Design Space Exploration in Heterogeneous Platforms Using OpenMP

Ángel Álvarez, Íñigo Ugarte, Víctor Fernández and Pablo Sánchez
Microelectronics Engineering Group
University of Cantabria
Santander, Spain
Email: {alvarez,ugarte,victor,sanchez}@teisa.unican.es

Abstract—**In the fields of high performance computing (HPC) and embedded systems, the current trend is to employ heterogeneous platforms which integrate general purpose CPUs with specialized accelerators such as GPUs and FPGAs. Programming these architectures to approach their theoretical performance is a complex issue. In this article, we present a design methodology targeting heterogeneous platforms which combines a novel dynamic offloading mechanism for OpenMP and a scheduling strategy for assigning tasks to accelerator devices. The current OpenMP offloading model depends on the compiler supporting each target device, with many architectures still unsupported by the most popular compilers, such as GCC and Clang. In our approach, the software and/or hardware design flows for programming the accelerators are dissociated from the host OpenMP compiler and the device-specific implementations are dynamically loaded at runtime. Moreover, the assignment of tasks to computing resources is dynamically evaluated at runtime, with the aim of maximizing performance when using the available resources. The proposed methodology has been applied to a video processing system as a test case. The results demonstrate the flexibility of the proposal by exploiting different heterogeneous platforms and design particularities of devices, leading to a significant performance improvement.**

I. Introduction

Heterogeneous computing architectures which combine general purpose multi-core CPUs and dedicated accelerators (e.g. GPUs and FPGAs) are becoming more and more common in both large data centers and embedded systems as they achieve better energy efficiency than CPU-only architectures [1]. Several programming models have emerged to help developers program these hardware accelerators, such as OpenCL [2] and CUDA [3]. Also, powerful and user-friendly synthesis tools (e.g. Vivado HLS from Xilinx) are available to generate FPGA-based accelerators from high-level languages, such as C, C++ and OpenCL.

A proper design methodology, which allows to identify the algorithm bottlenecks and efficiently map the set of tasks to the available computing resources, is of primary importance for high-performance execution of programs. Research on general-purpose multi-core CPU scheduling is abundant. A large number of scheduling heuristics targeting heterogeneous systems have also been presented in the literature. To cite just a few, Topcuoglu *et. al* in [4] presented some scheduling algorithms with the objective of simultaneously meet high performance and fast scheduling. In [5], Augonnet *et al.* developed various strategies that could be selected at runtime, outperforming static scheduling. In this paper, a methodology targeting acceleration of applications over heterogeneous architectures is presented. Our approach combines design-time and run-time strategies and features a simple heterogeneous scheduling algorithm.

After a heterogeneous scheduling is decided, tasks have to be moved to the corresponding computation resources. Computation *offloading* is a programming model in which the performance of a code is improved by transferring computation *kernels* from a host machine (e.g. CPU) to an accelerator device (e.g. GPU). OpenMP [6] is a well known API which is extensively used to program symmetric multiprocessor architectures (SMP). Support for offloading and a number of dedicated directives to define target regions and devices were introduced back in 2013 with the OpenMP 4.0 version. In order to use these offloading capabilities, the OpenMP compiler on the host machine must support code generation for the target accelerator device. The main drawback is that many architectures are still unsupported in the most popular compilers, such as GCC [7] and Clang [8]. Some previous researches have implemented code offloading from OpenMP annotated programs to accelerator devices. Most of them integrate offloading support for a certain architecture into the LLVM compiler infrastructure. For instance, Bertolli *et al.* [9] delivered offloading support for OpenPower systems targeting NVIDIA GPUs. Also, different optimization strategies were implemented into Clang with the aim of achieving CUDA-like performance. In [10], Antao *et al.* generalize the previous approach to allow compilation for multiple host and device types and describe their initial work to support code generation for OpenMP device offloading constructs in LLVM/Clang. Pereira *et al.* [11] designed an open source compiler framework based on LLVM/Clang which automatically converts OpenMP annotated code fragments to OpenCL/SPIR kernels. Finally, a proof-of-concept implementation of OpenMP offloading to FPGA devices integrated into the LLVM infrastructure was presented by Sommer *et al.* [12]. Compared to previous work, our proposal describes an alternative offloading methodology in which the device-specific compilation is independent from

This work has been funded by FEDER/Ministerio de Ciencia, Innovación y Universidades – Agencia Estatal de Investigación/TEC2017-86722-C4-3-R, also under the FitOptiVis Project (ECSEL2017-1-737451), which is funded by the EU (H2020) and *Ministerio de Ciencia, Innovación y Universidades.*

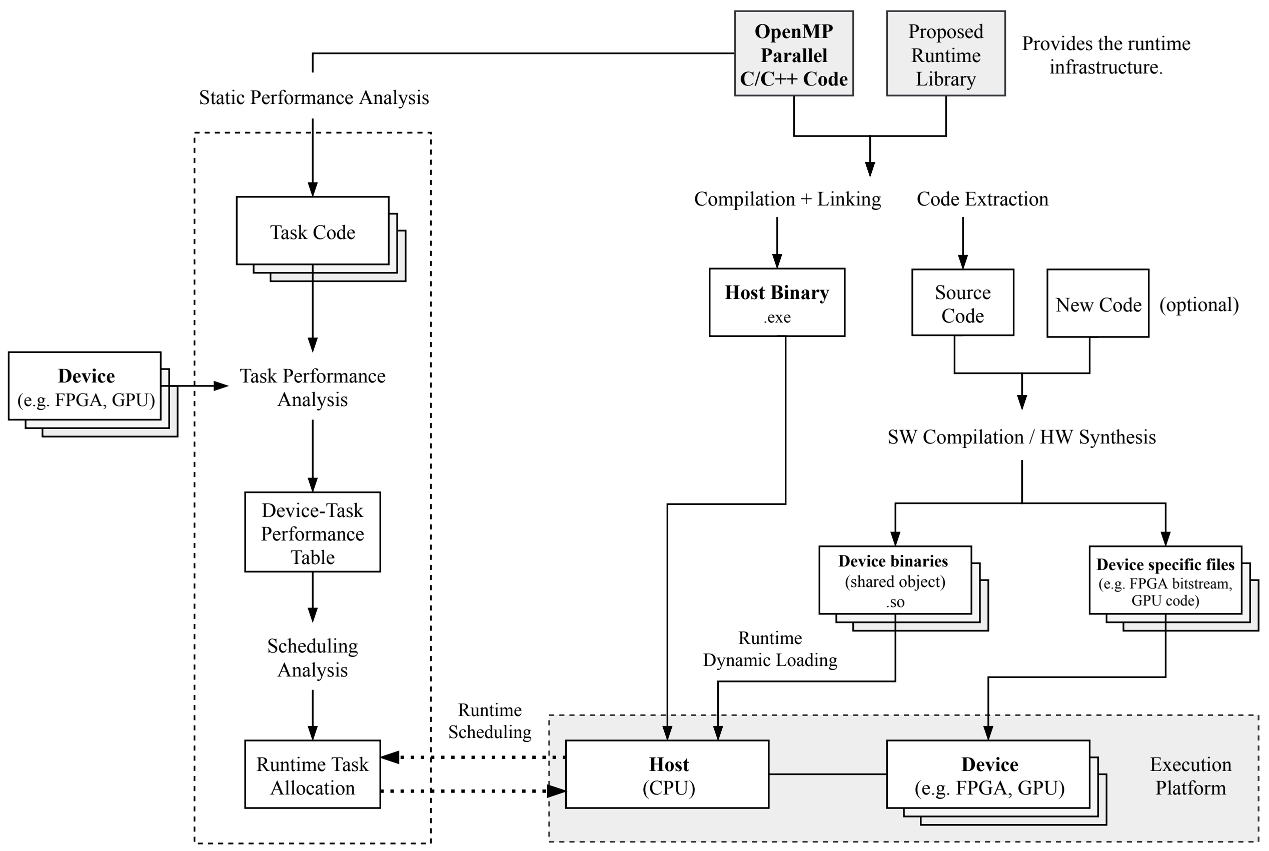

Fig. 1. Overview of the proposed design and execution methodology.

the OpenMP host compiler, thus requiring little compiler support and integration effort.

The rest of the paper is organized as follows. In Section II, the proposed methodology for accelerating applications over heterogeneous platforms is presented. Section III describes how tasks are distributed on the various processing units of a heterogeneous platform. In Section IV, the proposed methodology is applied to a case study video application and performance numbers are provided. Finally, the conclusions are drawn in Section V.

II. Methodology

This work introduces a novel methodology to efficiently accelerate applications over heterogeneous architectures. As explained in the previous section, one disadvantage of the current OpenMP offloading model is the dependence on the OpenMP host compiler to support each particular accelerator device architecture. In addition, this scheme does not allow the designer to manually optimize the device-specific implementations to achieve an effective performance improvement on the application. The key idea of our approach consists on dissociating the device-specific processes (software compilation and/or hardware synthesis) from the OpenMP host compiler. Since implementations for accelerator devices are generated from an independent design flow, a flexible runtime infrastructure has been developed to dynamically load and manage all available device implementations at runtime.

The design flow for the proposed methodology is summarized in Fig. 1. Consider a computing system with a host device (e.g. CPU) connected to one or multiple accelerator devices (e.g. GPUs, FPGAs). The starting point is a C/C++ source code which may be serial or parallel with various OpenMP threads concurrently executing different tasks. As shown in the left part of the figure, the application is profiled in order to obtain the execution time of the different tasks on the CPU of the target platform. If implementations corresponding to accelerator devices are available, their execution time is calculated as well. These results can be used to establish an *a priori* static scheduling in which tasks are associated to computing resources with the aim of minimizing the overall execution time.

The right part in Fig. 1 corresponds to compilation and execution processes. The original C/C++ source file must be compiled along with a library which provides a runtime infrastructure that allows the use of the proposed methodology.

The binary executable for the host CPU is generated. Also, the source code of the program regions marked for offloading can be extracted and used to obtain a device implementation (e.g. by using a high-level synthesis tool to generate an FPGA-based accelerator). At this point, the designer is able to manually optimize this code or even provide an alternative source for the design flow, such as VHDL/Verilog RTL descriptions or even third party IP cores for FPGA. A proof-of-concept implementation of OpenMP offloading to FPGA devices has already been shown in [12], in which Vivado HLS is used to generate the hardware. The authors claim that synthesis directives (pragmas) can be added to the original code to guide the HLS tool. However, the code cannot be modified with the aim of optimizing the generated hardware (e.g. buffers must be explicitly described in C/C++). As we show in Section IV, a deep understanding of the synthesis tool and applying both code modifications and synthesis directives are mandatory requisites to produce an efficient implementation.

Once that device implementations have been generated, some device-specific files may be produced, such as a bit-stream to configure the programmable logic of an FPGA device. On the other hand, some executable code for the host CPU is generated (e.g. software in charge of initializing hardware devices, transferring data to/from devices or launching the execution of offloaded tasks). This code is compiled into a shared object (i.e. a dynamic library). While the application executes, the developed runtime infrastructure is able to dynamically identify and load all the available implementations. A data table is built at runtime with information about the available devices and implementations, which includes data (such as the associated device, the number and type of arguments and performance numbers) and pointers to the executable code itself. The runtime system is able to provide dynamic task allocation while the application executes. The above makes it possible to perform a dynamic scheduling. More details on this are given in Section III. The flexibility of this approach allows to run the same application executable binary on platforms where devices are added/removed, or machines shared between many applications where devices may be busy. Moreover, new device implementations can be developed after compilation of the host code.

The main advantages of the presented methodology are: (i) multiple device architectures (such as GPUs and FPGAs from different families) can be used to offload program tasks, without depending on the compiler supporting each particular architecture; (ii) great flexibility is provided to the designers by enabling code modifications/optimizations and language/tool selection, with the aim of generating efficient implementations; and (iii) the runtime infrastructure allows to dynamically allocate tasks to computing resources after compilation of the host code, even if new implementations are loaded or the heterogeneous platform has changed.

III. Generic Task Scheduling

The offloading methodology presented in the previous section allows to dynamically map tasks to the available computational resources at runtime. In this section, a simple yet efficient heterogeneous scheduling strategy integrated into our runtime infrastructure is explained. While the application executes, the available device-specific implementations are dynamically loaded and a table of implementations for different tasks (i.e. potentially offloaded regions) is built, as represented in Table I. Every task has a CPU implementation available (known as *default*), corresponding to the first column in the table, and optionally device implementations for accelerators. Every element in the table is associated with a data structure, including metadata (such as target device and performance information used by the scheduler) and a pointer to the executable code.

TABLE I
Table of task implementations handled by the runtime.

	CPU	Device 1	...	Device N
Task 0	Impl. 0	Impl. 1	...	Impl. N
Task 1	Impl. 0	Impl. 1	...	Impl. N
⋮	⋮	⋮	⋮	⋮
Task K	Impl 0	Impl 1	...	Impl N

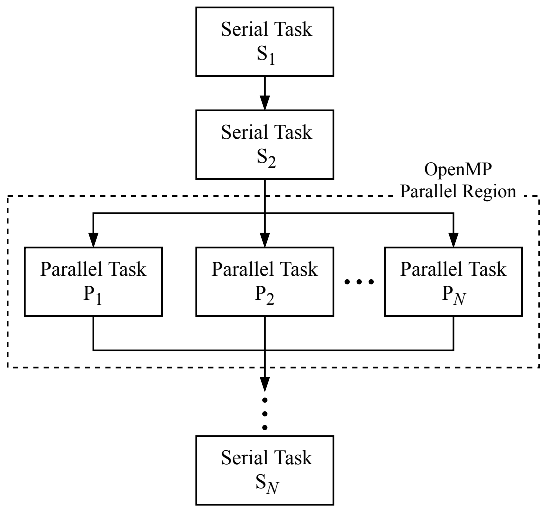

Fig. 2. Example execution flow of an application. Computational tasks can be potentially offloaded to accelerator devices.

Consider the execution flow of a generic application shown in Fig. 2, with execution time T_t given by (1). Then, the tasks are scheduled as follows. When a single thread is running on the system and a task is selected to be offloaded by the programmer, the implementation with the lowest execution time available is executed on the corresponding processing unit. When multiple threads concurrently execute different tasks, the most computationally complex task is given priority and accelerated on the hardware device associated with the fastest implementation available. The successive slowest implementations are sent to the devices associated with the fastest implementation available, given that the required computational resource is free. For instance, a GPU computing another task with higher priority would not be considered, but several tasks could be offloaded to the same FPGA device. The execution time of a set of concurrent tasks is limited by the slowest computation. When no accelerator devices are available/free, the default CPU implementation is executed.

$$T_t = T_{S_1} + T_{S_2} + \max\{T_{P_1}, T_{P_2}, \ldots, T_{P_N}\} + \cdots + T_{S_N} \quad (1)$$

IV. Case Study

In this section, a proof-of-concept of the proposed methodology is presented. In order to evaluate the runtime infrastructure, two heterogeneous architectures have been used: a Zynq UltraScale MPSoC (CPU-FPGA) and a PC (CPU-GPU).

A. System Description

As a test case, a commonly used video processing system is considered (see Fig. 3). Next, the different processing tasks are detailed.

1) RGB to grayscale conversion: First, an RGB color image (3 bytes per pixel) is read from a database. Then, the image is converted to grayscale (1 byte per pixel) by forming a weighted sum of the R, G and B color components.

2) Noise removal: A median filter is used to achieve a reduction of noise while preserving the edges in the image. This step obtains each pixel as the median value of a 3x3 pixel window centered on the considered position in the image. The median value is computed by sorting the elements in the window into numerical order and selecting the one on the middle position.

3) Edge detection: Then, a Sobel filter is applied as an edge detection algorithm. Two 3x3 kernels are convolved with the input image to get horizontal and vertical gradients. At each pixel in the image, the horizontal and vertical gradients are combined to obtain the gradient magnitude. The 8 most significant bits of the gradient are kept to produce a single channel grayscale image.

4) Scaling: In order to improve the visibility of the gradient magnitude images, a scaling which takes pixels closer to either white or black is performed. Intermediate pixel values in the range (0,128) are moved towards 0 (black). Intermediate pixel values in the range (128,255) are moved closer to 255 (white).

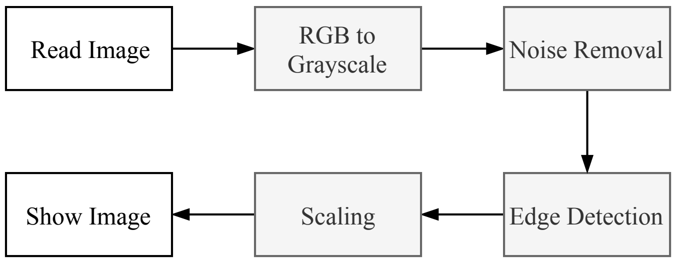

Fig. 3. Block diagram of the implemented video processing chain.

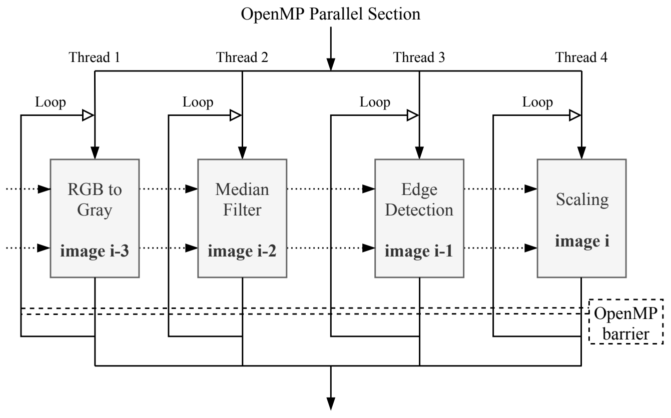

Fig. 4. Parallel implementation scheme for the video processing system.

B. System Implementation

As a starting point, an all-software version of the previous system has been developed. The proposed implementation is parallelized using OpenMP with four threads running on the host CPU, which concurrently executes the four processing tasks. Consequently, a pipeline is established with four images being processed at the same time. Since the output image of each stage is used as the input image of the following step, and the four stages are executing at the same time, a double buffer is used to communicate the modules. When a task (e.g. edge detection) is reading the first buffered imaged produced by the previous task (e.g. median filter), that previous task is writing in the second image buffer. As a consequence, memory coherence issues are avoided. The described implementation is depicted in Fig. 4.

C. Performance Evaluation

A detailed timing analysis is carried out in order to identify the bottlenecks of the all-sofware system on the platforms of interest. On the one hand, the Xilinx ZCU102 board features a Zynq MPSoC which integrates a 1.20 GHz quad-code ARM Cortex-A53 processor. On the other hand, the PC features a 2.30 GHz quad-core Intel Core i7-3610QM processor. Tables II and III summarize the CPU-only execution time of the previously described video processing tasks on these two platforms. The results show that the median filter stage has the biggest computational load on both architectures.

TABLE II
Detailed CPU Execution Time on Xilinx ZCU102

CPU	**Quad-core 1.20 GHz ARM Cortex-A53**
RGB to Grayscale	0.018 s
Median Filter	0.927 s
Edge Detection	0.099 s
Scaling	0.048 s

TABLE III
DETAILED CPU EXECUTION TIME ON PC

CPU	**Quad-core 2.30 GHz Intel Core i7-3610QM**
RGB to Grayscale	0.021 s
Median Filter	0.536 s
Edge Detection	0.062 s
Scaling	0.010 s

TABLE IV
MEDIAN FILTER HW IP PERFORMANCE ESTIMATES AFTER HIGH-LEVEL SYNTHESIS

Input to Vivado HLS	Ordinary C/C++	HW-oriented C/C++
Image resolution	1920x1080	1920x1080
Clock period	3.08 ns	4.58 ns
Clock cycles	360810722	2073609
IP throughput	**0.90 fps**	**105.29 fps**

TABLE V
SOBEL FILTER HW IP PERFORMANCE ESTIMATES AFTER HIGH-LEVEL SYNTHESIS

Input to Vivado HLS	Ordinary C/C++	HW-oriented C/C++
Image resolution	1920x1080	1920x1080
Clock period	3.85 ns	4.52 ns
Clock cycles	360810965	2073642
IP throughput	**0.72 fps**	**106.69 fps**

D. Device-Specific Implementations

In the previous section, some bottlenecks were detected in the system, mainly the median filter and the edge detection (Sobel filter) algorithms. Device-specific implementations have been designed in order to accelerate these functions on dedicated hardware.

1) GPU Implementations: OpenCL kernels (i.e. functions describing parallel execution on the device) have been generated to offload the median filter and the Sobel filter to a GPU. OpenCL kernels are defined in a C-based language called OpenCL C.

2) FPGA Implementations: Xilinx SDSoC has been used to produce the driver functions for the host, which are dynamically loaded by the runtime, and the files to configure the Zynq MPSoC device. Within the design flow, Vivado HLS is used by SDSoC to generate hardware IP cores from C/C++. Although high-level synthesis tools promise software-like development to generate hardware from pure high level code, the designer is usually required to significantly change and adapt the code in order to obtain an efficient implementation. Since one of the key points of our offloading methodology is allowing the designer to intervene in the design process for programming the accelerators, we will next justify this need with some synthesis details. SDSoC automates the connection of the generated HW IPs with the processing system through an AXI4 bus. The user IPs are connected to an AXI DMA IP to read from/write to the main system memory.

Although computation intensive algorithms such as the median and Sobel filters are natural candidates for implementation in hardware, synthesizing the original CPU-oriented C/C++ code generates cores with throughputs below 1 frame per second. The performance estimates of the synthesized IP cores are summarized in tables IV and V. As we show, good C/C++ code for a CPU or even a GPU may generate a poor implementation for an FPGA or ASIC. The low performance is due to the lack of an efficient memory architecture designed for such memory intensive algorithms as the ones we are facing. The high-level synthesis tool does not automatically introduce or manage memory buffers. The hardware designer must explicitly describe those structures in the code so that they are generated into the RTL.

In order to boost the performance, a double buffer memory architecture has been implemented, as suggested in [13] and described in Fig. 5. Both algorithms compute with data from a 3x3 pixel window. When pixels are sent to the hardware IP in the FPGA logic, they are pushed into a buffer capable of storing three lines of pixel data which acts as a shift register. Secondly, a window buffer which stores 9 elements from the three buffered lines is used. The algorithms compute with the elements of the window. A synthesis directive is applied to implement the window memory as flip-flops, so that all pixels are available to compute in a single clock cycle. The three-line buffer has been defined as three arrays and implemented as three different dual-port block RAMs since it requires simultaneous read and write access. By combining proper synthesis directives with the described code modifications, the throughput achieved by the generated hardware goes above 100 frames per second (x117 with respect to software-like code synthesis).

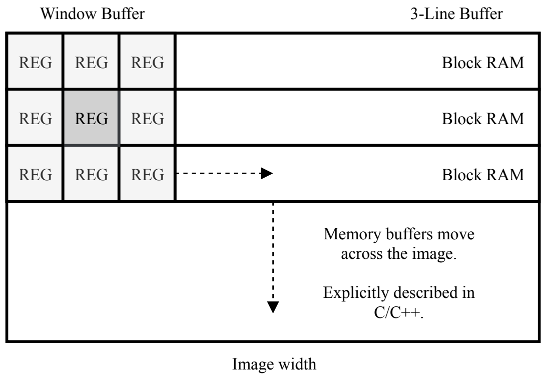
Window Buffer
3-Line Buffer
REG
Block RAM
Memory buffers move across the image.
Explicitly described in C/C++.
Image width

Fig. 5. Memory structures implemented to get efficient HLS implementations.

E. Experimental Results

The system has been tested over two heterogeneous architectures: (i) a Xilinx ZCU102 board featuring a Zynq

TABLE VI
OVERALL PERFORMANCE ON XILINX ZCU102 - ZYNQ MPSOC ARM CORTEX-A53 CPU + FPGA

Implementation	Frame Rate	Speed-up
Serial (CPU-only)	0.92 FPS	x1.0
Parallel (CPU-only)	1.13 FPS	x1.2
Parallel + Offloading (FPGA)	21.36 FPS	x23.2

TABLE VII
OVERALL PERFORMANCE ON PC - INTEL CORE I7-3610QM CPU + NVIDIA GT630M GPU

Implementation	Frame Rate	Speed-up
Serial (CPU-only)	1.60 FPS	x1.0
Parallel (CPU-only)	1.84 FPS	x1.2
Parallel + Offloading (GPU)	15.31 FPS	x9.6

UltraScale MPSoC with 1.20 GHz 4 cores ARM Cortex-A53 CPU integrated with FPGA programmable logic and (ii) a laptop PC with 2.30 GHz 4 cores Intel Core i7-3610QM CPU and a NVIDIA GT630M GPU, both running Linux.

Tables VI and VII summarize the overall execution times on these platforms for different implementations of the test case. Serial implementation corresponds to the all-software version of the diagram in Fig. 3 executing on the CPU (single thread). Parallel implementations correspond to the diagram in Fig. 4, which concurrently executes 4 threads on the host (CPU). Parallel implementation with offloading means that tasks with available implementations are sent to accelerator devices by the runtime infrastructure. In the case of the Zynq MPSoC, the median and Sobel filters have been offloaded to the FPGA logic. In the case of the PC, the median filter has been moved to the GPU (note that the GPU is not available for any other concurrent task). The processing has been applied to images with 1920x1080 resolution and the results are averaged over 100 executions. Final implementations with offloading show significantly lower execution times than either serial or parallel CPU-only versions. When compared to serial CPU implementations, a x9.6 speed-up was achieved on PC (CPU-GPU) and a x23.2 speed-up was reached on Xilinx ZCU102 with the Zynq SoC (CPU-FPGA).

V. CONCLUSION

This paper introduces and evaluates techniques for acceleration of OpenMP-annotated applications in heterogeneous platforms. This work overcomes some limitations of the standard OpenMP offloading model (e.g. limited devices are supported by popular compilers, like GCC and Clang). New features are also incorporated, such as dynamic loading of accelerator implementations and dynamic mapping of tasks to computing resources. Our proposal requires that the design and execution flows follow a new offloading methodology which dissociates the accelerator specific compilation and/or synthesis from the host OpenMP compiler. In order to enable the use of the above methodology and scheduling features, a runtime infrastructure has been developed. The presented approach has been evaluated using a video processing system as a test case over two heterogeneous architectures. The application has been parallelized with multiple OpenMP threads running on the host CPU. In the algorithm, some bottlenecks have been detected and offloaded (i.e. moved) to accelerators at runtime, depending on the available resources. When using FPGA accelerators, it has been proven the importance of letting the hardware designer modify the code taken by the HLS tool in order to generate efficient implementations. Analyzing the overall application performance against the starting CPU-only implementation, x9.6 and x23.2 speed-ups were achieved on PC (CPU-GPU) and Xilinx ZCU102 with the Zynq SoC (CPU-FPGA), respectively.

ACKNOWLEDGMENT

This work has been funded by FEDER/Ministerio de Ciencia, Innovación y Universidades – Agencia Estatal de Investigación/TEC2017-86722-C4-3-R, also under the FitOptiVis Project (ECSEL2017-1-737451), which is funded by the EU (H2020) and *Ministerio de Ciencia, Innovación y Universidades – Agencia Estatal de Investigación*.

REFERENCES

[1] M. Horowitz, "Computing's energy problem (and what we can do about it)", *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*,
[2] Khronos Group, "OpenCL: The open standard for parallel programming of heterogeneous systems", 2010, https://www.khronos.org/opencl/.
[3] NVIDIA, CUDA — Compute Unified Device Architecture, https://developer.nvidia.com/cuda-zone.
[4] H. Topcuoglu, S. Hariri and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, March 2002.
[5] Augonnet, C. , Thibault, S. , Namyst, R. and Wacrenier, P., "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures", *Concurrency and Computation: Practice & Experience*, pp. 187-198, Volume 23 Issue 2, February 2011
[6] Open MP API Specification. Version 5.0 November 2018 https://www.openmp.org/specifications/.
[7] Offloading support in GCC, https://gcc.gnu.org/wiki/Offloading
[8] Clang 9 documentation: OpenMP support. https://clang.llvm.org/docs/OpenMPSupport.html
[9] C. Bertolli, S. F. Antao, G. T. Bercea, A. C. Jacob, A.E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, K. O'Brien, "Integrating GPU Support for OpenMP Offloading Directives into Clang", *LLVM-HPC2015*, Austin, Texas, USA, November 15-20, 2015.
[10] S. F. Antao, A. Bataev, A. C. Jacob, G. T. Bercea, A.E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, K. O'Brien, "Offloading Support for OpenMP in Clang and LLVM", *LLVM-HPC2016*, Salt Lake City, Utah, USA, November 13-18, 2016.
[11] M. Pereira, R. Sousa and G. Araujo, "Compiling and Optimizing OpenMP 4.X Programs to OpenCL and SPIR", *13th International Workshop on OpenMP (IWOMP)*, Stony Brook, NY, USA, September 20-22, 2017.
[12] L. Sommer, J. Korinth and A. Koch, "OpenMP device offloading to FPGA accelerators", *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Seattle, WA, 2017, pp. 201-205.
[13] F. M. Vallina, "Implementing Memory Structures for Video Processing in the Vivado HLS Tool", *Xilinx Application Note XAPP793 (v1.0)*, September 20, 2012.