Seamless FPGA Integration with Stream Processing Engines

Alberto Ottimo[0009-0003-9411-2475], Gabriele Mencagli[0000-0002-6263-7723], and Marco Danelutto[0000-0002-7433-376X]

Department of Computer Science, University of Pisa, Pisa, 56127, Italy
{alberto.ottimo@phd.unipi.it, gabriele.mencagli@unipi.it, marco.danelutto@unipi.it}

Abstract. Stream processing is a computing paradigm enabling the analysis of data streams arriving at high speed from data producers. Its goal is to extract knowledge and complex events by processing streams with high throughput and low latency. To accomplish this goal, Stream Processing Engines (SPEs) try to exploit the parallel processing capabilities provided by modern hardware (usually multi-core CPUs and distributed systems). The exploitation of hardware accelerators, and in particular of FPGAs, is promising because they can maximize parallelism and reduce energy consumption. However, programming FPGAs is a very cumbersome and challenging task requiring a lot of expertise. In this paper, we discuss the seamless integration of FSPX, a prototype system for generating FPGA-based implementations of streaming pipelines, with an existing SPE (WindFlow). Our goal is to integrate these two tools by providing high-level programming interfaces to end users and guaranteeing high performance with efficient hardware utilization.

Keywords: Data Stream Processing, WindFlow, FSPX, FPGA

1 Introduction

A growing number of applications require the fast processing of data streams, i.e., unbounded sequences of data items (often records of attributes called *tuples*) generated by a plethora of data sources. The *Data Stream Processing* [2] (DSP) paradigm was introduced years ago as a way to process streams efficiently by *continuous queries* [10] described as data-flow graphs. Vertices of such graphs are *operators* doing computational steps, while arcs represent streams.

DSP applications need to process streams with high throughput and low latency to meet the Quality of Service constraints dictated by their specific use cases. To do that, several systems called *Stream Processing Engines* (SPEs) have been released by different communities and industries to facilitate the development of continuous queries, and to deploy them in a distributed architecture easily and transparently to the programmer.

One opportunity to accelerate DSP queries is represented by the exploitation of hardware accelerators. Field-Programmable Gate Arrays (FPGAs) represent valuable candidates since they have a very high performance-to-energy ratio. However, programming such devices is very complex and requires deep expertise by developers. FSPX [9] has been recently proposed as a solution to fill this gap. It provides a Python-based API to define the data-flow graph, and to generate starting from this representation the low-level code whose compilation results in the final bitstream to be loaded on the device. Although interesting, FSPX is currently a standalone project. It provides a host C/C++ library to interact with an FPGA pipeline generated by the tool. However, such an API is very low-level and error-prone to use by standard developers. Furthermore, FSPX do not have any integration with existing SPEs.

In this paper, we propose an integration between FSPX and WindFlow [6]. The latter is a C++17 parallel library for DSP, which exposes a very high-level programming interface and supports the execution on multi-core CPUs. This paper describes our integration of how to interact with an FSPX pipeline from a WindFlow application in an easy manner through a well-engineered interface and programming model. We also demonstrate with some experiments that the internal implementation and the host-to-device interaction still provide high performance and justify the use of FPGAs in this domain.

This paper is organized as follows. Sect. 2 provides the background by showing the features of DSP, WindFlow and FSPX. Sect. 3 shows the design of our integration and its implementation. Sect. 4 provides a preliminary experimental analysis, Sect. 5 describes related works, and Sect. 6 concludes the paper.

2 Background

This section presents a review of the DSP paradigm and a brief introduction to the WindFlow and FSPX research projects.

2.1 Data Stream Processing

DSP [2] is a computing paradigm enabling the continuous processing of data streams. DSP applications are continuous queries described as *data-flow graphs* of *operators*. Each operator is an intermediate transformation stage transforming inputs into outputs. Connections between operators represent data dependencies, implemented through data streams.

Operators can be classified into two categories: *stateless* operators produce outputs by computing pure functions that depend solely on the current input tuple. On the other hand, *stateful* operators maintain internal data structures to store statistics of the data stream. For each input tuple, the corresponding output (if any) will be computed by updating such internal data structures.

Data-flow graphs expose several parallelism patterns. Operators compute different input tuples in parallel, enabling *pipeline parallelism*. They can also compute different functions over the same or different tuples, enabling *task parallelism*. Furthermore, each operator can be internally replicated to increase its throughput: more threads can be used to replicate the operator processing logic.

This pattern is called *data parallelism* [10]. When operators are stateful, data parallelism is usually applied with *keyby* distributions [10], i.e., tuples are delivered to the replicas of the destination operator by assigning all tuples having the same key attribute to the same replica, which keeps the portion of the state associated with that key.

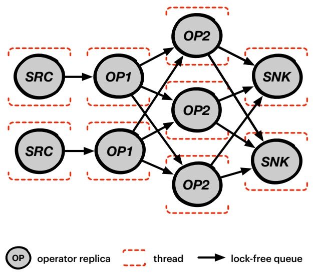
2.2 WindFlow

WindFlow is a parallel processing library [6] for shared-memory systems developed for accelerating streaming workloads. It provides a C++17 fluent API to instantiate operators and configure them, and two constructs (`PipeGraph` and `MultiPipe`) to create the data-flow graph. The library is built on top of the FastFlow parallel programming environment [1], so it inherits all the features of the FastFlow runtime system. This is based on lock-free single-producer single-consumer queues for fast data exchange between operators through memory pointers, while threads are pinned onto the cores of the machine.

Listing 1.1 shows an example of an application in WindFlow with four operators including the Source and the Sink. Each operator has its parallelism degree (`withParallelism`), and other configuration options. When an operator is declared with the `withOutputBatchSize`, the user specifies that output results will be delivered to the next operator in batches, to amortize runtime overheads.

Listing 1.1. Example of WindFlowAPI usage

```cpp
PipeGraph app;
Source src = Source_Builder(
  [](Source_Shipper<Raw_t> &s) -> void {...} )
  .withParallelism(2).build();
Map op1 = Map_Builder(
  [](const Raw_t &t) -> Input_t {...} )
  .withParallelism(2)
  .withOutputBatchSize(batchSize).build();
Keyed_Windows op2 = Keyed_Windows_Builder(
  [](const Input_t &a, const Input_t &b, Input_t &c) {
    c.value = a.value + b.value;
  })
  .withKeyBy( [](const Input_t &t) -> Key_t { return t.key; } )
  .withTBWindows(milliseconds(1000), milliseconds(100))
  .withParallelism(3).build();
Sink sink = Sink_Builder(
  [](std::optional<Input_t> &win_res) {...} )
  .withParallelism(2).build();
app.add_source(src).add(op1).add(op2).add_sink(sink);
app.run(); // execute the application
```



The example above provides two internal operators: the first is a Map computing one output per input according to a user-defined function provided to the constructor of the Map builder class (omitted in the snippet for the sake of brevity). The second is a Keyed_Windows operator maintaining the set of tuples received in the last second. The user computes a result upon completion of a new window of data, where windows move ahead of 100 ms each. Such a kind of window-based processing is usually adopted in DSP as a fundamental stateful operator, where the state management (e.g., window update with new tuples, and expiring of old tuples) is automatically done by the internal implementation.

2.3 FSPX

FPGAs are powerful hardware accelerators. They are notoriously challenging to program as they require specific engineering expertise. In recent years, High-Level Synthesis (HLS) tools [4] have been developed to mitigate such a problem. However, these tools are still challenging to be used by application domain experts. FSPX [9] is a research prototype that enables the development of DSP applications on FPGAs by abstracting many of the implementation details and guaranteeing optimal performance.

FSPX provides a Domain-Specific Language (DSL) written in Python that allows the high-level definition of a pipeline of operators that will be implemented in hardware by the resulting FPGA bitstream. The development of a DSP application in FSPX takes place in three stages. In the first stage, the developer should write a description of the application using the DSL, defining the operators and their properties (e.g. type, degree of parallelism, data distribution, results collection strategies). This description is used by FSPX to generate the application skeleton for the target FPGA. Such a skeleton fully instantiates the data-flow graph in terms of operators and data distributions according to the options selected in the DSL. However, the skeleton is incomplete because it misses the operator business logic (which should be user-defined). Therefore, in the second phase, the developer is required to implement the business logic code for each operator in the application, by defining some functions with predefined signatures depending on the operator type. As a last step, the application can be compiled into the FPGA bitstream and it is ready to run.

The current implementation of FSPX supports a set of stateless/stateful computing operators (i.e., Map, Filter, and FlatMap) that can be replicated according to their parallelism degree. An operator replica is directly connected to all the $m > 0$ subsequent operator replicas through m single-producer single-consumer FIFO streams. Each operator replica defines the policies to collect the tuples from $n > 0$ input streams (e.g., round-robin), and to dispatch the resulting output tuples to the replicas of the next operator in the pipeline.

In addition to computing operators (all implemented by an OpenCL kernel), FSPX also provides two built-in memory operators (each implemented by a separate OpenCL kernel): `MemoryReader` and `MemoryWriter`. The former produces an input stream that feeds the computing operators on the FPGA by reading a buffer filled by the host program with new inputs. The latter collects the stream of results from the computing operators and stores them in a pre-allocated global memory buffer that is eventually read by the host program.

FSPX provides a host library that should be used by the host program to interact with the FPGA pipeline and with its `MemoryReader` and `MemoryWriter` instances on the device. The host library includes the `StreamGenerator` class, which abstracts the low-level actions to push new inputs to a `MemoryReader` operator, and the `StreamDrainer` class, which collects the results generated by a `MemoryWriter` operator in a convenient way.

FSPX employs the K-buffering technique, a generalization of the well-known double-buffering optimization. The idea is to overlap the data transfers of the next up to $K-1$ buffers while the currently running kernel is using the previous one. The `StreamGenerator` and the `StreamDrainer` adopt this technique by allocating $K > 1$ buffers in the FPGA memory before the application starts. During the application lifetime, buffers that have been consumed are recycled to avoid repeated allocations and de-allocations.

3 Proposed Architecture

The seamless integration of FSPX into WindFlow requires a careful design to hide most of the low-level details of the FSPX configuration and of the FPGA board. In our solution, the idea is to allow the programmer to offload a partition of the data-flow graph (i.e., likely the most computationally demanding part that will be implemented as a FSPX pipeline) to the device. At the same time, data generation, results collection, and some pre-/post-processing activities are executed by the host. The general solution is depicted in Fig. 1.

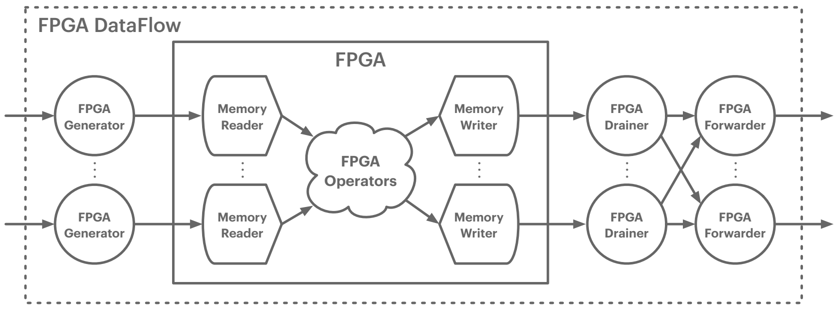


Fig. 1. WindFlow and FSPX integration: proposed architecture.

3.1 FPGA DataFlow

The `FPGA_Dataflow` is a new meta operator that we introduce to provide the integration with FSPX. As any library operator, it can be used by adding it to an existing `MultiPipe`. The main constraint is that the preceding operator(s) before the `FPGA_Dataflow` is/are obliged to produce inputs in batches, through the use of the method `withOutputBatchSize` in its/their builder/s.

The `FPGA_Dataflow` can be instantiated as shown in Listing 1.2. The class builder is a template with two parameters: the first is the data type of inputs arriving at the FPGA dataflow (in the snippet `input_t`), while the second is the data type of outputs produced by the dataflow (in the snippet `output_t`).

The user specifies the name of the operator for logging purposes and the pathname of the bitstream file previously generated through the FSPX programming model and the Vitis tool-chain. Other configuration options will be described later in this section. In the next part, we describe in detail the internal private operators implementing the dataflow, and the seamless interaction with the FSPX bitstream according to the architecture previously shown in Fig. 1.

Listing 1.2. Fluent interface to instantiate an `FPGA_Dataflow` meta operator in WindFlow.

```
FPGA_DataFlow df = FPGADF_Builder<Input_t, Output_t>
    .withName("dataflow")
    .withBitStream("/home/user/fpga_examples/bitstream.xclbin")
    .withParallelism(fpga_par)
    .withMaxBatchSize_Gen(max_batch_g)
    .withNumBatches_Gen(n_batch_g)
    .withMaxBatchSize_Drainer(max_batch_d)
    .withNumBatches_Drainer(n_batch_d)
    .withOutputBatchSize(batch_size)
```

3.2 FPGA Generator

The `FPGA_Generator` private operator is hidden to the programmer since it is added to a `MultiPipe` as an effect of adding the `FPGA_Dataflow` meta operator.

The operator makes use of the `StreamGenerator` class provided by the FSPX host library to forward incoming batches from preceding WindFlow operators to the FSPX pipeline. More precisely, each replica of the `FPGA_Generator` instantiates a private copy of the `StreamGenerator` that is coupled with a specific `MemoryReader` on the FPGA application. Therefore, the parallelism degree of the `FPGA_Generator` should match the number of `MemoryReader` in the bitstream. Since the FPGA bitstream is loaded at runtime, the `FPGA_Dataflow` automatically replicates the `FPGA_Generator` according to the number of `MemoryWriter` present in the FSPX application. This information can be obtained by calling the `clGetKernelInfo()` function with `CL_KERNEL_COMPUTE_UNIT_COUNT` (a parameter extension API provided by Xilinx) on the `MemoryReader` OpenCL kernel.

Each instance of the `FPGA_Generator` fetches an empty buffer from its own `StreamGenerator` object and copies the tuples of the current input batch into that buffer. Then, it notifies the `StreamGenerator` that the buffer is ready to be consumed by the FSPX pipeline and triggers the execution of the associated `MemoryReader` on the FPGA, which will be responsible for reading that buffer and feeding the FPGA operators with new tuples. Once the stream is ended, the `FPGA_Generator` replica notifies the End-of-Stream (EOS) special message by pushing an empty batch flagged with the EOS flag equal to `true`.

3.3 FPGA Drainer

The `FPGA_Dataflow` meta operator includes a second private operator called `FPGA_Drainer`. Its role is dual to the `FPGA_Generator`, i.e., it is in charge of receiving batches of computed results from the FSPX pipeline (batch of results are filled by the `MemoryWriter` replicas on the device) and delivering them to the `FPGA_Forwarder` (see the next section). Similar to the `FPGA_Generator`, this private operator is replicated with a parallelism degree that must match the number of `MemoryWriter` present in the FPGA bitstream.

In the initialization phase, the `FPGA_Drainer` allocates $K > 0$ different buffers in global memory and enqueues the same number of executions of the `MemoryWriter` kernel. This way, the operator ensures the overlapping of data transfers with the execution of `MemoryWriter` kernels. After the initialization phase, this operator waits for the completion of the launched `MemoryWriter` instances to read back a batch of results from the FPGA. Then, it delivers the obtained batch to the `FPGA_Forwarder` private operator.

3.4 FPGA Forwarder

The `FPGA_Drainer` is in charge of delivering results produced by the FSPX pipeline to the next operator running in the host. The delivery of such results might generate overheads. For example, suppose the `FPGA_Drainer` replicas are requested to deliver individual tuples in a keyby manner. In that case, tuples in each output batch must be read to push them one-at-a-time (or again in batches) to the right destination replicas. This might require tuning the parallelism degree of the `FPGA_Drainer` to remove potential bottlenecks in this phase. However, as previously discussed, the number of replicas of the `FPGA_Drainer` cannot be an arbitrary value, but it matches the number of `MemoryWriter` in the bitstream. Tuning the right parallelism would be needed in that case to produce different bitstreams (with different numbers of `MemoryWriter` each), which is time-consuming from the FPGA compilation perspective.

For the reason above, we introduce a new private operator included in the `FPGA_Dataflow` called `FPGA_Forwarder`. Each replica of the `FPGA_Drainer` sends pointers of the received batches of results from the device to the replicas of the `FPGA_Forwarder`. The latter can be easily configured to have any parallelism degree (it is conceptually a stateless operator), and it is responsible for delivering the results to the next operator by respecting the required distribution semantics.

4 Experiments

The experiments are conducted on a host machine with two Intel Xeon E5-2650 V3 CPUs and 128 GiB of DDR4 at 2133 MHz Quad-channel. Each CPU has 10 cores (20 hardware threads) sharing an Intel Smart Cache L3 of 25 MiB. Each core has a clock rate of 2.3 GHz (3.0 GHz with Turbo Frequency), and an L2 cache of 256 KiB. The host machine is equipped with a Xilinx Alveo U50 Data Center Acceleration Card. 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. The FSPX host library uses the OpenCL standard implemented by the Xilinx Runtime Library (XRT).

4.1 Benchmark Applications

We evaluate our work with a synthetic and a real-world application. The synthetic application filters those tuples that do not satisfy a given predicate. The FSPX pipeline is composed of a three-phased logical pipeline: a `MemoryReader`, a stateless Filter, and a `MemoryWriter`. The WindFlow host program is a pipeline comprising a Source generating a stream of tuples that satisfy the predicate in a given percentage (called *Keep Rate*), a `FPGA_Dataflow` to offload the filter computation to the FPGA, and a Sink counting the number of results.

The SpikeDetection (**SD**) application was chosen as a real example. This application analyses a stream of sensor readings and detects spikes of temperature. The FPGA implementation with FSPX is a pipeline of a `MemoryReader`, a stateful Map (Average Calculator) emitting a moving average value computed over a counting window, a Filter (Spike Detector) evaluating whether the current sensor reading is a spike, and a `MemoryWriter`. The Map operator implements a key-partitioned sliding window. The business logic code of the operator employs the shift register pattern [13]. The Filter implements a simple Boolean predicate, ensuring that only records that satisfy the predicate are delivered to the `MemoryWriter`. The WindFlow host program is a pipeline with a Source generating data by reading them from a dataset file, an `FPGA_Dataflow` interacting with the FPGA pipeline, and a Sink collecting results.

4.2 Experimental setup

In the experimental evaluation, we investigate the sustained throughput of the proposed architecture, denoted as *WindFlow+FSPX*. As our Baseline, we use a hard-coded C/C++ implementation of the host program that utilizes the low-level API provided by the FSPX host library. It employs a single thread per `MemoryReader` for the generation of data, and a single thread per `MemoryWriter` for the collection of results. This version tries to reduce at best the overheads from the host side. Therefore, our goal is to understand the additional overheads paid by programming the host program with a high-level library for stream processing like WindFlow. The tuple size of the synthetic application is a 16-byte struct composed of 4 fields. The SD application, instead, employs an input tuple of 8 bytes, and an output tuple of 16 bytes.

The experiments are repeated five times, configuring the batch size of both `MemoryReader` and `MemoryWriter` to 2^{20} tuples, setting $K = 4$ global buffers allocated using the Vitis Host Memory. On WF+FSPX, each source replica emits 512 batches, while in the Baseline we instruct the Source threads to emit the same amount of batches overall.

We set the `FPGA_Forwarder` parallelism degree to be the same as the Sink operator. However, Sink replicas and `FPGA_Forwarder` replicas are chained in the same threads to avoid thread over-subscription, leaving more cores available to increase the Source parallelism (i.e., the input rate of the application).

FSPX generates optimal bitstreams for our FPGA board with FMax of 300 MHz and with an Initiation Interval (II) of 1, meaning that every operator replica can process one new input per clock cycle. As a result, the ideal throughput of the bitstream can be predicted as the product of the parallelism of the operators and the operating frequency FMax.

Synthetic application. In this set of experiments, we vary the parallelism degree and the keep rate of the Filter. By controlling the keep rate, we can control the number of results produced by the FSPX pipeline, and so the utilization of the device-to-host bandwidth. Fig. 2 shows the throughput by changing the keep rate. We consider three parallelism degrees 1, 2, and 3. With parallelism 2 for example, the FSPX pipeline consists of 2 `MemoryReader`, 2 replicas of the Filter, and 2 `MemoryWriter`. The total number of threads is shown in Table 2.

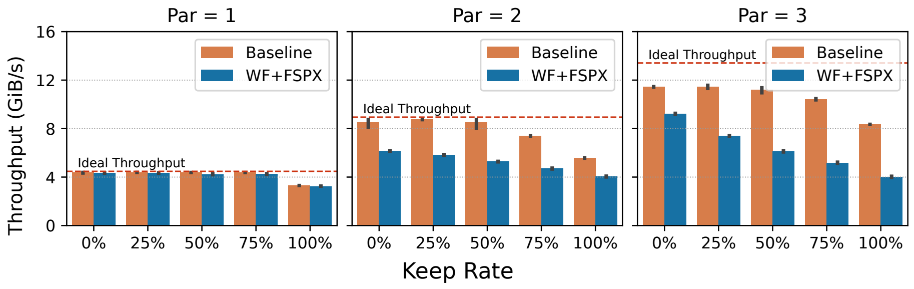


Fig. 2. Throughput of the synthetic application: comparison between WF+FSPX and the Baseline with different parallelism degrees and keep rates.

With parallelism 1, the throughput remains stable by changing the keep rate, while the reduction in terms of throughput between WF+FSPX and the Baseline remains limited (on average it is of 2%). With parallelism 2, the FPGA pipeline consumes more inputs per second, and more threads are used on the host side to try to exploit adequately the device. This generates higher overheads in the host program, more significant with the WF+FSPX version compared with the hard-coded Baseline (on average the throughput loss of using WF+FSPX is of 49%). In this case, the effect of the keep rate appears more remarkable, since with a higher keep rate (so more results produced by the FSPX pipeline), the overall throughput is lower. With parallelism degree 3 the effect is very evident, with a throughput loss of 71% of WF+FSPX compared with the Baseline.

Table 1 shows the number of threads required to achieve the highest throughput in WF+FSPX and with the Baseline. One of the reasons for the higher number of threads with WF+FSPX is the limited throughput of the Sources, which are only capable of producing 50–60 Mt/s each. This is likely because WindFlow, as well as other SPEs, are designed to process single tuples or small batches. These mechanisms are not designed for the generation of large batches, which are needed to maximize the utilization of the PCIe bandwidth and the FPGA capabilities.

Table 1. Bitstreams and host threads configuration of the synthetic application.

Par.	FMax MHz	II	Ideal Throughput MT/s	Ideal Throughput GiB/s	Baseline #Threads	WF+FSPX #Threads Keep Rate 0%	25%	50%	75%	100%
1	300	1	300	4.8	2	8	9	10	12	12
2	300	1	600	9.6	4	17	19	22	25	28
3	300	1	900	14.4	6	25	29	33	39	42

SpikeDetection application. SpikeDetection has been employed as a real-world application. Given that the keep rate is very low, we assigned the parallelism of the Sink to match the number of `MemoryWriter` of the FSPX pipeline. This saves cores that can be used to increase the number of Sources to maximize the throughput of the input generation phase.

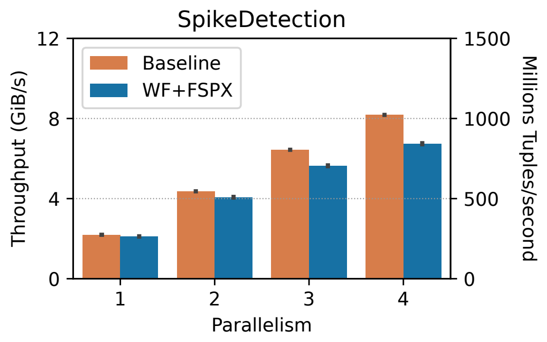


Fig. 3. Throughput of the SD application: comparison between WF+FSPX and the Baseline with different parallelism degrees.

Fig. 3 shows the throughput by changing the parallelism degree up to 4. With parallelism of 1, WF+FSPX performs slightly worse than the Baseline, with an average loss in throughput of 4%. With parallelism higher than 1, however, the loss is more significant, with an average decrease of 7%, 14%, and 21%.

5 Related works

The seamless utilization of hardware accelerators to enhance the performance of DSP applications has been the subject of intensive research. Saber [11] is an SPE accelerating sliding-window aggregates on GPUs. Similarly, FineStream [14] adopts a hybrid approach where operators can be scheduled on the host or the GPU based on their properties. In both works, operators are the ones of relational

Table 2. Bitstreams and host threads configurations of SD application.

Par.	FMax MHz	II	Ideal Throughput MT/s	Ideal Throughput GiB/s	Baseline #Threads	WF+FSPX #Threads
1	300	1	300	2.4	2	9
2	300	1	600	4.8	4	18
3	300	1	900	7.2	6	27
4	300	1	1200	9.6	8	36

algebra, and no support for general operators doing arbitrary imperative code is provided. G-Storm [3] provides general support to offload computations on a GPU in Apache Storm. However, the programmer is involved in the manual implementation of GPU kernels, which requires skills and expertise.

The integration of FPGAs is more challenging than GPUs since their programming abstractions usually pose additional challenges and often a deep hardware and compiler knowledge. Glacier [7] is an SPE oriented to relational algebra, with a limited set of built-in operators that can be implemented on the FPGA. F-Storm [12,5] integrates the use of FPGAs into Storm. The API is not adequately high level, since data transfers, bitstream management activities, and programming, and still the responsibility of the user.

FSPX [8,9] is a system designed to help the programmer in developing general-purpose DSP applications for FPGAs (both Intel and Xilinx). It is based on a Python DSL to define a pipeline of operators. A code generation approach is enforced to generate the pipeline skeleton, to be completed by the user with some C/C++ functions of the operator's business logic. FSPX provides a host library to copy input data to the device and to get results from it continuously. However, this API is low level and the integration of FSPX with existing SPEs is possible provided that proper operators to transfer data and to collect results with the device are implemented based on this library. This paper discussed the integration with the WindFlow library.

6 Conclusions

This paper presents an integration between WindFlow, a C++ parallel library for DSP, and FSPX, a tool for generating FPGA pipelines with a reduced programming effort. This work shows the API and implementation of such an integration, and we discussed the performance overheads induced by running the host program (feeding the FSPX pipeline and collecting the produced results) with a high-level general-purpose DSP library like WindFlow compared with a hard-coded low-level host program controlling the FPGA implementation.

Acknowledgments. 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.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280. John Wiley & Sons, Ltd (2017). https://doi.org/https://doi.org/10.1002/9781119332015.ch13
2. Andrade, H.C.M., Gedik, B., Turaga, D.S.: Fundamentals of Stream Processing: Application Design, Systems, and Analytics. Cambridge University Press, 1st edn. (2014)
3. Chen, Z., Xu, J., Tang, J., Kwiat, K., Kamhoua, C.: G-storm: Gpu-enabled high-throughput online data processing in storm. In: Proceedings of the 2015 IEEE International Conference on Big Data (Big Data). p. 307–312. BIG DATA '15, IEEE Computer Society (2015). https://doi.org/10.1109/BigData.2015.7363769
4. Cong, J., Lau, J., Liu, G., Neuendorffer, S., Pan, P., Vissers, K., Zhang, Z.: Fpga hls today: Successes, challenges, and opportunities. ACM Trans. Reconfigurable Technol. Syst. **15**(4) (2022). https://doi.org/10.1145/3530775
5. Li, H., You, J., Li, X., Song, W.: 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 (2022). https://doi.org/10.1109/ICBAIE56435.2022.9985849
6. Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., Danelutto, M.: Windflow: High-speed continuous stream processing with parallel building blocks. IEEE Transactions on Parallel and Distributed Systems pp. 1–1 (2021). https://doi.org/10.1109/TPDS.2021.3073970
7. Mueller, R., Teubner, J., Alonso, G.: Streams on wires: A query compiler for fpgas. Proc. VLDB Endow. **2**(1), 229–240 (2009). https://doi.org/10.14778/1687627.1687654
8. Ottimo, A., Mencagli, G., Danelutto, M.: 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 (2023). https://doi.org/10.1109/PDP59025.2023.00021
9. Ottimo, A., Mencagli, G., Danelutto, M.: Boosting general-purpose stream processing with reconfigurable hardware. The Journal of Supercomputing (2024). https://doi.org/10.1007/s11227-024-05894-4
10. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Comput. Surv. **52**(2) (2019). https://doi.org/10.1145/3303849
11. Theodorakis, G., Koliousis, A., Pietzuch, P., Pirk, H.: Lightsaber: Efficient window aggregation on multi-core processors. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. p. 2505–2521. SIGMOD '20, Association for Computing Machinery (2020). https://doi.org/10.1145/3318464.3389753
12. Wu, S., Hu, D., Ibrahim, S., Jin, H., Xiao, J., Chen, F., Liu, H.: When fpga-accelerator meets stream data processing in the edge. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). pp. 1818–1829 (2019). https://doi.org/10.1109/ICDCS.2019.00180
13. Xilinx: Inferring shift registers, https://docs.amd.com/r/2023.1-English/ug1399-vitis-hls/Inferring-Shift-Registers
14. Zhang, F., Yang, L., Zhang, S., He, B., Lu, W., Du, X.: Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures. In: Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference. USENIX Association (2020)