Exploring the Limits of Generic Code Execution on GPUs via Direct (OpenMP) Offload

Shilei Tian[1][0000−0001−6468−6839], Barbara Chapman[1][0000−0001−8449−8579], and Johannes Doerfert[2][0000−0001−7870−8963]

[1] Stony Brook University, Stony Brook, NY 11794, USA
{shilei.tian,barbara.chapman}@stonybrook.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
jdoerfert@llnl.gov

Abstract. GPUs are well-known for their remarkable ability to accelerate computations through massive parallelism. However, offloading computations to GPUs necessitates manual identification of code regions that should be executed on the device, memory that needs to be transferred, and synchronization to be handled. Recent work has leveraged the portable target offloading interface provided by LLVM/OpenMP, taking GPU acceleration to a new level. This approach, known as the direct GPU compilation scheme, involves compiling the *entire* host application for the GPU and executing it there, thereby eliminating the need for explicit offloading directives. Nonetheless, due to limitations of the current GPU compiler toolchain and execution, seamlessly executing CPU code on GPUs with certain features remains a significant challenge. In this paper, we examine the limits of CPU code execution on GPUs by applying the direct GPU compilation scheme to LLVM's `test-suite`, analyze the encountered errors, and discuss potential solutions for enabling more code to execute on GPUs without any changes if feasible. By studying these issues, we shed light on how to improve GPU acceleration and make it more accessible to developers.

Keywords: OpenMP · GPU · compiler testing.

1 Introduction

GPUs are renowned for their exceptional computational power, primarily attributed to their ability to leverage massive parallelism. Offloading computations to GPUs has proven to be an effective approach for accelerating various applications. However, this process typically requires manual identification of code regions suitable for GPU execution, as well as managing data transfers and synchronization between the CPU and GPU. To address this challenge, recent work [20, 21] has proposed the direct GPU compilation scheme, which leverages the portable target offloading interface offered by LLVM/OpenMP. This scheme involves compiling the *entire* host application for the GPU and executing it there, eliminating the need for explicit offloading directives.

Despite the potential benefits of the direct GPU compilation scheme, there are limitations to executing CPU code on GPUs seamlessly due to current toolchain and execution constraints. In this paper, we examine the limits of CPU code execution on GPUs by applying the direct GPU compilation scheme to LLVM's test-suite. Through our analysis, we identify and categorize a series of encountered errors into eight distinct types, which encompass issues with test cases, bugs in the compiler and runtime, and the absence of certain features that led to the failure of test case compilations. In addition, we delve into potential solutions that could enable a wider range of codes to be executed on GPUs, ideally without necessitating any alterations to user codes, provided it's feasible. The study's primary objective is to elucidate the potential areas of improvement in GPU acceleration, thereby making it more user-friendly and accessible to developers.

The paper is organized as follows. In Section 2, we provide an overview of the direct GPU compilation scheme, which is the approach we use in this study. In Section 3, we describe our methodology and implementation details. Section 4 presents the results of our study, along with a detailed analysis. We review related works in Section 5. Finally, we conclude the paper in Section 6.

2 Background

OpenMP 4.0 introduced the `target` construct, which allows code regions to be executed on target devices such as GPUs [3] and FPGAs [10]. An example of CUDA code and its equivalent OpenMP version is shown in Fig. 1. In addition to the `target` construct (as well as its combined variants), OpenMP provides the `declare target` directive that specifies that all associated variables and functions are to be mapped onto the target devices and thus are usable in device code [18]. The `device_type(nohost)` clause on a `declare target` construct forces the compiler not to generate host versions of the enclosed variables and functions.

While this approach provides a simpler programming model than traditional CUDA or OpenCL, it still requires users to wrap the code with the `target` construct. In particular, users need to identify the regions of code that would benefit from GPU acceleration and explicitly mark them with the `target` construct.

The proposed approach by Tian et al. [20] enables the compilation of an existing host application for GPU execution with minimal modification to the user code by leveraging the portable target offloading interface provided by LLVM/OpenMP. Users can provide simple stub code to delegate function calls to the host using the host remote procedure call (RPC) framework for functions that can not be executed directly on a GPU. Later, the approach was extended by augmenting the compiler with a custom link-time optimization pass, which can automatically generate RPC calls without the need for stub code from users, and expand source parallelism to the entire GPU device [21].

The compilation and execution path of this approach is illustrated in Fig. 2. In the following we will briefly introduce the compilation of the direct GPU compilation scheme.

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

__global__ void baz() { foo(); }

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

(a) 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(); }
}
```

(b) Equivalent OpenMP code using target offloading to Fig. 1a. Even though there is no explicit kernel specified by users, an OpenMP compiler will outline the target region and generate a kernel implicitly.

Fig. 1: An example of CUDA code and its equivalent OpenMP code.

2.1 Device Code Representation

The direct compilation framework facilitates executing the entire program on the GPU by marking all user code associated with the `declare target` directive, essentially prepending a `begin declare target device_type(nohost)` before any user source file. The framework offers a user wrapper header (shown in Fig. 3), which can be pre-included using `clang`'s `-include` command line option when compiling user code.

2.2 Loader

The GPU execution still follows a "host-centric" approach where the execution of a "GPU program" must be initiated from the host. Traditionally, the `main` function in the host code has been the entry point for user applications. However, since the entire user code is now considered device code, a new entry point for the host code is needed. The direct compilation framework provides a main wrapper

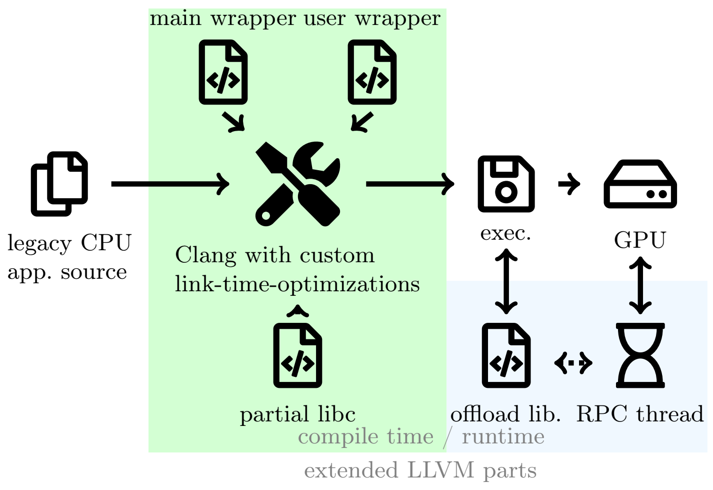

Fig. 2: Overview of the compilation and execution path of the direct GPU compilation framework introduced by Tian et al. [20] and the extended work [21]. The figure is from the work [21].

(also depicted in Fig. 2) that acts as the new host entry point. The main wrapper first maps all program arguments to the device so that the user code can access them and then invokes the user's `main` function. To avoid conflicts with the existing `main` function, the user's `main` function is renamed to `__user_main` (as illustrated in Fig. 3). The new host entry point must be compiled and linked with all other user source files into the executable by the user.

```
#pragma omp begin declare target device_type(nohost)
int main(int, char *[]) asm("__user_main");
```

Fig. 3: User wrapper header to take all user code as device code and rename `main` function to `__user_main`.

3 Methodology

This section introduces a new compiler driver wrapper designed to simplify the use of the direct compilation scheme. It then discusses the comprehensive handling of different `main` functions, followed by the test suite and system configuration employed in the exploration of the limits of generic code execution on GPUs using the direct compilation scheme.

3.1 Compiler Driver Wrapper

As described in Section 2, the direct GPU compilation scheme involves three steps: 1) compiling user source files with a user wrapper header included; 2) compiling the loader; and 3) linking the object files of user code and loader. However, the additional second step makes it difficult to seamlessly integrate the compilation scheme into build systems like CMake without significant changes to the configuration or build script.

To address this limitation, we have developed a solution by implementing a compiler driver wrapper, called `clang-gpu` and `clang-gpu++`, for C and C++ compilation, respectively. When the driver is used in compilation-only mode (`-c`), the wrapper forwards all arguments to the invocation of `clang`/`clang++` as well as all necessary extra compilation flags. Otherwise, the wrapper first compiles the loader and then adds the loader object file to the invocation of `clang`/`clang++` as an additional input file. With this approach, users can use `clang-gpu` and `clang-gpu++` as a regular compiler in build systems without requiring any changes.

3.2 Handling Different `main` Functions

In a host environment, a program must contain a global function named `main`, which serves as the designated start of the program. In C++, this function has one of two forms: `int main();` or `int main(int argc, char *argv[]);`. The C language also allows the form `void main();`.

As mentioned in Section 2, the user's `main` function is renamed to to `__user_main` to avoid ambiguity. However, this approach assumes that all users' `main` functions are in the form of `int main(int argc, char *argv[]);`, which may not always be the case and can lead to a compile error due to a conflict declaration of a function.

To address this issue, we implemented a compiler pass that "canonicalizes" the `main` function to the form `int main(int argc, char *argv[]);` and renames it accordingly. This approach ensures that the loader can correctly invoke the user's `main` function, regardless of its original form.

3.3 Test Suite and System Configuration

We used LLVM's `test-suite` to test the correctness and performance of the compilation scheme. This suite includes benchmarks and test programs, and provides tools to collect metrics such as benchmark runtime, compilation time, and code size. The suite includes several categories of tests, including:

- `SingleSource`: Contains single-file test programs.
- `MultiSource`: Includes entire programs with multiple source files.
- `MicroBenchmarks`: Programs using the `google-benchmark` library. The programs define functions that are run multiple times until the measurement results are statistically significant.
- `External`: Contains descriptions and test data for code that cannot be directly distributed with the test-suite. The most prominent members of this directory are the SPEC CPU benchmark suites.

- `Bitcode`: These tests are mostly written in LLVM bitcode.
- `CTMark`: Contains symbolic links to other benchmarks forming a representative sample for compilation performance measurements.

We chose not to perform runtime performance evaluation in this paper, as this has been extensively studied in prior work [20, 21]. We did not use the `Bitcode` tests, as these are written in LLVM bitcode, which is target dependent and can not be directly used for GPU testing.

Our system consisted of an NVIDIA A100 Tensor Core GPU (40GB) with AMD EPYC 7532 processors (32 cores with hyper-threading disabled) and 256 GB DDR4 RAM. We used CUDA 11.8.0 and compiled the entire test suite using the default configuration for release build. Fig. 4 shows how we configured, built, and executed the test suite.

```
$ cmake -G Ninja -S llvm-test-suite                                    \
  -DCMAKE_C_COMPILER=clang-gpu -DCMAKE_CXX_COMPILER=clang-gpu++
$ ninja -k 0
$ llvm-lit -v .
```

Fig. 4: Commands used to configure, build, and run the test suite. The compiler driver wrappers are used as compilers for C/C++ and there is no extra CMake configuration arguments nor changes in CMake files required.

4 Results and Analysis

The results of each subdirectory are presented in Fig. 5. In the subsequent sections, we delve into a detailed analysis of the different errors encountered, discussing its root causes and potential solutions.

4.1 Test Case Issue

We identified some issues in the test suite, such as the incorrect use of `parallel for` in the test case `SingleSource/Benchmarks/SmallPT/smallpt.cpp`, which caused a compile error (as shown in Fig. 6). This issue was not previously revealed because OpenMP was not enabled when compiling the `SingleSource` subdirectory. Another example is in `MultiSource/Applications/sgefa/driver.c`, where `malloc` is declared as `char *malloc();`, causing conflicting types for the function.

After fixing those issues, we got seven more passing tests: four in `MultiSource/Applications` and three in `MultiSource/Benchmarks`.

4.2 Compiler/Runtime Bug

We uncovered several bugs throughout the compiler, spanning front-end code generation, middle-end optimization, backend code generation, and runtime library. These bugs were discovered when assertions were triggered during compile

Sub Directory	Passed	Failed	Rate (%)
`SingleSource`	1641	185	89.9
`MultiSource`	125	75	62.5
`CTMark`	3	7	30
`MicroBenchmarks`	0	18	0

(a) Number of test cases and their compilation results in each subdirectory.

Sub Directory	Passed	Failed	Rate (%)
`SingleSource`	1641	0	100.0
`MultiSource`	5	120	4.0
`CTMark`	0	3	0.0

(b) Execution results of passed cases in Fig. 5a.

Fig. 5: Number of passed and failed test cases in each sub directory.

```
#pragma omp parallel for schedule(dynamic, 1) private(r)
fprintf(stderr,"Rendering (%d spp)\n",samps*4);
```

Fig. 6: Incorrect use of `parallel for` in the test case `SingleSource/Benchmarks/SmallPT/smallpt.cpp` that causes compile error.

or link time, indicating that certain errors were not caught beforehand and that certain assumptions made during development did not hold.

For example, while compiling the test case `CTMark/ClamAV`, `clang` crashed because the user code did not specify a size for a variable length array (VLA) in a way that was handled. The source excerpt is shown in Fig. 7. Despite of fact that the size of the array `dents` is a compile-time constant, rather than a literal, this error should have been detected earlier and an appropriate error message should have been produced, especially since VLAs are not currently supported when targeting GPUs (will be discussed in Section 4.6).

Another bug we encountered during our investigation was related to the LLVM Attributor framework. This bug manifests as an assertion error when compiling the test case `CTMark/tramp3d-v4`, as depicted in Fig. 8.

The majority of runtime failures shown in Fig. 5b were caused by illegal memory access. These failures can arise from various issues, including miscompilation or a faulty device runtime library. Further investigation is required to pinpoint the exact cause. Meanwhile, the other runtime failures were caused by issues in the automatic RPC implementation, where external functions were invoked on the host but the pointer arguments were not handled correctly.

In addition to the aforementioned bugs, we observed limitations in handling inline assembly and compiler intrinsics that are specifically target-dependent. Operations such as AVX512, which are specific to certain targets, are not portable by default. If inline assembly used in the code is not supported by the target, the compiler backend will crash instead of emitting an error. We will delve into this topic further in Section 4.8.

```
FAILED: CTMark/ClamAV/CMakeFiles/clamscan.dir/libclamav_readdb.c.o
...
clang-17: llvm-project/clang/lib/CodeGen/CodeGenFunction.cpp:2188:
clang::CodeGen::CodeGenFunction::VlaSizePair
clang::CodeGen::CodeGenFunction::getVLASize(const clang::
    VariableArrayType*):
Assertion ‘vlaSize && "no size for VLA!"’ failed.
```

(a) The assertion hit by `clang`.

```
/* MultiSource/Applications/ClamAV/libclamav_readdb.c */
static int cli_loaddbdir_l(...) {
  ...
  const unsigned MAX_DIRENTS = 20;
  struct dirent dents[MAX_DIRENTS];
  ...
```

(b) The corresponding source code caused the crash.

Fig. 7: `clang` crashed becasue an assertion is hit (top) for the source code shown in the bottom. Given the target does not support VLAs, an error should have been raised earlier.

```
FAILED: CTMark/tramp3d-v4/CMakeFiles/tramp3d-v4.dir/tramp3d-v4.cpp.o
...
clang-17:
llvm-project/llvm/lib/Transforms/IPO/AttributorAttributes.cpp:1536:
{anonymous}::AAPointerInfoFloating::updateImpl(llvm::Attributor&)::
<lambda(llvm::Value*, llvm::Value*, bool&)>:
Assertion ‘!PtrOI.isUnassigned() && "Cannot pass through if the input Ptr
     was not visited!"’ failed.
```

Fig. 8: `clang` crashed becasue an assertion is hit in LLVM's Attributor framework.

4.3 External Global Variable

Some header files contain external global variables, such as `extern std::ostream cout;` from `<iostream>` and `extern char *optarg;` from `<unistd.h>`. While in the extended work [21] external function calls are replaced with host RPC calls automatically by the compiler, external global variables are not handled in the same way. To address this, one possible solution is to replace access to an external global variable with host RPC calls, similar to the approach used for external functions. However, it may be difficult to handle pointers such as `extern char *optarg;`. Alternatively, with the emerging unified memory design where both CPU and GPU use the same memory, this will no longer be an issue.

4.4 Variadic Function

Variadic functions, such as `fprintf`, are commonly used in CPU code. However, they are not supported in GPU due to the lack of support from application binary interface (ABI). The extended work [21] managed to support external variadic functions in two steps: first, by creating a non-variadic wrapper on the device solely for host RPC calls, and second, by creating a non-variadic wrapper on the host side that recovers the call site. However, this approach may not work if users handle variadic arguments explicitly in the code, as shown in Fig. 9.

```
// CTMark/sqlite3/sqlite3.c
static int getDigits(const char *zDate, ...){
  va_list ap;
  ...
  va_start(ap, zDate);
  do{
    N = va_arg(ap, int);
    min = va_arg(ap, int);
    max = va_arg(ap, int);
    nextC = va_arg(ap, int);
    pVal = va_arg(ap, int*);
    ...
  }while( nextC );
  ...
  va_end(ap);
  return cnt;
}
```

Fig. 9: An example of explict handling of variadic arguments in the test case `CTMark/sqlite3`.

To solve this issue, a proper ABI for variadic functions needs to be defined. Some exploration has already been done in this area. For instance, NVIDIA GPUs can support the `printf` variadic function. In this case, the front end creates a structure at the call site that accommodates all variadic arguments, and then lowers the function call to `void vprintf(const char *fmt, void *args);`, where the second argument is a pointer to the structure. Fig. 10 demonstrates how this procedure works.

This approach can be extended to support the explicit handling of variadic functions, where both the caller and callee are compiled by the same compiler.

```
int a;
float b;
char *c;


printf("%d %f %s", a, b, c);
```

(a) Original function call to `printf`.

```
int a;
float b;
char *c;
struct {int a; float b; char *c;} s;
s.a = a; s.b = b; s.c = c;
vprintf("%d %f %s", &s);
```

(b) Pseudo code after lowering by `clang`.

Fig. 10: The lowering of `printf` in `clang` for NVIDIA GPUs.

4.5 C++ Exception Handling

Exceptions are a mechanism for handling exceptional circumstances, such as runtime errors, in programs by transferring control to special functions called handlers. An exception is thrown using the `throw` keyword from inside a `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block. However, no GPU compilers supports arbitrary C++ exception handling.

Full support for exceptions requires features such as stack unwinding, which are not yet available on GPUs. Moreover, the inherent dynamically divergent execution can cause problems on specific (lock step) targets. A reasonable alternative solution is to lower the `throw` expression to a built-in trap that aborts GPU execution. The `catch` statement then becomes a no-op, effectively equivalent to using the `-fno-exceptions` compiler flag except that the syntactic `throw` and `catch` statements would still be allowed.

4.6 Variable Length Array

Currently, GPU targets impose a limitation on stack allocation, requiring a statically known size. This constraint poses a challenge when dealing with variable length arrays that necessitate dynamic-sized stack allocation. Although NVIDIA has introduced a preview feature in PTX 7.3 that supports dynamic stack allocation [12], the compiler does not yet provide full support for this feature. To overcome this limitation, an alternative solution is to replace dynamic stack allocation with dynamic heap allocation. To ensure proper memory management, it becomes necessary to insert cleanup code that handles the deallocation of these dynamically allocated variables as their scope is left.

4.7 Unsupported Data Type

There are data types, such as `long double`, that are not supported by GPUs. Similar to the host side when the target CPU does not support certain types, software emulation can also be applied on GPUs. Moreover, more data types are likely to be supported in the future as GPUs evolves. For now, `clang` will allow `long double` and other unsuppoprted types to appear, e.g., as part of struct declarations, but it will not allow use of them, e.g., as part of arithmetic operations.

4.8 Inline Assembly

As mentioned earlier, both inline assembly and compiler intrinsics are inherently target-dependent and lack portability by default. To address this challenge, a potential solution is to translate the assembly code into the corresponding target-specific assembly code. This approach has been successfully employed in binary translation projects such as Apple's Rosetta 2 and Intel's Houdini, enabling cross-architecture execution. Similarly, for compiler intrinsics, a wrapper layer can be introduced to map them to a code sequence that is valid on the target architecture. This approach allows the intrinsic functions to be adapted and utilized in the context of the specific target architecture. A relevant study by Doerfert et al. [6] proposes techniques for mapping intrinsics to target-specific code sequences, offering a means to achieve compatibility across architectures. The OpenPOWER group provides functional equivalents of Intel MMX, SSE, and AVX intrinsic functions commonly used in Linux applications [17].

5 Related Work

Several prior works have investigated the execution of host programs on GPUs. Silberstein et al. [16] proposed direct access to the host's file system from GPU code and implemented an RPC protocol to facilitate data transfers between the CPU and GPU. Damschen et al. [4] explored transparent acceleration of binary applications using heterogeneous computing resources without manual porting or developer-provided hints. Matsumura et al. [9] introduced an automated stencil framework that transforms and optimizes stencil patterns in C source code, generating corresponding CUDA code. Mikushin et al. [11] presented a parallelization framework that detects parallelism and generates target code for both X86 CPUs and NVIDIA GPUs. To support functions that cannot be natively executed on GPUs, they replaced function calls in LLVM with an interface that uses a foreign function interface to execute the requested functions on the host. Jablin et al. [8] proposed a fully automatic system for managing and optimizing CPU-GPU communication, comprising a runtime library and compiler transformations. Pakin et al. [15] proposed reverse-acceleration model where the accelerators orchestrate the computation, offloading work that can not be accelerated to the general-purpose processors. Tian et al. [20] were the first to attempt running the entire host program on a GPU using OpenMP target offloading. They augmented the compiler with a custom link-time optimization pass to generate RPC calls automatically, eliminating the need for stub code from users and expanding source parallelism to the entire GPU device. Their work later has been extended in [21], where the compiler was augmented with a custom link-time optimization pass, which can automatically generate RPC calls without the need for stub code from users, and expand source parallelism to the entire GPU device.

In recent years, researchers have focused on compiler and runtime optimization for OpenMP after the introduction of target offloading in OpenMP 4.0. Bertolli et al. [2, 3] enabled OpenMP offloading to GPUs in LLVM. Flang, the PGI Fortran front-end, also supports OpenMP offloading through the LLVM OpenMP runtime [13]. Antão et al. [1] introduced front-end-based optimizations for NVIDIA GPUs, reducing register usage and avoiding idle threads. Doerfert et al. [5] presented the TRegion interface, enabling more kernels to execute in SPMD mode. Tian et al. [19] introduced runtime support for concurrent execution of OpenMP target tasks. Yviquel et al. [22] presented a framework for using the OpenMP programming model in distributed memory environments, combining OpenMP directives and MPI communication. Huber et al. [7] developed OpenMP-aware program analyses and optimizations for efficient execution of CPU-centric parallelism on GPUs. Ozen and Wolfe [14] demonstrated its implementation of the `loop` directive on NVIDIA GPUs.

6 Summary

In this paper we investigated the feasibility and effectiveness of executing CPU code on GPUs using the direct GPU compilation scheme. We highlighted the challenges and limitations in the current GPU compiler toolchain and hardware support. In addition, we discussed potential solutions to enable broader GPU execution capabilities. The findings can contribute to advancing GPU acceleration and facilitating the utilization of GPUs for a wider range of code without significant modifications from application developers.

This work highlights the effectiveness of the compilation scheme introduced in [20, 21], which enables straightforward execution of CPU codes on GPUs, to test "GPU compilers" using a vast collection of existing CPU code. In this initial study alone we detected multiple compiler bugs and categorized other the shortcomings; both will lead to improved capabilities and robustness.

Acknowledgement

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. 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 work was in parts prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-827970). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory.

References

[1] Antão, S.F., Bataev, A., Jacob, A.C., Bercea, G., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., O'Brien, K.: Offloading Support for OpenMP in Clang and LLVM. In: Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 14, 2016, pp. 1–11, IEEE Computer Society, Salt Lake

City, UT, USA (2016), https://doi.org/10.1109/LLVM-HPC.2016.006, URL https://doi.org/10.1109/LLVM-HPC.2016.006

[2] Bertolli, C., Antão, S., Bercea, G., Jacob, A.C., Eichenberger, A.E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D., O'Brien, K.: Integrating GPU support for OpenMP offloading directives into Clang. In: Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 15, 2015, pp. 5:1–5:11, ACM, Austin, Texas, USA (2015), https://doi.org/10.1145/2833157.2833161, URL https://doi.org/10.1145/2833157.2833161

[3] Bertolli, C., Antão, S., Eichenberger, A.E., O'Brien, K., Sura, Z., Jacob, A.C., Chen, T., Sallenave, O.: Coordinating GPU threads for OpenMP 4.0 in LLVM. In: Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 17, 2014, pp. 12–21, IEEE Computer Society, New Orleans, LA, USA (2014), https://doi.org/10.1109/LLVM-HPC.2014.10, URL https://doi.org/10.1109/LLVM-HPC.2014.10

[4] Damschen, M., Riebler, H., Vaz, G., Plessl, C.: Transparent offloading of computational hotspots from binary code to Xeon Phi. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) March 9-13, 2015, pp. 1078–1083, ACM, Grenoble, France (2015), URL http://dl.acm.org/citation.cfm?id=2757063

[5] Doerfert, J., Diaz, J.M.M., Finkel, H.: The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In: International Workshop on OpenMP (IWOMP), September 11-13, 2019, vol. 11718, pp. 153–167, Springer, Auckland, New Zealand (2019), https://doi.org/10.1007/978-3-030-28596-8_11, URL https://doi.org/10.1007/978-3-030-28596-8_11

[6] Doerfert, J., Jasper, M., Huber, J., Abdelaal, K., Georgakoudis, G., Scogland, T., Parasyris, K.: Breaking the vendor lock: Performance portable programming through openmp as target independent runtime layer. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), October 8-12, 2022, pp. 494–504, ACM, Chicago, Illinois (2022), https://doi.org/10.1145/3559009.3569687, URL https://doi.org/10.1145/3559009.3569687

[7] Huber, J., Cornelius, M., Georgakoudis, G., Tian, S., Diaz, J.M.M., Dinel, K., Chapman, B.M., Doerfert, J.: Efficient Execution of OpenMP on GPUs. In: International Symposium on Code Generation and Optimization (CGO), April 2-6, 2022, pp. 41–52, IEEE, Seoul, Republic of Korea (2022), https://doi.org/10.1109/CGO53902.2022.9741290, URL https://doi.org/10.1109/CGO53902.2022.9741290

[8] Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 4-8, 2011, pp. 142–151, ACM, San Jose, CA, USA (2011), https://doi.org/10.1145/1993498.1993516, URL https://doi.org/10.1145/1993498.1993516

[9] Matsumura, K., Zohouri, H.R., Wahib, M., Endo, T., Matsuoka, S.: AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs. In: International Symposium on Code Generation and Optimization (CGO), February, 2020, pp. 199–211, ACM, San Diego, CA, USA (2020), https://doi.org/10.1145/3368826.3377904, URL https://doi.org/10.1145/3368826.3377904

[10] Mayer, F., Knaust, M., Philippsen, M.: OpenMP on FPGAs - A Survey. In: International Workshop on OpenMP (IWOMP), September 11-13, 2019, vol. 11718, pp. 94–108, Springer, Auckland, New Zealand (2019), https://doi.org/10.1007/978-3-030-28596-8_7, URL https://doi.org/10.1007/978-3-030-28596-8_7

[11] Mikushin, D., Likhogrud, N., Zhang, E.Z., Bergstrom, C.: Kernelgen - the design and implementation of a next generation compiler platform for accelerating numerical models on gpus. In: International Parallel & Distributed Processing Symposium Workshops (IPDPSW), May 19-23, 2014, pp. 1011–1020, IEEE Computer Society, Phoenix, AZ, USA (2014), https://doi.org/10.1109/IPDPSW.2014.115, URL https://doi.org/10.1109/IPDPSW.2014.115

[12] NVIDIA: Parallel Thread Execution ISA Version 8.1. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#stack-manipulation-instructions-alloca (2023)

[13] Özen, G., Atzeni, S., Wolfe, M., Southwell, A., Klimowicz, G.: OpenMP GPU Offload in Flang and LLVM. In: Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 13, 2018, pp. 1–9, IEEE, Dallas, TX, USA (2018), https://doi.org/10.1109/LLVM-HPC.2018.8639434, URL https://doi.org/10.1109/LLVM-HPC.2018.8639434

[14] Ozen, G., Wolfe, M.: Performant Portable OpenMP. In: ACM SIGPLAN International Conference on Compiler Construction (CC), April 2 - 3, 2022, pp. 156–168, ACM, Seoul, South Korea (2022), https://doi.org/10.1145/3497776.3517780, URL https://doi.org/10.1145/3497776.3517780

[15] Pakin, S., Lang, M., Kerbyson, D.J.: The Reverse-Acceleration Model for Programming Petascale Hybrid Systems. IBM Journal of Research and Development **53**(5), 8 (2009), https://doi.org/10.1147/JRD.2009.5429074, URL https://doi.org/10.1147/JRD.2009.5429074

[16] Silberstein, M., Ford, B., Keidar, I., Witchel, E.: GPUfs: Integrating A File System with GPUs. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 16-20, 2013, pp. 485–498, ACM, Houston, TX, USA (2013), https://doi.org/10.1145/2451116.2451169, URL https://doi.org/10.1145/2451116.2451169

[17] System Software Work Group, OpenPOWER Foundation: Vector Intrinsics Porting Guide. https://openpowerfoundation.org/specifications/vectorintrinsicportingguide/ (2018)

[18] Tian, S., Chesterfield, J., Doerfert, J., Chapman, B.M.: Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1. In: International Workshop on OpenMP (IWOMP), September 14-16, 2021,

vol. 12870, pp. 159–169, Springer, Bristol, UK (2021), https://doi.org/10.1007/978-3-030-85262-7_11, URL https://doi.org/10.1007/978-3-030-85262-7_11

[19] Tian, S., Doerfert, J., Chapman, B.M.: Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In: Languages and Compilers for Parallel Computing (LCPC), October 14-16, 2020, vol. 13149, pp. 41–56, Springer, Stony Brook, NY, USA (2020), https://doi.org/10.1007/978-3-030-95953-1_4, URL https://doi.org/10.1007/978-3-030-95953-1_4

[20] Tian, S., Huber, J., Parasyris, K., Chapman, B.M., Doerfert, J.: Direct GPU Compilation and Execution for Host Applications with OpenMP Parallelism. In: Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 13-18, 2022, pp. 43–51, IEEE, Dallas, TX, USA (2022), https://doi.org/10.1109/LLVM-HPC56686.2022.00010, URL https://doi.org/10.1109/LLVM-HPC56686.2022.00010

[21] Tian, S., Scogland, T., Chapman, B., Doerfert, J.: GPU First – Execution of Legacy CPU Codes on GPUs (2023)

[22] Yviquel, H., Pereira, M., Francesquini, E., Valarini, G., Leite, G., Rosso, P.H.D.F., Ceccato, R., Cusihualpa, C., Dias, V., Rigo, S., Souza, A., Araujo, G.: The OpenMP Cluster Programming Model. In: Workshop of the International Conference on Parallel Processing (ICPP), 29 August 2022 - 1 September 2022, pp. 17:1–17:11, ACM, Bordeaux, France (2022), https://doi.org/10.1145/3547276.3548444, URL https://doi.org/10.1145/3547276.3548444