Benchmark DPC++ code and performance portability on heterogeneous architectures

Nenad Mijić*, Davor Davidović*
* Centre for Informatics and Computing, Ruđer Bošković Institute, Zagreb, Croatia
{nenad.mijic, davor.davidovic}@irb.hr

***Abstract*—Source code portability is becoming increasingly important in the development of new solutions in HPC due to the wide diversification of hardware and heterogeneity of systems. With Intel's oneAPI suite of programming tools and the Data Parallel C++ compiler, a single source code containing both host and device code can leverage hardware architectures from different vendors. Using the compiler's interoperability, it can be linked to existing libraries such as MPI to run the program on a distributed memory system.**

In this paper we benchmark and analyze the performance that can be achieved with the Intel DPC++ compiler, using the distributed Cholesky QR2 algorithm as an example and comparing it with the native CUDA and C++ implementation. The analysis shows that the performance degradation when using SYCL is negligible when a smaller number of nodes are used, but with the cost that some additional self-made optimizations are required in SYCL code.

***Keywords*—code portability, DPC++, GPU, MPI, HPC, Choleksy QR2**

I. Introduction

HPC is currently based on extremely heterogeneous systems, and there is no reason to believe that this situation will change in the near future. The most powerful systems in the TOP500 [1] supercomputers are dominated by such systems, and it is enough to look at the list to see the diversity of combinations of CPUs and accelerators. Intel, AMD and Arm dominate in the field of central processing units, while NVIDIA had a leading role for a decade in the domain of GPU accelerators, and more recently AMD has stepped forward by shipping systems with its GPU graphic processors. AMD has taken the first step forward by delivering the first exascale systems. In addition, thanks to the rise of artificial intelligence, specialised accelerators for AI are being developed and appearing on the scene. For developers and researchers, writing portable code that can run on a number of these architectures and maintain performance is a real challenge.

The diversity of hardware brings along a variety of programming models that achieve high performance using different approaches. Low-level programming models for heterogeneous systems are often completely vendor locked and therefore do not provide portability. The low-level architectural details are exposed directly by the programming language that can exploit them explicitly, often resulting in higher performance. For example, CUDA is exclusive to NVIDIA GPUs while ROCm is exclusive to AMD GPUs. The HIP programming model [2], provided by AMD, is an abstraction layer for CUDA and ROCm, targeting both GPU vendors. Opposite from previously described low-level programming models, OpenCL stands out as a portable solution but relies on the vendor to provide and maintain the backend for runtime execution.

The models based on directives provide direct portability by using high-level *pragma* directives to instruct the compiler to use different architectural implementations for exposing parallelism and provide solutions for offloading computation to the accelerator. They also handle memory management and data transfer between the host and accelerator. Notable examples include OpenMP, which was originally developed for multicore architectures but now supports offloading to the accelerator, and OpenACC [3]. All hardware vendors provide libraries and codes tuned for their respective hardware.

High-level programming models are often abstraction layers that enable the development of cross-platform portability, preferably from a single source code. They are intended as C++ portable APIs mapped to the specific backend for providing portability and features are often overlapped among these models. Such models take advantage of different libraries, directive programming approaches, and other languages to enable portability and leverage hardware specifics providing abstractions for parallel execution and data structures. The best known examples are Kokkos [4], [5], Raja [6], and SYCL [7] targeting shared memory systems. Kokkos and Raja include OpenMP, CUDA, Hip and SYCL backends. Celerity [8] extends SYCL with MPI to distributed memory systems and implements a task-based runtime system for distributed memory. Celerity creates task-based graphs and designates a master node to execute the schedule graph for the entire cluster and oversees how data and compute load is distributed.

This paper presents the code and performance portability of oneAPI solutions with respect to direct native implementation on distributed HPC systems. MPI is used for parallelization on a distributed memory system. The original contributions of this research are as follows:

- achieved single source code portability across different hardware architectures (CPUs and GPUs from different vendors),
- achieved performance portability for simple linear algebra algorithms,
- implements inter-node communication library for distributed memory systems,

- code benchmarked and analyzed on different architectures: AMD CPUs, Intel CPUs and NVIDIA GPU,
- detected several performance issues in Intel DPC++ compiler and libraries, and workarounds on how to resolve them were proposed.

The paper is divided into the following logical order. Section II presents related benchmarks for linear algebra algorithms implemented using the SYCL programming model. The distributed CholeslyQR2 algorithm used for benchmarking is briefly described in Section III with an overview of its implementation. Numerical results and achieved performance portability are presented in section IV, and the paper concludes with comments and an outlook on future work in section V.

II. SYCL and Related work

SYCL is an open standard published by the Khronos group [7] to provide a unified programming model for all heterogeneous architectures. The model is based on modern C++ with a single-source code approach that includes both host and device parts. Parallel part of code is scheduled and executed with kernels object written with lambda functions, offering several choices for parallel construction. During compilation, kernels intended to run on accelerators are extracted from code, compiled with a compiler suitable for that architecture and embedded in a single executable. It was developed with the intention of providing a plugin interface for various programming models. It is assumed that in the future the plugin interface will allow easy integration and support of new programming models that have not yet been announced and thus provide long-term HPC support. The SYCL runtime manages task scheduling, task dependencies and synchronisation by creating task graphs through knowledge of memory access while also being responsible for memory and data management.

The interoperability of SYCL with built-in *host_task* kernels allows the use of a common, unifying interface for accessing non-SYCL libraries. Sycl has been implemented by the following entities among others: Codeplay's ComputeCpp, hipSYCL [9] and Intel's Data Parallel C++ (DPC++) [10]. ComputeCpp supports any OpenCL platform that accepts SPIR/SPIR-V, and additionally NVIDIA by using the PTX backend from LLVM. hipSYCL is an open-source implementation led by the University of Heidelberg based on existing compiler toolchains for heterogeneous computing with Clang plugin via HIP or CUDA toolchain for GPUs, Intel Level Zero for Intels GPUs and OpenMP for CPUs. DPC++ is an open-source implementation based on the LLVM compiler that adopts the SYCL 2020 standard. Codeplay contributed to the Intel project by providing an interface to CUDA that allows direct programming for the NVIDIA GPUs. Lately, a beta version interface for AMD GPU has been announced.

The SYCL standard provides 2 containers for controlling data during code execution. A buffer is a memory object responsible for allocating or controlling the allocated memory on the host or device and implicitly takes care of transferring the memory between the devices and the host during the computation. Unified Shared Memory (USM) provides explicit control of memory allocation and transfer with traditional pointers. USM distinguishes two memory containers, one for the device and one for the host. Buffers provide modern C++ style memory management, while USM provides C style with fine-grained controls that give developers precise control over memory management, which can lead to better-optimized memory transfers.

Code portability has been tested in recent years due to various occurring computing architectures, especially in the area of high-performance computing. Most programming models that guarantee portability work well on different platforms, but often show unsatisfactory performance portability. Extensive testings of various programming models conducted in ([11]) showed at least good code portability but have performance problems with Kokkos and OpenMP models.

First benchmark suite [12] written in SYCL has 71 tests. To test all the capabilities of SYCL, the authors divided the benchmark into microbenchmarks, runtime tests, and application tests with ComputeCPP and hipSYCL. The microbenchmarks tested the performance of code on GPU devices targeting arithmetic or the memory subsystem. Application tested were mainly linear algebra algorithms *gemm*, *syrk*, *syr2k*, Gram-Schmidt *QR*, and *axpy*. For testing the runtime performance the benchmark includes tests that explicitly create complex inter-task dependencies. They showed close to the peak performance of microbenchmarks and divergent results for other tests with different toolchains. The paper [13] extended the work of SYCL-bench to 4 toolchain implementations showing the difference in performance between them but indicating somewhat problems within runtime systems targeting devices with non-optimal configuration.

In [14], the authors studied the performance of kernels from the RAJA performance suite written in the SYCL standard using the hipSYCL toolchain. The authors observed the competitive performance of their implementation against the native CUDA implementation.

To test the performance portability on distributed systems, the authors of Celerity tested the runtime system with a synthetic benchmark to evaluate performance portability. They observed negligible or slightly worse performance compared to the SYCL+MPI implementation.

III. Benchmark algorithm

One of the fundamental problems of linear algebra is the QR factorization, which decomposes the matrix A with linearly independent columns into an orthogonal matrix $Q \in \mathbb{R}^{m\times n}$ and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$. The importance of QR factorization is that most algorithms are based on it, such as solvers for linear systems, least squares solvers, eigenvalue and singular value solvers.

Numerically, the most stable algorithm for computing QR factorization is the Householder algorithm, which orthogonalizes the matrix column-wise by applying a series of Householder reflectors. Since the algorithm is mostly vector-oriented, the majority of computational time is spent on memory-bound BLAS2 operations (matrix-vector operations) which cannot take advantage of available hardware capabilities such as complex memory hierarchies. For tall and skinny matrices, that have significantly more rows than columns ($m \gg n$), this algorithm becomes even more challenging to achieve high performance.

The solution is the CholeskyQR [15] algorithm that avoids the described problems when computing the QR factorization of tall and skinny matrices and is based on Cholesky factorization. For input matrix A, the CholeskyQR algorithm first computes the Gram matrix $G = A^T A \in \mathbb{R}^{n \times n}$ which is a positive definite matrix. The triangular factor R (from QR decomposition) is obtained as an upper triangular factor of the Cholesky factorization of the matrix G. An orthogonal matrix Q is constructed by solving the linear systems $Q = AR^{-1}$. However, the CholeskyQR algorithm is numerically unstable, especially for ill-conditioned matrices. By repeating the CholeskyQR twice, the orthogonality of Q can be significantly improved, and the new algorithm is called CholeskyQR2. Numerical stability is parameterized with condition number of input matrix $\kappa(A) = \sigma_{max}(A)/\sigma_{min}(A)$ where σ are the smallest and the largest singular values of the matrix. The algorithm becomes numerically unstable for condition number around 10^8, after which Cholesky decomposition breaks down. The distributed algorithm divides the matrix into block rows, distributes the blocks to processes, and performs the computations independently except for a single collective routine (*Allgather*) called during the construction of the Gram matrix. This makes the algorithm communication-avoiding and very easy to parallelize on distributed systems. The total number of operations is $4mn^2 + n^3$ for non distributed version and $4mn^2/P + n^3$ per process for the distributed version where P is the total number of processes.

The CholeskyQR2 is implemented in both DPC++ and CUDA C/C++ with algebra routines called from appropriate external highly tuned libraries for specific computation architectures. We chose DPC++ because of the widespread use of Intel's solutions, that currently allows code to run on all architectures available to us. We emphasize that hipSYCL also supports the same architectures, but for simplicity, we have chosen DPC++. For the CUDA version, the libraries used were cuBLAS and cuSOLVER, and for DPC++, the routines are called from oneAPI open-source MKL Interface library [16]. This library is an implementation of oneMKL Data Parallel C++, following the oneMKL specification, and is a part of Intel's new oneAPI framework. oneAPI consists of languages and libraries for easy code portability between Intel's hardware, with the idea of providing uniform and consistent routine calls (API). The oneMKL interface library follows the same routine conventions and includes interfaces for multiple backends that target underlying libraries, such as oneMKL, cuBLAS, rocBLAS for BLAS functionality and oneMKL, cuSOLVER, rocSOLVER for LAPACK functionality. Listing 1 shows Cholesky routine call from oneMKL interface library.

Listing 1 Dpcpp Cholesky routine call.

```
namespace oneapi::mkl::lapack {
  void potrf(sycl::queue &queue,
             oneapi::mkl::uplo upper_lower,
             std::int64_t n,
             sycl::buffer<T,1> &a,
             std::int64_t lda,
             sycl::buffer<T,1> &scratchpad,
             std::int64_t scratchpad_size)
}
```

Message Passing Interface (MPI) is used for intra-node and inter-node parallelization. By using CUDA-aware MPI and passing pointers to CUDA memory, it is easy to bypass multiple staging of GPU buffers through host memory and back to the device, leaving only memory communication between MPI ranks. In addition, NVIDIA Collective Communication Library (NCCL) is included in place of CUDA-aware MPI for collective routines, as its primitives are optimized for collective communication between distributed NVIDIA GPUs.

For GPU versions, in the native CUDA, cuBLAS and cuSOLVER handles to cuBLAS and cuSOLVER context are initialize before the algorithm calls. These handles are responsible for resource allocation and used for dispatching the computation to selected device. In the oneMKL interface library, the same handles are created on the first calls that require them, without the possibility of explicit creation, resulting in a huge timing overhead. Accordingly, in SYCL codes, simple trivial functions are executed before the algorithm functions to avoid this overhead.

Here we describe implementations of SYCL that use NCCL primitives optimized for NVIDIA GPUs for collective communication. When NVIDIA NCCL communicator is used instead of MPI, NCCL uses the primary CUDA context. The same context is used for all other NCCL function calls. CUDA context is an object used to manage the CUDA graphics coprocessor, encapsulates resources and computation calls that are executed on it. The context is unique to each device, so a kernel from one CUDA context cannot run concurrently with a kernel from another CUDA context without context switching and large overhead. The first SYCL call creates a new context for later use during runtime. The possible overhead of context switching results in performance degradation.

To ensure that both NCCL and SYCL use the same context, the SYCL context is created with the `use_primary_context` property to force SYCL to use the same primary context as NCCL. In addition, we created a runtime interface library for communication that wraps MPI or NCCL calls depending on the desired backend. In this way, we reduced the multiple communications functions to a single function call.

IV. Results

In this section, the code and performance portability of the Intel oneAPI (DPC++) and SYCL environments were benchmarked and analyzed with the CholeskyQR2 algorithm. The idea was to test and analyze the Intel implementation of the SYCL standard on different CPU and GPU architectures and compare its performance with native implementations of the same code using standard C/C++ and CUDA compilers.

The benchmarks and analysis were performed on two testbed computing systems. The first is a local machine called Orthus located at the Ruđer Bošković Institute, equipped with 2x Intel Xeon Gold 6240R processors with a total of 48 cores (hyperthreading is disabled) and four NVIDIA A100 GPUs. The second is a GPU partition of the VEGA supercomputer at the Institute of Information Science (IZUM). Each Vega node consists of two 64-core AMD 7H12 CPUs and four NVIDIA A100 GPUs. All tests were performed in double- precision arithmetic.

DPC++ (and SYCL in general) can target different backends at runtime by using the device selector class to select a device, or by using the environment variable *SYCL_DEVICE _FILTER* to restrict the choice of devices available to selectors. In this way, it is easy to switch between the CPU and the GPU versions from a single SYCL executable compiled from a single source code. In the rest of the paper, the different variants of the CholeskyQR2 algorithm are named as follows (with the executing device in the brackets):

- ***Native C++*** - the code implementation in C++ with the calls to MKL library (CPU),
- ***Native CUDA*** - the code implementation in CUDA (GPU),
- ***SYCL buffer*** - SYCL (DPC++) code using *buffer* memory objects (CPU or GPU),
- ***SYCL usm*** - SYCL (DPC++) code using *usm* memory objects (CPU or GPU).

The *CUDA native* implementation has been written by calling the cuBLAS and cuSOLVER libraries directly. The SYCL buffer and SYCL usm versions denote versions implemented with the DPC++ compiler using the oneMKL interface library. Either the MPI or the NCCL library is used for communication.

The test matrix used in all benchmarks is a tall and skinny matrix of size 300000 × 3000, sufficiently large to simulate the data of a real problem, and a condition number 10^5 which guarantees the numerical stability of the algorithm without breakdown. The data (blocks of rows) are distributed among MPI processes. Without losing generality, the number of MPI ranks per node is set to 4 and one MPI rank is assigned to a single GPU, where the total number of MPI ranks per node equal to the number of devices per node.

To capture the possible overheads of the SYCL implementation, the entire SYCL function calls are included in the timing. Since SYCL implicitly transfers memory the first time a memory object is called, it is hard to avoid timing functions that also contain memory transfers. To allow a fair comparison of the codes, the corresponding memory transfers in the native codes are also included in the timings.

TABLE I: Numerical accuracy of the CPU code on Vega.

#MPI	Native C++		SYCL buffer		SYCL usm	
	ort	res	ort	res	ort	res
1	1.6e-14	2.5e-15	1.6e-14	2.5e-15	1.6e-14	2.5e-15
2	1.5e-14	2.5e-15	1.5e-14	2.5e-15	1.5e-14	2.5e-15
3	1.4e-14	2.5e-15	1.4e-14	2.5e-15	1.4e-14	2.5e-15
4	1.4e-14	2.5e-15	1.4e-14	2.5e-15	1.4e-14	2.5e-15

TABLE II: Numerical accuracy of the CPU code on Orthus.

#MPI	Native C++		SYCL buffer		SYCL usm	
	ort	res	ort	res	ort	res
1	9.1e-16	1.1e-15	9.1e-16	1.1e-15	7.4e-16	1.1e-15
2	7.9e-16	1.1e-15	7.9e-16	1.1e-15	7.9e-16	1.1e-15
3	3.8e-16	1.0e-15	3.8e-16	1.0e-15	3.8e-16	1.0e-15
4	3.3e-16	1.0e-15	3.3e-16	1.0e-15	3.2e-16	1.2e-15

A. Code portability

The code portability is tested by verifying the numerical accuracy of the calculated Q and R factors. The obtained accuracy of the SYCL-based codes should not be worse than that of their native counterparts. Orthogonality of obtained orthogonal matrix Q is computed as $||Q^TQ - I||_F/\sqrt{n}$ and residuals as $||A - QR||_F/||A||_F$.

Tables I II show the obtained orthogonality (*ort*) and residual (*res*) of the computed QR factorization of the test matrix on a single node. The tables show no decrease in numerical accuracy between the native implementation and the SYCL versions. The slight difference in accuracy between the testbed systems is the result of the different versions of SYCL and oneMKL (MKL) thath were used. The slight differences between the numerical results of the two testbed systems are due to the MKL libraries being optimized for different CPUs vendors (Orthus has Intel CPUs, while VEGA has AMD CPUs). Furthermore, there is no decrease in numerical accuracy when the number of MPI processes is increased.

The same behavior is observed for the GPU versions in the tables III IV. In the same system, the numerical results between the versions are identical and differences are observed only between the systems, which is to be expected.

Performance portability

The performance portability of DPC++ versions to native implementations in low-level programming models is investigated by analyzing the execution time of the algorithm. The execution time is divided into two parts, the computation part and the communication part. The

TABLE III: Numerical accuracy of the GPU code on Vega.

#MPI	Native CUDA				SYCL buffer				SYCL usm			
	MPI		NCCL		MPI		NCCL		MPI		NCCL	
	ort	res	ort	res	ort	res	ort	res	ort	res	ort	res
1	1.6e-14	2.7e-15	1.6e-14	2.7e-15	1.6e-14	2.7e-15	1.6e-14	2.7e-15	1.6e-14	2.7e-15	1.6e-14	2.7e-15
2	1.5e-14	2.7e-15	1.5e-14	2.7e-15	1.5e-14	2.7e-15	1.5e-14	2.7e-15	1.5e-14	2.7e-15	1.5e-14	2.7e-15
3	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15
4	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15	1.4e-14	2.7e-15

TABLE IV: Numerical accuracy of the GPU code on Orthus.

#MPI	Native CUDA				SYCL buffer				SYCL usm			
	MPI		NCCL		MPI		NCCL		MPI		NCCL	
	ort	res	ort	res	ort	res	ort	res	ort	res	ort	res
1	1.3e-14	2.3e-15	1.3e-14	2.3e-15	1.3e-14	2.3e-15	1.3e-14	2.3e-15	1.3e-14	2.3e-15	1.3e-14	2.3e-15
2	7.1e-15	2.3e-15	7.1e-15	2.3e-15	7.1e-15	2.3e-15	7.1e-15	2.3e-15	7.1e-15	2.3e-15	7.1e-15	2.3e-15
3	5.3e-15	2.3e-15	5.4e-15	2.3e-15	5.3e-15	2.3e-15	5.4e-15	2.3e-15	5.3e-15	2.3e-15	5.4e-15	2.3e-15
4	3.4e-15	2.3e-15	3.5e-15	2.3e-15	3.4e-15	2.3e-15	3.5e-15	2.3e-15	3.4e-15	2.3e-15	3.5e-15	2.3e-15

computation part, as mentioned earlier, includes the memory transfer to the device and could show the overhead introduced by SYCL. In terms of performance both CPU and GPU versions, perform the same number of floating point operations and a shorter execution time means better performance.

Performance portability is also investigated by analyzing the performance behaviour with strong scaling on the Vega HPC system using the same matrix. The results are plotted on a graph with execution time. For both the CPU and GPU versions, the test configuration is set up with 32 threads per MPI rank (process) using a range of processes up to 32. The results presented are based on the average of 10 benchmark runs for each configuration.

Figure 1 shows the execution time for all 3 CPU versions. As expected, the native implementation (*Native C++*) achieves the best performance with the shortest execution time. *SYCL* versions prove to be comparable to the native implementation. In most tests, *SYCL buffer* shows better times than the *SYCL usm* version, even though memory transfers are explicitly handled with pointers. The worst performance is observed for *SYCL usm* with 1 MPI rank, which has significantly more execution time than the other versions, which can be explained by the SYCL overhead.

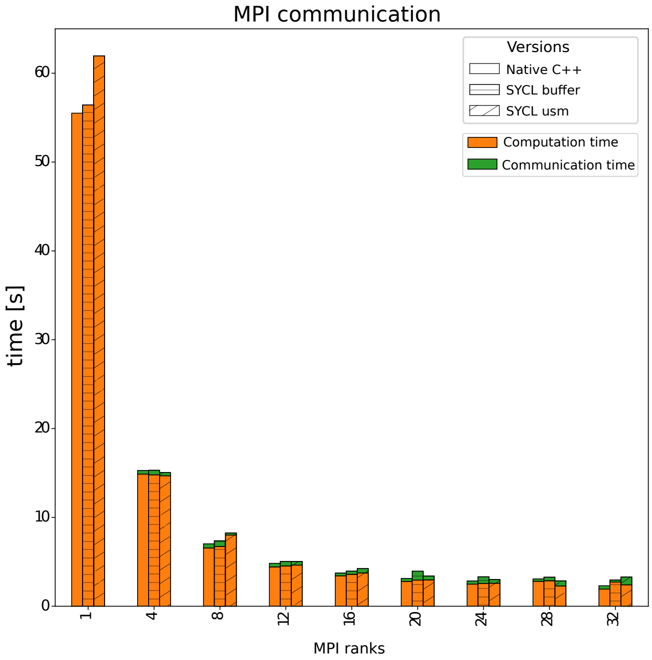


Fig. 1: Time of execution of CPU version on Vega.

Finally, the portability of performance to GPU devices, as shown in Figure 2, shows consistent performance increase (i.e. decrease of the execution time) as the number of processes (and GPUs) increases. As expected, NCCL is able to outperform MPI in terms of communication time for any number of processes. The native version shows better overall performance and takes less time to compute for both communication libraries, which means that the *SYCL* introduces a slight overhead in the memory transfers. The communication time varies considerably and we assume that the results are caused by the test system (VEGA) itself and not by the additional overhead of *SYCL*. In addition to the last arguments, testing on the Orthus system with 4 GPUs shows more consistent communication times (table V) in contrast to the VEGA system, where the standard deviation for 4 MPI ranks is larger than the average, showing a large discrepancy in the values. When testing with a single process, communication should be negligible, but significant communication time is observed for the *SYCL buffer* with NCCL and all versions with MPI.

TABLE V: Communication time of GPU versions on Orthus.

#MPI	SYCL buffer	
	MPI	NCCL
4	$(810 \pm 56)ms$	$(26 \pm 5)ms$
	SYCL usm	
	MPI	NCCl
4	$(786 \pm 74)ms$	$(31 \pm 12)ms$
	Native CUDA	
	MPI	NCCL
4	$(876 \pm 40)ms$	$(40 \pm 17)ms$

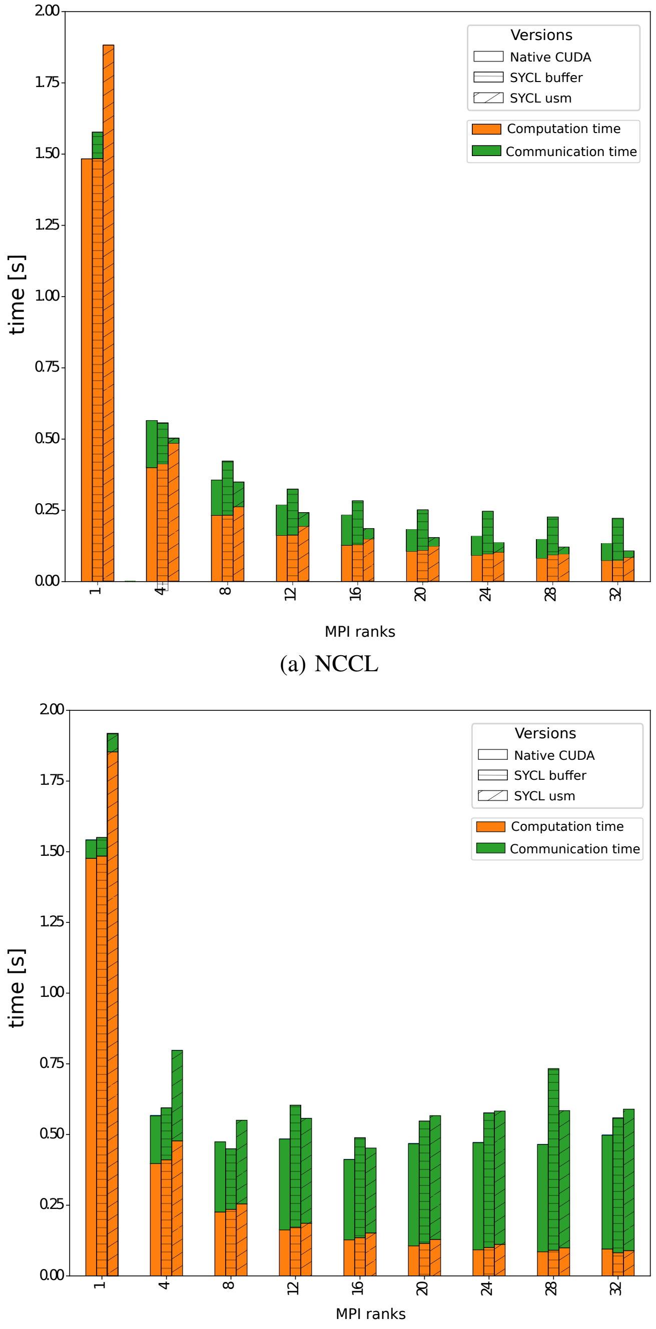
V. Conclusion

In this work, we tested Intel oneAPI solutions and their interoperability with current linear algebra and communication libraries to achieve code portability and performance portability for distributed algorithm. The benchmark code was the CholeskyQR2 algorithm. We showed that oneAPI solutions can be used for code portability, focusing on a single source code running on different architectures to simplify code maintenance and reduce development time. Code performance are also achieved with nearly the same performance as native solutions for a small number of nodes. The DPC++ language is maturing rapidly, as evidenced by the development of the compiler and new hardware support. The major drawback is the lack of and outdated documentation, but the open source code provides the opportunity to contribute to development, as evidenced by the large amount of support from the open source community and third parties. Future work would include other SYCL toolchains and extend the architectures used.

In addition, two issues were identified during implementation and testing. When using cuSOLVER, a new CUDA context is created each time a cuSOLVER function or kernel is called, resulting in significant overhead. We have implemented a solution by reusing solutions from the cuBLAS backend of the oneMKL interface library. The technical details of the improvement are beyond the scope of this paper and are not described here. The second problem was the reuse of the CUDA context created by NCCL for SYCL. The solution was described in the paper and a subsection is dedicated to it.

Acknowledgment

This work was supported by the Croatian Science Foundation under grant number HRZZ-UIP-2020-02-4559. The authors gratefully acknowledge the HPC RIVR consortium (www.hpc-rivr.si) and EuroHPC JU (eurohpc-ju.europa.eu) for funding this research by providing computing resources of the HPC system Vega at the Institute of Information Science (www.izum.si).



(a) NCCL

(b) MPI

Fig. 2: Time of execution on Vega using NCCL (2a) and MPI (2b) communication libraries.

References

[1] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, “TOP500.” [Online]. Available: https://www.top500.org/

[2] AMD, “Introduction to HIP Programming Guide.” [Online]. Available: https://docs.amd.com/bundle/HIP-Programming-Guide-v5.3/page/Introduction_to_HIP_Programming_Guide.html

[3] OpenACC, “OpenACC.” [Online]. Available: https://www.openacc.org/

[4] H. Carter Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, jul 2014.

[5] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang,

N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, apr 2022.

[6] D. A. Beckingsale, T. R. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryujin, "RAJA: Portable Performance for Large-Scale Scientific Applications," *Proceedings of P3HPC 2019: International Workshop on Performance, Portability and Productivity in HPC - Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 71–81, nov 2019.

[7] Khronos, "SYCL™ 2020 Specification (revision 6) | Enhanced Reader." [Online]. Available: https://registry.khronos.org/SYCL/

[8] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-Level C++ for Accelerator Clusters," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11725 LNCS, pp. 291–303, 2019. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-29400-7_21

[9] A. Alpay, T. Applencourt, G. Brown, R. Keryell, and G. Lueck, "Using interoperability mode in SYCL 2020," pp. 1–1, may 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3529538.3529997

[10] Intel Corporation, "Data Parallel C++ Language." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html

[11] T. Deakin, S. Mcintosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability across Diverse Computer Architectures," 2019. [Online]. Available: https://doep3meeting2019.lbl.gov/agenda

[12] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing." [Online]. Available: https://github.com/bcosenza/sycl-bench

[13] B. Johnston, J. S. Vetter, and J. Milthorpe, "Evaluating the Performance Portability of Contemporary SYCL Implementations."

[14] B. Homerding and J. Tramm, "Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs," 2020. [Online]. Available: https://doi.org/10.1145/nnnnnnn.nnnnnnn

[15] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto, "CholeskyQR2: A Simple and Communication-Avoiding Algorithm for Computing a Tall-Skinny QR Factorization on a Large-Scale Parallel System," *Proceedings of ScalA 2014: 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - held in conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 31–38, 2014.

[16] M. Krainiuk, M. Goli, and V. R. Pascuzzi, "OneAPI Open-Source Math Library Interface," *Proceedings of P3HPC 2021: International Workshop on Performance, Portability and Productivity in HPC, Held in conjunction with SC 2021: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 22–32, 2021.