PyMAGMA - A Python Library for MAGMA

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

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

3rd Wong, Kwai
Mechanical, Aerospace, and Biomedical Engineering
University of Tennessee, Knoxville
Knoxville, Tennessee
kwong@utk.edu

4th Halloy, Julian
Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, Tennessee
julian.halloy@gmail.com

***Abstract*—Despite C and Python both being very popular programming languages, each tool possesses unique advantages and disadvantages. Notably, computers can run C code very quickly, but C syntax can be difficult for new programmers to understand. Python code, however, sacrifices speed for an easy-to-understand syntax. Thankfully, it is possible to combine the benefits of Python and 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++ (with speeds similar to C), MAGMA offers extremely fast computations. To combine MAGMA's speed with Python's easy-to-understand syntax, I researched how to use C++ code with Python. By studying a tool known as Simple Wrapper and Interface Generator (SWIG), I created PyMAGMA - a library of MAGMA functions which can be imported in Python for use.**

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

I. Background

A. 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 matrix-matrix multiplication and LU decomposition. MAGMA's main advantage over other linear algebra libraries, such as Linear Algebra PACKage (LAPACK), is that it contains functions which can run on not only central processing units (CPUs) but also 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 obtains a higher performance than LAPACK (Fig. 1).

National Science Foundation

TABLE I
SGEMM Time Performance

Size	*LAPACK (ms)*	MAGMA (ms)
1088	16.47	1.49
2112	104.55	8.05
3136	342.17	25.79
4160	771.77	60.31
5184	1490.97	113.10
6208	2456.86	198.38
7232	3835.59	298.79
8256	5743.81	414.99
9280	8164.28	565.60
10304	11077.56	775.99

Fig. 1. For ten sets of random square matrices, we calculated the time taken by LAPACK and MAGMA to perform Single-precision GEneral Matrix Multiplication (SGEMM). Because MAGMA's SGEMM function runs on the GPU whereas LAPACK code runs on the CPU, MAGMA performs SGEMM much faster than LAPACK.

B. 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 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 [1]. This unique feature makes SWIG suited for programmers who might interface C/C++ functions with multiple languages in the feature.

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 import 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 C++ functions, we describe the main steps and files involved when using SWIG on a Linux machine. Information about interfacing C functions or using SWIG with different target languages is detailed in the SWIG 4.0 Documentation [2].

A. Installation

To install SWIG on the Linux operating system, users can write the command `apt-get install swig` in the Linux command line and then press `Enter`. To check if the latest version of SWIG, 4.0, is in use, users should input the command `swig -version` into the Linux command line after SWIG is installed.

B. Header File (.h)

To use SWIG after installing it, the Python user must decide what C++ functions they want to use with Python. Once the functions have been chosen, a header file must be created. This file should contain the declarations 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.

By creating the header file, a SWIG user can organize C++ functions which they want to use into a single file. By maintaining one file of C++ functions, the user can easily add or remove a function to/from the Python interface after it is created by simply adding or removing its declarations (or definitions) in the header file and then regenerating the SWIG wrapper, import, and shared library files.

C. Interface File (.i)

After the Python user creates a header file of declarations for all the C++ functions they want to interface, the user must create a special SWIG file known as the interface file.

At minimum, the file should contain two `include` statements for the header file and the name of the Python interface which the user wants to create. Additionally, optional features known as typemaps can be added to the interface file to give SWIG specific directions on how to convert between specific C++ and Python types. Typemaps are further discussed in Sections 11, 12, and 13 of the SWIG 4.0 Documentation [2].

D. Import File (.py)

The Python file which we refer as the "import file" contains Python's `import` command, which will let Python users import the C++ functions into Python once the shared library file is created. Also, inside the import file is a Python function for each C++ function whose declaration was put into the header file. Each of these Python functions its call the corresponding C++ function inside the shared library, letting Python users can use a desired C++ function by simply calling its Python counterpart.

To create the import file, the user should use the SWIG command `swig -c++ -python NAME.i`, where `NAME` represents the name of the interface file.

E. Wrapper File (_wrap.cxx)

SWIG generates the wrapper file when it creates the import file. Inside the wrapper file is its namesake wrapper code which will translate the C++ functions, which were declared in the header file (.h), to the Python interpreter. The file also incorporates any specific type conversions which the user enforced with typemaps in the interface file (.i). Like the import file (.py), the wrapper file is generated with the Linux command `swig -c++ -python NAME.i`.

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. To compile the wrapper file, users can run the line `-fPIC -c NAME_wrap.cxx PATH_TO_PYTHON`. `NAME` and `PATH_TO_PYTHON` both represent the same meaning from the C version above. In this command, `NAME` represents the part of the wrapper file's name before `_wrap.cxx` and `PATH_TO_PYTHON` represents the path to the '`Python.h` in the user's Linux machine.

G. Shared Library (.so)

To create a shared library for import the original C++ fuunctions into Python, the SWIG user should ensure that they have the compiled wrapper file (*_wrap.o*) and object code for all the C++ functions declared in the header file (.h). Assuming the user already has a library containing object code for the chosen C++ functions, they can create the Python interface's shared library with the Linux command `ld -shared OBJECT_LIBRARY COMPILED_WRAPPER.o -o _MODULE.so`. It is vital that the Python user writes an underscore (_) before `MODULE.so`.

In this command, `OBJECT_LIBRARY` represents the existing library of object code to use with Python, `COMPILED_WRAPPER.o` represents the name of the compiled wrapper file (.o), `MODULE` is the name of the Python interface specified in the interface file (*.i*), and `_MODULE.so` represents the name for the generated shared library file. It is vital that the Python user writes an underscore (_) before `MODULE.so`.

Because SWIG has generated the compiled wrapper file (.o), import file (.py), and shared library (.so), the Python interface has been created.

III. Creating PyMAGMA

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

```
1  // Naming the PyMAGMA library
2  %module pymagma
3
4  %{
5      #include "pymagma.h"
6  %}
7
8  %include "pymagma.h"
```

Fig. 2. The code inside the *pymagma.i* interface file. Notably, we specify the name of the PyMAGMA library and use include statements for the *pymagma.h* header file.

```
7  from sys import version_info as _swig_python_version_info
8  if _swig_python_version_info < (2, 7, 0):
9      raise RuntimeError("Python 2.7 or later required")
10
11 # Import the low-level C/C++ module
12 if __package__ or "." in __name__:
13     from . import _pymagma
14 else:
15     import _pymagma
```

Fig. 3. On lines 13 and 15, the import statement in the *pymagma.py* import file for importing the PyMAGMA library into Python.

A. Header File (pymagma.h)

Our original header file contained declarations of MAGMA functions for managing memory and queues, sending data between CPUs and GPUs, multiplying matrices of doubles, and performing LU factorization.

B. Interface File (pymagma.i)

The interface file (*pymagma.i*) for the first version of PyMAGMA simply contained two include statements for the *pymagma.h* header file and the name of the Python library we wanted to create: PyMAGMA (Fig. 2).

C. Import File (pymagma.py)

After creating the *pymagma.h* header file *pymagma.i* interface file, we used SWIG to generate the *pymagma.py* import file. With this file, we could try importing the first version of PyMAGMA after we building it (Fig. 3). For each MAGMA function whose declaration was listed in the *pymagma.h* header file, the import file contained the corresponding Python function for calling the C++ function (Fig. 4).

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` (Fig. 5).

```
73 def magma_init():
74     return _pymagma.magma_init()
75
76 def magma_finalize():
77     return _pymagma.magma_finalize()
78
79 def magma_print_environment():
80     return _pymagma.magma_print_environment()
81
82 def magma_malloc(ptr_ptr, bytes):
83     return _pymagma.magma_malloc(ptr_ptr, bytes)
```

Fig. 4. Examples of Python functions in the *pymagma.py* import file which users will call to use the corresponding C++ functions from MAGMA in PyMAGMA

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

Fig. 5. A list of Linux commands in our Makefile used to create the *pymagma.py* import file and *pymagma_wrap.cxx* wrapper file (line 12), create the *pymagma_wrap.o* object file, and create the *_pymagma.so* shared library.

```
3286  if (!SWIG_Python_UnpackTuple(args, "magma_malloc", 2, 2, swig_obj)) SWIG_fail;
3287  res1 = SWIG_ConvertPtr(swig_obj[0], &argp1,SWIGTYPE_p_p_void, 0 |  0 );
3288  if (!SWIG_IsOK(res1)) {
3289    SWIG_exception_fail(SWIG_ArgError(res1), "in method '" "magma_malloc" "', argument " "1"" of type '" "magma_ptr *""");
3290  }
3291  arg1 = (magma_ptr *)(argp1);
3292  ecode2 = SWIG_AsVal_size_t(swig_obj[1], &val2);
3293  if (!SWIG_IsOK(ecode2)) {
3294    SWIG_exception_fail(SWIG_ArgError(ecode2), "in method '" "magma_malloc" "', argument " "2"" of type '" "size_t""");
3295  }
3296  arg2 = (size_t)(val2);
3297  result = (magma_int_t)magma_malloc(arg1,arg2);
3298  resultobj = SWIG_From_int((int)(result));
3299  return resultobj;
3300 fail:
3301  return NULL;
3302 }
```

Fig. 6. The low-level wrapper code for MAGMA's *magma_malloc()* function in the *pymagma_wrap.cxx* wrapper file

D. Wrapper File (pymagma_wrap.cxx)

As its name suggests, the *pymagma_wrap.cxx* wrapper file contained wrapper code for translating the MAGMA functions (declared in the *pymagma.h* header file) to the Python interpreter.

To create the *pymagma_wrap.cxx* wrapper file and *pymagma.py* import file, we entered the following SWIG command into the Linux terminal: `swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i` (Fig. 5). Initially, we did not include the `-DSWIG_NO_CPLUSPLUS_CAST` text in the Linux command; however, we obtained an error when trying to compile the Wrapper File without `-DSWIG_NO_CPLUSPLUS_CAST`. 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 prevent SWIG from using the C++ typecasts at all, we forced SWIG to use C typecasts by including `-DSWIG_NO_CPLUSPLUS_CAST` in the Linux command used to generate the *pymagma_wrap.cxx* wrapper file. By creating the *pymagma_wrap.cxx* Wrapper File with the Linux command `swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i`, we successfully compiled the *pymagma_wrap.cxx* wrapper file.

```
g++ -fPIC -c pymagma_wrap.cxx -I/home/user1/anaconda3/include/python3.9
pymagma_wrap.cxx: In function 'PyObject* _wrap_magma_getvector_internal(PyObject*, PyObject*)':
pymagma_wrap.cxx:2699:86: error: reinterpret_cast from type 'const void**' to type 'void**' casts away qualifiers
 2699 | #define SWIG_as_voidptrptr(a) ((void)SWIG_as_voidptr(*a),reinterpret_cast< void** >(a))
                                                                                      ^
pymagma_wrap.cxx:1081:91: note: in definition of macro 'SWIG_Python_ConvertPtr'
 1081 | #define SWIG_Python_ConvertPtr(obj, pptr, type, flags)  SWIG_Python_ConvertPtrAndOwn(obj, pptr, type, flags, 0)
                                                                                           ^~~~
pymagma_wrap.cxx:4068:10: note: in expansion of macro 'SWIG_ConvertPtr'
 4068 |   res3 = SWIG_ConvertPtr(swig_obj[2],SWIG_as_voidptrptr(&arg3), 0, 0);
          ^~~~~~~~~~~~~~
```

Fig. 7. A list of errors which occurred when trying to compile the *pymagma_wrap.cxx* wrapper file after it was creating a Linux command excluding *−DSWIG_NO_CPLUSPLUS_CAST*. SWIG used C++ typecasting in the wrapper file in an invalid way, resulting in the errors shown.

E. Compiled Wrapper File (.o)

To compile the *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`. In the command, the `-I/home/.../python3.9` path is the path to the *Python.h* on our Linux machine (Fig. 5).

F. Shared Library (_pymagma.so)

In the first version of PyMAGMA, the *_pymagma.so* shared library contained object code from the *pymagma_wrap.o* file and *libmagma.so* file - a shared library of object code from MAGMA.

To create the *_pymagma.so* shared library, we successfully ran the following Linux command: `ld -shared /home/user1/magma/lib/libmagma.so pymagma_wrap.o -o _pymagma.so` (Fig. 5).

G. Testing

After successfully importing PyMAMGMA with Python, we tried calling the following three functions to test the functionality of PyMAGMA:

- `magma_init()`
- `magma_print_environment()`
- `magma_finalize()`

Even though we could successfully call `magma_init()` and `magma_finalize()` in a Python environment on the Linux terminal (Fig. 8), `magma_print_environment()` was not initially functional. Specifically, when trying to locally call `magma_print_environment()` with the Linux terminal, correct output was displayed before the warning *** *stack smashing detected* *** appeared. Somehow, data in the computer's stack memory was getting incorrectly overwritten.

Since locally calling `magma_print_environment()`, 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. While we do not know exactly why Google Colab did not display any warnings, we think that it could be because Google Colab used 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, the newer 4.0.1 version of SWIG contained code for giving "stack smashing" warnings while the older 3.0.12 version did not. Nevertheless, we wanted to call `magma_print_environment()` locally without seeing stack smashing occur because the version of PyMAGMA created locally used the newer SWIG version.

After relinking all the object files in the *libmagma.so* library and creating PyMAGMA locally again, we could run `magma_print_environment()` without any errors or warnings (Fig. 8).

```
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pymagma as pmg
>>> pmg.magma_init()
0
>>> 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
% Mon Aug  1 16:15:53 2022
>>>
>>> pmg.magma_finalize()
0
```

Fig. 8. Successfully calling the MAGMA functions *magma_init()*, *magma_print_environment()*, and *magma_finalize()* with PyMAGMA in a Python environment after relinking the object files in the *libmagma.so* library

```
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pymagma
>>> pymagma.magma_init()
0
>>> memory_address = 0
>>> number_of_bytes = 8
>>>
>>> pymagma.magma_malloc(memory_address, number_of_bytes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/user1/pymagma/pymagma.py", line 83, in magma_malloc
    return _pymagma.magma_malloc(ptr_ptr, bytes)
TypeError: in method 'magma_malloc', argument 1 of type 'magma_ptr *'
```

Fig. 9. Attempting to call PyMAGMA's *magma_malloc()* function, which dynamically allocates a block of user-specified amount of GPU memory starting at a given address. Because the first argument is of the pointer type *magma_ptr* * but Python users cannot normally create pointers, calling the function produces a TypeError.

IV. Added Functions to PyMAGMA

Python users do not normally have the ability to create pointer types, but many MAGMA functions in PyMAGMA require pointer arguments. Therefore, Python users would somehow need a way to create the pointers (Fig. 9). To combat this issue, we added to the *pymagma.h* header file new versions of MAGMA functions which would require Python users to pass in pointer arguments. Other than requiring arguments of pointer types, many of these new versions operate nearly identically to their MAGMA counterparts.

Furthermore, to test the accuracy of the linear algebra functions DGESV (Double-precision GEneral matrix SolVe) and SGEMM (Single-precision GEneral Matrix Multiplication) which would eventually be added to PyMAGMA, we created PyMAGMA functions for creating, editing, and printing C++ arrays on both the CPU and GPU.

For simplicity, we will provide one function which we added to PyMAGMA to avoid the pointer arguments necessary to use many PyMAGMA functions. Also, we will provide three functions we created for managing arrays on the CPU. Overall, however, we added many functions to PyMAGMA to avoid using pointer arguments and to manage arrays of both floats and doubles on the CPU and GPU.

A. pymagma_malloc()

The *magma_malloc()* function is designed to dynamically allocate memory on the current GPU but requires a *ptr_ptr* argument of type *magma_ptr** (identical to the *void*** type in C++). Therefore, we created a *pymagma_malloc()* function with the same purpose as the MAGMA version but does not require the *ptr_ptr* argument (Fig. 10). Specifically, *pymagma_malloc()* declares a pointer of type *void**, passes

```
63  magma_int_t
64  magma_malloc( magma_ptr *ptr_ptr, size_t bytes );
65
66  void*
67  pymagma_malloc(size_t bytes) {
68      void* a;
69      magma_malloc(&a, bytes);
70      return a;
71  }
```

Fig. 10. The definition for the *pymagma_malloc()* function which we added to the *pymagma.h* header file to let users create dynamically allocate GPU memory.

```
307  float*
308  pymagma_sarray_cpu(magma_int_t height, magma_int_t width) {
309      void* void_array = pymagma_malloc_cpu(sizeof(float) * height * width);
310      float* sarray = (float*)void_array;
311      return sarray;
312  }
```

Fig. 11. The definition for the *pymagma_sarray_cpu()* function which we added to the *pymagma.h* header file to let users create arrays of C++ floats on the CPU.

the memory address of that pointer into the *magma_malloc()* function to allocate GPU memory, and then returns the pointer.

B. pymagma_sarray_cpu()

To test the results of performing SGEMM, we wanted a way to create arrays of C++ floats on CPUs. We achieved this by creating the *pymagma_sarray_cpu()* function (Fig. 11). After a Python user passes *height* and *width* arguments into the function, the functions calls *pymagma_malloc_cpu()* to dynamically allocate a $height\ x\ width$ block of CPU memory and then return the base address of that array as a *float** pointer.

It should be noted that the return type of *pymagma_sarray_cpu()* is *float** instead of *void**. If the return type was *void** instead, then inputting the returned pointer in the *pymagma_sset_cpu()* function would return an error would void pointers cannot be de-referenced in C++.

C. pymagma_sset_cpu()

To test the results of performing SGEMM, we wanted a way to change values in arrays of floats on CPUs. We achieved this by creating the *pymagma_sset_cpu()* function (Fig. 12). After an user passes A, row, col, lda, and $value$ arguments into the function, the functions sets the float $value$ in the position at row row and column $lda\ x\ col$ in the array A returned by *pymagma_sarray_cpu()*.

```
321  void
322  pymagma_sset_cpu(float* A, magma_int_t row, magma_int_t col, magma_int_t lda, float value) {
323      // Since Fortran (-> MAGMA) is col. major, we multiply lda with col
324      A[row + lda * col] = value;
325  }
```

Fig. 12. The definition for the *pymagma_sset_cpu()* function which we added to the *pymagma.h* header file to let users change values in arrays of C++ floats on the CPU.

```
336  void
337  pymagma_sprint_cpu( magma_int_t m, magma_int_t n, const float* A, magma_int_t lda ) {
338      magma_sprint( m, n, A, lda );
339  }
```

Fig. 13. The definition for the *pymagma_sprint_cpu()* function which we added to the *pymagma.h* header file to let users print arrays of C++ floats on the CPU.

D. pymagma_sprint_cpu()

To test the results of performing SGEMM, we wanted a way to print arrays of floats on CPUs. We achieved this by creating the *pymagma_sprint_cpu()* function (Fig. 13). After an user passes m, n, A, and lda arguments into the function, the functions calls a MAGMA function known as *magma_sprint()*, which we also added to PyMAGMA, to print the first m rows and n columns in the array A generated with *pymagma_sarray_cpu()*.

V. Using PyMAGMA

After adding all functions for managing arrays on the CPU and GPU and avoiding pointer errors, we performed Single-precision GEneral Matrix Multiplication (SGEMM) with PyMAGMA and compared its performance to MAGMA's. Overall, MAGMA performed SGEMM both faster (Fig. 14) and with more floating point operations per second (Fig. 15) than PyMAGMA. However, the differences in both time and floating point operations per second between MAGMA and PyMAGMA were not large, showing that we can use PyMAGMA can perform very fast computations despite being having to be imported into Python.

TABLE II
MAGMA and PyMAGMA SGEMM Comparison (Time)

Size	MAGMA (ms)	PyMAGMA (ms)
1088	1.49	1.49
2112	8.05	8.10
3136	25.79	26.01
4160	60.31	60.81
5184	113.10	113.6
6208	198.38	194.19
7232	298.79	285.85
8256	414.99	396.34
9280	565.60	573.74
10304	775.99	782.41

Fig. 14. Comparing MAGMA's and PyMAGMA's time performances of performing Single-precision GEneral Matrix Multiplication (SGEMM).

Fig. 15. Comparing MAGMA's and PyMAGMA's floating-point-operations-per-second performances of performing Single-precision GEneral Matrix Multiplication (SGEMM).

VI. Conclusion

As discussed in Section 4.2, we can successfully perform DGESV and SGEMM with PyMAGMA, showing that PyMAGMA is functional. Furthermore, PyMAGMA's performance of SGEMM is very similar to MAGMA's performance

TABLE III
MAGMA AND PYMAGMA SGEMM COMPARISON (GIGAFLOP RATE)

Size	MAGMA (gflop/s)	PyMAGMA (gflop/s)
1088	1729.54	1729.54
2112	2341.09	2326.64
3136	2392.08	2371.85
4160	2387.66	2368.03
5184	2463.79	2452.95
6208	2412.25	2464.30
7232	2532.03	2646.65
8256	2712.24	2839.86
9280	2826.10	2786.01
10304	2819.76	2796.62

of SGEMM, showing that our PyMAGMA Python interface has sufficient performance to where Python users should use it. Also, we have an easy way to add functions to or remove functions from PyMAGMA by simply changing the list of function declarations and definitions included in our *pymagma.h* Header File.

VII. WAYS FOR IMPROVEMENT

A. Use Typemaps

As detailed in section 4, to use MAGMA functions which required pointer arguments, we created "pointerless" versions of the functions. Ideally, however, we would not create any new functions at all. Rather, we would like to tell SWIG how to wrap the input and output types for our MAGMA functions. A SWIG tool for doing this is known as "typemaps." For more information on what typemaps are and how to use them, please refer to Sections 11, 12, and 13 in the SWIG 4.0 Documentation [2].

B. Follow Python Enhancement Proposal 8

Because PyMAGMA is a Python library, we should make sure that the Python functions in the *pymagma.py* Import File which users call from the library follow the PEP 8 style guidelines [3].

Notably, we should change the case of every Python function in the Import File to snake case). For example, the *pymagma_getdevice()* function should be named as *pymagma_get_device()* instead. One possible way of accomplishing this is to directory change the names of the Python functions in the Import File. However, these changes would disappear each time PyMAGMA is regenerated with a new Header File or Interface File. Alternatively, we could create new C++ functions with snake case in the *pymagma.h* Header File which call their non-snake case equivalents. Such changes would not disappear everytime PyMAGMA is regenerated.

Furthermore, we could shorten the names of Pyton functions in the Import File. For example, the *pymagmablas_sgemm()* function should be simply named *sgemm()*. One possible way of accomplishing this is to directly shorten the names of the Python functions in the Import File. However, these changes would disappear each time PyMAGMA is regenerated with a new Header File or Interface File. Alternatively, we could create new C++ functions with shortened names in the *pymagma.h* Header File which call their non-snake case equivalents. Such changes would not disappear every time PyMAGMA is regenerated.

```
void*
malloc(size_t bytes, const char* processor) {
    if (processor == "cpu") {
        return pymagma_malloc_cpu(bytes);
    }

    if (processor == "gpu") {
        return pymagma_malloc(bytes);
    }
}
```

Fig. 16. Combining the *magma_malloc()* and *magma_malloc_cpu()* functions into a single *malloc()* function

C. Combine CPU and GPU Versions of Functions

Reducing the number of PyMAGMA functions available to users could make using PyMAGMA less overwhelming for Python users. Therefore, rather than having separate CPU and GPU functions for performing a given task (e.g., allocating memory), we could create a single function which a "processor" argument 16. The default value of the argument would be "CPU" and the CPU version of the function would be performed whenever the function is called. However, if the user inputs the value "GPU" instead, the GPU version of the function would be performed whenever the function is called. To do this, we could add the new function directly to the *pymagma.h* Header File.

D. Use PyMAGMA with Foreign Data Types

To increase the usability of PyMAGMA, we should find a way to use foreign data types with PyMAGMA. Specifically, we should learn how to input NumPy arrays and Python lists into functions which expect pointer arguments for an array. One way in which NumPy arrays can be used with SWIG interfaces is with typemaps inside the *numpy.i* interface file. However, using *numpy.i* depends heavily on the number and order of arguments in functions. Therefore, *numpy.i* should only be used if PyMAGMA is finalized and will not receive any other changes.

ACKNOWLEDGMENT

This research project was sponsored by the National Science Foundation (NSF) through a Research Experience for Undergraduates (REU) grant for the Research Experiences in Computational Science, Engineering, and Mathematics (RECSEM) program held at the University of Tennessee, Knoxville (UTK). During the project, research assistance was received by researchers from the Innovative Computing Laboratory (ICL) and University of Tennessee, Knoxville.

REFERENCES

[1] Welcome to SWIG. (2019, April 18). Retrieved from https://swig.org/.
[2] SWIG-4.0 Documentation [PDF file]. Retrieved from https://swig.org/Doc4.0/SWIGDocumentation.pdf.
[3] PEP 8 — the Style Guide for Python Code. Retrieved from https://pep8.org/