This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2023-07871C

The Kokkos OpenMPTarget Backend: Implementation and Lessons Learned

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

[1] Lawrence Berkeley National Laboratory, Berkeley, CA, USA `rgayatri@lbl.gov`
[2] Sandia National Laboratories, Albuquerque, NM, USA
`{slolivi, crtrott, jciesko}@sandia.gov`
[3] Lawrence Livermore National Laboratory, Livermore, CA, USA
`jdoerfert@llnl.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 high performance computing 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

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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-NA0003525.

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 use AMD GPUs, while Argonne National Laboratory's Aurora supercomputer will use Intel GPUs.

A challenge arising from this architectural 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 the Khronos SYCL standard [7], 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 [2, 12, 11] and RAJA [1] 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 [10] 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 [9], 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. Additionally, other performance portability frameworks have explored the use of OpenMP offloading in their backend implementations [8, 6].

In this paper we use the conjugate gradient solver (CG-Solve) described in [12] as an exemplar to discuss various concepts in Kokkos, how they are mapped to OpenMP, and the challenges which arise. The results demonstrate 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 (dated 5/15/2023) 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.

Our CG-solve exemplar is not an attempt to present 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 the 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. Also, note that while we have used CG-solve as vehicle to present issues arising when mapping Kokkos to OpenMP, the actual Kokkos backend must be robust and applicable to a wide variety of applications built upon Kokkos. Therefore, optimizations, e.g., OpenMP settings, that may benefit CG-solve but are not be universally appropriate would not be considered for inclusion in the Kokkos OpenMPTarget backend.

2 CG-Solve

The conjugate gradient solver (CG-Solve) [5] 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 many 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;
}
```

The remainder of this section discusses the Kokkos implementation of `axpby`, `dot` and `spmv`, mapping them to OpenMP, and the challenges we encountered.

2.1 AXPBY

The vector addition (`axpby`) function in CG-Solve 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 (`axpby`) as C++ lambda

```
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 the 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. The figure shows 5 versions AXPBY, where the labels on the legends represent the following:

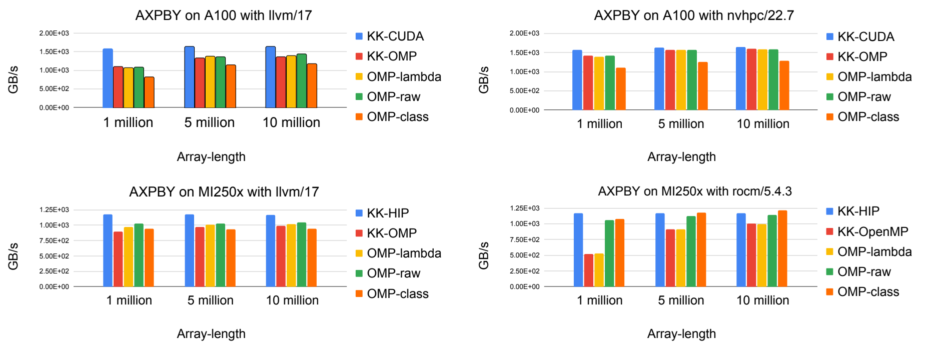

Fig. 1: `AXPBY` on NVIDIA A100 with LLVM and NVHPC compilers and on AMD MI250x with LLVM and ROCM compilers. Y axis is in GB/s, so higher is better.

1. *KK-CUDA* : Kokkos version with the CUDA backend
2. *KK-OMP* : Kokkos version with the OpenMPTarget backend
3. *OMP-lambda* : Direct OpenMP version using lambda inside a `target` region
4. *OMP-raw* : Direct OpenMP version not using lambda inside a `target` region
5. *OMP-class* : Variant of *OMP-raw* version using a class member inside the `target` region, instead of its equivalent local copy.

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 OpenMP-Target 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, NVHPC 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 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 (`dot`) function performs a single reduction on a given data type. In Kokkos this operation is expressed using the `parallel_reduce` pattern as shown in Listing 1.6. The equivalent direct OpenMP code is shown in Listing 1.7.

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;
}
```

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;
}
```

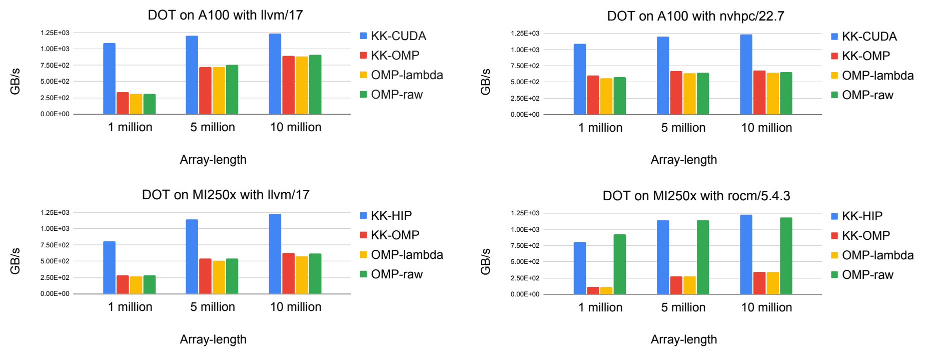

Fig. 2: `DOT` on NVIDIA A100 with LLVM and NVHPC compilers and on AMD MI250x with LLVM and ROCM compilers. Y axis is GB/s, so higher is better.

Figure 2 shows the bandwidth achieved by the `dot` kernel, with legend labels following the naming in Figure 1. 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% of the performance of the native backends. Unlike the `axpby` results, NVHPC with OpenMP 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. Numerous sparse matrices representations exist. Here we employ the common compressed sparse row (CSR) representation, which comprises 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: Sparse matrix vector multiply (`spmv`) sequential 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 operation is more complex than either `axpby` or `dot` since for good performance on GPUs, nested parallelism must be exploited. The nested parallelism exposes more concurrency in the algorithm, which becomes more important with increasing number of non-zeros per row. Since 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.

While the basic `spmv` algorithm requires only two loops, In practice Kokkos implements a somewhat more complex version using three levels of parallelism to expose appropriate amounts of work for each level of the GPU hierarchy. Often the number of non-zeros per row, and thus the inner loop length, is fairly small. Thus it is beneficial to use only the third and innermost level of parallelism to perform the reduction, but to still 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 concepts of teams, threads and vector parallelism. Kokkos provides 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. Both Kokkos and many OpenMP compilers are consistent in mapping the first level of parallelism across streaming multiprocessors (SMs) or compute units (CUs) and the second level of parallelism within SMs or CUs.

Differences between Kokkos and many OpenMP compilers arise regarding the third level of parallelism (or lack thereof). While conceptually the single instruction multiple data (SIMD) model of lock-step execution exemplified by CPU vectorization is stricter than the single instruction multiple threads (SIMT) model of GPUs, SIMD can be profitably mapped onto SIMT and indeed lockstep execution at the lowest level of the GPU's hierarchy can be the most performant. However, the LLVM compiler, and many vendor compilers, including NVHPC and ROCM, treat OpenMP's `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 third parallel level explicitly. Moreover, the third level of parallelization enables efficient memory coalescing on GPU architectures that Kokkos works to exploit. When that third level of parallelism is not present, the memory references are not coalesced, resulting in inefficient access patterns. That said, a dedicated three level mapping honoring the `simd` construct is currently under development as part of LLVM.

Listing 1.9: Kokkos Hierarchical Parallelism for `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;
         });
 });
```

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 that require tuning for each hardware platform. When using the CUDA or HIP backend, each team is mapped to a block, with the thread identifiers 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 is 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.

A direct mapping of the Kokkos semantics to OpenMP leads to an implementation as shown in Listing 1.10 In Kokkos, the loop body of the outer loop is executed by all threads within the team. This is achieved in OpenMP by a `parallel` region 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 annotated with the `simd` directive. As stated above, none of the compilers used for this work actually parallelize the `simd` loop for a GPU. 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.

There are other idioms of hierarchical parallelism where there is a mismatch of Kokkos and OpenMP semantics. Though not illustrated in CG-solve, Kokkos allows a team level reduction over a variable that is introduced within the team.

The semantics are that each thread has copy of the variable that is initialized to the identity at the start of the reduction operation, and the final partial values of all copies are combined and the resulting value redistributed to all threads' copies at the end of the reduction operation. In contrast, OpenMP reduction semantics require that the reduction variable must be shared by the threads in a team and hence it must be known at the start of the `parallel` region. However, in some use cases, it may not be possible to identify such reductions at the start of the `parallel` region, since the nested reduction may occur in other functions.

Listing 1.10: OpenMP Hierarchical Parallelism `spmv` - Version A

```
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;
    }
}
```

Listing 1.11: OpenMP Hierarchical Parallelism `spmv` - Version B

```
#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
        }
      }
    }
```

The native Kokkos backends implement the team level reduction using a memory buffer in device memory. Due to the mismatch in Kokkos and OpenMP semantics, this approach is also currently used for the OpenMPTarget backend. This workaround requires explicit control of the number of active teams using the `num_teams` clause to ensure that the correct amount of buffer space is allocated. Unfortunately adding that clause reduces the performance of Kokkos hierarchical parallelism on some compilers, even in the cases, such as `spmv`, where team level reductions are not present. We measured the impact of adding the `num_teams` clause for `spmv` in our experiments.

We also considered an alternative implementation strategy of Kokkos' hierarchical parallelism without the `distribute` construct that performs better in many cases. This strategy requires the loop over worksets to be a nested loop inside the target region as shown in Listing 1.11. Currently this approach is the default implementation strategy for the Kokkos OpenMPTarget backend on NVIDIA and AMD GPUs. However, different combinations of architecture and compiler can vary in their preference for implementations similar to Listing 1.10 or Listing 1.11, as our experiments will illustrate.

Figure 3 shows the performance of `spmv` on NVIDIA A100 and AMD MI250x GPUs. The labels for the legends of Figure 3 represent the following:

1. *KK-CUDA 3 levels* - Kokkos version with CUDA backend, using all 3 levels of hierarchical parallelism
2. *KK-CUDA 2 levels* - Kokkos version with CUDA backend, using only 2 levels of hierarchical parallelism. (Set `vector_size=1` for `ThreadVector` level.)
3. *KK-OMP-a* - Kokkos version with OpenMPTarget backend, implementing hierarchical parallelism similar to listing 1.10
4. *KK-OMP-b* - Kokkos version with OpenMPTarget backend, implementing hierarchical parallelism similar to listing 1.11.
5. *w/o num_teams* - allow the compiler to choose the number of teams
6. *OMP* - direct (non-Kokkos) OpenMP implementation

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, the hardware, and the specific variant of the OpenMP code. The experiment highlights the sensitivity of the OpenMP performance to specific implementation choices, with different choices resulting in better performance on different hardware and compiler combinations.

For example, consider the two compiler versions on NVIDIA's A100. Using LLVM compiler the native OpenMP version and the KK-OMP-B version without the `num_teams` clause come closest to the performance of the native backends, achieving approximately 70% of the KK-CUDA-3-level bandwidth. These optimized OpenMP versions using LLVM on A100 achieve performance similar to their equivalent native KK-CUDA-2-level version. In comparison, the KK-OMP-A version using LLVM shows a 25% performance gap, and this regression has been observed in other applications as well. The NVHPC compiler also prefers the KK-OMP-B style of parallel decomposition, and unlike LLVM it benefits immensely from the use of `num_teams` clause. Additionally in this combination of architecture and compiler versions, all OpenMP versions underperform compared to the equivalent native KK-CUDA-2-level version.

On AMD GPUs, even the 2-level native version significantly underperforms compared to the 3-level native version, highlighting the performance benefits that can be achieved by exploiting all 3 levels of hierarchical parallelism. Among the OpenMP versions, KK-OMP-A outperforms KK-OMP-B on both compilers.

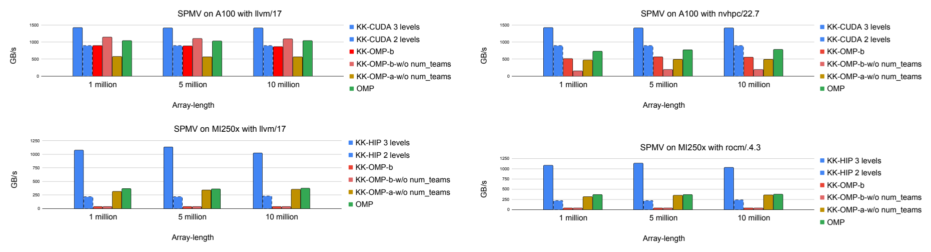

Fig. 3: SPMV on NVIDIA A100 with LLVM and NVHPC compilers and on AMD MI250x with LLVM and ROCM compilers. Y axis is GB/s, so higher is better.

The performance differences observed in this study make it difficult to maintain OpenMP code with consistent performance across different platforms.

The performance of the CG-Solve application as a whole is dominated by the performance of the `SPMV` kernel. Running the CG-Solve example with the OpenMPTarget backend of Kokkos without making any optimizations specific to CG-solve itself brings the performance close to 50% of the native backends.

3 Beyond the basics

Besides the initial issues mapping Kokkos to OpenMP already discussed above, there are a number of other challenges that we outline 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 principle 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 most 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 for which the OpenMP specification does not provide a mechanism.

3.2 Concurrency

Another capability in Kokkos which is difficult to reliably implement is querying available device concurrency. As mentioned in Section 2.3, there is a need to have tight control over the number of teams generated in order to support `TeamThread` level reductions in Kokkos. However we also do not want to restrict the parallelism that can be exploited by a compiler. A trade-off between the two constraints is to calculate the maximum number of in-flight teams possible on a given architecture based on the team size requested. This approach requires information about the number of execution resources available. Currently the backend uses a mix of hardware knowledge, OpenMP routines when applicable and an educated guess to determine this number since there is no single solution that reliably works on every applicable architecture-compiler combination.

One candidate solution is the `omp_get_num_procs()` routine. Because the routine returns the number of processors available on the current device, when called from the host it cannot provide information about concurrency on other devices. We suggest extending its functionality to take a device number as an argument and return the number of available processors on the device identified by that device number. A potential workaround is to open an empty target region at the start of the program only to call `omp_get_num_procs()` within it. Unfortunately, we have observed that the number returned by the routine when called from an accelerator device is not a consistent representation of the underlying hardware concurrency across implementations. Some implementations even return just `1` if the target region body contains only the call to that routine, because they try to optimize the amount of execution resources to match the computation in the target region.

Another use of device concurrency information in Kokkos is to support its `UniqueToken` feature, a locking mechanism that allows a caller to acquire a unique index. Ideally the number of unique index entries should match the number of execution resources. Otherwise, an arbitrarily large number of such unique indices must be created, which may not be practically useful.

Currently extensions to query device concurrency exist that are specific to some vendors, but we are not aware of a portable solution. We hope to converge onto a single cohesive and portable solution on this issue through collaboration with vendors and the community.

4 Conclusion

In this paper we have described mapping the Kokkos Performance Portability model to OpenMP for GPUs. Using a simple linear solver we have explored 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 specification. On average the OpenMP variants (including Kokkos OpenMPTarget backend and raw OpenMP code) provide 57% of the CUDA and HIP backend performance, but at its worst it is about 30x slower than the HIP backend. The performance of the OpenMP implementation is very sensitive to particular construct choices, but the effect of these choices depends on both hardware and compiler. It is thus 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. We acknowledge that the current state of OpenMP offloading for GPUs represents an improvement from the past, when performance and even basic portability had been universally poor even for simple loops. We look forward to future enhancements in the specification and improvements in compiler/runtime implementations, which are becoming more commonplace as a result of collaborations between vendors and the community to address the challenge of performance portability.

Acknowledgments

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 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. This work was supported by Exascale Computing Project 17-SC-20-SC, a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and 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.

References

1. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: RAJA: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 71–81. IEEE (2019)
2. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014). https://doi.org/https://doi.org/10.1016/j.jpdc.2014.07.003,

https://www.sciencedirect.com/science/article/pii/S0743731514001257, domain-Specific Languages and High-Level Frameworks for High-Performance Computing
3. Doerfert, J., Jasper, M., Huber, J., Abdelaal, K., Georgakoudis, G., Scogland, T., Parasyris, K.: Breaking the vendor lock: Performance portable programming through OpenMP as target independent runtime layer. In: Klöckner, A., Moreira, J. (eds.) Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022. pp. 494–504. ACM (2022). https://doi.org/10.1145/3559009.3569687, https://doi.org/10.1145/3559009.3569687
4. Doerfert, J., Patel, A., Huber, J., Tian, S., Diaz, J.M.M., Chapman, B., Georgakoudis, G.: Co-designing an OpenMP GPU runtime and optimizations for near-zero overhead execution. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 504–514 (2022). https://doi.org/10.1109/IPDPS53621.2022.00055
5. Hestenes, M.R., Stiefel, E., et al.: Methods of conjugate gradients for solving linear systems. Journal of research of the National Bureau of Standards **49**(6), 409–436 (1952)
6. Kelling, J., Bastrakov, S., Debus, A., Kluge, T., Leinhauser, M., Pausch, R., Steiniger, K., Stephan, J., Widera, R., Young, J., Bussmann, M., Chandrasekaran, S., Juckeland, G.: Challenges porting a C++ template-metaprogramming abstraction layer to directive-based offloading. In: Bhalachandra, S., Daley, C., Melesse Vergara, V. (eds.) 2021 International Workshop on Accelerator Programming Using Directives. pp. 92–111. Springer International Publishing, Cham (2022)
7. Khronos SYCL Working Group: SYCL specification (June 2020), https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf, version 2020 provisional
8. Killian, W., Scogland, T., Kunen, A., Cavazos, J.: The design and implementation of OpenMP 4.5 and OpenACC backends for the RAJA C++ performance portability layer. In: Chandrasekaran, S., Juckeland, G. (eds.) 2017 International Workshop on Accelerator Programming Using Directives. pp. 63–82. Springer International Publishing, Cham (2018)
9. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf (July 2013)
10. 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)
11. Trott, C., Berger-Vergiat, L., Poliakoff, D., Rajamanickam, S., Lebrun-Grandie, D., Madsen, J., Al Awar, N., Gligoric, M., Shipman, G., Womeldorff, G.: The kokkos ecosystem: Comprehensive performance portability for high performance computing. Computing in Science & Engineering **23**(5), 10–18 (2021). https://doi.org/10.1109/MCSE.2021.3098509
12. Trott, C.R., Lebrun-Grandie, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2021)