








# Just-in-Time Compilation and Link-Time Optimization for OpenMP Target Offloading

Shilei Tian<sup>1</sup>(✉) , Joseph Huber<sup>2</sup> , John Tramm<sup>3</sup> , Barbara Chapman<sup>1</sup> ,  
and Johannes Doerfert<sup>3</sup>(✉) 

<sup>1</sup> Stony Brook University, Stony Brook, USA  
{shilei.tian,barbara.chapman}@stonybrook.edu  
<sup>2</sup> Oak Ridge National Laboratory, Oak Ridge, USA  
huberjn@ornl.gov  
<sup>3</sup> Argonne National Laboratory, Lemont, USA  
{jtramm,jdoerfert}@anl.gov

**Abstract.** Following the mass adoption of external accelerators for high performance computing, the overall performance of many applications has become increasingly dependent on relatively small accelerated kernels. As static analysis is fundamentally limited by dynamic values and external definitions, standard ahead-of-time compilation is not always sufficient to achieve the best performance. Furthermore, many users looking to port an existing application to run on an external accelerator will not want to fundamentally restructure their programs. These and other problems can be addressed through both link-time optimization (LTO) and just-in-time (JIT) compilation, but until now had sparse and inconsistent support from the compiler.

In this work, we present a new compilation method that enables device-side LTO as well as a transparent JIT compilation tool-chain for OpenMP target offloading. Our contributions include an entirely new device linking and embedding scheme to enable LTO as well as a novel JIT engine to efficiently optimize OpenMP offloading regions at run-time. We also introduce a persistent caching system to improve end-to-end run-time using the JIT engine and minimize kernel launching overheads. We measure the performance of our LTO and JIT implementation via several real-world scientific applications. With our optimizations we observe significant improvements through LTO on large applications as well as significant end-to-end execution time improvement using JIT.

**Keywords:** OpenMP · GPU · LTO · JIT

## 1 Introduction

The dominance of massively-parallel GPGPU based accelerators in high performance computing systems has resulted many applications being highly dependent on small accelerated kernels executed on the device. This poses a challenge

for compilers looking to optimize applications targeting heterogeneous systems, especially through generic programming models, such as OpenMP target offloading. The massively parallel nature of these systems means that any missed optimizations or overhead can result in considerably large performance losses. Furthermore, the compiler’s ability to optimize these program is fundamentally limited by external definitions or dynamic values only known at runtime. OpenMP especially makes heavy use of environment variables whose values can only be known at runtime. This means that ahead-of-time (AoT) optimizations alone are not sufficient to determine important constants, such as the number of teams and threads in a region.

In this work we present a transparent implementation of link-time optimization (LTO) and just-in-time (JIT) compilation for OpenMP offloading for the LLVM/Clang compiler infrastructure. We first show an overhauled driver for compiling OpenMP offloading programs in LLVM/Clang that allows transparent embedding and linking of device LLVM IR. The JIT engine uses the linked bitcode to perform further optimizations and code generation at runtime with the knowledge of runtime values, e.g., environment variables. Finally, we present the performance improvements of the LTO and JIT compilation on several benchmarks and proxy-applications.

In the following we first briefly explain the necessary background on OpenMP offloading compilation via LLVM/Clang, LTO, and JIT. Sections 3 and 4 describe our LTO and JIT contributions in detail. The evaluation of our approach is given in Sect. 5, and finally, before the conclusion in Sect. 7, we discuss related works in Sect. 6.

## 2 Background

In this section, we will briefly introduce the current compilation pipeline used to create OpenMP target offloading applications and support LTO and JIT.

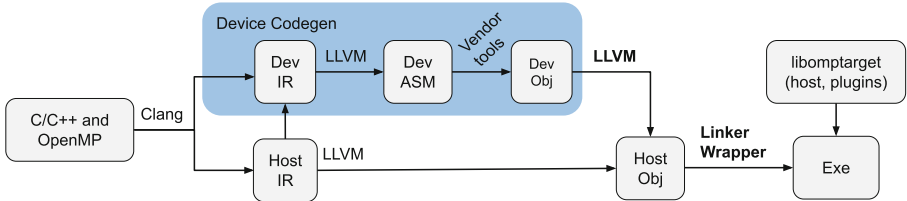
The LLVM/Clang compiler driver is responsible for creating the necessary actions to produce the compiled output. Compilation for LLVM/OpenMP offloading is more complex than a standard compilation job because the compiler must compile and link for multiple architectures at once. In order to maintain standard compilation semantics, the compiler driver will create a compilation for each target architecture and then embed the result into a single fat binary. Device linking occurs by extracting the device code inside the fat binary and first running the appropriate device linking job on it. After linking, the device image still needs to be registered with offloading runtime on the host. To register the device image we create a new module containing the necessary registration code and link it with the rest of the application.

Standard LTO in LLVM is performed by emitting LLVM IR bitcode instead of an object file. During linking, if the linker supports LTO, all the identified LLVM IR bitcode files will be merged and optimized together using symbol resolution information from the linker. The linked and optimized bitcode is then compiled and linked with the other input files. Similarly, JIT compilation uses LLVM IR bitcode to compile for the target architecture as-needed at runtime.

### 3 Link Time Optimization Support

As mentioned in Sect. 2, creating offloading binaries is more challenging than standard compilation. In this section, we first introduce our new offloading driver for LLVM/Clang. Then, we describe our new augmented linker to support device linking with LTO.

#### 3.1 Offload Driver



**Fig. 1.** The main phases of the new LLVM/Clang offloading driver. Source file compilation is done for every target architecture using the device tools. This is then embedded into the host to create a fat-binary and linked with the new augmented linker.

Our new driver supports OpenMP offloading compilation in a unified manner by utilizing a common embedding and linking scheme for each target architecture. The first step is to compile each input OpenMP offloading program to an object file using LLVM/Clang. If we are performing LTO, we instead emit LLVM-IR instead of a standard ELF object file.

We then complete the host compilation and create a fat-binary by storing each output device object file in a special section in the output ELF file. This special section contains both the embedded image and a binary blob containing necessary metadata to link the device image, such as its target and architecture. This section will be named `llvm.offloading` and is identified in the ELF using the new `SHT_LLVM_OFFLOADING` section type. Furthermore, we use the `SHF_EXCLUDE` section flag to indicate that this section should be dropped by the linker when creating the final executable. The final compilation step is to pass the new fat binary to the linker, which will use the embedded device code to create an executable device image. These steps are roughly outlined in Fig. 1.

#### 3.2 Offload Linker

We created a new augmented linker supporting offloading device linking and registration. This new augmented linker works as a thin wrapper over the original host linking job called the linker wrapper. First, the linker wrapper searches every

input file or library for embedded device code stored in the `SHT_LLVM_OFFLOADING` sections. Once the files are located and extracted we sort each input object using its target architecture that was extracted from the metadata stored previously. We then identify all the input files containing LLVM-IR and use LLVM's existing LTO library to create an object file output. All device objects for a single target architecture are then linked together using the vendor linker to create an executable.

The device executable is useless on its own, so first we need to register each linked executable with the vendor's runtime library. We perform this final step by creating a new module containing the executable data and the runtime calls necessary to register it. This module is then compiled to an ELF object file and added to the linker input. Finally, we run the original host linking job and obtain an executable containing offloading code.

## 4 Just-in-Time Compilation Support

In this section, we will introduce our support for JIT compilation of OpenMP offloading applications in LLVM. We will first talk about the necessary compiler support, followed by the code generation and sub-architecture portability support. Next, we propose three specializations to improve optimizations using JIT. Finally, we discuss a multi-level caching implementation used to mitigate host overhead caused by JIT compilation and optimization.

### 4.1 Compilation Flow

We utilized the LTO support shown in Sect. 3 to create linked LLVM-IR necessary for JIT compilation. The only change required to the compilation flow for JIT is to skip the LTO back-end in the linker and register the linked LLVM-IR directly. Now when the runtime attempts to register the device code it will use JIT if it encounters LLVM-IR instead of a device executable.

### 4.2 JIT Kernel Invocation

When the OpenMP runtime attempts to execute a kernel when performing JIT we first need to compile and register it as before. We will also use LLVM's LTO support for the code generation to call the same back-end we skipped during ahead-of-time linking. The previous LTO optimization pass is augmented with JIT-specific optimizations that will be described later as well as aggressive pruning of global definitions unused by the current kernel. After the LTO backend is run, we then need to register the kernel with the device runtime and proceed to the kernel launch.

### 4.3 Sub-architecture Portability

AoT compilation does not support sub-architecture portability because device images are not usually compatible with different compute capabilities. For example, if the program is compiled for `sm_35`, it can only run on `sm_35` GPUs. Programs instead must be recompiled for the desired target device.

With our JIT support, device images are generated at runtime using target information collected from the target device, thus we only need to compile programs once (AoT) and they can be executed on different target devices with varying sub-architectures. However, it is worth noting that this does not support the portability across different vendors. As we mentioned before, we embed LLVM IR which is still inherently vendor dependent. In addition, for AMD GPUs, an extra pass is set up to update all target features attached to functions to make sure its backend works properly.

### 4.4 Specialization

We propose three specializations with information only available at runtime. They are all enabled by default and can be configured via environment variables.

**Scalar Kernel Arguments.** One of the most important pieces of runtime information is kernel arguments. There are two kinds of kernel arguments: pointer values and scalar values. We do not specialize pointer values because it can easily invalidate caching, which will be discussed later, incurring more host side overhead. Scalar values are however specialized, hence replaced with their runtime value prior to optimizations. If the scalar values are loop bounds, it can make more aggressive loop optimization possible.

**Pointer Alignments.** An important characteristic of a pointer is its alignment, which plays an important role in vectorization and instruction selection. Although each target has a default pointer alignment, the actual alignment of a pointer can be more strict. For each pointer value  $p$ , we iterate a list of predefined alignments  $a \in \{128, 64, 32, 16, 8\}$  in decreasing order and find the first  $a$  that  $p$  is aligned to. If  $a$  is greater than the default alignment, an attribute `align` with value  $a$  is added to the pointer kernel argument.

**Launch Parameters.** Two kernel launch parameters, grid size and block size, are provided to the driver API when launching a kernel. If the `num_teams` clause or `thread_limit` clause is present and a compile time constant value is specified, the corresponding runtime functions to query the size are optimized away at compile time [1]. If the clause is not specified, the runtime will choose a default value. Given that these launch parameters are known to the JIT, specialization is performed as if the user provided constant values via the respective clauses.

## 4.5 Internalization

In the device runtime, there are global variables listed in `@llvm.used` to prevent to be optimized out when building the device runtime. At JIT time<sup>1</sup>, since the module has already been linked, all feasible global variables, except those that should be exposed to users, can be optimized. We mark all global variables, except those exposed to users, as `internal` and remove them from `@llvm.used`.

## 4.6 Caching

JIT compilation requires constructing an `LTOModule`, going through kernel arguments, modifying the module, generating a device image, and loading it to the target device. This can have significant overheads and can potentially cancel the benefits of our runtime optimizations. To mitigate the cost we need to reuse the generated device image for multiple kernel launches. To this end, we implement a novel classification system for kernel launches together with a persistent, two-level cache system that keeps specialized images in the host memory (L2) as well as in the device memory (L1) for future reuse. JIT compilation is only invoked if there is no compatible cached image available at launch time.

**Kernel Launch Identification.** In order to reuse an image, we need an efficient way to determine if it is compatible with an incoming kernel launch request. A kernel launch is effectively defined by the kernel (function) name, the kernel arguments, and the number of teams and threads (= grid dimensions). However, kernel launches do not require identical values to share/reuse the same optimized image. For example, pointers `0x1230` and `0x4560` are not exactly same, but they are both 16-byte aligned. If that is the only difference between two kernel launches, they are compatible. Additionally, if parameters are not involved in specialization their values do not impact kernel launch compatibility.

In order to efficiently query the cache we employ a *kernel launch descriptor*, which includes: the kernel name, kernel arguments, architecture, and a list of *specializations* applied to the kernel when the image was compiled. An existing optimized image with a kernel launch descriptor is compatible with a kernel launch, and hence can be reused, if (1) the kernel name and architecture match and (2) the specializations applied to obtain the image match what would have been applied for the new launch in question.

**L1: Target Table Cache.** A *target table* stores information about offloading entries, such as entry size, host pointer and its corresponding device pointer. It is constructed when an image is loaded to the device. Therefore, it is per device and every execution starts with an empty L1 cache that is filled on-demand.

We set up a target table cache, indexed by kernel entry name, for each target device. Each entry is a list of target tables for the same kernel entry but with different kernel launch descriptors.

---

<sup>1</sup> Technically, this does not have to be limited to JIT time but LTO time is sufficient.

**L2: Image Cache.** An *image* is a memory buffer that can be loaded to a target device. It can be used for all target devices with same sub-architecture. More importantly, it does not contain (dynamic) device pointers, which gives us the ability to reuse it across executions. Images are reused within and across program runs whenever a compatible kernel launch is encountered.

We set up an image cache for each sub-architecture. An image cache is organized similar to target table cache. In addition, during the runtime shutdown, the image cache writes all cached images and metadata to a file. When the runtime is loaded, it reads all images from the file and construct the image cache to be used by the application. Hence, prior runs with compatible kernel launch parameters can effectively eliminate most overheads of just-in-time compilation.

**Cache Lookup.** When a kernel  $k$  is launched, the cache lookup works as follows:

1. Check if there is a compatible entry in the target table cache (L1) for the target device. If yes, move to Step 2; otherwise, move to Step 3.
2. Iterate over the list of target tables. If there is a *match*, it is a L1 cache *hit*, and the target table can be used directly; otherwise, it is a L1 cache *miss* and we proceed with Step 3.
3. Check if there is a compatible entry in the image cache (L2) for the sub-architecture of the target device. If yes, move to Step 4; otherwise, it is a L2 cache *miss* and we proceed to Step 5.
4. Iterate over the list of images. If there is a *match*, it is a L2 cache *hit*. The image will be loaded to the target device, a new target table will be constructed and added to the L1 cache. If no *match* was found it is a cache *miss*. Move to Step 5.
5. JIT the device image, add it to the L2 cache (for cross-execution persistence), load it to the target device, and add it to the L1 cache.

## 4.7 Specialization Tracker

In spite of the multi-level caching system, it is still possible that scalar kernel arguments for a kernel vary in every (or many) different kernel launches. For example, `552.pep` in SPEC ACCEL [2] has one scalar argument that changes in every kernel launch. As consequence, we have to compile a new image and load it to the device for every launch. This situation can cause significant overheads and device resource waste.

We set up a specialization tracker for each kernel entry which records the total number of specializations, denoted by  $N$ , and the number of specializations for each kernel argument, including the launching parameters, represented by  $n_i$ . Before we apply any specialization, we check if  $N > T$  and  $n_i/N > R$ , where  $T$  is a threshold to always allow a certain amount of specialization, and  $R$  is an argument specialization control ratio. If both conditions are true we have exceeded the specialization quota for an argument and it is not specialized in the future. For any subsequent kernel launch, no matter whether the argument value change, there has to be a match as no argument specializations has been applied to one image. Both  $T$  and  $R$  can be configured via environment variables.

## 5 Evaluation

For our performance evaluation we used a Nvidia A100 GPU system with an AMD EPYC 7532 CPU and 256 GB DDR4 RAM. We used CUDA 11.4.0 for all experiments and collected kernel times with `nsys`. In addition to the Nvidia system, an AMD MI100 GPU system with two AMD EPYC 7532 CPUs and 512 GB DDR4 RAM is used for portability evaluation. Our prototype version ( $\mathcal{P}$ ) is based on `git 3723868d`.

### 5.1 Benchmarks

We looked at seven scientific proxy applications for our performance study for LTO and JIT compilation and evaluated both the end-to-end execution time and the performance of their main GPU kernels. Our results are presented relative to the performance of AoT compilation without LTO. We also test four of the seven proxy applications for sub-architecture portability with JIT.

**OpenMC** is a continuous-energy Monte Carlo particle transport application [3] that has recently been ported to the OpenMP target offloading programming model for use on GPU-based systems [4]. In addition to being an open source application, OpenMC also provides a host of advanced modeling and simulation capabilities including depletion, advanced geometry representations, on-the-fly Doppler broadening, and multigroup cross section generation.

**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 in memory bound, RSbench [6] provides a compute bound alternative implementation.

**MiniFMM** is a proxy application developed by the University of Bristol for Fast Multipole Method (FMM) [7]. It solves the Laplace equation in a three-dimensional polar coordinate plane by applying the FMM, which uses a dualtree traversal method.

**SU3** is a Lattice QCD SU(3) matrix-matrix multiply microbenchmark. The kernel is based on the `mult_su3_nn()` SU(3) matrix-matrix multiply routine in the MILC Lattice Quantum Chromodynamics(LQCD) code.

**Thermo4PFM** is a software library used for Phase-Field modeling of solidification in metallic alloys [8]. Given the thermodynamic properties of a materials in its various phases, it solves a small system of non-linear equations to compute the force that drives phase changes.

**miniMDock** is a GPU-accelerated performance portable particle-grid based protein ligand molecular docking tool. It is used for virtual drug discovery compound screens based on a molecular recognition model, that analysis a three-dimensional model of an interaction between a protein and a small molecule.



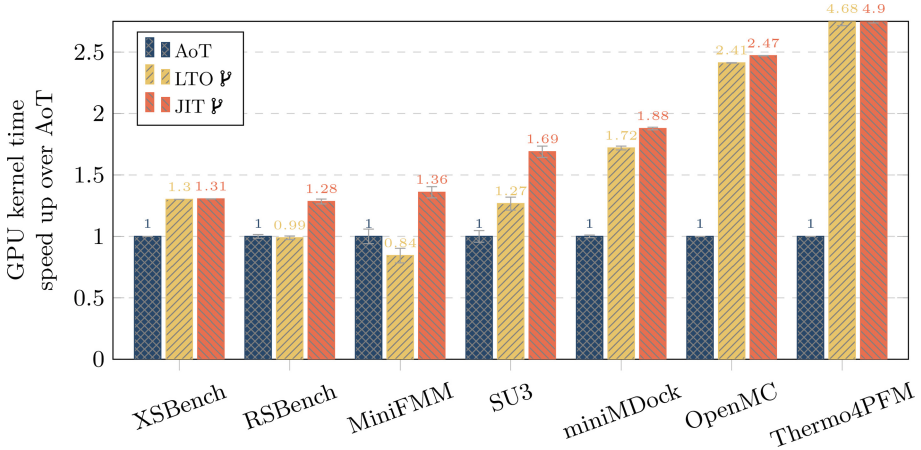


Fig. 2. Kernel execution time relative to the base AoT case without LTO.

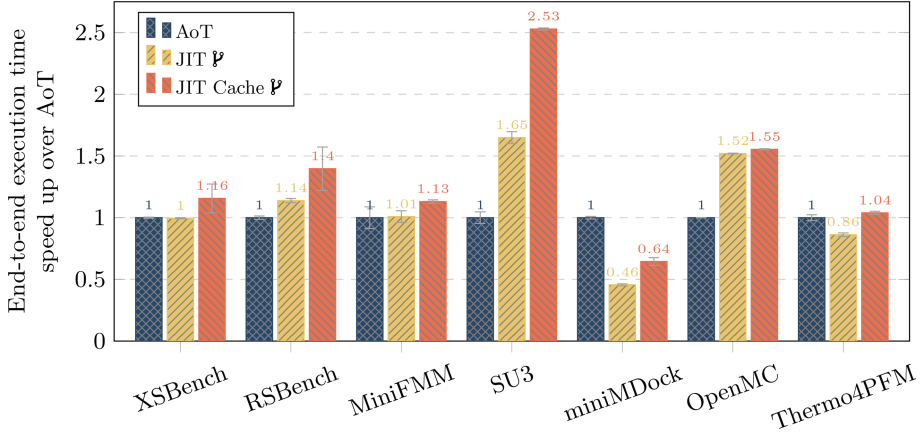
## 5.2 Performance Results

Figure 2 shows the relative improvement in kernel time for LTO and JIT. Our results show large improvements for miniMDock, OpenMC, and Thermo4PFM when LTO is used because these applications have code split between many files and benefit most from LTO. Although the other cases do not benefit from cross-file optimizations, LTO can still affect performance due to additional optimizations and internalization of external symbols. The performance evaluation for JIT uses only the kernel timings and does not include the overhead necessary to first compile the image. Hence, we assume a perfect pre-filled cache.

Figure 3 shows the relative improvement in end-to-end time for JIT with and without the offline cache. We can see that in most cases, JIT compilation can improve the end-to-end performance, except for miniMDock and Thermo4PFM. For miniMDock, the total kernel time is optimized to about 1.4 s from 2.6 s. However, because miniMDock uses random inputs the cache is easily invalidated which results in overall slower execution. In the worst case, the overall JIT overhead is more than 4.5 s, leading to the performance regression shown in Fig. 3. This demonstrates that for applications similar to miniMDock, specialization should be disabled (adaptively). Thermo4PFM also shows an end-to-end performance regression, but the offline cache allows it to retain performance.

## 5.3 Portability Results

Although performance can change when using JIT, another advantage is providing sub-architecture portability. The sub-architectures of our GPU systems are SM80 (Nvidia A100) and GFX908 (AMD MI100) respectively. Figure 4 shows the results of different benchmarks compiled with different sub-architectures on the two GPU systems, where  $\checkmark$  means the benchmark runs without any issue.



**Fig. 3.** End-to-end execution time relative to the base AoT case without LTO.

Benchmark	SM35	SM53	SM60	SM75	GFX701	GFX803	GFX902
XSbench	✓	✓	✓	✓	✓	✓	✓
RSBench	✓	✓	✓	✓	✓	✓	✓
MiniFMM	✓	✓	✓	✓	✓	✓	✓
SU3	✓	✓	✓	✓	✓	✓	✓

**Fig. 4.** Portability of benchmarks compiled with different SM versions for Nvidia A100 GPU and different GFX versions for AMD MI100 GPU.

## 6 Related Works

### 6.1 OpenMP Target Offloading

OpenMP 4.0 introduced target offloading. In LLVM/Clang, OpenMP offloading support for GPUs was first presented by [9, 10]. The (PGI) Fortran front-end, known as Flang, supports OpenMP offloading via the LLVM/OpenMP runtimes [11]. GCC 5 first supports OpenMP target offloading on Intel MIC architecture. Starting from GCC 7, Nvidia platforms support was added. All existing implementation feature ahead-of-time compilation of device code.

### 6.2 Just-in-Time Compilation

Just-in-time compilation has been used in software systems for decades [12]. However, the support in programming languages vary. For parallel programming models, OpenCL naturally employs JIT compilation for parallel code execution via an intermediate representation SPIR-V [13]. [14] implemented automatic translation of OpenACC to LLVM IR with SPIR kernels, optimization of the IR code by LLVM optimizer, and execution of the host LLVM IR by LLVM JIT. For OpenMP, [15] proposed an on-the-fly technique on top of the Pin binary

instrumentation [16] to detect data races in OpenMP programs. [17] presented support for parallel programs written in OpenMP executing on JVM using LLVM IR.

### 6.3 Link Time Optimization

Link-time optimization is not a new concept and has been supported by various compilers, including GCC and LLVM/Clang [18]. Nvidia has supported device-side LTO for CUDA following the CUDA 11.2 release [19], however the Nvidia compilers do not support LTO for OpenMP offloading. The AMDGPU toolchain uses LLVM IR bitcode as its relocatable object file format and used bitcode linking and LLVM optimizations as a part of its compilation.

### 6.4 Compiler Optimization for OpenMP

Regarding compiler-based optimizations on OpenMP, [20] introduced the first front-end based optimizations for Nvidia GPUs in LLVM/Clang, related to choosing the number of teams and threads for parallel loops to avoid idle threads and reduce register usage. [21] presented the TRegion interface which delayed the discovery of SPMD regions into LLVM, by contrast to the Clang-based approach, which enabled more kernels to execute in SPMD mode. [22] introduce in the IBM XL C/C++ compiler a lowering of OpenMP that executes without the control loop state machine in a mode where all threads execute in parallel, deemed SPMD mode of execution, when the target offloaded region encloses a single parallel construct. [1] presented OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. [23] presented a co-design methodology for optimizing applications using a specifically crafted OpenMP GPU runtime inducing near-zero overhead in most cases. Recent advances in architecture porting have made it feasible to extend our work of sub-architecture specialization to retargeting across different vendors [24, 25].

To our best knowledge, this is the first work to present JIT compilation and LTO for OpenMP target offloading.

## 7 Conclusion and Future Works

In this paper, we proposed just-in-time compilation and link-time optimization for LLVM/OpenMP target offloading. We showed a new compiler driver to embed and link device bitcode, and a novel JIT engine that features optimization, caching, and sub-architecture portability. The evaluation results show that link-time optimization can provide large performance benefits for certain applications and we can further optimize applications and offer sub-architecture portability using JIT compilation. In the future, we plan to further improved JIT compilation and specialization in LLVM/Clang as a default for better interoperability between architectures.

**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. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (<https://energy.gov/downloads/doe-public-access-plan>).

## References

1. Huber, J., et al.: Efficient Execution of OpenMP on GPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Republic of Korea, 2–6 April 2022, pp. 41–52 (2022)
2. Juckeland, G., et al.: 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, 16 November 2014. Revised Selected Papers. vol. 8966, pp. 46–67 (2014)
3. Romano, P.K., Horelik, N.E., Herman, B.R., Nelson, A.G., Forget, B.: OpenMC: a state-of-the-art Monte Carlo code for research and development. *Ann. Nucl. Energy* **82**, 90–97 (2015). <https://doi.org/10.1016/j.anucene.2014.07.048>, <https://doi.org/10.1016/j.anucene.2014.07.048>
4. Tramm, J., et al.: Toward portable GPU acceleration of the OpenMC Monte Carlo particle transport code. In: International Conference on Physics of Reactors (PHYSOR 2022). Pittsburgh, USA (2022)
5. Tramm, J.R., Siegel, A.R., Islam, T., Schulz, M.: XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In: PHYSOR (2014)
6. Tramm, J.R., Siegel, A.R., Forget, B., Josey, C.: 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, 2–3 April 2014, Revised Selected Papers. vol. 8759, pp. 39–56 (2014)
7. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 92–106. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-65578-9\\_7](https://doi.org/10.1007/978-3-319-65578-9_7)
8. Fattebert, J.L., Wickett, M., Turchi, P.: Phase-field modeling of coring during solidification of au-ni alloy using quaternions and calphad input. *Acta Materialia* **62**, 89–104 (2014). <https://doi.org/10.1016/j.actamat.2013.09.036>

9. Bertolli, C., et al.: Coordinating GPU threads for OpenMP 4.0 in LLVM. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, 17 November 2014, pp. 12–21 (2014)
10. Bertolli, C., et al.: Integrating GPU support for OpenMP offloading directives into clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, 15 November 2015. pp. 5:1–5:11 (2015)
11. Özen, G., Atzeni, S., Wolfe, M., Southwell, A., Klimowicz, G.: OpenMP GPU Offload in Flang and LLVM. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 1–9 (2018)
12. Aycock, J.: A brief history of just-in-time. *ACM Comput. Surv.* **35**(2), 97–113 (2003)
13. The Khronos Group Inc.: SPIR Overview (2022). <https://www.khronos.org/spir/>
14. Peng, H., Shann, J.J.: Translating OpenACC to LLVM IR with SPIR kernels. In: 15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, 26–29 June 2016. pp. 1–6 (2016)
15. Ha, O., Kuh, I., Tchamgoue, G.M., Jun, Y.: On-the-fly detection of data races in OpenMP programs. In: Proceedings of the 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012, Minneapolis, MN, USA, 16 July 2012. pp. 1–10 (2012)
16. Luk, C., Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005. pp. 190–200 (2005)
17. Gaikwad, S., Nisbet, A., Luján, M.: Hosting OpenMP programs on Java virtual machines. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, 21–22 October 2019. pp. 63–71 (2019)
18. Glek, T., Hubicka, J.: Optimizing real world applications with GCC link time optimization. arXiv preprint [arXiv:1010.2196](https://arxiv.org/abs/1010.2196) (2010)
19. Murphy, M., Sundaram, A.: Improving GPU application performance with NVIDIA CUDA 11.2 device link time optimization, February 2021. <https://developer.nvidia.com/blog/improving-gpu-app-performance-with-cuda-11-2-device-lto/>
20. Antão, S.F., et al.: Offloading support for OpenMP in clang and LLVM. In: Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, 14 November 2016. pp. 1–11 (2016)
21. Doerfert, J., Diaz, J.M.M., Finkel, H.: The TRegion interface and compiler optimizations for OPENMP target regions. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 153–167. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-28596-8\\_11](https://doi.org/10.1007/978-3-030-28596-8_11)
22. Tiotto, E., Mahjour, B., Tsang, W., Xue, X., Islam, T., Chen, W.: OpenMP 4.5 Compiler optimization for GPU offloading. *IBM J. Res. Dev.* **64**(3/4), 14:1–14:11 (2020)
23. Doerfert, J., Patel, A., Huber, J., Tian, S., Diaz, J.M.M., Chapman, B., Georgakoudis, G.: 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, 15–19 May 2023. IEEE (2022)

24. Doerfert, J., et al.: 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)
25. Moses, W.S., Ivanov, I.R., Domke, J., Endo, T., Doerfert, J., Zinenko, O.: High-performance GPU-to-CPU transpilation and optimization via high-level parallel constructs (2022). <https://doi.org/10.48550/ARXIV.2207.00257>, <https://arxiv.org/abs/2207.00257>