PyMAGMA: A Python Interface for MAGMA

Julian Halloy
University of Tennessee
Knoxville, TN, USA
jhalloy@utk.edu

Stephen Qiu
University of Tennessee
Knoxville, TN, USA
sqiu4@vols.utk.edu

Stanimire Tomov
NVIDIA
Santa Clara, CA, USA
tomov@icl.utk.edu

Kwai Wong
University of Tennessee
Knoxville, TN, USA
kwong@utk.edu

ABSTRACT

The Matrix Algebra for GPU and Multicore Architectures (MAGMA) library is used for dense linear algebra computations on both CPUs and GPUs. MAGMA offers an extensive selection of linear algebra routines similar to those found in BLAS and LAPACK, but optimized and specifically designed for GPU acceleration. However, as MAGMA is a C library, C/C++ and MAGMA's C function interfaces can be challenging for non-computer scientists to learn and use. To address this problem, PyMAGMA provides a user-friendly Python interface. PyMAGMA employs C++ object-oriented programming methodologies to simplify the interfaces, particularly by reducing the number of arguments required for routines and eliminating the need for specifying the floating-point data format and arithmetic precision used. In particular, function overloading is employed to abstract routines and avoid precision specification. We show that the PyMAGMA library provides access to high-performance linear algebra using GPUs, while offering the typical benefits of Python, including ease of learning, versatility of use, extensive access to external libraries, large community and support, cross-platform compatibility, and capability for interactive work.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented frameworks*; **Software libraries and repositories**; • **General and reference** → *Performance*; • **Mathematics of computing** → *Solvers*; **Mathematical software performance**; • **Computing methodologies** → **Linear algebra algorithms**.

KEYWORDS

SWIG, Interface, Wrapper, C++, MAGMA, PyMAGMA, Python

ACM Reference Format:
Julian Halloy, Stephen Qiu, Stanimire Tomov, and Kwai Wong. 2024. PyMAGMA: A Python Interface for MAGMA. In *Practice and Experience in Advanced Research Computing (PEARC '24), July 21–25, 2024, Providence, RI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626203.3670607

PEARC '24, July 21–25, 2024, Providence, RI, USA

ACM ISBN 979-8-4007-0419-2/24/07
https://doi.org/10.1145/3626203.3670607

Matrix Size	CPU (MKL)	GPU (MAGMA)
1088	42.25	609.80
5184	99.14	28780.97
10304	101.46	36749.73

Table 1: Double-precision matrix-matrix multiplication (dgemm) performance in GFlop/s for two Intel Xeon Silver 4309Y CPUs using Intel MKL vs. one Nvidia H100 PCIe GPU using MAGMA.

1 BACKGROUND

Linear algebra is a vital tool in scientific computing and finds applications in areas like machine learning and artificial intelligence. Utilizing GPUs as a computational resource has become increasingly advantageous with advancements in GPU technology. The MAGMA library [7] offers a framework for linear algebra that employs the GPU for significant speedup in routine calculations. Table 1 illustrates the significant performance improvement that can be achieved by using GPUs when comparing Double-precision GEneral Matrix Multiplication (DGEMM) with MAGMA (on GPUs) versus with Intel's MKL (on CPUs). Note that in this case the acceleration is up to 377× when comparing the latest H100 GPU from Nvidia with Thermal Design Power (TDP) of 350 Watts vs. two Intel Xeon Silver 4309Y CPUs with TDP of 210 Watts. In terms of Flops per Watt the GPU in this case is 226× more energy efficient. Our goal is to develop PyMAGMA that simplifies the MAGMA interfaces by adding object-oriented designs, callable from Python, while retaining the high-performance and energy efficiency of MAGMA across different GPU architectures.

2 INCORPORATING C++ OBJECT-ORIENTED DESIGNS INTO MAGMA

C++ is an object-oriented programming language that builds upon the syntax and features of the C language. While MAGMA primarily uses C and adheres to the BLAS and LAPACK interfaces, certain parts of the library utilize C++ templates for added functionality. To further enhance the accessibility and ease-of-use of MAGMA, a C++ interface was created via the MAGMA Templates library [2], which, like PyMAGMA, draws on existing state-of-the-art linear

algebra libraries such as MAGMA, SLATE [3], Trilinos [8], and vendor-optimized math libraries. By providing a simple, unified API for the latest algorithms and architecture-specific optimizations in MAGMA, PyMAGMA streamlines the programming process for Python users. Although PyMAGMA was not directly constructed as a Python interface for MAGMA Templates, we adopt the interface designs used in MAGMA Templates and SLATE, while adding in MAGMA custom C++ classes to create simple objects that can be utilized within the Python interface. By leveraging the object-oriented features inherent in Python, we are able to better abstract MAGMA routines in the PyMAGMA interface. Additionally, the use of C++ facilitates the creation of a reusable framework, ultimately reducing development time. As a result, the process of deriving PyMAGMA from MAGMA is now automated, and new developments in MAGMA will be automatically integrated into PyMAGMA.

The auxiliary objects added to MAGMA through C++ classes to facilitate the PyMAGMA development are described in detail below.

2.1 Vector Class

The Vector class added to MAGMA is illustrated in Listing 1. It is primarily used as a wrapper for the pointer to the matrix's data (denoted by data_) and the data size (denoted by n). The class is templated, so the pointer can be to real single-precision, real double-precision, complex magmaFloatComplex, or complex magmaDoubleComplex data. The class keeps track of which device the data is on with an integer called device_. This integer can be negative one for host (CPU), or any integer greater than or equal to zero that corresponds to the index of a GPU on the system. Finally, the class has an own_ Boolean to know whether the data contained in the pointer is owned by the object. This is used in the destructor to determine whether or not to free the memory when an object is destroyed.

The Vector class has three constructors. The first constructor simply takes the number of elements desired and allocates the memory. The second constructor wraps around an existing pointer in memory. The copy constructor takes the data from another Vector object and copies it to a newly allocated location in memory. The class also has assignment operator overloads for copying the data from another pointer and setting all the values to a constant.

Listing 1: Vector class member variables

```
template< typename FloatType >
class Vector {
 public:
    magma_int_t n;
    FloatType *data_;
    int device_;
    bool own_;
}
```

2.2 Matrix Class

The primary class in PyMAGMA is the Matrix class illustrated in Listing 2. It is a wrapper around the underlying Vector object to the data and contains important information such as the size of the matrix and leading dimension values. The class is templated to allow for the different precision types. This abstraction allows the information native to a matrix to be grouped together in a Matrix object and minimizes the number of arguments required from the user to call a routine. This also prevents the user from having to keep track of any pointers themselves. The values of the matrix are stored with column-major ordering. The class keeps track of whether the matrix should be transposed or not with the op_ variable.

Listing 2: Matrix class member variables

```
template< typename FloatType >
class Matrix {
public:
    magma_int_t m, n, ld;
    Vector<FloatType> data_;
    Op op_;
}
```

The primary constructor for the class takes the two dimension sizes and allocates memory on the CPU. If the user passes an optional device integer that does not equal negative one, then memory is allocated on the GPU with magma_malloc() instead of C++'s new operator.

2.3 Complex Class

The Complex class was created for the user to directly interact with complex numbers and set them as necessary. The Complex class is templated so that it can be of type magmaFloatComplex or magmaDoubleComplex. The class's primary constructor takes the real and imaginary values of the complex number and assigns them. The assignment overload takes another Complex object and assigns the real and imaginary values to the current object.

2.4 Function Overloading

Function overloading is crucial in making the interface simple. By creating multiple functions for a single routine, the user no longer needs to specify the type in the routine name. For example, simply calling "gemm" instead of "sgemm", "dgemm", "cgemm", or "zgemm". The type is determined by the Matrix objects that are passed as arguments.

3 DERIVING PYMAGMA FROM MAGMA USING SWIG

The Simplified Wrapper and Interface Generator (SWIG) [1] allows for the creation of a Python interface to C++ and C. Interface files are how SWIG knows which functions, types, and classes of the code should be wrapped. The interface file contains information such as the package and module name, header files to include, function definitions, and more. SWIG takes an interface file and creates a wrapper file with the suffix _wrap.cxx and an import file with the suffix .py. The wrapper file is then compiled to create an object file with the suffix _wrap.o. Finally, the object file and the shared library for the program being wrapped (libmagma.so) can be combined to create a module's shared library.

3.1 Matrix Interface File

The matrix.i file contains the interface data for the Matrix C++ class. It implements several Python magic methods, which are functions called internally by Python. For example, the method __str__() is called when the user calls Python's print() function. In the matrix interface file, __str__() is defined to loop through the matrix and print out each value so the user can print matrices in Python. Python's __getiem__() and __setiem__() are also implemented so we can access individual values in a matrix with bracket indexing.

The interface file must specify which types a templated class accepts with the directive "%template". The matrix.i file does this for Float, Double, magmaFloatComplex, and magmaDoubleComplex types. It maps each of these types to sMatrix, dMatrix, cMatrix, and zMatrix, respectively. The letter at the beginning representing the data type follows the naming convention that MAGMA uses. The interface file also expands upon the Matrix class to take a NumPy array and set its values to the Matrix object. This is done with the Assign() method. With this functionality, the user can utilize the already powerful NumPy library in combination with the PyMAGMA library.

3.2 Routine Interface Files

The routine interface files contain the overloaded functions of the routines to be called by the user. An example of this is the gemm.i file. The gemm() function defined in this file can take two Matrix objects of any of the four types. It will then call either magma_sgemm(), magma_dgemm(), magma_cgemm(), or magma_zgemm(). If the matrices are on the CPU, they will be copied to the GPU before calling the routine. Also, the function allows for the resulting matrix to be preallocated and passed as an argument for faster computation or allocated by the routine and returned to the user.

4 PYMAGMA

For initialization, magma.init() should be called before running any magma routines. Every magma.init() call must be paired with a magma.finalize() call. A matrix can be created by calling its constructor. The constructors for each precision are sMatrix(), dMatrix(), cMatrix(), and zMatrix(). Also, a matrix can be assigned with a NumPy matrix of the corresponding type.

The gemm() function is a wrapper of the magma_*gemm MAGMA routines. It is to be called with two or three Matrix objects of the same type and a queue created with 'magma.getqueue()'. It also can optionally take alpha and beta values, which default to zero and one, respectively. The PyMAGMA gemm routine copies the matrices to the GPU if they are on the CPU, calls the respective MAGMA routine, and then copies the result back to the CPU if the initial matrices were on the CPU. The function either returns the resulting C matrix from the matrix multiplication or sets the C matrix that was passed as an argument to the result. All PyMAGMA routines are synchronous, so they will only return once the computation is completed.

```
magma.init()
# get magma queue
queue = magma.getqueue()
A = magma.sMatrix(2,2) #creates a 2x2 matrix
B = magma.sMatrix(2,2) #creates a 2x2 matrix
# call gemm
C = magma.gemm(A, B, queue)
magma.finalize()
```

5 PERFORMANCE

In building an interface for a library that focuses heavily on speed, the verification that performance is maintained is crucial. To check this, DGEMM and SGEMM routines were run on both MAGMA and PyMAGMA with matrices of increasing sizes. The matrices in PyMAGMA were allocated on the GPU before the GEMM call, so the wrapper code did not have to allocate any new matrices or copy data from the CPU to the GPU. The tests were run on Nvidia's H100 GPU.

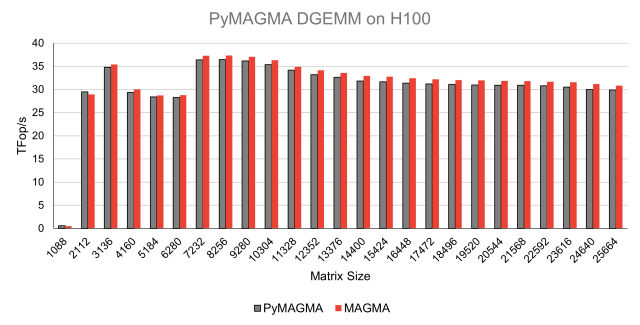

Figure 1: DGEMM performance in TFlop/s using PyMAGMA and MAGMA on one Nvidia H100 GPU

The resulting TFlop/s values matched between the two libraries with little differences. In fact, since PyMAGMA is occasionally faster than MAGMA it is likely that the discrepancies seen are due to resource utilization by other processes or slight differences in the timing algorithm used. From this, it is reasonable to conclude that PyMAGMA does not produce any significant overhead that hinders the performance of MAGMA. MAGMA is also compatible with AMD GPU architectures. We tested PyMAGMA on the Instinct MI210, which is one of AMD's latest professional GPUs for high-performance computing applications. Similar results between PyMAGMA and MAGMA were achieved.

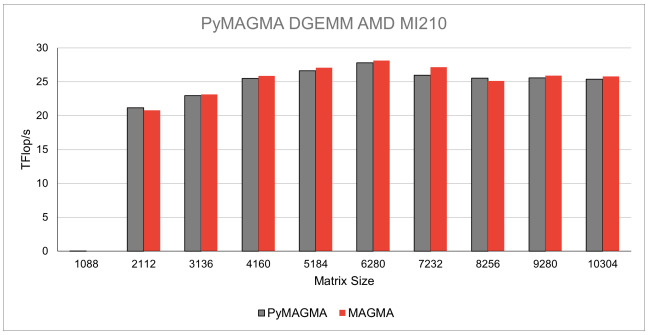

Figure 2: DGEMM performance in TFlop/s using PyMAGMA and MAGMA on one AMD MI210 GPU

6 FUTURE WORK

Going forward, we hope to expand PyMAGMA to encompass all of the routines in MAGMA. This includes batched routines, for which it may be useful to create a new class for batched matrices. Another goal is to use the techniques for PyMAGMA to create a Python interface for MagmaDNN [5]. MagmaDNN is a Deep Learning Framework in C++ that utilizes the linear algebra routines of MAGMA. A Python interface would expand the usability and reach of MagmaDNN.

ACKNOWLEDGMENTS

PyMAGMA was sponsored through the Research Experiences for Undergraduates (REU) award no. 2050534 of the National Science Foundation. Additional support was given by the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville (UTK). Special acknowledgment goes to Delario Nance, Jr. who began work on PyMAGMA in 2022 [4]. ICL's MAGMA Templates and the Software for Linear Algebra Targeting Exascale (SLATE) acted as reference for the object-oriented principles used in C++ linear algebra software. PyMFEM [6] from the Lawrence Livermore National Laboratory acted as a reference for utilizing SWIG to create a Python interface for an existing software library. We thank NVIDIA and AMD for their support and hardware donations.

REFERENCES

[1] David M. Beazley. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4* (Monterey, California) *(TCLTK'96)*. USENIX Association, USA, 15.

[2] Mohammed Al Farhan, Ahmad Abdelfattah, Stanimire Tomov, Mark Gates, Dalal Sukkari, Azzam Haidar, Robert Rosenberg, and Jack Dongarra. 2020. MAGMA templates for scalable linear algebra on emerging architectures. *The International Journal of High Performance Computing Applications* 34, 6 (2020), 645–658. https://doi.org/10.1177/1094342020938421 arXiv:https://doi.org/10.1177/1094342020938421

[3] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

[4] Delario Nance, Stanimire Tomov, and Kwai Wong. 2022. A Python Library for Matrix Algebra on GPU and Multicore Architectures. In *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. 770–775. https://doi.org/10.1109/MASS56207.2022.00121

[5] Daniel Nichols, Nathalie-Sofia Tomov, Frank Betancourt, Stanimire Tomov, Kwai Wong, and Jack Dongarra. 2019. MagmaDNN: Towards High-Performance Data Analytics and Machine Learning for Data-Driven Scientific Computing. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode (Eds.). Springer International Publishing, Cham, 490–503.

[6] The PyMFEM Project Team. 2024 (accessed March 2, 2024). *MFEM + PyMFEM (FEM library)*. https://github.com/mfem/PyMFEM

[7] S. Tomov, J. Dongarra, and M. Baboulin. 2010. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parellel Comput. Syst. Appl.* 36, 5-6 (2010), 232–240. DOI: 10.1016/j.parco.2009.12.005.

[8] The Trilinos Project Team. 2024 (accessed March 5, 2024). *The Trilinos Project Website*. https://trilinos.github.io