A Python Library for Matrix Algebra on GPU and Multicore Architectures

1st Nance, Jr., Delario
Mathematics and Computer Science
Davidson College
Davidson, United States
denance@davidson.edu

2nd Tomov, Stanimire
Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, United States
tomov@icl.utk.edu

3rd Wong, Kwai
National Institute for Computational Sciences
University of Tennessee, Knoxville
Knoxville, United States
kwong@utk.edu

2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS) | 978-1-6654-7180-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/MASS56207.2022.00121

***Abstract*—Despite C/C++ and Python both being very popular programming languages, each tool possesses unique advantages and disadvantages. Notably, computers can run C/C++ code very quickly, but C/C++ code has to first be compiled and the syntax can be difficult for new programmers to understand. Python code, however, sacrifices speed for an easy-to-understand syntax and can be run interactively. Thankfully, it is possible to combine the benefits of Python and C/C++. For example, NumPy is a popular package of linear algebra operations written in C but used with Python. Such a combination allows programmers to not only utilize the fast speeds of C code but also Python's simple syntax. NumPy's potential, however, is limited by its inability to run on graphics processing units (GPUs), processors specialized for handling computations. On the other hand, a linear algebra library known as Matrix Algebra on GPU and Multicore Architectures (MAGMA) is suited for running its code on GPUs. Coupled with the fact that its code is written in C/C++, MAGMA offers extremely fast computations. To combine MAGMA's speed with Python's easy-to-understand syntax, we researched how to use C++ code with Python. By researching a tool known as Simplified Wrapper and Interface Generator (SWIG), we created PyMAGMA - a library of chosen MAGMA functions which can be imported in Python 3.9 for use.**

***Index Terms*—BLAS, C++, MAGMA, Python, SWIG, wrapper**

I. Background

A. Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) is a package of routines for performing standard linear algebra operations involving vectors and matrices. Originally written in Fortran, BLAS routines are separated into three levels. Level 1 BLAS contains routines for vector-vector operations, such as the dot product [5]. Contrarily, Level 2 BLAS has routines for matrix-vector operations, including GEMV - the GEneral Matrix-Vector product [3]. Level 3 BLAS contains routines for matrix-matrix operations (e.g., GEMM - the GEneral Matrix-Matrix product) which, through block matrix multiplication, become more memory efficient than Level 1 and 2 routines when ran on "high-performance" computers [2].

B. Matrix Algebra on GPU and Multicore Architectures

Matrix Algebra on GPU and Multicore Architectures (MAGMA) is a computational library of C++ functions for performing linear algebra operations such as BLAS routines, LU decompositions, linear system solvers, and eigenvalue problem solvers [8]. MAGMA's main advantage over other linear algebra libraries, such as the Linear Algebra PACKage (LAPACK) [1] and NumPy [4], is that it contains not only functions which can run on central processing units (CPUs) but also functions which can run on graphics processing units (GPUs). Whereas CPUs are computer processors tasked with most processing roles like handling input and output (I/O), GPUs focus on performing computations, resulting in GPUs running code much faster than CPUs. Because LAPACK code is designed to run on CPUs but not GPUs, MAGMA redesigns the LAPACK algorithms to perform efficiently on GPUs. Thus, when using LAPACK on an Intel® Xeon® CPU X5650 and MAGMA on a NVIDIA GeForce GTX 1650 SUPER to perform Single-precision GEMM (SGEMM) on random square matrices with sizes not exceeding 10304 x 10304, MAGMA performs approximately ten times faster than LAPACK (Fig. 1).

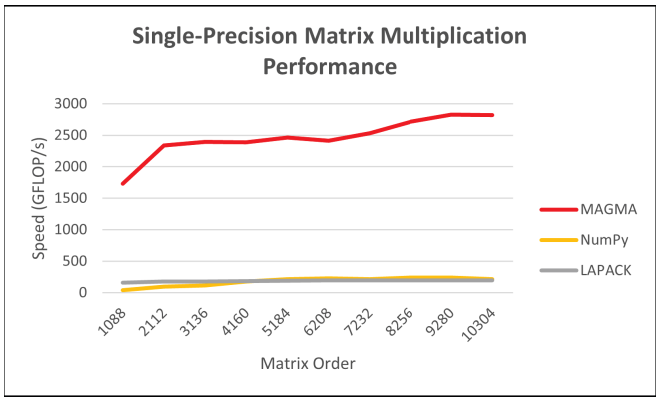

Fig. 1. Performing single-precision matrix multiplication with LAPACK (SGEMM), MAGMA (SGEMM), and NumPy (MatMul)

C. Simplified Wrapper and Interface Generator

Simplified Wrapper and Interface Generator (SWIG) is one of many tools for interfacing C/C++ code with other programming languages. For example, programmers can use SWIG to create interfaces through which C/C++ functions can be used

National Science Foundation (NSF)

2155-6814/22/$31.00 ©2022 IEEE
DOI 10.1109/MASS56207.2022.00121

in Python. Unlike other interface tools, however, SWIG can generate interfaces in many high-level languages (e.g., Java, Perl, Ruby, PHP), not only Python [9]. This unique feature makes SWIG suited for programmers who might interface C/C++ functions with multiple languages in the future.

For Python in particular, SWIG builds interfaces by generating three files: a wrapper file containing code for translating C/C++ functions to the Python interpreter, a shared library containing the compiled C/C++ code to interface as well as the compiled wrapper file's code, and a Python file allowing users to import the shared library into Python and use the C/C++ functions inside.

II. Simplified Wrapper and Interface Generator Workflow

To illustrate the process of using SWIG to generate a Python interface for a library of C++ functions, we give high-level descriptions of the main files involved when using SWIG on a Linux machine (Fig. 2). For more details on interfacing C functions and C++ classes, using SWIG on Windows, how to use SWIG with different target languages, or how SWIG works internally, please refer to the SWIG 4.0 Documentation [7].

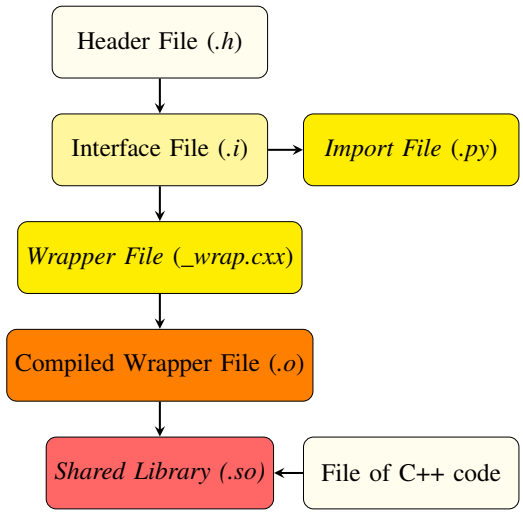

Fig. 2. A flowchart of files created when building Python interfaces for C++ code with SWIG

A. Installation

To install SWIG on the Linux operating system, users can type `apt-get install swig` in the command line and then press `Enter`. To check if SWIG's latest version (4.0.2 as of July 2022 [9]) was installed, users should input `swig -version` into the Linux command line.

B. Header File (.h)

To use SWIG after installing it, the user should decide what C++ functions they wish to interface with Python. Once the functions have been chosen, a header file must be created (Fig. 2). This file should contain the declarations (or definitions) of all the C++ functions to interface with Python. In addition to function declarations, the header file should include any macro definitions or typedefs used by the C++ functions.

With the header file, a SWIG user can organize C++ functions which they want to interface into a single file. By maintaining this file, the user can easily add functions to or remove functions from the created Python interface by adding or removing its declaration/definition in the header file and then following Fig. 2 to recreate the interface's shared library (*.so*).

C. Interface File (.i)

After the SWIG user creates a header file (*.h*) for the C++ functions which they wish to interface with Python, the user must create a special SWIG file known as the interface file (Fig. 2). According to Sec. 5.7.2 of [7], the interface file should contain a `#include` statement and SWIG's `%include` directive for the header file and the name of the Python interface which the user wants to create. Optionally, however, SWIG features known as "typemaps" can be added to customize how SWIG's wrapper code will convert between C++ and Python data types. Typemaps are further discussed in Sections 12 and 13 of [7].

D. Import File (.py)

The Python file which we will refer to as the "import file" contains Python's `import` command, which will let users import the interfaced C++ functions into Python once the shared library file (*.so*) is created. Also, inside the import file is a Python function for each C++ function whose declaration or definition is in the header file. Each of these Python functions will call the corresponding C++ function inside the shared library, letting Python users use a desired C++ function by simply calling its Python counterpart. To create the import file with the Linux command line, the user should use the SWIG command `swig -c++ -python NAME.i`, where `NAME.i` represents the name of the interface file (*.i*).

E. Wrapper File (_wrap.cxx)

SWIG generates the wrapper file after the user inputs into the Linux command line the same SWIG command used to create the import file (*.py*) (Fig. 2). Inside the wrapper file is namesake wrapper code for translating the C++ functions, which were declared in the header file (*.h*), to the Python interpreter. When the user calls a Python function from the import file (*.py*), the wrapper code converts the function inputs to their equivalent C++ data types, calls the corresponding C++ function with the generated C++ inputs, converts the return value into its equivalent Python data type, and then returns the Python value. If a user wants to customize specific type conversions in the wrapper file, the user should enforce the corresponding typemaps in the interface file (*.i*). For more detail on SWIG's wrapper code, please refer to Sections 4.2 and 5.2 in [7].

F. Compiled Wrapper File (.o)

Before the SWIG-generated wrapper code can translate C++ functions to the Python interpreter, the code must first be compiled into object code (Fig. 2). To compile the wrapper file (*_wrap.cxx*), users can try running the Linux command `g++ -fPIC -c NAME_wrap.cxx PATH_TO_PYTHON`, where `NAME_wrap.cxx` represents the name of the wrapper file, and `PATH_TO_PYTHON` represents the path to the folder containing *Python.h* on the user's Linux machine. According to Section 6.4 in [7], the Linux command used to compile the wrapper file differs across machines.

G. Shared Library (.so)

Assuming the user has a library containing object code for the C++ functions declared in the header file (*.h*), they can create the Python interface's shared library with the Linux command `ld -shared OBJECT_LIBRARY COMPILED_WRAPPER.o -o _MODULE.so`. In this command, `OBJECT_LIBRARY` represents the path to the existing library of C++ object code to use with Python, `COMPILED_WRAPPER.o` represents the name of the compiled wrapper file (*.o*), and `MODULE` is the name of the Python interface specified in the interface file (*.i*). The shared library file will be named `_MODULE.so`. After running the Python command `import MODULE` (where `MODULE` is the interface name defined in the interface file), the SWIG user can call the interfaced C++ functions from Python.

III. Generating PyMAGMA

We now discuss the process of creating the first version of PyMAGMA, our SWIG-generated library of C++ functions from MAGMA to be used with Python 3.9. While the first version of PyMAGMA could be successfully imported into Python, we could not use it to call MAGMA functions containing pointer arguments. Our work to solve this problem is detailed in Section IV.

A. Header File (pymagma.h)

To eventually interface many MAGMA functions in Python, we first tried interfacing twenty-one MAGMA functions required for performing many of MAGMA's GPU computations (e.g., GEMM). Notably, our first header file contained declarations of C++ functions for managing memory, managing queues, sending data between CPUs and GPUs, and managing the GPU in use (Listing 1). Additionally, the header file contained declarations for Double-precision GEMM (DGEMM) and Double-precision GEneral TRiangular Factorization (DGETRF).

```
magma_malloc // Dynamically allocates GPU memory
magma_malloc_cpu // Dynamically allocates CPU memory
magma_free_cpu // Frees allocated CPU memory
magma_free_internal // Frees allocated GPU memory
magma_getdevice // Returns the ID of the GPU in use
magma_setdevice // Sets the GPU to use with MAGMA
magma_getmatrix_internal // Sends a matrix from GPU to CPU
magma_setmatrix_internal // Sends a matrix from CPU to GPU
magmablas_dgemm // Performs DGEMM on GPU
```

Listing 1. Sample C++ functions declared in our first *pymagma.h* header file

B. Interface File (pymagma.i)

Our interface file (*pymagma.i*) for the first version of PyMAGMA contained an `include` statement and `include` directive for the *pymagma.h* header file and the name of the Python library we wanted to create: PyMAGMA (Listing 2). We did not enforce any SWIG typemaps.

```
// Naming the PyMAGMA library
%module pymagma
%{
    #include "pymagma.h"
%}
%include "pymagma.h"
```

Listing 2. The contents of our *pymagma.i* interface file

C. Import File (pymagma.py)

```
def magma_print_environment():
    return _pymagma.magma_print_environment()
def magma_malloc(ptr_ptr, bytes):
    return _pymagma.magma_malloc(ptr_ptr, bytes)
```

Listing 3. Sample Python functions in the *pymagma.py* import file for calling the interfaced C++ functions from MAGMA

After creating the *pymagma.h* header file and *pymagma.i* interface file, we used SWIG to generate our *pymagma.py* import file. With this file, we could try importing the first version of PyMAGMA after building it. For each C++ function which we declared in the *pymagma.h* header file, our import file contained a Python function for calling the C++ function (Listing 3). To create the *pymagma.py* import file and *pymagma_wrap.cxx* wrapper file, we entered the following SWIG command into the Linux terminal: `swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i` (Listing 4).

```
swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i
g++ -fPIC -c pymagma_wrap.cxx
    -I/home/user1/anaconda3/include/python3.9
ld -shared /home/user1/magma/lib/libmagma.so
    /usr/lib/x86_64-linux-gnu/libopenblas.so pymagma_wrap.o
    -o _pymagma.so
```

Listing 4. The Linux commands used to generate *pymagma.py* and *pymagma_wrap.cxx*, *pymagma_wrap.o*, and *_pymagma.so*

D. Wrapper File (pymagma_wrap.cxx)

```
SWIGINTERN PyObject *_wrap_magma_malloc(...) {
...
    arg2 = (size_t)(val2);
    result = (magma_int_t)magma_malloc(arg1,arg2);
    resultobj = SWIG_From_int((int)(result));
    return resultobj;
fail:
    return NULL;
}
```

Listing 5. Low-level wrapper code for MAGMA's *magma_malloc()* function in the *pymagma_wrap.cxx* wrapper file

As its name suggests, our *pymagma_wrap.cxx* wrapper file contained wrapper code for translating our chosen MAGMA functions to Python's interpreter (Listing 5). To create the *pymagma_wrap.cxx* wrapper file, we used the same SWIG command for creating our *pymagma.py* import file (Listing 4). Initially, we did not include `-DSWIG_NO_CPLUSPLUS_CAST` in the Linux command; however, not doing so resulted in an error when trying to compile the wrapper file (Listing

6). Specifically, SWIG's use of C++ typecasting (i.e., static, const, and reintepret) in the wrapper file was invalid, preventing the wrapper file from being compiled. Thus, to force SWIG to use C-style typecasts instead of C++ typecasts, an Innovative Computing Laboratory researcher helped us include `-DSWIG_NO_CPLUSPLUS_CAST` in the Linux command used to generate the *pymagma_wrap.cxx* wrapper file.

```
error: reinterpret_cast from type 'const void**' to type
    'void**' casts away qualifiers
```

Listing 6. The error from trying to compile *pymagma_wrap.cxx* without using *−DSWIG_NO_CPLUSPLUS_CAST* when creating the file

E. Compiled Wrapper File (.o)

To compile our *pymagma_wrap.cxx* wrapper file into the *pymagma_wrap.o* object file, we used the Linux command `!g++ -fPIC -c pymagma_wrap.cxx -I/home/user1/anaconda3/include/python3.9` (Listing 4). In the command, the `-I/home/.../python3.9` path is the path to the *Python.h* file on our Linux machine.

F. Shared Library (_pymagma.so)

In the first version of PyMAGMA, our *_pymagma.so* shared library contained object code from the *pymagma_wrap.o* file and *libmagma.so* file - a shared library of object code for various C++ functions in MAGMA. To create *_pymagma.so*, we successfully ran the Linux command `ld -shared /home/user1/magma/lib/libmagma.so pymagma_wrap.o -o _pymagma.so` (Listing 4).

G. Testing

```
>>> import pymagma as pmg
>>> pmg.magma_init() # 0 represents success
0
>>> pmg.magma_finalize() # 0 represents success
0
```

Listing 7. Successfully calling *magma_init()* and *magma_finalize()* with PyMAGMA

By using SWIG to generate the *pymagma_wrap.cxx*, *pymagma.py*, and *_pymagma.so* files, we successfully built our first version of the PyMAGMA library. To test the functionality of that PyMAGMA version, we tried calling three C++ functions which we declared in the *pymagma.h* header file: `magma_init()`, `magma_print_environment()`, and `magma_finalize()`. We successfully called `magma_init()` and `magma_finalize()` through PyMAGMA in a Python environment on the Linux terminal (Listing 7), but `magma_print_environment()` was not completely functional with PyMAGMA. Specifically, when trying to call `magma_print_environment()`, correct output was displayed before the warning **** stack smashing detected **** appeared. We believed that data in our local machine's stack memory was somehow getting incorrectly overwritten.

To resolve the issue, we tried creating PyMAGMA on Google Colab and calling `magma_print_environment()` with Google Colab's command line. Correct output was displayed with no warnings (Listing 8), possibly due to Google Colab using SWIG 3.0.12. Specifically, whereas SWIG *4.0.1* was used to create PyMAGMA locally, SWIG *3.0.12* was used to create PyMAGMA on Google Colab. Possibly, SWIG's newer 4.0.1 version contains code for triggering "stack smashing" warnings while the older 3.0.12 version does not. Nevertheless, we desired to call `magma_print_environment()` locally without stack smashing since SWIG 4.0.1 is newer than version 3.0.12. After relinking the object files in the *libmagma.so* library and locally creating PyMAGMA again with the same Linux commands (Listing 4), we could run `magma_print_environment()` without any *stack smashing* warnings (Listing 8).

```
>>> import pymagma as pmg
...
>>> pmg.magma_print_environment()
% MAGMA 2.6.0 svn 32-bit magma_int_t, 64-bit pointer.
Compiled with CUDA support for 3.5
% CUDA runtime 11030, driver 11040. OpenMP threads 24.
% device 0: NVIDIA GeForce GTX 1650 SUPER, 1740.0 MHz clock,
    3910.6 MiB memory, capability 7.5
% Tue Aug  2 14:33:20 2022
```

Listing 8. Successfully calling *magma_print_environment()* with PyMAGMA

IV. Extending PyMAGMA

```
>>> import pymagma
...
>>> address = 0
>>> bytes = 4
>>>
>>> pymagma.magma_malloc(address, bytes)
...
TypeError: in method 'magma_malloc', argument 1 of type
          'magma_ptr *'
```

Listing 9. Attempting to call PyMAGMA's *magma_malloc()* function, which expects a pointer as its first argument, even though Python users cannot normally create pointers

Python users do not normally have the ability to create pointer types, but many MAGMA functions in PyMAGMA require pointer arguments (Listing 9). Therefore, we wanted to give PyMAGMA users the ability to somehow generate pointers in Python. To combat this issue, we created and added to the *pymagma.h* header file C++ functions which act as new versions of some of the original functions in PyMAGMA (Listing 10). Other than not requiring arguments of pointer types, many of these new functions operate nearly identically to their original MAGMA counterparts.

To test PyMAGMA's accuracy of MAGMA's linear algebra routines, we desired the ability to create C++ arrays in Python with values chosen by us, pass those arrays into linear algebra routines, and then print the resulting array to see if its contents are correct. However, Section 5.4.5 of [7] shows that Python users cannot normally index or print interfaced C++ arrays like Python lists. Therefore, to put specific values into C++ arrays and then print them, we created PyMAGMA functions for managing C++ arrays with Python. Furthermore, to test PyMAGMA's speed, we created functions for generating random C++ arrays with Python.

In this section, we will discuss one of the functions which we added to PyMAGMA to generate pointers in Python

(Listing 10) and four of the functions which we created to manage C++ arrays with Python (Listings 11 - Listing 14).

A. *pymagma_malloc()*

```
magma_int_t
magma_malloc( magma_ptr *ptr_ptr, size_t bytes );

void*
pymagma_malloc(size_t bytes)
{
    void* a;
    magma_malloc(&a, bytes);
    return a;
}
```

Listing 10. The definition for the *pymagma_malloc()* function to let PyMAGMA users dynamically allocate GPU memory

The *magma_malloc()* function in MAGMA is designed to dynamically allocate memory on the current GPU but requires a *ptr_ptr* argument of type *magma_ptr** (acts as the *void*** type in C++). To let PyMAGMA users dynamically allocate GPU memory, we created a *pymagma_malloc()* function with a similar purpose to the MAGMA version but returns a pointer to the Python user and does not require the *ptr_ptr* argument (Listing 10). Specifically, *pymagma_malloc()* declares a pointer of type *void**, calls *magma_malloc()* function to allocate a block of GPU memory starting at the address stored in the created void pointer, and finally returns the pointer.

B. *pymagma_sarray_cpu()*

```
float*
pymagma_sarray_cpu( magma_int_t height, magma_int_t width )
{
    void* void_array = pymagma_malloc_cpu(
        sizeof(float) * height * width);
    float* sarray = (float*)void_array;
    return sarray;
}
```

Listing 11. The definition for the *pymagma_sarray_cpu()* function to let PyMAGMA users create arrays of C++ floats in CPU memory

To test PyMAGMA's accuracy, we desired a way to create matrices of C++ floats on CPUs. We achieved this by creating the *pymagma_sarray_cpu()* function (Listing 11). After a Python user passes *height* and *width* arguments into the function, *pymagma_sarray_cpu()* calls our *pymagma_malloc_cpu()* function to dynamically allocate a 1D block of CPU memory for $height * width$ C++ floats, and then returns the base address of that block as a *float** pointer. Despite the allocated block of CPU memory being stored linearly, PyMAGMA users can manipulate the float values stored in the block as if the block was a 2D matrix (Listing 12).

C. *pymagma_sset_cpu()*

```
void
pymagma_sset_cpu( float* A,
                  magma_int_t row, magma_int_t col,
                  magma_int_t lda, float value )
{
    A[row + lda * col] = value;
}
```

Listing 12. The definition for the *pymagma_sset_cpu()* function to let PyMAGMA users change values in arrays of C++ floats in CPU memory

To check if PyMAGMA gives accurate results for MAGMA routines, we wanted a way to test MAGMA routines with specific matrices chosen by us. Therefore, we created the *pymagma_sset_cpu()* function (Listing 12) for editing the values in arrays of C++ floats on CPUs. Specifically, after an user passes A, row, col, lda, and $value$ arguments into the function, the function sets the float $value$ in the position at $row + lda * col$ in the 1D memory block A returned by *pymagma_sarray_cpu()*. From a high-level, 2D matrix perspective, *pymagma_sset_cpu()* simply updates the value at row row and column col in the given matrix A.

D. *pymagma_sprint_cpu()*

```
void
pymagma_sprint_cpu( magma_int_t m, magma_int_t n,
                    const float* A, magma_int_t lda )
{
    magma_sprint( m, n, A, lda );
}
```

Listing 13. The definition for the *pymagma_sprint_cpu()* function to let PyMAGMA users print arrays of C++ floats in CPU memory

To verify PyMAGMA's accuracy of MAGMA routines like SGEMM, we desired a way to print matrices of C++ floats on CPUs. We achieved this by creating the *pymagma_sprint_cpu()* function (Listing 13). After an user passes m, n, A, and lda arguments into the function, *pymagma_sprint_cpu()* calls *magma_sprint()* - an existing MAGMA function which we also added to PyMAGMA - to print the first m rows and n columns of a submatrix with width lda from the 2D representation of the memory block A generated with *pymagma_sarray_cpu()*.

E. *pymagma_slarnv()*

```
void
slarnv_(int* IDIST, int* ISEED, int* N, float* X);

float*
pymagma_slarnv(int dist, int height, int width)
{
    int n = height * width;
    int iseed[4] = {0,0,0,1};
    float* rand_sarray = pymagma_sarray_cpu(height, width);
    slarnv_(&dist, iseed, &n, rand_sarray);
    return rand_sarray;
}
```

Listing 14. The definition for the *pymagma_slarnv()* function to let PyMAGMA users generate random arrays of C++ floats in CPU memory

To test PyMAGMA's speed at performing MAGMA routines (e.g., GEMM), we desired a way to quickly generate large arrays of random C++ floats on CPUs without manually setting values with *pymagma_sset_cpu()*. To do so, we created the *pymagma_slarnv()* function (Listing 14). After an user passes in $dist$, $height$, and $width$ arguments into the function, *pymagma_slarnv()* will first create a linear block of memory representing an empty $height$ x $width$ matrix, Next, the function calls *slarnv_()* - an LAPACK function which we added to PyMAGMA. At each position in the generated array, *slarnv_()* will place a C++ float from a random distribution, depending on the value of $dist$. For example, if $dist$ equals 2, then *slarnv_()* will use random floats from the normal distribution $(-1, 1)$.

V. Testing PyMAGMA

Due to Level 3 BLAS routines being very memory-efficient for "high-performance" computations [2], we evaluated the current state of PyMAGMA by testing the library's accuracy and performance of SGEMM.

A. Accuracy

To test PyMAGMA's accuracy of SGEMM, we first called *pymagma_sarray_cpu()* to allocate CPU memory for three C++ arrays of floats: A, B, and C. Arrays A, B, and C each respectively represented a 5 x 3 matrix, 3 x 5 matrix, and 5 x 5 matrix. Next, we used *pymagma_sset_cpu()* to set float values, chosen by us, at every position in the three arrays. After copying the arrays to GPU memory with a NVIDIA GeForce GTX 1650 SUPER, we obtained the result $C = -AB + C$ by passing the three GPU arrays into *pymagmablas_sgemm()* - a function which we created for calling MAGMA's *magmablas_sgemm()* function. By calling *pymagma_sprint_cpu*, we saw that the resulting array contained the values which we expected, showing that PyMAGMA accurately performed SGEMM.

B. Performance

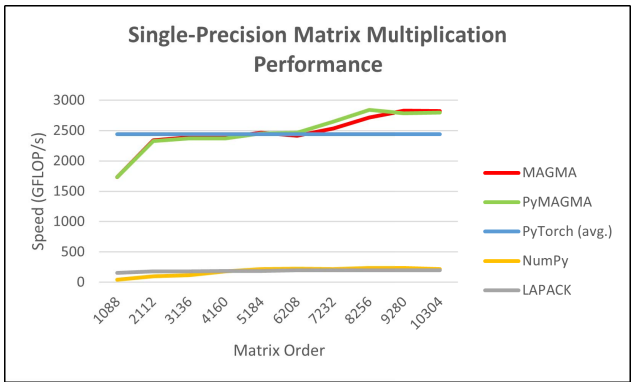

Fig. 3. Performing single-precision matrix multiplication with LAPACK (SGEMM), MAGMA (SGEMM), NumPy (MatMul), PyMAGMA (SGEMM), and PyTorch (MatMul)

To test PyMAGMA's performance of SGEMM, we used *pymagma_slarnv()* to generate three 1088 x 1088 square matrices *A*, *B*, and *C* with random floats from the normal distribution $(-1, 1)$. After copying each of the matrices to a NVIDIA GeForce GTX 1650 SUPER, we calculated the rate (gigaflop/s) at which PyMAGMA performed the computation $C = -AB + C$ with SGEMM. We then compared PyMAGMA's performance to that of MAGMA on the same GPU, as well as LAPACK's performance of SGEMM and NumPy's single-precision performance of MatMul on a Intel® Xeon® CPU X5650. Also, we obtained the single-precision, GPU performance of MatMul with PyTorch, a popular Python machine learning framework [6]. We repeated the performance tests for nine increasing matrix sizes.

After the ten tests, we saw that PyMAGMA's performance was almost identical to that of MAGMA (Fig. 3). Furthermore, like MAGMA, PyMAGMA performed approximately ten times faster than LAPACK and NumPy. Concerning PyTorch, we took its average performance on the NVIDIA GPU from the ten tests because its performance drastically increased and decreased across the tests. While PyTorch's average performance exceeded LAPACK's and NumPy's, (Py)MAGMA eventually outperformed PyTorch (Fig. 3).

VI. Results and Future Direction

By researching how to use the Simplified Wrapper and Interface Generator (SWIG) to interface C++ functions with Python, we successfully created PyMAGMA: a Python library for using chosen MAGMA functions with Python. Furthermore, we see that PyMAGMA obtains similar SGEMM performances to MAGMA for square matrices with sizes not exceeding 10304 x 10304 (Fig. 3). Currently, PyMAGMA contains thirty-two C++ functions from MAGMA as well as two functions from LAPACK and thirty functions which we created when extending PyMAGMA. By successfully defining and using added functions in PyMAGMA, we see that we can easily add functions to the PyMAGMA library by adding their declaration or definition to the *pymagma.h* header file. To use PyMAGMA functions containing pointer arguments without creating new functions, we plan to research how to use SWIG typemaps with Python.

Acknowledgments

Nance thanks God for granting him this opportunity. 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. Nance also thanks Dr. Kwai Wong, Dr. Stanimire Tomov, Julian Halloy, and his fellow 2022 REU participants for their assistance.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK Users' Guide (3rd ed.), Society for Industrial and Applied Mathematics, 1999, ISBN 0-89871-447-8 (paperback)

[2] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1—17.

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 14 (1988), pp. 1—17.

[4] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.

[5] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Soft., 5 (1979), pp. 308—323. Pages 232-240, ISSN 0167-8191.

[6] The PyTorch website. [Online]. Available: https://pytorch.org/.

[7] The SWIG 4.0 Documentation. [Online]. Available: https://www.swig.org/Doc4.0/index.html.

[8] Stanimire Tomov, Jack Dongarra, Marc 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.

[9] (2019, Apr.) The SWIG website. [Online]. Available: https://swig.org/.