OpenMP Dynamic Device Offloading in Heterogeneous Platforms*

Ángel Álvarez[0000−0003−2631−3817], Íñigo Ugarte[0000−0003−2586−2339], Víctor Fernández[0000−0003−0614−151X], and Pablo Sánchez

Microelectronics Engineering Group, University of Cantabria, Spain
{alvarez,ugarte,victor,sanchez}@teisa.unican.es

Abstract. Heterogeneous architectures which integrate general purpose CPUs with specialized accelerators such as GPUs and FPGAs are becoming very popular since they achieve greater performance/energy tradeoffs than CPU-only architectures. To support this trend, the OpenMP standard has introduced a set of offloading constructs that enable to execute code fragments in accelerator devices. The current offloading model heavily depends on the compiler supporting each target device, with many architectures still unsupported by the most popular compilers (e.g. GCC and Clang). In this article, we introduce a new methodology for offloading OpenMP annotated code to accelerator devices. In our proposal, the software compilation and/or hardware synthesis processes to program the accelerator are independent from the host OpenMP compiler. As a consequence, multiple device architectures can be easily supported through their specific compiler/design tools. Also, the designer is able to manually optimize the original offloaded code or provide an alternative input to the design flow (e.g. VHDL/Verilog or third party IP cores for FPGA), thus leading to an effective speed-up of the application. In order to enable the proposed methodology, a powerful runtime infrastructure that dynamically loads and manages the available device-specific implementations has been developed.

Keywords: OpenMP · Offloading · GPU · FPGA.

1 Introduction

Heterogeneous computing architectures which combine general purpose CPUs and dedicated accelerators such as GPUs and FPGAs have become extensively used both in large processing centers and on embedded systems. These platforms outperform homogeneous multi-core CPU systems in terms of computing capabilities and especially energy efficiency (operations per watt) [1]. In order to facilitate the design for hardware accelerators, programming models such as

* This work has been funded by EU and Spanish MICINN through project ECSEL2017-1-737451 and Spanish MICINN through project TEC2017-86722-C4-3-R.

OpenCL [2] and CUDA [3] have emerged and powerful hardware synthesis tools have been introduced by the industry to enable the use of high-level languages such as C, C++ and OpenCL to generate FPGA-based accelerators.

The performance of a code executing on a CPU (*host device*), can be improved by *offloading* a code fragment (*target region* or *kernel*) to a hardware accelerator (*target device*), like a GPU or FPGA. Since its origin, OpenMP has proven to be an efficient and widely used model for programming shared-memory symmetric multiprocessor (SMP) architectures. In recent versions, the standard [4] has introduced a set of extensions to support code offloading to accelerators. This model relies on the compiler to support the generation of the executable code for the accelerator device. The implementation of the OpenMP offloading features in GCC [5] and Clang [6] is still under development, with many architectures still unsupported.

This paper introduces a new offloading methodology which allows both large compatibility with different device architectures and flexibility in the design of the computation kernels. In our approach, the SW compilation/HW synthesis and (optionally) design flows for the accelerator device are independent from the OpenMP compiler. In order to support the above, a flexible and interoperable runtime infrastructure has been developed, which fully integrates with the standard OpenMP runtime.

The rest of this paper is organized as follows. In Section 2, the proposed offloading methodology is introduced. Section 3 describes the implemented runtime infrastructure, which we evaluate over some heterogeneous architectures in Section 4. Section 5 provides related work. Finally, Section 6 concludes the paper and discusses future work.

2 Methodology

2.1 Motivation

As explained in the previous section, one drawback of the current offloading process in OpenMP is the fact that target devices must be supported by the OpenMP compiler. Also, this scheme leaves the designer with very little or no flexibility to modify the design in some scenarios such as offloading to hardware accelerators, in which specific optimizations as well as code/algorithm modifications are required to generate efficient implementations.

This work focuses on the development of a new OpenMP offloading methodology. The key idea of our approach is to dissociate the OpenMP compiler from device specific compilation/synthesis processes and provide an efficient mechanism to integrate device implementations with the host executable during runtime. In order to make it possible, a runtime infrastructure which integrates with the OpenMP runtime is developed.

The elementary requirements that the proposed infrastructure has to meet are summarized as follows:

1. Allowing the development of new device implementations after compilation of the host code. Runtime mechanisms are defined for dynamic loading of the new device-specific implementations.
2. Enabling the runtime infrastructure to identify, during execution time, all the available implementations as well as computing resources required to execute them.
3. Enabling the runtime infrastructure to provide dynamic task allocation during execution time. The designer will be able to use runtime library routines to set the target device.
4. Allowing device-specific implementations to optionally include performance metadata, like memory requirements, execution time or power consumption. Similarly, identified computing resources may include information such as memory size and clock frequency. This could be used to guide device selection at runtime. In order to use this information, new OpenMP runtime functions should be defined.

Let offloading to FPGA serve as an example of application. A proof-of-concept implementation of OpenMP offloading to FPGA which integrates with the LLVM offloading infrastructure has already been presented in [7]. It uses Vivado HLS to generate the hardware from the C/C++ original code. Despite the fact that the designer can add synthesis directives (pragmas) in the original code to be used by the high-level synthesis tool, the code cannot be modified with the aim of optimizing the generated hardware. In practice, it is well known by hardware designers that a deep knowledge of the synthesis tool and wisely modifying the input code (along with the use of directives) are key points to get an efficient hardware design.

Our approach is based on generating a host binary which integrates device-specific implementations during runtime and breaking apart the device code compilation flow. Then, the original code of the OpenMP target regions can be used as an input to the HLS tool though the designer is able to get into the design flow and generate an optimized code as well. Moreover, hardware description languages such as VHDL or Verilog or even extern IP cores can be used depending on the designer preferences. The integration of these implementations is supported by the proposed runtime infrastructure. In addition, high flexibility in terms of supported devices is provided since designers use device-specific compilers or synthesis tools no matter whether they are supported by the current OpenMP compilers.

2.2 Target Platforms and Supported Devices/Accelerators

With the increasing importance of heterogeneous platforms which integrate CPUs, GPUs and FPGA-based hardware accelerators, supporting as many targets as possible is at the core of our methodology. From OpenMP API 4.0 (released in 2013) some directives to instruct the compiler and runtime to offload a region of code to a device are available to the programmer [4,8]. However,

support for the target devices must be included into the compiler infrastructure in order to allow device offloading. In practice, offloading support in the most commonly used compilers is still immature [5,6]. In our approach, by making the device-specific design flow independent from the OpenMP compiler, it is possible to offload a code region to almost any target provided that device compilers or synthesis tools are available to the designer. We will focus on proving the compatibility of the proposed methodology with: i) GPUs, which can be programmed through OpenCL or propietary languages such as CUDA, and ii) FPGA devices, through high-level synthesis or hardware description languages.

2.3 Offloading Design Flow

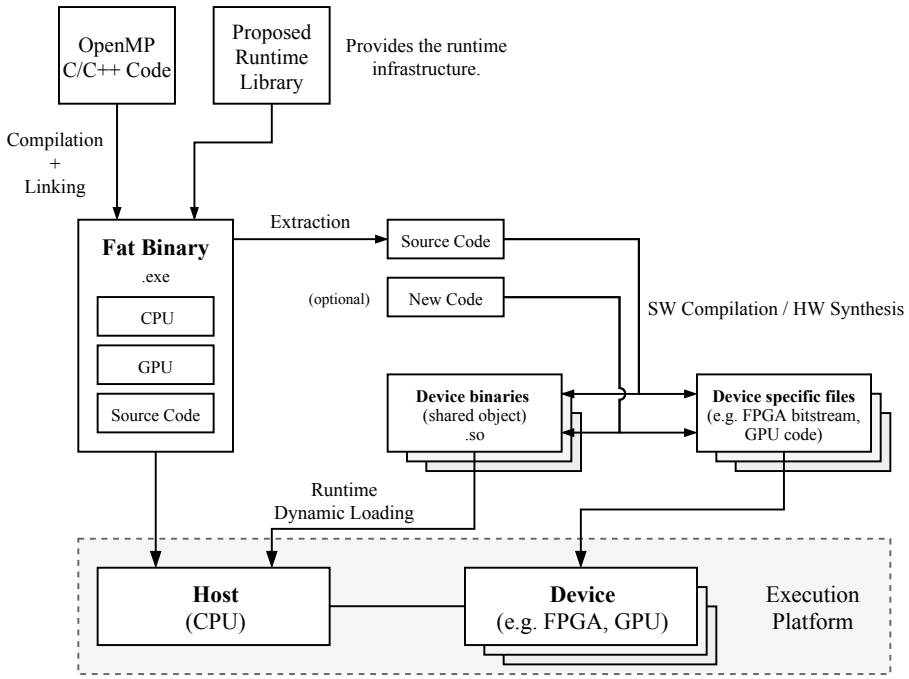

Fig. 1. Design flow for the proposed OpenMP dynamic device offloading methodology.

Consider a computation node with a host device (CPU) connected to one or multiple accelerators. The starting point in the OpenMP accelerator model flow is a source file with standard OpenMP code, in which the region of code (known as *target region*) to be offloaded to an accelerator (known as *target device*) is specified by the `target` directive.

In our proposal (summarized in Fig. 1), the original input source file has to be compiled and linked with a library which provides a runtime infrastructure

to allow the use of the new dynamic offloading methodology. This infrastructure integrates with the standard OpenMP runtime and will be detailed in section 3. When a compiler supports offloading to a certain architecture, a binary for each target is commonly inserted into the host fat binary file. In order to support any potential target device, the source code corresponding to the target region has to be included in the fat binary as well. Then, this code is extracted to be used as the input of a separate compilation/synthesis flow. Automation of these steps and integration of the proposed methodology into an existing OpenMP open-source compiler are out of the scope of this paper.

The compilation/synthesis of the target code for the accelerator device are dissociated from the OpenMP compilation process in the presented methodology. On the one hand, the original source code of the target region can be directly used to program the accelerator (e.g. as the input to a high-level synthesis tool to generate an RTL design for FPGA). Also, the OpenMP code may be converted to an OpenCL kernel to be executed on a GPU/FPGA [12]. On the other hand, the designer has the possibility of modifying the code to achieve an effective speed-up of the application in a particular device or even taking a different approach, such as VHDL or Verilog in the case of FPGA or CUDA for a GPU from NVIDIA. This flexibility is one of the biggest advantages of the proposed methodology. Compilation/synthesis for the accelerator are carried out prior to the program execution. Compilation at runtime, also known as just-in-time (JIT) compilation, would not be feasible in terms of performance (e.g. FPGA bitstream generation is too time-consuming) and flexibility (the designer must be able to provide new code). New device implementations can be generated and new accelerators can be supported without recompiling the host code.

In order to run the target region in the accelerator device, the necessary executable code for the host is generated in the form of a shared object (i.e. a dynamic library). The tasks performed by these shared objects include managing the device status, the data transfer and the execution on the device. They are not inserted into the fat binary — instead, they are designed to integrate with the original host binary during runtime. Also, some device specific files can be produced in the design process, such as a bitstream to configure an FPGA device. Different implementations to accelerate the target region in multiple devices may be available. A single shared object can contain different implementations, or individual shared objects corresponding to each accelerator can be used (e.g. `lib_GPU.so`, `lib_FPGA.so`, etc). During execution time, the runtime infrastructure is able to identify all the available devices and implementations.

3 Proposed Runtime Infrastructure

In this section, the features and implementation details of the proposed runtime infrastructure are presented. First, we illustrate how it can be used from a programmer's point of view. Then, internal implementation details are given.

3.1 Programmer's Perspective

The runtime library provides the programmer with a set of routines to select and check the accelerator device during execution time. Their functionality could be added to their OpenMP counterparts (see Table 1). It is fully interoperable with the OpenMP runtime and all functions are designed to have a C binding, so that it supports C and C++. An example of the use of the runtime library to offload a code region from a programmer's perspective is shown in Listing 1. In the code, two concurrent OpenMP host threads are created, with identifiers '0' and '1'. In thread 0, some code in `function1` will be executed on the CPU. In thread 1, some compute-intensive code is marked for offloading with the `omp target` directive. This code will be moved to the accelerator with device number '2', which is assumed to be an FPGA in the execution platform. In the example, the `map` clause has been used to explicitly indicate the variables to be copied to and from the device data environment.

Table 1. OpenMP runtime library routines modified to enable the new methodology.

Function	Description
`void omp_set_default_device(int device_num)`	Selects the default target device.
`int omp_get_default_device(void)`	Returns the default target device.
`int omp_get_num_devices(void)`	Returns the number of target devices.

3.2 Implementation Details

As a result of the proposed design flow, two kinds of files are used at execution time:

The host binary (required), which includes the original code and an implementation corresponding to every target region marked for offloading for, at least, the CPU. Implementations for other devices may be included into the executable as well when supported by the OpenMP compiler.

Shared objects (optionally), which include implementations for one or multiple additional target devices, corresponding to one or various of the target regions marked for offloading.

During the host program execution, the runtime infrastructure is initialized the first time that a runtime routine or target region is executed. The available shared objects containing device implementations are loaded and some runtime lists are built : (i) a list of devices, (ii) a list of target-region functions and (iii) a list of implementations for each function. This data structures are shown in Fig. 2 and explained below.

Listing 1. Device offloading example with the proposed infrastructure.

```
#define GPU  1
#define FPGA 2

void host_code(char *image_in, char *image_out, int width, int height)
{
    UC_set_default_device(FPGA);
    omp_set_num_threads(2);

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        // ----Thread #0----
        if(id == 0){
            function1();
        }
        // ----Thread #1----
        if(id == 1){
            #pragma omp target map(to: image_in[0:width*height], width,
                 height) map(from: image_out[0:width*height])
            {
                int x, y;
                for(y=0; y<height; y++){
                    for(x=0; x<width; x++){
                        // Some computation
                        image_out[y*width + x] = image_in[y*width + x] * 0.5;
                    }
                }
            }
        }
    }
}
```

Devices. The accelerators devices supported by the host compiler and the ones defined in loaded shared objects are added to the global list of devices. Device 0 corresponds to the CPU and is always present. Every element in the list of devices contains metadata (such as name, type, status...) and pointers to device-specific management functions, which are detailed in subsection 3.3. Also, performance characteristics can be included (number of cores, memory size...). The above may be useful to add new functionality to the OpenMP runtime in the future, such as guiding the device selection process during execution.

Functions. For each target region marked for offloading with the `target` directive in the code, a *target-region function* is extracted and added to the global list of functions. Every element in the list contains information related to the arguments of the function (number, type and direction) and points to a list of implementations targeting one or more target devices (al least, the default CPU version is available).

Implementations. The available implementations for each *target-region function* are added to the list of implementations. Every element in the list contains data (such as the target device), pointers to implementation-specific management functions (explained in subsection 3.3) and a pointer to the executable

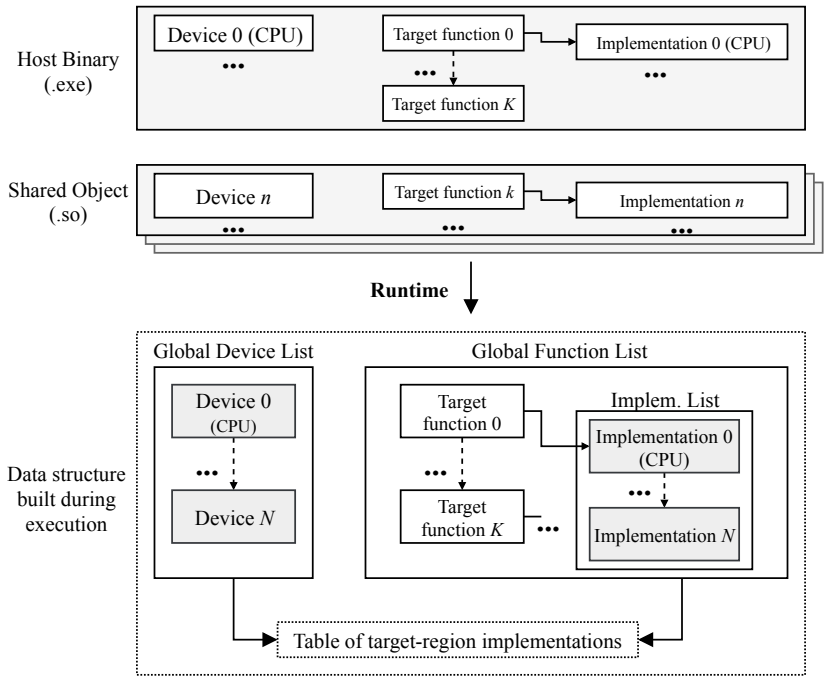

Fig. 2. Overview of data structures built by the runtime infrastructure.

code. Overall, the information handled by the proposed runtime method is organized in a *table of target-region implementations*, as represented in Table 2. In this version of the runtime, each device is associated with only one implementation and vice versa (i.e. for target region 0—first row in the table—`Impl.(0,0)` corresponds to `Device 0`, `Impl.(0,1)` to `Device 1`...). When the device required for offloading of a target region does not have an implementation available, the default implementation (`Implementation 0`) is launched on the CPU. The *table of target-region implementations* allows to integrate our proposal with the current OpenMP offloading methodology since it uses a similar approach.

Table 2. Table of target-region implementations handled by the runtime.

	Device 0	Device 1	...	Device N
Function 0	Impl. (0,0)	Impl. (0,1)	...	Impl. (0,N)
Function 1	Impl. (1,0)	Impl. (1,1)	...	Impl. (1,N)
⋮	⋮	⋮	⋮	⋮
Function K	Impl. (K,0)	Impl. (K,1)	...	Impl. (K,N)

3.3 Management of Devices and Implementations

For every device and implementation, a set of functions are provided to manage and configure the accelerator and the execution during runtime. These functions are summarized in Table 3. The runtime infrastructure internally employs these functions, although the tasks they perform are specific for each device/implementation. Owing to that reason, they are defined in the shared objects containing device implementations (default versions are in the host binary as well).

Table 3. Internal runtime routines to manage devices and implementations.

	Function	Description
Device	`open_device()`	Checks if the device is in the execution platform. If present, initializes the device. Allocates memory in the host to store device data.
	`close_device()`	Releases the device. Deletes device data stored in the host memory.
	`lock_device()`	Disables access to the device from other host thread.
	`unlock_device()`	Enables access to the device from other host thread.
Implementation	`init_implementation()`	Initializes the implementation (e.g. allocates memory in the device).
	`close_implementation()`	Clears the implementation (e.g. deallocates memory in the device).

Fig. 3 shows the execution flow when a code region is offloaded to a device, in order to illustrate how the runtime infrastructure makes use of the above functions. As an example, consider offloading to a GPU through OpenCL. The implementation and management functions have been loaded from a shared object. When the runtime is initialized, all the available devices are recognized and opened. Only the devices included in the global device list can be recognized. In this case, opening the device means initializing the OpenCL variables related to the device, such as the *context* and the *queue*, as well as allocating memory in the host to store these new information. When the host requires the execution of the target region, the required implementation is initialized, which in this example builds the OpenCL kernel and creates the buffers to store the transferred data in the GPU memory. Before and after the execution, lock and unlock routines set the device as busy/idle to control access to the device from other host threads while it is being used. If the host thread terminates, the implementation and devices are closed, deleting the stored information and releasing the allocated memory from the host and the device. Otherwise, the implementation is not closed by the runtime, since it is frequent that the target region needs to be executed repeatedly (e.g. when processing a sequence of video frames). In this

case, a 'soft' initialization is performed in successive executions. For example, there might be no need to rebuild the kernel or reallocate memory buffers — in the 'soft' initialization, this is checked to decide whether they can be reused from previous executions.

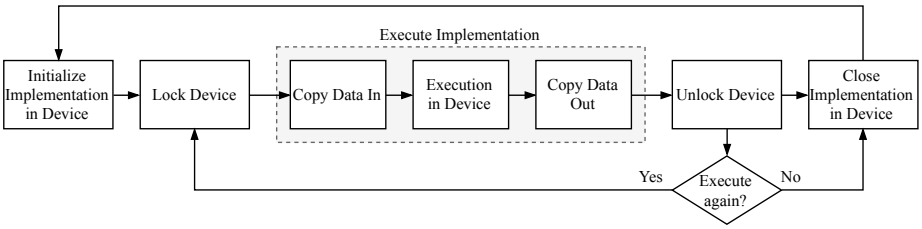

Fig. 3. Execution flow for device offloading performed by the runtime infrastructure.

3.4 Host Code Transformation

Previously, the implemented runtime support has been discussed. Allowing the execution of different device implementations requires some host side support. As a consequence, the host code needs to be preprocessed and transformed during compilation in order to enable the connection to multiple device implementations. Listing 2 shows how the code in Listing 1 is modified by replacing the target region by a call to a wrapper function which eventually manages the execution of any device implementation.

In Listing 3, the simplified implementation for the wrapper is shown. The arguments passed to the wrapper derive from the explicit mapping of variables defined by the programmer. When a target region is executed for the first time, an initialization is performed by building a list of the associated device implementations (which corresponds to a row in Table 2). This is carried out in code line 17, with `target1_struct` being an element with information about the *target-region function* as described in section 3.2, and `implementation_struct_array` being an array of pointers to elements with information about each implementation as described in section 3.2. An array of pointers to every implementation itself is then built (code line 19). The particular code for each device is external to the host code, since it is embedded in a shared object file. All shared objects available at the moment of execution are dynamically loaded at startup. The number of devices and implementations is unknown at the moment of compilation of the fat binary. The above allows to support new devices or optimize existing implementations without recompiling the host code. When the initialization for a target region has already been completed, the currently selected device is obtained and used to select the implementation to be launched (code lines 24-25).

Listing 2. Transformed host code, in which target regions marked in the original code are replaced by a wrapper function.

```
#define GPU  1
#define FPGA 2

void host_code(char *image_in, char *image_out, int width, int height)
{
    UC_set_default_device(FPGA);
    omp_set_num_threads(2);

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        // ----Thread #0----
        if(id == 0){
            function1();
        }
        // ----Thread #1----
        if(id == 1){
            wrapper1(char *image_in, char *image_out, int width, int height);
        }
    }
}
```

Listing 3. Code generated at the moment of compilation to connect the transformed host code to different device implementations (simplified).

```
// Create function pointer type
typedef int (*ptr_function)(char*, char*, int, int);
// Declare array of pointers to implementations
static ptr_function *implementations = NULL;

static int num = 0, initialized = 0;

// Wrapper to connect host code to different implementations
int wrapper1(char* image_in char* image_out, int width, int height)
{
   ptr_function fn;
   // Initialize list of implementations for current target region
    if(initialized == 0) {
        num = UC_get_num_devices();
        implementations = (ptr_function *) calloc(num, sizeof(ptr_function));

        (void) UC_Init_Impl( &target1_struct, &implementation_struct_array);
        for(int i=0; i<num; i++){
            implementations[i] = implementation_struct_array[i]->function;
        }
        initialized = 1;
    }
    // Select and launch implementation
    fn = implementations[UC_get_default_device()];
    return fn(image_in, image_out, width, height);
}
```

4 Experimental Evaluation

In this section, a proof-of-concept of the proposed methodology is presented. The runtime infrastructure has been evaluated using two heterogeneous architectures: a Zynq UltraScale+ MPSoC (CPU-FPGA) and a PC (CPU-GPU).

The serial video processing system represented in Fig. 4 is used as a test case. First, an RGB frame is taken from a camera. The image is converted to grayscale and a sobel filter is applied, which is an edge detection algorithm. The output image is shown on screen.

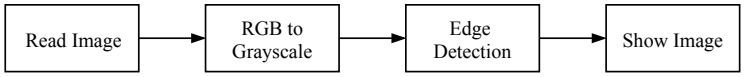

Fig. 4. Block diagram of the video processing sequence used as a test case.

The system is parallelized using OpenMP with four host threads concurrently executing the four tasks in which the system is divided. Therefore, a pipeline is established with four images being processed at the same time. In order to evaluate the proposed methodology, the edge detection function is marked for offloading with a `target` directive. In order to program the GPU attached to the PC, an OpenCL kernel has been generated. For execution on the Zynq MPSoC device, Xilinx SDSoC has been used to produce the driver functions for the host (dynamically loaded by the runtime) and the files to program the Zynq device (a hardware accelerator for the FPGA logic has been generated from the original target code with Vivado High-Level Synthesis). The host application code along with the developed runtime structure has been compiled with GCC for both x86 and ARM architectures.

Tables 4 and 5 summarize the execution time of the previously described example over two heterogeneous platforms: (i) a Xilinx ZCU102 board featuring a Zynq 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. In the experiments, the input images are 640x480 and obtained from the filesystem to avoid being limited by the camera framerate. The results are averaged over 100 executions.

Table 4. Performance on Xilinx ZCU102 - Zynq MPSoC ARM Cortex-A53 + FPGA.

Mode	Frames per second
Serial (CPU)	9.2
Parallel + Offloading (CPU)	16.9
Parallel + Offloading (FPGA)	39.5

Table 5. Performance on PC - Intel Core i7-3610QM CPU + NVIDIA GT630M GPU.

Mode	Frames per second
Serial (CPU)	79.2
Parallel + Offloading (CPU)	147.3
Parallel + Offloading (GPU)	295.3

5 Related Work

Several previous researches have studied and implemented code offloading from OpenMP annotated programs to accelerator devices. Liao *et al.* [8] first reviewed the *OpenMP Accelerator Model* when support for heterogeneous computation was introduced in the OpenMP API 4.0 back in 2013. They presented an initial implementation built upon an OpenMP compiler based on ROSE [9], with support for GPUs from NVIDIA by generating CUDA code.

More recently, some authors have worked to include OpenMP offloading support into the LLVM compiler infrastructure. To cite some of them, Bertolli *et al.* [10] focused on delivering efficient OpenMP offloading support for OpenPower systems and describe an implementation targeting NVIDIA GPUs. Their approach automatically translates the target region code to PTX language and eventually to low-level native GPU assembly, called SASS. Different optimization strategies were integrated into Clang with the aim of maximizing performance when compared to CUDA-based implementations. The CUDA device driver is used to map data to/from the GPU. In [11], Antao *et al.* generalize the previous approach to handle compilation for multiple host and device types and describe their initial work to completely support code generation for OpenMP device offloading constructs in LLVM/Clang. Pereira *et al.* [12] developed an open-source compiler framework based on LLVM/Clang which automatically converts OpenMP annotated code regions to OpenCL/SPIR kernels, while providing a set of optimizations such as tiling and vectorization. Lastly, a proof-of-concept implementation of OpenMP offloading to FPGA devices which also integrates with the LLVM infrastructure was presented by Sommer *et al.* [7]. In their work, Vivado HLS is used for generating the hardware from the C/C++ target regions. Compared to previous work, our proposal describes an alternative offloading methodology in which the device-specific compilation is no longer attached to the OpenMP host compiler, thus requiring little compiler support and integration effort.

6 Conclusions and Future Work

This paper introduces a new OpenMP device offloading methodology. In our proposal, the device-specific software compilation and/or hardware synthesis processes are dissociated from the OpenMP host compiler. The advantages of this approach include: (i) support for multiple devices (i.e. different architecture

GPUs, FPGAs...), while the standard offloading method heavily depends on the compiler supporting each architecture; (ii) large design flexibility (in terms of languages, design tools...) is provided to program the accelerator devices, being specially demanded by hardware designers to generate efficient FPGA implementations (iii) little compiler support and integration effort is required. To allow the application of the proposed methodology, we have presented a flexible runtime infrastructure that dynamically loads and manages the available device-specific implementations. Our future work includes integrating the presented runtime into an open-source compiler infrastructure and exploring the use of performance data to guide the selection of an available accelerator device during execution time.

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. Open MP API Specification. Version 5.0 November 2018 https://www.openmp.org/specifications/.
5. Offloading support in GCC, https://gcc.gnu.org/wiki/Offloading
6. Clang 9 documentation: OpenMP support. https://clang.llvm.org/docs/OpenMPSupport.html
7. 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.
8. C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan and B. Chapman, "Early Experiences with the OpenMP Accelerator Model", *9th International Workshop on OpenMP (IWOMP)*, Canberra, ACT, Australia, September 16-18, 2013.
9. C. Liao, D. J. Quinlan, T. Panas and B. R. de Supinski, "A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries", *6th International Workshop on OpenMP (IWOMP)*, Tsukuba, Japan, June 14-16, 2010.
10. 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.
11. 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.
12. 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.