Extending MAGMA Portability with OneAPI

Anna Fortenberry
University of North Texas

Stanimire Tomov (advisor)
Kwai Wong (advisor)
University of Tennessee, Knoxville

ACKNOWLEDGEMENTS

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.

INTRODUCTION

- Supercomputers (SC) provide computational power necessary to resolve problems in a vast number of important domains, such as High Performance Computing (HPC)
- SC architectures are becoming increasingly diverse in architecture types and designs
  - Eight of the top ten SCs ranked in the Top500 list use accelerators, coming from three different vendors [1]
  - Intel plans to join as a SC GPU vendor with their announcement of Aurora
- Interoperability decreases between code and hardware with this trend
- Intel presents an architecture-independent programming model interface called **oneAPI** as a solution
  - Scalar, vector, spatial, and matrix architectures are supported
- Matrix Algebra on GPU and Multicore Architectures (**MAGMA**), a dense linear algebra library, can be used to test the capability and value of this programming model

ONEAPI PROGRAMMING MODEL

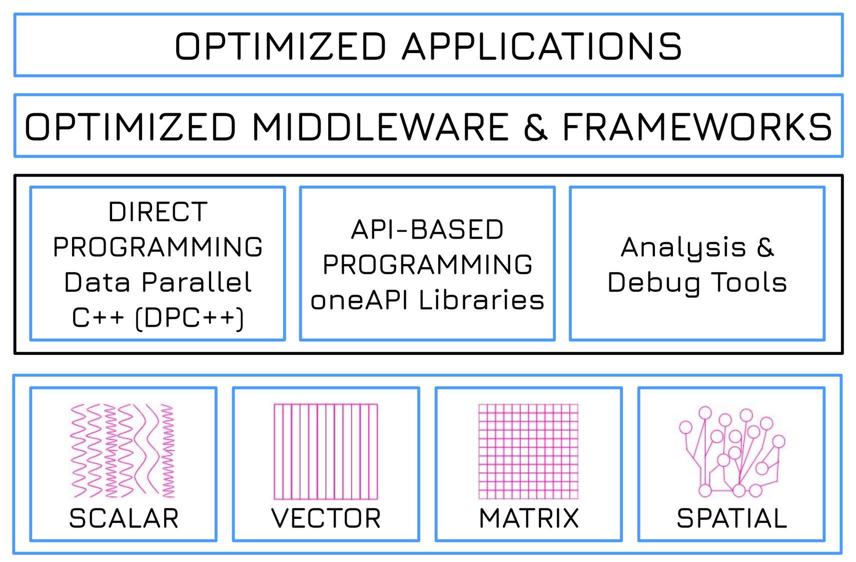

Figure 1: oneAPI Programming Model

- **oneAPI** consists of three main components: Data Parallel C++ (DPC++), oneAPI libraries, and analysis & debugging tools [2]
  - DPC++ is an implementation of the Khronos standard SYCL
- SYCL is an accelerator language that allows code reuse across hardware targets
  - Data parallelism and heterogeneous programming added to standard ISO C++

STRUCTURE OF MAGMA

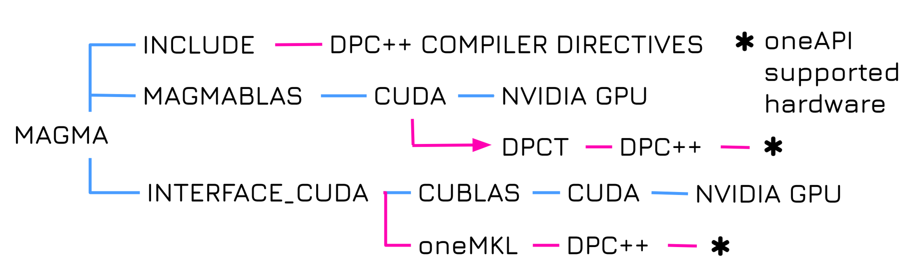

Figure 2: Extended Structure of MAGMA

MAGMA will adopt the oneAPI model by using the DPC++ Translation Tool (DPCT) and the oneAPI Math Kernel Library (oneMKL).

COMPUTATIONAL ENVIRONMENT

AMD EPYC 7742 PROCESSOR V4
Cores: 64
Base Clock: 2.25 Ghz
Threads: 128
Cache: 256 MB

INTEL® XEON® PROCESSOR E5-2698
Cores: 20
Base Clock: 2.20 Ghz
Threads: 40
Cache: 50 MB

NVIDIA GEFORCE RTX 3060
GPU Cores: 3584
Base Clock: 1320 MHz
Memory: 12 GB
Discrete GPU

INTEL UHD GRAPHICS P630 [0x3e96]
Cores: 192
Base Clock: 350 MHz
Memory: Shared System
Integrated GPU

- Intel's GPU is a mobile device and thus has significantly less computational power than the Nvidia GPU
- DPC++ can be compiled directly to multicore CPUs and Intel GPUs upon installation whereas Nvidia GPU requires installation of the DPC++-LLVM (Clang-LLVM) compiler [3]

GENERAL MATRIX-MATRIX MULTIPLICATION (GEMM) DESIGN AND IMPLEMENTATION IN DPC++

- GEMM design and implementation is fundamental in HPC
- Performance-portable GEMM implementations are challenging to develop
- Goal is to evaluate to what extent can a single DPC++ code be performance-portable
- References for comparison aid with evaluation
  - Lower-bound implementations, as given in Listings 1 and 2
  - Higher-bound, state-of-the-art implementations that are architecture-specific, e.g. MKL and MAGMA

```
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 1: ijk loop implementation of the SGEMM routine.

- Listing 1 is a reference GEMM implementation of low performance - a low bar to outperform
- Listing 2 shows a higher performance implementation of Listing 1, which blocks for computation for higher memory reuse
- Parallel implementation rendered using OpenMP - I and J loops are collapsed to be performed in data-parallel fashion on different CPU cores/threads
- Implementation is parametrized, allowing for tuning
- Target goal to outperform using DPC++ code.
- Denoted by C++(OpenMP)

```
#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 2: Blocked OpenMP implementation of the SGEMM routine.

- MAGMA's GEMM algorithm translated and tested in this research is shown in Figure 3
  - Used for GEMM of any type
- 9 important constants for tuning algorithm to a given hardware
  - DIM_X and DIM_Y determine dimensions of thread block executed in parallel
  - DIM_M and DIM_N determine dimensions of sub-matrix product which thread block computes
  - Remaining constants affect how matrices A and B are loaded into shared memory
- Product C_IJ is stored in registers of multiprocessor/core that computes C_IJ, which vary in number across hardware
- Size of C_IJ is critical to high performance
- Algorithm was developed and auto-tuned for optimal performance on NVIDIA GPUs [5]
- Parameters that optimize the GEMM algorithm on current high-end NVIDIA devices are labeled as **cuda** in Table 1

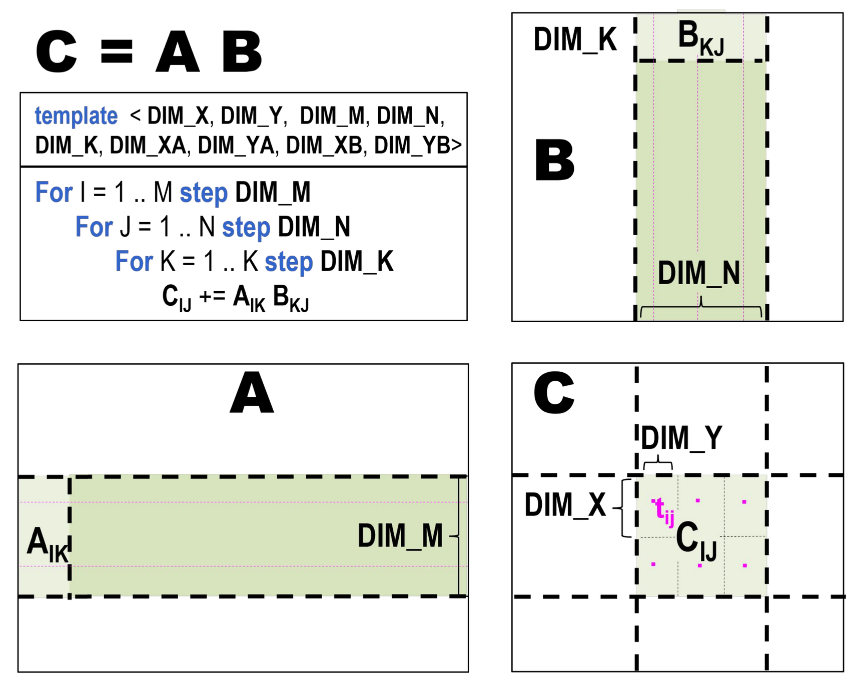

Figure 3: The GEMM algorithm in MAGMA [6].

GEMM PARAMETERS

- **cuda** set performed well on multicore CPUs
- DPC++ code migrated by the DPCT retained performance of CUDA
- **cuda** performed poorly on Intel GPU (Fig. 7)
  - New sets were tested - **ker2** and **ker11** were the best performing (Fig. 8)

	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
ker2	16	16	64	64	8	32	8	8	32
ker11	12	4	48	48	2	24	2	24	2

Table 1: GEMM Algorithm Constants

MAGMA TO DPC++ PORTABILITY ON MULTICORE CPUS

- Migrated DPC++ code ran successfully on multicore CPUs
  - Achieved 100% usage, confirmed with htop
- Migrated code demonstrated impressive performance
  - Able to outperform higher-bound MKL on the AMD CPU (MKL is tuned for Intel CPU)
  - Outperformed lower-bounds on Intel CPU
- DPC++(MAGMA) is the migrated MAGMA SGEMM code, DPC++(CUDA) is a migrated generic CUDA SGEMM code [4]

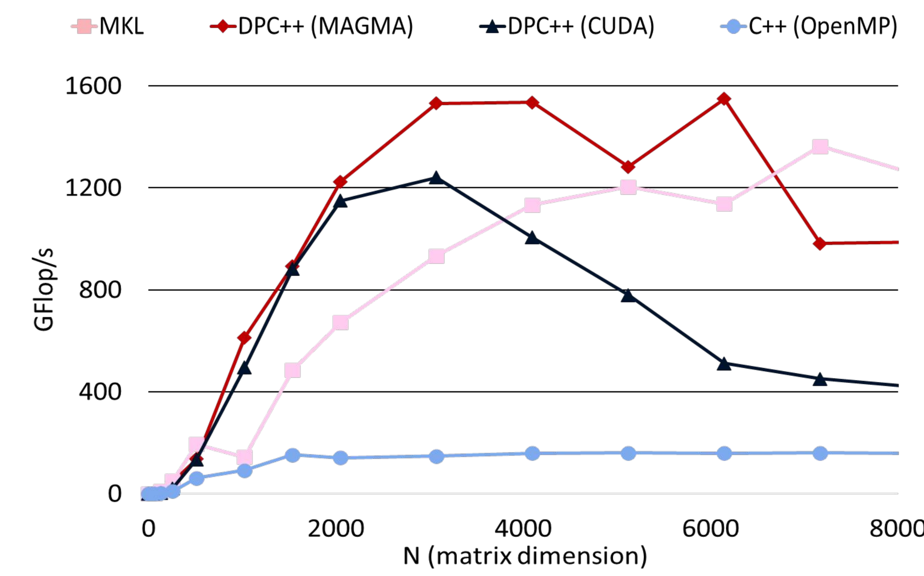

Figure 4: SGEMM Performance on AMD EPYC 7742 64-Core Processor @ 2.25GHZ

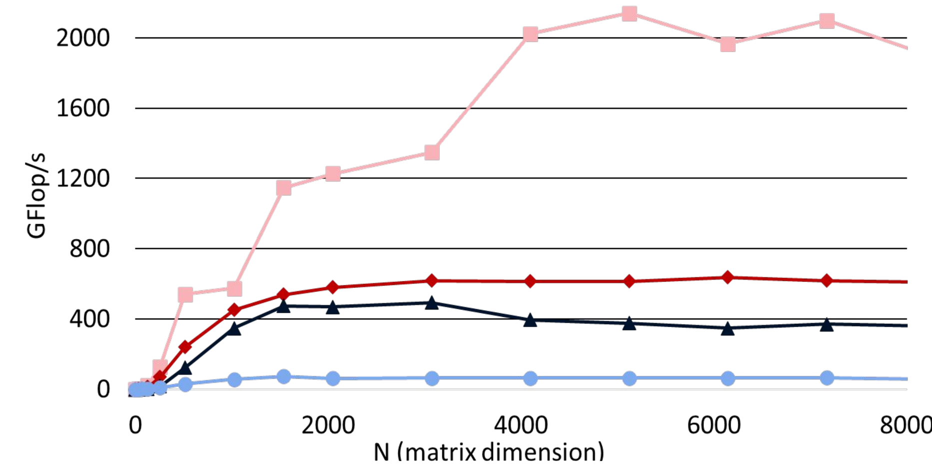

Figure 5: SGEMM Performance on INTEL® XEON® CPU E5-2698 V4 20-Core Processor @ 2.20GHZ

MAGMA TO DPC++ PORTABILITY ON NVIDIA GPUS

- Initial migrated code ran successfully on the Nvidia GPU with the DPC++-LLVM compiler
  - Achieved 100% usage, confirmed with watch -n0.5 nvidia-smi
- **cuda** constants remained optimal
- Achieved performance similar to that of MAGMA, which is optimized for Nvidia GPUs
- CUDA denotes the generic SGEMM algorithm

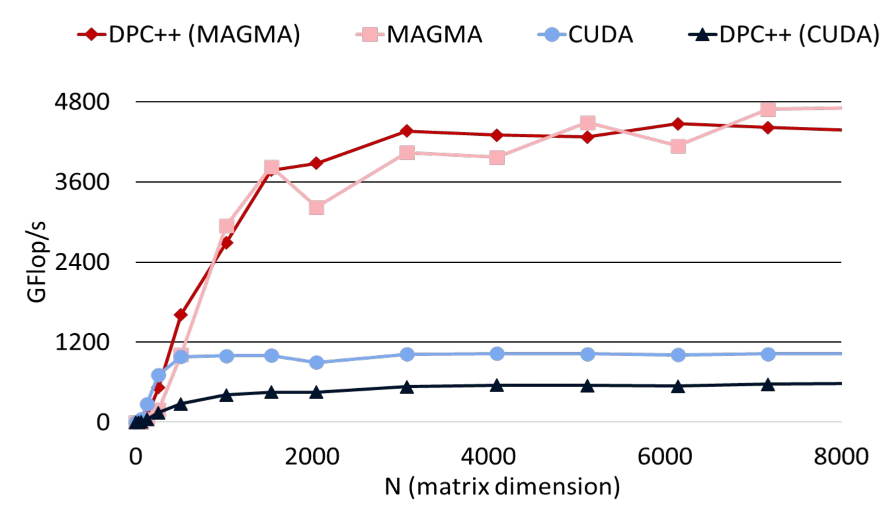

Figure 6: SGEMM Performance on NVIDIA GeForce RTX 3060

MAGMA TO DPC++ PORTABILITY ON INTEL GPUS

- Initial migration performance results were very poor (Fig. 6) - migrated MAGMA SGEMM code never exceeded two G/Flops
- **cuda** constants no longer optimal for the hardware
- Sets **ker2** and **ker11** increased the performance more than ten times (Fig. 7)
- Optimal algorithm parameters are yet to be determined
  - Without specific details about the architecture, parameters combinations are tested via trial and error

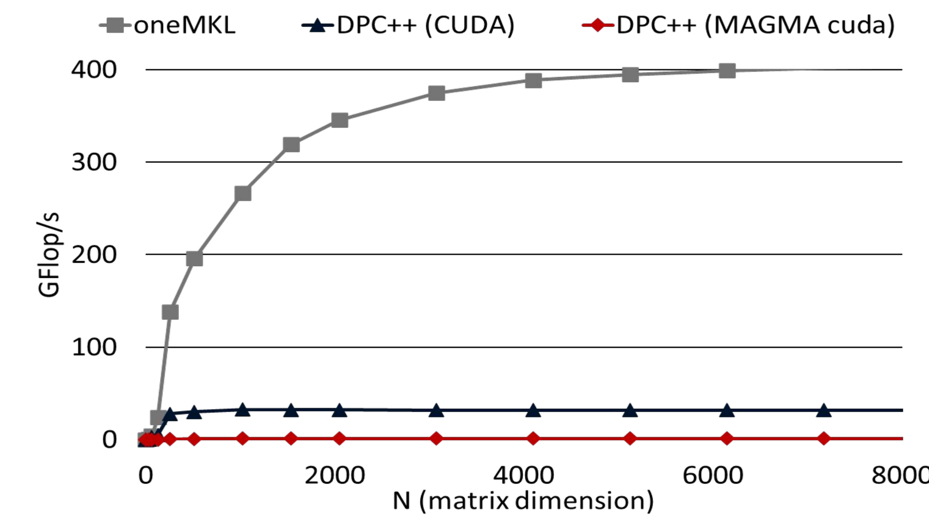

Figure 6: Initial SGEMM Performance on Intel UHD Graphics P630 [0x3e96]

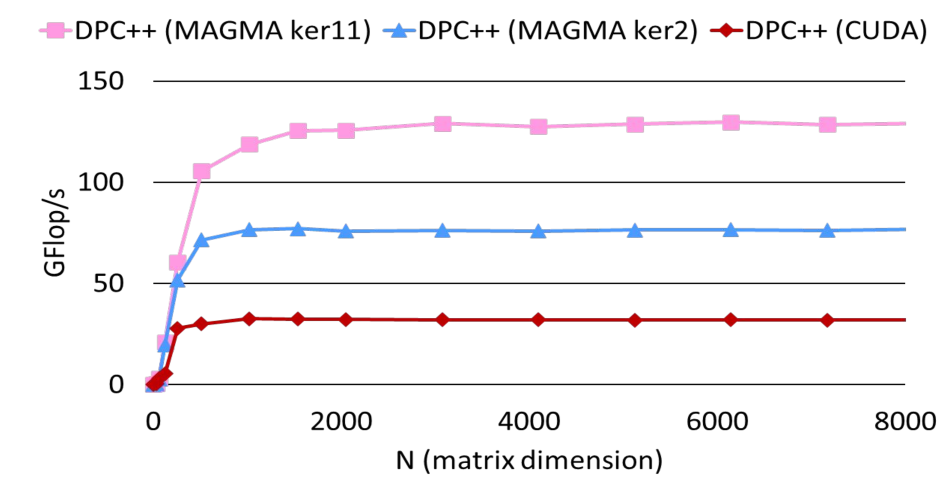

Figure 7: Tuned SGEMM Performance on Intel UHD Graphics P630 [0x3e96]

CONCLUSION AND FUTURE DIRECTIONS

- oneAPI is a promising approach for portable, parallel programming on heterogeneous systems and various architectures
- DPCT useful for initial port of CUDA to DPC++
- Large numerical libraries written in CUDA 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 well on multicore CPUs and retains performance on NVIDIA GPUs
- Further sets of constants must be tested using autotuning techniques to discover the best performing versions for taking full advantage of the computational capabilities of the Intel GPU for GEMM algorithms

REFERENCES

[1] June 2022 | TOP500. Top500 The List. Retrieved August 5, 2022 from https://www.top500.org/lists/top500/2022/06/
[2] 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
[3] 2022. Compiling SYCL* for Different GPUs. Intel. Retrieved August 5, 2022 from https://www.intel.com/content/www/us/en/developer/articles/tool/opencl-drivers.html
[4] 2022. NVIDIA/cuda-samples: Samples for CUDA Developers . - GitHub. GitHub. Retrieved August 5, 2022 from https://github.com/NVIDIA/cuda-samples
[5] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An Improved Magma Gemm For Fermi Graphics Processing Units. The International Journal of High Performance Computing Applications 24, 4 (2010), 511-515. DOI:https://doi.org/10.1177/1094342010385729