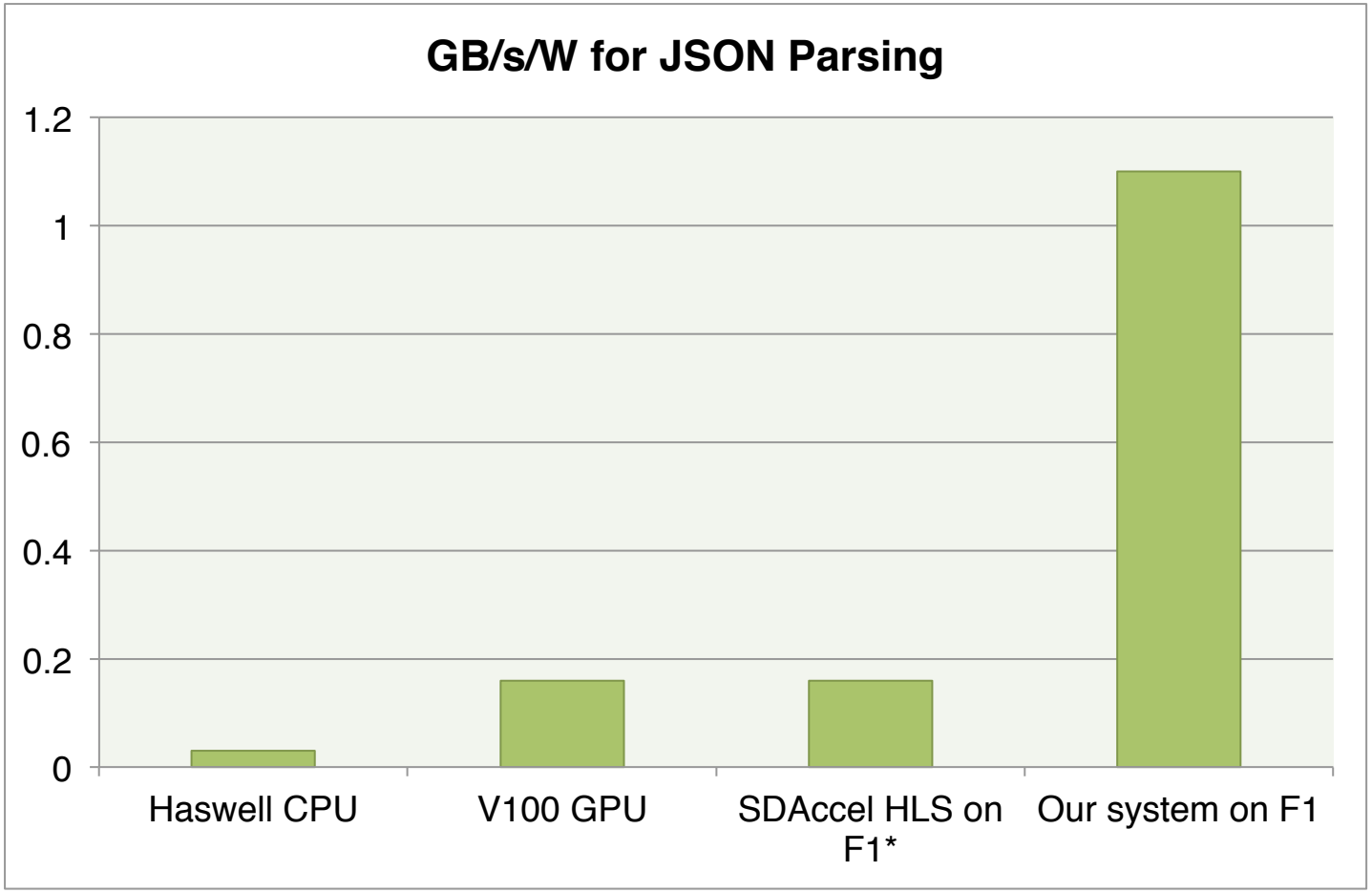
FPGAs as Streaming MIMD Machines for Data Analytics

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



*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:

```
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 `ifs`, `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

```
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

```
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

Stage 1	Stage 2
-Feed `hist_idx` or `input` to `histogram` read address	-Feed appropriate write data to `histogram` -`emit` if necessary -Update registers -If all `pre_while` conditions false (finalizer virtual cycle), load next token into input register

Example Pipeline Scheduling

Stage 1	Stage 2
-Feed `hist_idx` or `input` to `histogram` read address	-Feed appropriate write data to `histogram` -`emit` if necessary -Update registers -If all `pre_while` conditions false (finalizer virtual cycle), load next token into input register

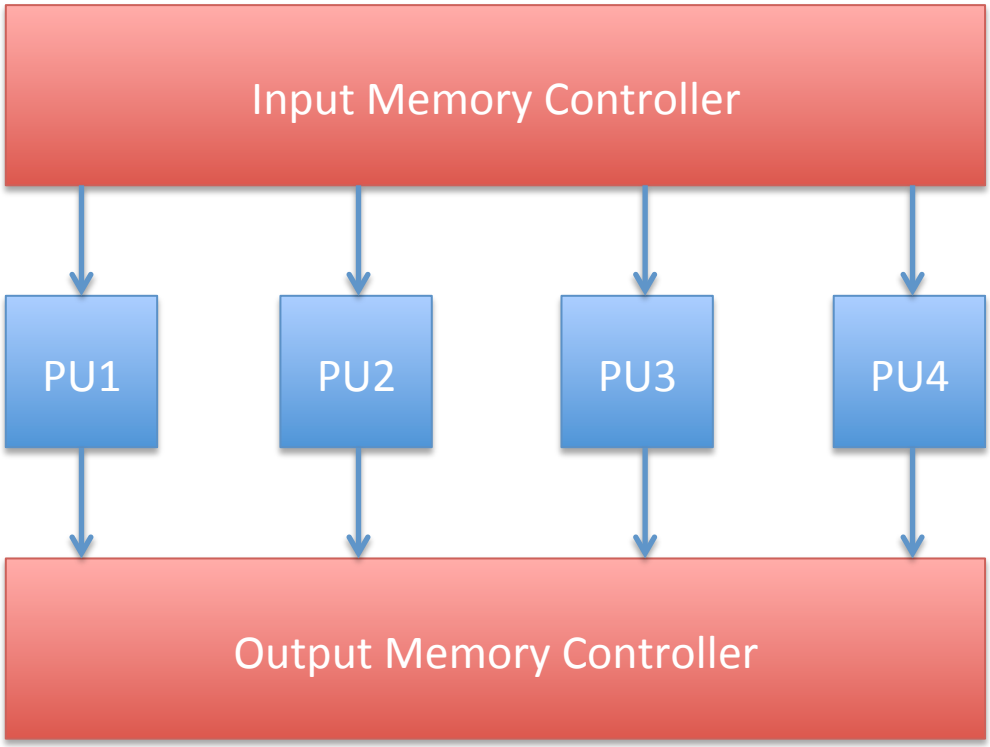
Virtual Cycles: 

Time	Stage 1	Stage 2
1		
2		
3		

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

Synthesis

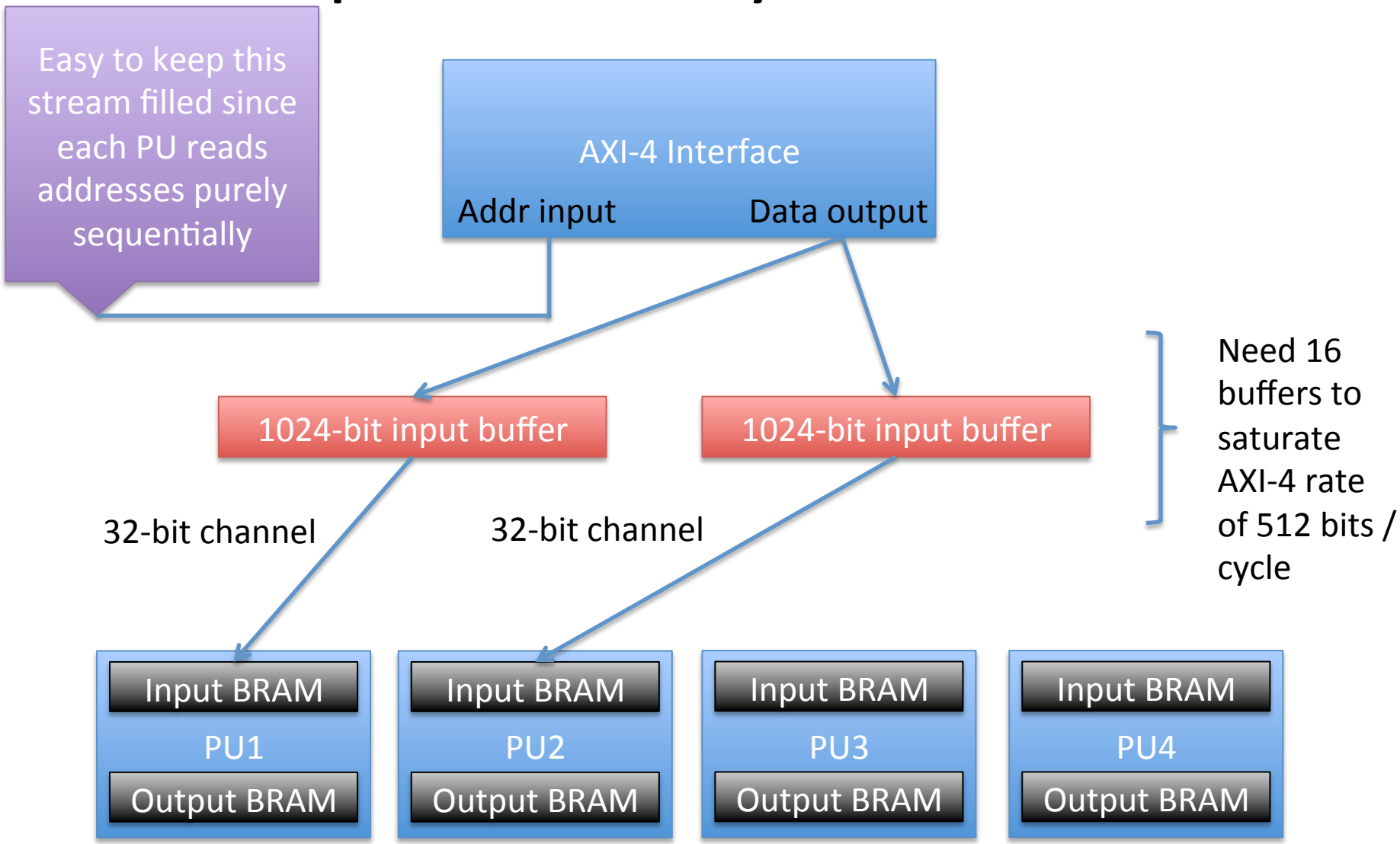
- Synthesize specification for one stream processing unit (PU) into many copies driven by input and output memory controllers



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



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

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

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

	JSON Parsing	Integer Compression	Decision Tree
Our cyles per input token	1	6	2
HLS cycles per input token	7	18	2

Future Directions

- Faster placement and routing due to repeated structure
- Integration with code generation frameworks like Weld

Thanks!

jjthomas@stanford.edu