The Kokkos OpenMPTarget Backend: Implementation and Lessons Learned

Rahulkumar Gayatri[1], Johannes Doerfert[2], Jan Ciesko[3], Stephen L. Olivier[3], Christian R. Trott[3], and Damien Lebrun-Grandie[4]

[1] Lawrence Berkeley National Laboratory, Berkeley, CA, USA rgayatri@lbl.gov
[2] Lawrence Livermore National Laboratory, Livermore, CA, USA
jdoerfert@llnl.gov
[3] Sandia National Laboratories, Albuquerque, NM, USA
{jciesko, slolivi, crtrott}@sandia.gov
[4] Oak Ridge National Laboratory, Oak Ridge, TN, USA
lebrungrandt@ornl.gov

Abstract. As the supercomputing landscape diversifies, solutions such as Kokkos to write vendor agnostic applications and libraries have risen in popularity. Kokkos provides a programming model designed for performance portability, which allows developers to write a single source implementation that can run efficiently on various architectures. At its heart, Kokkos maps parallel algorithms to architecture and vendor specific backends written in lower level programming models such as CUDA and HIP. Another approach to writing vendor agnostic parallel code is using OpenMP's directives based approach, which lets developers annotate code to express parallelism. It is implemented at the compiler level and is supported by all major high performance computing vendors, as well as the primary Open Source toolchains GNU and LLVM. Since its inception, Kokkos has used OpenMP to parallelize on CPU architectures. In this paper, we explore leveraging OpenMP for a GPU backend and discuss the challenges we encountered when mapping the Kokkos APIs and semantics to OpenMP target constructs. As an exemplar workload we chose a simple conjugate gradient solver for sparse matrices. We find that performance on NVIDIA and AMD GPUs varies widely based on details of the implementation strategy and the chosen compiler. Furthermore, the performance of the OpenMP implementations decreases with increasing complexity of the investigated algorithms.

Keywords: Kokkos · OpenMP · GPUs · parallel programming · performance portability.

1 Introduction

As the high performance computing community enters the exascale computing era, the largest supercomputers are dominated by GPU accelerated system designs. For almost a decade, these platforms, including the latest NERSC system, Perlmutter, exclusively deployed GPUs from NVIDIA. This single vendor trend

is changing with the first deployed exascale machines. The recently launched Frontier system at Oak Ridge National Laboratory and the upcoming El Capitan platform at Lawrence Livermore National Laboratory are based on a system design using AMD GPUs, while Argonne National Laboratory's Aurora supercomputer will use Intel GPUs.

A challenge arising from this architecture diversity is that each vendor has their own preferred programming model. NVIDIA provides CUDA, first introduced in 2007. AMD developed the HIP programming model, which is closely modelled after CUDA. Data Parallel C++ (DPC++), an extension of Khronos' SYCL, is Intel's preferred choice for implementing code on their GPUs. Writing applications and libraries directly in each vendor's preferred programming model thus requires the implementation of four versions, assuming one would want to support multicore CPU execution as well.

To eliminate this unmanageable software development and maintenance overhead, vendor independent higher-level frameworks such as Kokkos [1] and RAJA/Chai/Umpire [2] were developed. These frameworks promise performance portability by providing a common interface for expressing parallelism and data management, which is then mapped to the vendor specific programming models.

There are also efforts to make the vendor specific models portable across architectures. SYCL itself is designed as a hardware agnostic programming model, and Intel's DPC++ compiler has the ability to target NVIDIA GPUs and to a lesser degree AMD GPUs. AMD's HIP model can be mapped to CUDA by coupling AMD's toolchain to NVIDIA's. Community research efforts in LLVM are also working to compile CUDA to other architectures [3]. However, in practice there are very few projects relying on these portability efforts of the vendor models, due to concerns over full support on all architectures. In particular, support contracts which are part of the large supercomputing procurements generally only cover the vendor's own toolchain. The portability frameworks do not have the same issue, since they leverage the native toolchains on each architecture.

OpenMP is the one vendor independent node-level programming model standard which all the vendors support to varying degrees, and which is generally part of the contractual requirements in the large supercomputing procurements. Furthermore, it is not only supported by vendor specific compilers, but also by the two primary open source toolchains, LLVM and GCC. OpenMP uses a directive based approach, which allows developers to annotate existing code to express parallelism. This approach has been used to good effect on CPU based systems for two decades. Since version 4.0, OpenMP has also supported directives for accelerators such as GPUs, and those directives have evolved significantly with subsequent versions. However, the available subset of the specification, the quality of implementation of those subsets, and even the interpretation of intended behavior of some features are different in each toolchain, causing challenges when using OpenMP for performance portability.

In this paper we explore these challenges using the effort of porting Kokkos to use OpenMP as a hardware independent backend implementation. That effort was conceived as a means to provide for Kokkos a second toolchain path on

each platform, in addition to the vendor specific programming models. Having multiple toolchains, and specifically compilers, available on each system allows for redundancy and more overall robustness of the software stack. It also prepares Kokkos for a situation where a new hardware vendor may not develop a unique programming model, leveraging the OpenMP specification instead.

In this paper we will use the conjugate gradient solver (CG-Solve) described in [1] as an exemplar to discuss various concepts in Kokkos, how they are mapped to OpenMP, and the challenges which arise. The results shown in this paper show the performance achieved by the CG-Solve example and its individual kernels on NVIDIA A100 GPUs available on Perlmutter and AMD MI250x available on Crusher (testbed for Frontier). We use the latest clang compiler from the main branch of llvm (***date***) and vendor specific compilers for each of the GPUs, i.e., NVHPC/22.7 on A100 and amdclang available with rocm/5.4.3 on MI250x. We will refer to these as LLVM, NVHPC and ROCM respectively. This effort is not an attempt to find the very best implementation of CG-Solve, nor to improve upon the existing math algorithms. Specifically we are not exploring the use of different sparse matrix storage formats or various possible parallelization schemes for the algorithms. This paper is primarily concerned with the question of how Kokkos usage of OpenMP compares to native OpenMP implementations and how the OpenMP offload implementation compares to the use of native CUDA and HIP backends in Kokkos, given a specific algorithm and parallelization strategy.

2 CG-Solve

The congujate gradient solver (CG-Solve) is a simple iterative linear solver, which use three primary linear algebra functions: a vector addition (`axpby`), an inner product (`dot`) and a sparse matrix vector multiply (`spmv`). In each iteration the `axpby` is called four times, the `dot` twice and the `spmv` once. Listing 1.1 shows the pseudo code for the solver. The three operations exhibit three common patterns found in data parallel programming: simple data parallel loops, reductions, and nested loops. The overall algorithm is largely bandwidth limited. However the pure vector operations are often latency sensitive on GPU systems, since at typically observed vector lengths of 100,000 to 1,000,000 entries per device the vector operations can execute in under 20us there. Furthermore, `axpby`, `dot` and `spmv` are not just important for CG-Solve, but are also the fundamental building blocks in most other linear solvers.

Listing 1.1: CGSolve

```
for (int64_t k = 1; k <= max_iter && normr > tolerance; ++k) {
  if (k == 1) {
    axpby(p, one, r, zero, r);          // AXPBY
  } else {
    oldrtrans = rtrans;
    rtrans = dot(r, r);                 // DOT
    double beta = rtrans / oldrtrans;
    axpby(p, one, r, beta, p);          // AXPBY
  }
  normr = std::sqrt(rtrans);
```

```
  double alpha    = 0;
  double p_ap_dot = 0;
  spmv(Ap, A, p);                          // SPMV
  p_ap_dot = dot(Ap, p);                   // DOT
  if (p_ap_dot < brkdown_tol) {
    if (p_ap_dot < 0) {
      std::cerr << "miniFE::cg_solve␣ERROR,␣numerical␣breakdown!"
                << std::endl;
      return num_iters;
    } else
      brkdown_tol = 0.1 * p_ap_dot;
  }
  alpha = rtrans / p_ap_dot;
  axpby(x, one, x, alpha, p);              // AXPBY
  axpby(r, one, r, -alpha, Ap);            // AXPBY
  num_iters = k;
}
```

For the rest of the paper, we will discuss the Kokkos implementation of `axpby`, `dot` and `spmv`, how they can be mapped to OpenMP, and the challenges encountered.

2.1 AXPBY

The vector addition is a simple data parallel loop, with no dependencies between iterations. It is straightforward to express in most programming models, including Kokkos.

Listing 1.2: Kokkos Vector Addition (`axpby`)

```
void axpby (double a, Kokkos::View<double*> x,
            double b, Kokkos::View<double*> y) {
  Kokkos::parallel_for("AXPBY", x.extent(0), KOKKOS_LAMBDA(const int i) {
      y(i) = a*x(i) + b*y(i);
  });
}
```

A Kokkos `View` expresses a possibly multi-dimensional array. This function only uses its simplest version representing a plain one-dimensional contiguous vector. The Kokkos `parallel_for` execution pattern expresses a parallelizable loop. It takes as arguments a label (for debugging and profiling purposes), an iteration range, and the loop body expressed through a C++ lambda. Kokkos is a descriptive programming model, which does not guarantee any specific implementation strategy on architectures. Its parallel loops do not imply order nor concurrency, and thus can be mapped to thread, vector or pipeline parallelism.

An equivalent OpenMP implementation of `axpby` for GPUs (assuming manual data management) is given in Listing 1.3.

Listing 1.3: OpenMP Vector Addition (`axpby`)

```
void axpby (int N, double a, double* x,
                   double b, double* y) {
  #pragma omp target teams distribute parallel for simd nowait is_device_ptr(
      x,y)
  for(int i=0; i< N; i++) {
      y[i] = a*x[i] + b*y[i];
  }
}
```

In its implementation of `parallel_for`, Kokkos uses a partial specialization approach, where the lambda is handed to a backend specific implementation of the parallel loop. Simplified, this strategy looks like the code in Listing 1.4.

Listing 1.4: `parallel_for` OpenMPTarget backend

```
template<Functor>
struct ParallelFor<Functor, OpenMPTarget> {
  int N; Functor f;
  void execute() {
    #pragma omp target teams distribute parallel for simd nowait
    for(int i=0; i< N; i++) { f(i); }
  }
};

template<class Functor>
void parallel_for(string label, int N, Functor f) {
  ParallelFor<Functor, OpenMPTarget> closure{N,f};
  closure.execute();
}
```

Note that the only fundamental difference between the direct OpenMP implementation and the Kokkos backend implementation is the expression of the loop body via a C++ lambda. However, we have observed that the OpenMP compilers are very sensitive to the use of seemingly unrelated C++ patterns. Specifically, significant performance difference can be observed when writing algorithms in two different – but from the C++ perspective equivalent – ways. One such instance is the use of C++ lambdas. To illustrate that difference, we measured performance also for versions of the algorithms written directly in OpenMP, but using lambdas, as shown in Listing 1.5.

Listing 1.5: OpenMP Vector Addition as C++ lambda(axpby)

```
void axpby (int N, double a, double* x,
                   double b, double* y) {

    auto f = [=](i) {y[i] = a*x[i] + b*y[i];};
  #pragma omp target teams distribute parallel for simd nowait firstprivate(f
       )
  for(int i=0; i< N; i++) {
      f(i);
  }
}
```

A similar issue occurs with use of OpenMP target regions inside class member functions. When the `axpby` is implemented as a class member function, where `N` is a class data member, performance drops even more than with the use of lambdas, compared to creating a local copy of `N` inside the member function.

Figure 1 shows the performance of the different versions of `axpby` discussed above. For this kernel, we see that the direct OpenMP code when compiled with the vendor compilers can achieve almost the same performance as Kokkos with the native CUDA/HIP backends. At larger vector lengths, the Kokkos OpenMPTarget backend approaches the raw OpenMP performance, and most of the difference can be explained by the previously noted issues around the use of Lambdas. However, nvc++ does not exhibit the lambda specific performance penalty, and the Kokkos OpenMPTarget backend in each case achieves the same performance as the lambda OpenMP implementation. Comparing the relative

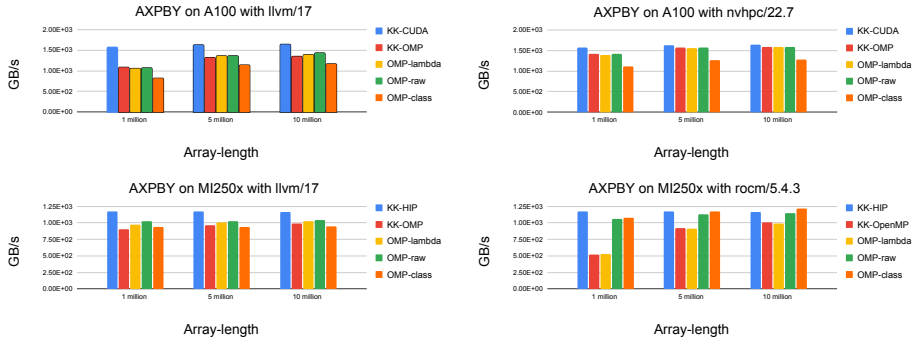

Fig. 1: AXPBY on NVIDIA A100 with llvm and nvhpc compilers and on AMD MI250x with llvm and amdclang compilers.

performance of the different implementations on the two different architectures, they appear to be a function of the compiler rather than the hardware.

2.2 DOT

The `dot` product kernel performs a single range reduction on a given data type. In Kokkos this is expressed with the the `parallel_reduce` pattern as shown in Listing 1.6.

Listing 1.6: Kokkos Reduction (dot)

```
double dot(Kokkos::View<double*> x, Kokkos::View<double*> y) {
  double result = 0.;
  Kokkos::parallel_reduce("DOT", x.extent(0), KOKKOS_LAMBDA(const int i,
       double &lsum) {
      lsum += x(i) * y(i);
  }, result);
  return result;
}
```

The equivalent direct OpenMP code is shown in Listing 1.7.

Listing 1.7: OpenMP Reduction (dot)

```
void dot (int N, double* x, double* y) {
  double result = 0.;
  #pragma omp target teams distribute parallel for simd reduction(+:result)
       is_device_ptr(x,y)
  for(int i=0; i< N; i++) {
      result += x[i] * y[i];
  }
  return result;
}
```

Figure 2 shows the bandwidth achieved by the `dot` kernel.

Only ROCM achieves the same performance as the native backends of Kokkos, and only in the absence of lambdas which otherwise reduce performance by 4-8x depending on the vector length. Here LLVM and NVHPC are not sensitive to the use of Lambdas. Still, with OpenMP, they only achieve between 30% and 70%

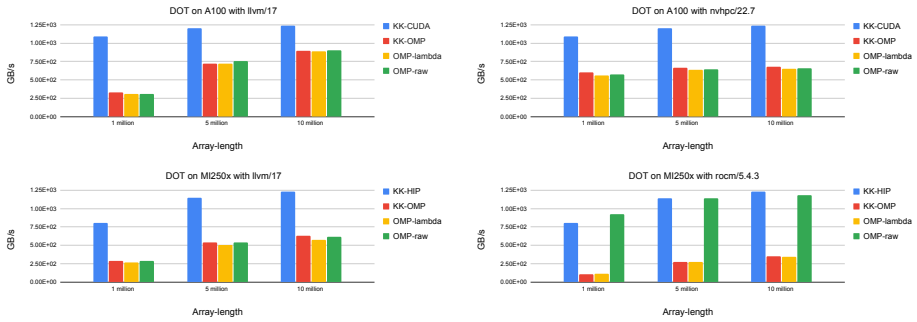

Fig. 2: DOT on NVIDIA A100 with llvm and nvhpc compilers and on AMD MI250x with llvm and amdclang compilers.

of the performance of the native backends. Unlike the `axpby` results, NVHPC only reaches about 50% of the CUDA backend performance. A 2022 paper documenting the current design of the LLVM OpenMP runtime [4] remarks that recent improvements of that runtime have not included any work on better implementations of GPU reductions, but our understanding is that some vendors are working on this topic.

2.3 SPMV

The third algorithm needed for CG-Solve is a sparse-matrix vector multiply. Sparse matrices can be represented in different ways, here we employ the commonly used compressed sparse row (CSR) representation, which uses an array storing the non-zero values of the matrix, an array with the associated column indicies, and a vector storing the row offsets into the value and column index arrays.

At its simplest the `spmv` can then be implemented as loop over rows, with a nested reduction to compute the `dot` product of each row. Listing 1.8 provides a simple implementation of the `spmv` algorithm.

Listing 1.8: SPMV-Algorithm

```
for(row = 0; row < num_rows; row++) { // Loop over all rows
  row_start = row_offsets[row];
  row_end = row_offsets[row+1];
  // Reduction over non-zeros in each row
  for(idx = row_start; idx<row_end; idx++)
    y(row) += m_values[idx] * x[m_cold_idx[idx]];
}
```

This kernel is more complex than either `axpby` or `dot` since for good performance on GPUs nested parallelism must be exposed. The nested parallelism helps expose more concurrency in the algorithm, and gains in importance with increasing number of non-zeros per row. Due to the nature of the loops, where the inner loop's trip count depends on the outer loop's iteration index, they can not be easily collapsed. Furthermore, the kernel exhibits a mix of streaming and

irregular data access. The matrix data is accessed continuously, while accesses of the `x` vector are irregular.

In practice Kokkos implements a somewhat more complex version using three levels of parallelism. Often the number of non-zeros per row, and thus the inner loop length, is fairly small. In that case we want to use only vector parallelism to perform the reduction, but still want to group adjacent rows in threads sharing a common cache, to exploit data access locality of the vector `x`.

Both Kokkos and the OpenMP specification support three levels of parallelism using the concept of team of threads, threads and vector parallelism. In Kokkos this is achieved using special execution policies with the execution patterns, namely `TeamPolicy`, `TeamThreadRange`, and `ThreadVectorRange`. OpenMP expresses the same conceptual ideas with the `teams distribute`, `parallel for`, and `simd` constructs.

However, the LLVM compiler, as many vendor compilers, including NVHPC and ROCM, treat `simd` as a hint, and do not map it to hardware parallelism. All threads in a GPU CUDA block or HIP group are instead activated together as part of the `parallel for` construct. This restriction, for now, limits the performance for any Kokkos application that uses the 3rd parallel level explicitly. That said, explicit control over the placement, including a dedicated three level mapping, is currently under development as part of LLVM.

Listing 1.9 shows the implementation of SPMV using hierarchical execution patterns in Kokkos. The `Kokkos::TeamPolicy` is used to specify the number of teams, team size and the number of vector lanes used per thread. For this algorithm the team size and the vector length are optimization parameters, which need to be tuned for each hardware platform. When using the CUDA or HIP backend, each team is mapped to a block, with the thread ids in each team mapped to `threadIdx.y` and vector lanes mapped to `threadIdx.x`. Vector lengths are limited by the warp or wavefront size respectively.

In the `spmv` algorithm, each team gets assigned a number of rows, which are then iterated over in parallel by the threads of the team. The nested reduction is performed by the vector lanes associated with each thread.

Listing 1.9: Kokkos Hierarchical Parallelism (SPMV)

```
Kokkos::parallel_for(
 "SPMV",
 Kokkos::TeamPolicy<>(num_teams, team_size, vector_size),
 KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::member_type &team) {
     const int64_t first_row = team.league_rank() * rows_per_team;
     const int64_t last_row = first_row + rows_per_team < nrows
                                  ? first_row + rows_per_team
                                  : nrows;
     // iterate over rows owned by this team
     Kokkos::parallel_for(
         Kokkos::TeamThreadRange(team, first_row, last_row),
         [&](const int64_t row) {
             const int64_t row_start = A.row_ptr(row);
             const int64_t row_length =
                 A.row_ptr(row + 1) - row_start;

             double y_row;
             // reduction over non-zeroes in the row
             Kokkos::parallel_reduce(
                 Kokkos::ThreadVectorRange(team, row_length),
```

```
                [=](const int64_t i, double &sum) {
                    sum += A.values(i + row_start) *
                           x(A.col_idx(i + row_start));
                },
                y_row);
            y(row) = y_row;
        });
});
```

A direct mapping of the Kokkos semantics to OpenMP leads to an implementation as shown in Listing 1.10

Listing 1.10: OpenMP Hierarchical Parallelism - b(SPMV)

```
int num_teams = (nrows + rows_per_team - 1)/rows_per_team;
#pragma omp target teams distribute is_device_ptr(x,y,A_row_ptr,A_values,
     A_col_idx)
for(int team = 0; team < num_teams; ++i)
#pragma omp parallel
{
     const int64_t first_row = omp_get_team_num() * rows_per_team;
     const int64_t last_row = first_row + rows_per_team < nrows ? first_row +
          rows_per_team : nrows;
     #pragma omp for
     for(int row = first_row; row < last_row; ++row)
     {
        const int64_t row_start = A_row_ptr[row];
        const int64_t row_length = A_row_ptr[row + 1] - row_start;

        double y_row;
        #pragma omp simd reduction(+:y_row)
        for(int i = 0; i < vector_size; ++i)
        {
            y_row += A_values[i + row_start] * x[A_col_idx[i + row_start]];
        }
        y[row] = y_row;
    }
}
```

In Kokkos, the loop body of the outer loop is executed by all threads within the team. This is achieved in OpenMP by opening a `parallel` scope inside the outer loop. Now every thread computes redundantly `first_row` and `last_row` - avoiding an otherwise necessary broadcast upon entering the nested parallel loop. The nested reduction is marked as intended to be vectorized with the `simd` directive.

As stated above, none of the compilers used for this work actually parallelize that `simd` loop. In order to identify how much of a performance reductions is caused by that lack of parallelization we also ran the native CUDA/HIP Kokkos backend code with a vector-size of one.

When implementing the Kokkos backend for OpenMPTarget, we encountered a number of additional issues which affect features of Kokkos' hierarchical parallelism not necessary for `spmv`. For example, Kokkos allows reductions on the team-thread level. In OpenMP, one is not allowed to simply add a reduction clause on a local variable to the `parallel for` construct as shown in Listing 1.10, because reduction variables need to be declared at the target level. However, in typical code, it is not always possible to identify such reduction variables at the Kernel dispatch point, since the nested reduction may occur in other functions.

To work around this limitation the Kokkos OpenMPTarget backend proactively allocates a memory buffer on the device memory and annotates it with

the `reduction` clause. Nested reductions then use that memory buffer as reduction variables. However, this workaround also necessitates adding a `num_threads` clause to the `teams distribute` construct to ensure that enough buffer space was allocated. Unfortunately adding that clause reduces the performance on some compilers, even in the cases - such as `spmv` - where it wasn't strictly needed. We measured the impact of adding the `num_threads` clause for `spmv` separately.

Furthermore, on NVIDIA and AMD GPUs an implementation strategy of Kokkos' hierarachical parallelism without the `distribute` construct turned out to be more performant. This strategy requires the loop over worksets to be a nested loop inside the target region as shown in Listing 1.11. Currently this is the default implementation strategy for the Kokkos OpenMPTarget backend on NVIDIA and AMD GPUs, while on Intel GPUs the implementation is similar to Listing 1.10.

Listing 1.11: OpenMP Hierarchical Parallelism - a(SPMV)

```
#pragma omp target teams num_teams(leage_size) thread_limit(team_size)
    is_device_ptr(x,y,A_row_ptr,A_values,A_col_idx)
#pragma omp parallel
    {
      const int blockIdx = omp_get_team_num();
      const int gridDim  = omp_get_num_teams();

      for (int league_id = blockIdx; league_id < num_teams; league_id +=
          gridDim) {
        #pragma omp for
        for(int row = first_row; row < last_row; ++row)
        {
            // similar to above
        }
      }
    }
```

Figure 3 shows the performance of the `spvm` on NVIDIA A100 and AMD MI250x GPUs. As with the previous algorithms, KK-CUDA/KK-HIP performance is significantly greater than any of the OpenMP variants. How, much however depends on the compiler, hardware and the specific variant of the OpenMP code. The experiment highlights the sensitivity of the OpenMP performance to specific implementation choices, with different ones resulting in better performance on different hardware and compiler combinations.

For example, not using the `num_teams` clause improves the performance on A100 when using the LLVM compiler, but reduces the performance dramatically with NVHPC. While with the LLVM compiler, implementation KK-OMP-A, based on Listing 1.11, generally performs better than KK-OMP-B, based on Listing 1.10, it performs exceptionally bad when using the ROCM compiler. The Kokkos OpenMPTarget backend - without the use of the `num_teams` clause - performs as well as the raw OpenMP implementation when using the LLVM compiler, but runs significantly slower when using the ROCM compiler.

In practice these variations make it difficult to maintain an OpenMP code with consistent performance across different platforms.

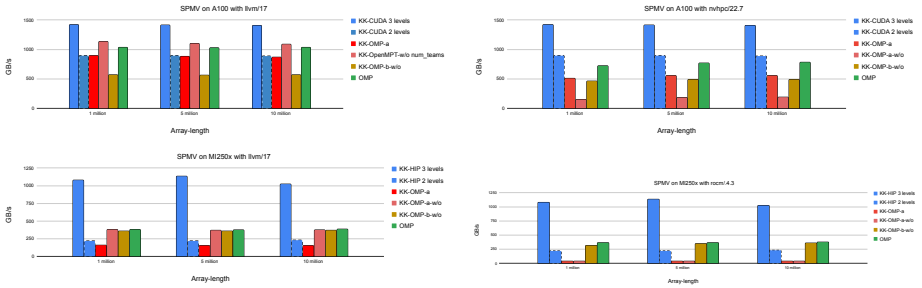

Fig. 3: SPMV on NVIDIA A100 with llvm and nvhpc compilers and on AMD MI250x with llvm and amdclang compilers .

3 Beyond the basics

Besides the already discussed issues with mapping Kokkos to OpenMP there are a number of other challenges we discuss briefly in this section. These challenges did not impact the CG-Solve example, but are of great concern when implementing more complex applications.

3.1 Scratch memory

Kokkos' hierarchical parallelism provides the ability to allocate team and thread private scratch pads, which act as fast user-managed cache. These scratch pads can be mapped to CUDA and HIP shared memory, and generally are useful for cooperative work within a thread-team. In principal the OpenMP specification has the concept of allocators which conceivably would be able to address part of the problem. However, currently this is not implemented by the compilers. Furthermore, in order to leverage aforementioned CUDA and HIP shared memory, the allocation size needs to be specified upon entry into a target region - something the OpenMP specification does not provide a mechanism for.

3.2 Concurrency

Another concept in Kokkos which is difficult to reliably implement is the idea of hardware concurrency. For example when users require private worksets for each active thread in an algorithm, they need to have the ability to acquire a unique workset, and also to size the number of worksets appropriately before launching a kernel. Kokkos' execution space concurrency provides that latter size. This is used in conjunction with Kokkos' `UniqueToken`, a locking mechanism allowing a caller to acquire a unique index.

OpenMP has directives such as `omp_get_max_threads` which can be used to address these issues. Currently the backend uses a mix of hardware knowledge, OpenMP directives on architecture;compiler combinations where they are supported to make an educated guess.

Since the OpenMPTarget backend allocates memory based on the assumptions on the number of active teams, there is a need to have a tight control on the number of teams generated. This is achieved using the `num_teams` clause. However the use of this clause restricts the ability of compilers to optimize on the occupancy of a GPU. The hack to avoid this by using `omp_set_num_teams` and assigning a huge value to it was unsuccessful as most of the compilers did not respect this routine.

We hope to converge onto a single cohesive portable solution on this issue in collaboration with various compiler teams that share similar interests as us.

3.3 Custom Reductions

Kokkos allows users to provide custom reduction operators, similar to how the C++ standard provides a capability to specify reducers in algorithms such as `std::reduce` and `std::transform_reduce`. While OpenMP has a mechanism to implement such custom reductions, which is based on free functions taking the value arguments. In Kokkos and C++ the reduction operator can be a stateful object. This is not possible in OpenMP. As a consequence the Kokkos OpenMPTarget backend only supports the use of pre-defined reduction operators, but not user provided ones.

4 Conclusion

In this paper we described the aspects of mapping the Kokkos Performance Portability model to OpenMP for GPUs. Using a simple linear solver we explore the state of the Kokkos OpenMPTarget backend on NVIDIA and AMD GPUs with multiple compilers. We find that the OpenMPTarget backend provides significantly less performance than the architecture specific CUDA and HIP backends, due to a mix of compiler implementation issues and limitations in the standard. On average the OpenMP variants (inlcuding Kokkos OpenMPTarget backend and raw OpenMP code) provide 57% of the CUDA and HIP backend, but at its worst it is about 30x slower than the HIP backend. An important observation is that the performance of the OpenMP implementation is very sensitive to particular construct choices, but that the effect of these choices depends on both hardware and compiler. This makes it difficult to write and maintain code which performs consistently across different platforms. Extending OpenMP testing and verification suites to include performance testing across different hardware and compilers could help improve this situation, identify regressions in implementations and help develop best practices.

5 Acknowledgements

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.

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 research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.
2. 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)*. IEEE, 2019, pp. 71–81.
3. J. Doerfert, M. Jasper, J. Huber, K. Abdelaal, G. Georgakoudis, T. Scogland, and K. Parasyris, "Breaking the vendor lock: Performance portable programming through OpenMP as target independent runtime layer," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022*, A. Klöckner and J. Moreira, Eds. ACM, 2022, pp. 494–504. [Online]. Available: https://doi.org/10.1145/3559009.3569687
4. J. Doerfert, A. Patel, J. Huber, S. Tian, J. M. M. Diaz, B. 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, pp. 504–514.