FLOWER: A Comprehensive Dataflow Compiler for High-Level Synthesis

Puya Amiri∗, Arsène Pérard-Gayot§, Richard Membarth†∗, Philipp Slusallek∗§, Roland Leiβa†, Sebastian Hack§

∗German Research Center for Artificial Intelligence (DFKI), Germany
†Technische Hochschule Ingolstadt (THI), Research Institute Almotion Bavaria, Germany
‡University of Mannheim (UMA), Germany
§Saarland University (UdS), Germany

Abstract—FPGAs have found their way into data centers as accelerator cards, making reconfigurable computing more accessible for high-performance applications. At the same time, new high-level synthesis compilers like Xilinx Vivis and runtime libraries such as XRT attract software programmers into the reconfigurable domain. While software programmers are familiar with task-level and data-parallel programming, FPGAs often require different types of parallelism. For example, data-driven parallelism is mandatory to obtain satisfactory hardware designs for pipelined dataflow architectures. However, software programmers are often not acquainted with dataflow architectures—resulting in poor hardware designs.

In this work we present FLOWER, a comprehensive compiler infrastructure that provides automatic canonical transformations for high-level synthesis from a domain-specific library. This allows programmers to focus on algorithm implementations rather than low-level optimizations for dataflow architectures. We show that FLOWER allows to synthesize efficient implementations for high-performance streaming applications targeting System-on-Chip and FPGA accelerator cards, in the context of image processing and computer vision.

Index Terms—high-level synthesis, dataflow, compiler, FPGA, transformations, high-performance computing

I. INTRODUCTION

Although Dennard scaling has broken down some time ago, it is generally assumed that Moore’s law will continue to hold for at least a few years. As a consequence, hardware vendors have built more and more specialized as well as parallel hardware such as multi-core CPUs, GPUs, or FPGAs. Since FPGAs are low-power, reconfigurable and highly parallel integrated circuits, they have already been extensively adopted in embedded systems and more recently have found their way into scientific high-performance computing (HPC).

Akin to languages for GPU computing such as CUDA or OpenCL, FPGA manufacturers offer various vendor-specific dialects of C/C++ that allow software developers to program at a high level of abstraction. So-called high-level synthesis (HLS) compilers translate untimed, C-based dialects down into a timed, high-performance, register-transfer level (RTL) language in dataflow style. These HLS languages entail two major drawbacks: First, each dialect is closely tied to its vendor which makes code incompatible between different HLS languages. Second, albeit HLS languages are typically C-based, they still require a hardware design mentality to be fully taken advantage of. For example, transforming an untimed language into an RTL language calls for several transformations [1] with different levels of compilation and hardware synthesis flows. These transformations have a different structure, depending on whether the input language is Xilinx C++ for HLS [1], or Xilinx and Intel OpenCL [2].

To sum up, FPGA vendors offer competing and incompatible HLS solutions. For this reason, programmers must rewrite the application for each of those solutions. Moreover, hardware and software interaction methods and optimizations are different among vendors and HLS dialects.

Most FPGA applications require some level of parallelism or concurrency to achieve the best performance, particularly for memory accesses. HLS programming allows to express such parallelism through dataflow regions. To take advantage of this, developers have to manually separate their application into tasks, and manually write all the necessary glue code to transform these separate tasks into a well-formed application. This process is notoriously difficult, because it requires a lot of effort and knowledge of the application.

An alternative to HLS languages are domain-specific languages or libraries (DSLs): By embedding the knowledge of a particular domain into a language, the compiler or library automatically applies efficient coding patterns, data movement mechanisms [3], or spatial designs.

Contributions: This paper introduces FLOWER, a framework for FPGA development that makes the following contributions:

- FLOWER provides a high-level syntax that helps in the design of dataflow-oriented FPGA applications—in particular by encouraging the separation between the core algorithm and data transfers. This simplifies low-level optimizations like vectorization or burst transfers.
- FLOWER automatically generates kernels that combine dataflow-oriented tasks from the dataflow graph of the application (see Section IV-B). While FLOWER’s main target is HLS for Xilinx FPGAs, it can also generate multi-target OpenCL kernels which are optimized for both Intel and Xilinx FPGAs.
- FLOWER automatically generates host-code from a single piece of code that describes the entire application (see Section IV-C).

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the HorME, HP-DLF, MetaDL, and REACT projects.
We show the applicability of our framework on image processing and computer vision applications, where our framework has comparable or even better performance than alternatives (see Section V).

II. Background

A. Stages, Kernels and Tasks

Typical streaming applications consist of several stages. For instance, Table I shows a list of applications with their respective number of stages. Considering a directed acyclic graph (DAG), each node of a DAG represents a stage. In practice, using HLS, these stages are mapped to kernels and tasks.

A kernel is a function that is scheduled and controlled from the host code, and not from within the FPGA design. On the contrary, a task is a function that is statically scheduled for execution from within the FPGA design, and not from the host. A kernel may contain one or more tasks.

Suppose we have a multi-stage application and its FPGA design consists of a single kernel: HLS tools will then apply a static model that will schedule every operation inside it. After synthesis, different segments of the resulting hardware run in lockstep with each other, and cannot run concurrently. While this coding style is simpler, it is not well-designed because dependencies or variable latency operations may introduce stalls.

A better approach is to apply a dataflow transformation that uses queues to transfer data between each task and enables task-parallelism. With that approach, HLS tools will then generate a kernel that has a latency equal to the latency of the task with the highest latency. This is in contrast to the previous approach, where the kernel had a latency equal to the sum of the latencies of each individual task. Fig. 1 shows the effect when applying dataflow optimization on a kernel that consists of five tasks.

The HLS compiler internally uses a Finite State Machine (FSM) to schedule individual parts of the kernel that does not use the dataflow transformation. When an expensive operation is running, this FSM waits for its completion. Hence, all other components of the kernel are in idle mode. In cases where the kernel needs a significant amount of data, it may happen that the FPGA does not have enough BRAM to buffer them all, which means that the FPGA design may not function properly. Moreover, such a kernel may need to access global memory with sporadic patterns, which may decrease the efficiency of the DMA engine.

In contrast to this, the dataflow-optimized kernel is made of several small tasks, which allows the HLS compiler to schedule each one independently, and generate one FSM per task. This means that tasks have their own independent controllers, connected via FIFO buffers; the buffering requirements get distributed among the tasks. As a result, when a task stalls in a clock cycle, other tasks continue running as long as there is enough data in their input buffers, resulting in a higher overall throughput. The dataflow transformation also has a significant impact on physical synthesis: Shortening the critical paths allows the design to run at a higher clock frequency. What is more, it benefits the fan-out of control signals.

B. AnyHLS and FLOWER

The work in this paper is built upon AnyHLS [4], a framework for FPGA application development that is itself built upon AnyDSL [5]. AnyHLS introduces high-level abstractions to design FPGA applications, and extends the AnyDSL compiler infrastructure to generate FPGA designs for Intel OpenCL and Xilinx HLS. For this, the syntax of AnyDSL is extended with additions for FPGA programming. The image processing applications in AnyHLS are written in a library that builds on top of these changes. This library allows programmers to develop point, local, and global image processing operators with very little effort. In this paper, we focus on addressing the shortcomings of AnyHLS listed below, in particular with the automation of host code generation and dataflow optimizations.

AnyHLS provides a way to abstract typical patterns found in high-level synthesis in the form of a library with the help of the partial evaluator provided by AnyDSL. These abstractions work well for single-kernel and single-task applications. However, AnyHLS is limited to generate disjoint kernels in the form of IP-blocks without system integration, task-level pipelining, dataflow optimization, host-code generation, or memory optimizations such as burst transfers. In fact, AnyHLS can only generate disjoint IPs from multi-stage applications, which then need manual wiring to connect them together to achieve a sequential execution. In order to drive the design, the user has to write a corresponding host-side code for each application. In FLOWER, we rely on AnyHLS abstractions to describe applications, and extend both AnyHLS and AnyDSL to support multi-stage applications by mapping them to different tasks and enable dataflow optimizations. For this, FLOWER is deeply integrated into the AnyDSL compiler in order to apply task-level optimization and transformations. We extend the AnyDSL compiler in order to extract the dataflow graph from application stages described by the user program and produce optimized dataflow pipelines according to the producer-consumer dependencies. Unlike AnyHLS, our
toolchain automates the whole design process from program-
ing to synthesis.

III. MOTIVATION

Software programming is very different from hardware de-
sign since traditional software design methods are not adapted
to execution on FPGAs. HLS tools have been introduced to
to help bridge that gap. However, there are still areas where HLS
does not help in the process of designing efficient hardware:

A. Dataflow Transformations

Vitis requires a canonical dataflow form to realize an archi-
tecture that takes advantage of task-pipelining and decreases
redundant host-kernel communication. This particular coding
pattern is not only alien to software programmers, but it
also demands a specific set of canonical rules which are
difficult to apply, tedious, and may distract the programmer
from implementing the actual algorithm. This particularly
applies for deep learning, image processing, and machine
vision applications, where stages of a kernel need to be split
into many tasks so that they can run concurrently or in parallel,
and where a final kernel calls each task in the order dictated
by the dataflow. This is tedious and can be automated, since
the order is in any case fixed by the dataflow.

B. Low-level Optimizations

With FPGA hardware, unlike CPU architectures, the notion
of a hierarchical memory that is transparent to the programmer
does not exist. Instead, there are plenty of BRAMs available on
FPGA fabric that the HLS programmer must explicitly use to
improve overall design performance and also increase global
memory access efficiency [6]. Considering that global memory
access is taking a significant amount of time compared to ker-
nel execution, missing a caching system becomes an essential
problem. In order to decrease this overhead and to exploit
BRAMs as a cache, a batch process strategy must be used,
typically with burst memory transfers. Using burst memory
transfers allows for minimizing the amount of control signal
transactions and for merging several memory access requests
into a single request. This optimization greatly maximizes the
application throughput and decreases global memory access
latency. Sadly, taking advantage of burst transfers requires to
perform the dataflow transformation first, with all the above
problems.

Another important low-level optimization for FPGA designs
is vectorization: One goal of this optimization is to widen the
bitwidth of the inputs of the kernel, so that many elements
can be loaded at the same time and processed in parallel. By
this, vectorization increases throughput and memory efficiency.
In order to vectorize the code, the HLS compiler requires
consecutive memory access indices and several copies of the
computation of interest. Writing entire applications in this style
is profoundly complex and error-prone. In order to fully benefit
from this optimization, the increase of the input’s bitwidth
should be matched by an appropriate number of units that
process the data. This will become substantially difficult if
the application is not tailored for that.

Another important low-level optimization for FPGA designs
is vectorization: One goal of this optimization is to widen the
bitwidth of the inputs of the kernel, so that many elements
can be loaded at the same time and processed in parallel. By
this, vectorization increases throughput and memory efficiency.
In order to vectorize the code, the HLS compiler requires
consecutive memory access indices and several copies of the
computation of interest. Writing entire applications in this style
is profoundly complex and error-prone. In order to fully benefit
from this optimization, the increase of the input’s bitwidth
should be matched by an appropriate number of units that
process the data. This will become substantially difficult if
the application is not tailored for that.

Another important low-level optimization for FPGA designs
is vectorization: One goal of this optimization is to widen the
bitwidth of the inputs of the kernel, so that many elements

This code uses the image processing DSL of AnyHLS [4]. DSL functions are highlighted in green. Within FLOWER, each of these functions creates a task, resulting in 4 different tasks. The function `split_image` creates the first task, reads the input image `in_img` from memory, and writes to two different virtual images (images that are mapped to channels). Those virtual images are then used in two different point operators `fun1` and `fun2`, and the results are passed back to two more virtual images `tmp_img1` and `tmp_img2`. Finally, a binary point operator `fun3` is applied to the result of the two previous tasks, and the final result is written back to global memory.

The DSL functions all use the constant `vector_length` internally, so that the resulting kernel is vectorized with burst transfers. FLOWER achieves this by unrolling the computation within the loop body:

```cpp
@iteration_point(output: Img, input: Img,
    body: fn(132, 132, Img) -> () -> () {
    /* ... */
    for v in unroll(0, vector_length) {
        body(); /* ... */
    } /* ... */
}
```

This results in several copies of the for-loop body. The HLS compiler is then able to determine the parts that can execute in parallel, resulting in the code being vectorized.

A. Dataflow Graph Extraction

From this example, FLOWER extracts a dataflow graph. Each task represents a node in that graph and each channel is mapped to an edge. FLOWER detects invalid graphs and emits error messages if applicable. In particular, it checks that the graph is acyclic and channels are written to or read from (incoming edges) or written to (outgoing edges).

B. Top-level Kernel Generation

Our scheduling algorithm generates an HLS kernel that combines all the tasks of the application. In the remainder of this text, we will refer to this kernel as the top-level kernel. For the HLS compiler to allow for concurrent or parallel execution of the tasks in the top-level kernel, FLOWER performs a topological sort of the graph in order to ensure that any task first writes to a channel before any tasks reads from that channel. As a side note, this scheduling algorithm also works with tasks that are isolated from the rest of the graph. Such tasks execute in parallel with the rest. Fig. 3 illustrates how the control flow changes between host/device code by introducing scheduling kernels as tasks (b).

![Fig. 3: Control flow before top-level generation (a) and after scheduling kernels as tasks (b).](image_url)
concepts are the same): generates the host code as LLVM IR, not C++, but the host code will be automatically generated (our framework instance, for the application above, the following equivalent required to launch the kernel and communicate with it. For the generated code contains the necessary XRT API calls Xilinx. Our framework generates such host code automatically. code, that is typically written with the XRT API provided by interface pragmas for different target platforms. However, this setting, we need to interface it with a host. FLOWER generates such host code automatically. FLOWER places calls of these tasks in topological order as discussed previously and tells the HLS compiler via #pragma HLS DATAFLOW of a dataflow region. Consequently, this structure results in a design in which all tasks are pipelined and execute concurrently.

While this example plainly introduces the fundamental functionality of our toolchain, FLOWER is not limited to that. Figure 4 introduces another example. It demonstrates a more complicated dataflow graph that implements the Lucas-Kanade method for optical flow estimation. Black nodes are not part of algorithm, they specify inputs and outputs and reside on the host-side. Splitting nodes are not shown for the sake of simplicity. Since there are parallel paths from inputs to outputs, a single memory interface cannot feed the tasks concurrently. Squared nodes named mem1−4 solve this issue. FLOWER designates 4 different memory bundles using interface pragmas to separate memory transactions. Assigning individual memory interfaces avoid congestion in host-device memory transfers.

C. Hardware/Software Interface

In order to use the generated FPGA design in a practical setting, we need to interface it with a host. FLOWER generates interface pragmas for different target platforms. However, this is not sufficient because the HLS code on its own does not specify how to communicate data from or to the host system. In order to do that, the HLS code has to be driven by a host code, that is typically written with the XRT API provided by Xilinx. Our framework generates such host code automatically. The generated code contains the necessary XRT API calls required to launch the kernel and communicate with it. For instance, for the application above, the following equivalent host code will be automatically generated (our framework generates the host code as LLVM IR, not C++, but the concepts are the same):

```c
int4_chan* chan4 = &chan4_slot;
#pragma HLS STREAM variable = chan1 depth = 2
#pragma HLS STREAM variable = chan2 depth = 2
#pragma HLS STREAM variable = chan3 depth = 2
#pragma HLS STREAM variable = chan4 depth = 2

task1(input_data, chan1, chan2);
task2(chan1, chan3);
task3(chan2, chan4);
task4(output_data, chan3, chan4);
```

Note that this code uses channels of type int4, since we have a vectorization factor of 4. FLOWER generates separate tasks as separate functions. The tasks task1 and task4 have a parameter that allows them to access global memory. For the same reason the top-level kernel hls_top expects two parameters: input_data and output_data. FLOWER annotates these parameters with pragmas to instruct the underlying HLS compiler to give them an AXI interface to connect to other peripherals. FLOWER defines the 4 channels chan1 to chan4 as FIFO channels to communicate data between tasks (using the #pragma HLS STREAM annotation). Finally, we see how FLOWER places calls of these tasks in topological order as discussed previously and tells the HLS compiler via #pragma HLS DATAFLOW of a dataflow region. Consequently, this structure results in a design in which all tasks are pipelined and execute concurrently.

Concisely, this code sets up the basic infrastructure to load the kernel, creates the OpenCL/XRT context and command queue, then creates buffers to hold the input and output data, loads the image, runs the kernel, and finally writes back the result image. In order to generate that code, FLOWER considers every loop that comes from the DSL (for instance, the loops created via iteration_point) as executing on the FPGA. Thus, these parts are translated into a single launch of the top-level kernel. The rest is considered as running on the host: The calls to read_image or write_image, for instance, will be executed there, and not on the FPGA. Internally, functions like write_image or read_image use compiler-provided intrinsics to copy data from the host to the FPGA: Those directly translate to calls to XRT that transfer the data in the right direction.

As mentioned previously, all the loops that are generated via the DSL translate into one top-level kernel launch on the host. The arguments of that kernel launch are set according to the parameters extracted during the top-level kernel generation.

```c
int4_chan* chan4 = &chan4_slot;
#pragma HLS STREAM variable = chan1 depth = 2
#pragma HLS STREAM variable = chan2 depth = 2
#pragma HLS STREAM variable = chan3 depth = 2
#pragma HLS STREAM variable = chan4 depth = 2

task1(input_data, chan1, chan2);
task2(chan1, chan3);
task3(chan2, chan4);
task4(output_data, chan3, chan4);
```

![Data flow graph for the Lucas-Kanade implementation](image)
phase. Those typically come from uses of input or output images in the DSL loops, like in this example.

Thanks to that automatic host code generation, the programmer only needs to focus on writing the application from a single piece of code written using FLOWER. Consequently, it’s easier to make modifications of the code, since the host code is automatically synchronized with the FPGA code.

V. EVALUATION

For experimental evaluation, we consider a range of prominent applications that have been used in comparable works [7], [8], [9], [10]. Table I lists the number of stages of each application. This number does not include two additional memory read/write stages required for burst transfer optimization.

TABLE I: Benchmarking applications.

<table>
<thead>
<tr>
<th>application</th>
<th>stage(s)</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean filter</td>
<td>1</td>
<td>5 × 5 filter reducing intensity variation</td>
</tr>
<tr>
<td>Gaussian blur</td>
<td>1</td>
<td>5 × 5 integer low-pass filter for noise reduction</td>
</tr>
<tr>
<td>Bilateral filter</td>
<td>1</td>
<td>5 × 5 floating-point filter for image smoothing while preserving edges</td>
</tr>
<tr>
<td>Sobel-Luma</td>
<td>2</td>
<td>Edge detection algorithm utilizing RGB to luma color conversion</td>
</tr>
<tr>
<td>Unsharp mask</td>
<td>3</td>
<td>Sharpens an image</td>
</tr>
<tr>
<td>Filter chain</td>
<td>3</td>
<td>3 × 3 filter chained 3 times</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1</td>
<td>3 × 3 filter for image smoothing</td>
</tr>
<tr>
<td>Optical flow (LK)</td>
<td>16</td>
<td>Lucas-Kanade method for motion estimation</td>
</tr>
<tr>
<td>Harris</td>
<td>9</td>
<td>Corner detection for finding features in images</td>
</tr>
<tr>
<td>Shi-Thomasi</td>
<td>9</td>
<td>Corner detection with improved scoring function</td>
</tr>
<tr>
<td>Laplace</td>
<td>1</td>
<td>Derivative operator for edge detection</td>
</tr>
<tr>
<td>Square</td>
<td>1</td>
<td>Pixel-wise operation for increasing image contrast</td>
</tr>
<tr>
<td>Sobel</td>
<td>1</td>
<td>3 × 3 filter for edge detection</td>
</tr>
</tbody>
</table>

We evaluate FLOWER on two different FPGA platforms: Xilinx Alveo U280 (xcu280-fsvh2892-2L-e) and Bittware 520N-MX (Intel Stratix 10 MX2100). Both are accelerator cards connected to the host via PCIe Gen3x16. However, we can only use PCIe Gen3x8 for the Bittware 520N-MX due to restrictions of the used BSP.

Intel OpenCL codes are synthesized by the Intel FPGA SDK for OpenCL 19.4. Xilinx HLS C++ and OpenCL codes for the accelerator card are synthesized by the Vitis v++ compiler 2020.1. We use Vitis_hls 2020.1 to synthesize the generated IPs. Host programs for the Xilinx card use the XRT command-line interface without any global memory control bundle, and thus they cannot access global memory. Consequently, we need to impose this restriction on FLOWER applications as well.

Synthesis results in Fig. 5 alongside with resource usage in Table III shows that FLOWER applications have lower latencies. This is true also when applications are vectorized. Thanks to that automatic host code generation, the program-mer only needs to focus on writing the application from a single piece of code written using FLOWER. Consequently, it’s easier to make modifications of the code, since the host code is automatically synchronized with the FPGA code.

We start by evaluating our framework against Hipacc [11], [12] and AnyHLS [4] on a set of image processing applications, before assessing the OpenCL support.

A. Hipacc

The FPGA support in Hipacc is mostly designed for Zynq SoPC (System on Programmable Chip) platforms, and the generated IP blocks obtained from Hipacc are not immediately ready to be linked with the accelerators’ platform shell. Therefore, to compare our work with Hipacc, we rely on the SoPC IP output of FLOWER that is synthesized for the FPGA part xcu280-fsvh2892-2L-e found in the Alveo U280 card. With Hipacc, we generate each application by first making it compatible with the vitis_hls command-line tool, and then by synthesizing it as streaming IPs for the same FPGA part. All applications in Hipacc have an AXIS interface without any global memory control bundle, and thus they cannot access global memory. Consequently, we need to impose this restriction on FLOWER applications as well.

Fig. 5 illustrates how different optimizations dramatically improve the applications’ performance in terms of execution time, measured using Vitis analyzer. In order to observe how task pipelining takes place with dataflow optimization, we
launch each kernel 6 times, so that several memory transactions happen consecutively. In all applications, the DMA engine transfers 25.166 MB from the host to the kernel’s global memory. FLOWER extracts global memory operations from the kernel and automatically generates the dataflow-optimized version, which in turn allows burst transfers for read/write memory. In fact, in FLOWER, even applications that consist of only a single stage, are divided into at least three tasks which are read from global memory, compute, and write to global memory, enabling pipelined dataflow executions. Fig. 7 demonstrate how we utilize global memories.

As described in Section IV, vectorization aggregates several input data (pixels) to vectors, and by replicating the arithmetic operations, processes them at the same time. Thanks to appropriate packing of data and sequential access patterns on inputs and outputs of the generated kernel, FLOWER enables Vitis to figure out the corresponding memory interface data-width for kernel to global memory transfers. This data-width is aligned with the size of the vectorized datapath, which in turn results in optimal dataflow transfers throughout the entire design. Memory controllers of common FPGA cards support a data bus width of up to 512 bits, which in practice defines an upper limit for constructive vectorization. By combining vectorization and burst transfer optimizations we get the best performance.

Since AnyHLS could not take advantage of the dataflow optimization, it also could not benefit from burst transfers, resulting in execution times that are up to 20× slower.

C. OpenCL

The OpenCL backend of FLOWER generates multi-platform OpenCL device and host codes from the same application. These codes can be synthesized with both Xilinx Vitis and Intel OpenCL SDK without any modifications. Fig. 8 and Fig. 9 show the total execution time of kernels launched 6 times for Xilinx (measured using Vitis analyzer) and Intel OpenCL (measured using the Intel dynamic profiler). While we cannot compare Intel’s OpenCL version to the Xilinx one, because of the user licenses of the vendor tools, these charts demonstrate that FLOWER works on both platforms without requiring any change in the application code.
Fig. 8: Total kernel runtime of the Gaussian filter. The kernel is generated using FLOWER’s OpenCI backend for Xilinx FPGAs. The naïve version is made of only one kernel/task. The kernel is executed 6 times for a 1024×1024 image on a Xilinx Alveo U280 PCIe accelerator card.

Fig. 9: Evaluation of FLOWER’s OpenCL backend on a Bittware 520N-MX card (Intel FPGA). Applications are executed 6 times and total kernel runtime is measured. Image size is 1024 × 1024.

VI. RELATED WORK

FPGA designs applying dataflow optimizations for data access and reuse are significantly more efficient than non-optimized designs: A climate prediction application runs 800× faster than a naïve version when developers correctly address data movement [1], [13]. Unfortunately, current C/C++ HLS languages lack the ability to express parallelism and dataflow properly, which means that a huge amount of low-level programming is still required when using those languages.

When it comes to image processing in particular, the literature contains several DSLs and compilers for FPGAs. They follow various approaches for generating hardware: SCORE [14] introduces basic elements of dataflow architectures and is based on TDF which is basically an RTL language. RIPL [15] and Spatial [16] target intermediate languages to generate HDL code, named CAL dataflow [17] and Chisel [18]. Through these intermediate languages, RIPL and Spatial apply basic dataflow optimizations at the HDL level. The LIFT [19] and Darkroom [20] generate HDL code directly from functional patterns suitable for dataflow programming and do not support data-parallelism. While these tools construct hardware IPs and interfaces, they do not provide the flexibility of HLS tools that generate low-level RTL through automatic allocation, binding, and scheduling of operators/registers [21], as in commercial or open-source HLS tools like LegUp [22].

Hipacc [11], [12], HeteroHalide [7], PolyMage [8], the Merlin compiler [23], and AnyHLS [4] target C/C++ codes in their backend while applying optimization passes suitable for HLS tools such as Xilinx Vitis or Intel HLS compilers. AnyHLS makes use of the AnyDSL compiler [24] to partially evaluate [25], [5] higher-order functions in order to generate optimized OpenCL/C++ HLS codes. Daliha [26] is similar to AnyHLS, in that it provides a general purpose language for generating specialized HLS C++ code with performance predictability. Daliha uses a rich type system to deliver performance, while AnyHLS relies on function specialization.

There are several studies presenting the importance of dataflow designs for stream processing, most notably OxiGen [27], which relies on the MaxCompiler, or Frost [28], which uses the SDAccel synthesis tool (now superseded by Xilinx Vitis) and doesn’t strictly follow the canonical form required for Vitis. Both tools require ad-hoc host programs to be written by the user. HLS dataflow transformations have also demonstrated benefits in packet processing and deep learning pipelines [29], [30], [31].

In contrast to FLOWER, most of these frameworks, including Hipacc and PolyMage, generate a dataflow design through C++ template metaprogramming, and their approach does not rigorously follow the canonical form recommended by Xilinx. They only support Vivado HLS and are not updated for the new Vitis toolchain from Xilinx. Additionally, most of these studies only evaluate the generated C/C++ IPs on Zynq SOC platforms, which are only designed for embedded systems. StencilFlow [32], and other frameworks [33], [34] show how an optimized Intel OpenCL implementation is beneficial for dataflow and stream processing designs. AFFIX [35] also provides a scalable OpenCL library for vision algorithms via Intel OpenCL on FPGAs. While our work is based on stateless synchronous dataflow graphs, others [36] introduce a data-centric model for stateful dataflow multigraphs, which relies on SDAccel and Xilinx OpenCL code.

VII. CONCLUSION

Dataflow transformations, low-level optimizations, and host code development are essential building blocks to achieve an efficient design for streaming applications, as shown by the examples in this paper. FLOWER allows programmers to write their applications in a high-level library, and automatically introduces the required transformations and optimizations. Our results demonstrate that our work is not only faster compared to similar tools, but also increases productivity, in contrast with alternatives where manual work is required to do those transformations. FLOWER is fully compatible with both Xilinx and Intel FPGA accelerator cards, allowing to use the same code to drive two different devices. In the future, we would like to take advantage of the LLVM IR backend of AnyDSL to target the recently open-sourced front-end of Vitis, in order to perform even more optimizations.