Extending MAGMA Portability with OneAPI

Anna Fortenberry
Department of Computer Science and Engineering
University of North Texas
Denton, USA
AnnaFortenberry@my.unt.edu

Stanimire Tomov
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, USA
tomov@icl.utk.edu

***Abstract*—As the architectures of super-computing systems are continually changing, it is important to maintain efficient code portability in order to continue to take advantage of the computing capabilities of the diverse and evolving hardware in these systems. Intel has adopted an open standard programming interface for heterogeneous systems called oneAPI, designed to allow code portability across different processor architectures. This paper evaluates Intel's oneAPI by migrating a general matrix-matrix multiplication (GEMM) CUDA algorithm from the dense linear algebra library Matrix Algebra on GPU and Multicore Architectures (MAGMA) to Data Parallel C++ (DPC++), the direct programming language of oneAPI. The DPC++ Compatibility Tool (DPCT) in Intel's oneAPI was used successfully for an initial port of MAGMA to DPC++. The performance of the migrated code is evaluated and compared to OpenMP GEMMs and state-of-the-art Intel MKL implementations on AMD EPYC 7742 multicore CPUs and Intel Xeon CPU E5-2698 V4 multicore CPUs, to the original native-CUDA code in MAGMA on NVIDIA GeForce RTX 3060 discrete GPUs, and to oneMKL on Intel UHD Graphics P630 [0x3e96] integrated GPUs. The initial migrated code demonstrates impressive performance on multicore CPUs as it significantly outperforms reference OpenMP implementations, and even MKL on AMD CPUs. Performance on Nvidia GPUs is also very surprising as the DPC++ code matches in performance the native CUDA code. The initial migrated code performed poorly on the Intel GPU, as expected, because the Intel GPU architecture used is quite different than the Nvidia GPU architecture for which the original code was designed. However, using the MAGMA's parameterized implementations to tune the GEMM algorithm to better match the Intel GPU architecture, improved the performance significantly. Intel's oneAPI allowed for a successful extension of MAGMA's functional and performance portability to multicore CPUs and Intel GPUs.**

***Index Terms*—oneAPI, DPC++, portability, MAGMA**

I. Introduction

Supercomputers are used to solve today's most challenging problems. Data science, applied mathematics, high performance computing, and quantum information science are all examples of domains in which supercomputers play a critical role [1], [2]. Researchers are able to analyze complex systems that would otherwise be impossible [3]. The architectures of supercomputing systems are becoming increasingly diverse in heterogeneous architectures. This started with the invention of the NVIDIA GPU in 1999 [4]. In 2012, a supercomputer called Titan was the first successful heterogeneous system, designed by combining AMD Opteron 6274 16-core Central Processing Units (CPUs) and NVIDIA Tesla K20X Graphics Processing Units (GPUs) [5]. NVIDIA GPUs progressed to become integral to several of the top supercomputers. Recently, new vendors have entered the supercomputing GPU domain. The TOP500 list shows that in June 2019, five of the top ten supercomputers used GPUs, all from NVIDIA [6]. As of June 2022, eight of the top ten use accelerators, including four from NVIDIA, three from AMD, and one from NUDT [7]. Further, with the announcement of Aurora, Intel plans to join as a supercomputer GPU vendor [8]. This illustrates how supercomputers are becoming increasingly diverse in both heterogeneous architecture types and designs. The effect of this is a decrease in interoperability between code and hardware. To take advantage of heterogeneous architectures without requiring significant code modifications, architecture-independent programming models are needed. Intel adopts such a programming model interface called oneAPI [22] with the claim that it is the solution to this problem. Intel states that applications which adopt this model gain portability to all supported hardware, including scalar, vector, spatial, and matrix architectures such as CPUs, GPUs, Field-Programmable Gate Arrays (FPGAs), and other accelerators. To encourage the use of oneAPI, Intel developed tools to aid in the migration of CUDA, the language for porting to NVIDIA GPUs, to Data Parallel C++ (DPC++), the direct programming language of oneAPI [16]. Specifically, the DPC++ Compatibility Tool (DPCT) [19] claims to migrate CUDA to DPC++ very effectively. This paper describes the efforts on testing the oneAPI model by porting the dense linear algebra library called Matrix Algebra on GPU and Multicore Architectures (MAGMA) [11] to Sycl/DPC++. The capability of the DPCT tool to migrate code successfully is tested first. It is then evaluated whether the process of using DPCT is more efficient than manual translation. This is followed by testing the value of the translated code. Performance is analyzed using MAGMA's single-precision general matrix-matrix multiplication (SGEMM) algorithms, translated to DPC++. Hardware tested includes two CPUs, the AMD EPYC 7742 64-Core Processor @ 2.25GHz and the Intel® Xeon® CPU E5-2698 V4 20-Core Processor @ 2.20GHZ, and two GPUs, the NVIDIA GeForce RTX 3060, which is a discrete GPU, and the Intel UHD Graphics P630 [0x3e96], which is integrated.

The rest of this paper is as follows. A related works section details recent evaluations of various heterogeneous

Research funded by NSF through RECSEM REU program at UTK 2022

programming models and a benchmark evaluation of oneAPI. A background section explains the structure of CUDA in MAGMA, overviews the design of oneAPI, and lists the specifications of the hardware mentioned above. A methodology overviews the approach to extending support to oneAPI in MAGMA. The next section discusses the GEMM algorithm design and parameters for testing performance. In the section that follows, performance results are provided; required test parameters are listed first, followed by hardware usage results and performance comparisons between original and migrated code. Lastly, a conclusion and future directions section summarizes the contributions of this paper and explains the areas where further work is required.

II. Related Works

Intel's oneAPI is relatively new, so few substantial papers related to migration of applications to oneAPI have been published. One case study [12] tests several portable programming models, including Kokkos and SYCL, with the latter related to oneAPI. The main component of oneAPI is the DPC++ language, which is an implementation of SYCL [23]. These models were tested with a parallel Milc-Dslash implementation, which is a benchmark involving millions of matrix-vector multiplications of the complex-double type. Hardware tested included the NVIDIA A100 GPU, AMD MI100 GPU, and Intel Gen9 GPU. Their results demonstrate satisfactory performance portability across all GPUs when compared to CUDA and HIP.

An Intel project [15] evaluates DPC++ in four aspects: memory bandwidth utilization, migration of well-established CUDA algorithms, performance of migrated DPC++ to native implementations of CUDA, and oneMKL performance. The results show that DPC++ achieves bandwidth performance comparable to OpenCL [24] and CUDA. With respect to migrating well-established CUDA algorithms, the computational science algorithms of lid-driven cavity flow, heart wall tracking, k-means clustering, and GROMACS were each successfully migrated and tested. Porting to the Tesla K40 and Tesla V100, DPC++ performed reasonably well. A comparison of SGEMM between cuBLAS and oneMKL showed poor performance for oneMKL comparatively. Their explanation is that context creation cannot be excluded from the oneMKL event recording like it is in cuBLAS. Sufficiently large matrices help mitigate the overhead.

Porting sparse linear algebra library to Intel GPUs by developing a kernel backend based on the DPC++ programming environment was investigated in [39]. The main sparse kernel, the sparse matrix product (SpMV), was tuned in the Ginkgo math library and compared to Intel's oneMKL vendor library. Similarly to this work, all Intel GPU experiments were conducted on hardware that is part of the Intel DevCloud. Results show SpMVs achieve about 90% of peak bandwidth on GEN12 and about 60-70% of peak bandwidth on GEN9 Intel GPUs, and the performance comparison to oneMKL show mixed results.

This report focuses specifically on the DPC++ implementation of SYCL and tests both CPU and GPU hardware. Previous works provide a baseline expectation for performance of DPC++ code, specifically migrated CUDA code.

III. Background

A. Structure of MAGMA

The MAGMA library has been originally designed and implemented for heterogeneous systems that use NVIDIA GPUs [41]. Support was later extended to OpenCL [13], Intel Xeon Phi [42], and more recently to AMD GPUs by translating CUDA code to HIP [40]. To extend portability to oneAPI supported hardware, this translation process must be completed again. Figure 1 depicts the directories in MAGMA that are written in CUDA and used for executing CUDA code. It has been updated to depict the intended structure of MAGMA after support has been extended to oneAPI. In the updated structure, CUDA code has been migrated with the DPCT tool, followed by manual changes to DPC++ completed as necessary. DPC++ code may then be used to port to any oneAPI supported hardware. The oneMKL library [20] replaces cuBLAS. Specific to MAGMA, DPC++ compiler directives must be implemented by hand.

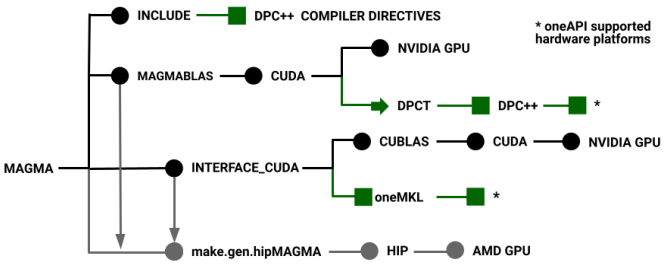


Fig. 1. Extended files structure of the MAGMA library and the MAGMA directories that contain CUDA code.

Note that the rest of the MAGMA library contains code and abstractions that are architecture-independent and do not need translation. The different ports of MAGMA to different architectures and programming models need to translate only the code in Figure 1.

B. The oneAPI Programming Model

There are three main components integrated into the design of the oneAPI programming model [10]. The most important component is DPC++, the direct programming language of oneAPI. It works in conjunction with oneAPI libraries, which are for API-based programming, and analysis and debugging tools. These together allow for a hardware-independent programming model where single-source code can be ported to scalar, vector, matrix, and spatial architectures. DPC++ itself is an implementation of the Khronos standard SYCL. SYCL is an accelerator language that allows code reuse across hardware targets. It is designed to add data parallelism and heterogeneous programming to standard ISO C++.

The oneAPI tools used in this project were downloaded through the Intel® oneAPI Base Toolkit [17]. Notably, this includes the DPC++ compiler [18], the DPCT tool, and oneMKL. The oneAPI compiler allows for code reuse by optimally compiling ISO C++ and SYCL across oneAPI supported hardware targets. The DPCT tool performs source-to-source migration from CUDA to DPC++ with high efficiency. For code that cannot be migrated, human readable comments are printed to aid in the completion of the migration by the user manually. The tool is based on LLVM/Clang [21], which, in this project, allowed for the possibility of porting to NVIDIA and AMD GPUs. The tool oneMKL refers to both an open-source math library interface and a library itself. It is designed for compatibility with a range of computational devices; thus, oneMKL is first called as an interface. It then calls the math kernel library that corresponds with the target device. For CPUs and Intel GPUs, oneMKL as a library is called and used. For NVIDIA GPUs, NVIDIA cuBLAS or MAGMA may be called and used instead.

Other compilers for C++ with SYCL besides the one in Intel's oneAPI, that are not used and investigated in this work, include implementations from Codeplay, Huawei, and Heidelberg University.

C. Computational Environment

The hardware used to test the performance of DPC++ includes two multicore CPUs (one from AMD and one from Intel), one NVIDIA GPU, and one Intel GPU. Ideally, more hardware configurations will be of interest to test, tune for, and use, and we have ran on some, including high-end Intel GPUs in the early-access precursors of the Aurora system, to find that the conclusions and main massage of this paper remain the same. The results that we show in this paper include the AMD EPYC 7742 CPU, the Intel® Xeon® Processor E5-2698 v4, the NVIDIA GeForce RTX 3060, and the Intel UHD Graphics P630 [0x3e96]. The AMD CPU comes with 64 cores, a base clock of 2.25GHz, 128 threads, and L3 cache of 256MB [34]. This contrasts to the Intel CPU, which has 20 cores, a base clock of 2.20 GHz, 40 threads, and 50MB of L3 smart cache [35]. The design of oneAPI allows DPC++ to be compiled directly to multicore CPUs upon installation of oneAPI.

The NVIDIA GPU used has 3,585 CUDA cores, a base clock of 1,320 MHz, and 12 GB of memory [36]. Additional software was installed to compile DPC++ to this GPU: DPC++-LLVM (Clang-LLVM) [21]. This corresponds to the LLVM/Clang design of the DPCT tool. DPC++-LLVM is an open source project on the DPC++ compiler. After the initial download, the compiler can be built for either an NVIDIA or AMD GPU target backend. The tutorial for NVIDIA support in [21] was successfully conducted for the NVIDIA GeForce RTX 3060 with one adjustment. The tutorial clones the compiler repository with git, runs a python script, and then performs the following command: *make install -j 'nproc'*. This was replaced with two commands, *make sycl-toolchain -j 'nproc'* and *make install*, for a successful installation.

The Intel GPU differs from the NVIDIA GPU in design. It is a mobile integrated GPU rather than a discrete GPU. It has 192 cores, a base clock of 350 MHz and a shared memory system [37], and as a mobile device has significantly less computational power than the NVIDIA GPU. At the time this research was conducted, a discrete Intel GPU was unavailable for public testing [25]. Contrary to the other computational devices, the Intel GPU was run on the cloud, through Intel DevCloud [26]. Intel DevCloud offers remote development environments that grant access to Intel hardware. This project used the JupyterLab environment to access the publicly available Intel GPU.

IV. Methodology

This research aimed to both measure the effort and evaluate the ease of portability, but it should be acknowledged that the research was limited in time and conducted over the course of a summer internship of ten weeks. The first step involved system configuration for running DPC++ code. The oneAPI Base Toolkit was installed on various architectures, and the process was documented. To test the DPCT for migration accuracy, the following CUDA samples where first downloaded from GitHub [27]. The example *matrixMul.cu* was used, available in the *matrixMul* directory. It provides a standard SGEMM algorithm in CUDA, derived from V. Volkov and J. Demmel's state-of-the-art SGEMM for Nvidia Tesla GPUs [38]. The *matrixMul.cu* file and corresponding header files were moved into a separate directory. This allowed us to test the DPCT tool with a file that had header dependencies, which would be the case for migration within MAGMA. Translated CUDA files are appended with an extension for DPC++; thus, *matrixMul.cu* was migrated to *matrixMul.dp.cpp* in an output directory. After this step, the DPC++-LLVM compiler was installed and built to target NVIDIA GPUs. This allowed us to test DPC++ code on a GPU. Next, an account to the University of Tennessee's Innovative Computing Laboratory (ICL) was granted to access more powerful CPUs and GPUs. The system configuration step must be repeated in this environment; the installation of oneAPI was again successful, but the installation of the DPC++-LLVM compiler was unsuccessful. At this point, the Intel and AMD multicore CPUs and the NVIDIA GPU were accessible, so the decision was made to prioritize the goal of translating MAGMA. The consequence of this prioritization left behind the testing of DPC++ on AMD GPUs. Next, to provide a self-contained example of the port for reproducible purposes, a directory was setup with MAGMA SGEMM CUDA code and all of the required dependencies (about 50 files, including all magma header files and GEMM sources). The DPCT tool was used to recursively migrate the SGEMM files and headers. Some embedded header files did not get translated by the tool, so these were migrated independently in a separate directory. They were afterwards copied into the directory with the translated DPC++ files. Compiler directives specific to MAGMA were implemented manually to complete the translation process. To test the MAGMA DPC++ SGEMM, a benchmark based on extending the *matrixMul.dp.cpp* file

```
if ( TransA == 0 && TransB == 0 ) {


    dim3 dimGrid ( magma_ceildiv( m, BLK_M_nn ),
                   magma_ceildiv ( n, BLK_N_nn ) );


    sgemm_kernel_fermi_nn<<< dimGrid, dimBlock, 0,
                             queue->cuda_stream() >>> (
            m, n, k, dA, ldda, dB, lddb,
            dC, lddc, alpha, beta,
            (int)offsetA, (int)offsetB );


}
```

Listing 1. Original CUDA code in MAGMA calling the $C = \alpha AB + \beta C$ sgemm routine.

```
if ( TransA == 0 && TransB == 0 ) {
    sycl::range<3> dimGrid(1,magma_ceildiv(n, BLK_N_nn),
                             magma_ceildiv(m, BLK_M_nn));
    /* DPCT1049:36: The work-group size passed to the SYCL
       kernel may exceed the limit. To get the device limit,
       query info::device::max_work_group_size.
       Adjust the work-group size if needed. */
    queue->submit( [&](sycl::handler &cgh ) {
      sycl::accessor<FloatingPoint_t, 2,
                     sycl::access_mode::read_write,
                     sycl::access::target::local>
          sA_acc_ct1(sycl::range<2>(BLK_K_nn, BLK_M_nn+1),cgh);
      sycl::accessor<FloatingPoint_t, 2,
                     sycl::access_mode::read_write,
                     sycl::access::target::local>
          sB_acc_ct1(sycl::range<2>(BLK_N_nn, BLK_K_nn+1),cgh);

      cgh.parallel_for( sycl::nd_range<3>(dimGrid * dimBlock,
                                          dimBlock),
                        [=](sycl::nd_item<3> item_ct1) {
        sgemm_kernel_fermi_nn( m, n, k, dA, ldda, dB, lddb,
                               dC, lddc, alpha, beta,
                               (int)offsetA, (int)offsetB,
                               item_ct1, sA_acc_ct1,
                               sB_acc_ct1);
      });
    });
}
```

Listing 2. DPC++ translated CUDA code from MAGMA calling the $C = \alpha AB + \beta C$ sgemm routine.

Fig. 2. Left: Original CUDA sgemm call in MAGMA; Right: The corresponding DPC++ code translated using DPCT plus some hand fixes.

was created to call the MAGMA SGEMM implementation and header files rather than the original, standard implementation of MAGMA's SGEMM benchmark.

Figure 2 shows an example of the DPCT translation of the MAGMA SGEMM call. The original code is on the left listing and the translated one followed by manual changes is to the right. Note that the DPCT tool looked into the original code and if texture memory or shared memory are used, the allocations get pulled and declared as objects outside the call and get passed to the routine as parameters. The `sA_acc_ct1` and `sB_acc_ct1` objects in this case are for shared memory. There were many more objects for texture memory use generated automatically that were giving errors, even though they were not used as a result of being into macro sections. Those had to be removed by hand and this is one example of what we had to fix manually.

To evaluate the performance of DPC++, different tests were set up for the CPUs, the NVIDIA GPU, and the Intel GPU. For the NVIDIA GPU, the original sample CUDA SGEMM and MAGMA SGEMM algorithms were compared with their migrated DPC++ counterparts. This corresponds with DPC++(MAGMA), MAGMA, DPC++(CUDA), and CUDA in Figure 6, included in the OneAPI Portability Results sections of this paper. For the two CPUs, since CUDA cannot run on CPU, the original sample CUDA and MAGMA SGEMM implementations were replaced with MKL and C++. This provides a performance baseline since MKL is written in optimized Assembly code and C++ uses OpenMP [28]. This corresponds to DPC++(MAGMA), DPC++(CUDA), MKL, and C++(OpenMP) in Figures 4 and 5. For the Intel GPU, oneMKL was first compared to the migrated standard SGEMM and MAGMA SGEMM algorithms. The MAGMA SGEMM algorithm, tuned for NVIDIA GPUs, performed poorly, so new algorithm parameters were tested. An explanation of the algorithm parameters is provided in the following section. The original test parameters are labeled *cuda*, and two new sets of SGEMM algorithm parameters are labeled *ker2*, and *ker11*. This corresponds to DPC++(MAGMA cuda), DPC++(MAGMA ker2), and DPC++(MAGMA ker11) in Figure 8. For all performance tests, the N matrix dimensions tested remained constant. The original CUDA SGEMM algorithm required N to be divisible by 32. The largest size of N tested was 8,192. The range of N was intended to mimic performance tests conducted in a previous extension of MAGMA to support OpenCL [13].

V. General Matrix-Matrix Multiplication (GEMM) Design and Implementation in DPC++

The GEMM design and implementation are of fundamental importance in HPC as many scientific computing applications are designed in terms of GEMM operations. GEMM is expected to run close to machine peak of the underlying hardware so that applications using GEMMs derive their high-performance and performance portability across different hardware from highly efficient GEMM implementations. This is the reason that we concentrate on the GEMM kernel in this paper. Performance portable GEMM implementations however are very challenging to develop – even for just one specific architecture, let alone having a single code to be performance-portable across architectures. Still, our goal is to evaluate exactly that, i.e., to what extent can a single DPC++ GEMM implementation be performance-portable.

To evaluate the performance-portability of a DPC++ code we need to set first references for comparison. These will be lower-bound implementations that are reasonable reference implementations (as given in Listings 1 and 2), that the DPC++ code will hopefully outperform, and higher-bound state-of-the-art implementations, typically architecture-specific and written in Assembly (e.g., MKL or MAGMA for Nvidia GPUs, cf. Figure 3), that the DPC++ code will ideally get close to in performance.

```
void sgemm_ijk(int m, int n, int k,
               float alpha, float *A, int lda,
                            float *B, int ldb,
               float beta,  float *C, int ldc) {
    for(int i = 0 ; i < m; ++i ) {
        for(int j = 0 ; j < n ; ++j ) {
            float sum = 0.0;
            for(int l = 0 ; l < k ; ++l ) {
                sum += A[i+lda*l] * B[l + j*ldb];
            }
            C[i+j*ldc] = beta * C[i+ j*ldc] + alpha * sum;
        }
    }
}
```

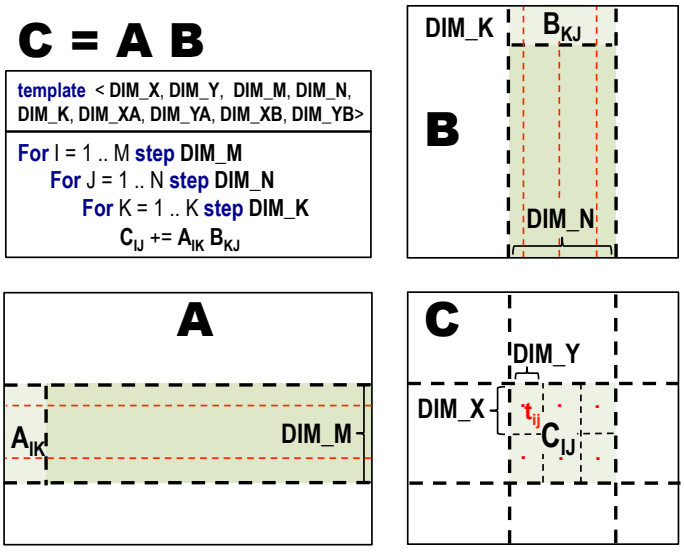
Listing 3. ijk loop implementation of the sgemm routine.

Listing 1 gives a reference GEMM implementation that is, in general, of very low performance, and thus a very low bar to outperform. Therefore, Listing 1 code is almost never used by itself in practice. A higher performance implementation is to block for computation for higher memory reuse, as given in Listing 2. Note that Listing 2 uses the Listing 1 code in its innermost loop, and also renders a parallel implementation using OpenMP, where the I and J loops are collapsed to be performed in data-parallel fashion on different CPU cores/threads. The implementation is also parameterized, allowing the tuning of this code to be of very high-performance. This code is used as a target goal to outperform using DPC++ code. In subsequent performance measurements we denote it by C++(OpenMP), as given in Figures 4 and 5.

```
#define BLK 96
void sgemm_bijk(int m, int n, int k,
                float alpha, float *A, int lda,
                             float *B, int ldb,
                float beta,  float *C, int ldc) {
  #pragma omp parallel for collapse(2) schedule(dynamic)
  for(int I = 0 ; I < m; I+=BLK )
    for(int J = 0 ; J < n ; J+=BLK )
      for(int K = 0 ; K < k ; K+=BLK ) {
        int bm = min(BLK, m-I);
        int bn = min(BLK, n-J);
        int bk = min(BLK, k-K);
        if (K==0)
          sgemm_ijk(bm,bn,bk, alpha, A+I+K*lda, lda,
                    B+J*ldb+K,ldb, beta, C+I+J*ldc,ldc);
        else
          sgemm_ijk(bm,bn,bk, alpha, A+I+K*lda, lda,
                    B+J*ldb+K,ldb, 1.0, C+I+J*ldc,ldc);
      }
}
```

Listing 4. Blocked OpenMP implementation of the sgemm routine.

The MAGMA general matrix-matrix multiplication (GEMM) algorithm translated and tested in this report is shown in Figure 3. This algorithm is used for GEMM of any type, including single precision. There are nine important constants that tune the algorithm to a hardware architecture. The first two, labelled DIM_X and DIM_Y, determine the dimensions of the thread block executed in parallel. The second set of parameters, DIM_M and DIM_N, provide the dimensions of the sub-matrix product that a thread block computes. The remaining five parameters affect how loading into shared memory is done for parts of the A and B matrices (as illustrated on Figure 3). Another important aspect to note is that the product sub-matrix C_{IJ} is stored in the registers of the multiprocessor/core that computes C_{IJ}, which vary in number across different hardware. Thus, selecting the size of C_{IJ} is crucial for obtaining high performance. This algorithm was developed and autotuned for optimal performance on NVIDIA GPUs [33]. The parameters that optimize the GEMM algorithm on current high-end NVIDIA devices are labeled as *cuda* in Table 1.



- Implementation is templated with 9 parameters
- Computation is done with thread blocks of size [**DIM_X** , **DIM_Y**]
- Thread t_{ij} computes [**DIM_M** / **DIM_X**, **DIM_N** / **DIM_Y**] elements of $\mathbf{C_{IJ}}$
- $\mathbf{A_{IK}}$ gets loaded in shared memory by [**DIM_XA** , **DIM_YA**] threads
- $\mathbf{B_{KJ}}$ gets loaded in shared memory by [**DIM_XB** , **DIM_YB**] threads
- $\mathbf{C_{IJ}}$ is held and computed in registers

Fig. 3. The GEMM algorithm in MAGMA [14].

Although double-precision (64-bit) floating-point arithmetic (FP64) is a standard in HPC, we choose to present results in single-precision 32-bit floating point arithmetic (FP32). The MAGMA codes have single implementation regarding precision and the rest are generated (alternative is to template precision and generate the different versions by a C++ compiler). In particular, MAGMA sources are written for double-complex arithmetic and single real, double real, and single complex are generated. Thus, the precision choice is not important for the choice and conclusion results in this paper. A motivation to choose FP32 is that FP32 arithmetic is often more than 2× faster than FP64 for many GPUs and many applications, including machine learning and deep neural networks, are currently aiming to leverage this performance

boost to somehow get away with reducing the accuracy (even to FP16) and possibly regaining it with mixed-precision calculations where the bulk of the computation is in reduced FP16 or FP32 precision [43] [44].

VI. OneAPI Portability Results

A. Test Parameters

The algorithm optimized for the NVIDIA GPUs, with parameters labeled as *cuda* and listed in Table 1, performed well on the multicore CPUs, significantly outperforming the reference OpenMP code and even the Intel MKL on AMD CPUs. This was confirmed by measuring the performance, monitoring the CPU usage, and profiling information. For the NVIDIA GPU, performance was also very good as the migrated code retained the performance of CUDA. The *cuda* GEMM kernel did not perform well on the integrated Intel GPU, as expected. New parameter sets were tested. The best two selected, labeled *ker2* and *ker11*, are listed in Table 1, along with a number of other parameters included in the search space used for this paper.

B. MAGMA to DPC++ Portability on Multicore CPUs

1) Hardware Usage: Profiling and performance measurements, as well as monitoring commands like *htop* [29], were used to verify CPU usage. These are important indicators on whether the algorithm parameters are a good fit for the architecture. In particular, *htop* was used as an interactive process viewer to confirm exact process usage and for troubleshooting, e.g., to observe and verify the number of threads used and their load. The migrated code used 100% for each of the CPU's cores available.

2) Performance: The migrated DPC++ code successfully ran on multicore CPUs. As initial migrated code, it demonstrated impressive performance. On the AMD CPU, shown in Figure 4, the migrated DPC++(MAGMA) SGEMM algorithm significantly outperforms the OpenMP C++(OpenMP) implementation, as well as the DPC++(CUDA), and even the MKL implementation (for N less than 7,000). This result is very significant as it shows that an algorithm designed for Nvidia GPUs can be very efficient for multicore CPUs as well. Note that the DPC++(MAGMA) implementation outperforms even our upper performance target, which is MKL in this case. This means parallelization, blocking for reduced communication, and vectorization have been all efficiently achieved. Thus, since a fast multicore GEMM is very challenging to develop, this is one illustration of both functional and performance portability of MAGMA GEMM, and arguably the entire MAGMA because of the GEMM importance, to multicore CPUs using the Intel's oneAPI programming model and toolkit. Furthermore, MAGMA never had a port to just multicore CPUs, and this result shows that the oneAPI port as being developed is a feasible solution to easily provide this functionality in a performance portable way.

The same conclusions can be derived from runs on Intel multicore CPUs. The results are shown in Figure 5. Note that as MKL is specifically tuned for Intel CPUs, MKL had the highest performance, as one would expect. However, the MAGMA DPC++(MAGMA) implementation is not much lower, and still outperformed the standard CUDA SGEMM algorithm translated to DPC++. More notably, the translated DPC++ code significantly outperformed the OpenMP implementation (on both the Intel and AMD CPUs). We note that performance improvements are possible for the DPC++(MAGMA) by autotuning, e.g., by testing a search space of algorithms like the one in Table 1 and as discussed in the Intel GPUs section. The result here is for the fixed GEMM kernel/algorithm (parameters) that is used for high-end Nvidia GPUs.

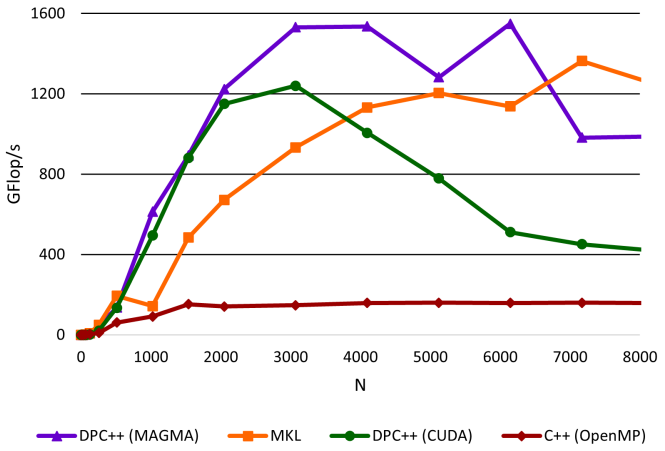


Fig. 4. SGEMM Performance on AMD EPYC 7742 64-Core Processor @ 2.25GHZ

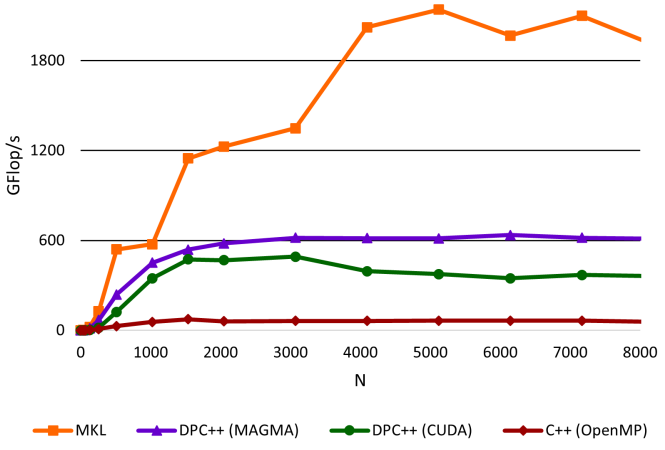


Fig. 5. SGEMM Performance on INTEL® XEON® CPU E5-2698 V4 20-Core Processor @ 2.20GHZ

C. MAGMA to DPC++ portability on NVIDIA GPUs

1) Hardware Usage: After successfully porting to the NVIDIA GPU with the DPC++-LLVM installation, usage was checked with the command *watch -n0.5 nvidia-smi* [30]. Migrated code was shown to use 100% of the NVIDIA GPU.

TABLE I
GEMM TEST PARAMETERS

constants	DIM_X	DIM_Y	DIM_M	DIM_N	DIM_K	DIM_XA	DIM_YA	DIM_XB	DIM_YB
cuda	16	16	96	96	16	32	8	8	32
ker1	32	4	64	64	4	32	4	32	4
ker2	16	16	64	64	8	32	8	8	32
ker3	4	32	64	64	4	32	4	32	4
ker4	4	32	64	64	4	4	32	4	32
ker5	32	4	64	64	4	4	32	4	32
ker6	16	4	64	64	2	32	2	32	2
ker7	16	4	64	64	1	64	1	64	1
ker8	8	4	32	32	1	32	1	32	1
ker9	8	4	64	32	1	32	1	32	1
ker10	4	4	32	32	2	8	2	8	2
ker11	12	4	48	48	2	24	2	24	2

This indicated that optimization of the SGEMM MAGMA algorithm was retained.

2) Performance: The performance of the initial migrated code on the NVIDIA GPU demonstrated impressive results. As shown in Figure 6, the migrated MAGMA DPC++(MAGMA) code, originally tuned for the NVIDIA GPU, retained the performance of the CUDA implementation. This is significant result as it illustrates that DPC++ is expressive enough for parallel algorithms that have to map and run well on Nvidia GPUs, as it matches in performance CUDA that is designed for Nvidia GPUs. Also, it shows that the Intel oneAPI compiler is very good in generating highly optimized code for Nvidia GPUs. This is the ideal outcome for a new language, its compiler, and a translation tool – the tool to translate a highly-optimized code to a new language, compile the new code, and achieve the same performance as the original code on hardware for which the original code had been tuned.

Combined with the results from the previous section, we conclude that after a full translation of MAGMA is complete, the same code can be used for both functional and performance portability to NVIDIA GPUs, multicore CPUs, and Intel GPUs.

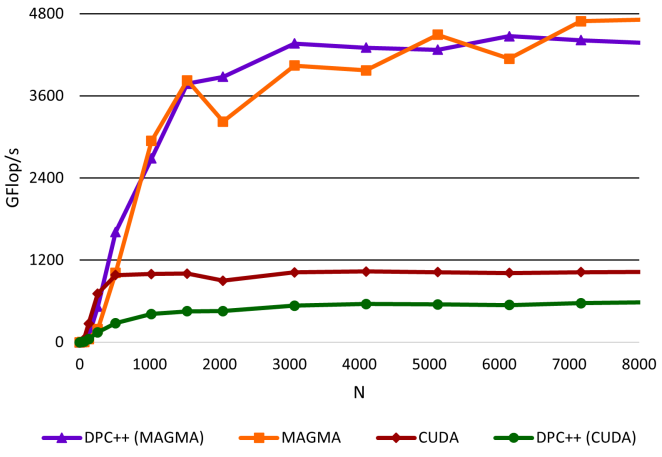


Fig. 6. SGEMM Performance on NVIDIA GeForce RTX 3060

D. MAGMA to DPC++ portability on Intel GPUs

1) Hardware Usage: The Intel GPU was tested on the cloud with Intel DevCloud. Inferences about the usage were made based on the performance obtained.

2) Performance: The initial migration to DPC++ did not compile on the Intel GPU. [15] offered insight into the issue by explaining the maximum work group size [31] for NVIDIA devices and the integrated Intel GPU. For NVIDIA devices, it is constant at 1024, while for this specific Intel device, it is 256. The variables that determine the work group size were updated in the migrated code. The code then compiled and executed. Initial performance results were collected. As shown in Figure 7, these results were very poor; the MAGMA DPC++ SGEMM algorithm never exceeded two GFlop/s in performance. This indicated that the parameters originally used in the MAGMA SGEMM algorithm, labeled *cuda* in Table 1, needed to be updated to accommodate for the Intel GPU architecture. Multiple constant sets were tested next, to discover *ker2* and *ker11*, which increased the performance to more than ten times that of the standard SGEMM algorithm, as shown in Figure 8. Optimal algorithm parameters are yet to be determined. Without specific details about the Intel GPU architecture, several hundred parameter combinations must be tested through trial and error (following an autotuning approach [33]).

VII. CONCLUSIONS AND FUTURE WORK DIRECTIONS

Intel's oneAPI proves to be a promising approach for portable parallel programming on heterogeneous systems with various hardware architectures. The DPCT tool can be used successfully for an initial port of CUDA code to DPC++. DPC++ code was successfully compiled and tested on multicore CPUs, NVIDIA GPUs, and the integrated Intel GPU. Thus, the MAGMA port to DPC++ can be used to provide support for Intel GPUs, NVIDIA GPUs, AMD GPUs, and multicore CPUs. DPC++ shows that large numerical libraries like MAGMA, originally written in CUDA to support NVIDIA GPUs, can be easily translated to DPC++ to provide functional portability to different vendor GPUs, as well as multicore CPUs. Initial migrated code tuned for NVIDIA GPUs performs

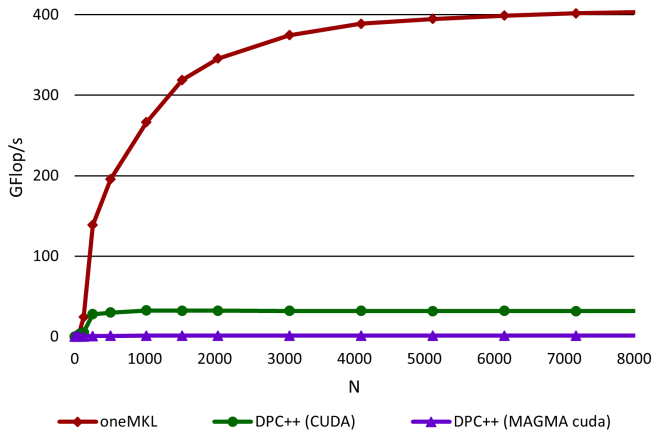


Fig. 7. Initial SGEMM Performance on Intel UHD Graphics P630 [0x3e96]

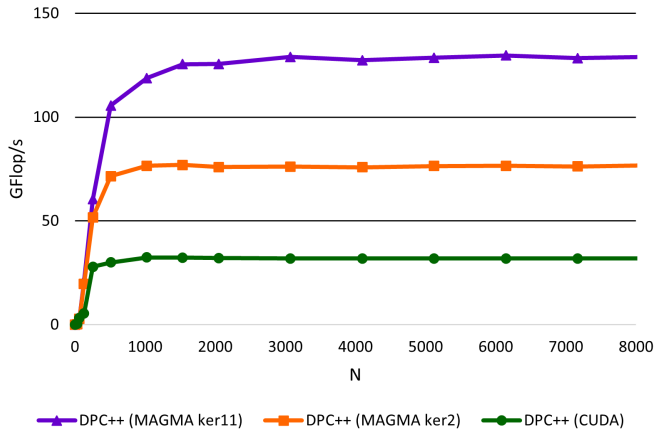


Fig. 8. Tuned SGEMM Performance on Intel UHD Graphics P630 [0x3e96]

well on multicore CPUs. It retains performance of algorithms optimized for NVIDIA GPUs. Initial migrated code that is tuned for NVIDIA GPUs performs poorly on the available Intel GPU, but tuning can improve performance. However, optimal parameters are difficult to find without specific knowledge of the hardware architecture as thousands of parameter combinations must be tested to find the best.

The success of the MAGMA SGEMM migration has established a process for the eventual full extension of MAGMA support to oneAPI. To take full advantage of the computational capabilities of the Intel GPU for GEMM algorithms, further sets of constants must be tested using autotuning techniques to discover the best performing versions.

Acknowledgment

This project was sponsored by the National Science Foundation through the Research Experience for Undergraduates (REU) award no. 2020534 with additional support from the National Institute of Computational Sciences and Innovative Computing Laboratory at the University of Tennessee, Knoxville. The contributions of the second author were supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early tested platforms, in support of the nation's exascale computing imperative.

References

[1] *Advancing computing and data capabilities for scientific discovery and continued U.S. technological leadership.* Oak Ridge National Lab. urlhttps://www.ornl.gov/directorate/ccsd

[2] *Computing at LLNL.* Lawrence Livermore National Laboratory. https://computing.llnl.gov/

[3] *High performance computing.* U.S. Department of Energy, Office of Science. https://www.energy.gov/science/high-performance-computing

[4] *NVIDIA history.* NVIDIA. https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/

[5] *New Titan supercomputer named fastest in the world.* Department of Energy. https://www.energy.gov/articles/new-titan-supercomputer-named-fastest-world-0

[6] *June 2019.* The Top 500 List. https://www.top500.org/lists/top500/2019/06/

[7] *June 2022.* The Top 500 List. https://www.top500.org/lists/top500/2022/06/

[8] *Aurora: HPC and AI at exascale.* Intel. https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html

[9] *Compare benefits of CPUs, GPUs, and FPGAs for different oneAPI compute workloads.* Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html#gs.83qstn

[10] *Intel oneAPI programming overview.* Intel. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/introduction-to-oneapi-programming/intel-oneapi-programming-overview.html

[11] *MAGMA.* Innovative Computing Laboratory, University of Tennessee at Knoxville. https://icl.utk.edu/magma/

[12] A. S. Dufek et al. "Case study of using Kokkos and SYCL as performance-portable frameworks for Milc-Dslash benchmark on NVIDIA, AMD and Intel GPUs." in 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2021, pp. 57-67, doi: 10.1109/P3HPC54578.2021.00009.

[13] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming." in Parallel Computing, vol. 38, 2012, pp. 391–407.

[14] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi Graphics Processing Units." The International Journal of High Performance Computing Applications 24, no. 4 (November 2010): pp. 511—515. https://doi.org/10.1177/1094342010385729.

[15] *Benchmarking the performance of oneAPI on heterogeneous computing Platforms.* Moasys, Intel Software. 2021. https://www.moasys.com/files/upload_oneapi_webinar_20210618.pdf?ckattempt=2

[16] *Data Parallel C++: the oneAPI implementation of SYCL*.* Intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html#gs.83xmmq

[17] *Intel® oneAPI Base Toolkit*. Intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html#gs.8anj8g

[18] *Intel® oneAPI DPC++/C++ Compiler*. Intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html#gs.vnkxf8

[19] Z. Wang, et al. "A source-to-source CUDA to SYCL code migration tool: Intel®DPC++ Compatibility Tool," presented at IWOCL'22: International Workshop on OpenCL. vol. A17, pp. 1-2, May 2022. https://doi-org.libproxy.library.unt.edu/10.1145/3529538.3529562

[20] M. Krainiuk, M. Goli, V. R. Pascuzzi. "oneAPI open-source math library interface," presented at 2021 International Workshop on Performance, Portability and Productivity in HPC (p3HPC). 2021. https://ieeexplore-ieee-org.libproxy.library.unt.edu/stamp/stamp.jsp?tp=&arnumber=9652858&tag=1
[21] *Compiling SYCL for different GPUs.* Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/compiling-sycl-with-different-gpus.html
[22] *oneAPI.* Intel. https://www.oneapi.io/spec/
[23] *SYCL Specification*. KHRONOS. https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf
[24] *The OpenCL specification.* Khronos. https://registry.khronos.org/OpenCL/specs/opencl-1.1.pdf
[25] *Intel® Iris® Xe Max Graphics to be retired from Intel® DevCloud on 07/29/2022.* Intel. https://community.intel.com/t5/Intel-DevCloud/Intel-Iris-Xe-Max-Graphics-to-be-retired-from-Intel-DevCloud-on/m-p/1402673#M5623
[26] *Intel DevCloud.* Intel. https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html
[27] *CUDA Samples.* NVIDIA. https://github.com/NVIDIA/cuda-samples
[28] *OpenMP application programming interface.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
[29] *Part 3.2 - Linux task managers, top, and htop* Microsoft. https://docs.microsoft.com/en-us/troubleshoot/developer/webapps/aspnetcore/practice-troubleshoot-linux/3-2-task-managers-top-htop
[30] *NVIDIA System Management Interface.* NVIDIA. https://developer.nvidia.com/nvidia-system-management-interface
[31] *SYCL* Thread Mapping and GPU Occupancy.* Intel. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top/thread-mapping.html
[32] G. Castaño, Y. Faqir-Rhazoui, C. García, M. Prieto-Matías. "Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing," in Journal of Parallel and Distributed Computing, vol. 165, pp. 120-129. 2022. https://doi.org/10.1016/j.jpdc.2022.03.017
[33] J. Kurzak, S. Tomov, J. Dongarra. "Autotuning GEMM kernels for the Fermi GPU," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 11, pp. 2045-2057, Nov. 2012, doi: 10.1109/TPDS.2011.311.
[34] *AMD EPYC™ 7742.* AMD. https://www.amd.com/en/products/cpu/amd-epyc-7742
[35] *Intel® Xeon® Processor E5-2698 v4.* Intel. https://ark.intel.com/content/www/us/en/ark/products/91753/intel-xeon-processor-e52698-v4-50m-cache-2-20-ghz.html
[36] *GEFORCE RTX 3060 FAMILY.* Nvidia. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3060-3060ti/
[37] *Intel UHD Graphics P630.* TechPowerUp. https://www.techpowerup.com/gpu-specs/uhd-graphics-p630.c3676
[38] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1-11, doi: 10.1109/SC.2008.5214359.
[39] Y. Tsai, T. Cojean, and H. Anzt, "Porting Sparse Linear Algebra to Intel GPUs," In Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers. Springer-Verlag, Berlin, Heidelberg, 57–68. https://doi.org/10.1007/978-3-031-06156-1_5
[40] C. Brown, A. Abdelfattah, S. Tomov and J. Dongarra, "Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs," 2020 IEEE High Performance Extreme Computing Conference (HPEC), 2020, pp. 1-7, doi: 10.1109/HPEC43674.2020.9286214.
[41] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," Parallel Computing, Volume 36, Issues 5–6, 2010, Pages 232-240, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2009.12.005.
[42] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "HPC programming on Intel many-integrated-core hardware with MAGMA port to Xeon Phi," Sci. Program. 2015, Article 9 (January 2015), Vol. 23, https://doi.org/10.1155/2015/502593.
[43] A. Haidar, S. Tomov, J. Dongarra and N. J. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers," SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018, pp. 603-613, doi: 10.1109/SC.2018.00050.
[44] A. Abdelfattah, H. Anzt, E. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. Higham, X. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. Tsai, and U. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," Int. J. High Perform. Comput. Appl. 35, 4 (Jul 2021), 344–369. https://doi.org/10.1177/10943420211003313.

APPENDIX

DATA AVAILABILITY STATEMENT

SUMMARY OF THE EXPERIMENTS REPORTED

The code is made available on GitHub. There is a Makefile. The Intel oneAPI tools have to be installed and one can compile with "make cpu" for multicore CPUs, or "make gpu" for Nvidia or Intel GPUs. The code can be run with "make runCpu" and "make runGpu" on multicore CPUs or GPUs, respectively. The kernels can be changed manually in the Makefile. The default is the cuda kernel and to change it, one must change "$(cuda)" to "$(ker11)" to run ker11, for example. The code is tested and is reproducible on the architectures specified. The code has been tested and runs on JLSE Arcticus, precursor for Aurora, to confirm portability on high-end Intel GPUs. If we receive approval to show NDA results, we plan to add them to the final paper.

ARTIFACT AVAILABILITY

a) Software Artifact Availability:: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

b) Hardware Artifact Availability:: There are no author-created hardware artifacts.

c) Data Artifact Availability: : There are no author-created data artifacts.

d) Proprietary Artifacts:: None of the associated artifacts, author-created or otherwise, are proprietary.

e) List of URLs and/or DOIs where artifacts are available:: `https://github.com/stomov/oneMAGMA-example`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

f) Relevant hardware details:: AMD EPYC 7742 64-Core Processor @ 2.25GHz, Intel® Xeon® CPU E5-2698 V4 20-Core Processor @ 2.20GHZ, NVIDIA GeForce RTX 3060, Intel UHD Graphics P630 [0x3e96]

g) Operating systems and versions:: CentOS Linux 7 (Core)

h) Compilers and versions:: Intel DPC++/C++ Compiler for Linux v2022.1.0, DPC++-LLVM (CLang-LLVM) v15.0.0,

i) Libraries and versions:: MAGMA 2.6.2

j) Key algorithms:: GEMM

ARTIFACT EVALUATION

k) Verification and validation studies:: Verification and validation were performed on several architectures. Multiple runs were performed. Accuracy is always compared to a known solution.

l) Accuracy and precision of timings:: Timing is given as an average.

m) Used manufactured solutions or spectral properties:: We used manufactured solutions as well as verified third party software to compare solutions.

n) Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment:: We present average timing to avoid qualifying sensitivity.

o) Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system.: Averaging time measurements.