FPGAs as Streaming MIMD Machines for Data Analytics

James Thomas, Matei Zaharia, Pat Hanrahan
CPU/GPU Control Flow Divergence

• For peak performance, CPUs and GPUs require groups of threads to have identical control flow (SIMD)

• Many important applications have divergent control flow across threads
  – String processing (parsing, regex)
  – Compression
  – Machine learning inference
  – Video encoding
Motivation

GB/s/W for JSON Parsing

*estimate
Our Goal

• Provide language allowing developers to specify the logic and state of a single thread processing a single stream

• Create many copies of that thread on FPGA, each operating on its own stream and diverging arbitrarily from the others (MIMD)
Our Goal

• Language should be easier for software developers to write than RTL
  – Target users: high-performance library writers

• Should be a more performant abstraction than HLS
Why Not HLS?

• High-level synthesis (HLS) systems attempt to go from C loop to gates

• Our model in HLS:

```c
input[], output[], output_idx = 0
other array and scalar state
repeat:
    load block from DRAM into input
    for i in input:
        // per-input token logic, for example:
        if (...)  
            output[output_idx++] = ...
        if (...)  
            output[output_idx++] = ...
        update state
    flush output to DRAM
```
HLS Abstraction Mismatch

- Local arrays, including input and output, are implemented as FPGA BRAMs, which have at most two read/write ports.
- C does not expose this restriction, meaning many cycles may be required to resolve the reads/writes on each iteration.
  - Pipelining across iterations difficult due to loop-carried dependencies.
More BRAM details

- BRAMs are synchronous read/write
  - Read data appears one cycle after address provided
  - Writes committed one cycle after data supplied
Our Language

• Scala-embedded DSL
• Specify the register and BRAM state explicitly
• Specify logic using if s, elses, assignments, and emits of output tokens
• Virtual cycle – one step in program logic, ignoring details of BRAM timing
Our Language

• Restriction: only one read and one write per BRAM per virtual cycle, and only one emit
  – Checked by simulator
• Concurrent rather than sequential semantics
  – BRAM/register writes (including emit) occur at end of virtual cycle
  – Avoid BRAM read-after-write delays
pre_while Loops

- Convert each pre_while to if
- Repeatedly run virtual cycle pipeline, masking out operations outside of pre_while blocks
- When all pre_while conditions become false, run a finalizer virtual cycle for the operations outside of pre_whiles, and finally load new input token
Example

```plaintext
reg  counter(bitWidth = 8, init = 0)
bram histogram(numElts = 8, bitWidth = 9)
reg  hist_idx(bitWidth = 4, init = 0)

if (counter == 0) {
    pre_while (hist_idx < 8) { // multiple
        // virtual cycles for current input
        emit(histogram[hist_idx])
        histogram[hist_idx] = 0
        hist_idx += 1
    }
    hist_idx = 0
}

histogram[input] += 1
counter += 1 // wraps around
```
General Virtual Cycle Scheduling Strategy

• Generate pipeline long enough to handle the maximum length BRAM read dependency chain (e.g. \( a[b[x]] \))

• Add an extra pipeline stage to perform BRAM/register writes

• Always try to start next virtual cycle during previous virtual cycle’s write stage
Example Pipeline

```plaintext
reg counter(bitWidth = 8, init = 0)
bram histogram(numElts = 8, bitWidth = 9)
reg hist_idx(bitWidth = 4, init = 0)

if (counter == 0) {
    if pre_while (hist_idx < 8) {
        // multiple virtual cycles for current input
        emit(histogram[hist_idx])
        histogram[hist_idx] = 0
        hist_idx += 1
    }
    hist_idx = 0
}
histogram[input] += 1
counter += 1 // wraps around
```

Two-stage pipeline

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Feed hist_idx or input to histogram read address</td>
<td>-Feed appropriate write data to histogram</td>
</tr>
<tr>
<td>-emit if necessary</td>
<td>-emit if necessary</td>
</tr>
<tr>
<td>-Update registers</td>
<td>-Update registers</td>
</tr>
<tr>
<td>-If all pre_while conditions false (finalizer virtual cycle), load next token into input register</td>
<td>-If all pre_while conditions false (finalizer virtual cycle), load next token into input register</td>
</tr>
</tbody>
</table>
## Example Pipeline Scheduling

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Feed <code>hist_idx</code> or input to histogram read address</td>
<td>- Feed appropriate write data to histogram</td>
</tr>
<tr>
<td></td>
<td>- emit if necessary</td>
</tr>
<tr>
<td></td>
<td>- Update registers</td>
</tr>
<tr>
<td></td>
<td>- If all <code>pre_while</code> conditions false (finalizer virtual cycle), load next token into input register</td>
</tr>
</tbody>
</table>

### Virtual Cycles:

<table>
<thead>
<tr>
<th>Time</th>
<th>Stage 1</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="blue" alt="Stage 1" /></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><img src="red" alt="Stage 1" /></td>
<td><img src="blue" alt="Stage 2" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="red" alt="Stage 1" /></td>
<td><img src="red" alt="Stage 2" /></td>
</tr>
</tbody>
</table>

Start stage 1 of immediately by forwarding new register and BRAM data from stage 2 of.
Synthesis

• Synthesize specification for one stream processing unit (PU) into many copies driven by input and output memory controllers
Input Memory Controller

• Interacts with 512-bit AXI-4 interface, uses a burst size of 2 (2 sequential 512-bit transfers for each input address)
• Blocking round-robin: wait for each PU to be ready for next input block
  – Expect PUs to read input tokens at roughly same rate
• Reaches 90% of peak memory performance on Amazon F1
**Input Memory Controller**

Easy to keep this stream filled since each PU reads addresses purely sequentially.

- **AXI-4 Interface**
  - Addr input
  - Data output

- **1024-bit input buffer**
- **32-bit channel**
- **Need 16 buffers to saturate AXI-4 rate of 512 bits / cycle**

PU1
- Input BRAM
- Output BRAM

PU2
- Input BRAM
- Output BRAM

PU3
- Input BRAM
- Output BRAM

PU4
- Input BRAM
- Output BRAM
Experimental Setup

• Amazon F1 instances (Xilinx UltraScale+ xcvu9p), 125 MHz system clock
• CPU comparison: c4.8xlarge (36 Haswell vcores)
• GPU comparison: p3.2xlarge (V100)
• HLS comparison: Xilinx SDAccel 2017.1
Applications

1. JSON field extractor that can be configured with list of fields to extract at runtime
2. Integer compression: parallel search over 16 schemes to compress blocks of 4 integers
3. Gradient-boosted decision tree evaluation: decision tree nodes loaded at runtime
## CPU/GPU Comparison

<table>
<thead>
<tr>
<th></th>
<th>JSON Parsing</th>
<th>Integer Compression</th>
<th>Decision Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td># of FPGA processing units</td>
<td>512</td>
<td>192</td>
<td>384</td>
</tr>
<tr>
<td>FPGA GB/s</td>
<td>23</td>
<td>12</td>
<td>4.2</td>
</tr>
<tr>
<td>FPGA bottleneck (compute GB/s if memory-bound)</td>
<td>Memory (64)</td>
<td>Compute</td>
<td>Compute</td>
</tr>
<tr>
<td>CPU GB/s</td>
<td>6.3</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>GPU GB/s</td>
<td>30</td>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>FPGA GB/s/W (w/ DRAM)</td>
<td>1.1 (0.64)</td>
<td>0.75 (0.39)</td>
<td>0.22 (0.12)</td>
</tr>
<tr>
<td>CPU GB/s/W</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>GPU GB/s/W</td>
<td>0.16</td>
<td>0.11</td>
<td>0.48</td>
</tr>
<tr>
<td>FPGA GB/s/W speedup vs. CPU (w/ DRAM)</td>
<td><strong>37x (21x)</strong></td>
<td>75x (39x)</td>
<td>22x (12x)</td>
</tr>
<tr>
<td>FPGA GB/s/W speedup vs. GPU (w/ DRAM)</td>
<td><strong>6.9x (4x)</strong></td>
<td>6.8x (3.5x)</td>
<td>0.46x (0.25x)</td>
</tr>
</tbody>
</table>
Applications Discussion

• GBDT has few branches, very low computational intensity per BRAM read
  – GPU has much better SRAM bandwidth
• JSON parsing and integer compression have many branches and many operations per BRAM read that can be parallelized
## HLS Comparison

<table>
<thead>
<tr>
<th></th>
<th>JSON Parsing</th>
<th>Integer Compression</th>
<th>Decision Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our cycles per input token</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>HLS cycles per input token</td>
<td>7</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>
Future Directions

• Faster placement and routing due to repeated structure
• Integration with code generation frameworks like Weld
Thanks!

jjthomas@stanford.edu