FSP: a Framework for Data Stream Processing Applications targeting FPGAs

1st Alberto Ottimo
Department of Computer Science
University of Pisa – Italy
a.ottimo@phd.unipi.it

2nd Gabriele Mencagli
Department of Computer Science
University of Pisa – Italy
gabriele.mencagli@unipi.it

3rd Marco Danelutto
Department of Computer Science
University of Pisa – Italy
marco.danelutto@unipi.it

***Abstract*—FPGA architectures are becoming popular because of their high performance-to-energy ratio. Nonetheless, their effective exploitation is often counterbalanced by a high programming effort, since most of the modern hardware description languages provide only low-level programming abstractions. This paper proposes FSP, a framework to productively support the development of Data Stream Processing applications on CPU+FPGA System-on-Chip devices (SoCs). By exploiting a code generation approach starting from a high-level DSL in Python, FSP generates an efficient OpenCL skeleton implementation of the parallel pipeline on FPGA and the library to be used by host programs to transfer inputs and collect results to/from the FPGA program. The experimental results showcase the effectiveness of FSP on an SoC equipped with an Intel Arria 10 FPGA by running two streaming benchmark applications.**

***Index Terms*—Data Stream Processing, FPGA, OpenCL, Shared Memory, System-on-Chip Devices**

I. Introduction

The last years have been witnessing rapid growth in the volume of data that is being generated in the form of *data streams*, i.e., unbounded sequences of inputs received at high speed from thousands of data sources spread in our cities, buildings, environment and from the metaverse. With such a proliferation of streams, analyzing them to extract information, insights, and hidden value is a capital asset for many companies and industries.

Processing streams in an efficient and effective manner has been the subject of several research activities falling under the theoretical umbrella of the *Data Stream Processing* (DSP) [1] research field. Most of the first attempts have been conducted by the database research community, with SQL-like formalisms to process streams analogously to finite relations. To support the execution of such relational streaming queries, Data Stream Management Systems (DSMSs) [1] have been released in the past. In recent years, this topic has been extended to face two important issues: *i)* the support of non-relational applications with DSP frameworks capable of running general computations beyond the space of SQL-like queries; *ii)* the efficient exploitation of parallel hardware of different kinds, including traditional servers equipped with multi-core CPUs and clusters.

With the current trend of hardware evolution, DSP frameworks are demanding the efficient support of accelerators to speed up their performance by providing high-throughput and low-latency processing capabilities. GPUs and FPGAs are fundamental building blocks of supercomputers nowadays. Furthermore, they represent hardware capabilities also available on resource-constrained embedded resources like System-on-Chip (SoC) devices present in the Internet of Things. So, supporting accelerators for DSP is pivotal both for enabling efficient stream processing on high-performance architectures as well as on Edge/IoT computing resources.

Although GPU programming has improved in terms of programming abstractions, and recent attempts to exploit GPUs for DSP have been presented in the literature (they will be mentioned in § VII), the exploitation of reconfigurable hardware in a productive manner is still a research issue. FPGAs are very powerful, and they promise effective performance-to-energy ratios useful for embedded architectures. However, the learning curve to use FPGAs is quite challenging and the definition of novel programming frameworks assisting programmers in developing FPGA-based stream processing applications represents an open research direction.

This paper presents FSP, a framework for DSP applications for SoCs equipped with a multi-core CPU and an integrated FPGA. The contributions of this paper are the following:

- FSP reduces the programming effort in using reconfigurable hardware for general-purpose non-relational streaming applications owing to its code generation approach which, starting from a Domain-Specific Language (DSL) in Python, generates the OpenCL skeleton code to be integrated by the developer with few business logic functions in the C programming language;
- the generation of the OpenCL skeleton of a DSP application includes a ready-to-use efficient implementation of intra-FPGA communications between OpenCL kernels representing the application operators (introduced in § II-A), so alleviating the programming effort for this part of the runtime code by the application developer;
- FSP also generates from the DSL the header-only library to be used by host programs to interact with the DSP application on the FPGA.We provide several mechanisms

This research has been supported by the INTERCONNECT project no. PRA_2022-2023_9, by the EuroHPC TEXTAROSSA project no. 956831, and by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on HPC, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali di R&S (M4C2-19)" - Next Generation EU (NGEU).

to offload streaming data to the FPGA and to collect results efficiently.

The experimental analysis has been conducted using two streaming applications from an open-source benchmark suite [2] and discussing the performance results on the Terasic Han Pilot Platform SoC, equipped with a dual-core ARM CPU and an Intel Arria 10 FPGA.

This paper is organized as follows. § II introduces the background. § III provides an overview of FSP with its workflow. § IV describes the DSL in Python. § V shows the code generation approach and the implementation. § VI shows the experimental evaluation, § VII provides the related works and § VIII draws the conclusions of this paper.

II. Background

In this part, we briefly summarize the main concepts of the DSP paradigm and FPGA devices.

A. Data Stream Processing

DSP applications are data-flow graphs where arcs model unbounded streams while vertices are operators doing intermediate computations applied over the inputs to produce outputs. In relational streaming applications (i.e., the ones that can be developed using SQL-like formalisms [3]), operators are the ones of linear algebra (e.g., projection, selection, joins) while graphs have regular structures (e.g., trees). In general-purpose DSP, which is the paradigm of interest in this paper, graphs are generic while operators do generic computations.

DSP applications expose different parallelism exploitation patterns. Operators usually work in parallel on different inputs, while each operator can be internally replicated, with replicas doing the same processing logic on different inputs distributed by the previous operator in the graph (the so-called data parallelism [4]). Distributions can follow different dispatching policies: *round-robin* tries to assign the same number of inputs to the replicas, while *key-by* assigns all the inputs having the same key (e.g., a specific field of the input) to the same replica.

Furthermore, operators can be *satateless* if outputs depend on the inputs only, without keeping any state information of the stream history. Otherwise, operators are *stateful* if they maintain a state used to compute outputs. A very used pattern in DSP is to use stateful operators with a key-by dispatching policy (*partitioned-stateful operators*), where each distinct key value is associated with a separated state object accessed privately by the replica receiving inputs having that key.

B. FPGA Architectures

Field Programmable Gate Arrays (FPGAs) are integrated circuits consisting of an array of configurable hardware blocks connected via programmable interconnects. The main blocks are *Basic Logic Elements* (BLEs), which are composed of a *look-up table* (LUT), a *flip-flop*, and a multiplexer. Modern FPGAs have higher-level functionality blocks, including memory blocks (BRAM), which are used to store data and transfer data between on-chip resources, dedicated hard blocks such as *Digital Signal Processing* (DSP) blocks that have dedicated circuitry implementing multiply and accumulate operations, and I/O blocks used to communicate with external devices such as microcontrollers or external memories.

Developers are in charge of providing a description of the functionality they want to accelerate at hardware using a proper formalism, which will be translated into a bitstream file (i.e., describing the configuration of the FPGA blocks) by a so-called *offline compiler*. In recent years, both Intel and Xilinx have proposed OpenCL [5] for FPGA programming to ease the effort spent by developers. OpenCL increases programmability by abstracting the complexity of direct hardware programming with Hardware Description Languages like Verilog or VHDL.

The default OpenCL model is called *NDRange* (NDR), which employs multiple work-items grouped into work-groups. Although NDRange kernels are well suited for GPU programming, this execution model does not always guarantee optimal hardware design for FPGAs [6]. Single Work-item (SWI) kernels, instead, allow developers to avoid partitioning of data across work-items, while the whole kernel code is developed using a sequential programming model (with loops) similar to C programming. Such kind of kernels enables the offline compiler to extract *pipeline parallelism* at compile time, in order to run loop iterations in parallel.

The Intel FPGA SDK for OpenCL [6] provides a mechanism for kernel-to-kernel communication based on FIFO queues called *Intel Channels*, which are directly implemented using the embedded memory blocks of the FPGA. This feature decouples data movements between executing kernels, alleviating the memory pressure on the off-chip FPGA memory and removing synchronization overheads with the host program.

III. FSP Overview

FSP[1] is a framework prototype to leverage FPGA devices to accelerate DSP applications. This prototype aims to ease the development of streaming applications by offloading part or the whole computation of a pipeline of streaming operators onto an FPGA device. The development of an application with FSP starts by describing the pipeline of operators through a DSL as shown in Fig. 1. Such a DSL provides an easy-to-use way for high-level programmers to structure the application DAG (i.e., a pipeline) by expressing different kinds of operators, with different input dispatching policies and state management properties, and connecting them according to the logical dataflow driven by the application semantics.

The DSL is expressed in the Python programming language and provides a basic set of operators which run on the FPGA:

- the *Filter* operator removes from the streams all inputs not respecting a user-defined predicate;
- the *Map* operator applies a user-defined function on each input by producing a corresponding output having the same or a different data type.

In addition to these operators, FSP provides some built-in operators (still run by the FPGA) that interact with the host

[1]The source code is available at https://github.com/blackwut/FSP.

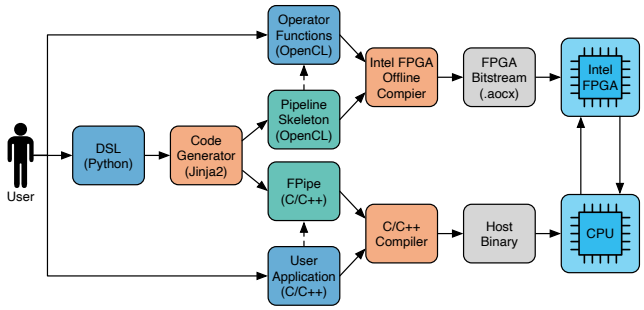

Fig. 1. FSP workflow to develop a DSP application leveraging FPGA.

application to receive a stream of inputs to be consumed, or to produce a stream of output results:

- the *Source* operator produces an input stream feeding the other operators on the FPGA by forwarding data continuously arriving from a host application;
- the *Sink* operator collects the streams of results and makes them continuously available to a host application.

Each FSP operator can be internally parallelized by running more functionally equivalent replicas that will work on different inputs in parallel, according to the data-parallelism paradigm introduced in § II. From the DSL perspective, the FSP programmer specifies the degree of parallelism of each operator (i.e., number of replicas), and the dispatching policy to route inputs produced by the previous operator to such destination replicas. The dispatching policies provided by FSP are the following:

- the *round-robin policy* allows inputs to be dispatched 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;
- the *key-by policy* allows inputs having the same *key* (i.e., an attribute of the input data structure or the result of a function that computes the key based on some fields of the input) to be always sent to the same replica of the next operator;
- the *broadcast policy* allows sending a copy of each input to each destination replica.

After defining the structure of the operator pipeline, the number of replicas, and the dispatching policies, FSP generates an OpenCL code for the device (*pipeline skeleton*) using Jinja2 [7]. In the pipeline skeleton (see Fig. 1), the code generated for each operator replica consists of an OpenCL kernel doing the following logical steps:

- it handles the gathering of inputs received from the replicas of the previous operator by polling from a set of Intel Channels FIFO queues implemented directly on the FPGA memory resources;
- for each input, it calls the user-defined function associated with the operator;
- it handles the transmission of the computed outputs to the replicas of the next operator according to the specified dispatching policy chosen by the programmer;
- it handles the *End-of-Stream* (EoS) propagation logic to terminate the pipeline correctly.

Therefore, the generated code contains several pieces of the pipeline logic that are efficiently implemented for the device without any programming effort by the FSP user. However, since FSP 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 complemented by the programmer with the user-defined functions (one per operator) that must respect a given signature based on the operator type (denoted as *operator functions* in Fig. 1).

FSP generates also the code to be used by the host application (a set of C++ header files). It consists of a set of functions containing the so-called OpenCL boilerplate to launch all the OpenCL kernels as they are defined in the generated device code as well as all the runtime calls to instantiate the memory buffers. Furthermore, the header files implement an API that will be used by the host application to push and pop batches of inputs/outputs to be read/consumed by the device (so transparently interacting with the Source and Sink operators on the FPGA).

IV. DSL

This section introduces the basic feature of our DSL. The first step to declare an application is to create an instance of the `FApplication` class as shown in Listing 1.

```
app = FApplication('./myApp', 'input_t', FTransferMode.SHARED)
```

Listing 1. Example of declaring an application with FSP.

The programmer can specify the destination folder of the generated code (i.e., a directory named `myApp`), the input datatype of the pipeline (i.e., the Source operator will accept inputs with `input_t` as datatype), and the transfer protocol used to perform the host-to-device (H2D) and device-to-host (D2H) data transfers (see § V-C).

Operators are declared by instantiating the `FOperator` class. Listing 2 shows the declaration of a Map operator using a key-by dispatching policy. The programmer can tune some options provided at the implementation level about the way in which inputs are retrieved from the incoming buffers (e.g., using the non-blocking policy as it will be described in the following sections). In this example, the Map operator is parallelized with two internal replicas to increase throughput and it produces a stream of output results having a new datatype `output_t`.

```
map = FOperator('map',                          # Name
                2                               # Parallelism degree
                FOperatorKind.MAP,              # Base operator kind
                FGatherMode.NON_BLOCKING,# Policy for reading inputs
                FDispatchMode.KEYBY,            # Policy for sending outputs
                'output_t')                     # Output datatype
```

Listing 2. Map operator with key-by dispatching policy.

A stateless operator, excluding Source and Sink operators, becomes stateful by adding a buffer to it. Such a buffer is a memory space that the user decides to make available to the operator, where it can save historical information about

the received inputs, and can be used to compute results in a generic stateful manner. Listing 3 shows the methods to add a buffer residing in the three different memory spaces (i.e., private, local, and global according to the OpenCL standard). Private buffers are implemented by the offline compiler with registers and should be used to store variables or arrays with few elements. A local buffer is implemented in dedicated memory blocks inside the FPGA and should be used to store a small state. Global buffers are stored in external memory (i.e., DDR and HBM) and can hold large states. Private and local buffers are visible only by a replica of an operator. Global buffers can instead be accessed by a single replica or by all the replicas of the operator depending on the access type specified with the DSL. A global buffer can be specified as *read only*, *write only*, or *read-write*, while private buffers are implicitly read-write.

```
map.add_private_buffer('char', 'array', size=16)
map.add_local_buffer('float', 'matrix', size=(32, 128))
map.add_global_buffer('int', 'img', size=2**30, FBufferAccess.RW_ALL)
```

Listing 3. Add private, local, and global states to an operator.

After the declaration of all the operators, they are inserted in the right order to the pipeline as shown in Listing 4. By calling the `generate_code()` method, FSP generates the device code and the host header-only library.

```
app.add_source(source)
app.add(map)
app.add_sink(sink)
app.generate_code()
```

Listing 4. Adding operators and device/host code generation.

V. FSP Implementation

This section shows the implementation of FSP, i.e., the choices we made to generate the OpenCL code of the device program as well as of the generated API to interact with the FPGA pipeline through a host C/C++ application.

Each operator is an active running component on the FPGA whose logic is split into three phases: *1)* a `begin` phase in which the operator initializes its internal state (if any); *2)* a `compute` phase in which the operator processes each input received from its incoming FIFO queues by running the business logic code provided by the FSP user for that operator; *3)* and the `end` phase, which can be used to finalize the operator state. The three phases are functions that have complete access to all the buffers declared for that operator.

A. Operator Replication

As said in § II, operators can be replicated to increase their processing throughput. Each replica runs the same business logic of the operator on a subset of inputs distributed by the preceding operator of the pipeline.

In the FSP generated device code, each operator replica is implemented as an OpenCL SWI kernel. The structure of a kernel is shown in Listing 5 in which each function represents a component that is generated according to the DSL. The first two attributes instruct the offline compiler to infer the SWI model. As a result, the offline compiler reduces the logic utilization needed for NDRange kernels and applies additional performance optimizations. Before the main loop, the kernel calls the `begin` phase function of the operator. Inside the main loop, a generated *gather* component applies the gather policy to receive inputs from the replicas of the preceding operator of the pipeline. This component is also in charge of updating the `done` variable to `true` once the stream ends. After an input has been received, the `compute` phase function is called to process it. At the end of the main loop, the *dispatch component* sends the computed result to the next operator replica(s) according to the dispatch policy chosen by the programmer through the DSL. Once the EoS signal arrives, the kernel calls the `end` phase function and propagates the EoS signal to each of its destinations. The EoS is a mechanism to signal that the stream is closed and the receiving operator replicas can terminate its execution.

```
__attribute__((uses_global_work_offset(0)))
__attribute__((max_global_work_dim(0)))
__kernel void op_kernel(...) {
  op_begin();
  while (!done) {
    gather_input();
    op_compute();
    dispatch_result();
  }
  op_end();
  propagate_EoS();
}
```

Listing 5. Stucture of a kernel running an operator replica.

The logic of an operator needs to be replicated in the device code based on the number of replicas specified for the operator (i.e., its parallelism degree). To implement this, two main strategies can be followed:

- replicas of the same operator can be implemented by generating the code of a unique SWI kernel composed by multiple *compute units*, one per replica;
- the whole kernel code can be replicated during the code generation phase in order to have one full definition of the kernel for each replica (each having a unique name in the code and the same implementation).

Although the first approach is quite natural in OpenCL programming, and the offline compiler implements each compute unit as a unique pipeline, only *autorun kernels* can be used to exploit such a feature. Autorun kernels run on their own without any host interaction and receive no input arguments from the host. This is a limiting factor since they have no access to global memory buffers and cannot receive any information from the host program. For this reason, FSP follows the second approach to replicate operators.

B. Inter-operator Data Distribution

Let us consider the case of two consecutive operators, the left-hand side operator (LHS) and the right-hand side one (RHS). Each replica of LHS is connected to each of the RHS replicas by dedicated Intel Channels FIFO queues. A datatype (see Listing 6) is generated in the device code to wrap the user datatype with information used by FSP.

The struct name is generated as the concatenation of the two operator names and contains the data type of the output mes-

sage produced by LHS and delivered to LHS (`data_type`), and a boolean flag that carries the EoS signal.

```
typedef struct {
  data_type data;
  bool EoS;
} <LHS>_<RHS>_t;
```

Listing 6. Datatype wrapping a user-defined datatype exchanged between two consecutive operators.

Each of the $M > 0$ RHS replicas receives inputs from the $N > 0$ previous LHS replicas according to a *blocking* or a *non-blocking* gathering policies. Listing 7 shows the RHS operator replica adopting the latter policy using the non-blocking `read` functions to poll each input channel coming from all the LHS replicas. The use of non-blocking `reads` allows the operator replica to poll one channel at each loop iteration. This implementation is useful to avoid kernel stalls because the polling phase keeps going even when no data is present in some channels. We also support a *blocking* gathering policy, which uses blocking `read` functions. In this case, the `read` call can cause the kernel to stall until at least one element becomes available in that specific channel.

```
channel <LHS>_<RHS>_t ch[N][M];

__kernel void operator_<RHS>(...) {
  uint id = 0;
  uint r = 0;
  while (!done) {
    <LHS>_<RHS>_t in;
    bool v = false;      // valid flag
    switch (r) {
      case 0:   in = read_channel_nb_intel(ch[  0][id], &v); break;
      case 1:   in = read_channel_nb_intel(ch[  1][id], &v); break;
      ...
      case N-1: in = raed_channel_nb_intel(ch[N-1][id], &v); break;
    }
    r = (r + 1) % N;
    if (v) {             // check valid flag
      op_compute(in);
    }
  }
}
```

Listing 7. Non-blocking gathering implementation.

Listing 8 shows the implementation of the key-by dispatching policy. The destination to deliver a new output result depends on the value of the key obtained by calling a user-provided function (e.g., `get_key()`), modulo the number of the next operator replicas. Doing so, outputs with the same key are sent to the same replica of the next operator.

```
channel <LHS>_<RHS>_t ch[N][M];

__kernel void operator_<LHS>(...) {
  uint id = 0;
  while (!done) {
    gather_input();
    <LHS>_<RHS>_t out = op_compute();
    const uint w = result_get_key(out.data) % M;
    switch (w) {
      case 0:   write_channel_intel(ch[id][  0], out); break;
      case 1:   write_channel_intel(ch[id][  1], out); break;
      ...
      case M-1: write_channel_intel(ch[id][M-1], out); break;
    }
  }
}
```

Listing 8. Key-by dispatching policy implementation.

C. H2D and D2H Data Transfers

FPGAs can improve the performance of streaming applications by accelerating the computation of operator pipelines. However, the performance gain can be nullified by the overhead of H2D and D2H data transfers. Therefore, FSP supports several techniques to mitigate the communication overhead, including data batching, the K-buffering optimization, and a custom protocol designed for SoCs with shared memory between CPU and FPGA.

1) Data batching: transferring inputs one by one to the FPGA can result in low communication bandwidth, which may drastically reduce the application throughput. Therefore, FSP adopts a strategy that groups inputs together to form batches and transfers them to the FPGA as a whole. The batch size is a user-configurable parameter exposed by the host API for both the H2D and D2H communications. If the batch size is set to one, once a new input arrives it is immediately transferred to the FPGA to be processed. By setting the batch size to be greater than one, inputs are buffered and transferred once the batch has been entirely filled. The batch size obviously impacts the application performance. Small batches (a few tens of inputs) can lower the computational latency but can drastically reduce throughput because of the communication overhead. Large-sized batches (e.g., thousands of inputs) often increase throughput at the expense of latency.

2) K-buffering: when a kernel is executed multiple times over different input batches, batch transfers occur between successive kernel 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, FSP employs a generalization of the well-known 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 kernel is computing the previous one. This can be done by using separated buffers (one per batch), or a single properly aligned buffer large enough to host K distinct batches.

3) Shared-memory protocol: on SoC CPU+FPGA devices, read and write operations from the CPU to the FPGA DDR memory (i.e., memory accessible only by the FPGA) are often very slow because they do not use Direct Memory Access (DMA) hardware capabilities. However, such devices are often equipped with a physically shared memory that can be simultaneously accessed by both CPU and FPGA. When this memory is used, OpenCL memory calls are done as zero-copy transfers for buffer reads, writes, maps, and unmaps. Therefore, we design a shared-memory synchronization protocol to take full advantage of this feature. This protocol adopts two shared buffers, allocated once at the beginning of the application to store $K > 1$ batches (to implement the K-buffering technique previously described) and their corresponding headers. A header is a 32-bit unsigned integer that contains information about the number of inputs contained in a batch, a flag to signal the EoS, and a `ready` flag to synchronize the access to the shared memory space.

Let us consider the example of an H2D communication in which a thread on the host CPU pushes data to a Source operator running on the FPGA. The host thread performs active waiting on the `ready` flag of the header through a spin

loop. Once the `ready` flag is set to `false` by the Source operator, the host obtains access to the corresponding batch and can fill it with arriving inputs of type `input_t`. Then, it executes a memory barrier to ensure that all the writes are completed and updates the header by setting the number of inputs in the batch, the `ready` flag to `true`, and the `QoS` flag to `true` if the stream is ended, `false` otherwise.

On the device side, the Source operator waits until the value of the `ready` flag goes from `false` to `true`, then reads and dispatches the inputs of the batch to the next operator according to its dispatching policy, and after a memory barrier, it resets the header again (Listing 9).

```
__kernel source(__global volatile header_t *restrict headers,
                __global const volatile input_t *restrict data) {
  uint idx = 0;
  while (!done) {
    header_t h;
    while (!header_ready(h = headers[idx]);
    const uint batch_size = header_size(h);
    for (uint i = 0; i < batch_size; ++i) {
      dispatch_inputs();
    }
    done = header_close(h);
    mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);
    header_reset(headers[idx]);
    idx = (idx + 1) % N;
  }
}
```

Listing 9. Source implementing the shared-memory protocol.

VI. Experiments

We evaluate FSP on the Han Pilot Platform SoC shipped by Terasic, depicted in Fig. 2. The SoC includes an Intel Arria 10 FPGA and a 1.5GHz dual-core ARM Cortex-A9 processor with 1GB on-board DDR4-2400 with 32-bit data width (shared with the FPGA), running Angstrom GNU/Linux v2014.12. The Intel Arria 10 FPGA includes 660K Logic Elements, 250K Adaptive Logic Modules (ALM), 1M Registers, 42,620 M20K, 5,788 MLAB, and 1,687 DSPs, and access 1GB on-board DDR4-2400 with 32-bit data width (FPGA DDR). Host programs are compiled with the `arm-linux-gnueabihf-g++` cross-compiler with `-O3` optimizer flag. To compile device programs, we use the Intel FPGA SDK for OpenCL Offline Compiler `aoc` Version 19.1 Build 240 Pro Edition, compiling with `-g0` flag, and the `-board=a10s_ddr` flag that instructs the compiler to use the HAN Pilot Platform OpenCL Board Support Package 19.1.

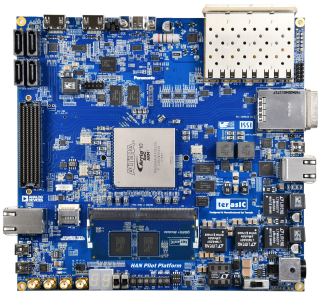

Fig. 2. Han Pilot Platform SoC by Terasic.

We have chosen two applications in DSPBench [2]. The first application, SpikeDetection (SD), processes a stream of sensor readings to monitor spikes. It is a pipeline of four operators: a Source, a stateful Map computing a moving average, a stateless Filter evaluating if the current sensor reading is a spike, and a Sink. We implemented the Map by keeping key-partitioned windows in local memory (BRAM), and it computes the moving average by using the shift register pattern [6]. The Filter is implemented with a simple Boolean predicate.

The second application, FraudDetection (FD), processes a stream of credit card transactions to detect fraud. It is composed of a pipeline of four operators: a Source, a stateful Map applying a Markov model to calculate the probability of fraud, a Filter evaluating if the current transaction is a fraud, and a Sink. We implemented the Map operator such that, for each input, it reads the Markov model (a matrix of floating-point probabilities) from global memory (FPGA DDR) to increase the granularity of the computation.

A. Shared-memory Protocol Evaluation

To assess the effectiveness of the shared-memory protocol, we consider three different data transfer methods:

- `Copy`: the host allocates buffers on the FPGA memory and transfers data from the host buffers to the FPGA buffers with the `clEnqueueWriteBuffer()` and back using the `clEnqueueReadBuffer()` OpenCL functions. A Source kernel is launched after transferring each new batch, to start reading its inputs and distributing them to the next FPGA operator of the pipeline. The Sink kernel is instead launched at the beginning, to start filling the first batch of results, and then re-started each time a new output batch is consumed by the host application;
- `Hybrid`: buffers directly accessible by host and FPGA are allocated in shared memory. Kernels are launched as for `Copy`, but the host employs memory barriers to ensure that all writes are completed before launching each Source kernel and reads are complete before launching each Sink kernel. So, with this approach, we avoid the overhead of additional OpenCL copies, but we still pay the cost of launching Source/Sink kernels;
- `Shared`: buffers are allocated in shared memory and managed with the shared-memory protocol in V-C3. Source and Sink kernels are launched once at the beginning of the application and they run until the EoS signal is received from the host.

SD is a fine-grained application. To exploit the H2D bandwidth, we evaluate SD with one and two replicas for the Source, while we fix all the other operators to have one replica only. The host program spawns a dedicated thread responsible for writing batches read by each Source on the FPGA (so we can have one or two of these threads), and a thread reading output data batches from the Sink operator on FPGA. Fig. 3 shows the impact of the data transfer methods, the Source parallelism degree (SP), and the Sink batch size ($BSize_{sink}$) on the overall application throughput. In each case, we report the throughput by varying the Source batch size ($BSize_{source}$) from 16 inputs to $4,096$ inputs per batch.

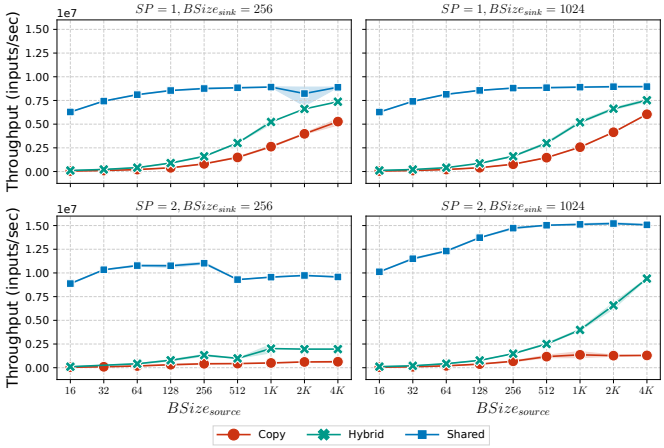

Fig. 3. Impact of the transfer methods with different configurations of Source Parallelism (SP), and the Sink Batch Size ($BSize_{sink}$) on throughput (inputs/sec) for the SD application.

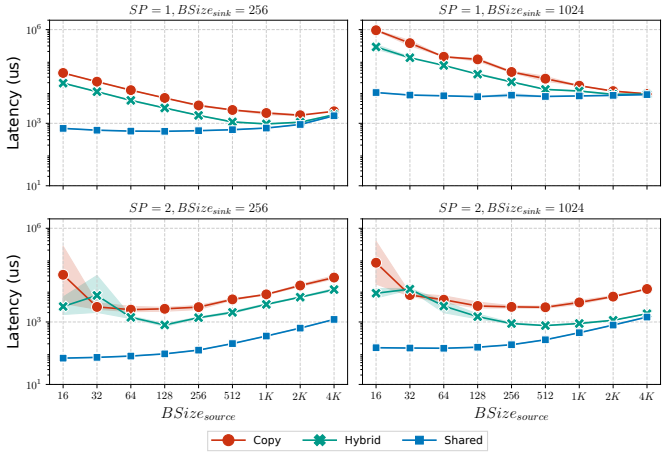

Fig. 4. Impact of the transfer methods with different configurations of Source Parallelism (SP), and the Sink Batch Size ($BSize_{sink}$) on latency (milliseconds) for the SD application.

With one Source replica and with the two considered $BSize_{sink}$ values (256 and $1,024$), the maximum throughput is of 8.9M inputs/sec achieved by `Shared` with a $BSize_{source}$ of $4,096$, which is $1.6\times$ and $1.2\times$ higher than the maximum throughput with `Copy` and `Hybrid`.

With small $BSize_{source}$ (< 128 inputs), `Shared` obtains $39\times$ to $117\times$ higher throughput than `Copy` and $19\times$ to $56\times$ than `Hybrid`. With larger $BSize_{source}$, this gap is reduced but `Shared` is still the better transfer method as it achieves up to $20\times$ higher throughput than `Copy` and `Hybrid`.

With two Source replicas on FPGA, the application is composed of a total of three host threads that compete for the two cores of the ARM CPU. The `Copy` method is negatively affected by thread concurrency in all scenarios, while the throughput using `Hybrid` is reduced up to 73% with $BSize_{sink}$ of 256, and up to 22% with $BSize_{source}$ of $1,024$. The `Shared` method shows an improvement from 10% to 40% with $BSize_{sink}$ of 256, and $50 - 70\%$ with $BSize_{sink}$ of $1,024$. The $BSize_{sink}$ has a positive impact on the throughput of the application using the `Shared` method, gaining from 11% to 62% with the larger $BSize_{sink}$, reaching a peak of 15.2M inputs/sec in the best case.

Fig. 4 shows the impact of the transfer methods, the Source parallelism degree, and the $BSize_{sink}$ on the latency. With one Source replica, increasing the $BSize_{sink}$ leads to a higher latency because it needs more time to fill the batch of output results. The latency with the `Shared` method is under the millisecond with $BSize_{source} < 2,048$ inputs. With small $BSize_{source}$ (< 64), `Shared` outperforms `Copy` by $20\times$ to $59\times$ and $17\times$ to $98\times$ with $BSize_{sink}$ of 256 and $1,024$ respectively, and $9\times$ to $28\times$ `Hybrid` with both $BSize_{sink}$ values, on average. The gap between the three methods is reduced by using large $BSize_{source}$ as the overheads of the `Copy` and `Hybrid` methods became negligible.

By employing two Source replicas, the input rate of the application doubles and thus improving the latency in all configurations. Despite the latency by using the `Copy` and `Hybrid` methods being under 10 milliseconds for 32, 64, and 128 values of the $BSize_{source}$, both methods are outperformed by `Shared`, which shows latencies under 100 microseconds in the same configurations. In particular, the `Copy` method, on average, registers $27\times$ and $20\times$ higher latency compared to `Shared` with $BSize_{sink}$ of 256 and $1,024$, respectively. The `Hybrid` method performs better with $BSize_{source} > 64$ compared to the `Copy` method. However, `Shared` obtains $2\times$ lower latency on average, even with large $BSize_{source}$ (> 512) compared to the `Hybrid` method.

B. Replication Evaluation

We use the FD application to show the impact of replication since it exposes a more coarse-grained computation. Fig. 5 shows the throughput resulting from varying the number of replicas of the Map operator under several $BSize_{source}$ configurations. In these experiments, we run the application by fixing the $BSize_{sink}$ to 32 and using the `Shared` transfer method. The baseline is the application with parallelism set to one for each operator on FPGA, which reaches a maximum throughput of 1.2M inputs/sec with batches greater than 32 inputs. With small input batches (< 64), the replication of the Map operator has a low impact on throughput. However, by increasing the $BSize_{source}$, and thus the input rate, we obtained $1.69\times$, $3.29\times$, and $3.97\times$ of peak throughput with 2, 4, and 6 replicas of the Map operator, respectively, which is the most compute-intensive operator of the pipeline.

Operator replication can have a significant impact on latency too, as shown in Fig. 6. With a $BSize_{source}$ of 16, we obtain a minimum latency of 71 microseconds by using 4 and 6 Map replicas, which is $2\times$ lower than using one replica only. With larger $BSize_{source}$, the baseline experiences from $2.29\times$ to $3.76\times$ more latency than using 4 and 6 Map replicas.

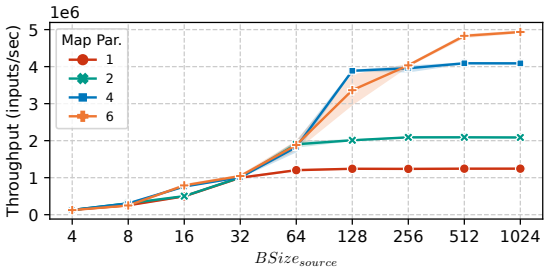

Fig. 5. Impact of replication on throughput (inputs/sec) of the FD application.

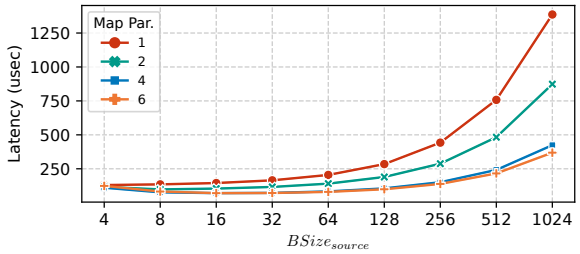

Fig. 6. Impact of replication on latency (microseconds) in the FD application.

VII. Related Works

DSP on heterogeneous resources is a promising research direction to speed up streaming tasks with an effective performance-to-energy ratio. Although some papers have investigated how to support GPUs for relational stream processing [8], [9], the space of FPGA solutions is still small.

The most complete FPGA-oriented framework for DSP is Glacier [10] to support relational SQL streaming queries. Although not comparable with FSP, because we target non-relational computations, Glacier is no longer maintained and is hard to be smoothly ported on modern FPGA resources. Another work is F-Storm [11], a version of Apache Storm to support FPGA. The approach is interesting, although it does not support the offloading of whole pipelines like FSP does.

Other research papers have focused on the FPGA acceleration of specific well-known relational streaming operators. The work described in [12] presents an efficient and scalable FPGA-based acceleration of sliding-window aggregates. The implementation proposed in that paper applies a combination of previously studied techniques such as window panes [13]. This technique is used to avoid replicating a large number of aggregation modules for overlapping sliding windows by dividing each window into non-overlapping sub-windows called panes. The implementation has been evaluated using a Xilinx FPGA and directly developed using VHDL. Our FSP currently supports basic general-purpose streaming operators only. Extensions to include window-based operators are ongoing.

Other accelerated operators are joins, a computationally expensive class of stateful operators. An FPGA implementation has been proposed in [14] with the name of StreamJoin targeting the Convery HC-2ex hybrid computer featuring an Intel CPU and a Xilinx FPGA. A generic high-throughput architecture for DSP targeting FPGAs has been proposed in [15]. It is oriented to data stream mining problems but actually covers join operators and their porting on FPGAs.

VIII. Conclusions and Future Works

This paper presented FSP, a programming framework to develop DSP applications exploiting FPGAs. The approach generates code in OpenCL starting from a pipeline description provided in Python. The generated code adopts several optimizations for efficient kernel-to-kernel communications and to optimize H2D and D2H data transfers. The experiments show interesting results in terms of latency and throughput. In the future, we plan to extend FSP to support FlatMap and window-based operators. Furthermore, the code generation approach can be extended to support Xilinx FPGAs.

References

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. USA: Cambridge University Press, 2014.

[2] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.

[3] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, p. 121–142, jun 2006. [Online]. Available: https://doi.org/10.1007/s00778-004-0147-z

[4] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Comput. Surv.*, vol. 52, no. 2, apr 2019. [Online]. Available: https://doi.org/10.1145/3303849

[5] OpenCL™. [Online]. Available: https://www.khronos.org/opencl/

[6] Intel. Intel FPGA SDK for OpenCL Software Tecnology. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

[7] Jinja 2. [Online]. Available: https://jinja.palletsprojects.com

[8] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "Lightsaber: Efficient window aggregation on multi-core processors," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2505–2521. [Online]. Available: https://doi.org/10.1145/3318464.3389753

[9] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, *FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures*. USA: USENIX Association, 2020.

[10] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: A query compiler for fpgas," *Proc. VLDB Endow.*, vol. 2, no. 1, p. 229–240, aug 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687654

[11] S. Wu, D. Hu, S. Ibrahim, H. Jin, J. Xiao, F. Chen, and H. Liu, "When fpga-accelerator meets stream data processing in the edge," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1818–1829.

[12] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on fpga," in *2013 First International Symposium on Computing and Networking*, 2013, pp. 112–121.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Rec.*, vol. 34, no. 1, p. 39–44, mar 2005. [Online]. Available: https://doi.org/10.1145/1058150.1058158

[14] C. Kritikakis, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos, "An fpga-based high-throughput stream join architecture," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[15] C. Rousopoulos, E. Karandeinos, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos, "A generic high throughput architecture for stream processing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–5.