DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Reference herein to any social initiative (including but not limited to Diversity, Equity, and Inclusion (DEI); Community Benefits Plans (CBP); Justice 40; etc.) is made by the Author independent of any current requirement by the United States Government and does not constitute or imply endorsement, recommendation, or support by the United States Government or any agency thereof.

ORNL/TM-2025/4398

Experiences with SYCL on AMD GPUs with Kokkos

Daniel Arndt
Damien Lebrun-Grandie
Christian Trott

Approved for public release.
Distribution is unlimited.

December 2025

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2025/4398

Computational Sciences and Engineering Division

EXPERIENCES WITH SYCL ON AMD GPUS WITH KOKKOS

Daniel Arndt
Damien Lebrun-Grandie
Christian Trott

December 2025

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

ABSTRACT

With the recent diversification of the hardware landscape in the high-performance computing (HPC) community, performance-portability solutions are becoming more and more important. One of the most popular choices is Kokkos, which recently became a Linux Foundation project. Most of its development is supported by the US Department of Energy and the French Alternative Energies and Atomic Energy Commission. Kokkos is implemented as a C++ library with multiple backends to support CPUs as well as various GPU architectures. These backends include OpenMP, CUDA, HIP, and also SCYL. This approach enables users to leverage the preferred vendor toolchain for the respective platform (e.g. CUDA, ROCm, OneAPI). The SYCL backend is used to target Intel GPUs, in particular to support the Aurora exascale supercomputer.

However, SYCL itself also offers a large degree of portability, and in fact Kokkos' CI for SYCL has been running on NVIDIA hardware due to a lack of access to Intel GPUs. In this report, we describe our experience with using Kokkos SYCL backend on AMD GPUs targeting the Frontier supercomputer at Oak Ridge National Laboratory. The two major SYCL implementations are DPC++ and AdaptiveCpp. While the Kokkos SYCL backend has been implemented using the former, the latter was the first implementation to target AMD GPUs. We will discuss the experience with both of these SYCL implementations in terms of functionality and performance.

Using Kokkos to evaluate SYCL toolchains has a number of benefits. Kokkos' use of SYCL is fairly complex, exercising features such as graphs, relocatable device functions, atomics – including for non-arithmetic types, as well as pinned and page migratable memory allocations. Kokkos also needs to implement capabilities such as Kokkos' hierarchical parallelism that are not a straight-forward mapping to SYCL capabilities. Furthermore, a large number of libraries and applications that represent diverse use cases are implemented in Kokkos, providing readily available test cases for a toolchain evaluation.

Preliminary results show that support for AMD GPUs in DPC++ is much less mature than for NVIDIA GPUs or Intel GPUs. While the situation has improved significantly over the last year, we still encounter many runtime failures, dispatching problems, and code generation issues. With AdaptiveCpp the challenges arise even earlier in the evaluation process. Since Kokkos' SYCL implementation is largely focused on supporting Intel GPUs, we opted to leverage SYCL extensions which are available in DPC++ but not in AdaptiveCpp. Furthermore, AdaptiveCpp appears to be less conformant with the SYCL2020 standard which Kokkos relies on. In some cases, we are able to work around the lack of feature support, in other cases we have to disable certain Kokkos capabilities to evaluate the toolchain.

Our evaluation will leverage Kokkos' unit tests to establish basic functionality and feature completeness. We then use simple benchmarks for components of a CG implementation as a measure of usability and performance of the SYCL toolchains.

1. A BRIEF HISTORY OF THE SYCL BACKEND IN KOKKOS

Since the announcement that Aurora, the latest supercomputer at Argonne National Laboratory (ANL), will use Intel GPUs, Kokkos has started to implement a SYCL backend in addition to a OpenMPTarget backend. The earliest attempts date back to 2019[1] but a full implementation started towards the end of 2020[2]. At that point, there existed test beds at ANL for predecessors of the Intel GPUs deployed in Aurora but no Intel GPUs where available in our unit testing and continuous integration (CI) framework. Since SYCL is designed as a performance-portability solution and our CI had many NVIDIA GPUs, we used those for ensuring the correctness of the implementation on top of manual builds on the ANL testbeds. Therefore, the Kokkos SYCL backend has been working correctly on NVIDIA GPUs since we started implementing it. Nowadays, we can also have access to various resources with Intel GPUs including the ones used in Aurora (Intel Data Center Max 1100 and 1550) but we still keep a SYCL+Cuda build in our CI (NVIDIA A100). Note that Kokkos has separate (primary) backends for targeting NVIDIA and AMD GPUs built on top of CUDA and HIP respectively. Therefore, Kokkos SYCL backend is designed and optimized for Intel GPUs and support for other GPU architectures requires users to explicitly enabling the `Kokkos_ENABLE_UNSUPPORTED_ARCHS` `CMake` option. Nevertheless, there is interest for comparing performance between different backends both from the perspective of the Kokkos developer team and users. Hence, we allowed running Kokkos with SYCL on AMD GPUs after a user request[3] but never tested that option.

This report sheds some light on the compatibility with various oneAPI versions. Choosing oneAPI over any other SYCL implementation is an obvious choice as many of the extensions needed to implement all Kokkos features were only developed within oneAPI's DPC++ compiler in response to user requests for the Aurora supercomputer.

1. https://github.com/kokkos/kokkos/pull/2257
2. https://github.com/kokkos/kokkos/pull/3447
3. https://github.com/kokkos/kokkos/pull/6321

2. EARLY RESULTS ON FRONTIER WITH RECOMMENDED TOOLCHAIN

The most interesting platform for testing the SYCL backend implementation is the `Frontier` supercomputer at Oak Ridge National Laboratory (ORNL). In fact, there is a Frontier guide for SYCL recommending to load modules for ROCm 5.4.3 and oneAPI 2024.2.0.

Configuring Kokkos with these modules loaded is as easy as

```
cmake \
-DCMAKE_CXX_COMPILER=icpx \
-DKokkos_ENABLE_SYCL=ON \
-DKokkos_ARCH_AMD_GFX90A=ON \
-DKokkos_ENABLE_UNSUPPORTED_ARCHS=ON \
-DKokkos_ENABLE_TESTS=ON \
..
```

resulting in adding the architecture-specific compiler flags

```
-fsycl-targets=amdgcn-amd-amdhsa
-Xsycl-target-backend --offload-arch=gfx90a
```

in accordance with the documentation for Codeplay's plugin for AMD GPUs[4] that is used by the OneAPI compiler.

2.1 TEST RESULTS

As motivated earlier, a corresponding setup on NVIDIA and Intel GPUs passed the Kokkos test suite but with the setup above we were seeing internal compiler errors for the following executables:

- Kokkos_CoreUnitTest_SYCL1A.dir/sycl/TestSYCL_DeepCopyAlignment.cpp
- Kokkos_CoreUnitTest_SYCL1B.dir/sycl/TestSYCL_MDRangeReduce.cpp
- Kokkos_CoreUnitTest_SYCL3.dir/sycl/TestSYCLSharedUSM_ViewAPI_e.cpp
- Kokkos_AlgorithmsUnitTest_StdSet_Team_B
- Kokkos_AlgorithmsUnitTest_StdSet_Team_C
- Kokkos_AlgorithmsUnitTest_StdSet_C

and the test status looked like

- Kokkos_CoreUnitTest_SYCL_ViewSupport (Subprocess aborted)
- Kokkos_CoreUnitTest_Serial1 (Failed)
- Kokkos_CoreUnitTest_SYCL1A (Not Run)
- Kokkos_CoreUnitTest_SYCL1B (Not Run)
- Kokkos_CoreUnitTest_SYCL2A (Timeout)
- Kokkos_CoreUnitTest_SYCL3 (Not Run)
- Kokkos_CoreUnitTest_SYCLInterOpInit (SEGFAULT)

4. https://developer.codeplay.com/products/oneapi/amd/2025.1.1/guides/get-started-guide-amd#run-a-sample-application

- Kokkos_CoreUnitTest_SYCLInterOpStreams (SEGFAULT)
- Kokkos_CoreUnitTest_SYCLInterOpGraph (Failed)
- Kokkos_CoreUnitTest_Default (Failed)
- Kokkos_ContainersUnitTest_SYCL (Failed)
- Kokkos_UnitTest_Sort (Failed)
- Kokkos_UnitTest_Random (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_C (Not Run)
- Kokkos_AlgorithmsUnitTest_StdSet_D (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_E (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_B (Not Run)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_C (Not Run)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_I (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_L (Failed)

The used oneAPI SYCL implementation included an extension to support `printf` in device code that caused compilation errors like:

```
/autofs/nccs-svm1_sw/frontier/ums/ums015/oneapi/2024.2.0/compiler/2024.2/bin/
compiler/../../include/sycl/ext/oneapi/experimental/builtins.hpp:84:36: error: SYCL
kernel cannot call a variadic function
   84 |   return ::printf(__format, args...);
      |                                    ^
/ccs/home/darndt/kokkos/core/src/Kokkos_Printf.hpp:43:38: note: called by 'printf<const
char *>'
   43 |     sycl::ext::oneapi::experimental::printf(format, args...);
      |                                      ^
```

Therefore, we decided that this compiler version is unusable with Kokkos and that updating the oneAPI module on Frontier might be crucial for usability.

2.2 A NEWER ONEAPI VERSION

In a new attempt we tried the following compiler and toolchain versions (with the same Kokkos configuration):

- DPC++ 2025.0.0
- ROCm 6.1.3

This time, we were able to compile all unit tests but there were a lot of test failures:

- Kokkos_CoreUnitTest_SYCL_ViewSupport (Subprocess aborted)
- Kokkos_CoreUnitTest_Serial1 (Failed)
- Kokkos_CoreUnitTest_SYCL1A (Subprocess aborted)
- Kokkos_CoreUnitTest_SYCL1B (Subprocess aborted)

- Kokkos_CoreUnitTest_SYCL2A (Timeout)
- Kokkos_CoreUnitTest_SYCL3 (Failed)
- Kokkos_CoreUnitTest_SYCLInterOpInit (Subprocess aborted)
- Kokkos_CoreUnitTest_SYCLInterOpInit_Context (Failed)
- Kokkos_CoreUnitTest_SYCLInterOpStreams (Failed)
- Kokkos_CoreUnitTest_SYCLInterOpStreamsMultiGPU (Failed)
- Kokkos_CoreUnitTest_Default (Failed)
- Kokkos_CoreUnitTest_InitializeFinalize (Failed)
- Kokkos_CoreUnitTest_KokkosP (Failed)
- Kokkos_IncrementalTest_SYCL (Failed)
- Kokkos_ContainersUnitTest_SYCL (Failed)
- Kokkos_UnitTest_Sort (Failed)
- Kokkos_UnitTest_Random (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_C (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_D (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_E (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_I (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_L (Failed)
- Kokkos_UnitTest_SIMD (Failed)

Furthermore, calling `Kokkos::printf`/`sycl::ext::oneapi::experimental::printf` still resulted in runtime segmentation faults such as

```
6: [ RUN      ] sycl.kokkos_printf
6: Memory access fault by GPU node-4 (Agent handle: 0x557cb80) on address (nil).
Reason: Unknown.
```

making debugging any of these failing tests very hard.

One of the common themes among the failing unit tests seemed to be related to the reduction algorithm used. The Kokkos SYCL backend maps `parallel_reduce` to SYCL `parallel_for` calls instead of using reduction variables due to additional features Kokkos exposes.
Mapping just those `Kokkos::parallel_reduce` capabilities that could be easily implemented using SYCL's reduction interface doesn't seem to be worth the effort as we were seeing comparable performance on NVIDIA and Intel GPUs.

The Kokkos implementation exposes both a version taking advantage of shuffle instructions and one operating on local memory. Previous benchmarks have shown that the variant using local memory yields better performance than the version based on shuffles on Intel GPUs. Thus, Kokkos SYCL backend only uses local memory reduction by default. Unfortunately, there seem to be a memory fence issue when running on AMD GPUs that has been acknowledged by Codeplay that forces us to use the shuffle-based implementation instead. This means, in particular, that array reductions can't be used.

Therefore, the test results look much better after disabling all tests that use array reductions and use shuffle-reductions where we can:

- Kokkos_CoreUnitTest_SYCL_ViewSupport (Subprocess aborted)
- Kokkos_CoreUnitTest_Serial1 (Failed)
- Kokkos_CoreUnitTest_SYCL1A (Subprocess aborted)
- Kokkos_CoreUnitTest_SYCL1B (Failed)
- Kokkos_CoreUnitTest_SYCL2A (Timeout)
- Kokkos_CoreUnitTest_SYCL3 (Timeout)
- Kokkos_CoreUnitTest_SYCLInterOpInit (Failed)
- Kokkos_CoreUnitTest_SYCLInterOpInit_Context (Subprocess aborted)
- Kokkos_CoreUnitTest_SYCLInterOpStreamsMultiGPU (Failed)
- Kokkos_CoreUnitTest_KokkosP (Failed)
- Kokkos_ContainersUnitTest_SYCL (Timeout)
- Kokkos_UnitTest_Sort (Subprocess aborted)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_I (Failed)
- Kokkos_AlgorithmsUnitTest_StdSet_Team_L (Failed)
- Kokkos_UnitTest_SIMD (Failed)

Some remaining errors where related to implicitly nested `sycl::queue::submit` calls[5] and easily fixable. However, the issue with segmentation faults when calling `sycl::ext::oneapi::experimental::printf` remained.

Furthermore, we were still seeing launch errors such as

```
20: [ RUN      ] kokkosp.async_deep_copy
20: <HIP>[ERROR]:
20: UR HIP ERROR:
20:     Value:           209
20:     Name:            hipErrorNoBinaryForGpu
20:     Description:     no kernel image is available for execution on the device
20:     Function:        buildProgram
20:     Source Location: /tmp/tmp.6nq6FrBCn9/intel-llvm-mirror/build/_deps/unified-
runtime-src/source/adapters/hip/program.cpp:234
20:
20: unknown file: Failure
20: C++ exception with description "Enqueue process failed." thrown in the test body.
```

After reaching out to Codeplay, they discovered that enforcing the availability of a device-side assert implementation, possibly via a fallback mechanism was the culprit for the launch issues[6].

With all these fixes and workarounds (including making `Kokkos::printf` a no-op, we ended up with the following failing tests that are under active investigation:

- sycl.*_require

5. https://github.com/kokkos/kokkos/pull/7886
6. https://github.com/kokkos/kokkos/pull/8160

- sycl.reducers_*
- sycl.int_combined_reduce_mixed
- sycl.reduce_device_view_*
- sycl.large_parallel_for_reduce
- sycl_DeathTest.abort_from_device
- sycl.team_broadcast_long
- sycl.team_broadcast_char
- std_algorithms_unique_team_test.test_default_predicate
- std_algorithms_unique_team_test.test_default_predicate
- std_algorithms_partition_copy_team_test.all_true
- std_algorithms_partition_copy_team_test.all_false
- std_algorithms_partition_copy_team_test.random

suggesting that most Kokkos applications should run fine since the features used in these tests are less common in user code.

Of course, there are still some optimizations/features that are only available for Intel GPUs when using the SYCL backend:

- intel_get_cycle_counter -> clock_tic
- sycl::ext::oneapi::experimental::bfloat16;
- compile-time shuffles

3. ADAPTIVECPP

Compiling with `AdaptiveCpp` runs into a bunch of missing features such as

- sycl::ctz
- sycl::nan
- sycl::ext::oneapi::experimental::printf
- sycl::kernel_id, sycl::get_kernel_id
- sycl::ext::oneapi::group_local_memory_for_overwrite
- sycl::ext::intel::device_ptr, sycl::ext::intel::host_ptr
- sycl::ext::oneapi::experimental::this_sub_group
- sycl::atomic_fence

and we couldn't find a working compiler setup. Thus, we leave a proper evaluation of compatible features for a later study.

4. PERFORMANCE ONEAPI

After establishing that most Kokkos functionalities work after some tweaking, we move our focus to some performance benchmarks.

The following results were obtained on one node of the Frontier supercomputer at ORNL using one AMD Instinct® MI250X GPU.

As benchmarks, we choose the three main ingredients for a sparse matrix conjugate gradient solver (CG-solver). These are vector add (AXPBY), dot product (DOT), and sparse matrix-vector multiplication (SPMV) operations. These are the same benchmarks as in the corresponding comparison between raw SYCL and the Kokkos SYCL backend for Intel GPUs, see Arndt, Lebrun-Grandie, and Trott 2024, and can also be found in Daniel Arndt 2025.

4.1 AXPBY

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

Figure 1 shows that we essentially achieve the same effective bandwidth for the raw SYCL implementation and the Kokkos SYCL backend. However, Kokkos HIP backend performance better in the launch latency-limited regime of less than around 10^6 elements. This can be attributed to having different launch mechanisms available (constant, local, and global memory) while the SYCL implementation generally doesn't expose constant memory at all. For more work items we reach the same limit with all three implementations of around 1200 GB/s (75% of the peak memory bandwidth of 1.6 TB/s).

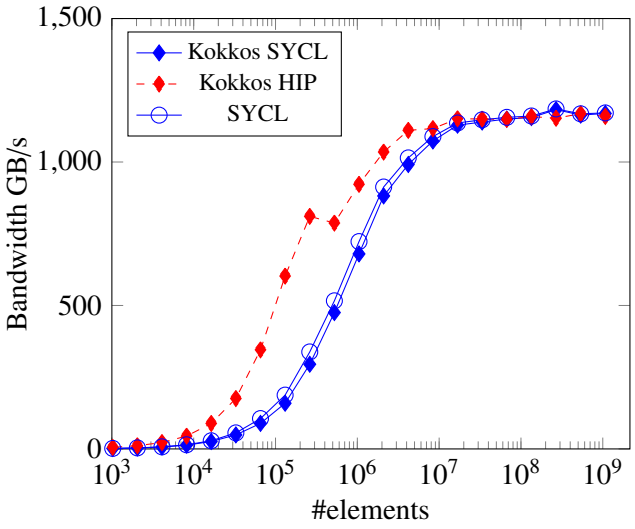
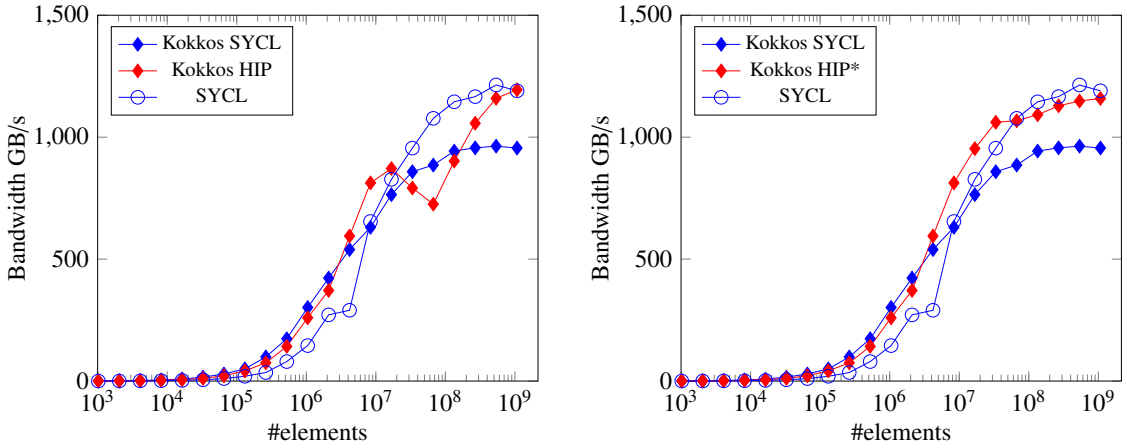


Figure 1. Achieved effective bandwidth for the AXPBY benchmark.

4.2 DOT

The `DOT` benchmark performs a single reduction with a `double` value type and has native implementations both in Kokkos and SYCL.

Figure 2a and 2b show the achieved effective bandwidth for the three implementations. Of note is that both Kokkos implementations achieve better performance for up to 10^7 elements (with the Kokkos SYCL implementation even slightly outperforming the Kokkos HIP implementation for up to 10^6 elements). For larger element sizes, the SYCL and HIP implementations both approach the same effective bandwidth as for the AXPBY benchmark while the Kokkos SYCL implementation only achieves up to around $1 \cdot 10^3$ GB/s. This is somewhat surprising as the same limitation can't be found when running on Intel GPUs. Of course, we were forced to use shuffle-based reductions instead of using local memory but that doesn't seem to have a significant impact here (not shown). Apart from that, the Kokkos HIP implementation drops in performance in Figure 2a due to suboptimal choices for block sizes. This can be overcome by forcing a larger block size as seen in Figure 2b making it arguably the best performing implementation. With kokkos#8268, these results are achieved when specifying `WorkItemProperty::HintLightWeight` in the kernel submission.



(a) HIP implementation unmodified

(b) HIP with 1024 instead of 256 threads per block when #elements ≥ 16777216.

Figure 2. Achieved effective bandwidth for the DOT benchmark.

4.3 SPMV

The final benchmark considered here is a sparse matrix-vector product for a finite element matrix corresponding to a heat conduction problem with a cubic grid and a maximum row length of 27. Since the sparse-matrix vector product exhibits multiple 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), this benchmark models hierarchical parallelism well.

Since the implementation here isn't trivial in Kokkos or SYCL, the implementations can be found below in Listing 1 and 2. Note that the SYCL implementation uses the hierarchical parallelism feature `parallel_for_work_group` that has now been deprecated and recommend to refrain from using[7]. The Kokkos SYCL implementation is based on a `sycl::parallel_for` implementation using `sycl::nd_range` in comparison.

7. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_hierarchical_data_parallel_kernels_deprecated and https://github.khronos.org/SYCL_Reference/iface/group.html#sycl-group

Listing 1. SPMV SYCL implementation

```
int rows_per_team = 128; //optimized for GPU
int team_size = 128;     //optimized for GPU
int vector_size = 8;     //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;
            // 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));
        });
    });
```

Listing 2. SPMV SYCL implementation

```
int rows_per_team = 1024; //optimized for GPU
int team_size = 1024;     //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;
    });
  });
});
```

Figure 3 compares the achieved bandwidth between the three implementations. Similar to the results in Arndt, Lebrun-Grandie, and Trott 2024, the SYCL implementation performs very poorly, in fact, even worse than on Intel GPUs. For the Kokkos HIP implementation we again observe a drop in performance

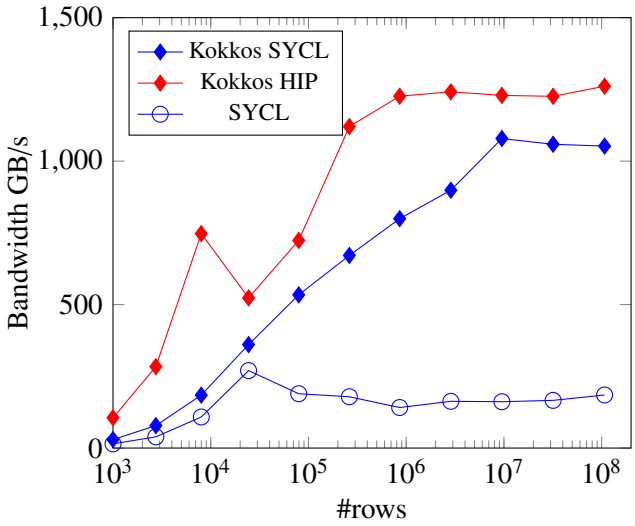


Figure 3. Achieved bandwidth for the SPMV benchmark

for moderate problem sizes but recover the same bandwidth as seen for the AXPBY and DOT benchmark for 10^6 and more rows. Finally, the Kokkos SYCL implementation needs around 10^7 rows to reach similar values as for the DOT benchmark.

5. SUMMARY

In this report, we discussed how well Kokkos' SYCL implementation works on AMD GPUs using Frontier's MI250X as an example. It turned out that running with the SYCL backend on AMD GPUs is much more challenging than on NVIDIA and Intel GPUs. Even after multiple more rounds of fixes, issues remain with the shuffle-based reduction algorithm and a small amounts of tests is still failing. On the performance side, we observe decent performance with the Kokkos SYCL implementation that still lacks behind the HIP backend by around 20% in two out of the three benchmarks considered. The SYCL native implementations show performance comparable with the Kokkos HIP backend for the AXPBY and the DOT benchmark.

In the future, we will incorporate testing the SYCL backend on AMD GPUs in Kokkos' CI and work on fixes or workarounds for the remaining test failures with the goal of providing Kokkos users with an alternative to the HIP backend.

6. REFERENCES

Arndt, Daniel, Damien Lebrun-Grandie, and Christian Trott. 2024. "Experiences with implementing Kokkos' SYCL backend." In *Proceedings of the 12th International Workshop on OpenCL and SYCL,* 1–11.

Daniel Arndt, Christian Trott, Damien Lebrun-Grandié. 2025. "Experiences with Kokkos' SYCL Backend using AMD GPUs." IWOCL 2025. https://www.iwocl.org/wp-content/uploads/iwocl-2025-daniel-arndt-kokkos.pdf.