Leveraging LLVM OpenMP GPU Offload Optimizations for Kokkos Applications

Rahulkumar Gayatri
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
rgayatri@lbl.gov

Shilei Tian
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY, USA
shilei.tian@stonybrook.edu

Stephen L. Olivier
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
slolivi@sandia.gov

Eric Wright
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
wright117@llnl.gov

Johannes Doerfert
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
doerfert1@llnl.gov

***Abstract*—OpenMP provides a cross-vendor API for GPU offload that can serve as an implementation layer under performance portability frameworks like the Kokkos C++ library. However, recent work identified some impediments to performance with this approach arising from limitations in the API or in the available implementations. Advanced programming concepts such as hierarchical parallelism and use of dynamic shared memory were a particular area of concern. In this paper, we apply recent improvements and extensions in the LLVM/Clang OpenMP compiler and runtime library to the Kokkos backend that targets GPUs via OpenMP offload. We focus on efficient hierarchical parallelism and use of fast GPU scratch memory. We compare the performance of applications written using the Kokkos library with this improved OpenMP backend against the same programs using the CUDA and HIP backends. This evaluation shows progress toward closing the performance gaps between native and OpenMP backends and offers insights that may be useful to users and implementers of other runtime systems and programming frameworks for GPUs.**

***Index Terms*—Computing methodologies → Parallel programming languages → Kokkos, OpenMP, CUDA, HIP**

I. Introduction

Kokkos is a performance portability library that allows a single C++ code base to efficiently execute on diverse GPUs and CPUs [1]. Many scientific applications, especially those developed via the United States Exascale Computing Project (ECP), use it as a programming model. The Linux Foundation assumed ownership of Kokkos to ensure lasting sustainability.

While the front end interface of Kokkos is modern ISO standard C++, its implementation provides several different backends depending on the target architecture. Backends using the vendor preferred programming models (NVIDIA's CUDA, AMD's HIP, and Intel's DPC++ dialect of SYCL) provide the best performance on each vendor's hardware. However, vendor preferred programming models are poorly supported, if at all, on the other vendors' hardware. OpenMP's device offload model [2] provides an alternative to vendor preferred programming model backends of Kokkos. The OpenMP API enjoys broad compiler support, including GCC, Clang/LLVM, and vendor implementations (many of which are based on LLVM). Kokkos includes one backend for OpenMP on host CPUs only, and another with GPU support using the offload model. The latter, which Kokkos refers to as the OpenMP-Target backend, is the subject of this paper. For the rest of the paper, whenever we refer to Kokkos-OMP backend, it is specifically to the OpenMPTarget backend that targets GPUs.

Previous work investigated the performance of Kokkos-OMP backend compared to the CUDA and HIP backends on NVIDIA and AMD GPUs, respectively, in the context of a conjugate gradient solver [3]. That study found that while performance of single level parallelism was adequate, the Kokkos-OMP backend handled hierarchical parallelism, e.g., sparse-matrix vector multiplication, poorly.

In this paper, we demonstrate improvements to the Kokkos-OMP backend by integrating recent capabilities from LLVM/OpenMP in the following new extensions:

- Support for GPU scratch memory (often referred to as "shared" memory on NVIDIA GPUs or "Local Data Storage (LDS)" on AMD GPUs), with a focus on dynamic scratch memory allocation.
- Implementation of grid parallelism in OpenMP, analogous to CUDA/HIP, along with comprehensive support for all three levels of hierarchical parallelism.
- Integration of advanced reduction techniques, including shuffle instructions in OpenMP.

We evaluate these improvements using Kokkos-based applications, comparing their performance against existing implementations and vendor-preferred backends for each target architecture.

The remainder of the paper is organized as follows. In Section II and Section III we explain the motivation and background for the work. Section IV discusses the extensions to the LLVM/OpenMP ecosystem and how they are integrated into the Kokkos framework, along with empirical results to

demonstrate performance impact. We discuss some related works in Section V, and conclude the paper in Section VI.

II. Motivation

Kokkos has native backends for all three major GPU vendors, i.e., NVIDIA, AMD and Intel. The OpenMPTarget backend is considered a secondary backend for the GPUs. The motivation for a secondary backend is two-fold: 1) risk mitigation, especially in scenarios where Kokkos applications might need to interact with external libraries that use OpenMP and 2) preparedness for any future hardware that relies on OpenMP as its first or primary framework, so that applications based on Kokkos can compile and run on such an architecture.

As a C++ framework, Kokkos templates its backend implementation on vendor specific programming models such as CUDA, HIP and SYCL. It also provides constructs to users that may be unavailable even to the native backends such as atomics on user defined data types. Providing implementation for such involved and advance concepts using the OpenMP offload directives provides a litmus test for the compiler implementations and a proof of concept for C++ applications that might consider OpenMP for their portable implementations.

Additionally, the OpenMPTarget backend enables interoperability between OpenMP tooling infrastructure such as record and replay [4] and advanced OpenMP features in LLVM such as remote OpenMP offloading [5] and JIT compilation [6].

However, the performance of Kokkos applications using the OpenMPTarget backend is typically slower than the native backends. While compiler maturity is one of the reasons for such slowdowns, a major contributor is also the lack of features in the OpenMP standard to completely exploit the available parallelism on a GPU. In our study, we aim to demonstrate how OpenMP can compete against native backends through small changes and extensions to OpenMP and efficient design choices in the implementation of the API.

Our evaluation used NVIDIA A100 GPUs (40GB HBM) available on NERSC Perlmutter located at Lawrence Berkeley National Laboratory and AMD MI250X GPUs on OLCF Frontier at Oak Ridge National Laboratory. For the CUDA builds we used cuda/12.2 and for HIP we used rocm/6.0.

III. Background

Kokkos enables developers to write a common code base that can compile and run on multiple CPU and GPU architectures with minimal changes. Fig. 1 shows its available backends and supported architectures. Execution patterns such as `parallel_for` and `parallel_reduce` are provided for parallel iteration over loop ranges or items in a view (Kokkos multidimensional array). The body of the loop is specified as a C++ lambda. The parallelism may be flat or hierarchical. This paper presents modifications to the implementation of the execution patterns in one backend and hence user code changes are not required to benefit from the optimizations described.

While OpenMP has been available since the late 1990's for CPUs, OpenMP GPU offload is a more recent development, both in the API specification and in compiler implementations.

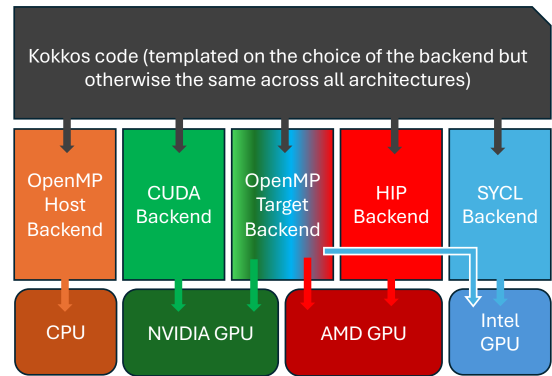


Fig. 1: Kokkos backends and supported architectures.

Level	Kokkos	CUDA	OpenMP
Top	teams	block	teams
Mid	threads	Y dim	parallel
Low	vector	X dim	simd

Fig. 2: Parallelism levels in Kokkos, CUDA, and OpenMP

The widely used `parallel for` directive creates a team of threads that execute a loop in parallel. When used in a `target` construct along with `teams distribute` for GPU offload, it results in the creation of multiple teams spread across the blocks of a GPU, wherein, each team has multiple threads running in parallel.

Fig. 2 shows how OpenMP offload parallelism corresponds to parallelism in Kokkos and CUDA. However, the mapping does not indicate fully equivalent behavior. Firstly, below the thread level, the `simd` directive can enable vector parallelism, but this directive is commonly ignored by many compilers when generating GPU code. Secondly, simple block and thread indexing in grid languages like CUDA keeps overheads low, in comparison to the more heavyweight state required by OpenMP semantics. A major focus of the work described in this paper is the adaptation of the grid style expression of parallelism to OpenMP offload, as well as GPU scratch memory use and optimization of reduction operations.

IV. LLVM OpenMP Extensions

In this section we discuss extensions proposed to the LLVM/OpenMP ecosystem. We explain the motivation for these extensions and their use in Kokkos-OMP backend. We also show the performance impact on Kokkos applications using the newly optimized Kokkos-OMP backend.

A. Dynamic Shared Memory Extension

One limitation of current OpenMP API is the inability to expose dynamic memory as "shared" among threads in an OpenMP team. While OpenMP 6.0 has proposed extensions to address the issue for stack or compile time constant variables, support for sharing of dynamically allocated memory among members of a team is still unavailable. LLVM/OpenMP has introduced extensions to ad-

```
using ScratchViewType = Kokkos::View<int*, Kokkos
    ::DefaultExecutionSpace::scratch_memory_space
    ,...>;

 // Create N teams with 32 threads per team.
 Kokkos::TeamPolicy team_policy(N, 32);
 size_t scratch_size = team_size *
     scratch_per_thread * sizeof(int);

 Kokkos::parallel_for(team_policy.set_scratch_size
     (0, Kokkos::PerTeam(scratch_size) ),
     KOKKOS_LAMBDA ( const member_type &teamMember
     ) {
  ...
  int num_scratch_elems = scratch_size / sizeof(
      int);
  ScratchViewType scratch(teamMember.team_scratch
      (0),  num_scratch_elems);
  ...
 });
```

Fig. 3: Scratch memory invocation in Kokkos.

```
__global__ cu_kernel(...) {
 extern shared int scratch[];
 ...
}
void calling_kernel(...){
 size_t scratch_size =  scratch_per_thread *
     team_size * sizeof(int);
 cu_kernel<<<N, 32, scratch_size>>>(...)
}
```

Fig. 4: Scratch memory invocation in CUDA.

```
template <class Functor, class... Properties>
class ParallelFor<Functor, Kokkos::TeamPolicy<
    Properties...>, Kokkos::OpenMPTarget> {
  ...
  Functor f;
  void execute() {
 int device = omp_get_default_device();
#if (CLANG_VERSION > 1800)
void* scratch_ptr = omp_target_alloc(shmem_size_L1
    , device);
#pragma omp target teams thread_limit(team_size)
    firstprivate(m_functor) num_teams(...)
    is_device_ptr(scratch_ptr) ompx_dyn_cgroup_mem
    (shmem_size_L0)
 #pragma omp parallel
 {
  for (int lId = blockIdx; lId < league_size;
        lId += gridDim) {
      team(lId, league_size, team_size,
          vector_length, scratch_ptr, blockIdx,
          shmem_size_L0, shmem_size_L1);
      f(team); // Inner loop inside
     }
 }
#else
void* scratch_ptr = omp_target_alloc(shmem_size_L0
    +shmem_size_L1, device);
#pragma omp target teams thread_limit(team_size)
    firstprivate(a_functor) num_teams(...)
    is_device_ptr(scratch_ptr)
  // Same parallel region as above
#endif
 }
};
```

Fig. 5: Scratch memory implementation in Kokkos-OMP.

dress this issue. It provides a new `target` directive clause `ompx_dyn_cgroup_mem(<N>)` that allocates N bytes of data per team that can be shared among threads in that team. This feature corresponds to the shared memory that is allocated during a CUDA kernel invocation. The `llvm_omp_target_dynamic_shared_alloc` routine, called inside the target region, returns a pointer to the shared memory. The routine returns the same value to each thread of a target team and returns a `NULL` pointer on the host. The extension is available in upstream LLVM since release 18.

Kokkos provides a similar abstraction called the "scratch-pad" to share data among threads in a team. Multiple levels of scratch-pad are provided in Kokkos. The first level (level-0) of the scratch-pad is typically mapped to the team specific allocatable local storage in the memory hierarchy, and hence is restricted to a few kilobytes. The native backends of Kokkos, those for CUDA and HIP, use the unified L1 / shared memory and local data share (LDS), on NVIDIA and AMD GPUs respectively. The second level (level-1) of scratch-pad is typically mapped to the high bandwidth memory (HBM). Until the availability of the `ompx_dyn_cgroup_mem(<N>)` clause in LLVM/OpenMP, Kokkos-OMP backend provided support for this feature by allocating the required scratch memory (for both level-0 and level-1) on the main memory of a GPU by using the `omp_target_alloc` routine. This led to negative performance implications in certain applications since the expectation that shared reads/writes within a team take place in the fast access local storage specific to each team did not match the implementation. We resolved this issue with the use of the new clause when LLVM/Clang is 18 or newer.

The basic behavior of Kokkos and its ability to pass functors to the backends and its subsequent handling of them is explained in prior work [1]. Subsequent work [3] provides a deeper discussion specific to the Kokkos-OMP backend.

Fig. 3 shows an example of how the scratch-pad can be used in Kokkos. Elements in scratch memory are accessed through a custom version of `Kokkos::View`, the primary Kokkos data structure. The first line in Fig. 3 defines such a view, `ScratchViewType`. The second template parameter indicates that the memory space where the data would reside is the scratch space of the default execution space, i.e., the default architecture on which a Kokkos `parallel_pattern` would be executed. Lines 4 and 5 determine how much scratch memory is needed per team. The `Kokkos::team_policy` specifies the dimensions of a work grid of thread teams, in this case `N` teams with 32 threads per team. To request scratch space shared among threads of a team we use a member function to the `team_policy` called `set_scratch_size` as shown in the Kokkos parallel pattern. The first parameter to `set_scratch_size` is the scratch level (0 or 1). The second parameter is the amount of scratch memory needed per team. To access scratch memory inside the `parallel_for` kernel, we create a view of `ScratchViewType`. Fig. 4 is the CUDA equivalent implementation of Fig. 3.

In Fig. 5, we show two implementations for scratch memory in Kokkos-OMP. The choice of the implementation is based on LLVM versions. `shmem_size_L0` and `shmem_size_L1` refer to scratch memory requested by the user for level-0

```
1 #if (CLANG_VERSION > 1800)
2   char* l0_scratch_ptr = static_cast<char*>(
        llvm_omp_target_dynamic_shared_alloc());
3   team_scratch_view = scratch_memory_space(
        l0_scratch_ptr, shmem_size_L0, scratch_ptr,
        shmem_size_L1);
4 #else
5   team_shared_view = scratch_memory_space(
        scratch_ptr, shmem_size_L0, scratch_ptr +
        shmem_size_L0, shmem_size_L1);
6 #endif
```

Fig. 6: Scratch view in Kokkos-OMP.

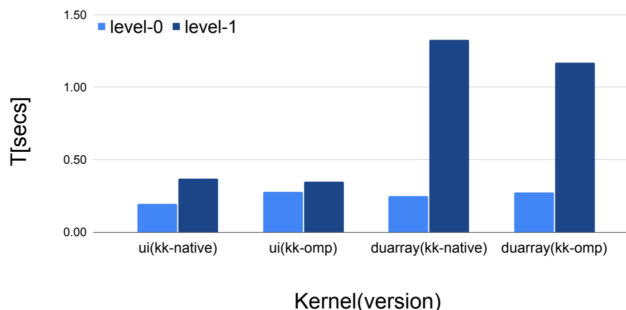
and level-1 of a `TeamPolicy`. For LLVM 18 and higher, we use the LLVM extension `ompx_dyn_cgroup_mem` to allocate `shmem_size_L0` in shared memory. Level-1 of scratch memory is then allocated on HBM using `omp_target_alloc`. For LLVM prior to 18, both level-0 and level-1 scratch memory is allocated on HBM and indexed accordingly when accessed in a scratch view. The implementation using LLVM extensions in Kokkos-OMP is our contribution and is already available since the 4.3 Kokkos release. The code inside the `parallel` region in Fig. 5 leverages hierarchical parallelism, as explained in later sections.

Fig. 6 shows how a scratch memory view is created in Kokkos. It is a common interface across all backends that uses a routine named `scratch_memory_space`. The first two parameters to the routine are pointers for level-0 and its corresponding size. The last two parameters follow the same pattern for level-1 of the scratch memory. `scratch_ptr` refers to the memory allocated using `omp_target_alloc` from Fig. 5. For LLVM 18 and above, level-0 scratch is accessed via the `llvm_omp_dynamic_shared_alloc` extension. For LLVM 17 or lower, we index into `scratch_ptr` according to the size of level-0 and level-1 scratch memory requested.

As shown in Fig. 5 and 6, using the extensions to allocate dynamic shared memory within a team enables Kokkos-OMP to match the implementation of level-0 scratch memory as intended by the framework.

To demonstrate the impact of dynamic shared memory in the Kokkos-OMP backend, consider TestSNAP proxy application that is modelled on Spectral Neighborhood Analysis Potential (SNAP) computations in the LAMMPS molecular dynamics simulator. It calculates the total energy of a configuration of atoms as the sum of energies of individual atoms, each of which is dependent on its neighbor atoms within a certain distance. We select a problem from the Exascale Computing Project, with 2000 atoms on one GPU and 26 neighbors per atom. Of the three main kernels in TestSNAP that consume more than 98% of the total execution time, two (`ui` and `duarray`) use level-0 scratch memory to store team specific intermediate information for fast access [7].

The `ui` kernel calculates expansion coefficients for each pair of neighbor atoms. It is implemented by generating the number of teams based on the outer loop while the inner loops are iterated in a 2-dimensional thread-block. The kernel uses scratch memory to store partial updates to coefficients for fast access. The `duarray` kernel computes the partial derivative



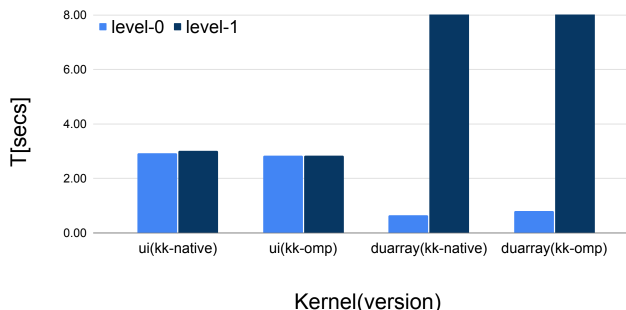


Fig. 7: `TestSNAP` performance of native and OMP backend under scratch-levels 0 (shared memory) and 1 (HBM).

of the coefficients that impact the force on each atom and neighbor pair. The parallelism exposed in this kernel is similar to the `ui` kernel and hence implemented similarly.

Fig. 7 shows the impact of of using dynamic shared memory extension in the Kokkos implementation of TestSNAP. We compare the benefits gained by using level-0 scratch memory compared to level-1, i.e., implementation prior to the use of the LLVM extension. The Y axis shows the time for each kernel to run 100 timesteps while the X-axis indicates the two kernels using scratch memory under Kokkos-native and Kokkos-OMP backends. For the rest of the paper we follow the convention of prefixing the Kokkos implementations with "kk" in figures. For a fair evaluation of the benefits of dynamic shared memory we also show the performance using native backends under the two scratch levels. The comparison with the native backend illustrates the performance regression observed if the scratch memory is allocated on HBM rather than in the L1 cache.

SNAP and TestSNAP use all three levels of hierarchical parallelism available in Kokkos in `ui` and `duarray`, i.e., the kernels using scratch memory. Kokkos-OMP backend however has only two levels of effective hierarchical parallelism [3]. Hence, for a fair comparison, in Fig. 7, we restrict the native backends to two levels of hierarchical parallelism.

Fig. 7 shows that on NVIDIA A100, `ui` performance improves by 15% with Kokkos-OMP when data is stored in level-0 scratch versus level-1 scratch while Kokkos-native backend shows a $2\times$ speedup in the same scenario. In `duarray` we see a $\approx$4.5$\times$ performance improvement with Kokkos-OMP backend and a $5\times$ speedup with the native backend. For both

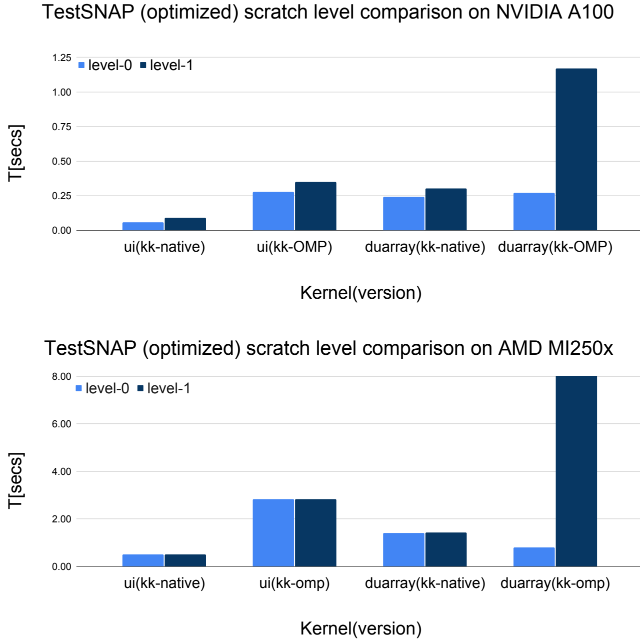


Fig. 8: `TestSNAP` performance with optimized native and OMP backends under scratch-levels 0 and 1.

kernels, the speedup achieved by Kokkos-native backend is higher than the Kokkos-OpenMP backend when team-shared data is stored in fast access local storage rather than HBM.

On AMD GPUs, going from level-1 to level-0 scratch memory gives the same rate of performance improvements on both Kokkos-native and Kokkos-OMP backends. For `ui` in both the backends, we do not see any observable performance improvement. However, in the case of `duarray` we observe an $\approx 15\times$ performance improvement when using the LLVM extensions to store team shared data in LDS on AMD MI250X. This matches the observation from the Kokkos-native backend.

While we only show one application, our goal of this paper is to show a proof of concept of how the `ompx_dyn_cgroup_mem` extension in LLVM/OpenMP can bridge one of the gaps between OpenMP and native frameworks. There are several other applications written in Kokkos that can take advantage of the fast access local storage per team available through the level-0 scratch-pad memory interface. With the new extension in LLVM, Kokkos-OMP can provide the intended implementation to the user. The ability to allocate and access dynamic scratch memory that can be shared among threads in a team is a major capability previously missing in OpenMP but available in the native frameworks.

Fig. 7 might give an impression that the dynamic memory extension largely closes the performance gap between native and OpenMP backends. However, as mentioned earlier, we had restricted the Kokkos-native backends to use only two hierarchical parallelism levels for an equivalent comparison with the Kokkos-OMP backend. Fig. 8 shows a comparison with the optimized native versions that use all three levels of hierarchical parallelism. The differences in kernel execution times between the Kokkos backends is still significantly different. For `ui` on both NVIDIA A100 and AMD MI250X, the native backend benefits significantly with the addition of three levels of parallelism, making it $\approx 4.5\times$ faster on Kokkos-native compared to Kokkos-OMP. Currently the Kokkos-OMP backend has no meaningful way to extract the 3rd parallelism level [3]. The remainder of this paper addresses that need.

B. LLVM OpenMP Kernel Mode Extension

The second LLVM extension we leverage is perhaps an even more important addition to the LLVM/OpenMP ecosystem, enabling us to bridge the gap between native frameworks and OpenMP. While existing OpenMP offers a rich set of parallel semantics, which includes a fork-join model and automatic workload distribution, it needs the support of an extensive runtime library to manage execution. Runtime operations often limit performance and consume substantial resources, particularly on a GPU. Also this overhead has been generally regarded as unavoidable under the existing API semantics [8].

To overcome the runtime overhead, LLVM/OpenMP proposed a new set of extensions that allow OpenMP target regions to execute in a "bare metal" mode, also known as *kernel mode* [9]. This feature enables OpenMP GPU code to be written in single instruction multiple threads (SIMT) style, facilitating an easy transition of an existing GPU code written in kernel languages such as CUDA and HIP to OpenMP thereby benefiting from the portability offered by OpenMP. Furthermore, the OpenMP kernel mode only requires a thin layer of runtime library support compared to existing OpenMP, eliminating runtime overhead to potentially improve performance. This work is built on the work of Tian et al. [9] to implement features necessary for supporting the Kokkos programming model using a combination of the existing OpenMP and the extended OpenMP kernel mode.

We use the LLVM extension `ompx_bare` as an additional clause to the `pragma omp target teams` construct that directs the compiler to execute the associated code block in "bare metal" SIMT mode. As with the dynamic shared memory feature, the additional clause is an extension and not in the OpenMP standard. Therefore it is prefaced with `ompx` instead of the usual `omp`. Multiple extensions are needed to support CUDA/HIP style code generation in OpenMP, the first of which is the implementation of multi-dimensional grid and blocks. For that LLVM extends the `num_teams` and `thread_limit` clauses to accept a list of integers. For instance, a three-dimensional CUDA block size represented as `dim3 blockSize(4, 32, 2)` can be equivalently expressed using `thread_limit(4, 32, 2)`. Similarly, a 3-D grid can be generated as `num_teams(x,y,z)`, which would be equivalent to `dim3 gridsize(x,y,z)`.

We can replicate the CUDA style kernel launch shown in Fig. 9 in OpenMP by using the two kernel mode extensions: 1) the SIMT style code generation clause and 2) multi-dimensional grid and blocks. The OpenMP version using the extensions is shown in the first line of Fig. 10. In both programming models, it generates a grid of `128` thread-blocks

```
cu_kernel<<<dim3(128,1,1), dim3(4,32,2)>>>(a);
```

Fig. 9: CUDA kernel launch.

```
#pragma omp target teams ompx_bare num_teams(128,
    1, 1) thread_limit(4, 32, 2) firstprivate(a)
{
 // number of teams in X-dimension
 int blockDimx = ompx::block_dim(ompx::dim_x);
 // team-id in X-dimension
 int blockIdx  = ompx::block_id(ompx::dim_x);
 // thread-id in X-dimension
 int threadIdx = ompx::thread_id(ompx::dim_x);
 // thread-id in Y-dimension
 int threadIdy = ompx::thread_id(ompx::dim_y);
 ...
}
```

Fig. 10: OpenMP target region (kernel launch).

in X-dimension, in which each thread-block/team contains 4, 32 and 2 threads in X,Y and Z dimensions respectively.

Fig. 10 also shows the C++ specific access mechanisms for grid and thread dimensions and the corresponding team and thread ID inside the kernel mode.

The kernel mode extensions allow the Kokkos-OMP backend to have a meaningful implementation for three levels of hierarchical parallelism which include, 1) a league of teams, 2) each team comprising multiple threads and 3) each thread comprising of multiple vectors. Having a vector level allows optimizations on CPUs with hardware vector units and instructions. Fig. 11 shows how multi-level parallelism can be exposed using the three levels of parallel hierarchy in Kokkos.

Fig. 11 creates a 3-dimensional `Kokkos::View` that is initialized by traversing each dimension in each of the hierarchical levels. The current implementation of the Kokkos-OMP backend available in upstream Kokkos maps the outer league level to `omp target teams`, the intermediate thread level to `omp for` and the final vector level to `omp simd`. However, the `simd` clause inside a `target` region is serialized in most OpenMP implementations, including LLVM. Prior work Gayatri et al. [3] discusses in detail the drawbacks of such a mapping and its impact on the performance of Kokkos applications that use the Kokkos-OMP backend.

```
Kokkos::View<int***,Kokkos::DefaultExecutionSpace>
     a("a",N,N,N);
Kokkos::parallel_for(
 Kokkos::TeamPolicy<>(N, team_size, vector_size),
KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::
    member_type &team) {
  const int i = team.league_id();
  // iterate over rows owned by this team
  Kokkos::parallel_for(
    Kokkos::TeamThreadRange(team, N), [&](const
        int64_t j) {
    // reduction inside the vector range
    Kokkos::parallel_for(
      Kokkos::ThreadVectorRange(team, N), [&](
          const int64_t k) {
      a(i,j,k) = ...;
  });
  });
 });
```

Fig. 11: Kokkos hierarchical parallelism.

The introduction of the `ompx_bare` extension allows SIMT style kernel generation such that the Kokkos-OMP backend can now exploit all three levels of hierarchical parallelism, similar to native Kokkos backends. The CUDA/HIP backends map the outer level to teams (thread blocks). Within each team, the native backends generate a 2-dimensional block of threads. Within a thread block, `Kokkos::TeamThreadRange` is mapped to Y-dimension of threads and `Kokkos::ThreadVectorRange` is mapped to X-dimension of the threads. The innermost loop is mapped onto consecutive threads to improve memory coalescing.

Fig. 12 shows our implementation of Fig. 11 in Kokkos-OMP when kernel mode extensions are enabled. This implementation is currently in a separate fork [1]. When the extensions are available in upstream LLVM, we will submit a pull request for the implementation to be adopted in the main Kokkos repository. For the rest of the paper, to differentiate between our implementation of Kokkos-OMP backend using the kernel mode extensions and the upstream Kokkos-OMP backend discussed in [3], we call our implementation the Kokkos-OMPX backend. Fig. 12 shows the effective code and not the exact Kokkos code as we want to focus on the Kokkos-OMPX backend rather than the Kokkos API.

The functor from the league (outermost) level `parallel_for` pattern in Fig. 11 is passed to the `ParallelFor` class shown in Fig. 12. The execution policy and its associated information is passed as a template parameter, shown in the second template parameter to the `ParallelFor` class. The `execute` member function in `ParallelFor` implements the parallel pattern. This style is consistent for all patterns in all Kokkos backends.

Requests for the two levels of scratch memory are represented by `shmem_size_L0` and `shmem_size_L1`. The dynamic shared memory extension, discussed in the previous section, is used to request `shmem_size_L0` amount of shared memory. Kernel mode is entered using the `ompx_bare` clause and the grid dimensions are generated using the `num_teams` and `thread_limit` extensions. Instead of generating a kernel with the same number of teams as requested by the user, the implementation calculates an optimized number of teams to maximize performance of the kernel. Using a larger number of teams can improve GPU occupancy for better code performance. However, it can also increase the memory footprint required by Kokkos to maintain team specific meta-data, e.g., the level-1 scratch memory allocated on HBM. Conversely, generating fewer teams can limit the available parallelism. Kokkos backends try to optimize the number of teams generated based on the underlying architecture and scratch memory requested. In Fig. 12, this number is represented as `max_teams`. A loop inside the kernel mode handles cases where `max_teams` is smaller than the requested `league_size` (iteration space of the outermost loop in Kokkos hierarchical parallelism).

[1] https://github.com/rgayatri23/kokkos/tree/ompt_kernel_mode.

```
1 template <class Functor, class... Properties>
2 class ParallelFor<Functor, Kokkos::TeamPolicy<
      Properties...>, Kokkos::OpenMPTarget> {
3   ...
4   Functor f;
5   void execute() {
6     // scratch request for level-0 and 1
7     int shmem_size_L0 = ...;
8     int shmem_size_L1 = ...;
9     // number of teams based on occupancy
10    int max_teams = ...;
11 #pragma omp target teams ompx_bare num_teams(
      max_teams) thread_limit(vector_size,team_size
      ,1) ompx_dyn_cgroup_mem(shmem_size_L0)
12    {
13    int blockId = ompx::block_id(ompx::dim_x);
14    int gridDim = ompx::grid_dim(ompx::dim_x);
15
16    for (int lId = blockId; lId < league_size; lId
           += gridDim) {
17      team(lId, league_size, team_size,
            vector_length, scratch_ptr, blockId,
            shmem_size_L0, shmem_size_L1);
18      f(team); // Inner loop inside
19    }
20  }
21 }
22
23 template <class Lambda>
24 void parallel_for(const TeamThreadRangeBoundaries
      <...>& loop_boundaries, const Lambda& lambda)
      {
25   int start = loop_boundaries.start;
26   int end = loop_boundaries.end;
27   int blockDimy=ompx::block_dim(ompx::dim_y);
28   int threadIdy=ompx::thread_id(ompx::dim_y);
29   for (int i=start+threadIdy; i<end; i+=blockDimy)
30     lambda(i);
31 }
32
33 template <class Lambda>
34 void parallel_for(
35 const ThreadVectorRangeBoundaries<...>&
      loop_boundaries, const Lambda& lambda) {
36   int start = loop_boundaries.start;
37   int end = loop_boundaries.end;
38   int blockDimx=ompx::block_dim(ompx::dim_x);
39   int threadIdx=ompx::thread_id(ompx::dim_x);
40   for (int i=start+threadIdx; i<end; i+=blockDimx)
41     lambda(i);
42 }
```

Fig. 12: Hierarchical parallelism in Kokkos-OMPX backend.

Below the `ParallelFor` class, we show the code snippet for the `parallel_for` implementation of the `TeamThreadRange` hierarchy. The Kokkos-OMPX backend iterates through the Y-dimension of the team threads rather than using `omp for` directives as in Kokkos-OMP.

At the end of Fig. 12, we demonstrate the `parallel_for` implementation for `ThreadVectorRange`. In Kokkos-OMPX we iterate through the X-dimension of threads in a team until we reach the loop boundaries. The same is implemented using `simd` in Kokkos-OMP as mentioned earlier. The ability to use parallelism in all 3-levels of parallel hierarchy is one of the main advantages of kernel mode extensions.

Fig. 13 shows the performance of TestSNAP when running with the Kokkos-OMPX backend compared to Kokkos-native and Kokkos-OMP backends. The team and vector sizes in the kernel mode are similar to the native backend, matching the best grid dimension for the backend, i.e., a `vector_size` of

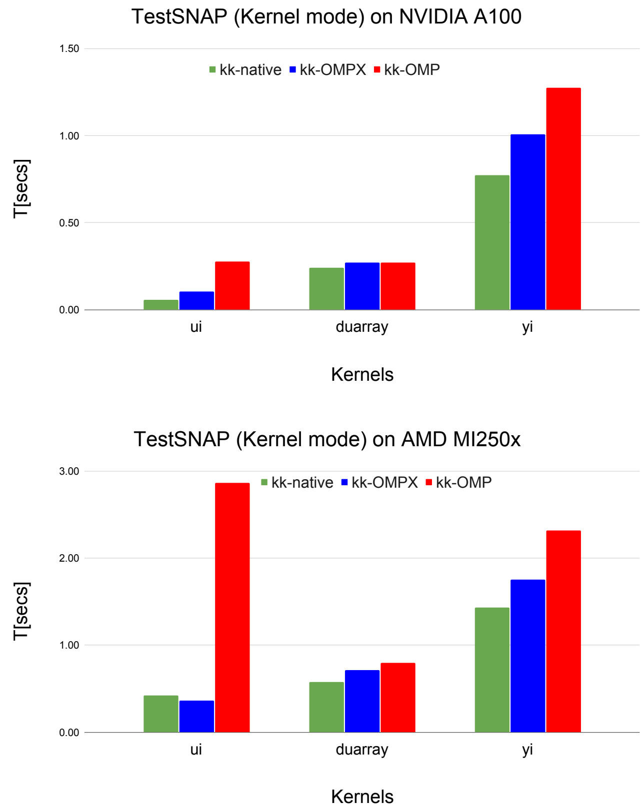


Fig. 13: `TestSNAP` performance with Kokkos-OMPX backend.

32 on NVIDIA A100 and 64 on AMD MI250X. The vector size is a constant across TestSNAP and team sizes are chosen based on the amount of scratch memory that can be requested without oversubscribing the resource.

On NVIDIA A100, the impact of kernel mode is minimal for `duarray`, but using dynamic shared memory (also available in Kokkos-OMP) already improved the performance of the kernel, now only 3-4% slower than the native backend. For `ui`, Kokkos-OMPX is 4$\times$ faster than Kokkos-OMP, although 40% slower than the Kokkos-native backend. However on AMD MI250X, Kokkos-OMPX performs better on `ui` by 15% compared to the Kokkos-native (HIP) backend. Although in `duarray` Kokkos-OMPX is faster than Kokkos-OMP, it is still slower than the native backend. We are currently investigating the slowdowns. The slowdown might be due to the small but still existing LLVM/OpenMP runtime overhead, or differences in generating the optimal kernel parameters, i.e., `max_teams`. The overhead can also be caused by the presence of lambdas as explained in [3].

Kokkos supports multi-dimensional parallelism in closely nested loops using `MDRangePolicy`, passing in the number of nested loops and iteration ranges for each of the loops. `MDRangePolicy` is semantically equivalent to the OpenMP `collapse` clause, and the Kokkos-OMP backend implements it using that clause.

```
1 Kokkos::parallel_for(
2 Kokkos::TeamPolicy<>(num_teams, team_size,
      vector_size),
3 KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::
      member_type &team) {
4  int64_t first_row = ...; // per team
5  int64_t last_row = ...;  // per team
6
7  // iterate over rows owned by this team
8  Kokkos::parallel_for(
9   Kokkos::TeamThreadRange(team, first_row,
        last_row),
10   [&](const int64_t row) {
11   const int64_t row_length = ...;
12   ...
13   double y_row;
14   // reduction over non-zeroes in the row
15   Kokkos::parallel_reduce(
16   Kokkos::ThreadVectorRange(team, row_length),
17    [=](const int64_t i, double &sum) {
18    sum += A.values(i + row_start);
19    ...
20   }, y_row);
21   y(row) = y_row;
22  });
23 });
```

Fig. 14: Kokkos hierarchical parallelism for `SPMV`.

A bonus of kernel mode is that kernel `yi`, which uses `MDRangePolicy`, now exposes more parallelism because the `vector_size` is now 32 and 64. The `yi` kernel performs a Clebsh-Gordon product on the coefficients calculated in `ui`. It is a 3 dimensional perfectly nested loop whose parallelism is exploited using the `collapse` clause from OpenMP where the iteration range of the innermost loop is equal to `vector_size`. Since there was no effective implementation for `simd` used for the `ThreadVector` loop in Kokkos-OMP, we had to restrict `vector_size` to "1" in Kokkos-OMP to maintain correctness. This restriction is eliminated in kernel mode, so the `collapse` clause can now extract more parallelism and performance of `yi` improves by 25% on both NVIDIA A100 and AMD MI250X as shown in Fig. 13.

C. Reductions in OpenMP Kernel Mode

The introduction of kernel mode also allows us to perform inter team reductions using an efficient implementation that can exploit dynamic shared memory and advanced techniques of shuffle instructions. Gayatri et al. [3] used a conjugate gradient solver (CGSolve) application as a vehicle to understand performance differences between the Kokkos-OMP and Kokkos-native backends. It discussed how the performance of CGSolve is heavily dependent on the Sparse Matrix Vector (SPMV) computation which uses three levels of hierarchical parallelism to extract maximum parallelism out of the kernel.

Fig. 14 describes how all three hierarchical parallelism levels are used to implement SPMV. A group of rows is distributed among teams. Within each team, each thread is assigned a set of rows and a reduction over all non-zero elements in each row is performed by vectors in a thread.

The Kokkos-OMPX backend with LLVM/OpenMP extensions preallocates a dynamic shared buffer sized to the number of threads in a team to store partial results for nested reductions. Fig. 15 shows the Kokkos-OMPX backend's allocation

```
1 template <class Functor, class... Properties>
2 class ParallelFor<Functor, Kokkos::TeamPolicy<
      Properties...>, Kokkos::OpenMPTarget> {
3  ...
4  Functor f;
5  void execute() {
6    int scratch_size = shmem_size_L0 + team_size*
        vector_size*sizeof(size_t);
7
8 #pragma omp target teams ompx_bare num_teams(
      max_teams) thread_limit(vector_size,team_size
      ,1) firstprivate(...) ompx_dyn_cgroup_mem(
      scratch_size)
9     ... // Same as Fig. 12
10    }
11 };
```

Fig. 15: Kokkos team specific scratch size.

in scratch memory of one element per thread of a team for team local reductions. Fig. 16 shows how the reduction in Fig. 14 is implemented in the Kokkos-OMPX backend. For simplicity, we omit the fallback code used for compilers that do not support kernel mode.

Fig. 16 shows the implementation of reductions using dynamic shared memory in the native frameworks. Fig. 16 shows the implementation of `parallel_reduce` in KK-OMPX, which has an additional parameter compared to `parallel_for` shown in Fig. 12 to store the final `result`. The pointer to the shared memory buffer is acquired via the LLVM/OpenMP extension `llvm_omp_target_dynamic_shared_alloc` as shown in line 7. The right element within the buffer is indexed by advancing the pointer with the amount of level-0 scratch requested. The first elements in the buffer are used for level-0 scratch memory as shown in Fig. 6. In `SPMV`, `scratch_0` would be zero, since it does not request any scratch memory. The type of the reduction result is abstracted in `ValueType` class. The indexed shared memory is then initialized to the default value and the partial update of each thread is accumulated into the thread specific index. The default constructor for the `ValueType` class can be used to assign an initial value. A sync operation is performed at the end of the partial updates for all threads in a team to finish their work. This would be equivalent to the `__syncthreads` operation available in both CUDA and HIP. The LLVM/OpenMP has been extended by the authors to provide a portable synchronization called `ompx::sync_block_acq_rel`. The partial results are then accumulated by a single thread, `thread-0`, to calculate the final result, which is stored in a location that is accessible to each thread in the team. Kokkos semantics do not require any specific thread to do the final write, and every thread in the ThreadVectorRange shall have the final result to maintain correctness. Ultimately, an optimized implementation would provide a common mask for each set of threads in the X-dimension corresponding to a single thread in the Y-dimension and broadcast to all threads with the same mask.

Fig. 18 shows a performance comparison of SPMV using the Kokkos-OMPX backend against Kokkos-OMP and Kokkos-native backends. There is also a direct OpenMP version that uses LLVM extensions without Kokkos, which we

```
1 template <class Lambda>
2 void parallel_reduce(
3 const ThreadVectorRangeBoundaries<...>&
      loop_boundaries, const Lambda& lambda,
      ValueType& result) {
4  int start = loop_boundaries.start;
5  int end = loop_boundaries.end;
6  size_t scratch_0 = //level-0 scratch;
7  ValueType* buf = static_cast<ValueType*>(
       llvm_omp_target_dynamic_shared_alloc()) +
       scratch_0;
8
9  int blockDimx=ompx::block_dim(ompx::dim_x);
10 int threadIdy=ompx::thread_id(ompx::dim_y);
11 int threadIdx=ompx::thread_id(ompx::dim_x);
12 int gridId = threadIdy*blockDimx+threadIdx;
13 buf[gridId] = ValueType();
14
15 for (int i=start+threadIdx; i<end; i+=blockDimx)
       {
16  ValueType tmp = ValueType();
17  lambda(i,tmp);
18  buf[gridId] += tmp;
19 }
20 ompx::sync_block_acq_rel(); //team sync
21
22 if (threadIdx == 0) {
23  for (int tid = 0; tid < blockDimx; ++tid)
24   vector_reduce += buf[threadIdy * blockDimx +
         tid];
25
26  buf[threadIdy * blockDimx] = vector_reduce;
27 }
28 ompx::sync_block_acq_rel(); //team sync
29
30 // Every thread should have the final value
31 result = buf[threadIdy*blockDimx];
32 }
```

Fig. 16: Kokkos-OMPX ThreadVector reduction.

call “direct-ompx”. The Y-axis of Fig. 18 shows the bandwidth achieved by SPMV. There are three different blocks of bars in the figure to illustrate the impact of problem size. On NVIDIA A100, the Kokkos-OMPX backend performs at-least 15% better than the the upstream Kokkos-OMP backend. The difference between the two versions increases with the amount of data transferred. The Kokkos-OMPX backend is however always slightly less performant compared to the native backend. The best performance was achieved by the direct-ompx version. On AMD MI250X, Kokkos-OMPX backend is at-least 2× more performant than the Kokkos-OMP backend. However the Kokkos-OMPX backend is 2× slower than the Kokkos-native backend and the direct-ompx implementation.

D. Optimizing Occupancy and Shuffle Operations

Remaining performance gaps concern GPU occupancy and shuffle operations. Both native and OpenMP backends use multiple heuristics to improve the occupancy of a Kokkos kernel. One heuristic is based on the maximum number of possible teams that can simultaneously run on a given architecture, such as the number of thread blocks that can be simultaneously scheduled on a single streaming multiprocessor (SM) on NVIDIA GPUs. A kernel is generated using this heuristic to determine the number of teams, implying that a loop is needed to meet the user provided `league_size`. Even if the number of teams generated in the backend is similar to the requested `league_size`, the overhead of a loop is still inevitable.

```
1 #ifdef ompx_shuffle
2  int gridId = threadIdy*blockDimx + threadIdx;
3  vector_reduce = buf[gridId];
4  for (int offset = blockDimx / 2; offset > 0;
         offset /= 2) {
5   vector_reduce += ompx::shfl_down_sync(-1,
          vector_reduce, offset);
6  }
7 #else
8  if (threadIdx == 0) {
9   for (int tid = 0; tid < blockDimx; ++tid)
10    vector_reduce += buf[gridId + tid];
11 }
12 #endif
13 ompx::sync_block_acq_rel(); //team sync
14
15 // Store the final value in a common location
16 if (threadIdx == 0)
17  buf[gridId] = vector_reduce;
18 ompx::sync_block_acq_rel();
19
20 // Every thread have the final value
21 result = buf[threadIdy*blockDimx];
```

Fig. 17: Kokkos shuffle reductions in a team.

To avoid this overhead, we modified both backends to create special instances of kernel generation in which the loop can be eliminated depending on the `league_size` requested.

Kokkos native backends use shuffle primitives from CUDA and HIP to implement optimized reductions. An equivalent interface (`ompx::shfl_down_sync`) has been added as an LLVM OpenMP extension. The parameters to this routine are similar to those in the native frameworks, i.e., mask, value and offset. Fig. 17 shows how the accumulation of values from Fig. 16 can be modified to use the shuffle primitive.

Fig. 19 shows the performance of SPMV when both Kokkos-native and Kokkos-OMPX backends optimize the number of teams generated, i.e., elimination of the loop when `league_size` is at most `max_teams`. The Kokkos-OMPX backend additionally uses shuffle primitives to accumulate updates same as what the direct-ompx version does.

On NVIDIA A100, the Kokkos-OMPX backend sees 15% better performance when running in the optimized mode and reaches the equivalent performance as native backend in cases of higher data transfer. However, the version of SPMV using OpenMP extensions still achieves a 5% higher performance compared to the Kokkos versions. In this scenario we can attribute some overhead to using the Kokkos framework.

On AMD MI250X, the optimized Kokkos-OMPX versions are 50% faster for higher data transfers compared to non-optimized versions. The optimized Kokkos-OMPX backend outperforms the direct-ompx version with LLVM/OpenMP extensions and the Kokkos-native backend. Profiling reveals several key differences between optimized the Kokkos-OMPX version and the direct-ompx version that contribute to their performance difference on AMD MI250X: Kokkos-OMPX uses fewer total registers (64 versus 80) for direct-ompx, resulting in slightly higher occupancy (29.42% vs 26.49%). Kokkos-OMPX also exhibits a higher L2 cache hit rate (65.07% versus 61.65%), indicating more efficient memory access patterns. Inspection of the intermediate representation

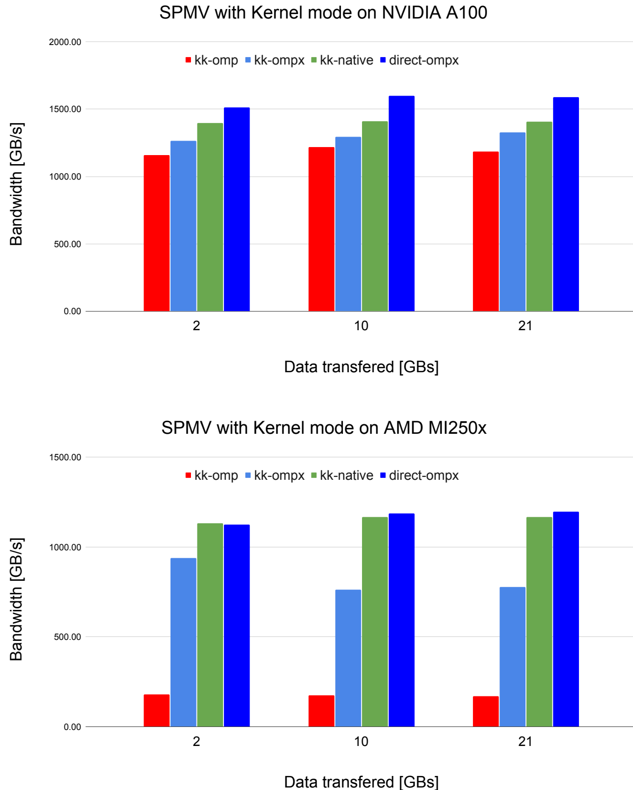


Fig. 18: `SPMV` performance using kernel mode.

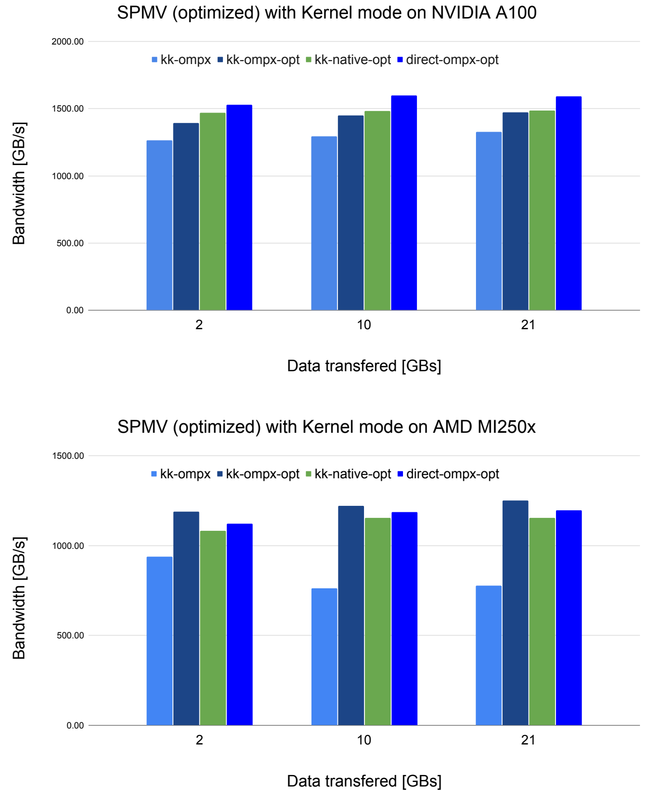


Fig. 19: Optimized `SPMV` performance using kernel mode.

(IR) reveals differences in instruction sequence order rather than significant structural variations, suggesting that the performance gap stems from subtle optimizations rather than fundamental algorithmic differences. This target-dependent performance variation strongly indicates that the observed differences on the AMD platform are likely due to distinct backend optimization pipelines employed by the two targets.

In summary, we have shown how LLVM/OpenMP extensions can be used to optimize Kokkos parallel patterns on GPUs. The new Kokkos-OMPX backend can now match the native backends of Kokkos on NVIDIA and AMD GPUs.

V. Related Work

Due to the high potential performance benefits of its use, GPU shared memory has been a topic of several recent efforts regarding OpenMP offload. Huber et al. [10] describe how recent versions of the LLVM/OpenMP runtime examine variables and determine placement, including use of shared memory where safe and appropriate [10]. Gammelmark et al. [11] show how programs can be structured to allow the compiler to infer that GPU shared memory can be used, in the absence of explicit annotations. Talaashrafi et al. [12] show how the OpenMP runtime library can in some cases automatically make use of GPU shared memory.

Although we focus on NVIDIA and AMD GPUs in this study, TestSNAP can also run on Intel GPUs [13]. Testing our use of LLVM/OpenMP extensions in the Kokkos-OpenMP backend on Intel GPUs is a topic for future work, as upstream LLVM does not currently support OpenMP offload for them.

Kokkos is one of several C++ performance portability frameworks. Others, such as RAJA [14], may also benefit from LLVM-OpenMP extensions using techniques similar to ours. Additionally, since the OpenMP API also supports Fortran, eventual inclusion of the extensions in the specification would also benefit programmers in that language, which currently has few performance portable options besides OpenMP offload.

VI. Conclusion

OpenMP is a widely used parallel programming model with support from open source and vendor compilers. However, adoption of OpenMP offload for GPUs has been more limited. Contributing factors include lack of support for advanced optimization techniques, e.g., use of dynamic memory shared among a team of threads, and heavyweight runtime library requirements compared to the simple multi-dimensional grid model of CUDA and HIP.

In this paper we have discussed two extensions to the LLVM/OpenMP ecosystem for GPUs, 1) the ability to request dynamic shared memory within a team and 2) the option to write SIMT style code in OpenMP. Together the extensions bridge key feature gaps between native frameworks such as CUDA/HIP and OpenMP. We have also shown how the extensions can be combined with additional performance tuning extensions in LLVM/OpenMP such as shuffle instructions to further optimize reductions on GPUs. The result is that users

can leverage OpenMP's wider portability compared to vendor supported frameworks without major performance penalties.

We have demonstrated the use of new extensions in performance portable frameworks through the exemplar of Kokkos, in which we have extended the OpenMPTarget backend to use LLVM/OpenMP extensions when offloading Kokkos execution patterns to GPUs. Using these extensions, the performance of representative programs is now competitive with the native (CUDA/HIP) backends of Kokkos on NVIDIA and AMD GPUs. Our demonstration of these extensions provides motivation for the OpenMP Language Committee to consider adoption of them in the API for eventual availability in other OpenMP implementations beyond LLVM. Moreover, our evaluation of Kokkos with LLVM/OpenMP extensions provides evidence of viability for OpenMP GPU offload in large C++ based projects for performance portability.

ACKNOWLEDGMENT

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award DDR-ERCAP0030946. 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 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-2000576). Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

[1] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[2] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 5.2," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, November 2021.

[3] R. Gayatri, S. L. Olivier, C. R. Trott, J. Doerfert, J. Ciesko, and D. Lebrun-Grandie, "The Kokkos OpenMPTarget backend: Implementation and lessons learned," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 99–113.

[4] K. Parasyris, G. Georgakoudis, E. Rangel, I. Laguna, and J. Doerfert, "Scalable tuning of (OpenMP) GPU applications via kernel record and replay," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607098

[5] A. Patel and J. Doerfert, "Remote OpenMP offloading," in *High Performance Computing*, A.-L. Varbanescu, A. Bhatele, P. Luszczek, and B. Marc, Eds. Springer International Publishing, 2022.

[6] S. Tian, J. Huber, J. R. Tramm, B. M. Chapman, and J. Doerfert, "Just-in-time compilation and link-time optimization for OpenMP target offloading," in *OpenMP in a Modern World: From Multi-device Support to Meta Programming - 18th International Workshop on OpenMP, IWOMP 2022, Chattanooga, TN, USA, September 27-30, 2022, Proceedings*, ser. Lecture Notes in Computer Science, M. Klemm, B. R. de Supinski, J. Klinkenberg, and B. Neth, Eds., vol. 13527. Springer, 2022, pp. 145–158. [Online]. Available: https://doi.org/10.1007/978-3-031-15922-0_10

[7] R. Gayatri, S. Moore, E. Weinberg, N. Lubbers, S. Anderson, J. Deslippe, D. Perez, and A. P. Thompson, "Rapid exploration of optimization strategies on advanced architectures using TestSNAP and LAMMPS," *arXiv preprint arXiv:2011.12875*, 2020.

[8] J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. M. Chapman, and G. Georgakoudis, "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*. IEEE, 2022, pp. 504–514. [Online]. Available: https://doi.org/10.1109/IPDPS53621.2022.00055

[9] S. Tian, T. Scogland, B. Chapman, and J. Doerfert, "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*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 876–883. [Online]. Available: https://doi.org/10.1145/3624062.3624164

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

[11] M. Gammelmark, A. Rydahl, and S. Karlsson, "OpenMP target offload utilizing GPU shared memory," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 114–128.

[12] D. Talaashrafi, M. M. Maza, and J. Doerfert, "Towards automatic OpenMP-aware utilization of fast GPU memory," in *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, M. Klemm, B. R. de Supinski, J. Klinkenberg, and B. Neth, Eds. Cham: Springer International Publishing, 2022, pp. 67–80.

[13] N. A. Mehta, R. Gayatri, Y. Ghadar, C. Knight, and J. Deslippe, "Evaluating performance portability of OpenMP for SNAP on NVIDIA, Intel, and AMD GPUs using the roofline methodology," in *Accelerator Programming Using Directives: 7th International Workshop, WACCPD 2020, November 20, 2020*. Springer, 2021, pp. 3–24.

[14] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.