OpenMP GPU Offload in Flang and LLVM

Güray Özen, Simone Atzeni, Michael Wolfe
Annemarie Southwell, Gary Klimowicz
NVIDIA Corp.
{gozen, satzeni, mwolfe, asouthwell, gklimowicz}@nvidia.com

***Abstract*—Graphics Processing Units (GPUs) have been widely adopted to accelerate the execution of High Performance Computing (HPC) workloads due to their enormous computational throughput, ability to execute a large number of threads inside SIMD groups in parallel, and their use of multithreaded hardware to hide long pipelining and memory access latency. However, developing applications able to exploit the high performance of GPUs requires proper code tuning. As a consequence, computer scientists proposed different approaches to simplify GPU programming, including directive-based programming models such as OpenMP and OpenACC. Their intention is to solve the aforementioned programming challenges with a directive-based approach which allows the users to insert non-executable pragma constructs that guide the compiler to handle the low-level complexities of the system.**

Flang, a Fortran front end for the LLVM Compiler Infrastructure, has drawn attention from the HPC community. Although Flang supports OpenMP for multicore architectures, it has no capability of offloading parallel regions to accelerator devices. In this paper, we present OpenMP Offload support in Flang targeting NVIDIA GPUs. Our goal is to investigate possible implementation strategies of OpenMP GPU offloading into Flang. The experimental results show that our implementation achieve similar performance to those of existing compilers with OpenMP GPU offload support.

***Index Terms*—Flang, NVIDIA, GPU, Offload, OpenMP, Clang, LLVM, compiler**

I. Introduction

GPUs have taken a central role in increasing the performance of HPC applications. Engineers and scientists from different fields are using this type of accelerator to speed up their simulations and research to obtain answers faster. However, GPU programming is hard, and it requires a greater effort from the developer to expose a higher degree of parallelism compared to the parallelism achieved by most existing HPC applications. Higher-level programming models, such as OpenMP [1] and OpenACC [2], provide a directive-based approach, which makes it easier for the programmer to express parallelism. Non-computer science experts, such as scientists and engineers of different fields, can parallelize their existing applications in a more straightforward way by adding to specific code regions the directives provided by the language, which guide the compiler to handle the low level complexities of the systems (e.g., GPUs).

OpenMP is the de facto standard for achieving on-node parallelism in HPC applications. Starting from OpenMP version 4.0, the specification introduces target offloading directives to allow programmers to annotate specific regions of code which will be transferred at runtime to an accelerator device. An OpenMP 4.0 compliant compiler manages the offloading directive and generates code for the specific accelerator device.

Fortran is one of the most widely-used programming languages for scientific applications in the HPC community. In 2015, the US Department of Energy and NVIDIA announced the development of a Fortran front end for LLVM [3], [4], which was open sourced in 2017 under the name of Flang [5]. Flang supports OpenMP for multicore but does not have any support for OpenMP offloading to accelerator devices.

In this paper, we describe our work with Flang to implement OpenMP GPU offloading target regions onto NVIDIA GPUs. Our work focuses on delivering an efficient implementation of the OpenMP offloading model by taking advantage of the current LLVM OpenMP project. We aim to maintain compatibility with Clang, seamlessly generating device code for the NVPTX back end [6] in LLVM.

The main contribution of this work involves the Flang front end, from the parsing of the OpenMP offload directives to the code generation for the NVIDIA GPU architecture. We touch all the main phases of a compiler front end, including the compiler driver which guides the compilation toolchain. In our work, we implement the parsing and semantic analysis of the OpenMP offload directives and make the required changes to generate the Abstract-Syntax Tree (AST) enhanced with OpenMP offload information. Furthermore, we modified all the Flang components that involve the transformation of the AST to an intermediate representation of the device code. Eventually, the AST is transformed into LLVM IR and finally into the device code for NVIDIA GPUs.

To summarize, we make the following contributions:

- We show our design for integrating the OpenMP offload model in Flang; our model targets NVIDIA GPUs.
- We describe our implementation which is interoperable with Clang's offload support.
- We conduct performance analysis on different benchmarks. We compare our results to modern

compilers with offload support for NVIDIA GPUs.

This paper is organized as follows. Section II provides some background about the Flang compiler and the OpenMP programming model. We discuss related work in Section III. Section IV describes our implementation and gives relevant details about integrating a GPU offload mechanism in Flang. Section V shows experimental results on well-known benchmarks using our implementation, and we compare them against the results obtained with existing compilers such as PGI, Clang, and IBM XL. Finally, Section VI discusses future work.

II. Background

In this section, we describe the background concepts and details about the LLVM compiler infrastructure and the Flang Fortran front end necessary to understand our work. We also explain the OpenMP offloading model and the challenges of its implementation.

A. LLVM

LLVM Core [3], [4] is the main project of the LLVM Project. It provides a source and target independent optimizer, along with code generation support for different processor architectures. LLVM defines a strongly typed Intermediate Representation inside the compiler called LLVM IR which aims to abstract details from the target architecture. The LLVM IR is the input source for the LLVM back end which translates the IR instructions to code for a specific target architecture (e.g., X86, PowerPC, ARM, etc.).

1) Clang: Clang [7] is a front end for C and C++ which uses and follows the implementation guidelines defined by the LLVM community. One of the key efforts of Clang is to ensure that each feature is structured into logical modules in order to ease the integration and reduce disruption caused by inter-dependencies. Following this modular design, each action accomplished by the consumers of each basic element (e.g., token, AST node) tends to be self-contained, either by extending a default action applied on top of a class or by creating a new class.

2) OpenMP Runtime: The OpenMP sub-project in the LLVM Compiler Infrastructure defines a collection of runtime libraries and plugins to implement OpenMP functionality in the Clang/LLVM compiler. The OpenMP sub-project consists of a main runtime library, known as *libomp* [8], which provides runtime functions and routines to support the OpenMP model during the execution of a program. A second runtime library, called *libomptarget* [8], [9], supports the offloading of OpenMP regions to target devices, including NVIDIA GPUs. *Libomptarget* contains different plugins to support multiple devices in a single program, while *libomp* provides support only for the host device (i.e., CPU multicore).

The implementation of *libomptarget* consists of three parts. The first one is a target-agnostic library, which manages the mapping of data onto the accelerator device and the execution of the OpenMP target regions. The second part contains the device plugins for generic ELF-based host devices and NVIDIA GPUs. The last part is the device runtime library *libomptarget-nvptx*, which is written in CUDA and implements the OpenMP runtime for NVIDIA GPUs.

```
$ flang -### -o myexample myexample.f90 -save-temps
 "flang1"       # .f   --> ILM   [Upper]
 "flang2"       # .ILM --> .ll   [Lower]
 "clang-6.0"    # .ll  --> .bc   [Compile]
 "clang-6.0"    # .bc  --> .s    [Backend]
 "clang-6.0"    # .s   --> .o    [Assemble]
 "/usr/bin/ld" # .o   --> (exe) [Link]
```

Listing 1. An example of the Flang driver to compile a Fortran program.

B. Flang

In this section, we describe Flang's internals and compilation pipeline.

1) Flang driver: The compiler driver is an important part of the compilation pipeline because it allows the user to combine several compiler steps into one. The driver calls a set of tools which compiles the source code into an executable program. The collection of related tools for the compilation process is known as a *toolchain*. For example, different compilation "requests" (commands) can have specific flags that require different toolchains. The driver is responsible for selecting the correct toolchain to proceed with the compilation and produce the executable.

The Clang front end provides a compiler driver for C and C++ which is usually invoked using the command line alias *clang*. The Flang front end extends the Clang driver with Fortran-specific features and defines an alias to the *clang* command called *flang*. Listing 1 shows an example of the Flang driver, and all its components invoked through the *flang* command, to compile a Fortran code.

2) Upper - flang1 : The *flang1* command, known as "upper", is the first step of the compiler invoked by the driver; it reads the Fortran source code as input. The upper consists of three main phases: (1) the *scanner* which turns the Fortran code into tokens; (2) the *parser* which builds the AST from the tokens, (3) and *lower* which lowers the AST into ILM (Intermediate Language Macros) code, which is an internal representation of a subprogram's executable statements created by the semantic analyzer. The ILM code is written in a temporary text file and is the input to the next step.

3) Lower - flang2 : The second step, known as "lower", executes the command *flang2* on the ILM code produced by the previous step. The first phase of lower is the *expander*, which transforms the input ILM code into ILI (Intermediate Language Instructions, also known as PGI

IR). The second phase consists of the *optimizer*, which performs a first degree of optimization at the ILI level. The last phase, performed by *schedule*, converts the ILI to LLVM IR and saves it in a file. Once `flang2` generates the LLVM IR, the other driver steps (compiler, back end, assembler, linker) eventually generate the executable. The steps of the compilation toolchain after the LLVM IR generation are not of interest in this work.

4) OpenMP Directive Processing: Flang supports OpenMP for parallelization on multicore CPUs, based on the LLVM OpenMP runtime. The OpenMP code is handled by the same Flang compilation phases previously discussed. The command *flang1* parses the OpenMP directives and clauses, and performs the semantic analysis before turning them into ILM. *flang1* also produces error messages and warnings in case of directive misuse. An important step during the compilation process is *outlining*. The outlining process creates additional functions whose bodies are the code scope associated with the OpenMP directives. *flang2* is responsible for this outlining, and inserts the necessary runtime calls in the ILI code to manage the OpenMP constructs such as `target`, `parallel`, `task`, etc. This step will be discussed in more detail in Section IV.

C. OpenMP

OpenMP announced support for accelerators with version 4.0, and the specification has been evolving continuously since then. The current stable version is 4.5 which offers a set of directives for offloading the execution of code regions onto accelerator devices. The directives can define target regions that will be mapped to the target device, or define data movement between the host memory and the target device memory. The main directives are `target data` and `target`, which create the data environment and offload the execution of a code region on an accelerator device, respectively. Both directives can be paired with a `map` clause to manage the data associated with it; the map clause specifies the direction (`to, from`, or `tofrom`) of the data movement between the host memory and the target device memory. Another important directive defined by the OpenMP specification is `teams`, which creates teams of threads. The programmer can control the number of teams and the maximum number of threads in each team by specifying the values with the clauses `num_teams` and `thread_limit`, respectively. All the threads in a team, except the master, wait to begin execution until the master thread encounters a parallel region. The `distribute` directive specifies how the iterations of one or more loops are distributed across the master threads of all teams that execute the team region. For instance, the loop at line 2 in Listing 2 is parallelized by the master threads of each team, while the loop at line 6 is parallelized across the threads within a team.

```
1 !$omp target teams distribute map(tofrom:a) map(to:b)
2 do i = 1, N
3  alpha = find_alpha()
4
5  !$omp parallel do reduction(+:alpha)
6  do j = 1, N
7    call do_process(a, b, alpha)
8  enddo
9  call finalize(a)
10 enddo
```

Listing 2. Example OpenMP offload code in Fortran.

a) Transformation Challenge for GPUs: OpenMP provides three levels of parallelism constructs: `teams`, `parallel`, `simd`. All three are designed to map threads across loops. These parallelism directives are designed for the fork-join execution model. They can be used together to create different levels of parallelism within the same `target` region. The directives also introduce implicit sequential and parallel regions, and implicit barriers. The mapping of sequential and parallel regions to a GPU is not trivial. GPU kernels can only be launched with the number of threads and blocks specified in advance, so once the kernel is running it means that all GPU threads are available and active. To make different parallelism levels coexist in the `target` region, the compiler needs to follow complex code transformation patterns.

A possible solution is to use "dynamic parallelism" [10], which is a CUDA feature supported by Kepler and newer NVIDIA GPU architectures. Dynamic parallelism allows a CUDA kernel to launch another CUDA kernel, emulating the fork-join model. Experimental results show that dynamic parallelism introduces significant overhead due to the high number of kernel launches [11]; consequently, this approach is not an efficient way to implement the fork-join model in a GPU.

To understand the challenges of OpenMP we use the example in Listing 2, where the `target` directive creates a data device environment with two arrays *a* and *b* as specified in the `map` clause, and a scalar *alpha* which will be implicitly declared firstprivate and automatically mapped in the GPU memory. This code contains a loop at line 2 whose iterations will be distributed among the master threads of each team. Inside of this code block, which will be executed by each team, there are two hidden sequential regions, one at line 3 and one at line 9, which must be executed only by the master thread of the team. One way to run a code region sequentially on a GPU is to guard the region with an if-statement. Later on, at line 7, this scalar will be used by all threads in the team. Since the scalar is in the thread private region, its value must be distributed among the rest of the threads in the team when the code reaches the inner `parallel` region at line 5. This operation can be difficult for the compiler and the runtime in the presence of more complex code blocks. In our example, the compiler will

map all the threads in the team to the loop's iterations. If, however, the user specifies a `num_threads` clause with fewer threads, the runtime still launches the maximum number of threads for the `target`; the compiler must handle the rest of the (unneeded) threads.

III. Related Work

The OpenMP target offload feature was added to the specification in version 4.0. In the last few years, compiler vendors have been working on adding OpenMP target offload support for NVIDIA GPUs. As of this writing, IBM's XL compiler supports OpenMP offload in C, C++, and Fortran; Clang/LLVM supports OpenMP offload in C, and C++. In this section, we describe some of the relevant work that has been done to implement OpenMP GPU offload in these compilers.

The first approach [12], [13], [14], [15], which we call *CUDA-like* in this paper, introduces additional control flow to coordinate GPU threads. When a program encounters a parallel region, the threads first execute a conditional statement; if the threads are needed in the region, they will execute it collectively. In this method, one of the threads is chosen as the master thread, it executes the sequential code regions, writes to a common memory location, and synchronizes with the rest of the threads at the end of a parallel region. The studies [12], [13], [14], [15] using this approach show that this method obtains the closest performance to that achieved by a native CUDA programming model since it does not increase the complexity of code transformation.

In [16], the authors introduce a novel approach based on the concept of control-loop and inspection/execution; this design is easier to implement respecting Clang modularity than the CUDA-like design. They suggest the implementation of a state machine for each code region. In this approach, the master thread leads the other threads to specific states; at the end of a state the threads synchronize before proceeding to the next state. This approach provides a straightforward integration for modular compilers (e.g., Clang/LLVM), unfortunately, it requires complex control flow which affects performance of the generated code.

In [17], the authors propose an efficient fork-join algorithm as a code transformation strategy for the implementation of directive compilation. It reserves a warp (a unit of 32 threads) to serve as helper thread to execute sequential code on the GPU. When the helper thread reaches the parallel region, it can wake up other threads to execute the parallel code.

There are other compilers that support the OpenMP GPU offload targeting NVIDIA GPUs, such as Cray and IBM XL Compiler. In particular, in [18], [19] the authors describe IBM's OpenMP 4.5 implementation with numerous examples.

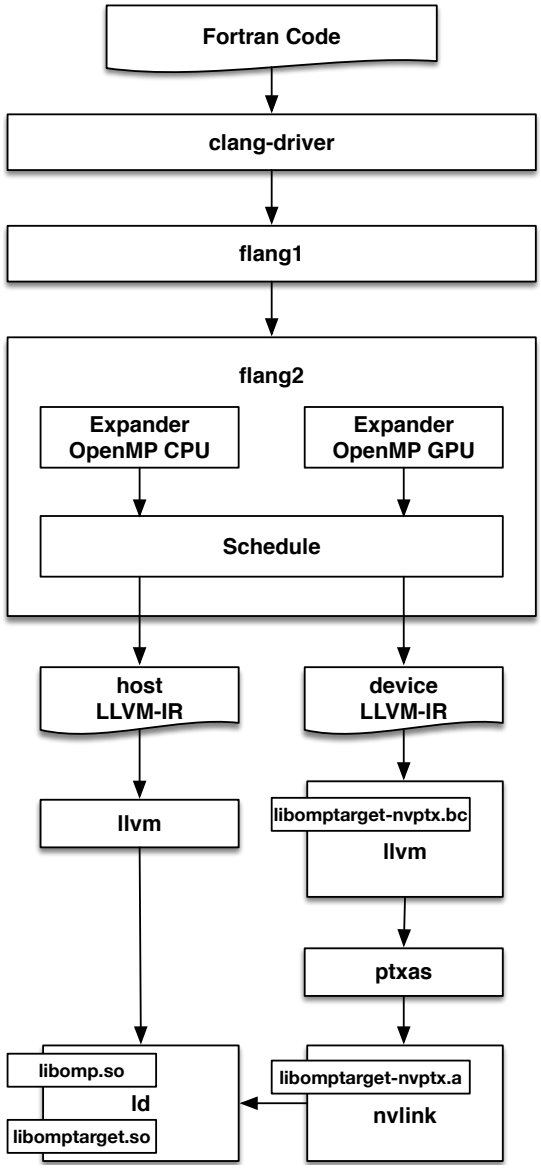

Figure 1. Compilation workflow of Flang.

IV. Integration of GPU Offload Support into Flang

In this section, we describe our OpenMP offload implementation in Flang targeting NVIDIA GPUs. We give details about the design principles and our implementation choices.

A. Design Principles

In our implementation of OpenMP GPU offload support in Flang, we aim to keep the same design as Clang and guarantee full interoperability. In fact, we rely on existing tools provided by the Clang front end, and we follow its modular implementation. The first design goal is to leverage the Clang/LLVM compiler driver, which already offers support for OpenMP offloading targeting NVIDIA GPUs. The second goal is the interoperability between Flang and Clang, which we achieve by using

```
$ flang -o daxpy daxpy.f95 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda

Steps   Tool                    Action                                  Inputs          Outputs
(1)     "clang-offload-bundler" [unbundler]                             ["daxpy.f95"]   ["tmp1.f95","tmp2.f95"]
(2)     "flang1"                [(host)   flang-frontend - Upper]       ["tmp1.f95"]    ["tmp1.ilm"]
(3)     "flang2"                [(host)   flang-frontend - Lower]       ["tmp1.ilm"]    ["tmp1.ll"]
(4)     "clang-6.0"             [(host)   Compile]                      ["tmp1.ll"]     ["tmp1.s"]
(5)     "flang1"                [(device) flang-frontend - Upper]       ["tmp2.f95"]    ["tmp2.ilm"]
(6)     "flang2"                [(device) flang-frontend - Lower]       ["tmp2.ilm"]    ["tmp2.ll"]
(7)     "clang-6.0"             [(device) Compile]                      ["tmp2.ll"]     ["tmp2.s"]
(8)     "ptxas"                 [(device) NVPTX::Assembler]             ["tmp2.s"]      ["tmp2.o"]
(9)     "nvlink"                [(device) NVPTX::Linker]                ["tmp2.o"]      ["tmp2.out"]
(10)    "/usr/bin/ld"           [(host)   Linker]                       ["tmp1.o"]      ["daxpy"]
```

Listing 3. An example of the Clang/LLVM driver to compile a Fortran program using OpenMP GPU Offload.

the same code generation logic to ensure compatibility with the LLVM OpenMP runtime. Figure 1 illustrates Flang's compilation workflow with OpenMP offload support for NVIDIA GPUs. This support can be enabled by passing the command line option *-fopenmp-targets=nvptx64-nvidia-cuda* to the driver.

We present our initial design which provides driver and code generation support. We have implemented a CUDA-like code transformation scheme based on a set of OpenMP combined directives. This implementation choice is driven by the fact that the combined directives are the most common pattern used in OpenMP, and the CUDA-like code transformation scheme introduces the least complexity to the generated code. At the time this paper is being written, we do not fully support the OpenMP 4.5 accelerator model for Flang, but we have chosen a commonly used set of OpenMP offload directives.

B. Driver

As we mentioned in Section II-B1, the LLVM project has a driver, *clang*, designed for the Clang front end. In version 6.0 it started supporting OpenMP offloading to NVIDIA GPUs. Clang's driver consists of two main parts: the *clang-offload-bundler* and the *NVPTX toolchain*. The clang-offload-bundler is an external tool in Clang designed to combine multiple files generated by different compiler host and device toolchains. The NVPTX toolchain is used to produce the object binary for NVIDIA GPUs starting from the PTX code generated by the LLVM compiler. It first invokes the CUDA *ptxas* program to assemble PTX instructions into SASS, the low-level assembly language for NVIDIA GPUs. Eventually, it calls the CUDA linker, *nvlink*, to produce the object file for the device.

We adapted the current offloading mechanism of the clang driver to support OpenMP GPU offloading in Flang. Listing 3 shows the overall workflow of flang compiling Fortran code with OpenMP offload support targeting NVIDIA GPUs. Initially, the driver invokes the clang-offload-bundler, which makes two temporary copies, tmp1.f95 and tmp2.f95, of the source input file. Later on, these temporary files are used for the host and device compilation. To adapt this step, we made the clang-offload-bundler aware of the Fortran file extensions.

The driver then invokes `flang1`, `flang2`, and `clang -cc1` in series to compile the code for the host target. The compilation of the device code for the target triple specified by the *-fopenmp-targets* option, which is the NVPTX target in our example, happens when the driver calls `flang1` and `flang2` to generate the device code for the NVPTX back end, then `clang -cc1` transforms the generated code into PTX code. Finally, the NVPTX toolchain produces the object code for the device. As the last step, the driver calls the linker to generate the executable.

The driver's compilation pipeline provides interoperability with Clang, but it reveals two major problems due to the clang-offload-bundler tool. The first one is that given an object file, the driver tries to unbundle it. If this object file was compiled by a compiler other than Flang/Clang, the clang-offload-bundler cannot unbundle it and the compilation process fails. Another issue is that bundled object files can be part of static libraries, which requires unbundling by the clang-offload-bundler. In this case, the driver needs to know where the files of the library are before the linking step, making linking against static libraries impossible. We understand there is an attempt to solve both issues with the clang-offload-bundler in future versions of Clang.

C. Code Transformation

In this section, we describe the code transformation algorithms we implemented in `flang1` and `flang2`. Initially, we enhanced the parser and the semantic analyzer in `flang1` to make them aware of the new device-specific OpenMP offload directives. Then, we added new functionality in `flang2` to generate output code using the NVPTX back end of LLVM.

1) `flang1` Extensions: The upper, `flang1`, was already able to parse, perform semantic analysis, lower the host OpenMP directives, and produce the ILM file. We extended `flang1` with support for the device-specific OpenMP directives and clauses.

```
1 PROGRAM AXPY
2   implicit none
3   integer,parameter :: N = 8192
4   real :: y(N), x(N), a
5   integer :: i
6
7   !$omp target parallel do map(to:x) map(tofrom:y)
8   do i = 1, N
9     y(i) = a * x(i) + y(i)
10  enddo
11 END PROGRAM
```

Listing 4. Fortran example of a target region with mapped variables.

The OpenMP implementation in `flang1` leverages the semantic analysis phase to prepare ILM code for use by `flang2`. During this phase, `flang1` performs semantic error checking and generates the symbol table and the AST. In particular, the AST contains a node for each OpenMP construct in the source code. `flang1` then transforms the AST into ILM code. We expand and adapt this mechanism to generate the appropriate ILM code (from the AST nodes) for the device-specific OpenMP constructs.

2) `flang2` Extensions: We extended `flang2` to generate the LLVM IR for the NVPTX back end. To add this feature we modified four phases of `flang2`: *outliner*, *expander*, *schedule*, *assemble*. We left untouched the phases for reading the input ILM and optimization.

a) Outliner: Outlining is a code generation concept which consists of extracting a function from the code scope of the host function to generate the device equivalent function. Outlining is the most common technique used to implement OpenMP constructs that transform sequential code into parallel code; LLVM's OpenMP runtime is based on this approach. The example in Listing 4 shows the directives `target` and `parallel` which Flang outline into two new functions, and adds the appropriate runtime calls to invoke these functions. Eventually, `flang2` generates LLVM IR code as shown in Listing 5.

Although the outlining concept is implemented in Flang for CPU OpenMP, it is not in a form suitable for the *libomptarget* runtime which we use for Flang OpenMP offload. For NVIDIA GPU offloading, we use the outlined functions that are created in GPU code, since the GPU has a separate memory space and can be called only via the CUDA driver. The *libomptarget* library provides two functions, `tgt_target` and `tgt_target_teams`, to launch outlined functions and process the necessary data transfer for them. These outlined functions are required to have a specific signature—one parameter for each symbol that occurs inside of the host region. With this information, the function can be launched and the data mapping can be done correctly between host and device. The functions require arrays of the base address of the symbols, map-types, and size.

```
1 %struct.BSS1 = type <{ [65536 x i8] }>
2 ; Global BSS1 keeps array x and array y on the main memory
3 @BSS1 = internal global %struct.BSS1 zeroinitializer
4
5 define void @MAIN_() {
6   ; < other IRs>
7   %0 = call i32 @__kmpc_global_thread_num(...)
8   call void @target(i32* %0, ...)
9   ret void
10 }
11 define internal void @target(i32* %A0, i64* %A1, i64* %A2){
12   call @__kmpc_fork_call(..., %par, ...)
13   ret void
14 }
15 define internal void @par(i32* %A0, i64* %A1, i64* %A2){
16   call void @__kmpc_for_static_init_4(...)
17   ; find array y from @BSS1
18   %0 = bitcast %struct.BSS1* @.BSS1 to i8*
19   %1 = getelementptr i8, i8* %0, i64 32764
20   %2 = bitcast i8* %38 to float*
21   ; find array x from @BSS2
22   %3 = bitcast %struct.BSS1* @.BSS1 to i8*
23   %4 = getelementptr i8, i8* %3, i64 -4
24   %5 = bitcast i8* %4 to float*
25   ; <the rest of loop body>
26   call void @__kmpc_for_static_fini(...)
27   ret void
28 }
```

Listing 5. Outlining example by Flang OpenMP multicore. The output LLVM IR is generated from the Fortran source in Listing 4

Flang does not extract the outlined functions in this way, which introduces two issues. The first, as shown in Listing 5 at lines 11 and 15, is that Flang always generates functions with three parameters. The second issue is that Flang's outlining does not create new symbols, pack, and pass them to the function. Instead, it uses the existing symbols no matter where they are located. Unfortunately, this approach cannot be used for GPU offload since the GPU device has its own memory space. In the example in Listing 4, there are two static arrays, x and y, at line 4, these are also read and written in the loop body at line 9. Flang defines them as global variables at lines 1-3 in Listing 5. Thus, in the rest of code, these arrays are read and written via these global variables as shown at lines 18-20 and 22-24. These two arrays should be created on the GPU, and mapped according to their map information specified in the `map` clause at line 7 in Listing 4.

To adapt Flang's outlining to the *libomptarget* interface, we implemented a new outlining mechanism in Flang. The new mechanism extracts a function with parameters for each symbol that is used in the region. In addition, the new outliner generates a true autonomous function which is not dependent on any global or shared variables of the host function. In Listings 6 and 7 we illustrate the output LLVM IR core of the new outliner, in particular one for the NVIDIA GPU and one for the host machine; we will explain the code separation process in the next paragraphs.

The new outlining mechanism extracts the function for the target region (at line 4 of Listing 6) and generates it in a different file. We did not do any outlining in

```
target triple = "nvptx64-nvidia-cuda"
; < other IRs>

define ptx_kernel void @MAIN__1F1L10_([8192 x float]*
      %Arg_y, [8192 x float]* %Arg_x, float %Arg_alpha) {
  call void @__kmpc_for_static_init_4(...)
  ; find array y from %Arg_y
  ; find array x from %Arg_x
  ; find array x from %Arg_alpha
  ; <the rest of loop body>
  call void @__kmpc_for_static_fini(...)
  ret void
}
```

Listing 6. Generated device code for NVPTX back end of LLVM. The output LLVM IR is generated from the Fortran in the Listing 4.

the device code because creating extra functions increases the complexity for compiler optimization, and may also cause a performance slowdown. Thus, the new mechanism does not extract any additional functions for any OpenMP directives after the `target` construct. The extracted function is called by the *__tgt_target_teams* runtime function of the libomptarget library, as shown at line 29 of Listing 7. The base addresses of each function parameter are packed into an array *%args*. Later on, the runtime separates this array into multiple arguments and communicates it to the CUDA driver to pass these arguments to the outlined function.

b) Expander: The primary purpose of the expander phase is to translate the ILMs produced by `flang1` into ILIs, the internal language that is used by later phases of the compiler such as optimization or LLVM IR transformations. During ILM to ILI translation, the ILIs are grouped into blocks, where the extent of a block is determined by certain compiler options and the characteristic of the ILMs control flow. The internal representation is created by the expander, which represents the terminal nodes of an ILI statement. An ILI statement may be a function call, a store instruction, a register move, or a branch. A sequence of ILTs represents a block of ILIs. An ILT is the terminal node of an ILI statement which roughly corresponds to a source language statement. Each ILT includes links to previous and next pointer of the ILI tree. In addition, the expander is also responsible for making the correct ILI transformation from OpenMP ILMs.

OpenMP GPU offloading is a different concept of parallelism than multicore parallelism, thus, it requires different code transformations. To support different types of parallelism in Flang, we extended the current *expander*, in particular the transformation functions of the OpenMP ILM nodes. We did not change the expander's sequential code generation.

c) Schedule - LLVM-Bridge: The LLVM-Bridge consists of *schedule* and *assemble* phases of `flang2`. In the schedule phase, the ILI generated by the *expander* is

```
target triple = "powerpc64le-unknown-linux-gnu"
; < other IRs>

%struct.BSS1 = type <{ [65536 x i8] }>
; Global BSS1 keeps array x and array y on the main memory
@BSS1 = internal global %struct.BSS1 zeroinitializer

define void @MAIN_() {
  %args = alloca [3 x i8*], align 16
  ; < other IRs>
  %2 = bitcast %struct.BSS1* @.BSS1 to i8*
  %3 = getelementptr i8, i8* %2, i64 32768
  %4 = bitcast [3 x i8*]* %args to i8**
  store i8* %3, i8** %4, align 8
  ; < other IRs>
  %12 = bitcast %struct.BSS1* @.BSS1 to i8*
  %13 = bitcast [3 x i8*]* %args to i8*
  %14 = getelementptr i8, i8* %13, i64 8
  %15 = bitcast i8* %14 to i8**
  store i8* %12, i8** %15, align 8
  ; < other IRs>
  %28 = load float, float* %alpha_305, align 4
  %29 = bitcast [3 x i8*]* %args to i8*
  %30 = getelementptr i8, i8* %29, i64 16
  %31 = bitcast i8* %30 to float*
  store float %28, float* %31, align 4
  ; < other IRs>
  %47 = bitcast [3 x i8*]* %args to i64*
  %50 = call i32 @__tgt_target_teams(..., i64* %47, ...)

  ret void
}
```

Listing 7. Generated host code for PowerPC64le of LLVM. The output LLVM IR is generated from the Fortran in the Listing 4.

transformed into LLVM IR. While doing this, the schedule creates its own structures to build LLVM IR internally. This phase generates a *LL_Module* struct which keeps all the information about the llvm module. Eventually these internal structures are written as an LLVM IR file.

To implement OpenMP GPU offload, we extended the *schedule* phase. First, we make it aware of multiple *LL_Module* structs, where each struct corresponds to a different target. The schedule phase is capable of generating LLVM IR into an *LL_Module*. Eventually, the *assemble* phase turns them into LLVM IR and writes them into different files.

V. Experimental Evaluation

We evaluate Flang OpenMP offload support for NVIDIA GPUs on two benchmark applications. We com-

Compiler	Flags
Flang OpenMP Multicore	-O3 -Mpreprocess -fopenmp
Flang (this work) OpenMP GPU	-O3 -Mpreprocess -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda
NVIDIA PGI	-Mpreprocess -fast -ta=tesla,cc60,cuda8.0 -mp
IBM XLF	-qsmp -qoffload -qpreprocessor -O3
Clang 7.0	-O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda

TABLE I
Compilers studied in our experiments.

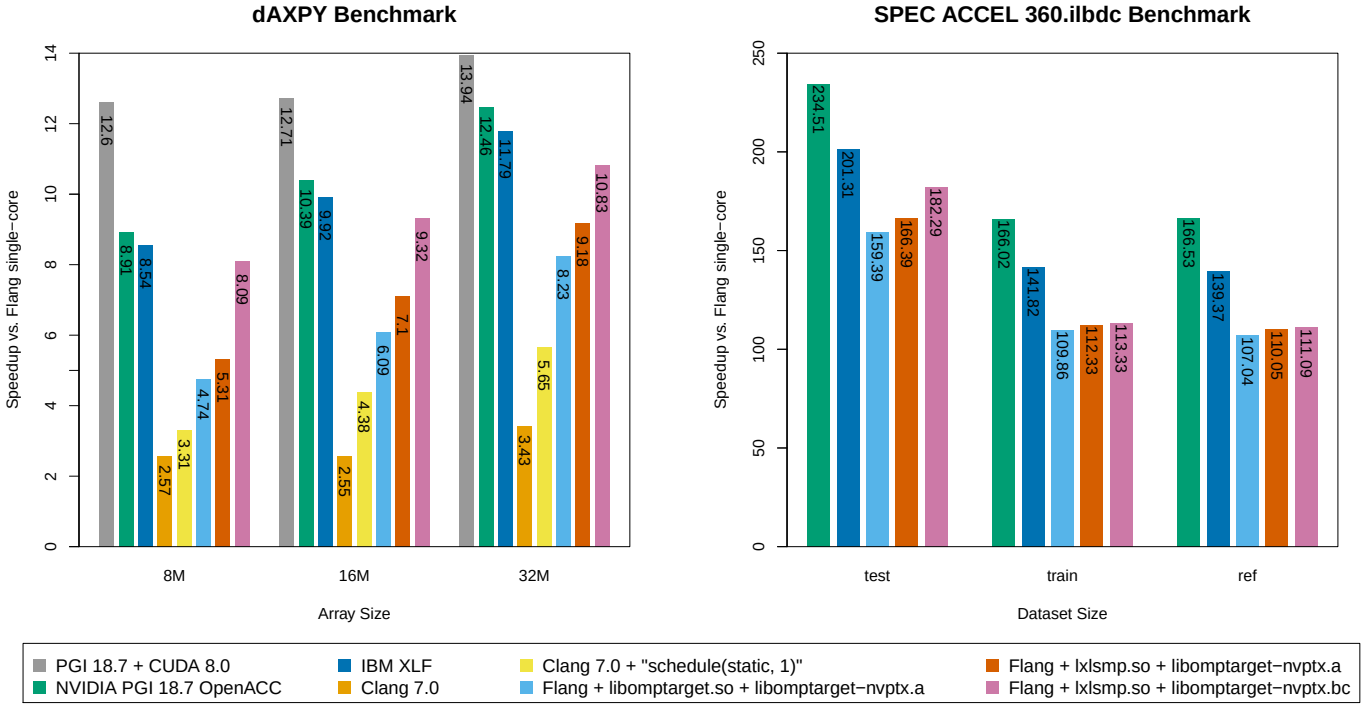

Figure 2. Execution speedup evaluation for the dAXPY application compiled with different compilers, and executed with three different array sizes.

Figure 3. Execution speedup evaluation for the 360.ilbdc application from the SPEC Accel benchmark suite compiled with different compilers, executed with different dataset sizes.

pare Flang's results with those of two other compilers that currently support OpenMP offloading: Clang 7.0 and IBM XL 16.0.1. Additionally, we make a comparison of Flang's results to the OpenACC version of the same benchmarks compiled with NVIDIA PGI 18.7.

We run our experiments on a system with 2 10-core IBM POWER8NVL processors, with 1024GB of memory, running Ubuntu 16.04, with three NVIDIA Pascal GPUs with 3584 CUDA cores, compute capability 6.0, and 12GB of device memory. Each Pascal GPU includes 56 Streaming Multiprocessors (SM), each with 64 CUDA cores, and NVlink [20] technology. We obtain Flang from the NVIDIA internal repository. CUDA 8.0 toolkit is used with all the compilers.

Table I shows the compilers and compilation options we used for the experiments in this paper.

1) AXPY: AXPY is an application whose name stands for A*X plus Y, which is a function from the Basic Linear Algebra Subroutines (BLAS) [21] library. AXPY is a combination of scalar multiplication and vector addition, it takes as input two vectors X and Y with N elements each, and a scalar value A. We use different versions of AXPY. We implement a version of the AXPY application in Fortran and annotate the compute loop with the OpenMP `!$omp target teams distribute parallel do` directive. We also created an OpenACC version using `!$acc parallel loop` from the OpenACC directives. Finally, we implemented a C version which we also parallelized with the same OpenMP directives as the Fortran version.

Figure 2 shows the speed up of AXPY using different compilers over sequential version runs on a CPU. The results show poor performance for the Clang compiler. Our investigation revealed that the Clang compiler schedules the loop with block scheduling, which breaks the coalesced memory access pattern, resulting in poor performance. We added the `schedule(static,1)` clause to the code and the experimental results show the effectiveness of our solution. Even though these experiments used the same runtime libraries for both Clang and Flang, the results yield better performance with the Flang compiler. The reason is that Flang does not apply outlining in the device code, which creates opportunities for the compiler to enable different optimizations. Conversely, Clang extracts a function for each OpenMP construct which might decrease the performance of generated code. We also compare the performance of the Flang compiler using the *libxlsmp* offload runtime library from the IBM XL compiler to *libomptarget*. Since these two runtimes support the same interface, we did not have to modify the code generation. In this case, we observed that IBM's offload runtime performs better than *libomptarget*. We show the performance of NVIDIA PGI and IBM XL compilers, on the OpenACC and OpenMP versions of AXPY, which both have better performance

compared to the LLVM compilers. The best performance, as shown in the plot, are obtained with the native CUDA implementation of the benchmark, that for our experiments we compiled with PGI.

2) SPEC ACCEL - 360.ilbdc: The 360.ilbdc is one of the applications of the SPEC ACCEL [22] benchmark suite; it implements an algorithm to solve problems related to fluid mechanics. The code is written in Fortran and uses a minimal number of OpenACC and OpenMP directives.

Figure 3 shows the speedups of the application built with different compilers compared to the sequential version run on a CPU. We ran the benchmark with three datasets of different sizes called test, train, and ref. Flang shows performance close to the other compilers, demonstrating its capability to take advantage of the massive parallel architecture of GPUs. Again, we followed the strategy of linking Flang-generated code with IBM's offload runtime. This experiment also yields better performance, we suspect the runtime processes data transfers and kernel launches in a more efficient way. Finally, the OpenACC version compiled with the NVIDIA PGI compiler shows the best performance. We believe PGI compiler applies several GPU-specific optimizations.

VI. Conclusions and Future Work

In this paper, we describe our initial implementation of OpenMP GPU offload in Flang targeting NVIDIA GPUs. Our study shows that Flang can efficiently take advantage of the massive parallel architectures of GPUs using OpenMP directives. Our compiler implementation in Flang allowed us to experiment with different code generation strategies. In this work we used the CUDA-like code transformation strategy in an attempt to obtain performance closer to native CUDA. Finally, we tried to emphasize this aspect supported by an experimental evaluation on two benchmark applications. We also compared our implementation with other modern compilers.

A possible next step in Flang would be to provide a complete OpenMP device model. We left supporting uncombined directives as future work, since that implementation requires a different code transformation strategy. More extensions in the programming model itself may be introduced for user-directed optimizations that allow the compiler to use more sophisticated approaches.

References

[1] "OpenMP – Application Program Interface," https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[2] "The OpenACC – Application Programming Interface," https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf.

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[4] C. Lattner, in *FOSDEM*, 2011.

[5] N. PGI, "Flang compiler," https://github.com/flang-compiler, 2017.

[6] N. Corp., "NVVM IR Specification," https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.

[7] "Clang: a C language family frontend for LLVM," http://clang.llvm.org.

[8] "OpenMP Runtime Library," https://openmp.llvm.org.

[9] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for OpenMP in Clang and LLVM," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '16, 2016, pp. 1–11.

[10] "CUDA dynamic parallelism API and principles," https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/.

[11] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, 2014, pp. 51–60.

[12] S. Lee and J. S. Vetter, "OpenARC: extensible OpenACC compiler framework for directive-based accelerator programming study," in *Proceedings of the First Workshop on Accelerator Programming using Directives, WACCPD '14, New Orleans, Louisiana, USA, November 16-21, 2014*, 2014, pp. 1–11.

[13] V. G. V. Larrea, W. R. Elwasif, O. Hernandez, C. Philippidis, and R. Allen, "An in depth evaluation of GCC's OpenACC implementation on cray systems," 2017.

[14] G. Ozen, E. Ayguadé, and J. Labarta, "On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP," in *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, 2014, pp. 215–229.

[15] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. M. Chapman, "Compiling a high-level directive-based programming model for gpgpus," in *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*, 2013, pp. 105–120.

[16] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O. Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU threads for OpenMP 4.0 in LLVM," in *2014 LLVM Compiler Infrastructure in HPC*, Nov 2014, pp. 12–21.

[17] A. C. Jacob, A. E. Eichenberger, H. Sung, S. F. Antao, G. Bercea, C. Bertolli, A. Bataev, T. Jin, T. Chen, Z. Sura, G. Rokos, and K. O'Brien, "Efficient fork-join on GPUs through warp specialization," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, Dec 2017, pp. 358–367.

[18] L. Grinberg, C. Bertolli, and R. Haque, "Hands on with openmp4.5 and unified memory: Developing applications for ibm's hybrid cpu+gpu systems (part i)," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds., 2017, pp. 3–16.

[19] ——, "Hands on with openmp4.5 and unified memory: Developing applications for ibm's hybrid cpu+gpu systems (part ii)," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds., 2017, pp. 17–29.

[20] "NVlink," https://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink.

[21] "Basic linear Algebra Subprograms (BLAS)," in *Encyclopedia of Parallel Computing*, 2011, p. 120.

[22] G. Juckeland, W. C. Brantley, S. Chandrasekaran, B. M. Chapman, S. Che, M. E. Colgrove, H. Feng, A. Grund, R. Henschel, W. W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. A. Stratton, A. Titov, K. Wang, G. M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran, "SPEC ACCEL: A standard application suite for measuring hardware accelerator performance," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, 2014, pp. 46–67.