An Intermediate Representation for Composable Typed Streaming Dataflow Designs

Zaid Al-Ars
Matthijs A. Reukers, Yongding Tian, Peter Hofstee, Matthijs Brobbel, Johan Peltenburg and Jeroen van Stratens
Accelerated Big Data Systems group
Email: z.al-ars@tudelft.nl
Web: abs.ewi.tudelft.nl

Delft University of Technology
Overview

• Need for efficient big data analytics
• Challenges in accelerating big data analytics
• Addressing data conversion overhead
• Addressing HW design limitations
• Details of the Tydi specification
• Impact on reducing design complexity
Efficiency challenges in big data analytics

- Data centers energy consumption is increasing rapidly
- By 2030, it is projected that the ICT industry will consume more than 20% of world energy
- Need to increase efficiency using optimized HW acceleration

Nature: Nicola Jones, “How to stop data centres from gobbling up the world’s electricity”, 2018
Efficiency challenges in big data analytics

3 main efficiency challenges in big data analytics

- Virtualized programming languages (e.g., JVM)

- Inefficient accelerator HW implementations
A string

C++
- String size
- Pointer to char buffer
- Internal char array (optionally used)
- Optionally allocated char array

Java 8
- JVM object header
- UTF-16 Array reference
- Hash cache
- JVM array object header
- UTF16 array

Python
- Python variable length object header
- Hash
- State
- Variable length character array

FPGA
- Length stream
- Character stream
FPGA integration results

Regex on 1GiB of tweet-like strings.

J. Peltenburg et al., “Pushing big data into accelerators: Can the JVM saturate our hardware?”, International Conference on High Performance Computing, 2017

<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>
- 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
Technology Stack

Our tool to build applications for FPGA

Legend

- Technology
- Applications
- Advantage

Targeting DATA COLLECTION issues

Targeting DATA ANALYTICS issues

J. Hoozemans et al., “FPGA Acceleration for Big Data Analytics: Challenges and Opportunities”, IEEE CSM, 2021
Data analysis:
Regex acceleration in Dremio

- Accelerating regular expression matching with Dremio.
- Runs on AWS EC2 FPGA-enabled instances.
- After physical planning, apply FPGA Acceleration Planning.
- Strings in, matching indices out.
Regex acceleration in Dremio results

J. Peltenburg et al., “Battling the CPU bottleneck in apache parquet to arrow conversion using FPGA”, FPT, 2020

- FPGA acceleration of regular expression matching
- Matches on tweet-like strings
- Needed to go native to squeeze out all performance from CPU
  - Optimized using Google RE2
  - Best CPU performance
- Over 10x higher throughput for Dremio + FPGA
- Try it out:
  https://github.com/teratide/tidre-demo
Designing FPGA accelerators is complex

- Lack of data abstractions
- Low-level attributes (like assembly programming)
=> Large codebase
=> not composable
=> complex to debug
=> etc.

Adrian Sampson, https://www.sigarch.org/hdl-to-adl/
Tydi specification to facilitate streaming of complex data

- **Tydi** is open specification to abstract streaming data in HW
- Automates HW design of streaming data interfaces
- Allows HW components (aka Streamlets) to be composed together
- It provides the following:
  - Data types
  - Data organization
  - Interface requirements

---

J. Peltenburg et al., Tydi: an open specification for complex data structures over hardware streams, IEEE Micro, 2020
Tydi specification to facilitate streaming of complex data: **Data types**

**Tydi** provides a type system for composite and variable-length data.

Type system defines the following data types:

1. **Stream**: represents physical stream carrying the following logical types
2. **Bits(N)**: represents a data signal of N bits
3. **Group**: composites of multiple types (all types set at the same time)
4. **Union**: composites of multiple types (one type can be active at a time)
5. **Null**: user-defined data type
Tydi specification to facilitate streaming of complex data: Data organization

- **Tydi** defines how data elements are organized in transfers
- Dimensionality property indicates if data is part of a sequence
- Translated to a “last” signal in HW
- Higher dimensionality need multiple last signals for nested sequences
Tydi specification to facilitate streaming of complex data: Requirements

• **Tydi** defines the requirements system needs from transfers
• Streams describe how transfers should be organized in space and time
• Tydi provides the following requirements attributes
  • Throughput
  • Direction
  • Synchronicity
  • Complexity
• Complexity encodes guarantees on how elements of a sequence are transferred
• Lower logical complexity imposes more restrictions on a source making it more difficult to implement
Example Tydi spec data definition

Suppose we would like to transfer “Hello World” on hardware:

• Each character is **8bit**. We need two dimensions to indicate the end of a word and the end of a sentence

• “hello world” => \textit{Stream}(\texttt{Bits(8)}, \texttt{dimension}=2)

• To satisfy the throughput requirement, we can specify 3 lanes to deliver the data
Example Tydi spec data definition

- We can also adjust the complexity to prevent problems when data isn't available or when sink components are busy.
Tydi-IR toolchain implementation

• Tydi-IR system implemented using three components
  1. Parser and grammar (stores results in query system)
  2. Query system to store IR’s declarations & expressions (types & components) on-demand
  3. Backend which uses the query system and emits VHDL
Tydi-IR toolchain implementation

• Tydi-IR describes components and their connections.
• Example shows Tydi types “a”, “c”, “b”, “d”

```
-- documentation (optional)
component my_example_space_comp1_comp1
port (  
    clk : in std_logic;  
    rst : in std_logic;  
    a_valid : in std_logic;  
    a_ready : out std_logic;  
    a_data : in std_logic_vector(53 downto 0);  
    b_valid : in std_logic;  
    b_ready : in std_logic;  
    b_data : out std_logic_vector(53 downto 0);  
    -- this is port
    -- documentation
    c_valid : in std_logic;  
    c_ready : out std_logic;  
    c_data : in std_logic_vector(53 downto 0);  
    d_valid : in std_logic;  
    d_ready : in std_logic;  
    d_data : out std_logic_vector(53 downto 0)  
);  
end component;
```
Tydi-IR evaluation

• Compare with AXI4 streams, notice that “Types” only need to be declared once

<table>
<thead>
<tr>
<th></th>
<th>Type Declaration</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXI4 equiv. (TIL)</td>
<td>48*</td>
<td>5</td>
</tr>
<tr>
<td>AXI4 equiv. (TIL, Group)</td>
<td>59*</td>
<td>1</td>
</tr>
<tr>
<td>AXI4 equiv. (VHDL)</td>
<td>-</td>
<td>28</td>
</tr>
<tr>
<td>AXI4</td>
<td>-</td>
<td>44</td>
</tr>
<tr>
<td>AXI4-Stream equiv. (TIL)</td>
<td>15*</td>
<td>1</td>
</tr>
<tr>
<td>AXI4-Stream equiv. (VHDL)</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>AXI4-Stream</td>
<td>-</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1
Lines of code to represent an interface in TIL, compared to the resulting number of signals in VHDL or for an equivalent interface standard. *Only required once.

M.A. Reukers et al., “An intermediate representation for composable typed streaming dataflow designs”, VLDB, 2023
Data collection: JSON parsing

- Accelerating JSON parsing for low-latency
- SigmaX application and system
- Network-attached FPGA – low latency
- Parsing multiple JSONs to Arrow RecordBatch in FPGA
- Resizing and serialization to Arrow IPC message on CPU
- Reduction of design time from weeks to days

J. Peltenburg et al., “Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators”, FPT, 2021
Conclusions

- High performance big data analytics efficiency can be improved
  - Using high-level abstractions that are aware of HW and application
- Tydi prevents SW constructs from being lost-in-translation in HW
- Implemented Tydi-IR, a toolchain to for streaming HW components
- Results indicate improved code readability & reduced HW design effort