Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow

@ FPL2019, Barcelona, September 11, 2019

Johan Peltenburg¹, Jeroen van Straten¹, Lars Wijtemans¹
Lars T.J. van Leeuwen¹, Zaid Al-Ars¹, H. Peter Hofstee²

1. Delft University of Technology, Netherlands
2. IBM, Austin, Texas, USA

Thanks to our supporters:

Fitoptivis European ECSEL project
no. ECSEL2017-1-737451
Xilinx
Outline

- The challenge of FPGA integration with Big Data Analytics
- Overcoming serialization bottlenecks with Apache Arrow
- Fletcher
- Mini-tutorial (if time)
- Results
- Conclusion & future work
An FPGA Accelerator Dev. Perspective

- High-performance datapath
- Write a host-side C lib
- Byte or even bit-level control of data structure in memory
- Structs, unions, bitspecs, padding
- Sculpt datastructure to feed datapath efficiently
A Big Data Analytics Dev. Perspective:

Ease of Use

Write applications quickly in Java, Scala, Python, R, and SQL.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python, R, and SQL shells.

- **DataFrame**: like a database table or excel spreadsheet, but...

- **Huge. Typically in the order of GiBs to TiBs.**
- **Distributed** over multiple worker nodes (also in storage).
- Operations on it build Directed Acyclic Graphs (DAGS) and are lazily evaluated.
- DAGs are optimized, planned and scheduled to execute in parallel over a cluster.
- **Resilient** to node failure, provides automatic recovery and continuation.

```
import org.apache.spark.sql.DataFrame

val df = spark.read.json("logs.json")
df.where("age > 21")
  .select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

Source: https://spark.apache.org/
Big Data Analytics SW Ecosystem

Frameworks for storage, scalability, resilience, analysis, etc..

Software languages & run-times

- Apache Spark™
- Hadoop
- pandas
- DASK
- dremio
- Parquet
- R
- Python
- Java™
- Scala
- MATLAB
- C#
- Rust
- Golang
Serialization

- Iterate over all objects in collection (data is big)
- Traverse all object graphs (memory latency)
- Copy fields to some intermediate format both A and B understand (bandwidth lost)
- Reconstruct objects in B ((de)allocation overhead)
I/O bandwidth catching up

Relative impact on accelerators

Original process on CPU:

Process on GPGPU/FPGA with serialization (potentially, but not necessarily, exaggerated)

Desired profile:

- CPU compute time
- (De)serialize / copy time
- Accelerator compute time

Serialization throughput on collection of Java (OpenJDK) objects on POWER8 [1]:

Overcoming serialization bottlenecks

- In-memory formats determined by:
  - Programming languages
    - Run-time system design choices
    - Standard libraries
  - Algorithms
  - Programmers
- Increased heterogeneity → more IPC → more serialization overhead
- What if data is...
  - In a **standardized format**?
    - That every language can use (through libraries or otherwise).
  - As **contiguous** as possible?
    - We can move it using large bursts, no pointer chasing, less misalignment overhead
Apache Arrow\(^3\)

- **Standardized representation in-memory – Common Data Layer**
- **Columnar format**
  - Hardware friendly while iterating over entries in single column (SIMD, caches, etc…)
  - Better for many algorithms, worse for some others.
- **Libraries and APIs for 10+ languages to build and access data sets (zero-copy)**

---

Arrow terminology:

**Schema:**
Description of data types in a *RecordBatch*

**RecordBatch:**
Tabular structure containing Arrow *arrays*

**Arrays:**
A RecordBatch “column”. Combination of Arrow *buffers*, can be nested

**Buffers:**
Contiguous C-like arrays

---

**Schema** MySchema {
  **A:** Float (nullable)
  **B:** List<Char>
  **C:** Struct{
    **E:** Int16
    **F:** Double
  }
}

**Index**

<table>
<thead>
<tr>
<th>Index</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.33f</td>
<td>ola</td>
<td>{1, 3.14}</td>
</tr>
<tr>
<td>1</td>
<td>7.01f</td>
<td>fpl</td>
<td>{5, 1.41}</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>@upc</td>
<td>{3, 1.61}</td>
</tr>
</tbody>
</table>

**Index**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.33f</td>
</tr>
<tr>
<td>1</td>
<td>7.01f</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
</tr>
</tbody>
</table>

**Index**

<table>
<thead>
<tr>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

**Index**

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Offset**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>o</td>
</tr>
<tr>
<td>1</td>
<td>l</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>p</td>
</tr>
<tr>
<td>5</td>
<td>l</td>
</tr>
<tr>
<td>6</td>
<td>@</td>
</tr>
<tr>
<td>7</td>
<td>u</td>
</tr>
<tr>
<td>8</td>
<td>p</td>
</tr>
<tr>
<td>9</td>
<td>c</td>
</tr>
</tbody>
</table>

**Index**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.14</td>
</tr>
<tr>
<td>1</td>
<td>1.41</td>
</tr>
<tr>
<td>2</td>
<td>1.61</td>
</tr>
</tbody>
</table>
Integrating FPGA and Arrow

- Arrow ‘turns out’ to be hardware-friendly
  - In-memory format clearly specified, to every bit
  - Highly contiguous & columnar format
    - Iterate over a column in streaming fashion
    - Useful for: maps, reductions, filters, etc...
  - Parallel accessible format
    - E.g. uses offsets, not lengths, for variable length data – we can start anywhere
    - Useful for: maps, reductions, filters, etc...
- Backed by a large and ever growing community
- Integration in many BDA frameworks, even without official format stability
- Can we generate easy-to-use, high throughput hardware interfaces automatically?
Main contribution:

FLETCHER

Fully open-source (Apache-2.0),
Vendor agnostic,
Generates easy-to-use high-throughput Interfaces.
Integrate FPGA accelerators with Apache Arrow.
Example: Interface for accelerator parsing strings

**Typical:**

- Memory interface
- Byte Address
- Bus Word
- Manual implementation of interface
- Computational part

**Fletcher:**

- Memory interface
- Fletcher-generated interface
- RecordBatch
- Length
- Stream
- Char stream
- Computational part

**Easy-to use:**
Data is delivered as streams that make sense w.r.t. schema field types.

**High-throughput:**
Number of values delivered per cycle configurable.
Generated interface overview

- Architecture based on library with streaming primitives
- **BufferReader/Writer**: Basic unit to read (N) Arrow Buffer elements
- **ArrayReader/Writer**: Combination of BufferReaders/Writers [1]
  - Dictated by the schema field and format specification
  - Generated through pure HDL; vendor agnostic
- **RecordBatchReader/Writer**: Combination of ArrayReaders/Writers
- **Mantle**: Wraps multiple RecordBatchR/W + bus infrastructure

Combining BufferReaders into ArrayReaders

- Arrow Schema & format spec dictate how to combine buffers.
- Passed to ArrayReaders through configuration string in HDL.
- Seeking the limits of synthesis tools :-)
- Over 10k+ random field types simulated.
High-level architecture generation: Fletchgen

- **Need syntactically pleasing interfaces**
  - Grouping of ArrayReader/Writer interfaces for RecordBatches
  - Stream names must correspond to schema fields
  - *Synthesizable* HDL too limited
- **Need kernel template generation** for kernel implementation in HDL/HLS
- **Need simulation**
- **Need platform integration**
- **High-level architecture generator: Fletchgen**
Reap the benefits of Arrow:
- Create **one accelerator**.
- Leverage in **any supported language**.

Fletcher Generated Hardware Interface is platform agnostic — requires no IP, tcl scripts, etc...

Top level with AXI4 interface available.
Mini-tutorial: Fletcher “Hello, World!”

Trivial example:
- Sum a column of integers

Get to know the toolchain

More realistic applications:
- Complex types
- More Arrow Arrays
- More input/output RecordBatches

Also on GitHub:
https://github.com/abs-tudelft/fletcher
Step 1: Create an Arrow Schema

```
import pyarrow as pa

number_field = pa.field('number', pa.int64(), nullable=False)
schema = pa.schema([number_field])

metadata = {b'fletcher_mode': b'read',
            b'fletcher_name': b'ExampleBatch'}
schema = schema.add_metadata(metadata)
```
Step 2: Create a RecordBatch
(optional, for simulation)

data = [pa.array([1, -3, 3, -7])]
recordbatch = pa.RecordBatch.from_arrays(data, schema)

writer = pa.RecordBatchFileWriter('recordbatch.rb', schema)
writer.write(recordbatch)
writer.close()
Step 3: Generate the design

RecordBatch input

Design output languages

$ fletchgen -n Sum -r recordbatch.rb -s recordbatch.srec -l vhdl dot --sim

Kernel name

Memory model file

Generate simulation top-level
Step 4: Implement the kernel

- Start from template.
- Use your favorite tools:
  - Custom HDL
  - Vivado HLS
  - ...
- Kernel interfaces:
  - AXI4-lite MMIO
  - Command streams to generated interface
  - Data streams from generated interface

```cpp
int sum(RecordBatchMeta ExampleBatch_meta, hls::stream<f_int64>& ExampleBatch_number) {
  ...
}
```
Step 5: Simulate the design

https://github.com/abs-tudelft/vhdeps

$ vhdeps -i path/to/fletcher/hardware -i . ghdl SimTop_tc

Fletcher hardware libs

Generated simulation top-level

Simulator target

GHDL, Questa, ...

Final summary:
* PASSED simtop_tc
Test suite PASSED
Step 6: Write host-side software

```python
import pyarrow as pa
import pyfletcher as pf

...  

platform = pf.Platform()
platform.init()

context = pf.Context(platform)
context.queue_record_batch(batch)
context.enable()

kernel = pf.Kernel(context)
kernel.start()
kernel.wait_for_finish()

...  
```
Step 7: Target a Platform

- **Supported platforms:**
  - **OpenPOWER CAPI SNAP**
    - If implementation allows, directly streamable from host-memory using virtual addresses on FPGA
  - **AWS EC2 F1**
    - Requires copy to on-board memory
Regular Expression Matching

- Given N strings
- Match M regular expressions
- Count matches for each regexp
- Example:

```
REGULAR EXPRESSION
/.*[Kk][Ii][Tt][Ee][Nn].*

TEST STRING
My snake just escaped! Be careful!  
I love kittens!!!  
Who let the #dogs out?
```
# Regex throughput/speedup result

## Table I

<table>
<thead>
<tr>
<th>System</th>
<th>Language</th>
<th>Native data set w/ CPU</th>
<th>Serialization</th>
<th>FPGA Copy</th>
<th>FPGA Kernel</th>
<th>Speedup w/ Serialization</th>
<th>Arrow/Fletcher</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS/F1 (16 regex units)</td>
<td>C++</td>
<td>0.08</td>
<td>0.55</td>
<td>7.13</td>
<td>14.27</td>
<td>6.18</td>
<td>59.73</td>
<td>9.67</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>0.04</td>
<td>0.83</td>
<td>7.17</td>
<td>14.28</td>
<td>15.93</td>
<td>107.73</td>
<td>6.76</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>0.03</td>
<td>0.27</td>
<td>7.13</td>
<td>14.27</td>
<td>8.24</td>
<td>152.91</td>
<td>18.56</td>
</tr>
<tr>
<td>P9/SNAP (8 regex units)</td>
<td>C++</td>
<td>0.43</td>
<td>0.81</td>
<td>n/a</td>
<td>7.61</td>
<td>1.70</td>
<td>17.78</td>
<td>10.44</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>0.11</td>
<td>0.81</td>
<td>n/a</td>
<td>7.61</td>
<td>6.77</td>
<td>70.72</td>
<td>10.45</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>0.16</td>
<td>0.16</td>
<td>n/a</td>
<td>7.61</td>
<td>0.95</td>
<td>46.49</td>
<td>48.69</td>
</tr>
</tbody>
</table>
Regex on 1GiB of tweet-like strings
## Writing random length (0-255) strings

<table>
<thead>
<tr>
<th>System</th>
<th>Language</th>
<th>Throughput (GB/s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>To native container</td>
<td>To Arrow RecordBatch</td>
</tr>
<tr>
<td>AWS/F1</td>
<td>C++</td>
<td>0.85</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>0.96</td>
<td>2.60</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>0.59</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>-</td>
<td>9.76</td>
</tr>
<tr>
<td>P9/SNAP</td>
<td>C++</td>
<td>0.76</td>
<td>7.52</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>1.60</td>
<td>7.68</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>0.28</td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>-</td>
<td>9.76</td>
</tr>
</tbody>
</table>
K-Means clustering, internal iteration bandwidth & total run-time

### TABLE III

**K-MEANS CLUSTERING RESULTS**

<table>
<thead>
<tr>
<th>Language</th>
<th>Avg. GB/s</th>
<th>Total run-time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>FPGA</td>
</tr>
<tr>
<td>C++</td>
<td>1.40</td>
<td>11.15</td>
</tr>
<tr>
<td>Python</td>
<td>1.29</td>
<td>11.15</td>
</tr>
<tr>
<td>Java</td>
<td>1.00</td>
<td>11.15</td>
</tr>
</tbody>
</table>

AWS EC2 F1 only
Conclusion

- Big data analytics systems increasingly heterogeneous – many different tools in many different technologies.

- Apache Arrow: one in-memory format for IPC through shared memory for most languages / runtimes / technologies.

- Fletcher: Arrow format allows us to generate high-throughput, easy-to-use hardware interfaces for FPGA.

- Streaming kernels benefit the most, more computationally oriented kernels less.

- Paves the way for more efficient FPGA accelerator integration with any of the supported big data analytics tools.
Spin-off projects & future work

- **Dynamic Arrow Buffers in Hardware** to support buffer resizing (Lars Wijtemans)
- **Parquet-to-Arrow decoder and decompressor** (Lars van Leeuwen, Jian Fang)
- HLS integration for map, reduce, **SQL-defined filters** (Erwin de Haan)

- Data-defined architecture.

- Can we turn this into a closed-loop self-optimizing interface generation and profiling framework?
  - Long-running workload: plenty of time to synthesize.
  - We only need one node of a cluster to do it.
  - Are the gains worth the cost?
Thank you for your attention!

References:

https://github.com/abs-tudelft/fletcher

Open-sourced example projects / existing applications:
- Regular Expression matching example / benchmark: https://github.com/abs-tudelft/fletcher-example-regexp
- K-Means clustering example/benchmark: https://github.com/abs-tudelft/fletcher-example-kmeans
- Writing strings to Arrow format using CAPI 2.0 and SNAP @ 10 GB/s: https://github.com/abs-tudelft/fletcher/blob/develop/examples/stringwrite
- Posit arithmetic on FPGA, BLAS and PairHMM accelerators by Laurens van Dam: https://github.com/lvandam/posit_blas_hdl
  https://github.com/lvandam/pairhmm_posit_hdl_arrow

Thanks to our supporters:
Fitoptivis European ECSEL project no. ECSEL2017-1-737451
Xilinx
## Area utilization

### Tab. 2: Area utilization statistics for a Xilinx XCVU9P device

<table>
<thead>
<tr>
<th>Type</th>
<th>Resource</th>
<th>W=8</th>
<th>W=16</th>
<th>W=32</th>
<th>W=64</th>
<th>W=128</th>
<th>W=256</th>
<th>W=512</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Column Reader Prim(W)</strong></td>
<td>CLB LUTs</td>
<td>0.30%</td>
<td>0.28%</td>
<td>0.26%</td>
<td>0.24%</td>
<td>0.22%</td>
<td>0.20%</td>
<td>0.21%</td>
</tr>
<tr>
<td></td>
<td>CLB Registers</td>
<td>0.20%</td>
<td>0.20%</td>
<td>0.20%</td>
<td>0.20%</td>
<td>0.22%</td>
<td>0.24%</td>
<td>0.26%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B36)</td>
<td>0.65%</td>
<td>0.65%</td>
<td>0.65%</td>
<td>0.65%</td>
<td>0.65%</td>
<td>0.65%</td>
<td>0.65%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B18)</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
</tr>
<tr>
<td><strong>Column Reader List of Prim(W)</strong></td>
<td>CLB LUTs</td>
<td>2.34%</td>
<td>1.81%</td>
<td>1.46%</td>
<td>1.32%</td>
<td>1.03%</td>
<td>1.04%</td>
<td>0.78%</td>
</tr>
<tr>
<td></td>
<td>CLB Registers</td>
<td>1.01%</td>
<td>1.01%</td>
<td>1.01%</td>
<td>1.01%</td>
<td>1.00%</td>
<td>1.00%</td>
<td>1.00%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B36)</td>
<td>1.30%</td>
<td>1.30%</td>
<td>1.30%</td>
<td>1.30%</td>
<td>1.30%</td>
<td>1.30%</td>
<td>1.30%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B18)</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.09%</td>
</tr>
<tr>
<td><strong>Column Writer Prim(W)</strong></td>
<td>CLB LUTs</td>
<td>0.20%</td>
<td>0.19%</td>
<td>0.19%</td>
<td>0.20%</td>
<td>0.20%</td>
<td>0.22%</td>
<td>0.23%</td>
</tr>
<tr>
<td></td>
<td>CLB Registers</td>
<td>0.28%</td>
<td>0.28%</td>
<td>0.28%</td>
<td>0.28%</td>
<td>0.29%</td>
<td>0.31%</td>
<td>0.33%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B36)</td>
<td>0.37%</td>
<td>0.37%</td>
<td>0.37%</td>
<td>0.37%</td>
<td>0.37%</td>
<td>0.37%</td>
<td>0.37%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B18)</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.02%</td>
<td>0.02%</td>
</tr>
<tr>
<td><strong>Column Writer List of Prim(W)</strong></td>
<td>CLB LUTs</td>
<td>1.03%</td>
<td>0.97%</td>
<td>0.91%</td>
<td>0.87%</td>
<td>0.80%</td>
<td>0.78%</td>
<td>0.52%</td>
</tr>
<tr>
<td></td>
<td>CLB Registers</td>
<td>1.18%</td>
<td>1.12%</td>
<td>1.11%</td>
<td>1.11%</td>
<td>1.06%</td>
<td>1.06%</td>
<td>0.73%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B36)</td>
<td>1.11%</td>
<td>1.11%</td>
<td>1.06%</td>
<td>1.06%</td>
<td>1.06%</td>
<td>1.06%</td>
<td>0.74%</td>
</tr>
<tr>
<td></td>
<td>Block RAM (B18)</td>
<td>0.07%</td>
<td>0.05%</td>
<td>0.07%</td>
<td>0.07%</td>
<td>0.07%</td>
<td>0.07%</td>
<td>0.05%</td>
</tr>
</tbody>
</table>