FastFlow targeting FPGAs

Marco Danelutto, Gabriele Mencagli & Alberto Ottimo
Dept. of Computer Science, Univ. of Pisa, Italy
Email: {name.surname}@unipi.it

Francesco Iannone & Paolo Palazzari
ENEA, Italy
Email: {name.surname}@enea.it

***Abstract*—Writing good code for FPGA is a challenge "per se", but also running already existing and optimized FPGA kernels often requires writing specific "host side" code and some target hardware knowledge to achieve good performances. In this work, we describe a FastFlow extension supporting seamless offloading of tasks to FPGA, once an FPGA kernel is available. In particular, we show how kernels implemented in Vitis and running on XILINX Alveo FPGA boards may be integrated to implement "normal" parallel stages (pipeline stages, map/farm workers) in a structured parallel FastFlow computation. Preliminary experimental results are shown, demonstrating the feasibility of the approach.**

***Index Terms*—structured parallel programming, design patterns, FPGA offloading, accelerators.**

I. Introduction

FPGA offer substantial advantages when compared to both CPU and GPU based accelerators. They offer the possibility to implement "in hardware" particular algorithms such that their execution may be completed with order of magnitude speedup with respect to CPU implementation. In addition, the power needed to run algorithms on FPGA is usually considerably smaller than the power needed to run the same algorithms with comparable performances on both CPUs and GPUs. Unfortunately, efficient FPGA programming is itself orders of magnitude more complex than CPU programming and even of GPU programming. Indeed, in recent years, a number of innovative programming models and environments have been developed, that can be used to produce quite efficient FPGA code from properly annotated high level language code rather than from low level RTL code such as VHDL, Verilog or other surrogates that look like slightly higher level but still are fundamentally Register Transfer Languages [1]–[3].

Several structured and non structured parallel programming environments provide ways to target FPGA through the support of kernels compiled from high level user code. In the context of the REPARA EU funded FP7 project, FastFlow [4], has been extended with basic possibilities to address FPGAs equipped and programmed with the ThreadPoolComposer programming environment developed by the Darmstad University [5]. The OpenMP family introduced the `target` directive to support computation redirection to FPGA, and a similar approach has been taken in OmpSS. The whole directive based programming model universe is again trying to supply programmers with the possibility to target FPGAs using properly annotated sequential code [6]. In most cases, the usage of directives to identify the sequential code to be offloaded on the FPGA leads to decent performances with "standard kernels" but does not compete with hand written FPGA kernel whose execution is managed through explicit OpenCL code. Whatever the programming model used, FPGAs are separate accelerators and host management code is therefore required to orchestrate the execution of kernels on the FPGA.

In this work, we discuss a new support of FPGAs in FastFlow. It provides efficient ways to offload computations to existing, pre-compiled kernels running on Xilinx Alveo FPGAs via the efficient and seamless embedding of the needed OpenCL offloading code within the FastFlow run time. Differently from the REPARA FastFlow implementation, the new support does not require any additional logic on the FPGA, thus leaving all the available FPGA resources free for "business" kernel implementation. The new FPGA FastFlow support ensures separation of concerns as well as maximisation of the different expertise sported by FPGA and parallel application programmers, greatly improving the time-to-solution while developing CPU+FPGA parallel applications.

II. FastFlow

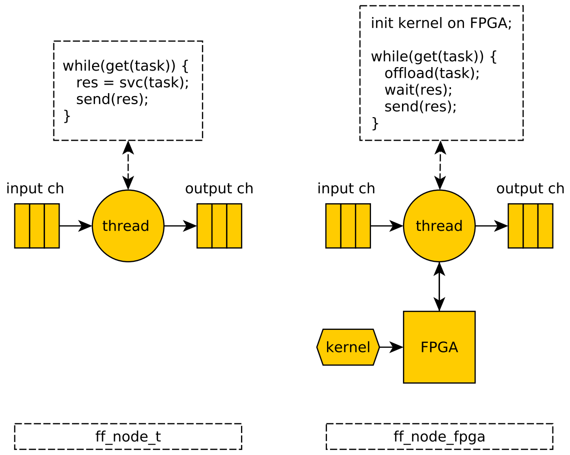
FastFlow[1] is an open-source parallel programming framework developed by the University of Pisa and Torino since early 2000s [4]. It is a header only C++ library providing the application programmer with a set of pre-defined parallel programming patterns that can be completed with business logic code and composed to model complex parallel applications without charging the application programmers of the burden typically involved in parallelism exploitation.

FastFlow was originally designed to target shared memory multi-cores only. Recently, FastFlow has been extended in such a way the patterns provided within the single shared memory node may be used to orchestrate the execution of a parallel application on COW/NOW architectures. This extension is built on top of two different backs ends, namely plain TCP/IP and MPI [7].

FastFlow parallel computations are arranged as compositions of parallel patterns. Parallel patterns include standard stream parallel (pipeline, task farm) and data parallel (map

This work has been partially funded by EU funded project TEXTAROSSA and by the Spoke 1 "FutureHPC BigData" of the Italian Research Center on High-Performance Computing, 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 RS (M4C2-19)" - Next Generation EU (NGEU).

[1]https://github.com/fastflow/fastflow.git



```
while(get(task)) {
  res = svc(task);
  send(res);
}
```

```
init kernel on FPGA;

while(get(task)) {
  offload(task);
  wait(res);
  send(res);
}
```

Fig. 1. Standard FastFlow component (left) and new FPGA offloading component (right)

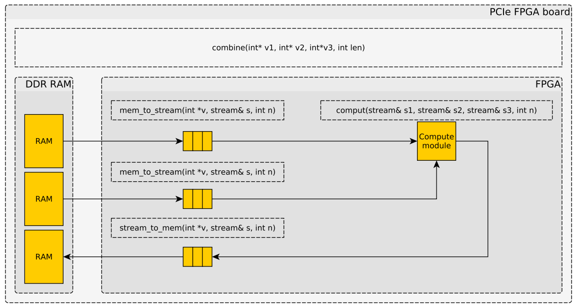

PCIe FPGA board
combine(int* v1, int* v2, int*v3, int len)
DDR RAM
FPGA
mem_to_stream(int *v, stream& s, int n)
comput(stream& s1, stream& s2, stream& s3, int n)
RAM
Compute module
mem_to_stream(int *v, stream& s, int n)
RAM
stream_to_mem(int *v, stream& s, int n)
RAM

Fig. 2. Structure of typical Vitis kernel: dotted text represent C/C++ signatures, orange boxes are the hardware components, synthesised from C/C++ user code and library/IP code

(parallel for), reduce, stencil) patterns as well as higher level and more general pattern (divide&conquer, data flow) and a sequential "wrapper" patterns suitable to re-use existing business code in a parallel computation.

The FastFlow sequential business logic code wrapper is implemented by a a `ff_node_t<Tin,Tout>` class specialization that provides a method processing `Tin*` data item and producing `Tout*` result. The `ff_node_t` nodes may be constructed from properly written classes sub-classing `ff_node_t` public interface and providing a `Tout *svc(Tin* t)` method as well as simply providing a `std::function<Tout*(Tin*)>` object in the constructor. An `ff_node_t` node is implemented using a thread in charge of processing `Tin*` data taken from FastFlow (input) channel and of delivering results to a FastFlow output channel (see Fig. 1 left). Every component in a FastFlow program is a `ff_node_t`: sequential nodes wrapping business logic code, pipelines with multiple stages, data parallel map and stencil computations and so on.

III. Vitis

Vitis [8] provides a high-level synthesis flow supporting the implementation of "quasi" standard C/C++ programs (kernels) on the FPGA; "quasi" standard because some constructs (mainly, recursion) are not supported by the flow. The FPGA is divided into two sections: a static section, programmed once at the board installation time, containing all the HW IP's needed to interface with the outer world (PCIe and memory controllers, clock and reset generators) and a dynamic section, programmed via the PCIe interface, which is configured with the kernels defined by the user. To achieve efficient behavior, the kernel code must be carefully written to allow the compiler to extract the inherent parallelism (both spatial and pipeline). To this aim, the HLS flow puts at the programmer's disposal some pragmas to guide the compilation step (to pipeline or unroll a loop, to inform the compiler that some variables are not dependent so parallelization can be carried out, to force the implementation of certain operations on specific HW modules, ...). The set of techniques used by Vitis programmers to achieve efficiency in the implementation of the C/C++ kernels onto the FPGA are peculiar to FPGAs and quite far from the techniques used to achieve efficiency in the programming of CPU or GPU code. To be compliant with the streaming behavior, which best suits the pipeline style of FPGA programming, we structured the kernel to be mapped on the FPGA through two kinds of components (see Fig. 2)

- Data mover components: they access the external memory and streams, copying in a pipelined way data from input streams to the memory or from the memory to the output stream. Data movers can be seen as the starting and ending points of streamed processing.
- Compute components: these are functions taking streams as arguments and computing output stream items from the input stream items.

The two kinds of components are structured in a "dataflow" computation graph and compiled to produce a configuration file (.xclbin) describing the set of hardware components to be mapped onto the FPGA, possibly exploiting any kind of data-flow parallelism available. The .xclbin file is loaded into the dynamic section of the FPGA via the run-time APIs that control the FPGA through the PCIe interface. It's worth to be evidenced that the translation of the C/C++ kernel code into the .xclbin file is a long process, requiring hours to be executed.

IV. FFPGA prototype

Within FastFlow, we designed a specialized node (`ff_node_fpga`) implementing the `ff_node_t` public interface and embedding all the additional logic needed to offload task computation to an FPGA kernel rather than executing the task on a CPU thread. The constructor of the `ff_node_fpga`) accepts parameters specifying the kernel name and the bit-stream file to be used for offloading. Constructor code manages to discover the FPGA resources available and pre-load the bitstream and set up all the necessary OpenCL context and command queues. The constructor parameters include also a specific data structure filled up by programmer detailing the kind (input or output parameter, vector or scalar) of the kernel parameters. The `svc` method–i.e. the method actually computing the result out of any input task reaching the `ff_node`–manages to "unpack" the input task, set up the buffers needed to transfer the input parameters to the FPGA and to retrieve the results from the FPGA, starts input data transfers, starts kernel execution and finally starts result copy in host memory. Depending on some specific class constructor parameters, these sequence of actions may be executed sequentially, in such a way computation/communication time is overlapped, with or without using "pinned memory" and therefore with different memory usage and performances. Fig. 1 (right) illustrates the usage of the new node within a regular FastFlow computation to offload tasks to the FPGA Vitis kernels. The new node type may be used in any place where a normal `ff_node_t` can be used in FastFlow, for instance as a pipeline stage, a farm or map worker, etc.

Therefore, a `ff_node_fpga` may be declared such as:

```
FTaskCL task_description;
task_description.addIn(size_in_bytes);
task_description.addIn(size_in_bytes);
task_description.addOut(size_in_bytes);
task_description.addScalar(sizeof(int));
...
auto ComputeFpga =
  new ff_node_fgpa(kernelName, bitstreamFilename,
                   task_descr);
```

and used in a code such as:

```
ff_pipeline p;                  // create a pipeline
p.add_stage(new Source(...));  // gen input stream
p.add_stage(new Compute1(...));// comp. tasks (CPU)
p.add:stage(ComputeFpga);       // comp. tasks (FPGA)
p.add_stage(new Drain(...));    // store  results

p.run_and_wait_end();           // run to completion
```

to implement a pipeline with a FPGA offloader stage.

V. Experimental results

We discuss a set of experiments running on a local server equipped with an FPGA. This is a Dell Power Edge dual socket Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz host node, with 128G of memory and a Xilinx Alveo U50 FPGA board [9]. We have tested our prototype implementation of the `ff_node_fpga` FastFlow node with different kernels and different parallel application structures. As far as FPGA kernels are concerned, we used simple kernels from Vitis examples (`vecadd`) and different kind of much more complex kernels we developed, including a file compressor kernel and a set of video filters used to develop more complex applications operating on standard mpeg video frames. The compressor kernel, in particular, is about 700 lines of code, including some 50 `#pragma HLS` directives the finally describes the compressor kernel as a dataflow graph as follows:

```
static hls::stream<io_stream_48B_tuser>
     LZ77Enc2Huffman_stream("LZ77Enc2Huffman_stream");
static hls::stream<io_stream_16B>
    DDR2Deflate_stream("DDR2Deflate_stream");
static hls::stream<io_stream_32B>
    Deflate2DDR_stream("Deflate2DDR_stream");

#pragma HLS dataflow
    Memory2Stream(in,DDR2Deflate_stream,input_size);
    LZ77_Encoder(DDR2Deflate_stream,
                 LZ77Enc2Huffman_stream, input_size);
    huffman_encoder(LZ77Enc2Huffman_stream,
                    Deflate2DDR_stream);
    Stream2Memory(out,outSizeMem,Deflate2DDR_stream);
```

which is fully compliant to the structure described in Sec. III and represented in Fig. 2.

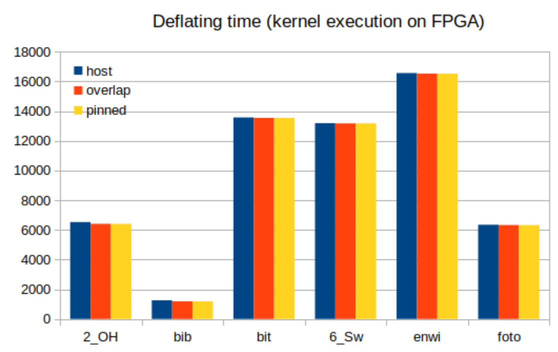


Fig. 3. Comparison of kernel execution times on FPGA with different input tasks "host" refers to C++/OpenCL host code, "overlap" and "pinned" refer to two different implementations of the FastFlow node `ff_node_fpga` (times in μsecs)

A. Functional test

Functional tests have been completed demonstrating that the offloading actually works as expected both on the attached FPGA board and in software emulation mode, which is particularly useful during program development and tuning. Functional test experiments show that offloading a computation on the FPGA requires times in the order of hundreds of μsecs at the steady state, that is after the initialization phase that includes bit-stream upload, which are the same times experimented offloading the same computation from within standard OpenCL host code. The execution times on the FPGA are obviously the same of the ones measured when using OpenCL/XRT code to manage offloading, as expected (see Fig. 3).

B. FastFlow wrapper vs plain OpenCL host code

We implemented our wrapper node in two different ways, that can be used specifying proper parameters while creating the wrapper `ff_node_fpga`. The two versions exploit different kind of buffering/communications strategies:

- the version labelled as "overlap" in the following, issues data transfer and kernel execution commands on the FPGA OpenCL queue by using actual, asynchronous copy commands. Different threads are used on the host to start FPGA kernel computations (moving input data and starting kernel) and to wait kernel termination and to copy back the kernel results. Communication time hiding thus results out of the overlap of different operations relative to different kernel executions ordered by different threads.

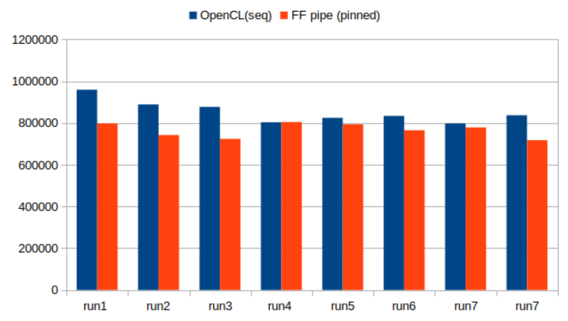


Fig. 4. OpenCL vs FastFlow pipe ”pinned” compression completion times

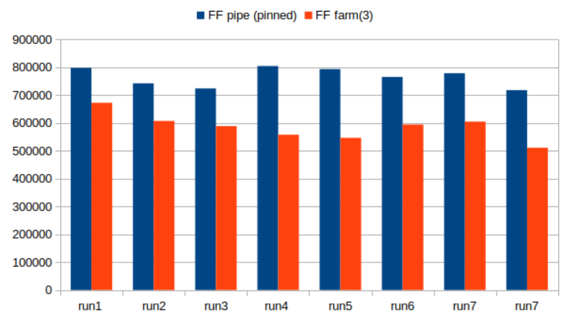


Fig. 5. FastFlow farm vs FastFlow pipe ”pinned” compression completion times

- the version, labelled as “pinnned”, enforces buffer set up such that input buffer access is granted through DMA accesses commanded by the FPGA IP rather than by the execution of explicit, synchronous copy operations. This version uses aligned and pinned memory buffers which are transparently managed by the `FTaskCL` implementation.

These different versions have been used to run a stream compressor application modelled as a three stage pipeline:

```
1 ff_pipeline p;
2 p.add_stage(new gen());
3 fnode_pinned_overlap fnp(bitStream, kernelName);
4 p.add_stage(fnp.stage());
5 p.add_stage(new drain());
6 p.run_and_wait_end();
```

The first stage, the `gen` node added at line 2, reads files from disk, the second stage (lines 3–4) offloads tasks to the FPGA and the third stage, the `drain` node added at line 5, eventually write zipped files to disk. The second node in this code excerpt is the one using pinned memory (zero copy while offloading to FPGA) and communication/computation overlap via double buffering as described above. ”Pinned” version works better than the simple ”overlap” version and both outperform the plain OpenCL host code, offloading the kernels sequentially, one after the other. The ”pinned” version of the compressor pipeline actually takes some 765 msec to complete compressing a short stream of tasks, while the OpenCL host code spends for the same stream 853 msec (in both cases these are averages computed on 32 runs). Overall this represents a small 10% (see Fig. 4).

C. Threads vs. OpenCL advanced features

Despite the fact double buffering techniques are a *de facto* standard when programming and exploiting PCIe based accelerators with OpenCL, we explored the possibilities offered by directing computations to a single FPGA accelerator from different host threads, as an example from workers in a farm parallel pattern. The idea is to avoid complex buffer management and to simply direct different computations to the same kernel running on the FPGA through the same interface. According to the FPGA documentation, this should enable the controller to order the different requests coming to the FPGA through the very same manager in such a way that data buffer copies (to and from FPGA memory) overlap with the actual kernel computations. We therefore modified the application sketched in Sec. V-B in such a way the compressor FPGA node in the pipeline is substituted by a farm (i.e. it has been parallelized) with 3 workers. All the workers direct computations to the same FPGA board and kernel. The code is simply modified by substituting lines 3–4 in Sec. V-B code with lines:

```
ff_farm farm;
FDevice *devic=new FDevice(bitStream,kernelName);
std::vector<ff_node *> w;
for(int i = 0; i < nworkers; ++i) {
  fnode_pinned_overlap * worker =
          new fnode_pinned_overlap(device);
  w.push_back(worker->stage());
}
farm.add_workers(w);
p.addStage(farm);
```

This brings down the completion time for the same file stream of the previous section to an average of 585 msecs, that is with a 31% improvement over the OpenCL code and a 23% improvement with respect to the ”pinned” pipeline (see Fig. 5).

D. On going work

The results shown so far where somehow expected, but confirmed the proper design of the “FastFlow single node offloader”. Indeed, FastFlow offers much more possibilities that are worth begin explored relatively to task offloading to accelerators. We are currently evaluating different aspects related to computation offloading to FPGA:

a) Multiple kernel instances: we are trying to measure the advantages derived from the usage of different instances of the same kernel used to compute task offloaded from different FastFlow concurrent activities (e.g. farm or map workers, see Fig. 6 (left)). Also, we are considering the possibility to use different kernels to support task offloading from FastFlow concurrent activities running diverse “business logic code”, such as stages in a pipeline. In both cases, we hope to be able to take advantage of the communication/computation overlap such that we are able to saturate the host–board memory bandwidth.

b) Multiple board support: we want to figure out how different FPGA boards attached to the same multicore node

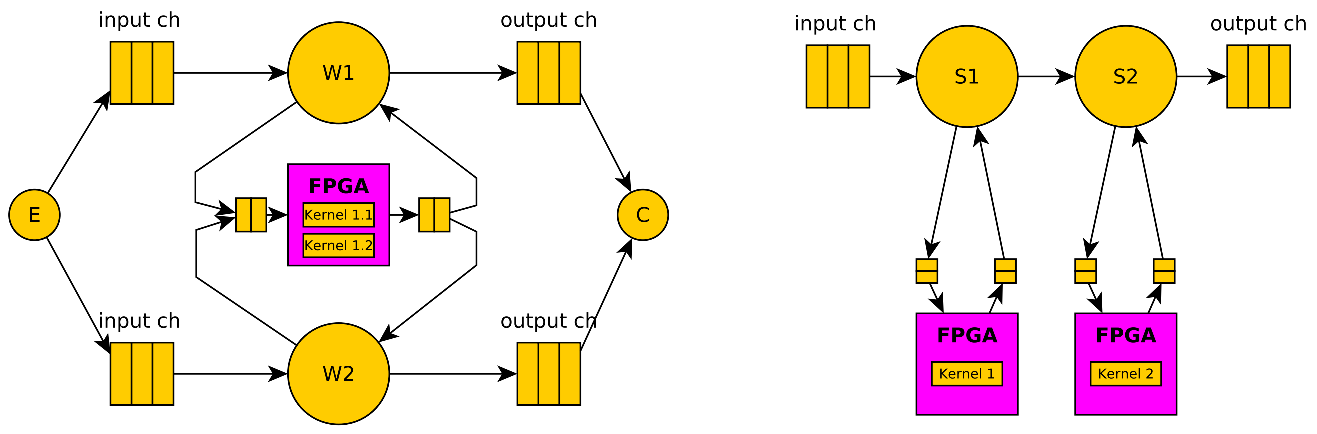


Fig. 6. Alternative usage of FPGA kernels: Multiple kernel instances from multiple farm workers (left) and Different kernel instance on different FPGA boards from different pipeline stages (right)

may be conveniently exploited to offload computations from different components (patterns) in FastFlow structured parallel applications (see Fig. 6 (right)).

c) Multiple cooperating boards: we want to explore the possibility to use different FPGA boards, interconnected as a pipeline of boards, to exploit pipeline parallelism at the FPGA kernel level as a plain `ff_node_t` in a FastFlow computation. We are currently investigating the possibilities offered by an FPGA networking library developed at INFN within the project that partially funded this work.

VI. Conclusions

We described preliminary results relative to the support of computation offloading to FPGA from FastFlow. We showed that tasks may be seamlessly offloaded to pre-compiled kernels provided that a) the `xclbin` file and kernel name are known and b) the structure (type and input/output direction) of the kernel parameters is also known. In addition to providing the kernel name and the name of the bit-stream file, the FastFlow programmer is only required to set up a special object to declare how many parameters are to be given in input and output to the kernel as well as their sizes. All the rest of the effort of locating the FPGA, loading the kernel, managing the host-to-board and board-to-host communications, offloading computations to FPGA and the whole set of associated synchronization actions is managed through our FastFlow `ff_node_t` extension. Our support run time does not introduce any kind of delay with respect to the offloading to the very same pre-compiled kernels using different, lower level programming tools and environments. We also showed that the possibilities offered by FastFlow to seamlessly set up different parallel structures for the very same application can be exploited to manage efficient offloading of tasks to available FPGA kernels. Our FastFlow FPGA support does not require to load specific IP on the FPGA to manage offloaded kernels but the standard FPGA dynamic configuration IP. This is a step ahead w.r.t. to what we alredy experimented in previous projects [5].

We *de facto* rely on the existence of pre-compiled (and optimized) kernels, and therefore our proposed approach does not solve the problem of reducing the time-to-solution relative to FPGA kernel programming. On the other hand, it fully preserves separation of concerns, in that parallel application programmers may easily and seamlessly experiment different parallel exploitation patterns relying on the existence of the optimized FPGA kernels while the FPGA kernel developers should not care about kernel usage from host parallel code while developing and optimizing the FPGA code.

References

[1] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," DAC Design Automation Conference 2012, 2012, pp. 1212-1221, doi: 10.1145/2228360.2228584.

[2] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[3] Panda Home Page, https://panda.dei.polimi.it/, 2022

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: High-Level and Efficient Streaming on Multicore. In Programming multi-core and many-core computing systems (eds S. Pllana and F. Xhafa) 2017, DOI: https://doi.org/10.1002/9781119332015.ch13.

[5] J. Korinth, D. de la Chevallerie, A. Koch, An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. FCCM 2015: 195-198

[6] Mayer, F., Knaust, M., Philippsen, M. (2019). OpenMP on FPGAs—A Survey. In: Fan, X., de Supinski, B., Sinnen, O., Giacaman, N. (eds) OpenMP: Conquering the Full Hardware Spectrum. IWOMP 2019. Lecture Notes in Computer Science(), vol 11718. Springer, Cham. https://doi.org/10.1007/978-3-030-28596-8$_7$

[7] N. Tonci, M. Torquati, G. Mencagli, M. Danelutto. "Distributed-memory FastFlow Building Blocks", International Journal of Parallel Programming (IJPP), Springer, HLPP 2022 Special Issue, 2022, DOI: 10.1007/s10766-022-00750-5

[8] Xilinx, Vitis Unified Software Platform Documentation, Embedded Software Development, UG1400 (v2022.2) October 19, 2022, available at dos.xilinx.com/r/en-US/ug1400-vitis-embedded/Getting-Started-with-Vitis

[9] Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet, DS965, 2020, available at https://docs.xilinx.com/v/u/en-US/ds965-u50