Heterogeneous Computing in Julia MAGMA binding

Student: Yushao Chen*
Mentor: Roger Luo, Viral B. Shah

Synopsis

MAGMA (Matrix Algebra on GPU and Multicore Architectures) is a collection of next-generation linear algebra libraries for heterogeneous architectures. By completing the Julia interface of MAGMA, we can provide powerful heterogeneous computing toolkits to users who require a large amount of linear algebra computation and face great computational complexity along with huge memory costs, such as condensed matter physicists who focus on tensor networks and machine learning.

1 Introduction

1.1 Background of MAGMA

Julia is becoming more and more popular and excellent for big data analytics as well as high-performance computation. Also, it is suitable for CPU and GPU computing especially with Julia packages for CUDA (e.g., Project JuliaGPU).

Meanwhile, recent activities of major chip manufacturers, such as Intel, AMD, IBM, and NVIDIA, make it more evident than ever that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

- many-cores CPU technology, where the number of cores will continue to escalate because of the desire to pack more and more components on a chip while avoiding the power wall, instruction level parallelism wall, and the memory wall; and
- special purpose hardware and accelerators, especially Graphics Processing Units (GPUs), which are in commodity production, have outpaced standard CPUs in floating point performance in recent years, and have become as easy, if not easier to program than multicore CPUs.

*CONTACT Email: chenys13@outlook.com; GitHub: JerryChen97; Tele No. +86 13051670552

While the relative balance between these component types in future designs is not clear, and will likely to vary over time, there seems to be no doubt that future generations of computer systems, ranging from laptops to supercomputers, will consist of a composition of heterogeneous components.[1]

Among the various libraries used for CPU/GPU computing, MAGMA (Matrix Algebra on GPU and Multicore Architectures) is a collection of next-generation linear algebra libraries for heterogeneous architectures. MAGMA is designed and implemented by the team that developed LAPACK and ScaLAPACK, incorporating the latest developments in hybrid synchronization- and communication-avoiding algorithms, as well as dynamic runtime systems. Interfaces for the current LAPACK and BLAS standards are supported to allow computational scientists to seamlessly port any linear algebra reliant software components to heterogeneous architectures. MAGMA allows applications to fully exploit the power of current heterogeneous systems of multi/many-core CPUs and multi-GPUs to deliver the fastest possible time to an accurate solution within given energy constraints.[2]MAGMA can help scientists and engineers with different work in machine learning and other popular computing techniques.

For the ease of Julia users' CPU/GPU linear algebra computing, one natural thought is to implement the Julia binding for MAGMA. By binding MAGMA to Julia and providing corresponding BinaryProvider.jl for installation, both of which should be fitted in with existing Julia GPU project JuliaGPU, we will definitely benefit many MAGMA and Julia users sustainably.

1.2 Benefits to Community

Since MAGMA is meant for the general matrix algebra computation, e.g. QR factorization and singular value decomposition (SVD), all the high-performance computation users in Julia community will definitely benefit from the convenience of complete MAGMA binding. For many scientists, the extremely powerful multiple dispatch of Julia provides better and easier expression as well as the mathematics-friendly syntax for scientific computing. A straightforward API of MAGMA will not only attract more researchers but also help scientists and engineers more naturally develop the computing programs which should have either demanded too complicated and counter-intuitive designing or consumed too much time and memory.

Recently, many physicists have been trying to combine the automatic differentiation scheme with tensor networks to solve some classical but difficult many-body system problems. Lying in the core technical challenges are numerically stable differentiation through singular value decomposition and memory-efficient implementation for fixed point iterations, which can be apropriately solved by MAGMA.[3] Hence, our Julia MAGMA binding will combine the existing tensor toolkits and high-performance heterogeneous computing, and make significant contribution to computational condensed-matter physics.

[1]https://icl.utk.edu/magma/overview/index.html

[2]http://www.icl.utk.edu/files/print/2017/magma-sc17.pdf

[3]Differentiable Programming Tensor Networks, https://arxiv.org/pdf/1903.09650.pdf

2 Implementation

2.1 The Structure Overview of MAGMA

The similar interface of MAGMA to LAPACK facilitates porting existing codes, and many routines share the same base names and arguments. The following are five classes of MAGMA routines:

- **Computational routines** execute jobs like matrix factorization and reduction, similar to subroutines of driver routines.
- **Driver routines** solve an entire problem like linear system solver, eigenvalue solver, and singular value decomposition.
- **BLAS routines** provide vector operations (level-1), matrix-vector operations (level-2), and matrix-matrix operations (level-3). These routines consist of wrappers around CUBLAS indicated by prefix *magma_* and MAGMA implementation indicated by *magmablas_*.
- **Auxiliary routines** contain additional BLAS-like routines, many of which originally defined in LAPACK. All MAGMA auxiliary routines are GPU native and take the matrix in GPU memory.
- **Utility routines** in charge of initialization/finalization, memory allocation, and communication between CPU and GPU, along with error handling.

Besides, the conventions for data types, variables, and functions are slightly different from the typical.

MAMGA uses *magma_int_t* for C/C++ int type and *magmaFloatComplex/magmaDoubleComplex* to indicate complex numbers. For macros, constants, variables, and functions, it basically keeps the conversions but adds prefix *magma_* or *MAGMA_*.

2.2 Related Work and Existing Project

GitHub JuliaGPU/MAGMA

Pinned repositories CuArrays.jl, CUDAnative.jl, CLArrays.jl, OpenCL.jl and CUDAdrv.jl

MAGMA.jl repository forked from mweastwood/MAGMA.jl 5 years ago and some source codes fixed 9 months ago; have already finished parts of enums constants and some types defined by MAGMA, e.g., MagmaVector and MagmaMatrix.

Finished wrappers

- MagmaVector(x::Vector)
- Get(x::MagmaVector)
- MagmaMatrix(x::Matrix)
- Axpy!
- (Level 1 BLAS) Dot
- (Level 2 BLAS) Gemv!

In a word, some general constants and parts of essential linear algebraic types and functions have been finished, though there are still lots of work to be done.

2.3 Implementation Plan

2.3.1 Resources

Object

- MAGMA v2.5.0

Tools

- Julia v1.1.0
- JuliaGPUthe latest version
- Clangas the wrapper auto-generator
- BinaryProvier.jlto provide downloads and installation
- CPU/GPU for testing. Maybe my personal laptop.

2.3.2 Basic Techniques

By following the style of CUBLAS binding existing in JuliaGPU, for every routine in MAGMA, we will

1. create low-level wrappers by simply using **ccall**
2. create high-level wrappers by manually handling the linear algebraic input/output issues, considering what a mathematician/physicist will need for completing an SVD or QR.
3. realize the multiple dispatch for common routines, taking into considerartion people's general demands to these tasks.

2.3.3 Steps and Schedule

- **25th March ~26th May: Pre-GSoC Investigation and Training** Get myself familiar with Julia, CUDA, and MAGMA; community bonding.
- **1st Week, 27th May ~2nd June: Prepare Utilities** Prepare the low-level wrappers for the essential utility routines, such as init(), finalize(), queue_create(...), dcsrset(...), dvset(...) and so on. (We omitted the prefix *magma_* here for simplification and same to the following)
- **2nd Week, 3rd June ~9th June: Wrapping Trial** Prepare the low-level wrappers for the computational routines. For examples, geqrf or some other geqr functions, or some level-1 BLAS routines.
- **3rd Week, 10th June ~16th June: Trial Testing** Construct the first runnable demo and test the performance to spot in advance some of the potential problems and obstacles that we may confront later.
- **4th Week, 17th June ~23rd June: First-Round Wrapping** After successfully wrapping part of the most low-level routines without loss of high performance, continue the further wrapping and finish the work for all the computational routines and level-1 BLAS routines, especially the QR routines.
- **5th Week, 24th June ~30th June: First-Round Testing** Test the results of 1st Wrapping Round, and then deal with possible problems.
- **6/7th Week, 1st July ~14th July: Second-Round Wrapping** Wrap all the driver routines, left BLAS routines and auxiliary routines, especially the symmetric eigensolver and all kinds of SVD routines.
- **8th Week, 15th July ~21st July: Second-Round Testing** Test the result of 2nd Wrapping Round, and then fixed possible bugs.
- **9th Week, 22nd July ~28th July: Overall White-box Testing** Design the unit testing for every routine, considering especially the data type, input/output and the interaction with CPU/GPU; catch any defects that may cause damage to the performance or memory safety.
- **10th Week, 29th July ~4th August: Overall Black-box Testing** Apply the finished binding to some practical physics research techniques such as tensor renormalization group approaches to the square lattice and watch the performance of our MAGMA.jl package.
- **11th Week, 5th August ~11th August: Polishing and BinaryProvider** Re-check the whole project source codes and improve existing imperfections. Provide ways to download and install the MAGMA.jl package using BinaryProvier.jl.
- **12th Week, 12th August ~18th August: Documentation and Publishment** Make the formal documents for both users and future contributors. Publish the first official version and welcome testing, usage, and issues.
- **Future Maintenance**

2.4 Sample Code

All my sample codes have been added to my personal GitHub repository JerryChen97/MAGMA.jl. The *common.jl* and *samples.jl* will be presented in the appendix.
However, my sample only provides some prototypes for routines gesvd, which utilize QR factorization to implement matrix SVD. Because of the time constraint, I did not finish all the utility routines' wrappers which are essential and necessary for a successful running of MAGMA library. Hence, it is actually not a real testing but something used for presenting the codes' styles and wrapper method.

3 Curriculum Vitae

3.1 Contact

Name: Yushao Chen
Email: chenyushao5@gmail.com; chenys13@outlook.com;
Telephone No.: +86 13051670552
Postal Address: Room 527, Student Dorm #28, Tsinghua University, Haidian District, Beijing, China
GitHub: JerryChen97
Slack: Jerry Chen

3.2 Education Background

I am a senior undergraduate student majoring in physics at Tsinghua University, Beijing, China. Two years ago, I started minoring in Computer Technology and Application at the Department of Computer Science and Technology, and then I began my programming with C/C++.
I got good grades in my CS courses, and most of them included many projects, which offered me much training in programming practice; meanwhile, as a quasi-physicist, I equipped myself with sufficient toolkits commonly used by physics researchers.
Over these years, I have finished some projects both in and out of class, ranging from GUI programming and web development to numerical computation and deep learning. Most of them are not large-scale and usually finished locally, but still teach me so much about how a project should be developed and what I should do when faced with brand new areas.
Now I am capable of several different programming languages, including Fortran, C/C++, Java, Python. Also, I am familiar with common computational software like MATLAB, Mathematica.
I got to know Julia a couple of months ago, introduced by one of my friends to its amazing capacity to provide high productivity and performance at the same time. After a while of light usage, I indeed felt pleasure and convenience but also found that unlike in Python, there were still limited packages in Julia and more in demand for development.
Quite by accident, I heard about GSoC and spotted many exciting projects. Among all of them, I believe the "MAGMA binding" fits me most because I major in computational condensed-matter physics and understand how important it is to be able to get programs with both high-performance and legibility. Especially for the matrix computation which

will be the most challenging technical problem for tensor renormalization group, it is never too late to tackle it with more modern programming language like Julia. If we finally create a good enough MAGMA.jl, it will benefit lots of physicists in the future.

3.3 Development Experiences

3.3.1 Softwares Development

- A Six-in-a-row Game with Deep Reinforcement Learning AI
- ECoinFor: A Financial Info Software powered by AI

3.3.2 Numerical Computation

- Computing Mean Field Properties of Fermi Gases in BCS-BEC Crossover

3.4 Logistics

I will be totally free from 27th May to 18th August in 2019 and able to spend over 40 hours per week on our GSoC project.

Appendix

common.jl

```
# sample code for magma binding



# MAGMA enum constants
# the whole file will be stored as enums.jl
#just like in JuliaGPU/MAGMA.jl

# MAGMA constants indicating the vectors status
# as input/output for some functions
# For example, the gesvd functions will use
# MagmaNoVec, MagmaSomeVec, MagmaAllVec and
# MagmaOverwriteVec to indicate the
# strategies that will be applied to the SVD
# U matrix and VT matrix in A = U Σ V**T
# (for MagmaOverwriteVec it is going to overwrite A)
const MagmaNoVec         = 301
const MagmaVec           = 302
const MagmaIVec          = 303
const MagmaAllVec        = 304
const MagmaSomeVec       = 305
const MagmaOverwriteVec  = 306
const MagmaBacktransVec  = 307

### some certain setting for this testing
using CUDAdrv
using CUDAapi
using CUDAnative
using CuArrays
# indicate the library position
const libmagma = "/usr/local/magma/lib/libmagma.so"
```

samples.jl

```
# low-level wrappers of the MAGMA library
include("common.jl")


# magma_init
function magmaInit()
    ccall((:magma_init, libmagma),Cint,())
end

# magma_finalize
function magmaFinalize()
    ccall((:magma_finalize, libmagma),Cint,())
end

# magma_cgesvd wrapper
function magmaCgesvd(jobu, jobvt, m, n, A, lda, s, U,
    ldu, VT, ldvt, work, lwork, rwork, info)
    ccall((:magma_cgesvd, libmagma),
               Cint,
               (Cint, Cint, Cint, Cint, PtrOrCuPtr{ComplexF32},
                Cint, PtrOrCuPtr{Float32}, PtrOrCuPtr{ComplexF32},
                Cint, PtrOrCuPtr{ComplexF32}, Cint,
                PtrOrCuPtr{ComplexF32}, Cint, PtrOrCuPtr{Float32},
                PtrOrCuPtr{Cint}),
                jobu, jobvt, m, n, A, lda, s, U, ldu, VT, ldvt,
                work, lwork, rwork, info)
end

# magma_dgesvd wrapper
function magmaDgesvd(jobu, jobvt, m, n, A, lda, s, U,
     ldu, VT, ldvt, work, lwork, info)
    ccall((:magma_dgesvd, libmagma),
               Cint,
               (Cint, Cint, Cint, Cint, PtrOrCuPtr{Float64},
                Cint, PtrOrCuPtr{Float64}, PtrOrCuPtr{Float64},
                Cint, PtrOrCuPtr{Float64}, Cint,
                PtrOrCuPtr{Float64}, Cint, PtrOrCuPtr{Cint}),
                jobu, jobvt, m, n, A, lda, s, U, ldu, VT, ldvt,
                work, lwork, info)
end

# magma_sgesvd wrapper
function magmaSgesvd(jobu, jobvt, m, n, A, lda, s, U,
     ldu, VT, ldvt, work, lwork, info)
    ccall((:magma_sgesvd, libmagma),
               Cint,
               (Cint, Cint, Cint, Cint, PtrOrCuPtr{Float32},
                Cint, PtrOrCuPtr{Float32}, PtrOrCuPtr{Float32},
                Cint, PtrOrCuPtr{Float32}, Cint, PtrOrCuPtr{Float32},
                Cint, Ptr{Cint}),
                jobu, jobvt, m, n, A, lda, s, U, ldu, VT, ldvt,
                work, lwork, info)
end

# magma_cgesvd wrapper
function magmaZgesvd(jobu, jobvt, m, n, A, lda, s, U,
     ldu, VT, ldvt, work, lwork, rwork, info)
    ccall((:magma_zgesvd, libmagma),
               Cint,
               (Cint, Cint, Cint, Cint, PtrOrCuPtr{ComplexF64},
                Cint, PtrOrCuPtr{Float64}, PtrOrCuPtr{ComplexF64},
                Cint, PtrOrCuPtr{ComplexF64}, Cint, PtrOrCuPtr{ComplexF64},
                Cint, PtrOrCuPtr{Float64}, PtrOrCuPtr{Cint}),
                jobu, jobvt, m, n, A, lda, s, U, ldu, VT, ldvt,
                work, lwork, rwork, info)
end
```

```
# wrappers of the low-level MAGMA functions
for (function_name, element_type, singular_value_type) in
    ((:magmaCgesvd,:ComplexF32,:Float32),
                    (:magmaZgesvd,:ComplexF32,:Float64))
    @eval begin
        # magma_int_t magma_cgesvd  (   magma_vec_t     jobu,
        #   magma_vec_t     jobvt,
        #   magma_int_t     m,
        #   magma_int_t     n,
        #   magmaFloatComplex *     A,
        #   magma_int_t     lda,
        #   float *     s,
        #   magmaFloatComplex *     U,
        #   magma_int_t     ldu,
        #   magmaFloatComplex *     VT,
        #   magma_int_t     ldvt,
        #   magmaFloatComplex *     work,
        #   magma_int_t     lwork,
        #   float *     rwork,
        #   magma_int_t *   info
        # )
        function gesvd!(jobu::Cint,
                    jobvt::Cint,
                    A::CuMatrix{$element_type},
                    ldu::Cint,
                    ldvt::Cint,
                    lwork::Cint,
                    rwork::PtrOrCuPtr{$singular_value_type})
            convert(Cint,jobu)
            convert(Cint,jobvt)
           m, n = size(A)
           # there should be some certain checking
           # for the input matrix and arrays
           # before we can calculate further input
           # and call the lower C function wrappers
           lda = max(1,stride(A,2))
           s = CuArray{$singular_value_type}(0,min(m,n))

           # there should be some conditions for the size of U and VT
           # but we will deal with them later
           U = zeros($element_type,ldu,m)
           VT = zeros($element_type,ldvt,n)

           work = zeros($element_type,max(1,lwork))
           info = zeros(Cint,1)

           $function_name(jobu, jobvt, m, n, A, lda, s, U, ldu, VT,
            ldvt, work, lwork, rwork, info)
           return U, s, VT, work, info
       end
    end
end
for (function_name, element_type, singular_value_type) in
     ((:magmaDgesvd,:Float64,:Float64),
                    (:magmaSgesvd,:Float32,:Float32))
    @eval begin
        # magma_int_t magma_dgesvd  (   magma_vec_t     jobu,
        #   magma_vec_t     jobvt,
        #   magma_int_t     m,
        #   magma_int_t     n,
        #   magmaFloatComplex *     A,
        #   magma_int_t     lda,
        #   float *     s,
        #   magmaFloatComplex *     U,
        #   magma_int_t     ldu,
        #   magmaFloatComplex *     VT,
        #   magma_int_t     ldvt,
        #   magmaFloatComplex *     work,
        #   magma_int_t     lwork,
```

```
        #   magma_int_t *   info
        # )
        Base.unsafe_convert(::CuPtr{$element_type},x::CuArray{$element_type})=Base.unsafe_convert(CuPt
        Base.unsafe_convert(::PtrOrCuPtr{$element_type},x::CuArray{$element_type})=Base.unsafe_convert
        function gesvd!(jobu,
                        jobvt,
                        A::CuMatrix{$element_type},
                        ldu,
                        ldvt,
                        lwork)
            convert(Cint,jobu)
            convert(Cint,jobvt)
           m, n = size(A)
           # there should be some certain checking
           # for the input matrix and arrays
           # before we can calculate further input
           # and call the lower C function wrappers
           lda = max(1,stride(A,2))
           s = cu(zeros(m,n))

           # there should be some conditions for the size of U and VT
           # but we will deal with them later
           U = cu(zeros($element_type,ldu,m))
           VT = cu(zeros($element_type,ldvt,n))

           work = cu(zeros($element_type,max(1,lwork)))
           info = (zeros(Cint,1))

           $function_name(jobu, jobvt, m, n, A, lda, s, U, ldu, VT,
            ldvt, work, lwork, info)
           return U, s, VT, work, info
       end
    end
end

print(1+1)

jobu = MagmaAllVec
jobvt = MagmaAllVec
A = cu(zeros(2,2))
A[1,1]=2.0
A[1,2]=1.0
A[2,1]=1.0
A[2,2]=2.0
ldu=2
ldvt=2
lwork=134
success=magmaInit()

print("Magma Initialization success=")
print(success)
print('\n')

U, s, VT, work, info = gesvd!(jobu,jobvt,A,ldu,ldvt,lwork)

print("U=")
print(U)
print('\n')

print("s=")
print(s)
print('\n')

print("V**T=")
print(VT)
print('\n')

print("work=")
print(work)
print('\n')
```

```
print("info=")
print(info)
print('\n')
```