The Fletcher Framework for Programming FPGAs

OpenPOWER Summit Europe
October 3, 2018

Johan Peltenburg
Accelerated Big Data Systems
Quantum & Computer Engineering Department
Delft University of Technology
Additional Credits

Development (TU Delft)
Jeroen van Straten
Matthijs Brobbel
Laurens van Dam
Lars Wijtemans
Lars van Leeuwen

Support
Zaid Al-Ars (TU Delft)
Peter Hofstee (IBM/TU Delft)
Jinho Lee (IBM)
CAPI SNAP team (IBM)
Cathal McCabe (Xilinx)
## Heterogeneous software processes

<table>
<thead>
<tr>
<th>Language</th>
<th>Runs on top of...</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Fortran</td>
<td>CPU</td>
<td>Compiled to machine instructions (sometimes called native instructions)</td>
</tr>
<tr>
<td>Rust Julia</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Scala</td>
<td>Java Virtual Machine</td>
<td>Compiled to Java Bytecode Could be Just-In-Time compiled to machine instructions</td>
</tr>
<tr>
<td>Python R</td>
<td>Interpreter</td>
<td>Interpreted Strong integration with native libraries</td>
</tr>
</tbody>
</table>
Heterogeneous computing

• Big data systems are becoming increasingly **heterogeneous**.
  - Many different “types” of processes in both SW and HW.

• Example: TensorFlowOnSpark[1]
  - You can run a **Python** program
    • That uses NumPy (Python bindings on top of a **C** core)
  - Interfacing with TensorFlow, programmed in **CUDA**
    • Running on a **GPU**
  - On top of Spark, written in **Scala/Java**
    • Running on a **Java Virtual Machine**
    • That runs on your **CPU**

• What challenges does this bring?

Serialization

- Iterate over all objects in collection
- Traverse all object graphs (memory latency)
- Copy fields to some intermediate format both A and B understand (bandwidth lost)
- Reconstruct objects in B (allocations)
Relative impact on accelerators

Original process on CPU:

Accelerated process on GPGPU/FPGA with serialization:

Desired acceleration profile:

- CPU compute time
- (De)serialize / copy time
- Accelerator compute time
Overcoming serialization bottlenecks

• We (de)serialize a lot… Can we do this smarter?

• What if data is…
  – In a **standardized format**?
    • Every language run-time can use it.
  – As **contiguous** as possible?
    • We can move it quickly without traversing object graphs
• Standardized representation in-memory: Common Data Layer
• Columnar format
  – Hardware friendly while iterating over data (SIMD, caches, etc…)
• Libraries and APIs for various languages to build and access data
Schema X { 
A: Float (nullable) 
B: List<Char> 
C: Struct{
   E: Int16 
   F: Double 
}
}

Arrow terminology:
Schema:  
Description of data types in a RecordBatch
RecordBatch:  
Tabular structure containing arrays
Arrays:  
Combination of buffers, can be nested
Buffers:  
Contiguous C-like arrays
Integrating FPGA and Arrow

- Arrow is hardware-friendly
  - *Standardized* format
    - If you know the schema you know exactly where the data is.
  - *Contiguous & columnar* format
    - Iterate over a column in *streaming* fashion
    - Useful for: maps, reductions, filters, etc...
  - *Parallel accessible* format
    - Uses offsets, not lengths, for variable length data
    - Useful for: maps, reductions, filters, etc...

- We can *generate easy-to-use hardware interfaces automatically*
Fletcher\cite{3} architecture:

1. Data source (disk, network) → Fletcher run-time libraries
2. Apache Arrow libraries → Arrow Table
3. Fletcher run-time libraries → Arrow Table
4. Fletcher run-time libraries → Host Memory
5. Accelerator Design (manual, HLS) → Accelerator sources
6. Generated Interface → Hardware Accelerated Function
7. Hardware Accelerated Function → FPGA

Generated Interface internals

- Based on **streaming primitives**
  - Slices, splitters, etc...
  - Arbiters, serializers, parallellizers, etc...
  - Normalizer, accumulators, etc...
- Each Arrow buffer gets its own **BufferReader/Writer**
- Combination of BufferReader/Writer forms a **ColumnReader/Writer**
- Generated through **pure HDL; vendor agnostic**
  - Simulation in GHDL, QuestaSim, XSIM, PSLSE
- Verification: random schema and testbench generation, **over 1000+ schemas tested**
Internals: Fixed length data
(with validity bitmap)

- User streams in first and last index in the table.
- Column Reader streams the requested rows in order.

- Internal command stream:
  - First element offset in the data word.
  - No. valid elements in the data word.
- Response handler aligns and serializes or parallelizes the data.

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</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>

<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>
Internals: Variable length data
(without validity bitmaps)

<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>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>r</td>
</tr>
<tr>
<td>4</td>
<td>i</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
</tr>
</tbody>
</table>
Internals: Structs
(without validity bitmaps)
Motivating use case: Regular Expression Matching

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

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

TEST STRING
My snake just escaped! Be careful!
I love kittens!!!
Who let the #dogs out?
```
Results

RegExp on 1GiB of tweet-like strings.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Throughput (GB/s)</th>
<th>Speedup over fastest / serialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCVU9P / VCU1525 / AWS EC2 F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fastest C++</td>
<td>0.067</td>
<td>1</td>
</tr>
<tr>
<td>FPGA &amp; C++ serialization</td>
<td>0.418</td>
<td>6.2</td>
</tr>
<tr>
<td>FPGA &amp; Arrow</td>
<td>1.611</td>
<td>24.0 / 3.9</td>
</tr>
<tr>
<td>Fastest Java</td>
<td>0.027</td>
<td>1</td>
</tr>
<tr>
<td>FPGA &amp; Java serialization</td>
<td>0.293</td>
<td>11.0</td>
</tr>
<tr>
<td>FPGA &amp; Arrow</td>
<td>1.611</td>
<td>60.6 / 5.5</td>
</tr>
<tr>
<td>XCKU060 / ADKU3 / POWER8+CAPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fastest C++</td>
<td>0.413</td>
<td>1</td>
</tr>
<tr>
<td>FPGA &amp; C++ serialization</td>
<td>0.281</td>
<td>0.7</td>
</tr>
<tr>
<td>FPGA &amp; Arrow</td>
<td>2.963</td>
<td>7.2 / 10.5</td>
</tr>
<tr>
<td>Fastest Java</td>
<td>0.144</td>
<td>1</td>
</tr>
<tr>
<td>FPGA &amp; Java serialization</td>
<td>0.197</td>
<td>1.4</td>
</tr>
<tr>
<td>FPGA &amp; Arrow</td>
<td>2.963</td>
<td>20.5 / 15.1</td>
</tr>
</tbody>
</table>
Hands on: sum example

• Suppose we want to add all integers in a column.

... weight[2], weight[1], weight[0]  
\[\text{Accumulate}\]  
result
Step 1: Generate Schema

- Currently C++ libraries are most advanced in Arrow
- Examples will use C++.
  - Python, Java, R, Rust, Go, etc… also possible
- Create list of Schema fields; **name, type, nullable**
- Add **metadata** for Fletcher
  - read/write, fields to ignore
  - bus width, elements per cycle
  - No. MMIO registers for user, etc...
- Save Schema as **Flatbuffer file**

```cpp
std::vector<std::shared_ptr<arrow::Field>> schema_fields = {
  arrow::field("weight", arrow::int64(), false)
};
```
Step 2: Fletchgen Wrapper Generator

- Generates **wrapper** based on schema
- Specify desired top-level
  - Currently **AXI** is supported
    - AXI4 master interface to (host/on-board) memory
    - AXI4-lite slave interface for MMIO
  - Simulation top-level available
    - Can provide Arrow RecordBatch to simulation
- Compatible with baseline project for **CAPI SNAP** / AWS EC2 F1

```bash
$ fletchgen \
  --input sum.fbs \
  --output fletcher_wrapper.vhd \
  --axi axi_top.vhd
```
Step 3: Implement Accelerator Kernel

- Accelerator kernel template.
- Two streams appear (for this example):

```vhdl
weight_cmd_firstIdx : out std_logic_vector(INDEX_WIDTH-1 downto 0);
weight_cmd_lastIdx  : out std_logic_vector(INDEX_WIDTH-1 downto 0);
weight_cmd_ready    : in  std_logic;
weight_cmd_valid    : out std_logic;
weight_valid        : in  std_logic;
weight_ready        : out std_logic;
weight_data         : in  std_logic_vector(63 downto 0);
weight_last          : in  std_logic;
```
Step 4: Finishing touches

- Simulate, Debug, Place, Route
- Easy to use run-time interfaces provided
  - C++ available
  - Python incoming
  - Other languages with Arrow support in future
- Set custom MMIO with desired configuration
- Put data in Arrow RecordBatch
- Throw at Fletcher
Future work

- Continued development
  - More applications for showcasing/verification
  - Support for more Arrow-supported languages
- HLS integration for map/reduce/filter lambdas
- SQL integration
Summary

- Accelerators can be heavily burdened by serialization overhead from heterogeneous systems

- Apache Arrow format prevents serialization overhead and allows hardware interface generation

- Paves the way for more efficient FPGA acceleration in any of the supported languages

- Fletcher is the framework!

  https://github.com/johanpel/fletcher
References


Example projects / existing applications:

• Regular Expression matching example:
  https://github.com/johanpel/fletcher/tree/master/examples/regexp

• Writing strings to Arrow format using CAPI 2.0 and SNAP @ 11 GB/s:
  • https://github.com/johanpel/fletcher/tree/master/examples/stringwrite

• Posit arithmetic on FPGA, accelerated through Fletcher/SNAP by Laurens van Dam:
  • https://github.com/lvandam/posit_blas_hdl

• PairHMM accelerator with posit arithmetic by Laurens van Dam & Johan Peltenburg:
  • https://github.com/lvandam/pairhmm_posit_hdl_arrow