Make Life Easier for Embedded Software Engineers Facing Complex Hardware Architectures

Romain Leconte, Eric Jenn, Guy Bois, Hubert Guérard

▶ **To cite this version:**

Romain Leconte, Eric Jenn, Guy Bois, Hubert Guérard. Make Life Easier for Embedded Software Engineers Facing Complex Hardware Architectures. 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020), Jan 2020, Toulouse, France. ⟨hal-02474476⟩

HAL Id: hal-02474476

https://hal.science/hal-02474476v1

Submitted on 11 Feb 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Make Life Easier for Embedded Software Engineers Facing Complex Hardware Architectures

Romain Leconte*†, Eric Jenn*‡, Guy Bois†, Hubert Guérard†
*IRT Saint-Exupéry, 3 Rue Tarfaya, CS 34436, 31400 Toulouse, France
{romain.leconte, eric.jenn}@irt-saintexupery.com
†Space Codesign Systems Inc, 450 rue St-Pierre, Suite 1010 H2Y 2M9 Montreal, Quebec, Canada
{romain.leconte, hubert.guerard, guy.bois}@spacecodesign.com
‡Thales AVS, 105 Avenue du Général Eisenhower F31036 Toulouse, France
eric.jenn@fr.thalesgroup.com

***Abstract*—The increasingly parallel execution platforms - mixing multi-cores, GPUs, and programmable logic (namely, FPGA) - require new development techniques and technologies to be used efficiently. Software developers are hampered by the complexity of modern SoCs and MPSoCs. In particular, the complexity of the hardware design flow makes the exploitation of FPGAs difficult and expensive, especially in cases where the design space to be explored is large. Therefore, this paper proposes a design flow that offers joint support of both hardware and software flows making life easier for embedded software engineers. It is based on a HW/SW codesign approach where a sequential C code annotated with OpenMP offloading directives is progressively transformed into an FPGA implementation. OpenMP has been selected because it is a widely adopted solution in the high-performance computing domain, but also because work is currently going on to extend its scope to embedded real-time systems. This paper identifies the important properties required in such a flow, demonstrates how they are supported by our workflow, and, finally, presents results of our approach on an image processing function deployed on both Zynq and Cyclone platforms.**

***Index Terms*—OpenMP, FPGA, Design Space Exploration, Embedded systems**

I. Introduction and contributions

Optimizing the use of complex modern Multi-Processor System on Chip (MPSoC) require significant efforts for the software and hardware developers. They have to cope with an ever increasing number of homogeneous or heterogeneous cores, the presence of SIMD units, AI accelerators, GPGPUS, and now, FPGA.

On the software side, to fully *exploit* the high level of parallelism offered by the platform, software engineers have now to *expose* as much as possible the parallelism of their applications, at coarse (e.g., process or thread), and fine grain (e.g., loop). When the level of hardware parallelism is high, this activity definitely needs to be supported by appropriate abstractions and modeling or programming languages. OpenMP [1] is one possible solution.

OpenMP is one of the most popular technologies to support the large scale parallelization of applications. OpenMP is an API specification maintained by the OpenMP workgroup[1]. It proposes a set of source-level annotations to expose the parallelism of applications at multiple levels: loops, tasks, data, etc., and manage communications between the parallel components on the basis of the shared memory communication paradigm. Since version 4.0 [1], OpenMP also supports heterogeneous architectures. Using the `target` directive, parts of the code can be offloaded from the main CPU (or *host*) to one or several accelerators implemented using general purpose processors, GPGPUs, or FPGAs. Using these offloading annotations, the code running on the host can move data from its memory to the devices (or *target*) memory, execute logic on the target device and move data back to the host memory.

On the hardware side, software developers are hampered by the complexity of modern SoCs and MPSoCs. Although FPGAs are now competitive with respect to state-of-the-art processors – thanks to the latest technology breakthroughs, including low latency communication links between FPGAs and CPUs, and the relatively low power consumption of todays FPGAs – moving part of the code from software to hardware (FPGA) still requires a significant effort.

So, in order to deploy a complex application on those execution platforms, developers require an integrated design flow offering joint support of both hardware and software.

Towards that goal, our proposal has 3 main characteristics:

- It starts with a unique and "high-level" formalism to expose parallelism. This formalism allows parallelization and deployment decisions to be taken gradually during the design process, and with as few changes to the source code as possible.
- It provides the means to translate an OpenMP code into an hybrid CPU-FPGA virtual platform that can be validated, HW/SW partitioned and evaluated by software developers.
- It comes with an semi-automatic synthesis and bitstream generation process targeting Xilinx' or Intel's FPGAs.

[1]See www.openmp.org.

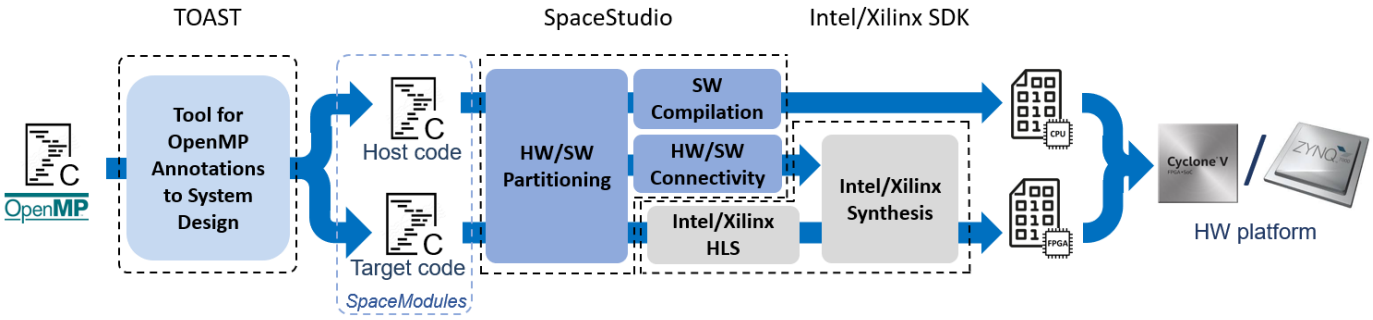

Fig. 1. From OpenMP to SoC FPGA workflow (TOAST stands for *Tool for OpenMP Annotations to System design Translation)*

Note that even if the last point may require some manual interventions of hardware engineers, for instance to select appropriate constructs or add appropriate annotations to facilitate or guide the High-Level Synthesis process, we estimate that over 75% of effort can be achieved by software engineers.

II. Overview of the Proposed Workflow

A. Overview

The complete workflow is depicted in Figure 1. The input is an OpenMP annotated code in which the developer has identified and annotated the code regions to be parallelized and offloaded. Before entering the process leading to the FPGA implementation (i.e., entering TOAST), the annotated code can be first compiled and functionally tested on devices easier to setup, such as a GPU (e.g., NVidia) or on an acceleration board (e.g., a Xeon Phi accelerator). This is achieved using an OpenMP-aware compiler (e.g., gcc or Clang) supporting the target.

During the first phase of the workflow, the source code is transformed into two components: the *host code* aimed at being executed on the host, and the *target code* aimed at being deployed on the accelerator (another general purpose processor or an FPGA-implemented hardware accelerator). Both components interact using services provided by a generic control and communication API. This first phase, implemented by the TOAST tool, is detailed in Section III.

During the second phase, the host code and the target code enter the design exploration process in which candidate solutions – including deployment and target technologies – are successively designed, evaluated, and refined. These evaluation steps are performed on a SystemC virtual platform during the early phases, and on the actual hardware implementation during the late phases. The parts of the application deployed on the CPU (the *host*) are compiled using a standard toolchain (gcc, Clang, etc.); the parts that are deployed to the FPGA go through High Level Synthesis (HLS). In our case, this phase is implemented by a System Design Environment (SDE) called SpaceStudio from Space Codesign Systems [2] and by FPGA vendors specific toolchains (e.g., Vivado from Xilinx [3], and Quartus Prime from Intel [4]).

B. System Design Environment and FPGA vendors specific toolchains

A System Design Environment facilitates the development and deployment of applications that target heterogeneous systems (CPU and FPGA). Xilinx proposes a SDE called SDSoC[2], but unfortunately it only targets the Xilinx Zync platform. We based our toolchain on Space Codesign's SDE called SpaceStudio [2], which presents the advantage to be platform-independent and allows the system architect to deploy on either Xilinx or Intel platforms.

Figure 2 summarizes the HW/SW codesign flow of SpaceStudio. A system architect creates a set of architectures representing different mappings of the application's components onto a specified platform and specifies the type of communication used between those components.

SpaceStudio supports virtual platform simulation for performance prediction and algorithm validation. It integrates moni-

[2]See https://www.xilinx.com/sdsoc.html.

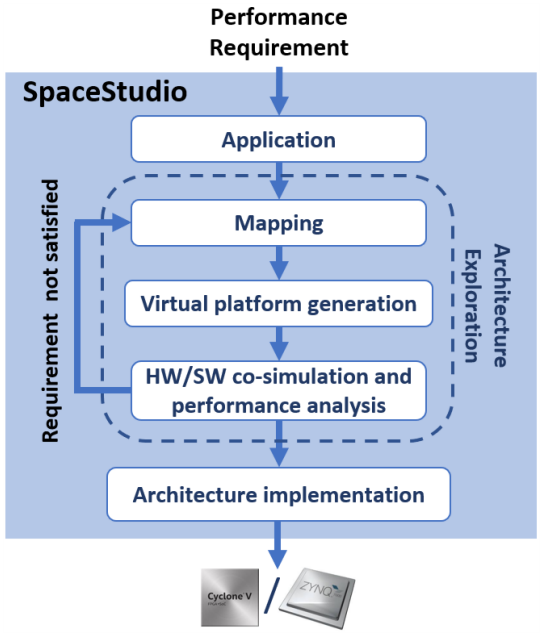

Fig. 2. SpaceStudio workflow

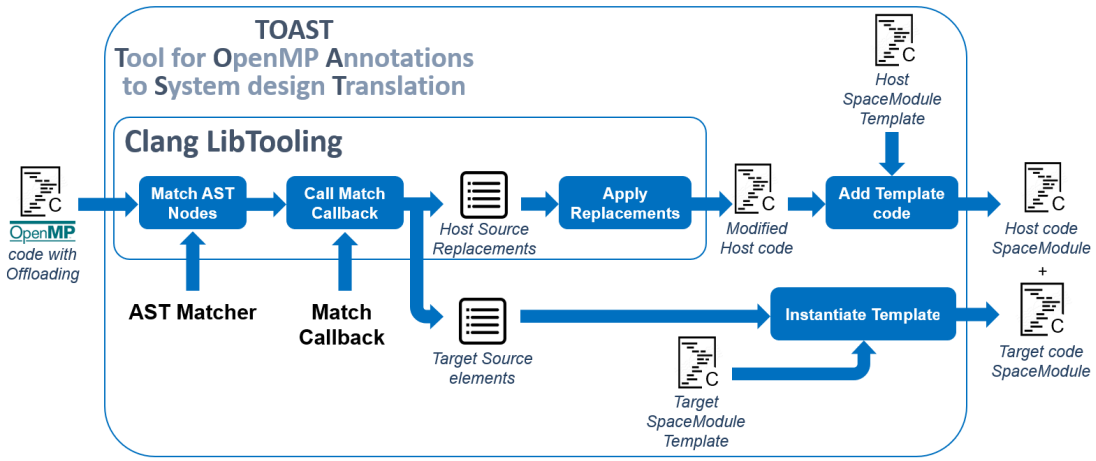

Fig. 3. TOAST Architecture

toring and analysis capabilities for non-intrusive performance profiling of both hardware and software. Examples of collected metrics are bus performance (latency and throughput), queue and memory usage, processor load, task scheduling, context switching, as well as timing data. Using the collected information, the system designer can determine which architectures allow the application to exhibit the best performance. Those architectures can then be directly implemented on the target platform as SpaceStudio generates the code and project files for downstream HLS tools to synthesize application-tasks mapped as co-processor. The result of this process can be directly deployed on the target to be tested and to confirm the performance characteristics of the program.

The input of SpaceStudio is a SystemC code partitioned into several logical modules (called "Space Modules") that communicate using a dedicated communication API. Those modules are directly generated from the program source code. OpenMP annotations are used to designate the parts to be offloaded, and to capture the control and data dependencies. This generation process is implemented in the TOAST tool described hereafter.

III. From OpenMP to System Level Design with TOAST

Offloading directives have been introduced in OpenMP 4.0. They support the deployment of part of a program on one or several *devices*. A device may be another processor, a GPU, a FPGA coprocessor, etc. This allows to exploit the various forms of hardware-level parallelism using a unique technology and a consistent model of computation and communication, rather than using specialized languages or APIs to target directly these kind of accelerators. Using these offloading annotations, the code running on the main CPU (or *host*) can move data from its memory to the device's (or *target*) memory, execute logic on the target device and move data back to the host memory.

TOAST (Tool for OpenMP Annotations to System design Translation) is the bridge between OpenMP and the system level design. It interprets the OpenMP offloading directives (`omp target`) to generate a functionally equivalent architecture ready to be deployed on one or several device configurations.

The input of TOAST is a source file containing the host code as well as the target regions to offload (i.e., the code that will be ultimately run on the device).

The output of the tool is composed of several source files:

- One for the host, consisting of the input file with the modifications listed in the next section applied to it.
- One file for every Space Module generated out of a target directive.

Next, for each Space Module device specification, there are two possible refinements:

- The use of an HLS tool to produce a bitstream if the device is an FPGA. In that process, OpenMP constructs still in the device source file (such as parallel regions or tasks constructs) will be discarded.
- The use of an appropriate compilation toolchain depending on the device the module is mapped to. For example, a version of gcc to compile code for soft cores such as the MicroBlaze or Nios.

Note that TOAST keeps all OpenMP annotations of the initial OpenMP file, except those interpreted during the offloading process implementation. In practice, this means that those annotations can still be processed by the host or the target[3] dedicated compilers (for example gcc for the Cortex-A9 or Cortex-A53), as long as they support OpenMP. For instance, if the OpenMP source file contains both offloading annotations (`omp target`) and loop or task parallelization annotations (i.e., `omp parallel for`, `omp task`), the latter may be exploited by the host compilation toolchain to generate multi-threaded and multi-core implementations of the host part.

[3]If the target device is another processor, not a FPGA.

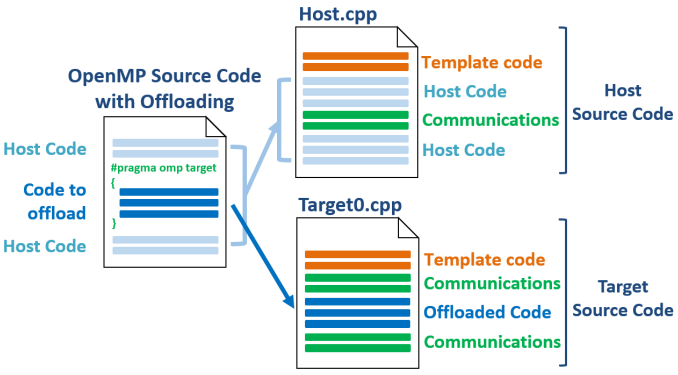

Fig. 4. TOAST Code generation

Interconnects and communication between host and devices are handled by the SpaceStudio platform and its API. They are not covered in this paper.

IV. TOAST Implementation Details

TOAST is implemented using the Clang LibTooling framework [5]. This framework provides a set of services to access and modify the Abstract Syntax Tree (AST) of a C program.

Figure 3 presents the architecture of TOAST and how it integrates Clang LibTooling by implementing:

- **AST Matchers**, which are used to find the elements of the AST corresponding to the OpenMP offloading directives
- **Match Callback Functions**, which are used to replace elements of the original source code and create the source files to be deployed on the devices.

Figure 4 schematically describes the structure of the generated files for one OpenMP offloading directive (`omp target`). As shown on this figure, some code parts are simply "translocated" from the input file to the host or target files whereas other, "template code" and "communication code" are generated according to the OpenMP directives.

A. Subset of OpenMP currently covered by TOAST

The current implementation of TOAST supports the following OpenMP directives:

- `pragma omp target`
- `pragma omp target data`
- `pragma omp target enter data`
- `pragma omp target exit data`
- `pragma omp target update`

It also fully supports the *map* clause that is the core of the OpenMP API to manage communications and avoid unnecessary transfers between the *host* and *device* memories.

It is worth noting that a `omp target` clause does not fork any treatment: it simply offloads some code block to some external device, in order to accelerate its computation. Accordingly, the host thread that initiates this computation is blocked until the computation is complete. Stated differently, the offloaded part is executed synchronously with the host thread. If one wants to parallelize computations, he has to add other clauses such as *nowait* and *depend* that allow to decouple the execution of the target region from the host control flow. Those clauses are also supported by TOAST. Their use is illustrated later in subsection V-B.

The following runtime library routines are also implemented:

- `omp_target_alloc`
- `omp_target_free`
- `omp_target_is_present`

They are part of the runtime library and are used by the higher level memory management services that compose the runtime library. They are also available to the user as specified in the OpenMP standard.

B. Runtime library technical details

The runtime library is the part of the implementation that will be linked to the final host binary. As shown in Figure 5, the runtime library has the following roles:

- Manage the device memory
- Trigger execution of computations on the device

1) Device Memory Management: Memory allocations on the device are done through the RunTime Library (RTL). Memory can be manually allocated by the user by calling the `omp_target_alloc` service. It can also be allocated automatically by the RTL when it encounters a target or target data directive that requires memory allocation on the device. The RTL uses three structures to keep track of the state of the device memory:

- Memory page containing the address of the memory page in the device memory, the number of memory blocks used, and whether it is allocated or not.
- A hash table holding all references to memory pages created during the lifetime of the program, indexed by their address in memory.
- A set of free memory page addresses.

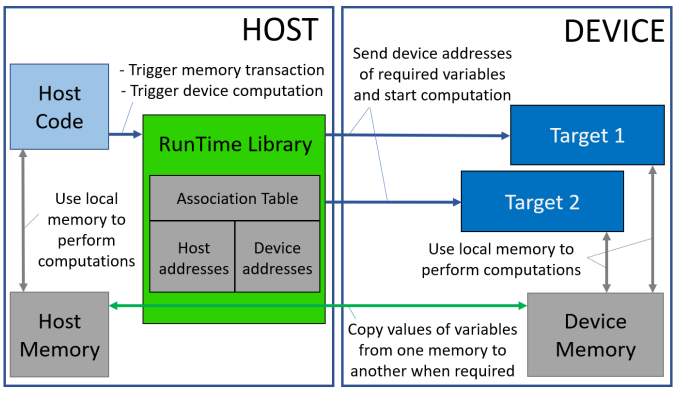

Fig. 5. Runtime library interaction with the *host* and the *device* at runtime

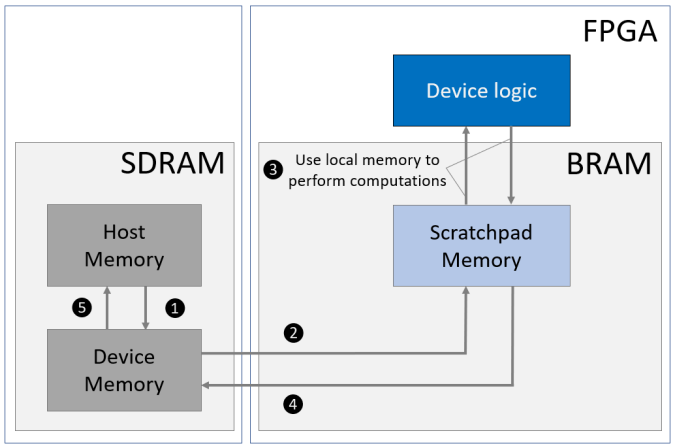

Fig. 6. TOAST Memory structure

The RTL uses two structures to keep track of memory associations between the host and the device as well as the reference count associated with each memory block in the device:

- A hash table holding references to memory pages associated with a host pointer (which is the index).
- A hash table holding the reference count of a given host pointer (therefore the reference count of the associated memory on the device).

These structures consume memory on the the host, but they also reduce the number of data copies between the host and the device at runtime. The interconnection between the host and the accelerators being a bottleneck, it is important to limit those copies as much as possible to get the benefits of offloading computations.

This constraint also applies to the choice of the memory used to implement communications. Device memory is allocated in the central SDRAM for it provides faster communications (and, also, larger storage) than the FPGA-located Block RAMs (BRAMs). This way the BRAM is only used for the scratchpad memory of the different accelerators that are generated. The memory transactions within this structure shown in Figure 6 are the following:

1) variables are copied from the Host memory into the Device memory by the Host, following the map clause if specified
2) variables are copied from the Device memory into the scratchpad memory by the accelerator
3) local scratchpad memory is modified by the accelerator execution.
4) variables are copied from the scratchpad memory into the Device memory by the accelerator
5) variables are copied from the Device memory into the Host memory by the Host, following the map clause if specified

2) Trigger Execution: As mentionned previously, the target device can be an FPGA or a CPU (e.g., a softcore). In both cases, the code to run on the device is extracted from the original source code, synthesized (or compiled), and loaded on the FPGA (or CPU) before execution. Note that, contrary to off-line offloading scheme implemented by TOAST, in the gcc implementation, the logic to be executed on the device when the device is a CPU is uploaded to the device at runtime. This scheme is extremely flexible but it can hardly be implemented on a FPGA. In addition, it makes temporal analysis (e.g., Worst-Case Execution Time estimation) much harder to achieve.

A typical OpenMP offloading implementation (such as the one from gcc or clang) can be summarized as follows:

1) The host asks the runtime library to run a target region
2) The runtime library updates the device memory state
3) The runtime library loads the associated binary on the target along with the device memory addresses of all variables accessed during the execution of the offloaded region
4) The runtime library launches the execution of the offloaded region
5) The runtime library updates the host memory state once the offloaded region has finished its execution

In our implementation, the RTL does not need a dedicated "trigger device computation" service. Instead, control flow is transferred to/from the device using the SpaceStudio API. Therefore, our offloading implementation can be summarized as follows:

1) The host asks the runtime library to update the device memory state
2) The host gets the addresses of the variables required to execute the offloaded region (from the RTL)
3) The host sends these addresses to the device
4) The host sends the signal to start the device computation
5) The host waits for the signal indicating that the device has finished its execution
6) The host asks the runtime library to update the host memory state according to the device memory state

V. Application and Evaluation

A. Offloading with one accelerator

To validate our approach, we have applied it to the development of a line detection program. The initial program, written in C++, implements a sequence of three steps: filtering, edge detection, and line detection. After profiling this mono-threaded program, the filtering step was found to be the most CPU-intensive, so we used TOAST to offload it to the FPGA. The resulting architecture is described in Figure 7. TOAST created the Filtering Space Module and added the communications to the host source code.

The original OpenMP code of the Filtering function is presented in Listing 1. The only difference from the original code is the `pragma` added on line 3 to offload the included computations. The `map` clause requires to copy the input

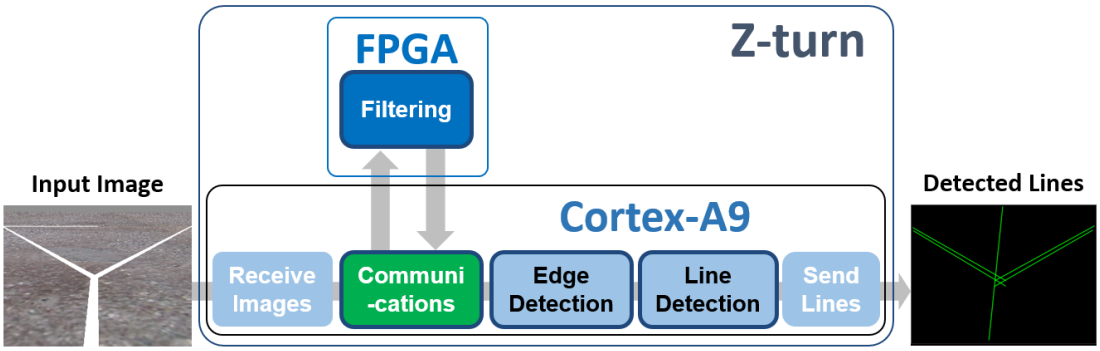

Fig. 7. TOAST HW/SW partitioning results applied to a Line Detection example

image to the FPGA before execution of the offloaded region. It is not copied back to the host after execution since the image is not changed by the algorithm. This clause also prevents to copy the initial `result_array` to the device memory, as the value of the result is only set by the device.

The code generated by TOAST for the host and target modules are given in Listing 2 and 3, respectively. The following elements are worth being commented (for an overview of the memory management, please refer to Section IV-B1).

On the host side (Listing 2):

- Lines 9–10: create the mappings to access the input and output images
- Lines 11–12: get the addresses of the input and output images
- Lines 13–17: send the addresses of the input and output images, and other parameters to the device
- Line 19: trigger the processing
- Line 20: wait until the processing is complete
- Lines 21–22: deallocate the memory mappings used to access the input and output images

On the target side (Listing 3):

- Lines 14–18: read the addresses of the input and output images, and other parameters
- Lines 19–20: copy the input and output images to the local scratchpad memory
- Line 22: wait for the execution trigger
- Lines 23–26: do the filtering operation
- Lines 27–28: copy the input and output image from the local scratchpad to the main memory
- Line 29: signal the end of the computation

For this simple example, TOAST generates the Space Module source files in less than a second thanks to the speed of the Clang Libtooling framework. The overhead of the transformation is negligible compared to the rest of the compilation process which takes around 15 seconds to compile the host

```
1 void line_detection::convolution_filter(int height,
      int width, int kernel_size, unsigned char*
      source_image, unsigned char* result_image) {
2
3 #pragma omp target map(always, to: source_image[0:
      IMAGE_SIZE]) map(from: result_image[0:IMAGE_SIZE
      ])
4 {
5     // convolution accessing source_image and
      writing results in result_image
6     ...
7 }
8 }
```

Listing 1. Original OpenMP source code

```
1  void fifo_write(T* host_ptr, int device_num);
2  void fifo_read(T* host_ptr, int device_num);
3  void map(T* host_ptr, int map_type, int map_modifier
       , int device_num,  int nb_elements, unsigned
       long offset);
4  void end_map(T* host_ptr, int map_type, int
       map_modifier, int device_num, int nb_elements,
       unsigned long offset);
5  ...
6
7  void line_detection::convolution_filter(int height,
       int width, int kernel_size, unsigned char*
       source_image, unsigned char* result_image) {
8
9  device.map(source_image, OMP_MAP_to, OMP_MOD_always,
        DEVICEMEMORY_ID, IMAGE_SIZE, 0);
10 device.map(result_image, OMP_MAP_from, OMP_MOD_none,
        DEVICEMEMORY_ID, IMAGE_SIZE, 0);
11 unsigned long source_image_addr0 = device.
       get_addr_in_device_memory(source_image);
12 unsigned long result_image_addr0 = device.
       get_addr_in_device_memory(result_image);
13 device.fifo_write(&source_image_addr0, 0);
14 device.fifo_write(&result_image_addr0, 0);
15 device.fifo_write(&kernel_size, 0);
16 device.fifo_write(&height, 0);
17 device.fifo_write(&width, 0);
18 int go_0;
19 device.fifo_write(&go_0, 0);
20 device.fifo_read(&go_0, 0);
21 device.end_map(source_image, OMP_MAP_to,
       OMP_MOD_always, DEVICEMEMORY_ID, IMAGE_SIZE, 0);
22 device.end_map(result_image, OMP_MAP_from,
       OMP_MOD_none, DEVICEMEMORY_ID, IMAGE_SIZE, 0);
23 }
```

Listing 2. Host code after TOAST transformation

```
1  void read_from_memory(T* local_ptr, unsigned long
       memory_address, int nb_elements, int elem_offset
       );
2  void write_to_memory(T* local_ptr, unsigned long
       memory_address, int nb_elements, int elem_offset
       );
3  ...
4
5  void Target0::thread(void) {
6      unsigned char source_image[IMAGE_SIZE];
7      unsigned char result_image[IMAGE_SIZE];
8      int kernel_size;
9      int height;
10     int width;
11     unsigned long source_image_addr;
12     unsigned long result_image_addr;
13     while (true) {
14     ModuleRead(HOST0_ID, SPACE_BLOCKING, &
       source_image_addr);
15     ModuleRead(HOST0_ID, SPACE_BLOCKING, &
       result_image_addr);
16     ModuleRead(HOST0_ID, SPACE_BLOCKING, &
       kernel_size);
17     ModuleRead(HOST0_ID, SPACE_BLOCKING, &height);
18     ModuleRead(HOST0_ID, SPACE_BLOCKING, &width);
19     read_from_memory(source_image, source_image_addr
       , IMAGE_SIZE, 0);
20     read_from_memory(result_image, result_image_addr
       , IMAGE_SIZE, 0);
21     int go;
22     ModuleRead(HOST0_ID, SPACE_BLOCKING, &go);
23     {
24     // convolution accessing source_image and
       writing results in result_image
25     ...
26     }
27     write_to_memory(source_image, source_image_addr,
        IMAGE_SIZE, 0);
28     write_to_memory(result_image, result_image_addr,
        IMAGE_SIZE, 0);
29     ModuleWrite(HOST0_ID, SPACE_BLOCKING, &go);
30     }
31 }
```

Listing 3. Target code after TOAST transformation

binary and the SystemC simulation of the hardware modules. After this phase, the complete design can be simulated, profiled, and evaluated at the functional level using SpaceStudio's own virtual platform. Coupled with OpenMP's flexibility to designate the code parts to be offloaded, this toolchain allows for a fast – hence large – architectural exploration process. Once the best architecture is identified on the basis of profiling figures, it takes around 15 minutes to generate the bitstream and test the complete application in its production environment on the target board.

The focus of this work was to obtain a functional validation of the application on the device. The example application has been successfully deployed on both a Xilinx XC7Z020 (ZedBoard from Digilent) and an Intel Cyclone V 5CSEMA5F31C6N (DE1-SoC from Terasic), both SoCs are fitted with a 667MHz dual-core ARM Cortex-A9 processor and an on-chip programmable logic area. At this phase of the experiment, no low-level optimizations have been applied (i.e., no increase of the clock frequency, no burst or DMA transfers, etc). Furthermore, the filter computation has only been offloaded, not parallelized. So, considering the performance of the host CPU (one dual-core ARM Cortex A9 at 667 MHz), performance increase can only be achieved by a better usage of the FPGA resource. This improvement is investigated in the next section.

B. *Offloading to multiple accelerators*

The example previously shown does a simple offloading, without parallelization. In order to improve performances, the initial program is slightly tweaked to dispatch the filter work on multiple accelerators, as shown in Listing 4.

This listing shows two target directives (starting line 15 and line 20, respectively), each one covering a call to the same function (`sub_filter`) but with specific `map` clauses designating the part of the initial image to be processed by each target. Each clause corresponding to one target, this implementation generates two accelerators, but it could easily

```
1  #pragma omp declare target
2  void sub_filter(int line_start, int line_end, int
       height, int width, int kernel_size, unsigned
       char* source_image, unsigned char* result_image)
        {
3      // convolution accessing source_image and
       writing results in result_image
4      ...
5  }
6  #pragma omp end declare target
7
8  void line_detection::convolution_filter(int height,
       int width, int kernel_size, unsigned char*
       source_image, unsigned char* result_image) {
9
10 #pragma omp parallel
11 {
12 #pragma omp master
13 #pragma omp target data map(always, to: source_image
       [0:IMAGE_SIZE]) map(from: result_image[0:
       IMAGE_SIZE])
14 {
15 #pragma omp target map(to: source_image[0:
       SLICE_SIZE + OVERLAP_SIZE]) map(from:
       result_image[0: SLICE_SIZE]) nowait
16     {
17         sub_filter(0, height / 2, height, width,
       kernel_size, source_image, result_image);
18     }
19
20 #pragma omp target map(to: source_image[SLICE_SIZE -
        OVERLAP_SIZE: SLICE_SIZE + 2 * OVERLAP_SIZE])
       map(from: result_image[SLICE_SIZE:SLICE_SIZE])
       nowait
21     {
22         sub_filter(height / 2, height, height, width
       , kernel_size, source_image, result_image);
23     }
24 }
25 }
26 }
```

Listing 4. Parallelized OpenMP source code

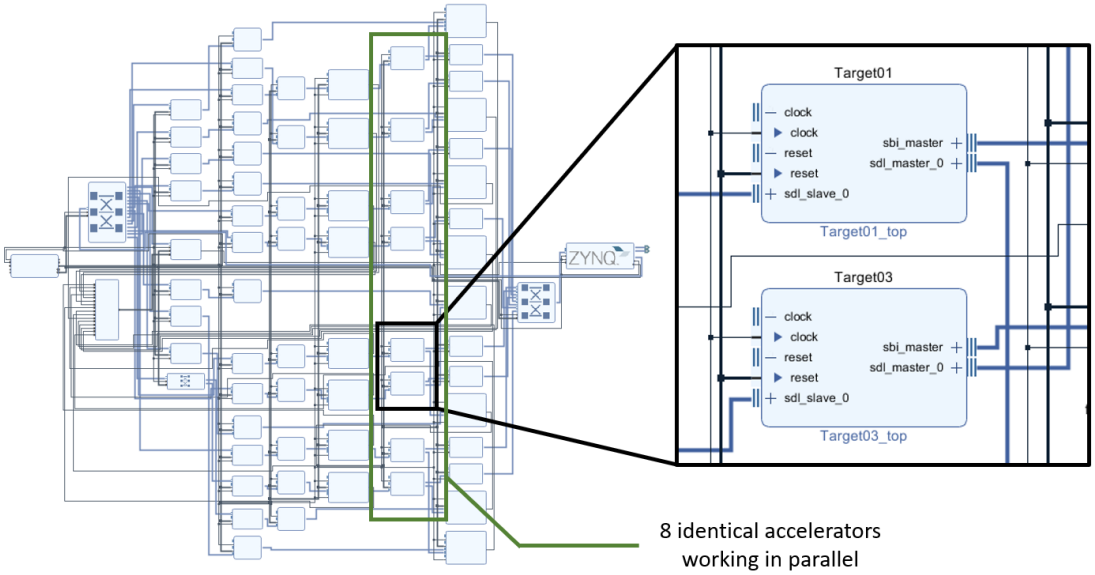

Fig. 8. Block design of the generated FPGA configuration with 8 accelerators

be extended to generate as many accelerators as supported by the FPGA. It is worth noting that each `target` directive uses the `nowait` clause in order to prevent the serialization of computations (i.e., call to accelerator n only once call to accelerator $n-1$ is complete).

Using the workflow presented in Figure 1, we have generated variants with 2, 4, 8 and 10 accelerators and used the SpaceStudio environment to estimate the FPGA occupation of each variant and generate its implementation on the target board.

The performance characteristics of the generated variants on the board are presented in Table I. At this stage, we chose to use 8 accelerators as adding more accelerators does not improve the performance (as shown by the data collected). We parallelize the filtering process by dividing the computation of the filter on slices of the image. However, the algorithm needs the preceding and following lines of the current line being processed to produce a result, meaning that, as we increase the number of accelerators computing a slice of the image, the number of overlapping lines that needs to be transferred to several accelerators increases, hence the total number of communications happening between the host and the accelerators. This phenomenon leads to diminishing returns when adding accelerators as they increase the time of communication more than the time gained by having lines processed concurrently on the FPGA.

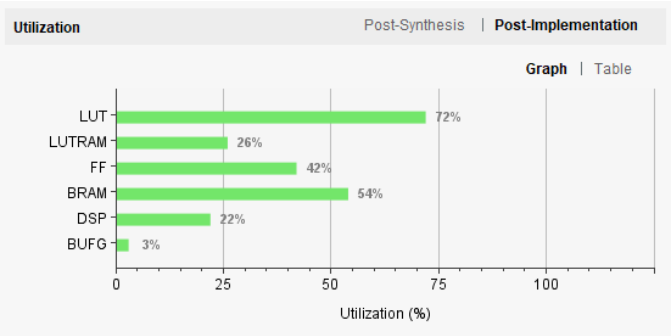

Fig. 9. Utilization of the generated FPGA configuration with 8 accelerators

TABLE I
PERFORMANCE OF VARIANTS

Accelerators	0 (ref)	1	4	8	10
filter time (ms)	61.4	202.3	63.9	40.9	41.1
acceleration (times)	1	0.3	0.96	1.5	1.49

The block representation of the generated design is shown in Figure 8, and its utilization of the target FPGA (ZedBoard) is shown in Figure 9.

Performance figures given in Table I show that at least 8 accelerators are needed to observe an acceleration with respect to the reference implementation. The performance vs. resource ratio is not as high as what would be achieved by manual VHDL or Verilog coding, but the performance vs. development cost is certainly much higher. Indeed, this result is obtained

1) with very few modifications of the source code
2) with very few manual operations, thanks to the automated tool chain,
3) using a high-level formalism (OpenMP) allowing a comprehensive and consistent expression and exploitation of application-level parallelism,
4) using a compilation chain and runtimes (OpenMP compliant compilers) allowing the deployment of the same application on diverse targets (GPU, acceleration boards, or FPGA),

5) with practically no knowledge of HDL programming.

Moreover, performances can actually be significantly improved by simply adding HLS pragmas to the source code. For instance, with the addition of two HLS pragmas in our algorithm, we were able to obtain a total execution time of the filter of 16.7 ms, which correspond to a 3.7x speedup.

This example illustrates the feasibility of the workflow presented in Section II. However, this demonstration has been performed on a simple, integer code. Will our workflow be applicable to applications making an extensive use of floating point operations (FPOps)? Indeed, deploying FP-extensive code on a FPGA is very resource demanding. In some cases, it may be necessary to re-code the offloaded part so that it uses fixed-point operations instead of FPOps. In that case, accelerating the application does not simply consists to select the code to be offloaded using OpenMP annotations and to press a button: the offloaded part must be re-coded. If the developer relies on the HLS compiler to handle floating point operations, he must ensure that the computed results are always "identical" to those that would be obtained on the host. Finally, performances gains achieved by offloading FP-intensive code may be limited due to the necessity to comply with the operation ordering in the C code in order to ensure identical results [6].

VI. Related works

Several academic works have addressed the problem of translating an OpenMP annotated code to an FPGA implementation. A comprehensive bibliography can be found in [7, Section B]).

In [8], Podobas *et al* propose a solution where an OpenMP code is transformed into custom FPGA hardware units orchestrated by a NiosII softcore. This approach differs from ours for three main reasons: (i) they translate OpenMP *tasks* (`#pragma omp task`), whereas we process OpenMP offloading directives (`#pragma omp target`); (ii) they provide a specific HLS chain that generates "hyper-tasks", whereas we use standard HLS chains provided by FPGA vendors (Xilinx or Intel) in order to leverage on the maturity of these products; (iii) they start HLS from the compiler's intermediate representation (IR) whereas we use a source-to-source transformation in order to maintain a direct traceability from the OpenMP code to the code provided as input to the HLS toolchain.

In [7], Sommer *et al* also process OpenMP target directives, and also uses commercial HLS tools to generate an FPGA implementation. The transformation relies on the ThreadPool-Composer (TPC) API (see [9]), later improved in TaPaSCo[4], whereas our workflow relies on the communication API implemented in the SpaceStudio environment. Leveraging the capabilities of SpaceStudio, our workflow allows to evaluate the performance of different deployment solutions starting from a unique OpenMP source code. It also allows to deploy to either Intel or Xilinx platforms.

Finally, in [10], Ceissler *et al* propose *HardCloud*[5], which extends OpenMP 4.x with clauses expressing the fact that the annotated software code will be replaced by either a pre-synthesized hardware implementation (mode 1) or an hardware implementation generated from source code (mode 2). Mode 2 seems to be similar to our solution, but as of the writing of this paper, only the first mode is described and supported.

VII. Conclusion

Software developers need appropriate programming paradigms and languages to expose the parallelism of their applications, and exploit efficiently the huge processing capabilities of the recent hardware platforms of the embedded market. OpenMP is a possible solution towards that goal thanks to the large scope of parallelization approaches supported, including the capability to offload parts of a software application to an external processing unit such as a GPU or an FPGA, to its very large user base, and its maturity. In domains where *size, weight* and *power* constraints are stringent, FPGA solutions are a promising alternative to general purpose CPUs or GPUs, as long as the transformation from software to hardware (i) does not mean re-coding manually the application in some low-level hardware description language such as VHDL or Verilog, and (ii) does not hinder the capability to explore various implementation solutions. To comply with conditions (i) and (ii), we have implemented a complete workflow based on two main components: TOAST, a new tool that extracts and "encapsulates" the OpenMP target code, and SpaceStudio, that provides Design Space Exploration and platform-independant implementation capabilities. Thanks to the independence of OpenMP with respect to the hardware platform, this solution can be used to deploy the same software first on a multi-core for early verification and functional validation activities, then on non-embeddable hardware accelerators (such as GPUs), and finally on FPGAs. We have presented the application of this workflow on a simple use case.

In the short term, we are completing TOAST to support the complete offloading capabilities of OpenMP and add optimizations to the generated code.

On the long term, several issues remain to be addressed. First, our workflow strongly relies on the capabilities of the HLS toolchain and of the target FPGA. If some significant part of code needs to be modified to be deployed on an FPGA, this means that some design choice must be done very early in the process, i.e., before the coding phase, and this would strongly affect the DSE capabilities of the approach. For instance, an application containing floating point operations may need to be rewritten using fixed point operations before being given to the HLS toolchain. Second, we strongly believe that OpenMP can make its way into the domain of safety critical

[4]Available at git.esa.informatik.tu-darmstadt.de

[5]See www.hardcloud.org.

embedded systems, but this is conditioned by the improvement of OpenMP determinism, which will require the application of specific implementation and usage constraints. Hopefully, several academic works have been carried out towards that goal (see e.g., [11]).

REFERENCES

[1] OpenMP Architecture Review Board. (2015) OpenMP Application Programming Interface Version 4.5. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[2] G. Bois. (2017) CPU plus FPGA design flow for software developers: A new tangible reality. https://www.embedded.com/electronics-blogs/say-what-/4458785/CPU-plus-FPGA-design-flow-for-software-developers–A-new-tangible-reality.

[3] (2019). [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[4] (2019). [Online]. Available: https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-software/qsys.html

[5] LLVM Project. (2018) Libtooling Documentation. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html

[6] J. Hrica, "Floating-PointDesign with Vivado HLS," XILINX, Tech. Rep. XAPP599, Sep. 2012. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp599-floating-point-vivado-hls.pdf

[7] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators," in *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*. IEEE, 2017, pp. 201–205. [Online]. Available: https://www.esa.informatik.tu-darmstadt.de/twiki/pub/Staff/AndreasKochPublications/2017_ASAP_LS_paper_preprint.pdf

[8] A. Podobas and M. Brorsson, "Empowering OpenMP with Automatically Generated Hardware," in *SAMOS XVI*, 2016. [Online]. Available: http://www.diva-portal.org/smash/record.jsf?pid=diva2:1033786

[9] J. Korinth, D. d. l. Chevallerie, and A. Koch, "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 195–198.

[10] C. Ceissler, R. Nepomuceno, M. Pereira, and G. Araujo, "Automatic Offloading of Cluster Accelerators," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO, USA: IEEE, Apr. 2018, pp. 224–224. [Online]. Available: https://ieeexplore.ieee.org/document/8457673/

[11] R. Vargas, E. Quinones, and A. Marongiu, "OpenMP and timing predictability: A possible union?" in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 617–620. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755893