HPC-Friendly Workflows in Julia

“Julia + Jupyter + GPU = 🧪🔬🧬🥰”

(phrase borrowed from Marius Millea)

Johannes Blaschke
Data Science Engagement Group
NERSC, LBNL
Credit and Disclaimers

None of this would be possible without:

- Tim Bersard, Valentin Churavy, Julian Samaroo (MIT Julia Lab) + Anton Smirnov (AMD) + Carsten Bauer (NHR, PC2)
  - Providing the Infrastructure
- Marius Millea (UC Davis) + Mark Hirsbrunner (LBNL*) + William Godoy, Pedro Valero Lara (OLCF)
  - Inspiring applications
- The Julia for HPC working group
  - [https://github.com/JuliaParallel](https://github.com/JuliaParallel)
  - Meets monthly on Zoom (cf. [https://julialang.org/community/](https://julialang.org/community/) ) and is very active on Discord
  - Julia for HPC BoF at SC and JuliaCon
- Soham Ghosh (NERSC)
  - Exploring AI applications: just-in-time AI, AI for science, UQ

Disclaimer:

- I work on Perlmutter, but can be (and has been) easily applied to HPC more broadly.
Overview

1. Julia in 60s
2. Julia + Jupyter as an interactive workflow engine
3. Network Discovery
4. Programming GPUs
5. Inspiration: Particles in Potentials
6. Machine Learning using Flux.jl
7. Using Dagger.jl to parallelize your workflow
Julia in 60s
Julia is a High-Productivity Language

- It has all the modern HP features (rich stdlib, gc, …)
Julia is a High-Productivity Language

- It has all the modern HP features (rich stdlib, gc, …)
- + a powerful REPL
Julia is a High-Productivity Language

- It has all the modern HP features (rich stdlib, gc, …)
- + a powerful REPL
- + a comprehensive package manager (which integrates with system software)

Project.toml:

```toml
[extras]
MPIPreferences = "3da0fdf6-3ccc-4f1b-acd9-58bbaa6c99267"
CUDA_Runtime_jll = "76a88914-d11a-5bdc-97e0-2f5a05c973a2"
```

LocalPreferences.toml:

```toml
[MPIPreferences]
  _format = "1.1"
  abi = "MPICH"
  binary = "system"
  cclibs = ["cupti", "cudart", "cuda", "sci_gnu_82_mpi", "sci_gnu_82", "dl", "dsmml", "xpmem"]
  libmpi = "libmpi_gnu_91.so"
  mpiexec = "srun"
  preloads = ["libmpi_gtl_cuda.so"]
  preloads_env_switch = "MPICH_GPU_SUPPORT_ENABLED"

[CUDA_Runtime_jll]
  local = "true"
  version = "11.7"
```
Julia has LLVM under the Hood

```julia
1: f() = @call clock::Cint
[10]: f (generic function with 1 method)
[11]: f()
[11]: 8598048
[12]: @code_llvm(f())
    ; @ In[10]:1 within `f`
    define i32 @julia_f_1782() #0 {
    top:
      %0 = call i32 inttoptr (i64 139954008063022 to i32 )() ()
    ret i32 %0

[ ]: 1
```
Julia has LLVM under the Hood

Julia data types are binary-compatible with C

`@ccall` equivalent to C function call
Julia has LLVM under the Hood

@code_llvm exposes the LLVM IR for debug purposes
Julia has a Powerful Type System
Julia has a Powerful Type System

Structured data types are also compatible with C

{T<:Number} represents a type template for all types inheriting from Number
Julia has a Powerful Type System

{T<:Number} represents a type template for all types inheriting from Number.
Julia has a Powerful Type System

Julia has multiple dispatch: a function can have several implementations (methods) depending on the input types.
Julia + Jupyter as an interactive workflow engine
DOE SC User Requirements Are Evolving

Users require support for

- End-to-end DOE SC Workflows involving multiple facilities
- New modes of scientific discovery through the integration of simulation & modeling, AI and experiment.
- Interactive/Real-Time Workflows
Why NERSC Cares about Julia Workflows

<table>
<thead>
<tr>
<th>Function signature</th>
<th>Pybind11</th>
<th>ccall</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fn0()</td>
<td>132</td>
<td>±14.9</td>
<td>2.34</td>
</tr>
<tr>
<td>int fn1(int)</td>
<td>217</td>
<td>±20.9</td>
<td>2.35</td>
</tr>
<tr>
<td>double fn2(int, double)</td>
<td>232</td>
<td>±11.7</td>
<td>2.32</td>
</tr>
<tr>
<td>char* fn3(int, double, char*)</td>
<td>267</td>
<td>±28.9</td>
<td>6.27</td>
</tr>
</tbody>
</table>

pip install ...
Rapid scripting

C/C++
Python
Julia
Workflows
Building a Distributed Julia Application (without MPI)
Building a Distributed Julia Application (without MPI)

User WF

WF node

High-speed network

User SW  User SW  User SW  User SW

Compute 1  Compute 2  Compute 3  Compute 4
Building a Distributed Julia Application (without MPI)

<table>
<thead>
<tr>
<th>User WF</th>
<th>Jupyter</th>
<th>WF node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dagger.jl</td>
<td>Distributed.jl</td>
<td>High-speed network</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User SW</th>
<th>User SW</th>
<th>User SW</th>
<th>User SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA.jl</td>
<td>CUDA.jl</td>
<td>CUDA.jl</td>
<td>CUDA.jl</td>
</tr>
<tr>
<td>Vendor SW</td>
<td>Vendor SW</td>
<td>Vendor SW</td>
<td>Vendor SW</td>
</tr>
<tr>
<td>Compute 1</td>
<td>Compute 2</td>
<td>Compute 3</td>
<td>Compute 4</td>
</tr>
</tbody>
</table>
Building a Distributed Julia Application (without MPI)

- User WF
  - Jupyter
  - WF node

- Dagger.jl
- Distributed.jl
- High-speed network

- User SW
  - CUDA.jl
  - Vendor SW
  - Compute 1
- User SW
  - CUDA.jl
  - Vendor SW
  - Compute 2
- User SW
  - CUDA.jl
  - Vendor SW
  - Compute 3
- User SW
  - CUDA.jl
  - Vendor SW
  - Compute 4

Possibly also login node, or head node

JACC.jl and KernelAbstractions.jl provide portability

- Possibly also login node, or head node
- JACC.jl and KernelAbstractions.jl provide portability
Cluster Managers: Interaction with Slurm

```python
using Distributed, ClusterManagers

addprocs(SlurmManager(2), N="2", constrain="cpu", qos="debug", time="00:5:00")

connecting to worker 1 out of 2
connecting to worker 2 out of 2
srun: job 18107782 queued and waiting for resources
srun: job 18107782 has been allocated resources

2-element Vector{Int64}:
  2
  3

hosts = []
pids = []
for i in workers()
    host, pid = fetch(@spawn @i (gethostname(), getpid()))
push!(hosts, host)
push!(pids, pid)
end

hosts

2-element Vector{Any}:
  "nid004709"
  "nid004953"
pids

2-element Vector{Any}:
  1505084
  1781811
```
Cluster Managers: Interaction with Slurm

Request two cpu nodes

ElasticManager can be used instead of SlurmManager to manually manage workers from within an allocation
Cluster Managers: Interaction with Slurm

Get hostnames and pids from each worker

```python
using Distributed, ClusterManagers

addprocs(SlurmManager(2), N="2", constrain="cpu", qos="debug", time="00:5:00")
    connecting to worker 1 out of 2
    connecting to worker 2 out of 2

hosts = []
pids = []
for i in workers()
    host, pid = fetch(@spawnat i (gethostname(), getpid()))
    push!(hosts, host)
    push!(pids, pid)
end
```

2-element Vector{Any}:
"nid004709"
"nid004953"

2-element Vector{Any}:
1505084
1781811
Cluster Managers: Interaction with Slurm
Distributed.jl supports basic workflows

Basic task: count number of heads from 2x10^8 fair coin tosses.

- Serial implementation: for loop + increment counter
- Parallel (distributed) implementation: @distributed for loop + reduction (summation) on counter

- Most Julia data types are trivial to serialize and communicate over network (the users doesn’t have to do anything “special” to enable this)

```julia
1 function n_heads()
2     n = 0
3     for i = 1:200000000
4         n += Int(rand(Bool))
5     end
6     n
7 end

8 function n_heads_parallel()
9     nheads = @distributed (+) for i = 1:200000000
10        Int(rand(Bool))
11     end
12     nheads
13 end
```
Performance Gains from Distributing Work
Performance Gains from Distributing Work

```python
[31]: 1 @benchmark(n_heads())

[31]: BenchmarkTools.Trial: 21 samples with 1 evaluation.
    Range (min ... max): 245.963 ms ... 246.808 ms | GC (min ... max): 0.00% ... 0.00%
    Time (median): 245.978 ms            | GC (median): 0.00%
    Time (mean ± σ): 246.023 ms ± 181.318 µs | GC (mean ± σ): 0.00% ± 0.00%

Memory estimate: 0 bytes, allocs estimate: 0.
```
Performance Gains from Distributing Work

Distributing work over 2 nodes results in a 2x performance increase.

```
[32]: @benchmark(n_heads_parallel())

BenchmarkTools.Trial: 51 samples with 1 evaluation.
  Range (min ... max):  99.827 ms ... 100.605 ms  |  GC (min ... max): 0.00% ... 0.00%
  Time (median):       99.855 ms
  Time (mean ± σ):     99.871 ms ± 106.362 μs  |  GC (median):         0.00%
                                                                       GC (mean ± σ): 0.00% ± 0.00%

99.8 ms  Histogram: frequency by time  99.9 ms <

```
Tangent: Hybrid CPU/GPU Jobs
Tangent: Hybrid CPU/GPU Jobs
Tangent: Hybrid CPU/GPU Jobs

```python
using Distributed, ClusterManagers

addprocs(2, N="2", constrain="cpu", qos="interactive", time="00:15:00", A="nstaff")
    connecting to worker 1 out of 2
    srn: job 18134163 queued and waiting for resources

srn: job 18134168 has been allocated resources
    connecting to worker 2 out of 2

2-element Vector{Int64}:
4
5

features = []
for i in workers():
    scontrol = fetch(@spawnat i get_features())
    push!(features, scontrol)
end

features

4-element Vector{Any}:
"ActiveFeatures=cpu,milan,ss11"
"ActiveFeatures=cpu,milan,ss11"
"ActiveFeatures=gpu,ss11,a100,hbm40g"
"ActiveFeatures=gpu,ss11,a100,hbm40g"
```
Tangent: Hybrid CPU/GPU Jobs

```python
[48]: features = []
    for i in workers():
        scontrol = fetch('spawnat', i, get_features())
        push!(features, scontrol)
```

```
[49]: 4-element Vector(Any):  
    "ActiveFeatures=cpu,milan,ss11"  
    "ActiveFeatures=cpu,milan,ss11"  
    "ActiveFeatures=cpu,milan,ss11"  
    "ActiveFeatures=cpu,milan,ss11"
```
Tangent: Hybrid CPU/GPU Jobs

[2]: using Distributed, ClusterManagers

[3]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", A='nstaff')

[4]: addprocs(SlurmManager(2), N="2", constrain="gpu", qos="interactive", time="00:15:00", A='nstaff_g')

[49]: features

[49]: 4-element Vector{Any}:
"ActiveFeatures=cpu,milan,ss11"
"ActiveFeatures=cpu,milan,ss11"
"ActiveFeatures=gpu,ss11,a100,hbm40g"
"ActiveFeatures=gpu,ss11,a100,hbm40g"
Tangent: Network Discovery
Workflow Support Story

- Unexpected poor performance and scaling
- User application 100x slower on Perlmutter
Perlmutter is a Heterogeneous System

<table>
<thead>
<tr>
<th>Partition</th>
<th>Nodes</th>
<th>CPU</th>
<th>RAM</th>
<th>GPU</th>
<th>NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>1536</td>
<td>1x AMD EPYC 7763</td>
<td>256GB</td>
<td>4x NVIDIA A100 (40GB)</td>
<td>4x HPE Slingshot 11</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1x AMD EPYC 7763</td>
<td>256GB</td>
<td>4x NVIDIA A100 (80GB)</td>
<td>4x HPE Slingshot 11</td>
</tr>
<tr>
<td>CPU</td>
<td>3072</td>
<td>2x AMD EPYC 7763</td>
<td>512GB</td>
<td>–</td>
<td>1x HPE Slingshot 11</td>
</tr>
<tr>
<td>Login</td>
<td>40</td>
<td>1x AMD EPYC 7713</td>
<td>512GB</td>
<td>4x NVIDIA A100 (40GB)</td>
<td>–</td>
</tr>
<tr>
<td>Large Memory</td>
<td>4</td>
<td>1x AMD EPYC 7713</td>
<td>1TB</td>
<td>4x NVIDIA A100 (40GB)</td>
<td>1x HPE Slingshot 11</td>
</tr>
</tbody>
</table>

- Each GPU node has 4 NICs
  - 1 NIC and 1 GPU per host bridge
- Each CPU node has 1 NIC
Eg. GPU Node Topology

```
Hwloc.Object: Machine
  └── Hwloc.Object: Package [L#0 P#0]
    │   └── Hwloc.Object: Machine
    │      └── Hwloc.Object: Package [L#0 P#0]
    │         └── Hwloc.Object: Machine
    │            └── Hwloc.Object: Package [L#0 P#0]
    │                └── Hwloc.Object: Machine
    │                   └── Hwloc.Object: Package [L#0 P#0]
    │                      └── Hwloc.Object: Machine
    │                           └── Hwloc.Object: Package [L#0 P#0]
    │                                └── Hwloc.Object: Machine
    │                                   └── Hwloc.Object: Package [L#0 P#0]
    │                                        └── Hwloc.Object: Machine
    │                                             └── Hwloc.Object: Package [L#0 P#0]
    │                                                └── Hwloc.Object: Machine
    │                                                   └── Hwloc.Object: Package [L#0 P#0]
    │                                                        └── Hwloc.Object: Machine
    │                                                            └── Hwloc.Object: Package [L#0 P#0]
```

...
Eg. GPU Node Topology

Hwloc.Object: Machine
  └─ Hwloc.Object: Package [L#0 P#0]
    ├─ Hwloc.Object: Group
    │   └─ Hwloc.Object: NUMANode
    │       └─ Hwloc.Object: Bridge [HostBridge]
    │       │   └─ Hwloc.Object: Bridge [PCIBridge]
    │       │       └─ Hwloc.Object: PCI_Device [c2:00.0 (Ethernet)]
    │       │       └─ Hwloc.Object: OS_Device [Net "hsn0"]
    │       │       └─ Hwloc.Object: Bridge [PCIBridge]
    │       │       └─ Hwloc.Object: PCI_Device [c3:00.0 (Ethernet)]
    │       │       └─ Hwloc.Object: OS_Device [Net "nmn0"]
    ├── Hwloc.Object: Group
    │   └─ Hwloc.Object: NUMANode
    │       └─ Hwloc.Object: Bridge [HostBridge]
    │       │   └─ Hwloc.Object: Bridge [PCIBridge]
    │       │       └─ Hwloc.Object: PCI_Device [81:00.0 (Ethernet)]
    │       │       └─ Hwloc.Object: OS_Device [Net "hsn1"]
    ├── Hwloc.Object: Group
    │   └─ Hwloc.Object: NUMANode
    │       └─ Hwloc.Object: Bridge [HostBridge]
    │       │   └─ Hwloc.Object: Bridge [PCIBridge]
    │       │       └─ Hwloc.Object: PCI_Device [42:00.0 (Ethernet)]
    │       │       └─ Hwloc.Object: OS_Device [Net "hsn2"]
    │   └─ Hwloc.Object: Group
    │       └─ Hwloc.Object: NUMANode
    │           └─ Hwloc.Object: Bridge [HostBridge]
    │           │   └─ Hwloc.Object: Bridge [PCIBridge]
    │           │       └─ Hwloc.Object: PCI_Device [01:00.0 (Ethernet)]
    │           │       └─ Hwloc.Object: OS_Device [Net "hsn3"]
    ├── Hwloc.Object: Group
    │   └─ Hwloc.Object: NUMANode
    │       └─ Hwloc.Object: Bridge [HostBridge]
    │       │   └─ Hwloc.Object: Bridge [PCIBridge]
    │       │       └─ Hwloc.Object: PCI_Device [01:00.0 (Ethernet)]
    │       │       └─ Hwloc.Object: OS_Device [Net "hsn3"]
    └─ Hwloc.Object: Group
      └─ Hwloc.Object: NUMANode
        └─ Hwloc.Object: Bridge [HostBridge]
          └─ Hwloc.Object: Bridge [PCIBridge]
Finding the right NIC is easy now: pick the (non-nmn) interface with lowest tree distance between your core and the PCI device

```python
for net in collect(network_devs):
    found, dist = distance_to_core(hwloc_tree, net, cpu_id)
    print("\$(dist): ")
    print_tree(net)
end
```
Topo distance to NIC

- Finding the right NIC is easy now: pick the (non-nmn) interface with lowest tree distance between your core and the PCI device.

```python
for net in collect(network_devs):
    found, dist = distance_to_core(hwloc_tree, net, cpu_id)
    print("$(dist): ")
    print_tree(net)
end
```

This one!
Future

• This is pre-alpha, so far only deployed at NERSC
  o Looking for folks to test this at NERSC and on their favorite HPC systems

• Distributed.jl to use distance between NIC and Core (on Hwloc tree) to select preferred tree NIC
  o JuliaParallel/NetworkInterfaceControllers.jl
Programming GPUs
CUDA.jl: Interfacing with Nvidia GPUs

```julia
44

```
CUDA.jl: Interfacing with Nvidia GPUs

Basic example: 1000x1000 matmul using OpenBLAS

```julia
using OpenBLAS

arr1 = rand(1_000, 1_000);
arr2 = rand(1_000, 1_000);

@benchmark arr1 * arr2

 BenchmarkTools.Trial: 1493 samples with 1 evaluation.
 Range (min ... max):  2.626 ms ...  7.337 ms  GC (min ... max):  0.00% ...  2.79%
 Time  (median):       3.152 ms       GC (median):     0.00%
 Time  (mean ± σ):    3.347 ms ±  490.147 μs  GC (mean ± σ):  0.90% ±  1.83%

Memory estimate: 7.63 MiB, allocs estimate: 2.
```

```julia
@benchmark cu(arr1) * cu(arr2)

 BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min ... max):  22.533 μs ...  390.183 μs  GC (min ... max):  0.00% ...  0.00%
 Time  (median):       121.285 μs       GC (median):     0.00%
 Time  (mean ± σ):    114.948 μs ±  24.119 μs  GC (mean ± σ):  0.00% ±  0.00%

Memory estimate: 1.09 KiB, allocs estimate: 44.
```
CUDA.jl: Interfacing with Nvidia GPUs

Copy arrays to device

[11]: using CUDA

[12]: carr1 = cu(arr1);
   carr2 = cu(arr2);
CUDA.jl: Interfacing with Nvidia GPUs

Using cuBLAS to perform matmul decreases run time from 3.15ms to 121μs (26x)
Write Your Own CUDA Kernels in Julia

```
function my_kernel(carr_out, carr)
    start = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    stride = blockDim().x * gridDim().x
    len = length(carr)
    for i = start:stride:len  # "grid-stride" loop
        carr_out[i] = sin(carr[i]) + 1
    end
    return
end

my_kernel (generic function with 1 method)
```

carr = cu(rand(10_000_000))
carr_out = similar(carr);
@cuda threads=256 my_kernel(carr_out, carr)

CUDA.HostKernel for my_kernel(CuDeviceVector{Float32, 1}, CuDeviceVector{Float32, 1})

```
carr_out
```

```
10000000-element CuArray(Float32, 1, CUDA.Mem.DeviceBuffer):
1.1730438
1.8287812
1.5149846
1.3774228
1.606185
1.431187
1.4560564
1.1032873
```
Define kernels using Julia functions:

```julia
function my_kernel(carr_out, carr)
    start = (blockIdx(.x - 1) * blockDim().x + threadIdx().x
    stride = blockDim().x * gridDim().x
    len = length(carr)
    for i = start:stride:len  # "grid-stride" loop
        carr_out[i] = sin(carr[i]) + 1
    end
    return
end
```

Example usage:

```julia
# Generate random input array

carr = cu(rand(10_000_000))
carr_out = similar(carr);

@cuda threads=256 my_kernel(carr_out, carr)
```

This creates a CUDA Host Kernel for `my_kernel` with the given parameters.
Write Your Own CUDA Kernels in Julia

Launch kernel using the @cuda macro:

```julia
function my_kernel(carr_out, carr)
    start = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    stride = blockDim().x * gridDim().x
    len = length(carr)
    for i = start:stride:len # "grid-stride" loop
        carr_out[i] = sin(carr[i]) + 1
    end
    return
end
```

```julia
my_kernel (generic function with 1 method)
```

```julia
[carr = cu(rand(10_000_000))]
carr_out = similar(carr);
@cuda threads=256 my_kernel(carr_out, carr)
```

CUDA.HostKernel for my_kernel(CuDeviceVector{Float32, 1}, CuDeviceVector{Float32, 1})
CUDA.jl provides detailed profiling interface.
(advanced) LLVM + Julia

Julia provides interfaces to the LLVM backend.

Eg.:

- `loopinfo`
- `llvmcall`
Inspiration: Particles in Potentials
Transition Rates between Potential Minima

```
[1]: using Plots

[4]: energy(x; a=2, k=1) = 0.25 * k * ((x-a)^2) * ((x+a)^2)
   force(x; a=2, k=1) = -k * x * (x-a) * (x+a)

[4]: force (generic function with 1 method)

[9]: x = range(-3, 3, 100)
   e = energy(x)
   plot(x, e, legend=false)
```
Transition Rates between Potential Minima

This is a highly-simplified (1D) model for loads of interesting science: chemical reactions; protein conformations; etc.
Integrating SDEs

```julia
using Random, Distributions
Random.seed!(123)
TaskLocalRNG()

#this is step A
position_update(x, v, dt) = @. x + v * dt / 2.
#this is step B
velocity_update(v, F, dt) = @. v + F * dt / 2.

function random_velocity_update(v::AbstractVector, gamma::Number, kBT::Number, dt::Number; d=Normal(0, 1))
    R = rand(d, length(v))
    c1 = @. exp(-gamma * dt)
    c2 = @. sqrt(1-c1) = sqrt(kBT)
    v_new = @. c1 * v + c2
    return v_new
end

random_velocity_update(v::Number, gamma::Number, kBT::Number, dt::Number; d=Normal(0, 1)) = only(random_velocity_update([v], gamma, kBT, dt; d))

function baob(energy::Function, force::Function, max_time, dt,
              gamma, kBT, initial_position::T, initial_velocity::T;
              save_frequency = 3, kw...)
    where T <: Number

    x = initial_position
    v = initial_velocity
    t = 0
    while t < max_time
        update_position!
        update_velocity!
        # save...!
        t += dt
    end

    return x, v, t
end
```
Integrating SDEs

Julia makes RNG easy!

```julia
using Random, Distributions

Random.seed!(123)
TaskLocalRNG()
```

```julia
function random_velocity_update(v::AbstractVector, gamma::Number, kBT::Number, dt::Number; d=Normal(0, 1))
    R = rand(d, length(v))
    c1 = .* exp(-gamma*dt)
    c2 = .* sqrt(1-c1*c1)*sqrt(kBT)
    v_new = .* c1*v + c2
    return v_new
end

random_velocity_update(v::Number, gamma::Number, kBT::Number, dt::Number; d=Normal(0, 1)) = only(random_velocity_update([v], gamma, kBT, dt; d))

function baob(
    energy::Function, force::Function, max_time, dt,
    gamma, kBT, initial_position::T, initial_velocity::T;
    save_frequency = 3, kwargs...)
    where T <: Number
    x = initial_position
    v = initial_velocity
    t = 0
    # other code...
end
```
Integrating SDEs

Define SDE algorithm

```julia
[10]: ```
```julia
using Random, Distributions
```
```
```
```

```julia
[11]: ```
```julia
Random.seed!(123)
```
```
```
```

```julia
[11]: ```
```julia
TaskLocalRNG()
```
```
```
```

```julia
[12]: ```
```julia
#this is step A
position_update(x, v, dt) = @. x + v*dt/2.

#this is step B
velocity_update(v, F, dt) = @. v + F*dt/2.

function random_velocity_update(
    v::AbstractVector, gamma::Number, kBT::Number, dt::Number;
    d=Normal(0, 1)
)
    R = rand(d, length(v))
    c1 = @. exp(-gamma*dt)
    c2 = @. sqrt(1-c1*c1)*sqrt(kBT)
    v_new = @. c1*v + R*c2
    return v_new
end

random_velocity_update(
    v::Number, gamma::Number, kBT::Number, dt::Number; d=Normal(0, 1)
) = only(random_velocity_update([v], gamma, kBT, dt; d))
```
Integrating SDEs

SDE algorithms can be long and complex!
Transitions are Rare!
Transitions are Rare!

Often you will 100s of millions of data points in order to collect a few thousand transitions.
Solution Strategy: Local Monte-Carlo Sampling
Solution Strategy: Local Monte-Carlo Sampling

```julia
# Define the variables
times, positions, velocities, total_energies = baoab(
    energy, force, max_time, dt,
    gamma, kBT, initial_position, initial_velocity;
    save_frequency=save_frequency, kwars...
)

ensemble_t = Array[T, 1][]
ensemble_p = Array[T, 1][]
ensemble_v = Array[T, 1][]

for (t, p, v) in zip(
    times[1:ensemble.st:end],
    positions[1:ensemble.st:end],
    velocities[1:ensemble.st:end]
)
    p_r = rand(ensemble.d, ensemble.n)
    v_r = rand(ensemble.d, ensemble.n)

    for i in 1:ensemble.n
        t_i, p_i, v_i, _ = baoab(
            energy, force, ensemble.len*dt, dt,
            gamma, kBT, p + p_r[i], v + v_r[i];
            save_frequency=save_frequency, kwars...
        )
        push!(ensemble_t, t + t_i)
        push!(ensemble_p, p_i)
        push!(ensemble_v, v_i)
    end
end
```
Solution Strategy: Local Monte-Carlo Sampling

For every time step, run a short simulation with slightly different starting conditions.
Solution Strategy: Local Monte-Carlo Sampling

Borderline state

Stable state
Solution Strategy: Adaptive Algorithms
Solution Strategy: Adaptive Algorithms
Start Mapping a Phase Space

```julia
scatter(positions, velocities, markershape=:circ, zcolor=ensemble_hi-ensemble_lo)
xlabel!("position")
ylabel!("velocity")
```

```julia
using Flux, CUDA, Statistics, ProgressMeter
CUDA.versioninfo()
```

CUDA runtime 12.2, local installation
CUDA driver 12.3
NVIDIA driver 525.185.17, originally for CUDA 12.2
CUDA libraries:
- CUBLAS: 12.2.1
- CUFFT: 10.3
Start Mapping a Phase Space

```python
scatter(positions, velocities, markershape=:circ, zcolor=ensemble_hi-ensemble_lo)
xlabel("position")
ylabel("velocity")
```
Machine learning using *Flux.jl*
Flux.jl Automatically Detects CUDA.jl
Flux.jl Automatically Detects CUDA.jl

Julia introspection is a powerful tool to detect / confirm system configuration.

At NERSC Julia is configured to automatically detect the system’s CUDA runtime.
Define Input Data

```python
# Phase-space coordinate
pcord = hcat(positions, velocities) |> transpose

# Phase-space class
lrate = ensemble_hi - ensemble_lo  # Lyapunov rate
pclass = Array(Int64, lrate)...
fill!(pclass, 0)
pclass[lrate .> 2] .= 1
pclass[lrate .> 3] .= 2;

# N_coord = 2
N_class = 3
N_neuron = 8

model = Chain(  
    Dense(N_coord => N_neuron, tanh),  
    BatchNorm(N_neuron),  
    Dense(N_neuron => N_class),  
    softmax  
)  
    |> gpu  # move model to GPU, if available

# Total: 6 trainable arrays, 67 parameters,  
# plus 2 non-trainable, 16 parameters, sum: ~1.266 KiB.

# The model encapsulates parameters, randomly initialised. Its initial output is:
out1 = model(pcord |> gpu) |> cpu  # 2*N Matrix(Float32)

3x833 Matrix(Float32):
0.38174 0.29260 0.280616 0.283165 0.401954 0.436592 0.44925 0.291435 0.206022 0.244021 0.235257 0.362114 0.384997 0.391649 0.486825 0.446776 0.469063 0.481578 0.235932 0.17841 0.159102

# To train the model, we use batches of 128 samples, and one-hot encoding:
target = Flux.onelhotbatch(pclass, [0, 1, 2])  
# 2*N OneHotMatrix
loader = Flux.BatchLoader(pcord, target) |> gpu  
batchsize=128 (shuffle=true);
```
Define Input Data

Classify point in phase space based on local MC sample's spread rate (~Lyapunov rate)
Define a (simple) Neural Network Model

Chain conveniently chains together layers. Ingests a 2D (position, velocity) vector, outputs 3D class probability vector

```
N_coord = 2
N_class = 3
N_neuron = 8

model = Chain(
    Dense(N_coord => N_neuron, tanh),
    BatchNorm(N_neuron),
    Dense(N_neuron => N_class),
    softmax
) |> gpu  # move model to GPU, if available
```

```
Chain(
    Dense(2 => 8, tanh),
    BatchNorm(8),
    Dense(8 => 3),
    NNlib.softmax,
)  # Total: 6 trainable arrays, 67 parameters,
    # plus 2 non-trainable, 16 parameters, summarysize 1.266 KiB.
```
Use \( \rightarrow \) gpu and \( \rightarrow \) cpu to Move Data

Column-major inputs. Model outputs the likelihoods (columns) of each class.
Train the Model

```julia
# To train the model, we use batches of 128 samples, and one-hot encoding:
target = Flux.onehotbatch(pclass, [0, 1, 2])  # 2*N OneHotMatrix
loader = Flux.DataLoader(pcoord, target) |> gpu, batchsize=128, shuffle=true;

optim = Flux.setup(Flux.Adam(1e-2), model)  # will store optimiser momentum, etc.

losses = []
@showprogress for epoch in 1:1_000
    for (x, y) in loader
        loss, grads = Flux.withgradient(model) do m
            # Evaluate model and loss inside gradient context:
            y_hat = m(x)
            Flux.crossentropy(y_hat, y)
        end
        Flux.update!(optim, model, grads[1])
    end
end

Progress: 100%   Time: 0:00:45

plot(losses, xaxis=:log, yaxis=:log)
```
Train the Model

> gpu also works with complex data types (tuples, vectors, structs, tuples of structs, …)

```javascript
# To train the model, we use batches of 128 samples, and one-hot encoding:
target = Flux.onehotbatch(pclass, [0, 1, 2])  # 2xN OneHotMatrix
loader = Flux.DataLoader((pcoord, target) |> gpu, batchsize=128, shuffle=true);
```

```
losses = []
@showprogress for epoch in 1:1_000
    for (x, y) in loader
        loss, grads = Flux.withgradient(model) do m
            # Evaluate model and loss inside gradient context:
            y_hat = m(x)
            Flux.crosentropy(y_hat, y)
        end
        Flux.update!(optim, model, grads[1])
        push!(losses, loss)  # logging, outside gradient context
    end
end

Progress: 100%| Time: 0:00:45
```

```
[38]: plot(losses, xaxis=:log, yaxis=:log)
```

```
[38]:
```
Train the Model

Flux.train!(model, loader, optim) does all of this
We’ve Finished Building our AI Classifier
We’ve Finished Building our AI Classifier

```plaintext
function classify(coords)
    prob_class = model(coords |> gpu) |> cpu
    return map(x->x[1], argmax(prob_class, dims=1))[;]
end
```
ML Model Infers

```julia
function classify(coords)
    prob_class = model(coords |> gpu) |> cpu
    return map(x->x[1], argmax(prob_class, dims=1))[1];
end

classify (generic function with 1 method)

rand_coord = 6 .* rand(Float32, 2, 10_000) .- 3

2×10000 Matrix{Float32}:
  1.58479  -2.6759  -0.757071  -1.1845   -0.673121  0.585823  1.14873
  -1.77659  -2.31748  0.752013   -1.3579  -0.0657971  -2.94806   -1.41852

p_true = scatter(pcoord[1, :], pcoord[2, :], zcolor=`rate, title="Raw Data")
p_nn   = scatter(pcoord[1, :], pcoord[2, :], zcolor=classify(pcoord), title="NN Classifier")
p_rand = scatter(rand_coord[1, :], rand_coord[2, :], zcolor=classify(rand_coord), title="NN Classifier")
plot(p_true, p_nn, p_rand, layout=[1,3], size=(1500,330), legend=false)
```

[39]

[40]

[41]

[41]
[40]: rand_coord = 6 .* rand(Float32, 2, 10_000) .-= 3

[40]: 2×10000 Matrix{Float32}:
    1.58479   -2.6759   -0.757071  -1.1845    -0.673121  0.585823  1.14873
    -1.77659  -2.31748   0.752013   -1.3579    -0.0657971 -2.94806  -1.41052

[41]: plot(p_true, p_val, p_rand, layout=(1, 3), size=(1200, 350), legend=false)
Using Dagger.jl To parallelize your workflow
**Dagger.jl: Easy work distribution**

```julia
using Distributed, ClusterManagers, BenchmarkTools

addprocs(2, N="2", constrain="cpu", qos="interactive", time="08:15:00", A="nstaff")

connecting to worker 1 out of 2
srun: job 18135272 queued and waiting for resources
srun: job 18135272 has been allocated resources
connecting to worker 2 out of 2

2-element Vector(Int64):
  2
  3

@everywhere using Dagger

darr1 = rand(Blocks(500, 500), 1_000, 1_000);
darr2 = rand(Blocks(500, 500), 1_000, 1_000);

function test_mul()
    darr1 .*= darr2
    wait;
end

test_mul (generic function with 1 method)

@benchmark(test_mul())

BenchmarkTools.Trial: 8120 samples with 1 evaluation.
  Range (min ... max): 340.665 μs - 1.253 s  GC (min ... max): 0.00% ... 87.19%
  Time  (median): 432.180 μs    GC (median): 0.00%
  Time  (mean ± σ): 612.252 μs ± 13.935 ms  GC (mean ± σ): 21.96% ± 0.97%

Memory estimate: 167.05 KiB, allocs estimate: 3113.
```
Dagger.jl: Easy work distribution

```julia
using Distributed, ClusterManagers, BenchmarkTools

addprocs(1, N="2", constrain="cpu", qos="interactive", time="08:15:00", A="nstaff")

@everywhere using Dagger

darr1 = rand(Blocks(500, 500), 1_000, 1_000);
darr2 = rand(Blocks(500, 500), 1_000, 1_000);

darr1 ∘ darr2
wait;

@test_mul (generic function with 1 method)
@benchmark(test_mul())

BenchmarkTools.Trial: 8128 samples with 1 evaluation.
  Range (min ... max):  340.665 μs  -  1.253 s  |  GC (min ... max):  0.00%  ... 87.19%
  Time (median):  432.180 μs  |  GC (median):  0.00%
  Time (mean ± σ):  612.252 μs ± 13.935 ms  |  GC (mean ± σ):  21.96% ± 0.97%

341 μs  Histogram: frequency by time  676 μs <

Memory estimate: 167.05 KiB, allocs estimate: 3113.
```
Dagger.jl: Easy work distribution

```julia
using Distributed, ClusterManagers, BenchmarkTools

@everywhere using Dagger

darr1 = rand(Blocks(500, 500), 1000, 1000);

function test_mul()
    darr1 * darr2
    wait;
end

BenchmarkTools.Trial: 8120 samples with 1 evaluation. 
  Range (min .. max):  340.665 µs ..  1.253 s  |  GC (min .. max):  0.00% ..  87.19%
  Time (median):    432.180 µs |  GC (median):   0.00%
  Time (mean ± σ):  612.252 µs ± 13.935 ms |  GC (mean ± σ): 21.96% ±  0.97%

Memory estimate: 167.05 KiB, allocs estimate: 3113.
```
Dagger.jl: Easy work distribution

For references: openBLAS (single node) = 3.15ms, cuBLAS = 121μs
Dagger Provides Async Task Spawning

```julia
[10]: th = Dagger.@spawn sleep(10)
    while true
        st = isready(th)
        println("Is ready = $(st)")
        if ! st
            sleep(1)
        else
            break
        end
    end

    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = false
    isready = true

[11]: @everywhere function f(dist, len, reps, o)
    v = Vector(Float64)(undef, len)  # avoiding allocations
    maximum(mean(rand!(dist, v)) for _ in 1:reps)/o, myid()
end

[14]: function experiments_dagger(distls, lens, K=1000)
    res = DataFrame()
    @sync for T in distls
        dist = T()
        o = Dagger.@spawn std(dist)
        for L in lens
            z = Dagger.@spawn f(dist, L, K, o)
            push!(res, (;T, o, L, z))
        end
    end
end
```
Dagger Provides Async Task Spawning

Tasks start immediately. Spawning is non-blocking. This is called EagerThunk.
Dagger Provides Async Task Spawning

Functions to be used by `@spawn` need to be defined with `@everywhere`.
Use Loop to Submit Tasks to Workers
Use Loop to Submit Tasks to Workers

```julia
function experiments_dagger(dists, lens, K=1000)
    res = DataFrame()
    @sync for T in dists
        dist = T()
        σ = Dagger.@spawn std(dist)
        for L in lens
            z = Dagger.@spawn f(dist, L, K, σ)
            push!(res, (;T, σ, L, z))
        end
    end
    res.z = fetch.(res.z)
    res.σ = fetch.(res.σ)
    res
end
```

@spawn is non-blocking, fetch or using a variable are blocking
Where Did All The Tasks Run?
Where Did All The Tasks Run?

All the tasks are distributed over the Distributed.jl processors.
Time to Accelerate the MC Sampler

Serial version:

1. Simulate N steps
2. Simulate `ensemble.n` independent trajectories
3. Aggregate data

```python
times, positions, velocities, total_energies = baoab(
    energy, force, N*dt, dt,
    gamma, kBT, initial_position, initial_velocity;
    save_frequency=save_frequency, kwars...
)

p_r = rand(ensemble.d, ensemble.n)
v_r = rand(ensemble.d, ensemble.n)

dtasks = Any[]
for i in 1:ensemble.n
    p = initial_position + p_r[i]
    v = initial_velocity + v_r[i]

    dtask = baoab(
        energy, force, ensemble.len*dt, dt,
        gamma, kBT, p + p_r[i], v + v_r[i];
        save_frequency=save_frequency, kwars...
    )

    push!(dtasks, dtask)
end

ensemble_t = Array{T, 1}[]
ensemble_p = Array{T, 1}[]
for task in dtasks
    t_i, p_i, v_i, _ = task
    push!(ensemble_t, t .+ t_i)
    push!(ensemble_p, p_i)
end
```
Time to Accelerate the MC Sampler

Naive parallelization:

1. Simulate N steps
2. Spawn `ensemble.n` independent trajectory simulation tasks
3. Fetch data

```python
times, positions, velocities, total_energies = baoab(
    energy, force, N*dt, dt,
    gamma, kB*T, initial_position, initial_velocity;
    save_frequency=save_frequency, kwars...
)

p_r = rand(ensemble.d, ensemble.n)
v_r = rand(ensemble.d, ensemble.n)

dtasks = Dagger.EagerThunk[]
@sync for i in 1:ensemble.n
    p = initial_position + p_r[i]
    v = initial_velocity + v_r[i]

    dtask = Dagger.@spawn baoab(
        energy, force, ensemble.len*dt, dt,
        gamma, kB*T, p + p_r[i], v + v_r[i];
        save_frequency=save_frequency, kwars...
    )

    push!(dtasks, dtask)
end

ensemble_t = Array{T, 1}[]
ensemble_p = Array{T, 1}[]
for task in dtasks
    t_i, p_i, _, _ = fetch(task)
    push!(ensemble_t, t_i]
    push!(ensemble_p, p_i)
end
```
There is No Free Lunch
There is No Free Lunch

```python
benchmarks integrate_ensemble(4, initial_position, initial_velocity)
```

Benchmark: 100 samples with 1 evaluation.

<table>
<thead>
<tr>
<th>Range (min ... max)</th>
<th>Time (median)</th>
<th>Time (mean ± σ)</th>
<th>GC (min ... max)</th>
<th>GC (median)</th>
<th>GC (mean ± σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>46.908 ms ... 58.275 ms</td>
<td>49.374 ms</td>
<td>50.150 ms ± 2.361 ms</td>
<td>14.97% ... 26.60%</td>
<td>16.31%</td>
<td>16.60% ± 2.01%</td>
</tr>
</tbody>
</table>

```

Memory estimate: 55.27 MiB, allocs estimate: 861963.
```

```
benchmarks integrate_ensemble_dagger(4, initial_position, initial_velocity)
```

Memory estimate: 264.10 MiB, allocs estimate: 4976193.

```
benchmarks integrate_ensemble(4, initial_position, initial_velocity)
```

Memory estimate: 3.21 MiB, allocs estimate: 62962.
There is No Free Lunch

You’re not keeping workers busy enough to amortize @spawn costs (+too much data movement)
Keep Workers Busy Enough

Unit of work: simulate several trajectories (or make each trajectory longer).

Don’t transfer data unnecessarily: We’re only interested in how much trajectories spread out.
Keep Workers Busy Enough

Improved parallelization:

1. Simulate N steps
2. Spawn batches of trajectory simulation tasks
3. Fetch only needed data

```
function baoab_ensemble_step_dagger_2(
    energy::Function, force::Function, t, N, dt,
    gamma, kBT, initial_position::T, initial_velocity::T;
    save_frequency = 3, ensemble = (len=10, d=Normal(0, 0.01), n=10),
    kwars...
) where T <: Number

    times, positions, velocities, total_energies = baoab(
        energy, force, N*dt, dt,
        gamma, kBT, initial_position, initial_velocity;
        save_frequency=save_frequency, kwars...
    )

    dtasks = Dagger.EagerThunk[]
    for i in 1:ensemble.n/ensemble.batch
        dtask = Dagger.@spawn baoab_ensemble_batch(
            energy, force, ensemble.len*dt, dt, ensemble.batch,
            gamma, kBT, initial_position, initial_velocity;
            save_frequency=save_frequency, kwars...
        )
        push!(dtasks, dtask)
    end

    hi_p = T[]
    lo_p = T[]
    for task in dtasks
        lo_p_i, hi_p_i = fetch(task)
        append!(hi_p, hi_p_i)
        append!(lo_p, lo_p_i)
    end

    return t .+ times, positions, velocities, total_energies, (:p_spread=max(hi_p)-min(lo_p))
end
```
Busy Workers Amortize Overhead

Also note: you can scale to many nodes
Conclusions
Conclusion

● Julia provides a rich ecosystem to build performant distributed applications on HPC systems
  ○ Saw examples of Jupyter (IJulia.jl); Beginnings of sophisticated multi-node workflows (Distributed.jl, Dagger.jl); Programming GPUs (CUDA.jl); and AI (Flux.jl)
● Modern high-productivity design
● HPC vendor aware. Built on top of LLVM, with vendor backends (CUDA.jl, AMDGPU.jl, oneAPI.jl, etc)
● Provides interfaces to examine and manipulate what you’re doing (including LLVM IR)
Noteworthy Julia Packages (for HPC)

- **JuliaIO**: [https://github.com/JuliaIO](https://github.com/JuliaIO)
  JuliaData: [https://github.com/JuliaData](https://github.com/JuliaData)
  Collects many Julia packages around I/O and Data

- **JuliaParallel**: [https://github.com/JuliaParallel](https://github.com/JuliaParallel)
  Collects many Julia packages around distributed and parallel computing

- **JuliaGPU**: [https://github.com/JuliaGPU](https://github.com/JuliaGPU)
  Collects many Julia packages used for GPU computing
Noteworthy I/O Packages

- **Pidfile.jl**: Provides the linux/unix pidfile mechanism to hold mutex’es – useful for locking files
- **HDF5.jl**: HDF5-file support
- **Zarr.jl**: Julia Zarr (N-D array compressed data) support
- **JLD.jl / JLD2.jl**: Julia-native serialization support
- **Tables.jl / DTables.jl / DistributedArrays.jl**: arrays and tables build on distributed / **CSV.jl**: Tabular data support
- **JuliaDB.jl**: A distributed database for tables (implemented in pure Julia)
Noteworthy REST and Web Frameworks

- **HTTP.jl**: Send and receive HTTP requests
- **Mux.jl / Oxygen.jl**: Routing middleware for HTTP requests – Oxygen is newer and makes multithreading easier (considered an all-Julia replacement for FastAPI)
- **Genie.jl**: Fully-fledged web development framework (Julia’s answer to Flask)
Noteworthy HPC Packages

“Traditional” HPC support: (https://github.com/JuliaParallel)

• **MPI.jl**: no explanation needed (it is CUDA/ROCM-aware)
• **ClusterManagers.jl**: manager HPC resources on the fly (also note SlurmClusterManager.jl and MPIClusterManagers.jl for HPC clusters)
• **ImplicitGlobalGrid.jl / MPIArrays.jl**: implement a global address space (using the Array interface) built on MPI.jl
Noteworthy HPC Packages

Tasking (producer-consumer) style HPC support: (https://github.com/JuliaParallel)

- **Distributed.jl / Dagger.jl**: task-based parallelism (like Dask and Ray)
- **DTables.jl / DistributedArrays.jl**: arrays and tables build on distributed

ML support: **Flux.jl** (like pytorch, but different)
Noteworthy HPC Packages

GPU Support:
(https://github.com/JuliaGPU)

- **CUDA.jl / AMDGPU.jl / oneAPI.jl**: low-level GPU support (expose GPU Array interface + helper functions to manage GPU resources)
- **KernelAbstractions.jl**: lets you write portable code by writing portable kernels (a bit “like” Kokkos)
- + Many Many more
Perlmutter system configuration

NVIDIA "Ampere" GPU Nodes
- 4x GPU + 1x CPU
- 40 GiB HBM + 256 GiB DDR
- 4x 200G "Slingshot" NICs

AMD "Milan" CPU Node
- 2x CPUs
- > 256 GiB DDR4
- 1x 200G "Slingshot" NIC

Centers of Excellence
- Network
- Storage
- App. Readiness
- System SW

Compute racks
- 64 blades

Perlmutter system
- GPU racks
- CPU racks
- ~6 MW

Blades
- 2x GPU nodes or 4x CPU nodes
CUDA.jl provides detailed profiling interface

### Host-side activity: calling CUDA APIs took 1.96 ms (87.81% of the trace)

<table>
<thead>
<tr>
<th>Time (%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.29%</td>
<td>653.51 µs</td>
<td>1</td>
<td>653.51 µs</td>
<td>653.51 µs</td>
<td>653.51 µs</td>
<td>cuMemAllo</td>
</tr>
<tr>
<td>18.15%</td>
<td>404.83 µs</td>
<td>1</td>
<td>404.83 µs</td>
<td>404.83 µs</td>
<td>404.83 µs</td>
<td>cudaEvent</td>
</tr>
<tr>
<td>1.12%</td>
<td>25.03 µs</td>
<td>1</td>
<td>25.03 µs</td>
<td>25.03 µs</td>
<td>25.03 µs</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>0.82%</td>
<td>18.36 µs</td>
<td>1</td>
<td>18.36 µs</td>
<td>18.36 µs</td>
<td>18.36 µs</td>
<td>cudaMemset</td>
</tr>
<tr>
<td>0.44%</td>
<td>9.78 µs</td>
<td>2</td>
<td>4.89 µs</td>
<td>953.67 ns</td>
<td>8.82 µs</td>
<td>cudaOccup</td>
</tr>
<tr>
<td>0.28%</td>
<td>6.2 µs</td>
<td>3</td>
<td>2.07 µs</td>
<td>476.84 ns</td>
<td>4.77 µs</td>
<td>cudaStream</td>
</tr>
<tr>
<td>0.26%</td>
<td>5.72 µs</td>
<td>1</td>
<td>5.72 µs</td>
<td>5.72 µs</td>
<td>5.72 µs</td>
<td>cudaEvent</td>
</tr>
<tr>
<td>0.00%</td>
<td>0.0 ns</td>
<td>1</td>
<td>0.0 ns</td>
<td>0.0 ns</td>
<td>0.0 ns</td>
<td>cudaGetLastError</td>
</tr>
</tbody>
</table>

### Device-side activity: GPU was busy for 160.22 µs (7.18% of the trace)

<table>
<thead>
<tr>
<th>Time (%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.09%</td>
<td>158.07 µs</td>
<td>1</td>
<td>158.07 µs</td>
<td>158.07 µs</td>
<td>158.07 µs</td>
<td>ampere_sg</td>
</tr>
<tr>
<td>0.10%</td>
<td>2.15 µs</td>
<td>1</td>
<td>2.15 µs</td>
<td>2.15 µs</td>
<td>2.15 µs</td>
<td>[set devi]</td>
</tr>
</tbody>
</table>

1 column omitted
CUDA.jl is compatible with Structs
CUDA.jl is compatible with Structs

Julia converts struct to cuda-compatible type

```
[17]:
struct Point{T}
    x :: T
    y :: T
end

[18]:
arr = Point.(rand(100), rand(100))
carr = cu(arr)
```
```julia
using Distributed

addprocs(10)

10-element Vector{Int64}:
2 3 4 5 6 7 8 9 10 11

@everywhere using Dagger, Random, Distributions, StatsBase, DataFrames

WARNING: using Dagger.In in module Main conflicts with an existing identifier.
WARNING: using Dagger.Out in module Main conflicts with an existing identifier.

th = Dagger.@spawn sleep(10)
while true
    st = isready(th)
    println("isready = "+st)
    if !st
        sleep(1)
    else
        break
    end
end

isready = false
isready = false
isready = false
isready = false
isready = false
isready = false
```
Why does NERSC care about Julia?
NERSC is the mission HPC and data facility for the U.S. Department of Energy Office of Science.

Top Science Disciplines
(By computational hours used)

- Nuclear Physics
- Chemical Sciences
- High Energy Physics
- Geosciences
- Plasma Science
- Biological Systems Science
- Fusion Energy

Scientific User Facilities
- Material Sciences
- Small Business Innovation Research
- Biosciences
- Climate and Environmental Science

Breakdown of Compute Used by DOE Program

- <1% Small Business Innovation Research
- 1% Biological Systems Science
- 2% Scientific User Facilities
- 11% Climate and Environmental Science
- 13% Nuclear Physics
- 19% High Energy Physics
- 18% Chemical Sciences, Geosciences, and Biosciences
- 18% Material Sciences
- 15% Fusion Energy and Plasma Science

~1,000 Projects

~9,000 ANNUAL USERS FROM ~800 Institutions + National Labs

2021 NERSC USERS ACROSS US AND WORLD
- 50 States + Washington D.C. and Puerto Rico
- 46 Countries

27% Graduate Students
17% Postdoctoral Fellows
14% Staff Scientists
11% University Faculty
7% Undergraduate Students
6% Professional Staff
3% Industry
1% Small Businesses
<1% Private Labs
59% Universities
5% Other Government Labs
29% DOE Labs
1% Small Businesses
<1% Private Labs

>2,000 Scientific Journal Articles per Year
NERSC is the mission HPC and data facility for the U.S. Department of Energy Office of Science.

- Most users at NERSC are not HPC experts
  - and we can’t force them to become ones
- Workflows running at NERSC are incredibly varied
  - in response, NERSC systems provide a range of capabilities
- => Julia needs to “know what to do” by default
  - Need: intelligent, easy to support, and robust interface with HPC resources

>2,000 Scientific Journal Articles per Year