Experiences with implementing Kokkos' SYCL backend

Daniel Arndt
Damien Lebrun-Grandié
arndtd@ornl.gov
lebrungrandt@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Christian Trott
crtrott@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

ABSTRACT

With the recent diversification of the hardware landscape in the high-performance computing community, performance-portability solutions are becoming more and more important. One of the most popular choices is Kokkos. In this paper, we describe how Kokkos maps to SYCL 2020, how SYCL had to evolve to enable a full Kokkos implementation, and where we still rely on extensions provided by Intel's oneAPI implementation. Furthermore, we describe how applications can use Kokkos and its ecosystem to already explore upcoming C++ features also when using the SYCL backend. Finally, we are providing some performance benchmarks comparing native SYCL and Kokkos and also discuss hierarchical parallelism in the SYCL 2020 interface.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

Performance portability, programming models, high-performance computing, heterogeneous computing, exascale

ACM Reference Format:
Daniel Arndt, Damien Lebrun-Grandié, and Christian Trott. 2024. Experiences with implementing Kokkos' SYCL backend. In *International Workshop on OpenCL and SYCL (IWOCL '24), April 8–11, 2024, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3648115.3648118

1 INTRODUCTION

Over the last decade, writing performance-portable applications has become one of the most pressing concerns of the high-performance computing (HPC) community. With the introduction of GPUs into the hardware mix for supercomputers, developers of scientific and engineering codes had to contend with the challenge of using multiple toolchains and vendor programming models to make their software work and perform on every relevant platform. Up until recently, that largely meant supporting CPUs and Nvidia GPUs. Many projects approached this challenge by maintaining both a CPU implementation and a CUDA implementation - either of the entire code or just the most compute-intensive pieces. However, for some teams, having multiple implementations of their software was not considered maintainable. Thus, the need to develop single-source performance-portability solutions arose. Two early solutions for this issue in the HPC community were OpenACC and Kokkos[2, 9], but adoption was limited and many projects considered the potential benefit of fine-tuned specialized code worth the additional effort for separate CPU and CUDA code paths.

This attitude changed when the Department of Energy (DOE) announced in 2019 that the first exascale supercomputers would introduce much more hardware diversity. Frontier and El Capitan - to be sited at Oak Ridge and Lawrence Livermore National Laboratory respectively - were going to use AMD GPUs, and Aurora - to be sited at Argonne National Laboratory - would depend on Intel GPUs for the majority of its performance. This posed an urgent challenge to the members of DOE's Exascale Computing Project (ECP) which was tasked with preparing DOE's HPC software ecosystem for these new machines. Code teams had to support current platforms using Nvidia GPUs with CUDA as well as CPU-based systems while preparing their software for two new architectures with programming models that either had not existed in the recent past (AMD's HIP), had virtually no use within the HPC community (SYCL for Intel GPUs), or only provided very immature implementations (OpenMP target offload). For the majority of ECP projects that could not afford to target all of the different (native/vendor-preferred) programming modules (CPU, CUDA, HIP, SYCL) directly, ECP offered four possible solutions: Kokkos, RAJA[1], OpenMP target offload, and a domain-specific library called AMReX[10].

Today, Kokkos is used by about half of all C++-based ECP projects [4], while RAJA is mostly used by projects led by Lawrence Livermore National Laboratory where it originated. AMReX is a domain-specific C++ library for applications that need adaptive mesh refinement. It is directly implemented on top of the vendor native programming models but isolates those from the applications using AMReX. OpenMP target offload was initially thought to be the primary approach for writing vendor-independent code in ECP but in the end, only a few projects used it, mostly in connection with Fortran. Experiences with this program model and the implementation of the corresponding backend within Kokkos can be found in [5].

To support their hardware, Intel recommended SYCL as the primary programming model, and engaged with ECP teams as well as participants in Argonne's separate Early Science Program (ESP).

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

IWOCL 2024, April 08–11, 2024, Chicago, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3648115.3648118

This recommendation and interactions within ECP prompted the Kokkos team to develop a SYCL backend using Intel's oneAPI implementation, leading to today's hardware mapping of Kokkos' primary backends:

- Nvidia GPU → CUDA backend
- AMD GPU → HIP backend
- Intel GPU → SYCL backend
- CPUs → Serial, Threads, or OpenMP backend

In this paper, we describe how Kokkos maps to SYCL 2020, how SYCL had to evolve to enable a full Kokkos implementation, and where we still rely on extensions provided by Intel's oneAPI implementation. Furthermore, we describe how applications can use Kokkos and its ecosystem [8] to already explore upcoming ISO C++ features also when using the SYCL backend.

The paper is structured as follows. We first describe data abstractions in Kokkos, their relationship to ISO C++ mdspan, and issues around using SYCL buffers and its unified shared memory (USM) abstraction to implement them. Next, we cover parallel constructs and their execution policies and briefly discuss Kokkos' implementation of ISO C++ parallel algorithms. Before concluding with a summary, we also touch on gaps in the SYCL 2020 standard that requires Kokkos to rely on Intel-specific extensions for full support and present some performance benchmarks comparing native SYCL with Kokkos' SYCL backend.

Note: this paper only provides the briefest of descriptions of Kokkos' functionality, with the intent to just introduce enough to be able to follow the discussion of specific SYCL concerns. For more details on Kokkos please refer to [2, 9] as well as the Kokkos documentation.

2 DATA ABSTRACTION

One of the most challenging aspects of writing code for heterogeneous architectures is data management. Developers have to control and reason about where data resides as well as how to optimize architecture-dependent data access patterns. In their original design, Kokkos and SYCL have different approaches to address these issues (also see [2, 6, 9]): Kokkos was designed with explicit user-controlled data management in mind, while SYCL's primary design philosophy centered on implicit data management. More specifically, Kokkos introduces explicit concepts of memory spaces, as well as memory layouts with strongly typed data structures, and a requirement for the user to explicitly express data movement. SYCL, on the other hand, uses an opaque data abstraction called buffers where dependencies of algorithms are expressed in terms of the use of those buffers, and the SYCL runtime manages data movement and layout implicitly.

2.1 Kokkos::View and std::mdspan

The fundamental data class in Kokkos is called View. It is a reference-counted, multidimensional array that is memory layout- and memory space-aware. At its simplest, View is used as the basic smart pointer of Kokkos applications. ISO C++23 introduced std::mdspan with a design that is based to a large degree on Kokkos::View. Effectively, std::mdspan is an unmanaged (i.e. non-owning) version of Kokkos::View. The properties expressed in the template arguments of the two classes can be explicitly mapped to each other. Consider the signature of std::mdspan and Kokkos::View:

```
template <class DataType [, class Layout = @see below@]
  [, class MemorySpace = @see below@]
  [, class MemoryTraits = @see below@]
> class View;

template <class T, class Extents,
  class LayoutPolicy = std::layout_right,
  class AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

Where mdspan takes an element type and the extents of the array as two separate parameters T and Extents, View accepts a combined parameter via DataType. For example, a two-dimensional array (with dynamic extents) of double elements, is expressed with parameters [T = double] and [Extents = extents<int, dynamic_extent, dynamic_extent>] for mdspan and [DataType = double**] for View. The Layout template parameter maps directly with Kokkos' default argument depending on the architecture a code is compiled for.

Since ISO C++ does not include a notion of memory spaces (let alone accessibility properties) in its abstract machine model, MemorySpace was not made a first-class concept for mdspan. Kokkos uses MemorySpace to encode semantics such as the accessibility of memory allocations from specific execution resources in a type-safe manner. Thus, an algorithm implemented in Kokkos can at compile time decide on a valid execution mechanism (e.g. GPU or CPU) based on the types of the Views handed to it. It also enables static checking for data access violations instead of discovering them at runtime.

The MemoryTraits parameter expresses additional data access behavior such as atomic accesses, non-aliasing (restrict) access, and data reuse behavior. With mdspan, both MemorySpace and MemoryTraits can be expressed with the AccessorPolicy. For example, one could implement an AccessorPolicy for atomic accesses in a CUDA kernel, which would give compile-time errors for mdspan data accesses in host code and calls CUDA atomic intrinsics in device code.

In contrast to View, mdspan allows for arbitrary mixing of compile-time and runtime extents, and it has no limitation on the dimensionality, whereas View currently only supports up to 8 dimensions.

Currently, a reference implementation for mdspan is maintained within the Kokkos ecosystem and will soon become a crucial building block for View which should then only be a small wrapper around it managing the owning semantics. This implementation is also tested in SYCL code and is ready to be used with Kokkos or independently.

Data movement between different memory spaces is managed explicitly in Kokkos via its deep_copy function. Since the View arguments to deep_copy are strongly typed on their respective memory spaces, it does not require trailing arguments for the direction of the copy which could be mismatched. In the CUDA runtime, for example, cudaMemcpy accepts a cudaMemcpyKind argument specifying the memory spaces between which to copy (note: on systems with unified virtual address space, one can ask cudaMemcpy to infer the copy direction from the runtime pointer values, thus avoiding the mismatch problem).

2.2 SYCL Buffers and the issue of higher level data structures

As stated above, `View` and `mdspan` are intended not just as standalone multi-dimensional arrays, but also as the fundamental building blocks for higher-level data structures. For example a typical implementation of a compressed sparse row (CSR) matrix[2] in Kokkos would contain three `Views` for the row offsets, column indices and matrix values:

```
template<class MemSpace>
struct CsrMatrix {
  View<int64_t*, MemSpace> row_ptr, col_idx;
  View<double*, MemSpace> values;
  /*...*/
};
```

With Kokkos (or pure C++ for that matter), such a struct is seamlessly usable in parallel code sections. For example, a naive sparse matrix-vector product would capture the matrix, and then use its members in the implementation:

```
CsrMatrix<MemSpace> M(...);
View<MemSpace> x(...);
View<MemSpace> y(...);
parallel_for(M.num_rows(), [=](int i) {
  for(int j=M.row_ptr[i]; j<M.row_ptr[i+1]; j++)
    y[i] += M.values[j] * x[M.col_idx[j]]
}
```

However, this scenario poses an issue when attempting to use SYCL `buffers` as the fundamental data abstraction. SYCL `buffers` do not directly provide a method to access data elements. One first has to obtain an `accessor`, through which data accesses are performed. The lifetime of an `accessor` is limited to the scope of a single `queue::submit` call, and thus they can't be permanently stored as members of higher-level data structures that persist across multiple kernel dispatches. In simple programs, one can work around that limitation by explicitly extracting the members of classes and not using the class itself inside a parallel region. A possible SYCL implementation could look something like

```
CsrMatrix<MemSpace> M(...);
View<MemSpace> x(...);
View<MemSpace> y(...);
sycl::queue q;
q.submit([&](sycl::handler &h) {
  auto rows = M.get_row_buffer().get_access(h);
  auto cols = M.get_col_buffer().get_access(h);
  auto values = M.get_values_buffer().get_access(h);
  auto x_access = x.get_access(h);
  auto y_access = y.get_access(h);
  h.parallel_for(sycl::range<1>(N),[=](sycl::item<1> id)
  {
    for(int j=rows[i]; j<rows[i+1]; j++)
      y_access[i] += values[j] * x_access[cols[j]]
  });
});
```

This limits the usability of higher-level structures significantly. For example, one couldn't call a function that takes a `CsrMatrix` as an argument inside the parallel region. Furthermore, it requires that all class members are part of the public interface of that class. Last but not least, this does not allow for some higher-level dispatch functions where users can pass in an opaque callable object. Specifically, the above `parallel_for` function can not be implemented in a way compatible with SYCL `buffers` as the data handles in `CsrMatrix`. Consider the following implementation:

```
template<class Callable>
void parallel_for(int N, Callable lambda) {
  sycl::queue q;
  q.submit([&](sycl::handler &h) {
    // can only create accessors here
    h.parallel_for(sycl::range<1>(N),
      [=](sycl::item<1> id) {
        lambda(id);
      });
  });
}
```

An `accessor` can only be created inside the scope of `queue::submit`, but at that point, there is no way to get access to any indirectly contained `buffers` in the `lambda`. Besides this fundamental usability issue of the SYCL `buffer` construct, compared to `View` and `mdspan` it also only supports up to 3-dimensional arrays.

2.3 Using USM instead of Buffers

To address the usability issues of `buffers` described above, SYCL 2020 introduced Unified Shared Memory (USM). USM provides C-style memory management functions operating on raw pointers in conjunction with arguments indicating memory spaces.

Kokkos memory abstractions are exclusively built on top of USM (instead of the `buffer`/`accessor` model) and the `SYCL` backend `View` implementation just wraps USM pointers/allocations. There are no modifications necessary to the `View` class itself. Each `SYCL` USM allocation type is mapped to a separate Kokkos memory space type:

`sycl::usm::alloc::host`	`Kokkos::SYCLHostUSMSpace`
`sycl::usm::alloc::shared`	`Kokkos::SYCLSharedSpace`
`sycl::usm::alloc::device`	`Kokkos::SYCLDeviceSpace`

Furthermore, Kokkos defines a scratch memory space (also see Section 3.1) that wraps either global (device) address space or shared address space.

One outstanding issue with `View` and other data containers is that they are not trivially-copyable which prevents them from being usable by default in `SYCL` device code. Since Kokkos doesn't have any control over the functor's members (that in most cases contain at least one `View`), we have to declare every function object passed to a parallel construct as `sycl::is_device_copyable`. Unfortunately, this is not quite sufficient and we also have to make sure that the functor's special member functions behave as if they were trivially-copyable (at least with the oneAPI implementation) since those might not be device-callable. Note that the SYCL 2020 standard says[3]:

> It is unspecified whether the implementation actually calls the copy constructor, move constructor, copy assignment operator, or move assignment operator of a class declared as is_device_copyable_v when doing an inter-device copy. Since these operations must all be the same as a bitwise copy, the implementation may

[2] https://github.com/kokkos/code-examples/tree/main/papers/IWOCL2024/cgsolve/generate_matrix.hpp

[3] https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec::device.copyable

> simply copy the memory where the object resides. Likewise, it is unspecified whether the implementation actually calls the destructor for such a class on the device since the destructor must have no effect on the device.

This works fine with other backends like CUDA since problematic member functions are never invoked[4]. One way to work around the issue is to define a functor wrapper as follows

```
template <typename Functor>
class SYCLFunctionWrapper<Functor> {
  union TrivialWrapper {
    TrivialWrapper(){}
    TrivialWrapper(const Functor& f) {
      std::memcpy(&m_f, &f, sizeof(m_f));
    }
    TrivialWrapper(const TrivialWrapper& other) {
      std::memcpy(&m_f, &other.m_f, sizeof(m_f));
    }
    TrivialWrapper(TrivialWrapper&& other) {
      std::memcpy(&m_f, &other.m_f, sizeof(m_f));
    }
    TrivialWrapper& operator=(const TrivialWrapper& other) {
      std::memcpy(&m_f, &other.m_f, sizeof(m_f));
      return *this;
    }
    TrivialWrapper& operator=(TrivialWrapper&& other) {
      std::memcpy(&m_f, &other.m_f, sizeof(m_f));
      return *this;
    }
    ~TrivialWrapper(){}
    Functor m_f;
  } m_functor;
 /*...*/
 public:
  SYCLFunctionWrapper(const Functor& functor, Storage&)
    : m_functor(functor) {}
  const Functor& get_functor() const {
    return m_functor.m_f;
  }
};

template <typename Functor>
struct sycl::is_device_copyable<SYCLFunctionWrapper<Functor>
  : std::true_type {} ;
```

Finding a way to circumvent the `is_device_copyable` check was probably the biggest initial hurdle we had to overcome when implementing the SYCL backend.

3 PARALLEL CONSTRUCTS AND ALGORITHMS

Kokkos implements three primary parallel constructs: `parallel_for`, `parallel_reduce`, and `parallel_scan`. These parallel constructs take at least two arguments: an execution policy and a callable object or functor that defines the operation executed. The three primary execution policies are `RangePolicy` which expresses a simple 1-D iteration range, `MDRangePolicy` which allows for multi-dimensional iteration, and `TeamPolicy` for implementing hierarchical parallel algorithms.

Based on these three parallel constructs, Kokkos provides algorithms that match the ISO C++ parallel algorithms. In addition to interfaces taking iterators, our implementations also have overloads accepting `Kokkos::View` arguments. These overloads have the advantage that a runtime size matching check is possible between input and output arguments. Another deviation from the ISO C++ standard is that our implementations only support `par_unseq` semantics, and take Kokkos execution space instances (i.e. the equivalent of SYCL queues) instead of ISO C++ execution policies. An additional capability provided by the Kokkos parallel algorithms is the ability to call them as part of our hierarchical parallelism implementation, i.e. a user can call these algorithms instead of nested parallel loops. This is not supported by any other implementation of the ISO C++ standard algorithms we are aware of, including oneDPL, Thrust, and a seemingly abandoned SYCL effort by the Khronos-Group (https://github.com/KhronosGroup/SyclParallelSTL).

3.1 parallel_for

In its simplest form, `Kokkos::parallel_for` using `RangePolicy` directly wraps a `sycl::parallel_for`. It describes the execution of a kernel for a 1-D iteration range where each work item is independent. A prototypical invocation like

```
Kokkos::parallel_for(
  Kokkos::RangePolicy(execution_space, start, end)),
  KOKKOS_LAMBDA(int i) {/*...*/});
```

is mapped to SYCL code as

```
Functor functor;
q.parallel_for(sycl::range<1>(end - begin),
  [=](sycl::id<1> idx) {
    int i = idx + begin;
    functor(i);
});
```

When using an `MDRangePolicy`, the parallel iteration range can be 2 to 6-dimensional and is internally divided into tiles. Individual tiles are then mapped to thread groups such as a single thread on a CPU, blocks in CUDA and HIP, and a workgroup for SYCL. The iteration over tiles is flattened onto a single dimension of parallelism. The iteration within each tile is then mapped to the threads in the thread groups. With SYCL, HIP, and CUDA sharing the concept of 3-dimensional thread groups, we were able to share the code for mapping the multi-dimensional tile iteration to threads across the three backends. An example of a 5-dimensional iteration space looks like:

```
struct Functor{
  KOKKOS_FUNCTION void operator(
    int i, int j, int k, int l, int m) const {/*...*/}
};
Kokkos::parallel_for(
  Kokkos::MDRangePolicy(execution_space,
    {s0,s1,s2,s3,s4}, {e0,e1,e2,e3,e4}),
  Functor{});
```

Implementation-wise the basic idea here is to pack multiple dimensional ranges into the three dimensions exposed by sycl::nd_range in the same way as for the CUDA and HIP backend implementation.

[4]See https://github.com/intel/llvm/issues/5320 for further discussion.

	nd_range dimension		
# dimensions	0	1	2
2	0	1	
3	0	1	2
4	0,1	2	3
5	0,1	2,3	4
6	0,1	2,3	4,5

One interesting distinction between the CUDA/HIP backend and the SYCL backend is that we map indices differently. For optimal memory access patterns with CUDA and HIP the `threadIDx.x` (i.e. the first) index is the one that needs to access consecutive memory locations, while it is the third dimension with SYCL.

Finally, there is `TeamPolicy` that describes hierarchical parallelism. In contrast to the SYCL concept for hierarchical parallelism, which exposes a fork-join-style interface, in Kokkos, threads perform redundant execution in the outer loop - that is all threads are active upon entry into the parallel region. A true fork-join style implementation is not suitable for many cases of hierarchical parallelism (such as the sparse matrix-vector product later discussed), because the work inside the nested parallel loops is smaller than the cost of fork-join[5]. The redundant execution concept however does map well to CUDA and HIP. It is also trivially implementable with any model where hierarchical parallelism is only provided via fork-join mechanisms, by forking immediately and simply splitting the nested parallel iterations across the appropriate threads (also compare [3] that proposed another approach to hierarchical parallelism with SYCL). Note that the SYCL 2020 standard itself discourages using its native interface[6].

An example for the `TeamPolicy` API is:

```
parallel_for("Label",
  TeamPolicy<>(numberOfTeams, teamSize, vectorLength),
  KOKKOS_LAMBDA (const member_type & teamMember) {
    /* beginning of outer body */
    parallel_for(
      TeamThreadRange(teamMember, thisTeamsRangeSize),
      [=] (const int indexWithinBatch[, ...]) {
        /* begin middle body */
        parallel_for(ThreadVectorRange(teamMember,
                     thisVectorRangeSize),
           [=] (const int indexVectorRange) {
             /* inner body */
           });
        /* end middle body */
      });
    parallel_for(TeamVectorRange(teamMember, someSize),
      [=] (const int indexTeamVector) {
        /* nested body */
      });
    /* end of outer body */
  });
```

Kokkos `Teams` consist conceptually of *threads* with *vector lanes*. In the example above, the number of *vector lanes* per thread is controlled by the `vectorLength` parameter, while the number of *threads* per team is controlled by the `teamSize` parameter. Again, this concept is implemented using `sycl::parallel_for` using `nd_range`. We map a team to a `sycl::group` and the vector lanes of an individual thread to a subset of a `sycl::subgroup`. For example, if the subgroup size is 32, and `vectorLength` is 8, each `sycl::subgroup` will consist of four Kokkos *threads*.

The execution policies for nested parallel constructs are mapping iterations to *threads* in a team (`TeamThreadRange`), *vector lanes* within a *thread* (`ThreadVectorRange`) or both (`TeamVectorRange`).

Since we divide subgroups into multiple *threads*, we require a mechanism to synchronize part of a subgroup. This is implicit on some hardware (Intel GPUs, and AMD GPUs), but would require the non-uniform group extension proposed in the oneAPI implementation, in particular `tangle_groups`, on Nvidia GPUs[7]. However, `tangle_groups` aren't implemented in the oneAPI compiler for Nvidia hardware, yet[8]. Thus, our implementation of hierarchical parallelism with SYCL is not fully functional when compiling for Nvidia architectures.

`TeamPolicy` is the only Kokkos policy that allows using scratch memory space in the form of *team scratch*. In particular, Kokkos provides two levels of scratch memory for a team: one intended to map to an L1-like cache and one for larger scratch requirements. An early design decision in Kokkos, where this distinction is not expressed through strongly-typed memory spaces, posed some minor problems when porting to SYCL. For SYCL, we want to map small scratch allocations to SYCL workgroup local memory, and large allocations to plain USM allocations. Unfortunately, the oneAPI compiler has problems figuring out the correct address space when accessing the scratch memory in kernels, and thus generates instructions for either case with runtime checks on the pointer. This makes the compiler emit confusing warnings and may have a negative impact on the instruction cache efficiency. We are currently investigating a change in the Kokkos interface to make different scratch allocations strongly-typed as well so that Kokkos can take advantage of the extra address space information SYCL exposes in comparison to HIP and Cuda. Note that `libcudacxx` provides with `cuda::annotated_ptr`[9] a similar concept that Kokkos could exploit.

3.2 parallel_reduce

The `parallel_reduce` construct in Kokkos provides a mechanism for combining contributions from all iterations of a parallelized loop. As with `parallel_for`, it can be used with different execution policies, expressing various iteration and parallelization schemes. A typical use case looks as follows:

```
double result;
Kokkos::parallel_reduce(
  Kokkos::RangePolicy(execution_space, start, end)),
  KOKKOS_LAMBDA(int i, double& partial_sum) {
    partial_sum += i;
  }, result);
```

[5]There are compiler approaches to avoid true fork-join by having threads execute redundantly every instruction needed to build the thread state, and masking out instructions with side effects. It is not clear whether this optimization is employed by any SYCL implementation.

[6]https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_hierarchical_data_parallel_kernels

[7]https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_non_uniform_groups.asciidoc

[8]https://github.com/intel/llvm/blob/ec7fb7c4aebae3ea642a269e8cc4d4ab57f721ef/sycl/include/sycl/ext/oneapi/experimental/tangle_group.hpp#L152-L159

[9]https://nvidia.github.io/libcudacxx/extended_api/memory_access_properties/annotated_ptr.html

When providing just a result argument, `parallel_reduce` will perform a sum-reduction. However, users can provide so-called *reducers* which specify different reduction operations, such as minimum, maximum, or product:

```
double minimum;
Kokkos::View<double*> a(/*...*/);
Kokkos::parallel_reduce(
  Kokkos::RangePolicy(execution_space, start, end)),
  KOKKOS_LAMBDA(int i, double& partial_min) {
    if (a[i] < partial_min)
      partial_min = a[i];
  }, Kokkos::Min(minimum));
```

This interface is not too different from the provided by SYCL's reduction variables:

```
double minimum;
double* data = /*...*/;
q.parallel_for(sycl::range<1>(N_),
  sycl::reduction(result_ptr, 0., sycl::minimum<double>()),
  [=](sycl::id<1> idx, auto& min) {
    int i = idx;
    min.combine(data[i]);
  });
  q.memcpy(&result, result_ptr, sizeof(double));
  q.wait();
}
```

However, Kokkos does not currently implement `parallel_reduce` via the `sycl::reduction` interface. Partly this is due to historically incomplete support for the full SYCL reduction interface (in fact the SYCL 1.2 did not have a reduction interface at all), and partly due to lack of support for certain capabilities of Kokkos' `parallel_reduce` which exceed the scope of the `sycl::reduction` interface. Among the features of Kokkos `parallel_reduce` are

- simple reductions (sum)
- multiple reductions per parallel construct
- custom reductions with arbitrary value types and reduction operations
- runtime sized array reductions
- pre- and post-callbacks for reductions (`init`, `final`)

SYCL only supports compile-time-sized reductions and none of the callbacks. The Kokkos implementation uses `sycl::parallel_for` with `sycl::nd_range` and follows the following pseudocode

```
per_thread:
  value& tmp=init(local_tmp);
  for (i in local range)
    functor(i, tmp)
call join for merging values between threads
  in the same workgroup
let one (the last) workgroup merge all results
  from all workgroups
call final(result) on one thread
```

Initially, we tried to use a shuffle-based implementation but quickly discovered that using local memory gives better performance on Intel GPUs - an observation also reported in [7].

3.3 parallel_scan

Contrary to the SYCL API, Kokkos implements an interface for inclusive/exclusive prefix sums called `parallel_scan` as a first-class citizen in the Kokkos programming model. The interface is similar to the one for `parallel_reduce` but only supports a flat, one-dimensional policy (`RangePolicy`) with a single scalar reducer. A typical usage example looks like

```
Kokkos::parallel_scan(
  Kokkos::RangePolicy(execution_space, start, end),
  KOKKOS_LAMBDA (const int index, value_type& update,
                 const bool is_final) {
    const value_type local_value = in_data(i);
    // exclusive scan
    if (is_final)
      out_data_exclusive(i) = update;
    update += local_value;
    // inclusive scan
    if (is_final)
      out_data_inclusive(i) = update;
 });
```

The data in `in_data` is used to simultaneously compute an inclusive prefix sum (stored in `out_data_inclusive`) and an exclusive prefix sum (stored in `out_data_exclusive`) storing the sum across values in `total`. One important distinction of `parallel_scan` is that it requires a two-pass algorithm for parallelization. In the first pass, every work item calculates its contribution, and in the second pass, the `update` value passed to the work item contains the exclusive scan value. The `is_final` argument indicates whether the second pass is performed.

Note that within the oneAPI framework, oneDPL implements `[inclusive/exclusive]_[transform]_scan` that provide similar capabilities based on the ISO C++ parallel algorithms. However, the Kokkos `parallel_scan` facility is somewhat more powerful. With the C++ parallel algorithms, the actual scan values are only available in the data structure referred to by an output iterator, and can only be consumed during a subsequent kernel launch. In the Kokkos construct, the scan values are available during execution of the callable and don't have to be stored. This allows, for example, the implementation of communication pack routines as a single parallel construct, such as copying all particles that left the domain of an MPI rank to a buffer for transfer to another rank.

The following pseudocode shows the Kokkos' `parallel_scan` implementation using two `sycl::parallel_for` with calls using `sycl::nd_range`:

```
first kernel:
  per_thread:
    value& tmp=init(local_tmp);
    for (i in local range)
      functor(i, tmp, /*is_final*/ false)
  call join for implementing a prefix sum
    in the same workgroup
  let the last workgroup compute the prefix sum for the
    totals of all workgroups and store the result
  store intermediate results on each thread
second kernel:
  combine workgroup totals with thread intermediate results
  call the functor again for final result (with final=true)
```

For finding the "last" workgroup, every workgroup increases a counter in global memory atomically. The one that reaches the total number of workgroups is identified as the last workgroup. Note that an optional last argument for `parallel_scan` allows to also return the total sum of contributions (i.e. the last iterations inclusive scan value).

For several of the C++ parallel algorithms Kokkos provides, the straightforward implementation does require a `parallel_scan`

primitive strengthening the importance of treating it as a first-class citizen. These include:

- `copy_if`
- `exclusive_scan`
- `inclusive_scan`
- `partition_copy`
- `remove_if`
- `remove_copy_if`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `unique`
- `unique_copy`

4 SYCL EXTENSIONS NEEDED FOR KOKKOS

Most of Kokkos' features can be implemented using pure `SYCL 2020` but there are also several features that require extensions, which are provided by the oneAPI implementation[10] (namespace `sycl::ext::oneapi` suppressed for readability):

- random number generator:
  - `experimental::this_nd_item`
- arbitrary size atomics:
  - `experimental::this_sub_group`,
  - `experimental::device_global`,
  - `group_ballot`,
  - `sub_group_mask`
- user code, debugging:
  - `experimental::printf`
- half type support:
  - `bfloat16`
- workgroup-level reductions:
  - `group_local_memory_for_overwrite`

The most critical of the missing features are related to the implementation of arbitrary size atomics - i.e. atomics for objects of a size where no native atomic compare exchange operation exists. Note that the latter are part of the ISO C++ standard `atomic_ref`, but are not supported by either the SYCL or NVIDIA libcu++ library `atomic_ref` implementation. Places that use `this_nd_item` or `this_sub_group` are exposed to users and the Kokkos implementation doesn't provide a way to pass the active thread through the respective interface in a portable way, e.g, `Kokkos::atomic_add(T* const dest, const T val)`.

One capability that isn't strictly needed to implement Kokkos itself, but is frequently requested by users of Kokkos, is the support for `virtual` functions. This capability does exist for all other primary toolchains used by Kokkos, including CUDA and ROCm, but is missing in SYCL and is not currently available as an extension in oneAPI [11].

5 PERFORMANCE RESULTS

The following performance benchmarks were obtained on one node of the Sunspot testbed for Aurora at Argonne National Laboratory with 6 Intel® Data Center GPU Max 1550 GPUs and 2 Intel® Xeon® CPU Max 9470C (52 physical cores supporting 2 hardware threads per core). The GPU is composed of two so-called tiles that can either be addressed as one device (also called "implicit scaling") or two devices (also called "explicit scaling"). In the following, we will refer to the former as "2-tile" and the latter as "1-tile" expressing how many tiles per device are used.

We are employing a sparse matrix conjugate gradient solver (CG-solver) as our basic benchmark. It consists of vector add (AXPBY), dot product (DOT), and sparse matrix-vector multiplication (SPMV) operations. These exercise the fundamental parallel operations in Kokkos via simple `RangePolicy` `parallel_for` and `parallel_reduce` kernels for AXPBY and DOT respectively, and hierarchical parallelism (`TeamPolicy`) for the implementation of the SPMV. We used this benchmark previously in our reference Kokkos paper [9], and the discussion of the OpenMPTarget backend of Kokkos [5]. This benchmark has the advantage of being fairly simple, while simultaneously exposing performance concerns in hardware and basic software stack capabilities (e.g. compiler optimizations, thread synchronization overhead, etc.). For all the kernels we implement a raw SYCL variant, in addition to the Kokkos version, to identify performance costs associated with the abstraction level introduced by Kokkos.

The following code is an implementation of the CG-solve (`A` is the matrix, texttttp, `r`, and `Ap` are vectors):

```
for (int64_t k = 1; k <= max_iter && normr > tolerance; ++k) {
  if (k == 1) {
    axpby(p, one, r, zero, r);
  } else {
    oldrtrans = rtrans;
    rtrans = dot(r, r);
    double beta = rtrans / oldrtrans;
    axpby(p, one, r, beta, p);
  }
  normr = std::sqrt(rtrans);
  double alpha = 0;
  double p_ap_dot = 0;
  spmv(Ap, A, p);
  p_ap_dot = dot(Ap, p);
  if (p_ap_dot < brkdown_tol) {
    if (p_ap_dot < 0) {
      std::cerr << "numerical breakdown!\n";
      return num_iters;
    } else
      brkdown_tol = 0.1 * p_ap_dot;
  }
  alpha = rtrans / p_ap_dot;
  axpby(x, one, x, alpha, p);
  axpby(r, one, r, -alpha, Ap);
  num_iters = k;
}
```

Ignoring the initial iteration, AXPBY is called three times per iteration, DOT twice, and SPMV once. Since SPMV operates on the matrix, and thus accesses by far the most memory, it commonly dominates the overall performance of the CG-solve.

The code for all the experiments performed here, including reproduction instructions can be found at https://github.com/kokkos/code-examples/tree/main/papers/IWOCL2024.

5.1 AXPBY

The `AXPBY` algorithm is a simple vector add with scaling factors and thus exhibits no dependencies between iterations. As such, it is

[10] https://github.com/intel/llvm

[11] There are some efforts in https://github.com/intel/llvm/pull/10540 to enable virtual function support for SYCL in oneAPI.

a good candidate for a RangePolicy parallel benchmark, and in fact, can be considered a variant of the widely used STREAM benchmark. The following implementations in Kokkos and raw SYCL can also be found at https://github.com/kokkos/code-examples/tree/main/papers/IWOCL2024/axpy.

```
// Kokkos
for (int r = 0; r < R; r++) {
  Kokkos::parallel_for("axpby", N, KOKKOS_LAMBDA(int i) {
   z(i) = alpha*x(i) + beta*y(i);
  });
}

// SYCL
sycl::queue q{sycl::property::queue::in_order()} ;
for (int r = 0; r < R; r++) {
  q.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx){
    int i = idx;
    z[i] = alpha*x[i] + beta*y[i];
  });
}
```

Figure 1 shows that the achieved effective bandwidth for both implementations is identical which isn't very surprising given the direct mapping from Kokkos to SYCL code.

In the standalone benchmark, we run the AXPBY kernel on the same data repeatedly. This exposes a caching effect, where the measured bandwidth is significantly higher than the steady state. All in all, however, the observed performance is significantly lower than one may expect based on the theoretical peak throughput of the hardware which is 1.6TB/s per tile. Indeed, we see about 95% and 75% of the peak memory bandwidth for the same benchmark on NVIDIA and AMD GPUs respectively in [5].

It is worth noting that below 10^5 elements AXPBY is largely latency-limited.

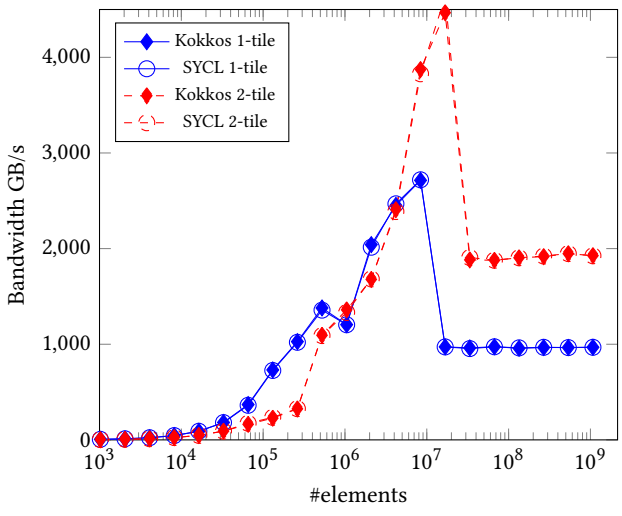

Figure 1: Achieved effective bandwidth for the AXPBY benchmark.

5.2 DOT

The DOT benchmark performs a single reduction with a double value type and thus is trivially implementable with a RangePolicy and parallel_reduce:

```
// Kokkos
for (int r = 0; r < R; r++) {
  Kokkos::parallel_reduce("dot", N,
    KOKKOS_LAMBDA(int i, double& sum) {
      sum += x(i) * y(i);
    },
  result);

// SYCL
sycl::queue q{sycl::property::queue::in_order()} ;
for (int r = 0; r < R; r++) {
  q.parallel_for(sycl::range<1>(N_),
    sycl::reduction(result_ptr, 0., sycl::plus<double>()),
      [=](sycl::id<1> idx, auto&sum) {
        int i = idx;
        sum += x[i] * y[i];
      });
  q.memcpy(&result, result_ptr, sizeof(double));
  q.wait();
}
```

Figure 2 and 3 show the achieved effective bandwidth for both implementations as well as the ratio of the time. There are slight differences in performance, where the Kokkos variant is up to 30% faster for very small arrays, the SYCL variant is up to 20% faster around 1 million elements, and then the differences mostly disappear for larger arrays.

Ultimately, the two perform very similarly and the spikes seen in Figure 3 can be attributed to differences in the workgroup size selection. Similarly to the AXPBY benchmark, we observe that the performance of the DOT product is limited by latency in the regime of fewer than 10^6 elements - and in fact using two tiles does not provide any benefit.

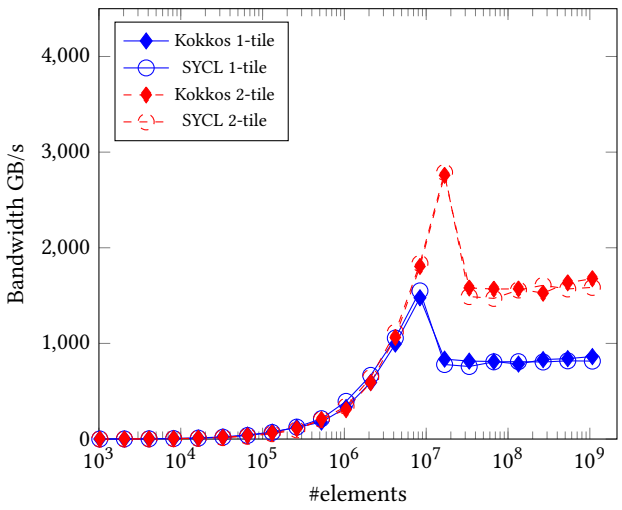

Figure 2: Achieved effective bandwidth for the DOT benchmark.

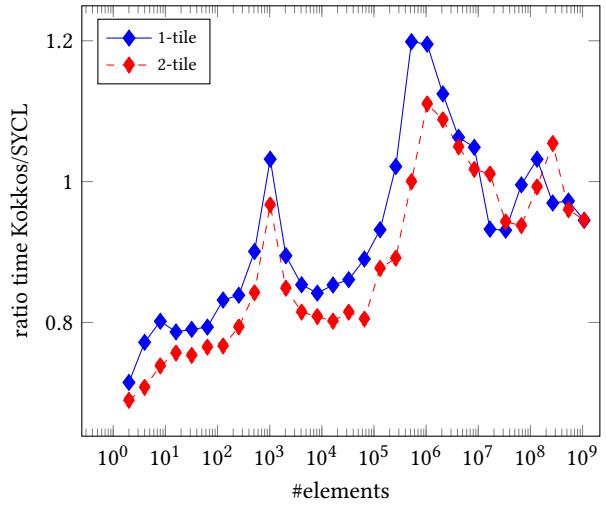

Figure 3: Ratio between run time for the DOT benchmark between Kokkos and SYCL implementation. Values smaller than 1 indicate that the Kokkos version is faster.

5.3 SPMV

The sparse-matrix vector product exhibits two levels of parallelism: for each row of the matrix one has to perform a (sparse) dot product of the matrix elements with the right-hand side vector. In Kokkos this is implemented using a `TeamPolicy` with an outer `parallel_for` algorithm, and nested `parallel_reduce`. One optimization the canonical SPMV implementation in Kokkos does is to introduce a third level of parallelism by grouping multiple rows and assigning them to a single team, while leveraging the vector level parallelism to perform the matrix-row product with the right-hand side vector. This optimization is done for two reasons: i) for many sparse matrices there are not enough entries per row to actually exploit the available concurrency per team on a GPU and ii) sparse matrix rows often can be sorted in a way that subsequent rows need to be multiplied with elements in the right-hand side vector that are near to each other. Assigning those rows to the same team enables implicit cache reuse of elements in the vector:

```
int rows_per_team = 32; //optimized for GPU
int team_size = 16;     //optimized for GPU
int vector_size = 4;    //optimized for GPU
int n_teams = (nrows + rows_per_team - 1)/rows_per_team;
using TeamMember = Kokkos::TeamPolicy<>::member_type;
// parallelize over the row blocks
Kokkos::parallel_for("SPMV",
  Kokkos::TeamPolicy<>(n_teams, team_size, vector_size),
    KOKKOS_LAMBDA(const TeamMember &team) {
      int64_t first_row=team.league_rank()*rows_per_team;
      int64_t last_row=first_row + rows_per_team < nrows
        ? first_row + rows_per_team : nrows;
      // parallelize over rows owned by the 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;
            // perform the dot-product of a matrix row
            // with vector
            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;
        });
    });
```

Since SYCL does not natively support three-level parallelism, the raw SYCL implementation only has two levels for hierarchical parallelism and we miss out on the parallelizing over the entries in a row:

```
int rows_per_team = 32; //optimized for GPU
int team_size = 16;     //optimized for GPU
int n_teams = (nrows + rows_per_team - 1)/rows_per_team;
q.submit([&] (sycl::handler& cgh) {
  // parallelize over the row blocks
  cgh.parallel_for_work_group(sycl::range<1>(n),
    sycl::range<1>(team_size), [=](sycl::group<1> g) {
      int64_t first_row= g.get_group_id(0)*rows_per_team;
      int64_t last_row=first_row + rows_per_team < nrows
        ? first_row + rows_per_team : nrows;
      // parallelize over rows owned by the team
      g.parallel_for_work_item(
        sycl::range<1>(last_row-first_row),
        [&](sycl::h_item<1> item) {
          int64_t row = item.get_local_id(0)+first_row;
          int64_t row_start = row_ptr[row];
          int64_t row_length = row_ptr[row+1]-row_start;
          double y_row = 0.;
          for (int64_t i = 0; i < row_length; ++i)
            y_row += values[i + row_start] *
              xp[col_idx[i + row_start]];
          yp[row] = y_row;
    });
  });
});
```

Note that the choices for the rows per team, the team size and the vector size have been optimized for the particular sparse matrix structure used in this paper. In real math libraries such as Kokkos Kernels, these parameters are usually set by some type of heuristic that takes the sparsity pattern of the matrix into account.

Figure 4 shows performance results for a finite element matrix corresponding to a heat conduction problem with a cubic grid and a maximum row length of 27. The number of rows in the matrix - i.e. the number of elements in the grid - determines the number of elements in the vector operations. We also report on times of SPMV when using the oneMKL implementation of SPMV to better judge the effectiveness of the Kokkos implementation in comparison to hand-tuned code. The comparison demonstrates that it is crucial to also parallelize over the entries in a row to not miss out on almost 2x of performance for the sparse matrix-vector product - at least in the not latency-limited regime. We also see that the implementation here is at least as good as using oneMKL that only really reaches a similar bandwidth with more than 10^6 rows. Note that we are again observing caching between 10^5 and 10^6 rows for one tile and between $2 \cdot 10^5$ and $2 \cdot 10^6$ rows with two tiles.

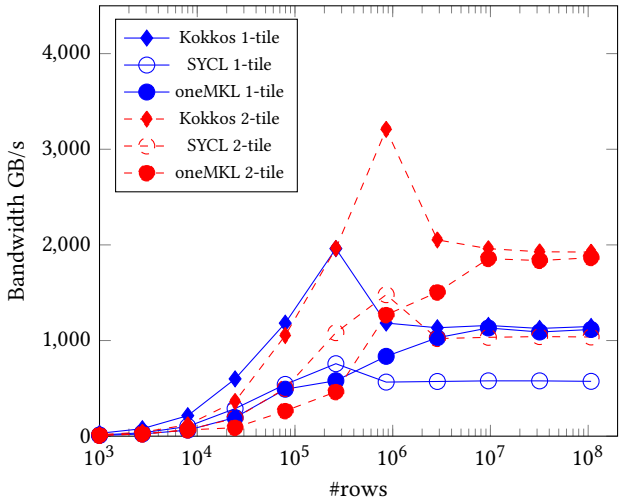

Figure 4: Achieved effective bandwidth for the SPMV benchmark on the GPU.

5.4 CG-solve

Finally, we report on the full CG-solve performance[12] using the same matrix as in Subsection 5.3. Compared with running the individual algorithms there are no caching effects expected for the DOT product and the AXPBY since the caches are getting flushed in between during the SPMV.

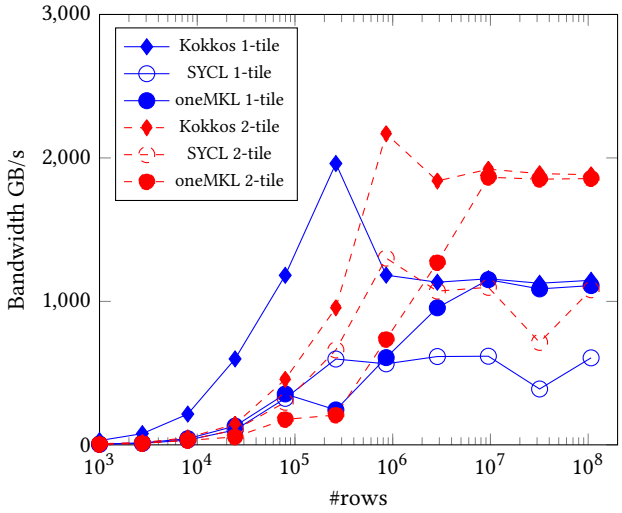

Figure 5: Achieved effective bandwidth for the CG benchmark on the GPU.

During the CG-Solve 1/5th of the time is spent on vector operations, while SPMV takes the remaining 4/5ths, thus the observed performance follows closely the pure SPMV data. The effective bandwidth is in the previously seen regime of about 2/3rds of the peak bandwidth. The implicit scaling to two tiles works fairly well with an observed speed-up of 1.8x. It is also worth noting that the three-level Kokkos algorithm achieves the same level of performance as oneMKL for SPMV for large matrices with 10M rows or more, and actually outperforms the oneMKL implementation for problem sizes smaller than that.

[12]The full code can be found at https://github.com/kokkos/code-examples/tree/main/papers/IWOCL2024/cgsolve

5.5 Running on the CPU

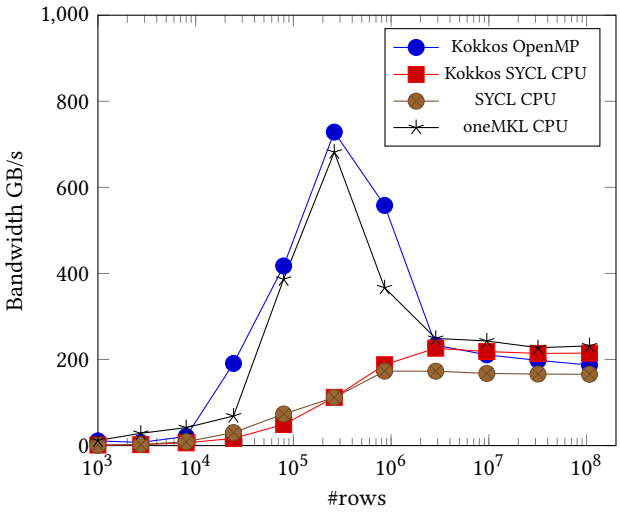

Figure 6: Achieved effective bandwidth for the CGSOLVE benchmark on the CPU.

Finally, we ran this example with the Kokkos `OpenMP` backend, the Kokkos `SYCL` backend, as well as the native SYCL implementation leveraging a `OpenCL` driver for Intel CPUs. It is worth noting that this is not officially supported by Kokkos, and some minor modifications to Kokkos were necessary to disable static checks that Kokkos is getting compiled for a GPU, when SYCL is enabled. In either case, we adjusted the team size and the number of rows per team to be tuned for CPUs. We are using a team size of 1 (and vector length=1) in both cases so that there is effectively no real hierarchical parallelism. We also measured the Kokkos `SYCL` backend version where the Kokkos SPMV call is replaced by a oneMKL call.

The results reported in Figure 6 show that the raw SYCL implementation again is somewhat slower than the Kokkos versions. The Kokkos `SYCL` backend, however, does perform comparable and even better than the Kokkos OpenMP backend for larger problem sizes. In the mid regime, oneMKL and Kokkos OpenMP outperform the SYCL-based implementations of SPMV. This may be an indication of parallel dispatch overhead but may require more detailed investigations for a full understanding.

6 CONCLUSION

Although we only covered basic Kokkos functionalities, we can see that `SYCL` and Kokkos are largely compatible and Kokkos can be implemented on top of `SYCL`. Only a few Kokkos features require

oneAPI extensions, interestingly enough `printf` being the one most often asked for by users.

Kokkos provides programming model elements that improve developer productivity significantly beyond what SYCL 2020 enables. That includes the first-class support for multi-dimensional arrays via `Kokkos::View`, three-level hierarchical parallelism, parallel algorithms usable in the context of hierarchical parallelism, and `parallel_scan`. Of these, `Kokkos::View` is arguably the most critical, in particular since it provides a strongly typed object that allows reasoning about data accessibility at compile time, and can be used as a member of C++ classes to build higher-level data structures.

An open question is if Kokkos users could or should leverage the `SYCL` backend for architectures other than Intel GPUs. In practice, it is possible to compile the `SYCL` backend for NVIDIA GPUs, AMD GPUs, and even CPUs. However, how well these toolchains are supported for each of the architectures isn't clear yet, and it may also require different implementation choices inside the Kokkos backend accounting for fundamental architectural differences to get optimal performance. In particular, support for CPUs via the `SYCL` backend brings up a number of complicated questions related to the compile time choices made for accessibility of memory spaces, and default memory layouts. For example, the memory footprint of the benchmark used in this paper is actually twice as large as it would need to be since there are separate "Host" and "Device" allocations.

Ultimately, our experience with the oneAPI implementation for supporting Intel GPUs has been good. However, the toolchain has not been as stable as the ones used for other Kokkos backends such as CUDA and HIP. A significant fraction of that instability can be attributed to the transition of the oneAPI toolchain to SYCL 2020. Thus, we expect the situation to be much better going forward as the support for this newer standard is getting finalized.

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 manuscript has been authored by UT-Battelle, LLC, under Grant DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). This work was done on a pre-production supercomputer with early versions of the Aurora software development kit. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, United States, 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[2] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[3] Tom Deakin, Simon N McIntosh-Smith, Aksel Alpay, and Vincent Heuveline. 2021. Benchmarking and Extending SYCL Hierarchical Parallelism. In *Workshop on Hierarchical Parallelism for Exascale Computing*. IEEE Computer Society, United States, 10 pages. https://doi.org/10.1109/HiPar54615.2021.00007

[4] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. 2022. A survey of software implementations used by application codes in the Exascale Computing Project. *The International Journal of High Performance Computing Applications* 36, 1 (2022), 5–12. https://doi.org/10.1177/10943420211028940 arXiv:https://doi.org/10.1177/10943420211028940

[5] Rahulkumar Gayatri, Stephen L. Olivier, Christian R. Trott, Johannes Doerfert, Jan Ciesko, and Damien Lebrun-Grandie. 2023. The Kokkos OpenMPTarget Backend: Implementation and Lessons Learned. In *OpenMP: Advanced Task-Based, Device and Compiler Programming*, Simon McIntosh-Smith, Michael Klemm, Bronis R. de Supinski, Tom Deakin, and Jannis Klinkenberg (Eds.). Springer Nature Switzerland, Cham, 99–113.

[6] Jeff R. Hammond, Michael Kinsner, and James Brodman. 2019. A Comparative Analysis of Kokkos and SYCL as Heterogeneous, Parallel Programming Models for C++ Applications. In *Proceedings of the International Workshop on OpenCL* (Boston, MA, USA) *(IWOCL'19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 2 pages. https://doi.org/10.1145/3318170.3318193

[7] Esteban Miguel Rangel, Simon John Pennycook, Adrian Pope, Nicholas Frontiere, Zhiqiang Ma, and Varsha Madananth. 2023. A Performance-Portable SYCL Implementation of CRK-HACC for Exascale. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1114–1125. https://doi.org/10.1145/3624062.3624187

[8] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science & Engineering* 23, 5 (2021), 10–18. https://doi.org/10.1109/MCSE.2021.3098509

[9] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[10] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software* 4, 37 (2019), 1370.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009