Ginkgo - A Math Library designed for Platform Portability

Terry Cojean[⋆], Yu-Hsiang Mike Tsai[⋆], and Hartwig Anzt[⋆†]

[⋆]*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany*
[†]*Innovative Computing Lab, University of Tennessee, USA*

Abstract

In an era of increasing computer system diversity, the portability of software from one system to another plays a central role. Software portability is important for the software developers as many software projects have a lifetime longer than a specific system, e.g., a supercomputer, and it is important for the domain scientists that realize their scientific application in a software framework and want to be able to run on one or another system. On a high level, there exist two approaches for realizing platform portability: 1) implementing software using a portability layer leveraging any technique which always generates specific kernels from another language or through an interface for running on different architectures; and 2) providing backends for different hardware architectures, with the backends typically differing in how and in which programming language functionality is realized due to using the language of choice for each hardware (e.g., CUDA kernels for NVIDIA GPUs, SYCL (DPC++) kernels to targeting Intel GPUs and other supported hardware, ...). In practice, these two approaches can be combined in applications to leverage their respective strengths. In this paper, we present how we realize portability across different hardware architectures for the Ginkgo library by following the second strategy and the goal to not only port to new hardware architectures but also achieve good performance. We present the Ginkgo library design, separating algorithms from hardware-specific kernels forming the distinct hardware executors, and report our experience when adding execution backends for NVIDIA, AMD, and Intel GPUs. We also present the performance we achieve with this approach for distinct hardware backends.

Keywords: Porting to GPU accelerators; Platform Portability; Performance portability; AMD; NVIDIA; Intel

1. Introduction

When surveying the research software landscape, we can identify some software products that have been successful for several decades ([1, 2]). On the other hand, some libraries are successful for a while, and then fade out. When investigating the source of decline for some products, it is often that the jump from one hardware architecture to the next was too big, and the product failed to keep up with the development of other software and hardware ecosystems.[1] In that sense, the lack of software portability and the lack of flexibility to embrace future hardware designs is a time bomb that limits the lifetime of software.

The lack of platform portability becomes even more critical as we see an explosion of diversity in hardware architectures employed in supercomputers. In the last century, the hardware development was mostly incremental, as it was driven by the clock frequency increase of the processors ([3]). During that time, the software developers usually succeeded in transferring to newer chip technologies by applying minor modifications or by simply leveraging the "free lunch" ([4]) that came with higher operating frequency. That said, the move from single-core processors to multi-core processors in the early 21st century was incremental enough to be mastered by many software products that did not embrace platform portability as a central design principle. This is partly because using pragmas and the OpenMP language allowed for a smooth transition. In addition, only the performance—not the functionality—of software was endangered when ignoring multi-threaded or multi-core hardware capacity. In fact, single-threaded software remains functional and can still achieve acceptable performance. However, at least since the rise of many-core accelerators (e.g., GPUs) and the adoption of special function units and lightweight ARM processors for supercomputing, software libraries can no longer ignore the hardware changes. As a consequence, the lack of platform portability for emerging and future hardware technology is among the main threats to the sustainability of a given software product.

In the design of the GINKGO [5] open-source software library, we have the burden and privilege to start from scratch, and to apply the lessons learned in other software projects. In this paper, we first recall in Section 2 on different levels of portability before we detail in Section 3 how we develop GINKGO with platform portability as a central design principle. We present in Section 4 and Section 5 our strategy for adopting new hardware architectures and report our experiences in porting to new backends, namely AMD GPUs and Intel GPUs. In Section 6 we present a brief performance evaluation on how architecture-native backends perform in comparison to using platform portability layers. We conclude by summarizing our central findings in Section 7.

[1]Another reason is often the lack of resources for sustained software development, but here we refrain from addressing the topic of under-funding the field of research software engineering.

2. Platform Portability

There are at least two dimensions to platform portability. One aspect is the mapping of a parallel algorithm onto different sets of hardware which relates to scheduling techniques and parallel programming representations such as task graphs. Another aspect is the expression of algorithms or basic building blocks (kernels) in an ecosystem that allows for the efficient execution on specific hardware architectures. We will here focus on this second dimension of performance portability.

2.1. The Levels of Portability

There are multiple levels of platform portability. Depending on the use case, platform targets, and objectives, some applications may find it sufficient to restrict themselves to a specific portability level. The first distinct level is *no portability*, where the code compiles and runs for only one type of high-performance computing (HPC) system. The same sort of hardware and compute capabilities are expected. Another option is to support *partial software portability*. An application using such a model will be dependent on some platform model abstraction. For example, the model could expect any CPU type combined with one or more accelerators, either from AMD or NVIDIA. In such a case, a hybrid programming approach featuring a CPU programming model like OpenMP is combined with an accelerator programming model like HIP to ensure portability (and possibly good performance) on the machine. As a more advanced case, one might consider *full software portability*, where the application can execute and run on any type of platform, including hypothetical future machines that might feature field-programmable gate arrays (FPGAs). In this case, a practical example is the SYCL programming model, which features compiler backends that support some FPGAs, all mainstream HPC accelerators, and ARM-based hardware. Finally, and especially important for HPC applications, there is the level of *performance portability*, which means that the code will not only compile and run on target platforms, but it will also achieve high efficiency by providing performance close to the machine's total capabilities. To achieve performance portability, one needs good software design practices (e.g., code portability) *and* full command and understanding of the problems inherent in computing unit granularity vs. problem granularity. The latter requires using specific programming techniques to fully express an application's parallelism and scheduling to spread the workload, dynamically, depending on the machine hardware's computing units.

2.2. Writing Performance Portable Applications

Ignoring efforts that are likely doomed to fail, as they are based on a permanent redesign to reflect the changes in hardware architecture, one can identify two different approaches to enable cross-platform portability and readiness for future hardware architectures.

Relying on a Portability Layer The idea behind a portability layer is that the user writes the code once in a high-level language, and the code is then automatically transformed into a source code tailored for a specific architecture and its ecosystem before being executed thanks to an abstraction. This idea was already applied to software performance portability at the assembly level by using preprocessor macros [6]. For the more modern problem of expressing parallel algorithms on a variety of architectures including accelerators, the portability layer enabling such systems can take very different shapes. They range from using tailored high level languages [7], [8], macro-based systems [9], or C++-lambda based like Kokkos [10], RAJA [11] or SYCL [12]. Also advanced, compiler-based strategies are becoming popular, maybe most notably in the SYCL programming ecosystem [12] or OpenCL [13] which both rely on the SPIR-V intermediate representation. In addition, modern compilers themselves help architecture portability with recent features such as multi-level transformations. These are proposed in LLVM with the MLIR [14] or SPIRAL [8], and offer the opportunity to run compiler passes at a coarser granularity than a few assembly instructions.

Relying on a portability layer removes the burden of platform portability from the library developers and allows them to focus exclusively on the development of sophisticated algorithms. This convenience comes at the price of a strong dependency on the portability layer, and moving to another programming model or portability layer is usually extremely difficult or even impossible. Furthermore, relying on a portability layer naturally implies that the performance of algorithms and applications is determined by the quality and hardware-specific optimization of the portability layer. This performance penalty may not always be insignificant, as portability layers usually have a wide user base, and dramatic changes to the interface, logic, or kernel design of the portability layer would likely result in the failure of some applications that rely on the portability layer [15], [16], [17]. Hence, performance portability layers should avoid modifying the design or hardware coverage, which can limit the opportunities to heavily optimize kernels for a new hardware architecture.

Performance Portability as a Library Design In this approach, the hardware-specific kernels are written separately for the different types of hardware targeted. Standards are a good example of this portability strategy. The standards define an API and a set of functionalities, the respective kernels are then realized in the hardware-tailored languages. Many standards have been successfully adopted by vendors such as for linear algebra the BLAS standard [18], [19], but also for distributed communications with the MPI standard [20]. Libraries that cannot entirely rely on a standard (or become one), can realize portability themselves by acknowledging the principles of controlling the interfaces, isolating dependencies, and thinking portable [21]. Several libraries are using this model effectively, like deal.II [1] or PETSc [22]. To use this model, a library must be designed with modularity and extensibility in mind. Only a library design that relies on the separation of concerns between the parallel algorithm and the different hardware backends can allow such a feature. The different backends need to be managed and interacted with thanks to a specific interface layer between algorithms and kernels. However, the price for the higher performance potential is high: the library developers have to synchronize several hardware backends, monitor and react to changes in compilers, tools, and build systems, and adopt new hardware backends and programming models. The effort of maintaining multiple hardware backends and keeping them synchronized usually results in a significant workload that can easily exceed the developers' resources. However, unlike the first approach, this supports hand-written optimized kernels for each hardware architecture. In addition, the abstraction and kernel development being focused on a single product allows for a more consumer-specific kernel design and performance optimization. In consequence, libraries following this path can apply much more aggressive hardware-specific optimization and often achieve higher performance. One reason is that the set of kernels is usually much smaller than what portability layers provide as the hardware-specific backends, because only the kernels required by the library's core algorithms are included. A second reason is that a library has more freedom to phase out support for a specific hardware architecture. This can usually be justified because the dependency on a library is generally much looser than the dependency on a portability layer, and applications "just" need to find a new library that provides the same functionality, while the much deeper dependency on a portability layer virtually prohibits moving to an alternative portability layer.

A Hybrid Approach for Platform Portability The two strategies for achieving platform portability presented are not necessarily exclusive, and the usage of a hybrid model can be more efficient. This is to some extent leveraged in some recent developments of the Ginkgo library (Section 4 and other recent developments) but can also be seen in other libraries such as PETSc [22] or Trilinos [2]. Another example of a hybrid approach is the C++ standard library which is rapidly becoming a performance portability vector itself. On the one hand are vendors such as NVIDIA and Intel implementing hardware-accelerated versions of key algorithms that are accessible through different namespaces for targeting accelerators [23], [24]. On the other hand is C++ increasingly including advanced performance portability interfaces such as `mdspan` [25] or `for_each` which work similarly to the SYCL, RAJA, or Kokkos libraries. One reason for adopting such a hybrid approach is that not all building blocks are as relevant, performance-critical, or as complex to optimize as others. For those kernels, relying on a performance portability layer allows reducing the code maintenance and testing complexity as well as focus on the more performance-critical aspects of the library.

3. Developing Gingko using Platform Portability as Central Design Paradigm

GINKGO is an open-source numerical linear algebra library focusing on sparse computations [5]. Aside from basic building blocks like sparse matrix-vector multiplication (SpMV) and sparse matrix addition (SpGeAM), it features iterative solvers including a variety of Krylov solvers (BiCG, BiCGSTAB, CG, CGS, FCG, GMRES, IDR) and sophisticated preconditioners (sparse approximate inverses, parallel ILU preconditioners, and threshold ILU preconditioners). GINKGO is developed by acknowledging platform portability as a central design principle. This becomes apparent in the visualization of the software architecture radically separating the library core from the hardware-specific kernels, see Figure 1. All classes, library logic, and generic algorithm skeletons are accumulated in the library "core" which, however, is useless without the driver kernels available in the distinct hardware backends. We note that the "reference" backend contains sequential CPU kernels used to validate the correctness of the algorithms and as a reference implementation for the unit tests realized using the googletest[26] framework. The "OpenMP" backend contains kernels for multicore architectures that are parallelized using OpenMP. The "CUDA" and "HIP" backends are heavily optimized GPU backends written in the hardware-native language: CUDA for NVIDIA GPUs, HIP for AMD GPUs [27]. Given the significant level of similarity between the "CUDA" and the "HIP" executor, a shared folder (omitted in the visualization for simplicity) contains kernels that are up to architecture-specific parameter configurations identical for the AMD and the NVIDIA GPUs, respectively [28]. Collecting those kernels in a folder included by the backends reduces code duplication and maintenance efforts. The latest addition is the "DPC++" backend tailored towards Intel GPUs but usable also on other architectures supporting DPC++ code [29]. In opposition to the other backends that are already in production mode, the "dpc++" backend is currently under heavy development as also the technical details about Intel's future discrete GPUs are still unknown. During library configuration, the user decides which backends to compile. In GINKGO, we have chosen to use for each target architecture the vendor's native language. While languages like OpenACC or OpenMP offload can be used to target GPUs from different vendors, features like sub-warp operations which are critical for the performance of the CUDA and HIP backends would not be usable and this would impact the obtained performance.

The "executor" allows the selection of the target architecture. The respective kernels are then chosen during execution via dynamic polymorphism. This is possible as the executor is a central class in GINKGO that provides all important primitives for allocating/deallocating memory on a device, transferring data to other supported devices, and basic intra-device communication (e.g., synchronization). The chosen executor is also the library feature that orchestrates the interplay between the core library and the backends. An executor always has a `master` executor which is a CPU-side executor capable of allocating/deallocating space in the main memory. This concept is convenient when considering devices such as CUDA or HIP accelerators, which feature their own separate memory space. Although implementing a GINKGO executor that leverages features such as unified virtual memory (UVM) is possible via the interface, to attain higher performance, we manage all copies by direct calls to the underlying APIs [30]. This removes the burden of having to manage the memory from the GINKGO users but allows us to instruct explicit memory transfers in the places we want them to happen.

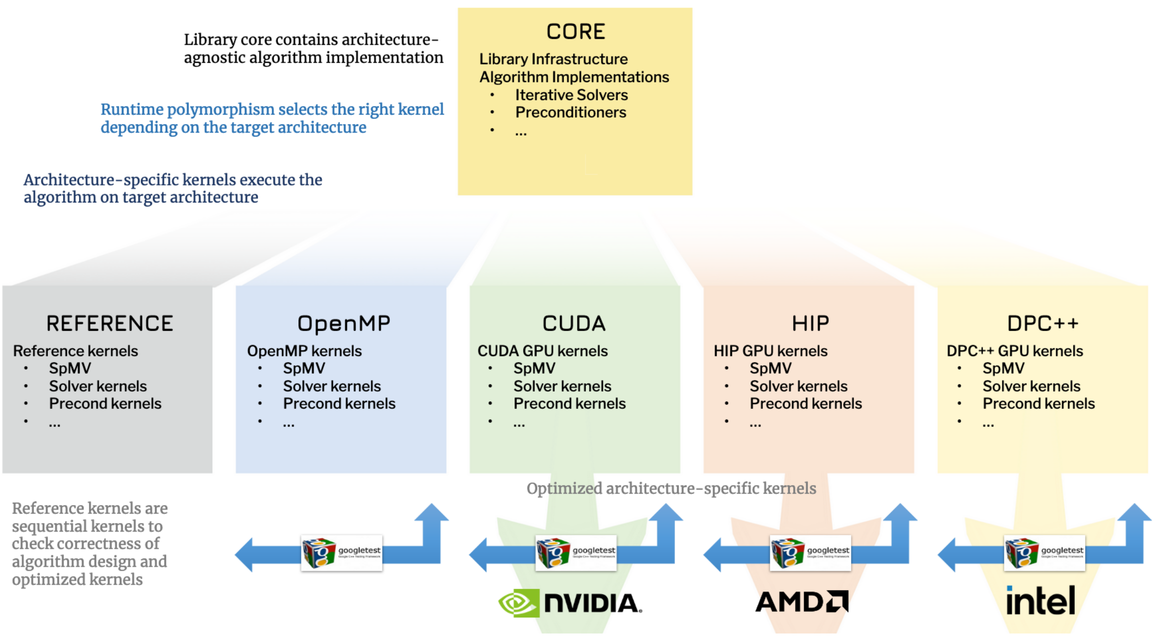


Figure 1: Overview of the GINKGO library design.

```
 1 #include <iostream>
 2 #include <ginkgo/ginkgo.hpp>
 3
 4 int main()
 5 {
 6   // Instantiate a CUDA executor
 7   auto omp = gko::OmpExecutor::create();
 8   auto ex = gko::CudaExecutor::create(0, omp);
 9   // Read data
10   auto A = gko::read<gko::matrix::Csr<>>(std::cin, ex);
11   auto b = gko::read<gko::matrix::Dense<>>(std::cin, ex);
12   auto x = gko::read<gko::matrix::Dense<>>(std::cin, ex);
13   // Create the solver factory
14   auto factory =
15     gko::solver::Cg<>::build()
16       .with_preconditioner(
17         gko::preconditioner::Jacobi<>::build().on(ex))
18       .with_criteria(
19         gko::stop::Iteration::build().with_max_iters(20u)
20           .on(ex),
21         gko::stop::ResidualNormReduction<>::build()
22           .with_reduction_factor(1e-15).on(ex))
23       .on(ex);
24   // Generate the solver and solve the system
25   factory->generate(give(A))->apply(lend(b), lend(x));
26   // Write result
27   write(std::cout, lend(x));
28 }
```

Listing 1: A minimal usage example of GINKGO to solve a linear system. The system is solved by a CG method enhanced with a block-Jacobi preconditioner. Everything is ran on a CUDA-capable device thanks to the use of a CudaExecutor.

Listing 1 shows a simple GINKGO use-case. In this example, we read the system parameters from the standard input in lines 10-12 before creating a CG solver factory preconditioned by a Block Jacobi preconditioner in lines 14-23. Using the two criteria, the iterative process will stop either after 20 iterations or when the residual norm was reduced by 15 orders of magnitude. In line 25, a concrete solver instance is generated using these factory parameters for the specific matrix *A* before solving the system thanks to the `apply()` method. Finally, the solution is printed. Lines 7-8 show how GINKGO can be set to run on specific hardware. In this example, a `CudaExecutor` was chosen. Replacing all instances of the variable `ex` in the code by `omp` would make the code run on the CPU using OpenMP instead.

4. Adopting AMD GPUs

Even though GINKGO is developed with platform portability as a central design principle, it initially only featured executors for sequential execution ("Reference"), OpenMP-parallelized multicore execution ("OpenMP"), and CUDA-based NVIDIA GPU execution ("CUDA"). Therefore, the addition of a "HIP" executor enabling the execution of code on AMD GPUs is the proof-of-concept test for the executor-based platform portability model [28]. Due to GINKGO's software design for portability, adding a new architecture always follows the same workflow: 1) the core library interfaces that manage the backends such as the executor needs to be manually extended as well as CMake support, and dummy kernels need to be added to obtain a first compiling backend; 2) one by one, the stub kernels can be replaced by concrete kernel implementations – in this step, we can often rely on automatic vendor-provided porting tools; 3) once the kernels have been ported, they can be tuned, specialized or rewritten if needed to obtain better performance.

CMake integration.

When adding an executor based on a new programming language, a first step is the integration of the compiler in the CMake configuration and compilation process. In the adoption of the HIP ecosystem, we make heavy use of the CMake build system generator. One way of integrating HIP into the build process would be to use the `hipcc` compiler for the entirety of the project. We chose a less intrusive approach, by relying on the HIP packages which thanks to modern CMake features allows us to integrate them via CMake's `find_package()` command. AMD provides the following important packages for GINKGO:

- `hip-config.cmake` etc., included via `find_package(hip)`;
- `hipblas-config.cmake`, included into a project via `find_package(hipblas)` (resp. for `hipsparse`);
- `FindHIP.cmake`, included into a project via `find_package(HIP)`.

The last file provides the main macros for creating HIP projects, and defines a new CMake HIP language to compile the HIP files. The most important macros are `hip_add_executable()` and `hip_add_library()`, similar to the ones declared in `FindCUDA.cmake` file which provided equivalent CUDA functionality until CUDA became a natively-supported CMake language in CMake version 3.8.

HIP allows compiling code either for CUDA support or for ROCm support. Depending on whether a physical GPU is detected or the environment variable HIP PLATFORM is set to either nvcc (for ROCm<=4.0) or NVIDIA (for ROCm>4.0), the code is compiled for CUDA. If HIP PLATFORM is set to either clang (for ROCm<=4.0) or AMD (for ROCm>4.0), the code is compiled for ROCm support. During compilation, the HIP header libraries bind to either the CUDA libraries or the ROCm libraries.

Despite providing this convenient feature of supporting both AMD and NVIDIA architectures via one language, currently, several pitfalls exist that require applying customized workarounds. The `hip-config.cmake` depends on all the ROCm subcomponents, which implies that compiling the HIP executor for NVIDIA architectures creates a complicated dependency structure. Also, while the process of locating the ROCm libraries is automated by the main FindHIP.cmake file, the user has to manually provide the CUDA locations when compiling HIP for NVIDIA architectures.

Another example of the difficult interoperability between the HIP language and NVIDIA hardware is the example shown in Listing 2. This example works for recent versions of ROCm with some backward compatibility to ROCm <= 4.0, but older versions fail. One issue is that the platform names changed in version 4.1, and two sets of platforms need to be matched for supporting older ROCm versions.

In addition, the `hip-config.cmake` file hard-codes several AMD device-specific flags which are non-standard for several HIP targets (i.e., the file sometimes supposes the full project is compiled with `hipcc`) which need to be manually removed for general compilation.

Finally, `FindHIP.cmake` struggles with compiling shared libraries in complex settings but throws an exception on the AMD backend[2]. In this case, a workaround is to explicitly set the property `LINKER_LANGUAGE` of the library to `HIP`.

[2]https://github.com/ROCm-Developer-Tools/HIP/issues/1029

We note that AMD and Kitware plan to implement native support for HIP in CMake, such integration would likely fix most of the issues we outlined. Overall, we found that the code skeleton in Listing 2 is the only one that allows us to successfully integrate HIP code as a subcomponent into a complex project featuring also other subcomponents which rely on CUDA, OpenMP, and other libraries for most versions of ROCm including newer ones.

```
# set ROCM_PATH, HIP_PATH, HIPBLAS_PATH,
# HIPSPARSE_PATH, etc. the path of the
# respective installations if not set,
if(HIP_PLATFORM MATCHES "nvcc|nvidia")
  # ensure the CUDA library can be found
  # in the default path or CUDA_PATH is
  # set (requirement of hipcc),
  set(GINKGO_HIP_NVCC_ARCH_FLAGS ...)
elseif(GKO_HIP_AMDGPU AND HIP_PLATFORM MATCHES "hcc|
    amd")
  # On AMD backend, set --amdgpu-target
  # options when selecting specific GPU targets
  foreach(target ${GKO_HIP_AMDGPU})
    list(APPEND GINKGO_HIP_AMD_ARCH_FLAGS --amdgpu-
     target=${GKO_HIP_AMDGPU})
  endforeach()
endif()
# set CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH
find_package(HIP REQUIRED)
find_package(hipblas REQUIRED)
find_package(hipsparse REQUIRED)

set(GKO_HIP_SOURCES path/to/source/files.hip.cpp)
set_source_files_properties(${GKO_HIP_SOURCES}
  PROPERTIES HIP_SOURCE_PROPERTY_FORMAT TRUE)
# All options must be given to HIP_ADD_LIBRARY
hip_add_library(ginkgo_hip ${GKO_HIP_SOURCES}
  CLANG_OPTIONS ${GINKGO_HIP_AMD_ARCH_FLAGS}
  NVCC_OPTIONS ${GINKGO_HIP_NVCC_ARCH_FLAGS})

if(HIP_PLATFORM MATCHES "hcc|clang")
  find_package(hip REQUIRED)
  target_link_libraries(ginkgo_hip hip::amdhip64)
elseif(HIP_PLATFORM MATCHES "nvcc|nvidia")
  find_package(CUDA 9.0 REQUIRED)
  target_link_libraries(ginkgo_hip
    ${CUDA_LIBRARIES})
endif()

target_link_libraries(ginkgo_hip roc::hipblas
  roc::hipsparse)
target_include_directories(${HIP_INCLUDE_DIRS}
  ${HIPBLAS_INCLUDE_DIRS}
  ${HIPSPARSE_INCLUDE_DIRS})
```

Listing 2: Schematic example for integrating a HIP module into an existing CMake project.

Porting CUDA code to HIP via the Cuda2Hip script. For easy conversion of CUDA code to the HIP language, we use a script based on the hipify-perl script provided by AMD with several modifications to meet our specific needs. First, the script generates the target filename including the path in the "hip" directory. Then AMD's hipify-perl script is invoked to translate the CUDA kernels to the HIP language, including the transformation of NVIDIA's proprietary library functions to AMD's library functions and the kernels launch syntax. Next, the script changes all CUDA-related header, namespace, type- and function names to the corresponding HIP-related names. By default, the script hipify-perl fails to handle namespace definitions. For example, the hipify-perl script changes

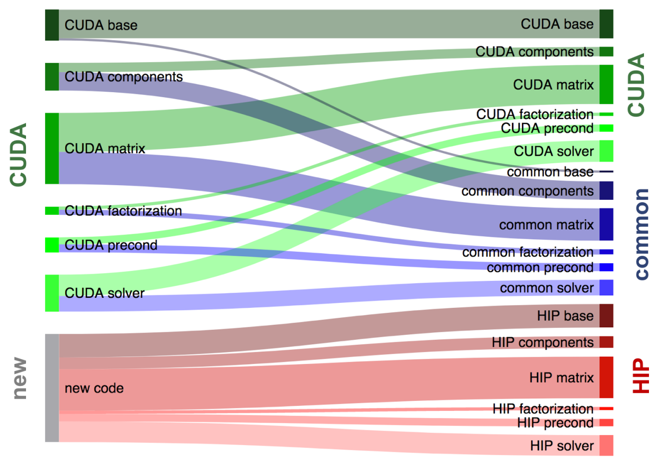


Figure 2: Reorganization of the Ginkgo library to provide a HIP backend for AMD GPUs.

`namespace::kernel<<<...>>> (...)`
to
`namespace::hipLaunchKernelGGL(kernel, ...)`
while the correct output would be
`hipLaunchKernelGGL(namespace::kernel, ...)`. In the Cuda2Hip script, we correct the namespaces generated by the hipify-perl script after having applied the hipify-perl script to all kernels.

Avoiding code duplication. When adding a HIP backend, we notice a high level of similarity in both kernel design and syntax between the CUDA kernels designed for NVIDIA GPUs and the HIP kernels for AMD GPUs. Thus, the straightforward addition of a HIP backend would introduce a significant level of code duplication. For this purpose, we create the "common" folder containing all kernels and device functions that are identical or the CUDA and the HIP executor except for architecture-specific kernel configuration parameters (such as warp size or `launch_bounds`). These configuration parameters are not set in the kernel file contained in the "common" folder but in the files located in "cuda" and "hip" that are interfacing these kernels. The CUDA and HIP backends then include the files in the "common" similar to header files and configure the parameters. Obviously, for separating kernel skeletons shared by the CUDA and HIP executors, the Cuda2Hip script discussed previously needs to be extended by identifying hardware-specific kernel execution parameters, replacing these with variables, and generating only the kernel call functions in the CUDA and HIP backends while placing the parameterized kernel skeletons in the common folder that are then included by the executors. Even though this reorganization requires the derivation of sophisticated scripts, it pays off as we can avoid code duplication while still configuring the parameters for optimal kernel performance on the distinct hardware backends [28]. The effect on Ginkgo's code stack is visualized in Section 4 where the left-hand side represents the code statistics (lines of code) before the addition of the HIP executor, and the right-hand side the reorganized code with a significant portion of the kernel code ending up in a "common" folder shared by the CUDA executor and the HIP executor.

Cooperative groups. CUDA 9 introduced cooperative groups for flexible thread programming. Cooperative groups provide an interface to handle thread block and warp groups and apply the shuffle operations that are used heavily in Ginkgo for optimizing sparse linear algebra kernels. HIP [31] only supports block and grid groups with `thread_rank()`, `size()` and `sync()`, but no subwarp-wide group operations like shuffles and vote operations. As such, the cooperative group interface is a convenient way to describe parallel operations and ease kernel writing.

For enabling platform portability where applicable, the same ease of kernel writing a small codebase, and preserving the performance of the optimized CUDA kernels, we implement cooperative group functionality for the HIP ecosystem. Our implementation supports the calculation of size/rank and shuffle/vote operations inside subwarp groups. We acknowledge that our cooperative group implementation may not support all features of CUDA's cooperative group concept, but all functionality we use in Ginkgo.

The cross-platform cooperative group functionality we implement with shuffle and vote operations covers CUDA's native implementation. HIP only interfaces CUDA's warp operation without `_sync` suffix (which refers to deprecated functions), so we use CUDA's native warp operations to avoid compiler warning and complications on NVIDIA GPUs with compute capability 7.x or higher. We always use subwarps with contiguous threads, so we can use the block index to identify the threads' subwarp id and its index inside the subwarp. We define

$$
\begin{aligned}
\texttt{Size} &= \text{Given subwarp size}\\
\texttt{Rank} &= \texttt{tid \% Size}\\
\texttt{LaneOffset} &= \lfloor \texttt{tid \% warpsize / Size} \rfloor \times \texttt{Size}\\
\texttt{Mask} &= \sim 0 \texttt{ >> (warpsize - Size) << LaneOffset}
\end{aligned}
$$

where `tid` is local thread id in a thread block such that `Rank` gives the local id of this subwarp, and ~ 0 is a bitmask of 32/64 bits, same bits as `lane_mask_type`, filled with 1 bits according to CUDA/AMD architectures, respectively. Using this definition, we can realize the cooperative group interface, for example for the `shfl_xor`, `ballot`, `any`, and `all` functionality:

$$
\begin{aligned}
\texttt{subwarp.shfl\_xor(data, bitmask)} &= \texttt{\_\_shfl\_xor(data, bitmask, Size)}\\
\texttt{subwarp.ballot(pred)} &= \texttt{(\_\_ballot(pred) \& Mask) >> LaneOffset}\\
\texttt{subwarp.any(pred)} &= \texttt{(\_\_ballot(pred) \& Mask) != 0}\\
\texttt{subwarp.all(pred)} &= \texttt{(\_\_ballot(pred) \& Mask) == Mask}
\end{aligned}
$$

Note that we use the `ballot` operation to implement `any` and `all` operations. The original warp `ballot` returns the answer for the entire warp, so we need to shift and mask the bits to access the subwarp results. The `ballot` operation is often used in conjunction with bit operations like the population count (*popcount*), which are provided by C-style type-annotated intrinsics `__popc[ll]` in CUDA and HIP. To avoid any issues with the 64bit-wide lane masks on AMD GPUs, we provide a single function `popcnt` with overloads for 32 and 64 bit integers as well as an architecture-agnostic `lane_mask_type` that provides the correct (unsigned) integer type to represent a (sub)warp lane mask. Experimental results have shown that GINKGO's custom platform portable cooperative group implementation is highly competitive to the vendor-provided functionality, see Figure 3 [28].

5. Adopting Intel GPUs

Intel is the vendor who is expected to provide one of the three US Exascale machines, named Aurora, and hosted at Argonne National Laboratory which will feature Intel discrete GPUs. To program these new architectures, Intel is focusing on the SYCL programming framework [12], which was extended with several programming features allowing both better kernel performance and usability. The new additions in Intel's DPC++ language comprise, but are not limited to, features such as Unified Shared Memory (USM, which allows direct memory transfer control) as well as the concept of subgroups, a CUDA-warp equivalent. Many of the Intel extensions part of DPC++ were released as part of the SYCL 2020 specification.[3] The Intel extensions are packaged into the DPC++ compiler [29], which is part of the Intel oneAPI framework[4]. GINKGO started the portability effort to the new Intel oneAPI platform to target future Intel hardware. Intel also supports OpenMP offloading for programming its GPUs. Since the oneAPI framework seems to be centered around DPC++, and this programming model seems more adapted for high-performance programming (subgroups, USM, and other features), we have chosen this new language to target Intel GPUs.

CMake integration. At the time of writing, the only way the authors found to integrate DPC++ into our framework is to use the DPC++ compiler for the whole compilation process of our library (e.g., with the CMake option `CMAKE_CXX_COMPILER=dpcpp`). This usage model means that the CMake integration effort is minimal, but on the other hand, there is no proper CMake isolation of the DPC++ submodule, which prohibits compiling any other GINKGO submodule at the same time, except for `Reference`.

Adding a DPC++ Executor to GINKGO. Like SYCL, DPC++ is a C++ single-source heterogeneous programming framework targeting the full range of device programming APIs, such as OpenCL, CUDA, or Intel oneAPI Level Zero. A core concept of these frameworks is the use of command queues which are objects able to schedule kernels on devices with specific execution contexts. By default, SYCL and DPC++ execute kernels asynchronously, and only the destruction of the queue object synchronizes with the device. Memory copies are all implicit thanks to SYCL providing a specific buffer type for registering data, together with the access mode specification of these data inside kernels thus SYCL is a task-based programming language. Another key aspect of DPC++ is that (like SYCL) it aims at being compilable on most existing devices, ranging from NVIDIA to AMD GPUs, Intel FPGAs, GPUs, and general-purpose processors.

This usage model of standard SYCL/DPC++ does not fit well with the GINKGO framework, as GINKGO handles devices via the `Executor` model (which can be either a CPU or an accelerator). On the other hand, a new DPC++ executor should be able to target any device type. For this purpose, we have decided to add an optional device selection to our DPC++ executor constructor which allows specifying whether the user wants to target a GPU or a CPU (or any other device type). In addition, we decided that a DPC++ executor always comes with a CPU executor to manage the host-side data (as it is needed for GPUs). We acknowledge that this can result in redundant data in case the DPC++ executor targets a CPU device.

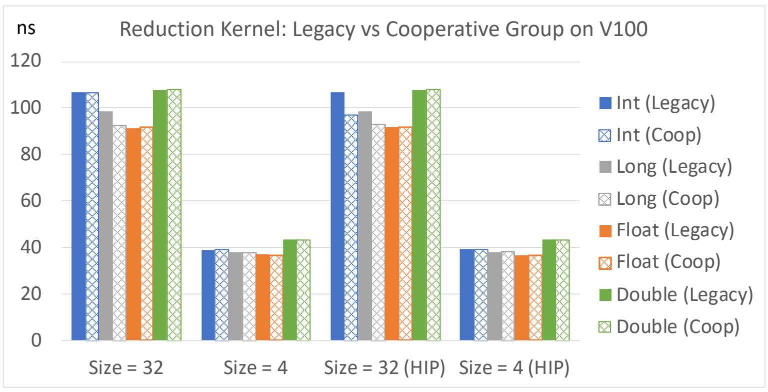
Another aspect important to consider when creating a DPC++ executor for GINKGO is the "subgroup" concept DPC++ introduces to represent a SIMD lane on CPUs or a warp/wavefront on GPUs. Depending on the architecture, the `subgroup_size` can differ dramatically, from a size of 1 on some simple devices to 4 or 8 for powerful CPUs, to 16, 32, or 64 on some GPUs. While this concept is not completely new – GINKGO already uses warp size 32 for NVIDIA GPUs and wavefront sizes 64 for AMD GPUs – it can heavily affect performance or even cause errors if selecting the `subgroup_size` inappropriate for a given hardware architecture. To resolve this, in GINKGO, we precompile kernels for all possible `subgroup_sizes` (from 1 to 64, in powers of two increments), and dynamically select at runtime the kernel version which fits best to the selected DPC++ device.

Since GINKGO executors need to provide synchronization features, we have decided that one executor is represented by one DPC++/SYCL queue object which exists during the whole lifetime of the executor. This object is automatically managed thanks to a `unique_ptr`. The DPC++ queue primitive `wait_and_throw()` can be used for explicit synchronization. A specific aspect of a DPC++ queue is that, by default, no execution order is guaranteed for the objects submitted to it, since it operates asynchronously. This means that when executing three tasks on the same queue, 1) copy: host $\rightarrow$ GPU, 2) a GPU kernel, 3) copy: GPU $\rightarrow$ host, there is no guarantee that the three operations will be executed in this order. To ensure the right execution order, DPC++ introduces the queue property `in_order` which can be used. Another solution to this problem is given by manually synchronizing after every DPC++ queue operation.

Finally, to allow explicit memory management, we rely on the new DPC++ concept of Unified Shared Memory, which can be used via the functions for the explicit memory (de)allocation (`sycl::malloc_device()`, `sycl::free()`). When using memory allocated via these functions, the memory copies can be controlled manually via the DPC++ queue itself using the function `queue.memcpy()`. The limitation of this concept is that it can not handle settings where another non-DPC++ GINKGO executor (or device) is used in an application (for example, a HIP

[3]The release happened on March 4, 2021, see `https://www.khronos.org/registry/SYCL`

[4]`https://www.oneapi.com/`



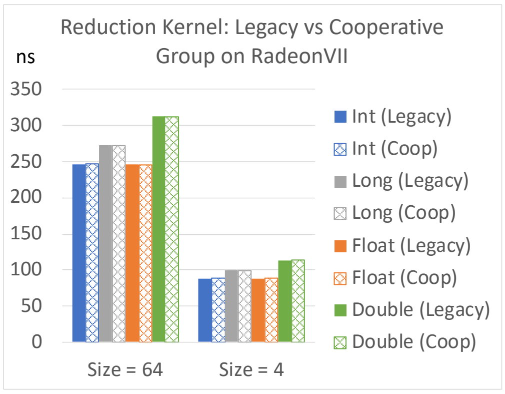


Figure 3: GINKGO's cooperative groups vs. legacy functions for different data types on V100 (left) and RadeonVII (right) [28].

enhanced device). In this case, DPC++ can not directly use the `queue.memcpy()` function to copy data to the other executor. To resolve the challenge of copying data from the DPC++ executor to another (non-CPU) executor, we use the workaround of creating a temporary copy on the master executors running on the CPU and controlling the execution of the DPC++ executor and the other device executor, respectively.

Kernel programming

```
 1 #include <CL/sycl.hpp>
 2
 3 template<int subgroup_size>
 4 [[ intel::reqd_sub_group_size(subgroup_size) ]]
 5 void reduce(float *a, sycl::nd_item<3> i) {
 6   auto subgroup = i.get_sub_group();
 7   auto local_data = a[i.get_local_id(2)];
 8   #pragma unroll
 9   for (int bitmask = 1; bitmask < subgroup_size;
10      bitmask <<= 1) {
11       const auto remote_data =
12         subgroup.shuffle_xor(local_data, bitmask);
13       local_data = local_data + remote_data;
14     }
15     a[i.get_local_id(2)] = local_data;
16 }
17
18 int main() {
19   sycl::queue q{}; // get default device's queue
20   float *dev_A;
21   float A[32]; // missing: populate data
22   dev_A = sycl::malloc_device<float>(32, q);
23   q.memcpy(dev_A, A, 32 * sizeof(float)).wait();
24   q.submit([&](sycl::handler &cgh) {
25     cgh.parallel_for(
26       sycl::nd_range<3>(sycl::range<3>(1, 1, 32),
27                         sycl::range<3>(1, 1, 32)),
28                         [=](sycl::nd_item<3> i) {
29                           reduce<8>(dev_A, i);
30                         });
31     }).wait();
32   q.memcpy(A, dev_A, 32 * sizeof(float)).wait();
33 }
```

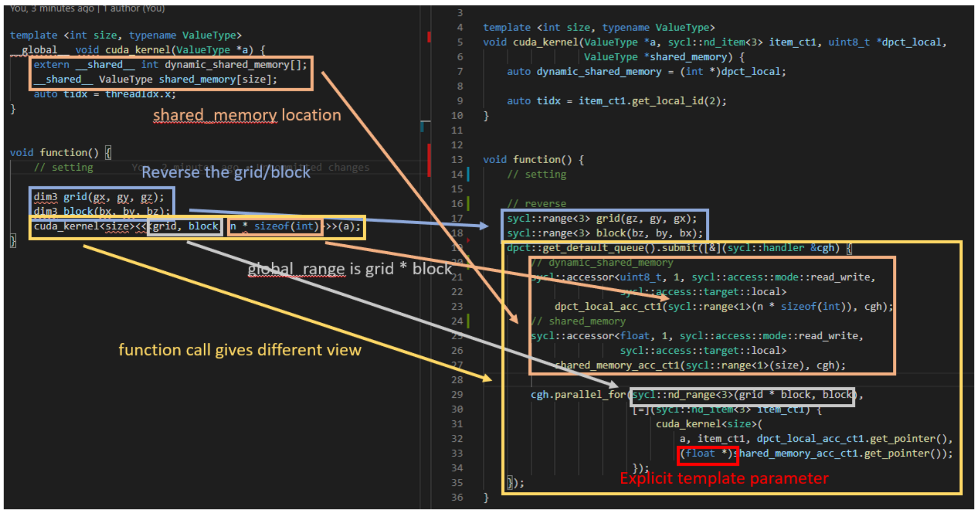
Listing 3: Minimal example of a DPC++ kernel call featuring a subgroup based reduction.

Listing 3 showcases a minimal DPC++ example for setting up and running a subgroup-based reduction. We only omit in this code the generation of the data the kernel operates on. The code is composed of two parts, lines 3-16 are device-side functions that handle the reduction on a subgroup. The template parameter reflects the subgroup size (line 3) to allow the compiler to unroll the loop in line 9. The second part in lines 18-33 is the main function containing data management as well as the kernel invocation. In some detail, we first use a default queue, which will access the default device (line 19) and can be used for data management as well as submitting tasks. Lines 22-23 reflect the DPC++ Unified Shared Memory usage, where initially the device-side memory is allocated before copying this data from the host to the device. Line 32 similarly copies back the data to the host before kernel completion. Lines 24-31 show the creation and submission of a task that operates in a "parallel-for" fashion (line 25). The task initially selects its `nd_range` parameter with both the global (line 26) and local, or workgroup level, (line 27) dimensions. Since the global and workgroup dimensions are identical, this implies that only one workgroup is instantiated which then processes the whole kernel range. The kernel executed by this "parallel-for" is the lambda function defined in lines 28-30 which only calls the reduction kernel on the device data at each respective "item" of the parallel for. Note that in this program, we always call the function `wait()` of the queue to ensure proper ordering of the operations (we could also use the DPC++ property "in order" of the queue). Lines 3-16 present the reduction kernel. On line 4, the keyword `intel::reqd_sub_group_size` is used to notify that this function targets a subgroup with a size depending on the template parameter (in this example the subgroup size is 8, see line 29). In summary, the code will create one workgroup containing four subgroups each composed of 8 elements. Line 6 then accesses this subgroup and stores the representation object. All subgroups can then execute a standard reduction in lines 9-15, which is based on `subgroup.shuffle_xor` (line 12).

Custom porting script based on DPCT

Similar to AMD, Intel provides a tool that converts CUDA code to the DPC++ language. This "DPC++ Compatibility Tool" (DPCT[5]) intends to make porting existing CUDA kernels to the oneAPI ecosystem a simple and convenient step. However, at the time of writing, the oneAPI ecosystem is still in its early

[5]https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top.html



shared_memory location
Reverse the grid/block
global_range is grid * block
function call gives different view
Explicit template parameter

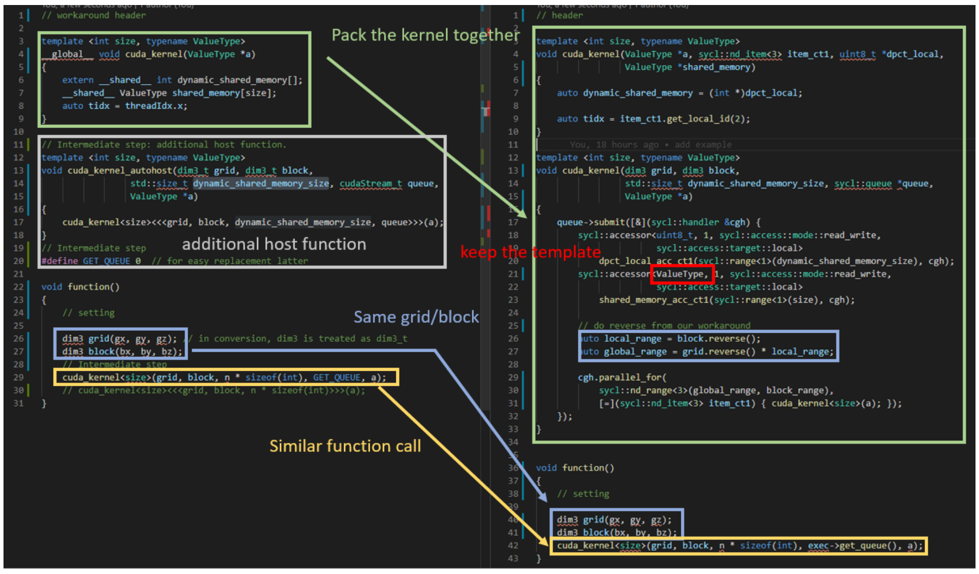
```
template <int size, typename ValueType>
__global__ void cuda_kernel(ValueType *a) {
    extern __shared__ int dynamic_shared_memory[];
    __shared__ ValueType shared_memory[size];
    auto tidx = threadIdx.x;
}


void function() {
    // setting

    dim3 grid(gx, gy, gz);
    dim3 block(bx, by, bz);
    cuda_kernel<size><<<grid, block, n * sizeof(int)>>>(a);
}
```

```
 3
 4  template <int size, typename ValueType>
 5  void cuda_kernel(ValueType *a, sycl::nd_item<3> item_ct1, uint8_t *dpct_local,
 6                   ValueType *shared_memory) {
 7      auto dynamic_shared_memory = (int *)dpct_local;
 8
 9      auto tidx = item_ct1.get_local_id(2);
10  }
11
12
13  void function() {
14      // setting
15
16      // reverse
17      sycl::range<3> grid(gz, gy, gx);
18      sycl::range<3> block(bz, by, bx);
19      dpct::get_default_queue().submit([&](sycl::handler &cgh) {
20          // dynamic_shared_memory
21          sycl::accessor<uint8_t, 1, sycl::access::mode::read_write,
22                         sycl::access::target::local>
23              dpct_local_acc_ct1(sycl::range<1>(n * sizeof(int)), cgh);
24          // shared_memory
25          sycl::accessor<float, 1, sycl::access::mode::read_write,
26                         sycl::access::target::local>
27              shared_memory_acc_ct1(sycl::range<1>(size), cgh);
28
29          cgh.parallel_for(sycl::nd_range<3>(grid * block, block),
30                           [=](sycl::nd_item<3> item_ct1) {
31                               cuda_kernel<size>(
32                                   a, item_ct1, dpct_local_acc_ct1.get_pointer(),
33                                   (float *)shared_memory_acc_ct1.get_pointer());
34                           });
35      });
36  }
```

Figure 4: Example of a CUDA code (left) and the DPCT-converted code (right) which uses static and dynamic shared memory.



```
// workaround header

template <int size, typename ValueType>
__global__ void cuda_kernel(ValueType *a)
{
    extern __shared__ int dynamic_shared_memory[];
    __shared__ ValueType shared_memory[size];
    auto tidx = threadIdx.x;
}

// Intermediate step: additional host function.
template <int size, typename ValueType>
void cuda_kernel_autohost(dim3_t grid, dim3_t block,
                          std::size_t dynamic_shared_memory_size, cudaStream_t queue,
                          ValueType *a)
{
    cuda_kernel<size><<<grid, block, dynamic_shared_memory_size, queue>>>(a);
}
// Intermediate step
#define GET_QUEUE 0  // for easy replacement latter

void function()
{
    // setting

    dim3 grid(gx, gy, gz); // in conversion, dim3 is treated as dim3_t
    dim3 block(bx, by, bz);
    // Intermediate step
    cuda_kernel<size>(grid, block, n * sizeof(int), GET_QUEUE, a);
    // cuda_kernel<size><<<grid, block, n * sizeof(int)>>>(a);
}
```

```
// header

template <int size, typename ValueType>
void cuda_kernel(ValueType *a, sycl::nd_item<3> item_ct1, uint8_t *dpct_local,
                 ValueType *shared_memory)
{
    auto dynamic_shared_memory = (int *)dpct_local;

    auto tidx = item_ct1.get_local_id(2);
}

template <int size, typename ValueType>
void cuda_kernel(dim3 grid, dim3 block,
                 std::size_t dynamic_shared_memory_size, sycl::queue *queue,
                 ValueType *a)
{
    queue->submit([&](sycl::handler &cgh) {
        sycl::accessor<uint8_t, 1, sycl::access::mode::read_write,
                       sycl::access::target::local>
            dpct_local_acc_ct1(sycl::range<1>(dynamic_shared_memory_size), cgh);
        sycl::accessor<ValueType, 1, sycl::access::mode::read_write,
                       sycl::access::target::local>
            shared_memory_acc_ct1(sycl::range<1>(size), cgh);

        // do reverse from our workaround
        auto local_range = block.reverse();
        auto global_range = grid.reverse() * local_range;

        cgh.parallel_for(
            sycl::nd_range<3>(global_range, block_range),
            [=](sycl::nd_item<3> item_ct1) { cuda_kernel<size>(a); });
    });
}


void function()
{
    // setting

    dim3 grid(gx, gy, gz);
    dim3 block(bx, by, bz);
    cuda_kernel<size>(grid, block, n * sizeof(int), exec->get_queue(), a);
}
```

Figure 5: Conversion of the same CUDA code (left) to DPC++ (right) as in Figure 4 via the improved DPCT containing our modifications. The CUDA code is also adapted by the script (before conversion only) with the addition of a new host function to force DPCT to create a more consistent output.

stages, and the conversion tool has several flaws that require manual fixes. In addition, the tool in some cases requires manual involvement of the programmer as the tool cannot be made general to work for all cases (in these cases a warning is printed by DPCT). In the context of developing a DPC++ backend for GINKGO, we enhanced DPCT with several additions and error fixes – some of them customized to the GINKGO library design, some of them useful for any code conversion. Even though DPCT is at the time of writing still under development, we want to list some of these changes and modifications we introduced to customize this tool to our purpose.[6] We show a concrete example for converting a CUDA code using dynamic and shared memory to DPC++ via the DPCT in Figure 4, and an intermediate version which is improved thanks to some techniques used by our script in Figure 5.

We observe in Figure 4 that, by default, DPC++ code has a very different structure and aspect than the CUDA equivalent, and also creates some subtle errors. One error concerns the static shared memory type which was based on a template parameter in the left (CUDA) code but becomes evaluated to the underlying data type (float) in this example on the right (DPC++) code (lines 29 and 33). In addition, there are inconsistencies compared to the CUDA code that are confusing at first glance. One such issue concerns the grid and block representation that is used in DPC++ (lines 17-18): the dimension which always moves in DPC++ is the right-most one, or "z" index,

[6]Our porting script can be accessed online `https://github.com/ginkgo-project/ginkgo/blob/develop/dev_tools/oneapi/convert_source.sh`.

whereas it is the left-most one in CUDA, or “x” index. Similarly, DPC++ uses a different concept of global and local range (line 29), where the global range is grid×block. In addition, the DPC++ kernel code is partly embedded in the main host function (the lambda function in the right part, lines 30-34), whereas there is a strict separation in CUDA.

The modifications we apply to DPCT as a workaround for these issues are shown in Figure 5. The main difference is that instead of calling DPCT on the original CUDA file, we generate a temporary CUDA file that uses a properly templated extra host function to invoke the kernel itself (lines 11-18 on the left side). In addition, we also create a placeholder `GET_QUEUE` (line 20, left) which we pass as `cudaStream_t` in the new host function (line 29, left). These two changes allow DPCT to generate better and more structured DPC++ code on the right side, where the main host function (lines-36-43, right) looks indeed similar to the original CUDA code. The `GET_QUEUE` placeholder can be replaced with our executor's queue accessor by our script (line 42, right), and DPCT properly converts the new `cudaStream_t` parameter in the new host function to a `sycl::queue` object. Note that we now need to reverse the `dim3` grid and block descriptions before using them (lines 26-27, right), which we realize via our custom `dim3` interface.

Overall, we advertise the following improvements that our custom conversion script provides compared to the plain DPCT tool:

- As previously detailed, we generate an intermediate CUDA host function which invokes the kernel with the correct configurations to keep a code structure that is more similar to CUDA (see Figures 4 and 5);
- To generate a code which is similar to CUDA, we introduce a DPC++ kernel interface layer, that abstracts the differences to a CUDA or HIP kernel such as:
  - We create a DPC++ `dim3` type to keep the CUDA kernel configuration and kernel call syntax consistent across the distinct executors;
  - We implemented our own cooperative group environment as this functionality is currently not directly supported in DPC++;
  - As DPC++ kernels need to access the kernel call configuration, we forward this information through the interface;
- We automatically comment out in the original CUDA code any `sync()` functions as these would cause problems in the DPC++ conversion, which are then correctly replaced by our script;
- We disable some of the template instantiations (especially calling a macro which takes as parameter another macro and other arguments) as these would cause DPCT errors.
- As `static_cast<Type>(kernel-index)` in templated kernels fails to be converted correctly, we fix these expressions;
- We automatically convert the GINKGO CUDA-specific namespaces, variable names, types, . . . to equivalent names containing DPC++ (e.g., in Figure 5 there are still several mentions of CUDA remaining);
- We also automatically move the generated DPC++ kernel file into the correct GINKGO location.

6. Performance Survey

Name	Description	Prog. Lang.	Release	Theo. BW (GB/s)
NVIDIA V100	Discrete HPC	CUDA	2017	920
NVIDIA A100 (40 GB)	Discrete HPC	CUDA	2020	1555
AMD RadeonVI	Discrete consumer	HIP	2019	1024
AMD MI100	Discrete HPC	HIP	2020	1228.8
Intel Gen. 9	Integrated	DPC++	2015	41.6

Table 1: List of the GPU architectures which we consider in the performance evaluation. The last column reports the theoretical bandwidth.

GINKGO aspires to not only provide platform portability, but also a satisfying level of performance portability. A good indicator to assess whether this goal is achieved is to quantify the performance GINKGO achieves on different hardware architecture relative to the hardware-specific performance bounds. We acknowledge that the Reference executor is designed to check the correctness of the algorithms and provide a reference solution for the unit tests and that GINKGO's primary focus is on high-performance accelerators. Thus, we limit the performance analysis to the CUDA executor running on high-end NVIDIA GPUs, the HIP executor running on high-end AMD GPUs, and the DPC++ executor running on Intel integrated GPUs. It is important to relate GINKGO's performance to the hardware-specific performance bounds. Thus, before providing performance results for GINKGO's routines, we present in Figure 6, Figure 7, and Figure 8 the results of the mixbench [32] and BabelStream [33] open source benchmarks. Table 1 lists the GPU hardware architectures we use in the performance evaluation along with a short architecture description. We note that not all architectures are high-end server-line GPUs. The AMD RadeonVII is a consumer-line GPU, the Intel Gen. 9 GPU is an embedded GPU, however already supporting the DPC++ execution model. In the performance evaluation, we use CUDA v. 11.0 for the NVIDIA GPUs, ROCm v. 3.8 for the AMD GPUs, and DPC++ v. 2021.1-beta10 for the Intel GPUs. We report the performance of Ginkgo using double precision arithmetic in all functionality.[7]

The bandwidth analysis for the NVIDIA V100 ad NVIDIA A100 GPUs reveals a significant peak bandwidth improvement from the NVIDIA V100 GPU (peak bandwidth 860 GB/s) to the NVIDIA A100 GPU (peak bandwidth 1,400 GB/s), see the top row in Figure 6. At the same time, the bandwidth for small and moderate-sized data reads is higher on the V100 GPU. For the arithmetic performance, we note that the benchmark does not leverage the tensor cores on the V100 or the A100 GPU.

[7]GINKGO supports IEEE single precision, double precision, single complex, and double complex and can be extended for other value types.

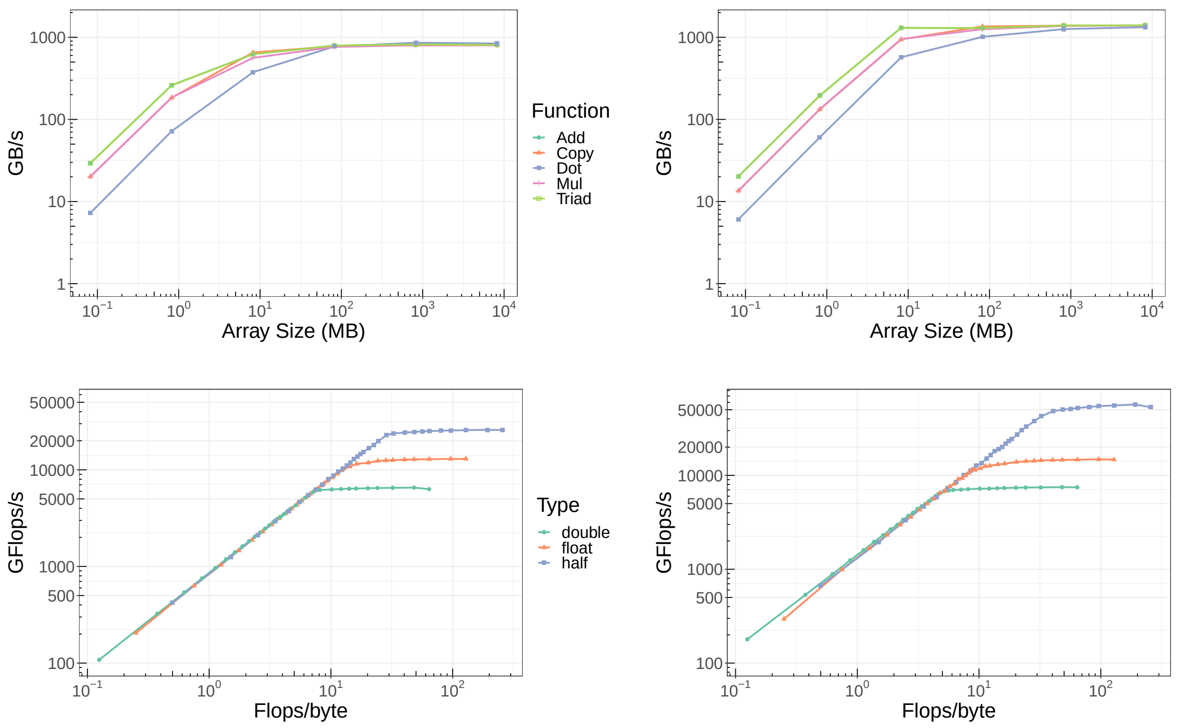


Figure 6: Performance evaluation of the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right) using the BabelStream benchmark (top) and the mixbench benchmark (bottom).

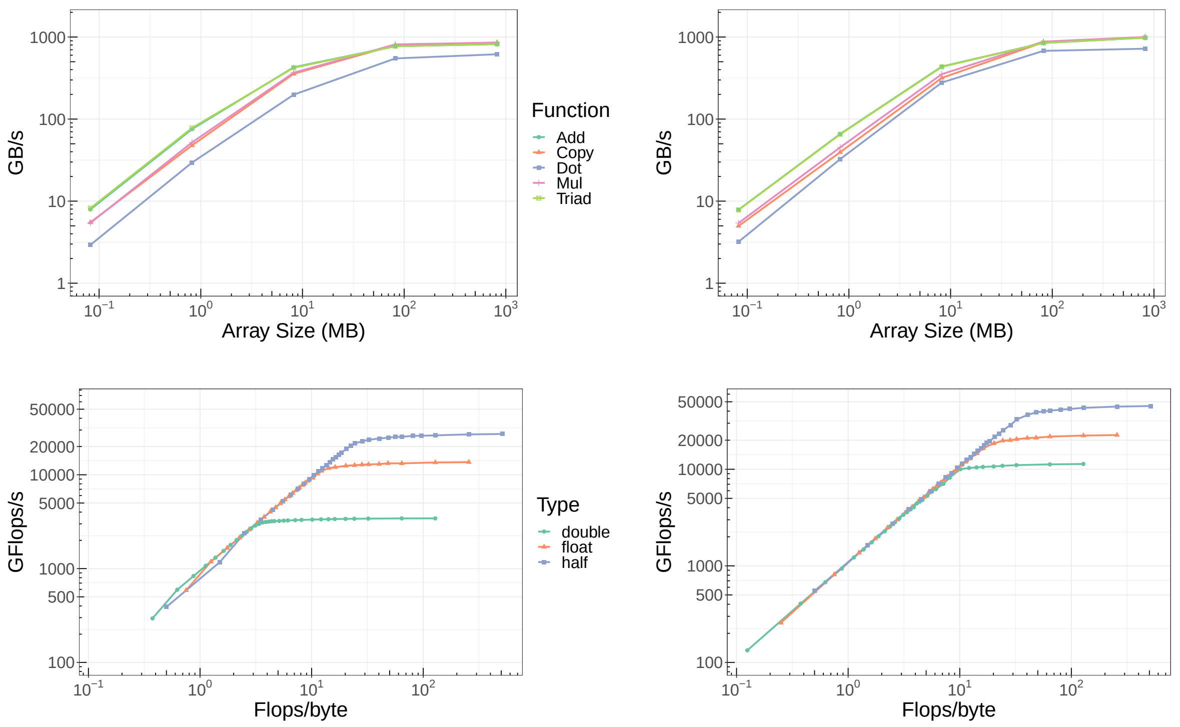


Figure 7: Performance evaluation of the AMD RadeonVII GPU (left) and the AMD MI100 GPU (right) using the BabelStream benchmark (top) and the mixbench benchmark (bottom).

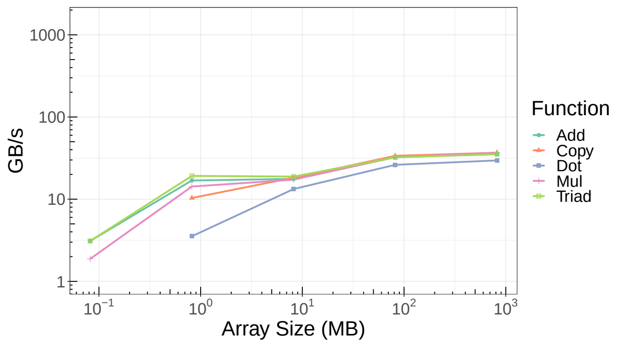


Figure 8: Performance evaluation of the Intel Gen. 9 GPU using the BabelStream benchmark.

For the general GPU cores, the peak performance of the A100 is for IEEE754 double precision and single precision about 1.2× the V100 performance. For IEEE754 half precision, the performance improvements are larger, approaching almost 2.5×, see the bottom row in Figure 6.

The two AMD GPU architectures we consider are comparable in their measured bandwidth: 850 GB/s for the AMD RadeonVII GPU vs 1,000 GB/s for the AMD MI100 GPU, see the top row in Figure 7. At the same time, they significantly differ in their arithmetic performance, see the bottom row in Figure 7.

At the time of writing the mixbench benchmark does not provide a DPC++ version for the execution on Intel GPUs. Thus, in Figure 8, we only report the bandwidth performance of the considered Intel Gen. 9 GPU. We note that the performance for the dot kernel is about 10% below the maximum achievable bandwidth. For the other operations, the bandwidth plateaus for array sizes exceeding 100 MB at a rate of 35 GB/s.

6.1. *Ginkgo SpMV performance*

We first investigate the performance GINKGO's sparse matrix-vector product kernels (SpMV) kernels achieve on the distinct executors. For this, we use matrices from the complete Suite Sparse Matrix Collection [34] as a benchmark and run a set of heavily tuned sparse matrix-vector product kernels on the distinct architectures. All computations use double precision arithmetic. The distinct SpMV kernels differ in terms of how they store the sparse matrix and which processing strategy they apply [35]. In general, we may expect a performance peak of (memory bandwidth) / (8byte/entry + 4 byte/entry + 4 byte/entry) * 2 ops/entry = (memory bandwidth) / 8 for the COO SpMV kernels, and (memory bandwidth) / 6 for the CSR kernel. For a better evaluation, we report the performance relative to the theoretical peak bandwidth for the machine, see Table 1.

In Figure 9 we report the performance of different SpMV kernels taken from either GINKGO or the vendor library (NVIDIA cuSPARSE [36]) on the NVIDIA V100 GPU (left) and NVIDIA A100 GPU (right). Each dot represents one combination of SpMV kernel and test matrix. We cannot identify a clear winner in the performance graphs, which is expected as the distinct kernels differ in their efficiency for the distinct problem characteristics. However, we can identify the GINKGO kernels to be competitive to the cuSPARSE SpMV kernels. The maximum performance numbers achieved are around 135 GFLOP/s and 220 GFLOP/s on the NVIDIA V100 and NVIDIA A100 GPUs, which amount to between 80 and 90% of the theoretical machine bandwidth. We notice that this performance difference exceeds the expectations based on the bandwidth improvements of 1.5× but remains close to the machine limits.

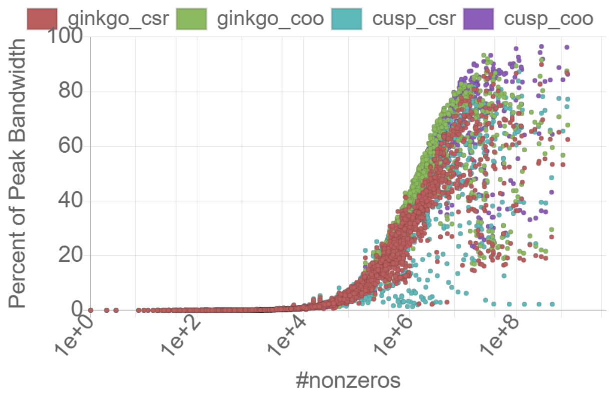
In Figure 10, we present performance results obtained from running SpMV benchmarks on AMD hardware, the AMD RadeonVII GPU (left), and the AMD MI100 GPU (right). We use the same experimental setup and again include both SpMV routines from the vendor library (hipSPARSE [31]) and from GINKGO. For both architectures, there is no clear performance winner, but we recognize GINKGO's SpMV kernels being highly competitive to the vendor library hipSPARSE. We observe that the GINKGO and hipSPARSE CSR SpMV kernels achieve up to 110 GFLOP/s on the AMD RadeonVII GPU and up to 138 GFLOP/s on the AMD MI100 GPU, or about 60 to 70% of the machine theoretical bandwidth. We advertise that the rocSPARSE library may provide higher performance than hipSPARSE, but we report the performance of the latter since our library focuses on the portable HIP layer. Furthermore, we acknowledge that the hardware and software stack for the AMD MI100 GPU is still under active development, and further performance increases can be expected.

Figure 11 presents the performance results we obtain from running the GINKGO SpMV on the Intel Gen. 9 GPU. As expected from the significantly lower bandwidth, the SpMV performance is only a fraction of the performance achieved on AMD and NVIDIA performance, which is expected given the embedded design of the Gen. 9 GPU. However, with a peak of 5 GFLOP/s for the GINKGO CSR SpMV kernel, the performance is close to the bandwidth-induced theoretical peak. We also note that the GINKGO SpMV generally outperforms the Intel MKL CSR SpMV.

6.2. *Ginkgo solver performance*

Next, we assess the performance GINKGO's Krylov subspace solvers achieve on recent hardware architectures from AMD, NVIDIA, and Intel using the respective hardware-native executors. For the performance evaluation, we select 10 test matrices that are different in size and origin to cover a wide spectrum of applications. We note that all Krylov solvers we consider are memory-bound and that the performance is hence limited by the memory bandwidth. All solvers are run for 10,000 iterations to account for machine noise. The GINKGO solvers employ the GINKGO COO SpMV for generating the Krylov subspace. Like in the SpMV results, we report the performance relative to the theoretical GPU memory bandwidth.

As the CUDA executor was developed as GINKGO's first high-performance GPU backend, we report in Figure 12 the performance of a subset of GINKGO's Krylov solvers on the NVIDIA V100 GPU (left) and the newer NVIDIA A100 GPU (right). We observe that on both V100 and A100 architectures, the results are between 60 and 80% of the peak bandwidth.



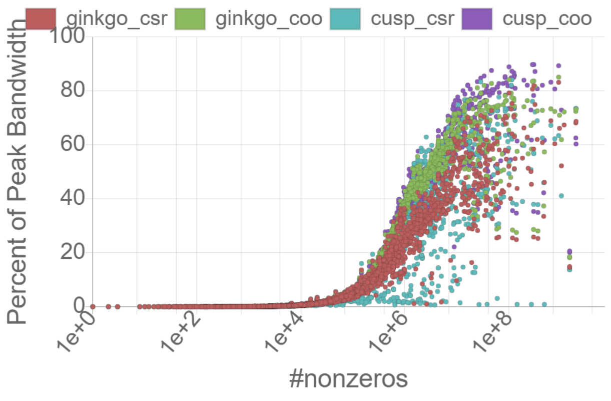
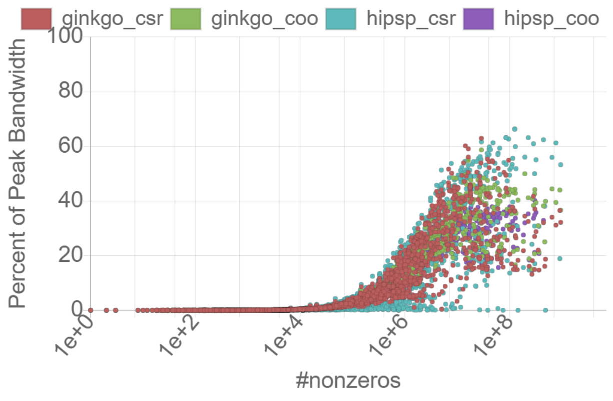


Figure 9: Performance of the GINKGO and cuSPARSE SpMV on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right).



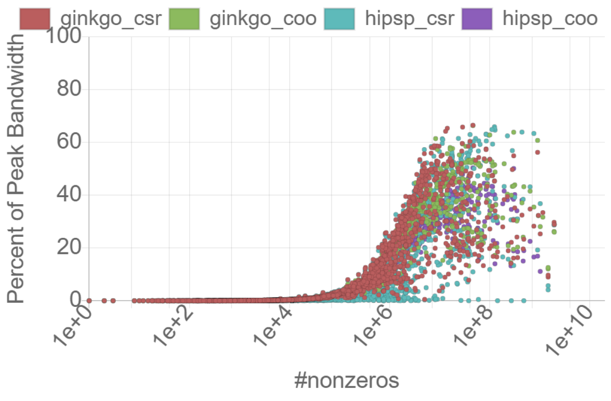


Figure 10: Performance of the GINKGO and hipSPARSE SpMV on the AMD RadeonVII GPU (left) and the AMD MI100 GPU (right).

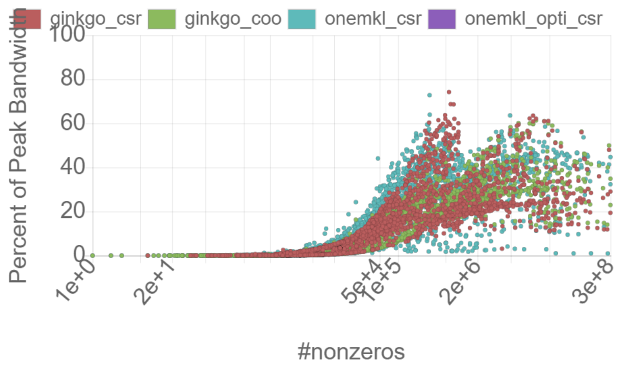


Figure 11: Performance of the GINKGO SpMV and the Intel MKL SpMV on the Intel Gen. 9 GPU.

On the A100, the first three matrices benefit from some cache effects (namely, the orthogonalization) which allows them to reach above 90% of the peak performance.

Even though originally focused on NVIDIA GPU execution via the CUDA executor, we can from the similarity in architecture design expect the algorithms to achieve high-efficiency values also on the AMD architectures. In Figure 13 we report the GINKGO Krylov solver performance on the AMD Radeon VII GPU (left) and the AMD MI100 GPU (right). These results are obtained using the HIP executor designed for AMD backends. Aside from the hardware-specific parameter configurations, the HIP kernels invoked by the Krylov solvers are very similar to the CUDA kernels used by the CUDA executor. The evaluation on the AMD RadeonVII GPU reveals the same problem-specific performance variation that we observed in the CUDA backend evaluation. On both RadeonVII and MI100 architectures, the solvers can reach more than 50% of the theoretical bandwidth. We acknowledge that the HIP executor achieves a smaller fraction of the theoretical performance bound, however, these results are in line with the SpMV performance observed in Figure 10.

The results in Figure 14 reveal the actual performance results being between 30 and 50% of the theoretical bandwidth on the Intel Gen. 9 GPU. This may be attributed to GINKGO's DPC++ executor being in his early stages, and the oneAPI ecosystem still being under development. We, again, note that the Intel Gen. 9 GPU is an integrated GPU, and not expected to achieve high performance. The purpose of this exercise is rather to show the validity of GINKGO's design, and the technology readiness for the Intel high-end GPU platform to come.

7. Summary and Outlook

In this paper, we elaborate on how GINKGO tackles platform portability by separating the numerical core from the hardware-

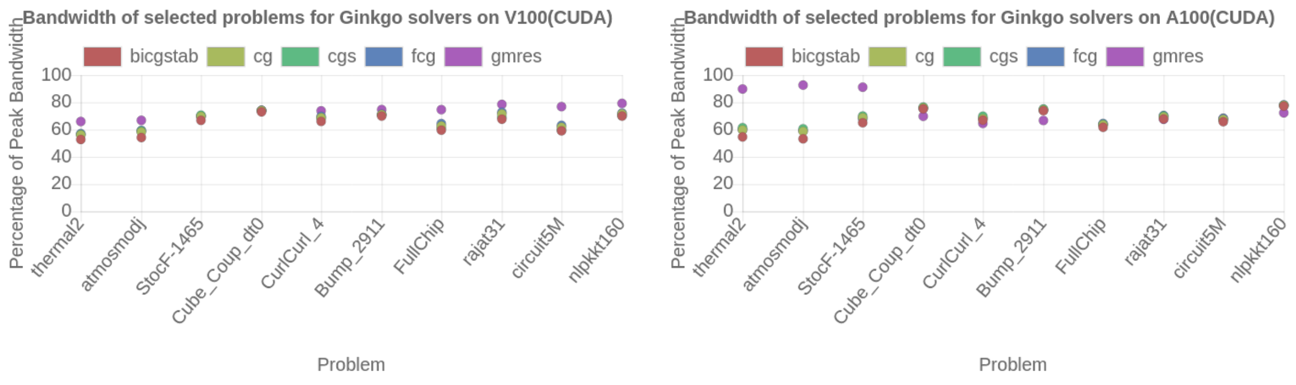


Figure 12: Krylov solver performance of the GINKGO CUDA backend on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right).

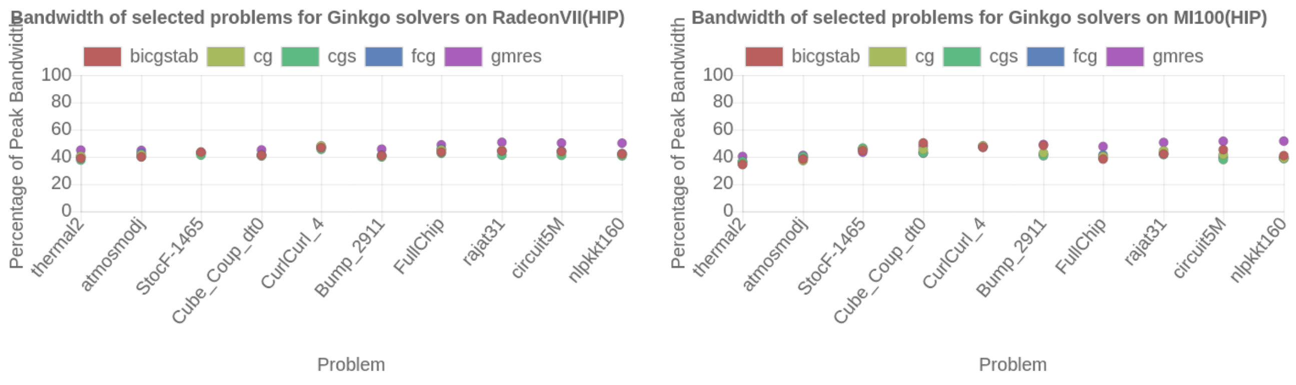


Figure 13: Krylov solver performance of the GINKGO HIP backend on the AMD RadeonVII GPU (left) and the AMD MI100 GPU (right).

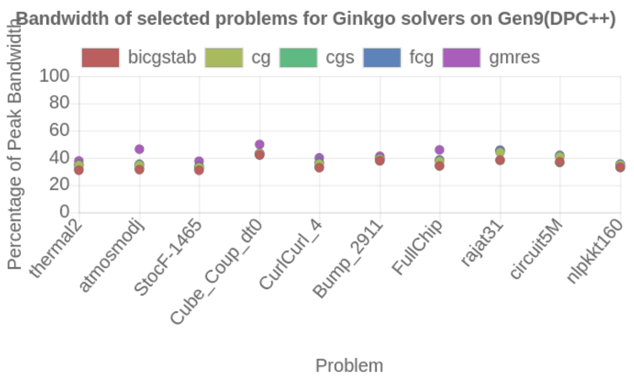


Figure 14: Krylov solver performance of the GINKGO DPC++ backend on the Intel Gen. 9 GPU.

specific backends. We discuss how we adopt the execution space to NVIDIA GPUs via the CUDA language, AMD GPUs via the HIP language, and Intel GPUs via the DPC++ language. We also report performance results for running basic sparse linear algebra operations and complete Krylov solvers on the newest hardware architectures from these vendors, and demonstrate GINKGO's performance portability and identify GINKGO's sparse matrix-vector product being highly competitive or even outperforming the vendor libraries.

Acknowledgments This work was supported by the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

References

[1] D. Arndt, W. Bangerth, T. C. Clevenger, D. Davydov, M. Fehling, D. Garcia-Sanchez, G. Harper, T. Heister, L. Heltai, M. Kronbichler, R. M. Kynch, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The `deal.II` Library, Version 9.1, Journal of Numerical Mathematics 27 (4) (2019) 203–213. `doi:10.1515/jnma-2019-0064`.
URL `https://dealii.org/deal91-preprint.pdf`

[2] The Trilinos Project Team, The Trilinos Project WebsiteAccessed on 2021/09/03 (2020).
URL `https://trilinos.github.io`

[3] R. R. Schaller, Moore's Law: Past, Present, and Future, IEEE Spectr. 34 (6) (1997) 52–59. `doi:10.1109/6.591665`.
URL `https://doi.org/10.1109/6.591665`

[4] H. Sutter, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal 30 (3) (2005) 202–210.
URL `http://www.gotw.ca/publications/concurrency-ddj.htm`

[5] H. Anzt, T. Cojean, Y.-C. Chen, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.-H. Tsai, Ginkgo: A High Performance Numerical Linear Algebra Library, Journal of Open Source Software 5 (52) (2020) 2260. `doi:10.21105/joss.02260`.
URL `https://doi.org/10.21105/joss.02260`

[6] P. J. Brown, Levels of Language for Portable Software, Communications of the ACM 15 (12) (1972) 1059–1062. doi:10.1145/361598.361624.
URL https://doi.org/10.1145/361598.361624

[7] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, D. Padua, Performance Portability with the Chapel Language, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, 2012. doi:10.1109/ipdps.2012.60.
URL https://doi.org/10.1109/ipdps.2012.60

[8] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Puschel, J. C. Hoe, J. M. F. Moura, Spiral: Extreme Performance Portability, Proceedings of the IEEE 106 (11) (2018) 1935–1968. doi:10.1109/jproc.2018.2873289.
URL https://doi.org/10.1109/jproc.2018.2873289

[9] A. Gray, K. Stratford, A Lightweight Approach To Performance Portability With Targetdp, The International Journal of High Performance Computing Applications 32 (2) (2016) 288–301. doi:10.1177/1094342016682071.
URL https://doi.org/10.1177/1094342016682071

[10] H. Carter Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. doi:10.1016/j.jpdc.2014.07.003.
URL https://doi.org/10.1016/j.jpdc.2014.07.003

[11] D. Beckingsale, R. Hornung, T. Scogland, A. Vargas, Performance Portable C++ Programming with RAJA, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 455–456. doi:10.1145/3293883.3302577.
URL https://doi.org/10.1145/3293883.3302577

[12] Khronos Group, SYCL programming framework, https://www.khronos.org/sycl/, accessed on 2021/09/03 (2014).

[13] Khronos Group, OpenCL programming framework, https://www.khronos.org/opencl/, accessed on 2021/09/03 (2009).

[14] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, O. Zinenko, Mlir: Scaling compiler infrastructure for domain specific computation, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 2–14. doi:10.1109/CGO51591.2021.9370308.

[15] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, J. Salmon, Performance Portability across Diverse Computer Architectures, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019. doi:10.1109/p3hpc49587.2019.00006.
URL https://doi.org/10.1109/p3hpc49587.2019.00006

[16] A. Poenaru, W.-C. Lin, S. McIntosh-Smith, A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application, in: Chamberlain, Bradford L. and Varbanescu, Ana-Lucia and Ltaief, Hatem and Luszczek, Piotr (Ed.), High Performance Computing, Springer International Publishing, Cham, 2021, pp. 332–350.

[17] J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, S. McIntosh-Smith, Interpreting and Visualizing Performance Portability Metrics, in: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2020, pp. 14–24. doi:10.1109/P3HPC51967.2020.00007.
URL https://doi.org/10.1109/P3HPC51967.2020.00007

[18] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans. Math. Softw. 5 (3) (1979) 308–323. doi:10.1145/355841.355847.
URL https://doi.org/10.1145/355841.355847

[19] C. L. Lawson, Background, Motivation and a Retrospective View of the BLAS, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1999, San Antonio, Texas, USA, March 22-24, 1999, SIAM, 1999.

[20] MPI Forum, MPI: A Message-Passing Interface Standard, Tech. rep., University of Tennessee, USA (1994).

[21] J. D. Mooney, Developing Portable Software, Information Technology, Kluwer Academic Publishers, nil, pp. 55–84. doi:10.1007/1-4020-8159-6_3.
URL https://doi.org/10.1007/1-4020-8159-6_3

[22] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. G. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, J. Zhang, Toward Performance-Portable PETSc for GPU-Based Exascale Systems, CoRR abs/2011.00715 (2020). arXiv:2011.00715.
URL https://arxiv.org/abs/2011.00715

[23] NVIDIA Corporation, libcu++: the NVIDIA C++ Standard Library, https://nvidia.github.io/libcudacxx/, accessed on 2021/06/19 (2020).

[24] Intel Corporation, oneDPL, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-library.html, accessed on 2021/06/19 (2021).

[25] D. S. Hollman, B. A. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, C. R. Trott, mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019. doi:10.1109/p3hpc49587.2019.00011.
URL https://doi.org/10.1109/p3hpc49587.2019.00011

[26] Google Inc., Googletest, https://google.github.io/googletest/, accessed on 2021/09/03 (2008).

[27] Y. M. Tsai, T. Cojean, H. Anzt, Sparse linear algebra on AMD and NVIDIA GPUs–the race is on, in: International Conference on High Performance Computing, Springer, Cham, 2020, pp. 309–327.

[28] Y. M. Tsai, T. Cojean, T. Ribizel, H. Anzt, Preparing Ginkgo for AMD GPUs–A Testimonial on Porting CUDA Code to HIP, arXiv preprint arXiv:2006.14290 (2020).

[29] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, J. Sewall, Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3388333.3388653.
URL https://doi.org/10.1145/3388333.3388653

[30] H. Anzt, T. Cojean, G. Flegar, F. Goebel, T. Gruetzmacher, P. Nayak, T. Ribizel, Y.-H. Tsai, E. S. Quintana-Orti, Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing, arXiv preprint arXiv:2006.16852 (2020).

[31] AMD Corporation, HIP: C++ Heterogeneous-Compute Interface for Portability, https://github.com/ROCm-Developer-Tools/HIP, accessed on 2021/09/03 (2016).

[32] E. Konstantinidis, Y. Cotronis, A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling, Journal of Parallel and Distributed Computing 107 (2017) 37 – 56. doi:https://doi.org/10.1016/j.jpdc.2017.04.002.
URL http://www.sciencedirect.com/science/article/pii/S0743731517301247

[33] Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream, Int. J. Comput. Sci. Eng. 17 (3) (2018) 247–262.

[34] SuiteSparse Matrix Collection, https://sparse.tamu.edu, accessed on 2019/03/29 (2019).

[35] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, W. Wang, Load-Balancing Sparse Matrix Vector Product Kernels on GPUs, ACM Transactions on Parallel Computing (TOPC) 7 (1) (2020) 1–26.

[36] NVIDIA Corporation, CUDA Toolkit, https://developer.nvidia.com/cuda-zone, accessed on 2021/09/03 (2007).