Offloading Support for OpenMP in Clang and LLVM

Samuel F. Antao(*), Alexey Bataev, Arpith C. Jacob(*), Gheorghe-Teodor Bercea(*),
Alexandre E. Eichenberger(*), Georgios Rokos(*), Matt Martineau(*), Tian Jin(*), Guray Ozen(*), Zehra Sura(*),
Tong Chen(*), Hyojin Sung(*), Carlo Bertolli(*), Kevin O'Brien(*)
(*)IBM T.J. Watson Research Laboratory, NY USA,
Email: {sfantao,acjacob,gheorghe-teod,alexe}@us.ibm.com
{grokos,mattmar,tjin,gozen,zsura,chentong,hsung,cbertol,caomhin}@us.ibm.com
a.bataev@hotmail.com

***Abstract*—OpenMP 4.5 allows performance portability by enabling users to write a single application code and run it on multiple types of accelerators. Our goal is to deliver a high-performance implementation of OpenMP into the Clang/LLVM project. This paper describes our initial work to fully support code generation for OpenMP device offloading constructs. We describe a new driver implementation to handle compilation for multiple host and device types, which generalizes the current Clang CUDA implementation and supports OpenMP. It can also be extended to any offloading based language including OpenCL and OpenACC. We describe an implementation of the OpenMP offloading constructs in the runtime library, giving details on two critical aspects. First, how data mapping is implemented. Second, how different device code sections in the binaries are handled to enable application execution on different devices without recompilation. We report initial performance on a prototype that extends current LLVM trunk repositories with all our proposed patches plus future ones, showing near-CUDA performance of our solution.**

I. Introduction

OpenMP® [1] is the main programming language for single node parallelism used within the HPC scientific community on large clusters. According to a widely cited report, it accounts for 50% of all applications composing MPI with some other intra-node language at NERSC [2]. Following its extension to target accelerator-based systems starting with version 4, it is projected as the main programming model for the CORAL systems [3] to be procured by the Department of Energy laboratories (DoE) in the near future.

An application portability challenge arises when each architecture supports either a different native language or different optimization patterns for the same language. This is particularly critical for users who need their applications to execute efficiently on systems with different node architectures, e.g., manycore vs. GPU-accelerated nodes.

OpenMP addresses performance portability by: (i) exposing a single programming interface that is architecture independent; (ii) introducing several mechanisms that enable high-performance compiler and runtime implementations. For example, OpenMP allows programmers to express data visibility and consistency across device data spaces in different application components without introducing unnecessary data copies or having to deal with multiple device memory references of the same data. This supports large applications composed of multiple legacy, third-party, and/or independently implemented libraries (e.g. see [4]). In previous work [5] we showed that near-CUDA® performance can be obtained with OpenMP.

In this paper we describe our efforts in implementing of the OpenMP 4.5 specification [6] into the Clang®/LLVM® [7] project. Our main target architecture is a GPU-accelerated system based on POWER® processors and NVIDIA® GPUs. This is comparatively more challenging than for other homogeneous systems. Addressing this challenge requires general solutions that need to be integrated with existing accelerator language support, such as CUDA and OpenCL®.

We describe the following contributions to the Clang/LLVM community:

- We submitted several Clang driver patches that enable a generic accelerator offloading scheme and include support for OpenMP and CUDA.
- We implemented libomptarget, a library that provides OpenMP runtime support for offloading on NVIDIA GPUs and generic ELF-based devices.
- We implemented support for code specialization for devices in the semantic analysis and code generation parts of Clang.

The paper is organized as follows: Section II gives necessary background information on OpenMP. Section III gives an outline of the main logic behind driver patches and gives relevant details for integration. Section IV describes the design of the libomptarget offloading library and its implementation as offered in patches to the LLVM's OpenMP library. Section V describes general design concepts in implementing OpenMP in Clang. Section VI gives an update on performance results that can be obtained using our fully-fledged OpenMP 4.5 implementation prototype. We discuss related work in Section VII. Finally, Section VIII concludes the paper.

II. Background

The following list includes OpenMP concepts that are necessary for the full understanding of this paper (see [6] for the full OpenMP specification):

978-1-5090-3878-7/16 $31.00 © 2016 IEEE
DOI 10.1109/LLVM-HPC.2016.6

- Offloading is expressed by marking a code region with the **target** pragma. User functions can be called from within a target region by declaring them for target execution. The compiler generates device code for all regions marked with target.
- Data mapping is expressed with pragmas through **map** clauses. These can be used to enable visibility of a user memory area (e.g. a variable, an allocated buffer, an array, etc.) on a device memory, and to enforce consistency between different views of the same variable in host and device memory. Allocation in device memory and data transfer to device memory is issued according to a reference count mechanism (for details, see page 216 of the specification [6]).
- Each device is identified by a unique non-negative integer, which can be used in the **device** clause. There is no mechanism at application-level to identify the type of device to be targeted (e.g. CUDA- or OpenCL-enabled). The user can target a device type by passing the compiler an option that determines what device types the compiler should generated code for.
- All OpenMP pragmas are allowed in target regions, with special pragmas that can only be used within target. **teams** is used to create groups of threads and **distribute** allows scheduling a set of teams for execution of a loop. There is no synchronization primitive to perform a barrier between threads belonging to different teams. This makes it natural to implement teams as CUDA threadblocks on NVIDIA GPUs, which share a similar property.
- From a compilation viewpoint, OpenMP pragmas typically result in code transformation and calls to the OpenMP runtime library. For instance, a **collapse** clause on pragma **for** will transform the associated loop nest into an LLVM IR fused single loop implementation. In our implementation, different device types may use different OpenMP runtime interfaces. An OpenMP implementation requires both compiler and runtime library support.

III. Driver Support for OpenMP Offloading

Clang is the C/C++ LLVM *driver* which represents the main entry point for application building. It is in charge of selecting the right commands from the appropriate tool chains given the options passed by the user and the available information about the input files, e.g. known file extensions. Adding support for OpenMP offloading introduces extra complexity in the implementation because for the same set of input files different tool chains need to be invoked, and the partial outputs of each tool chain is combined into a fully functional binary file containing host and device executable images.

Based on early feedback from both users and developers on our initial clang-based OpenMP 4 compiler prototype, we identified a set of characteristics for an offloading-enabled driver:

i. Add as few new options as possible to enable offloading, i.e. reuse existing options and avoid the proliferation of offloading-specific options.
ii. Require minimal changes to existing build systems, i.e. save the users the trouble of having to deal with a new set of files related with offloading.
iii. Operate seamlessly on libraries that contain offloading code, i.e. support libraries whose components contain not only host code but device codes as well.
iv. Support a convenient way to read and modify intermediate files that have a textual format, e.g. assembly and LLVM IR files.

Whereas *i)*, *ii)* and *iii)* aim at reducing the users learning curve when they start using the offloading capabilities, *iv)* eases the fine tuning of the application and helps in the identification of opportunities for performance improvement and bug fixing. In the following of this section we describe the proposed implementation for offloading support in the driver. Some of the material presented herein have been already discussed in LLVM/Clang and OpenMP community and materialized in patches to the project codebase.

A. Current Driver Implementation Overview

Clang's driver implementation consists of mainly two components. One is mostly target agnostic, and its role is to identify the right sequence of commands and dependences between them given the input files' type and the last phase requested by the user (linking, assembling, etc.). The other component contains all the tool chain information that should be used for a given device and is queried by the target-agnostic components to obtain the target-specific arguments to be used in the invocation of the commands.

1) Target-Agnostic Component: Actions and Jobs: In the implementation there is the notion of *action* and *job*. In a first step, taking into account the input file type and other options, the driver builds a graph of actions that represents the required compilation phases and dependencies between them. The driver uses a set of supported actions for the nodes of this graph - Preprocess, Compile, Assembly, Linking, etc. Depending on the user-requested final phase, there may a single graph or one for each input file; multiple files are only combined into a single graph if the root action of the graph is a linking action that combines outputs referring to different input files.

Once all the graphs are obtained, they are scanned from root to leaves, and a sequence of *jobs* will be created for patterns of actions. A job describes the tool, the input files, and other target-specific arguments. Based on the information encoded in a job, the final command is produced to materialize the user request. Multiple actions can be combined and materialized by a single job, when a single tool supports it. For instance, Clang can preprocess and generate code in a single run. For some targets there is an integrated assembler that enables combining code generation and assembler.

2) Target-Dependent Component: Tool Chains and Tools: Different targets may use different tools (assemblers, linkers) or customize the invocation of tools that are used by other targets. All this information about *tool chains* and *tools* is organized in the target-dependent part of the implementation

and is queried during the creation of the jobs. This includes logic to detect include paths, libraries, and tools in the host system.

B. Existing CUDA Offloading Implementation

Support for the CUDA programming model has been contributed in parallel to the OpenMP related efforts in Clang. At the time of writing, it is the only other functional programming model requiring offloading in Clang. The original CUDA implementation includes changes that are specific for CUDA and the devices it is meant to support - GPUs. It is based on the *CUDA host action* and *CUDA device action*.

CUDA compilation is prepared to support multiple GPU devices in a single run. A CUDA device action is used to set which specific device its dependences refer to and it does not map to any specific job.

A CUDA host action is used as a dependence to a host *Backend* action and its role is to mark where a device image should be integrated with the host image. The current CUDA implementation does not support relocation code (no user-specified external symbols to be resolved) and, therefore, does not require any actual linking of device objects. Instead, the device images are incorporated in the corresponding host object, and the resulting binary contains as many device images as compilation units.

In addition to the new actions, the CUDA implementation also introduces the ability to define in the host frontend options meant for the device, as well as define in the device frontend options meant for the host. This ensures that correct macros, include paths, and built-in functions are enabled for both host and device even though they are only meaningful for one target.

Several tool chain modifications were also introduced by the CUDA implementation, namely the definitions of the assembling and linking tools. The linking tool, however, is not meant to do actual linking but to bundle objects obtained for different kinds of GPUs (CUDA compute capabilities) into a single one. Other CUDA-specific modifications are also introduced in the frontend tool mostly to enable an extra input file to be passed when the device image is embedded in the host resulting binary.

C. Proposal of a Generic Offload Action

The existing CUDA implementation has many parts in common with OpenMP offloading support. However, it is tailored for the specifics of the model and contains several limitations that are not compatible with a fully functional OpenMP implementation, like the inability to deal with relocatable code for devices. Therefore, our initial step toward a full OpenMP driver support is to make the CUDA implementation more generic. We propose a generic *offload action* that contains information related to offloading that can support different programming models. This action replaces completely the CUDA host and device actions and contains the same information contained in those actions with some additions. This information consists of the offloading kind the action refers to, the tool chain that should be used by the dependences, and the specific device architecture (if any).

The use of the offload action is extended to add *i)* support for several device dependences to a host action - different offloading models can potentially be used simultaneously - and *ii)* support for a host dependence to a device action - device action may need to use host result files if the implementation requires it.

The use of the generic offload action allows the driver to cleanly support action generation for both CUDA and OpenMP programming models. In particular for OpenMP, the offload actions are used to:

- Specify a host dependence to the device compile action. Unlike CUDA, the device frontend cannot derive what declarations should be emitted based on attributes (e.g. `__device__`). Instead, it relies on metadata generated by the host code generation that specifies all the declarations emitted for the host that should also be emitted for device. Therefore, the host compile action is a dependence to the device compile action.
- Specify a device dependence to the host link action. For most OpenMP codes, not supporting relocations is not an option - whatever is supposed to work for the host should work for the device as well. Therefore, the device image is only injected in the resulting host binary after the actual linking of all independent device objects takes place. This injection happens during the host linking action and is controlled by a linker script that is generated by the compiler.

In order to make the implementation more scalable, a common infrastructure to generate offload actions was also proposed. It consists of an action builder that appends dependences to an host action or uses a host action as a dependence. The hooks that make the decision on whether the offload action depends or is a dependence to an host action can be conveniently overloaded for each programming model. This infrastructure is also prepared to propagate the offload kind (e.g. Host, CUDA, or OpenMP) to all its dependences, which allows the different components of the driver that process an action to access relevant offloading information directly from it.

Figure 1 presents the action graph generated by the driver when compiling two source files with OpenMP offloading support for host and a single device.

D. Job Generation

Given the graph of actions, the generation of jobs and their associated commands is done similarly to what was done before the offloading support was available. Actions are scanned from root to leaves and the tool chain information is queried for the different action patterns. New logic was proposed to additionally ensure that offload actions do not interfere with the combining of actions (use of integrated assembler or preprocessor).

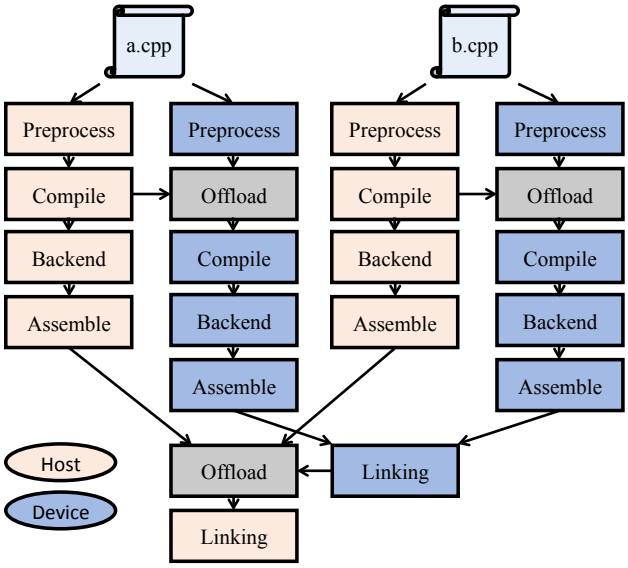

Fig. 1: Action graph obtained when compiling two source files for host and an OpenMP device. An offload action is used to add a host dependence to the device compile actions and add a device dependence to the host linking action.

1) Tool Chain Information Tuning: The tool chain information is extended and tuned in such a way that the queries that take place during the creation of jobs return the desired commands. This was materialized by the following changes:

- Append *libomptarget* to the linking command. For the NVPTX tool chain - the tool chain that supports NVIDIA GPUs - *libomptarget-nvptx* is used. This is different from the case where using the host as a device, where the OpenMP runtime library implementation (libomp) contains all necessary entry points and no additional library is needed.
- Create and append a linker script option for the host tool chain. This linker script controls how the device images are embedded in the host binary and also defines a set of symbols expected by the generated code to easily retrieve information about the different device images and OpenMP mapped kernels and globals.
- Add logic to accept more than one file in the OpenMP device frontend action - the user source file and the resulting file obtained from the host frontend.
- Use *nvlink* as the linking tool for the NVPTX tool chain when generating the linking command for an NVIDIA GPU device. This enables actual linking to take place, instead of the multi-device bundling accomplished by the *fatbinary* tool used by CUDA.
- The assembler for NVIDIA GPU devices is similar to CUDA - *ptxas*. OpenMP uses a similar command except that it enables code-relocation support by default.

E. Clang Offload (Un)Bundler

An highly desired feature for an offload implementation is to save the users from the hassle of having to deal with multiple files generated for the different tool chains. This is especially critical in presence of partial compilation.

In order to tackle this issue, we propose the use of an external tool to bundle multiple files together and/or unbundle a bundle into its individual files - the *clang-offload-bundler* tool. This tool is meant to be directly invoked by the driver relieving the user from management of the temporary files obtained for host and device. An *offload bundling action* is created at the root of the action graph each time partial compilation is used. An *offload unbundling action* is created each time the input file is not a plain source file. The unbundling action is the only action that is meant to produce multiple results, therefore it requires extra logic to ensure that the action is visited a single time during job creation, even though it may have multiple depending actions. If the input file is neither a source file or a bundle, the tool will assume that it is meant only for the host and will generate empty results for the devices that may be active for that compilation.

The offload bundle tool has a bundling and unbundling mode and is prepared to deal with different formats:

- Text format (e.g. assembly, LLVM IR): the tool bundles files with textual format by concatenating them and using as separator a comment with a magic string. This makes the reading/editing of intermediary files as easy as doing it in the individual files independently.
- Binary format (e.g. AST, LLVM bitcode): the tool bundles the binary formats by creating an header that contains the device information and the different offsets at which they are stored in the bundle.
- Object format (e.g. ELF, COFF): the tool uses the logical sections that host object files already provide in order to pack device code in them. Each section has a designated name that will be used to determine later on which device they refer to. By maintaining the format of the host file, the implementation allows the user to employ tools she/he is familiar with that operate on that format. That includes the ability to link the resulting file with other tools - which will likely ignore the section that has device code but still provide correct results for the host. Notwithstanding the object formats are device-specific, we were able to keep the implementation device-agnostic. That was accomplished by invoking Clang during bundling with a custom IR and with variables initialized with the device object assigned to the designated sections, and by using the existing object library already available in LLVM for unbundling.

In the following we show an example on how the tool is invoked to bundle a host (POWER8® - triple *powerpc64le-ibm-linux-gnu*) and device (NVIDIA GPU - triple *nvptx64-nvidia-cuda*) object:

```
clang-offload-bundler -type=o
-targets=host-powerpc64le-ibm-linux-gnu,
openmp-nvptx64-nvidia-cuda
-inputs=a-host.o,a-device.o -outputs=a.o
```

The `-type=` specifies the type of the files to bundle and it takes the default extension for that filetype (in this example it is an object file). `-targets=<programming mode>-<triple>,...` specifies the list of programming models and device triples the bundles refer to. `<programming mode>` can currently take the options `cuda`, `openmp`, or `host`. The order of the list passed to the `-targets=` option has to be consistent with the files passed to the option `-inputs=` when bundling and with `-outputs=` when unbundling. The option `-unbundle` makes the tool operate in unbundling mode.

Separating this tool from the driver enabled shifting all the complexity of bundling from the driver common infrastructure and provided opportunity to expand the current capabilities - e.g. support more complex formats like static libraries which consist of bundles of object files.

F. Options Grouping

In order to have better control in the compilation, users may be interested in passing options that should be observed for a given tool chain. E.g. users may have different include path for a given target, may wish to link against a specific library, or select a given CPU model. In order to avoid proliferation of options meant to be interpreted only by some tool chains we propose option grouping. The following example shows how the compiler driver can be invoked with option grouping to compile the file `foo.c` with OpenMP support for a POWER8 host and support for an OpenMP and a CUDA GPU device with CUDA compute capabilities 3.5 and 3.7, respectively:

```
clang -fopenmp -target
powerpc64le-ibm-linux-gnu
-mcpu=pwr8 -c foo.c -o foo.o
-fopenmp-target=nvptx64-nvidia-cuda
-Llibx_path -llibx -mcpu=sm_35
-fcuda-target=nvptx64-nvidia-cuda
-mcpu=sm_37
-fhost-target -lliby
```

In the example above the options that appear after `-f<programming model>-target=<target triple>` are only forwarded to the target tool chain with triple `<target triple>` when compiling code for that specific `<programming model>`. There is also a reserved *host* `<programming model>` that can be used to start a new group of options that should be forwarded to the host tool chain. As a result, `libx` is only used to compile the OpenMP device code and `liby` is only used for the host tool chain.

IV. Runtime Library: Generic Offloading and NVIDIA GPUs

Offloading mechanisms introduced with OpenMP 4 require an extension of the existing OpenMP runtime library. This was initially materialized in the form of a multi-company design document [8] that lists both the design principles of the offloading logic, as well as the interfaces between the various components. In this section we give a brief description of the design document and we give details on a current proposal of implementation.

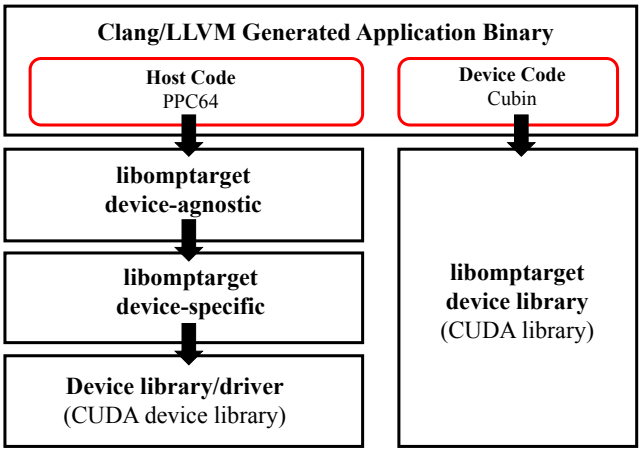

Fig. 2: Component structure of libomptarget where the host is a POWER processor and the device an NVIDIA GPU.

A. Design Principles

Figure 2 shows a graphical representation of the main components of libomptarget for the specific case where the host is a POWER8 processor and the device is an NVIDIA GPU. A first design goal of libomptarget is to separate code generation aspects within the compiler from device-specific mechanisms, which may vary across different architectures. When synthesizing an offloading construct, like "target", Clang generates a set of calls to libomptarget regardless of the device being targeted. In other words, there is no parameter or specific call that the user or Clang itself can generate to express offloading for a specific device type. All *device-agnostic* logic is implemented within libomptarget as a first component facing the application code.

A second logical component in the design is device-specific and can be instantiated multiple times, each for a different device. The role of this components is to implement calls from the device-agnostic level using low-level device-specific libraries or, more generally, mechanisms. An example of such a device-specific logic is the one we implemented for NVIDIA GPUs, which uses CUDA device driver calls. The design includes the possibility of these different device-specific instances to coexist within the same libomptarget instance. For this purpose, these device-specific runtime libraries are implemented as plug-ins of the first component.

Through this double layer of support, we can abstract away device-specific mechanisms and merge them in single abstractions at the device-agnostic level. In a clear example, OpenMP supports asynchronous target scheduling. The device-agnostic component should abstract away from device-specific

mechanisms and integrate device-specific mechanisms with the second component. For instance, the plug-in for CUDA-enabled device can use the *stream* mechanism, while other architectures may expose a *queue*-like mechanism (e.g. OpenCL-based architectures).

A main task accomplished by the device-agnostic library through interactions with the device-specific plug-ins is to build so-called *offloading tables* that map the addresses of host functions and global variables into their corresponding entities in each device. There is one such table for each device, and each device type can use its own object format, e.g. different from the host one, to encode its device entry functions (kernels) and variables. The compiled user code references its unique addresses that identify the kernels and global variables as registered in its offloading table. This is enabled at binary registration time, when a target region for a specific device is encountered, the device plug-in populates the offloading table with device-specific objects and returns it to the device-agnostic component. When a host address for a function or a global variable is forwarded to the runtime library by the application code, it is translated using that table. In the implementation part of this section we describe how this translation happens for two relevant device types upon binary registration.

There is a third component in libomptarget which is not directly connected (called from) the other two levels. This component implements OpenMP runtime library calls generated by the compiler within offloaded regions, i.e. region executing on the device. It is target-specific and can be instantiated differently for different devices. These so-called *device libraries* are offered to implement the OpenMP compiler-generated calls on various devices. These could be a mere host implementation of OpenMP, when offloading to host-like devices, or a complete re-implementation as is the case for NVIDIA GPUs. A potential goal is to homogenize the various implementations while at the same time retaining mechanisms for optimization opportunities. For instance, dynamic parallelism on NVIDIA GPUs could be used to implement varying parallelism degrees which are typical in and required by OpenMP, without disrupting host code generation schemes. However, careful analysis (see [9], [10]) showed that performance is heavily affected by the use of such mechanisms, and a different runtime interface and code generation scheme is required for performance reasons.

Finally, the runtime library design is based on a *fat binary* organization of application binaries (see Section III). The same binary can contain multiple sections each related to a different device type, i.e. containing generated code for a different architecture and a different OpenMP device library. This permits avoiding recompilation of the binary when switching between architectures with same host but different accelerator/device. To support multiple devices in a same application binary, libomptarget is designed to support multiple device plug-ins and OpenMP runtime libraries within the same execution. It is worth noticing that this is an over-design as current OpenMP specifications only allow one device type.

B. Implementation

Our implementation of libomptarget is proposed in three parts corresponding to the three components described in the design document. The first component is the target-agnostic library and that is the only library the host code has to interface with in order to map data and transfer execution for devices.

The second component consists of two device plug-ins and is instantiated as two dynamic libraries, one for a generic ELF-based host processor and one for CUDA-enabled devices. The former enables emulation of a device using host threads and is meant to be used mostly for debugging purposes. The latter allows execution of OpenMP target regions with NVIDIA GPUs. Each of these plug-ins are loaded by the first device-agnostic component. In this way, two different implementations of the same abstract interface for the second component can coexist within the same application execution. We use libdl to load the plug-in logic in the first component of libomptarget and the obtained handler is associated with the supported device types.

The third component is only instantiated for NVIDIA GPUs as a static CUDA library. That is not required when offloading regions use the host as a device, as the OpenMP implementation on the host without further specialization can be directly used.

This implementation scheme represents an initial version of libomptarget for the LLVM project. It is intentionally kept as clean and simple as possible to facilitate the review phases from the most critical aspects. At the end of this section we discuss optimizations of the implementation and other performance aspects relevant for future work on top of the baseline implementation once it will become part of the LLVM project.

C. Data Mapping

OpenMP data mapping is used to program visibility and consistency of data across host and device memories. The specifications contain detailed design hints that can be used to implement a high-performance data mapping support, where data copying across devices can be minimized. The main mechanism is called *reference count* and it allows avoiding data copying when a data has already been mapped previously during execution. This enables an application composed of multiple independent libraries to avoid unnecessary copies, while at the same time prevents incorrect or missing data mapping under different uses of the same library by different applications. For instance, a data mapping clause used within a library will result in data allocation and copy if the device reference count associated with the data maintained by the runtime is equal to zero, meaning that the data was not mapped earlier. It will not result in an allocation and copy if the reference count associated with the data is larger than zero.

Data mapping is implemented by the first two components of libomptarget. The device-agnostic component is not directly involved with the lower level device actions required by mapping. It only keeps track of the correspondence between a host and device addresses of variables. Each device-specific

plug-in implements the actual data allocation and movement and returns to the first component all information necessary to retrieve mapped data after the initial allocation.

The device-agnostic component of libomptarget contains a std::list of structs per device, each related to a data currently being mapped on a specific device. The struct contains the data base pointer, the beginning and ending of the array section, if present, and the reference count information. It is task of this first component to update the reference count of each mapped data.

The choice of std::list is to avoid reallocations when inserting new data mapping within the data structure, which permits fast insertion of initial mappings. We will extend this implementation with a hash map structure for fast retrieval of items in this list.

1) Data Mapping for Generic ELF-enabled Devices: The plug-in library for a generic ELF-based host processor implements the expected behavior. Data allocation is implemented as a call to *malloc* with the size specified by the user. Copies of data between host and device (in this case the host itself acts as a device) is implemented with calls to *memcpy*. De-allocation of data in the plug-in is implemented as a call to *free*.

2) Data Mapping for CUDA-based Devices: The plug-in library for CUDA-enabled devices is implemented with calls to the CUDA device library. Initial data mapping is implemented using *cuMemAlloc*, copying with *cuMemcpyHtoD* and *cuMemcpyDtoH*, and de-allocation with *cuMemFree*. The plug-in library maintains a list of contexts, one per device being used by the application. Each call to the plug-in is preceded by a call to cuCtxSetCurrent to set the context to the correct device. This is needed because the same CUDA plug-in instance supports multiple devices identified by the user with an integral ID that is matched to the context.

D. Implementation of Binary Registration

As shown in the previous section, application binaries contain device code organized in sections, with multiple device code types possibly present in the same binary. The device-agnostic component of libomptarget does not deal with each device section directly, but it delegates registration of them within each specific device type to the second level plug-ins.

As specified by the design document, the first level of libomptarget only handles tables that map each function entry point on the host and each mapped statically allocated variable to the corresponding object on each device. These tables are initialized upon library/section registration on a device by the plug-in, and returned to libomptarget for successive use.

The reason for this separation of functionalities is due to the different ways in which each device handles registration. This is the case for a ELF-based binary targeting a host CPU like PPC64, where we use libdl, and NVIDIA GPUs where the CUDA device driver interface is the only available tool for this purpose at the present time.

1) Binary Registration in ELF-enabled Devices: The ELF-enabled plug-in uses libelf to inspect and retrieve information from the ELF section of the binary, and libdl to dynamically load and use a set of entry points and global variables. The goal is to load an ELF section indicated by the device-agnostic level and to populate a list of function pointers and static data pointers and return it to the caller for successive use.

As described in the libomptarget design document, the device-agnostic library indicates to a plug-in library the start and end pointers of the relevant ELF section. These are determined by using appropriate external variables defined with the linker script employed by the driver as described in the previous section. The plug-in library for ELF devices performs the following actions to register a binary:

- It locates the requested ELF section within the binary, obtaining an ELF section handler, using the section start address indicated by the device-agnostic component.
- It copies the content of the ELF section into a temporary file for loading from libdl. This will be optimized in future versions of the library by loading the section in memory.
- It scans the ELF file looking for the section containing entries information. The offset of the section is translated to the dynamic address where the library is loaded and the result (the address of the table) is returned to the device-agnostic component.

2) Binary Registration in CUDA-based Devices: We have seen that, when compiling and linking for NVIDIA GPUs, LLVM generates PTX assembly. PTX is still an intermediate representation, and not the final executable GPU code. It is successively translated into the NVIDIA-specific executable *CUBIN* (or object) file by two tools available as part of the CUDA toolkit, namely ptxas and nvlink.

The CUBIN object file is embedded in the fat binary as an opaque object of which libomptarget is only communicated the start and end address in the binary. The CUDA plug-in implementation of libomptarget uses the the memory address of the CUBIN object file to *load* its content onto GPU memory. This is done by passing the CUBIN image address to the *cuModuleLoadDataEx* function. This CUDA driver call returns an opaque object called CUDA module (CUmodule type) which can be used to retrieve handles for global variable and functions.

The CUDA module variables returned by each CUBIN registration are stored per device by the plug-in along with the contexts, in std::vector's. This enables multiple registration and de-registration of CUBIN files in presence of dynamically loaded libraries using OpenMP offloading constructs.

To retrieve a kernel function handle from a CUDA module, libomptarget uses the CUDA device driver call *cuModuleGetFunction*. This takes as input the name of the function entry point to be retrieved, as a character string, and returns a further opaque object (CUfunction type). libomptarget stores the names of the target entry points in its internal offloading table, where there is an entry for each target region of the input program containing the character string. Similarly, global variable references are obtained as opaque CUdeviceptr's by passing the global variable name to the *cuModuleGetGlobal*

function. These opaque objects, i.e. CUfunction and CUdeviceptr, are stored in the offloading table returned from the plug-in to the device-agnostic component of libomptarget. This allows libomptarget to re-use the opaque object directly when referencing multiple times to the same object or kernel function, eliminating the cost of obtaining the handle multiple times.

V. Code Generation

At the time this paper is written, full support for OpenMP 4.5 language is not available in Clang when targeting GPUs. Several steps have been undertaken in both semantic analysis and code generation phases to ease the introduction of main patches. In this section we discuss some of these points: the interaction between semantic analysis and code generation; the introduction of *PerPostActions* to enable device-specific code generation.

A. Coordination of Semantic Analysis and Code Generation

The OpenMP implementation in Clang leverages the semantic analysis phase of the various constructs to prepare Clang expressions that will be used later on at code generation time. These *helper expressions* are generated after semantic checks and appended to the AST node being built for each OpenMP construct. This has several advantages. First, code generation becomes cleaner as all expression details (e.g. the data type) is decided upfront at semantic analysis time. Code generation only needs to emit the helper expression required using the existing infrastructure.

Second, in some cases there is a simplified handling of expressions in code generation which results in performance improvements. For instance, when emitting a private clause on an object, semantic analysis can directly access the default object constructors and destructors and append them to the AST node. To do the same in code generation would require code duplication in the scanning of the entire AST to search for the right default constructor and destructor, i.e. an expensive operation.

In the specific case of OpenMP offloading pragmas, we expand this mechanism to build appropriate helper expressions when implementing data mapping. At code generation time, we may need to create several LLVM IR values for each mapped expression. Each expression may contain indirections and array sections that describe non-contiguous memory, therefore for each pointer dereference a new set of map information has to be generated. The semantic analyzer detects all the valid cases for these mappable expressions, recording a list of all the components in each expression that have to be mapped separately. Code generation emits all these individual components taking into account the requirements of the programming model, mapping pointers as first-private variables whenever needed. For each expression, the following values are generated: the base address of the data being mapped; a pointer to the first byte actually being mapped, which may be different from the base address for array sections that do not start at the first element; the size of the mapped data; and the type of mapping (to, from, etc). This information is passed to OpenMP runtime functions [8] whose calls are also emitted during code generation.

```
1 int n = ... // size of dynamic arrays
2 double pi = 3.14;
3 double *a;
4 // allocate and populate a
5 ...
6 #pragma omp target map(tofrom: a[:n])
7   #pragma omp parallel for
8   for (int i = 0 ; i < N ; i++)
9     a[i] += pi;
```

Fig. 3: Example of target region with mapped variables to show how the helper expression mechanism is used to communicate information between semantic analysis and code generation passes.

To better understand how semantic analysis and code generation are related, consider the offloading region in Figure 3. There is an array section mapped as part of the target pragma (*a*). The compiler generates code to request libomptarget to perform the mapping. The runtime function parameters are: the base pointer (e.g. *a*); the pointer to the first byte being mapped (e.g the address of *a[0]*); the number of bytes to be mapped (e.g. n*sizeof(int)); the type of mapping (e.g. "tofrom"). This information is obtained in semantic analysis as AST expressions. Code generation will emit these expressions to produce LLVM IR that calls libomptarget with the appropriate parameters described above.

B. Device Code Specialization Through PrePostActions

As described in our previous paper [9], code generation of OpenMP target regions is significantly different when targeting a GPU or a host CPU. On the GPU, it is required to generate appropriate coordination code as threads cannot be simply put in a idle state once they are activated. In code generation this means that we need to create additional control code before and after a pragma is classically handled similarly to the host.

This is achieved with an OpenMP-specific mechanism of PrePostActions, i.e. code generation and parameter configuration commands that are invoked before and after an OpenMP construct code generation is performed. For instance, this is used to generate special code when emitting a target region implementation for NVPTX target. The following code generation actions are needed upon entry:

- Exclude all unnecessary threads for execution
- All worker threads are required to call a special worker assignment function, while the master thread remains in the same function and continues executing sequential code. This is achieved by checking the thread identifier against a pre-defined macro.
- Workers in the work assignment function loop waiting for the master thread to signal that some work is to be executed.
- The master thread signals tasks to the workers every time a parallel OpenMP construct is encountered and then

signals termination to the workers when it reaches the end of the offload region.
- When termination is signaled from the master to the workers, all threads converge back into the kernel entry point and return to the host.

This sequence of code generation statements is created within the NVPTX-specific code that overloads part of the default (CPU) implementation. It is therefore instantiated with the default interface when emitting the content of the outlined region related to the pragma target. As expected, this is only done when targeting NVPTX: the related code emission entry point used for the CPU does not contain/require any specialization code.

VI. Preliminary Performance of OpenMP on NVIDIA GPUs

To evaluate the performance of our implementation we used the LULESH [4] proxy application. The architecture used in these experiments includes an NVIDIA K40m GPU, with a POWER8 host, using CUDA 7.5. As one of the main proxy applications published by Lawrence Livermore National Laboratories, LULESH has been ported to several architectures including NVIDIA GPUs using CUDA. Similarly to previous work presented in [5], we perform a two-way comparison between the OpenMP 4.5 accelerator offloading capabilities of Clang and the existing reference CUDA implementation of LULESH [4].

Since LULESH has already been ported to OpenMP 4.0 [4], the use of our OpenMP 4.5 Clang compiler involved minimal changes to the source code.

LULESH consists of several parallel OpenMP loops which have been offloaded using the combined construct:`target teams distribute parallel for`. By default, the schedule chosen by the compiler for the offloaded single loops is static with a chunk size of one. This is a simplification from the previous OpenMP 4.0 implementation where such a schedule had to be made explicit by adding the `schedule(static, 1)` clause. A chunk size of one makes a significant performance difference since it ensures coalesced memory accesses when executing the target region on the GPU.

In the previous version of the compiler the number of registers allowed to be allocated to each thread was capped to 64. In the current version this restriction has been eliminated. The number of registers addressable per thread has been increased up until the hardware imposed limit of 255. This is similar to what NVCC does, where no software imposed cap on the number of registers exists. The Clang compiler is therefore responsible for starting an appropriate number of threads such that the total number of allocated registers does not exceed the hardware limit for a given SMX.

An important optimization which can be performed early on is the tuning of the number of teams and threads. In the previous version of the compiler tuning of these parameters involved performing an exhaustive search through all admissible values. In the current version of the compiler, we pick a number of threads, T, using the `thread_limit(T)` clause, and a number of teams equal to the size of the iteration space divided by the number of threads: $NdivT + 1$. The number of threads may be changed by the compiler to avoid running out of registers.

In Table I we present the results of the comparison between the CUDA version of LULESH compiled with NVCC and LULESH compiled with Clang supporting OpenMP 4.5 and 4.0 versions. We show execution times of all kernels in LULESH. The comparison is only approximate since for certain kernels on the CUDA side there may be several parallel loops on the OpenMP side. Although results obtained with OpenMP 4.0 have been partially mentioned in previous work [5], this is the first time results have been published for all LULESH OpenMP 4.0.

The CUDA implementation is, in general, the fastest while the OpenMP 4.5 implementation is showing improved performance for the more compute intensive kernels. Overall, the total runtime the OpenMP 4.5 implementation spends performing kernel level computations is twice that of CUDA but less than half of the one of OpenMP 4.0.

All kernels are affected by the latest improvements to LLVM code generation which have impacted register allocation. For smaller kernels such as OpenMP kernel 1, the number of allocated registers has increased from 18 to 22 which may account for more instructions being generated and, consequently, an increase in runtime. There are also small kernels, such as kernel 2, for which the number of registers allocated has decreased from 24 to 22, and the runtime has remained almost the same. For kernel 4, the number of registers remained unchanged (i.e. 42 registers) while the runtime improved slightly.

Kernel slow-downs can be attributed to having a different number of teams and threads. The number of teams used in the new LULESH code depends on the size of the iteration space and the number of threads. By using the same number of threads for all kernels, the smaller kernels do not reach the same performance with OpenMP 4.5 as with OpenMP 4.0.

Long-running kernels, on the other hand, show significant speed-ups compared to their OpenMP 4.0 counterparts of up to 4.8x for kernel 3. Both Clang OpenMP 4.0 and OpenMP 4.5 toolchains use the same strategy for generation code for the `target teams distribute parallel for` combined construct. The difference in performance occurs due to several changes which lowered the number of allocated registers (as reported by the nvlink tool): changes to LLVM's code generation and changes to the OpenMP 4.5 specification (and therefore implementation) which states that all pointers are `firstprivate` by default. The latter leads to the elimination of temporary registers used for dereferencing arguments passed to the kernel function.

The reduction of the number of allocated registers is significant. The number of register for kernel 16 has dropped from 176 in the OpenMP 4.0 Clang compiler to about 95 for the new compiler. This is significantly closer to the register usage of the equivalent CUDA kernel that uses 58 registers in total.

CUDA Kernel	NVCC (us)	OpenMP kernel	Clang OpenMP 4.5 (us)	Clang OpenMP 4.0 (us)
		1: CalcForceForNodes	4.5	2.8
CalcVolumeForce	12.1	2: InitStressTermsForElems	3.2	3.1
		3: IntegrateStressForElems_1	15.7	73.1
AddNodeForces	7.6	4: IntegrateStressForElems_2	13.1	15.1
		5: CalcHourglassControlForElems	35.9	59.4
		6: CalcFBHourglassForceForElems_1	41.1	115.2
		7: CalcFBHourglassForceForElems_2	15.4	17.4
CalcAcceleration	2.7	8: CalcAccelerationForNodes	4.8	4.3
ApplyAcceleration	4.2	9:ApplyAccelerationBoundaryConditionsForNodes	5.4	4.8
CalcPosAndVel	3.1	10:CalcVelocityForNodes	4.8	4.8
		11:CalcPositionForNodes	4.7	4.1
		12: CalcKinematicsForElems	32.7	58.4
CalcKinematics	15.1	13: CalcLagrangeElements	7.8	6.5
		14: CalcMonotonicQGradientsForElems	13.8	40.1
CalcMonotonic	9.3	15: CalcMonotonicQRegionForElems	11.2	15.7
ApplyMaterialPr	80.3	16: ApplyMaterialPropertiesForElems	43.4	101.3
Total	134.4	-	257.5	526.1

TABLE I: Execution time of LULESH kernels on the K40 NVIDIA GPU using the NVCC and Clang compilers.

Register usage with the new compiler is still fairly high when compared to the CUDA implementation. For kernels for which we have no direct comparison to a CUDA kernel (i.e. kernels 1 to 7, 10, 11 and 12 to 14), register usage is responsible for the overall much larger runtime on the OpenMP side. For kernels for which we can establish a one-to-one correspondence, the number of registers used by the Clang toolchain is around 1.5-5 times larger than that of CUDA. An extreme case is kernel 15 which uses 149 registers with OpenMP 4.5 and only about 32 registers on the CUDA side. One of the main reasons for the high register count in this case are the calls to runtime functions which may increase the register count with up to 100 or more registers. Since we are running a small problem size, we minimize the impact on performance due to frequent register spilling. In this way we contain the spilling of registers to local L1 memory only. Increasing the problem size, increases the number of teams and therefore the local memory footprint.

In case of kernel 16, the runtime is improved beyond that of CUDA. At a closer inspection of the PTX code produced by NVCC, we notice the heavy usage of non-coherent loads from read-only memory as compared to the code produced by Clang. In general, the usage of non-coherent loads leads to improved performance. In certain low bandwidth use cases, non-coherent loads may negatively impact performance. Currently, Clang lacks the ability to leverage the read-only memory of the accelerator and therefore does not use any non-coherent loads. This favors workloads which do not involve memory intensive operations. In our tests we did not use shared memory support to give a fair comparison: while shared memory can be natively programmed in CUDA, OpenMP does not currently have a mechanism to select a special memory type for variables. Clang supports the keyword *__address_space* for variables that in NVPTX backend maps to shared (*__address_space(3)*) and constant (*__address_space(4)*) memories. While this can be used to program shared and constant memories in an OpenMP program, it is not a portable OpenMP construct.

VII. Related Work

Stream Executor (SE) is a project that aims to deliver offloading support for accelerators within the LLVM community through CLANG extensions. Although it has similar goals and design [11] as OpenMP, there are significant user-level differences.

Similarly to OpenMP, Stream Executor (SE) abstracts the complexity and vendor-specific aspects of offloading to accelerators. SE exposes a single C++-based runtime interface that supports both CUDA and OpenCL specific host interfaces. For NVIDIA GPUs, SE builds on top of the CUDA device driver replacing the CUDA runtime library. Users can offload a kernel to CUDA or OpenCL platforms using the same SE runtime calls by passing a different type of platform to the related runtime call. A platform type is identified by a string, such as “cuda” or “opencl”. In the runtime interface exposed by *libomptarget* to applications, devices are logically identified through an integer, and this is independent of the supported programming interface (i.e. CUDA, OpenCL, etc.). The target device type is selected by passing the `-fopenmp-targets` compile-time option to Clang.

Unlike the OpenMP approach described in this paper, SE requires that any user kernel be implemented in the native programming language of the device [12]. For example, a kernel offloaded to an NVIDIA GPU has to be written in CUDA. This is in stark contrast to the OpenMP approach which leverages compiler support for generating device-specific code given a single version of the input kernel (C/C++ or Fortran) annotated with OpenMP directives. Users therefore maintain a single version of their application and can run on different device types by compiling with the appropriate `-fopenmp-target` option.

SE relies on users to make explicit calls for allocating memory on device and transferring between host and device memory. OpenMP, Instead, translates data mapping directives

into runtime calls to *libomptarget*. In turn, *libomptarget* makes appropriate calls to device memory data allocation and transfer functions according to the OpenMP reference count semantics. This allows OpenMP to avoid the need of maintaining explicit device data references in the user code.

SE's Clang extensions described in[11] perform a subset of Clang OpenMP functionalities. It extracts interface information, such as the type and number of parameters, for each input user kernel. This is then passed to a SE kernel registration function whose call is automatically generated in place with the extracted parameters. Similarly, OpenMP generates *libomptarget* function parameters (i.e. type and number of kernel arguments) by analyzing the input code, but it relies on outlining of offloading regions marked with `pragma target`.

A main merging possibility between the two projects is in the host runtime library. *libomptarget* and SE share most characteristics related to device abstraction and runtime calls. Common functionality between the two projects includes the interfacing towards CUDA and OpenCL device drivers. Performance considerations play a major role in this decision, as *libomptarget* is tailored to interface with code generated from an input OpenMP program. The extent of this interface/performance dependency will be explored in future work.

gcc®supports offloading to external devices [13] such as Intel®Xeon Phi®and NVIDIA GPUs. Like libomptarget: (i) gcc uses function tables for binary registration onto devices; (ii) libgomp®handles device-specific offloading libraries as plug-ins.

VIII. Status and Future Directions

The status of the patches at the time this paper was written is as follows. Patches to the driver are being reviewed and gradually accepted, and as such merged into Clang. As mentioned in the paper, libomptarget was published as three library component patches. These received reviews and are being considered for inclusion as a subproject of LLVM's OpenMP runtime library. The code generation patches described in this paper are already integrated as part of Clang, including the use of helper expressions and Pre-PostActions.

The next steps in enabling offloading, after all patches described in this paper are accepted, is to provide an implementation of common OpenMP parallelism, worksharing, and tasking constructs on NVIDIA GPUs. This requires several code generation patches that represent an extension of our previous work [10], [9]. They are being produced at the time this paper was written and were published as a full LLVM project fork and extension on github® [14].

Acknowledgment

The authors would like to thank scientists at U.S. DoE laboratories, NVIDIA, TOTAL, and GENCI for their valuable input to the optimization process of the OpenMP compiler and runtime libraries for the OpenPower platform. This paper is partially supported by the CORAL project LLNS Subcontract No. B604142.

References

[1] "OpenMP standard webpage," http://openmp.org/.

[2] Y. H. He, A. Koniges, R. Gerber, and K. Antypas, "Using OpenMP at NERSC," September 2015, https://www.nersc.gov/assets/Uploads/HelenHe-OpenMPCon-2015.pdf (08/24/16).

[3] "CORAL award announcement," http://energy.gov/articles/department-energy-awards-425-million-next-generation-supercomputing-technologies.

[4] "LULESH webpage," https://codesign.llnl.gov/lulesh.php.

[5] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Performance analysis of OpenMP on a GPU using a CORAL proxy application," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, ser. PMBS '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/2832087.2832089

[6] *OpenMP Application Program Interface*, Version 4.5 ed., OpenMP Language Committee, July 2013, http://www.openmp.org/mp-documents/openmp-4.5.pdf.

[7] "The LLVM compiler infrastructure webpage," http://llvm.org/.

[8] S. Antao, C. Bertolli, A. Bokhanko, A. Eichenberger, H. Finkel, S. Ostanevich, E. Stotzer, and G. Zhang, "OpenMP offload infrastructure in LLVM," 2016, https://github.com/clang-omp/OffloadingDesign.

[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, and K. O'Brien, "Integrating GPU support for OpenMP offloading directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:11. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833161

[10] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 12–21. [Online]. Available: http://dx.doi.org/10.1109/LLVM-HPC.2014.10

[11] J. Henline, "Documentation for StreamExecutor open source proposal," https://github.com/henline/streamexecutordoc.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[13] "Offloading support in gcc," August 2016, https://gcc.gnu.org/wiki/Offloading.

[14] "Source code for full OpenMP 4.5 implementation with offloading to NVIDIA GPUs," September 2016, https://github.com/clang-ykt.