Maximizing Parallelism and GPU Utilization For Direct GPU Compilation Through Ensemble Execution

Shilei Tian
shilei.tian@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

Barbara Chapman
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

Johannes Doerfert
jdoerfert@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA, USA

ABSTRACT

GPUs are renowned for their exceptional computational acceleration capabilities achieved through massive parallelism. However, utilizing GPUs for computation requires manual identification of code regions suitable for offloading, data transfer management, and synchronization. Recent advancements have capitalized on the LLVM/OpenMP portable target offloading interface, elevating GPU acceleration to new heights. This approach, known as the direct GPU compilation, involves compiling the entire host application for execution on the GPU, eliminating the need for explicit offloading directives. However, direct GPU compilation is limited to the thread parallelism a CPU application exposes, which is often not enough to saturate a modern GPU.

This paper explores an alternative approach to enhance parallelism by enabling ensemble execution. We introduce a proof-of-concept implementation that maps each invocation of an application on a different input to an individual team executed by the same GPU kernel. Our enhanced GPU loader can read command line arguments for different instances from a file to simplify the usability. Through extensive evaluation using four benchmarks, we observe up to 51X speedup for 64 instances. This demonstrate the effectiveness of ensemble execution in improving parallelism and optimizing GPU utilization for CPU programs compiled and executed directly on the GPU.

KEYWORDS

LLVM, OpenMP, accelerator offloading, GPU, ensemble execution

ACM Reference Format:
Shilei Tian, Barbara Chapman, and Johannes Doerfert. 2023. Maximizing Parallelism and GPU Utilization For Direct GPU Compilation Through Ensemble Execution. *ACM/IMS J. Data Sci.* 37, 4, Article 111 (August 2023), 7 pages. https://doi.org/10.1145/3605731.3606016

Authors' addresses: Shilei Tian, shilei.tian@stonybrook.edu, Stony Brook University, 100 Nicolls Road, Stony Brook, NY, 11794, USA; Barbara Chapman, barbara.chapman@stonybrook.edu, Stony Brook University, 100 Nicolls Road, Stony Brook, NY, 11794, USA; Johannes Doerfert, jdoerfert@llnl.gov, Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore, CA, 94550, USA.

1 INTRODUCTION

GPUs have gained widespread recognition for their remarkable ability to accelerate computations through massive parallelism. As a result, leveraging GPUs for accelerating various applications has become a prominent approach in high-performance computing [2]. However, achieving optimal GPU utilization requires careful identification of code regions that should be executed on the device, efficient memory transfers, and proper synchronization [9].

Recent advancements in compiler technology and programming frameworks have facilitated GPU acceleration by providing portable target offloading interfaces. One notable example is a recent work [26] that takes advantage of LLVM/OpenMP framework, which offers a flexible approach for offloading computations to GPUs. This approach, known as the direct GPU compilation scheme, involves compiling the *entire* host application for the GPU and executing it on the device, thereby eliminating the need for explicit offloading directives. This scheme simplifies the development process and enables transparent GPU acceleration.

However, the direct GPU compilation scheme has revealed significant performance limitations due to the insufficient parallelism of single-team execution, which is essential to ensure compliance with OpenMP semantics. This limitation hampers the ability to fully exploit the computational power of GPUs and achieve optimal performance. An extended work [27] has explored the approach of launching a parallel kernel when a parallel region is encountered if it is semantically allowed.

In contrast, ensemble-based simulations are extensively employed in high-performance computing to compute multiple individual simulation trajectories and analyze statistical properties across them [3, 4, 10, 12]. In this paper, we explore the concept of ensemble execution as a means to enhance parallelism and maximize GPU utilization. Ensemble execution involves running multiple instances of an application concurrently within the same GPU kernel launch. The primary goal of this study is to evaluate the effectiveness of ensemble execution in enhancing parallelism and GPU utilization. We present a proof-of-concept implementation that demonstrates the feasibility of mapping application instances to individual teams and provides an enhanced loader to handle command line arguments for different instances. We evaluate the performance using a set of benchmarks and analyze the scaling behavior under varying numbers of concurrent instances.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the direct GPU compilation scheme. Section 3 details the methodology and implementation of ensemble execution. Section 4 presents the evaluation results and discusses the observed performance trends. We review related works in Section 5. Finally, Section 6 concludes the paper and outlines future directions for enhancing ensemble execution.

2 BACKGROUND

OpenMP 4.0 introduced the `target` construct, which enables the execution of code regions on target devices like GPUs [6] and FPGAs [16]. To illustrate, Figure 1 depicts an example of CUDA code and its equivalent OpenMP version. Alongside the `target` construct, OpenMP provides the `declare target` directive, which specifies that all associated variables and functions should be mapped onto target devices, making them usable in device code [24]. Moreover, the `device_type(nohost)` clause on a `declare target` construct forces the compiler not to generate host versions of the enclosed variables and functions.

```
__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) Corresponding OpenMP code using target offloading to Figure 1a. Even though there is no explicit kernel specified by users, an OpenMP compiler will outline the target region and generate a kernel implicitly.

Figure 1: An example of a CUDA code and the corresponding OpenMP offload.

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. [26] 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 [27].

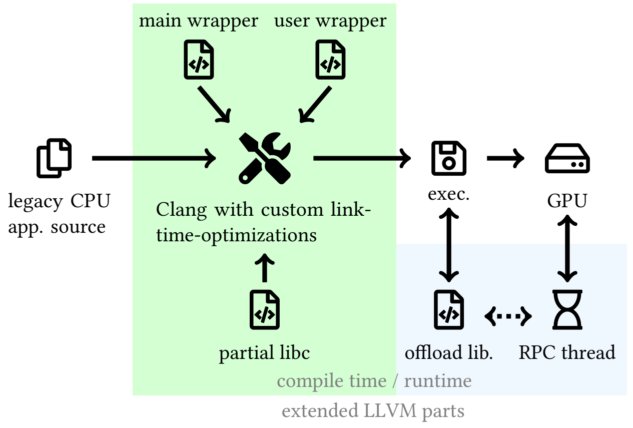


Figure 2: Overview of the compilation and execution path of the direct GPU compilation framework introduced by Tian et al. [26] and the extended work [27]. The figure is taken from the work [27].

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 and execution model.

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 Figure 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 (also depicted in Figure 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 Figure 3). In the extension work [23], the user's `main` function is canonicalized to the form `int main(int argc, char *argv[]);` and renamed accordingly. 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");
```

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

2.3 Single and Multiple Teams Execution

When a `target` region executes, it is executed by the initial thread sequentially. A `teams` construct is usually used to create a league of teams together with the `target` region where each team starts execution independently with its own initial thread. When a `parallel` construct is encountered, the enclosed region is executed by the encountering thread as well as the new threads associated with the respective team.

In scenarios where there are multiple teams (represented by N), the body of the construct will be executed by all N initial threads. However, in most cases, the user code, excluding the OpenMP parallel regions, is typically designed to run in a single-threaded manner. To maintain consistency with the behavior of host execution, the approach introduced by Tian et al. [26] supports only single team execution. In LLVM OpenMP, an OpenMP team is mapped to a thread block (or wavefront for AMD GPU). Each thread block has a limited number of threads that can be used, such as 1024 for NVIDIA GPUs. Therefore, the maximum number of threads that can be utilized is constrained by the total number available in the thread block. This limitation significantly impacts performance.

In the extension work [27], this limitation is addressed by launching a new kernel with multiple teams if the parallel region allows for it based on semantic considerations. By enabling multiple teams in parallel regions, the performance of these regions can be significantly improved.

3 IMPLEMENTATION

As discussed in Section 2.3, the restriction to single team execution significantly hinders performance. Although multi-team execution, as presented in [27], can improve performance, it may not be applicable in cases where the parallel region does not permit it semantically.

An alternative approach to fully utilize the capabilities of the GPU is to concurrently execute multiple instances of the application, each with a different input. This allows for parallel processing of independent tasks. In this section, we will introduce our enhanced loader, which builds upon the loader discussed in Section 2.2, to support ensemble execution. This enhancement enables the execution of multiple instances simultaneously, harnessing the full potential of the GPU for improved performance and efficiency.

3.1 Instance Mapping

Similar to previous work [26], we adopt a mapping strategy where each application instance is mapped to a team. In our enhanced loader, we utilize the `target teams distribute` construct to distribute the instances across multiple teams. The capability of our enhanced loader to support the invocation of multiple instances is demonstrated in Figure 4.

```
std::vector<std::string> StringCache;
std::vector<int> Argc;
std::vector<std::vector<char *>> Argv;
/* Construct arguments from argument file */
for (auto &Line : ArgumentFile) {
  Argv.emplace_back();
  auto &AV = Argv.back();
  AV.push_back(argv[0]);
  for (auto &Arg : Line.split(' ')) {
    StringCache.push_back(Arg);
    AV.push_back(StringCache.back().c_str());
  }
  Argc.push_back(AV.size());
}
#pragma omp target teams distribute num_teams(N) \
                    thread_limit(T) map(from:Ret[:NI])
for (int I = 0; I < NI; ++I) {
  Ret[I] = __user_main(Argc[I], &Argv[I][0]);
}
```

Figure 4: The invocation of `NI` instances of application with `N` teams, each of which can utilize up to `T` threads. `Argc[I]` and `Argv[I]` is the corresponding command line arguments of each instance.

The number of instances that can execute concurrently is limited by the number of teams available. However, there are alternative approaches to increase concurrency without introducing more teams. Both NVIDIA and AMD GPUs support three-dimensional thread blocks, while LLVM OpenMP currently uses only one dimension. By mapping `M` instances into a single team (thread block) at different dimensions, we can increase concurrency. In this mapping scheme, the size of the thread block becomes `(N/M,M,1)` when the thread limit is `N`. This approach allows for a reduction in the parallelism of each individual instance while improving overall concurrency. This mapping strategy is particularly beneficial for applications with limited parallelism. However, due to current limitations in the LLVM OpenMP implementation, this mapping scheme is not currently supported. As a result, we have not included it in our proof-of-concept implementation. Nonetheless, from a conceptual perspective, there are no difficulties in implementing this mapping scheme, and it can be explored in future enhancements of our approach.

3.2 Command Line Arguments

In previous works [23, 26, 27], the loader simply passed all command line arguments to the user's `main` function. However, in our research, we have extended the functionality of the loader to support ensembles of execution.

In our proof-of-concept implementation, the loader now accepts three command line arguments to enable ensemble execution:

- `-f <file>`: This argument specifies the command line arguments file. Each line in the file contains the arguments for each application instance.
- `-n <num instances>`: This argument specifies the number of instances to be launched simultaneously.
- `-t <thread limit>`: This argument specifies the maximum number of threads that each instance can utilize. However, the actual number of threads used by each instance may be lower due to hardware resource limitations.

An example of using our GPU ensembler to run four instances of a user application concurrently on a GPU is illustrated in Figure 5. This new capability allows for efficient parallel execution of multiple application instances, maximizing GPU utilization and potentially improving overall performance.

```
$ ./user_app_host -a 1 -b -c data-1.bin
$ ./user_app_host -a 2 -b -c data-2.bin
$ ./user_app_host -a 1 -b -c data-3.bin
$ ./user_app_host -a 3 -b -c data-4.bin
```

(a) An example of running the original user application four times on the host with different command line arguments.

```
-a 1 -b -c data-1.bin
-a 2 -b -c data-2.bin
-a 1 -b -c data-3.bin
-a 3 -b -c data-4.bin
```

(b) The content of command line argument file `arguments.txt`.

```
$ ./user_app_gpu -f arguments.txt -n 4 -t 128
```

(c) Executing the GPU version of the user application using the enhanced loader, where we simultaneously launch four application instances capable of utilizing up to 128 threads each.

Figure 5: An example of ensembling execution.

In our future work, we have plans to design a script language specifically for the command line argument file. This script language will enable the generation of command line arguments for each instance dynamically, providing greater flexibility.

3.3 Limitation

Running multiple instances of an application within the same kernel launch can pose challenges to maintaining the natural isolation between instances. This can be particularly problematic when shared global variables are involved, as it can introduce the possibility of data races and compromise the correctness of the application. To address this issue, a possible solution is to develop a compiler transformation that relocates global variables to shared memory, which is team-local on a GPU.

4 EVALUATION

In this section, we present the evaluation of our approach. We begin by introducing the benchmarks that were used in our experiments. Next, we provide details about the experimental configuration, including the hardware and software setup. Finally, we present the results of our evaluation and provide an analysis of the findings.

4.1 Benchmarks

We conducted our evaluation using two proxy applications, XSBench [29] and RSBench [28], as well as two microbenchmarks from the HeCBench suite [13].

XSBench and **RSBench** serve as proxies for the Open Monte Carlo (OpenMC) project, specifically focusing on the computation of continuous energy macroscopic neutron cross-section lookup in neutron transport simulations. XSBench represents a memory-bound kernel from the OpenMC project, while RSBench provides an alternative implementation that is compute-bound.

Additionally, we selected two benchmarks, **AMGmk** and **Page-Rank**, from **HeCBench**, a GitHub repository housing a collection of heterogeneous computing benchmarks, for our evaluation. AMGmk measures the performance of the relax kernel from the original AMGmk proxy application [14]. This benchmark focuses solely on the relax kernel's execution. Page-Rank, on the other hand, implements the page-rank algorithm for graphs, specifically measuring the propagation step of the algorithm.

4.2 Configuration

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 all benchmarks with `-O3`.

We performed runs using different numbers of instances for each benchmark: 1, 2, 4, 8, 16, 32, and 64 instances. The number of teams was set equal to the number of instances, ensuring that each team executed a single instance. Due to memory limitations, we did not utilize a larger number of instances in the experiment. We selected two thread limits: 32 and 1024. The limit of 32 corresponds to the size of a warp, which is the smallest unit of the hardware scheduler. The limit of 1024 represents the maximum number of threads that can be utilized by the hardware. It is important to note that the thread limit serves as an upper bound for the number of threads a kernel can utilize. In practice, a kernel may not be able to fully utilize the maximum thread count. Therefore, when we refer to 1024 as the thread limit, it implies that the kernel can utilize as many threads as it can effectively use, while 32 represents the minimum number of threads a kernel can utilize.

4.3 Results and Analysis

Figure 6 illustrates the relative speedup of each benchmark with varying numbers of instances and different thread limits. The speedup is computed using the formula $T_1 \times N/T_N$, where T_1 represents the time taken for executing a single instance, N denotes the number of instances, and T_N represents the time taken for executing N

instances concurrently. The line "Linear" in the figure corresponds to the upper bound of the speedup. It indicates that if we have N instances executing simultaneously, each instance would have the same execution time as when only a single instance is running. In other words, the speedup would be perfectly linear if the execution time scales linearly with the number of instances.

As observed in the results, all the benchmarks exhibited a "sub-linear" scaling behavior, particularly evident when the number of instances was 16 or less. As the number of instances increased, the scaling gap became more pronounced, particularly notable in the case of AMGmk with a thread limit of 1024. When executing parallel regions, global memory access within a single thread block tends to be coalesced due to the contiguous nature of the accesses. However, unlike the case of a common kernel execution, in our GPU assembling execution, threads from different thread blocks are unlikely to exhibit coalesced memory accesses since they process data allocated in different heap allocations, which are typically non-contiguous. This scenario does not maximize the utilization of the global memory bandwidth. Due to memory limitations, we were only able to show the results for two and four instances in the case of Page-Rank.

5 RELATED WORKS

Previous research has delved into the execution of host programs on GPUs, exploring various techniques and optimizations. Pakin et al. [21] introduced a reverse-acceleration model where accelerators orchestrate computations, offloading non-acceleratable work to general-purpose processors. Jablin et al. [11] proposed a fully automatic system for managing and optimizing CPU-GPU communication, encompassing a runtime library and compiler transformations. Silberstein et al. [22] proposed direct access to the host's file system from GPU code and implemented an RPC protocol for CPU-GPU data transfers. Mikushin et al. [17] presented a parallelization framework that detects parallelism and generates target code for both X86 CPUs and NVIDIA GPUs, employing a foreign function interface for executing functions on the host. Damschen et al. [7] investigated transparent acceleration of binary applications using heterogeneous computing resources without manual porting or developer-provided hints. Noack et al. [18] discussed the built-in reverse-offloading mechanism in the low-level Vector Engine Offloading library. Matsumura et al. [15] introduced an automated stencil framework that transforms and optimizes stencil patterns in C source code to generate corresponding CUDA code. Tian et al. [26] explored running the entire host program on a GPU using OpenMP target offloading, augmenting the compiler with a custom link-time optimization pass to generate RPC calls automatically and expand source parallelism to the GPU device. Subsequent work [27] extended this approach in the form of compiler enhancements and source parallelism expansion without requiring stub code from users. Tian et al. [23] explored the limit of executing generic code on GPUs using the direct GPU compilation scheme.

Several notable works have explored the practical applications and advancements in ensemble execution. Jiang et al. [12] developed a pulling-based workflow execution system tailored for efficient execution of large-scale scientific workflow ensembles in public cloud environments, specifically Amazon EC2. Balasubramanian et al. [4] introduced the ensemble toolkit, a comprehensive framework designed to facilitate the dynamic and efficient execution of ensembles on heterogeneous computing resources. Balasubramanian et al. [3] implemented a scalable and adaptive ensemble execution system on top of the ensemble toolkit.

Additionally, researchers have focused on compiler and runtime optimization for OpenMP after the introduction of target offloading in OpenMP 4.0. Bertolli et al. [5, 6] enabled OpenMP offloading to GPUs in LLVM, while Flang, the PGI Fortran front-end, supports OpenMP offloading through the LLVM OpenMP runtime [19]. Antão et al. [1] introduced front-end-based optimizations for NVIDIA GPUs, reducing register usage and minimizing idle threads. Doerfert et al. [8] presented the TRegion interface to enable more kernels to execute in SPMD mode. Tian et al. [25] introduced runtime support for concurrent execution of OpenMP target tasks. Yviquel et al. [30] developed a framework for using the OpenMP programming model in distributed memory environments, combining OpenMP directives and MPI communication. Huber et al. [9] developed OpenMP-aware program analyses and optimizations for efficient execution of CPU-centric parallelism on GPUs. Ozen and Wolfe [20] demonstrated the implementation of the `loop` directive on NVIDIA GPUs in NVIDIA's compiler.

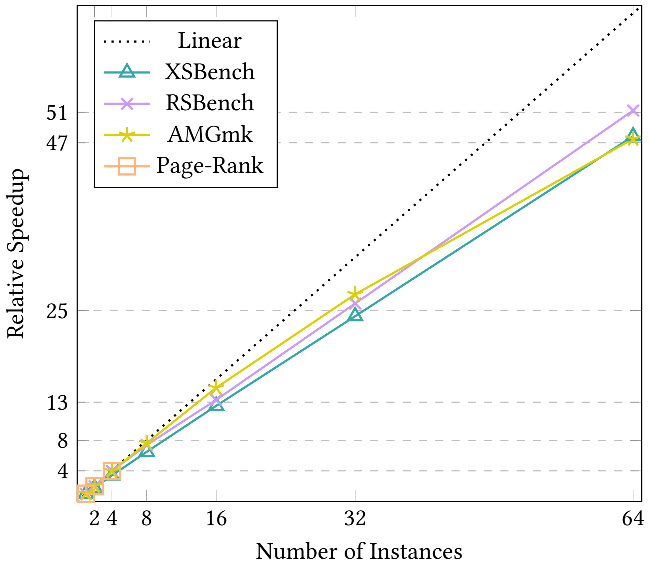
6 CONCLUSION AND FUTURE WORK

In this study, we have explored ensemble execution as a means to enhance parallelism and maximize GPU utilization. By mapping each instance of an application to an individual team and leveraging an enhanced loader capable of handling command line arguments for different instances, we have demonstrated the effectiveness of ensemble execution in increasing parallelism. Through our proof-of-concept implementation and evaluation of four benchmarks, we observed up to 51X speedup for 64 instances. These results highlight the potential of ensemble execution to improve parallelism and exploit the capabilities of modern GPUs.

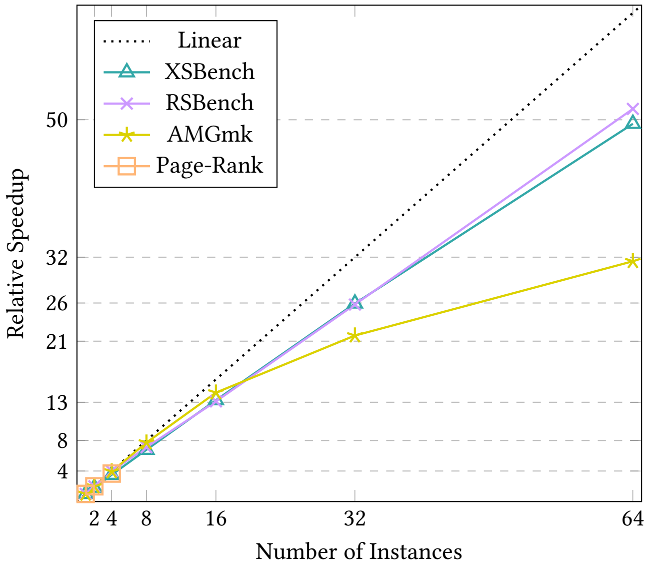
Looking ahead, our future research will focus on further optimizing the mapping of application instances to achieve even better scalability. We will explore advanced mapping strategies and investigate techniques to dynamically generate command line arguments using a script language, enabling more flexibility and convenience in ensemble execution.

ACKNOWLEDGMENTS

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



(a) Thread limit: 32



(b) Thread limit: 1024

Figure 6: The relative speedup of the benchmarks with different number of instances and two thread limits.

in parts prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-851058). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory.

REFERENCES

[1] Samuel F. Antão, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 14, 2016.* IEEE Computer Society, Salt Lake City, UT, USA, 1–11. https://doi.org/10.1109/LLVM-HPC.2016.006

[2] Seonmyeong Bak, Colleen Bertoni, Swen Boehm, Reuben D. Budiardja, Barbara M. Chapman, Johannes Doerfert, Markus Eisenbach, Hal Finkel, Oscar R. Hernandez, Joseph Huber, Shintaro Iwasaki, Vivek Kale, Paul R. C. Kent, JaeHyuk Kwack, Meifeng Lin, Piotr Luszczek, Ye Luo, Buu Pham, Swaroop Pophale, Kiran Ravikumar, Vivek Sarkar, Thomas Scogland, Shilei Tian, and P. K. Yeung. 2022. OpenMP Application Experiences: Porting to Accelerated Nodes. *Parallel Comput.* 109 (2022), 102856. https://doi.org/10.1016/j.parco.2021.102856

[3] Vivek Balasubramanian, Travis Jensen, Matteo Turilli, Peter M. Kasson, Michael R. Shirts, and Shantenu Jha. 2020. Adaptive Ensemble Biomolecular Applications at Scale. *SN Computer Science* 1, 2 (2020), 104. https://doi.org/10.1007/s42979-020-0081-1

[4] Vivekanandan Balasubramanian, Antons Treikalis, Ole Weidner, and Shantenu Jha. 2016. Ensemble Toolkit: Scalable and Flexible Execution of Ensembles of Tasks. In *International Conference on Parallel Processing (ICPP), August 16-19, 2016.* IEEE, Philadelphia, PA, USA, 458–463. https://doi.org/10.1109/ICPP.2016.59

[5] Carlo Bertolli, Samuel Antão, Gheorghe-Teodor Bercea, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O'Brien. 2015. Integrating GPU support for OpenMP offloading directives into Clang. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 15, 2015.* ACM, Austin, Texas, USA, 5:1–5:11. https://doi.org/10.1145/2833157.2833161

[6] Carlo Bertolli, Samuel Antão, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 17, 2014.* IEEE Computer Society, New Orleans, LA, USA, 12–21. https://doi.org/10.1109/LLVM-HPC.2014.10

[7] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. 2015. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Design, Automation & Test in Europe Conference & Exhibition (DATE) March 9-13, 2015.* ACM, Grenoble, France, 1078–1083. http://dl.acm.org/citation.cfm?id=2757063

[8] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. 2019. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *International Workshop on OpenMP (IWOMP), September 11-13, 2019*, Vol. 11718. Springer, Auckland, New Zealand, 153–167. https://doi.org/10.1007/978-3-030-28596-8_11

[9] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose Manuel Monsalve Diaz, Kuter Dinel, Barbara M. Chapman, and Johannes Doerfert. 2022. Efficient Execution of OpenMP on GPUs. In *International Symposium on Code Generation and Optimization (CGO), April 2-6, 2022.* IEEE, Seoul, Republic of Korea, 41–52. https://doi.org/10.1109/CGO53902.2022.9741290

[10] Stephen Hudson, Jeffrey Larson, John-Luke Navarro, and Stefan M. Wild. 2022. libEnsemble: A Library to Coordinate the Concurrent Evaluation of Dynamic Ensembles of Calculations. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 977–988. https://doi.org/10.1109/TPDS.2021.3082815

[11] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU communication management and optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 4-8, 2011.* ACM, San Jose, CA, USA, 142–151. https://doi.org/10.1145/1993498.1993516

[12] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. 2015. Executing Large Scale Scientific Workflow Ensembles in Public Clouds. In *International Conference on Parallel Processing (ICPP), September 1-4, 2015.* IEEE, Beijing, China, 520–529. https://doi.org/10.1109/ICPP.2015.61

[13] Zheming Jin. 2023. HeCBench. https://github.com/zjin-lcf/HeCBench

[14] Lawrence Livermore National Laboratory. 2023. CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/

[15] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs. In *International Symposium on Code Generation and Optimization (CGO), February, 2020.* ACM, San Diego, CA, USA, 199–211. https://doi.org/10.1145/3368826.3377904

[16] Florian Mayer, Marius Knaust, and Michael Philippsen. 2019. OpenMP on FPGAs - A Survey. In *International Workshop on OpenMP (IWOMP), September 11-13, 2019*, Vol. 11718. Springer, Auckland, New Zealand, 94–108. https://doi.org/10.1007/978-3-030-28596-8_7

[17] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, and Christopher Bergstrom. 2014. 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.* IEEE Computer Society, Phoenix, AZ, USA, 1011–1020. https:

//doi.org/10.1109/IPDPSW.2014.115

[18] Matthias Noack, Erich Focht, and Thomas Steinke. 2019. Heterogeneous Active Messages for Offloading on the NEC SX-Aurora TSUBASA. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 20-24, 2019*. IEEE, Rio de Janeiro, Brazil, 26–35. https://doi.org/10.1109/IPDPSW.2019.00014

[19] Güray Özen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. 2018. OpenMP GPU Offload in Flang and LLVM. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC@SC), November 13, 2018*. IEEE, Dallas, TX, USA, 1–9. https://doi.org/10.1109/LLVM-HPC.2018.8639434

[20] Guray Ozen and Michael Wolfe. 2022. Performant Portable OpenMP. In *ACM SIGPLAN International Conference on Compiler Construction (CC), April 2 - 3, 2022*. ACM, Seoul, South Korea, 156–168. https://doi.org/10.1145/3497776.3517780

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

[22] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating A File System with GPUs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 16-20, 2013*. ACM, Houston, TX, USA, 485–498. https://doi.org/10.1145/2451116.2451169

[23] Shilei Tian, Barbara Chapman, and Johannes Doerfert. 2023. Exploring the Limits of Generic Code Execution on GPUs via Direct (OpenMP) Offload. In *Under Review*.

[24] Shilei Tian, Jon Chesterfield, Johannes Doerfert, and Barbara M. Chapman. 2021. Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1. In *International Workshop on OpenMP (IWOMP), September 14-16, 2021*, Vol. 12870. Springer, Bristol, UK, 159–169. https://doi.org/10.1007/978-3-030-85262-7_11

[25] Shilei Tian, Johannes Doerfert, and Barbara M. Chapman. 2020. 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. Springer, Stony Brook, NY, USA, 41–56. https://doi.org/10.1007/978-3-030-95953-1_4

[26] Shilei Tian, Joseph Huber, Konstantinos Parasyris, Barbara M. Chapman, and Johannes Doerfert. 2022. 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*. IEEE, Dallas, TX, USA, 43–51. https://doi.org/10.1109/LLVM-HPC56686.2022.00010

[27] Shilei Tian, Tom Scogland, Barbara Chapman, and Johannes Doerfert. 2023. GPU First – Execution of Legacy CPU Codes on GPUs. arXiv:2306.11686 [cs.DC]

[28] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *International Conference on Exascale Applications and Software (EASC), April 2-3, 2014*, Vol. 8759. Springer, Stockholm, Sweden, 39–56. https://doi.org/10.1007/978-3-319-15976-8_3

[29] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *International Conference on Physics of Reactors (PHYSOR), September 28 - October 3, 2014*. JAEA, Kyoto, Japan, 1–12. http://dx.doi.org/10.11484/jaea-conf-2014-003

[30] Hervé Yviquel, Marcio Pereira, Emilio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Henrique Di Francia Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. 2022. The OpenMP Cluster Programming Model. In *Workshop of the International Conference on Parallel Processing (ICPP), 29 August 2022 - 1 September 2022*. ACM, Bordeaux, France, 17:1–17:11. https://doi.org/10.1145/3547276.3548444