OpenMP Device Offloading to FPGA Accelerators

Lukas Sommer
Embedded Systems and Applications Group
TU Darmstadt
sommer@esa.tu-darmstadt.de

Jens Korinth
Computer Systems Group
TU Darmstadt
korinth@rs.tu-darmstadt.de

Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
koch@esa.tu-darmstadt.de

***Abstract*—Future high-performance computing systems will need to include multiple specialized accelerators in a single heterogeneous system to overcome power-density limitations of CPU performance.**

To program such heterogeneous systems without the need to maintain multiple code bases, OpenMP device offloading constructs can be used to execute compute-intensive regions on different kinds of accelerators.

In this work we present a proof-of-concept implementation of OpenMP offloading for FPGA-based hardware accelerators. Our implementation seamlessly integrates with the existing LLVM offloading infrastructure, and enables the user to move computations to a custom FPGA accelerator by simply adding OpenMP offloading directives to the input program.

I. Introduction

With new process technologies for CPUs no longer translating into performance improvements due to power density limitations, the performance increase for CPUs has declined in recent years. As a consequence, CPU-only systems are no longer able to meet the ever-increasing demands for computing power, especially in high-performance computing (HPC) scenarios.

In order to overcome these limitations and provide sufficient computational power for future computing tasks, future high-performance computing systems will need to incorporate multiple dedicated, specialized accelerators into a single heterogeneous system. Each accelerator is suitable for a limited set of operational tasks and is able to deliver a better power-efficiency for this set of tasks than the general-purpose CPU.

Beyond the GPUs that are nowadays very common in high-performance heterogeneous systems, dedicated FPGA-based hardware accelerators have received increasing attention recently. Their reconfigurability facilitates the adaptation of the accelerator to multiple tasks, delivering better performance and power-efficiency than general-purpose processors. An example for the use of FPGAs in heterogeneous systems is the deployment of FPGAs in Microsoft's Azure cloud [1].

However, programming heterogeneous systems is hard, as each system comprises multiple (potentially varying) different computational units. Adapting the software to each accelerator and system requires significant rewriting of existing code bases, resulting in high development effort and cost.

The use of a single input program for all kinds of accelerators, on the other hand, is also challenging. This is especially true as not all accelerators are suitable for all computational tasks, and typically only the most computation-intensive sections ("hot-spots") of a program should be offloaded to a dedicated device, whereas more sequential or unsuitable (e.g. control-flow intensive) regions of code should remain on the general-purpose CPU.

The OpenMP target directive, introduced and refined in recent versions of the standard [2], is a perfect fit for denoting regions of code that should be offloaded to a device in a heterogeneous system. The directive does not only allow to specify the device-suitable regions of code by easy-to-use pragmas, but the associated data mapping clauses also give the programmer full control over what and how data is mapped to the device. Therefore, the OpenMP target directive is a good choice for the programming of heterogeneous systems including an FPGA as dedicated hardware accelerator.

The LLVM compiler framework C/C++-language frontend, Clang, is currently being extended for target directives. In this work, we build on the existing Clang infrastructure to provide support for offloading to FPGA-based accelerators using OpenMP target directives. Our work is based on *ThreadPoolComposer (TPC)* [3], an automated framework for the synthesis and HW/SW interfacing of FPGA-based accelerators (available from [4]). It uses Xilinx Vivado HLS for generating hardware accelerators from C/C++. Our implementation enables the user to directly offload OpenMP target regions that are compatible with the input restrictions of Vivado HLS to FPGA hardware accelerators. The user is not required to provide specialized code in the input program to interface with the hardware accelerator and our implementation also manages data mapping to the FPGA memory.

The rest of this work is structured as follows. Section II gives an overview of related work and the existing OpenMP offloading infrastructure in Clang/LLVM. Section III provides a short introduction to the ThreadPoolComposer toolchain, on which our work is based. In Section IV we describe our new compile and runtime flow, which we evaluate and compare in Section V. Section VI concludes our work and gives an overview on future work.

II. Prior work

We will begin the discussion of related work by looking at other tools which use OpenMP as input for High-Level Synthesis targeting FPGAs and compare them to our approach. Afterwards, we describe some efforts to implement OpenMP

978-1-5090-4825-0/17/$31.00

device offloading to non-FPGA targets and explain the existing LLVM offloading infrastructure in more detail.

A. OpenMP-based FPGA-acceleration

OpenMP-annotated source code has been used as input and starting point for a number of High-Level Synthesis approaches targeting FPGAs. These approaches focus on efficiently mapping *parallel* OpenMP-constructs to FPGA-based hardware accelerators, either using a pure hardware-flow or aiming for a mixed software/hardware-environment. For example, in [5], [6] a dedicated accelerator is synthesized for every OpenMP task in the program. Other efforts, such as the ones presented in [7], [8], [9], [10], efficiently map OpenMP worksharing loops to FPGA-accelerators capable of executing computations from multiple threads in parallel.

However, none of these approaches makes use of the OpenMP *target* directives to allow the user to specify which regions to offload to the FPGA, or how to map data from the host to the FPGA.

In contrast, the approach presented in this work uses the OpenMP *target* directive to allow programming a heterogeneous system, including FPGA-based accelerators, with a single, portable input code. The focus of our work is the offloading itself, i.e. the management of the on-device execution and the efficient mapping of data from host- to device-memory.

Using the memory mapping specified by the user in the appropriate pragma allows our tool to clearly determine which data must be transferred to/from the device memory, whereas the previously discussed approaches must employ a conservative approximation and tend to transfer more data than strictly necessary.

For the mapping of the code inside the target region to FPGA-accelerators, we currently rely on Vivado HLS. However, note that we could also integrate other HLS-approaches (e.g., those already listed above or other generic HLS systems such as Nymble [11], Bambu [12] etc.), to efficiently map the code inside the target region, potentially containing parallel OpenMP constructs, to the FPGA. Thus, our approach can be seen as complementary to the ones discussed above in that it can *additionally* provide the user with clearly defined means to specify a memory mapping and denote regions that are to be executed as FPGA-accelerators.

B. OpenMP Device Offloading

OpenMP device offloading has previously been implemented within the LLVM infrastructure for a number of device types. [13] presents results for OpenMP execution in a system featuring a Xeon Phi accelerator. In [14], an implementation of OpenMP target offloading for DSP accelerators is described. The target architecture comprises of a multi-core, general-purpose ARM CPU and a multi-core DSP. The volume of data transferred is optimized by allocating a contiguous block of physical memory that is shared between host and device, making the transfer to/from the device obsolete.

In [15], Bertolli et al. present an extension of the Clang language frontend and the LLVM OpenMP runtime, which facilitates the use of CUDA-enabled Nvidia GPUs for OpenMP offloading. Code regions intended for offloading are automatically translated to CUDA ptxas assembly, with OpenMP parallel constructs transformed to parallel CUDA constructs. The runtime uses the CUDA device driver to map data to/from the device and to initiate computation on the GPU.

Finally, [16] presents a concerted effort for generic and extensible support of OpenMP device offloading within the Clang/LLVM compiler infrastructure. The implementation is designed to provide easy access to common functionality, and to be extended for further device types with limited implementation effort.

During the compilation phase, a separate device-specific translation is performed. The resulting binaries for all devices and the host are then combined to a single "fat" binary.

Additionally, the LLVM OpenMP runtime has been extended by two-layered library support for device offloading, with the design of the library described in [17]. The device-agnostic libomptarget provides common functionality and offers a standardized interface for data mapping and device execution control. To this end, libomptarget interacts with the device-specific plugins on the second layer of the library, which each provide support for offloading for a certain kind of device (e.g. CUDA-enabled GPUs).

Our own work integrates seamlessly with this LLVM offloading infrastructure. In Section IV we describe our custom Clang compilation flow and the implementation of our libomptarget device plugin.

III. ThreadPoolComposer

The ThreadPoolComposer [3] toolchain has been developed to fast-track the prototyping of FPGA-based accelerators using HLS tools, such as Vivado HLS. TPC automates the execution of the HLS tool to synthesize hardware accelerators from kernel code, and can assemble multiple instances of these accelerators (*processing elements, PEs*) in a complete top-level design, called *threadpool*, which also provides standardized connectivity to host and device memory. Regardless of its internal composition, every threadpool can be controlled by a two-layered, unified software interface, consisting of the user-facing *TPC API* and the internal *Platform API*. TPC API provides basic functions to query the device bitstream for available hardware modules, transfer data to/from the device and launch jobs on the threadpool. Platform API is only used to implement the TPC API and isolates platform-specific code (e.g., to control infrastructure hardware). This facilitates basic portability of TPC applications since the TPC API is identical for all hardware platforms (write once, run "everywhere"). Using TPC designs, the software programmer can thus take advantage of the specific capabilities of the executing platform, without having to rewrite any code. This makes TPC an ideal low-level foundation for the implementation of higher-level, heterogeneous, parallel runtimes and programming frameworks, such as OpenCL, or OpenMP.

In this work, we substantiate this claim by combining TPC and the OpenMP target offloading infrastructure to implement FPGA-based accelerators directly from OpenMP application

code. Our custom Clang toolflow (described in more detail in the next section) extracts standalone C/C++ functions from the OpenMP application code (marked with the target directive), which are then fed into the TPC toolchain to generate a hardware design. Our LLVM offloading plugin automatically generates the TPC API calls to interface with the hardware, enabling transparent offloading to the FPGA.

IV. Offloading for FPGAs

In this section we describe the integration of our work into the Clang/LLVM OpenMP offloading infrastructure. As stated in Section II, our implementation consists of two parts, namely a custom device toolflow for Clang and a device-specific plugin implementation for libomptarget, both described in detail in the following sections.

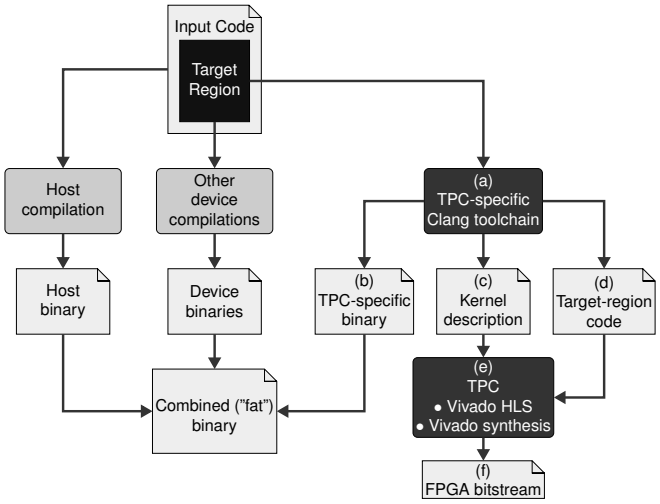

Fig. 1. Custom device toolflow for Clang.

A. Compilation Flow

As indicated in Section II, the compilation of an input program containing one or more OpenMP target directives results in multiple calls to Clang toolflows. In addition to the regular invocation for the host compilation, a distinct toolchain is invoked for each offloading target selected in the original call to the Clang driver. In contrast to the host compilation, the scope of per-device compilation is limited to the target regions present in the input program, which have each been extracted to a separate function before.

In order to support FPGA-based OpenMP offloading, we implement a custom Clang toolflow (see (a) in Fig. 1), which is identified by a custom LLVM target-triple we introduced. The output of our custom toolflow consists of three parts:

- A TPC-specific device software executable (Fig. 1.b), which is included in the fat binary.
- An input file for Vivado HLS for each target region in the original program (Fig. 1.d).
- A kernel description (Fig. 1.c) for each target region, used to drive the TPC synthesis flow.

These output files are now described in greater detail.

TPC device software executable The device software executable is the result of our custom code generation. It directly interacts with TPC using functions from the TPC API (cf. Section III). Its tasks are the transfer of parameter values (e.g. pointers to arrays used within the target region) to the FPGA, and the launch of the accelerator execution after a job ID has been acquired. The additional level of indirection introduced by this software executable (compared to direct invocation of TPC from the libomptarget-plugin) is beneficial, as it allows us to implement a more fine-grained control of the interfacing between software and hardware. For example, we could implement coarse-grain parallelism in the software executable, distributing the computations of an OpenMP work-sharing loops across multiple hardware processing elements by acquiring and simultaneously launching *multiple* offloaded jobs in the software executable.

Vivado HLS input file The Vivado HLS input files resulting from our custom device compilation toolchain each contain the code for a single target region of the original program, extracted to a function. Our toolchain also preserves pragmas unknown to Clang in this regions, allowing Vivado HLS specific pragmas annotated by the user to be still present in the Vivado HLS input file. These pragmas, indicating e.g., pipelining or unrolling of a loop, could also be added automatically in the future.

Kernel description The TPC-specific kernel description identifies the Vivado HLS input file and the target function for each target region, and also lists the type (*value* or *reference*) for each parameter of the target function.

From here on, TPC automates the entire design flow: Using the kernel description and the target region code source file, TPC synthesizes a hardware module for each target region in the input program via Vivado HLS. TPC then automatically assembles a complete top-level design with host connectivity and memory access. The top-level can contain multiple hardware instances of an individual kernel (so-called *processing elements*), and also mix hardware instances of different kernels (from distinct target regions), avoiding the need to dynamically reconfigure the FPGA at runtime. Finally, the hardware design is synthesized using the Vivado Design Suite. The resulting bitstream can be directly loaded and accessed with the TPC APIs (see Fig. 1.f).

In summary, our custom compilation flow allows users to go from a single input code, with OpenMP target directives annotated in the program code, to a complete FPGA-design including memory- and host-connectivity, without the need for the user to provide additional low-level specifications to the flow.

B. Runtime flow

The OpenMP offloading model is a host-centric approach, i.e., the execution starts on the host CPU. If a target region is encountered, device execution is initiated by calling libomptarget using calls from the LLVM OpenMP runtime library interface. Should this be the first offload to a device, the device is initialized now. Before the execution on the device

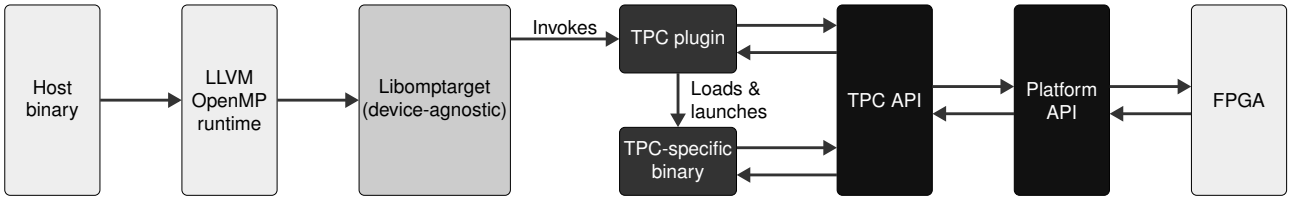

Fig. 2. Runtime flow for TPC FPGA offloading.

can start, data needs to be made available on the device. The OpenMP standard allows data to be shared between host and device, as well as for separate data spaces. In our current implementation, FPGA and host CPU do not share memory, therefore we need to allocate and transfer mapped data to the FPGA memory by invoking the TPC device plugin. During this process, libomptarget is responsible for keeping track of the mapping between host and device pointers for each variable mapped to the device, and the TPC-specific device plugin uses data transfer functions from the TPC API to initiate data transfers with the DMA engine in the FPGA bitstream.

In the next step, the TPC device-specific software executable is loaded and launched using libelf and libffi, a process similar to the one used for offloading to ELF-compatible devices in general. The loaded software executable in turn starts hardware execution (cf. Section IV-A). After hardware execution completes, data is copied back to the host memory, again using TPC API calls and the DMA engine on the FPGA. The entire runtime execution flow is shown in Fig. 2.

V. Evaluation

For this initial proof-of-concept system, we use the Xilinx VC709 board as accelerator. Here, an XC7VX690T FPGA, attached to 4 GiB on-board memory, is connected to the host via PCIe Gen3 x8. This platform uses the lightweight *ffLink PCIe Gen3 interface* which achieves close-to-optimal data transfer speeds [18]. The host uses a four core Intel Core-i7-6700K at 4.0 GHz with 16 GiB of DDR4 RAM running a Fedora 22 Linux with kernel 4.0.8.

We evaluate our approach using vector- and matrix-routines from the Adept benchmark suite [19]. For each workload we create two different hardware kernels: The first without any optimisations, the second with loop pipelining for the innermost loop by manually inserting the loop pipelining pragma available in Vivado HLS. All kernels run at a frequency of 250 MHz and the pipelined kernels were scheduled with an *initiation interval* of 1. For all benchmarks, we were able to offload the workload loop to the FPGA by just adding a simple OpenMP target pragma, demonstrating both the functionality and convenience of our implementation.

We compare our implementation to the x86-offloading implementation present in LLVM. The x86-executables were compiled with -O3 and use 4 cores for the OpenMP parallelised workload in each benchmark. Runtimes (normalized to the x86 offloading execution) are shown in Fig. 3. For the current prototype, the offloaded FPGA execution is slower (geomean ∼6.9x and ∼3.4x), with pipelining significantly

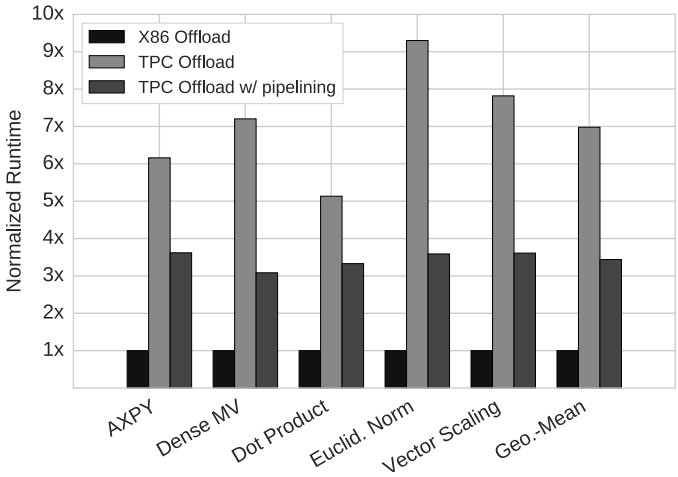

Fig. 3. Normalized runtime.

speeding up the hardware execution (geomean ∼2x improvement over non-optimised kernel). However, bear in mind that the x86 offloading execution uses 4 core CPU at 4.0 GHz, whereas our current proof-of-concept implementation is still limited to a single processing element at 250 MHz.

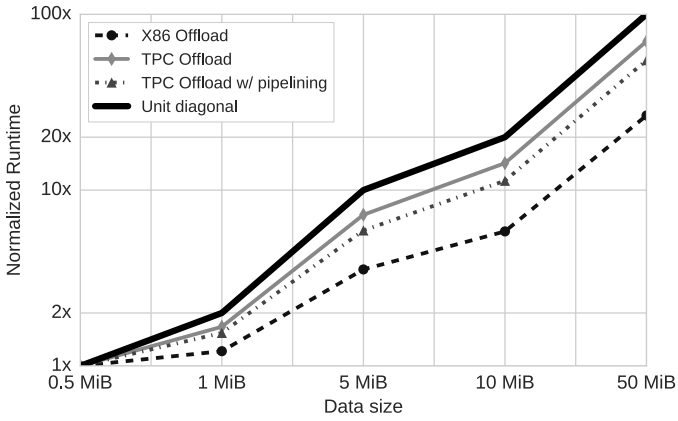

Fig. 4. Normalized runtime vs. data size in the *vector scaling* benchmark.

Despite being slower than the CPUs, the prototype is actually useful to evaluate the *overhead* of our offloading approach compared to the x86 offloading implementation. We measure the runtime for different data sizes, ranging from 0.5…50 MiB of the input vector in the vector scaling benchmark. The bold black line in Fig. 4 depicts the unit diagonal between input data size and runtime (note the logscale!). While the

overhead of offloading dominates for all three targets in the case of 0.5 MiB, all targets exhibit a gradient <1 for larger data sizes. Considering the fact that the overhead of our offloading implementation includes hardware initialization, data transfers via PCIe, and interrupt latencies, we conclude that our offloading approach via TPC is competitive.

VI. Conclusion and Future Work

In this work, we have presented the first fully functional implementation of the OpenMP device offloading model for FPGAs. Our implementation seamlessly integrates with the existing LLVM offloading infrastructure, and enables users to move computational workloads to a custom FPGA accelerator by simply adding OpenMP *target* directives to their code. Using our tool flow, which combines custom Clang extensions and TPC's automation of the hardware synthesis process, the user can generate a complete FPGA-design, including memory- and host-connectivity, from a single, portable input code. Furthermore, the OpenMP memory mapping clauses allow the user to precisely specify which data to transfer to/from the device memory. Our evaluation then showed that our approach does not introduce excessive overhead to the offloading process.

In our current implementation, a single PE occupies less than 1% of the FPGA's resources (not including PCIe, memory and host connectivity infrastructure). Therefore, in order to improve the performance of FPGA offloading, we intend to make use of coarse-grain parallelism in future work by automatically distributing the computation of parallel OpenMP workloads, such as target team distribute, across multiple identical PEs.

Beyond that, we intend to extend our implementation to additional platforms supported by TPC, e.g., the Xilinx *Zynq-series* reconfigurable SoC systems.

References

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[2] "OpenMP Application Programming Interface - OpenMP Standard 4.5," Nov. 2015.

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

[4] "ThreadPoolComposer," https://git.esa.informatik.tu-darmstadt.de/REPARA/threadpoolcomposer.git, accessed: 01-04-2017.

[5] A. Podobas, "Accelerating parallel computations with openmp-driven system-on-chip generation for fpgas," in *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, Sept 2014, pp. 149–156.

[6] A. Podobas and M. Brorsson, "Empowering openmp with automatically generated hardware," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 245–252.

[7] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 270–277.

[8] A. Cilardo, L. Gallo, and N. Mazzocca, "Design space exploration for high-level synthesis of multi-threaded applications," *Journal of Systems Architecture*, vol. 59, no. 10, Part D, pp. 1171–1183, Nov. 2013.

[9] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca, "Efficient and scalable OpenMP-based system-level design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, Mar. 2013, pp. 988–991.

[10] L. Sommer, J. Oppermann, and A. Koch, "C-based synthesis of area-efficient accelerators for openmp worksharing loops," *Second International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'16)*, 2016.

[11] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the nymble system," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*. IEEE, 2013, pp. 1–8.

[12] C. Pilato and F. Ferrandi, "Bambu: A free framework for the high-level synthesis of complex applications," *University Booth of DATE*, 2012.

[13] J. Barker and J. Bowden, "Manycore parallelism through OpenMP," in *International Workshop on OpenMP*. Springer, 2013, pp. 45–57.

[14] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture," in *International Workshop on OpenMP*. Springer, 2014, pp. 202–214.

[15] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans *et al.*, "Integrating GPU support for OpenMP offloading directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, p. 5.

[16] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura *et al.*, "Offloading support for OpenMP in Clang and LLVM," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. IEEE Press, 2016, pp. 1–11.

[17] Samuel Antao, Carlo Bertolli, Andrey Bokhanko, Alexandre Eichenberger, Hal Finkel, Sergey Ostanevich, Eric Stotzer, and Guansong Zhang, "OpenMP offload infrastructure in LLVM." [Online]. Available: https://github.com/clang-omp/OffloadingDesign

[18] D. de la Chevallerie, J. Korinth, and A. Koch, "ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators," *Highly Efficient Accelerators and Reconfigurable Technologies (HEART), International Symposium on*, 2015.

[19] Nick Johnson, Michle Weiland, Trevor Carlson, and Sudarshan Balaji, "The Adept Benchmark Suite," 2015. [Online]. Available: http://www.adept-project.eu/images/whitepapers/p_590591_1446568406_Adept_Whitepaper_Benchmarks.pdf