Direct GPU Compilation and Execution for Host Applications with OpenMP Parallelism

Shilei Tian
Department of Computer Science
Stony Brook University
Stony Brook, USA
shilei.tian@stonybrook.edu

Joseph Huber
Advanced Micro Devices
Austin, USA
Joseph.Huber@amd.com

Konstantinos Parasyris
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, USA
parasyris1@llnl.gov

Barbara Chapman
Department of Computer Science
Stony Brook University
Stony Brook, USA
barbara.chapman@stonybrook.edu

Johannes Doefert
Mathematics and Computer Science
Argonne National Laboratory
Lemont, USA
jdoerfert@anl.gov

***Abstract*—Currently, offloading to accelerators requires users to identify which regions are to be executed on the device, what memory needs to be transferred, and how synchronization is to be resolved. On top of these manual tasks, many standard (C/C++ library) functions, such as file I/O or memory manipulation, cannot be directly executed on the device and need to be worked around by the user explicitly. This makes it challenging to port programs in the first place and hinders developers from testing features on the GPU and within the GPU compilation pipeline. Existing tests and test suites for the host are effectively unusable for accelerators and need to be manually ported to provide the same benefits for the devices as they do on the host.**

In this paper, we propose a direct GPU compilation scheme that leverages the portable target offloading interface provided by LLVM/OpenMP. Utilizing this infrastructure allows us to compile an existing host application for the GPU and execute it there with only a minimal wrapper layer for the user code, command line arguments, and a compiler provided GPU implementation of C/C++ standard library functions. The C/C++ library functions are partially implemented for direct device execution and otherwise fallback to remote procedure call (RPC) to call host functions transparently. Our proposed prototype will allow users to quickly compile for, and test on, the GPU without explicitly handling kernel launches, data mapping, or host-device synchronization. We evaluate our implementation using three proxy applications with host OpenMP parallelism and three microbenchmarks to test the correctness of our prototype GPU compilation.

***Index Terms*—GPU, OpenMP, compilation, portability**

I. Introduction

Compiling and executing user code on the GPU has long been supported by kernel languages like CUDA, OpenCL, or HIP. However, these kernel languages require the user to explicitly outline which regions are to be compiled for and executed on the device. This does not only slow down the (compiler) developers' ability to test offloading but also users that want to determine what features, e.g., atomic operations, are properly supported by a compiler toolchain and the GPU hardware. Any test case needs to be written with explicit offload directives and data mapping in mind. As a result, existing host code, including test suites for compilers and (portable) parallel programming languages, e.g., OpenMP, are effectively unusable in the GPU world.

In this paper, we decided to take a step back to avoid the need for dedicated programming of host and device by the user. Instead of this multi-device view, our compiler toolchain will put the GPU (or any other supported accelerator) at the center of execution. Existing single-device programs are directly compiled for, and executed on, the GPU without user interaction or a specialized compiler. A modern LLVM/Clang and a few command line flags that pull in wrappers are all it needs for an OpenMP parallel host application to be executed "completely" on the GPU, including parallelism. This approach removes the need for a costly port to test features on the GPU or the GPU compilation pipeline itself. Any existing host program can simply be run on the GPU, assuming all dependencies are met. Users can easily determine what functionality is available, e.g., what OpenMP constructs work and what atomic operations are supported, by simply reusing the existing host test suites for those. Further, compiler developers can finally utilize the enormous amount of host code to identify problems in the GPU compilation toolchain and later boost confidence in the currently "comparatively sparsely tested" backends.

Summarized, we propose to use LLVM/OpenMP offloading to compile the entire host and potentially parallel application for the GPU device and execute it there. To achieve this, we implemented a user wrapper that calls the user's `main` function from a GPU kernel and further ensures all user code is compiled for the GPU. We also provide a new, but partial, standard C library (`libc`) for the device which is naturally present on the host. We reuse the math library (`libm`) for the device introduced by [1]. For system routines requiring information from the host, we implemented a remote procedure call (RPC) framework to periodically communicate with a running host thread during the execution of the user

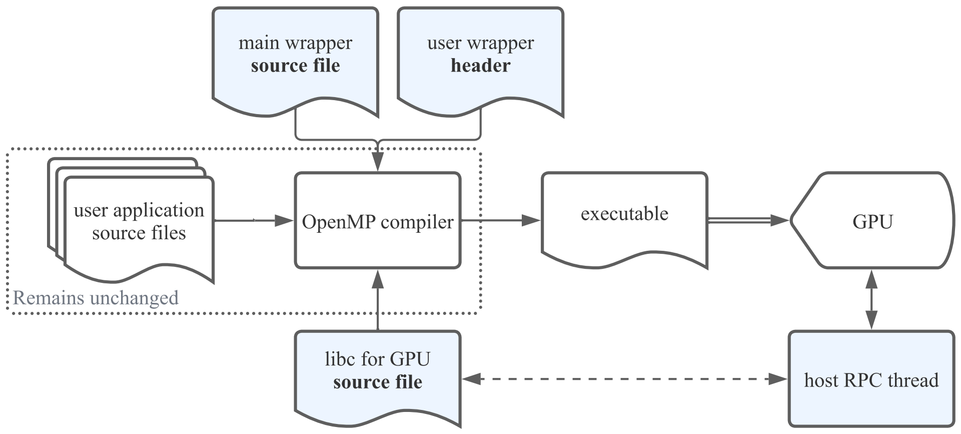

Fig. 1. An overview of our direct-GPU compilation pipeline. User code and compiler are unchanged (dotted rectangle). The parts with blue background (entire first and third row) are provided by this work. The source files for the "main wrapper" and the standard library `libc` are compiled as GPU-only code (via `-fopenmp-offload-mandatory` in addition to OpenMP device offlad flags). The "user wrapper" header file is pre-included into every user source file via `-include`. The final executable running on the GPU will communicate with the host RPC thread in case library functions cannot be reimplemented on the device and need host support.

program on the device. Fig. 1 shows an overview of our proposed framework. The blue parts (entire top row and bottom row) are added to allow direct GPU compilation. They are coupled with the user program through minimal changes in the users' compilation instructions, e.g., Makefile, which adds the necessary compilation flags. The host RPC thread is executing the host part of the standard library (`libc`) provided for the GPU as not all functionality can be executed natively on the device, e.g., system calls need host support. Note that our approach does neither require user code modifications nor changes to the functionality already present in a modern LLVM/Clang compiler (e.g., LLVM 15 and later).

This paper makes the following contributions:

- A novel compilation path that can compile an OpenMP parallel host program to make it "completely" run on a GPU without any change in the source code.
- A remote procedure call (RPC) implementation to transparently utilize the host from the device in case system calls or other unavailable features are needed during kernel execution.
- A LLVM/OpenMP-based prototype implementation that currently supports Nvidia GPUs but is not conceptually bound to a specific device. All parts but the RPC implementation are already portable.
- Thorough evaluations of our implementation using OpenMP proxy applications and micro-benchmarks that cover the most common patterns in HPC applications.

The rest of the paper is organized as follows. We first introduce some background of OpenMP target offloading in Section II. Next, we talk about our prototype implementation on top of LLVM/OpenMP in Section III, followed by the discussion of the evaluation results in Section IV. Related work is compared in Section V before we conclude the paper in Section VI and discuss future work, especially potential solutions to identified limitations.

II. Background

OpenMP 4.0 introduced the `target` construct, indicating that the enclosed region can be executed on a target device, such as a GPU or FPGA. This section briefly introduces the compilation flow, code representation, and execution model used by LLVM/OpenMP.

A. Compilation and Device Code Representation

LLVM/OpenMP offloading features a two-pass compilation by first performing host code compilation and then device code compilation. Host code compilation includes the regular compilation of code for the host and then generates calls to the LLVM/OpenMP offloading runtime in order to invoke kernels defined in the next stage. Device code compilation emits target-dependent device code for offloaded regions, as well as required functions and variables.

In addition to the `target` construct (as well as its combined variants), OpenMP provides the `declare target` directive which specifies that all associated variables and functions are to be mapped onto the target devices and thus are usable in device code [2]. All code associated with a `target` construct, the so-called target region, will be outlined, and a kernel will be generated for it such that it can be invoked from the host. Fig. 2 and 3 show an example of CUDA code and its corresponding OpenMP offload version. The `device_type(nohost)` clause on a `declare target` construct forces the compiler to not generate host versions of the enclosed variables and functions. Similarly, LLVM/Clang offers the command line option `-fopenmp-offload-mandatory` as an extension to prevent the compiler from generating host fallback code for target regions[1]. Together, this allows us to emit code only for the GPU without them being compiled for the host.

[1]Based on the OpenMP specification, a host version of a `target` region has to be generated and will be executed if offloading fails at runtime. The command line option effectively guarantees the compiler that an offload failure will end the program; therefore, a host fallback version is not required.

```
__device__ int g;
__device__ void foo();

__global__ void baz() { foo(); }

void bar() {
  baz<<<...>>>();
}
```

Fig. 2. An example of CUDA code. The function `baz` is a *kernel* that is the entry point of a GPU program and can be launched from host. The function `foo` is a device function that can be called in a kernel.

```
#pragma omp begin declare target device_type(nohost)
int g;
void foo();
#pragma omp end declare target

void bar() {
// The following region will be outlined to a new
// function, and will be launched from the host,
// similar to the function `baz` in the CUDA example.
#pragma omp target
  { foo(); }
}
```

Fig. 3. Equivalent OpenMP code to Fig. 2.

B. Execution Model

An OpenMP program begins as a single, or *initial thread*, executing sequentially. When any thread encounters a `parallel` construct, the thread creates a new team of itself with zero or more additional threads, each of which executes the code associated with the `parallel` construct.

```
int main(int argc, char **argv) {
  /* region 1 */
#pragma omp parallel
  { /* region 2 */ }
  /* region 3 */
}
```

Fig. 4. An example of a host OpenMP program.

Fig. 4 shows an example of a program with a `parallel` construct. Region 1 is executed by the initial thread. When the `parallel` construct is encountered, the initial thread forks a team of threads, each of which executes region 2. After region 2 is executed by all threads in the team, only the initial thread continues the sequential execution of region 3.

Similar to the host model described above, when a `target` region executes, it is executed by the initial thread sequentially. However, a `teams` construct is usually used to create a league of teams together with the `target` region such that each *team* starts execution with one initial thread independently. When a `parallel` construct is encountered, the enclosed region will, as above, be executed by the encountering thread and the (new) threads that are part of the respective team.

```
#pragma omp target teams num_teams(N)
  {
    /* region 1 */
#pragma omp parallel
    { /* region 2 */ }
    /* region 3 */
  }
```

Fig. 5. OpenMP target offloading program excerpt.

For the example in Fig. 5, when the target region starts execution, N teams will be created. Only one thread, the initial thread, in each team executes region 1 independently. When those threads come to the `parallel` construct, all threads in each team will execute region 2. After the execution of region 2, only the N initial threads will execute region 3.

III. Design and Implementation

We envisioned a toolchain that allows seamless execution of host codes on device accelerators without any code modifications. Thus, our approach is by design minimally invasive.

Traditionally, for any code to run on any architecture the code needs to be represented in the target architecture's instruction set. Moreover, the executable in a conventional architecture is loaded for execution by the operating system, At execution time the operating system provides support through system calls, e.g., to access the file system. These system calls are abstracted by the *standard C library*. For example, there is functionality to write and read files, send packets through the network, and allocate memory.

Intuitively, our approach needs to provide the same high-level functionality, summarized as: 1) The application source code needs to be translated into the underlying GPU instruction set architecture. 2) The application executable needs to be loaded to the device and execution must start there together with the command line options passed by the user being available. 3) The *standard C library* (`libc`) needs to be available on the device to provide common functions.

A. Source code translation to device instruction set

In order to make the program execute completely on the GPU, we need to ensure that all user code is compiled for the device, not the host. As mentioned in Section II-A, the `declare target` directive allows us to mark which code and global symbols should be present on the target device. To ensure all user code is associated with such a directive we effectively prepend a `begin declare target` before any user source file. Instead of requiring the user to add these annotations manually, we provide a wrapper header (presented in Fig. 6) and automatically include the wrapper header using Clang's `-include` command line option when compiling user code. Specifically, the user is required to extend the build system and pass the following flag `-include UserWrapper.h` as part of the compilation command, e.g., at the beginning of the `CFLAGS`. To ensure we avoid declaration conflicts with the host we use the `device_type(nohost)` clause to only emit the user code for the target device. The secondary use of the wrapper header is to rename the original `main` function of the program into `__user_main`. This allows us to provide our own main function while making sure we can call the user-provided one from our target region.

```
#pragma omp begin declare target device_type(nohost)

int main(int, char *[]) asm("__user_main");
```

Fig. 6. User wrapper header (`UserWrapper.h` in the text).

B. Loading and invoking the device execution

Typically, the startup of a process requires, among other steps, loading the application executable to host memory and initializing the statics object to be constructed and loaded into host memory. Finally, once the setup is done, execution control is given to the `main` function. Thus, the `main` function is the starting point of a traditional host program.

Similarly, in the typical device execution scheme, the host code launches explicitly a kernel and that kernel is one of the entry points of the GPU execution. Fig. 2 illustrates the syntax of launching a device kernel.

Respectively, in direct GPU compilation, we need to initialize all static objects in the device and then start execution by invoking the original `main` on the device. The LLVM/OpenMP implementation already allocates and initializes static device objects upon device initialization. Since the entire user code is considered device code, OpenMP will initialize all the required variables automatically. However, we still need to invoke the original `main` function on the device. To simplify this, we already renamed the user `main` function to `__user_main` using the assembly shown in Fig. 6.

To transfer control to the user main function we provide the host main function illustrated in Fig. 7. At first, all program arguments are mapped to the device such that the user's `main` function can access them. Then it launches the user's `main` function `__user_main` by calling it from a `target` region. The return value of the host program is taken from the `__user_main` function. This new host entry point is implemented in `Main.c` and this file needs to be compiled by the user and linked into the executable together with all other user source files.

```c
#include <string.h>

extern int __user_main(int, char *[]);

int main(int argc, char *argv[]) {
  #pragma omp target enter data map(to: argv[:argc])

  for (int I = 0; I < argc; ++I) {
    size_t Len = strlen(argv[I]);
    #pragma omp target enter data map(to: argv[I][:Len])
  }

  int Ret;
  #pragma omp target teams num_teams(1)     \
        thread_limit(1024) map(from: Ret)
  { Ret = __user_main(argc, argv); }

  return Ret;
}
```

Fig. 7. The file contains actual `main` function (`Main.c` in the text).

C. Standard C library support

Typically, applications interact with memory allocators, files, and other system parts through the omnipresent *Standard C Library* (`libc`). For our benchmarks, `libc` provides: 1) memory-related functionality, e.g., `malloc` and `free`; 2) utilities, including `stcmp`, `atof`, `atoi`, and `memcpy`; 3) I/O access via `fread`, `printf`, and similar functions. However, our application will not have direct access to the host runtime as it will be executed on the device. Therefore we provide a separate implementation of a subset of the standard C library as part of this work. Our implementation handles the aforementioned categories differently.

*1) **Memory allocation and deallocation**:* Dynamic memory allocation is widely used in a host program through `malloc` or the `new` operator in C++. Their support for GPUs varies widely between vendors. Nvidia, for example, has built-in `malloc` and `free` functions that can be called from the device. However, it is generally not recommended because of the poor performance as well as the limited heap size. AMD supports dynamic allocation in HIP, but currently not when using LLVM/OpenMP. In order to work around the variance and limitation among vendors' support, we instead implemented dynamic heap memory allocation using a pre-allocated memory pool and a simple bump allocator for the purpose of this prototype.

*2) **Utility functions**:* These functions are implemented in a device library that is linked with the application executable. Thus, the application can directly execute the library code inside the device, and through the link-time-optimizations (LTO) [3] overheads can be minimized.

*3) **I/O routines**:* The device is unable to access hardware that is memory mapped to the operating system, for example, hard drives and networks. In order to transparently support functions that rely on such access, we implement a host remote procedure call (RPC) scheme which coordinates the communication between the host and GPU whenever necessary. It features a synchronous, stateless client-server protocol, where the GPU (client) sends requests to the host (server) and waits for the host to acknowledge the completion. Fig. 8 illustrates how RPC communication between the host and GPU works.

All functions that require the delegation to host share the same basic pattern. We will use `fopen` as an example to discuss the communication between the device (as shown in Fig. 9) and host (as shown in Fig. 10) in more detail.

In the device implementation, a wrapper object that manages the lifetime of allocated resources encapsulates the main data structure, a `HostRPCDescriptor`. This descriptor stores all the necessary information that will be shared with the RPC server, such as the function call ID, number of arguments, and the value and type of each argument.

Arguments are added to the descriptor by calling `Wrapper.addArg(...)`. If an argument is a scalar or a pointer whose pointed memory buffer does not need to be accessed from the host, it will be copied into the descriptor directly. Otherwise, such as the two pointer arguments `filename` and `mode` in this case, two extra steps are required: 1) allocate new storage for the pointed-to memory (recursively if necessary), and 2) copy the memory over into the newly allocated one. This copy is in general necessary because we do not know if the original memory pointed by a pointer is accessible from the host[2]. For now, we need to conservatively

[2]For instance, on Nvidia GPUs, the memory buffer allocated from `malloc` on the device can not be copied to host. Similarly, stack memory of a thread is not host accessible.

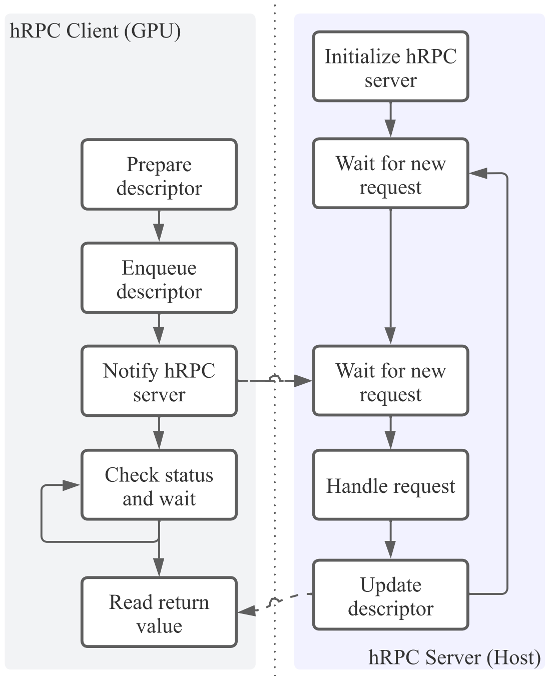

Fig. 8. An overview of the RPC state machine.

```
FILE *fopen(const char *filename, const char *mode) {
  HostRPCDescriptorWrapper Wrapper(ID_fopen, 2);
  if (!Wrapper.isValid())
    return nullptr;

  auto Len1 = strlen(filename) + 1;
  auto Len2 = strlen(mode) + 1;

  HostRPCObject<const char *> FileName(Len1);
  HostRPCObject<const char *> Mode(Len2);

  FileName.copyFrom((void *)filename, Len1);
  Mode.copyFrom((void *)mode, Len2);

  Wrapper.addArg(FileName.get(), ARG_POINTER, Len1);
  Wrapper.addArg(Mode.get(), ARG_POINTER, Len2);

  if (!Wrapper.sendAndWait())
    return nullptr;
  return Wrapper.getReturnValue<FILE *>();
}
```

Fig. 9. Implementation of `fopen` on the device.

assume all memory is non-accessible and perform the copy. Future work should utilize compiler analyses to alleviate this copy in the common case of heap-allocated memory. Similarly, a runtime check could be employed in the future.

After everything is set up, the RPC request is issued by the device thread which then waits for the result from the host. All this is encapsulated in the `Wrapper.sendAndWait()` method, which essentially works as follows. It first notifies the host that there is a new RPC request, and then actively checks the status of the request. If the request has not been fulfilled yet, it sleeps for a few nanoseconds using the exponential back-off algorithm and then checks again.

For now, our prototype only supports one request at a time. We expect future work to extend this and to support multiple requests, e.g., by using a queue.

```
bool handle_fopen(HostRPCDescriptor &SD) {
  ArgumentExtractor AE(SD);

  auto *FileName = AE.getArg<const char *>(0);
  auto *Mode = AE.getArg<const char *>(1);

  FILE *F = fopen(FileName, Mode);
  if (F == nullptr)
    return false;

  SD.ReturnValue = (void *)F;
  return true;
}
```

Fig. 10. Implementation of `fopen` on the host.

On the host side, once a new request is received, it is dispatched to the corresponding handler (in this case `handle_fopen`). The `ArgumentExtractor` encapsulates the logic to prepare arguments for the host runtime call. This includes moving memory buffers from the device to the host if unified shared memory is not supported. Finally, the corresponding host runtime function is called. Values and memory that need to be transferred to the device are now moved and the return value of the host call is saved. Note that the `FILE *` in this example is not mapped to the device but is taken as a literal value to be exclusively used on the host. The RPC server will update the status of the request on the device side and wait for the next request. As soon as the status was updated the RPC client on the device will continue execution.

D. *Limitations*

Arbitrary Library Functions: Our prototype implementation does not support function calls to an arbitrary library because delegation support needs to be customized. As with missing `libc` functionality, arbitrary library functions can be added as needed.

Variadic Functions: Variadic functions, such as `printf` and `fscanf`, are widely used in real-world applications. However, we do not support the ability to extract the variadic arguments through `va_start`, `va_arg`, etc., on the GPU. For now, LLVM/OpenMP does support `printf` for Nvidia GPUs using a workaround. The compiler first packs up all variadic arguments into a struct, replaces the function call to `printf` with `__llvm_omp_printf`, and passes the pointer to the struct as an argument. `__llvm_omp_printf` is implemented in the device runtime library [4]. It calls `vprintf`, which is an external function provided by the Nvidia device library. For AMD GPUs, this method does not work as it does not have its counterpart of `vprintf`. Furthermore, we can not simply apply the same method to other variadic functions either, even though we have RPC, because the host version of `vprintf` does not accept a pointer to a buffer of all packed arguments, as what Nvidia provides for `vprintf`. Once (partial) variadic argument support for GPUs is implemented in LLVM/Clang we can support these functions as we do any others already.

Single Team Execution: Based on the execution model of OpenMP target offloading, as introduced in Section II-B, the body of a `target teams` construct will be executed by the initial thread of each team. If there are N teams, the body will be executed by all the N initial threads. However, most part of the user code (except those OpenMP `parallel` regions) should only run single-threaded. As a result, we can only use one team, as the `num_teams(1)` shown in Fig. 7. In LLVM OpenMP, an OpenMP team is mapped into a thread block (or wavefront for AMD GPU). There is usually a limited number of threads that can be used in one thread block, such as 1024 for Nvidia GPUs. Therefore, we can only use at most that total number of threads, even for the `parallel` region, which limits the usage of GPUs. Mapping a parallel region onto multiple teams is not in general legal but could be done with compiler support. We also expect *ensemble* use cases that run the application many times on different inputs, e.g., random seeds, to be able to utilize the entire device rather than a single team/thread block/wavefront at a time.

Variable-Length Array: Variable-length array (VLA) is currently not supported on GPUs. As a consequence, a compiler (front end) error will be emitted if a VLA is used. Future work could enable support if the use case is important enough. However, VLAs on the GPU would potentially need to utilize dynamic memory rather than static memory.

E. Putting Everything Together

Our approach is transparent to the developer, as they do not need to modify the initial application code. In essence, the user is required to only modify the build configuration, e.g., the description passed to `make` or `cmake`. Figures 11 and 12 show the usage of LLVM/OpenMP with our proposed method. The required steps are: 1) Enable standard OpenMP offloading using `-fopenmp --offload-arch=<arch>`. 2) Include the wrapper header using `-include UserWrapper.h` to map all code to the device. 3) Compile and link `Main.c` which invokes the original `main` on the device.

```
$ clang -c <user source files>           \
        -fopenmp --offload-arch=<arch>    \
        -include <path to>/UserWrapper.h
```

Fig. 11. Compile the user code with LLVM/OpenMP offload flags and include the user wrapper header file.

```
$ clang <path to>/Main.c -c -o __Main.o   \
        -fopenmp --offload-arch=<arch>    \
        -fopenmp-offload-mandatory
$ clang -fopenmp --offload-arch=<arch>    \
        __Main.o <other object files>     \
        -o <exec name>
```

Fig. 12. Compile the provided host main function that offloads the user code application to the device and link it together with the other object files to generate the executable.

IV. Evaluation

A. System Configuration

For our performance evaluation, we used an Nvidia A100 GPU system with an AMD EPYC 7532 CPU (32C/64T) and 256 GB DDR4 RAM. We used CUDA 11.4.0 for all experiments. Our prototype version (ℵ) is based on git `0a8dd8ef`.

B. Benchmarks

We used three proxy applications and three microbenchmarks from HeCBench for the evaluation. Each benchmark has two versions: host OpenMP and OpenMP target offloading.

XSBench and RSBench are two proxy applications for the Open Monte Carlo (OpenMC) project. Both proxies compute the continuous energy macroscopic neutron cross-section lookup when studying neutron transport, and both are available in multiple programming languages and frameworks. While XSBench [5] extracts one of the main kernels in OpenMC, which is memory-bound, RSBench [6] provides a compute-bound alternative implementation.

miniBUDE is an implementation of the core computation of the Bristol University Docking Engine (BUDE) in different HPC programming models [7]. The benchmark is a virtual screening run of the NDM-1 protein and runs the energy evaluation for a single generation of poses repeatedly for a configurable number of iterations. The execution of the benchmark requires reading data from a file.

HeCBench is a GitHub repository that contains a collection of Heterogeneous Computing benchmarks written with CUDA, HIP, SYCL (DPC++), and OpenMP 4.5 target offloading for studying performance, portability, and productivity [8]. We used three benchmarks out of it: hotspot3D, page-rank, and amgmk, and ported them to the host OpenMP version by removing all `target` related code.

Source code modifications: The objective of our work is to allow direct GPU compilation without any source code modifications. However, due to the limitations that we discussed in Section III-D we modified some of the benchmarks to apply our approach. Specifically, we performed the following modifications in some of the benchmarks:

- Removed all `schedule` clauses from `parallel` directives because they are currently not supported by LLVM OpenMP when being used in a `target` region. Note that it is unrelated to our work.
- Removed or rewrote the code using features that are not supported by our framework due to the limitation mentioned in Section III-D, such as variadic functions.

C. Results and Analysis

Fig. 13 shows the slowdown of the host OpenMP code compiled with our work directly to the GPU (version GPU-D) over the host OpenMP version (version HST) and the existing OpenMP target offloading version (version TGT). Both GPU-D and TGT versions were executed on the Nvidia A100 GPU, and the HST version was executed on the AMD CPU. Fig. 14 shows the resource usages of TGT and GPU-D version as reported by `nsys`. All the `libc` functions supported by our prototype are listed in Fig. 15.

The experiment results demonstrate that our prototype implementation can successfully compile a host program to make it completely run on a GPU, especially for miniBUDE that uses many `libc` functions, such as `fopen`, `fread` to read file data, and `strcmp`, `strtoul` to manipulate strings. The performance degradation is also expected because of reasons

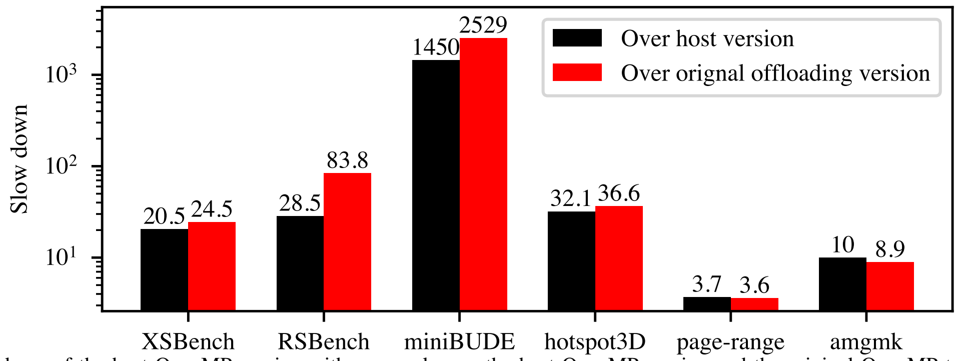

Fig. 13. Slowdown of the host OpenMP version with our work over the host OpenMP version and the original OpenMP target offloading version.

	Host version with our work			Original target offloading version			
	# regs	# threads	StcSMem	% comp.	# regs	# threads	StcSMem
XSBench	255	256	14,480	8.96%	174	17000064	9,772
RSBench	250	256	12,412	76.32%	254	10200064	9,772
miniBUDE	193	256	10,440	23.81%	255	16384	9,772
hotspot3D	170	256	10,096	1.18%	142	262144	9,772
page-rank	122	512	9,912	0.40%	42/72/32	10240	9,772
amgmk	255	256	9,968	1.71%	86	125184	9,772

Fig. 14. Comparison of the resource usage between the host version compiled with our work and the original target offloading version. "% comp." stands for the percentage of overall runtime spent on the actual computation, aka. kernel execution. It indicates how much serial work is done on the host to prepare for the computation task on the GPU. The number of threads is calculated by grid_size × block_size, where grid_size is always 1 for our work. Three numbers appear in the "# regs" for page-rank because it has three `target` regions.

as follows. First, the "% comp." column in Fig. 14 shows the amount of time the benchmark spends in actual "computation" rather than data initialization. Since most benchmarks spent a large portion of time running sequential initialization tasks, e.g., 91% for XSBench, and the performance of one GPU thread is much worse than one CPU thread, this code is executed much slower with GPU-D. Although version GPU-D does not need data transfer between host and GPU for data mapping, the performance gain can not compensate for the loss in serial execution for the given programs and inputs. For miniBUDE, there are four calls to `fread` in the source code, and each call only reads a small number of bytes (either 6 or 16 bytes) from the file. That leads to a large number of function calls to `fread` at runtime, which means a large number of RPC calls between host and GPU, causing large overheads. Next, as we mentioned in Section III-D, we can only use *up to* 1024 threads organized in a single team (aka. thread block). In fact, as shown in Fig. 14 none of the benchmarks can use 1024 threads because the actual number is capped by the function call to query the max number of threads per block that the kernel can use on the target device. It depends on both the function (e.g., the resource usage of the kernel) and the device on which the function is currently loaded. That limits the performance of benchmarks such as RSBench with heavy computation workload as the native GPU application will utilize the entire GPU with 10200064 threads in total. If the number of threads is similar, e.g., for the page-rank benchmark, the performance degradation is reduced significantly. While the original GPU version of this benchmark uses 20× more threads (=parallelism), it is only 3.6× faster.

These results allow us to predict four paths that could deliver competitive performance without manual porting efforts:

1) Run multiple application instances in parallel to utilize the various thread blocks, one per instance. Initial experiments show promising results, but resources like memory can be limiting.
2) Execute parallel regions via multiple teams. This requires compiler analysis to determine correctness, potentially combined with transformations and additional code to combine results across teams, e.g., for reductions.
3) Use compiler transformation to split off multi-threaded code into separate kernels with reduced resource (mostly register) usage. As a consequence, more threads might be available to participate in the parallel regions.
4) "Reverse offload" single-threaded code to the host to utilize fast host threads. This is simpler in a unified shared memory (USM) environment but requires automatic memory movement in non-USM systems.

V. Related Works

A. OpenMP Target Offloading

OpenMP 4.0 introduced target offloading. The OpenMP offloading support for GPUs in LLVM can be traced back to the two works presented by [9], [10]. The (PGI) Fortran front-end, known as Flang, supports OpenMP offloading via the LLVM OpenMP runtime [11]. Since then, researchers

fseek	fflush	fclose	time	ftell
rewind	free	gettimeofday	fread	gmtime
fwrite	getc	fopen	feof	fgets
pclose	strftime	popen		
memcmp	printf	malloc	strchr	strncmp
memcpy	strcpy	stat	srand	rand
strtoul	atof	strncpy	strcat	abs

Fig. 15. A full list of `libc` functions that our prototype supports. The upper group requires support from the host via host RPC, while the lower group can run directly on the GPU.

have been working on compiler and runtime optimization for LLVM OpenMP. [12] introduced the first front-end-based optimizations for Nvidia GPUs that can avoid idle threads and reduce register usage. [13] presented the TRegion interface which delays the discovery of SPMD regions to compiler middle end, contrary to the front-end based approach used before, which can support more kernels to execute in SPMD mode. [14] introduced the runtime support for concurrent execution of OpenMP target tasks. [15] presented OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. [4] presented a co-design methodology for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases.

B. Host Program on GPUs

Several works have explored the support for executing a host program on GPUs. [16] proposed making the host's file system directly accessible to GPU code. They also implemented an RPC protocol to coordinate data transfers between the CPU and GPU. [17] introduced a parallelization framework that can detect parallelism and generate target code for both X86 CPUs and Nvidia GPUs. To support those function calls that can not be natively executed on GPU, they replaced the function call in LLVM with an interface that eventually the host will execute the requested function using foreign function interface. In contrast, our work does not need a specialized compiler and can be directly used with vanilla LLVM. [18] studied transparently accelerated binary applications with novel heterogeneous computing resources without requiring any manual porting or developer-provided hints.

VI. Conclusion and Future Work

In this paper, we proposed a direct GPU compilation framework using existing LLVM/OpenMP support. We showed how our novel generic framework can be used to make an existing host program execute directly on a GPU without changing the existing source code. Our wrapper device `libc` allows us to execute code that could originally not be compiled on the GPU, even when it requires features that only work on the host, such as file processing. The evaluation results prove the correctness of our implementation and the ability to run full applications on the GPU with minimal changes. Our prototype will allow programs to be tested easily on the GPU, as well as testing the existing GPU backend on less common use cases.

In the future, we would like to refine our approach to make the process more automatic without the need for explicitly passing wrapper libraries. Additionally, we believe this could be used to achieve reasonable performance given additional optimizations as mentioned in Section IV-C. This would allow users to easily test their host applications on the GPU without needing to change any user code.

Acknowledgement

The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This work was partially prepared by LLNL under Contract `DE-AC52-07NA27344` and was partially supported by the LLNL-LDRD Program under Project No. 21-ERD-018.

References

[1] J. Doerfert, M. Jasper, J. Huber, K. Abdelaal, G. Georgakoudis, T. Scogland, and K. Parasyris, "Breaking the Vendor Lock — Performance Portable Programming Through OpenMP as Target Independent Runtime Layer," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022, (to appear).

[2] S. Tian, J. Chesterfield, J. Doerfert, and B. Chapman, "Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1," in *International Workshop on OpenMP*, Bristol, UK, 2021.

[3] S. Tian, J. Huber, K. Dinel, B. M. Chapman, and J. Doerfert, "Just-in-Time Compilation and Link Time Optimization for OpenMP Target Offloading," in *International Workshop on OpenMP (IWOMP)*, 2022, pp. 1–14.

[4] J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. Chapman, and G. Georgakoudis, "Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution," in *36th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, St. Petersburg, FL USA, May 15-19, 2023*. IEEE, 2022.

[5] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR*, 2014.

[6] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations," in *Solving Software Challenges for Exascale - International Conference on Exascale Applications and Software, EASC 2014, Stockholm, Sweden, April 2-3, 2014, Revised Selected Papers*, vol. 8759, 2014, pp. 39–56.

[7] A. Poenaru, W. Lin, and S. McIntosh-Smith, "A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application," in *High Performance Computing - International Conference (ISC)*, 2021, pp. 332–350.

[8] Z. Jin. [Online]. Available: https://github.com/zjin-lcf/HeCBench

[9] C. Bertolli, S. Antão, 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 *LLVM Compiler Infrastructure in HPC, LLVM-HPC*, 2014, pp. 12–21.

[10] C. Bertolli, S. Antão, G. 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 *Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC*, 2015, pp. 5:1–5:11.

[11] G. Özen, S. Atzeni, M. Wolfe, A. Southwell, and G. Klimowicz, "OpenMP GPU Offload in Flang and LLVM," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2018, pp. 1–9.

[12] S. F. Antão, A. Bataev, A. C. Jacob, G. 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 *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2016, pp. 1–11.

[13] J. Doerfert, J. M. M. Diaz, and H. Finkel, "The TRegion Interface and Compiler Optimizations for OpenMP Target Regions," in *International Workshop on OpenMP (IWOMP)*, vol. 11718, 2019, pp. 153–167.

[14] S. Tian, J. Doerfert, and B. M. Chapman, "Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads," in *Languages and Compilers for Parallel Computing (LCPC)*, 2020, pp. 41–56.

[15] J. Huber, M. Cornelius, G. Georgakoudis, S. Tian, J. M. M. Diaz, K. Dinel, B. M. Chapman, and J. Doerfert, "Efficient Execution of OpenMP on GPUs," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 41–52.

[16] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: integrating a file system with gpus," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 485–498.

[17] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergstrom, "KernelGen - The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs," in *International Parallel & Distributed Processing Symposium Workshops*, 2014, pp. 1011–1020.

[18] M. Damschen, H. Riebler, G. Vaz, and C. Plessl, "Transparent offloading of computational hotspots from binary code to xeon phi," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 1078–1083.