FSP: a Framework for Data Stream Processing Applications targeting FPGAs

Alberto Ottimo

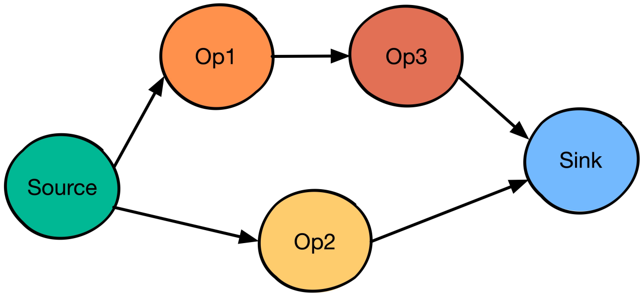
1st year PhD student

Outline

- Context
- FSP Framework
- Running Example
- Evaluation
- Conclusion & Future Work

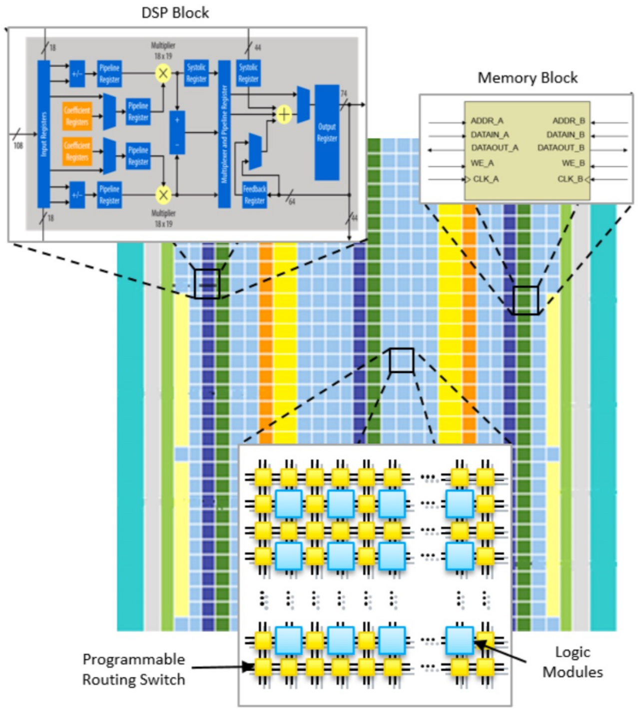
Data Stream Processing

- Unbounded sequences of data items (tuples)
- Stringent requirements
  - High-Throughput (tuples/sec)
  - Low Latency
- Modeled as a Data-Flow graph
- Existing solutions:
  - Apache Flink and Storm for distributed systems
  - WindFlow for multicore shared-memory systems



FPGAS

- Programmable integrated circuits composed by:
  - Programmable Logic Modules
  - Programmable Routing Switches
  - DSP Blocks
  - Memory Blocks

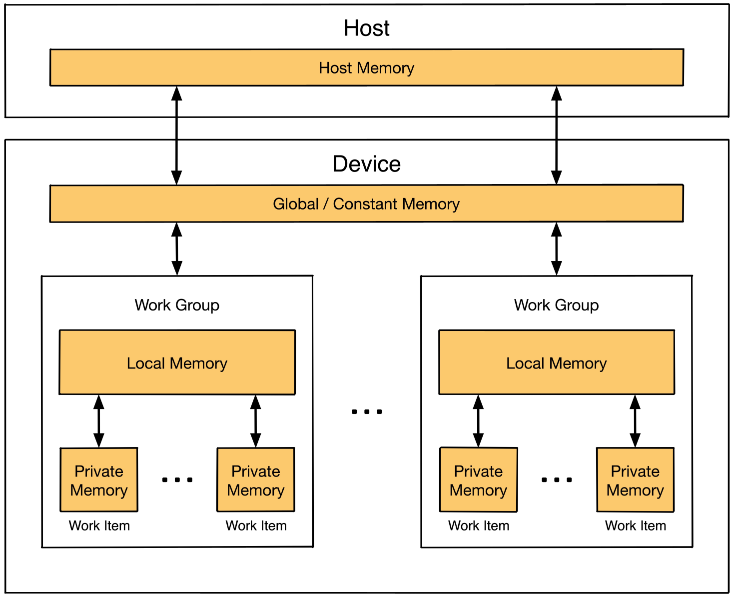


- Synthesis
  - Hardware Description Languages: Verilog and VHDL
  - High Level Synthesis: C/C++ and OpenCL

OpenCL™

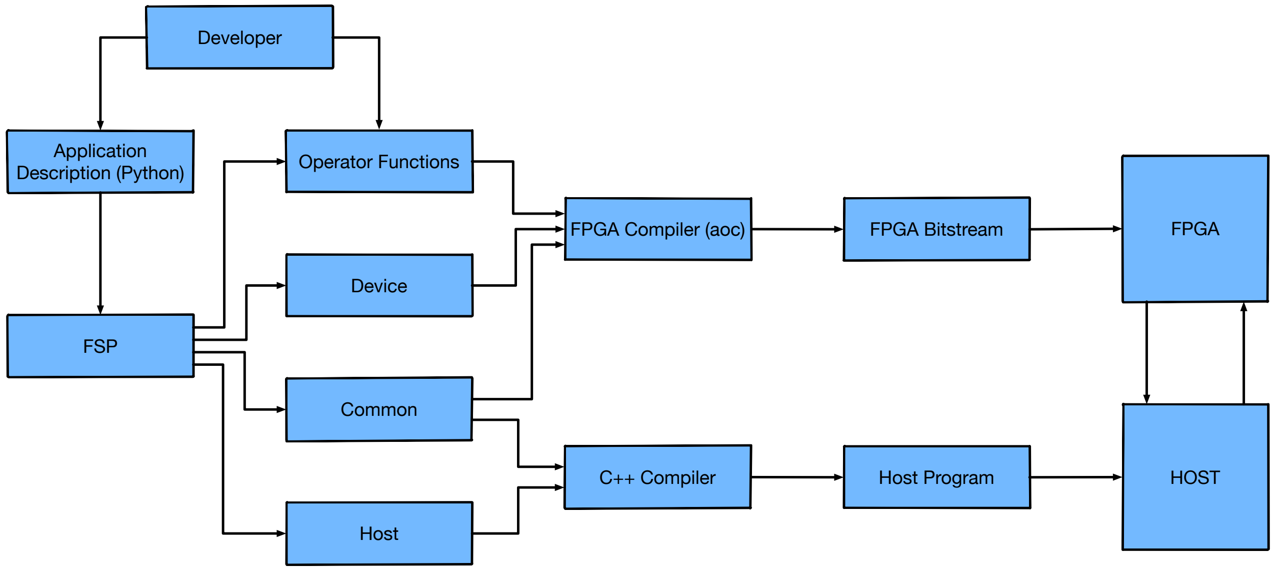
Intel FPGA SDK for OpenCL

- OpenCL provides a framework for parallel programming
  - Platform Model: Host connected to one or more devices
  - Execution Model: Host program and Device kernels
  - Programming Model: Data Parallel and Task Parallel
  - Memory Model:
    - Global / Constant Memory
    - Local Memory
    - Private Memory
- Intel FPGA SDK for OpenCL
  - Single Work-Item programming model
  - Channels Extension



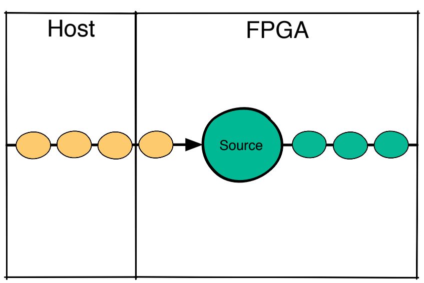
FSP Framework

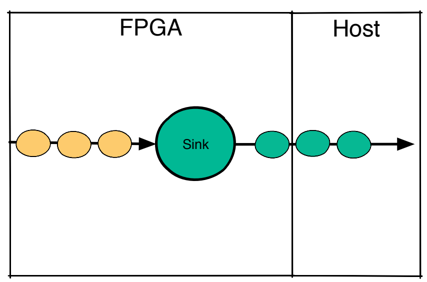
- Develop of DSP application targeting FPGAs:
  - A set of base operators
  - Efficient Host<->Device communication mechanisms
  - Hide the complexity of the implementation of the application structure
- Developer provides:
  - the Application description using our DSL in Python
  - the business logic OpenCL code of operators

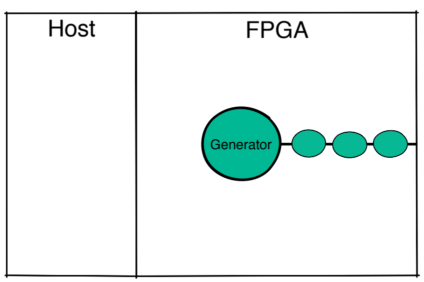


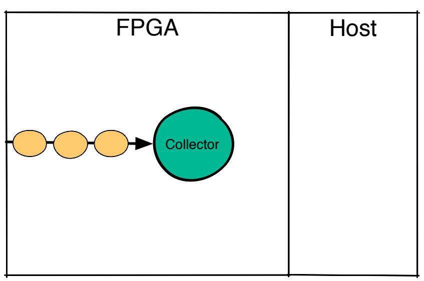
FSP Base Operators

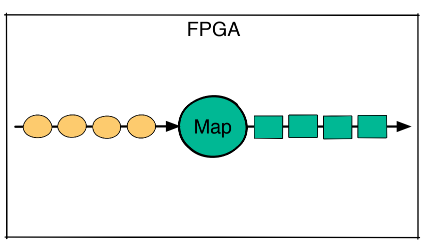
- FSP provides a set of Base Operators
  - Source: distribute tuples received from the Host
  - Generator: generates tuples within the FPGA
  - Sink: make available received tuples to the Host
  - Collector: collects tuples without interact with the Host
  - Map: applies one-to-one transformation. Output datatype can differ
  - Filter: drops tuples if predicate is False, keeps them otherwise

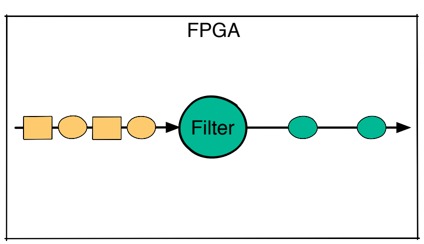






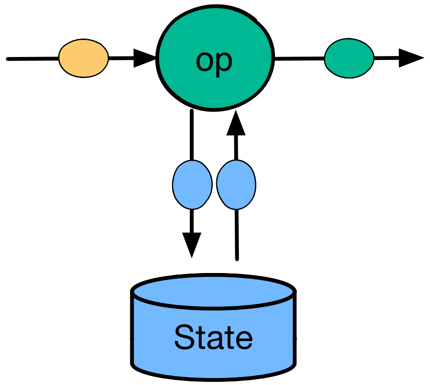


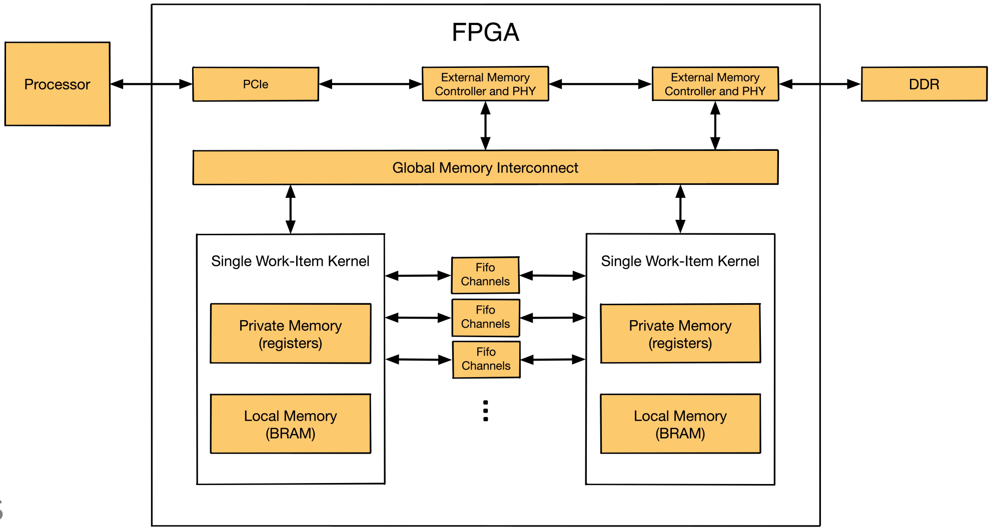




Operators: Stateless and Stateful

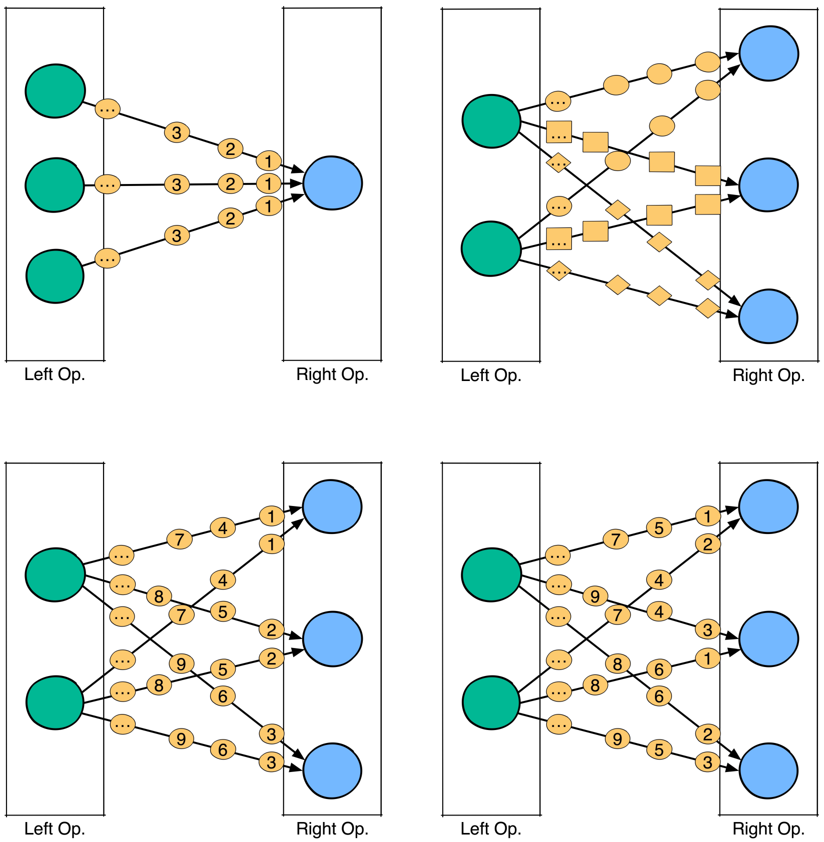
- Operators are Single Work-Item kernels
- Stateless: no needs of information to compute incoming tuples
- Stateful: compute an incoming tuple based on its State
- State implemented as:
  - Private Memory
  - Local Memory
  - Global/Constant Memory
- Global Memory access can be:
  - read/write only
  - both read and write
- Global Memory visibility:
  - one for each replica
  - one shared among all replicas





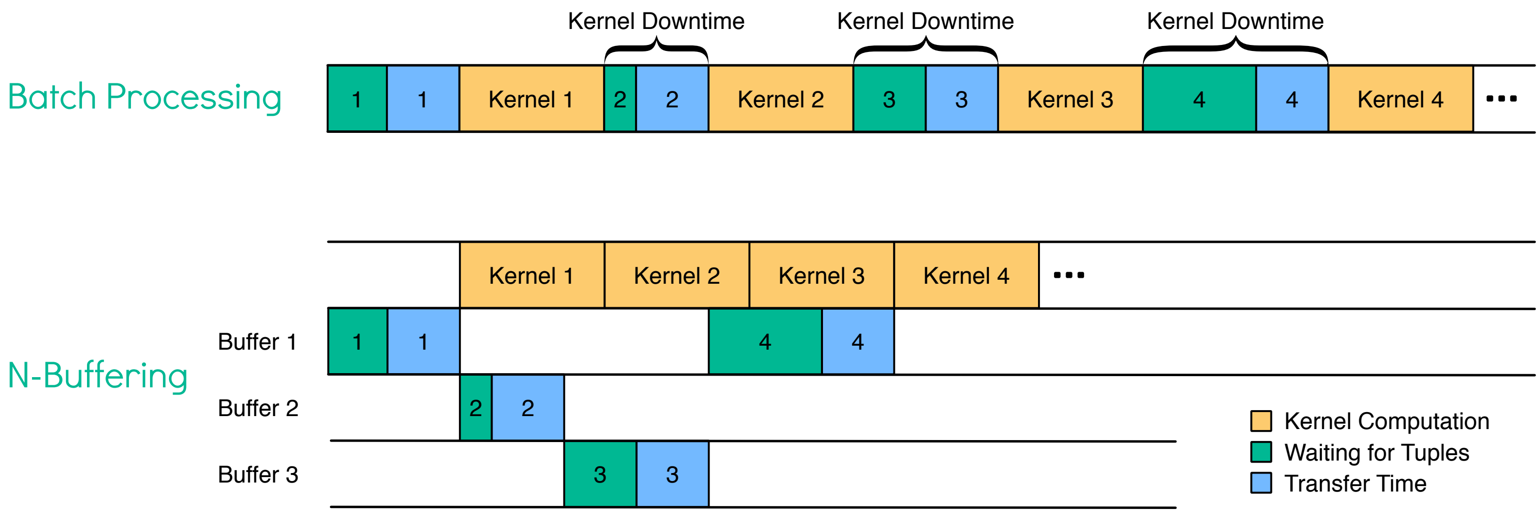
Gather and Dispatch Policies

- Inter-kernel communication by using the Intel Channels extension
- Gather Policy:
  - Blocking Mode
  - Non-Blocking Mode
- Dispatch Policy:
  - Forward
  - RoundRobin
    - Blocking mode
    - Non-Blocking mode
  - KeyBy
  - Broadcast



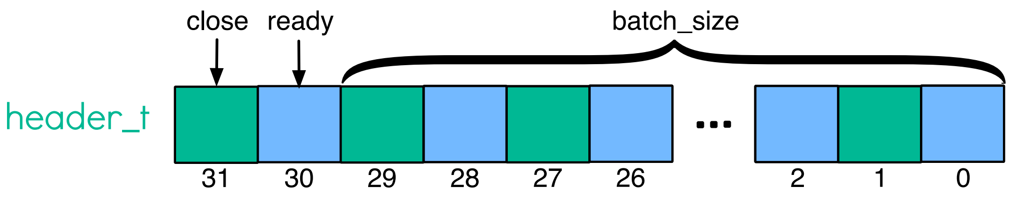
Host<->Device | N-Buffering

- Batch processing:
  - one buffer
  - long kernel downtime between kernel launches
- Stream processing:
  - N buffers recycled in circular manner
  - minimal/zero kernel downtime between kernel launches
  - good for variable arrival rates



Host<->Device | Shared Memory Protocol

- No use of OpenCL read/write buffers
- Exploits Shared Memory between CPU and FPGA
- Two circular buffers:
    - headers buffer
    - batches buffer



Host Source operator

```
void push(const tuple_t * batch, const size_t size, const bool close)
{
    while (header_ready(headers[id]);
    // write tuples on the batch
    WRITE_MEMORY_BARRIER();
    headers[id] = header_new(close, true, batch_size);
}
```

Device Source operator

```
__kernel source(__global volatile header_t * restrict headers,
                __global const volatile tuple_t * restrict batches)
{
    while (!done) {
        header_t h;
        while (!header_ready(h = headers[id]));
        // read items of the batch
        done = header_close(h);
        mem_fence(CLK_GLOBAL_MEM_FENCE);
        headers[id] = header_new(false, false, 0);
    }
}
```

DSL PYTHON APIS

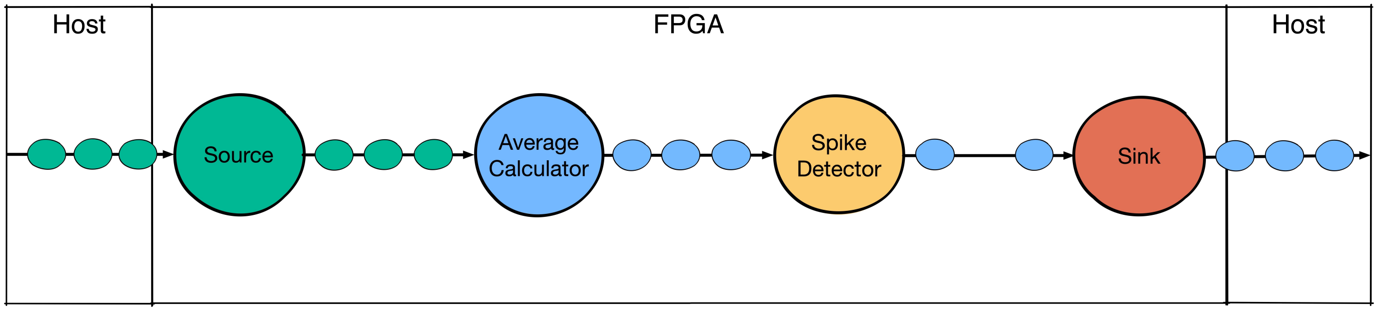
Domain Specific Language in Python

- FNode
  - parallelism
  - gather/dispatch policy
  - output datatype
  - phase functions (begin, compute, end)
  - add private/local/global buffer
  - add RNG state (only Generator Operator)
- FPipe
  - directory to generate code
  - input datatype of the Source Operator
  - host <-> device communication protocol
  - codebase directory

```
FNode(name,
      parallelism,
      node_kind,
      gather_policy,
      dispatch_policy,
      datatype,
      channel_depth,
      begin_function,
      compute_function,
      end_function)
```

```
pipe = FPipe('./codedir', 'tuple_t')
pipe.add_source(source_node)
pipe.add(map_node)
pipe.add(filter_node)
pipe.add_sink(sink_node)
pipe.finalize()
pipe.generate()
```

Running Example | Spike Detection



Several sensors produce information regarding temperature

Tuple format: {device_id, temperature}

Pipeline of 4 stages:

- Source
- Average Calculator: calculates the average over a window of tuples
- Spike Detector: checks the predicate $|x_n - \mu_n| > (threshold\ * \mu_n)$
- Sink

Device Program Implementation

Developer Python Description

```python
avg_node = FNode('average_calculator',
                 avg_par,
                 FNodeKind.MAP,
                 FGatherMode.NON_BLOCKING,
                 FDispatchMode.RR_BLOCKING,
                 'tuple_t')
avg_node.add_private_buffer('int', 'sizes',
                            size=avg_keys)
avg_node.add_local_buffer('float', 'windows',
                          size=(avg_keys, win_dim))
```

OpenCL operator generated code

```c
__attribute__((uses_global_work_offset(0)))
__attribute__((max_global_work_dim(0)))
__kernel void avg_kernel(...)
{
    __private int sizes[AVG_KEYS];
    __local float windows[AVG_KEYS][WIN_DIM];
    bool done = false;

    // call of the begin phase function

    while (!done) {
        // gather component
        // call of the compute phase function
        const tuple_t result = avg_compute(...);
        // dispatch component
    }

    // call of the closing phase function
    // send End-Of-Stream to the next operator replicas
}
```

Developer Compute Phase Function implementation

```c
inline tuple_t avg_compute(input_t in,
                           __private int sizes[AVG_KEYS],
                           __local float windows[AVG_KEYS][WIN_DIM])
{
    const uint idx = in.device_id / __AVERAGE_CALCULATOR_PAR;
    const float val = in.temperature;

    if (sizes[idx] == WIN_DIM - 1) {
        sizes[idx] = WIN_DIM;
    } else {
        sizes[idx] += 1;
    }

    float sum = 0.0f;
    #pragma unroll
    for (uint i = 0; i < WIN_DIM - 1; ++i) {
        windows[idx][i] = windows[idx][i + 1];
        sum += windows[idx][i];
    }
    windows[idx][WIN_DIM - 1] = val;
    sum += val;

    tuple_t out;
    out.device_id = in.device_id;
    out.temperature = in.temperature;
    out.average = sum / sizes[idx];

    return out;
}
```

Evaluation: Applications

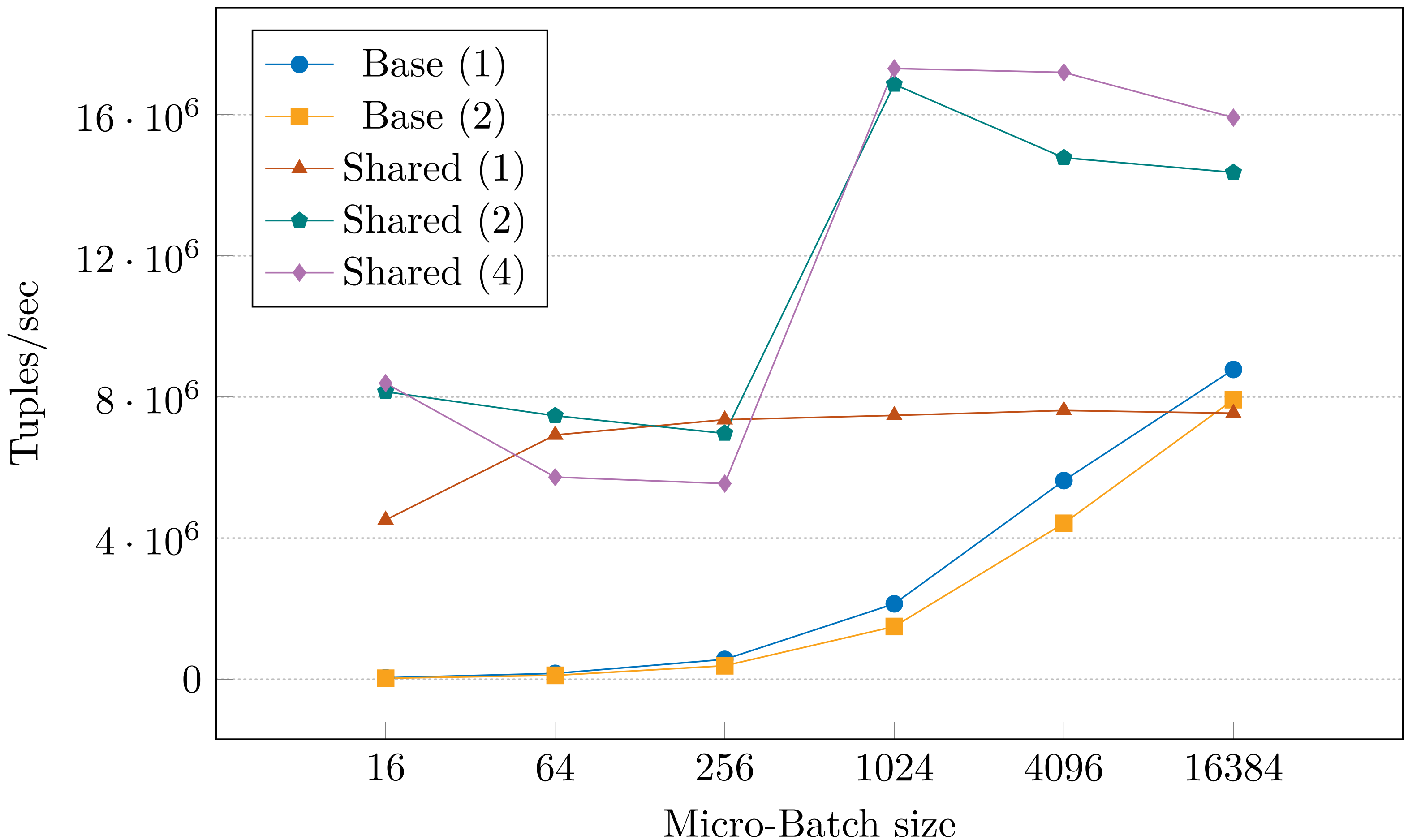
Four version of Spike Detection application:

- Base: using the N-Buffer technique for Host<->Device communication
- Shared: using the Shared Memory Protocol for Host<->Device communication
- Skeleton: Generator in place of Source op. and Collector in place of Sink op.
- WindFlow: application implemented using the WindFlow library

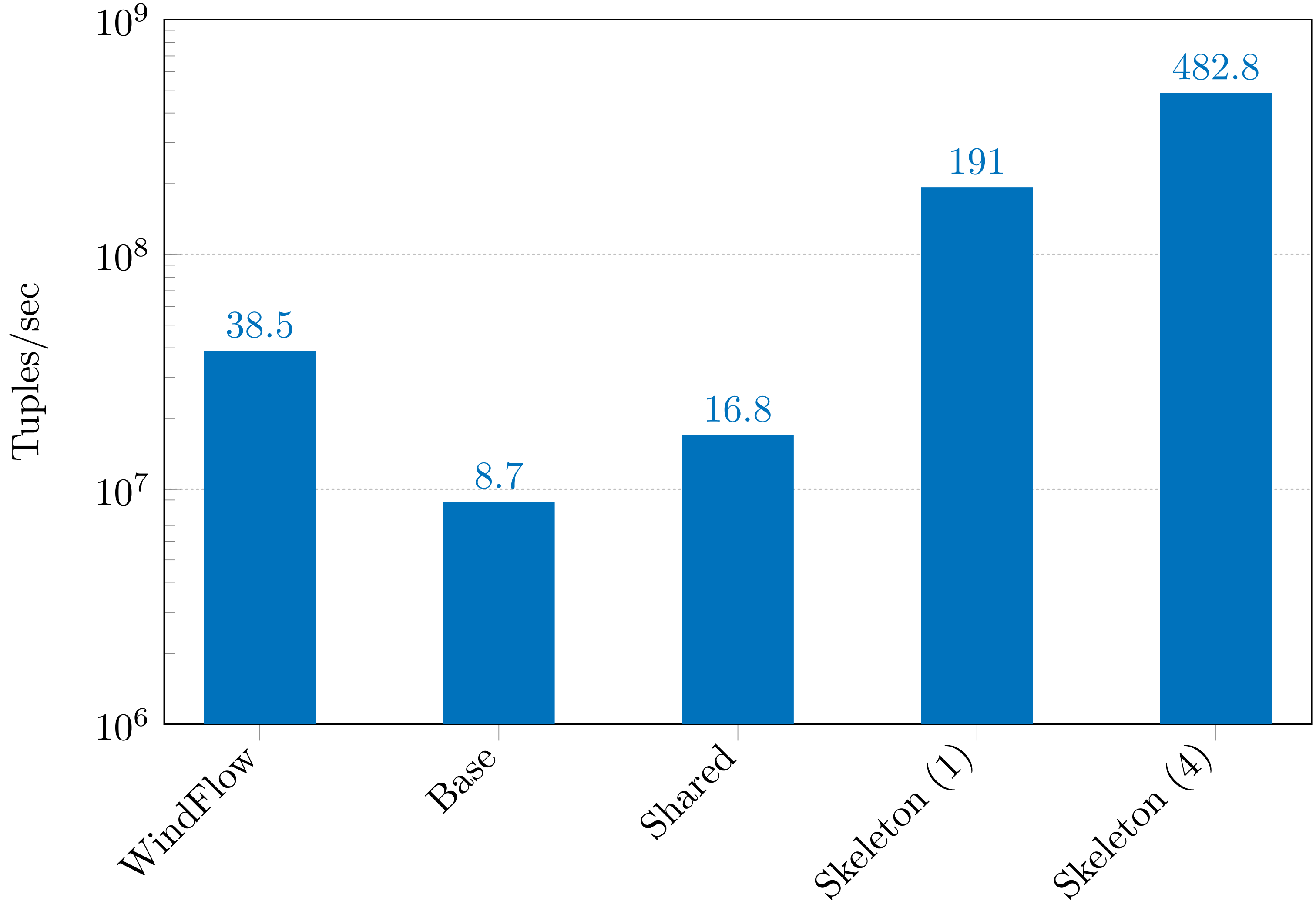
Hardware Configuration

- Intel Arria 10 SoC FPGA (dual-core ARM Cortex-A9, 1GB DDR4-2200)
- 2x AMD EPYC 7551 32 cores (64 threads) with 128 GB of RAM

Evaluation: Base vs Shared



Evaluation: FSP vs WindFlow



Conclusions & Future Work

FSP enables programmers to develop DSP application targeting FPGAs

- a set of Base Operators
- different ways to manage operator state
- two Host<->Device communication protocols

Our tests demonstrate the potential of adopting FPGAs for DSP applications

Future Work

- Provide more Base Operators (FlatMap, Windowing Operators)
- Improve Shared Memory Protocol
- Tests on different hardware configurations

Available on GitHub: https://github.com/blackwut/FSP

Thank You!

Any Questions?