The Journal of Supercomputing (2024) 80:13048–13078
https://doi.org/10.1007/s11227-024-05894-4

Boosting general-purpose stream processing with reconfigurable hardware

Alberto Ottimo[1] · Gabriele Mencagli[1] · Marco Danelutto[1]

Accepted: 4 January 2024 / Published online: 27 February 2024

Abstract

Reconfigurable devices such as field-programmable gate arrays (FPGAs) offer flexible solutions to workload acceleration with high energy efficiency. Despite such a potential advantage, they often reveal hard to program by application programmers. High-level synthesis languages have been developed to provide higher-level abstractions, allowing the developers to define the FPGA behavior using an imperative programming approach based on C/C++ languages. However, such approaches still leave the developer with the responsibility to harness the low-level optimizations required to develop efficient FPGA programs. Along this line, this paper introduces FSPX, a framework helping programmers to develop FPGA-accelerated *data stream processing* (DSP) applications. The approach provides a high-level Python API to develop the data-flow graph of operators, which is automatically translated into an efficient Vitis source code targeting Xilinx devices. The execution of the bitstreams implementing two benchmark applications showcases the efficiency of using FPGAs for DSP workloads. In general, FSPX provides, with a reasonable time-to-solution, higher performance compared with state-of-the-art DSP frameworks.

Keywords Data stream processing · Stream processing engines · FPGA · OpenCL · Xilinx

✉ Alberto Ottimo
alberto.ottimo@phd.unipi.it

Gabriele Mencagli
gabriele.mencagli@unipi.it

Marco Danelutto
marco.danelutto@unipi.it

[1] Department of Computer Science, University of Pisa, 56127 Pisa, Italy

1 Introduction

An increasing number of applications need to process huge volumes of data in the form of *data streams*, i.e., unbounded sequences of inputs received at high speed. Applications in this domain often rely on the *data stream processing* [1] (DSP) computing paradigm to process continuous streams and extract valuable data on the fly. According to this paradigm, applications are data-flow graphs executed on the underlying physical computing resources by *stream processing engines* (SPEs), which facilitate their development and deployment.

DSP requires the use of parallel hardware to deliver high throughput and low latency. Traditional SPEs (e.g., Apache Storm [3] and Flink [2]) are designed for distributed environments, where the focus is to efficiently and seamlessly process multiple applications on homogeneous clusters. Some specialized SPEs target scale-up machines equipped with multi-core CPUs and GPUs (e.g., as in [13]). However, a significant challenge lies in the expressiveness of such tools in terms of supported applications. Many existing systems focus on specific DSP domains (such as relational algebra [4] or Complex Event Processing [8]), making their API very user-friendly and abstracting most of the low-level implementation details. However, this specialization often sacrifices flexibility, making the execution of generic applications operating on any data type unfeasible.

Field-programmable gate arrays (FPGAs) are powerful accelerators but notoriously challenging to program. Writing code using RTL (register transfer level) languages requires specific engineering expertise. In recent years, high-level synthesis (HLS) tools [7] have been developed to address this challenge. While HLS languages offer high-level programming abstractions, developing efficient hardware designs to run on FPGAs is still a hard task and needs code that can be efficiently synthesized by the compiler. To make DSP accessible on FPGAs, new approaches are needed to simplify application development while maintaining high flexibility in terms of the variety of applications supported. In this regard, the literature is lacking: programmer-friendly approaches are restricted to specific domains (e.g., relational algebra), while general-purpose use cases are addressed by relatively few incomplete works (i.e., in terms of features supported) or with implementations no longer maintained or targeting obsolete FPGA devices (more details in §6).

This paper presents FSPX, a framework to accelerate general-purpose DSP applications on FPGAs. Our approach starts with a user-friendly Python DSL (domain-specific language) to define the data-flow of operators, declaring the kinds of operators and their interconnection topology in an easy manner. The execution of such a Python program produces, through a code generation process, the code targeting the FPGA platform. Such a code efficiently implements operators, and their interconnection with predefined data distribution policies, and addresses the replication of operators to increase processing throughput on the device. So, the application programmers are no longer responsible for the low-level implementation details, while they are requested to provide the business logic code of each operator as a sequential program (e.g., incorporated in functors having the right signature).

The specific research contributions of this paper are the following:

- FSPX extends the previous work of FSP [15], which was capable of generating OpenCL device code for Intel FPGAs only. We introduce a new code generation layer to target Xilinx FPGAs, preserving all the compatible features of FSP;
- we design a new device library from scratch developed for Vitis HLS, implementing efficient data-flow components (e.g., to perform data distribution, data collection, and operator replication) tailored for Xilinx FPGAs. The library includes all the components needed by the device runtime, greatly simplifies the code generation process, and minimizes the resulting code footprint;
- the experimental evaluation is based on two real-world streaming applications. The evaluation shows that FSPX considerably outperforms industrial and research SPEs (i.e., Apache Flink and WindFlow [13]);
- the source code of FSPX and all the applications implemented with all frameworks used in the comparison have been released open source[1].

The paper is organized as follows. §2 introduces the main concepts of the DSP paradigm and the FPGA architecture. §3 provides an overview of FSPX and its Python DSL. §4 shows the implementation in detail, both the device library and the host library. §5 presents the experimental evaluation with the results in terms of throughput and latency. §6 outlines related works and §7 draws the conclusion of this paper.

2 Background

In this section, we review the basic concepts needed to understand the paper. We focus on the DSP paradigm first and then give the essential information about FPGAs and their programming model.

2.1 Data stream processing

DSP applications can be modeled as data-flow graphs, where arcs are unbounded *data streams* and vertices are *operators* doing intermediate computational stages applied over the inputs to produce outputs. In relational streaming applications (i.e., the ones that can be developed using SQL-like formalisms [4]), operators are the ones of relational algebra (e.g., projection, selection, joins) while graphs have regular structures (e.g., trees). In general-purpose DSP [19], which is the paradigm of our interest, graphs are generic and operators execute generic computations.

Figure 1 shows an example of a DSP application having one source, two intermediate operators, and a sink. This structure is the *logical* one, showing only operators and their connections. However, the application can expose different parallelism exploitation patterns [16]. In particular, each operator can be internally replicated to work in parallel on different inputs as shown in Fig. 2, which can be considered

[1] FSPX is available at this link: https://github.com/blackwut/FSPX.

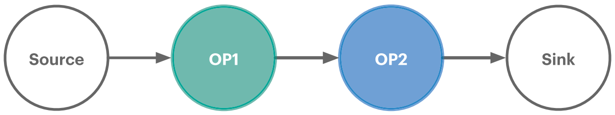
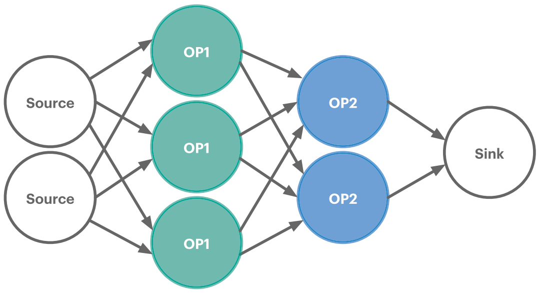


Fig. 1 Logical view of a DSP application having four operators

Fig. 2 Physical view of a DSP application having four operators with different numbers of replicas each



the *physical* view of the graph. Replicas are instances of the same processing logic working on different inputs distributed by the previous operator. Data distributions can follow different dispatching policies: the *round-robin policy* tries to assign the same number of inputs to the replicas by distributing them in a circular manner to the destinations, while the *key-by policy* assigns all inputs having the same key (e.g., a specific field of the input) to the same replica.

Depending on the nature of the computation, operators can be *stateless*, if their outputs depend on the inputs only. Otherwise, operators are *stateful* if they maintain a state used to compute outputs. A very common pattern is to use stateful operators with a key-by dispatching policy (i.e., *partitioned-stateful operators* [1]), where each distinct key is associated with a separated state object accessed privately by the replica receiving inputs having that key.

2.2 FPGA architectures

An FPGA is an integrated circuit consisting of a grid of configurable hardware blocks. Unlike CPUs and GPUs, which have a fixed hardware structure executing instructions, FPGAs build custom hardware to implement a program. The basic structure of an FPGA is made of look-up tables (LUTs) performing logic operations, flip-flops (FFs) storing register data, wires and I/O pads implementing the interconnects between blocks. To enhance the computational density and efficiency, modern FPGAs combine basic elements with specialized hardware blocks, including: *block random access memory* (BRAM) storing and transferring data between on-chip resources, *digital signal processing* implementing common fixed-point and floating-point arithmetic, and I/O blocks employed to communicate with external devices such as micro-controllers or off-chip memories.

FPGA programming techniques employ *hardware description languages* [7] (HDL), such as Verilog or VHDL, to describe digital circuits, either behaviorally or structurally. However, a recent trend is to use *high-level synthesis* (HLS) languages and tools to raise the level of abstraction and to reduce the design time. Both academia and industry have been involved in releasing HLS tools to develop, verify the functional correctness, debug, and synthesize FPGA hardware starting from a code written in C/C++ or using other programming languages.

2.3 Vitis HLS

Vitis HLS [11] is a tool allowing developers to design and synthesize hardware from an application code written using C/C++ and OpenCL. We describe the main concepts of Vitis HLS that are useful to understand the core choices of our work and the used jargon.

We refer to a *kernel* as a coarse-grained computational unit defined by a top-level function in the source code of a Vitis HLS program. Kernels can take advantage of *dataflow* and *pipelining* optimizations to perform their computations in parallel on the FPGA resources, leading to a significant performance improvement. Dataflow parallelism within the kernel source code can be achieved by adding the `#pragma HLS DATAFLOW` directive. This way, functions called within the top-level kernel function are synthesized as separate units that can run in parallel according to their data dependencies. Communications between functions are performed through *streams*, which are usually synthesized as FIFO queues on the FPGA hardware.

Multiple kernels are typically integrated into larger FPGA designs to create complex applications. Integrating kernels into larger designs involves connecting kernels and interfacing with other components of the FPGA. Kernels can communicate with each other using special streams implementing the AXI4-Stream protocol. Vitis also provides a host runtime system, which is a software library for controlling and interacting with the FPGA. It enables host-to-device (H2D) and device-to-host (D2H) memory copies, and kernel launching from the host code. Vitis kernels have several *memory ports* that can be mapped to different memory resources in the system (e.g., HBM banks) to optimize the bandwidth of kernel-to-memory connectivity in order to read data produced by the host program, as well as to write results that will be consumed by the host program, efficiently.

3 FSPX overview

FSPX supersedes our prior work FSP [15], which was aimed at generating DSP applications targeting Intel FPGAs only. FSPX is instead capable of generating code for both Intel and Xilinx FPGAs, producing the appropriate code for the chosen target. The general aim is to lower the effort in the development of DSP applications that want to exploit FPGA devices to accelerate their processing. To enable the targeting of both platforms, we introduce an abstraction layer within the code generation module to facilitate the production of the right source code for the target

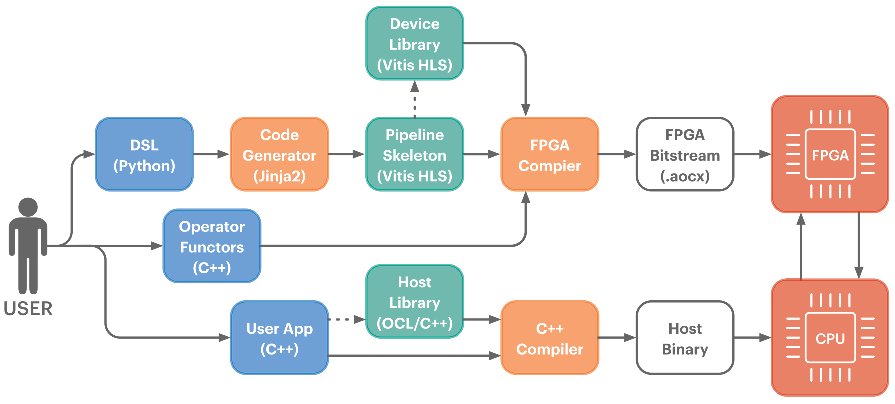


Fig. 3 FSPX workflow to develop a DSP application leveraging Xilinx FPGAs

FPGA device. Furthermore, we implemented from scratch a novel library of device components (shortly called *device library*) that will be combined and used to synthesize data-flow graphs on Xilinx FPGAs, which is the main focus of this paper.

The development of an FSPX application targeting Xilinx FPGAs is fully described by the workflow in Fig. 3. In this perspective, FSPX shares the same programming model of FSP and the same goal of targeting general-purpose stream processing applications, where operators can execute arbitrary code and not built-in functions like in relational stream processing. We achieve this goal by hiding the use of HLS from the application programmer, since the kernel definitions, and the use of HLS streams to perform the data-flow on the device are completely done by the generated code, while the programmer is only involved in defining the sequential code of the operators using plain C/C++ (with some optional code annotations to help the FPGA compiler). In fact, while HLS languages are often considered high-level approaches for FPGA programming, they are still low-level from the perspective of a traditional stream processing programmer, who is usually a data scientist or practitioner without the adequate expertise to cope with HLS programming in an efficient and productive manner.

To develop an application in FSPX, we start by defining the application and its operators using the domain-specific language (DSL) introduced by FSP. Such a DSL has been extended enabling the programmer to specify the class of target FPGA architecture (i.e., Intel or Xilinx), in order to generate the right code implementing the data-flow graph. The DSL is expressed in the Python programming language and allows the high-level definition of the data-flow, i.e., to identify the used operators and how they are interconnected. We currently support the following types of computing operators that will be implemented on the FPGA hardware resources:

- The *Map* operator applies a user-defined function on each input by producing a corresponding output having the same or a different data type
- The *Filter* operator removes from the stream all inputs for which a user-defined predicate is evaluated to *false*

- The *FlatMap* operator applies a user-defined function on each input and produces zero or more outputs. Outputs are of the same data type but can differ from the input data type.

In addition to computing operators, FSPX also provides two built-in memory operators that interact with the host program:

- The *MemoryReader* operator produces an input stream feeding computing operators on the FPGA by continuously forwarding data arriving from the host program;
- The *MemoryWriter* operator collects the stream of results from computing operators and makes them continuously available to the host program.

Each FSPX operator can be parallelized by running more functionally equivalent replicas that will work on different inputs in parallel, as introduced in §2. From the DSL perspective, the programmer specifies the degree of parallelism of each operator (i.e., number of replicas), and the *dispatching policy* to route outputs produced by the operator to the next destination replicas. The dispatching policies provided by FSPX are the following:

- *Round-Robin* (`RR`): the operator replica delivers its outputs to the replicas of the next operator in a circular manner, thus balancing the number of inputs assigned to each replica of the next operator. This policy makes use of so-called *Blocking streams*, meaning that the sending operator replica is blocked until the chosen receiver operator replica can accept a new input
- *Load-Balancing* (`LB`): the operator replica delivers outputs in a circular manner as with the `RR` policy. This policy is *non-blocking*, so if the receiving operator replica cannot receive a new input, the sender tries to deliver it to another destination operator replica in the next clock cycle
- *Key-By* (`KB`): inputs having the same *key* (the key value can be extracted from the input through a user-defined function) are always sent to the same replica of the next operator
- *Broadcast* (`BR`): the sending operator replica sends a copy of each output to all the replicas of the destination operator.

Analogously, FSPX allows the programmer to choose the *gathering policy*, which controls how an operator replica receives data from the replicas of the previous operator. `RR` and `LB` policies can be used for this purpose, by polling data in a blocking or non-blocking manner, respectively.

After defining the structure of the pipeline and specifying for each operator the number of replicas and the chosen dispatching and gathering policies, FSPX generates a C++ code for the device (called *pipeline skeleton* in Fig. 3) using Jinja2 [17]. The device implementation (partially generated and partially using the device library functions of FSPX) contains several pieces of the pipeline logic that are efficiently implemented for the device without any programming effort by the FSPX user. However, since FSPX is aimed at supporting general-purpose streaming applications with operators doing arbitrary computations expressed in imperative code, the produced pipeline skeleton is not ready to run, but it must be completed by the programmer with the specification of the user-defined functors (one per operator) that must respect a given signature based on the operator type (denoted as *operator functors* in Fig. 3).

In the next part, we describe more in detail the DSL and the code generator of FSPX.

3.1 Domain-specific language

In this section, we introduce the DSL provided to the user to design a DSP application in FSPX. To declare an operator, the developer instantiates the `FOperator` class. Listing 1 shows the declaration of a map operator that is replicated two times, gathering inputs from the previous operator with a `LB` policy, and dispatching results of datatype `output_t` with the `KB` policy.

```
map = FOperator('map',                # Name
                2,                    # Parallelism degree
                FOperatorKind.MAP,    # Base operator kind
                FGatherPolicy.LB,     # Gather policy
                FDispatchPolicy.KB,   # Dispatch policy
                'output_t')           # Output data type
```

Listing 1: Example of declaring a Map operator, with two replicas, gathering inputs with LB gather policy, and dispathching output t results with KB policy

A stateless operator, excluding memory operators, can become stateful by adding a buffer to store its state. Such a buffer is a memory space that the user decides to make available to the operator, where it can store historical information about the received inputs, and can be used to compute results in a generic stateful manner. Listing 2 shows how to add a *private* and a *local* buffer to an operator. Private buffers are synthesized by the compiler with registers and should be used to store variables or arrays with few elements. A local buffer is instead implemented in dedicated on-chip memory blocks and can be useful to store a small state. If the operator is replicated, each replica has its own buffer as specified for the operator.

```
map.add_private_buffer('char', 'array', size=16)
map.add_local_buffer('float', 'matrix', size=(32, 128))
```

Listing 2: Functions to add private and local buffers to an operator

The application is eventually declared by creating a `FApplication` object as shown in the first line of Listing 3. The programmer can specify the destination folder where to generate the source code (i.e., a directory named `myApp`), the input datatype of the pipeline (e.g., in the example the MemoryReader operator will accept inputs of `input_t` type), and the target platform on which the application should run (e.g., a Xilinx FPGA). The developer must add the two memory operators and the application operators in the right order. By calling the `generate_code()`

Table 1 Lines of code of FSPX to generate code based on the target FPGA

Target	Jinja2	Host	Device	Other	Total
Intel	1168	1409	-	-	2577
Xilinx	168	1160	1279	84	2691

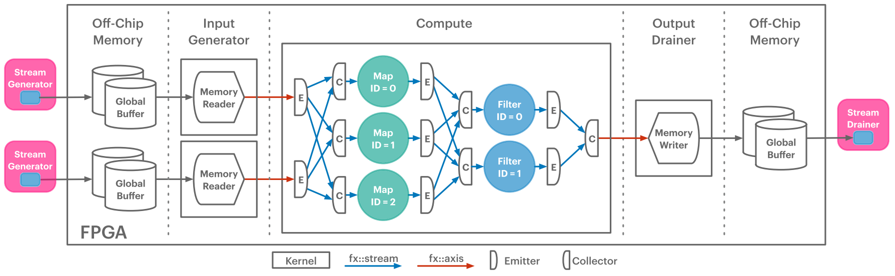


Fig. 4 Implementation of a DSP application in FSPX targeting Xilinx FPGAs

method, FSPX generates the device code, which mainly consists of the kernel instantiating the specific pipeline of computing operators, which in turn makes use of a set of components that we implemented in the device library.

```
app = FApplication('myApp', 'input_t', FTarget.XILINX)
app.add(memory_reader)
app.add(map)
...
app.add(memory_writer)
app.generate_code()
```

Listing 3: Example of declaring and generating code for an application with FSPX

3.2 Code generation

The Python DSL is used by the programmer to express a pipeline with an arbitrary number of operators (each having an arbitrary number of replicas) connected using a set of available dispatching and gathering policies. By running the Python script with the DSL program, some Jinja2 template files will be instantiated by passing as parameters the values of a set of Python variables used in the DSL. A Jinja2 template contains variables and expressions that will be replaced with the provided values when the template is rendered.

The generated code consists of a set of kernels as represented in Fig. 4 by referring to the pipeline example in Fig. 2. It contains the declaration and definition of the MemoryWriter and MemoryReader kernels, as well as of the so-called *Compute* kernel encapsulating the computing operators. The code of such kernels instantiates and refers to a set of C++ template functions (with Vitis annotations) that belong to the *device library* of FSPX, where each function corresponds to specific components of the device part for which the code has been already implemented (e.g., dispatching components or gathering components adopting the different available policies). Analogously, the host program will use the API provided by the header files belonging to the *host library* of FSPX, to interact with the MemoryReader and MemoryWriter kernels properly.

Such an approach minimizes the size of the generated code, keeping it small and moving most of the implementation burden to the device and host library files. This approach is quite different compared with what was done for the Intel targets [15], since in such a version most of the device code was generated with Jinja2, including all kernels and the whole implementation of the data-flow components. Table 1 quantifies these differences.

When targeting Xilinx FPGAs, Jinja2 templates occupy only 168 lines of code, while more than a thousand lines were necessary for the templates in the case of Intel FPGAs. The most remarkable difference is in the device library, actually absent for Intel targets (i.e., the whole device implementation is generated through Jinja2 every time the DSL program is run).

4 Implementation

In this section, we focus on the device code for the Xilinx FPGAs. This includes the code generated through Jinja2 from the DSL, and the device library containing several template functions instantiating the various components for an efficient FPGA implementation of the pipeline.

The device code implements the physical pipeline of operators with replication. It is composed of three main conceptual parts: *i)* the input generator part, *ii)* the compute part, and *iii)* the output drainer part. Figure 4 depicts a logical representation of the code of the physical pipeline in Fig. 2.

The input generator part is essentially composed of one or more instances of the MemoryReader operator, each reading data from a specified *global buffer* (i.e., a memory buffer allocated on the FPGA off-chip memory) and generating the stream of individual inputs. The compute part contains the computing operators. The output drainer part is composed of one or more instances of the MemoryWriter operator, each receiving data from the previous operators and storing them in specified global buffers.

4.1 Stream abstractions

Vitis HLS provides the `hls::stream<Type, Depth>` template class, which represents a unidirectional data stream facilitating efficient communication between hardware modules. The hardware implementation varies depending on the template parameter `Type`. It allows the use of any C++ native data type, Vitis HLS arbitrary precision type (e.g., `ap_int<>`), or user-defined `struct` types. When any of these types are used, the compiler generates a First-In-First-Out (FIFO) queue of size `Depth`. This hardware implementation can be used to transmit data between modules within the same kernel. However, when working with streams between independent kernels, it is essential to employ the AXI4-Stream protocol. In such a case, the `hls::axis<>` interface is used to wrap the data type, ensuring that the compiler implements streams that adhere to the protocol.

To enhance code simplicity and prevent mistakes during code generation, we have designed a class for each type of stream, both having the same member functions. As a result, the classes and functions of the device library can accept both stream types as template arguments and use them interchangeably. The `fx::stream<T, D>` class wraps the type of a stream used within the same kernel, while the `fx::axis<T, D>` wraps a stream using the AXI4-Stream protocol. This design enables us to achieve parametric polymorphism at compile time. Hereinafter, we refer to a generic stream implementing the AXI4 protocol as *axis*.

Both streams and axis types internally consist of two independent sub-streams: the `data` sub-stream, used to exchange data items, and the `eos` stream, which carries the end-of-stream (EoS) signal. These internal sub-streams can be managed independently by calling the associated functions to write, read, check for emptiness, and check for fullness. It is worth noting that the `write` member function (of both `fx::stream<T, D>` and `fx::axis<T, D>`) sends out not only the data item in the `data` sub-stream but also a `0` value in the `eos` sub-stream. The EoS signal will be transmitted by a specific `write_eos` member function that writes `1` in the `eos` sub-stream.

4.2 Memory operators

During the execution, data exchange between the host program and the FPGA pipeline occurs through the off-chip global memory of the device. The host program copies data on global buffers that will be read by MemoryReaders and delivered to the FPGA pipeline. Results computed by the FPGA pipeline will be stored by MemoryWriters in global buffers that are copied into the host memory by the host program. The MemoryReader operator implements the reading of new inputs from previously allocated global buffers and forwards them to the `compute` kernel through proper axis streams.

In our design, global buffers used by a single MemoryReader reside in a single HBM2 memory bank, which is accessed by the MemoryReader operator through a single memory port. To exploit more HBM2 banks if available, MemoryReader operators can be replicated, each having a dedicated host thread pushing new data onto global buffers allocated in a given HBM2 bank, and associated with a given MemoryReader replica.

A simplified implementation of the MemoryReader operator is shown in Listing 4. The outermost `for` loop reads `count` lines of size `W` bits from the global buffer `in`. In the innermost `for` loop, each line is then unpacked into individual data items of type `T` and sent to the `compute` kernel. Finally, the operator can propagate the end-of-stream (EoS) signal if requested. This implementation, with a proper choice of the `W` parameter, allows the FPGA hardware implementing the

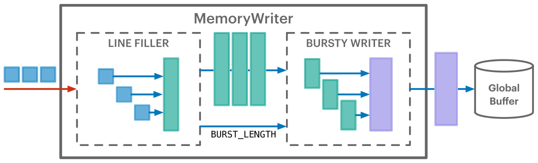


Fig. 5 Graphical representation of the MemoryWriter implementation

MemoryReader to better exploit the memory bandwidth, by reading W bits for each memory request.

```
template <int W, typename T>
void mr(ap_uint<W>* in, int n, int eos, fx::axis<T>& out)
{
    constexpr int B = sizeof(T) * 8;
    constexpr int ITEMS = W / B;
    for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II = ITEMS
        ap_uint<W> line = in[i];
        for (int j = 0; j < ITEMS; ++j) {
        #pragma HLS PIPELINE II = 1
            ap_uint<B> v = line.range(B * (j + 1) - 1, B * j);
            out.write(*reinterpret_cast<T>(&v));
        }
    }
    if (eos) out.write_eos();
}
```

Listing 4: MemoryReader implementation (W: bus width, T: output datatype)

The MemoryWriter operator is structured into two distinct stages working in a pipeline manner as depicted in Fig, 5, to ensure efficient bursts accessing the off-chip global memory. The first stage called *Line Filler* (LF) collects the data items originating from the preceding operator and packs them into a line of size `W` bits. When a new line is formed, this stage forwards it to the subsequent stage called *Bursty Writer* (BW) through a stream `lines` with depth $2 \cdot K$.

When a given number $K > 0$ of lines have been collected and transmitted by LF and buffered in the stream `lines`, LF writes the number of lines K in a second stream `bursts` transmitted to the same stage BW. In the second stage, BW is continuously waiting for new data from the stream `bursts`. As soon as the constant K is read from that stream, BW starts reading all the buffered K lines from the stream `lines` and writes each one at a proper offset of a global buffer `out` in the off-chip memory, which has been properly created by the host program before running the MemoryWriter kernel. Since the number of lines to be written is known before starting the first write, the generated FPGA hardware can aggregate write requests in a single burst request to the memory, improving the memory bandwidth utilization. Furthermore, at the steady state, the two stages work in parallel: the first one is filling lines of the next burst and buffering them into `lines`, while the second stage is writing lines of the previous burst in memory.

4.3 Compute kernel

All the computing operators of the pipeline are instantiated inside the `compute` kernel. We choose to implement it as a free-running kernel, i.e., a kernel starting on the FPGA automatically when the bitstream is loaded, and thus it does not need any interaction with the host program. In addition, it has no memory input or output ports and thus interacts with other kernels through axis streams only. The general structure of the kernel is shown in Listing 5 by referring to the example represented in Fig. 4.

This kernel is produced by the code generator. It makes use of several functions whose implementation is part of the device library. Each function is the implementation of a specific component of the FSPX runtime system, and such components are interconnected through `fx::stream` instances forming the data-flow graph implementation as sketched in Fig. 4. During the process of compilation of the kernel, through the `Vitis HLS` compiler, such components will be properly synthesized as separate hardware units. To do that, we use the `#pragma HLS DATAFLOW` directive to instruct the compiler in synthesizing each function as an independent and parallel hardware unit.

```
 1 void compute(fx::axis<input_t> in[2], fx::axis<output_t> out[1])
 2 {
 3     #pragma HLS INTERFACE ap_ctrl_none port=return
 4     #pragma HLS DATAFLOW
 5 
 6     fx::stream<input_t> e2m[2][3];
 7     fx::stream<input_t> m2f[3][2];
 8     fx::stream<output_t> f2c[2][1];
 9 
10     fx::Emitter<fx::Policy_t::KB, 2, 3>(in, e2m, keyby_fun);
11     fx::Replicate<fx::Operator_t::Map, 3, ...>(e2m, m2f);
12     fx::Replicate<fx::Operator_t::Filter, 2, ...>(m2f, f2c);
13     fx::Collector<fx::Policy_t::LB, 2, 1>(f2c, out);
14 }
```

Listing 5: Compute kernel generated by FSPX

The `compute` kernel accepts input and output axis streams as parameters, allowing the interconnection with MemoryReaders producing data streams by reading from global buffers, and with MemoryWriters receiving outputs from the kernel and writing them in global buffers.

An emitter is a component having a single input stream and more output streams, and implementing a specific dispatching policy. In the data-flow graph, we use some emitters to interface each MemoryReader with the replicas of the first operator in the pipeline. Each MemoryReader communicates with the replicas of the first operator through a dedicated emitter. This is done at line 10, where the `fx::Emitter` function is instantiated with template arguments representing the policy type (in this example we use the `KB` policy), and the number of MemoryReaders and replicas of the first operator. The function generates a number of components implementing the given dispatching policy that are equal to the number of MemoryReaders, as specified by the second template argument. The `fx::Emitter` takes in input the two axis streams from the MemoryReaders (in), and the streams to be used to connect with the replicas of the first operator. Since the dispatch policy chosen by the user for the MemoryReader is KB, the user is requested to provide the keyby_fun lambda function that specifies how to extract the key attribute from each input_t data item. Listing 6 shows an example of this lambda function.

```
auto keyby_fun = [](const input_t& data) {
    return data.id;
};
```

Listing 6: Lambda function to extract the key from an input t data item

Other more complex dispatching policies (like LB) are supported. This policy uses non-blocking streams and, before sending a new data item to an operator replica via the associated stream, the routing component checks if the stream is full. If it is full, it tries to dispatch the current data item to the next replica operator in the next clock cycle. This dispatch policy should be used when the subsequent operator replicas may have different II. This can happen, for example, when the subsequent operator is a FlatMap, which decides the number of dispatched data items at runtime. In this case, each FlatMap replica might produce a data item at each clock cycle, so it can have a different output rate during the application execution.

The operators are instantiated and replicated using the fx::Replicate library function. This function, described in detail afterward, is responsible for replicating the operator and the components to connect each replica to the subsequent and preceding ones through streams. Lastly, the fx::Collector function connects the replicas of the final operator in the pipeline to the MemoryWriters, by implementing the gather policy chosen. In the example, the policy is LB.

This implementation schema of the data-flow has been designed to reduce the number of axis streams that would connect MemoryReaders and MemoryWriters to the compute kernel. This is done by having dedicated emitters and collectors before the first operator and after the last one, instead of connecting each MemoryReader directly with the operator replicas with dedicated axis streams (symmetrically for the MemoryWriter). Consequently, we save the resources that would be required for the implementation of the AXI4 protocol for a large number of axis streams.

4.3.1 Replication

The replication of each computing operator (we assume here a map and a filter as an example) is achieved through the use of the template function fx::Replicate<>, which is shown in Listing 7.

```
template <
   fx::Operator_t OP_T, int PAR, typename FUN_T,
   fx::Policy_t IPOLICY, fx::Policy_t OPOLICY, int N, int K,
   typename ISTR, typename OSTR, typename KEY_EXTRACTOR_T
>
void Replicate(ISTR istrms[N][PAR], OSTR ostrms[PAR][K],
               KEY_EXTRACTOR_T&& keyby_fun)
{
    #pragma HLS DATAFLOW
    for (int i = 0; i < PAR; ++i) {
    #pragma HLS UNROLL
        fx::Replica<OP_T, PAR, FUN_T, IPOLICY, OPOLICY, N, K>(
            istrms, ostrms[i], i,
            std::forward<KEY_EXTRACTOR_T>(keyby_fun)
        );
    }
}
```

Listing 7: Implementation of fx::Replicate<>

This function has a set of template parameters describing the operator type `OP_T`, the number of replicas `PAR`, the type of the functional logic to call on each input `FUN_T`, the gathering and dispatching policies `IPOLICY` and `OPOLICY`, the number of emitters `N` of the previous stage of the pipeline, the number of collectors `K` of the next stage of the pipeline, the type of the input streams `ISTR` and of the output streams `OSTR`, and the type `KEY_EXTRACTOR_T` of the logic used to extract the key attribute from the inputs. The body of the function is composed of a fully unrolled `for` loop. Within this loop, the template function `fx::Replica`<> is invoked, leading to the creation of `PAR` replicas. The `DATAFLOW` directive declared at the start of the body function is essential to instruct the synthesis tool to generate replicas as independent and parallel hardware units.

The template function `fx::Replica`<> in Listing 8 shows the internal structure of an operator replica, which has similar template parameters of the calling `fx::Replicate`<> function. The structure is a three-staged pipeline made of a collector, the operator replica, and an emitter, as shown in Fig. 4.

```
1  template <
2     fx::Operator_t OP_T, int PAR, typename FUN_T,
3     fx::Policy_t IPOLICY, fx::Policy_t OPOLICY, int N, int K,
4     typename ISTR, typename OSTR, typename KEY_EXTRACTOR_T
5  >
6  void Replica(ISTR istrms[N][PAR], OSTR ostrms[K], int i,
7              KEY_EXTRACTOR_T&& keyby_fun)
8  {
9      #pragma HLS DATAFLOW
10     fx::stream<typename ISTR::data_t> c2o;
11     fx::stream<typename OSTR::data_t> o2e;
12
13     fx::Collector<IPOLICY, N, 1>(istrms[i], &c2o);
14
15     if constexpr (OP_T == MAP) {
16         fx::Map<FUN_T>(c2o, o2e);
17     } else if constexpr (OP_T == FILTER) {
18         fx::Filter<FUN_T>(c2o, o2e);
19     } else if constexpr (OP_T == FLATMAP) {
20         fx::FlatMap<FUN_T>(c2o, o2e);
21     }
22
23     fx::Emitter<OPOLICY, 1, K>(&o2e, ostrms, keyby_fun);
24 }
```

Listing 8: Implementation of fx::Replica<>

The collector implements the gathering policy associated with the operator as specified by the template parameter IPOLICY. At each clock cycle, it selects the input stream to read from and pulls a new data item from it. Then, it forwards the data item to the operator through the c2o stream (collector-to-operator). At lines 15–21, a chain of compile-time if statements is employed to conditionally instantiate the operator based on the template parameter OP_T. Finally, the emitter receives the resulting data items of the operator computation from the o2e stream (operator-to-emitter). It is responsible for routing processed data items to the next operator replica implementing the dispatch policy associated with the operator as specified by the template parameter OPOLICY.

4.3.2 Operator structure

Listing 9 shows the implementation of the Map operator (the implementation of the other supported operators is omitted for the sake of space).

At line 6, the functor FUN_T is instantiated. This functor must respect the signature void(input_t, output_t &) based on the operator type. It must be provided by the user and define the business logic code of the operator. For this reason, FSPX generates a stub with the right signature in the generated code, and the programmer is expected to fill the business logic code before compiling the code with the Vitis compiler. Within the while loop, the functor is applied on each input data item read from the istr stream, producing the result that is stored in the out variable. Then, the result is written to the ostr stream. When the EoS signal arrives, the loop stops and the EoS signal is forwarded to all the replicas of the next operator.

```
template <typename FUN_T, typename ISTR, typename OSTR>
void Map(ISTR & istr, OSTR & ostr) {
    using IN_T  = typename ISTR::data_t;
    using OUT_T = typename OSTR::data_t;

    FUN_T func;
    bool last = istr.read_eos();
    while (!last) {
    #pragma HLS PIPELINE II = 1
        IN_T in = istr.read();
        last = istr.read_eos();
        OUT_T out;
        func(in, out);
        ostr.write(out);
    }
    ostr.write_eos();
}
```

Listing 9: Implementation of the Map operator

4.4 Host library

The potential performance gain of using FPGAs might be nullified by the overhead of H2D and D2H data transfers. Therefore, FSPX provides a host library to mitigate the communication overhead by using data batching and K-Buffering optimizations. The host library includes the *StreamGenerator* class abstracting the low-level actions to push new inputs to a MemoryReader operator, and the *StreamDrainer* class to collect results from a MemoryWriter operator in an easy manner by the application programmer. More details will be given in this section.

4.4.1 Data batching

Transferring inputs one by one to the FPGA results in low communication bandwidth, which may drastically reduce the application throughput. Therefore, FSPX adopts a strategy that groups data items in batches, and transfers them to/from the FPGA as a whole. The batch size is a user-configurable parameter for both StreamGenerator and StreamDrainer interfaces. Small batches (e.g., of a few hundred bytes) can lower computational latency but drastically reduce throughput because of the communication overheads. Large-sized batches (e.g., order of kilobytes or megabytes) often increase throughput at the expense of latency, so a balance should be found based on the performance requirements dictated by the application use case. To help users achieve optimal throughput levels with our framework, the host library includes the `fx::batch_size_hint<T>()` function. This function is designed to return a batch size that is likely to maximize transfer bandwidth. This simple heuristic is based on the observation that batches having sizes of at least 2MB are often sufficiently large to amortize H2D and D2H data transferring overheads in most of the experiments that we conducted with FSPX.

In terms of kernels, by referring to Fig. 4, we observe that while the `compute` kernel is a free-running kernel always running on the FPGA, MemoryReader and MemoryWriter kernels have a life-cycle strictly related to a given batch. Every time a host thread using the StreamGenerator interface pushes data forming a new batch, the batch is copied from a host buffer to a global buffer on the device. Then, a MemoryReader kernel is launched to read that buffer and to start the streaming to the pipeline in the `compute` kernel. Analogously, once a MemoryWriter kernel is launched, it fills a batch in global memory with data items received from the last operator in the pipeline. Once the buffer is filled (or if the EoS signal is present), the MemoryWriter terminates and a host thread transfers the results from the global buffer to a host buffer. After that, a new MemoryWriter can be launched assigning the same or a different global buffer.

4.4.2 K-Buffering

A naive solution serializes the H2D and D2H transfers with MemoryReader and MemoryWriter executions. Therefore, there is a gap in time between one kernel execution and the next one. We call this time interval *kernel downtime*. To minimize the kernel downtime, FSPX employs a generalization of the double-buffering optimization that we call the K-Buffering technique. The idea is to overlap the data transfers of the next up to $K - 1$ batches while the currently running kernel is using the previous one. The StreamGenerator and the StreamDrainer adopt this technique by allocating $K > 1$ buffers in global memory before the application starts. During the application lifetime, buffers that have been consumed are recycled to avoid repeated allocations and deallocations.

The implementation requires special care. The StreamGenerator API directly copies a batch from a host buffer to an available buffer in global memory and enqueues the execution of the associated MemoryReader kernels. Once a new batch is ready to be transferred, the StreamGenerator can copy it into a different available buffer and enqueues a new kernel execution. When all global buffers are in use, the StreamGenerator awaits the completion of the oldest execution, ensuring that at least a global buffer becomes available for overwriting. This implementation ensures that data transfer remains continuous and minimizes the kernel downtime.

The StreamDrainer API, to ensure the overlapping of data transfers with the execution of MemoryWriter kernels, allocates K different buffers in global memory to be filled with output results from the pipeline, and enqueues the same number of executions of the MemoryWriter kernels (each working on a different global buffer). This way, once a MemoryWriter execution is complete, the StreamDrainer can transfer the results back from the global buffer to a host buffer of the application and enqueues a new execution of the MemoryWriter working on the same buffer in global memory. At the end of the computation, the EoS signal arrives and the StreamDrainer waits for the $K - 1$ MemoryWriter executions to terminate.

The number of buffers K is a user-configurable parameter for both StreamGenerator and StreamDrainer interfaces. When K is too high, the application performance can degrade since there are too many transfers in parallel, thus resulting in lowered transfer efficiency. On the other hand, when K is too low, it can lead to low efficiency since it does not allow transfers to be properly overlapped with kernel execution. To help the user find a good value of K, the host library offers the function `fx::kbuffers_hint<size_t>()` that calculates K as a function of the batch size. The idea of this simple heuristic solution is to adopt at least two buffers (to enable double buffering) if the size of the batch is more than 2MiB. Otherwise, the function returns the number of buffers needed to achieve an overall size of at least 2MiB.

5 Evaluation

We evaluate FSPX on the Xilinx Alveo U50 Data Center Acceleration Card. The FPGA includes 872K LUTs, 1743K registers, 5952 DSP slices, 1344 BRAM of 36Kb each, and accesses 8GiB off-chip HBM2 memory. The host machine is equipped with two CPUs Intel Xeon E5-2650 V3 and 128GiB of DDR4 at 2133MHz Quad-channel. Each CPU has 10 cores (20 hardware threads) sharing an Intel Smart Cache L3 of 25MiB. Each core has a clock rate of 2.3GHz (3.0GHz Turbo Frequency) and an L2 cache of 256KiB. The FPGA is connected to the host machine through the PCIe Gen3x16. Bitstreams are generated using the Vitis `v++ v2023.1` compiler within the Vitis Core Development Kit 2023.1. The host program is compiled with `g++ 11.4.0` with the `-O3` optimization flag. FSPX applications use the OpenCL standard implemented by Xilinx with the Xilinx Runtime Library (XRT).

We also provide a comparison with a scale-out open-source SPE and a scale-up research SPE. More specifically, we compare with Flink [2] (version 1.9.0) compiled with `OpenJDK 11`, which is a widely popular SPE to process data streams at scale, and WindFlow [13], a research project for scale-up multicore-based architectures written in `C++17`. Since Flink and WindFlow exploit CPU cores only, to provide a fairer comparison we run them on a more modern machine equipped with an Intel i7-12700K and 16GiB DDR4 at 3600MHz. The CPU has 8 Performance cores with Hyper-Threading running at 4.8GHz, and 4 Efficiency cores running at 3.7GHz, for a total of 20 threads. The cores share an Intel Smart Cache L3 of 25MiB. So doing, we compare the performance achieved by FSPX using the U50 board (released in 2019) against the performance obtained by WindFlow/Flink running on a modern Intel CPU (released in 2021).

5.1 Benchmark applications

We have chosen two applications from DSPBench [5], which is an established benchmark repository for DSP applications. This choice is the result of the evaluation of the available applications by taking into account two aspects: the first is related to the fact that FSPX is currently able to express arbitrarily long pipelines of operators (each with its parallelism degree), while some applications in DSPBench are more complex DAGs; second, some applications use in the business logic code of their operators some external libraries (e.g., for geo-localization) whose porting on FPGA is not straightforward and, in general, is outside the evaluation of FSPX and so the scope of this paper.

SpikeDetection (**SD**[2]) analyzes a stream of sensor readings and checks for spikes. We implement it on the FPGA as a logical pipeline of four operators: a MemoryReader, a stateful Map (AverageCalculator) emitting a moving average value computed over a counting window, a Filter (SpikeDetector) evaluating if the current sensor reading is a spike, and a MemoryWriter. The Map operator implements a key-partitioned sliding window. To efficiently compute the moving average, the business logic code of the operator uses the shift register pattern [22]. The Filter implements a simple Boolean predicate, so only records respecting the predicate are delivered to the MemoryWriter.

FraudDetection (**FD**[3]) processes a stream of credit card transactions to detect frauds. We implement it on the FPGA as a logical pipeline of three operators: a MemoryReader, a stateful FlatMap (Predictor) applying a Markov model to calculate the probability of fraud, which emits the current transaction if that probability is higher than a given threshold value, and a MemoryWriter. The FlatMap keeps a subset of the last transactions per credit card identifier and computes the probability of being a fraud using a static matrix of values of the Markov model, which is stored in the on-chip memory blocks.

We also consider a modified version of **FD** that incorporates two stateful operators in the `compute` kernel. The idea of this version (called FraudFreqDetection, shortly **FFD**) is to add a stateful Map (FrequencyEvaluator) after the predictor operator. Timestamps are initially assigned by the host program when data items are generated. This operator updates the count of frauds for each type of transaction.[4]

In the experimental evaluation, we consider two performance metrics. The first is related to the speed of the FPGA pipeline, i.e., its *throughput*, measured as the total number of inputs processed per second. We count the number of inputs that the MemoryReader operators can transmit to the rest of the pipeline (all such inputs are completely processed before termination) and we divide this count by the running time of the whole program (i.e., host program and the FPGA pipeline). The second metric is the so-called *latency*, computed as the average time between the time instant when an output result is received by the host program after the FPGA computation, and the timestamp attribute incorporated in the output itself. Each timestamp is then propagated in the outputs up to the host program after the FPGA processing. In the case of operators using more inputs to compute the same output, the timestamp of the last input contributing to the output is propagated for the latency evaluation.

5.2 Resource utilization and performance

Tab. 2 summarizes the resource utilization and the frequency of the bitstreams implementing SD and FD applications with parallelism from one to six. The

[2] SpikeDetection source code: https://github.com/blackwut/FSPX/Apps/SpikeDetection

[3] FraudDetection source code: https://github.com/blackwut/FSPX/Apps/FraudDetection

[4] FraudFreqDetection source code: https://github.com/blackwut/FSPX/Apps/FraudFreqDetection

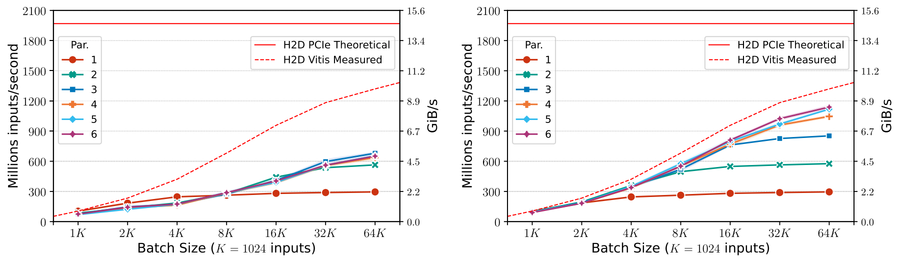
Table 2 Resources utilization and maximum frequency (FMax) of SD and FD bitstreams with different parallelism degrees

App.	Par.	LUT	REG	BRAM	DSP	II	FMax MHz	Theoretical throughput M inputs/sec	Measured throughput M inputs/sec
SD	1	16,715	21,673	74	40	1	300	300	299.40
	2	33,801	41,921	148	80	1	300	600	598.80
	3	53,910	65,695	222	132	1	300	900	898.21
	4	70,181	85,111	296	160	1	300	1200	1197.61
	5	93,977	111,433	370	220	1	300	1500	1497.01
	6	116,288	135,348	444	264	1	296	1776	1769.41
FD	1	23,387	37,275	274	155	1	265	265	254.99
	2	47,689	73,598	348	310	1	300	600	509.99
	3	74,403	114,886	447	477	1	293	879	764.98
	4	96,462	151,012	496	620	1	300	1200	1019.97
	5	126,927	193,172	620	795	1	300	1500	1274.96
	6	153,963	232,902	744	954	1	290	1740	1529.95

generated code and the device library components (e.g., emitters and collectors, MemoryReaders and MemoryWriters), as well as the specific implementation of the business logic functions within the operator replicas, allow obtaining an *Initiation Interval* (II) of one, meaning that every operator replica can process one new input per clock cycle. As a result, the theoretical throughput provided by the bitstream can be predicted as the product of the parallelism of the operators (we suppose the same parallelism degree for each operator) and the operating frequency FMax of the bitstream as obtained by the Vitis compiler. The table also reports the *measured* throughput, which corresponds to the actual throughput that we measured by running properly modified bitstreams where no H2D and D2H interactions with the host program are needed. Indeed, such bitstreams are capable of generating randomly input data without reading from the global memory, as well as results are not written in global buffers. These throughput results confirm that the FPGA implementation of the data-flow (i.e., operators interconnection, dispatching, and gathering policies) behaves as expected. In terms of utilized FPGA resources, the table shows that their utilization increases with parallelism, mainly because of the replication of the computing operators that require more hardware resources on the boards. The BRAM increase is mainly due to the replication of the MemoryReader and the MemoryWriter operators, which consume 57.5 and 16.5 BRAM per instance.

5.3 Throughput evaluation

We evaluate the throughput obtained by the applications described in 5.1 with FSPX in the realistic setup where a host program continuously feeds the FPGA pipeline with new batches to compute, and results produced by the pipelines are continuously



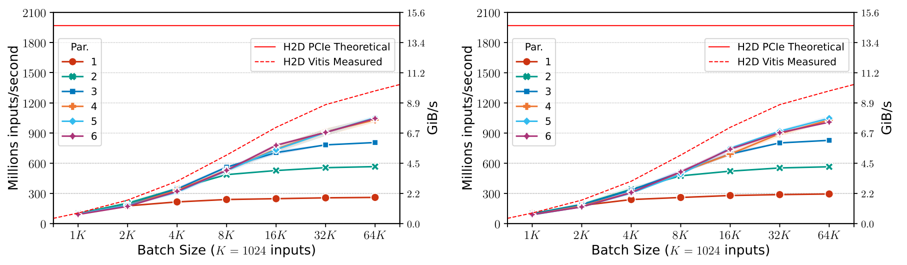
(a) SpikeDetection (0.025) (b) SpikeDetection (0.25)

Fig. 6 Throughput of SD with 0.025 (**a**) and 0.25 (**b**) threshold configurations by varying the parallelism and the batch size

read by the host program. In the throughput calculation, to measure the whole runtime of the program, we exclude the setup and the release of the OpenCL resources at the beginning and the end of the execution. We report in the plots, with a dotted red line, the maximum PCIe bandwidth measured with a bandwidth profiling benchmark provided by Vitis to test the data transfer rate (H2D) to the FPGA device with the OpenCL APIs with a specific buffer size. In this evaluation, for both SD and FD applications, the input data size is 8 bytes and the output data size is 16 bytes.

SD is a fine-grained application and its output rate is heavily dependent on the threshold used by the Filter operator. Therefore, we execute this application with two threshold values: 0.25 and 0.025. Figure 6a shows the results with the threshold value set to 0.025. The plot reports the throughput in terms of inputs per second and the corresponding in GiB/s. The achieved throughput is nearly optimal with parallelism 1 and 2 (i.e., it approximately matches the theoretical throughput provided by the FPGA pipeline according to the bitstream FMax), while for the other configurations of parallelism, it slightly grows but with clear sub-ideal performances. The main reason is that, with this chosen threshold value, the Filter operator drops more than 95% of the processed items with a single source, and an average of 34% with multiple sources, with a minimum of $15-25\%$ with large batches. By considering that the output data size is 16 bytes (so larger than the input data size), and since the dropping rate of the Filter is quite low with large batches with such a threshold value, the maximum throughput is limited by the StreamDrainer in the host program. We reach approximately 8.75 GiB/s of the D2H bandwidth, which corresponds to an exploitation of the H2D bandwidth (as reported in the figure) of 4.69 GiB/s. By setting the threshold to 0.25 (Fig. 6b), the Filter removes more than 99% items from the stream. In this case, with parallelism up to 4, we are very close to the trend of the H2D data rate measured by the Vitis profiling tool with the same batch sizes.

Figure 7a shows the achieved throughput by FD by varying the parallelism degree of the operators and the input batch size. With parallelism 1 and 2, the application achieves almost the theoretical performance reported in Table 2. Starting from parallelism equal to 4, the application reaches its peak performance of 1,050 millions of inputs per second.



(a) FraudDetection (b) FraudFreqDetection

Fig. 7 (**a**) Throughput of FD by varying the parallelism and the batch size. (**b**) Throughput of FFD by varying the parallelism and the batch size

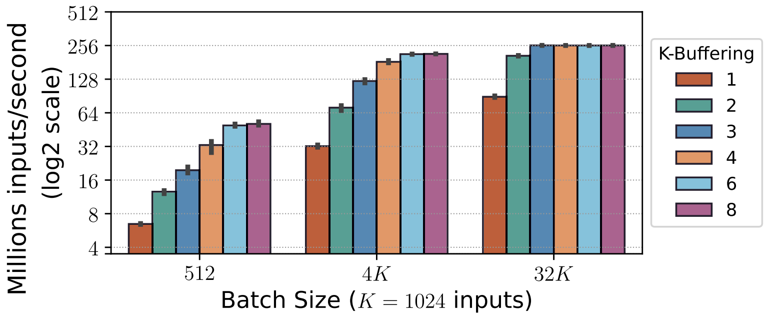


Fig. 8 Impact of the K-Buffering technique on FD using different batch sizes

Although the FFD application is a modified version of FD with two stateful operators in the `compute` kernel instead of one like in FD, its performance remains unaffected. Indeed, Fig. 7b shows the throughput trend of FFD and it is clear that mirrors the one obtained by FD.

Finally, we evaluate the impact of the K-Buffering technique. We select FD with parallelism 1 while we variate the used batch size. Figure 8 reports the result with a fairly small batch size (512 inputs), a medium batch size (4,096 inputs), and a large batch size (32,768 inputs) according to our previous experiments. With a small batch size, increasing the number of buffers up to three leads to a remarkable throughput increment. However, with 4 and 6 buffers, we obtain a significant increment of 5.09× and 7.62×, respectively. With 8 buffers the performance gain is slightly lower than 7.9×. With a medium batch size, we observe a quick increment up to 6 buffers. Using 8 buffers is no more beneficial. With the largest used batch size, the impact of 2 and 3 buffers is still significant, obtaining 2.33× and 2.88× higher throughput. However, with 4 or more buffers there is no additional gain.

In summary, FSPX achieves satisfactory throughput levels, although a certain margin of improvement is still possible to completely saturate the H2D and D2H memory bandwidth provided by the PCIe interconnect of the machine. However,

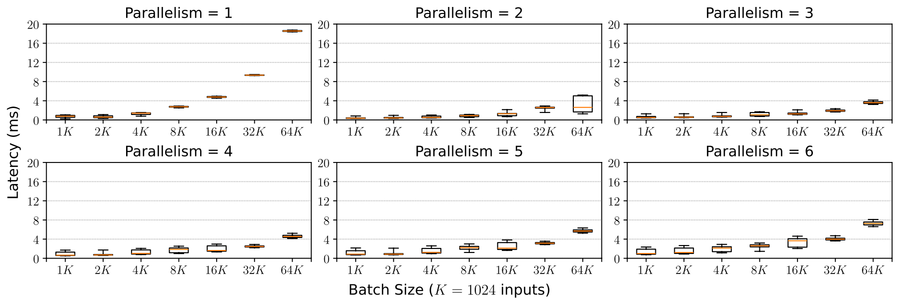


Fig. 9 Latency of SD by varying the parallelism and the batch size

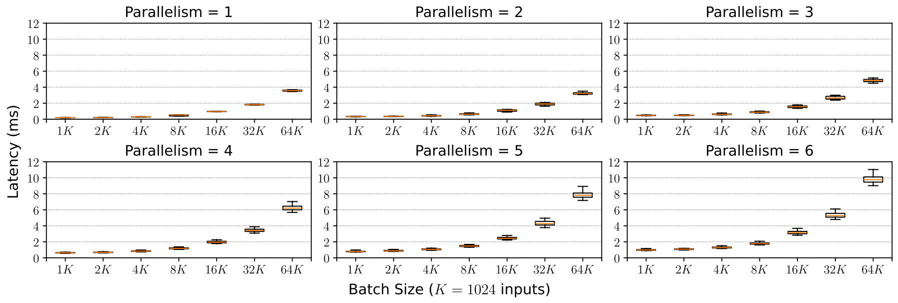


Fig. 10 Latency of FD by varying the parallelism and the batch size

DSP applications exacerbate this issue, since batches need to be continuously produced from external sources and cannot be pre-allocated and pre-filled in memory.

5.4 Latency evaluation

To measure the latency, we modify the bitstreams of the two applications to incorporate a timestamp (4 bytes) value in each input and output value. We collected the statistics by taking 1,024 samples per second and we reported the 5th, 25th, 50th, 75th, and 95th percentiles of the latency distribution.

In FSPX, the latency is affected by many factors, including the size of the input and output batches, the parallelism of the MemoryReader/MemoryWriter, and consequently the one of the StreamGenerator/StreamDrainer in the host program. Figure 9 shows the latency obtained by SD with a threshold equal to 0.025. With multiple sources of input streams (i.e., parallelism >1), we experience higher latency with larger batches, as expected. With a single source, we empirically found that more than 90% of the data items processed are removed from the output stream, while with multiple sources only 30 – 40%. As a result, with parallelism equal to 1 the latency is much higher compared to the other configurations because the output batches need more time to be filled.

An analogous behavior can be observed for FD in Fig. 10. In that case, the batch size affects latency as expected. Also, higher parallelism contributes to increased latency notably with large batches. We point out that FSPX can achieve stable latency results, even in the presence of multiple concurrent/parallel data transfers from and to the FPGA device, which is of great importance for latency-sensitive applications requiring stable processing delays.

5.5 Comparison with state-of-the-art SPEs

In this last part of the paper, we discuss the performance results achieved by FSPX against state-of-the-art SPEs. As it will be presented in §6, a direct comparison with other DSP frameworks targeting FPGAs revealed impossible. This is mainly due to the lack of FPGA tools having the same flexibility of FSPX in offloading the processing of whole pipelines on the FPGA. Furthermore, some works do not publicly provide their source code or, in other cases, the projects are quite dated and the targeted FPGA platforms obsolete. We refer to §6 for further details.

For the above reasons, we consider a comparison with two SPEs targeting the execution on traditional computing platforms. The first is Apache Flink [2], an open-source SPE targeting scale-out systems with a runtime system written in Java. It represents a *de-facto* standard SPE widely adopted by practitioners and researchers in the DSP domain. The second is WindFlow [13], a C++ DSP library specifically optimized for scale-up multi-core architectures. So doing, we showcase the efficiency of FSPX compared with industrial and research SPEs.

We use the source code of SD and FD provided by [5], which still includes Flink and WindFlow implementations of the two applications. To make a fair comparison, the applications have been slightly adapted to use the same input data size and output data size of the FPGA versions developed in FSPX, and with identical configuration parameters of the stateful operators. We tested the two applications using different configurations of parallelism per operator and batch size (the latter is available only in WindFlow, since Flink processes inputs one at a time only).

We point out that the performance results of FSPX are collected using a machine (equipped with two Intel CPUs released in 2014) equipped with the U50 FPGA board (released in 2019), while the experimental results of WindFlow and Flink have been collected using a different host equipped with a more modern Intel CPU (released in 2021) in order to have a comparison not necessarily in favor of the FPGA board.

Figure 11 reports the best throughput achieved for SD and FD. We report different cases: Flink, WindFlow, WindFlow using a batch size of 1,024 inputs (denoted by WindFlow+B), FSPX-1 is the FSPX bitstream with parallelism 1 per operator while FSPX-4 is the version with 4 replicas per operator. We report an additional case denoted by FSPX-Max, which represents the execution of FSPX with parallelism 4 per operator, where we use the bitstream version previously used in our analysis to measure the peak throughput. We recall that FSPX-Max does not incorporate a timestamp value to gather latency measurements per input, so reducing the input size and the impact on the PCIe bandwidth

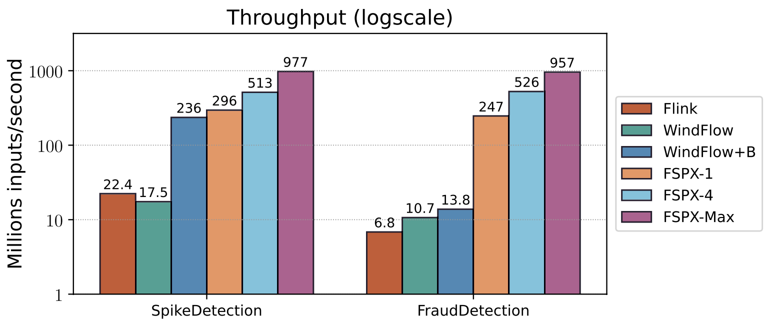


Fig. 11 Throughput comparison between FSPX and state-of-the-art SPEs

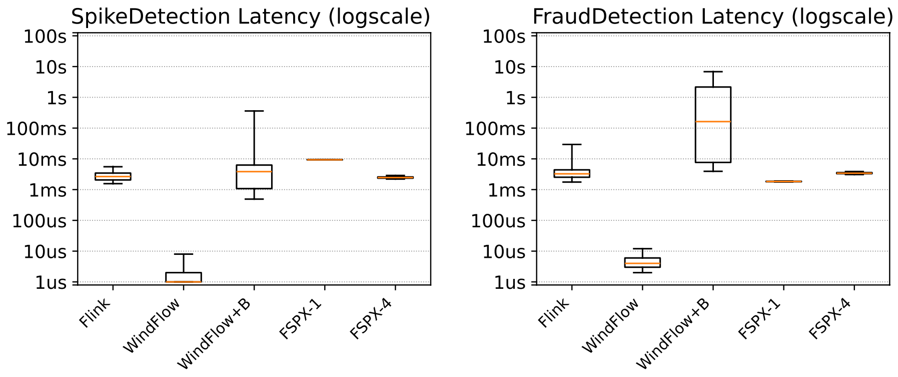


Fig. 12 Latency comparison between FSPX and state-of-the-art SPEs

consumption to transfer the same amount of data. In all cases, all versions of FSPX in the figure are executed with batches of 32,768 inputs, which represents the best choice.

On average, FSPX is capable of achieving 26.6× and 34.1× higher throughput against Flink and WindFlow with no-batching for SD, and 84.3× and 54× for FD. The use of batching in WindFlow demonstrates a performance boost of approximately 13.5× and 1.3× for SD and FD, respectively. By considering the Max configuration, FSPX achieves 43.6× and 4.1× better throughput for SD, and 140× and 69.2× for FD, outperforming, respectively, Flink and WindFlow in a remarkable manner.

In terms of latency, Fig. 12 shows the latency distribution of the different versions used before for the throughput comparison. As we can observe, although Flink does not adopt batching, its implementation targeting scale-out architectures (so incorporating useless data serialization/deserialization) exhibits a median latency of around 4ms for both applications, while WindFlow without batching provides latencies in the order of microseconds, so three orders of magnitudes lower. However, these two implementations sustain relatively low throughput (in the order of a few million inputs/second). With relatively small

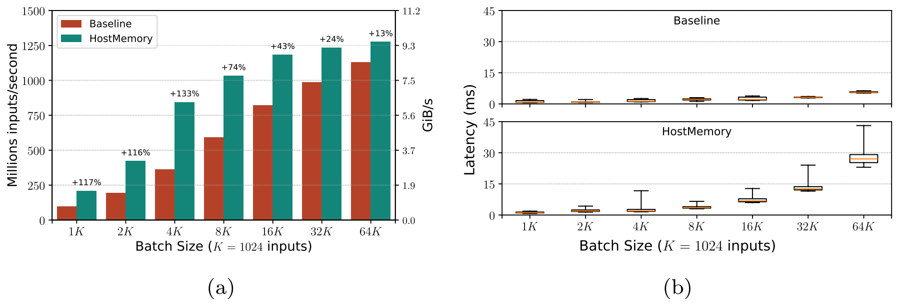


Fig. 13 (**a**) Vitis Host Memory throughput evaluation of SD application by varying the batch size. (**b**) Vitis Host Memory latency evaluation of SD application by varying the batch size

batching of $1K$ inputs, WindFlow provides higher throughput at the expense of latency, which becomes quite unstable, going from sub-milliseconds to hundreds of milliseconds (SD), or from a few milliseconds to almost 10 seconds (FD). We can observe instead that the behavior of FSPX is quite satisfactory: in all cases, it provides stable latency results, with a median in the order of a few milliseconds. Median latencies appear comparable with WindFlow (which however adopts smaller batches) and lower than Flink, confirming that FSPX generates FPGA bitstreams that provide highly efficient and reliable performance results.

5.6 Performance of using Vitis Host Memory

In this section, we evaluate throughput and latency when using Vitis Host Memory (VHM). This feature can be enabled on some data center platforms to let kernels access directly the host memory without making a memory copy on the off-chip memory of the FPGA device. The use of VHM requires changing the host program and the device memory port configuration. To enable such a feature, the user can specify it by adding the parameter `HOSTMEM=1` when compiling with the generated makefile. FSPX set all the memory ports to `HOST[0]` for the kernel link configuration to instruct the compiler to use VHM instead of the off-chip memory.

Figure 13 shows the experimental results obtained by the SD application with parallelism set to five, using VHM against the copy approach (named *baseline* in the plots). As we can see in Fig. 13a, the use of VHM results in a significant throughput improvement, more remarkable in percentage with small batches where the overheads of the copies in off-chip memory dominate. Therefore, VHM allows pushing the throughput of FSPX more closely to the theoretical PCIe bandwidth. In terms of latency (see Fig. 13b), VHM generally produces slightly higher latencies than the baseline since the reading and access time to the host memory is generally slower than using the off-chip memory directly.

5.7 Final considerations

The experimental results show the great benefit of using FPGAs for stream processing applications since FPGA-based implementations greatly outperform CPU implementations based on standard SPEs both in terms of throughput and latency. The actual performance is limited by the available host–device PCIe bandwidth, which still allows for achieving several hundreds of millions per second.

Traditional SPEs, based on CPU processing only, provide much lower performance due to the limited physical parallelism available, and their sources of overheads as studied in prior works [23]. The high potential of FPGAs is however counterbalanced by their programming approach, usually more complicated than the use of traditional SPEs offering high-level APIs friendly for data scientists and practitioners in this field. FSPX tries to fill also this gap, answering the need for new programming models for stream processing on FPGAs that hide (at least partially) the use of reconfigurable hardware as much as possible to permit a user-friendly way to exploit FPGA-based accelerators for the standard users of this application domain.

6 Related works

Considerable attention has been recently given to the use of hardware accelerators to speed up the execution of DSP applications. Such kind of hardware facilities are quite popular nowadays, both as PCIe boards of scale-up servers and on embedded systems as system-on-chip architectures. The papers addressing the challenge of using FPGAs for DSP focus on different aspects of the problem: most of them are related to specific use cases, for which a high-level programming approach hiding the complexity of FPGA programming has been given, in other (few) cases instead the focus was on the expressiveness of such tools in terms of variety of applications supported.

One work belonging to the first category is Glacier [14], a library supporting relational algebra queries over Xilinx FPGAs. Although based on boards of the same vendor as the ones targeted by FSPX (although older models since the paper was published in 2009), Glacier is strictly bounded to relational algebra applications, without any support to generic streaming algorithms working on unstructured streams. Along the same vein, the work in [20] proposes a tool to accelerate a class of DSP applications to detect complex event patterns in data streams. They propose efficient ways to execute automata on FPGAs. Analogously to Glacier, the domain of supported applications is narrow although the proposed approach is quite high level from the programmer's perspective.

A few papers have proposed approaches for general-purpose DSP targeting FPGAs. F-Storm [21] in an extension of Apache Storm featuring a new bolt (operator) interface to offload the processing on FPGAs. Although advocating a user-friendly programming model to incorporate FPGA computing in Storm, F-Storm actually requires adequate levels of expertise by the application programmer, since the development of OpenCL kernels is part of the business logic code of the operator, so under user responsibility. Furthermore, such a prototype has been mostly evaluated on benchmarks (run on an Arria10 PCIe board) not usual for DSP and outside the domain of streaming analytics (e.g., linear algebra kernels). Recently, a similar approach has been described in [12] for Storm and evaluated on small Cyclone-V Altera FPGAs. This project shares several limitations with F-Storm since one single operator per benchmark has been accelerated on the FPGA and not complete pipelines. Unfortunately, both projects are not currently maintained and their source node has not been publicly made available to the community.

A broader system to support FPGAs for several application domains like Machine Learning, Compressing, and Parsing is Fleet [18]. It requires the user to provide pieces of RTL code (Register Transfer Language) to process streams of tokens. However, the RTL code can be generated by a high-level program written in Scala, so targeting programmability. This approach looks promising, although the supported applications are very different from traditional DSP queries commonly used in analytics. Therefore, Fleet's contribution appears orthogonal to FSPX, and more DSP-oriented high-level programming approaches are still needed.

Other papers accelerate specific compute-intensive DSP operators on FPGAs. ShuntFlow [9] specifically targets sliding-window aggregation for a set of predefined associative binary operators, while other operators of the application run on CPU only. Although window-based operators are an important component of several DSP workloads, FSPX targets the processing of whole graphs of operators on FPGAs, while specific implementations for special classes of stateful operators will be the focus of our future works. Along the same line, we mention NetFPGA [10], a system supporting line-rate packet processing for network monitoring on FPGAs. Although such a domain is part of DSP, the approach is bounded to this use case only, and the FPGA bitstream is generated starting from a representation of the query using P4 [6], a special-purpose language for programming network devices.

7 Conclusions and future work

This paper presented FSPX, a programming framework to develop DSP applications exploiting FPGAs. This work extends the FSP prototype framework by adding an abstraction layer within the code generation module, and completely new device and host libraries, to produce tailored code for Xilinx FPGAs. The programmer develops the data-flow graph using a Python DSL. By running the Python program, a Vitis skeleton code is generated, which uses the new device library components of FSPX. The programmer is in charge of completing the implementation by providing the business logic code of the operators as proper functors. The compilation produces the bitstream to run on the FPGA. The experimental results show the effectiveness of FSPX, which outperforms open-source SPEs both in terms of throughput and latency.

Funding Open access funding provided by Università di Pisa within the CRUI-CARE Agreement. This research has been supported by the National Resilience and Recovery Plan (PNRR) through the

National Center for HPC, Big Data and Quantum Computing, and by the Italian PRIN project OUTFIT n. 2022BAL2F3.

Data availability FSPX source code and application datasets are available publicly and can be easily accessed from the Internet.

References

1. Andrade HCM, Gedik B, Turaga DS (2014) Fundamentals of stream processing: application design, systems, and analytics, 1st edn. Cambridge University Press, Cambridge
2. Apache: Apache flink. https://flink.apache.org/
3. Apache: Apache storm. https://storm.apache.org/
4. Arasu A, Babu S, Widom J (2006) The cql continuous query language: semantic foundations and query execution. VLDB J 15(2):121–142. https://doi.org/10.1007/s00778-004-0147-z
5. Bordin MV, Griebler D, Mencagli G, Geyer CFR, Fernandes LGL (2020) Dspbench: a suite of benchmark applications for distributed data stream processing systems. IEEE Access 8:222900–222917. https://doi.org/10.1109/ACCESS.2020.3043948
6. Bosshart P, Daly D, Gibb G, Izzard M, McKeown N, Rexford J, Schlesinger C, Talayco D, Vahdat A, Varghese G, Walker D (2014) P4: programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. 44(3):87–95. https://doi.org/10.1145/2656877.2656890
7. Cong J, Lau J, Liu G, Neuendorffer S, Pan P, Vissers K, Zhang Z (2022) Fpga hls today: successes, challenges and opportunities. ACM Trans Reconfigurable Technol Syst. https://doi.org/10.1145/3530775
8. Cugola G, Margara A (2015) The complex event processing paradigm. Springer International Publishing, Cham, pp 113–133
9. Gong S, Li J, Lu W, Yan G, Li X (2019) Shuntflow: An efficient and scalable dataflow accelerator architecture for streaming applications. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6
10. Ibanez S, Brebner GJ, McKeown N, Zilberman N (2019) The p4->netfpga workflow for line-rate packet processing. Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. https://api.semanticscholar.org/CorpusID:67873043
11. Kathail V (2020) Xilinx vitis unified software platform. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20, pp. 173–174. Association for Computing Machinery. https://doi.org/10.1145/3373087.3375887
12. Li H, You J, Li X, Song W (2022) Implementation and optimization of distributed stream processing system based on fpga. In: 2022 3rd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE), pp. 303–307. https://doi.org/10.1109/ICBAIE56435.2022.9985849
13. Mencagli G, Torquati M, Cardaci A, Fais A, Rinaldi L, Danelutto M (2021) Windflow: High-speed continuous stream processing with parallel building blocks. IEEE Transactions on Parallel and Distributed Systems pp. 1–1. https://doi.org/10.1109/TPDS.2021.3073970
14. Mueller R, Teubner J, Alonso G (2009) Streams on wires: A query compiler for fpgas. Proc VLDB Endow 2(1), 229–240. https://doi.org/10.14778/1687627.1687654

15. Ottimo A, Mencagli G, Danelutto M (2023) Fsp: a framework for data stream processing applications targeting fpgas. In: 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 92–99. https://doi.org/10.1109/PDP59025.2023.00021
16. Röger H, Mayer R (2019) A comprehensive survey on parallelization and elasticity in stream processing. ACM Comput Surv. https://doi.org/10.1145/3303849
17. Ronacher A Jinja 2. https://jinja.palletsprojects.com
18. Thomas J, Hanrahan P, Zaharia M (2020) Fleet: A framework for massively parallel streaming on fpgas. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, pp. 639–651. Association for Computing Machinery, New York https://doi.org/10.1145/3373376.3378495
19. Wingerath W, Ritter N, Gessert F (2019) General-Purpose Stream Processing. Springer International Publishing, Cham Chap. 5, pp. 57–74.
20. Woods L, Teubner J, Alonso G (2010) Complex event detection at wire speed with fpgas. Proc VLDB Endow **3**(1–2), 660–669. https://doi.org/10.14778/1920841.1920926
21. Wu S, Hu D, Ibrahim S, Jin H, Xiao J, Chen F, Liu H (2019) When fpga-accelerator meets stream data processing in the edge. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1818–1829. https://doi.org/10.1109/ICDCS.2019.00180
22. Xilinx: Inferring shift registers. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Inferring-Shift-Registers
23. Zhang S, He B, Dahlmeier D, Zhou AC, Heinze T (2017) Revisiting the design of data stream processing systems on multi-core processors. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 659–670. https://doi.org/10.1109/ICDE.2017.119