Porting Sparse Linear Algebra to Intel GPUs

Yuhsiang M. Tsai[1(✉)], Terry Cojean[1], and Hartwig Anzt[1,2]

[1] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{yu-hsiang.tsai,terry.cojean,hartwig.anzt}@kit.edu
[2] University of Tennessee, Knoxville, TN 37996, USA

Abstract. With discrete Intel GPUs entering the high performance computing landscape, there is an urgent need for production-ready software stacks for these platforms. In this paper, we report how we prepare the Ginkgo math library for Intel GPUs by developing a kernel backed based on the DPC++ programming environment. We discuss conceptual differences to the CUDA and HIP programming models and describe workflows for simplified code conversion. We benchmark advanced sparse linear algebra routines utilizing the converted kernels to assess the efficiency of the DPC++ backend in the hardware-specific performance bounds. We compare the performance of basic building blocks against routines providing the same functionality that ship with Intel's oneMKL vendor library.

Keywords: oneAPI · Intel GPUs · Ginkgo · Math library · SpMV

1 Introduction

In the past, Intel GPUs were primarily available as an integrated component of Intel consumer-grade CPU architectures. With the announcement that the Aurora Supercomputer will be composed of general purpose Intel CPUs complemented by discrete Intel GPUs, it becomes clear that Intel has committed to enter the arena of discrete high performance GPUs. Compared to integrated GPUs, discrete GPUs are usually not exclusively intended to accelerate graphics, but they are designed to also deliver computational power that can be used, e.g., for scientific computations. To enable the programmers to use Intel GPUs, Intel has teamed up with partners from academia and industry to create the oneAPI ecosystem, a platform for C++ developers to develop code in the DPC++ language, based on the SYCL language, that can be executed on any Intel device, including CPUs, GPUs, and FPGAs. As application scientists are in need of high performance math functionality for Intel GPUs, we develop a DPC++ backend for the Ginkgo open source math library that enables to run both basic linear algebra building blocks and complex algorithms like iterative Krylov solvers on Intel's GPUs. Up to our knowledge, we are the first to present the functionality and performance of an open source math library on Intel discrete GPUs.

© Springer Nature Switzerland AG 2022
R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 57–68, 2022.
https://doi.org/10.1007/978-3-031-06156-1_5

In this paper, we describe the process of preparing Ginkgo for Intel's GPUs by first providing an overview of the Ginkgo library design in Sect. 2 and introducing the oneAPI ecosystem and the DPC++ programming model in Sect. 3. The core of the paper is Sect. 4, where we discuss some differences between the CUDA/HIP programming environment and the oneAPI environment, detail how we reflect these particularities in the development of the DPC++ backend, and report how we developed a small framework for converting CUDA kernel code to DPC++ equivalents. In Sect. 5, we evaluate the performance of Ginkgo on different Intel GPU generations: we initially benchmark both the Intel generation 9 and 12 GPUs in terms of feasible bandwidth and peak performance to derive a roofline model, then evaluate the performance of Ginkgo's SpMV kernels (also in comparison to Intel's oneMKL library), and finally assess the performance of Ginkgo's Krylov solvers. We conclude with a summary of the porting effort and performance evaluation in Sect. 6.

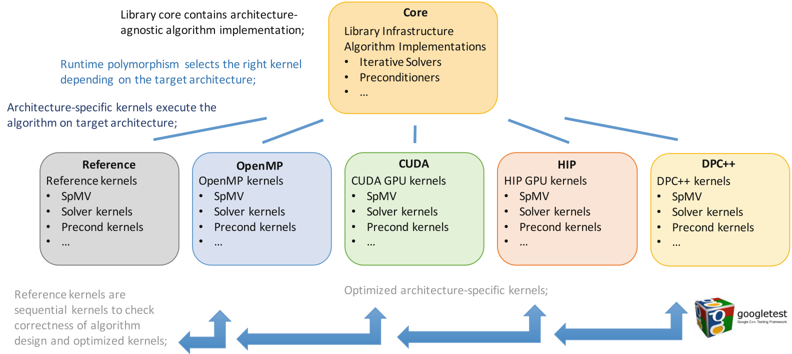

Fig. 1. The Ginkgo library design overview.

2 Ginkgo Design

Ginkgo [1] is a GPU-focused cross-platform math library focusing on sparse linear algebra. The library design is guided by combining ecosystem extensibility with heavy, architecture-specific kernel optimization using the platform-native languages CUDA (NVIDIA GPUs), HIP (AMD GPUs), or OpenMP (Intel/AMD/ARM multicore) [2]. The software development cycle ensures production-quality code by featuring unit testing, automated configuration and installation, Doxygen code documentation, as well as continuous integration and continuous benchmarking framework. Ginkgo provides a comprehensive set of sparse BLAS operations, iterative solvers including many Krylov methods, standard and advanced preconditioning techniques, and cutting-edge mixed precision methods.

A high-level overview of Ginkgo's software architecture is visualized in Fig. 1. The library design collects all classes and generic algorithm skeletons in

the "core" library which, however, is useless without the driver kernels available in the "omp", "cuda", "hip", and "reference" backends. We note that "reference" contains sequential CPU kernels used to validate the correctness of the algorithms and as the reference implementation for the unit tests realized using the googletest framework. We note that the "cuda" and "hip" backends are very similar in kernel design, so we have "shared" kernels that are identical for the NVIDIA and AMD GPUs up to kernel configuration parameters [6]. Extending Ginkgo's scope to support Intel GPUs via the DPC++ language, we add the "dpcpp" backend containing corresponding kernels in DPC++.

3 The oneAPI Programming Ecosystem

oneAPI[1] is an open and free programming ecosystem that aims at providing portability across a wide range of hardware platforms from different architecture generations and vendors. The oneAPI software stack is structured with the new DPC++ programming language at its core, accompanied by several libraries to ease parallel application programming.

DPC++ is a community-driven (open-source) language based on the ISO C++ and Khronos' SYCL standards. The concept of DPC++ is to enhance the SYCL [4] ecosystem with several additions that aim at improving the performance on modern hardware, improving usability, and simplifying the porting of classical CUDA code to the DPC++ language. Two relevant features originally introduced by the DPC++ ecosystem now also integrated into the SYCL standard are[2]: 1) a new subgroup concept that can be used inside kernels. This concept is equivalent to CUDA warps (or SIMD on CPUs) and allows optimized routines such as subgroup-based shuffles. In the Ginkgo library, we make extensive use of this capability to boost performance. 2) a new Unified Shared Memory (USM) model which provides new `malloc_host` and `malloc_device` operations to allocate memory which can either be accessed both by host or device or respectively accessed by a device only. Additionally, the new SYCL `queue` extensions facilitates the porting of CUDA code as well as memory control. Indeed, in pure SYCL, memory copies are entirely asynchronous and hidden from the user, since the SYCL programming model is based on tasking with automatic discovery of task dependencies.

Another important aspect of oneAPI and DPC++ is that they adopt platform portability as the central design concept. Already the fact that DPC++ is based on SYCL (which leverages the OpenCL's runtime and SPIRV's intermediate kernel representation) provides portability to a variety of hardware. On top of this, DPC++ develops a plugin API that allows to develop new backends and switch dynamically between them[3]. Currently, DPC++ supports the standard OpenCL backend, a new Level Zero backend which is the backend of

[1] https://spec.oneApi.com/versions/latest/index.html.

[2] These extensions are now part of the SYCL 2020 Specification: https://www.khronos.org/news/press/khronos-releases-sycl-2020-final-specification.

[3] https://intel.github.io/llvm-docs/PluginInterface.html.

choice for Intel hardware[4], and an experimental CUDA backend for targeting CUDA-enabled GPUs. As our goal is to provide high performance sparse linear algebra functionality on Intel GPUs, we focus on the Intel Level Zero backend of DPC++.

4 Porting to the DPC++ Ecosystem

Though porting Ginkgo to a new hardware ecosystem requires acknowledging the hardware-specific characteristics, the Ginkgo design exposed in Sect. 2 induces a general porting workflow: 1) As a first step, core library infrastructure needs to be ported manually. This includes the Ginkgo Executor which allows transparent and automatic memory management as well as the execution of kernels on different devices. Another example of manual porting in this preparatory step is the cooperative group and other shared kernel helper interfaces used for writing portable kernels and simplify advanced operations. 2) A set of scripts can be used to generate non-working definitions of all kernels for the new backend. The completion of this step creates a compilable backend for the new hardware ecosystem. 3) For an initial kernel implementation, we rely whenever possible on existing tools to facilitate the automatic porting of kernel implementations from one language to the target language, doing only manual fixes when appropriate. The successful completion of this step provides a working backend. 4) Finally, we analyze and validate the observed performance for the ported kernels. Often, simple kernels already provide competitive performance, but advanced kernels require either manual tuning or algorithmic adaptation to reach the hardware limits.

In this section, we concentrate on step 3) of this workflow and parts of step 1). These steps which we detail now are the more library agnostic aspect of the porting workflow and our lessons learned can be of practical use to other libraries. In addition, step 4) is a more complex effort and some portions of the library have been tuned, such as the Ginkgo SpMV kernels, and their performance will be showcased in Sect. 5. To facilitate the porting in step 3), we can rely on the Intel "DPC++ Compatibility Tool" (DPCT), which converts CUDA code into compilable DPC++ code. DPCT is not expected to automatically generate a DPC++ "production-ready" executable code, but "ready-to-compilation" and it requires the developer's attention and effort in fixing conversion issues and tuning it to reach performance goals. However, with oneAPI still being in its early stages, DPCT still has some flaws and failures, and we develop a customized porting workflow using the DPC++ Compatibility Tool at its core, but embedding it into a framework that weakens some DPCT prerequisites and prevents incorrect code conversion. In general, DPCT requires not only knowledge of the functionality of a to-be-converted kernel, but also knowledge of the complete library and its design. This requirement is hard to fulfill in practice, as for complex libraries, the dependency analysis may exceed the DPCT capabilities. Additionally, many libraries do not aim at converting all code to DPC++, but

[4] https://spec.oneApi.com/level-zero/latest/core/INTRO.html.

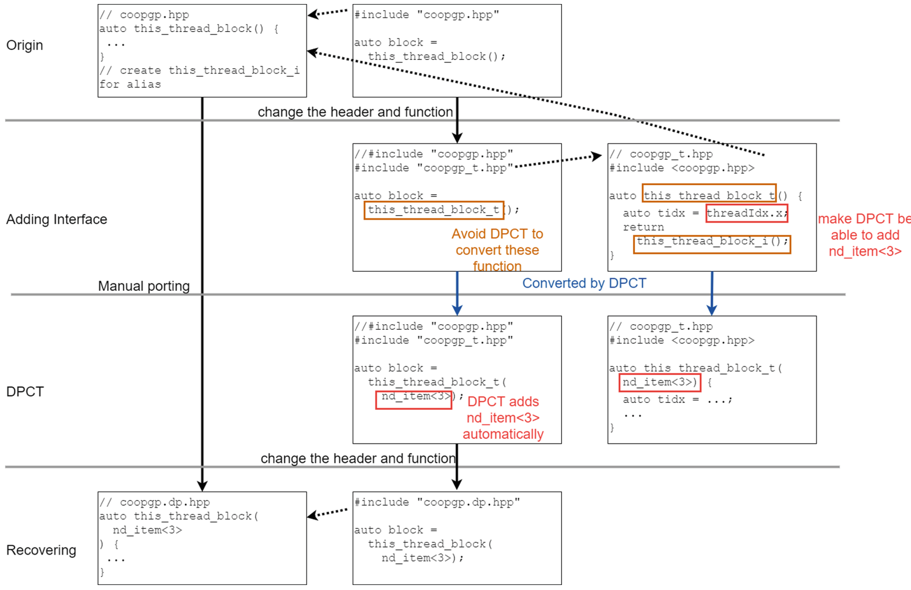

Fig. 2. Summary of the workflow used to port the cooperative groups functionality and isolating effort such that we get the correct converted DPC++ codes.

only a subset to enable the dedicated execution of specific kernels on DPC++-enabled accelerators. Thus, we employ a strategy where we first isolate kernels we want to convert and then re-integrate them into the library.

Isolated Kernel Modification. DPCT converts all files related to the target file containing any CUDA code that are in the target (sub)folders. To prevent DPCT from converting files that we do not want to be converted, we have to artificially restrict the conversion to the target files. We achieve this by copying the target files into a temporary folder and considering the rest of the Ginkgo software as a system library. After the successful conversion of the target file, we copy the file back to the correct destination in the new DPC++ submodule. By isolating the target files, we indeed avoid additional changes and unexpected errors, but we also lose the DPCT ability to transform CUDA kernel indexing into the DPC++ `nd_item<3>` equivalent. As a workaround, we copy simple headers to the working directory containing the thread_id computation helper functions of the CUDA code such that DPCT can recognize them and transform them into the DPC++ equivalent. For those complicated kernels, DPCT fails in the kernel conversion, and we need a fake interface that enables DPCT to apply the code conversion for `nd_item<3>`.

Fake Interface - Workaround for Cooperative Groups. While DPC++ provides a subgroup interface featuring shuffle operations, this interface is different from CUDA's cooperative group design as it requires the subgroup size as a function attribute and does not allow for different subgroup sizes in the

same global group. As Ginkgo implementations aim at executing close to the hardware-induced limits, we make heavy use of cooperative group operations. Based on the DPC++ subgroup interface, we implement our own DPC++ cooperative group interface. Specifically, to remove the need for an additional function attribute, we add the `item_ct1` function argument into the group constructor. As the remaining function arguments are identical to the CUDA cooperative group function arguments, we therewith achieve a high level of interface similarity. This workflow resolves the porting not only for the cooperative group functionality but also other custom kernels replacing the automated DPCPP conversion.

A notable difference to CUDA is that DPC++ does not support subgroup vote functions like "ballot", or other group mask operations yet. To emulate this functionality, we need to use a subgroup reduction provided by oneAPI to emulate these vote functions in a subgroup setting. This lack of native support may affect the performance of kernels relying on these subgroup operations. We visualize in Fig. 2 the workflow we use to port code making use of the cooperative group functionality via four steps:

1. Origin: We prepare an alias to the cooperative group function such that DPCT does not catch the keyword. We create this alias in a fake cooperative group header we only use during the porting process.
2. Adding Interface: As explained previously, we isolate the files to prevent DPCT from changing other files. We also add the simple interface including `threadIdx.x` and make use of the alias function. For the conversion to succeed, it is required to return the same type as the original CUDA type, which we need to extract from the CUDA cooperative group function `this_thread_block`.
3. DPCT: Apply DPCT on the previously prepared files. Adding `threadIdx.x` indexing to the function allows DPCT to generate the `nd_item<3>` indexing.
4. Recovering: During this step, we change the related cooperative group functions and headers to the actual DPC++ equivalent. We implement a complete header file that ports all the cooperative group functionality to DPC++.

In Fig. 3, the final result of the porting workflow on a toy example with cooperative groups. For the small example code in Fig. 3a, if we do not isolate the code, DPCT will throw an error like Fig. 3b once encountering the cooperative group keyword. A manual implementation of the cooperative group equivalent kernel is shown in Fig. 3c. Our porting workflow generates the code shown in Fig. 3d, which is almost identical to the original CUDA code Fig. 3a.

Pushing for Backend Similarity. To simplify the maintenance of the platform-portable Ginkgo library, our customized porting workflow uses some abstraction to make the DPC++ code in this first version look more similar to CUDA/HIP code. We note that this design choice is reflecting that Ginkgo was originally designed as a GPU-centric sparse linear algebra library using the CUDA programming language and CUDA design patterns for implementing GPU kernels and that the developers of Ginkgo are currently used to designing GPU kernels in CUDA. However, this may not be preferred by developers

```
template <unsigned subwarp_size, typename ValueType>
__global__ __launch_bounds__(32) void reduce(ValueType *a) {
    auto subwarp = cooperative_groups::tiled_partition<16>(
        cooperative_groups::this_thread_block());
    auto local_data = a[threadIdx.x];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[threadIdx.x] = local_data;
}
```

(a) CUDA cooperative group example

```
template<unsigned subgroup_size, typename ValueType>
void reduce(ValueType *a, sycl::nd_item<3> item_ct1) {
    /*
    DPCT1007:0: Migration of this CUDA API is not suppoted by the Intel(R)
    DPC++ Compatibility Tool.
    */
    sycl::group<3> subwarp = item_ct1.get_sub_group();
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

(b) DPCT conversion reports an error

```
template<unsigned subgroup_size, typename ValueType>
[[ intel::reqd_sub_group_size(subgroup_size) ]]
void reduce(ValueType *a, sycl::nd_item<3> item_ct1) {
    auto subwarp = item_ct1.get_sub_group();
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subgroup_size; bitmask <<= 1) {
        const auto remote_data = subwarp.shuffle_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

(c) Manual DPC++ subgroup implementation. The main difference from CUDA are in orange

```
template<unsigned subgroup_size, typename ValueType>
void reduce(float *a, sycl::nd_item<3> item_ct1)  {
    auto subwarp = group::tiled_partition<subgroup_size>(
        group::this_thread_block(item_ct1));
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

(d) The result converted by our porting script

Fig. 3. The cooperative group example

used to programming in task-based languages, and it may also narrow down the tasking power of the SYCL language. We may thus decide at a later point to move closer to the SYCL programming style, which is possible given GINKGO's strict decoupling between algorithms and hardware backends. For now, we aim for a high level of code similarity by not only adding the customized cooperative group interface previously discussed, but also adding a dim3 implementation layer for DPC++ kernel launches that uses the same parameters and parameter order as CUDA and HIP. We simply reverse the dim3 in the interface layer.

One fundamental difference remaining between the CUDA or HIP ecosystems and DPC++ is that the latter handles the static and dynamic memory allocation in the main component. CUDA and HIP handle the allocation of static shared memory inside the kernel and the allocation of dynamic shared memory in the kernel launch parameters. Another difference is the kernel invocation syntax since DPC++ relies on a hierarchy of calls first to a queue, then a parallel instantiation. For consistency, we add another layer that abstracts the combination of DPC++ memory allocation and DPC++ kernel invocation away from the user. This enables a similar interface for CUDA, HIP, and DPC++ kernels for the main component, and shared memory allocations can be perceived as a kernel feature, see Fig. 4. In Fig. 4, the right purple block (additional_layer_call) has the same structure as the left gray block (cuda_kernel_call). The enhanced porting script not only handles the kernel conversion but also the addition of the intermediate layer.

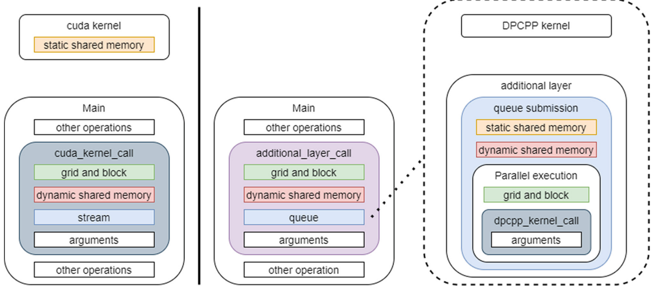

Fig. 4. Hierarchical view of usual CUDA (left) and DPC++ (right) kernel call and parameters. Wrapping the hardware-specific kernels into an intermediate layer enables consistency in the kernel invocation across all backends.

5 Performance Assessment of Ginkgo's DPC++ Backend

Experiment Setup. In this paper, we consider two Intel GPUs: the generation 9 (Gen9) integrated GPU UHD Graphics P630 with a theoretical bandwidth of 41.6 GB/s and the generation 12 Intel® Iris® Xe Max discrete GPU (Gen12)[5] which features 96 execution units and a theoretical bandwidth of 68 GB/s. To better assess the performance of either GPUs, we include in our analysis the performance we can achieve in bandwidth tests, performance tests, and sparse linear algebra kernels. We note that the Gen12 architecture lacks native support for IEEE 754 double precision arithmetic, and can only emulate double precision arithmetic with significantly lower performance. Given that native support for double precision arithmetic is expected for future Intel GPUs and using the double precision emulation would artificially degrade the performance results while not providing insight whether Ginkgo's algorithms are suitable for Intel GPUs, we use single precision arithmetic in all performance evaluation on the Gen12 architecture[6]. The DPC++ version we use in all experiments is Intel oneAPI DPC++ Compiler 2021.1 (2020.10.0.1113). All experiments were conducted on hardware that is part of the Intel DevCloud.

Bandwidth Tests and Experimental Performance Roofline. Initially, we evaluate the two GPUs in terms of architecture-specific performance bounds. For that purpose, we use the BabelStream [3] benchmark to evaluate the peak bandwidth, and the mixbench [5] benchmark to evaluate the arithmetic performance. In the upper part of Fig. 5, we visualize the bandwidth we achieve for different memory-intense operations. On both architectures, the Dot kernel requiring a global synchronization achieves lower bandwidth than the other kernels. We furthermore note that the Gen12 architecture achieves for large array

[5] https://ark.intel.com/content/www/us/en/ark/products/211013/intel-iris-xe-max-graphics-96-eu.html.

[6] Ginkgo is designed to compile for IEEE 754 double precision, single precision, double precision complex, and single precision complex arithmetic.

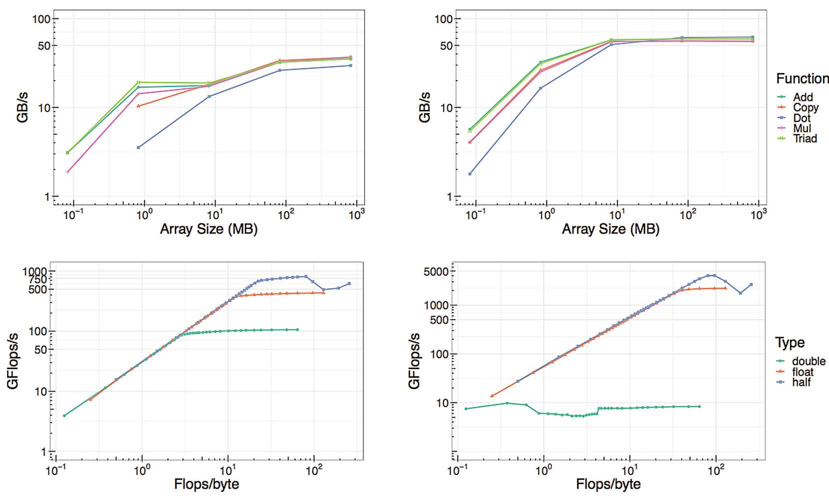

Fig. 5. Top: Bandwidth analysis on the Intel GEN9 (left) and the GEN12 (right) GPUs using double and single precision values, respectively. Bottom: Experimental performance roofline for the GEN9 (left) and GEN12 (right) GPUs.

sizes about 58 GB/s and the GEN9 achieves 37 GB/s. The experimental roofline visualized in the lower part of Fig. 5 reveals that the GEN9 architecture achieves about 105 GFLOP/s, 430 GFLOP/s, and 810 GFLOP/s for IEEE double precision, single precision, and half precision arithmetic, respectively. The GEN12 architecture does not provide native support for IEEE double precision, and the double precision emulation achieves only 8 GFLOP/s. On the other hand, the GEN12 architecture achieves 2.2 TFLOP/s and 4.0 TFLOP/s for single precision and half precision floating point operations.

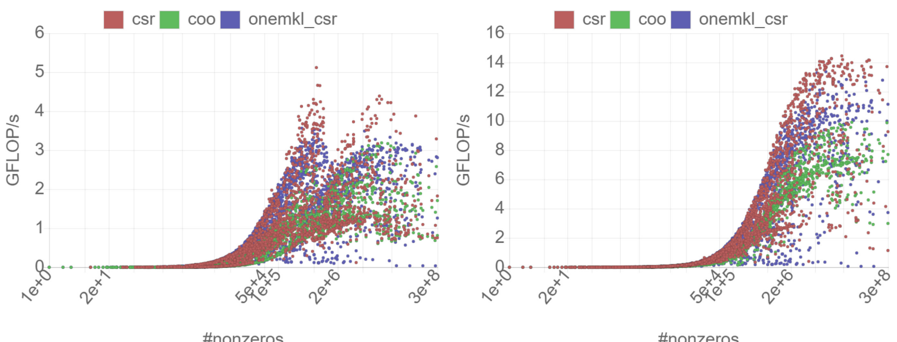

Fig. 6. SPMV kernel performance for GINKGO and Intel's oneMKL library on GEN9 (left) and GEN12 (right) using double and single precision, respectively.

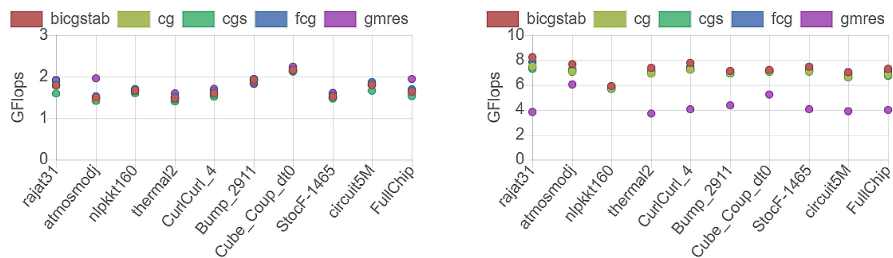

Fig. 7. Performance evaluation of Ginkgo's Krylov solvers on Intel's Gen9 (left) and Gen12 (right) GPUs.

SpMV Performance Analysis. An important routine in sparse linear algebra is the **Sparse Matrix Vector product** (SpMV). This kernel reflects how a discretized linear operator acts on a vector, and therewith plays the central role in the iterative solution of linear problems and eigenvalue problems. We consider two sparse matrix formats: 1) the "COOrdinate format" (COO) that stores all nonzero entries of the matrix along with their column- and row-indices, and the "Compressed Sparse Row" (CSR) format that further reduces the memory footprint of the COO format by replacing the row-indices with pointers to the first element in each row of a row-sorted COO matrix. We focus on these popular matrix formats not only because of their widespread use, but also because Intel's oneMKL library provides an optimized CSR-SpMV routine for Intel GPUs.

In Fig. 6, we visualize the performance of the CSR and COO SpMV kernels of the Ginkgo library along with the performance of the CSR SpMV kernel from the oneAPI library. Each dot represents the performance achieved for one of the test matrices of the Suite Sparse Matrix Collection. Ginkgo's CSR reaches up to 4 GFlop/s for several problems using double precision arithmetic, oneMKL CSR up to 3 GFlop/s similarly to Ginkgo's COO format. For Gen12, Ginkgo's CSR reaches up to 14 GFlop/s, oneMKL 13 GFlop/s and Ginkgo's COO 10 GFlop/s. These results highlight that Ginkgo's formats CSR and COO are at least competitive with the oneMKL CSR on both Gen9 and Gen12[7]. The achieved performance in terms of percentage of peak bandwidth are exposed in Fig. 8.

Krylov Solver Performance Analysis. We now turn to advanced numerical algorithms typical to scientific simulation codes. The Krylov solvers we consider – CG, BiCGSTAB, CGS, FCG, and GMRES – are all iterative methods popular for solving large sparse linear systems. They all have the SpMV kernel as the central building block, and we use Ginkgo's COO SpMV kernel and test matrices from the Suite Sparse Matrix Collection that are orthogonal in their characteristics and origin. We run the solver experiment for 1,000 solver iterations after a warm-up phase. In Fig. 7, we visualize the performance of the Krylov solvers on the

[7] At the point of writing, oneMKL does not provide a COO implementation and CSR can only operate on shared memory on the Gen12 architecture.

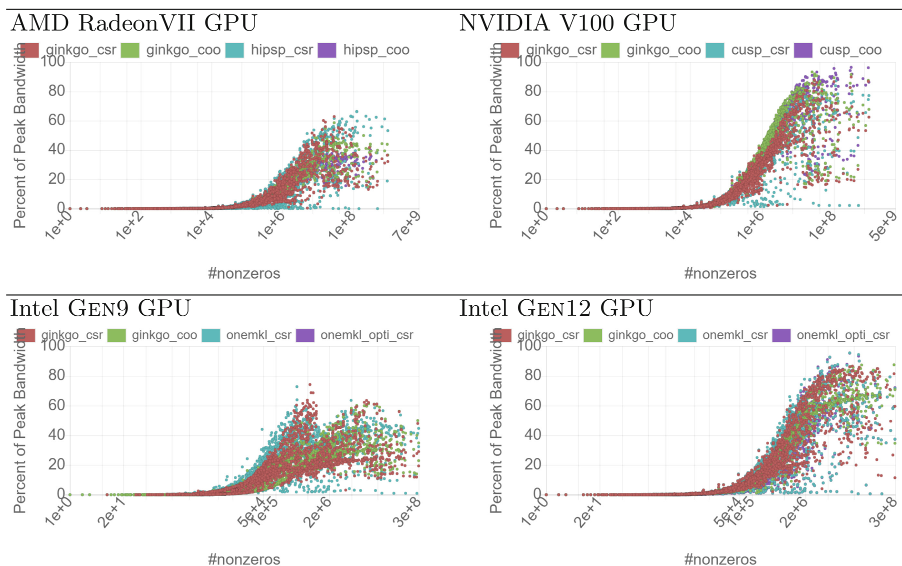

Fig. 8. SpMV performance relative to the hardware bounds on various GPUs.

Gen9 architecture (left) and Gen12 architecture (right). On the Gen9, the performance varies between 1.5 GFLOP/s and 2.5 GFLOP/s. We notice that the performance differences in-between the solvers are quite small compared to the performance differences for the distinct problems. Running Ginkgo's Krylov solvers in single precision on the Gen12 architecture, we achieve between 5 GFLOP/s and 9 GFLOP/s for the distinct systems. We note that all Krylov solvers based on short recurrences (BiCGSTAB, CG, CGS, FCG) are very similar in terms of performance, while GMRES usually achieves lower performance. This highlights that the kernels of GMRES require specific tuning.

Platform Portability. Finally, we evaluate the hardware efficiency of the Ginkgo DPC++ backend compared to the other backends. For that, we focus on the relative performance the functionality achieves on GPUs from AMD, NVIDIA, and Intel, taking the theoretical performance limits reported in the GPU specifications as the baseline. This approach reflects the aspect that the GPUs differ significantly in their performance characteristics, and that Intel's oneAPI ecosystem and GPU architectures are still under active development and have not yet reached the maturity level of other GPU computing ecosystems. At the same time, reporting the performance relative to the theoretical limits allows us to both quantify the suitability of Ginkgo's algorithms and to estimate the performance we can expect for Ginkgo's functionality when scaling up the GPU performance. In Fig. 8 we report the relative performance of different SpMV kernels on AMD Radeon VII ("hip" backend), NVIDIA V100 ("cuda" backend), and Intel Gen9 and Gen12 GPUs (both "dpcpp" backend). As expected, the achieved bandwidth heavily depends on the SpMV kernel and

the characteristics of the test matrix. Overall, the performance figures indicate that the SpMV kernels achieve about 90% of peak bandwidth on V100 and Gen12, and about 60–70% of peak bandwidth on RadeonVII and Gen9. On all hardware, Ginkgo's SpMV kernels are competitive to the vendor libraries, indicating the validity of the library design and demonstrating good performance portability.

6 Summary and Outlook

We have prepared the Ginkgo open source math library for Intel GPUs by developing a DPC++ backend. We presented strategies that are practical to accommodate the design differences between CUDA/HIP and the oneAPI ecosystem. We also evaluated the efficiency of Ginkgo's functionality in terms of translating hardware performance into algorithm performance and comparing basic building blocks against equivalent kernels shipping with Intel's oneMKL library. In this performance evaluation, we demonstrated that Ginkgo's kernels are competitive to Intel's oneMKL library, and that Ginkgo's advanced math functionality is readily available to run on Intel GPUs. While the oneAPI ecosystem itself aims for providing portability to GPUs from other vendors, we have acknowledge that this is currently not possible, and we thus have to postpone the evaluation of Ginkgo's DPC++ backend on AMD and NVIDIA platforms.

References

1. Anzt, H., et al.: Ginkgo: a high performance numerical linear algebra library. J. Open Source Softw. **5**(52), 2260 (2020). https://doi.org/10.21105/joss.02260
2. Cojean, T., Tsai, Y.H.M., Anzt, H.: Ginkgo - a math library designed for platform portability (2020). https://www.sciencedirect.com/science/article/abs/pii/S0167819122000096
3. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: Evaluating attainable memory bandwidth of parallel programming models via babelstream. Int. J. Comput. Sci. Eng. **17**, 247–262 (2017)
4. Keryell, R., Reyes, R., Howes, L.: Khronos SYCL for OpenCL: a tutorial. In: Proceedings of the 3rd International Workshop on OpenCL, IWOCL 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2791321.2791345
5. Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. J. Parallel Distrib. Comput. **107**, 37–56 (2017). https://doi.org/10.1016/j.jpdc.2017.04.002
6. Tsai, Y.M., Cojean, T., Ribizel, T., Anzt, H.: Preparing Ginkgo for AMD GPUs – a testimonial on porting CUDA code to HIP. In: Balis, B., et al. (eds.) Euro-Par 2020. LNCS, vol. 12480, pp. 109–121. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71593-9_9