FOTV: A Generic Device Offloading Framework for OpenMP

Jose Luis Vazquez and Pablo Sanchez(✉)

University of Cantabria, TEISA Dpto, Santander, Spain
{vazquezjl,sanchez}@teisa.unican.es

Abstract. Since the introduction of the "target" directive in the 4.0 specification, the usage of OpenMP for heterogeneous computing programming has increased significantly. However, the compiler support limits its usage because the code for the accelerated region has to be generated in compile time. This restricts the usage of accelerator-specific design flows (e.g. FPGA hardware synthesis) and the support of new devices that typically requires extending and modifying the compiler itself.

This paper explores a solution to these limitations: a generic device that is supported by the OpenMP compiler but whose functionality is defined at runtime. The generic device framework has been integrated in an OpenMP compiler (LLVM/Clang). It acts as a device type for the compiler and interfaces with the physical devices to execute the accelerated code. The framework has an API that provides support for new devices and accelerated code without additional OpenMP compiler modifications. It also includes a code generator that extracts the source code of OpenMP target regions for external compilation chains.

In order to evaluate the approach, we present a new device implementation that allows executing OpenCL code as an OpenMP target region. We study the overhead that the framework produces and show that it is minimal and comparable to other OpenMP devices.

Keywords: OpenMP · Heterogeneous computing · Offloading · Generic devices

1 Introduction

Over the course of the past few years, more and more projects are looking to harness the efficiency and computing power of specific accelerator devices through heterogeneous computing techniques. These programs typically offload defined sections of highly data parallel computation to GPU devices [3, 5] or FPGA-based accelerators. In order to support heterogeneous computing programing, OpenMP 4.0 [4] introduced the "target" family of directives that the latest specifications have extended and improved. The "target" directives instruct the compiler to allocate and move data to and from a target device and to execute code on that device. Nowadays, compiler support for these directives extends to multiple CPU and GPU architectures [7–9].

However, the current approach has certain limitations. For example, the code for the target region is generated and compiled in parallel with rest of the program [5, 6], and

S. McIntosh-Smith et al. (Eds.): IWOMP 2021, LNCS 12870, pp. 170–182, 2021.
https://doi.org/10.1007/978-3-030-85262-7_12

therefore the OpenMP compiler has to generate the target code, produce the target binary with a specific compilation chain and integrate all target binaries in a fat binary. Therefore, the integration of a new device typically requires specific compiler modifications that require OpenMP-compiler internal knowledge. Additionally, the OpenMP compiler generates the target binaries, which limits device-specific optimizations, performance analysis and target compilation chain. For FPGA-based accelerators, this limitation could have an important impact on performance. OpenMP 5.1 introduces the interop capability [4], allowing OpenMP to interface with other heterogeneous computing runtimes (e.g. OpenCL). While this is a step forward, it does not directly facilitate new device support.

The complex integration of new devices could limit the usage of OpenMP for heterogeneous computing. Other approaches, for example OpenCL [2], do not require modifying the host compiler, only some functions have to be implemented in a shared library. Additionally, OpenCL and OpenMP require a device specific compilation toolchain but, in OpenCL, the toolchain is not defined in the host compiler and the users can provide the kernel code or the kernel binary. This work explores a similar approach for OpenMP in order to facilitate new device integration.

We have integrated a generic device, FOTV (Future Offload Target Virtualization) in an open-source compiler (LLVM/Clang, version 13). The generic device is a container that supports new devices and target-region implementations in an OpenMP compiler. All the functionality that FOTV requires from a new device is encapsulated in a device-management component. Additionally, a target-region component defines all the functions that FOTV requires to execute an offloaded code in a particular device. These components define the set of functions (API) that the FOTV device requires to support in OpenMP a new device and/or device-specific target-region implementations. The implementations of these components are included in dynamic libraries that are loaded during the execution of the OpenMP program.

Additionally, a code generator extracts the target region code during OpenMP compilation. With this code, a device-specific toolchain could optimize, analyze and produce specific binary code after OpenMP compilation. These specific binaries will be loaded at runtime. The approach has been evaluated with a device that uses OpenCL code to implement the target region code.

The main contributions of the paper are the identification of an API for target device and implementation definition, the integration of a generic device (FOTV) in an OpenMP compiler and the definition of a methodology for dynamic loading of device-specific behaviors.

The rest of the paper is structured as follows. In Sect. 2, we describe the standard OpenMP offloading strategy and analyze its limitations to motivate this work. Section 3 presents the framework architecture and details two main components: the runtime library and the code extraction tool. The API for adding new devices and implementations is introduced in Sect. 4. Section 5 presents an example of a new device that uses OpenCL code. Section 6 evaluates the new device with several examples. Section 7 analyzes the related works and finally, Sect. 8 presents the conclusions and future work.

2 Background: OpenMP Offloading Infrastructure

This section briefly describes the OpenMP offloading strategy of Clang and analyzes the limitations that are faced in this work. A detailed definition of the offloading strategy is described in [5, 6]. Other compilers use a similar approach and therefore the conclusions of this work can be extended to them.

2.1 Offloading Strategy

The OpenMP offloading infrastructure integrates a runtime manager ("libomptarget"), some device specific libraries (device plugins), a target code-generation infrastructure and a binary bundler. The "libomptarget" module implements the interface between the host and the target devices. It defines a set of functions that the target devices have to implement in a specific device plugin, which provides device management functionalities such as device startup, data allocation and movement, target region launch and binary load.

The target-code generation infrastructure extracts target-region code in a compiler intermediate representation. This code is then compiled through a device-specific compilation toolchain that is compatible with the host compiler representation. The output binaries of all toolchains are embedded in a "fat binary": a single executable binary with the host and target binaries of all target regions. The bundler is used to make sure that the device binaries have the correct format when they are embedded into the host binary.

At runtime, the device plugins load the target specific binaries from the "fat binary" and launch target-region executions when it is required.

2.2 Advantages and Limitations

The previously commented strategy has several advantages such as a generic API for device management and device plugins for the specific implementations. However, it has several limitations:

- The runtime library ("libomptarget") is focused on device management but it has not control of the target regions that are loaded by the device plugins.
- Every time that a new device plugin has to be defined, the source code of the host compiler has to be modified.
- All the device tool chains have to be able to interact with the host compiler. Therefore, the integration of a new tool chain is not trivial.
- The host and device binaries have to be generated during the host compilation process and the OpenMP compiler has to support all required target devices.
- The target-code generation infrastructure produces a compiler-related code that cannot be easily adapted or modified for a specific device. OpenMP has introduced a new feature, the variant directive, that partially face this problem but the specific variant devices have to be supported by the host compiler.

Our proposal tackles these concerns by implementing a single OpenMP device plugin, an external runtime device/target-region manager and a source code extraction tool to interface with external device toolchains. This approach has several advantages:

- The extracted source code has the original syntax, which makes it easy to modify or adapt to specific targets.
- The approach supports off-line compilation with tool chains that are independent of the host compiler.
- Decoupling the process from the host compiler also means that new device support does not need any host compiler modifications.
- The framework loads both device and target region implementations at runtime. Consequently, there is no need to generate all implementations for all target regions during compilation.

For device manager, the proposed approach uses an API that is compatible with "libomptarget" in order to be easily integrated in an OpenMP compiler.

3 Architecture of the FOTV Generic Device Framework

The main elements of FOTV architecture are presented on Fig. 1. An open-source compiler (Clang) analyzes the OpenMP code. We have extended the last development version (v13) of Clang to support the FOTV device. The runtime library ("FOTVManager") manages FOTV-based devices and device-specific implementations at runtime. The compiler extension also includes a code generator that extracts, during OpenMP code compilation, the target-region code and some additional metadata such as file name, statement line, pragma directive and "for-loop" boundaries. All this information is stored in a "json" file.

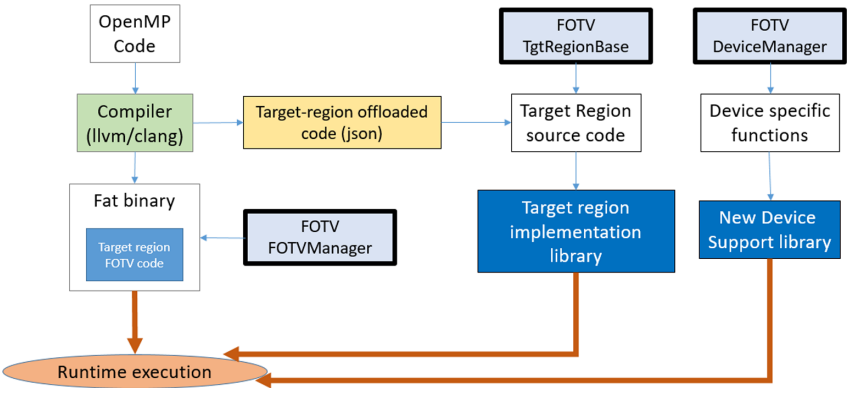

Fig. 1. Main elements of the FOTV architecture

3.1 The Runtime Library Components

The FOTV framework defines two components ("DeviceManager" and "TgtRegionBase") that encapsulate all device and target-region specific functionality. For a particular device, the "DeviceManager" methods have to be implemented with device-specific code.

The "DeviceManager" component integrates most of the operations of a device, centered around data movement and target region association and execution. The "TgtRegionBase" component represents a device-specific implementation of a target region. The "DeviceManager" and "TgtRegionBase" objects are integrated in dynamic libraries. When these libraries are loaded, all declared objects are registered in specific lists that are used to manage the FOTV components at runtime.

At runtime, the "FOTVManager" component controls all devices that have been defined with the generic device framework at runtime. During execution startup, the manager looks for dynamic libraries that include information about new devices and implementations. It loads the device/target-code libraries, checks their integrity and associates the implementations to devices. When a target-region is executed in the FOTV device, the manager checks if an implementation is available. If there is no device-specific implementation, the host target-region code will be executed.

3.2 The Code Extraction Tool

The proposed approach can use compilation tool chains that are independent of the host compiler. These toolchains require target-region source code to produce device specific binaries. The code extractor identifies, during host compilation, the target-region source code and some additional metadata that can be used for automatic code generation. It also generates the infrastructure that is needed to load the target-region code at runtime. This infrastructure can be used to integrate device specific implementations that are not related to the OpenMP target region code. All the information that the extraction tool produces for a compilation unit is included in two JSON files: a target-region and a data region file. The JSON file includes parameter mapping information, the outlined function identifier and the source code of the target region. Among the meta-data generated information is the full generating pragma, the mapping clauses, source location and mapped variables. The tool will generate the files during the host compilation process and it is integrated in a Clang optimization pass. Figure 3 presents a simplified diagram of the tool and next figure shows an example of the generated JSON file.

```
'axpy_concurrent_region_l11":{
  "source_file":"axpy_concurrent.c",
  "generating_pragma":"#pragma omp target parallel loop",
  "pragma_start_line":11,
```

Fig. 2. JSON file example.

4 Device Management API Description

This section introduces the basic elements of the approach. The DeviceManagement component functions are similar to those of "libomptarget", but they are modified to accommodate for the functionality of our runtime. The methods in the "TgtRegionBase" component are used to manage the target region implementations.

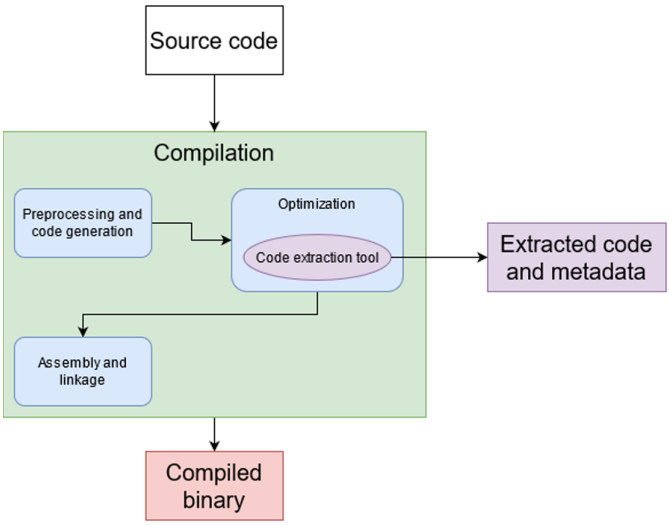

Fig. 3. Simplified compilation process and location of the extraction tool

4.1 DeviceManagement Component

A DeviceManagement implementation requires a minimum block of information that is used for identification and future compatibility with the OpenMP interop feature. This information includes an identifier string that is necessary for target region registration, a parameter block, an event queue and an internal host-to-device pointer mapping. The component contains 3 types of functions: basic runtime functions, memory management functions and synchronous and asynchronous data movement.

Basic Runtime Functions. These functions handle device operation. These include the implementation registration function for each device, functions to query the information block of a device, device synchronization routine, implementation registration, resource initialization and pause routines, and pointer map querying functions.

Synchronous and Asynchronous Data Movement Functions. These functions manage synchronous and asynchronous data movement in all directions: to the device, from the device and between devices.

Memory Management Functions. These functions are used to handle device memory and how it maps to the device. Includes the basic "alloc" and "free" operations as well as explicit pointer association and disassociation functions.

4.2 TgtRegionBase Component

The TgtRegionBase component is a simple wrapper with three fields. In certain cases, the user might need to extend the component with device-specific information. The core component fields are presented on Table 1.

Table 1. Target region fields

Name	Type	Explanation
dev_id	string	Device identifier. It is used for target region registration in a particular device
fn_id	string	Function identifier that is used for implementation querying. The identifier is derived from the original offloaded function symbol, and it is available in the code extraction output
fn	void *	The implementation function to be executed

5 Case Study: Running OpenCL Kernels as OpenMP Regions

In order to evaluate the generic-device framework, we have developed a FOTV-based device. This device uses the OpenCL runtime to identify the available platforms and hardware devices. These devices are discovered at runtime and mapped to new OpenMP devices. Therefore, in this case the OpenMP runtime provides the application with the compiler supported devices and all devices that are identified by the OpenCL runtime. The proposed approach also requires that the OpenMP target-regions be transformed into OpenCL kernels. Next sections describe the structure of the FOTV-based device.

5.1 The OpenCL Device Requirements

When a generic device infrastructure uses the OpenCL runtime, some problems have to be fixed. On the one hand, the runtime typically identifies more than one device at runtime and the "DeviceManager" component is designed to support only one specific device or device family. On the other hand, OpenCL requires some scaffolding code to be operated and it depends of the device and kernel. The generic device infrastructure assumes that the device functions are independent from the target-region functions and the OpenMP runtime uses these device functions to provide all the infrastructure that the target-region needs. This approach is not directly aligned with the OpenCL runtime.

These specific requirements make the OpenCL device a good use case to demonstrate the flexibility and efficiency of the proposed generic device. To comply with these requirements, the OpenCL device includes three elements that are presented on Fig. 4:

- A device-specific target region module (TgtRegionOpenCL). This new module includes the OpenCL kernel code. This code could be pre-compiled or an OpenCL C source built at runtime.
- A shared infrastructure (FOTVOCLEnvironment) between the device and the implementation. This object contains pointers to the OpenCL scaffolding as well as various ordered data mapping interfaces to be used by the target region module.
- An OpenCL specific device manager extension (OpenCLDeviceManagement). This extension consists of three major blocks: The first one is an OpenCL-specific implementation register. We also add initializer code that initializes OpenCL scaffolding. Finally, every extension requires device-specific implementations for the API introduced in the previous section, which in this case means OpenCL-specific implementations.

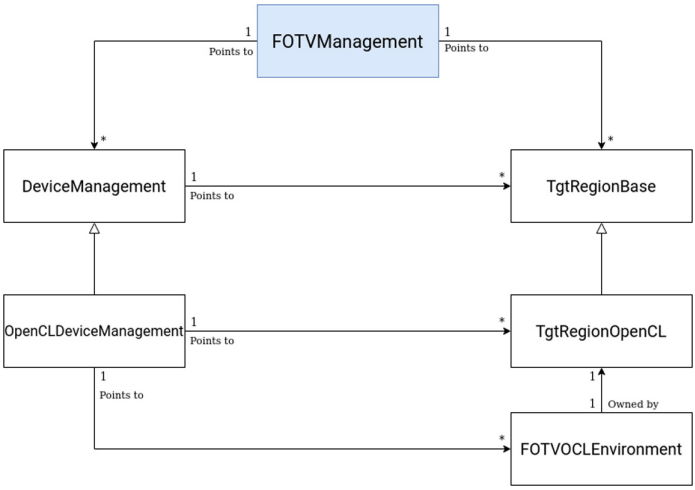

Fig. 4. Extended component diagram for OpenCL operation

6 Results

For benchmarking, we used an edge detection algorithm (Canny filter) in an image processing pipeline. The video pipeline includes four modules: a video capture, an image converter, an image filter and an image display module. The filter has a "target teams distribute parallel for collapse(2)" pragma in its top loop.

The tests were produced with code compiled using a FOTV version integrated in a development build of LLVM/Clang 13, using a standard workstation with the specs included in Table 2. The platform includes a CPU with 16 cores and a NVIDIA GPU with 768 cores. The FOTV device integrates the OpenCL backend that was presented in the previous section.

Table 2. Execution platform specs

Component	Type	Capabilities
CPU	Intel Xeon E5-2687W	16c/32t @ 3.1GHz
GPU	NVidia GTX1050Ti	6 CUs, 768 CUDA cores, 1392 MHz
Main memory	64 GB DDR3	
Operating system	Ubuntu Linux 18.04	OpenCL 2.0, LLVM/Clang 13

The OpenCL kernel code for the filter was manually generated from the JSON files, although it could be automatically created.

The main goal of the test is to evaluate the impact of the FOTV infrastructure and the flexibility that it provides to OpenMP. Two tests are proposed to evaluate these features.

The first test evaluates the FOTV performance impact. Figure 5 presents a comparison between the FOTV system offloading to GPU via OpenCL and the OpenMP native GPU offloading. The figure presents three data: the FOTV execution time including OpenCL data transfers, the FOTV execution time without the OpenCL data transfers and the standard OpenMP execution time with GPU offloading. As we can see, the OpenCL data transfer overhead is responsible for the difference between the FOTV and the standard OpenMP offloading strategies. If the OpenCL data transfer overhead is removed, the execution time of FOTV and OpenMP is very low.

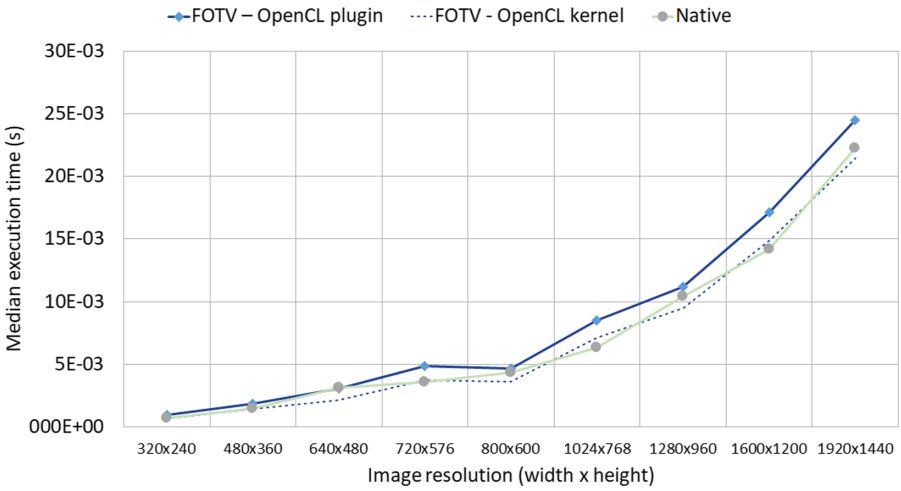

Fig. 5. Median execution times for different resolutions

The second test is oriented to evaluate the flexibility of FOTV. As we previously mentioned, FOTV can handle multiple devices with multiple interfaces. In Fig. 6, we are running the same test with a live device offloading reconfiguration of device swapping. In this case, the system has 2 offloading devices handled by FOTV: a local OpenCL-based GPU plugin and a remote POCL-r [15] based plugin. The remote POCL server provides access to a NVidia Quadro RTX4000 GPU. During the test, the remote access causes the large increase in execution time.

We observe a spike in the plot, just through a few images after device switching. This is produced largely by the implementation switch, while the extra execution time is caused by the remote communications and server load. This spike represents device switchover and the manager caching the new implementation function as well as device warmup, and averages out over the course of the run.

7 Related Works

The usage of dynamic libraries to define new devices and implementations was presented in Álvarez et al. [1]. However, it is a proof of concept that is not integrated in an OpenMP

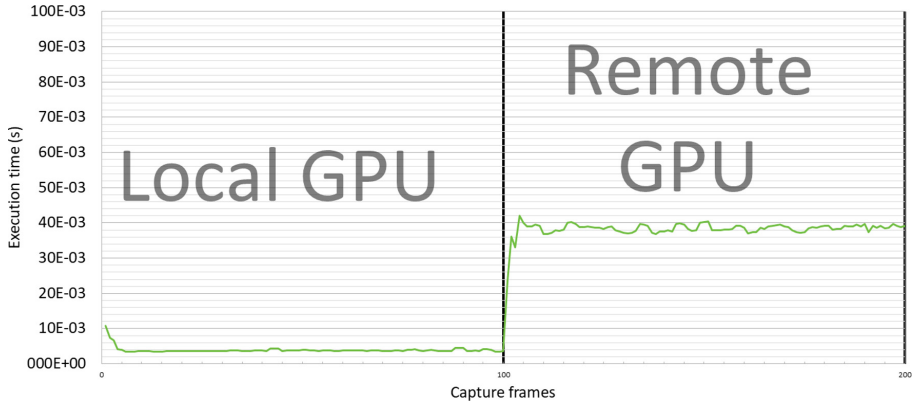

Fig. 6. Response time with a live device swapping for a local to a remote GPU

compiler and forces to use custom device implementations. The proposed FOTV generic device works seamlessly as an OpenMP device without any special requirements other than adding it as an offload target in compile time.

The addition of an external library for device management was inspired by the Aurora VE offload system [9]. The Aurora paper describes how to introduce the VE as an OpenMP offload target by introducing a custom toolchain that executes a source-to-source compiler and proprietary tools to generate the fat binary. After this process, they use the standard VE management library tools within a target device plugin to manage the device. This design allows higher flexibility than other device integrations such as the NVidia GPU [5, 6] support module that requires embedding architecture-specific runtime calls into the fat binary for operation. Unlike [9], this paper uses a compiler toolchain independent approach and defines a set of functions that facilitate the integration of new devices. Additionally, the proposed approach also dynamically loads the device-specific target-region implementations instead of including them in the fat binary. This provides a higher flexibility, as new devices don't require a tool-chain and/or backend to be integrated in the library, as well as giving the option of providing only relevant target implementations instead of having all target regions built for all devices.

Multiple works exist that extend the device pool of OpenMP with new features. The works range from adding an existing device into a different frontend [11], to adding entire cloud infrastructures as offloading devices [10]. In most cases, the new target still requires compile time code generation, making the approach device-specific and hard to extend. One of these implementations is the FPGA offloading device from Sommer et al. [12] that requires a High-Level Synthesis (HLS) compiler for the target device. This typically leads to very high compile times and very low FPGA occupation and performance, since CPU- and GPU-optimized code is notably inefficient in the FPGA architectures. Further work by Knaust [13] and Huthmann [14] attack this problem in different ways. The first one opts to prototype the FPGA device with OpenCL and compiler-specific interfaces, requiring IR (Intermediate Representation) backporting to make use of the HLS system and OpenCL interfaces. The second one creates a more

efficient compilation toolchain, obtaining OpenMP-optimized HDL code to synthesize it in a faster and more efficient way. These approaches are still device-specific, which was our main concern when creating the generic interface. The proposed approach allows using the standard HLS process that provides the most efficient implementations. These implementations use dynamic loading at runtime.

More flexible than [13, 14] is the aforementioned Yviquel et al. [10] work. It does not generate target binary code but rather a Scala implementation (as a Java runtime binary) to be ran on any Apache Spark cluster. It uses a configuration file for authentication and configuration of a cloud cluster that is started on the RTL plugin device initialization call. The nature of this offloading infrastructure allows for more target flexibility than standard extensions, as the target is a cloud cluster middleware instead of specific hardware, but it's still limited to that specific runtime and would require rebuilding the code generation in case of an API extension. This work outperforms these approaches with the integration of a generic device in the OpenMP compiler and the definition or an API that simplifies the definition of new devices and the integration of specific implementations.

8 Conclusions and Future Works

In this paper we introduce FOTV, an offloading target for OpenMP that allows for extended device support through a runtime management library. We prove that it can handle otherwise unsupported devices by running OpenCL code through it, and show that it has minimal overhead over traditional OpenMP offloading. We also present a way the end user can optimize both device and implementations, allowing for better performance than other OpenMP device implementations.

This first version presents the base form of FOTV as a device bridge. A future goal would be to introduce an extended runtime library that manages resource loads for optimal performance and efficiency across multiple heterogeneous devices, exposing the entire platform as a single device to OpenMP. We also hope to create an automatic implementation generator, using information from the code extractor to create a rudimentary implementation function code that end users can then adapt to their particular devices.

Acknowledgement. This work was done as part of the FitOptiVis project, funded by the ECSEL Joint Undertaking, grant H2020-ECSEL-2017–2-783162, and the Spanish MICINN, grant PCI2018–093057. It was partially funded by the Platino project, funded by the MICINN, grant TEC2017–86722-C4–3-R.

References

1. Álvarez, Á., Ugarte, Í., Fernández, V., Sánchez, P.: OpenMP dynamic device offloading in heterogeneous platforms. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 109–122. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_8

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. Bertolli, C., et al.: Integrating GPU support for OpenMP offloading directives into Clang. In: LLVM-HPC2015, Austin, Texas, USA, 15–20 November 2015
6. Antao, S.F., et al.: Offloading support for OpenMP in Clang and LLVM. In: LLVM-HPC2016, Salt Lake City, Utah, USA, 13–18 November 2016
7. Clang 13 documentation: OpenMP Support. https://clang.llvm.org/docs/OpenMPSupport.html
8. Offloading support in GCC. https://gcc.gnu.org/wiki/Offloading
9. Cramer, T., Römmer, M., Kosmynin, B., Focht, E., Müller, M.S.: OpenMP target device offloading for the SX-Aurora TSUBASA vector engine. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 12043 LNCS, pp. 237–249 (2020)
10. Yviquel, H., Cruz, L., Araujo, G.: Cluster programming using the OpenMP accelerator model. ACM Trans. Archit. Code Optim. **15**(3), 1–23 (2018)
11. Özen, G., Atzeni, S., Wolfe, M., Southwell, A., Klimowicz, G.: OpenMP GPU offload in flang and LLVM. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2018, pp. 1–9 (2018)
12. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2017, pp. 201–205 (2017)
13. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA offloading prototype using OpenCL SDK. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 387–390 (2019)
14. Huthmann, J., Sommer, L., Podobas, A., Koch, A., Sano, K.: OpenMP device offloading to FPGAs using the nymble infrastructure. In: Milfeld, K., de Supinski, B., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems. IWOMP 2020. Lecture Notes in Computer Science, vol. 12295. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_17
15. Solanti, J., Babej, M., Ikkala, J., Jääskeläinen, P.: POCL-R: distributed OpenCL runtime for low latency remote offloading. In: Proceedings of the International Workshop on OpenCL (IWOCL 2020). Association for Computing Machinery, New York, NY, USA, Article 19, pp. 1–2 (2020). https://doi.org/10.1145/3388333.3388642