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

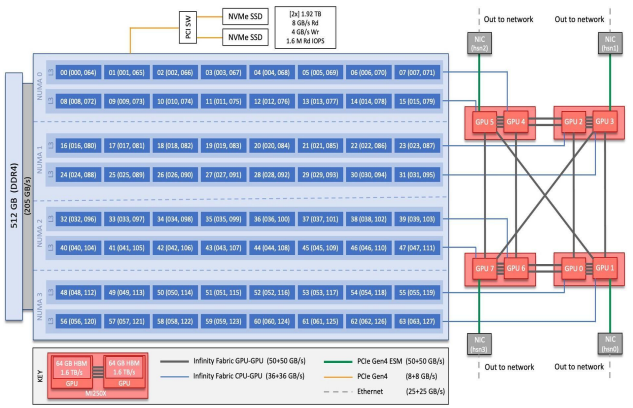
U.S. DEPARTMENT OF ENERGY

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

Motivation: Performance Portability and Programming Productivity

Program Once and Deploy Everywhere!

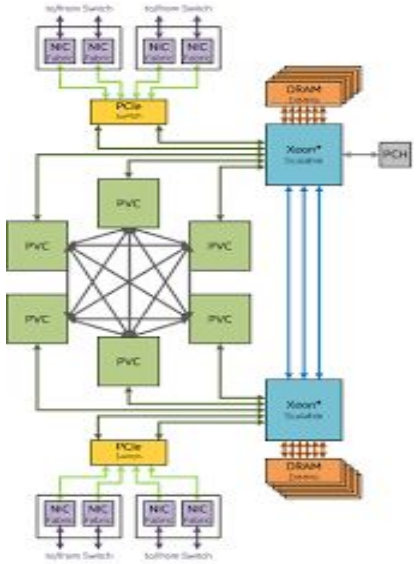
FRONTIER



AMD ROCm

Aurora

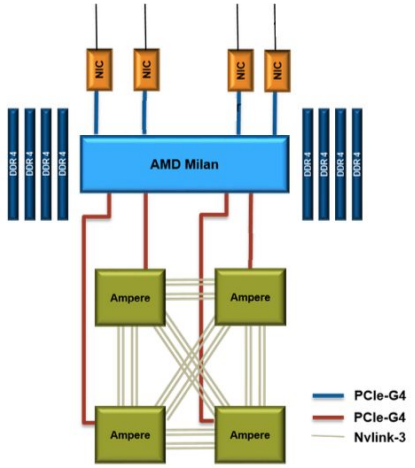
Argonne ENERGY intel CRAY UChicago Argonne



oneAPI

SYCL

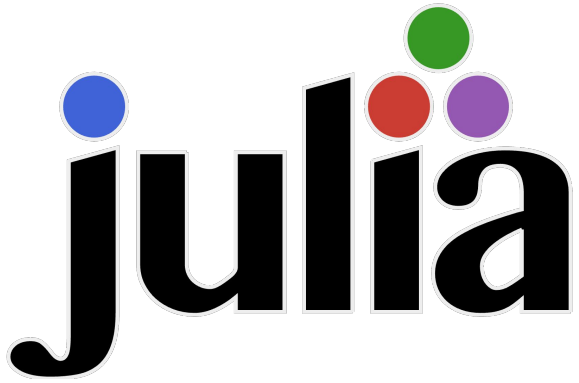
NERSC Perlmutter



NVIDIA CUDA

OpenMP

Why Julia?



Think in Python or Matlab, but now imagine that it works well on HPC

- A JIT language on top of LLVM
- Easy to use and agile interface with C performance
- Julia syntax is optimized for science
- Native syntax (no manual implemented wrappers of wrapper of …) for HPC support
  - Threads, CUDA, AMDGPU, OneAPI, MPI, DAGGER etc
- Integrated and efficient support for packaging, reproducibility, CI/CD, …
- All this makes Julia an ecosystem motivated by performance and productivity

https://julialang.org/

Options for (Data) Parallelism in Julia

CPU multi-threading:

- Threads (built-in)
- Polyester
- OhMyThreads

GPU kernel model (fine-granularity):

- Vendor-specific packages:
  - CUDA, AMDGPU, oneAPI, Metal
- KernelAbstractions
  - Portable interface implemented by each vendor package

Performance-Portable (CPU or GPU with smart defaults):

- JACC

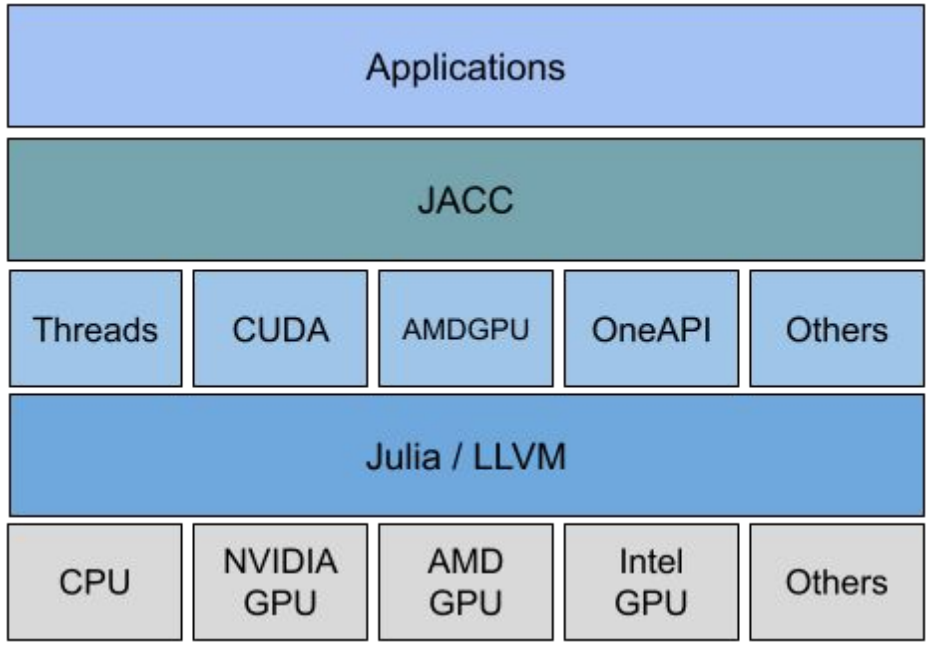
JACC.jl, Julia for ACCelerators

Simple to use performance portable model

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

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

- High-level performance portability model of Julia (`parallel_for` & `parallel_reduce`)
- **JACC is a unified Julia front-end on top of multiple backends**
  - Threads, CUDA, AMDGPU, and oneAPI
- Hide low-level and device specific implementation
  - Memory layout, granularity, etc.
- Improve programming productivity for science and HPC
- A growing community (family)
  - LBNL/NERSC, Argonne, MIT, ETHZ, BSC, Cerfacs, FI/CCQ, …
  - You are welcome to join (JACC community meetings once a month)



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")
```

   - This will install the appropriate backend package as a dependency (don't commit this).
   - In some cases you will need to configure the backend package.

3. Restart julia

4. Load extension before use

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

   This imports the appropriate backend package for the preference set with JACC.set_backend

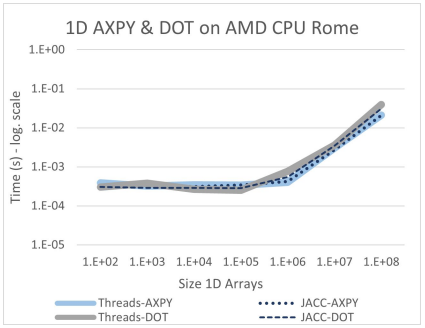
JACC paradigm basics

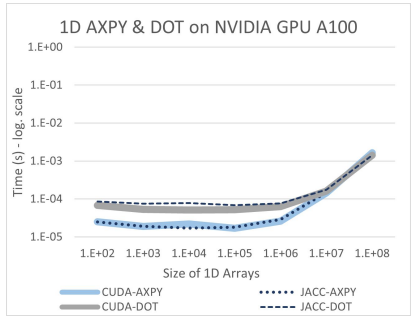
- **JACC.array**
  - Returns backend-managed array
  - No separate array type for JACC
- **JACC.parallel_for and JACC.parallel_reduce**
  - Kernel function passed as an argument
  - Unidimensional and multidimensional APIs
- **JACC.ones or JACC.zeros**

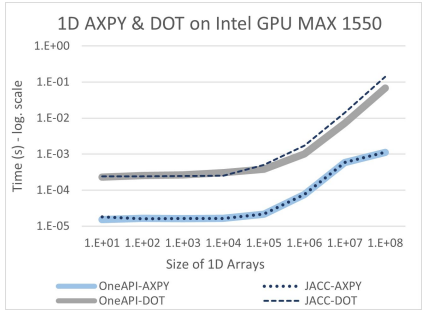
```
using JACC
JACC.@init_backend

# Unidimensional arrays
function axpy(i, alpha, x, y)
    x[i] += alpha * y[i]
end
function dot(i, x, y)
    return x[i] * y[i]
end
SIZE = 1_000_000
x = round.(rand(Float64, SIZE) * 100)
y = round.(rand(Float64, SIZE) * 100)
alpha = 2.5
dx = JACC.array(x)
dy = JACC.array(y)
JACC.parallel_for(SIZE, axpy, alpha, dx, dy)
res = JACC.parallel_reduce(SIZE, dot, dx, dy)

# Multidimensional arrays
function axpy(i, j, alpha, x, y)
    x[i, j] += alpha * y[i, j]
end
function dot(i, j, x, y)
    return x[i, j] * y[i, j]
end
SIZE = 1_000
x = round.(rand(Float64, SIZE, SIZE) * 100)
y = round.(rand(Float64, SIZE, SIZE) * 100)
alpha = 2.5
dx = JACC.array(x)
dy = JACC.array(y)
JACC.parallel_for((SIZE, SIZE), axpy, alpha, dx, dy)
res = JACC.parallel_reduce((SIZE, SIZE), dot, dx, dy)
```

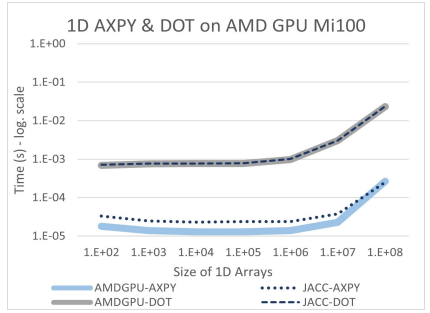
Performance Results

1D AXPY & DOT

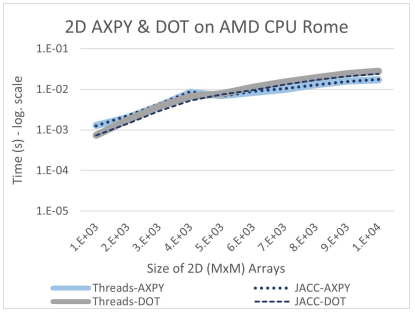


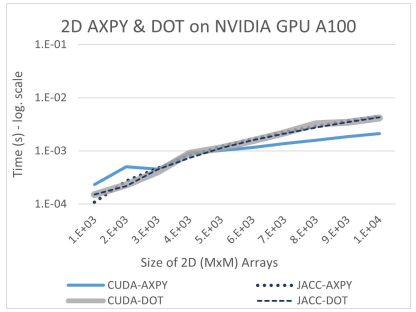


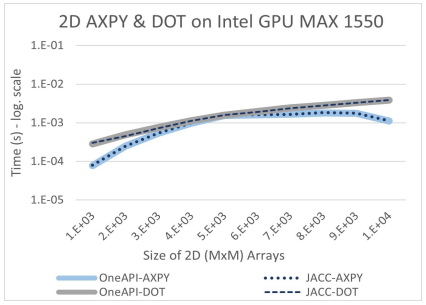


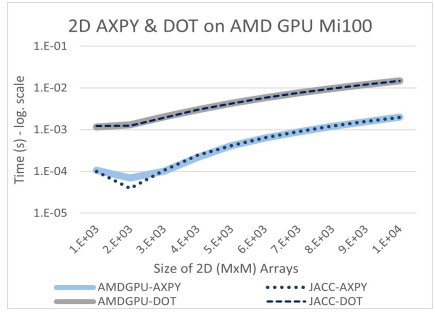


2D AXPY & DOT

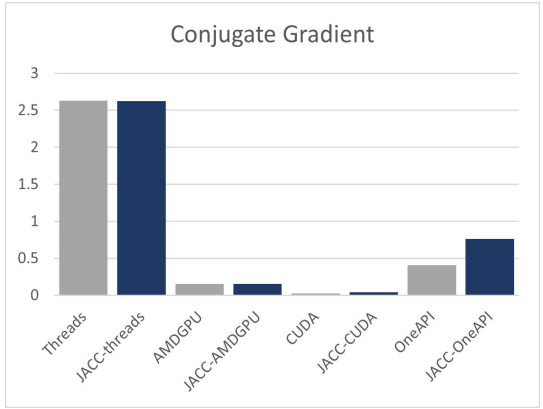




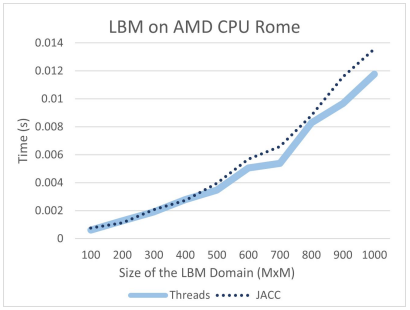


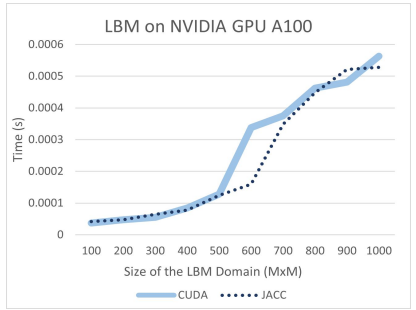


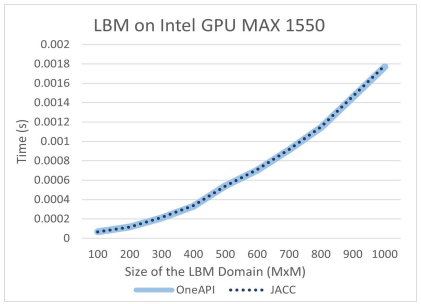
Conjugate Gradient

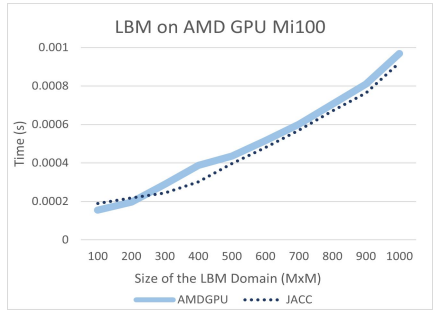


Lattice – Boltzmann Method









JACC paradigm basics - arrays

`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 paradigm basics - parallel_for

JULIA FOR ACCELERATORS

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,
    (i, alpha, x, y) -> begin            #
        @inbounds x[i] += alpha * y[i]   # anonymous function
    end,                                 #
    alpha, x, y)
```

(or, from the previous example)

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

Note that items must be passed explicitly.

JACC paradigm basics - parallel_for

Take a slightly more involved example and use a named tuple for the anonymous function

```
for i = 1:N
    @inbounds begin
        event = events[i, 6:8]
        weight = events[i, 1]
        for op in transforms
            v = op * event
            atomic_push!(hist, v, weight)
        end
    end
end
```

```
JACC.parallel_for(length(events),
    (i, t) -> begin
        @inbounds begin
            event = t.events[i, 6:8]
            weight = t.events[i, 1]
            for op in t.transforms
                v = op * event
                atomic_push!(t.h, v, weight)
            end
        end
    end,
    (h = hist, events, transforms), # named tuple
)
```

JACC paradigm basics - parallel_reduce

```
parallel_reduce(N, op, f, x…; init) -> typeof(init)
parallel_reduce(N, f, x…) = parallel_reduce(N, +, f, x…; init = 0.0)
parallel_reduce([op = +,] a::AbstractArray; init = default_init(eltype(a), op)) -> typeof(init)
    # op ∈ {+, *, min, max, custom}
```

Examples

```
a = JACC.array(randn(Float64, SIZE))
mysum = JACC.parallel_reduce(a)
mymin = JACC.parallel_reduce(min, a)
```

```
maxIx = JACC.parallel_reduce(N, max,
    (i, t) -> begin
        @inbounds return compute_something(
            t.signal,
            t.theta[i],
            t.phi[i],
            t.transform,
        )
    end,
    (
        signal = signal,
        theta,
        phi,
        transform,
    ),
    init = typemin(SizeType)
)
```

Notes for GPU programming

- Any function can be used in a kernel (without special annotation)
  (as long as it doesn't allocate)
- JACC imports the @atomic macro from Atomix.jl
- Use StaticArrays.jl for small fixed-size arrays
- Use Adapt.jl if you need to put GPU arrays in a struct

Simple exercises

1. Allocate an array and use a kernel to initialize the elements (e.g., using the index values)
2. Allocate an array of ones and find the sum of the elements using parallel_reduce.
3. Try #2 using a parallel_for with @atomic

(Bonus) Create a struct holding an array and use it in a kernel. (This is not a challenge for CPU-only code)

Some Implementation Details

- Most JACC functions have backend-specific implementations
- Use tag dispatch
- Backend tags are consistent with KernelAbstractions
- Example (from CUDA backend):
- API function dispatches to default backend

```
function parallel_for(N::Integer, f::Callable, x...)
    return parallel_for(default_backend(), N, f, x...)
end
```

```
function JACC.parallel_for(::CUDABackend, N::Integer, f::Callable, x...)
    kargs = kernel_args(N, f, x...)
    kernel, maxThreads = kernel_maxthreads(_parallel_for_cuda, kargs)
    threads = min(N, maxThreads)
    blocks = ceil(Int, N / threads)
    shmem_size = attribute(
        device(), CUDA.DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK)
    CUDA.@sync kernel(
        kargs...; threads = threads, blocks = blocks, shmem = shmem_size)
end
```

LaunchSpec for parallel_for

- `JACC.parallel_for` defaults:
  - Synchronize
  - Use default stream
  - Compute threads, blocks, and shmem_size
- New version allows fine-grain control
- Specify one or more of the keywords
- Call `parallel_for(spec, N, f, x…)`

```
@kwdef mutable struct LaunchSpec{Backend}
    stream = default_stream(Backend)
    threads = 0
    blocks = 0
    shmem_size::Int = 0
    sync::Bool = false
end

launch_spec(; kw...) = LaunchSpec{typeof(default_backend())}(; kw...)
```

JACC.Multi: Exploiting Multi-GPU Nodes

Most current HPC nodes are multi-GPU

- **JACC.Multi**
  - Deploy JACC's specification to multi-GPU
- **JACC.Multi.array:**

  `JX = JACC.Multi.array(X)`
  - X is evenly distributed into the different GPUs
  - JX is an implementation-specific structure for managing a set of sub-arrays
- **JACC.Multi.parallel_for** and **JACC.Multi.parallel_reduce**
  - Same specification, but launches portion of workload on all devices
- **Manage ghost elements between devices**

More thorough example in tests

```
# Unidimensional arrays
function axpy(i, alpha, x, y)
    x[i] += alpha * y[i]
end
function dot(i, x, y)
    return x[i] * y[i]
end
SIZE = 1_000_000
x = round.(rand(Float64, SIZE) * 100)
y = round.(rand(Float64, SIZE) * 100)
alpha = 2.5
dx = JACC.Multi.array(x)
dy = JACC.Multi.array(y)
JACC.Multi.parallel_for(SIZE, axpy, alpha, dx, dy)
res = JACC.Multi.parallel_reduce(SIZE, dot, dx, dy)

# Multidimensional arrays
function axpy_2d(i, j, alpha, x, y)
    x[i, j] += alpha * y[i, j]
end
function dot_2d(i, j, x, y)
    return x[i, j] * y[i, j]
end
SIZE = 1_000
x = round.(rand(Float64, SIZE, SIZE) * 100)
y = round.(rand(Float64, SIZE, SIZE) * 100)
alpha = 2.5
dx = JACC.Multi.array(x)
dy = JACC.Multi.array(y)
JACC.Multi.parallel_for((SIZE, SIZE), axpy_2d, alpha, dx,
dy)
res = JACC.Multi.parallel_reduce((SIZE, SIZE), dot_2d, dx,
dy)
```

JACC.Multi: Performance Results

Synchronous vs Asynchronous

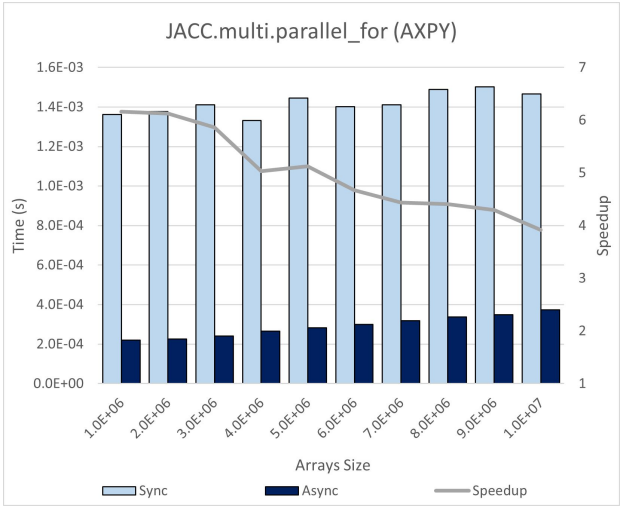
Frontier

4x AMD Mi250x GPUs

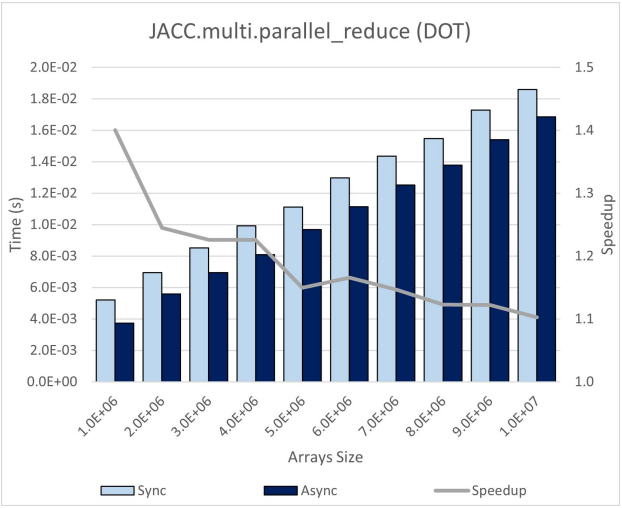
ExCL's Hudson Node

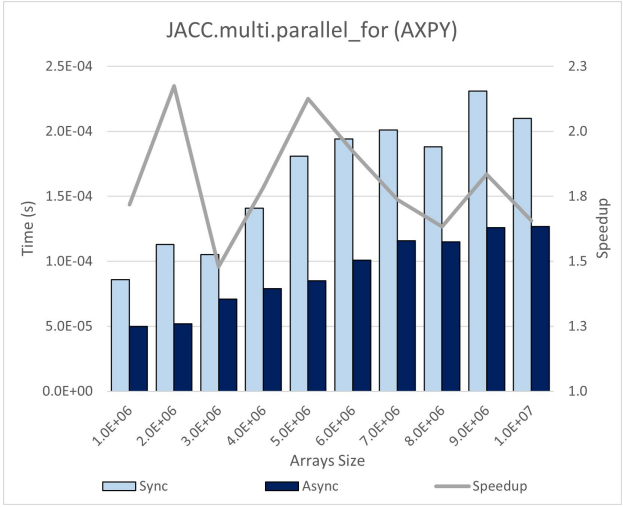
2x NVIDIA Hopper H100 GPUs

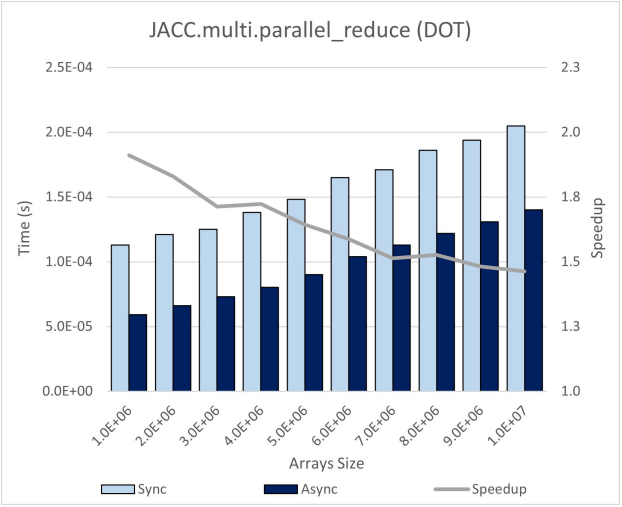
AXPY



DOT







JACC.shared: Exploiting High-Bandwidth on-chip GPUs Memory

https://ieeexplore.ieee.org/document/10938453

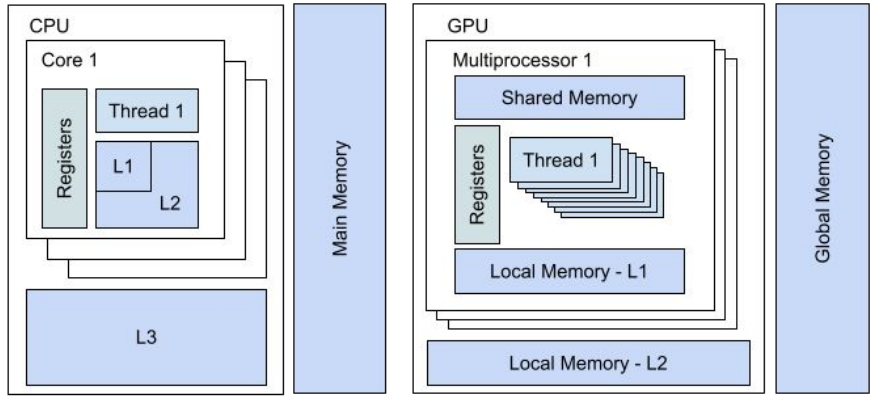
- Shared memory on-chip GPUs memory are **orders of magnitude faster** than global memory

- Shared memory is useful to accelerate computations where multiple threads within the same block must access the same data multiple times

- ***Implementation***

  Y = JACC.shared(X)

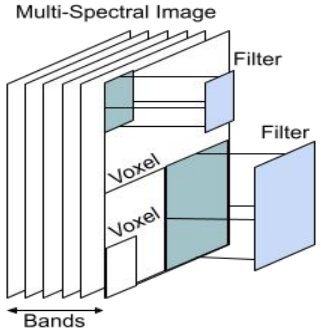
  - X is a JACC.array and Y is a copy of X stored in on-chip shared memory
  - X can be any dimension
  - Y can only be a unidimensional array
  - To be used inside functions
  - The array passed as argument must fit the capacity of the shared memory

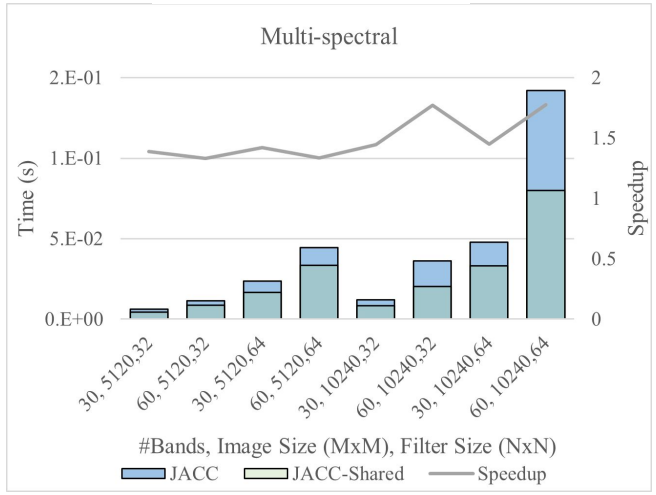


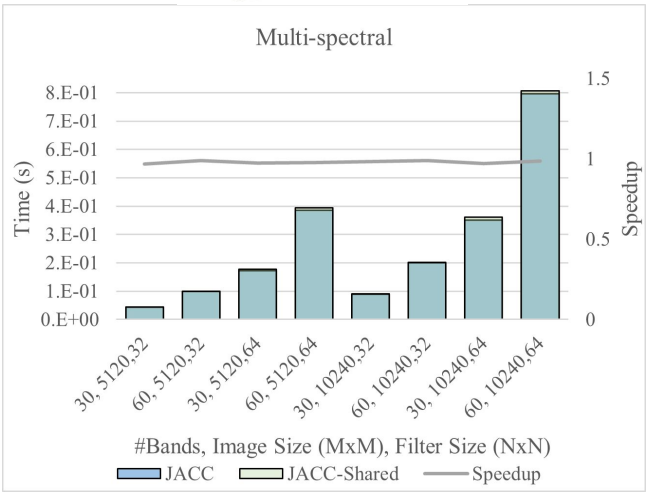

```
function spectral(i, j, image, filter, num_bands)
 for b in 1:bands
    @inbounds image[b, i, j] *= filter[j]
  end
end

function spectral_shared(i, j, image, filter,
    num_bands)
 #Shared memory initialization
 filter_shared = JACC.shared(filter)
 for b in 1:bands
  @inbounds image[b, i, j] *= filter_shared[j]
 end
end

num_bands = 60
num_voxel = 10_240
size_voxel = 64*64
image = init_image(Float32,
        num_bands, num_voxel, size_voxel)
filter = init_filter(Float32, size_voxel)
jimage = JACC.Array(image)
jimage_shared = JACC.Array(image)
jfilter = JACC.Array(filter)
JACC.parallel_for((num_voxel,size_voxel),
     spectral[_shared], jimage, jfilter, num_bands)
```

JACC.Shared: Performance Results

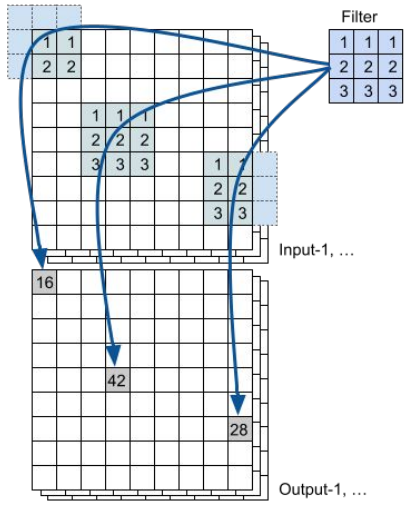
Multi-Spectral Imaging

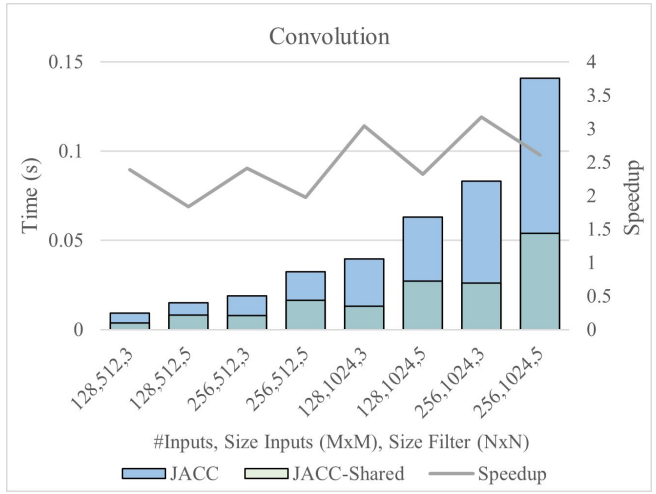


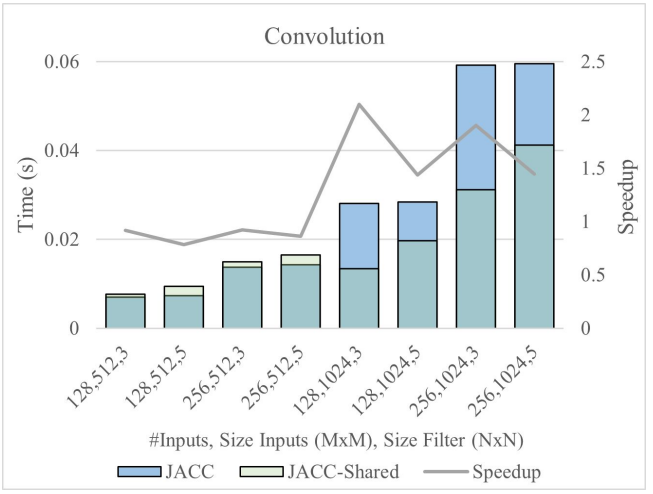




AI Convolutions







Applications & Future Work

- Ongoing Efforts
  - Performance benchmarking (closing gaps)
  - More intuitive kernel launch
  - JACC.Async : Concurrency on multi-device nodes
  - 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?