Parallel & Distributed Computing: Lecture 39

Alberto Paoluzzi

January 20, 2020

Julia on the GPU

Some web resources

https://nextjournal.com/sdanisch/julia-gpu-programming
https://discourse.julialang.org/t/can-we-make-a-comparison-overview-of-different-gpu-computing-implementations/24294
https://github.com/JuliaGPU/CuArrays.jl/tree/master/src/sparse
https://discourse.julialang.org/t/cudanative-is-awesome/17861/6
https://docs.nvidia.com/cuda/cusparse/index.html
https://discourse.julialang.org/t/performance-of-view-with-cuarrays/17387
https://juliagpu.gitlab.io/CUDA.jl/
https://juliagpu.github.io/GPUArrays.jl/stable/#The-Abstract-GPU-interface-1
https://nextjournal.com/sdanisch/julia-gpu-programming
https://juliagpu.gitlab.io/CUDA.jl/tutorials/introduction/
https://github.com/JuliaAttic/CUSPARSE.jl
https://github.com/JuliaGPU/CuArrays.jl https://github.com/JuliaGPU/CUDA.jl
https://github.com/JuliaGPU/GPUArrays.jl https://github.com/JuliaGPU
https://juliagpu.gitlab.io/CUDA.jl/usage/overview/
https://discourse.julialang.org/t/performance-of-view-with-cuarrays/17387
https://discourse.julialang.org/search?q=CUSPARSE

Too many implementations?

Can we make a comparison/overview of different GPU computing implementat...

Domains GPU

freemint 1 May '19

Ok let me summarize for compute:

GPUArrays.jl provides an abstract interface which is implemented by

```
KnetArrays
CLArrays
HSAArrays
CuArrays
ArrayFire
```

Some of these implementations are partial and broken. Are there some general heuristics which library to pick when?

The libraries

```
AMDGPUnative
CUDAnative
```

implement the same interface. CUDAnative is more developed since it is older. `OpenCL` provides a level at a similar level of abstraction but is not related to the VENDORnative family.

Did i get that right?

Here are some other questions:
Is there a performance difference between passing a VENDORnative kernel GPU arrays and broadcasting a kernel over an VENDORArrays array?

May 2019

3 / 7
May 2019

May 2019

Intro to GPU programming

sdanisch tutorial

An Introduction to GPU Programming in Julia

How does the GPU work

This article aims to give a quick introduction about how GPUs work and specifically give an overlook of the current Julia GPU ecosystem and how easy it is to get simple GPU programs running. To make things easier, you can run all the code samples directly in the article if you have an account and click on **edit**.

First of all, what is a GPU anyways?

A GPU is a massively parallel processor, with a couple of thousand parallel processing units. For example the Tesla k80, which is used in this article, offers 4992 parallel CUDA cores. GPUs are quite different from CPUs in terms of frequencies, latencies and hardware capabilities, but this is somewhat similar to a slow CPU with 4992 cores!

```
using CUDAdrv; CUDAdrv.name(CuDevice(0))
```

✓ 1.4s Julia Julia GPU+Flux

```
"Tesla K80"
```

The sheer number of parallel threads one can launch can yield massive speed-ups, but also makes it harder to utilize the GPU. Let's have a detailed look at the disadvantages one buys into when utilizing this raw power:

A gentle introduction to parallelization and GPU programming in Julia

CUDA.jl

Search docs

Tutorials / Introduction — Edit on GitHub

Introduction

A gentle introduction to parallelization and GPU programming in Julia

Julia has first-class support for GPU programming: you can use high-level abstractions or obtain fine-grained control, all without ever leaving your favorite programming language. The purpose of this tutorial is to help Julia users take their first step into GPU computing. In this tutorial, you'll compare CPU and GPU implementations of a simple calculation, and learn about a few of the factors that influence the performance you obtain.

This tutorial is inspired partly by a blog post by Mark Harris, An Even Easier Introduction to CUDA, which introduced CUDA using the C++ programming language. You do not need to read that tutorial, as this one starts from the beginning.

A simple example on the CPU

We'll consider the following demo, a simple calculation on the CPU.

```
N = 2^20
x = fill(1.0f0, N)  # a vector filled with 1.0 (Float32)
y = fill(2.0f0, N)  # a vector filled with 2.0

y .+= x             # increment each element of y with the corresponding element of x
```

```
1048576-element Array{Float32,1}:
 3.0
 3.0
 3.0
 3.0
```

CUDA.jl quick start and overview 1/2

CUDA.jl

Search docs

Home

- Quick Start
- Acknowledgements
- Supporting and Citing

Tutorials

- Introduction

Installation

- Overview
- Conditional Dependency
- Troubleshooting

Usage

- Overview
- Workflow
- Memory Management
- Multiple GPUs

FAQ

CUDA programming in Julia

Julia has several packages for programming NVIDIA GPUs using CUDA. Some of these packages focus on performance and flexibility, while others aim to raise the abstraction level and improve performance. This website will introduce the different options, how to use them, and what best to choose for your application. For more specific details, such as API references or development practices, refer to each package's own documentation.

If you have any questions, please feel free to use the `#gpu` channel on the Julia slack, or the GPU domain of the Julia Discourse.

Quick Start

The Julia CUDA stack requires a functional CUDA-setup, which includes both a driver and matching toolkit. Once you've set that up, continue by installing the three core packages:

```
using Pkg
Pkg.add(["CUDAdrv", "CUDAnative", "CuArrays"])
```

To make sure everything works as expected, try to load the packages and if you have the time execute their test suites:

```
using CUDAdrv, CUDAnative, CuArrays

using Pkg
Pkg.test(["CUDAdrv", "CUDAnative", "CuArrays"])
```

For more details on the installation process, consult the Installation section. To understand the toolchain in more detail, have a look at the tutorials in this manual. **It is highly recommended that new users start with the Introduction tutorial**. For an overview of the available functionality, read the Usage section. The following resources may also be of interest:

- Effectively using GPUs with Julia: video, slides
- How Julia is compiled to GPUs: video

CUDA.jl quick start and overview 2/2

CUDA.jl

Search docs

Tutorials

- Introduction

Installation

- Overview
- Conditional Dependency
- Troubleshooting

Usage

- Overview
  - CuArrays.jl
  - CUDAnative.jl
  - CUDAdrv.jl
  - Others
- Workflow
- Memory Management
- Multiple GPUs

FAQ

Usage / Overview

Edit on GitHub

Overview

There are three key packages that make up the Julia stack for CUDA programming:

- CUDAdrv.jl for interfacing with the CUDA APIs
- CUDAnative.jl for writing CUDA kernels
- CuArrays.jl for working with CUDA arrays

You probably won't need all three of these packages: Much of the Julia CUDA programming stack can be used by just relying on the `CuArray` type, and using platform-agnostic programming patterns like `broadcast` and other array abstractions.

CuArrays.jl

The CuArrays.jl package provides an essential part of the toolchain: an array type for managing data on the GPU and performing operations on its elements. Every application should use this type, if only to manage memory because it is much easier then doing manual memory management:

```
using CuArrays

a = CuArray{Int}(undef, 1024)

# essential memory operations, like copying, filling, reshaping, ...
b = copy(a)
fill!(b, 0)
@test b == CuArrays.zeros(Int, 1024)

# automatic memory management
a = nothing
```

Julia GPU packages

Pinned repositories

JuliaGPU

GPU Computing in Julia

https://juliagpu.org/ Verified

Pinned repositories

CuArrays.jl
A Curious Cumulation of CUDA Cuisine
Julia ★ 219 Forks 73

CUDAnative.jl
Julia support for native CUDA programming
Julia ★ 353 Forks 53

OpenCL.jl
OpenCL Julia bindings
Julia ★ 186 Forks 32

AMDGPUnative.jl
Julia interface to AMD/Radeon GPUs
Julia ★ 31 Forks 1

ArrayFire.jl
Julia wrapper for the ArrayFire library
Julia ★ 157 Forks 30

juliagpu.org
The JuliaGPU landing page.
HTML ★ 3

Array operations defined for all kind of GPU backends 1/2

Why another GPU array package in yet another language?

Julia offers great advantages for programming the GPU. This blog post outlines a few of those.

E.g., we can use Julia's JIT to generate optimized kernels for map/broadcast operations.

This works even for things like complex arithmetic, since we can compile what's already in Julia Base. This isn't restricted to Julia Base, GPUArrays works with all kind of user defined types and functions!

GPUArrays relies heavily on Julia's dot broadcasting. The great thing about dot broadcasting in Julia is, that it actually fuses operations syntactically, which is vital for performance on the GPU. E.g.:

```
out .= a .+ b ./ c .+ 1
#turns into this one broadcast (map):
broadcast!(out, a, b, c) do a, b, c
    a + b / c + 1
end
```

Will result in one GPU kernel call to a function that combines the operations without any extra allocations. This allows GPUArrays to offer a lot of functionality with minimal code.

Also, when compiling Julia for the GPU, we can use all the cool features from Julia, e.g. higher order functions, multiple dispatch, meta programming and generated functions. Checkout the examples, to see how this can be used to emit specialized code while not losing flexibility:

Array operations defined for all kind of GPU backends. 1/2

GPUArrays.jl

Search docs

Home

- The Abstract GPU interface
- The abstract TestSuite

The Abstract GPU interface

janvitek.org/pubs/oopsla18a.pdf

Different GPU computation frameworks like CUDA and OpenCL, have different names for accessing the same hardware functionality. E.g. how to launch a GPU Kernel, how to get the thread index and so forth. GPUArrays offers a unified abstract interface for these functions. This makes it possible to write generic code that can be run on all hardware. GPUArrays itself even contains a pure Julia implementation of this interface. The julia reference implementation is a great way to debug your GPU code, since it offers more informative errors and debugging information compared to the GPU backends - which mostly silently error or give cryptic errors (so far).

You can use the reference implementation by using the `GPUArrays.JLArray` type.

The functions that are currently part of the interface:

The low level dim + idx function, with a similar naming scheme as in CUDA:

```
# with * being either of x, y or z
blockidx_*(state), blockdim_*(state), threadidx_*(state), griddim_*(state)
# Known in OpenCL as:
get_group_id,      get_local_size,    get_local_id,       get_num_groups
```

Higher level functionality:

`GPUArrays.gpu_call` — Function

```
gpu_call(kernel::Function, A::GPUArray, args::Tuple, configuration = length(A))
```

Calls function `kernel` on the GPU. `A` must be an GPUArray and will help to dispatch to the correct GPU backend and supplies queues and contexts. Calls the kernel function with `kernel(state, args...)`, where state is dependant on the backend and can be used for getting an index into `A` with `linear_index(state)`. Optionally, a launch configuration can be supplied in the following way:

CUDA programming in Julia

CUDA programming in Julia — Introduction 1/3

CUDA.jl

CUDA programming in Julia

This repository hosts a Julia package that bundles functionality from several other packages for CUDA programming, and provides high-level documentation and tutorials for effectively using CUDA GPUs from Julia. The documentation is accessible at juliagpu.gitlab.io.

CUDA.jl includes functionality from the following packages:

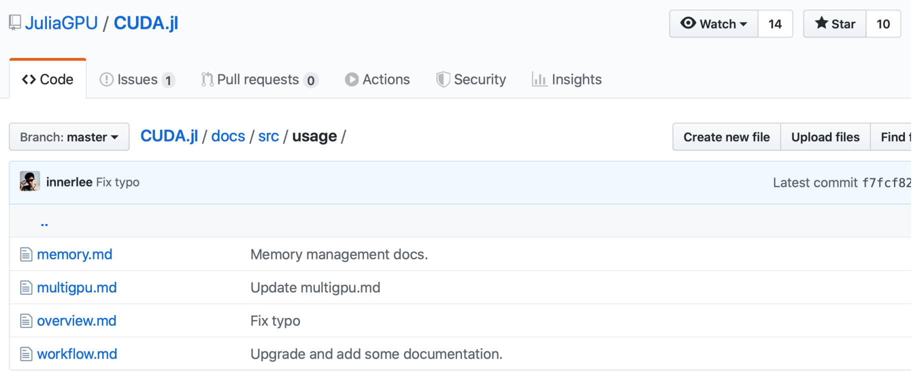
- CUDAdrv.jl: interface to the CUDA driver
- CUDAnative.jl: kernel programming capabilities
- CuArrays.jl: GPU array abstraction

For details on the APIs that these packages expose, refer to the associated documentation.

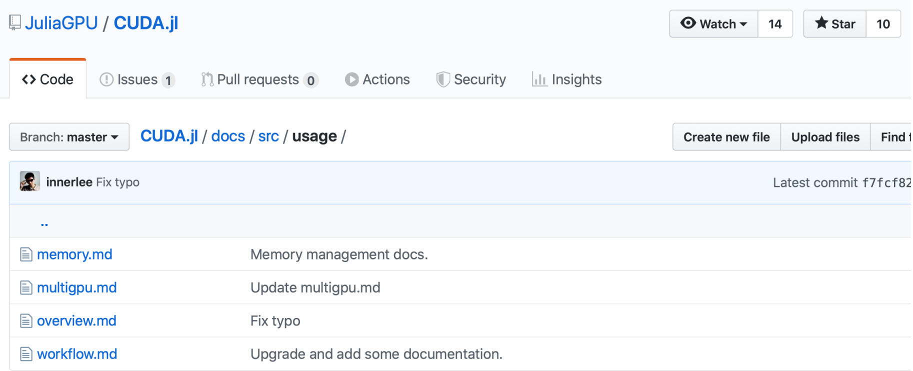
API stability

Versioning of this package follows SemVer as used by the Julia package manager: Depending on a specific major version of CUDA.jl should guarantee that your application will not break, as long as it only uses functionality from the package's public API. For CUDA.jl, this API includes certain non-exported functions and macros that would otherwise clash with implementations in Julia. Refer to src/CUDA.jl for more details.

CUDA programming in Julia — Introduction 2/3



CUDA programming in Julia — Introduction 3/3



GPU arrays documentation 1/2

GPUArrays Documentation

GPUArrays is an abstract interface for GPU computations. Think of it as the AbstractArray interface in Julia Base but for GPUs. It allows you to write generic julia code for all GPU platforms and implements common algorithms for the GPU. Like Julia Base, this includes BLAS wrapper, FFTs, maps, broadcasts and mapreduces. So when you inherit from GPUArrays and overload the interface correctly, you will get a lot of functionality for free. This will allow to have multiple GPUArray implementation for different purposes, while maximizing the ability to share code. Currently there are two packages implementing the interface namely CLArrays and CuArrays. As the name suggests, the first implements the interface using OpenCL and the latter uses CUDA.

The Abstract GPU interface

Different GPU computation frameworks like CUDA and OpenCL, have different names for accessing the same hardware functionality. E.g. how to launch a GPU Kernel, how to get the thread index and so forth. GPUArrays offers a unified abstract interface for these functions. This makes it possible to write generic code that can be run on all hardware. GPUArrays itself even contains a pure Julia implementation of this interface. The julia reference implementation is a great way to debug your GPU code, since it offers more informative errors and debugging information compared to the GPU backends - which mostly silently error or give cryptic errors (so far).

You can use the reference implementation by using the `GPUArrays.JLArray` type.

The functions that are currently part of the interface:

The low level dim + idx function, with a similar naming scheme as in CUDA:

GPU arrays documentation 2/2

CUDAnative is awesome!

Domains GPU

LaurentPlagne 2 cuchxq Dec '18

Hi,
I apologize because my statement is wrong or at least unclear. AFAIK, there is no centralized Julia GPGPU programming guide for beginners, and it is almost impossible to learn GPGPU only from Julia based documentation.

What I wanted to say is that Julia GPU tools (especially CUDANative.jl) makes GPGPU learning much easier an efficient in different ways:

- Large reduction of the GPGPU setup cost compared to C/C++ CUDA programming where you have to make all the compilation/installation setup (environement.CMake, test of drivers…). On my ubuntu system, once I have installed the CUDA tookit package (apt-get install nvidia-cuda-toolkit), everything works nicely from Julia.
- Nice integration with Julia native Arrays: the syntax to create and transfer a Julia Array to the GPU and the copy back to the CPU is transparent and intuitive:

```
n=1024
a=ones(Float32,n,n) # normal CPU matrix of float
d_a = CuArray(a)    # copy to a GPU
copyto!(a,d_a)      # copy back from CPU to GPU
```

- Compared to C/C++ dynamic language like Julia (this is also true for PyCUDA) accelerates experiments on optimal threads and block numbers.
- CUDA kernels syntax is Julia syntax for arrays (real multi-dim arrays). For example, compare the (excellent)
  Mark Harris tutorials nvidia_transpose 6, example to its Julia translation:

Nov 2018

7 / 13
Dec 2018

Dec 2018

Performance of view with CuArrays

CUDANative is awesome!

Domains GPU

LaurentPlagne 2 cuchxq Dec '18

Hi,
I apologize because my statement is wrong or at least unclear. AFAIK, there is no centralized Julia GPGPU programming guide for beginners, and it is almost impossible to learn GPGPU only from Julia based documentation.

What I wanted to say is that Julia GPU tools (especially CUDANative.jl) makes GPGPU learning much easier an efficient in different ways:

- Large reduction of the GPGPU setup cost compared to C/C++ CUDA programming where you have to make all the compilation/installation setup (environement.CMake, test of drivers…). On my ubuntu system, once I have installed the CUDA tookit package (apt-get install nvidia-cuda-toolkit), everything works nicely from Julia.
- Nice integration with Julia native Arrays: the syntax to create and transfer a Julia Array to the GPU and the copy back to the CPU is transparent and intuitive:

```
n=1024
a=ones(Float32,n,n) # normal CPU matrix of float
d_a = CuArray(a)    # copy to a GPU
copyto!(a,d_a)      # copy back from CPU to GPU
```

- Compared to C/C++ dynamic language like Julia (this is also true for PyCUDA) accelerates experiments on optimal threads and block numbers.
- CUDA kernels syntax is Julia syntax for arrays (real multi-dim arrays). For example, compare the (excellent)
  Mark Harris tutorials nvidia_transpose 6, example to its Julia translation:

Nov 2018

7 / 13
Dec 2018

Dec 2018

CUSPARSE

Cuda Sparse Arrays in Julia 1/2

Note: This package is being phased out.

The same functionality is available with CuArrays.

CUSPARSE.jl

Build status:

Code coverage: codecov unknown

Julia bindings for the NVIDIA CUSPARSE library. CUSPARSE is a high-performance sparse matrix linear algebra library.

Table of Contents

Cuda Sparse Arrays in Julia 2/2

JuliaGPU / **CuArrays.jl**

Sponsor | Watch 25 | Star 219

Code | Issues 92 | Pull requests 10 | Actions | Security | Insights

Branch: master | **CuArrays.jl / src / sparse /** | Create new file | Upload files | Fin

maleadt Improve error display. … | Latest commit 0284c

..

CUSPARSE.jl	Adapt to initialization API changes.
array.jl	remove extraneous where and add some tests
error.jl	Improve error display.
interfaces.jl	Reorganize files of subprojects.
libcusparse.jl	Use at-runtime_ccall's ability to delay the library lookup.
libcusparse_common.jl	Regenerate the CUSPARSE wrappers.
util.jl	Regenerate the CUSPARSE wrappers.
wrappers.jl	Reindent to reduce visual ASCII salad a little.

Topic CUSPARSE on Julia Discourse forum

julia

CUSPARSE

+ New Topic

14 results for **CUSPARSE**

Sort by Relevance

Support for Sparse Matrices on GPU (CUSPARSE)

Domains GPU gpu knet flux

Sep '18 - Do any of the ML libraries (Flux, KNet) support sparse matrices on the GPU?

Initializing Sparse Matrices with CuArrays.jl

Domains GPU first-steps

Feb '19 - ...arse matrices with CuArrays.jl, but I have not found a way to initialize them. I tried to with the example described at https://github.com/JuliaAttic/**CUSPARSE**.jl but did not worked. The used command and the error are shown below. Does anyone has a MWE with sparse matrices? Thank you! julia > CuArrays.allows...

Use GPU to generate known sparse matrix

Domains GPU

14d - ...x in CSR or CSC format in GPU. The code is as follows: using CuArrays D_I = CuArray{Int64,1}(I) D_J = CuArray{Int64,1}(J) D_V = CuArray{Float32,1}(V) **CUSPARSE**.sparse(D_I, D_J, D_V) Result: ┌ Warning: Performing scalar operations on GPU arrays: This is very slow, consider disallowing these operations with `a...

Advanced Search

Posted by

Categorized

All categories

Tagged

All the above tags

Only return topics/posts...

Matching in title only

I liked

In my messages

I read

any

Where topics

any

Posted

before

Nvidia cusparse

DEVELOPER ZONE CUDA TOOLKIT DOCU Click to close this tab; Option-click to close all tabs except this one Search

CUDA Toolkit v10.2.89

cuSPARSE

cuSPARSE (PDF) - v10.2.89 (older) - Last updated November 28, 2019 - Send Feedback

cuSPARSE

The API reference guide for cuSPARSE, the CUDA sparse matrix library.

1. Introduction

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on top of the NVIDIA® CUDA™ runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++. The library routines can be classified into four categories:

- Level 1: operations between a vector in sparse format and a vector in dense format
- Level 2: operations between a matrix in sparse format and a vector in dense format
- Level 3: operations between a matrix in sparse format and a set of vectors in dense format (which can also usually be viewed as a dense tall matrix)
- Conversion: operations that allow conversion between different matrix formats, and compression of csr matrices.

The cuSPARSE library allows developers to access the computational resources of the NVIDIA graphics processing unit (GPU), although it does not auto-parallelize across multiple GPUs. The cuSPARSE API assumes that input and output data reside in GPU (device) memory, unless it is explicitly indicated otherwise by the string `DevHostPtr` in a function parameter's name (for example, the parameter `*resultDevHostPtr` in the function `cusparse<t>doti()`).

It is the responsibility of the developer to allocate memory and to copy data between GPU memory and CPU memory using standard CUDA runtime API routines, such as `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`, and `cudaMemcpyAsync()`.

1.1. Naming Conventions

The cuSPARSE library functions are available for data types `float`, `double`, `cuComplex`, and `cuDoubleComplex`. The sparse Level 1, Level 2, and Level 3 functions follow this naming convention:

`cusparse`<`t`>[<`matrix data format`>]<`operation`>[<`output matrix data format`>]

where <`t`> can be `S`, `D`, `C`, `Z`, or `X`, corresponding to the data types `float`, `double`, `cuComplex`, `cuDoubleComplex`, and the generic type, respectively.

The <`matrix data format`> can be `dense`, `coo`, `csr`, `csc`, or `hyb`, corresponding to the dense, coordinate, compressed sparse row, compressed sparse column, and hybrid storage formats, respectively.

Finally, the <`operation`> can be `axpyi`, `doti`, `dotci`, `gthr`, `gthrz`, `roti`, or `sctr`, corresponding to the Level 1 functions; it also can be `mv` or `sv`, corresponding to the Level 2 functions, as well as `mm` or `sm`, corresponding to the Level 3 functions.

All of the functions have the return type `cusparseStatus_t` and are explained in more detail in the chapters that follow.