Received: 20 December 2021 | Revised: 5 July 2022 | Accepted: 24 August 2022

DOI: 10.1002/cpe.7400

SPECIAL ISSUE PAPER

WILEY

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Providing performance portable numerics for Intel GPUs

Yu-Hsiang M. Tsai[1] | **Terry Cojean**[1] | **Hartwig Anzt**[1,2]

[1]Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Baden-Württemberg, Germany

[2]The Innovative Computing Laboratory, University of Tennessee, Knoxville, Tennessee,

Correspondence
Hartwig Anzt, The Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA.
Email: hanzt@icl.utk.edu

Funding information
Helmholtz Association, Grant/Award Number: VH-NG-1241; U.S. Department of Energy, Grant/Award Number: 17-SC-20-SC

Summary

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 article, we report how we enable the Ginkgo math library to execute on Intel GPUs by developing a kernel backed based on the DPC++ programming environment. We discuss conceptual differences between the CUDA and DPC++ programming models and describe workflows for simplified code conversion. We evaluate the performance of basic and advanced sparse linear algebra routines available in Ginkgo's DPC++ backend in the hardware-specific performance bounds and compare against routines providing the same functionality that ship with Intel's oneMKL vendor library.

KEYWORDS

Ginkgo, Intel GPUs, math library, oneAPI, SpMV

1 | INTRODUCTION

To complement the arrival of the first discrete Intel GPUs, Intel has teamed up with partners from academia and industry to create the oneAPI open standard for a unified application programming interface intended to be used across different compute accelerator architectures, including GPUs, AI accelerators, and field-programmable gate arrays (FPGAs). oneAPI relies on the SYCL-based DPC++ programming language, and despite Intel currently being the driving force, the intention of the oneAPI standard is to enable platform portability across hardware from Intel and other vendors. A challenge in this context is that parallelization concepts like workgroup size can have very different characteristics for a hardware zoo that spans from multicore CPUs to manycore GPUs and special function units.

In Tsai et al.,[1] the authors reported how they prepared the math library Ginkgo[2] to execute on Intel GPUs by adding a backend containing kernels written in the DPC++ programming language. This article is an extended version of Tsai et al.,[1] building upon the previous effort, and carrying it further with the development of a kernel selection scheme that allows for compiling kernels for different hardware characteristics and automatically selecting an appropriate kernel via a unique kernel signature. Furthermore, we now also provide DPC++ kernels for sophisticated algorithms and preconditioners, including parallel incomplete factorizations and mixed-precision block-Jacobi and sparse approximate inverse preconditioners. This article is structured in the following way. We first introduce the oneAPI ecosystem in Section 2 before reviewing the GINKGO library design in Section 3. Section 4 is the main contribution of the article, which contains the porting strategy as well as the kernel selection strategy enabling performance portability. We evaluate the performance of sparse linear algebra workflows in Section 5 before concluding in Section 6.

2 | THE ONEAPI PROGRAMMING ECOSYSTEM

oneAPI[*] 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.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2022 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

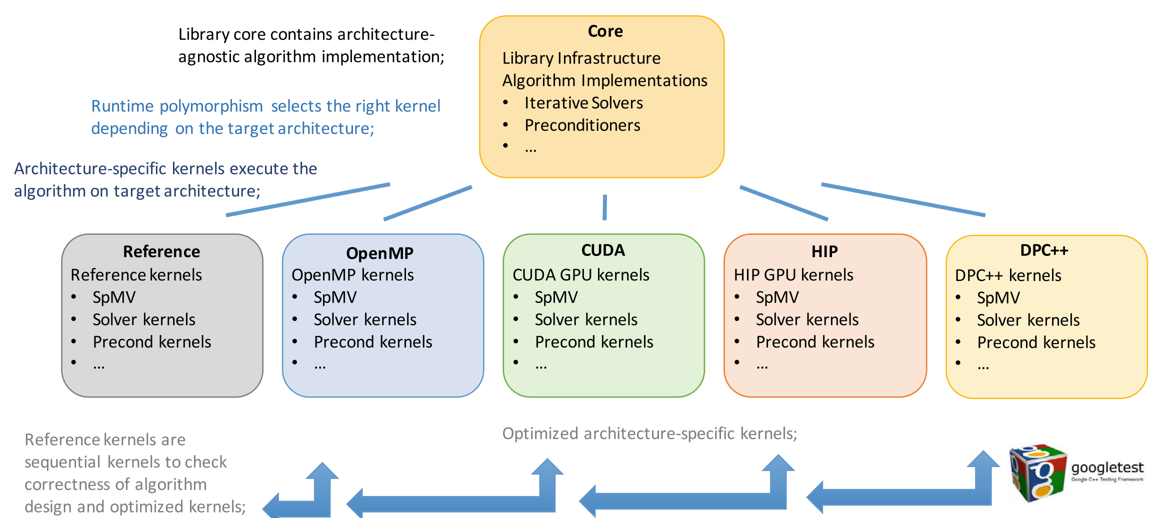

FIGURE 1 The GINKGO library design overview.

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[3] 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[†]: (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 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 facilitate 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++ introduces a plugin API that allows the development of new backends and switches dynamically between them[‡]. Currently, DPC++ supports the standard OpenCL backend, a new Level Zero backend which is the backend of choice for Intel hardware[§], 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++.

3 | GINKGO DESIGN

GINKGO[2] 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).[4] 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 Figure 1. The library design collects all classes and generic algorithm skeletons in the "core" library which, however, cannot be used without the driver kernels available in the "omp," "cuda," "hip," and "reference" backends. We note that "reference" contains sequential CPU kernels designed to validate the correctness of the parallel algorithms and 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.[5] Extending GINKGO 's scope to support Intel GPUs via the DPC++ language, we add the "dpcpp" backend containing corresponding kernels in 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 Section 3 promotes a natural porting workflow: (1) As a first step, core library infrastructure needs to acknowledge the addition of a new backend.

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 simplifying 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.

We will expose Steps 1–3 of this workflow in Section 4.1. The later sections introduce three of the aspects of Step 4 of the workflow, where we discuss challenges and optimization opportunities specific to the oneAPI/SYCL ecosystem. In Section 4.2, we focus on the SYCL concept of subgroup, discuss its relation and mapping to CUDA and propose viewing it as CUDA subwarp instead of CUDA warp. In Section 4.3, we introduce some performance and portability aspects on Intel oneAPI of GINKGO'S advanced preconditioners which are some of the most complex algorithms we support. Finally, in Section 4.4, we introduce a new C++ concept and relevant compiler interactions which allow our kernels to be portable to all Intel hardware while being able to be tuned for performance.

4.1 | Porting CUDA code to DPC++

To facilitate an initial porting of the functionality without optimization, 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 code "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 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.

4.1.1 | Isolated kernel modification

DPCT converts all files related to the target file containing any CUDA code that is 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>`.

4.1.2 | 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 performing 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 for 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 for subgroups. This lack of native

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

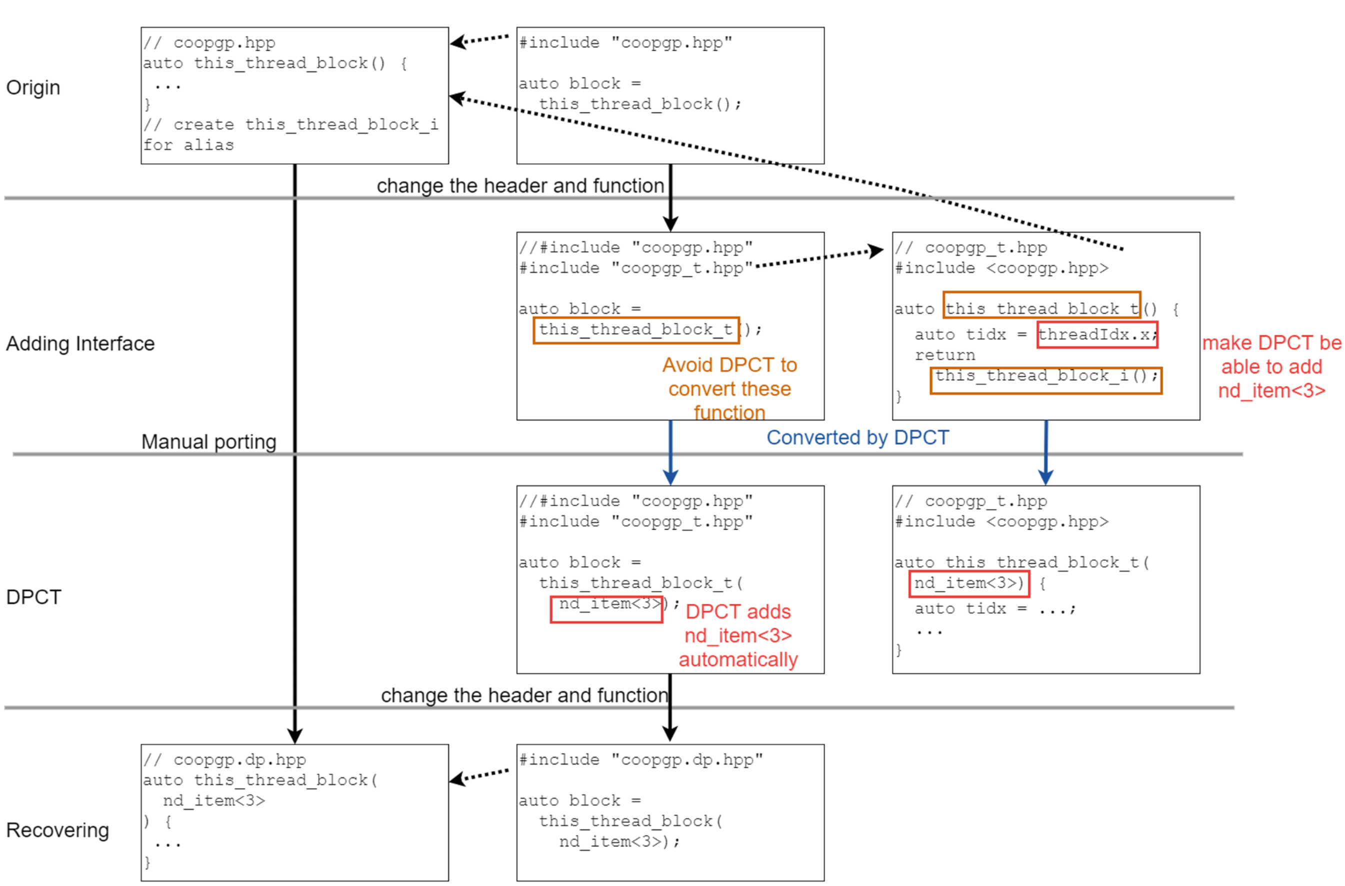

FIGURE 2 Summary of the workflow used to port the cooperative groups' functionality and isolating effort such that we get the correct converted DPC++ codes.

support may affect the performance of kernels relying on these subgroup operations. In Figure 2, we visualize the porting workflow for 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 Figure 3, the final result of the porting workflow on a toy example with cooperative groups. For the small example code in Figure 3A, if we do not isolate the code, DPCT will throw an error like Figure 3B once encountering the cooperative group keyword. A manual implementation of the cooperative group equivalent kernel is shown in Figure 3C. Our porting workflow generates the code shown in Figure 3D, which is almost identical to the original CUDA code Figure 3A.

4.1.3 | 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 look similar to CUDA/HIP code[¶]. In particular, we not only add the customized cooperative group interface, but also a dim3 implementation layer for DPC++ kernel launches that uses the same parameter order as CUDA and HIP kernel launches.

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

```
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 is 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

FIGURE 3 The cooperative group example. (A) CUDA cooperative group example; (B) DPCT conversion reports an error; (C) Manual DPC++ subgroup implementation. The main difference from CUDA is in orange; (D) The result converted by our porting script

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. We extend this design for supporting different device configurations as well in Section 4.4 and use both of these approaches in GINKGO . If the kernels require a subgroup or workgroup size selection, we add the configuration selection into the call, otherwise, we use the simple kernel interface without configuration input.

4.2 | SYCL subgroups compared to CUDA warps and subwarps

In SYCL and DPC++, subgroups are equivalent to the CUDA warp concept. However, there are two differences between SYCL subgroups and CUDA warps: (1) SYCL subgroups are adjustable for kernels (2) SYCL does not have a sub-subgroup (subwarp) concept for the moment. In this situation, we decide to rather match subgroups to CUDA subwarps. This allows the usage of small subgroups similar to subwarps as long as there is no major communication out of the given subgroup. In addition, choosing a suitable subgroup size is of importance because it gives better performance than the automatic SYCL choice, see the small reduction example in Figure 4. In the original SYCL-1.2.1 standard, Intel OneAPI has proposed a DPC++ extension that introduces the subgroup concept along with propagation rules for the kernel attribute from deep inside a kernel call nesting to the

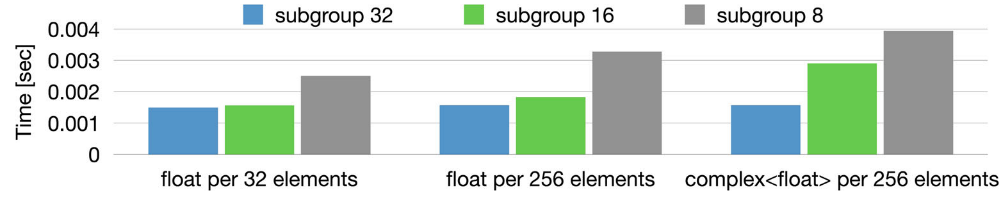

FIGURE 4 The performance of subset reduction on 10,000,000 elements on GEN12 GPU

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lice

outer-most level, such that the compiler recognizes this setting to generate a tailored version of the kernel. This allows a very similar usage of cooperative groups in DPC++ and CUDA and a smooth conversion of CUDA kernels to DPC++ via our porting workflow with DPCT. The conversion process as well, as the cooperative group porting process, are described in Section 4.1.

However, the new SYCL-2020 standard changes the attribute propagation for the subgroup setting such that all kernel attribute propagation is disabled. This virtually prevents the strategy of setting the subgroup size inside the cooperative group. From the standard's perspective, it might be reasonable to ignore subgroup settings from deep inside the kernel because it is essentially considered like a CUDA warp and not a subwarp. An overhead-free workaround is given by re-using an intermediate layer that manages the subgroup settings, see "Pushing for backend similarity" of Section 4.1. However, the cooperative groups themselves under SYCL will only have a debug-mode check and no real effect on the subgroup setting.[#]

```
1  // Build a parilu factory with 15 iterations
2  auto par_ilu_fact =
3      gko::factorization::ParIlu<ValueType, IndexType>::build().with_iterations(15u).on(exec);
4  // Create the adaptive block jacobi factory with max block size 32
5  auto bj_factory =
6      Jacobi<ValueType>::build()
7          .with_max_block_size(32u)
8          .with_storage_optimization(gko::precision_reduction::autodetect())
9          .on(exec);
10 // Create IR solver with the adaptive block jacobi as triangular solver
11 auto trisolve_factory =
12     ir::build()
13         .with_solver(share(bj_factory))
14         .with_criteria(
15             gko::stop::Iteration::build().with_max_iters(sweeps).on(exec))
16         .on(exec);
17 // Create the ILU preconditioner and use the IR to solve lower/upper triangular solver
18 auto ilu_pre_factory =
19     gko::preconditioner::Ilu<ir, ir>::build()
20         .with_l_solver_factory(gko::clone(trisolve_factory))
21         .with_u_solver_factory(gko::clone(trisolve_factory))
22         .on(exec);
23 // Generate the ParILU factorization of system matrix A
24 auto par_ilu = par_ilu_fact->generate(A);
25 // Use incomplete factors to generate ILU preconditioner
26 auto ilu_preconditioner = ilu_pre_factory->generate(gko::share(par_ilu));
27 // Generate a PCG with the ILU preconditioner
28 auto ilu_cg_factory =
29     cg::build()
30         .with_criteria(gko::share(iter_stop), gko::share(tol_stop))
31         // without the following line, it will be plain cg.
32         .with_generated_preconditioner(gko::share(ilu_preconditioner))
33         .on(exec);
34 // Generate preconditioned solver for a system matrix A
35 auto ilu_cg = ilu_cg_factory->generate(A);
36 // Solve the system
37 ilu_cg->apply(lend(b), lend(x));
```

Listing 1: ParILU IR PCG example

4.3 | Porting GINKGO 's advanced preconditioners on Intel oneAPI

GINKGO contains a set of advanced preconditioners designed to leverage the compute power of modern GPUs by allowing for fine-grain parallelism and using lower precision formats for parts of the computations. We have ported the following preconditioner functionality to the DPC++ backend: (1) adaptive precision block-Jacobi preconditioning,[6] (2) routines for generating incomplete factorization preconditioners, in particular the parallel incomplete LU factorization (ParILU[7]), the parallel incomplete Cholesky factorization (ParIC), and their respective threshold versions (ParILUT/ParICT[8,9]), and (3) incomplete sparse approximate inverse preconditioners (ISAI[10]). In comparison to the vendor libraries for AMD and NVIDIA GPUs, oneMKL is in the early stage and still missing some key functionality such as sparse triangular solves that we use for incomplete factorization preconditioning on NVIDIA and AMD GPUs. However, incomplete factorization preconditioning is still possible also in the DPC++ backend by using approximate triangular solves that employ a relaxation method for solving the triangular systems.[11] Listing 1 demonstrates the use of approximate triangular solves: we first build a ParILU factorization factory with its settings (line 2–3). Next, we declare an adaptive block-Jacobi factory, which is used as a solver in an iterative refinement (IR) solver, in (lines 7–18) and replace the missing triangular solver. Then the ILU preconditioner is built using all the previous pieces (lines 20–26). Finally, the incomplete factorization preconditioner using approximate triangular solves is passed as a preconditioner to a CG solver (lines 28–37).

4.3.1 | Challenges with ParIC/ILU(T)

A challenge when porting the ParILUT[8,9] algorithm to DPC++ is the limitation in terms of using only one fixed-size subgroup in a kernel (i.e., the lack of a CUDA subwarp equivalent). ParILUT employs divide and conquer to handle the bitonic sorting, such that the working space is: block -> block/2

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

-> · · · -> warp -> subwarp(16) -> subwarp(8) -> · · · -> single thread. Its implementation relies on cooperative groups to efficiently handle each subwarp level, which is not possible in DPC++ due to the fixed subgroup size. Acknowledging that all subwarps execute the same algorithm, we can still use the algorithm with all subgroups (subwarps) having the same size. This workaround does however not follow the cooperative group concept. That is, the kernels on DPC++ leverage the divide and conquer strategy on the workload view and stop when reaching the cooperative group level. When reaching the subgroup size, for example, 32, the divide and conquer logic is implemented within the kernel without changing the subgroup size. To make this work, the kernel needs to compute or count indices carefully. For example, `ffs` gets the first set bit (1-based) in the given mask. `len(mask) - clz(mask)` gives the wrong answer when the `len(mask)` does not match the size of subgroup. We need to use `ctz(mask) + 1` to ensure correctness.

4.3.2 | Challenges with adaptive precision block-Jacobi

Also, the adaptive precision block-Jacobi[6] algorithm makes heavy use of subwarps as matching diagonal blocks to subwarps requires adjusting the subwarp size to the size of the diagonal block. The original CUDA implementation checks the diagonal blocks for the precision requirements in a subwarp and then uses the strongest requirement for all subwarps of the same warp. This scheme means that we also face the issue that SYCL/DPC++ does not support variable sub-subgroup sizes. It is possible to always use the same subgroup size to handle the blocks, which means that blocks need to be considered individually but this will produce different block precisions compared to the other backends. As long as oneAPI does not allow for flexibility in the subwarp size, we limit the block-Jacobi preconditioner to a uniform block size of 32.

4.4 | Improving performance and portability of GINKGO 's DPC++ backend

An important aspect of oneAPI is hardware portability. In effect, oneAPI is expected to execute not only on Intel architectures (CPUs, GPUs, and FPGAs) but also on NVIDIA GPUs and AMD GPUs. In practice, this requires supporting a wide range of hardware characteristics and adjustment of the kernel execution parameters. For example, in the past, Intel CPUs supported subgroup sizes 4, 8, or 16, and Intel GPUs supported subgroup sizes 8, 16, or 32. Starting with DPC++ 2021.4, the Intel CPUs can handle subgroup sizes 4, 8, 16, 32, or 64 such that the settings 8, 16, and 32 are available on both Intel architectures. But when considering NVIDIA hardware, the subgroup size is fixed to 32. Another example of a hardware-dependent kernel launch characteristic is the maximum workgroup size that depends on the specific device type. To enable kernels to execute with high performance on different architectures, we introduce the concept of a `ConfigSet` that encodes kernel execution parameters in a kernel signature. This concept allows to compile a kernel in different configurations and then select the kernel variant with the optimal configuration at runtime.

4.4.1 | Subgroup configurations impacting kernel performance

To demonstrate the necessity to not only select the kernel configuration in the device-specific valid range but to also optimize it in the performance metric, we visualize in Figure 4 the performance of a reduction kernel acting on an array of 100,000,000 elements on the Intel GEN12 GPU. The reduction kernel contains a group-local reduction with a group size larger or equal to the subgroup size but smaller than the workgroup size. The subgroup sizes are 8, 16, or 32, the workgroup size is 256, and the reduction lengths are 32 or 256 elements. All configurations are valid for the GEN12 architecture, however, result in very different runtime performance. For a reduction size of 32, we notice small differences between the subgroup sizes 16 and 32, but a subgroup size of 8 results in lower performance. The trend remains when moving to reduction size of 256, suggesting that subgroup sizes 16 and 32 always result in similar performance. The picture changes when operating on the data type complex float: here, the subgroup size 16 results in significantly lower performance. This reveals that not only hardware characteristics but also datatype can matter when selecting a suitable kernel configuration.

4.4.2 | Encoding kernel parameters with the `ConfigSet`

We propose a compile-time structure `ConfigSet` which supports compile-time encoding of multiple parameters into a single bitset and both compile-time and runtime decoding of the configuration. The architecture of the `ConfigSet` is detailed in Figure 5. First, the type needs to be declared and specifies the sizes and order of the encoded numbers, for example, `Config<3, 11, 7>` indicating that the first element uses up to 3 bits, the second element uses up to 11 bits, and the last element uses up to 7 bits. The user can encode a set configuration (e.g., `4, 256, 16`) into a number at compilation-time. This number can be used as a template parameter value for the kernel which greatly simplifies the implementation process of the strategy, as all relevant numerical parameters are specified and checked against this single encoded result. Inside the kernel, the `ConfigSet` is then decoded and the kernel parameters such as subgroup size and group size are extracted.

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

FIGURE 5 The encoding example of ConfigSet

```
 1 // the code in practice deletes all comments and add \ to the end of each line
 2 #define GKO_ENABLE_IMPLEMENTATION_CONFIG_SELECTION(_name, _callable)
 3 template <typename Predicate,
 4           // BoolArgs, IntArgs, SizeTArgs is a union of all kernel's template arguments
 5           bool... BoolArgs, int... IntArgs, gko::size_type... SizeTArgs,
 6           // TArgs is for no-auto deduced template type
 7           typename... TArgs, typename... InferredArgs>
 8 // this is for the last case - non matching throw error
 9 inline void _name(::gko::syn::value_list<std::uint32_t>, Predicate,
10                   ::gko::syn::value_list<bool, BoolArgs...>,
11                   ::gko::syn::value_list<int, IntArgs...>,
12                   ::gko::syn::value_list<gko::size_type, SizeTArgs...>,
13                   ::gko::syn::type_list<TArgs...>,
14                   InferredArgs...) GKO_KERNEL_NOT_FOUND;
15
16 // K is the current number will be check by is_eligible
17 // Rest are the rest of value in the list
18 template <std::uint32_t K, std::uint32_t... Rest, typename Predicate,
19           bool... BoolArgs, int... IntArgs, gko::size_type... SizeTArgs,
20           typename... TArgs, typename... InferredArgs>
21 inline void _name(
22     // the given possible list all are known in compile time
23     ::gko::syn::value_list<std::uint32_t, K, Rest...>,
24     // is_elligible allows the user to control whether K is matching or not
25     Predicate is_elligible, ::gko::syn::value_list<bool, BoolArgs...> bool_args,
26     ::gko::syn::value_list<int, IntArgs...> int_args,
27     ::gko::syn::value_list<gko::size_type, SizeTArgs...> size_args,
28     ::gko::syn::type_list<TArgs...> type_args, InferredArgs... args)
29 {
30     if (is_elligible(K)) {
31         // runtime decision: if K is elligible, call the real implementation _callable
32         _callable<BoolArgs..., IntArgs..., SizeTArgs..., TArgs..., K>(
33             std::forward<InferredArgs>(args)...);
34     } else {
35         // If K is not passed, go to check the rest of the possible solution
36         _name(::gko::syn::value_list<std::uint32_t, Rest...>(), is_eligible,
37               bool_args, int_args, size_args, type_args,
38               std::forward<InferredArgs>(args)...);
39     }
40 }
```

Listing 2: Runtime decision of compiled time generation

4.4.3 | Runtime selection of compile-time parametrized kernels

Leveraging the `ConfigSet` in kernels requires a surrounding infrastructure. We realize this in a C++ compile-time and runtime component which we call "implementation selection." The main purpose of this component is to automatize the process of generating compile-time kernels for a set of parameters, and providing runtime interfaces: (1) The developer implements a kernel and adds a template parameter to this kernel which contains the relevant parameters (e.g., `ConfigSet` encoded integer values) that are required in the kernel. (2) The developer specifies a compile-time (constexpr) instantiation list for all parameters. To facilitate this step we provide a type named `syn::value_list<int_type, value1, value2, ...>`. (3) A wrapper function is generated for the templated kernel, which takes the `syn::value_list` as a parameter. A user-specified predicate, calls the correct kernel instantiation or throws an exception if a kernel satisfying the predicate is not found. This function is generated at compile-time for all the values of the `syn::value_list`. At runtime, only a few branches are evaluated for picking the correct implementation. Listing 2 lists the runtime selection with the default case throwing an exception (lines 3–14) and looping over the `syn::value_list` elements one by one trying to find a kernel matching the `is_elligible` predicate (line 30). (4) The user implements a host interface to the kernel which calls the wrapper function of Step 3 with the correct predicate and types to pick the desired kernel based on his behavior. We visualize this compile-time generation of templated kernels in Figure 6.

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

FIGURE 6 Compile-time selection of configuration list <A, B, C, … >. The func<A, RestA … > will check the first configuration - A. If it is matched, then the kernel<A> is run, otherwise, the function is called with the configurations <B, C, … > = RestA as func<RestA … >. The func<RestA … > = func<B, RestB … > has the same behavior as above func<A, RestA … > but checks configuration B. The template is expanded automatically to generate the selection phase for runtime and prepare the possible kernels at compile time.

```
using cfg_t = ConfigSet<11, 7>;
constexpr auto cfg_list =
    syn::value_list<int, cfg_t::encode(512, 32), cfg_t::encode(256, 16)>;

template <int cfg, typename ValueType>
void reduction_kernel(sycl::queue* queue, ...) {
    queue->submit(/* ... kernel call implementation */);
}
// generate the runtime parameter -> compile time wrapper
GKO_ENABLE_IMPLEMENTATION_CONFIG_SELECTION(reduction_config,
                                           reduction_kernel);
// calls wrapper with the predicate to find the desired config
// in practice, generated automatically
template <typename ValueType>
void reduce_add_array_call(int desired_cfg, ...) {
    reduction_config(cfg_list,
        [&desired_cfg](int cfg) { return cfg == desired_cfg; },
        ...);
}

template <typename ValueType>
void reduce_add_array(shared_ptr<DpcppExecutor> exec, ...) {
    auto queue = exec->get_queue();
    // get_first_cfg will return the first valid number in the first input array
    const auto cfg = get_first_cfg(cfg1d_array, [&](int cfg){
        if (cfg_t::decode<0>(cfg) in exec->get_subgroup_list()
            and cfg_t::decode<1>(cfg) < exec->get_max_workgroup()) {
            return true;
        }
    });
    // generate the corresponding grid and block according to the cfg
    // each block performs reduction on partial array and results in a block-size array
    reduction_call(cfg, grid, block, 0, queue, ...);
    // one block performs reduction on the block-size array
    reduction_call(cfg, 1, block, 0, queue, ....);
}
```

Listing 3: The reduction kernel call example

The `ConfigSet` concept allows selecting a valid and well-performing kernel during runtime without significantly increasing the development effort. In Listing 3, we present the full implementation selection via the `ConfigSet` for a reduction kernel. In this example, we consider (256, 16) as a valid (workgroup, subgroup) configuration for the GEN9 GPU and (512, 32) for the GEN12 (lines 1–3) GPU due to the devices' max workgroup size limitation and other hardware characteristics. We note that because `get_first_cfg` returns the first valid config, the ordering of the list entries affects which configuration is selected. We also note that it is possible to extend the `ConfigSet` and the implementation selection process to account also for other hardware characteristics.

4.4.4 | Overhead of the runtime selection

To evaluate the overhead of the runtime selection, we take the kernel performing reduction per 256 elements with single precision from Figure 4 as an example, because the reduction is a common kernel requiring subgroup functionality to get good performance. We generate a certain number of kernels at compile time and select the last generated kernel at runtime. Our experiment runs one warmup and measures the average of 100 runs,

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

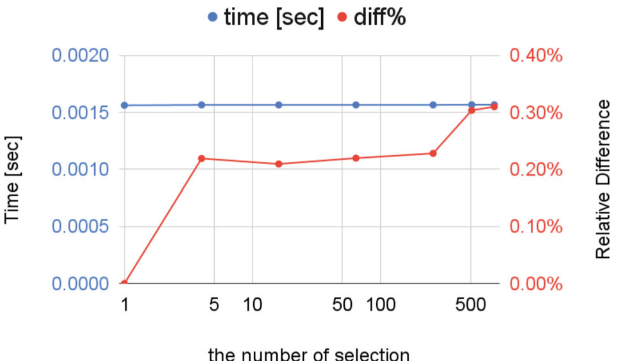

FIGURE 7 The overhead of runtime selection on Gen12LP GPU. "x" is the number of kernels in the selection list generated at compile-time and we select the last kernel in the list at runtime. "time" includes the selection and the kernel run. "relative difference" is the relative difference (in percentage) between the current case and the 1-selection case.

where each run selects kernels on the GEN12 GPU and we check using from 1 to 768 kernels in the compile time list. In Figure 7, there is a time increase when we choose the last kernel in a list with more than 512 kernels, but it is still very small with only up to 0.3% of the time (i.e., 4.84 μs)[‖]. Thus, the overhead from the selection is not a major issue for the kernel performance.

4.4.5 | Compiling GINKGO in the oneAPI ecosystem

GINKGO allows for integration with the oneAPI and SYCL toolchains by changing the default C++ compiler, for example, via `-DCMAKE_CXX_COMPILER=dpcpp`. This compiler setting implies that not only the DPC++ backend, but the full project then uses the DPC++ compiler. GINKGO 's DPC++ functionality relies on other oneAPI-specific libraries, namely Intel oneMKL and oneDPL. oneMKL is used as a replacement for cuBLAS, cuSPARSE, and other CUDA math libraries, whereas oneDPL is used as a replacement for some thrust[12] functionality. These third-party libraries also have different (linking) behavior depending on the C++ compiler. For example, oneMKL sets extra flags for just in time (JIT) compilation and other features. All these options are available only when using `dpcpp` as the main C++ compiler.

There are two compilation modes for DPC++ functionality. The first mode is JIT compilation where the specific kernel code is automatically generated at runtime, whereas ahead of time compilation (AOT) pre-generates the kernel code for the specific architecture[**]. The selection of AOT and JIT compilation is controlled via the `-fsycl-targets` compiler option[††]. The default mode `spir64` enables JIT whereas `spir64_gen` enables AOT compilation for specific hardware types (in this case, a GPU). In addition, a specific device generation can be targeted by using the flag `-Xsycl-target-backend` such as the Intel GEN9 . Whether AOT or JIT is used, and in which manner, has important implications for the previous `ConfigSet` and implementation selection concepts: when generating exhaustively the kernel configuration for all potential hardware settings, the kernel compilation may fail because it is not supported for the target hardware type and device pair. JIT compilation can work transparently by using the compiler option `-fsycl-device-code-split` which controls the granularity of device kernel code modules: the default `auto` uses an algorithm to try to recognize it, whereas `per_source` compiles each source file as a device code module, and `per_kernel` generates a different device code module for every kernel (including template parameters). Setting a value of `per_kernel` in combination with JIT implies that all code paths for which the hardware is not compatible are not evaluated at runtime, and therefore never compiled. On the other hand, a `per_source` value raises a runtime JIT error if an invalid kernel configuration is encountered. However, the convenience of the `per_kernel` compilation comes with an important cost: all kernel dependencies (like cooperative group and other pieces of code) are bundled in every kernel module separately, which can significantly increase the compilation overhead as well as library size, particularly in debug mode. To address this downside and enable more flexibility, we also support the JIT `per_source` mode and AOT if the user explicitly specifies the relevant hardware parameters of the target device, and this information is then used to automatically generate the configuration list like in line 2 of Listing 3 from within CMake.

5 | PERFORMANCE ASSESSMENT OF GINKGO'S DPC++ BACKEND

In this section, we evaluate all aspects of the newly ported GINKGO backend on available Intel hardware. In Section 5.1, we present the platforms we consider and the experimental setup. We then start with evaluating the performance of the SPMV kernel in Section 5.2 and its performance portability on Intel, NVIDIA, and AMD GPUs. The performance portability of the Krylov solvers and the performance of the advanced preconditioners are evaluated in Sections 5.3 and 5.4, respectively. All performance results including the above testings are available in the dataset.[13]

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lice

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

5.1 | Experimental setup and platform overview

In this article, 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)[‡‡] which features 96 execution units and a theoretical bandwidth of 68 GB/s. We start with initially evaluating the performance and bandwidth of the Intel GPUs using 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 into whether GINKGO'S algorithms are suitable for Intel GPUs, we use single-precision arithmetic in all performance evaluations on the GEN12 architecture[§§]. The DPC++ version we use in all experiments is Intel oneAPI DPC++ compiler 2021. All experiments were conducted on hardware that is part of the Intel DevCloud. We have performed bandwidth tests and experimental performance roofline in the article.[1] We use the BabelStream[14] benchmark to evaluate the peak bandwidth, and the mixbench[15] benchmark to evaluate the arithmetic performance. The GEN9 architecture achieves 37 GB/s and the GEN12 architecture achieves about 58 GB/s for large array sizes. 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. 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.

5.2 | SPMV performance analysis and portability

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 Figure 8, 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 [¶¶]. The achieved performance in terms of percentage of peak bandwidth is exposed in Figure 9.

We also 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

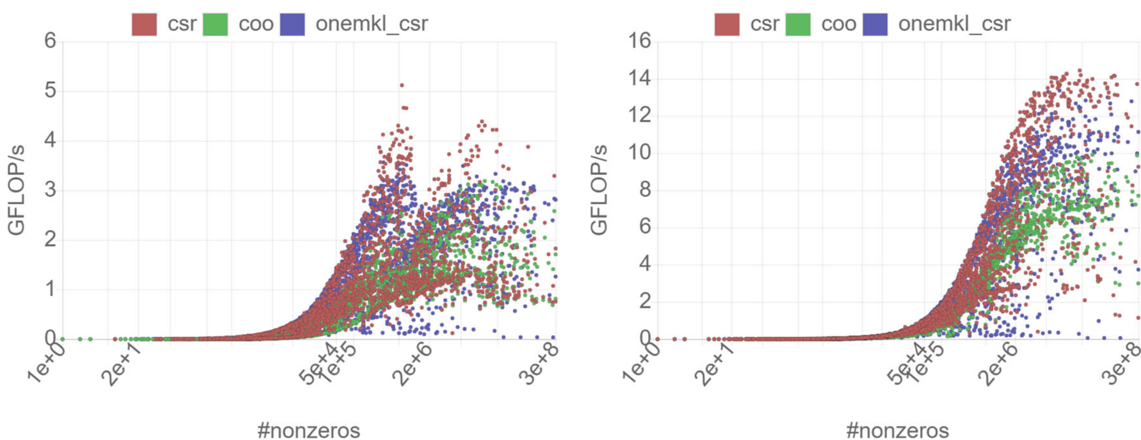

FIGURE 8 SPMV kernel performance for GINKGO and Intel's oneMKL library on GEN9 (left) and GEN12 (right) using double and single precision, respectively.

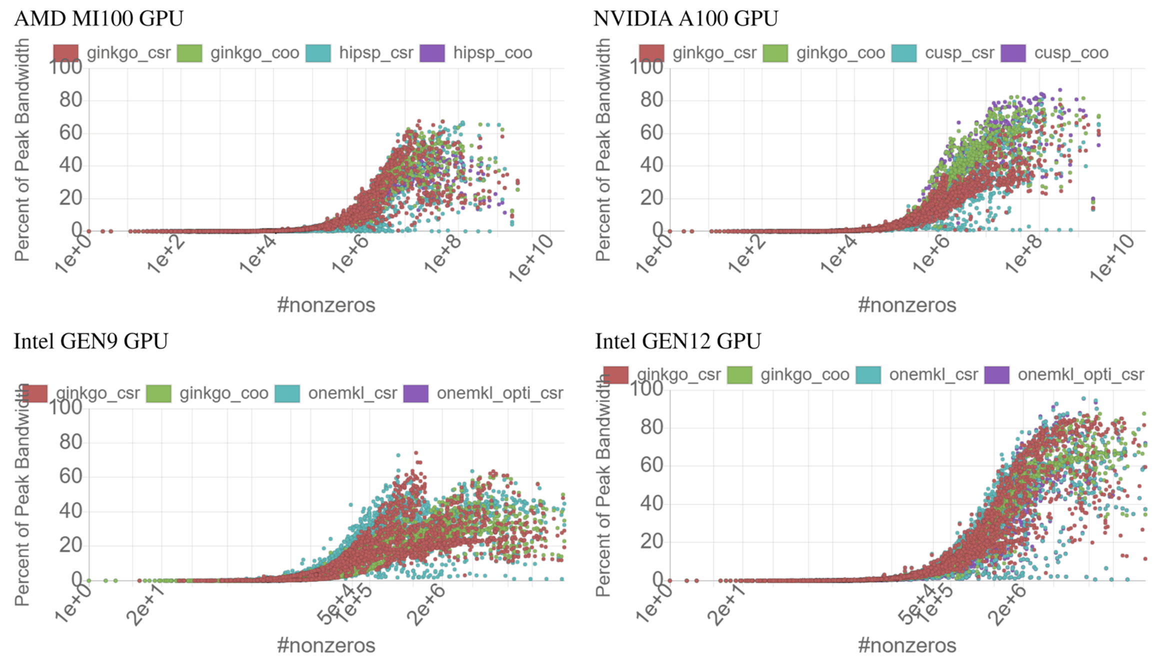

FIGURE 9 SpMV performance relative to the hardware bounds on various GPUs.

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 Figure 9 we report the relative performance of different SPMV kernels on AMD MI100 ("hip" backend), NVIDIA A100 ("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 80% of peak bandwidth on A100, 90% of peak bandwidth on GEN12 , and about 60%–70% of peak bandwidth on MI100 and GEN9 . On all hardware, GINKGO'S SPMV kernels are competitive with the vendor libraries, indicating the validity of the library design and demonstrating good performance portability.

5.3 | Krylov solver performance analysis and performance portability

We now turn to advanced numerical algorithms typical of scientific simulation codes. The Krylov solvers we consider—CG, BiCGSTAB, CGS, FCG, and GMRES[16]—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 SuiteSparse Matrix Collection[17] that are orthogonal in their characteristics and origin, and their characteristics are listed in Table 1. We run the solver experiment for 1000 solver iterations after a warm-up phase.

We report the relative performance of different Krylov solvers on different vendors' GPUs in Figure 10. However, we do not have the vendors' library as a reference. A100 achieves the highest ratio of bandwidth (50%–85%) and the ratio is similar with the SPMV case Figure 9. GEN12 achieves the second highest ratio of bandwidth, and GEN9 and MI100 achieve similar ratios of the devices' bandwidth. GEN12 and A100 achieve a similar ratio of bandwidth in the short recurrences Krylov solvers. GMRES on GEN12 achieves a lower performance than other short recurrences Krylov solvers. This highlights that the kernels of GMRES require specific tuning as the article[1] mentioned.

5.4 | Evaluation of advanced preconditioners

For evaluating the performance of GINKGO 's preconditioners, we select a list of test matrices from the Suite Sparse Matrix Collection, see Table 1 for the matrices' key characteristics. For the preconditioner performance evaluation, we embed the GINKGO 's DPC++ preconditioner kernels into a CG iterative solver. We apply the preconditioned CG to a linear system that complements the test matrices with an all-ones right-hand side. The

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

TABLE 1 Matrix characteristics

	Name	Kind	#rows	#nonzeros
Matrices for solver performance	rajat31	Circuit simulation problem	4,690,002	20,316,253
	atmosmodj	CFD problem	1,270,432	8,814,880
	nlpkkt160	Optimization problem	8,345,600	225,422,112
	thermal2	Thermal problem	1,228,045	8,580,313
	CurlCurl_4	Model reduction problem	2,380,515	26,515,867
	Bump_2911	2D/3D problem	2,911,419	127,729,899
	Cube_Coup_dt0	Structural problem	2,164,760	124,406,070
	StocF-1465	CFD problem	1,465,137	21,005,389
	circuit5M	Circuit simulation problem	5,558,326	59,524,291
	FullChip	Circuit simulation problem	2,987,012	26,621,983
Matrices for preconditioner experiments	finan512	Economic problem	74,752	596,992
	Dubcova3	2D/3D problem	146,689	3,636,643
	thermal1	Thermal problem	82,654	574,458
	ted_B	Thermal problem	10,605	144,579
	Muu	Structural problem	7102	170,134

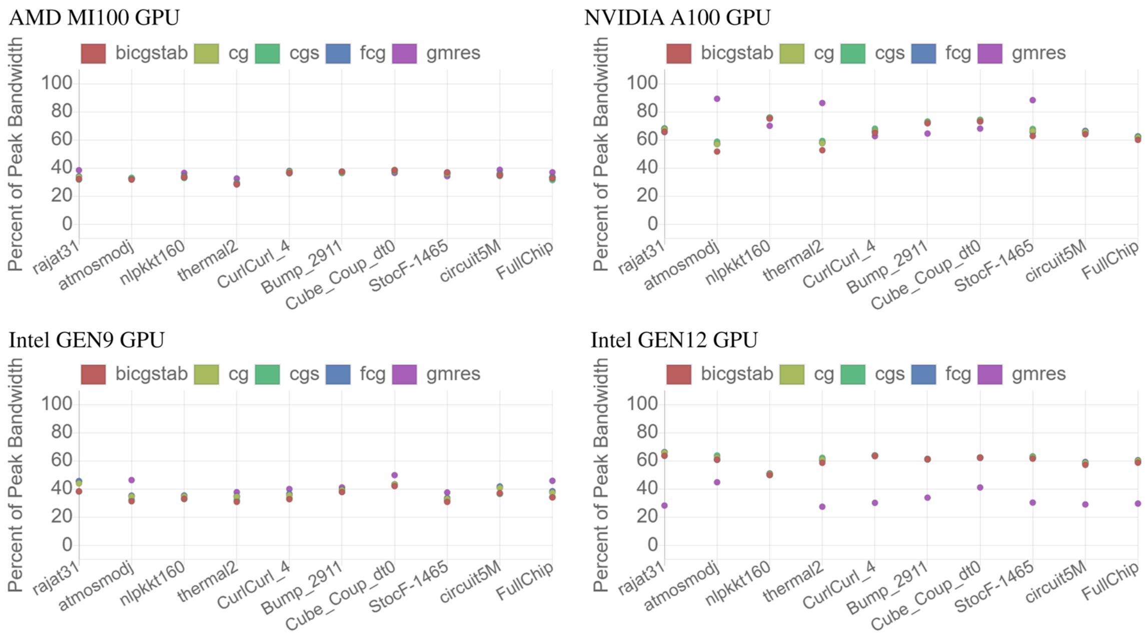

FIGURE 10 Solver performance relative to the hardware bounds on various GPUs.

CG uses an all-zeros initial guess and a relative residual, which is reported by CG, stopping criterion of 10^{-7} while allowing for at most 1000 CG iterations. The preconditioners are executed in the following configurations:

- CG—plain conjugate gradient without any preconditioner.
- Adaptive block-Jacobi PCG—PCG using GINKGO 's adaptive precision block-Jacobi preconditioner with block size 32.

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic

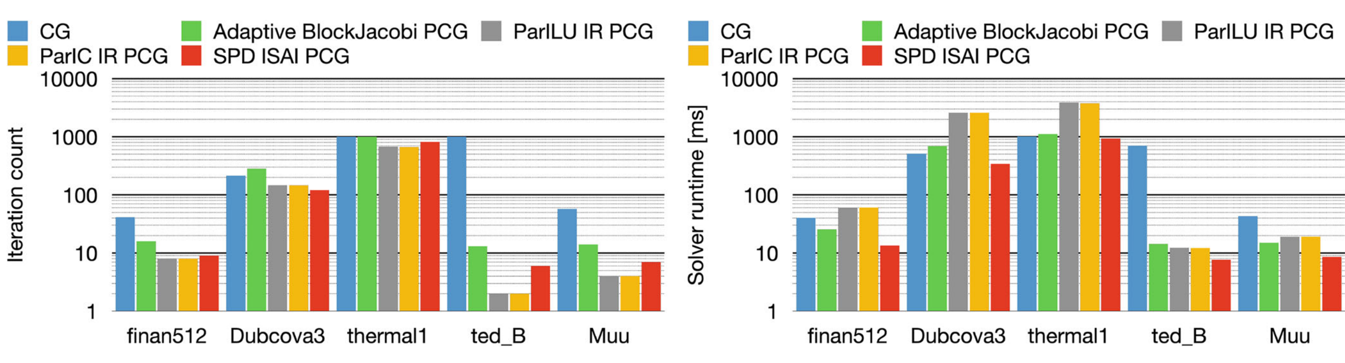

FIGURE 11 Effectiveness and efficiency of GINKGO 's preconditioners on the Intel GEN12 GPU.

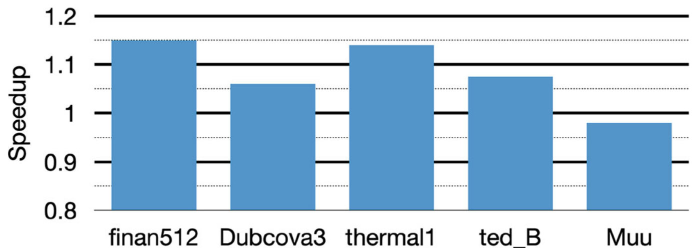

FIGURE 12 Adaptive precision block-Jacobi speedup over the fixed precision block-Jacobi on the Intel GEN9 GPU.

- ParILU IR PCG—PCG using the ParILU preconditioner. The triangular systems are solved using 5 sweeps of IR with adaptive precision block-Jacobi (block size 32) as shown in Listing 1. Except for the block size, this configuration is identical to the configuration used in Goebel et al.[11]
- ParIC IR PCG—Identical to ParILU IR PCG, except for using the symmetric (ParIC) incomplete factorization.
- SPD ISAI PCG—PCG using (symmetric positive definite) incomplete sparse approximate inverse (ISAI) preconditioner.

Figure 11 reveals that a plain CG is not converging within the iteration limit for the thermal1 and ted_B problems, but adding a preconditioner enables convergence. Adaptive precision block-Jacobi PCG provides a significant iteration improvement for the ted_B matrix but it is less effective for the Dubcova3 and thermal1 matrices. ParILU or ParIC generally provide larger convergence improvement, see for example, finan512, thermal1, ted_B, and Muu matrices. The larger convergence improvement of ParILU/ParIC preconditioning comes at higher preconditioner generation and application costs. The time-to-solution comparison in Figure 11 reveals that the ParILU/ParIC preconditioned CG is despite the lowest iteration count often not the fastest choice. In fact, the ISAI-preconditioned CG providing moderate convergence improvement at low cost is the fastest choice for all test problems we consider in this evaluation.

We next evaluate the performance of GINKGO 's adaptive precision block-Jacobi in comparison to the fixed precision block-Jacobi. For this experiment, we use IEEE double precision and run on the GEN9 GPU. We use a relative residual threshold of 10^{-12}, start the iterations with an all-zero initial guess, and solve for an all-one right-hand side. In Figure 12, we visualize the speedup we observe when running on the GEN9 architecture. For the finan512 and thermal1 problems, adaptive precision block-Jacobi executes about 15% faster than fixed-precision block-Jacobi. For the rest matrices, adaptive precision block-Jacobi gives a similar performance with fixed precision. This demonstrates that the DPC++ kernels for the adaptive precision block-Jacobi are working as intended and can offer performance advantages over the fixed precision preconditioning.

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. Acknowledging the oneAPI ecosystem spans from general-purpose CPUs to GPUs, and special function units, we also discuss a strategy that enables the automatic selection of a valid and

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lice

well-performing kernel configuration out of a set of pre-compiled kernels when running on specific DPC++ enabled architecture. Using these porting strategies, we added DPC++ kernels for basic building blocks, Krylov solvers, and advanced preconditioners to the GINKGO library. This makes GINKGO the first math library able to run complete simulation workflows on Intel GPUs. We evaluated the efficiency of the new functionality in terms of translating hardware performance into algorithm performance. Comparing the execution time of basic building blocks with kernels available in Intel's oneMKL library, we demonstrate that GINKGO 's kernels are competitive to the vendor implementation while providing more flexibility. Future research will focus on evaluating GINKGO 's DPC++ on GPUs from other vendors.

ACKNOWLEDGMENTS

This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under Grant VH-NG-1241, and 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. The authors want to acknowledge the access to the Intel® Devcloud early access development platform. Open Access funding enabled and organized by Projekt DEAL.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

Performance data supporting this study findings are openly available in Zenodo at https://doi.org/10.5281/zenodo.6787304.

ENDNOTES

[*] https://spec.oneapi.com/versions/latest/index.html
[†] These extensions are now part of the SYCL 2020 Specification: https://www.khronos.org/news/press/khronos-releases-sycl-2020-final-specification
[‡] https://intel.github.io/llvm-docs/PluginInterface.html
[§] https://spec.oneApi.com/level-zero/latest/core/INTRO.html
[¶] This design choice reflects the expertise of GINKGO developers, but may not be the strategy for future development.
[#] Intel is aware of the community requesting the option of configuring a subgroup inside a kernel, and we may expect that feature in future releases.
[‖] The compiler also limits the number of recursive function generated by template. In our test case, 1024 kernels in the list hit the limitation and lead to a compiler error.
[**] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/compilation/ahead-of-time-compilation.html
[††] https://intel.github.io/llvm-docs/UsersManual.html
[‡‡] https://ark.intel.com/content/www/us/en/ark/products/211013/intel-iris-xe-max-graphics-96-eu.html
[§§] GINKGO is designed to compile for IEEE 754 double-precision, single precision, double precision complex, and single-precision complex arithmetic.
[¶¶] At the point of writing, oneMKL does not provide a COO implementation.

ORCID

Yu-Hsiang M. Tsai https://orcid.org/0000-0001-5229-3739
Terry Cojean https://orcid.org/0000-0002-1560-921X
Hartwig Anzt https://orcid.org/0000-0003-2177-952X

REFERENCES

1. Tsai YM, Cojean T, Anzt H. Porting sparse linear algebra Intel GPUs. Proceedings of the Euro-Par 2021: Parallel Processing Workshops; 2022:57-68; Springer International Publishing, Cham.
2. Anzt H, Cojean T, Chen YC, et al. Ginkgo: a high performance numerical linear algebra library. *J Open Source Softw*. 2020;5(52):2260. doi:10.21105/joss.02260
3. Keryell R, Reyes R, Howes L. Khronos SYCL for OpenCL: a tutorial. *IWOCL '15*. Khronos, Association for Computing Machinery; 2015.
4. Cojean T, Tsai YHM, Anzt H. Ginkgo – A math library designed for platform portability. preprint; 2020.
5. Tsai YM, Cojean T, Ribizel T, Anzt H. Preparing Ginkgo for AMD GPUs - a testimonial on porting CUDA code to HIP. Proceedings of the Euro-Par 2020: Parallel Processing Workshops; Vol. 12480, 2020:109-121.
6. Flegar G, Anzt H, Cojean T, Quintana-Ortí ES. Adaptive precision block-jacobi for high performance preconditioning in the ginkgo linear algebra software. *ACM Trans Math Softw*. 2021;47(2). doi:10.1145/3441850
7. Chow E, Patel A. Fine-grained parallel incomplete LU factorization. *SIAM J Sci Comput*. 2015;37(2):C169-C193.
8. Anzt H, Chow E, Dongarra J. ParILUT—A new parallel threshold ILU factorization. *SIAM J Sci Comput*. 2018;40(4):C503-C519.
9. Anzt H, Ribizel T, Flegar G, Chow E, Dongarra J. ParILUT - A parallel threshold ILU for GPUs. Proceedings of the 33rd International Parallel and Distributed Processing Symposium (IPDPS 2019); 2019; IEEE. http://bit.ly/ParILUTGPU.
10. Anzt H, Chow E, Huckle T, Dongarra J. Batched generation of incomplete sparse approximate inverses on GPUs. *ScalA'16*. IEEE; 2016:49-56.
11. Goebel F, Anzt H, Cojean T, Flegar G, Quintana-Ortí ES. Multiprecision block-Jacobi for iterative triangular solves. Proceedings of the European Conference on Parallel Processing; 2020:546-560; Springer.
12. Hoberock J, Bell N. Thrust: a parallel template library. Version 1.15.0. Online, accessed in December 2021; 2010.
13. Tsai YH, Cojean T, Anzt H. Dataset for providing performance portable numerics for Intel GPUs; 2022. 10.5281/zenodo.6787304

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

14. Deakin T, Price J, Martineau M, McIntosh-Smith S. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *Int J Comput Sci Eng*. 2017;17:247-262.
15. Konstantinidis E, Cotronis Y. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J Parallel Distrib Comput*. 2017;107:37-56. doi:10.1016/j.jpdc.2017.04.002
16. Saad Y, Schultz MH. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comput*. 1986;7:856-869.
17. Davis TA, Hu Y. The university of Florida sparse matrix collection. *ACM Trans Math Softw (TOMS)*. 2011;38(1):1-25.

How to cite this article: Tsai Y-HM, Cojean T, Anzt H. Providing performance portable numerics for Intel GPUs. *Concurrency Computat Pract Exper*. 2022;e7400. doi: 10.1002/cpe.7400

20634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7400 by University Of Tennessee, Knoxville, Wiley Online Library on [07/11/2022]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons Lic