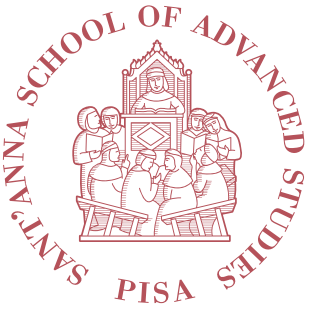


Master Thesis

FSP: a Framework for Data Stream Processing Applications targeting FPGAs

Author

Alberto Ottimo

Supervisor

Prof. Gabriele Mencagli

Master's Degree in Computer Science and Networking

October 08, 2021

Contents

List of Figures

Listings

Introduction

The Data Stream Processing (DSP) paradigm studies novel algorithms and parallel processing techniques to process continuous streams of high-speed data. Based on the application domain, this might require to maintain stringent performance requirements such as high throughput, in terms of processed inputs per second, and low latency. In recent years, state-of-art DSP engines like Apache Storm [1], Apache Flink [2] have been release to provide user-friendly solutions for scale-out environments (i.e. clusters of homogeneous machines). More recently, streaming libraries for single machines (i.e. scale-up servers with several multi-core CPUs) have been released in order to achieve higher performance on these computing platforms. One of them, WindFlow [3], has been developed by the Parallel Programming Models group of the University of Pisa, Department of Computer Science.

In this thesis, we propose a novel FPGA-based Stream Processing (FSP) framework that enables developers of DSP applications to employ FPGA co-processors to accelerate streaming applications or some of their critical parts. The framework is characterized by a high-level Python API to describe DSP processing pipelines of intermediate transformations (operators) with high-level abstractions. From this description, the OpenCL code is generated using `Jinja 2.11` [4] as a templating engine for both the host program and for the device program. The generated code for the host program manages the streams and the computation onto the FPGA device. The device code is generated according to the provided high-level description of their operators and their interconnections, and it is completed by the developer with the business logic code of each operator. To present the framework API, we use a previously studied streaming benchmark, named SpikeDetection, in order to describe how the framework can be used and which kind of performance can be achieved using it.

In this thesis, we outline the most important components needed to construct a DSP

application for the FPGA and we describe the implementation solutions adopted during this work. The implementation of the operators is based on the Single-Work Item programming model, which is one of the programming models provided by OpenCL and strongly recommended by Intel for FPGA-based accelerators. In order to increase their processing bandwidth, operators of streaming applications can be replicated according to a parallelism degree chosen by the programmer with the high-level API. To proper replicate an operator, we adopt a code replication approach that is easily achievable with the code generation and gives us flexibility and the possibility of optimizations. The communication between operator replicas is a central part of the design, in order to achieve satisfactory performance. We use the channels provided by the Intel FPGA SDK for OpenCL Channels Extension. This mechanism allows to easily and quickly exchange data between kernels through FIFO buffers implemented directly on the FPGA fabric, without the need to use memories with high latencies such as the Global Memory. The programmer can choose the dispatching policies, i.e., how the outputs produced by an operator are sent to the next operator in the pipeline and to its internal replicas. Finally, two solutions are presented concerning the communication between the host and the device to properly manage streams. The first one uses the basic APIs provided by OpenCL, while the second one is specific for hardware solutions in which there is a shared memory between the host CPU and the device FPGA board. The first implementation is designed for FPGAs connected via PCI-Express interfaces and uses an N-Buffering technique to overlap the device computation with the data transfers. The second one tries to keep the overhead as low as possible in order to minimize or completely remove the communication overhead. A custom protocol has been designed to exploit the shared memory, which is accessible by both the host and the device at the same time.

Finally, in the final stages of this study we employ some benchmarks to show the efficiency of the adopted solutions, as well as to offer a comparison with the same application developed using WindFlow and targeting a large server machine.

Document structure

This chapter outlined the reasons that led to the creation of this thesis and gave a brief introduction to our work.

Chapter 1 introduces the basic concepts of the stream processing paradigm and outlines the current state-of-art DSP engines and their main approaches to facilitate the programmer with the implementation of DSP applications. In addition, it shows the FPGA architectural models and the abstraction levels needed to program such kinds of devices, focusing on the OpenCL programming model and in particular on the Intel FPGA SDK for OpenCL programming model that will be used in this work.

Chapter 2 presents the implementation of the main components of a generic DSP application targeting FPGAs, including how to model the operator computation, how to manage the state of stateful operators, the inter-kernel communication and the communication between host and device to move data on the streams.

Chapter 3 describes how our FSP framework works and how a sample application can be implemented leveraging our programming model, going from the high-level description to the code generation and to the implementation of the operators business logic to complete the implementation.

Chapter 4 shows the benchmark results of the Spike Detection DSP application implemented with our framework targeting the Intel Arria 10 SoC FPGA. Benchmarks show how the two communication mechanisms designed during our work behave with the use of different micro-batch sizes. Furthermore, we compare our best results against the ones of the WindFlow implementation of Spike Detection, which runs on a large server machine.

Finally, Chapter 5 concludes this work by summarizing the objectives, contributions and possible future goals of this study.

Chapter 1

Background

1.1 Data Stream Processing

In recent years the number of interconnections have significantly increased: data streams are generated by an infinite quantity of sources such as Internet of Things sensors, servers, applications and many more; this data comes with all types of volumes and formats, from various locations and clouds. In such complex context, legacy technologies are obsolete and can't guarantee low-latency and high-throughput that are fundamental to nowadays applications [5]. The answer to these requirements is a new model of data management, exploiting data as *streams*: Data Stream Processing (DSP) is the paradigm that continuously analyzes and processes data, monitoring and providing new information in real-time. Streams are sequences of single elements, or *segments*, belonging to the same logical data structure. The defined continuous computation (*query*) ingests events and produces results in a continuous manner, whose operators must be executed "on the fly" [6]. These continuous queries over streams can be programmed by using the abstractions provided by the currently available Data Stream Processing Systems, which are based on the Data-Flow programming paradigm. Indeed, the main approach to design Data Stream Processing Applications is by using Data-Flow direct graphs. Each vertex of the graph represents a task or computation, called *operator*, and every edge models the stream defining dependencies.

An operator is the basic functional unit of an application. In general, an operator processes input data from incoming streams, applies an arbitrary function, and emit results towards one or more output streams for further processing. Operators without

input ports are called *data sources* and operators without output ports are called *data sinks*. A Data-Flow graph must have at least one data source and one data sink.

Operators can be classified depending on their state [7]:

- *stateless operator* which works on a item-by-item basis in a streaming scenario and processes a new item without any information about past processed data.
- *stateful operator* which require instead proper mechanisms to maintain and update a set of internal data structures while processing input data, and the result of the internal processing logic also depends on the current value of the state.

A special case of stateful operators is the *partitioned stateful operator*: while this type of operator manages an internal state, the data structures supporting the state can be separated into independent partitions, and each partition corresponds to a data substream multiplexed together to a unique stream. The correspondence between elements and substreams is made by applying a predicate on a partitioning attribute called *key* or a unique combination of the data item attributes. An example of operators of this kind are the ones that process network traces partitioned by sender and receiver IP address or the ones that analyze trading transactions from a stock exchange partitioned by the company symbols.

Over the years, Stream Processing Engine (SPE) have been proposed as they allow developers to have more flexibility in defining their applications without dealing with low level implementations. Among them, we remember Apache Flink [2] and Apache Storm [1], which are state-of-art solutions for distributed systems of homogeneous machines. Furthermore, we find WindFlow [3], a C++ library for parallel data stream processing targeting heterogeneous shared-memory architectures.

1.2 Apache Storm

Apache Storm is a framework for real-time data processing and the main scopes of the framework can be summarized as follows [8]:

- Stream processing
- Continuous computation

- Distributed remote procedure call

Storm is *highly-scalable*, *fault-tolerant*, i.e., it can reassign tasks of failed nodes, *reliable*, i.e., re-sending lost data, and *language agnostic*.

The Storm stream processing system provides an abstraction layer that allows programmers to manually build the processing graph of workers and their connections needed for real-time processing. The abstraction level is given by the Storm concept of *topology* which is the data-flow graph of the desired application built with processing logic nodes and links. The final representation of the topology, as shown in figure 1.1 is an acyclic-graph that contains two different type of nodes:

- *Spouts*: the sources of the stream;
- *Bolts*: the processing logic units that transforms input data and produce output streams.

Each node in the topology is executed in parallel, specifying for each node, the desired parallelism degree. The topology can be submitted to a Storm cluster for the real processing of data streams.

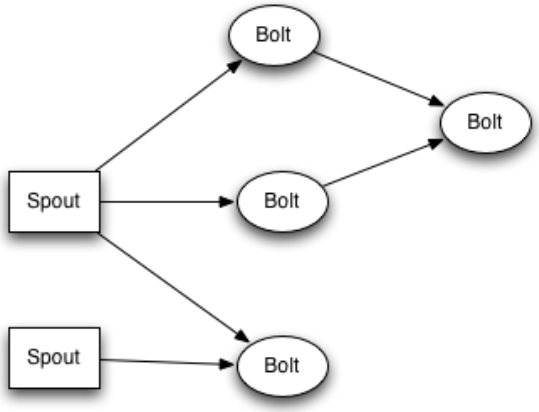


Figure 1.1: High-Level view of a Storm Topology.

A stream in Storm is defined as a sequence of *tuples* in the form of key-value pairs. Bolts can subscribe to specific streams to receive tuples from them, while both spouts and bolts emit tuples to a stream which is sent to every bolt that subscribed to it.

Storm provides stream groupings as a way to define how to send tuples between parallel entities. A stream grouping defines the routing of the tuples between the parallel replicas (i.e., tasks) of each operator. Among them we remember:

- *Shuffle Grouping*: tuples are distributed uniformly at random among the receiver bolt's tasks.
- *Fields Grouping*: tuples are distributed according to a subset of its fields. It is also guaranteed that tuples with equal values for that subset of fields go to the same replica of the bolt operator.
- *Custom Grouping*: the user defines a specific routing strategy to forward tuples.

At the task level, a topology can look like the one in the figure 1.2.

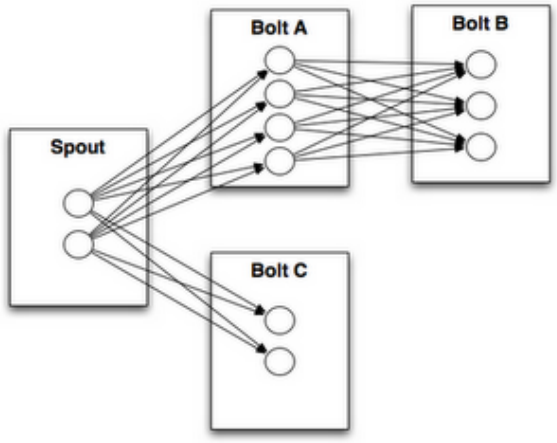


Figure 1.2: Task-Level view of a Storm Topology.

1.3 Apache Flink

Flink is a framework and a distributed processing engine for stateful computations developed by the Apache Software Community, written in Java and Scala. It provides APIs for both stream processing and batch processing use cases simultaneously, based on fluent interfaces. A fluent interface is designed to increase code readability, exploiting *method chaining* where return values from methods are used to relay instruction context.

Flink offers multiple levels of abstraction, shown in Figure 1.3, providing the two concepts which are the basement of the framework: stateful and timely stream processing.

The lowest levels provide stateful streaming, allowing users to process events from one or more streams and use consistent fault tolerant state. This lower lever is not always necessary and the programmers can directly use the `DataStream` API (bounded/unbounded

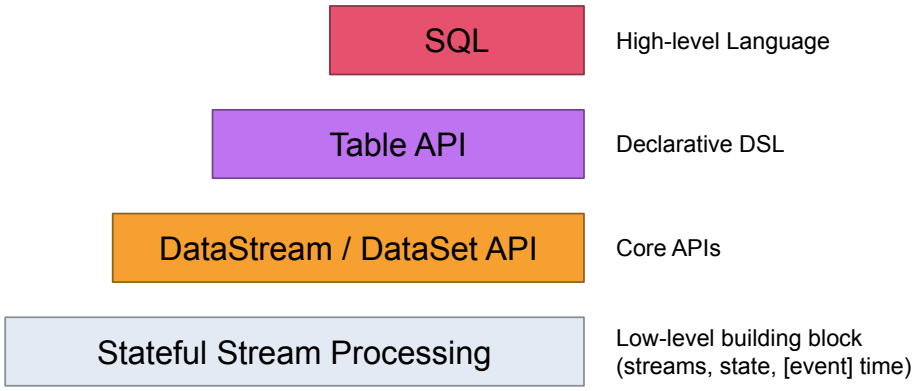


Figure 1.3: Levels of abstraction offered by Flink.

streams) and the `DataSet` API (bounded data sets). These fluent interfaces offer the common building blocks for data processing (e.g., user-specified transformations, joins, aggregations, windows, states, etc.) and data types represented as classes. The `Table` API is an extended version of the relational model with tables with a schema and comparable operations (e.g `select, project, join, group-by, aggregate`, etc.). The highest level is the SQL language which is similar to the previous layer but represents programs as SQL queries. It can also interact with the `Table` API underlying layer executing queries over tables defined in it.

Flink provides different predefined operators such as `Map, Filter, Reduce`, etc., and it also provides an extensive support for windowed operators, which made *a-tuple-at-a-time* transformations, that can be defined over *keyed* and *unkeyed* data stream. Flink offers the classical *tumbling/sliding* and *count/time* based windows. Other types of window can be defined by specifying the eviction and trigger policies.

Flink has a completely distributed and efficient architecture mainly targeting clusters of heterogeneous node. The runtime consists of two types of processes:

- the *JobManager* coordinates task scheduling and reacts to finished tasks or execution failures.
- one or more *TaskManagers*, also called workers, are the components that execute the tasks of a data-flow graph, buffering and exchanging the data streams.

In Figure 1.4 we show the architecture during the runtime execution of a Flink application. The execution of a Flink program can happen in a local Java Virtual Machine (JVM) or on clusters of many machines.

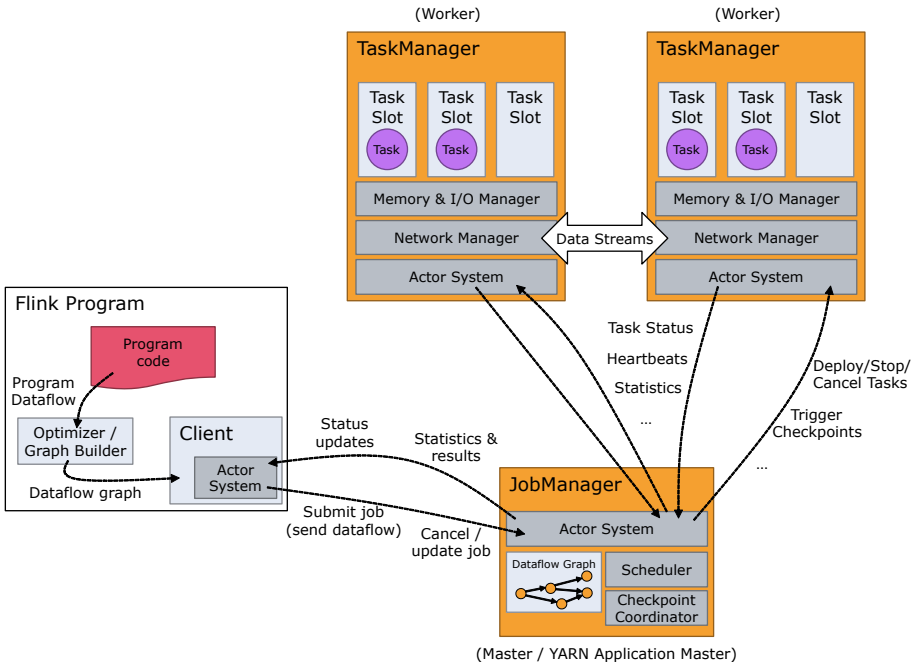


Figure 1.4: Runtime architecture of Flink.

When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each stream is split in more stream partitions, and each operator has one or more operator subtasks, which execute independently and in different threads or machines.

The number of on operator subtasks is called parallelism of that particular operator. The parallelism of a stream always coincides to that of its source, but different operators may have different parallelism.

We can distinguish two different *patterns* for transporting data between two operators:

- *One-to-one (or forwarding)*: it preserves partitioning and ordering of the elements, meaning that the subtask i of the receiver operator will see the same elements in the same order, as they were produced by subtask i of the source operator. An example of this pattern is shown in figure 1.5 between the Source operator and the `map` operator;

- *Redistributing*: it changes the partitioning of stream since each subtask sends data to different target subtasks, depending on the selected transformation. In this pattern the order among elements is only preserved for each pair of sending-receiving task. Examples are:

  – `keyBy`: re-partitions by hashing (i.e., assigning all records with the same key

to the same partition, and to the same receiver's subtask);

- `rebalance`: distributes the data in a round-robin fashion, creating equal load per partition (i.e, balanced work-load across the receiver's subtasks in this case of infinite input streams);
- `shuffle`: randomly spreads elements according to a uniform distribution, balancing the load among the receiver's subtasks;
- `broadcast`: broadcasts elements to every subtask of the receiver operator.

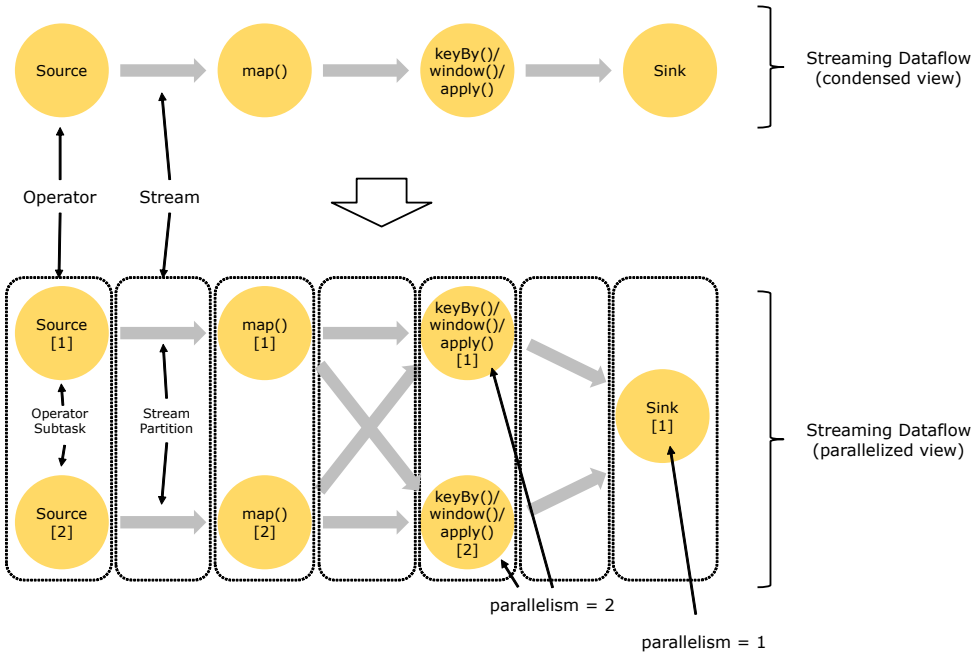


Figure 1.5: Parallelized view of the Flink streaming Data-Flow.

1.4 WindFlow

WindFlow is a C++17 application DaSP parallel library, developed by the Parallel Programming Models group at the Department of Computer Science of the University of Pisa [1]. It is built on top of FastFlow [9] C++ library, and it allows to create parallel streaming applications providing several parallel patterns (operators). The main operators that can be used to compose the desired application (Storm-like approach) are the following:

[1]This is the official web site of the PPMs research group: http://calvados.di.unipi.it/paragroup

- `Source`: produces a stream of items characterized by the same data type;
- `Sink`: absorbs the input stream and possibly stores the received results of the computation into files or databases;
- `Map`: applies a transformation on each received item, producing exactly one output item for each processed item;
- `Filter`: applies a boolean predicate to each input, dropping all items of the stream for which the predicate evalutes `False`;
- `FlatMap`: applies a transformation on each input item, producing zero, one or more than one output per each input data;
- `Accumulator`: applies a fold function on the received inputs partitioned by key. It combines the current item with the last reduced values.

Furthermore, different windowed operators can be employed: *Keyed Farm* executes a windowed query in parallel on different key partitions, while windows relative to the same key are processed sequentially by the same parallel entity; *Windowed Farm* executes a windowed query in parallel on distinct streaming windows; *Paned Farm* executes a windowed query in parallel by exploiting window-overlapping, splitting consecutive windows into panes; *Windowed MapReduce* executes a windowed query in parallel by exploiting data parallelism within each window. For further information, please refers to [10]. Once the programmer have built the functional logic chain of the computation between operators, they are grouped into a `MultiPipe`. Each operator of the `MultiPipe` has its own parallelism (number of operator replicas), and each operator can be connected with the next one in two different ways:

1. *direct connection*: one replica of the operator is connected with one replica of the next operator;
2. *shuffle connection*: one replica is connected with all replicas of the next operator, the shuffling can be random, key-based, window-based or pane-based depending on the type of the operators.

The following figure 1.6 shows an example of the `MultiPipe` structure in which there is a *direct connection* between the Source operator and the A operator, while the others are *shuffle connections*.

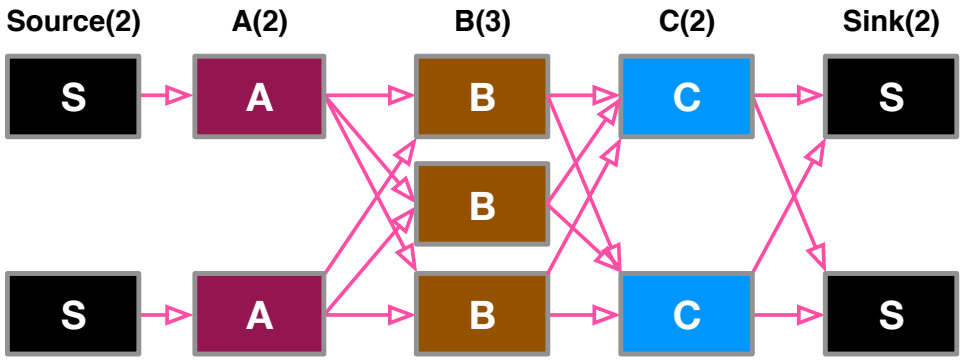


Figure 1.6: Example of MultiPipe structure.

1.5 Field Programmable Gate Array (FPGA)

An FPGA is a programmable integrated circuit composed by an array of programmable logic blocks and routing switches. The main building block of an FPGA is the *basic logic element* (BLE), which has a *look-up table* (LUT) and a *flip-flop* (FF). The LUT can implement any logic function by programming its *SRAM* bits, and it can either feed or bypass the FF via programmable multiplexer. Modern FPGAs have a hierarchical architecture in which several BLEs are grouped into logic clusters referred to as *logic array blocks* (LABs) in Altera device or *slices* and *configurable logic blocks* (CLBs). They employ an island-style architecture in which LABs are arranged into a two dimensional mesh with rows and columns of *routing channels*. *Switch blocks* connect wires of adjacent channels through programmable switches and *connection blocks* have programmable switches that connect the LAB's inputs and outputs to the wire segments surrounding it. BLEs can be connected via a series of wire segments by configuring the SRAM bits of programmable switches in the connection blocks and routing switches. Moreover, FPGAs contain other programmable blocks: I/O blocks to communicate with external devices such as micro-controller or external memories; memory blocks (BRAM) which are configurable random access memory used to store data and to transfer data between on-chip resources; dedicated hard blocks such as *digital signal processing* (DSP) block that has dedicated circuitry that implements multiply and accumulate operations.

FPGA Synthesis

To create an FPGA design, the application is described using a Hardware Description Language (*HDL*) like *Verilog* or *VHDL*, and converted into a *bitstream* using a synthesis

tool which follows 8 steps:

1. RTL elaboration: generates a *netlist*, that is a set of elements (FFs, LUTs, etc.), pins (inputs and outputs), and interconnections (routing channels and switches), needed to implement the given HDL behaviour;

2. Logic synthesis: optimizes the usage of logic gates in the netlist in terms of area occupancy and delay;

3. Technology mapping: since the netlist is hardware-independent, in this step it is converted into a netlist that depends on the logic units included in the specific target FPGA;

4. Clustering: forms the LABs, grouping the BLEs mapped in the previous step;

5. Placement: physical association between the netlist and the resources of the FPGA;

6. Routing: configuration of the interconnection network, i.e., routing channels and switchboxes;

7. Timing analysis: determines each needed timing characteristic, such as the maximum clock frequency;

8. Bitstream generation: the last step produces the bitstream to configure all FPGA programmable units needed by the netlist.

As designs became more and more complex and the time-to-market pressures increased, developers and the vendor community have strived to provide more software-based tool chains to help reduce development times; one of these techniques is *high level synthesis* (HLS), which can be thought of as a productivity tool for hardware designs. It typically uses C/C++ source files to generate RTL that is, in most cases, optimized for a particular target FPGA device. In recent years, Intel has developed an SDK for programming its FPGA devices in OpenCL. OpenCL (Open Computing Language) [11] is an open and royalty-free standard for programming heterogeneous systems in a host/device fashion.

1.5.1 OpenCL

OpenCL provides a framework for parallel programming and includes a programming language, APIs, libraries and a runtime system to support software development. The OpenCL standard describe its core ideas using a hierarchy of models:

- Platform Model
- Execution Model
- Programming Model
- Memory Model

The Platform model consist of a *host* connected to one or more *devices*. A device is divided into one or more *compute unites* (CUs) wich are further divided into one or more *processing elements* PEs. An OpenCL application runs on a host according to the models native to the host platform; the host submits commands to execute computation on the processing elements within a device.

The execution of an OpenCL program occurs in two parts: *kernels* that execute on devices and a *host program* that executes on the host. The host program defines the context for the kernels and manages their execution. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel, called *work-item*, executes for each point in this index space. Work-items are organized into *work-groups* to provide a more coarse-grained decomposition of the index space. The index space supported in OpenCL is called *NDRange*, a N-dimensional index space, where N is one, two, or three. Each work-item can be uniquely identified by a *global ID*, relative to the NDRange index space, or a combination of *local ID* and work-group ID, respectively the ID within the work-group and the work-group where it resides.

A wide range of Programming Models can be mapped onto this Execution model, but OpenCL support only two of these: the data parallel programming model and the task parallel programming model.

The Memory Model defines four distinct memory regions:

- Global Memory. This memory region permits read/write access to all work-items in all work-groups. Usually this type of memory is generally the largest but the slowest memory that a device can access.

- Constant Memory. A region of global memory that remains constant during the execution of a kernel. This memory is usually cached to guarantee lower latency access.
- Local Memory. This memory region is visible only by work-items within a work-group. It can be used to allocate variables that are shared by all work-items in that work-group. Usually it is implemented as low latency memory but with limited capacity.
- Private Memory. A region of memory private to a work-item, usually implemented as registers.

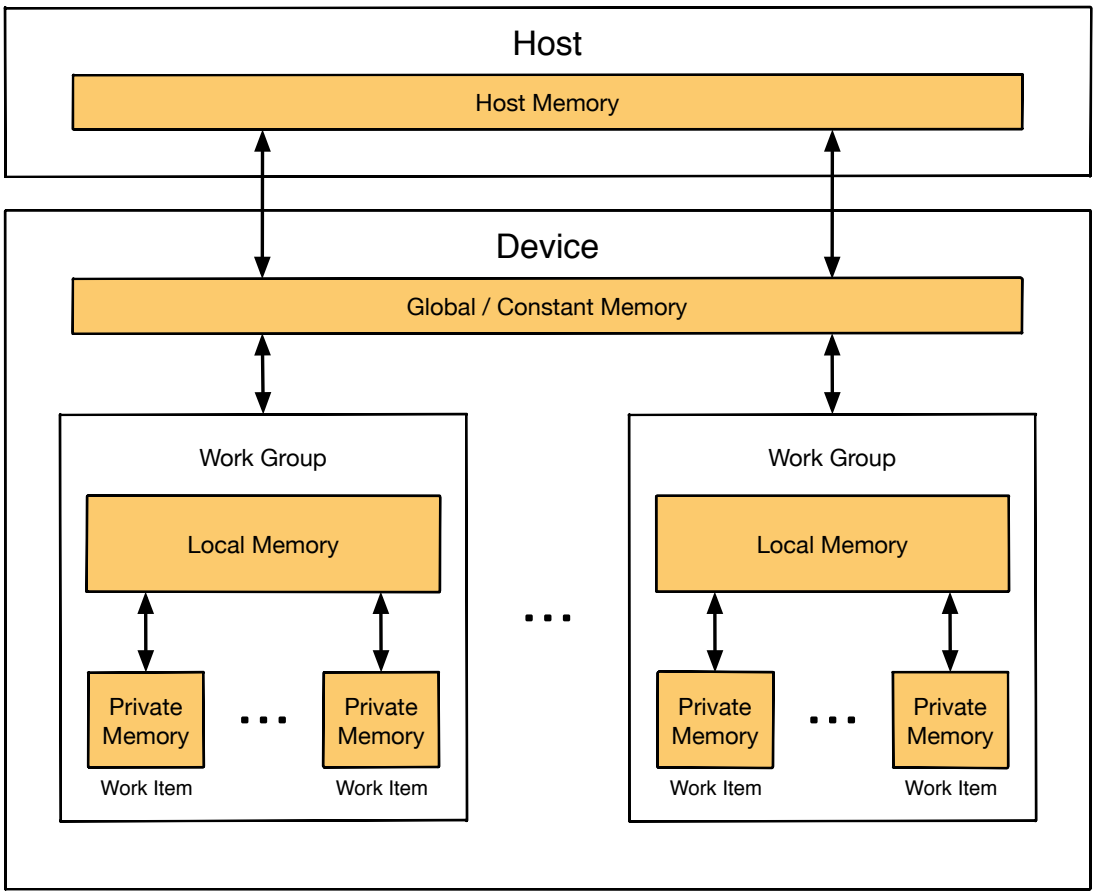


Figure 1.7: OpenCL memory hierarchy.

1.5.2 Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL provides the APIs and the run-time to program and use Intel FPGAs. Unlike CPUs and GPUs, run-time OpenCL kernels compilation is not

feasible due to very long placement and routing time. Intel FPGA SDK for OpenCL Offline Compiler is used to compile kernels to an image file, the bitstream, that the host program uses at runtime to program the FPGA. To properly compile the application and to execute on the FPGA, the Custom Platform is needed. Typically, the board manufacturer develops the Custom Platform that supports a specific OpenCL board. The offline compiler targets the Custom Platform when compiling the OpenCL kernel to generate the hardware programming image.

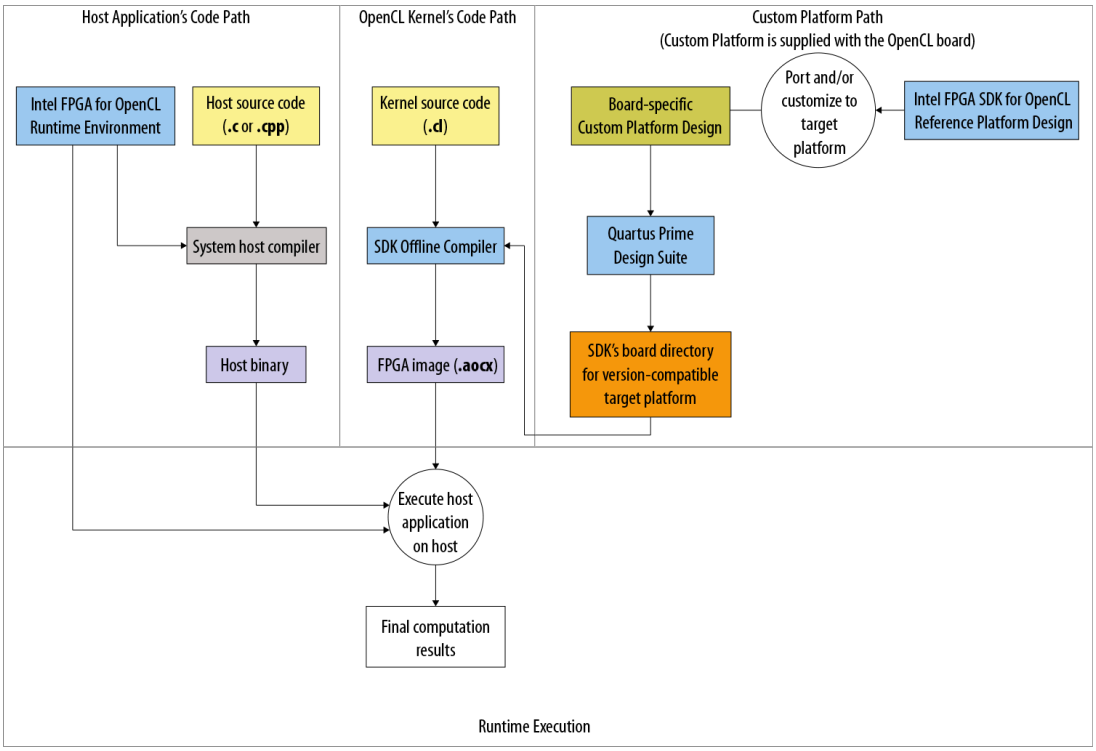


Figure 1.8: Intel FPGA SDK for OpenCL Programming Model

Programming Flow

The Intel FPGA SDK for OpenCL Offline Compiler can create the FPGA programming bitstream in a single or multistep process. From the kernel source file (.cl), the offline compiler groups one or more kernels into a temporary file and then compiles this file to generate the following files and folders:

- .aoco object file, an intermediate object file that contains information for later stages of compilation;
- .aocx image file, the bitstream, that is the hardware configuration file and contains

information necessary to program the FPGA at runtime;

- a work folder or subdirectory, which contains data necessary to create the bitstream file.

By default, the offline compiler uses the single step process to generate the bitstream. Note that this process is very time consuming as it requires to perform a full compilation of the bitstream, which takes hours.

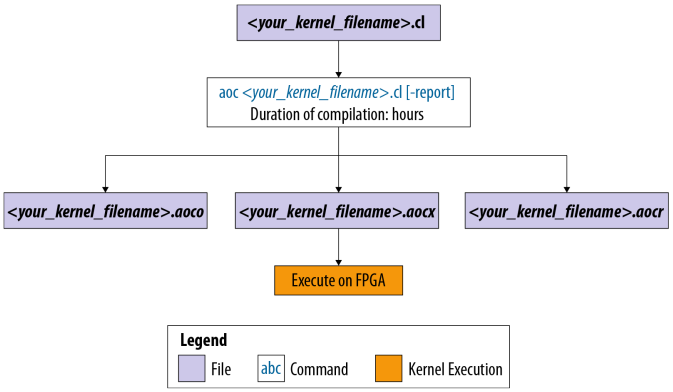


Figure 1.9: Intel FPGA SDK for OpenCL Single-Step Kernel Compilation Flow

The multistep process is preferable as it allows to iterate on OpenCL kernel design to implement optimizations or other iterative modification. The stages in the following design flow serve as checkpoints for identifying functional errors and performance bottlenecks. They allow to modify your OpenCL kernel code without performing a full compilation on each iteration, performing some or all of the compilation steps.

The multistep design flow includes the following steps:

- Emulation: assesses the functionality of the OpenCL kernel by executing it on one or multiple emulation devices. For Linux systems, the Emulator offers symbolic debug support, allowing to locate the origins of functional errors in the kernel code.

- Intermediate Compilation: there are two available intermediate compilation steps. Compiling the kernel source files using the -c flag instructs the offline compiler to generate the .aoco object file that contain the output from the OpenCL parser; or, instead, compiling them using -rtl flag, instructs the offline compiler to generate the .aoco file and the .aocr file.

- Review HTML Report: compiling the kernel source files using the -report, instructs the offline compiler to generate a report of those kernels. This report is very useful

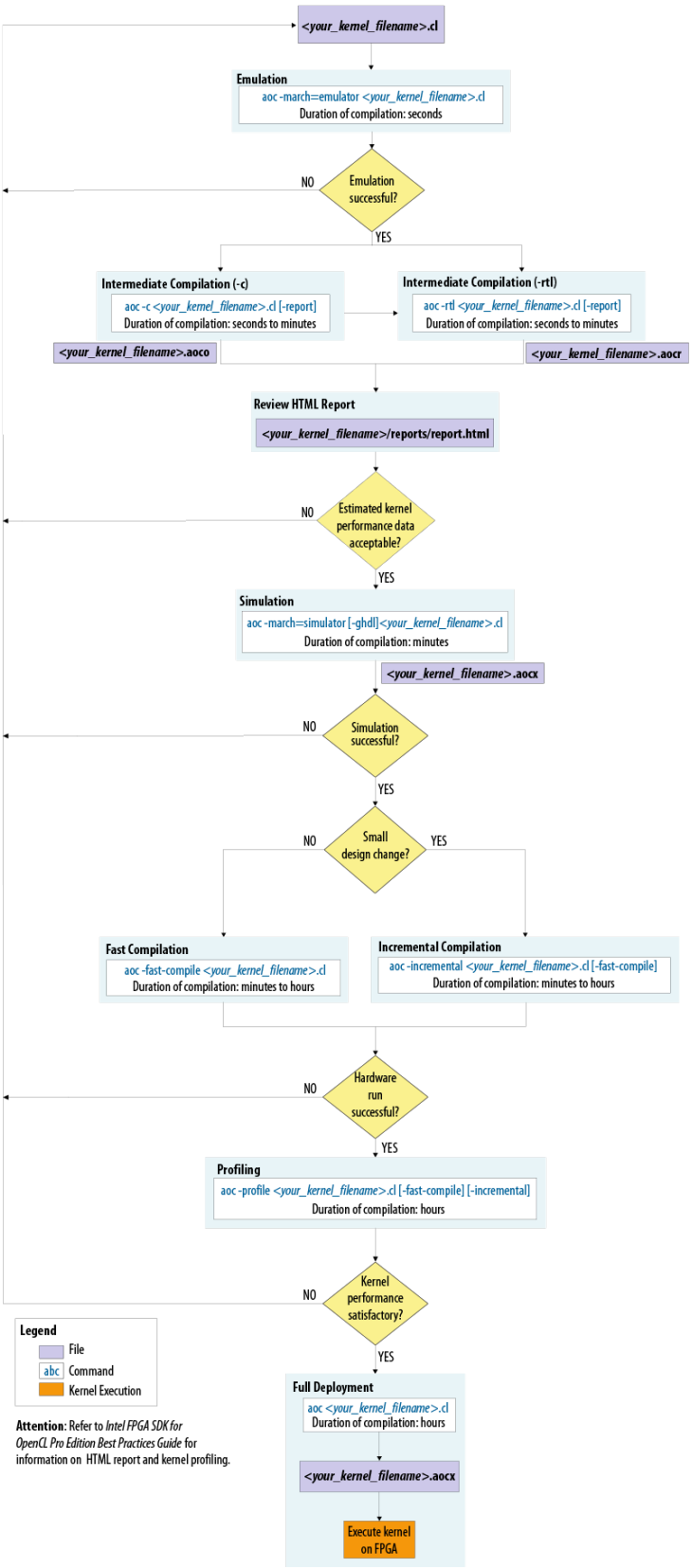


Figure 1.10: Intel FPGA SDK for OpenCL Multi-Step Kernel Compilation Flow

tool as it contains detailed information about the implementation such as utilized resources, timings, memory, etc.

- Simulation (Preview): assesses the functionality of your OpenCL kernel by running it through simulation without a long compilation time.
- Fast Compilation: assesses the functionality of your OpenCL kernel in hardware. The fast compilation step generates the bitstream in a fraction of the time required to complete a full compilation, by performing only light optimization.
- Profiling: the offline compiler inserts performance counters in the bitstream. During execution, the counters collect performance information which can be reviewed in the Intel FPGA Dynamic Profile for OpenCL GUI.
- Full deployment: performs a full compilation and the resulting bitstream will be suitable for deployment.

Programming Model

Intel FPGA SDK for OpenCL specifies two programming models: the NDRange kernel and the Single Work-Item kernel. Typically, in the NDRange kernel programming model, the host launches multiple work-items in parallel. This programming model is implemented by the offline compiler as a deep pipeline, as the kernel usually consists of hundreds of stages. Each work-item is processed one at a time, trying to schedule them one per clock cycle by the run-time scheduler, to minimize pipeline stalls and maximize efficiency. An example of a NDRange kernel code is shown in the following Listing 1.1:

Listing 1.1: NDRange Kernel implementation example

```
1 __kernel void ndrange_kernel(__global const float * restrict a,
2                               __global const float * restrict b
3                               __global float * restrict result)
4 {
5     int i = get_global_id(0);
6     result[i] = a[i] + b[i];
7 }
```

With the Single Work-Item kernel programming model, instead, the entire kernel is executed by one work-item. This programming model follows the sequential model like C programming, and is well suited for fine-grain computations. Loop iterations are converted into pipeline stages and dependencies are resolved by the offline compiler.

An example of a Single Work-Item kernel code is shown in the following Listing 1.2:

Listing 1.2: Single Work-Item Kernel implementation example

```
__kernel void single_work_item_kernel(__global const float * restrict a,
                                      __global const float * restrict b
                                      __global float * restrict result)
{
    for (int i = 0; i < size; ++i) {
        result[i] = a[i] + b[i];
    }
}
```

One programming model is not better than the other. Intel suggests to use the NDRange kernel programming model if:

- data is present before computation;
- kernel does not have loop and memory dependency;
- kernel can execute multiple work-items in parallel efficiently;
- kernel can take advantage of SIMD processing.

And it suggests to use the Single Work-Item programming model if:

- data is processed in stream fashion;
- algorithm can not easily break down into work-items due to data dependencies;
- data can not be easily partitioned.

However, Intel in [12] recommends to use the Single Work-Item kernel programming model.

Channels Extension

The Intel FPGA SDK for OpenCL channels extension provides a mechanism for inter-kernel communication and for kernel to I/O communication. The implementation is based on FIFO buffers. Implementation of channels decouples data movement between concurrently executing kernels from the host processor.

To read from and write to a channel, the kernel must pass the channel variable to each of the corresponding API calls. The channel handle has to be declared as a file scope variable using the syntax shown in the Listing 1.3:

Listing 1.3: Intel Channel declaration syntax

```
channel <type> <variable_name>
```

The type can be a built-in data type or a user defined data structure.

To send data across a channel, one of the following signatures in the Listing 1.4 can be used:

Listing 1.4: Intel Channel write blocking and non-blocking functions

```
1 void write_channel_intel(channel <type> channel_id, const <type> data);
2 bool write_channel_nb_intel(channel <type> channel_id, const <type> data);
```

The former allows to write into the channel with a blocking semantics; this means that, if the channel is full, the caller stalls and waits until at least one data slot becomes available in the FIFO buffer. In order to avoid stalls, the latter can be used as it provides a non-blocking semantic. Indeed, the call returns a boolean value that indicates whether data was written successfully to the channel.

Data can be read with the function signatures presented in the Listing 1.5:

Listing 1.5: Intel Channel read blocking and non-blocking functions

```
1 <type> read_channel_intel(channel <type> channel_id);
2 <type> read_channel_nb_intel(channel <type> channel_id, bool * valid);
```

Similarly to the previous case we examined, the first of these signatures has a blocking semantics: if the channel is empty, the caller stalls until at least one element is written to the channel. Non-blocking reads are performed with the latter function signature and are useful to avoid stalls if data is not available on the channel. On a successful read, the boolean pointed by valid is set to true and the call returns a valid data read from the channel. On a failed read, valid is set to false and the value read from the channel is undefined.

Data written to a channel remains in a channel as long as the kernel program remains loaded on the FPGA device. However, data is not persistent across multiple or different execution of programs that lead to FPGA device reprogramming. Considering the code example shown in Listing 1.6:

Listing 1.6: Producer-Consumer example using Intel Channels

```
1  channel int c;
2  __kernel void producer() {
3      for (uint i = 0; i < 10; ++i) {
4          write_channel_intel(c, i);
5      }
6  }
7  __kernel void consumer(__global uint * restrict dst) {
8      for (uint i = 0; i < 5; ++i) {
9          dst[i] = read_channel_intel(c);
10     }
11 }
```

Both kernels are launched as Single Work-Item kernels. The producer kernel writes ten elements to the channel per invocation. The consumer reads five elements from the channel per invocation. During the first invocation, the consumer kernel reads only the first five elements. In this example, to avoid deadlocks, the consumer must be executed twice for every invocation of the producer. If consumer is invoked less than twice, the producer stalls because the channel becomes full. If, on the other hand, the consumer is invoked more than twice, consumer stalls because there is insufficient data in the channel.

There are few design restrictions in the implementation of channels:

- A kernel can read from the same channel multiple times, however, multiple kernels can not read from the same channel. Similarly, a kernel can write to the same channel multiple times but multiple kernels cannot write to the same channel.
- Channels within a kernel can be either read-only or write-only. Performance of a kernel that reads and writes to the same channel might be poor.
- Vectorizing a kernel (specifying the *num_simd_work_items attribute* which uses channels, creates multiple channel access inside the same kernel and requires arbitration, which negates the advantages of vectorization. As a result, the SDK's channel extension does not support kernel vectorization.
- The Intel FPGA SDK for OpenCL channels extension does support indexing into arrays of channel IDs, but it could lead to inefficient hardware. A slightly more efficient hardware can be generated with a `switch` statement, as shown in the following Listing 1.7:

Listing 1.7: Improved implementation of the Consumer kernel with a switch statement

```
1  channel int c[N];
2  __kernel void consumer(__global uint * restrict indexes, const uint n) {
3
4      for (uint i = 0; i < n; ++i) {
5          int value;
6          switch(i % N)
7          {
8              case   0: value = read_channel_intel(c[0]);   break;
9              case   1: value = read_channel_intel(c[1]);   break;
10             ...
11             case N-1: value = read_channel_intel(c[N-1]); break;
12         }
13 }
```

- If no data dependencies exist between channel calls, the offline compiler attempts to execute them in parallel. As a result, the offline compiler might execute these calls in an order that does not follow the one expressed in the code.

Chapter 2

Data Stream Processing on FPGA

In this section we are going to show how to model a DSP application in OpenCL for FPGA. We start introducing the operator implementation, which is the basic functional unit composing the Data-Flow graph of the application. An operator receives input data items (which we will call "tuples" from now on) from incoming streams, applies an arbitrary function to process them, and emits the resulting tuples to outgoing streams. We will further show how to implement an operator and we provide a base set of operators. Operators can be stateless or stateful: the former processes a tuple without any additional information, while the latter has to maintain an operator state that it uses to process an incoming tuple. We also show the different possibilities to store and manage an operator state and how operators interact with each other. Lastly, we show how special operators on FPGA like Source and Sink operators communicate with the host program depending on the hardware configuration.

2.1 Operator implementation

In our framework, operators are implemented as OpenCL kernels. Both the NDRange kernel model and the Single Work-Item model can be used to properly implement a kernel. Our implementation is based on the Single Work-Item kernel model as it is well documented as well as strongly suggested by Intel.

With the Single Work-Item implementation, once the kernel is compiled, loop iterations inside the operator are converted into pipeline stages. This way the operator can take advantage from the pipeline parallelism provided by the hardware implementation.

To further improve parallelism, the operator can be *replicated* according to its parallelism degree. Intel SDK for OpenCL gives the possibility to replicate operators by specifying the number of *compute units* to the kernel, such as in the following Listing 2.1.

Listing 2.1: Replicating FPGA operator by specifying the number of compute units

```
__attribute__((num_compute_units(N)))
__kernel kernel_name(...) {
    size_t cid = get_compute_id();
    ...
}
```

where `N` is the parallelism degree of the operator, and the `get_compute_id()` function is an intrinsic function that returns the ID of the specific compute unit. However, we adopted a different solution that gives us more flexibility for stateful operator implementation. The kernel code of each operator is copied an amount of times equal to its parallelism degree, and a unique identifier is assigned to each copy. Once compiled, this translates into physical hardware copies of the same operator; this give us the possibility to manage directly each replica and the state associated to them. From now on, we will use *logical operator* to refer to the operator that needs to be replicated, and *operator replica* to designate the actual implemented kernel.

With regards to the implementation of an operator replica, we chose to split the operator business code into three phases. The first phase is the *begin phase*, in which the operator initializes its own internal variables and the operator state. Next we find the *compute phase*, in which the operator receives tuples, processes and sends them to the next operator. Lastly, the *closing phase*, which can be used to clean up the operator state. As shown by the following listing, each phase is implemented as a call to the corresponding phase function. Phase function implementation have to be provided by the developer and they have access to the operator state variables, which are provided as function parameters.

In the following Listing 2.2 we show a basic structure of an operator:

Listing 2.2: FPGA base operator implementation

```
__attribute__((uses_global_work_offset(0)))
__attribute__((max_global_work_dim(0)))
__kernel void kernel_name(/* global memory objects and global variables */)
{
    // kernel scope variables
    ...
    bool done = false;

    // call of the begin phase function

    while (!done) {
        // gather component
```

```
13          // call of the compute phase function
14          // dispatch component
15      }
16
17      // call of the closing phase function
18      // send End-Of-Stream to the next operator replicas
19  }
```

The first two lines instruct the compiler to implement this kernel with the Single Work-Item kernel model, while the signature of the kernel is on line 3. Kernel arguments can be global memory objects and global variables. Global memory objects are pointers to buffers allocated in the global memory region of the OpenCL device, and they are used to write data that has to be processed (i.e., by the previous operator in the pipeline or by the operator interfacing the operators running on the host with the ones running on the FPGA) or memory to store the computed results. Global variables, on the other hand, are constant values provided by the host program. They are usually values representing the size of global memory objects or constant values needed for the computation. The body of the kernel consists of several components. In the first part of the kernel body we find the list of kernel scoped variables, declared and initialized both from the framework and the developer. In the next part, the kernel calls the begin phase function of the operator. The main loop of the kernel starts at line 8, in which incoming tuples from the input stream are gathered, computed and dispatched to the next operator. Inside the main loop, a gather component applies the gather policy to receive incoming tuples from previous operator replicas. This component is also in charge of updating the `done` variable to `True` once the stream ends. After a tuple is received, the compute phase function is called to process the tuple. At the end of the main loop, the dispatch component applies the dispatch policy to send the computed tuple to each next operator replicas. The closing function is called at line 14, and, at the next line, the End-of-Stream (`EOS`) is sent each next operator replicas. The EOS is a mechanism to signal that the stream is closed and the receiving operator replica can terminate its execution.

We devised a set of base operators that are common on a variety of DSP applications:

- Source: this operator reads tuples from the input stream provided by the host program. It may apply a user provided transformation function on those tuples before to send them to the next operator replicas.

- Generator: similar to the Source operator, but it does not have memory communication with the host program. Streams can be generated directly on the device

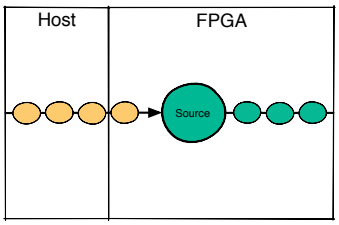
by this operator. Built-in functions are provided to generate random integer and floating point numbers.

- Filter: evaluates a user-defined predicate on each incoming tuple. It drops tuples for which the predicate is `False`, and keeps the others.
- Map: this operator applies a one-to-one user-defined transformation of the incoming tuples by means of the compute phase function. Processed tuples are sent to the next operator replicas. Outgoing tuples can have a different datatype w.r.t. the incoming tuple.
- Sink: it gathers the tuples sent by the previous operator and makes them available to the host program.
- Collector: similar to the Sink operator, but does not provide tuples to the host program.

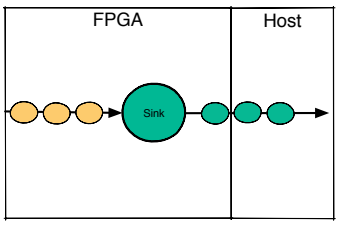
2.2 Operator State

An operator can be characterized by the kind of state that it maintains. An operator is stateless if it does not maintain any history about past received tuples. Stateful operators process incoming tuples on key basis and the operator has to maintain a state per each key.

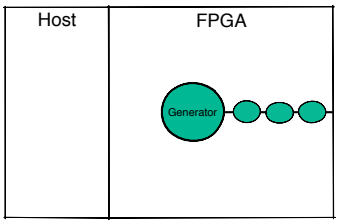
The operator state is maintained using the four types of memory provided by the OpenCL abstractions. Private and local memories are visible by one replica of the operator. Private buffers can be used to store scalars and small arrays. This type of memory is usually implemented by the offline compiler as registers of various configurations (for example, plain registers, shift registers, and barrel shifter), which have a high-bandwidth (100s TB/s) and low latency (1 clock cycle). Large arrays, or arrays with some sort of dependencies (i.e., write-after-read), or arrays which are accessed with a variable index, are promoted to local memory, which is usually implemented with a Block RAM. This type of hardware memory has a different cost in terms of access and hardware resource: it usually has a lower bandiwdth (TB/s), twice the latency and consumes more hardware resources compared with the register implementation.



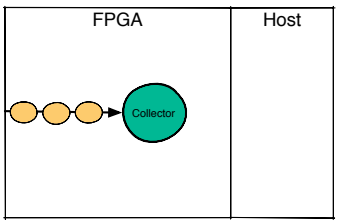
(a) Source Operator



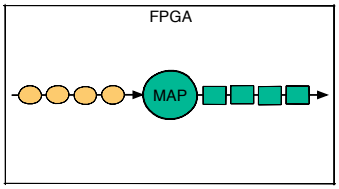
(b) Sink Operator



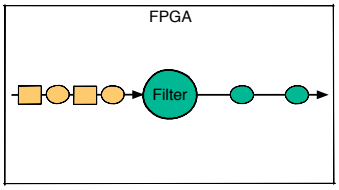
(c) Generator Operator



(d) Collector Operator



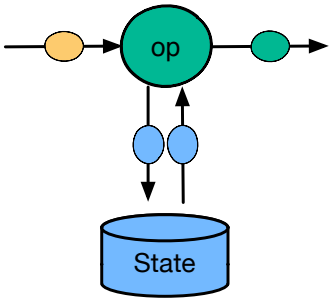
(e) Map Operator



(f) Filter Operator

Figure 2.1: Base operators provided by the FSP framework

(a) Stateless Operator



(b) Stateful Operator

Figure 2.2: Stateless and Stateful operators

Global memory can be used in various ways: a pointer to global memory can be shared by all the replicas of the operator, or even by the entire application. Otherwise, it can be used in a private manner by a replica. Furthermore, a global memory buffer can be accessed in read-only mode, in write-only mode and in read-write mode. Depending on the memory access mode, the compiler can apply specific optimizations to improve latency

and bandwidth. For example, a global memory used in a read-only manner can be coupled with a cache memory, and the compiler can be able to prefetch data if the access pattern has some predefined properties. In addition, it is possible to declare constant memory, which is essentially implemented with global memory. However, the offline compiler can apply more aggressive optimizations to speed up read-only memory accesses on such areas.

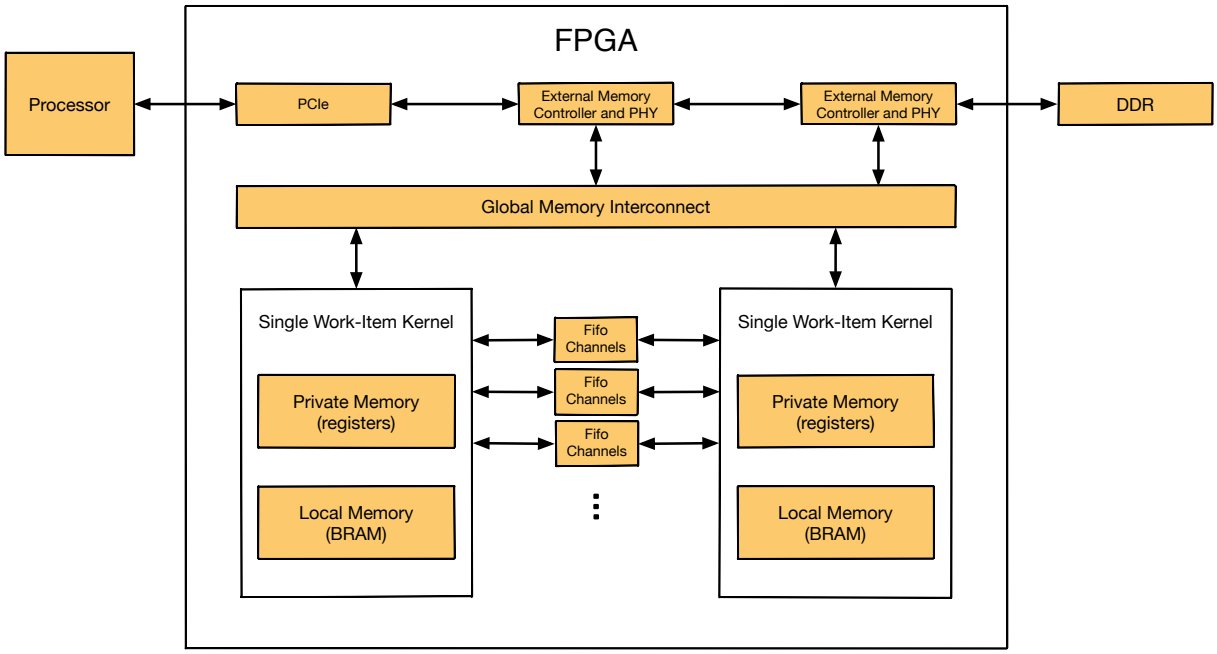


Figure 2.3: OpenCL memory hierarchy on FPGA.

The developer must carefully choose which type of memory to use as it can significantly impact the performance of the operator replicas, and may affect the whole application as well.

Our framework provides useful APIs for managing global memory for stateful operators. The developer can declare one or more global memory buffers for a logical operator and for each of them it is possible to specify the visibility between the replication operators. The state implemented with the global memory can be declared private for each replica, or shared among all replicas of the operator. Furthermore, the access mode can be specified as read-only, write-only, or read-write. If the state is shared among all replicas, the developer must implement the operator (and its replicas) either to manually partitioning the state or by managing the concurrent access. To handle concurrent accesses, the developer can use atomic functions, although Intel strongly discourages them as they can significantly impair the kernel performance.

Our framework does not provide APIs for creating and managing high-level data

structures to maintain the operator state: the developer has the burden of implementing the necessary data structures, i.e. hash tables, on top of the raw memory buffers declared with our API. However, in the next part of this section we show useful implementations for managing the state, which can be useful for a broad range of DSP applications.

2.2.1 Shift Registers using Private or Local memory

The shift register is a very important design pattern for efficient implementation of many applications on the FPGA. Suppose we want to implement a stateful operator of a DSP application in which the computation of each incoming tuple is based on a window containing the last 16 tuples with the same key attribute. Each tuple `tuple_t` has a field `key` which has a value from 0 to 31. If the tuples are small enough they can be stored in private or local memory.

In the following Listing 2.3 we show a possible implementation using the shift registers:

Listing 2.3: Implementation of the shift registers using local memory

```
1  #define KEYS   32
2  #define DIM    16
3  ...
4  __kernel compute_function(const tuple_t in, __local tuple_t windows[KEYS][DIM]) {
5      ...
6      const uint key = in.key;
7      #pragma unroll
8      for (int i = 0; i < DIM - 1; ++i) {
9          windows[key][i] = windows[key][i + 1];
10     }
11     windows[key][DIM - 1] = in;
12     ...
13 }
```

The compute function has two parameters: the incoming tuple `in` and the `windows` variable which stores the last 16 tuples for each key. In the for loop, all tuples inside the corresponding window pointed by the key are shifted to the left. The tuple `in` is then inserted into the last position of the window. The `pragma unroll` is necessary so that the offline compiler can infer that the shift registers pattern is being used and therefore can properly implement it in hardware. This technique is very useful because, if implemented correctly, it guarantees II = 1 even when using Local memory. When implementing a shift register in the kernel, the developer has to keep in mind the following points:

- Unroll the shifting loop so that it can access every element of the array.
- All access points must have constant data accesses. For example, if the computation is inside a nested loop using multiple access points, unroll these loops to establish

the constant access points.

- Initialize all elements of the array to the same value. Alternatively, leave the elements uninitialized as they do not require a specific initial value.
- If some accesses to a large array are not inferable statically, the offline compiler creates inefficient hardware. If these accesses are necessary, use Local memory instead of Private memory.
- Do not shift a large shift register conditionally. The shifting must occur in every loop iteration that contains the shifting code to avoid creating inefficient hardware.

2.2.2 Direct Address Table using Local Memory

Many DPS applications have stateful operators that keep a state through hash tables. There are some implementations of Hash Tables on FPGAs but they are used in specific contexts, such as in network applications [13], and implemented with low-level languages such as Verilog. A simpler yet effective alternative is the Direct Address Table, which can be implemented using all the memory levels provided by OpenCL. We recommend using Private and Local memory as they have low latency, but in this case, tuples of the stream have to have a finite and small set of keys.

Suppose we want to implement a stateful operator with four internal replicas, which internally maintains a Direct Address Table that keeps track of the number of tuples for each key. Each tuple of type `tuple_t` has a field `key` that stores an integer between 0 and 255 (we assume that the set of keys have cardinality 256 in this example, and each key is a unique identifier).

The Direct Address Table of each replica is implemented as an array of size 64 (the number of keys divided by the number of replicas). The following Listing 2.4 shows a possible implementation of the compute phase function of the stateful operator.

Listing 2.4: Implementation of the compute phase function using a Direct Address Table

```
1 #define PAR    4    // parallelism degree of the map operator
2 #define KEYS   256  // max value of a key
3 ...
4 __kernel computation_function(const tuple_t in, __local histogram[KEYS / PAR]) {
5     ...
6     const uint key = in.key / PAR;
7     histogram[key]++;
8     ...
9 }
```

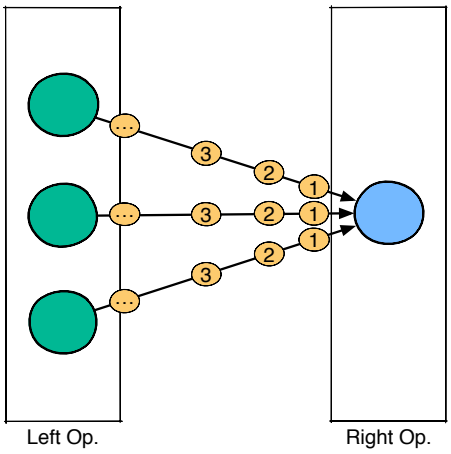
The `histogram` parameter is the Direct Address Table provided by a replica and it is implemented with Local memory. The state associated to a tuple is accessed by indexing the `histogram` array with the value of the tuple key divided by the parallelism degree of the stateful operator.

2.3 Operator Dependencies

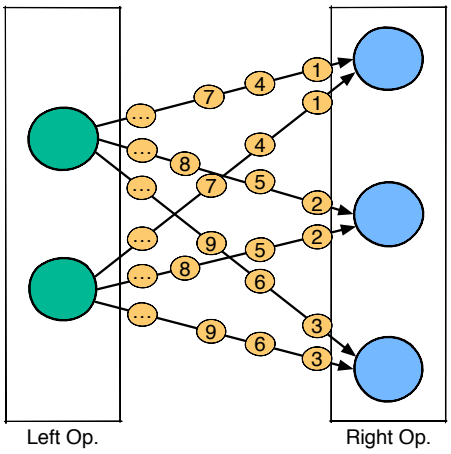
As described above, each logical operator is replicated according to the parallelism degree assigned to it. Let us consider the case of two consecutive logical operators, we call them left-hand logical operator (L) and right-hand logical operator respectively (R). Each replica of L is directly connected to each of the replicas of R. The communication pattern is defined by two policies assigned to a logical operator: the dispatch policy and the gather policy. The dispatch policy does not apply to the sink and the collector operators, as they do not have a next operator to dispatch their tuples. There are four dispatch policies:

- Forward: this policy is always applied when R has a parallelism degree equal to one. Each replica of L sends the tuple directly to the replica of R.
- RoundRobin: this policy has two different modes, namely a *blocking* and a *non-blocking mode*.In the blocking mode, tuples are dispatched in a circular order, meaning that the first tuple is sent to the first operator replica, the next tuple to the second operator replica, and so on. With the non-blocking mode, tuples are still dispatched in a circular order, but if the destination replica is not capable to process the tuple, it is dispatched to the next available operator replica following the circular ordering.
- KeyBy: this policy is used by operators that maintain a local state. A user-provided function extracts a key attribute from the tuple, and the tuple is transmitted to the operator replica responsible for receiving all the tuples with that key attribute.
- Broadcast: this policy is used to dispatch a copy of the tuple to each replica of the next operator.

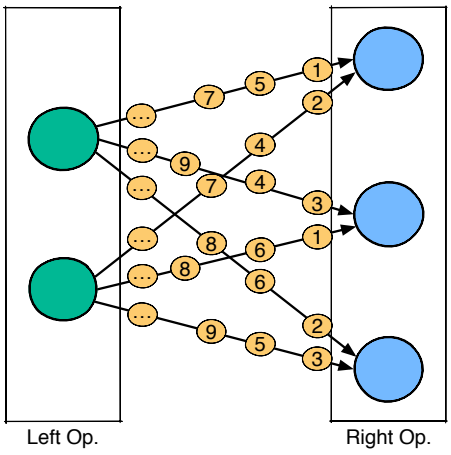
The gather policy does not apply to the source and to the generator operators as they do not have a preceding operator to gather tuples. As already explained with the



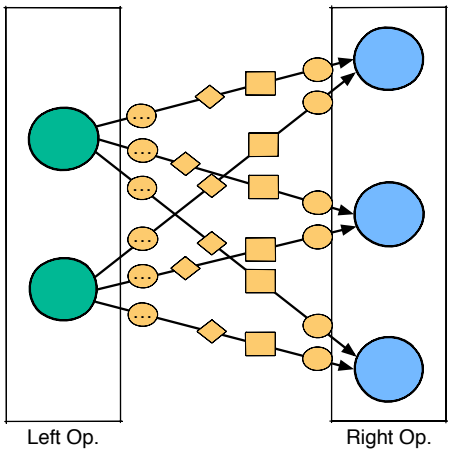
(a) Forward dispatch



(b) RoundRobin dispatch, Blocking mode



(c) RoundRobin dispatch, Non-Blocking mode



(d) KeyBy dispatch

Figure 2.4: Dispatch policies

RoundRobin dispatch policy, the gathering of tuples can happen in blocking or non-blocking mode. With the blocking mode semantics, each R replica waits for a tuple from the first replica of L, then from the second replica, and so forth. If the L operator has a parallelism degree equal to one, this is the default policy. The non-blocking mode semantics is very similar to the RoundRobin dispatching policy described earlier. In this case, each replica of R tries to receive a tuple from the first replica of L. If the tuple is not immediately available, the replica will try to receive a tuple from the next L replica following the circular ordering (therefore executing a polling of the input channels).

These policies can be implemented in many ways. For example, we have an application consisting of a pipeline of operators with parallelism degree equal to one. In this case the dispatch policy of each operator is Forward, and the gather policy is blocking.

A possible implementation consists of controlling the flow of messages between FPGA operators by the host program via global memory. Suppose that a host thread is assigned to each operator: the thread waits for a global memory pointer from the previous operator and then launches its assigned FPGA kernel by passing the received pointer. Once the operator finishes the computation, it sends the global memory pointer of the computed result to the next thread of the pipeline (e.g., using any communication mechanism available between host threads such as lock-free data structures to host addresses to shared data structures [14]). After the operator has finished the computation, the last thread (associated with the sink operator of the pipeline) reads the tuple from the global buffer and frees or recycles the pointer.

This procedure requires a lot of synchronizations between threads and FPGA operators, which could lead to a massive overhead. Furthermore, reading and writing to global memory buffers is costly as the bandwidth is low and shared among all the kernels accessing to it. This can cause a lot of contention on global memory, and hence, the latency of each access operation on global memory could be huge.

Our implementation, instead, is based on the Intel Channel Extension, which provides mechanisms to perform inter-kernel communications directly on FPGA, without the need of the host program to control the flow of messages exchanged between operators, and so by reducing contention on global memory. Such channels are implemented as FIFO buffers directly on the FPGA fabric. Hence, the host program has less interaction with the operator replicas, meaning less code and no need for expensive synchronizations. Furthermore, the logic of the dispatch and gather policies is part of the operator replicas only, since the host is not involved. Indeed, tuples can be directly sent and received by means of channels, without storing them in global memory and by providing immediate access to the new tuples to process by the next operator of the application. Consequently to this choice, latency is drastically reduced as there is no synchronization overhead and no store in global memory that is very costly.

To control the termination of the streaming application on the FPGA, we wrap each tuple in a special wrapper message encapsulating the tuple itself and a boolean flag named EOS (end-of-stream). A wrapper data structure is declared for each pair of connected logical operators, since each operator can emit its own tuple data type which can be different from the input one. The wrapper type name is obtained by concatenating the

names of the L and R operators. The following Listing 2.5 represents an example of a wrapper data type with its definition:

Listing 2.5: Wrapper data structure for channel communication

```
typedef struct {
    <tuple_type> data;
    bool EOS;
} <left>_<right>_t;
```

where `<tuple_type>` is the tuple data type exchanged between the `<left>` and the `<right>` logical operator (L and R). The `EOS` is a boolean value used to signal the end of the stream to the receiving operator replica so that the replica can terminate its execution.

Channels involved in the communication of two subsequent logical operators have to be declared as global scope variables, and they use right wrapper data type. They can be declared with the C array notation. In the following listings, the value of `N` is the parallelism degree of the L operator and the value `M` is that of the R operator.

Now, we present the dispatch policy implementations in the Listing 2.6, showing the first replica of the L operator with the unique identifier 0. In the Forward dispatch policy implementation, we need only one call to the `write_channel_intel()` primitive on the channel `ch` indexed with the unique identifier of the replica.

Listing 2.6: Forward Dispatch policy with right-hand side with parallelism degree = 1

```
<left>_<right>_t ch[N];
...
__kernel void <left>_0(...) {
    const uint idx = 0;
    ...
    while (!done) {
        // gather component
        <left>_<right>_t out;
        // call of the computing function
        write_channel_intel(ch[idx], out);
    }
    ...
}
```

In the RoundRobin dispatch policy, both in blocking and non-blocking mode, we introduce a `switch-case` statement to generate a more efficient hardware with respect to the single indexing. The `w` variable keeps track of the last used channel. It is initialized with the unique identifier, instead of 0, to avoid contention on channels, especially in the blocking mode version. After a write on a channel, this variable is incremented as `w = (w + 1) % M`. We do not show the update code of `w` for readability purposes, as it is implemented with 4 lines of code to instruct the offline compiler to generate efficient hardware. We implement the RoundRobin dispatch policy as in the Listing 2.7.

Listing 2.7: RoundRobin dispatch policy in blocking mode with right-hand side with parallelism degree > 1

```
1  <left>_<right>_t ch[N][M];
2  ...
3  __kernel void <left>_0(...) {
4      const uint idx = 0;
5      uint w = idx;
6      ...
7      while (!done) {
8          // gather component
9          <left>_<right>_t out;
10         // call of the computing function
11         switch (w) {
12             case   0: write_channel_intel(ch[idx][  0], out); break;
13             case   1: write_channel_intel(ch[idx][  1], out); break;
14             ...
15             case M-1: write_channel_intel(ch[idx][M-1], out); break;
16         }
17         // update `w`
18     }
19     ...
20 }
```

For the RoundRobin dispatch policy in non-blocking mode, we have to check if a write is executed successfully as shown in the Listing 2.8.

Listing 2.8: RoundRobin dispatch policy in non-blocking mode with right-hand side with parallelism degree > 1

```
1  <left>_<right>_t ch[N][M];
2  ...
3  __kernel void <left>_0(...) {
4      const uint idx = 0;
5      uint w = idx;
6      ...
7      while (!done) {
8          // gather component
9          bool success = false;
10         <left>_<right>_t out;
11         // call of the computing function
12         do {
13             switch (w) {
14                 case   0: success = write_channel_nb_intel(ch[idx][  0], out); break;
15                 case   1: success = write_channel_nb_intel(ch[idx][  1], out); break;
16                 ...
17                 case M-1: success = write_channel_nb_intel(ch[idx][M-1], out); break;
18             }
19             // update `w`
20         } while (!success);
21     }
22     ...
23 }
```

For the KeyBy dispatch policy, the developer has to provide a mechanism to extract a key from the received tuple. We calculate the index of the destination operator replica by taking the remainder of the integer division of the extracted key and the parallelism degree of the next logical operator as shown at line 10 of the Listing 2.9.

Listing 2.9: KeyBy dispatch policy with right-hand side with parallelism degree > 1

```
1  <left>_<right>_t ch[N][M];
2  ...
3  __kernel void <left>_0(...) {
```

```
    const uint idx = 0;
    ...
    while (!done) {
        // gather component
        <left>_<right>_t out;
        // call of the computing function
        const uint w = tuple_get_key(out.data) % M;
        switch (w) {
            case   0: write_channel_intel(ch[idx][  0], out); break;
            case   1: write_channel_intel(ch[idx][  1], out); break;
            ...
            case M-1: write_channel_intel(ch[idx][M-1], out); break;
        }
    }
    ...
}
```

With respect to the Gather policies, we must point out that the call of the computing function and the dispatch component are injected into the gather component implementation. If a EOS is received, it has to be handled properly. It keeps track of the EOS of each preceding operator replica and updates the `done` variable with `true` once all the channels are closed. Moreover, the `r` variable is initialized with the unique identifier of the operator replica and incremented as `r = (r + 1) % N`. For the sake of brevity, we are not going to show the code related to the handling of the EOS messages.

If the L operator has parallelism degree equal to 1, the gather policy implementation is in blocking mode and needs only one call to `read_channel_intel()` indexed with the unique identifier of the replica, as shown in the Listing 2.10.

Listing 2.10: Gather policy in blocking mode with left-hand side with parallelism degree = 1

```
<left>_<right>_t ch[M];
...
__kernel void <right>_0(...) {
    const uint idx = 0;
    ...
    while (!done) {
        <left>_<right>_t in = read_channel_intel(ch[idx]);
        if (in.EOS) {
            // handle EOS
        } else {
            // call of the computing function
            // dispatch component
        }
        ...
    }
    ...
}
```

Otherwise, if the L operator has a parallelism degree greater than 1, a different implementation is needed and it is shown in the Listing 2.11. A boolean array is needed to keep track of the EOS received by each replica to avoid the kernel to be blocked on a closed channel. For each `case` of the `switch-case` statement, we check if a replica has completed its computation and closed the communication before calling the read on the

corresponding channel.

Listing 2.11: Gather policy in blocking mode with left-hand side with parallelism degree > 1

```
1  <left>_<right>_t ch[N][M];
2  ...
3  __kernel void <right>_0(...) {
4      const uint idx = 0;
5      uint r = idx;
6      bool EOS[N];
7      ...
8      while (!done) {
9          bool valid = false;
10         <left>_<right>_t in;
11         switch (r) {
12             case   0: if (!EOS[  0]) {in = read_channel_intel(ch[  0][idx]); valid = true;} break;
13             case   1: if (!EOS[  1]) {in = read_channel_intel(ch[  1][idx]); valid = true;} break;
14             ...
15             case N-1: if (!EOS[N-1]) {in = read_channel_intel(ch[N-1][idx]); valid = true;} break;
16         }
17
18         if (valid) {
19             if (in.EOS) {
20                 // handle EOS
21             } else {
22                 // call of the computing function
23                 // dispatch component
24             }
25             ...
26             // update `r`
27         }
28     }
29     ...
30 }
```

In the non-blocking version, instead, we do not need to keep track of the EOS received for each replicas. The non-blocking read primitive of Intel channels, which is defined as `read_channel_nb_intel()`, requires a `boolean` variable `valid` that states if the returned value of the primitive is meaningful or not. So, after the read, the `valid` variable is checked before proceeding into the computation, as shown at line 17 in the Listing 2.12.

Listing 2.12: Gather policy in non-blocking mode with left-hand side with parallelism degree > 1

```
1  <left>_<right>_t ch[N][M];
2  ...
3  __kernel void <right>_0(...) {
4      const uint idx = 0;
5      uint r = idx;
6      ...
7      while (!done) {
8          bool valid = false;
9          <left>_<right>_t in;
10         switch (r) {
11             case   0: in = read_channel_nb_intel(ch[  0][idx], &valid); break;
12             case   1: in = read_channel_nb_intel(ch[  1][idx], &valid); break;
13             ...
14             case N-1: in = read_channel_nb_intel(ch[N-1][idx], &valid); break;
15         }
16
17         if (valid) {
18             if (in.EOS) {
19                 // handle EOS
```

```
20              } else {
21                  // call of the computing function
22                  // dispatch component
23              }
24          }
25          ...
26          // update `r`
27      }
28      ...
29  }
```

2.4 Managing Streams

Accelerators usually perform computations once all the data is located in their own memory hierarchy. As a first step, the host program copies the required data to the memory of the accelerator, then instructs the accelerator to process the data, and finally copies back the results on the host memory.

Due to the characteristics of stream processing applications, data is not available immediately as it arrives during the computation. In an ideal scenario, each input tuple should be processed as soon as it arrives (one-at-a-time processing [5]). In case of using an accelerator such as an FPGA with its own memory, incoming tuples have to be copied to the FPGA memory.

Each memory access has a fixed cost and a variable one. The fixed cost is usually due to the initialization of the memory communication protocol,while the variable cost is proportional to the data size transferred. In case of small/medium size tuples (i.e., less than some KB), the individual copy of each tuple to the FPGA memory can be quite costly since the fixed cost dominates.

In our implementation, we adopted a micro-batching data processing approach already used in the literature for similar purposes [5]. The host program is in charge of collecting input tuples from incoming streams to form micro-batches of a predefined size. Then, each complete micro-batch is provided to the FPGA which processes it. Outgoing streams of results produced by the FPGA are managed with the micro-batch as well. Sink operators gather tuples and buffers them to form a micro-batch, which will be stored in global memory as soon as it is filled completely or the incoming streams end (by receiving the EOS message from all the preceding replicas).

2.5 Host ↔ Device Communication

The host ↔ device communication is an important aspect concerning the use of accelerators. The host program is in charge of providing tuples to the source operator, which reads them from global memory buffers. Furthermore, the sink operator will make the received tuples available to the host program by means of global memory buffers.

Communication overheads can have a significant impact on the overall application throughput: for example, applications with fine-grained computations are not able to hide the communication latency entirely. Another aspect that can play a significant role is the available hardware capabilities: for example, an FPGA connected through the PCI-Express interconnect needs to copy tuples from the host memory to the device one. In hardware configurations based on System-on-Chip (SoC), a small multicore and an FPGA are integrated on the same chip and they share the physical memory of the system. In this scenario, the use of zero-copy techniques can significantly reduce or possibly minimize communication latencies.

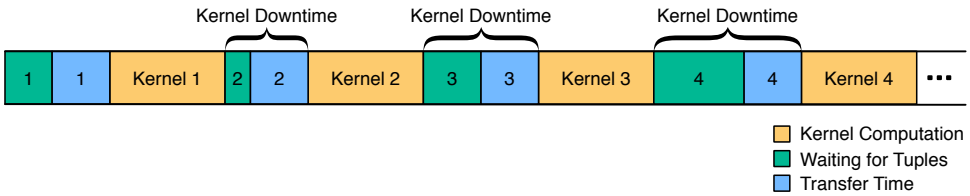
The host ↔ device communication is necessary to provide the micro-batches to the Source operator, as well as to extract the processed tuples from the Sink operator. In the first case, the host collects the tuples needed to create a micro-batch and makes them available to the Source operator on the FPGA through a global memory object. Subsequently, it launches the Source operator kernel and waits for its completion to provide a new micro-batch. In the second case, the host launches the Sink operator kernel, which collects the processed tuples from the preceding operators on the device, and makes them available to the host, which might be waiting for them.

To reduce the impact of the communication overhead, we provide two implementations: the first makes use of the standard functions provided by OpenCL, while the second one aims at reducing the communication overhead by exploiting the shared memory between CPU and FPGA with a custom synchronization protocol.

The first implementation is based on the generalization of the double-buffering technique, known as the N-Buffering technique. This technique allows the execution of kernels that occur in parallel to the transfers of micro-batches between the host and the device, improving the performance of the application. Without using this technique, the micro-batch transfer would occur between kernel executions if the transfer time is shorter than the computation time: this means that there would be a gap in time between one kernel

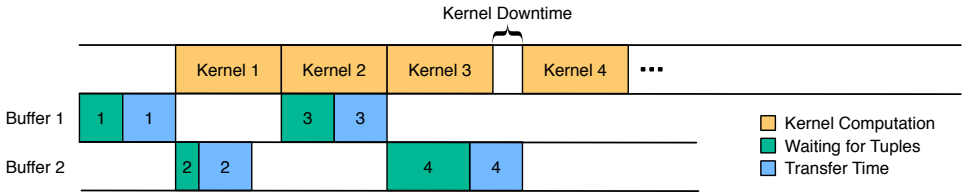
execution and the next one, which we refer to as *kernel downtime*. By using this technique,

Figure 2.5: Single-Buffering transfer implementation diagram



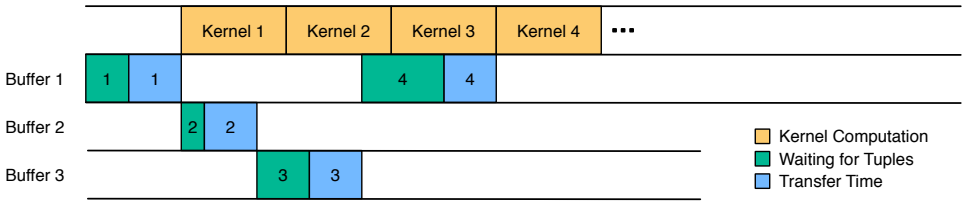
for example using N = 2 buffers, we can overlap the execution of the kernels with the transfer time, so that the kernels can be executed back-to-back with minimal downtime. In cases where tuples arrive with a variable frequency, the overall time to consider is the

Figure 2.6: Double-Buffering transfer implementation diagram



transfer time plus the time it takes to collect enough tuples to form a micro-batch. In this case, the use of N buffers is recommended, as it allows to maintain a lower downtime than using the double-buffering technique. The implementation of this technique makes use of

Figure 2.7: N-Buffering transfer implementation diagram with N = 3



N global memory buffers, which are recycled in a circular fashion. For buffer management and dependency enforcement, we make use of OpenCL *events*, which allows us to enforce dependencies between kernel execution and buffer access; in addition, we employ multiple command queues to perform micro-batch transfers and kernel executions in parallel. The following Listing 2.13 shows an example of the implementation of this technique using `N` buffers.

Listing 2.13: Host implementation of the N-Buffering technique

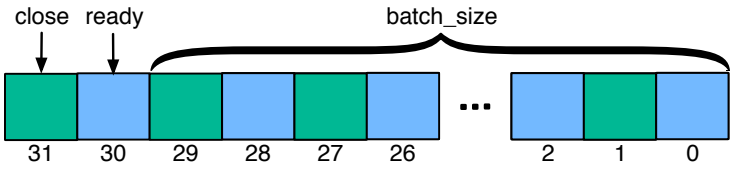
```
cl_event source_event[N];
cl_mem buffer_device[N];

void push(const tuple_t * buffer_host, const size_t buffer_size) {

    const size_t curr_it = it % N;
    cl_event write_event;

    if (iteration < N) {
        clEnqueueWriteBuffer(buffer_queue, buffer_device[curr_it], CL_FALSE, 0,
                             buffer_size, buffer_host,
                             0, NULL, &write_event);
        clFlush(buffer_queue);
    } else {
        clEnqueueWriteBuffer(buffer_queue, buffer_device[curr_it], CL_FALSE, 0,
                             buffer_size, buffer_host,
                             1, &source_event[curr_it], &write_event);
        clFlush(buffer_queue);
    }

    clSetKernelArg(source_kernel, 0, sizeof(cl_mem), &buffer_device[curr_it]);

    clEnqueueTask(source_queue, source_kernel,
                  1, &write_event, &source_event[curr_it]);
    clFlush(source_queue);
    iteration++;
}
```

The second to last parameter of the OpenCL functions `clEnqueueWriteBuffer` and `clEnqueueTask` is `event_wait_list`, which allows us to specify one or more event that needs to be completed before executing the function. An event is then associated with each write and launch operation of a kernel, which are used to keep track of the dependencies. The first N micro-batches are written to the buffer devices without any dependency, because the buffer devices are initially empty and no kernel has been previously launched to process them. Then, the device buffers are recycled and the micro-batch is copied with the `clEnqueueWriteBuffer` function, specifying the event associated with the launch of the oldest kernel, namely the kernel that has been launched to process the targeting buffer. The kernel is launched by specifying the event associated to the write the buffer call, to ensure that the buffer has been completely written. The `clFlush()` function is necessary to make sure that the previous commands have been submitted.

The second implementation is designed to take full advantage of SoC configurations where a physical shared memory exists between the host and the FPGA. Intel specifically states not to use the calls to `clEnqueueReadBuffer` and `clEnqueueWriteBuffers` in this case, but rather to use the `clEnqueueMapBuffer` function to map the shared memory to the host, which can directly use the pointer returned by the function for read and write accesses. Since the CPU and FPGA can access the shared memory simultaneously, we designed a proper synchronization protocol, which makes use of two shared memory buffers: one is used to exchange headers, while the other is used to exchange micro-

batches. The header is a 32-bit unsigned integer, where bit 31 is the `close` flag, which indicates whether the communication is terminated; bit 30 is the `ready` flag to indicate that the micro-batch is ready for read/write accesses; lastly, the remaining bits are used to specify the `size` of the micro-batch. Both buffers are implemented as circular buffers

Figure 2.8: Representation of `header_t`



with size N. The headers buffer is an array of `header_t` items, each one referring to each micro-batch contained in the second buffer. The micro-batches can have a variable size in terms of tuples, with a maximum size set by the developer. Support functions have been defined for header management and we present them in the following Listing 2.14:

Listing 2.14: Host and Device supporting functions for shared memory communication protocol

```
1  inline bool header_close(const header_t h) {
2      return (h >> 31);
3  }
4
5  inline bool header_ready(const header_t h) {
6      return (h >> 30) & 1;
7  }
8
9  inline uint header_size(const header_t h) {
10     return (h & 0x3FFFFFFF);
11 }
12
13 inline header_t header_new(const bool close, const bool ready, const unsigned int size) {
14     return (header_t)((close << 31) | (ready << 30) | (size & 0x3FFFFFFF));
15 }
```

In the following Listing 2.15 we present the implementation of the `push` function that the host uses to make a new micro-batch available to the Source operator.

Listing 2.15: Host implementation of `push()` function using shared memory protocol

```
1  // Host-side
2  void push(const tuple_t * batch, const size_t batch_size, const bool close) {
3      // busy waiting on the ready flag
4      while (header_ready(headers[id])) {};
5      // once the ready flag is false, copy micro-batch
6      memcpy(buffers[id], batch, batch_size * sizeof(tuple_t));
7      // ensure all writes are completed
8      WRITE_MEMORY_BARRIER();
9      // update the header    (close, ready, batch_size)
10     headers[id] = header_new(close,  true, batch_size);
11     // point to the next header
12     id = (id + 1) % N;
13 }
```

The host performs a busy-waiting loop on the ready flag of the header pointed by the `id` variable. Once the ready flag is set to false, the host copies the micro-batch on the shared buffer and then the header is updated specifying the number of tuples in the micro-batch, then it sets the ready flag to true and the close flag with the `close` parameter. We inserted a write memory barrier between the copy of the micro-batch and the writing of the header, in order to guarantee that the writes are executed in the same order as written in the code. Without the write memory barrier, the compiler could reorder the writes and so the header could be written before the copy of the micro-batch has finished. Finally, the id variable is incremented modulo N.

In the following listing we have an example of the Source operator on the device.

Listing 2.16: Implementation of Source base operator on FPGA using shared memory communication protocol

```
1  // Device-side
2  __kernel source(__global volatile header_t * restrict headers,
3                  __global const volatile tuple_t * restrict batches) {
4      uint id = 0;
5      bool done = false;
6      ...
7      while (!done) {
8          header_t h;
9          // busy waiting on the ready flag
10         while (!header_ready(h = headers[id]));
11
12         const uint batch_size = header_size(h);
13         for (uint i = 0; i < batch_size; ++i) {
14             // read tuples from batches and dispatch them
15         }
16         // update done variable
17         done = header_close(h);
18         // point to the next header
19         id = (id + 1) % N;
20         // ensure all reads completed
21         mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);
22         // update the header    (close, ready, batch_size)
23         headers[id] = header_new(false,  false, 0);
24     }
25     ...
26 }
```

Within the main loop, the operator performs a busy-waiting loop on the ready flag of the header pointed by the id variable. Once the ready flag is set to true, the actual size of the micro-batch is extracted and the tuples of the micro-batch are read and dispatched; the `done` variable is then updated with the `close` flag of the header and the id variable is incremented modulo N. Finally, the header is updated to make it available to the host. Before writing the header, we have a `mem_fence()` that guarantees the order of the writes to both the channels and the global memory.

For the sake of completeness, we show the listings of the Sink operator that makes the gathered tuples available to the host.

Listing 2.17: Implementation of `pop()` function using shared memory protocol

```
// Host-side
void pop(tuple_t * batch, size_t * batch_size, bool * close) {
    // busy waiting on the ready flag
    while (!header_ready(headers[id])) {};
    // update close and batch_size parameters
    *close = header_close(h_ptr[idx]);
    *batch_size = header_size(h_ptr[idx]);
    // copy processed tuple to the provided buffer
    memcpy(batch, buffers[id], *batch_size * sizeof(tuple_t));
    // ensure all reads are completed
    LOAD_MEMORY_BARRIER();
    // update the header    (close, ready, batch_size)
    headers[id] = header_new(false, false,          0);
    // point to the next header
    id = (id + 1) % N;
}
```

Listing 2.18: Implementation of Sink operator on FPGA using shared memory communication protocol

```
// Device-side
__kernel sink(__global volatile header_t * restrict headers,
              __global volatile tuple_t * restrict data)
{
    uint id = 0;
    bool done = false;
    ...
    while (!done) {
        // busy waiting on the ready flag
        while (header_ready(headers[id])) {};
        bool read_done = false;
        while (!read_done) {
            // gather tuples and write them to the micro-batch buffer
        }
        // ensure all writes completed
        mem_fence(CLK_GLOBAL_MEM_FENCE);
        // update the header    (close, ready, batch_size)
        headers[id] = header_new(done,   true,          n);
        id = (id + 1) % N;
    }
}
```

Chapter 3

Code Generation

As shown in the previous chapter, writing a DSP application in OpenCL targeting an FPGA device from scratch could be a complex task as there is the necessity to write repetitive code and manage the interaction between host and device properly. The host program needs to prepare the OpenCL context, initialize the device providing the bitstream, create the global memory buffers and fill them, launch and manage the kernel execution, and manage the FPGA input streams and output streams. Most of the OpenCL host code is boilerplate code and should be adapted to the application needs and the targeting device; in addition, operator replicas should be properly engineered for their replication and state management. Both dispatch and gather policies implementation must be properly designed taking into account the number of replicas, in order to generate efficient hardware and to avoid stalls. For these reasons, the FSP framework uses a code generation approach to produce efficient code starting from a high-level representation of a DSP application written in Python, and generating optimized code tailored for FPGAs.

3.1 FPS framework

The FSP framework enables a quick development of DSP applications targeting FPGA devices. It provides High-Level APIs to describe the application in a Python script that generates the host and the device application code. Furthermore, the host generated code presents supporting APIs to manage the application and the operator states. We have been employed the templating engine `Jinja 2.11` to implement a code generation

approach.

The code generation approach has many advantages for both the application developer and the framework developer. Application developers should be focused only on the implementation of their applications without having to consider problems related to the implementation of the operators on the FPGA, the host runtime support and data transfers. Starting from the same high-level description, different versions of the application can be generated by changing some parameters, e.g., the parallelism degree, memory implementation of the state, and dispatching policies. From a framework developer point of view, the framework presents a building-blocks structure that allows new components (i.e. base operators, memory transfer protocols) to be added or modified to enable optimizations. The code generation technique overcomes many of the limitations of the OpenCL language. For example, some implementations described in the previous chapter could be implemented using macro expansions, but this approach turns out to be poorly maintainable as it is error prone and not very flexible from many points of view.

In order to develop an application with our framework, the developer has to follow the workflow depicted in Figure 3.1. Its application must be described by creating a Python script using the Python APIs, which we describe in depth in the following sections, and it has to implement the operator phase functions generated by the framework. Once the

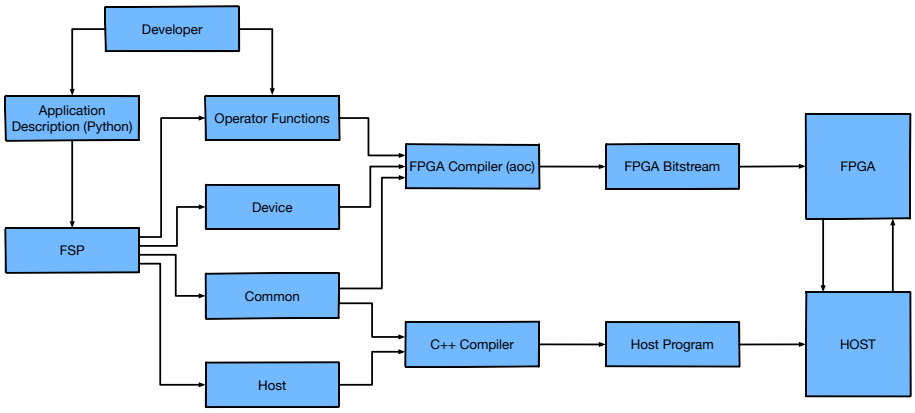


Figure 3.1: FSP Workflow.

script is executed, the FSP framework generates both the host and the device code. The generated host code is written in C/C++ and includes: the code for the management of the OpenCL environment; a class for each logical operator of the application that manages replica kernels and their state; a class for the management of the pipeline of operators on the FPGA. The generated device code is written in OpenCL for the Intel FPGA SDK and

includes: a file containing the operator replica kernels, channel declarations and kernel scoped constants and variables; a file for each logical operator containing the function signatures of the three phases; a file containing the structures necessary to define the specified tuples In addition, a Makefile is generated to compile the host program and the device code in emulator mode, as well as for the generation of the bitstream for both debug and release purposes.

3.2 Use case: Spike Detection

In this section, we will show how to use our FPS framework to implement the Spike Detection (SD) application. This application is a good example of a DSP application where data from a multitude of sensors is processed in an IoT scenario. Several sensors produce information regarding temperature, humidity, voltage, and other factors which the SD application analyzes in order to check for anomalies in real time. For each received value, its deviation from the average computed until that moment is calculated. The mean of the values received by that sensor is updated with each new value. An anomaly, called a spike, is detected when the deviation of that value exceeds a given threshold.

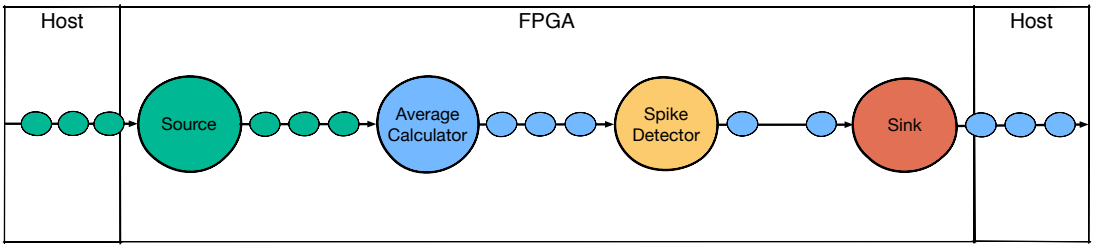
Let us suppose that our application needs to process values coming from temperature sensors. The minimum information we need is:

- `device_id`: a unique identifier of the device that produced the value;
- `temperature`: the temperature value perceived by the sensor.

This information represents the input tuple of our application.

The Data-Flow graph of our application will be a pipeline of 4 stages, as shown in the figure: We want to offload the pipeline computation entirely onto the FPGA.

Figure 3.2: FPGA Data Flow graph of Spike Detection



The Source operator receives the tuples from the host, which could read them from the

network or, as in our case, from a file. The tuples are sent by the Source operator to the Average Calculator operator with a KeyBy dispatch policy where `device_id` represents the distribution key. The Average Calculator operator is responsible for computing the moving average over the input data. The average is calculated over a window of tuples, so the Average Calculator operator has to maintain a state for each distribution key. A new tuple is defined, which contains all the information of the input tuple and the computed average. The resulting tuple is sent to the Spike Detector operator with a non-blocking RoundRobin dispatch policy. The Spike Detector operator checks the following predicate:

$$|x_n - \mu_n| > (threshold * \mu_n) \tag{3.1}$$

If it is False the tuple is dropped, otherwise the operator keeps it and sends it to the Sink operator with a RoundRobin dispatch policy in non-blocking mode. Finally, the Sink operator gathers the tuples and delivers them to the host program.

3.2.1 FSP Python APIs

To create the structure of this application using our framework, we need to describe each operator as follows:

Listing 3.1: FNode Python constructor

```
node = FNode(name,                      # The name of the operator
             par,                       # The parallelism degree of the operator
             kind,                      # The base operator to be implemented
             gather_policy,             # The gather policy
             dispatch_policy,           # The dispatch policy
             datatype = None,           # The output tuple datatype
             channel_depth = 0,         # The FIFO size of the output channels
             begin_function = False     # True if it has a begin phase, otherwise False
             compute_function = False,  # True if it has a compute phase, otherwise False
             end_function = False)      # True if it has an end phase, otherwise False
```

The parameter `kind` specifies which base operator the developer chooses to implement, selecting it from the following `enum`:

Listing 3.2: FNodeKind Python enumerator

```
class FNodeKind(Enum):
    SOURCE
    GENERATOR
    FILTER
    MAP
    SINK
    COLLECTOR
```

The parameter `gather_policy` specifies the gather policy that the operator has to utilize to receive tuples, including the two policies described in the previous chapter and

the NONE policy, which has to be utilized by the Source base operator as it does not have preceding operators.

Listing 3.3: FGatherPolicy enumerator

```
1 class FGatherPolicy(Enum):
2     NONE
3     BLOCKING
4     NON_BLOCKING
```

The parameter dispatch_policy specifies the dispatch policy that the operator has to utilize to send tuples to the next operator replicas, including the policies described in the previous chapter and the NONE policy which has to be utilized by the Sink base operator as it does not have subsequent operators.

Listing 3.4: FDispatchPolicy Python enumerator

```
1 class FDispatchPolicy(Enum):
2     NONE
3     FORWARD
4     RR_BLOCKING
5     RR_NON_BLOCKING
6     KEYBY
7     BROADCAST
```

The parameter datatype is an optional parameter, which specifies the datatype of the output tuple from the operator. If this parameter is omitted or is set to None, the datatype of the outgoing tuple will be the same as the input tuple.

The parameter channel_depth is an optional parameter and specifies the size of the FIFO buffer of the channels adopted in the implementation of the dispatch policy. If this parameter is omitted or it is set to 0, the offline compiler will choose the optimal size of these channels.

Lastly, the parameters begin_function, compute_function and end_function are optional parameters and can be set to True to enable the begin, compute and end phases respectively to the operator. The Map and the Filter operators have this parameter set to True by default, the other operators have to specify to them when declaring the corresponding FNode.

In the following listings we are going to show the declaration of each operator to compose the application pipeline.

In the declaration of the Source operator, which receives tuples from the host program, we have to set the gather policy to NONE. The dispatch policy has to be KEYBY as we want to distribute tuples of the incoming stream by device_id as a key. We do not specify the datatype of the outgoing tuple here, as this operator is used only to distribute tuples

provided by the host program.

Listing 3.5: Declaring the Source operator of the Spike Detection application

```
source_node = FNode('source',
                    source_par,
                    FNodeKind.SOURCE,
                    FGatherMode.NONE,
                    FDispatchMode.KEYBY,
                    channel_depth=16)
```

The Average Calculator operator is implemented as a Map base operator, as it applies a one-to-one transformation of the input tuples. The operator emits a tuple of type `tuple_t` that will contain the value of the calculated average along with the values present in the received tuple. The gather policy is set in non-blocking mode, as the previous logical operator will send tuples in KeyBy fashion and we want to be sure that this operator does not get blocked on a channel connected to a previous operator replica. Indeed, a Source operator replica might not send tuples to an Average Calculator replica for a certain amount of time, even if the incoming stream has a uniform distribution of tuples. We choose to apply the RoundRobin dispatch policy in blocking mode and the channels with a depth of 16 elements. Furthermore, we set the `begin_function` to True as we want to implement the begin phase in which we properly initialize the operator state.

Listing 3.6: Declaring the Average Calculator operator of the Spike Detection application

```
avg_node = FNode('average_calculator',
                 avg_par,
                 FNodeKind.MAP,
                 FGatherMode.NON_BLOCKING,
                 FDispatchMode.RR_BLOCKING,
                 'tuple_t',
                 begin_function=True)
```

The Spike Detector operator is implemented with the Filter base operator and it is in charge of removing or keeping tuples on the stream by checking the predicate mentioned above. Only outlier tuples will be sent to the Sink operator. Both the gather and the dispatch policies are set in non-blocking mode and the output tuple datatype is not specified.

Listing 3.7: Declaring the Spike Detector operator of the Spike Detection application

```
spike_node = FNode('spike_detector',
                   spike_par,
                   FNodeKind.FILTER,
                   FGatherMode.NON_BLOCKING,
                   FDispatchMode.RR_NON_BLOCKING)
```

Lastly, we declare the Sink operator that will receive tuples in non-blocking mode and no dispatch policy is specified.

Listing 3.8: Declaring the Sink operator of the Spike Detection application

```
sink_node = FNode('sink',
                  sink_par,
                  FNodeKind.SINK,
                  FGatherMode.NON_BLOCKING,
                  FDispatchMode.NONE)
```

We declare some constants that we will need both for declaring the state of Average Calculator and for implementing operator functions in OpenCL.

Listing 3.9: Declaring the constants of the Spike Detection application

```
win_dim = 16                              # window size
max_keys = 64                             # max num. of keys in total
avg_keys = round_up(max_keys // avg_par)  # max num. of keys per replica
threshold = 0.025                         # temperature threshold
constants = {'WIN_DIM': win_dim,
             'THRESHOLD': threshold,
             'MAX_KEYS': max_keys,
             'AVG_KEYS': avg_keys}
```

As shown above, the mean value is calculated over a window with a size of 16 items, and with a presumed maximum number of devices, and therefore of keys, of 64. Since the Average Calculator operator will be replicated according to its parallelism degree `avg_par`, and the stream is partitioned among replicas, the state size of each replica will be proportional to the number of keys in the stream. Thus, the number of windows to maintain will be exactly `max_keys` divided by `avg_par`. For efficiency purposes the number of windows is rounded to the next power of two. We then set the value of the threshold used by the Spike Detector operator to 0.025. Finally, we group these constants into a dictionary by specifying the name that each of these constants should have in the generated code.

Now, we present how to add the state to an operator. The following listings present the class functions of `FNode` that enable to add private, local and global state to an operator.

Listing 3.10: FNode class function to add a private buffer

```
def add_private_buffer(self,
                       datatype,
                       name,
                       size = 1,
                       value = None,
                       ptr = False,
                       attributes = None)
```

A private memory state is declared using the class function `add_private_buffer()` and has the following parameters:

- `datatype`: it is a string which specifies the datatype of the buffer. The datatype

can be a basic datatype or a developer-defined data structure.

- `name`: it is a string which specifies the name of the buffer.
- `size`: if it is set to 1, the resulting private buffer is a scalar; if it is set to a value > 1, the resulting private buffer is a unidimensional array; if its datatype is a Python Tuple or a Python List, the resulting private buffer is a multidimensional array.
- `value`: if it is passed as a string, it can be used to initialize the buffer with a constant or with a string; if it is passed as a python tuple or a python list, it can be used to initialize the private buffer with a multidimensional array of values.
- `ptr`: if the `size` parameter is set to 1, and this parameter is set to `False`, the resulting private buffer is passed to the phase functions as value, otherwise it is passed as a pointer. If the `size` parameter is > 1, this parameter is ignored.
- `attributes`: it is a Python dictionary and it is used to declare memory attributes that customize the on-chip memory architecture. For example it can be used to force the offline compiler to implement a private buffer in registers. Refer to Memory Attributes for Configuring Kernel Memory Systems on the Intel Programming Guide [15] on page 177 for more details about attributes.

A local memory state is declared using the class function `add_local_buffer()` and it has the same parameters of the `add_private_buffer()` class function, except for the `ptr` parameter as a local memory buffer is always passed by pointer to the phase functions.

Listing 3.11: FNode class function to add a local buffer

```
1 def add_local_buffer(self,
2                       datatype,
3                       name,
4                       size = 1,
5                       value = None,
6                       attributes = None)
```

A global memory state is declared using the class function `add_global_buffer()`, as shown in the following listing:

Listing 3.12: FNode class function to add a global buffer

```
1 def add_global_buffer(self,
2                        datatype,
3                        name,
4                        size = 1,
```

```
5                       access = FBufferAccess.READ_ALL,
6                       ptr = True):
```

The `datatype`, `name` and `size` parameters have the same semantics of the corresponding parameters of the previous two functions. The `access` parameters specifies the visibility of the declared global buffer and it can set to:

- `READ`: each replica has its own buffer in read mode. It could be used to make a partitioned state available to each replica.
- `WRITE`: each replica has its own buffer in write mode. It could be used to store updated information to the host program without performing reads on it.
- `READ_WRITE`: each replica has its own buffer in read-write mode. It could be used to a private global storage to implement a data structure.
- `READ_ALL`: all replicas share the same buffer in read mode. It could be used to make a global state available to all replicas, or it could be partitioned manually.
- `WRITE_ALL` all replicas share the same buffer in write mode. It could be used by all replicas to store updated information for the host program.
- `READ_WRITE_ALL`: all replicas share the same buffer in read-write mode. It could be used to implement a global shared memory data structure.

The `ptr` parameter specifies the implementation of the global buffer: it is implemented in global memory if the parameter is set to `True`, otherwise the global buffer is implemented as a scalar kernel parameter. This is could be useful when a scalar value or a developer defined data structure has to be passed as a value to the kernel and its value is known at runtime only.

The class functions shown above allow us to declare the state of the Average Detector operator, which is composed of two Direct Address Tables: one implemented as an array of integers in private memory to keep track of the number of tuples present in each window, and one to store the tuples contained in each window. The latter is implemented as a matrix of float numbers as we need to keep track only of the temperature value. It has a number of rows equal to the `map_keyes` value and a number of columns equal to `win_dim`.

Listing 3.13: Adding a private and a local state to the Average Calculator operator of the Spike Detection application

```
1 avg_node.add_private_buffer('int', 'sizes', size=avg_keys)
2 avg_node.add_local_buffer('float', 'windows', size=(avg_keys, win_dim))
```

To declare the operator pipeline, we need to create an object of the class `FPipe` that has the following parameters:

- `dest_dir`: it is a string which specifies the directory where the generated host and device code is placed.
- `datatype`: it is a string which specifies the datatype of the stream tuples that the host manage and sends to the Source operator.
- `codebase`: it is a string which specifies the directory where the operator files containing already implemented phase functions are stored.
- `constants`: it is a Python dictionary containing constants provided by the developer, which will be declared in both the host and device code. The constants related to the parallelism degree of each operator are generated by the framework and identified with `__<OPERATOR_NAME>_PAR`, e.g., `__SOURCE_PAR`.
- `transfer_mode`: accepts `TransferMode.COPY` and `TransferMode.SHARED`. The former is set by default and will generate the host and the device code implementing the N-Buffering technique. The latter is optional and will generate the host and the device code implementing the data transfers with the protocol for shared memory presented in previous chapter.

The `FPipe` class has the following class functions to add operators:

- `add_source(node)`: to add a Source base operator to the pipeline.
- `add(node)`: to add any node to the pipeline, except for Source and Sink base operators.
- `add_sink(node)`: to add a Sink base operator to the pipeline.

Furthermore, the `FPipe` class has the following class functions to finalize the pipeline and generate the host and the device code:

- `finalize()`: this function must be called before generating the code and performs the followings: it checks the operator names and verifies that they are unique; it infers and assigns the datatype of the input and the output tuples to operators; it properly constructs the channels used by the gather and dispatch policies implementation; it creates the necessary folders to contain the generated code.

- `generate_device()`: it is the function in charge of generating the device code, including: a file containing constants and functions used by the framework; a file containing data structures definitions necessary to the gather and dispatch policies implementation; a file containing the definition of datatypes specified by the developer in the `FNode` constructors; a file for each defined operator that will contain the definition of the selected phase functions; a file containing the actual device code, where operators are implemented as OpenCL kernels.

- `generate_host()`: it is the function in charge of generating the host code, including: some files containing wrapper functions of the OpenCL API; a file containing the runtime support for the management of the application; an example file where the application is launched.

We can finally declare our pipeline and add the previously declared operators. First, we create an `FPipe` object specifying the directory where to store the generated code, the tuple datatype of the stream provided to the Source operator, and the constants declared so far. Then, we add the operators to the pipeline by following the order depicted in the Figure 3.2.

Listing 3.14: Declaring and populating an FPipe to describe the Spike Detection application

```
1 pipe = FPipe('./spikeDetection', 'input_t', constants=constants)
2 pipe.add_source(source_node)
3 pipe.add(avg_node)
4 pipe.add(spike_node)
5 pipe.add_sink(sink_node)
```

Finally, we call the function `finalize()` and the two functions to generate the device and host code.

Listing 3.15: Generating the device and the host code of the Spike Detection application

```
1 pipe.finalize()
2 pipe.generate_device()
3 pipe.generate_host()
```

3.2.2 FSP Host and Device code

Starting from the description of the high-level Python application presented above, the framework generates the application implementation files in the directory specified by the developer. These files are distributed within the following directories:

- `common`: it contains supporting files for the host program implementation.
- `device`: it contains supporting files for the device program implementation, including a file for each operator that contains the signatures of the phase functions, and a the `device.cl` file containing the operator kernels.
- `host`: it contains an example file to show how to use the API generated by the framework to manage and to interface with the device operators.
- `includes`: it contains a file with the API generated by the framework for managing the application, and a file containing the datatype definitions specified by the developer in the constructors of the `FNode` operators.

In addition, the framework generates a Makefile that allows you to compile the host and device program.

3.2.3 Device Code

To actually implement the application, the developer has to provide the business logic of the operators by implementing the operator phase functions and completing the definition of the declared tuple datatypes. Inside the folder `./device/nodes`, we find a file for each operator containing the signatures of the declared phase functions and an implementation example for each of them. Inside the folder `./includes`, we find the file `tuples.h` where a `struct` is defined for each tuple datatype.

Moving forward with the SpikeDetection running example, we first complete the definitions of the datatype structs. Inside the file `tuples.h` we find the following code:

Listing 3.16: Tuples definitions generated by the FSP framework

```
1 typedef struct {
2     uint key;
3     float value;
4 } input_t;
5
6 inline uint input_t_getKey(input_t data) {
7     return data.key;
```

```
8 }
9
10 typedef struct {
11     uint key;
12     float value;
13 } tuple_t;
14
15 inline uint tuple_t_getKey(tuple_t data) {
16     return data.key;
17 }
```

This is the code that the framework autogenerated, which presents the two datatype definitions that we specified in the description of the operators: `input_t` and `tuple_t`. In addition, we find the function `<datatype>_getKey()` for each defined datatype, which allows us to implement the key extraction from the tuple needed by the KeyBy dispatch policy implementation. For the implementation of our application, we are going to modify the code as follows:

Listing 3.17: Tuples definitions customized for the Spike Detection application

```
1  typedef struct {
2      uint device_id;
3      float temperature;
4  } input_t;
5
6  inline uint input_t_getKey(input_t data) {
7      return data.device_id;
8  }
9
10 typedef struct {
11     uint device_id;
12     float temperature;
13     float average;
14 } tuple_t;
15
16 inline uint tuple_t_getKey(tuple_t data) {
17     return data.device_id;
18 }
```

The `struct input_t` has two fields: the information about the unique identifier of the device and the temperature value. We then implement the key extraction function by simply returning the value of the device id. The `struct tuple_t`, as we specified earlier, contains the same fields of the `input_t`, to which we add the `average` field to store the average calculated by the Average Calculator operator.

Now, we are going to show how to implement the business logic of the application operators. We find the `average_calculator.cl` file that contains the begin and the compute phase functions of the Average Calculator operator, and the `spike_detector.cl` file containing the compute phase function of the Spike Detector operator. The files of the Source and Sink operators are not present because we have not specified any phase function in their Python description.

In the following Listing 3.18 we show the signatures of the Average Calculator operator

phase functions that we generated:

Listing 3.18: Average Calculator phase functions generated by the FSP framework

```
inline void average_calculator_begin(__private int sizes[AVG_KEYS],
                                     __local float windows[AVG_KEYS][WIN_DIM])
{
    // initialize the operator state
}

inline tuple_t average_calculator_compute(input_t in,
                                          __private int sizes[AVG_KEYS],
                                          __local float windows[AVG_KEYS][WIN_DIM])
{
    // apply the computation function to 'in' and store result to 'out'
    tuple_t out;
    return out;
}
```

Both functions allow the state of the operator that we declared in the Python description to be accessed. Inside the begin phase function, we initialize each item of the array `sizes` with zeros. We do not initialize the array `windows` because, as we are going to show in the implementation of the compute function, it is not useful for our purposes.

Listing 3.19: Average Calculator begin phase function implementation for the Spike Detection application

```
inline void average_calculator_begin(__private int sizes[AVG_KEYS],
                                     __local float windows[AVG_KEYS][WIN_DIM])
{
    #pragma unroll
    for (int i = 0; i < AVG_KEYS; ++i) {
        sizes[i] = 0;
    }
}
```

We specify the directive `pragma unroll` because the number of elements in the array is small and allows us to initialize the whole array in one clock cycle.

We have to keep in mind that the compute phase function must calculate the average over a moving window of the last received `WIN_DIM` tuples. A possible implementation is as follows:

Listing 3.20: Average Calculator compute phase function implementation for the Spike Detection application

```
inline tuple_t average_calculator_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
```

```
16      for (uint i = 0; i < WIN_DIM - 1; ++i) {
17          windows[idx][i] = windows[idx][i + 1];
18          sum += windows[idx][i];
19      }
20      windows[idx][WIN_DIM - 1] = val;
21      sum += val;
22
23      tuple_t out;
24      out.device_id = in.device_id;
25      out.temperature = in.temperature;
26      out.average = sum / sizes[idx];
27
28      return out;
29  }
```

We define two support variables:

- `idx`: it stores the corresponding index of the device on the Direct Address Table implemented in local memory and identified by `windows` parameter. We calculate its value by dividing the value of `device_id` by the parallelism degree of the Average Calculator operator.
- `val`: it stores the temperature value of the input tuple.

In lines 8-12 we update the number of tuples received for that particular key within the array `sizes`. We declare the variable `sum` to accumulate the values of the window and use the shift-register pattern on the window at position `idx`. All values of the window are shifted to the left, thus eliminating the first element of the window that is the oldest inserted tuple, and the value of the current tuple is stored at the last position of the window. During this procedure, the accumulation of the values in the window is performed. On the following lines, the output tuple with datatype `tuple_t` is declared and populated with the values of the input tuple and the average calculation. Lastly, the new tuple is returned.

We compiled the device code with this implementation and we found that, in the `Throughput Analysis/Fmax II report` section of the report, the `II` is 2 and thus the replication operator is able to process a tuple every 2 clock cycles. Moreover, we found in the `System Viewers/Kernel Memory Viewer` section that the array `sizes` has been implemented as BRAM by the offline compiler. To solve these two problems, we can specify the `register` attribute on the `sizes` array to force the offline compiler to implement the array using registers. Recompiling the code with this change, the report shows that, indeed, the array `sizes` is implemented in registers and the `II` is now 1. Unfortunately, the `Fmax` value has dropped to about 194 compared to 240 which is the target value we usually aim for. Another option is to change the datatype of the array from

`int` to `char` since new datatype can store the maximum value `WIN_DIM`. We recompiled again and we get the implementation of the array in registers and `II` = 1, but again `Fmax` = 194.

The implementation of the Spike Detector operator is straightforward as it is necessary to specify the predicate seen earlier in the Equation 3.1:

Listing 3.21: Spike Detector compute phase function implementation for the Spike Detection application

```
1 inline bool spike_detector_function(tuple_t in)
2 {
3     return (fabsf(in.temperature - in.average) > (THRESHOLD * in.average));
4 }
```

For the sake of completeness, we show how to add a Random Number Generator (RNG) to the Generator base operator, which we use in our benchmark tests. We implemented a Random Number Generator, as shown in the Listing 3.22, based on the *Linear-feedback shift register* (LSFR) algorithm and we provide one function to generate an integer number and one to generate a floating point number.

Listing 3.22: Random Number Generator implementation

```
1  typedef union {
2      unsigned int i;
3      float f;
4  } rng_state_t;
5
6  inline unsigned int next_int(rng_state_t * s)
7  {
8      s->i = (s->i >> 1) | (((s->i >> 0) ^ (s->i >> 12) ^ (s->i >> 6) ^ (s->i >> 7)) << 31);
9      return s->i;
10 }
11
12 // Generates a float random number in the range [1.0, 2.0[
13 inline float next_float(rng_state_t * s)
14 {
15     rng_state_t _s;
16     s->i = (s->i >> 1) | (((s->i >> 0) ^ (s->i >> 12) ^ (s->i >> 6) ^ (s->i >> 7)) << 31);
17     _s.i = ((s->i & 0x007fffff) | 0x3f800000);
18     return _s.f;
19 }
```

Both functions take a pointer to a `rng_state_t`. The `next_int()` returns an integer number, while the `next_float()` function returns a floating point number between 1.0 included and 2.0 excluded.

To enable a random number generator, the developer has to add a RNG state to the Generator base operator in the Python description of the application as shown in the Listing 3.23.

Listing 3.23: Add a Random Number Generator state to the Generator base Operator

```
1 generator_node = FNode(...)
```

```
2 generator_node.add_rng_state('rng')
```

The developer can specify more than one RNG state. We strongly suggests to use a RNG state for each number generator function call, as it is a lightweight component and this way it does not cause memory dependencies.

Host Code

The listings that we are going to show concern the host program API generated by the FSP framework that is used to interact with the operators on the device. Each operator defined in the Python application description is implemented as a `C++ struct` in the host program containing specialized functions to interact with the operator replicas on the device and with its state. In the host operator implementation we find all the OpenCL objects needed to launch the replicas kernels, some buffers to maintain the operator state, and the support functions for writing and reading the state of each replica.

Listing 3.24: Host operator implementation

```
1  struct <operator_name>
2  {
3      std::string name;
4      size_t par;
5      ...
6      <operator_name>(const std::string name, const size_t par)
7      : name(name), par(par) {
8          // For each replica
9          for (size_t i = 0; i < par; ++i) {
10             // Create a 'cl_command_queue'
11             // Create a 'cl_kernel'
12         }
13         // Create 'cl_mem' and 'cl_command_queue' for each global buffer (except scalars)
14     }
15
16     // Declare a "write" function for each global scalar
17     void write_<buffer_name>(const <buffer_datatype> value,
18                              [const size_t replica_id]) {
19         // Set the value to the corresponding scalar
20     }
21
22     // Declare a "write" function for each global buffer with write access
23     void write_<buffer_name>(const <buffer_datatype> * buffer,
24                               const size_t buffer_size,
25                              [const size_t replica_id]) {
26         // Copy host data pointed by 'buffer' to the corresponding device buffer
27     }
28
29     // Declare a "read" function for each global buffer with read access
30     <buffer_datatype> * read_<buffer_name>([const size_t replica_id]) {
31         // Copy back data from device buffer
32     }
33
34     void launch_kernels() {
35         // For each replica
36         for (size_t i = 0; i < par; ++i) {
37             // Set kernel arguments to the corresponding kernel
38             // Launch the kernel calling clEnqueueTask() function
39         }
40     }
41
42     void finish() { /* Wait until all kernels complete the computation */ }
```

```
43      void clean() { /* Release all OpenCL stuff */ }
44  };
```

In particular, within the operator definition, a function is generated for each global scalar and for each global buffer to manage write and read operations on them. These functions have the parameter `replica_id` used to specify the replica on which to perform the write and read operations. If a global buffer has been declared with `<access>_ALL` access mode, the `replica_id` parameter is not generated because only one global buffer is created for all replicas.

The Source and Sink host operators start with the same implementation of the Listing 3.24, and are augmented with the necessary data structures and functions to implement a push operation of the new micro-batch by the Source operator and a pop operation to fetch a computed micro-batch. The implementation of the `push()` function is based on the implementation of the Listing 2.15, and has been modified to support the push of a new micro-batch on a specific replica of the Source base operator. The signature of this function is as follows:

Listing 3.25: `pop()` signature function of the Host Sink base operator

```
1  void pop(const <buffer_datatype> * batch,
2           size_t * batch_size,
3           const replica_id,
4           bool * close)
```

Once all host operators are defined, the `struct FPipe` representing the host-side application is created. In the constructor, the parameter `aocx_filepath` specifies the bitstream to use. The two parameters, `source_batch_size` and `sink_batch_size`, specify the size of the micro-batches adopted by the Source and the Sink operators respectively. Furthermore, if the developer has set `TRANSFER_COPY` to manage the communication between the host and the device, the parameter `number_of_buffers` is generated and so it is possible to define the number of buffers to be used for the N-Buffering technique. The functions `push()` and `pop()` are wrappers of the respective functions of the Source and Sink host operators.

Listing 3.26: Host FPipe implementation

```
1  struct FPipe {
2
3      std::string aocx_filepath;
4      size_t source_batch_size;
5      size_t sink_batch_size;
6      // Declare all operators
7      ...
8
```

```
    FPipe(const std::string aocx_filepath,
          const size_t source_batch_size,
          const size_t sink_batch_size,
          [const size_t number_of_buffers])
    : aocx_filepath(aocx_filepath)
    , source_batch_size(source_batch_size)
    , sink_batch_size(sink_batch_size)
    // Construct all operators
    ...
    {}

    void start() { /* Call the function launch_kernels() on each operator */ }

    // Wrapper of the 'push()' function of the Source node
    void push(...) { ... }

    // Wrapper of the 'pop()' function of the Sink node
    <buffer_datatype> * pop(...) { ... }

    void wait_and_stop() { /* Call the 'finish()' function on each operator */ }

    void clean() { /* Call the 'clean()' function on each operator */ }
};
```

The framework also generates a host file example showing the use of `FPipe`:

Listing 3.27: Host file example

```
...
int main(int argc, char * argv[]) {
    ...
    FPipe pipe(acox_filepath, source_batch_size, source_buffers, N);
    // Prepare all buffers for all internal nodes
    // e.g. pipe.filter.write_constant(42);
    pipe.start();
    // Create a thread for each Source operator replica to manage input streams
    // Create a thread for each Sink operator replica to manage output streams
    pipe.wait_and_stop();
    pipe.clean();

    return 0;
}
```

In the main function, an object of type `FPipe` is declared with specified attributed such as the filepath of the bitstream, the size of the micro-batches of the Source and Sink operators, and the number of buffers to be used for N-Buffering communication. In the following lines, we show how to use the functions to initialize the state of an operator. The `start()` class function is called on the pipeline to start the device computation. A number of threads equal to the number of replicas of the Source operator are created to handle the input streams. The same is done for the Sink operator to handle the output streams coming from the device. Then the `wait_and_stop()` function is called, which pauses the main thread until the computation is complete. Lastly, the `clean()` function is called. It releases all OpenCL resources created by the host program.

Chapter 4

Evaluation

4.1 Test Applications

In this last part of the thesis, we discuss some benchmarks based on the Spike Detection application presented in the previous chapters. Furthermore, we discuss the results of some additional tests that show the potential of our framework and the use of the FPGA for stream processing applications. We implemented three different versions of the application targeting the FPGA:

- Base: this version has the same implementation described in the previous chapters except for the Average Calculator operator, which uses an optimized compute phase function resulting in `II` $= 1$ that can be found in the Listing A.1 of the Appendix. The `transfer_mode` is set to `TRANSFER_COPY` and we set the number of buffers to 4 for the N-Buffer technique.

- Shared: this version is the same as the Base one except for the `transfer_mode`, which is set to `TRANSFER_SHARED`.

- Skeleton: this version uses a Generator operator in place of the Source Operator, which is in charge of generating tuples directly on the device instead of continuously transfer them through the global memory in batches. Similarly, the Sink operator has been replaced with a Collector operator in order to avoid writing results to the global memory in order to be used by the host. The Average Operator and the Spike Operator implementations are unchanged. With this application, we test the peak theoretical performance of the pipeline implemented on the FPGA without

any global memory interaction and without any interaction with the host except for the kernel launches.

In addition, we execute some rests with the Spike Detection application implemented using the WindFlow library, which we use to compare our results on the FPGA against the one obtained on a shared-memory server machine.

4.2 Results

We evaluate our applications targeting FPGA on a Intel Arria 10 SoC FPGA that features a dual-core ARM Cortex-A9 MPCore Hard Processor System, 660K Logic Eleements, 250K Adaptive Logic Modules (ALM), 1M Registers, 42,620 M20K, 5,788 MLAB, 1687 variable-precision DSPs and 1GB on-board DDR4-2400. The host program is compiled with the `arm-linux-gnueabihf-g++` cross-compiler with `-O2` optimizer flag. To compile the device program, we use the Intel FPGA SDK for OpenCL Offline Compiler `aoc` with `-g0` flag, which removes source information from the compiler reports and source code and customer IP information from the bitstream file.

We adopt the `boardtest` benchmark tool provided by Intel to test the Host-to-Memory bandwidth [16] on the SoC FPGA, and we measured a top write speed of 104 MB/s and a top read speed of 62 MB/s.

We evaluate the WindFlow version on a machine equipped with two CPUs AMD EPYC 7551 with 128GB of RAM. Each CPU has 32 cores (64 hard-ware threads) with groups of four cores sharing an L3 cache of 8MB. Each core has a clock rate of 2.4 GHz and an L2 of 512KB. The WindFlow version is compiled with the `gcc 9.0.1` compiler with `-O3` optimizer flag.

The Base, the Shared and the WindFlow applications process the same dataset [17], whereas in the Skeleton application the Generator base operator generates tuples by using the RNG engine provided by the framework. Each run of the Base and the Shared application is launched with 4096 micro-batches varying the micro-batch size. Each benchmark of the WindFlow version is executed for 60 seconds.

We run the Base application in different configurations in terms of parallelism degree and we conclude that replicating the Source operator is not useful because the Host-to-Memory bandwidth is a bottleneck. Moreover, in terms of tuples per second, by replicating

the Average Calculator and the Spike Detector operators we obtain the same results that we would by not replicating them; therefore, we advise to spare the hardware resources and avoid replication.

For the Shared application we also run different configurations in terms of parallelism degree, and we conclude that replicating the Source operator more than twice does not improve the performance as we expect, as the Host-to-Memory bandwidth is still a bottleneck. As shown in Figure 4.1, with two or four replicas of the Source operator we obtain a better bandwidth compared to the one with only one replica. With micro-batches of 1024 elements we reach a peak of 132 MB/s, that is even more than the one measured with the `boardtest` benchmark tool, but increasing the micro-batch size the performance only degrades: in this case as well, replicating the Average Calculator and the Spike Detector operators more than the Source operator parallelism degree does not improve the performance.

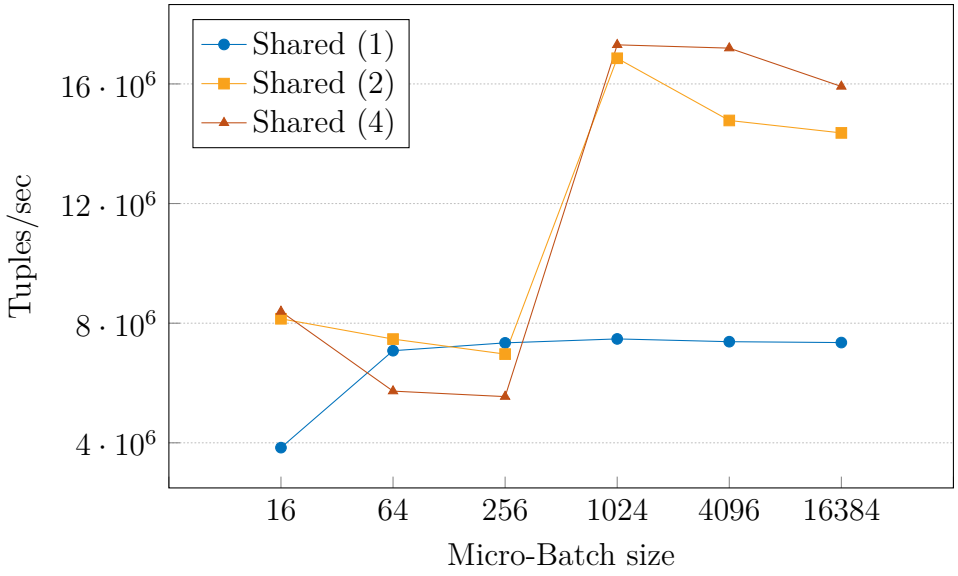


Figure 4.1: Results of the Shared application with Source operator with different parallelism degrees.

We collect the bandwidth measurements of the Base and the Shared applications in terms of tuples per second by varying the micro-batch sizes. The results are shown in Figure 4.2.

We test the Base and the Shared versions of the application with the Source operator replicated 1 and 2 times. For the Base application we can see that as the size of the micro-batch increases, the bandwidth increases as well in both configurations. This is not

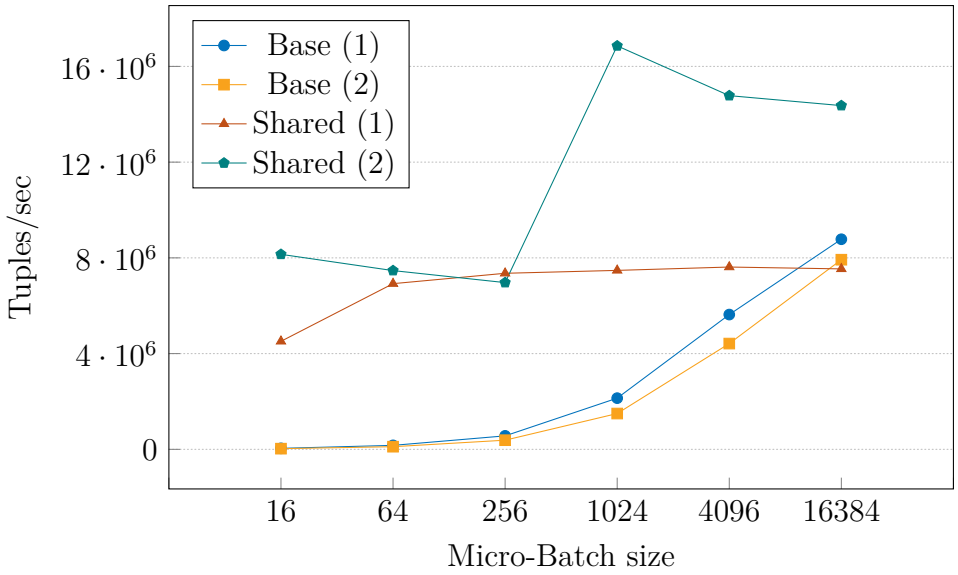


Figure 4.2: Comparison between the N-Buffer technique and Shared Memory protocol with different micro-batch sizes.

a surprising behavior since the communication overhead becomes less and less important as the size of the micro-batches to be transferred increases. Furthermore, as the data available to the FPGA increases, the performance of the device computation increases correspondingly. Indeed, with a micro-batch of size 16384 we can compute almost 9M tuples per second. The performance with small sized micro-batches is poor because the communication overhead dominates and can not be entirely overlapped by the computation. In the Share application with Source with parallelism degree 1, we note that it performs quite well with small sized micro-batches, managing to compute about 4M tuples per second with a batch size of 16 tuples. With batches of 64 tuples and higher, the number of computed tuples does not increase and reaches a maximum of 7M tuples per second. By replicating the Source operator twice, we observe a peculiar trend: using micro-batches of 16 tuples, we obtain just over 8M tuples per second, dropping to 7M tuples per second with a batch of size 256. With a batch of size 1024 we get the peak of the bandwidth, which is around 17M tuples per second, and as the batch size increases the bandwidth obtained decreases.

From these considerations, we can state that the Shared Memory Protocol performs much better than the N-Buffer technique for small sized micro-batches. This aspect is particularly important since in real applications the use of oversized batches would require very high input rates (to maintain acceptable latency), and this is not always justifiable.

Since in our tests we are limited by the memory bandwidth between the host and the FPGA, we run the benchmarks with the Skeleton application to show the potential capability of the code generated by our framework. In a configuration where the operators Source and Sink have a degree of parallelism set to 1 and the operators Average Calculator and Spike Detector have a degree of parallelism set to 2, we get a maximum bandwidth of about 200M tuples per second. If we increase the number of replicas of all operators except the Sink, and set the degree of parallelism to 4, we get a bandwidth of about 500M tuples per second. These are ideal results with no access to global memory, as the tuples are generated directly by the Generator operator on the FPGA. These results are remarkable, because the code generated by the framework has great potential and the implementation is only limited by Host-to-Memory bandwidth of the considered SoC used for the experiments. In a scenario where the bandwidth is much higher than the one we measured, or the tuples of the streams come from a network interface (such as a 10GB Ethernet) we could exploit the full capabilities of the FPGA computation for DSP applications generated by our framework.

In Figure 4.3 we show the results of the WindFlow implementation of the Spike Detection application by varying the size of the micro-batches. The results are obtained by choosing the best parallel configuration on that machine: the Source operator has a parallelism degree set to 17, and the Average Calculator, the Spike Detector and the Sink operators are chained so that they form a single operator which is replicated 12 times, with a total number of 29 threads. From these results we note that, as the size of the micro-batches increases, the number of tuples per second computed increases as well.

Figure 4.4 shows a comparison of the best results obtained by the versions of Spike Detection implemented using both our framework and the WindFlow library. As we can observe, the Base and Shared applications are about 4.3 and 2.3 times slower than the WindFlow one. With regards to the results of the Skeleton application, however, we note that there is a lot of room for improvement even using only one replica for the Source operator, in the scenario of having no limits on the memory bandwidth. We also note that the WindFlow version is about 5 times slower and about 10 times slower than the Skeleton application, with respectively 1 and 4 parallelism degree for the Source operator.

As a last point, we perform a test to verify the necessity of operator replication.

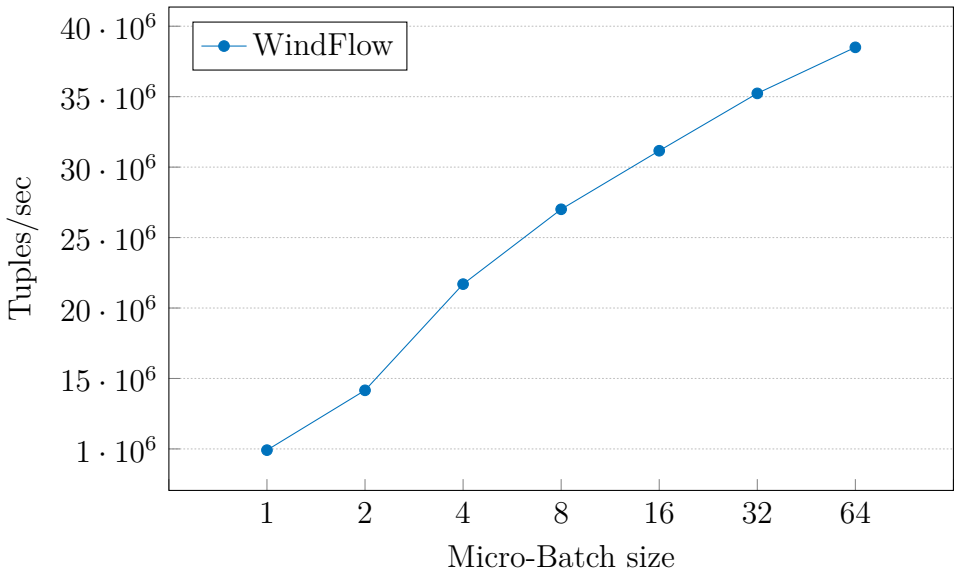


Figure 4.3: Results of the WindFlow implementation of Spike Detection by varying the micro-batch size.

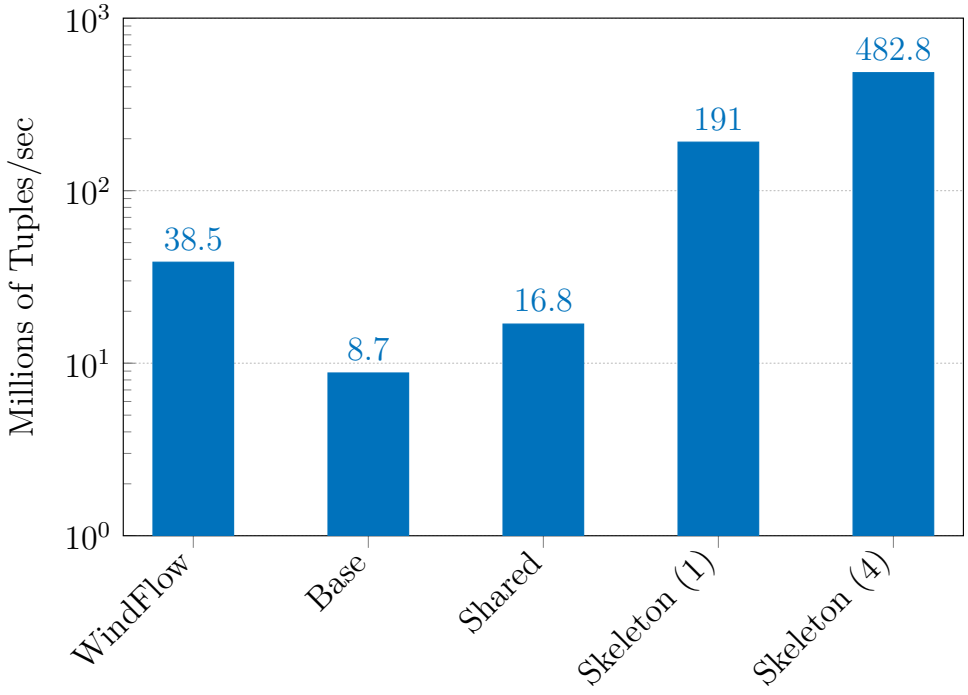


Figure 4.4: Comparison between the best results of all the Spike Detection versions.

We implemented the Average Calculator operator in an inefficient way and with data dependencies such as *write-after-read* dependency. This has the general goal of mimicking a situation that could happen during the development of real-world complex codes for the FPGA. Indeed, it might be possible that, due to the complexity of the code or due to the inexperience of the high-level programmer, the operator business logic compilation does

not result in optimized FPGA code with low `II`. The report shows that the operator is compiled with `II` = 22, and consequently it is able to compute a tuple every 22 loop iterations. In the Figure 4.5 we show the results of two inefficient configurations: one with all operators replicated only once, the other with the Average Calculator operator replicated 4 times. Comparing the results we obtain a speedup of 3, that is lower than the

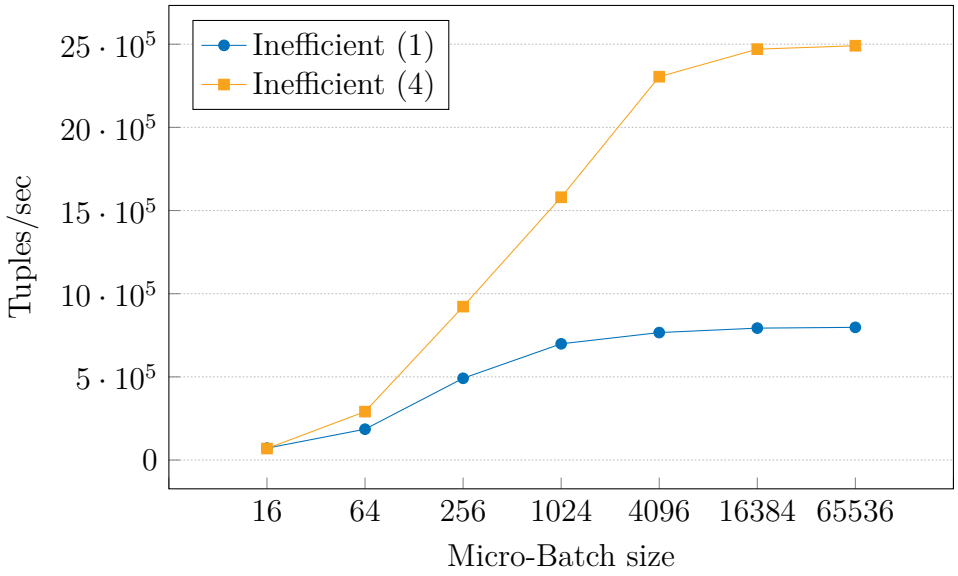


Figure 4.5: Results of the Inefficient version with Average Calculator operator replicated 4 times and with `II` = 22.

ideal one of 4. Therefore, the operator replication proves useful at least in cases where it is not possible to develop the operators' business logic code such that the offline compiler reaches `II` = 1.

Chapter 5

Conclusions

The aim of this project was to provide a solid foundation to the development of DSP application targeting FPGA. Within this work we introduced the components necessary to implement streaming application on FPGA. Among these components, we focused on the proper implementation of operators, on the communication between the host and the device, the inter communication between operators running on the FPGA, and so on. We developed a set of Python APIs to describe a DSP application and to generate the host and the device OpenCL code, to which developers can add the business logic code of operators. We used a running example to introduce the APIs step by step, while also explaining the reasons for the solutions we implemented. We showed the OpenCL code that our framework generated starting from the description of the example application Spike Detection.

Finally, we performed some benchmarks of the application produced by our framework. We evaluated the two implementations we proposed for the communication between host and device: the N-Buffering technique that makes use of the OpenCL API, and the share memory protocol we designed. We then compared it with the Spike Detection application implemented with the WindFlow library, and we observed that the bandwidth obtained by our implementations on the FPGA are respectively 4.3 and 2.3 times slower. From these tests, we noticed that the Host-to-Memory bandwidth is the bottleneck of our implementations. For this reason, we developed a new version in which we put the Generator operator in place of the Source operator, in order to generate the tuples inside the device and to prevent the operators on the FPGA from interacting with the host program. This version showed the potential of using our framework to target FPGAs for

streaming applications, obtaining up to 10 times the bandwidth reached by the application developed with the WindFlow library.

Considering the results achieved in this study, we aim to further develop this framework by providing new base operators such as the FlatMap operator which applies a user-defined transformation on each received tuple producing zero, one or more output tuples. In addition, we believe it would be useful to provide APIs to directly implement windowed operators, which execute a windowed query on key based streams partitions. We would also like to further improve the Share Memory Protocol we developed, in order to reduce the overhead of the busy waiting approach we adopted. Furthermore, there would be a need to test more applications and run them on different hardware configurations to consolidate the results collected so far. Finally, we believe we could achieve optimal results using autorun kernels to implement our dispatch policies, with the aim to improve performances and reduce usage of hardware resources.

Appendices

Appendix A

Average Calculator optimized compute phase function

Listing A.1: Average Calculator compute phase function optimized (`II` = 1) implementation for the Spike Detection application

```
1  inline tuple_t average_calculator_compute(input_t in,
2                                            __private int sizes[AVG_KEYS],
3                                            __local float windows[AVG_KEYS][WIN_DIM])
4  {
5      const uint idx = in.device_id / __AVERAGE_CALCULATOR_PAR;
6      const float val = in.temperature;
7
8      float N = 0.0f;
9      #pragma unroll
10     for (uint i = 0; i < WIN_DIM; ++i) {
11         if (sizes[idx] == ((1 << i) >> 1)) N = 1.0f / (i + 1);
12     }
13
14     if (sizes[idx] & (1 << (WIN_DIM - 2))) {
15         sizes[idx] = (1 << (WIN_DIM - 2));
16     } else {
17         sizes[idx] = (sizes[idx] == 0 ? 1 : sizes[idx] << 1);
18     }
19     float sum = 0.0f;
20     #pragma unroll
21     for (uint i = 0; i < WIN_DIM - 1; ++i) {
22         windows[idx][i] = windows[idx][i + 1];
23         sum += windows[idx][i];
24     }
25     windows[idx][WIN_DIM - 1] = val;
26     sum += val;
27
28     tuple_t out;
29     out.device_id = in.device_id;
30     out.temperature = in.temperature;
31     out.average = sum * N;
32     return out;
33 }
```

Bibliography

[1] Apache Storm: Distributed and Fault-Tolerant Real-Time Computation. URL: https://storm.apache.org.

[2] Apache Flink: Scalable Batch and Stream Data Processing. URL: https://flink.apache.org.

[3] Gabriele Mencagli et al. "WindFlow: High-Speed Continuous Stream Processing with Parallel Building Blocks". In: *IEEE Transactions on Parallel and Distributed Systems* (2021), pp. 1–1. DOI: 10.1109/TPDS.2021.3073970.

[4] Jinja 2.11.x. URL: https://jinja.palletsprojects.com/en/2.11.x/.

[5] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. 1st. USA: Cambridge University Press, 2014. ISBN: 1107015545.

[6] Gabriele Mencagli and Tiziano De Matteis. "Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach". In: *International Journal of Parallel Programming* 45 (Apr. 2017). DOI: 10.1007/s10766-016-0413-x.

[7] Brian Babcock et al. "Models and Issues in Data Stream Systems." In: June 2002, pp. 1–16. DOI: 10.1145/543613.543615.

[8] Wenhong Tian and Yong Zhao. "2 - Big Data Technologies and Cloud Computing". In: *Optimized Cloud Resource Management and Scheduling*. Ed. by Wenhong Tian and Yong Zhao. Boston: Morgan Kaufmann, 2015, pp. 17–49.

[9] Marco Aldinucci et al. "Fastflow: High-Level and Efficient Streaming on Multicore". In: *Programming multi-core and many-core computing systems*. John Wiley & Sons, Ltd, 2017. Chap. 13, pp. 261–280. ISBN: 9781119332015. DOI: https://doi.org/

10.1002/9781119332015.ch13. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13.

[10] Gabriele Mencagli. *WindFlow: a C++17 Data Stream Processing Parallel Library for Multicores and GPUs*. URL: https://paragroup.github.io/WindFlow/.

[11] Khronos Group. *OpenCL™*. URL: https://www.khronos.org/opencl/.

[12] Intel. *Intel® FPGA SDK for OpenCL™ Pro Edition - Best Practices Guide*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl-best-practices-guide-19-1.pdf.

[13] Da Tong, Shijie Zhou, and Viktor K. Prasanna. "High-Throughput Online Hash Table on FPGA". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015, pp. 105–112. DOI: 10.1109/IPDPSW.2015.149.

[14] Scott Schneider and Kun-Lung Wu. "Low-Synchronization, Mostly Lock-Free, Elastic Scheduling for Streaming Runtimes". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 648–661. ISSN: 0362-1340. DOI: 10.1145/3140587.3062366. URL: https://doi.org/10.1145/3140587.3062366.

[15] Intel. *Intel® FPGA SDK for OpenCL™ Pro Edition - Programming Guide*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-19-1.pdf.

[16] Intel. *Intel® FPGA SDK for OpenCL™ Pro Edition - Custom Platform Toolkit User Guide*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/ug_aocl_custom_platform_toolkit.pdf.

[17] Intel Berkeley Research lab. URL: http://db.csail.mit.edu/labdata/labdata.html.