PyMAGMA: A Python Library for MAGMA

Delario Nance, Jr. (Davidson College)
Mentors: Stanimire Tomov (UTK) and Kwai Wong (UTK)
Research Assistant: Julian Halloy (UTK)
Presentation Outline

1) Background
2) SWIG Workflow
3) Creating PyMAGMA
4) Extending PyMAGMA
5) Performing SGEMM with PyMAGMA
6) Conclusion and Future Work
Background
What is MAGMA?

➢ Stands for “Matrix Algebra on GPU and Multicore Architectures”
➢ A large package of C++ functions optimized for running linear algebra operations on GPUs
   ○ LAPACK and NumPy are linear algebra packages whose code only runs on CPUs

Comparing the times taken by LAPACK, MAGMA, and NumPy to perform SGEMM (C = -AB + C)

GPU Model: NVIDIA GeForce GTX 1650 SUPER
CPU Model: Intel(R) Xeon(R) CPU X5650
C++ vs. Python

**C++**
- Code is ran very quickly
- Syntax can be difficult for new programmers to understand

**Python**
- Code is ran relatively slowly
- Syntax is often easy to understand

Printing “Hello REU” in C++
```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello REU \n";
    return 0;
}
```

Printing “Hello REU” in Python
```python
print("Hello REU")
```
What is SWIG?

➢ Stands for “Simplified Wrapper and Interface Generator”
➢ A tool for interfacing C/C++ code with high-level programming languages (e.g., Python)
➢ Works by generating three files
  ○ Wrapper file - translates C/C++ functions to the target language
  ○ Shared library - contains the original C/C++ functions and wrapper code
  ○ Import file - lets users import the shared library into the target language
SWIG Workflow
First, the user must choose which C functions to interface with Python.

Each of the C functions should be declared in a file known as the “header file.”

By editing the header file, the user can easily extend the Python interface.

**Sample C functions to interface**

```c
4   // Returns the value n!
5   math_int my_fact(int n) {
6       if (n == 0) {
7           return 1;
8       }
9       return n * my_fact(n - 1);
10     }
11
12   // Returns the value x mod y
13   math_int my_mod(int x, int y) {
14       return MOD_MACRO(x, y);
15     }
```

**Header file for the chosen C functions**

```c
4   // User-defined macro
5   #define MOD_MACRO(x, y) (x % y)
6
7   // User-defined typedef
8   typedef int math_int;
9
10  // C declarations
11   math_int my_fact(int n);
12   math_int my_mod(int x, int y);
13   math_int my_range(int n);
```
File 2: Interface File (.i)

➢ Must contain the name of the Python library to create (Line 1)
➢ Usually contains two “include” statements for the previously created header file (Lines 4 and 6)
➢ Where users can insert typemaps to give SWIG directions on how to handle specific C-to-Python type conversions

Example of a SWIG Typemap

```c
%module example
#include "typemaps.i"

%apply double *OUTPUT { double *result };
%inline {%
extern void add(double a, double b, double *result);
%
```
File 3a: Import File (.py)

➢ The “payment” in our real-life analogy
➢ Lets users import the library of C code into Python after it is created (Line 15)
➢ Contains a Python function for each C function which was declared in the header file (Lines 65-72)

The Python import command for importing the Python library

```python
11 # Import the low-level C/C++ module
12 if __package__ or "." in __name__:
13     from . import _pymath
14 else:
15     import _pymath
```

The Python functions which users will call to use the C functions

```python
65     def my_fact(n):
66         return _pymath.my_fact(n)
67
68     def my_mod(x, y):
69         return _pymath.my_mod(x, y)
70
71     def my_range(n):
72         return _pymath.my_range(n)
```
File 3b: Wrapper File (_wrap.c)

➢ The “translator” in our real-life analogy
➢ Contains the “wrapper” code which will translate our chosen C functions to the Python interpreter
➢ Incorporates any typemaps which the user enforced in the interface file (.i)

Wrapper code for the *my_fact()* function

```c
if (largs) SWIG_fail;
swig_obj[0] = args;
ecode1 = SWIG_AsVal_int(swig_obj[0], &val1);
if (!SWIG_IsOK(ecode1)) {
    SWIG_exception_fail(SWIG_ArgError(ecode1), "in method ", "my_fact", ", argument ", "1" of type ", "int":"");
}
arg1 = (int)(val1);
result = (math_int)my_fact(arg1);
resultobj = SWIG_From_int((int)(result));
return resultobj;
fail:
    return NULL;
```
File 4: Shared Library (.so)

➢ The library of C functions which users will import into Python
➢ Contains the compiled wrapper code and object code for the C functions

Using the PyMath library in Python

```python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # Importing the PyMath Library
>>> import pymath

>>> # Calling the interfaced C functions
>>> pymath.my_fact(5)
120
>>> pymath.my_mod(4, 2)
0
>>> pymath.my_range(3)
3
```
Creating PyMAGMA
Header File (*pymagma.h*)

- **Contained typedefs and declarations for the MAGMA functions which we wanted to use in Python**
- **Previous Errors**
  - ‘Magma_trans_t was not declared in this scope’

Example Typedefs for MAGMA Functions

```c
// MAGMA types (from file magma_types.h)

typedef int magma_int_t;
typedef magma_int_t magma_device_t;

struct magma_queue;
typedef struct magma_queue* magma_queue_t;

typedef void *magma_ptr;
typedef void const *magma_const_ptr;

typedef double *magmaDouble_ptr;
typedef double const *magmaDouble_const_ptr;
```

Example C++ declarations from MAGMA

```c
magma_int_t magma_init( void );
magma_int_t magma_finalize( void );

void magma_print_environment();
magma_int_t magma_malloc( magma_ptr *ptr_ptr, size_t bytes );
magma_int_t magma_malloc_cpu( void **ptr_ptr, size_t bytes );
magma_int_t magma_malloc_pinned( void **ptr_ptr, size_t bytes );
magma_int_t magma_free_cpu( void *ptr );
```
Interface File (*pymagma.i*)

- Where we specified the name of the library we were creating (PyMAGMA)
- Contains two include statements for the *pymagma.h* header file

```plaintext
1 // Naming the PyMAGMA library
2 %module pymagma
3
4 // Including the Header File
5 %{
6   #include "pymagma.h"
7 %{
8
9 %include "pymagma.h"
```
Import File (*pymagma.py*)

➢ Contained the Python statement for importing the PyMAGMA library into Python once it was built (Line 15)

➢ Included Python functions for calling the C++ code from MAGMA (Lines 73-80)

➢ Created with the command `swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i`

---

The Python import statements

```python
# Import the low-level C/C++ module
if __package__ or "." in __name__:
    from . import _pymagma
else:
    import _pymagma
```

---

Python functions which call MAGMA functions in the PyMAGMA library

```python
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()
```
Wrapper File (*pymagma_wrap.cxx*)

- Contained the wrapper code for translating the MAGMA functions to the Python interpreter
- Created with the command: `swig -DSWIG_NO_CPLUSPLUS_CAST -c++ -python pymagma.i`
- Previous errors
  - `reinterpret_cast` from type `const void**`...

```c
#define SWIG_as_voidptr(a) const_cast< void * >(static_cast< const void * >(a))
#define SWIG_as_voidptrptr(a) ((void)SWIG_as_voidptr(*a), reinterpret_cast< void** >(a))
```

Compilation Error

```
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))
```
Shared Library (_pymagma.so)

- Created with the command:
  ```bash
cpython3 /home/user1/magma/lib/libmagma.so
```

Using three MAGMA functions in Python with PyMAGMA

```python
(base) user1@lapenna3-HP-Z800-Workstation:~/pymagma$ python
Python 3.9.12 (main, Apr  5 2022, 06:56:58)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # Importing PyMAGMA
>>> import pymagma as pnm
>>> # Initializing the MAGMA library
>>> pnm.magma_lnit()
0
>>> # Printing MAGMA info.
>>> pnm.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 11640. OpenMP threads 24.
% device 0: NVIDIA GeForce GTX 1650 SUPER, 1740.0 MHz clock, 3910.6 MiB memory, capability 7.5
% Tue Aug  2 10:36:42 2022
>>> # Finalizing the MAGMA library
>>> pnm.magma_finalize()
0
```
Extending PyMAGMA
Many C++ functions in MAGMA require pointer types as arguments, but Python users cannot normally create pointers in Python!

How do we resolve this pointer error???

Trying to call the `magma_malloc()`, which expects a pointer argument, through PyMAGMA

```python
>>> 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 *'
```
We create new “pointerless” functions in PyMAGMA which call their “pointer” counterparts!
pymagma_malloc_cpu()

➢ Purpose
  ○ Dynamically allocates a user-specified number of bytes for a block of CPU memory

➢ Returns
  ○ The base address of the allocated block of CPU memory

Additional Added Functions:
  • pymagma_malloc()
  • pymagma_free()
  • pymagma_malloc_pinned()
  • pymagma_free_pinned()
  • pymagma_queue_create()
  • pymagma_queue_destroy()
  • pymagma_queue_sync()

The definition for pymagma_malloc_cpu()

```c
magma_int_t
magma_malloc_cpu( void **ptr_ptr, size_t bytes );

void*
pymagma_malloc_cpu(size_t bytes) {
  void* a;
  magma_malloc_cpu(&a, bytes);
  return a;
}
```
pymagma_sarray_cpu()

Purpose:

- Creates a matrix of floats by dynamically allocating a \textit{height} \times \textit{width} block of memory for floats on the CPU

Returns:

- The base address of the allocated block of memory

The definition for \textit{pymagma_sarray_cpu()}

```c
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     }
```
**pymagma_sset_cpu()**

**Purpose:**
- Changes the value at a given position in a matrix of floats on the CPU

**Returns:**
- N/A

The definition for `pymagma_sset_cpu()`

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

**Purpose:**
- Prints an array of floats stored on the CPU

**Returns:**
- N/A

```c
void pymagma_sprint_cpu( magma_int_t m, magma_int_t n, const float* A, magma_int_t lda ) {
    magma_sprintf( m, n, A, lda );
}
```
Performing SGEMM with PyMAGMA
SGEMM Performance (C = -AB + C)

Takeaways:

➢ PyMAGMA performs SGEMM with a similar time and speed to MAGMA
➢ Like MAGMA, PyMAGMA performs faster than LAPACK and NumPy
Conclusion and Future Work

Conclusion

❖ We successfully used SWIG to build PyMAGMA, an interface through which currently ~34 functions in MAGMA can be used with Python
❖ We learned that PyMAGMA can perform SGEMM with similar speeds to MAGMA
❖ We learned that we can easily add functions to PyMAGMA by adding their declaration/definition to pymagma.h

Future Work

❖ Research how SWIG typemaps can be used to direct SWIG in how to wrap pointer arguments
❖ Research how to use SWIG with foreign data types (e.g., NumPy arrays)
Acknowledgments and References

Acknowledgments

- National Science Foundation (NSF)
- Innovative Computing Laboratory (ICL)
- National Institute of Computational Sciences (NICS)

References