Developing FPGAs as an Acceleration Platform for Data-Intensive Applications

James Thomas
AHA Affiliates Meeting
Data-Intensive Applications Growing

• Companies collecting more and finer-grained data about their systems, customers, and more

• Data is being dumped into cloud systems and CPUs are struggling to fully explore it – ML, SQL, and other workloads
FPGAs Are a Possible Solution

• Growing interest among hyperscalers such as Microsoft, Amazon, etc.
FPGA Accelerator Development Is Challenging

• How to maximize IO and DRAM performance?
• How to utilize full compute fabric?
• Analogous problems in CPU/GPU world, but the design space is much larger with FPGA
Solution: Domain-Specific Frameworks

• Goal: Build FPGA development frameworks specific to data-intensive applications

• In contrast to today’s push towards general frameworks like C-based high-level synthesis (HLS) that can’t perform in all domains
How can we simplify FPGA design for data-intensive applications?

• Many of these applications stream data sequentially from DRAM – reuse common memory controller
• Many of them are massively parallel – replicate a single core computational unit
Outline

• Fleet: a language & framework for parallel streaming applications on FPGAs
• Improving compile times for Fleet-like applications by reusing compilation work
• Accelerating subgroup analysis on FPGA using Fleet-like design patterns
Fleet: A Framework for Massively Parallel Streaming on FPGAs

James Thomas, Matei Zaharia, Pat Hanrahan
State-machine-based Stream Processing

• Each token affects next
  • Many important analytics apps: parsing, regex, compression, ML inference

• Often not intuitively parallelizable

• Easiest way to get high throughput is to process *multiple* streams in parallel

{"a":{"b":30}, "b":50}
Massively Parallel Streaming

Processing Unit (PU)

{ "a" ...
Massively Parallel Streaming

Processing Unit (PU)

{ "a" }

Processing Unit (PU)

{ "b" }

Processing Unit (PU)

{ "c" }
Fleet: Language and Compiler for Massively Parallel Streaming

- Language is an extension to Chisel oriented around token-based, state-machine streaming
- Compiler takes Fleet processing unit (PU), generates many copies and a memory controller to feed them with separate streams
Fleet System Overview

unit Histogram {
    bram a;
    if (...) ...
}

Fleet processing unit (PU)
Simple Fleet PU Example

```plaintext
unit CSVParser(inp_token_bits = 8, out_token_bits = 8) {
    reg cur_col(init = 0, ...)

    if (input == ',')
        cur_col += 1
    else if (input == '\n')
        cur_col = 0

    if (cur_col == 2)
        emit(input)
}
```

```
1, 2, 3, 4
5, 6, 7, 8
9, 9, 9, 9
```
Fleet PU Example

```
unit BlockHistograms(inp_token_bits=8, out_token_bits=8) {
  reg inp_counter(bitWidth = 7, init = 0)
  bram histogram(numElts = 256, bitWidth = 9)
  reg hist_idx(bitWidth = 9, init = 0)

  if (inp_counter == 100) { // emit histogram
    while (hist_idx < 256) {
      emit(histogram[hist_idx])
      histogram[hist_idx] = 0
      hist_idx += 1
    }
    hist_idx = 0
  }

  histogram[input] += 1
  inp_counter = (inp_counter == 100) ? 1 : inp_counter + 1
}
```
Compiler Generates Two-Stage Pipeline for PUs with BRAMs

- Allows programmer to think of BRAM as zero-latency
- Programmer must ensure each BRAM read & written at most once per PU firing

1. BRAM Read Stage
   - User logic for rd. addrs.
   - BRAM 1

2. BRAM & Reg. Write Stage
   - User logic for writes
   - User logic for writes
   - Reg 1
   - Forwarding
Compiler-Generated Verilog for Histogram PU

• Handles BRAM pipelining, stalling PU on input or output stalls – significant boilerplate for token-based streaming
Fleet Memory Controller

- Feeds multiple processing units in parallel, making data requests ahead of time for upcoming processing units
### Eval: Amazon F1 Performance

<table>
<thead>
<tr>
<th>App</th>
<th>Fleet # PUs</th>
<th>Perf/W improvement vs. V100 GPU</th>
<th>Fleet Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON Parsing</td>
<td>512</td>
<td>5.41x</td>
<td>Memory</td>
</tr>
<tr>
<td>Integer Coding</td>
<td>192</td>
<td>2.67x</td>
<td>Compute</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>384</td>
<td>0.35x</td>
<td>Compute</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>384</td>
<td>5.82x</td>
<td>Memory</td>
</tr>
<tr>
<td>Regex</td>
<td>704</td>
<td>2.62x</td>
<td>Memory</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>320</td>
<td>6.66x</td>
<td>Memory</td>
</tr>
</tbody>
</table>
Reasons for Speedup over CPU/GPU

• Large number of processing units
• Fusion of multiple CPU/GPU instructions into one FPGA cycle
• No penalty for control flow divergence across processing units
Fleet Conclusion

• Fleet provides an intuitive token-based streaming model for processing units that covers many applications

• It allows push-button generation of multiple-processing unit FPGA designs

https://github.com/jjthomas/Fleet
Software-like Compilation for Datacenter FPGA Accelerators

James Thomas, Chris Lavin, Alireza Kaviani
Stanford University and Xilinx Research Labs
## Fleet: Good Performance, Slow Compilation

<table>
<thead>
<tr>
<th>Application</th>
<th># of PUs</th>
<th>vs. CPU Perf/W</th>
<th>vs. GPU Perf/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON Parsing</td>
<td>512</td>
<td>26x</td>
<td>5.4x</td>
</tr>
<tr>
<td>Integer Coding</td>
<td>192</td>
<td>45x</td>
<td>2.7x</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>384</td>
<td>14x</td>
<td>0.4x</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>384</td>
<td>275x</td>
<td>5.8x</td>
</tr>
<tr>
<td>Regex</td>
<td>704</td>
<td>60x</td>
<td>2.6x</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>320</td>
<td>15x</td>
<td>6.7x</td>
</tr>
</tbody>
</table>

**Diagram:**
- **App Code** to **Fleet & Chisel Compiler** in 30-60 seconds
- **Verilog (.v)**
- **VIVADO** (Synthesis, P&R) in 8-12 hours *per try*
- **P&R Design (.dcp)**
- **Ingestion Flow** in 30-60 minutes

*Fleet: A Framework for Massively Parallel Streaming on FPGAs, ASPLOS 2020*
Data-Intensive FPGA App Compilation

• Memory controller often doesn’t change across apps – wasted work in redoing its place and route for each app

• Processing units identical – significant wasted work in redoing place and route for each one
Goal

• Fast compilation by:
  • Reusing compilation for replicated PUs
  • Take memory controller (shell) from a pre-implemented library

• Use standard Xilinx CAD tool (Vivado) for all placement and routing
Simple Solution: Vivado Out-of-context Flow

- Replicated PU
- Pre-implemented memory controller (shell)
Issue: Shell-to-PU Routing

• Problem: Vivado routing from replicated PUs to shell takes time (1-2 hours or more)
• Solution: Shell is pre-implemented, so route it to a two-column register block next to each PU location (“slot”) ahead of time
Example Shell

- Slot resource layout can be different per slot column (but resource counts are same) – requires separate implementations
180-Slot Shell
Online PU Flow

• Generate PU implementation for each slot column & replicate implementations

~8-10 minutes

*Register block added by RapidWright post Synthesis

~40-130 seconds
## Results

<table>
<thead>
<tr>
<th>PU</th>
<th>Interface Size (bits)</th>
<th># Logic Cells</th>
<th>PU Template Implementation Runtime</th>
<th>RapidWright PU Replication Runtime</th>
<th>Our Flow Total Runtime</th>
<th>Standard Flow Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot</td>
<td>46</td>
<td>71 (incl. DSP)</td>
<td>9m4s</td>
<td>1m1s</td>
<td>10m5s</td>
<td>82m50s</td>
<td>8.2×</td>
</tr>
<tr>
<td>Counter</td>
<td>22</td>
<td>109 (incl. BRAM)</td>
<td>10m37s</td>
<td>0m37s</td>
<td>11m14s</td>
<td>87m25s</td>
<td>7.9×</td>
</tr>
<tr>
<td>Summer</td>
<td>46</td>
<td>138</td>
<td>8m24s</td>
<td>0m41s</td>
<td>9m5s</td>
<td>82m51s</td>
<td>9.1×</td>
</tr>
<tr>
<td>JSON</td>
<td>22</td>
<td>352 (incl. BRAM)</td>
<td>10m57s</td>
<td>0m53s</td>
<td>11m50s</td>
<td>94m30s</td>
<td>8.0×</td>
</tr>
<tr>
<td>Time Series Pred.</td>
<td>22</td>
<td>512</td>
<td>8m31s</td>
<td>0m51s</td>
<td>9m22s</td>
<td>95m38s</td>
<td>10.2×</td>
</tr>
<tr>
<td>KNN</td>
<td>46</td>
<td>800 (incl. distr. RAM &amp; DSP)</td>
<td>8m25s</td>
<td>1m41s</td>
<td>10m6s</td>
<td>111m15s</td>
<td>11.0×</td>
</tr>
<tr>
<td>Integer Coder</td>
<td>46</td>
<td>1119 (incl. distr. RAM)</td>
<td>9m35s</td>
<td>2m7s</td>
<td>11m42s</td>
<td>117m19s</td>
<td>10.0×</td>
</tr>
</tbody>
</table>
Area Tradeoff

• Previously able to get 500+ PU’s with standard flow
• Still have room to add more slots
• Can still beat GPU with 180 PU’s in some cases, may be enough for some users
• For others, this can be a fast flow for prototyping, can use standard flow once design is finalized
Improving Utilization

• Need better support from CAD tools/architecture to make these techniques more effective
  • Consistent routing interface between shell and PUs within a single switchbox to avoid wastage from the two-column register block
  • More homogenous architecture supporting tighter packing of PU slots
  • More long wires to support routing of shell over PU slots
Fast Compilation Conclusion

• Fast compilation system for modular datacenter designs (~10x speedup)
• Open source at https://github.com/jjthomas/Fleet-Floorplanning
Accelerating Computation of Interesting Subgroups

with Trevor Gale, Pat Hanrahan, Matei Zaharia
Subgroup Analysis

• Want to know about interesting subgroups: e.g. users from New York aged 20-29 have a high propensity to churn

• Instead of analyst manually coming up with interesting two-dimensional subgroups to query, exhaustively search through all of them
Subgroup Table Computation

### Dataset

<table>
<thead>
<tr>
<th>Age</th>
<th>Region</th>
<th>Income</th>
<th>Metric (Churned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-29</td>
<td>Northeast</td>
<td>High</td>
<td>0</td>
</tr>
<tr>
<td>30-39</td>
<td>Midwest</td>
<td>Low</td>
<td>0</td>
</tr>
<tr>
<td>20-29</td>
<td>Pacific</td>
<td>Low</td>
<td>1</td>
</tr>
<tr>
<td>30-39</td>
<td>Pacific</td>
<td>High</td>
<td>1</td>
</tr>
</tbody>
</table>

### Subgroup tables: entries are (total row count, metric sum)

<table>
<thead>
<tr>
<th>Region</th>
<th>Age</th>
<th>Metric (Churned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Northeast</td>
<td>20-29</td>
<td>(2, 1)</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>Midwest</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20-29</td>
<td>(0, 0)</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>Pacific</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20-29</td>
<td>(1, 1)</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
<td>(1, 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Income</th>
<th>Age</th>
<th>Metric (Churned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>20-29</td>
<td>(1, 0)</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20-29</td>
<td>(2, 2)</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
<td>(1, 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Region</th>
<th>Income</th>
<th>Metric (Churned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Northeast</td>
<td>High</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>Midwest</td>
<td>Low</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>Pacific</td>
<td></td>
<td>(1, 1)</td>
</tr>
</tbody>
</table>
for row in dataset:
    for (f1, f2) in <all ordered pairs of features>:
        tables[(f1, f2)][row[f1]][row[f2]] +=
        (1, row[metric])
FPGA Design: Overview

• FPGA has enough SRAM to hold subgroup tables for about 2k feature pairs

• Need to “block” problem to fit in accelerator

• Divide feature set into blocks; one pair of blocks is crossed per accelerator invocation; with a block size of $B$ features there will be $B^2$ pair tables computed per invocation
FPGA Design: Architecture Example (B = 2)

- **F0/F2 RAM**
- **F1/F2 RAM**
- **F0/F3 RAM**
- **F1/F3 RAM**

<table>
<thead>
<tr>
<th>Metric</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 bits per feature</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Goes to all RAMs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8 bits for metric</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Goes to all RAMs</td>
<td></td>
</tr>
</tbody>
</table>
FPGA Design Details

• $B = 43$ was the largest block size we could fit on the FPGA (Amazon F1 instance)

• Implementation double-buffered – data transfer of the next pair of blocks overlapped with the computation
## Experimental Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Features</th>
<th># Rows (millions)</th>
<th>CPU Performance (billions of updates/s)</th>
<th>GPU Performance (billions of updates/s)</th>
<th>FPGA Performance (billions of updates/s)</th>
<th>CPU Performance/$ (billions of updates/s per $)</th>
<th>GPU Performance/$ (billions of updates/s per $)</th>
<th>FPGA Performance/$ (billions of updates/s per $)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Random</td>
<td>300</td>
<td>4.2</td>
<td>24.65</td>
<td>355.68</td>
<td>144.44</td>
<td>15.10</td>
<td>116.24</td>
<td>87.54</td>
</tr>
<tr>
<td>Power $a = 5$</td>
<td>300</td>
<td>4.2</td>
<td>24.47</td>
<td>183.37</td>
<td>145.49</td>
<td>14.99</td>
<td>59.93</td>
<td>88.18</td>
</tr>
<tr>
<td>Power $a = 10$</td>
<td>300</td>
<td>4.2</td>
<td>23.90</td>
<td>118.80</td>
<td>146.43</td>
<td>14.64</td>
<td>38.82</td>
<td>88.74</td>
</tr>
<tr>
<td>Jane Street</td>
<td>133</td>
<td>2.4</td>
<td>21.48</td>
<td>71.03</td>
<td>71.79</td>
<td>13.16</td>
<td>23.21</td>
<td>43.51</td>
</tr>
</tbody>
</table>

Table 1: Performance of each platform on our datasets in billions of updates per second and billions of updates per second per dollar. Bolded entries indicate the best-performing platforms for each dataset.
Subgroup Acceleration Conclusion

- Performant FPGA accelerator for a new analytics problem using the design patterns of streaming and PU replication
- FPGA has consistent performance that doesn’t depend on data distribution, unlike GPU

https://github.com/jjthomas/rule_engine
Talk Conclusion

• By focusing on streaming data-intensive applications we developed
  • A productive and performant framework for streaming FPGA apps
  • A fast compilation flow for highly replicated designs
  • A fast FPGA implementation of an important analytics computation