Portable and Efficient Julia Code for Heterogeneous Hardware Systems

Julian Samaroo (RSE @ MIT JuliaLab)

Why care about GPUs? CPUs are easier!

- GPUs basically are just CPUs with really good SIMD and fast memory
- GPU programming can be more difficult, but Julia does this better
- Some problems (ML/AI, image processing, crypto) only feasible with GPUs

Why AMD? NVIDIA is king!

- NVIDIA’s CUDA is dominant
- Competition is good!
- Unique features (proper UVM, debuggable drivers)
- Frontier/LUMI are available and are AMD-only
- HIP == CUDA
- Many of the same libraries exist (rocBLAS vs. CUBLAS, MIOpen vs. CUDNN)
- Julia support is solid

Who am I? What do I do?

- RSE @ MIT’s JuliaLab since April 2020
- My core mission: Develop better support in Julia for AMD GPUs and other heterogeneous compute
- Author and maintainer of AMDGPU.jl library

AMDGPU.jl

AMD GPU (ROCm) programming in Julia

AMDGPU.jl from 10,000 feet

- GPU computing library for AMD GPUs
- Compiles Julia code to GCN
- Manages GPU devices via ROCm’s HSA and HIP
- Access to ROCm’s scientific computing libraries (rocBLAS, rocFFT, MIOpen, etc.)
- Integration Julia libraries (ForwardDiff, etc.)

The early days - CUDA is the only game in town

- Pre-2020, Julia only had CUDA GPU support (sad!)
- Mostly hard-wired CUDA support, not fully generic
- ROCm was in its infancy
- I had AMD GPUs, and wanted to use them
- Valentin Churavy encouraged me to make it happen

Pre-JuliaLab development

- I ported AMD’s C examples to Julia
- Got bits and pieces working, launched pre-compiled vector add kernel
- Reused existing code Julia’s CUDA libs (quite time-intensive)
- Basic GCN compilation through LLVM
- Basically just a toy set of libraries, limited functionality

Some interesting features though!

- Unified virtual memory allowed some things to “just work” via CPU
- “Hostcalls” let the GPU call CPU functions dynamically - still unique to AMDGPU.jl even today
- Argument-based synchronization for easy cross-kernel sync

Joining the JuliaLab + DARPA project

- Valentin recruited me to JuliaLab
- Primary goals are developing Julia’s AMD GPU computing and distributed computing support
- As COVID befalls the world, AMDGPU development picks up steam
- ROCm is concurrently growing in features and notoriety
- My work funded by a DARPA grant for the “three-Ps”: Productivity, Performance, and Portability

DARPA grant pushes AMD GPU support forward

- Lots of work to do!
- Thanks to Valentin and Tim Besard (CUDA.jl maintainer) for guidance and mentorship
- Contributions from various interested Julia programmers
- At project end, demo’d ocean simulation across multiple servers, CPUs and AMD+NVIDIA GPUs

AMDGPU.jl becomes a truly capable library

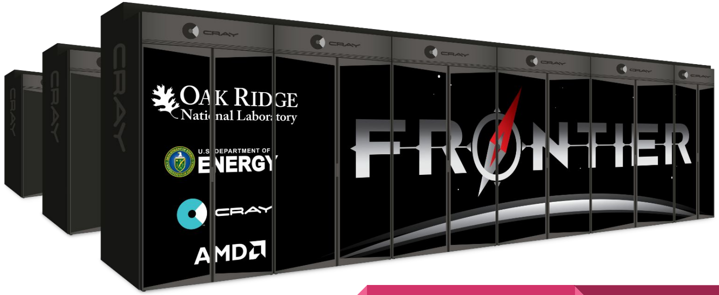
- Thanks to DARPA, Julia’s AMD GPU development accelerates
- Lots of code sharing - common code was moved to GPUArrays.jl and GPUCompiler.jl
- Less redundant code = less bugs, more features
- Code sharing promoted creation of oneAPI.jl for Intel GPUs and Metal.jl for Apple GPUs

Struggles along the way…

- Bugs in HSA, HIP, and LLVM - some fixed by AMD, some worked around
- Understanding LLVM (really compilers) takes time
- Adoption takes a long time (especially with CUDA’s dominance)
- New architecture means new never-before-seen bugs

Current day: Ice Flow team and ExaSGD

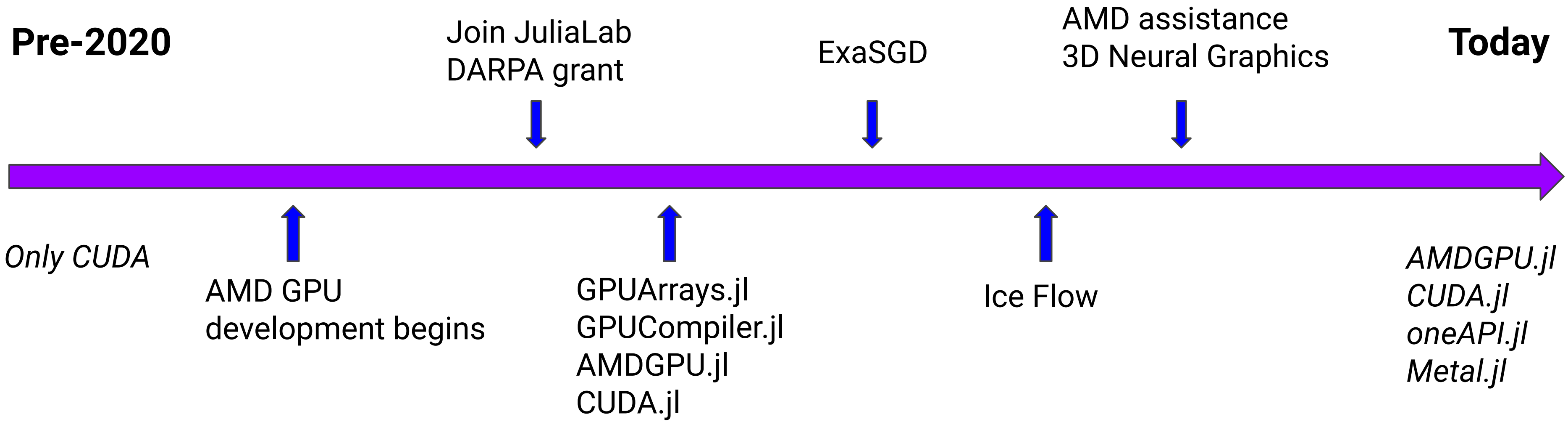
- LUMI and Frontier starting to come online, need Julia support
- New users and contributors: Ice flow team (Ludovic Rass, Samuel Omlin, Ivan Utkin @ CSCS) and ExaSGD team (Michel Schanen @ Argonne) helped improve and validate AMDGPU.jl's performance at scale
- AMDGPU.jl is on its way to the exascale!



AMD enters the room - 3D Neural Graphics

- Chrisitian Laforte and Mike Mantor (AMD) lead Julia 3D Neural Graphics team
- AMDGPU.jl performance improved for single-GPU
- Anton Smirnov (AMD) becomes co-maintainer
- JuliaNeuralGraphics libraries are open source

History Overview



Pre-2020

Only CUDA

AMD GPU
development begins

Join JuliaLab
DARPA grant

GPUArrays.jl
GPUCompiler.jl
AMDGPU.jl
CUDA.jl

ExaSGD

Ice Flow

AMD assistance
3D Neural Graphics

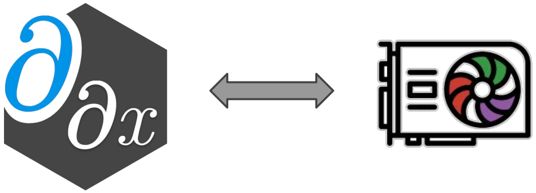
Today

AMDGPU.jl
CUDA.jl
oneAPI.jl
Metal.jl

Beyond AMDGPU.jl: Code portability

- Technology evolves quickly, code must keep pace
- GPUArrays.jl for arrays
- KernelAbstractions.jl for kernels
- Enzyme.jl for autodiff
- New backends are easy to add (e.g. Metal.jl)

Automatic Differentiation on GPUs



Layers	**Julia Code**			
Abstraction	GPUArrays.jl		KernelAbstractions.jl	
Generation	CUDA.jl	AMDGPU.jl	oneAPI.jl	Host/CPU
IR / C API	CUDA	ROCm	Intel Compute Runtime	LLVM

Scientific Machine Learning in Julia

SciML - Differential Equations solvers and more

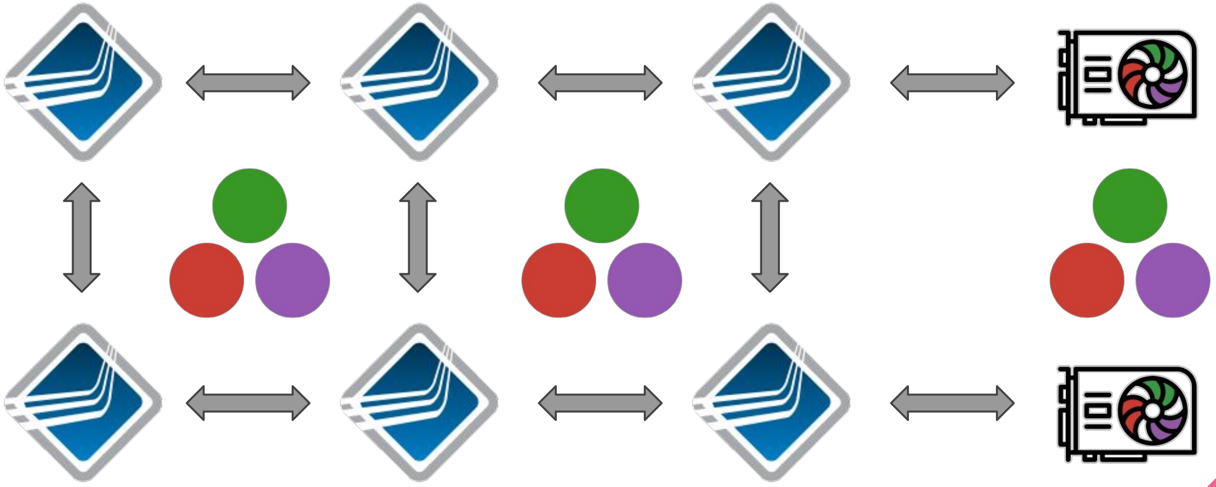
- Support AMD GPUs and others for pure-Julia solvers
- Parallelization across ensembles with GPUs

FluxML - Machine Learning and AI

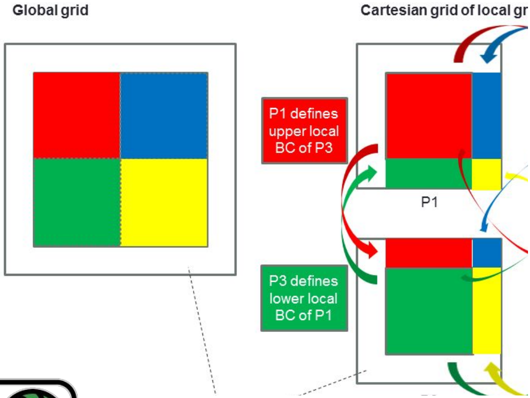
- Recently added AMDGPU support
- Integrates with SciML incl. GPU support

When one GPU just isn’t enough

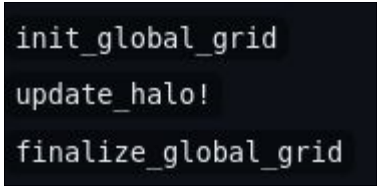
- MPI: the defacto standard for multi-node
- But writing MPI is not fun
- Many useful abstractions to make this easier!



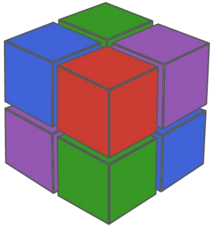
ImplicitGlobalGrid and ParallelStencil


Global grid
Cartesian grid of local gr
P1 defines upper local BC of P3
P1
P3 defines lower local BC of P1

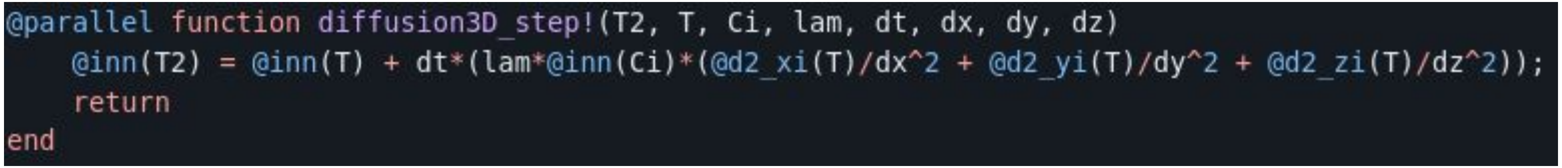
- ImplicitGlobalGrid.jl: Handles async halo region updates



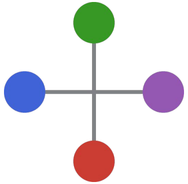
```
init_global_grid
update_halo!
finalize_global_grid
```



- ParallelStencil.jl: Helpers for stencils over ImplicitGlobalGrid



```
@parallel function diffusion3D_step!(T2, T, Ci, lam, dt, dx, dy, dz)
    @inn(T2) = @inn(T) + dt*(lam*@inn(Ci)*(@d2_xi(T)/dx^2 + @d2_yi(T)/dy^2 + @d2_zi(T)/dz^2));
    return
end
```



Non-traditional HPC: I don’t need a supercomputer?

- “Big homogenous grid on big homogenous cluster” pattern doesn’t always work
- Irregular problems: machine learning (“AI”), streaming image processing, etc.
- Users have unique setups - Beowulf clusters, laptop + workstation, cloud resources, etc.
- Not so many libraries for this in Julia, but they do exist

Dagger.jl for heterogeneous computing

- Task runtime + scheduler that utilizes whatever resources the user has available
- Support for automatically using AMD GPUs (and other vendors' GPUs) for supported operations
- Many different APIs and abstractions, such as Arrays, Tables, and Tasks
- MPI support planned for this summer
- Built-in parallel I/O subsystem
- Integration with Flux.jl for machine learning
- Planned integration with other languages

The future of Julia HPC is bright!

To summarize:

- AMDGPU.jl is alive and well, and rapidly maturing
- AMDGPU.jl is running on the latest, most powerful supercomputers
- Julia’s GPU libraries are sustainably maintained by sharing common code
- Julia has excellent abstractions for HPC at extreme scales
- Julia’s non-traditional HPC landscape is growing to meet user’s needs
- Julia makes it easy to become an HPC practitioner and scale codes