Oak Ridge National Laboratory

JACC
JULIA FOR ACCELERATORS

A Julia Package for HPC Meta-Programming and Performance Portability

Philip Fackler

In collaboration with Pedro Valero-Lara, William Godoy, Het Mankad, Keita Teranishi, and Jeffrey Vetter

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

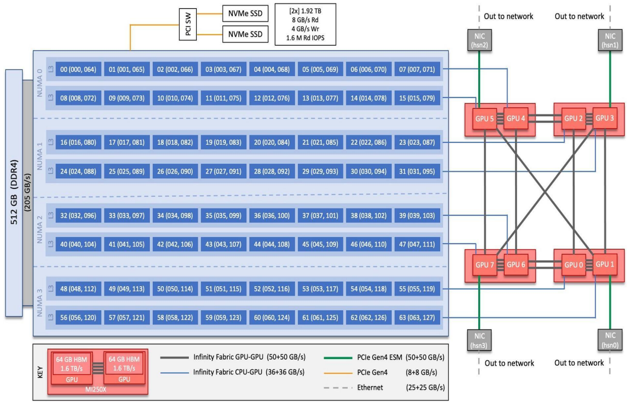
Why Julia?

- Easy to use
- Easy performance
- Easily supported (JIT on top of LLVM)
- Sensible syntax for science
- Easy for HPC packages to provide native syntax (not just wrapping)
- Integrated packaging and testing (reproducibility)
- Easy portability
  (usually just works on a new system…from a developer's standpoint)

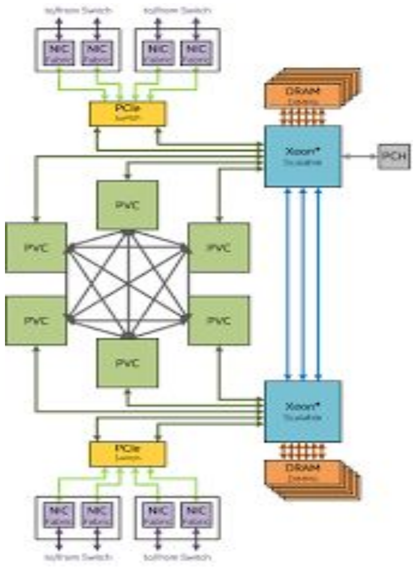
https://julialang.org/

Motivation: Programming Productivity Performance Portability Performance “Practicability”?

- Varied systems
- Varied user expertise

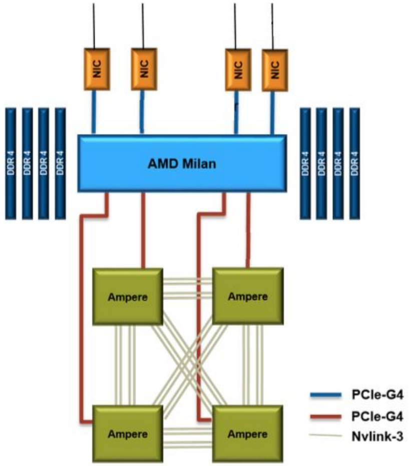


AMD ROCm



oneAPI

SYCL



NVIDIA CUDA

OpenMP

JACC.jl, Julia for ACCelerators

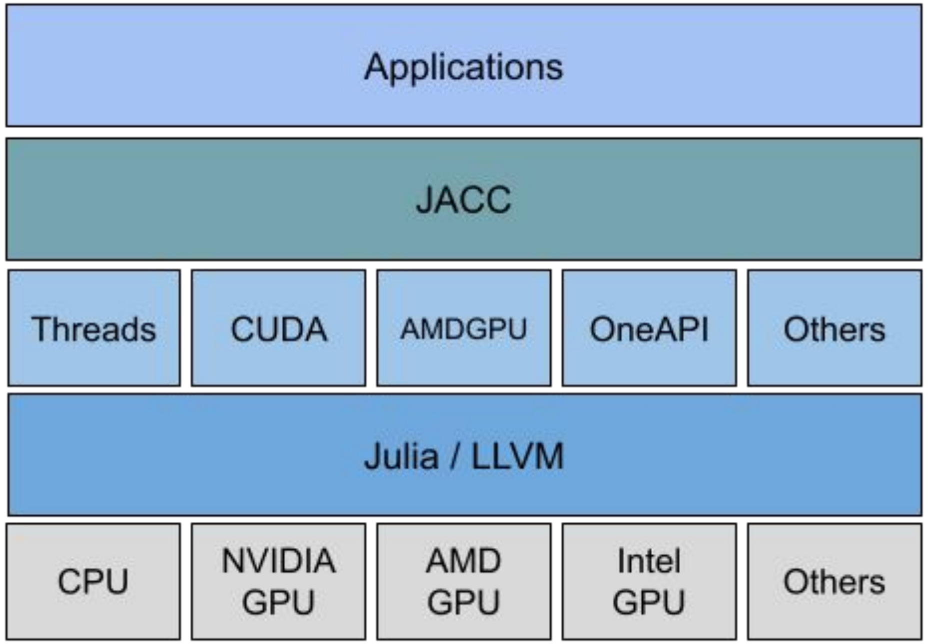
Simple to use performance portable model

- High-level performance portability model (parallel_for & parallel_reduce)
- Unified Julia front-end on top of multiple backends
  - Threads, CUDA, AMDGPU, oneAPI (Metal coming soon)
- Hide low-level, device-specific implementation
  - Memory layout, granularity, etc.
- ***Brings HPC closer to domain scientists***

JACC JULIA FOR ACCELERATORS

https://github.com/JuliaORNL/JACC.jl

https://ieeexplore.ieee.org/document/10820713



Getting started with JACC

1. Add JACC as a dependency

```
julia> Pkg.add("JACC")
```

2. Set backend

```
julia> using JACC
julia> JACC.set_backend("cuda")
```

   - Adds backend package

3. Restart julia

4. Load extension before use

```
julia> using JACC
julia> JACC.@init_backend
```

   - Imports backend package -> activates extension

Oak Ridge National Laboratory

JACC paradigm basics - arrays

No JACC-defined array type

```
JACC.array(x::Base.Array)
```

Constructs a backend-specific array (copying to device if necessary)

```
JACC.array_type()
```

Provides type of array that would be returned by JACC.array, e.g., CuArray

```
function foo(x::JACC.array_type(){Float64, 1})
    # ...
end
```

```
JACC.ones, JACC.zeros, JACC.fill
```

JACC paradigm basics - parallel_for

Moving from a serial loop to a JACC.parallel_for

```
for i = 1:N
    @inbounds x[i] += alpha * y[i]
end
```

```
JACC.parallel_for(N, alpha, dx, dy) do i, alpha, x, y
    @inbounds x[i] += alpha * y[i]
end
```

(or, using predefined function)

```
function axpy(i, alpha, x, y)
    @inbounds x[i] += alpha * y[i]
end
JACC.parallel_for(SIZE, axpy, alpha, dx, dy)
```

JACC paradigm basics - parallel_reduce

Examples:

```
elem_sum = JACC.parallel_reduce(a)
elem_min = JACC.parallel_reduce(min, a)

dp = JACC.parallel_reduce(10, a, b) do i, a, b
    @inbounds a[i] * b[i]
end
```

API overview:

```
parallel_reduce(N, f, x...; op, init) -> typeof(init)

parallel_reduce(N, f, x...) = parallel_reduce(N, f, x…; op = +, init = 0.0)

parallel_reduce([op = +,] a::AbstractArray; init = default_init(eltype(a), op)) -> typeof(init)

    # op ∈ {+, *, min, max, custom}
```

Extra Things

LaunchSpec for fine granularity

```
launch_spec(; kw...)

accepts keyword arguments:
     stream
     threads
     blocks
     shmem_size
     sync


spec = JACC.launch_spec(;
          sync = false, threads = 1000)

JACC.parallel_for(spec, N, a_device) do i, a
    @inbounds a[i] += 5.0
end
```

JACC.shared: on-chip shmem

```
function spectral(i, j, image, filter,
num_bands)
    for b in 1:num_bands
        @inbounds image[b, i, j] *= filter[j]
    end
end
```

(just add one line)

```
function spectral(i, j, image, filter,
num_bands)
    filter_sh = JACC.shared(filter)
    for b in 1:num_bands
        @inbounds image[b, i, j] *= filter_sh[j]
    end
end
```

Extra Things (multi-device nodes) (experimental)

JACC.Multi: multi-device “parallelism”

```
alpha = 2.5
dx = JACC.Multi.array(x)
dy = JACC.Multi.array(y)

JACC.Multi.parallel_for(N,a,dx,dy) do i, a, x, y
    @inbounds x[i] += alpha * y[i]
end
```

With extra methods for handling ghost elements

JACC.Async: multi-device “concurrency”

```
JACC.Async.parallel_for(1, ...)
JACC.Async.parallel_for(2, ...)
JACC.Async.synchronize()

# parallel_reduce return value is on device
res_d = JACC.Async.parallel_reduce(1, ...)
res = JACC.to_host(res_d)[]
```

Applications & Future Work

- Ongoing Efforts
  - Performance benchmarking (closing gaps)
  - More intuitive kernel launch
  - JACC.Auto : Autotuning
  - Have ideas? Send us a message!
- Examples and applications using JACC
  - Look at the tests
  - Tatiana's 7-point stencil
  - https://github.com/JuliaORNL/MiniVATES.jl
  - https://github.com/JuliaORNL/JACC-applications
  - https://github.com/JuliaORNL/GrayScott.jl
- Other HPC Julia tutorial resources
  - SC24 tutorial

Project:

- William F Godoy (ORNL)
- Pedro Valero Lara (ORNL)
- Philip W Fackler (ORNL)
- Narasinga Rao Miniskar (ORNL)
- Aaron Young (ORNL)
- Keita Teranishi (ORNL)

Contributions:

- Het Mankad (ORNL)
- Julian Samaroo (MIT)
- Michel Schanen (ANL)
- Karl Pierce (Flatiron Institute)
- Kelly Tang
- Tatiana Melnichenko

Thanks!

This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Sponsors:
ASCR Competitive Portfolios: Fairbanks project
ASCR NGSST PESO and S4PST projects
ASCR Facilities: OLCF/NERSC

Questions?