Evaluating LLVM OpenMP Offload Optimizations on NVIDIA GH200 Grace Hopper Superchip and AMD Instinct™ MI300A Accelerator Architectures

Kevin Sala[1], Stephen L. Olivier[2], Rahulkumar Gayatri[3], Shilei Tian[4], and Johannes Doerfert[1]

[1] Lawrence Livermore National Laboratory, Livermore, CA, USA
salapenades1@llnl.gov, jdoerfert@llnl.gov
[2] Sandia National Laboratories, Albuquerque, NM, USA
slolivi@sandia.gov
[3] Lawrence Berkeley National Laboratory, Berkeley, CA, USA
rgayatri@lbl.gov
[4] Advanced Micro Devices, Boxborough, MA, USA
shilei.tian@amd.com

Abstract. Wide cross-vendor support for the OpenMP API makes it appealing for portable parallel programming on CPUs and GPUs. While its use on CPUs is well-established, perceived or actual performance gaps relative to native APIs like CUDA and HIP have limited its adoption on GPUs. Previous work has shown that for simple parallel patterns such as single level loops, recent OpenMP implementations can be competitive in terms of performance against those native APIs. For more complex use cases, such as hierarchical parallelism, extensions to OpenMP have been necessary to attain the performance of those native APIs on discrete GPUs like the AMD Instinct™ MI250X and NVIDIA A100. In this paper, we evaluate and extend the use of these extensions in the Kokkos C++ performance portability framework on newer converged CPU-GPU architectures, the AMD Instinct™ MI300A and NVIDIA GH200. Our results show that the extensions help bridge performance gaps between native API and OpenMP in these architectures, and we identify areas that still have room for improvement.

Keywords: GPU · Kokkos · LLVM · OpenMP

1 Introduction

Support for accelerator programming in the OpenMP® API dates back over a decade to version 4.0, having been refined in subsequent versions of the specification. Yet while OpenMP remains widely used on CPU architectures, its use on GPUs remains limited. One contributing factor is that OpenMP's accelerator support has acquired a reputation for lower performance compared to native vendor-specific programming models, i.e., CUDA and HIP. As shown in a recent study comparing OpenMP with those models, recent OpenMP implementations

can exhibit commensurate performance for simple kernels of one-level loops, but they struggle with more complex patterns such as hierarchical parallelism [7]. Additionally, native models better exploit features such as shuffle operations and the shared memory local to subunits of the GPU. Moreover, the burden of providing library support for full OpenMP semantics on the GPU adds overhead cost compared to the simpler grid-based approach of native models.

The feature and performance gaps between OpenMP accelerator support and native models have motivated a series of extensions in the LLVM® OpenMP implementation. These include runtime calls to access local dynamic shared memory, to leverage shuffle operations for faster reductions, and to express parallelism in a grid-based "kernel mode" similar to HIP and CUDA [17]. A previous evaluation of these extensions as applied to the OpenMPTarget backend code of the Kokkos C++ performance portability library [18] showed how they help to close the gap with native models [8].

As the OpenMP API and its implementations evolve to better support accelerators, the devices themselves are also evolving. Beyond improvements in processing power and memory bandwidth, some vendors have also introduced converged architectural designs that integrate CPU and GPU components, such as the NVIDIA Grace Hopper Superchip [5, 3] and AMD Instinct™ MI300A Accelerator [14, 15]. In this paper, we explore the impact of LLVM's OpenMP extensions on these newer architectures. Moreover, we are able to use the release version of LLVM Clang in this study, as all extensions used in the prior paper, including kernel mode, have now been incorporated into the release series. Once again, the Kokkos library is the vehicle for comparison, allowing the same application level code to be transformed via C++ template metaprogramming to use equivalent native (HIP/CUDA) and OpenMPTarget backend implementations of Kokkos. Beyond the enhancements of the previous study [8], we also expand the use of the extensions in the Kokkos-OpenMPTarget backend to additional parallel patterns, and we present performance results for the newer and older GPU architectures.

2 Background

This paper uses the Kokkos library and its OpenMPTarget backend implementation as a vehicle to evaluate performance of LLVM OpenMP extensions on recent converged CPU+GPU architectures compared to previous generation discrete GPUs. In this section we provide basic descriptions of Kokkos, its OpenMPTarget backend, and the LLVM OpenMP extensions. More details on these topics can be found in previous work [8].

2.1 Kokkos and its OpenMPTarget Backend

Kokkos is a performance portability library that enables a single C++ codebase to run efficiently across a wide range of GPUs and CPUs [18]. It is widely adopted

in scientific applications, particularly those developed under the United States Exascale Computing Project (ECP), as a preferred programming model.

For a given device, the Kokkos library maps to a backend implementation that supports that device. Kokkos provides native backends for all three major server-class GPU vendors: NVIDIA, AMD, and Intel, as well as an OpenMP CPU backend. The OpenMPTarget backend is considered a secondary option for GPU support, serving several purposes: 1) risk mitigation, 2) preparation for future hardware that may adopt OpenMP as its primary programming model, and 3) interoperability with third party libraries using OpenMP GPU offload. For the purposes of evaluating new OpenMP features, the Kokkos OpenMPTarget backend provides a convenient way to test OpenMP features using the same code against other programming model backends, such as HIP and CUDA.

2.2 LLVM OpenMP Extensions

In addition to the standard features included in the OpenMP specification, OpenMP implementations can offer additional constructs, clauses, or runtime routines as extensions, prefixed by `ompx`. LLVM OpenMP includes several such extensions to help leverage GPU hardware in a way similar to native GPU programming models.

Dynamic Shared Memory Support. One limitation of OpenMP as currently specified is its inability to expose dynamically allocated memory as "shared" among threads within an OpenMP team. Native GPU programming models, such as CUDA and HIP, usually refer to this feature as dynamic shared memory. Although OpenMP 6.0 recently brought support for sharing variables (with static storage duration) among the threads of a team, it still lacks support for sharing a dynamically sized memory buffer across team members. LLVM OpenMP introduces an extension to address this limitation. Specifically, it adds a new clause for the `target` directive, `ompx_dyn_cgroup_mem(<N>)`, which provides a memory buffer of `N` bytes per team that will be shared among the threads within each team. In this way, OpenMP exposes a similar mechanism to that of CUDA or HIP when requesting dynamic shared memory at kernel launch. Within the target region, the `llvm_omp_target_dynamic_shared_alloc` routine can be used to obtain a pointer to this shared memory buffer.

Kokkos exposes dynamic shared memory through what it refers to as scratch memory, i.e., views created in its `scratch_memory_space`. Leveraging that LLVM extension enables access to dynamic shared memory in the OpenMPTarget backend, whereas previously, such buffers had to be allocated in the slower, more distant global memory.

Notably, OpenMP has recently accepted a new clause and routine that enable access to dynamic shared memory within target regions. This feature addition is expected to be included in the upcoming OpenMP 6.1 specification.

LLVM OpenMP Kernel Mode. While existing OpenMP provides a rich set of parallel semantics, including a fork-join model and automatic workload distribution, it relies on a substantial device runtime library to manage execution. These runtime operations can introduce significant overhead and resource usage, particularly on GPUs. This overhead has traditionally been considered unavoidable under the current OpenMP semantics [2]. To address this, LLVM OpenMP introduces a set of extensions that allow OpenMP target regions to execute in a "bare-metal" mode, also known as *kernel mode* [17]. This feature enables OpenMP GPU code to be written in a single-instruction, multiple-threads (SIMT) style, facilitating the transition of existing GPU code written in kernel languages such as HIP and CUDA to OpenMP, while leveraging OpenMP's portability benefits. Additionally, OpenMP kernel mode requires only a minimal runtime layer, significantly reducing overhead and potentially improving performance.

To that end, LLVM OpenMP provides the `ompx_bare` clause for the `target teams` construct, which enables kernel mode for that `target` region [17]. This clause configures the region to operate according to the SIMT model and prevents the generation of code related to the OpenMP device runtime. Additionally, the clause enables support for multidimensional OpenMP teams and leagues, similar to how CUDA and HIP operate with blocks and grids. The `num_teams` and `thread_limit` clauses are augmented to accept multidimensional sizes in the form of a list, and new routines are provided to retrieve the multidimensional indexes and sizes within the `target` regions. Fig. 3 shows an example of how to use the `ompx_bare` clause in a `target` region. Notice that the `target` region does not use any explicit `parallel` construct, as LLVM's `ompx_bare` clause implicitly initiates parallelism within each team.

With the addition of new extension routines for portable synchronization and shuffle operations in LLVM OpenMP, it becomes possible to efficiently implement multidimensional reductions. By relying on these capabilities, the Kokkos OpenMPTarget backend implementation can logically map parallel patterns like hierarchical parallelism in a manner similar to the CUDA and HIP backends with fewer overheads compared to the previous implementation based on `teams distribute parallel for`.

2.3 Exemplar Application Benchmarks

We use applications representing two different domains as exemplar workloads in our study. The first, CGSolve, implements a conjugate gradient solver both in Kokkos, enabling comparison of the various Kokkos backends (native HIP/CUDA and OpenMPTarget), and directly in OpenMP. In the evaluation, we used sparse matrices of size 150x150, 255x255, and 325x325, resulting in data sizes of 2GB, 11GB, and 22GB, respectively, as represented in compressed sparse row format. The program performs 200 iterations of CGSolve. For this application, the direct OpenMP implementation allows assessment of instances where the combination of OpenMP and C++ abstractions in Kokkos may be handled sub-optimally by the compiler. We focus on the `spmv` kernel, which dominates execution time. It

exhibits hierarchical parallelism that has been challenging to perform efficiently using OpenMP on GPUs [7, 8].

The TestSNAP proxy application [6] is modeled on Spectral Neighborhood Analysis Potential (SNAP) computations in the LAMMPS molecular dynamics simulation application. Our test problem simulates 2000 atoms with 26 neighbors per atom over 100 timesteps. We report performance on the three kernels that account for nearly all total execution time, two of which (`ui` and `duarray`) can exploit dynamic shared memory and hierarchical parallelism in programming models that support them well [8].

3 Applying OpenMP Extensions to Multidimensional Kokkos Parallel Patterns

The Kokkos library provides an extensive set of parallel execution patterns designed to efficiently exploit concurrency in applications. In our previous work, we investigated the feasibility of implementing hierarchical parallelism within the Kokkos-OpenMPTarget backend by leveraging LLVM OpenMP extensions. Our earlier study specifically targeted complex parallel patterns, aiming to demonstrate the potential for expressing CUDA/HIP-style hierarchical parallelism directly through OpenMP extensions. Detailed explanations of how the LLVM extensions are used in the backend can be found therein [8], and code for the benchmark applications is available online[5].

We intentionally excluded simpler Kokkos parallel patterns from our previous investigation, as they typically map directly onto existing OpenMP directives. For instance, a one-dimensional Kokkos parallel range pattern corresponds directly to the OpenMP directive `#pragma omp target teams distribute parallel for`, while multidimensional parallelism, represented by the Kokkos MDRange pattern, can be implemented using OpenMP's `collapse` clause.

In the workloads considered in our study, the TestSNAP application employs the MDRange parallel pattern for its most computationally intensive kernel, `compute_yi`. Although our previous enhancements to the Kokkos-OpenMPTarget backend indirectly improved performance for TestSNAP, the kernel execution remained significantly slower compared to native Kokkos backends. In this paper, we directly implement the MDRange parallel pattern using LLVM's OpenMP extensions to further address these performance limitations.

Fig. 1 illustrates the use of MDRange parallelism within the Kokkos programming model. All parallel patterns in Kokkos assume iterations in a pattern to be independent. Hence the current Kokkos-OpenMPTarget backend implements the MDRange policy described above using the OpenMP `collapse` clause. Fig. 2 shows a pseudocode representation of this implementation.

However, this implementation does not achieve performance comparable to native Kokkos backends. To address this performance gap, we leveraged the LLVM OpenMP extensions, enabling an implementation of the collapse functionality

[5] https://github.com/kokkos/code-examples/tree/HiPC2024/papers/HiPC-2024

```
// Creates a multidimensional iteration space of 3 levels.
// This will be created in the default execution space of Kokkos.
// The iteration space can be passed as parameters to the policy.
Kokkos::MDRangePolicy<Rank<3>> policy({begin_0,begin_1,begin_2},{end_0,
     end_1,end_2});

// Creates a 3-level nested loop with the outermost iteration index
// starting with i and then j and k indexes for the subsequent
// iteration spaces.
Kokkos::parallel_for(policy, KOKKOS_LAMBDA ( const int i, const int j,
      const int k ) {
 ...
});
```

Fig. 1: MDRange parallel pattern in Kokkos.

```
#pragma omp target teams distribute parallel for collapse(3)
for (int i = begin_0; i < end_0; ++i)
  for (int j = begin_1; j < end_1; ++j)
    for (int k = begin_2; k < end_2; ++k)
    {
      ...
    }
```

Fig. 2: MDRange implementation in the current Kokkos-OpenMPTarget backend.

analogous to that of HIP/CUDA. Fig. 3 presents our implementation of the three-level nested loop MDRange parallel pattern using LLVM OpenMP extensions.

The current implementation employs a straightforward approach, flattening the three nested loops into a single iteration space equal to the product of the loop ranges, as illustrated in line 2 of Fig. 3. We use teams consisting of 128 threads, a configuration empirically determined to yield good performance across all architectures in our benchmarks. Increasing team size further slightly improves performance on AMD devices but degrades performance on NVIDIA devices. Future studies may require tuning this parameter according to specific architectural characteristics and kernel properties.

Line 5 computes the number of teams needed based on the selected team size, carefully accounting for cases where the total iteration count is not evenly divisible by the team size. Line 7 initiates the kernel execution with the calculated grid dimensions. Lines 18–21 demonstrate how the original multidimensional indices are recovered from the flattened iteration index.

As noted previously, the `compute_yi` kernel in TestSNAP utilizes the MDRange parallel policy. Section 5.2 discusses the resulting performance of the kernel implementation presented in Fig. 3.

```
// Total elements
const int n = (end_0-begin_0) * (end_1-begin_1) * (end_2-begin_2);
if (!n) return; // Return if nothing needs to be done
const int team_size = 128;
int nteams = n / team_size + !!(n % team_size);

#pragma omp target teams ompx_bare num_teams(nteams) thread_limit(
    team_size)
{
   const Index blockIdx = ompx::block_id(ompx::dim_x);
   const Index blockDimx = ompx::block_dim(ompx::dim_x);
   const Index threadIdx = ompx::thread_id(ompx::dim_x);
   const Index tid = blockIdx * blockDimx + threadIdx;

   if (tid < n)
   {
       Index iter_ = tid;

       const int i = iter_ / (end_2*end_1) + begin_0;
       const int i1_ = iter_ % (end_2*end_1);
       const Index j = i1_ / end_2 + begin_1;
       const Index k = i1_ % end_2 + begin_2;

       ...
   }
}
```

Fig. 3: MDRange implementation using LLVM OpenMP extensions in Kokkos-OpenMPTarget.

4 Experimental Setup

The evaluation measures the performance of the Kokkos OpenMPTarget backend with and without the LLVM OpenMP extensions, comparing with native (HIP/CUDA) Kokkos backends, across different GPU architectures. Establishing a notion of overall equivalency across the GPU lines of different vendors can be challenging, especially as optimization targets for hardware designs have shifted to specialize for the growing market of machine learning applications. We therefore decline to do so, but nonetheless we provide some relevant hardware details for reference in Table 1.

On the software side, we used the LLVM Clang compiler, release version 20.1.1, which supports all OpenMP extensions described in Section 2. We used this OpenMP version in the optimized Kokkos OpenMPTarget backend. On AMD platforms, we used ROCm™ 6.4.0 on both MI250X and MI300A. On NVIDIA platforms, CUDA 12.6 was used for A100 and CUDA 12.5 for GH200 GPUs.

Table 1: GPU models used in this study. The numbers of NVIDIA streaming multiprocessors (SMs) and AMD compute units (CUs) are provided for comparison among GPUs of the same vendor but are not 1:1 equivalent and should not be used to compare processing capability across vendors.

Processor	GPU-CPU Connectivity	Peak HBM BW (TB/s)	HBM Size (GB)	Processing Elements	GPU Generation	Peak FP32/FP64 TFLOPs
NVIDIA A100	Discrete	1.6	40	108 SMs	Ampere	19.5 / 9.7
NVIDIA GH200	Integrated	4	96	132 SMs	Hopper	67 / 34
AMD MI250X	Discrete	3.2	128	220 CUs	CDNA 2	47.9 / 47.9
AMD MI300A	Integrated	5.3	128	228 CUs	CDNA 3	122.6 / 61.3

5 Results

In this section, performance results from prior generation discrete GPUs (A100 and MI250X) and more recent converged GPUs (GH200 and MI300A) shed light on the ongoing benefits of LLVM OpenMP extensions for grid-style kernel mode execution and dynamic GPU shared memory.

5.1 CGSolve

Fig. 4 shows the performance of the `spmv` kernel from CGSolve. *KK-Native* represents Kokkos using the vendor programming model and compiler, i.e., CUDA with `nvcc` on NVIDIA and HIP with `hipcc` on AMD. *KK-Native Clang* represents Kokkos using the vendor programming model (CUDA or HIP) compiled using Clang. *KK-OMPT* represents Kokkos using the OpenMPTarget backend without LLVM extensions. *KK-OMPT Kernel* represents Kokkos using the OpenMPTarget backend with LLVM extensions discussed in Section 2.2. *OMPT* represents direct OpenMP target code without LLVM extensions. *OMPT Kernel* represents direct OpenMP target code with LLVM extensions.

Since SpMV is a memory-bound computation, performance is often reported in terms of the memory bandwidth achieved. Here it is normalized to the bandwidth achieved by *KK-Native*; higher is better. First, there is little difference between the results of the vendor compilers and Clang using the CUDA and HIP Kokkos backends (on NVIDIA and AMD, respectively).

Next, consider the performance of the Kokkos OpenMPTarget backend with and without the LLVM kernel mode extensions. Without the LLVM extensions, the Kokkos OpenMPTarget backend reaches 52-54% of native performance on the older A100 GPU but only 29-31% of native performance on GH200. Using the extensions in the Kokkos OpenMPTarget backend improves performance to 91-95% of native on both devices. For direct OpenMP code without the LLVM extensions, performance reaches 64-66% of native performance on the older A100 GPU but only 40% of native performance on GH200. On both devices, direct OpenMP code with the extensions meets or exceeds native performance.

On the AMD devices, performance without the LLVM kernel mode extensions is uniformly poor, 12-16% of the native backend performance with the Kokkos

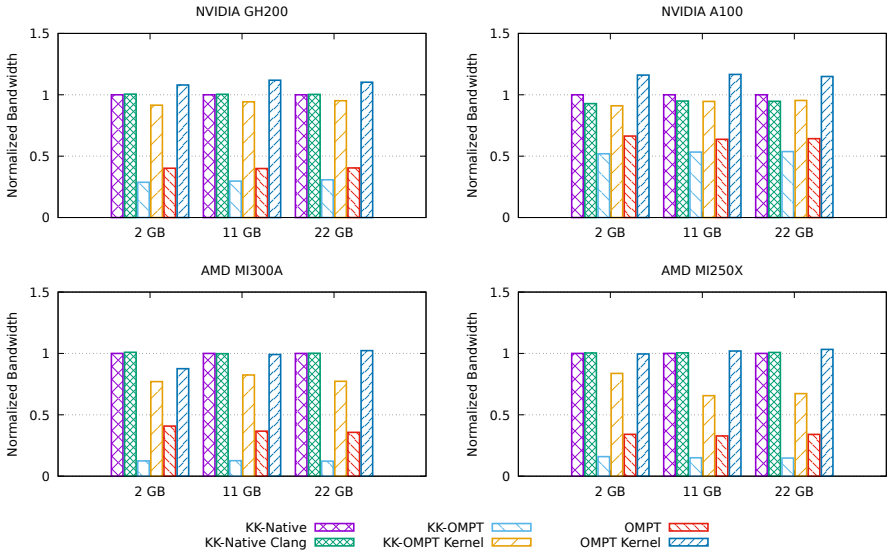

Fig. 4: Normalized bandwidth of SpMV relative to baseline *KK-Native* (CUDA/HIP) for various sizes of sparse matrix data. Higher is better.

OpenMPTarget backend, and 33-41% with direct OpenMP code. Using the LLVM extensions, the Kokkos OpenMPTarget backend performance reaches 66-84% of native on MI250X and 77-82% of native on MI300A. The direct OpenMP code with extensions is within ±4% of native performance, except for the smallest data size on MI300A.

While the extensions have helped to close the performance gap with the native backends, there is still some room for improvement on both NVIDIA and AMD devices at the intersection of OpenMP target offload and Kokkos C++ abstractions.

5.2 TestSNAP

Fig. 5 shows performance results for TestSNAP using the native and OpenMPTarget Kokkos backends. The execution time is normalized to that of *KK-Native*; lower is better, and the y-axis is cut off at 5X baseline performance for readability. In addition to the native backend and OpenMPTarget backend results (with and without extensions), the graphs in Fig. 5 also feature a fifth bar in each cluster of bars. This bar represents the performance using the OpenMPTarget backend kernel collapse implementation described in Section 3.

As with CGSolve, the results for the Kokkos native backends are mostly close between Clang and vendor compilers. The exception is the `compute_yi` kernel on the NVIDIA devices, which is 26-28% slower using Clang than `nvcc`. This

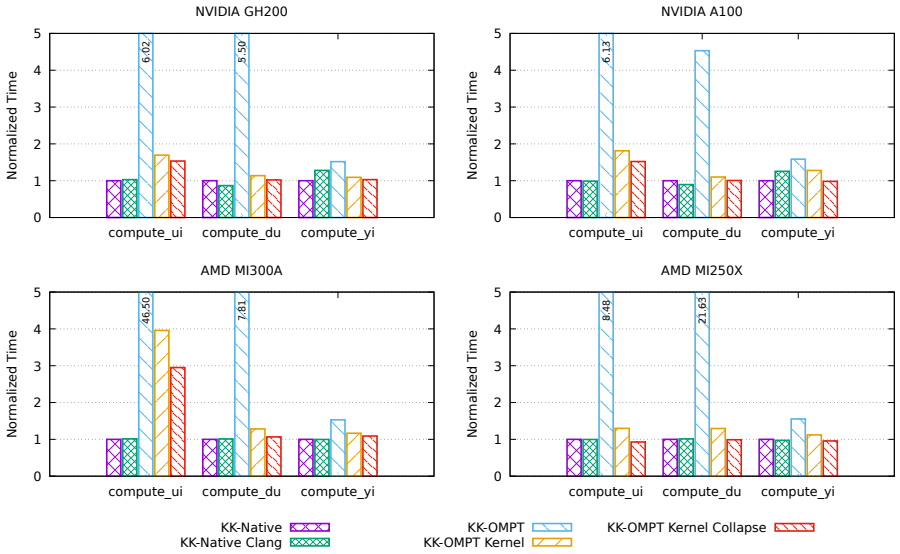

Fig. 5: Normalized execution time of TestSNAP relative to baseline *KK-Native* (CUDA/HIP) for several kernels. Lower is better.

discrepancy demonstrates the need to check for performance differences between compilers regarding GPU code generation.

For the `compute_ui` and `compute_du` kernels on NVIDIA, the Kokkos OpenMPTarget backend without LLVM extensions is 4X-6X slower. With the extensions, `compute_ui` and `compute_du` are only 69-81% and 7-10% slower than native, respectively. With the optimized kernel collapse implementation (Section 3), `compute_ui` is only 51-53% slower than native and `compute_du` is only 1-2% slower than native. The `compute_yi` kernel is 52-55% slower than native without the extensions. Using the extensions brings performance to within 9-12% of native, and using the kernel collapse gets it to within 3% on GH200 and on par with native on A100.

On AMD MI250X, the OpenMPTarget backend without extensions is more than 8X slower than native for `compute_ui`, but with extensions it is only 30% slower than native and with the kernel collapse it is slightly faster than native. Even with the extensions, OpenMPTarget backend performance for `compute_ui` on MI300A is over 4X slower than native, but it is a large improvement compared to the 46X slowdown over native without extensions. The kernel collapse reduces the slowdown to 3X over native, which is still a significant performance gap.

Profiling on MI300A indicates that kernel collapse has slightly lower register pressure compared to native for the `compute_ui` kernel, suggesting that the current team size (128) might not fully saturate the GPU resources. A local experiment using a larger team size (256) indeed demonstrates improved performance, although the gap compared to native remains. Further analysis of the generated instructions reveals that the compiler produced more optimized instructions

for native. Specifically, it utilizes native FP64 atomic-add instructions, while kernel collapse relies on compare-and-swap instructions. These differences likely contribute to the observed performance gap. A deeper investigation is required to identify the root cause more clearly.

Without extensions, `compute_du` is over 21X slower on MI250X and over 7X slower on MI300A. With extensions, the slowdown from native is only 28-29%. The kernel collapse results show roughly the same execution time as native on MI250X and within 7% of native on MI300A. On the two AMD devices, `compute_yi` is 53-55% slower than native without the extensions. Using the extensions brings performance to within 12-17% of native, and using the kernel collapse gets it to within 9% of native on MI300A and slightly faster than native on MI250X.

The order of magnitude of the improvements reflect the importance of the dynamic GPU shared memory support provided by LLVM OpenMP, as well as the kernel mode extensions that map efficiently to GPU compute resources and reduce overhead costs. Further improvements are provided by the OpenMPTarget backend kernel collapse implementation described in Section 3, which in several cases bring performance to parity with the native backends. Obviously, there is still room for improvement, and the remaining slowdown for `compute_ui` on MI300A merits special attention for further optimization.

6 Related Work

Several recent studies have investigated the use of OpenMP offload to program modern converged CPU + GPU architectures, such as GH200 and MI300A.

Jin [9] optimizes and analyzes the sum reduction operation using OpenMP offload in a Grace Hopper system with the OpenMP implementation from the NVIDIA HPC SDK. They demonstrate that, under certain conditions, co-executing the reduction in the CPU and GPU simultaneously can improve performance relative to the GPU-only version. Li et al. [10] implement a runtime tool to automatically offload BLAS operations executed by CPU code to the Hopper GPU, providing several memory management strategies, which leverage the high-speed cache-coherent NVLink C2C interconnect in GH200.

Tandon et al. [16] describes the MI300A APU and shows how to accelerate the code on it using ROCm's OpenMP. Relying on its unified physical memory, they obtain significant gains in the OpenFOAM application compared to discrete GPUs. Bertolli et al. [1] describe the changes in LLVM OpenMP to support zero-copy between CPU and GPU in OpenMP applications that map data on the MI300A APUs. Exploring the performance of CGSolve and TestSNAP enabling zero-copy support is left for future work.

Sfiligoi [12] accelerates PERMANOVA in MI300A using OpenMP offloading from the AOMP compiler. Although they only show GPU results for MI300A, this GPU version significantly outperforms OpenMP parallelization on the CPU. Elwasif [4] evaluates the support for Unified Shared Memory in OpenMP using different types of memory allocations on various compilers and GPU architectures, although MI300A and GH200 are not tested.

Other studies have also compared OpenMP with other high-level programming models, such as Kokkos, on these newer GPU architectures. Ruzicka et al. [11] compare the performance portability of OpenMP and Kokkos (with CUDA/HIP backends) in two plasma physics applications running on a broad variety of GPUs, including H100 and MI300A. Although both models show decent "out-of-the-box" performance, Kokkos performs better on new GPU architectures. Shan et al. [13] optimize the OpenACC and OpenMP implementations of a 3D stencil kernel, where NVIDIA's OpenACC runs 1.32X and 1.53X faster than NVIDIA's OpenMP on A100 and H100, respectively. Even so, it is shown that both programming models are narrowing the performance gap with native CUDA in newer architectures like H100. OpenMP optimizations such as the ones reviewed in this paper aim to help OpenMP versions reduce the gap against the CUDA/HIP, Kokkos and OpenACC counterparts.

7 Conclusion

The LLVM extensions to the OpenMP API allow the expression of parallelism and dynamic shared memory usage in a manner similar to native models like CUDA and HIP. We have shown how these extensions can be used in more complex parallel patterns and how they perform on the AMD Instinct™ MI300A Accelerator and NVIDIA GH200 Grace Hopper Superchip. These processors represent a new generation of CPU-GPU converged architectures, with increased computational prowess and additional memory bandwidth. Using the OpenMP-Target backend in the widely used Kokkos C++ performance portability library, we have demonstrated some application scenarios in which the extensions allow OpenMP to be competitive with the native programming models, and others in which they fall short.

In future work, we will more directly incorporate the innovative characteristics of the newer architectures – specifically, unified memory between host and device. The ability to avoid memory copies due to architectural design can be explored in the Kokkos-OpenMPTarget backend by avoiding the Kokkos View copies where the size of copy is lower than the granularity of hardware synchronization. We are also collaborating with LLVM OpenMP developers, who aim to optimize scalar copies on unified memory architectures for static variables within a given OpenMP scope.

Although more optimization work is needed to further improve performance, this paper provides evidence that the approaches represent a promising direction for OpenMP on GPUs. Our findings offer added motivation for their possible inclusion in future versions of the language specification.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

(LLNL-CONF-2006869). This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. Bertolli, C., Blass, T., Stringer, L., Aschenbrenner, N., Lehr, J.P., Bercea, D., Chakrabarti, D., Meadows, L., Lieberman, R.: Performance analysis of runtime handling of zero-copy for OpenMP programs on MI300A APUs. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1420–1429. IEEE (2024)
2. Doerfert, J., Patel, A., Huber, J., Tian, S., Diaz, J.M.M., Chapman, B.M., Georgakoudis, G.: Co-designing an OpenMP GPU runtime and optimizations for near-zero overhead execution. In: 2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022. pp. 504–514. IEEE (2022). https://doi.org/10.1109/IPDPS53621.2022.00055, https://doi.org/10.1109/IPDPS53621.2022.00055
3. Elster, A.C., Haugdahl, T.A.: Nvidia Hopper GPU and Grace CPU highlights. Computing in Science & Engineering **24**(2), 95–100 (2022). https://doi.org/10.1109/MCSE.2022.3163817
4. Elwasif, W.: Experimental characterization of OpenMP offloading memory operations and unified shared memory support. In: McIntosh-Smith, S., Klemm, M., de Supinski, B.R., Deakin, T., Klinkenberg, J. (eds.) OpenMP: Advanced Task-Based, Device and Compiler Programming. pp. 210–225. Springer Nature Switzerland, Cham (2023)
5. Evans, J.: Nvidia Grace. In: 2022 IEEE Hot Chips 34 Symposium (HCS). pp. 1–20. IEEE Computer Society, Los Alamitos, CA, USA (Aug 2022). https://doi.org/10.1109/HCS55958.2022.9895599
6. Gayatri, R., Moore, S., Weinberg, E., Lubbers, N., Anderson, S., Deslippe, J., Perez, D., Thompson, A.P.: Rapid exploration of optimization strategies on advanced architectures using TestSNAP and LAMMPS. arXiv preprint arXiv:2011.12875 (2020)

7. Gayatri, R., Olivier, S.L., Trott, C.R., Doerfert, J., Ciesko, J., Lebrun-Grandie, D.: The Kokkos OpenMPTarget backend: Implementation and lessons learned. In: McIntosh-Smith, S., Klemm, M., de Supinski, B.R., Deakin, T., Klinkenberg, J. (eds.) OpenMP: Advanced Task-Based, Device and Compiler Programming. pp. 99–113. Springer Nature Switzerland, Cham (2023)
8. Gayatri, R., Tian, S., Olivier, S.L., Wright, E., Doerfert, J.: Leveraging LLVM OpenMP GPU offload optimizations for Kokkos applications. In: 2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 277–287 (2024). https://doi.org/10.1109/HiPC62374.2024.00035
9. Jin, Z.: Sum reduction with OpenMP offload on NVIDIA Grace-Hopper system. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1006–1013 (2024). https://doi.org/10.1109/SCW63240.2024.00140
10. Li, J., Wang, Y., Liang, X., Liu, H.: Automatic BLAS offloading on unified memory architecture: A study on NVIDIA Grace-Hopper. In: Practice and Experience in Advanced Research Computing 2024: Human Powered Computing. PEARC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626203.3670561, https://doi.org/10.1145/3626203.3670561
11. Ruzicka, J., Asch, C., Meneses, E., Rampp, M., Laure, E.: A study of performance portability in plasma physics simulations. In: Latin American High Performance Computing Conference. pp. 19–35. Springer (2024)
12. Sfiligoi, I.: Comparing CPU and GPU compute of PERMANOVA on MI300A (2025), https://arxiv.org/abs/2505.04556
13. Shan, B., Araya-Polo, M., Chapman, B.: Evaluation of directive-based programming models for stencil computation on current GPGPU architectures. In: Espinosa, A., Klemm, M., de Supinski, B.R., Cytowski, M., Klinkenberg, J. (eds.) Advancing OpenMP for Future Accelerators. pp. 126–140. Springer Nature Switzerland, Cham (2024)
14. Smith, A., Chapman, E., Patel, C., Swaminathan, R., Wuu, J., Huang, T., Jung, W., Kaganov, A., McIntyre, H., Mangaser, R.: AMD InstinctTM MI300 series modular chiplet package – HPC and AI accelerator for exa-class systems. In: 2024 IEEE International Solid-State Circuits Conference (ISSCC). vol. 67, pp. 490–492 (2024). https://doi.org/10.1109/ISSCC49657.2024.10454441
15. Smith, A., Loh, G.H., Schulte, M.J., Ignatowski, M., Naffziger, S., Mantor, M., Kalyanasundharam, M.F.N., Alla, V., Malaya, N., Greathouse, J.L., Chapman, E., Swaminathan, R.: Realizing the AMD exascale heterogeneous processor vision : Industry product. In: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). pp. 876–889 (2024). https://doi.org/10.1109/ISCA59077.2024.00068
16. Tandon, S., Grinberg, L., Bercea, G.T., Bertolli, C., Olesen, M., Bna, S., Malaya, N.: Porting HPC applications to AMD Instinct™ MI300A using unified memory and OpenMP®. In: ISC High Performance 2024 Research Paper Proceedings (39th International Conference). pp. 1–9 (2024). https://doi.org/10.23919/ISC.2024.10528925
17. Tian, S., Scogland, T., Chapman, B., Doerfert, J.: OpenMP kernel language extensions for performance portable GPU codes. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. p. 876–883. SC-W '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624164, https://doi.org/10.1145/3624062.3624164

18. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283