

Distributed and GPU Computing Ecosystem in Julia



“Julia + Jupyter + GPU = 🐾 🧪 💃”
(phrase borrowed from Marius Milea)

Johannes Blaschke
Data Science Engagement Group
NERSC, LBNL

Credit and Disclaimers

None of this would be possible without:

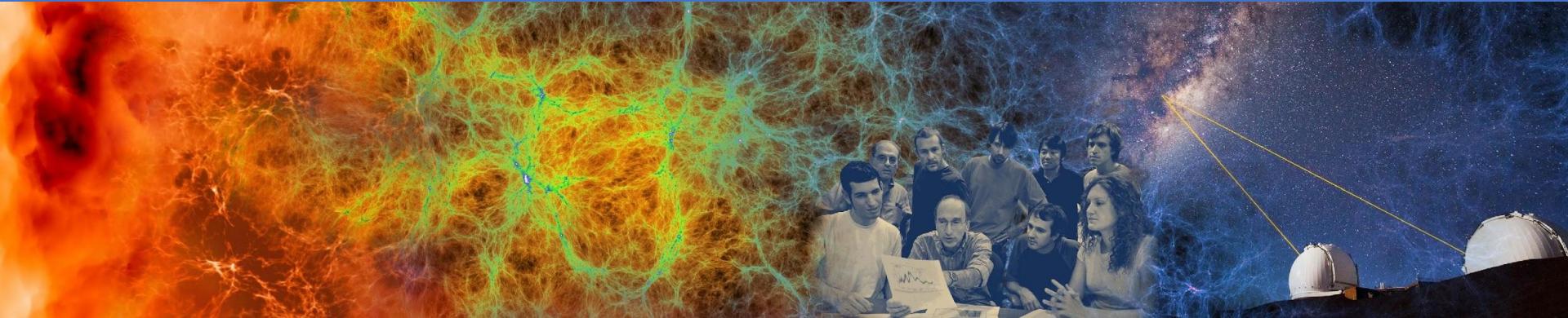
- Tim Bersard, Valentin Churavy, Julian Samaroo (MIT Julia Lab) + Anton Smirnov (AMD) + Carsten Bauer (NHR, PC2)
 - Providing the Infrastructure
- Marius Millea (UC Davis) + Mark Hirsbrunner (LBNL) + William Godoy, Pedro Valero Lara (OLCF)
 - Inspiring applications
- The Julia for HPC working group
 - <https://github.com/JuliaParallel>
 - Meets monthly on Zoom (cf. <https://julialang.org/community/>) and is very active on Discord
 - Julia for HPC BoF on Thursday

Disclaimer:

- I work on Perlmutter – All my GPU work is based on NVIDIA, but can be easily applied to AMD and Intel.



Julia in 60s

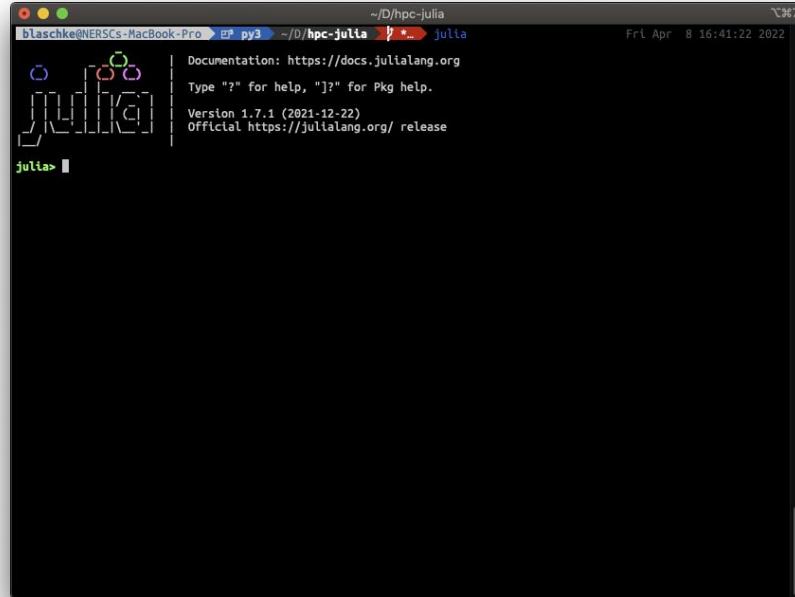


Julia is a High-Productivity Language

- It has all the modern HP features (rich stdlib, gc, ...)

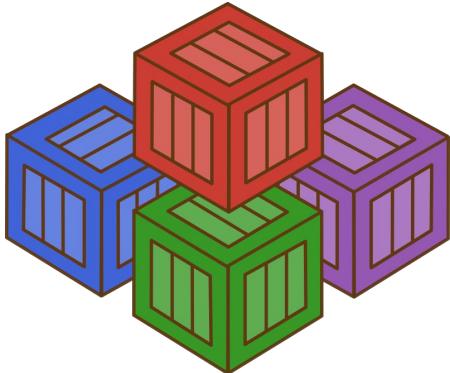
Julia is a High-Productivity Language

- 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 :

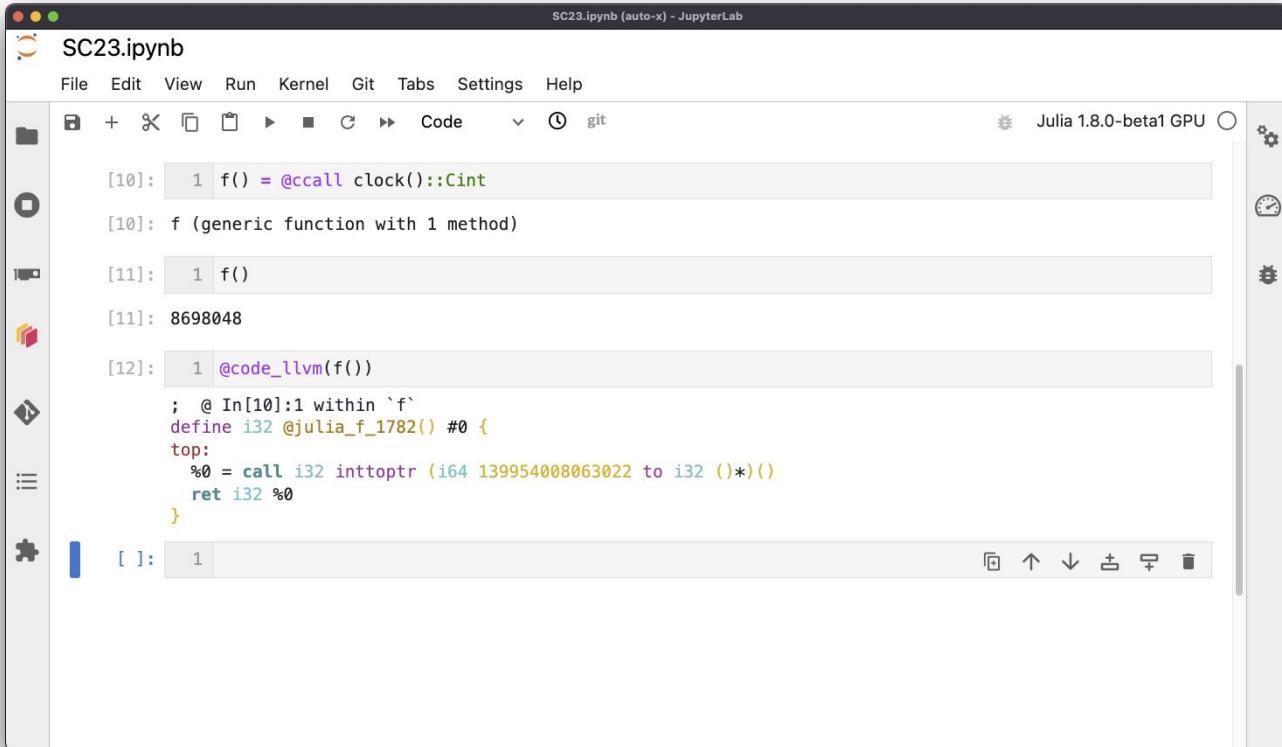
```
[extras]
MPIPreferences = "3da0fdf6-3ccc-4f1b-acd9-58baa6c99267"
CUDA_Runtime_jll = "76a88914-d11a-5bdc-97e0-2f5a05c973a2"
```

LocalPreferences.toml :

```
[MPIPreferences]
_format = "1.1"
abi = "MPICH"
binary = "system"
cclibs = ["cupti", "cudart", "cuda", "sci_gnu_82_mpi", "sci_gnu_82",
"dl", "dsmlm", "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



The screenshot shows a JupyterLab interface titled "SC23.ipynb". In the top right corner, it indicates "Julia 1.8.0-beta1 GPU". The notebook contains the following code and output:

```
[10]: 1 f() = @ccall clock()::Cint
[10]: f (generic function with 1 method)

[11]: 1 f()
[11]: 8698048

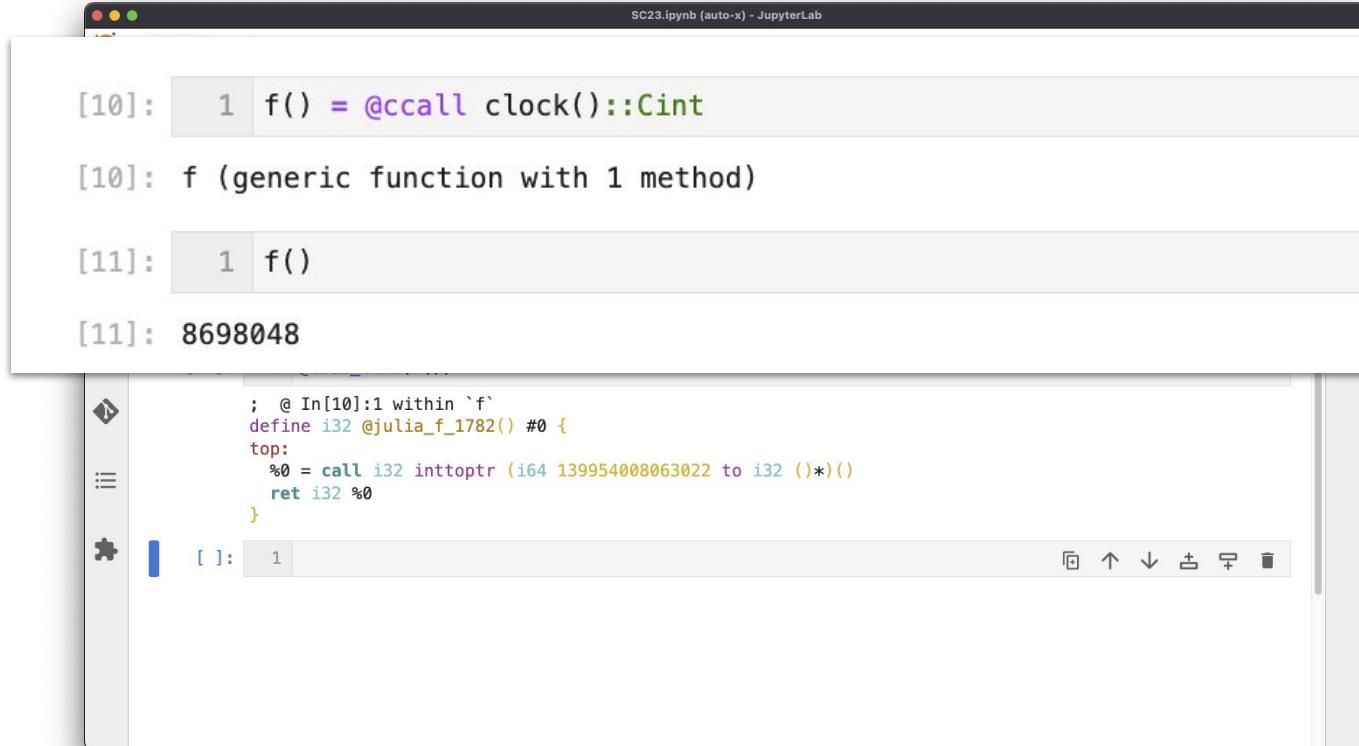
[12]: 1 @code_llvm(f())
; @ In[10]:1 within `f`
define i32 @_julia_f_1782() #0 {
top:
    %0 = call i32 inttoptr (i64 139954008063022 to i32 (*)())
    ret i32 %0
}

[ ]: 1
```

Julia has LLVM under the Hood

Julia data types are
binary-compatible with C

@ccall equivalent to c
function call



The screenshot shows a JupyterLab notebook interface with the following content:

```
SC23.ipynb (auto-x) - JupyterLab
```

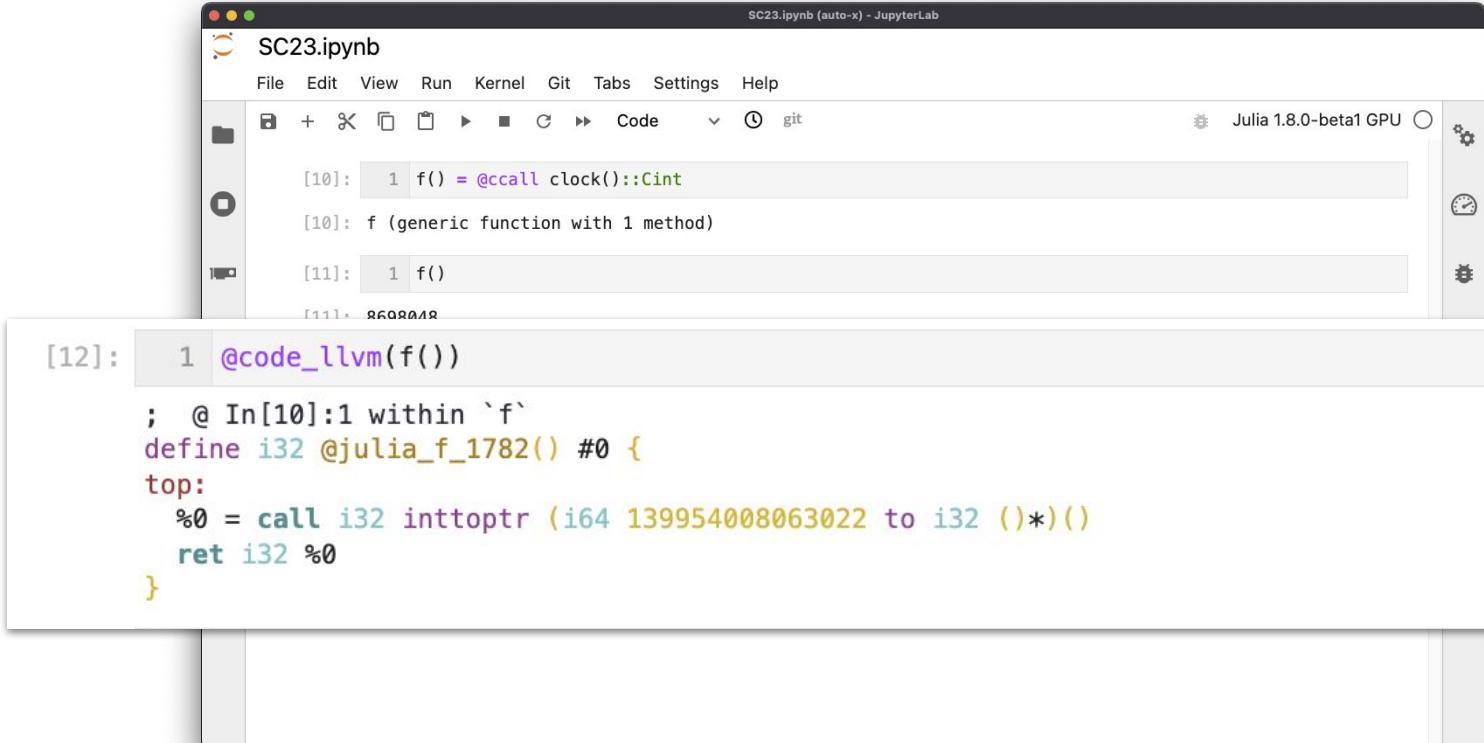
```
[10]: 1 f() = @ccall clock()::Cint
[10]: f (generic function with 1 method)

[11]: 1 f()
[11]: 8698048
```

```
; @ In[10]:1 within `f`
define i32 @_julia_f_1782() #0 {
top:
    %0 = call i32 inttoptr (i64 139954008063022 to i32 (*)())
    ret i32 %0
}
```

The interface includes a toolbar with icons for file operations, a search bar, and a status bar at the bottom.

Julia has LLVM under the Hood



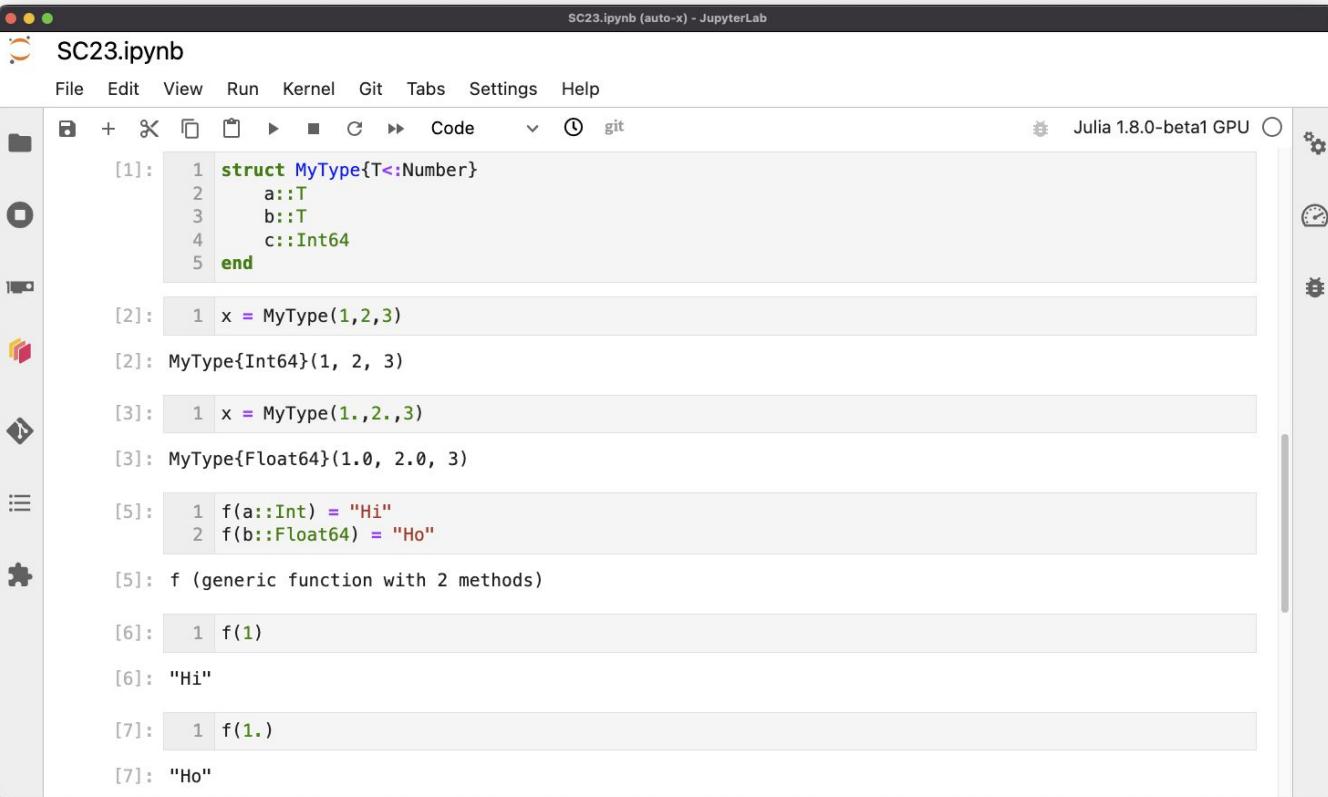
The screenshot shows a JupyterLab interface with a notebook titled "SC23.ipynb". The code cell [12] contains the command `@code_llvm(f())`. The resulting LLVM IR output is:

```
; @ In[10]:1 within `f`
define i32 @julia_f_1782() #0 {
top:
    %0 = call i32 inttoptr (i64 139954008063022 to i32 ()*)
    ret i32 %0
}
```

`@code_llvm` exposes the LLVM IR for debug purposes



Julia has a Powerful Type System



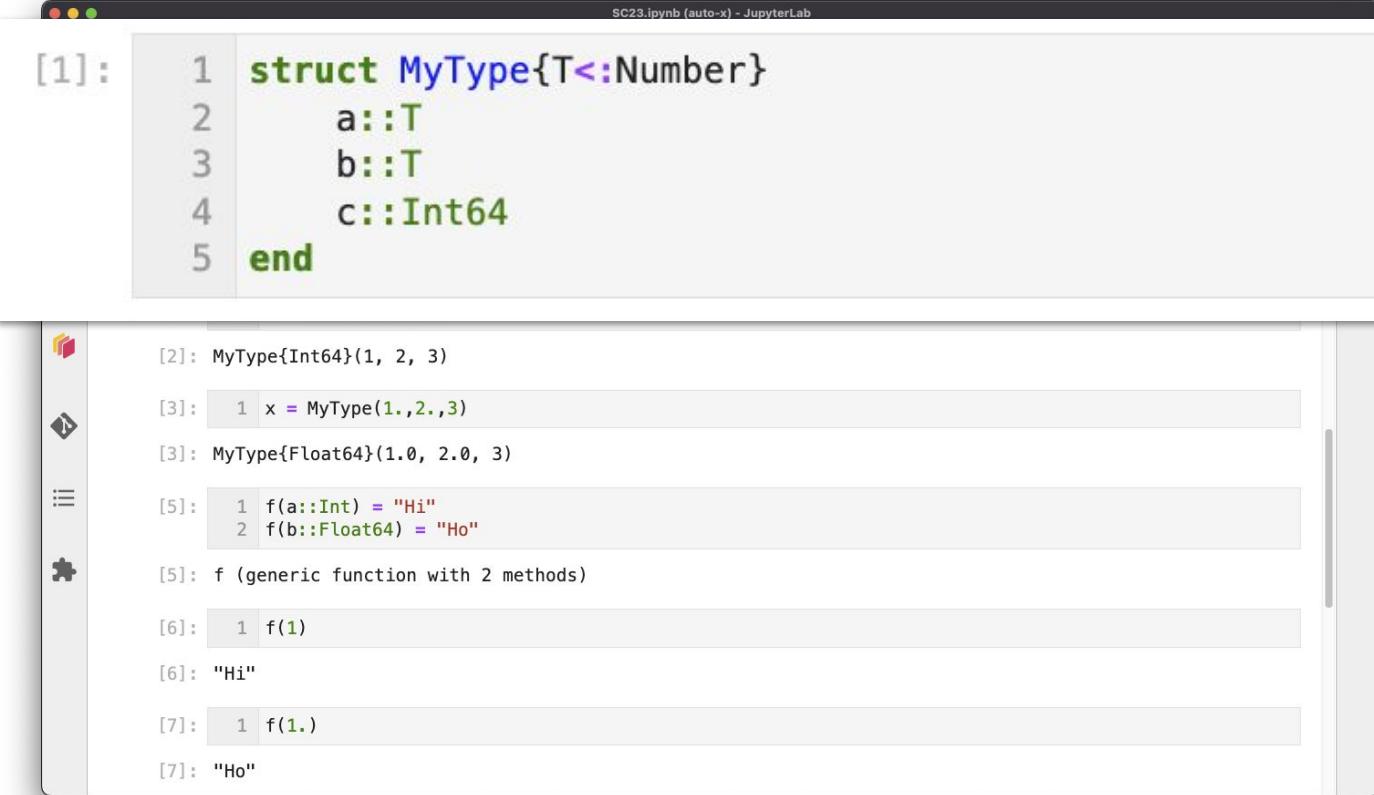
The screenshot shows a JupyterLab interface with a Julia kernel. The code cell [1] defines a struct `MyType{T<:Number}` with fields `a::T`, `b::T`, and `c::Int64`. The code cell [2] creates an instance `x = MyType(1, 2, 3)`. The code cell [3] creates another instance `x = MyType(1., 2., 3)`. The code cell [5] defines a generic function `f` with two methods: one for `Int` arguments and one for `Float64` arguments. The code cell [6] calls `f(1)` and prints "Hi". The code cell [7] calls `f(1.)` and prints "Ho".

```
SC23.ipynb (auto-x) - JupyterLab
File Edit View Run Kernel Git Tabs Settings Help
Julia 1.8.0-beta1 GPU
[1]: 1 struct MyType{T<:Number}
      2     a::T
      3     b::T
      4     c::Int64
      5 end
[2]: 1 x = MyType(1,2,3)
[2]: MyType{Int64}(1, 2, 3)
[3]: 1 x = MyType(1.,2.,3)
[3]: MyType{Float64}(1.0, 2.0, 3)
[5]: 1 f(a::Int) = "Hi"
      2 f(b::Float64) = "Ho"
[5]: f (generic function with 2 methods)
[6]: 1 f(1)
[6]: "Hi"
[7]: 1 f(1.)
[7]: "Ho"
```

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



The screenshot shows a JupyterLab interface with a code editor and a history pane.

Code Editor:

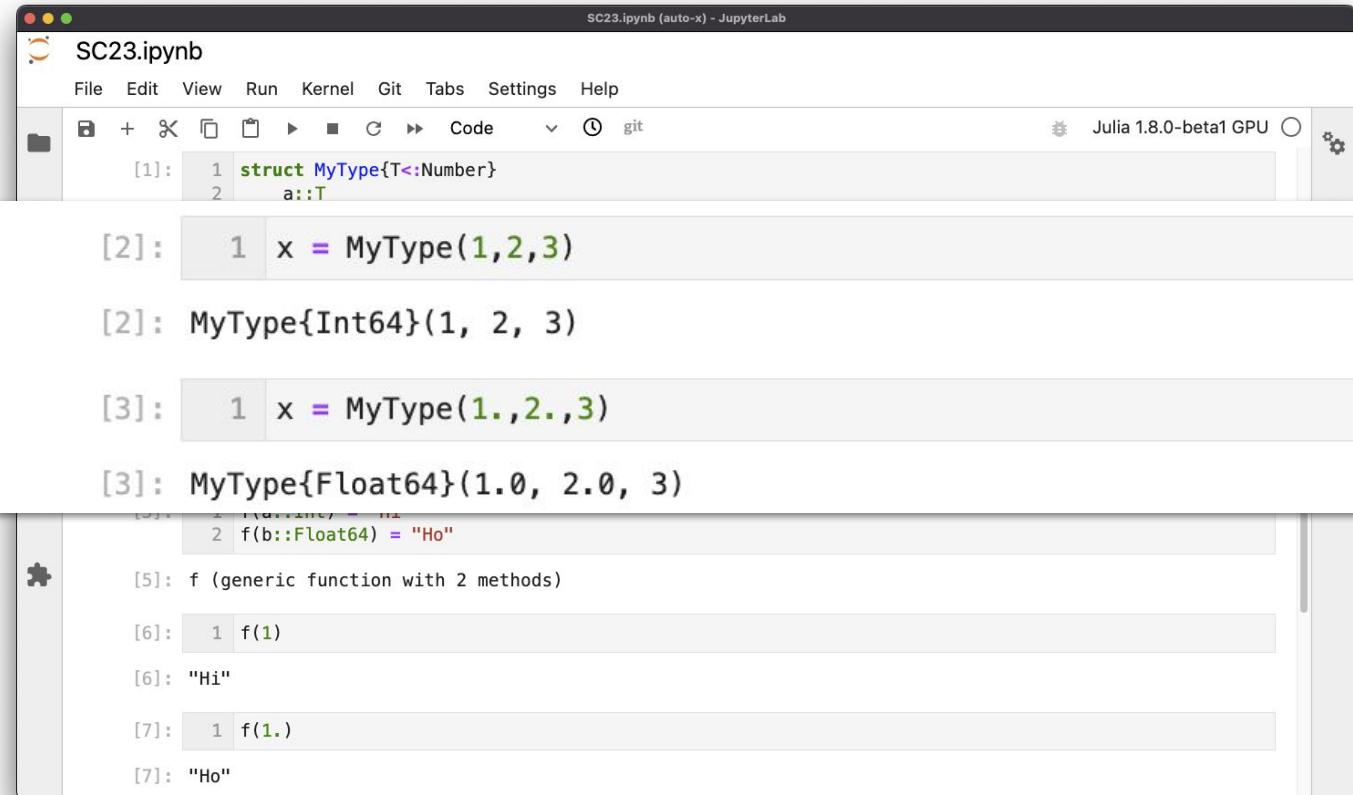
```
[1]: 1 struct MyType{T<:Number}
      2     a::T
      3     b::T
      4     c::Int64
      5 end
```

History Pane:

```
[2]: MyType{Int64}(1, 2, 3)
[3]: 1 x = MyType(1., 2., 3)
[3]: MyType{Float64}(1.0, 2.0, 3)
[5]: 1 f(a::Int) = "Hi"
      2 f(b::Float64) = "Ho"
[5]: f (generic function with 2 methods)
[6]: 1 f(1)
[6]: "Hi"
[7]: 1 f(1.)
[7]: "Ho"
```



Julia has a Powerful Type System



The screenshot shows a JupyterLab interface with a Julia kernel. The code cell [1] defines a type template:

```
[1]: struct MyType{T<:Number}
      a::T
```

The code cell [2] creates an instance of the type template with integers:

```
[2]: 1 x = MyType(1,2,3)
```

The code cell [2] also shows the resulting type inferred by the compiler:

```
[2]: MyType{Int64}(1, 2, 3)
```

The code cell [3] creates an instance of the type template with floating-point numbers:

```
[3]: 1 x = MyType(1.,2.,3)
```

The code cell [3] also shows the resulting type inferred by the compiler:

```
[3]: MyType{Float64}(1.0, 2.0, 3)
```

The code cell [5] defines a generic function f:

```
[5]: f (generic function with 2 methods)
```

The code cell [6] calls the function f with an integer argument:

```
[6]: 1 f(1)
```

The output of the function call is "Hi".

```
[6]: "Hi"
```

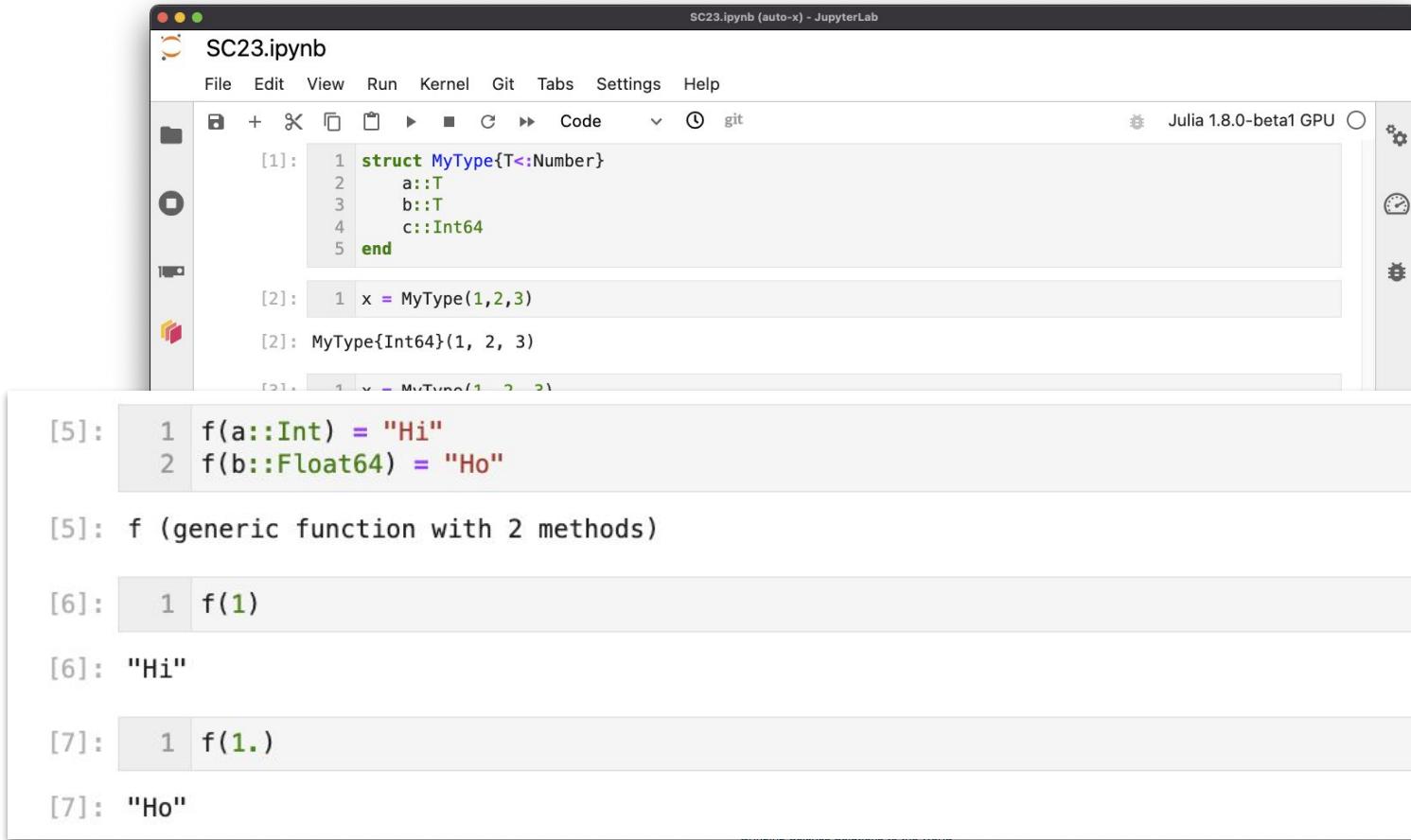
The code cell [7] calls the function f with a floating-point argument:

```
[7]: 1 f(1.)
```

The output of the function call is "Ho".

```
[7]: "Ho"
```

Julia has a Powerful Type System



The screenshot shows a JupyterLab interface with a Julia kernel. The top header bar indicates "SC23.ipynb (auto-x) - JupyterLab" and "Julia 1.8.0-beta1 GPU". The code editor displays the following Julia code:

```
1 struct MyType{T<:Number}
2   a::T
3   b::T
4   c::Int64
5 end

[2]: 1 x = MyType(1,2,3)

[2]: MyType{Int64}(1, 2, 3)

[5]: 1 f(a::Int) = "Hi"
2 f(b::Float64) = "Ho"

[5]: f (generic function with 2 methods)

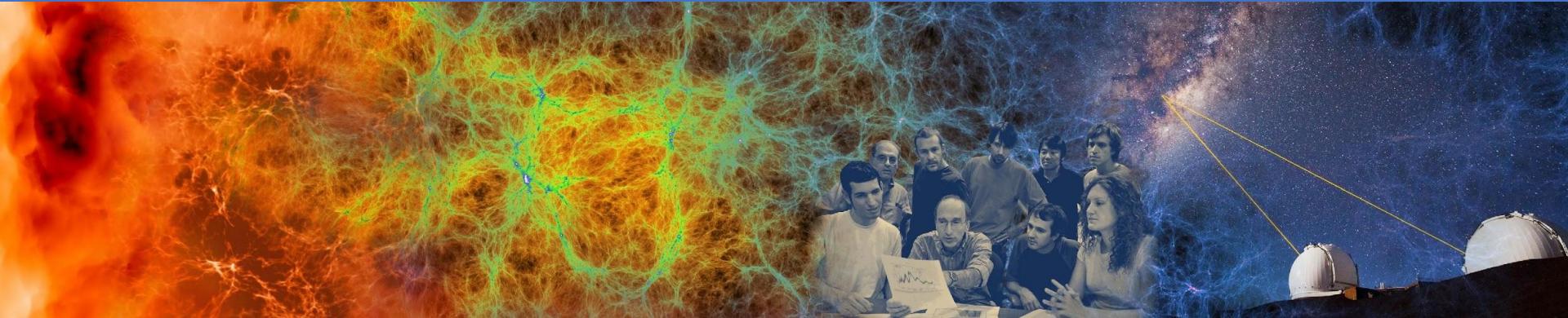
[6]: 1 f(1)

[6]: "Hi"

[7]: 1 f(1.)
[7]: "Ho"
```

Julia has multiple dispatch:
a function can have several
implementations (methods)
depending on the input
types

Jupyter as a workflow engine



Building a Distributed Julia Application (without MPI)

WF node

High-speed network

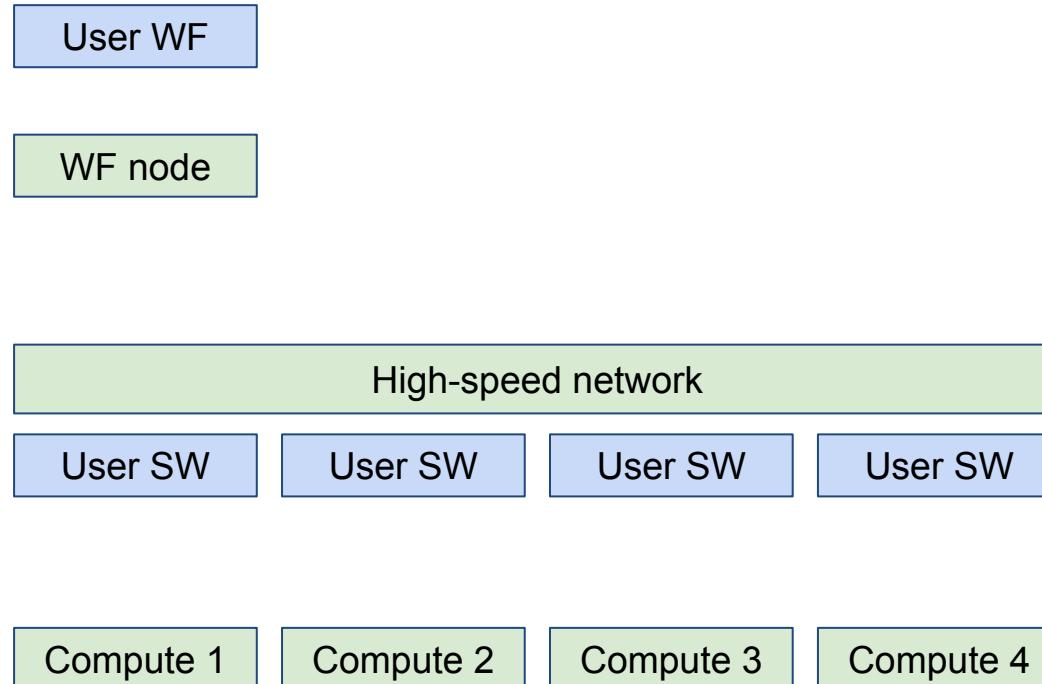
Compute 1

Compute 2

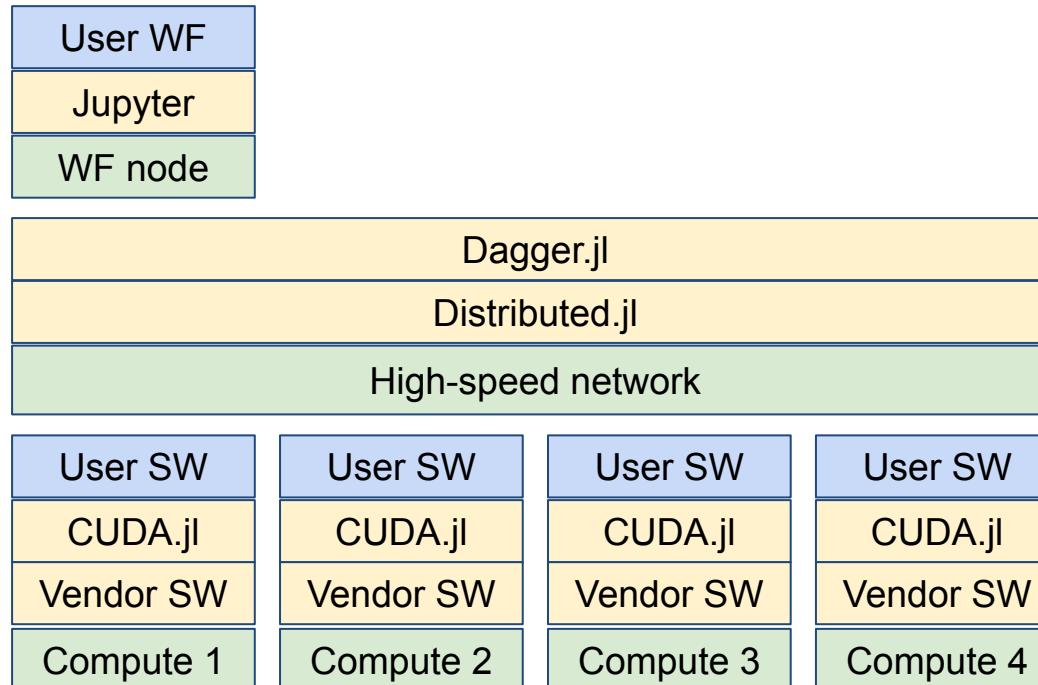
Compute 3

Compute 4

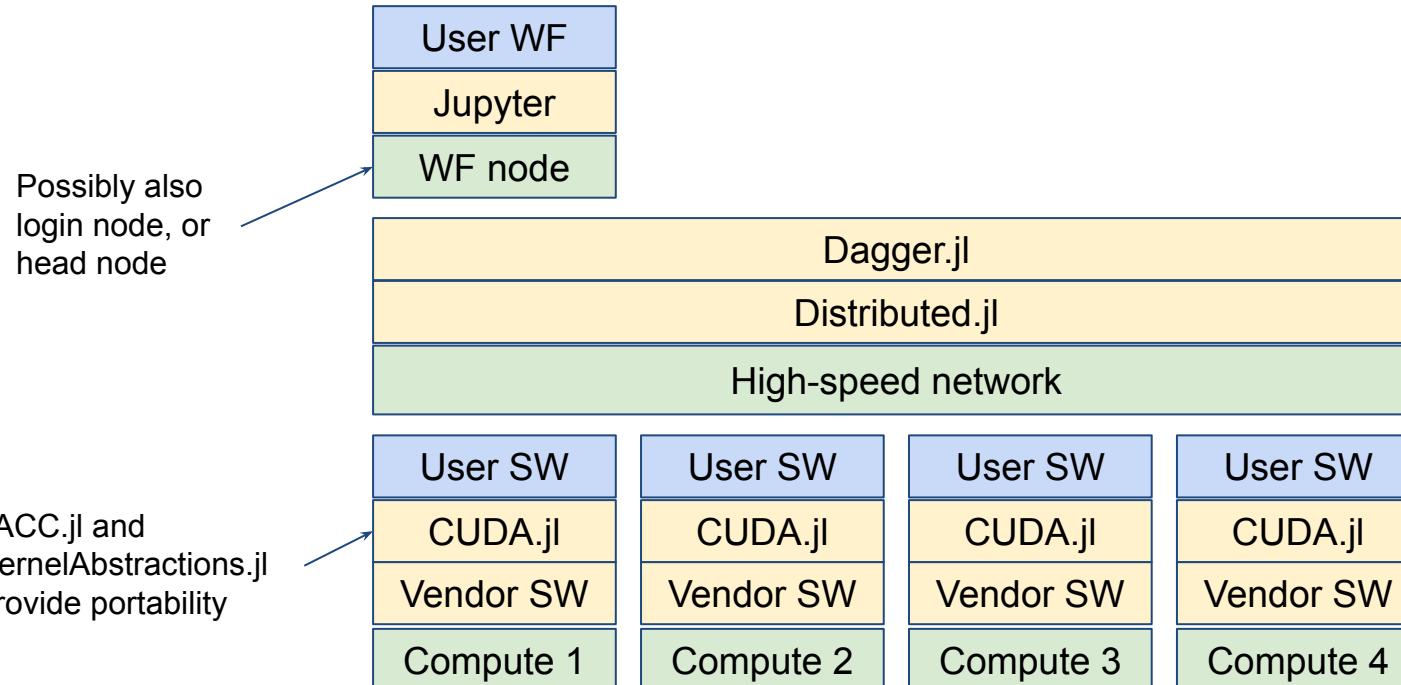
Building a Distributed Julia Application (without MPI)



Building a Distributed Julia Application (without MPI)



Building a Distributed Julia Application (without MPI)



Cluster Managers: Interaction with Slurm

The screenshot shows a JupyterLab interface titled "SC23.ipynb" running on "Julia 1.8.0-beta1 GPU". The code cell [25] imports the Distributed and ClusterManagers modules. Cell [26] adds two workers using the SlurmManager, with output showing the connection to workers and the submission of a job to the Slurm queue. Cells [26] and [27] show the creation of vectors for hosts and pids. Cells [28] and [29] print the contents of these vectors.

```
File Edit View Run Kernel Git Tabs Settings Help
SC23.ipynb (auto-x) - JupyterLab
Julia 1.8.0-beta1 GPU
[25]: 1 using Distributed, ClusterManagers
[26]: 1 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
[26]: 2-element Vector{Int64}:
      2
      3
[27]: 1 hosts = []
      2 pids = []
      3 for i in workers()
      4     host, pid = fetch(@spawnat i (gethostname(), getpid()))
      5     push!(hosts, host)
      6     push!(pids, pid)
      7 end
[28]: 1 hosts
[28]: 2-element Vector{Any}:
      "nid004709"
      "nid004953"
[29]: 1 pids
[29]: 2-element Vector{Any}:
      1605084
      1781811
[ ]: 1
```

Cluster Managers: Interaction with Slurm

Request two
cpu nodes

```
[26]: 1 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
```

ElasticManager
can be used instead
of SlurmManager
to manually manage
workers from *within*
an allocation

```
[27]: 1 hosts = []
2 pids = []
3 for i in workers()
4     host, pid = fetch(@spawnat i (gethostname(), getpid()))
5     push!(hosts, host)
6     push!(pids, pid)
7 end
```

```
[28]: 1 hosts
```

```
[28]: 2-element Vector{Any}:
 "nid004709"
 "nid004953"
```

```
[29]: 1 pids
```

```
[29]: 2-element Vector{Any}:
 1605084
 1781811
```

```
[ ]: 1
```



Cluster Managers: Interaction with Slurm

The screenshot shows a JupyterLab interface with a Julia 1.8.0-beta1 GPU kernel. The code cell [25] imports the Distributed and ClusterManagers modules. Cell [26] adds two workers using the SlurmManager, specifying constraints (cpu), quality of service (debug), and time limit (00:5:00). Cells [27] and [28] demonstrate fetching hostnames and pids from the workers. Cell [29] prints the pids.

```
File Edit View Run Kernel Git Tabs Settings Help
SC23.ipynb
Julia 1.8.0-beta1 GPU
[25]: 1 using Distributed, ClusterManagers
[26]: 1 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
[27]: 1 hosts = []
       2 pids = []
       3 for i in workers()
       4     host, pid = fetch(@spawnat i (gethostname(), getpid()))
       5     push!(hosts, host)
       6     push!(pids, pid)
       7 end
[28]: 2-element Vector{Any}:
       "nid004709"
       "nid004953"
[29]: 1 pids
[29]: 2-element Vector{Any}:
       1605084
       1781811
[ ]: 1
```

Get hostnames
and pids from
each worker

Cluster Managers: Interaction with Slurm

The screenshot shows a JupyterLab interface titled "SC23.ipynb" running in a "Julia 1.8.0-beta1 GPU" kernel. The code cell [25] imports the "Distributed, ClusterManagers" module. Cell [26] adds two workers using the "SlurmManager" with constraints and QoS settings. The output shows the connection to workers and the submission of a job to the Slurm queue. Cells [28] and [29] demonstrate querying the hosts and pids respectively.

```
File Edit View Run Kernel Git Tabs Settings Help
SC23.ipynb (auto-x) - JupyterLab
Julia 1.8.0-beta1 GPU
[25]: 1 using Distributed, ClusterManagers
[26]: 1 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
[26]: 2-element Vector{Int64}:
      2
      3
[28]: 1 hosts
[28]: 2-element Vector{Any}:
      "nid004709"
      "nid004953"
[29]: 1 pids
[29]: 2-element Vector{Any}:
      1605084
      1781811
```

Distributed.jl supports basic workflows

Basic task: count number of heads from 2×10^8 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)

```
1 function n_heads()
2     n = 0
3     for i = 1:200000000
4         n += Int(rand(Bool))
5     end
6     n
7 end
8
9 function n_heads_parallel()
10    nheads = @distributed (+) for i = 1:200000000
11        Int(rand(Bool))
12    end
13    nheads
14 end
```

Performance Gains from Distributing Work

SC23.ipynb (auto-x) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

Julia 1.8.0-beta1 GPU

```
[33]: 1 using BenchmarkTools
```

```
[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.
```

```
[32]: 1 @benchmark(n_heads_parallel())
```

```
[32]: 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 | GC (median): 0.00%
Time (mean ± σ): 99.871 ms ± 106.362 μs | GC (mean ± σ): 0.00% ± 0.00%
```

```
Memory estimate: 6.62 KiB, allocs estimate: 155.
```

```
[ ]: 1
```

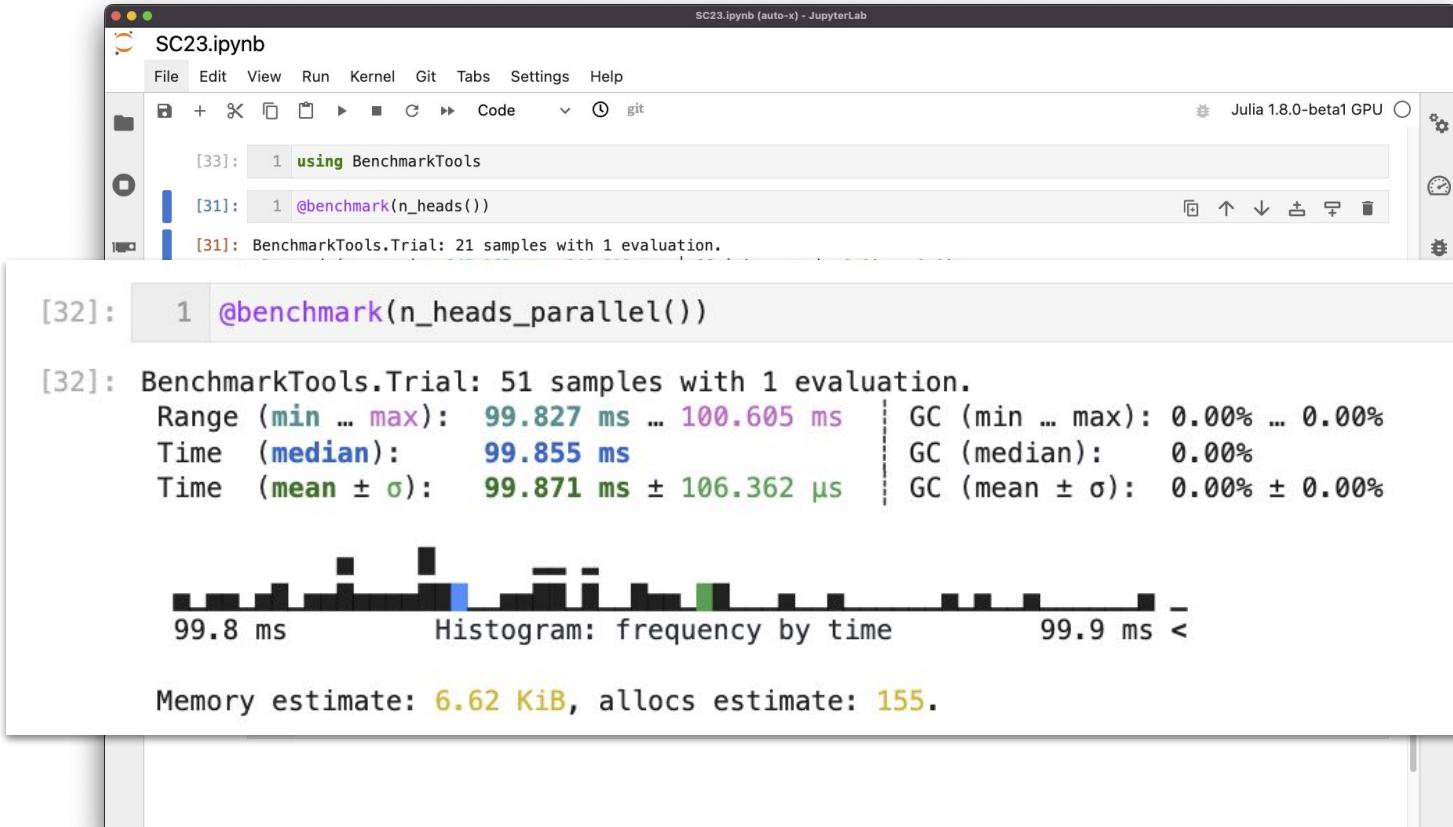
Performance Gains from Distributing Work

```
SC23.ipynb (auto-x) - JupyterLab
File Edit View Run Kernel Git Tabs Settings Help
[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%
Histogram: frequency by time
246 ms           Histogram: frequency by time           247 ms <
Memory estimate: 0 bytes, allocs estimate: 0.

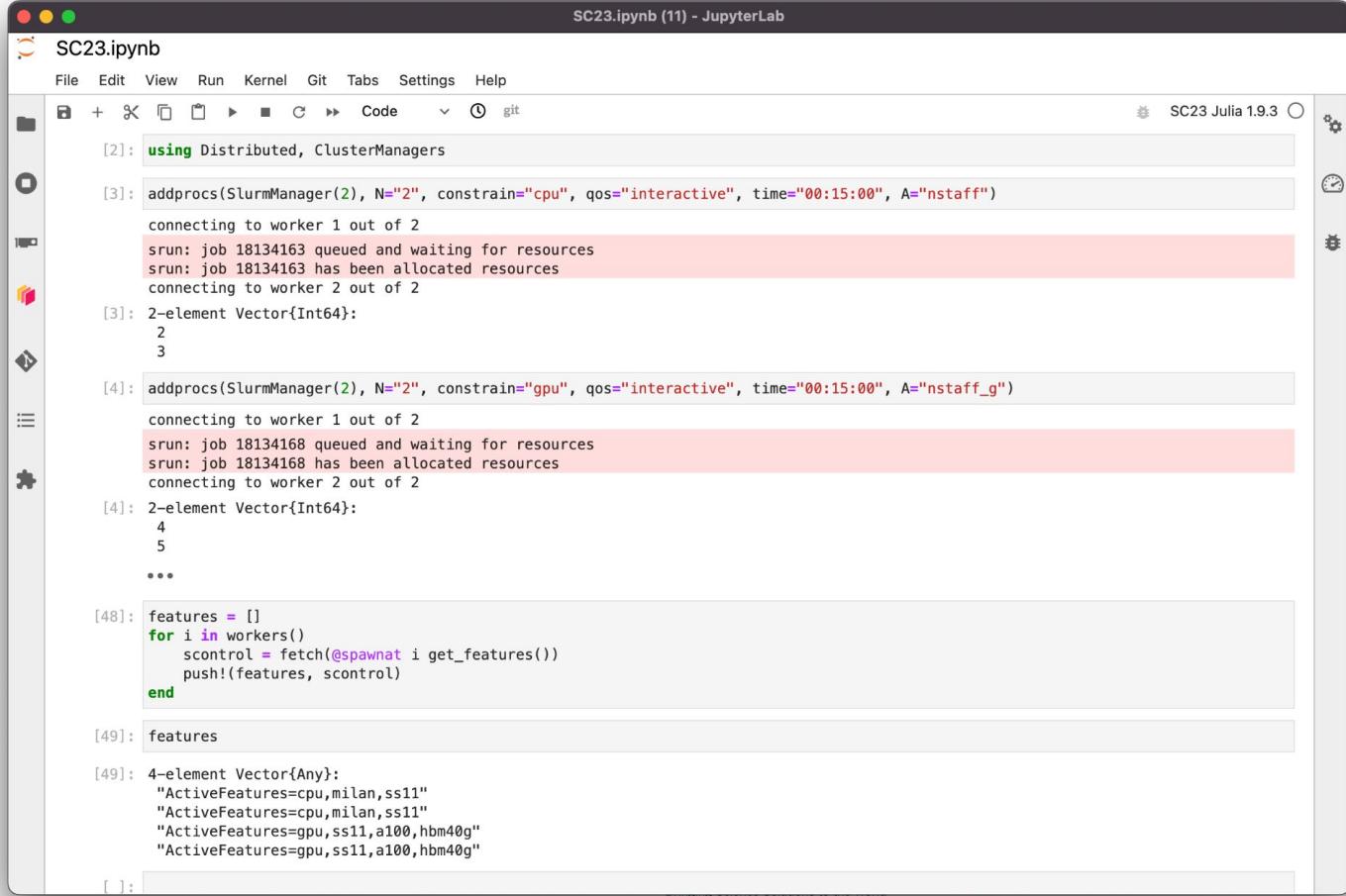
Time (mean ± σ): 99.871 ms ± 106.362 μs | GC (mean ± σ): 0.00% ± 0.00%
Histogram: frequency by time
99.8 ms           Histogram: frequency by time           99.9 ms <
Memory estimate: 6.62 KiB, allocs estimate: 155.
[ ]: 1
```

Performance Gains from Distributing Work

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



Tangent: Hybrid CPU/GPU Jobs



The screenshot shows a JupyterLab interface titled "SC23.ipynb (11) - JupyterLab". The code cell [2] imports the Distributed and ClusterManagers modules. Cells [3] and [4] demonstrate adding two workers (one CPU, one GPU) using the SlurmManager. Both workers are constrained to the "cpu" node and have a qos of "interactive" with a time limit of "00:15:00". The GPU worker is specifically labeled "A=nstaff_g". The output for both workers shows they are queued and waiting for resources, then allocated resources, and finally connected to the workers. The code in cell [48] fetches features from each worker using scontrol. The resulting features vector contains four entries: "ActiveFeatures=cpu,milan,ss11", "ActiveFeatures=cpu,milan,ss11", "ActiveFeatures=gpu,ss11,a100,hbm40g", and "ActiveFeatures=gpu,ss11,a100,hbm40g".

```
using Distributed, ClusterManagers

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

2-element Vector{Int64}:
 2
 3

addprocs(SlurmManager(2), N="2", constrain="gpu", qos="interactive", time="00:15:00", A="nstaff_g")
connecting to worker 1 out of 2
srun: job 18134168 queued and waiting for resources
srun: job 18134168 has been allocated resources
connecting to worker 2 out of 2

2-element Vector{Int64}:
 4
 5

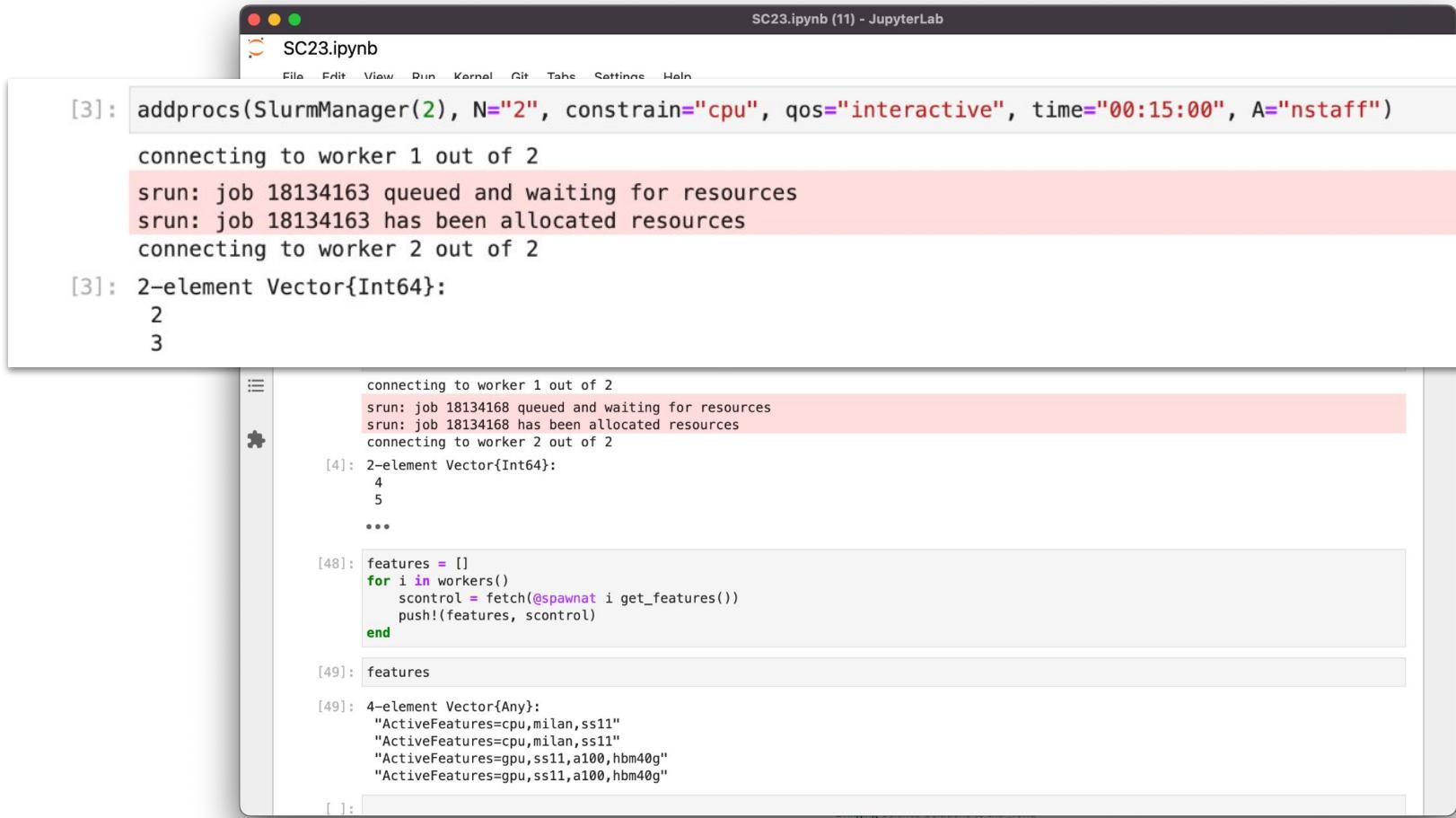
***

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



The screenshot shows a JupyterLab interface with a tab titled "SC23.ipynb (11) - JupyterLab". The code cell [3] contains the command `addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", A="nstaff")`. The output shows the process of connecting to two workers and the submission of a job to Slurm. The code cell [3] also displays a 2-element Vector{Int64} containing the values 2 and 3.

The code cell [4] shows the continuation of the job submission process, connecting to worker 2 and displaying a 2-element Vector{Int64} with values 4 and 5. It also includes an ellipsis (...).

The code cell [48] contains a loop that iterates over workers, fetches their features using scontrol, and pushes them into a vector named "features".

The code cell [49] displays the resulting "features" vector, which is a 4-element Vector{Any} containing four strings: "ActiveFeatures=cpu,milan,ss11", "ActiveFeatures=cpu,milan,ss11", "ActiveFeatures=gpu,ss11,a100,hbm40g", and "ActiveFeatures=gpu,ss11,a100,hbm40g".

Tangent: Hybrid CPU/GPU Jobs

The screenshot shows a JupyterLab interface titled "SC23.ipynb (11) - JupyterLab". The top menu bar includes File, Edit, View, Run, Kernel, Git, Tabs, Settings, and Help. A toolbar below the menu contains icons for file operations like Open, Save, and Run, along with a "Code" dropdown and a git status indicator. The main area displays a code cell [2] and its output:

```
[2]: using Distributed, ClusterManagers
```

Output:

```
connecting to worker 1 out of 2
srun: job 18134163 queued and waiting for resources
```

Cell [4] is currently being run:

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

Output:

```
connecting to worker 1 out of 2
srun: job 18134168 queued and waiting for resources
srun: job 18134168 has been allocated resources
connecting to worker 2 out of 2
```

Cell [4] has run and produced the following output:

```
[4]: 2-element Vector{Int64}:
 4
 5
```

Cells [48] and [49] are also visible in the bottom panel:

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

```
[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"
```

The bottom right corner of the interface shows the text "Running on SC23 Julia 1.9.3".

Tangent: Hybrid CPU/GPU Jobs

The screenshot shows a JupyterLab interface titled "SC23.ipynb (11) - JupyterLab". The code cell [2] contains:

```
[2]: using Distributed, ClusterManagers
```

Cell [3] shows the output of adding two workers:

```
[3]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", A="nstaff")  
connecting to worker 1 out of 2  
srun: job 18134163 queued and waiting for resources  
srun: job 18134163 has been allocated resources  
connecting to worker 2 out of 2
```

Cell [3] also displays a 2-element Vector{Int64} with values 2 and 3.

Cell [4] shows the output of adding two GPU workers:

```
[4]: addprocs(SlurmManager(2), N="2", constrain="gpu", qos="interactive", time="00:15:00", A="nstaff_g")  
connecting to worker 1 out of 2  
srun: job 18134168 queued and waiting for resources  
srun: job 18134168 has been allocated resources  
connecting to worker 2 out of 2
```

A callout box highlights the code in cell [48]:

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

Cell [49] shows the resulting 4-element Vector{Any}:

```
[49]: 4-element Vector{Any}:  
"ActiveFeatures=cpu,milan,ss11"  
"ActiveFeatures=cpu,milan,ss11"  
"ActiveFeatures=gpu,ss11,a100,hbm40g"  
"ActiveFeatures=gpu,ss11,a100,hbm40g"
```

Cell [] is partially visible at the bottom.

Tangent: Hybrid CPU/GPU Jobs

The screenshot shows a JupyterLab interface titled "SC23.ipynb (11) - JupyterLab". The code cell [2] contains the command `using Distributed, ClusterManagers`. Cells [3] and [4] show the execution of `addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", A="nstaff")` and `addprocs(SlurmManager(2), N="2", constrain="gpu", qos="interactive", time="00:15:00", A="nstaff_g")` respectively. Both commands output resource allocation details from Slurm, indicating jobs 18134163 and 18134168 are queued and then allocated resources. The output for cell [4] continues with "connecting to worker 2 out of 2", "2-element Vector{Int64}: 4 5", and "...".

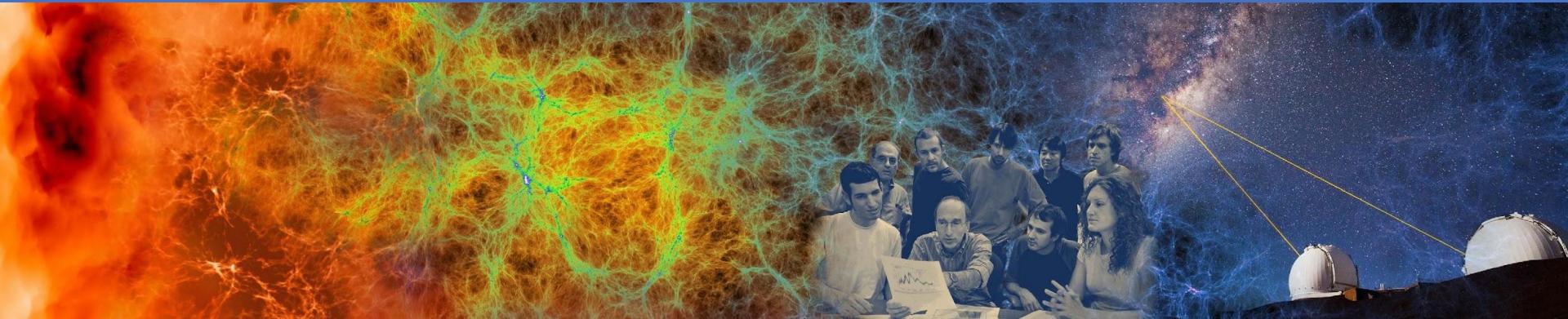
```
[2]: using Distributed, ClusterManagers
[3]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", A="nstaff")
      connecting to worker 1 out of 2
      srun: job 18134163 queued and waiting for resources
      srun: job 18134163 has been allocated resources
      connecting to worker 2 out of 2
[3]: 2-element Vector{Int64}:
      2
      3
[4]: addprocs(SlurmManager(2), N="2", constrain="gpu", qos="interactive", time="00:15:00", A="nstaff_g")
      connecting to worker 1 out of 2
      srun: job 18134168 queued and waiting for resources
      srun: job 18134168 has been allocated resources
      connecting to worker 2 out of 2
[4]: 2-element Vector{Int64}:
      4
      5
      ...

```

```
[49]: features
```

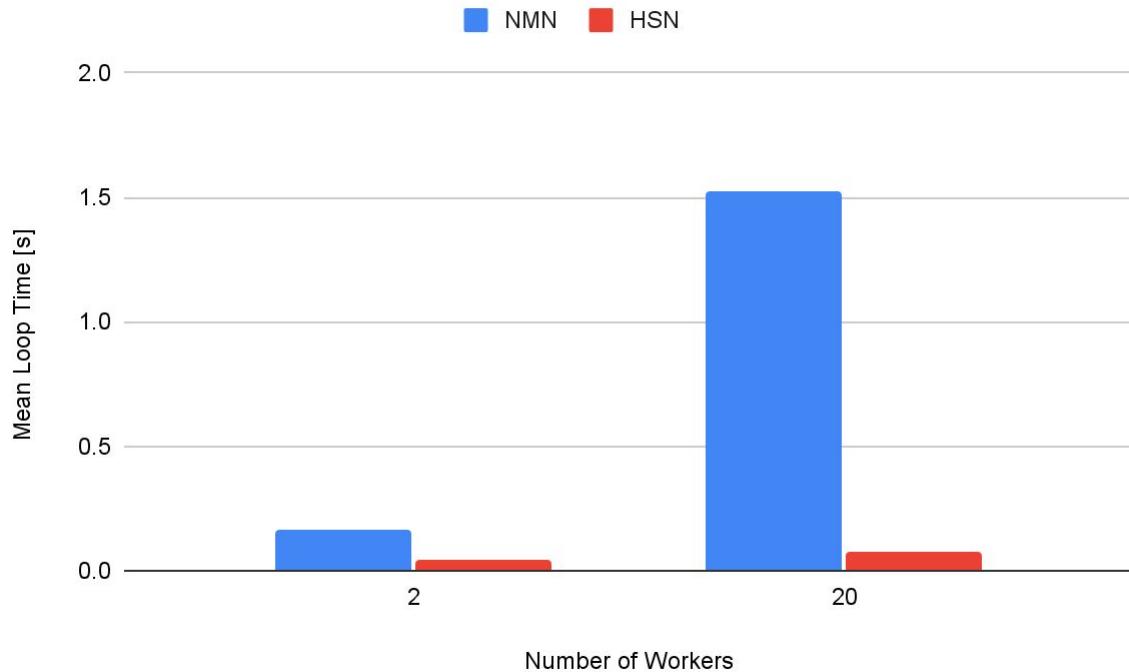
```
[49]: 4-element Vector{Any}:
      "ActiveFeatures=cpu,milan,ss11"
      "ActiveFeatures=cpu,milan,ss11"
      "ActiveFeatures=gpu,ss11,a100,hbm40g"
      "ActiveFeatures=gpu,ss11,a100,hbm40g"
```

Tangent: Network Discovery



Workflow Support Story

- Unexpected poor performance and scaling
- User application 100x slower on Perlmutter

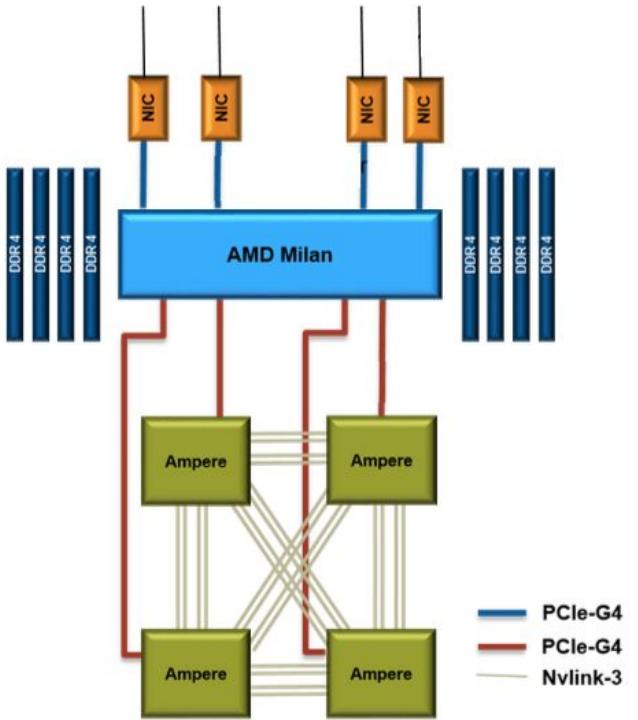


Perlmutter is a Heterogeneous System

Partition	Nodes	CPU	RAM	GPU	NIC
GPU	1536	1x AMD EPYC 7763	256GB	4x NVIDIA A100 (40GB)	4x HPE Slingshot 11
	256	1x AMD EPYC 7763	256GB	4x NVIDIA A100 (80GB)	4x HPE Slingshot 11
CPU	3072	2x AMD EPYC 7763	512GB	–	1x HPE Slingshot 11
Login	40	1x AMD EPYC 7713	512GB	4x NVIDIA A100 (40GB)	–
Large Memory	4	1x AMD EPYC 7713	1TB	4x NVIDIA A100 (40GB)	1x HPE Slingshot 11

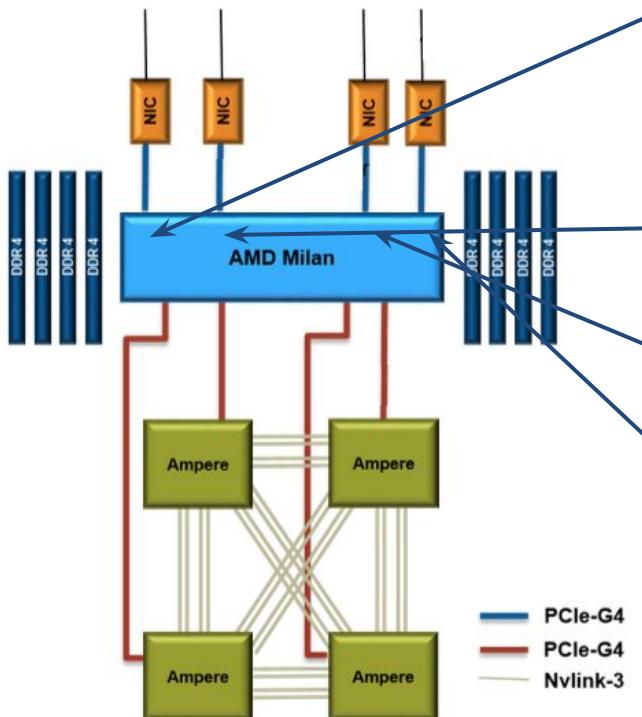
- Each GPU node has 4 NICs
 - 1 NIC and 1 GPU per host bridge
- Each CPU node has 1 NIC

Eg. GPU Node Topology



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

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"]
```

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



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

```
213: Hwloc.Object: PCI_Device [c2:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn0"]
213: Hwloc.Object: PCI_Device [c3:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "nmn0"]
47: Hwloc.Object: PCI_Device [81:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn1"]
379: Hwloc.Object: PCI_Device [42:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn2"]
379: Hwloc.Object: PCI_Device [01:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn3"]
```

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

This one!



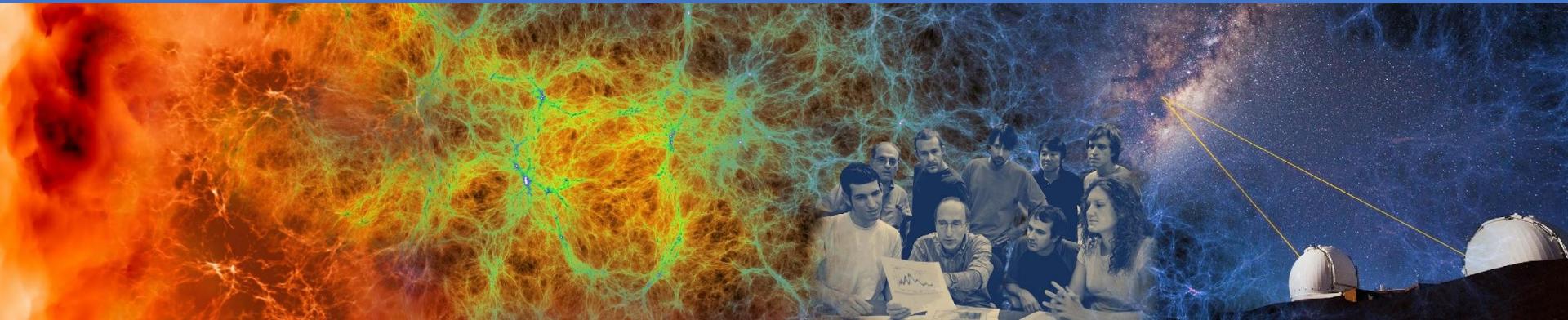
```
for net in collect(network_devs)
    found, dist = distance_to_core(hwloc_tree, net, cpu_id)
    print("${dist}: ")
    print_tree(net)
end
```

```
213: Hwloc.Object: PCI_Device [c2:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn0"]
213: Hwloc.Object: PCI_Device [c3:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "nmn0"]
47: Hwloc.Object: PCI_Device [81:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn1"]
379: Hwloc.Object: PCI_Device [42:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn2"]
379: Hwloc.Object: PCI_Device [01:00.0 (Ethernet)]
  └ Hwloc.Object: OS_Device [Net "hsn3"]
```

Future

- This is pre-alpha, relies on some non-merged PRs
 - Looking for folks to test this on their favorite HPC systems
- Distributed.jl to use distance between NIC and Core (on Hwloc tree) to select preferred tree NIC
 - [JuliaParallel/NetworkInterfaceControllers.jl](#)

Programming GPUs



CUDA.jl: Interfacing with Nvidia GPUs

SC23.ipynb (auto-L) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

SC23 Julia 1.9.3

```
[9]: arr1 = rand(1_000, 1_000);
arr2 = rand(1_000, 1_000);

[10]: @benchmark arr1 * arr2

[10]: 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%
```



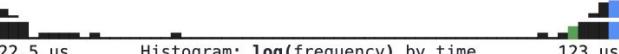
```
Memory estimate: 7.63 MiB, allocs estimate: 2.
```

```
[11]: using CUDA

[12]: carr1 = cu(arr1);
carr2 = cu(arr2);

[13]: @benchmark carr1 * carr2

[13]: 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): 121.205 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

Basic example:
1000x1000 matmul
using OpenBLAS

```
[9]: arr1 = rand(1_000, 1_000);
      arr2 = rand(1_000, 1_000);

10]: @benchmark arr1 * arr2

10]: 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%
```

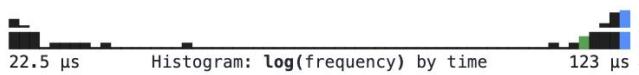


Memory estimate: 7.63 MiB, allocs estimate: 2.

```
[12]: carr1 = cu(arr1);
       carr2 = cu(arr2);

[13]: @benchmark carr1 * carr2

[13]: 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): 121.205 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

SC23.ipynb (auto-L) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

SC23 Julia 1.9.3

```
[9]: arr1 = rand(1_000, 1_000);
arr2 = rand(1_000, 1_000);

[10]: @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%

```

2.63 ms Histogram: frequency by time 4.16 ms <

```
[11]: using CUDA
```

```
[12]: carr1 = cu(arr1);
carr2 = cu(arr2);
```

```
[13]: 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): 121.205 μs | GC (median): 0.00%
Time (mean ± σ): 114.948 μs ± 24.119 μs | GC (mean ± σ): 0.00% ± 0.00%

```

22.5 μs Histogram: log(frequency) by time 123 μs <

Memory estimate: 1.09 KiB, allocs estimate: 44.

Copy arrays to device

CUDA.jl: Interfacing with Nvidia GPUs

```
SC23.ipynb (auto-L) - JupyterLab
SC23.ipynb

File Edit View Run Kernel Git Tabs Settings Help
git
SC23 Julia 1.9.3

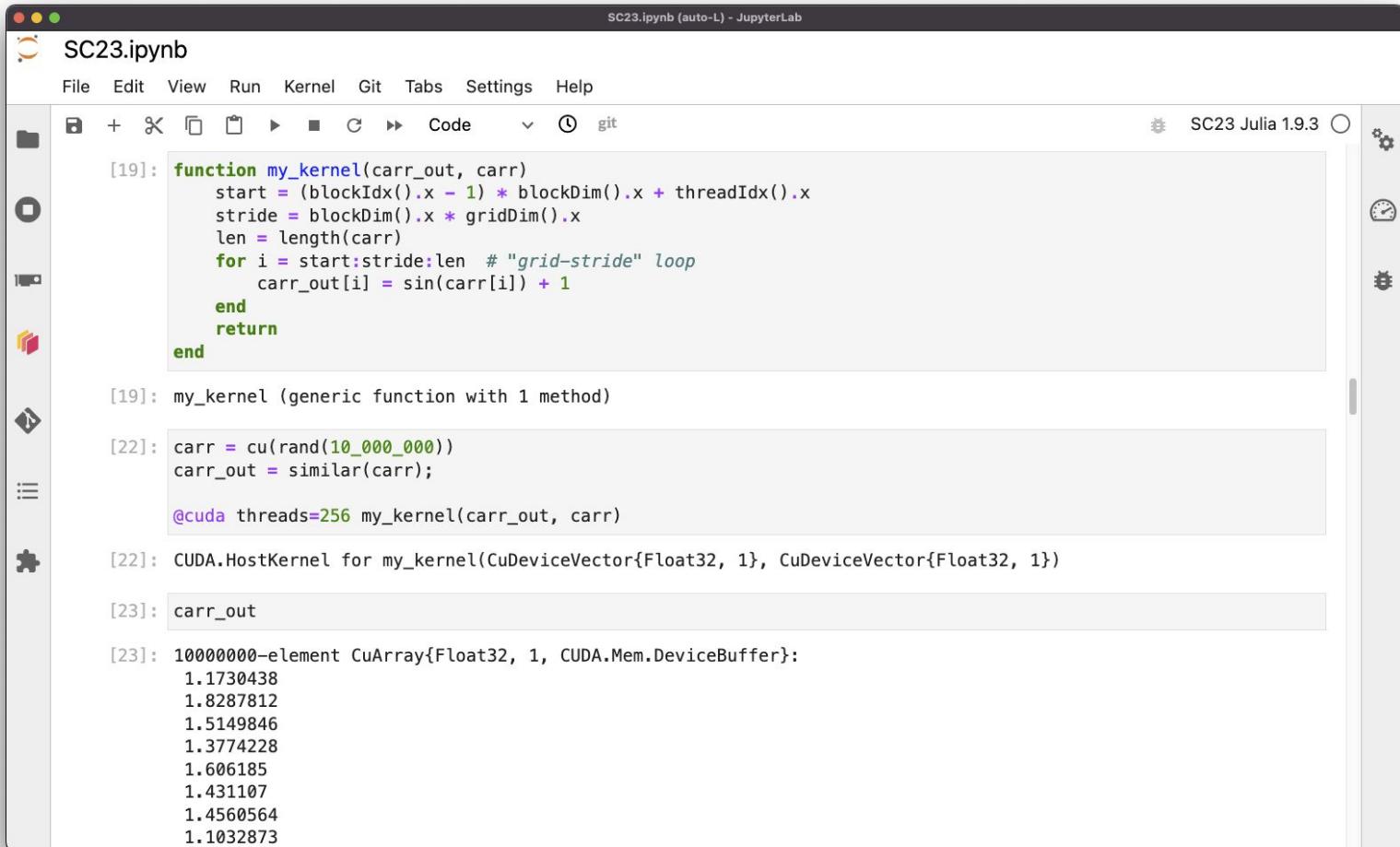
[9]: arr1 = rand(1_000, 1_000);
arr2 = rand(1_000, 1_000);

[10]: @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%
Histogram: frequency by time
2.63 ms      Histogram: frequency by time      4.16 ms <
Memory estimate: 7.63 MiB, allocs estimate: 2.

[13]: @benchmark carr1 * carr2
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): 121.205 μs | GC (median): 0.00%
Time (mean ± σ): 114.948 μs ± 24.119 μs | GC (mean ± σ): 0.00% ± 0.00%
Histogram: log(frequency) by time
22.5 μs      Histogram: log(frequency) by time      123 μs <
Memory estimate: 1.09 KiB, allocs estimate: 44.
```

Using cuBLAS to perform matmul decreases run time from 3.15ms to 121μs (26x)

Write Your Own CUDA Kernels in Julia



The screenshot shows a JupyterLab interface with a notebook titled "SC23.ipynb". The code cell [19] contains a Julia function named `my_kernel` that performs a grid-stride loop to calculate the sine of each element in a array and adds 1. The code cell [22] creates a CUDA array `carr`, initializes `carr_out`, and runs the `my_kernel` function with `@cuda threads=256`. The code cell [23] displays the resulting `carr_out` array.

```
[19]: 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

[19]: my_kernel (generic function with 1 method)

[22]: carr = cu(rand(10_000_000))
carr_out = similar(carr);

@cuda threads=256 my_kernel(carr_out, carr)

[22]: CUDA.HostKernel for my_kernel(CuDeviceVector{Float32, 1}, CuDeviceVector{Float32, 1})

[23]: carr_out

[23]: 1000000-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 1.1730438
 1.8287812
 1.5149846
 1.3774228
 1.606185
 1.431107
 1.4560564
 1.1032873
```

Write Your Own CUDA Kernels in Julia

Define kernels using
Julia functions

[19]:

```
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
```

[22]:

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

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

[22]: CUDA.HostKernel for my_kernel(CuDeviceVector{Float32, 1}, CuDeviceVector{Float32, 1})

[23]:

```
carr_out
```

[23]: 1000000-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:

1.1730438
1.8287812
1.5149846
1.3774228
1.606185
1.431107
1.4560564
1.1032873

Write Your Own CUDA Kernels in Julia

The screenshot shows a JupyterLab interface with a notebook titled "SC23.ipynb". The code editor contains the following Julia code:

```
[19]: 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

[19]: my_kernel (generic function with 1 method)
```

In cell [22], the user has typed:

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

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

The output of cell [22] is:

```
[22]: CUDA.HostKernel for my_kernel(CuDeviceVector{Float32, 1}, CuDeviceVector{Float32, 1})
```

The output of cell [23] is:

```
[23]: 1000000-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 1.1730438
 1.8287812
 1.5149846
 1.3774228
 1.606185
 1.431107
 1.4560564
 1.1032873
```

Launch kernel using the
@cuda macro

(advanced) LLVM + Julia

Julia provides interfaces to the LLVM backend.

Eg.:

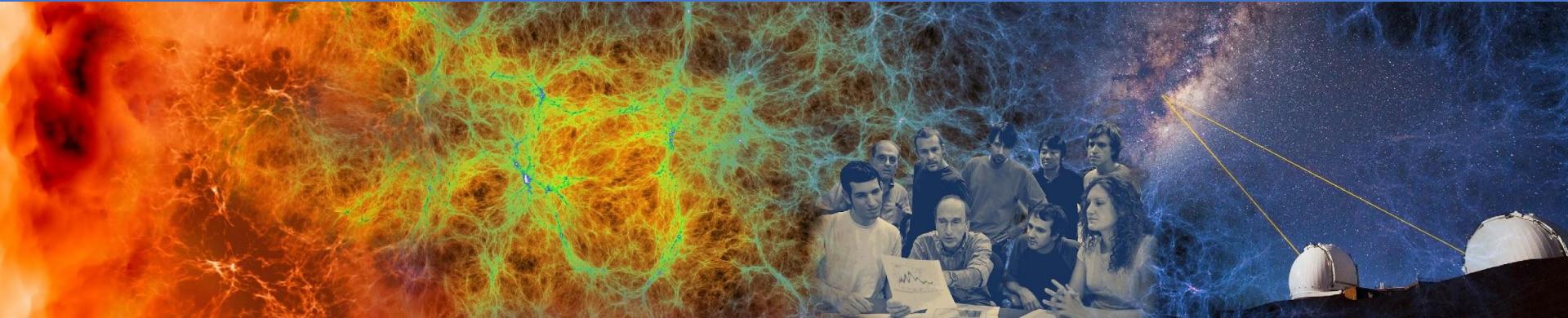
- loopinfo
- llvmlcall

```
[16]: macro unroll(expr)
    expr = loopinfo("@unroll", expr, (Symbol("llvm.loop.unroll.full"),))
    return esc(expr)
end

for (jlf, f) in zip(:+, :*, :-, (:add, :mul, :sub))
    for (T, llvmT) in ((:Float32, "float"), (:Float64, "double"))
        ir = """
            %x = f$f contract nsz $llvmT %0, %1
            ret $llvmT %x
        """
        @eval begin
            # the @pure is necessary so that we can constant propagate.
            @inline Base.@pure function $jlf(a::$T, b::$T)
                Base.llvmlcall($ir, $T, Tuple{$T, $T}, a, b)
            end
        end
    end
    @eval function $jlf(args...)
        Base.$jlf(args...)
    end
end
```



Distributed.jl is nice, but...



Dagger.jl: Easy work distribution

SC23.ipynb (11) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

SC23 Julia 1.9.3

```
[1]: using Distributed, ClusterManagers, BenchmarkTools

[2]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", 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]: 2-element Vector{Int64}:
      2
      3

[3]: @everywhere using Dagger

[11]: darr1 = rand(Blocks(500, 500), 1_000, 1_000);
      darr2 = rand(Blocks(500, 500), 1_000, 1_000);

[12]: function test_mul()
      darr1 * darr2
      wait;
end

[12]: test_mul (generic function with 1 method)

[13]: @benchmark(test_mul())

[13]: 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%
```

Histogram: frequency by time

Memory estimate: 167.05 KiB, allocs estimate: 3113.

Dagger.jl: Easy work distribution

SC23.ipynb (11) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

SC23 Julia 1.9.3

```
[1]: using Distributed, ClusterManagers, BenchmarkTools
```

```
[2]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", 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
```

```
[3]: @everywhere using Dagger
```

```
[11]: darr1 = rand(Blocks(500, 500), 1_000, 1_000);
darr2 = rand(Blocks(500, 500), 1_000, 1_000);

    darr1 * darr2
    wait;
end
```

```
[12]: test_mul (generic function with 1 method)
```

```
[13]: @benchmark(test_mul())
```

```
[13]: 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%
```

Memory estimate: 167.05 KiB, allocs estimate: 3113.

Dagger.jl: Easy work distribution

SC23.ipynb (11) - JupyterLab

File Edit View Run Kernel Git Tabs Settings Help

SC23 Julia 1.9.3

```
[1]: using Distributed, ClusterManagers, BenchmarkTools

[2]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", 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]: 2-element Vector{Int64}:
      2
      3

[3]: @everywhere using Dagger

[11]: darr1 = rand(Blocks(500, 500), 1_000, 1_000);

[12]: function test_mul()
      darr1 * darr2
      wait;
    end

[13]: 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%
      Histogram: frequency by time
      341 μs   676 μs <
      Memory estimate: 167.05 KiB, allocs estimate: 3113.
```

Dagger.jl: Easy work distribution

The screenshot shows a JupyterLab interface titled "SC23.ipynb (11) - JupyterLab". The code cell [1] imports `Distributed`, `ClusterManagers`, and `BenchmarkTools`. Cell [2] adds workers using `addprocs` with SlurmManager, specifying 2 workers, CPU constraint, interactive QoS, and a time limit of 15 minutes. It also creates a 2-element vector of Int64 values [2, 3]. Cell [3] uses `@everywhere` to import `Dagger`. Cell [11] generates two random matrices, `darr1` and `darr2`, both 500x500 with elements ranging from 1_000 to 1_000. The code in cell [13] is annotated with `@benchmark test_mul()`.

```
[1]: using Distributed, ClusterManagers, BenchmarkTools
[2]: addprocs(SlurmManager(2), N="2", constrain="cpu", qos="interactive", time="00:15:00", 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]: 2-element Vector{Int64}:
      2
      3
[3]: @everywhere using Dagger
[11]: darr1 = rand(Blocks(500, 500), 1_000, 1_000);
      darr2 = rand(Blocks(500, 500), 1_000, 1_000);

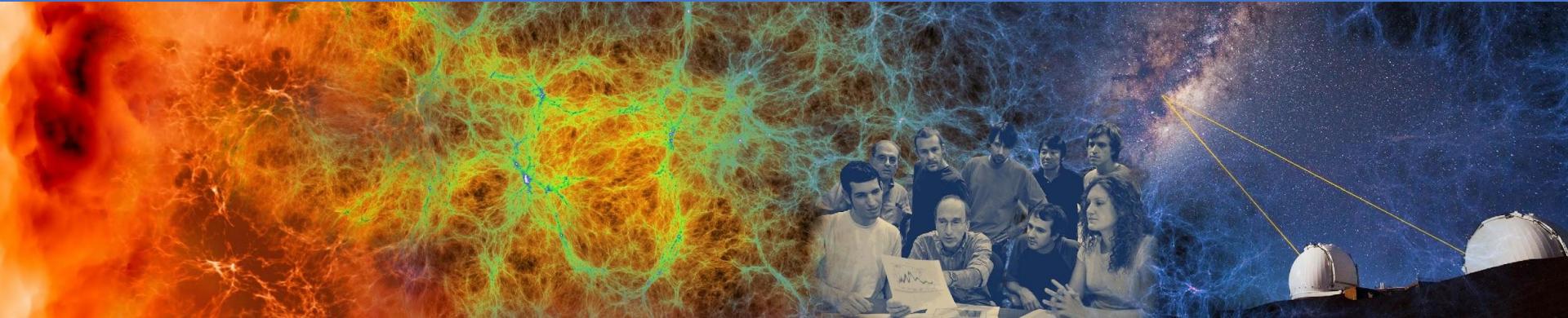
[13]: @benchmark test_mul()
```

```
[13]: 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%
```



For references: openBLAS
(single node) = 3.15ms,
cuBLAS = 121μs

Conclusions



Conclusion

- Julia provides a rich ecosystem to build performant distributed applications on HPC systems
- Modern high-productivity design
- Built on top of LLVM, with vendor backends (CUDA.jl, AMDGPU.jl, oneAPI.jl, etc)
- Provides interfaces to examine and manipulate LLVM IR



Extra Slides

Perlmutter system configuration

NVIDIA "Ampere" GPU Nodes

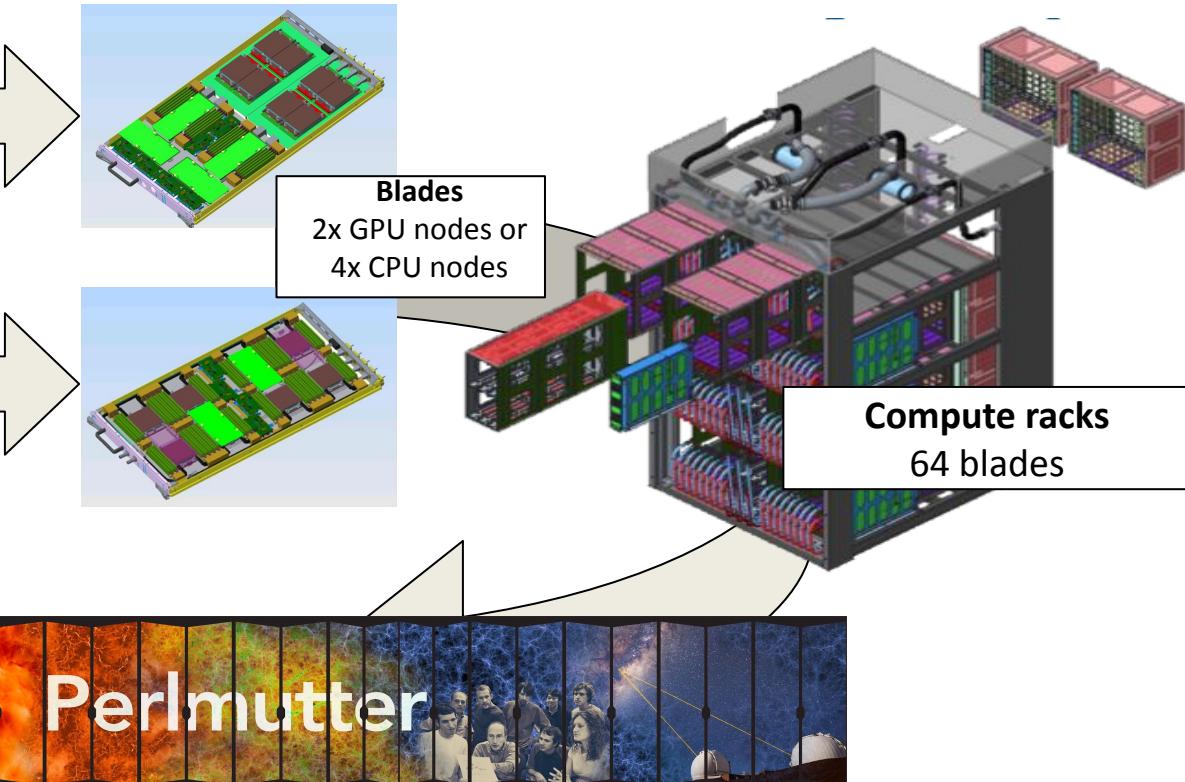
4x GPU + 1x CPU
40 GiB HBM + 256 GiB DDR
4x 200G "Slingshot" NICs

AMD "Milan" CPU Node

2x CPUs
> 256 GiB DDR4
1x 200G "Slingshot" NIC

Centers of Excellence
Network
Storage
App. Readiness
System SW

Perlmutter system
GPU racks
CPU racks
~6 MW



CUDA.jl provides detailed profiling interface

The screenshot shows a JupyterLab interface with the title "SC23.ipynb (auto-L) - JupyterLab". In the code editor, the following command was run:

```
[14]: CUDA.@profile carr1 * carr2
```

The output indicates that the profiler ran for 2.23 ms, capturing 23 events.

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

Time (%)	Time	Calls	Avg time	Min time	Max time	Name	...
29.29%	653.51 µs	1	653.51 µs	653.51 µs	653.51 µs	cuMemAlloc	...
18.15%	404.83 µs	1	404.83 µs	404.83 µs	404.83 µs	cudaEvent	...
1.12%	25.03 µs	1	25.03 µs	25.03 µs	25.03 µs	cudaLaunch	...
0.82%	18.36 µs	1	18.36 µs	18.36 µs	18.36 µs	cudaMemcpy	...
0.44%	9.78 µs	2	4.89 µs	953.67 ns	8.82 µs	cudaOccupiedGrid	...
0.28%	6.2 µs	3	2.07 µs	476.84 ns	4.77 µs	cudaStreamCreate	...
0.26%	5.72 µs	1	5.72 µs	5.72 µs	5.72 µs	cudaEventDestroy	...
0.00%	0.0 ns	1	0.0 ns	0.0 ns	0.0 ns	cudaGetLastError	...

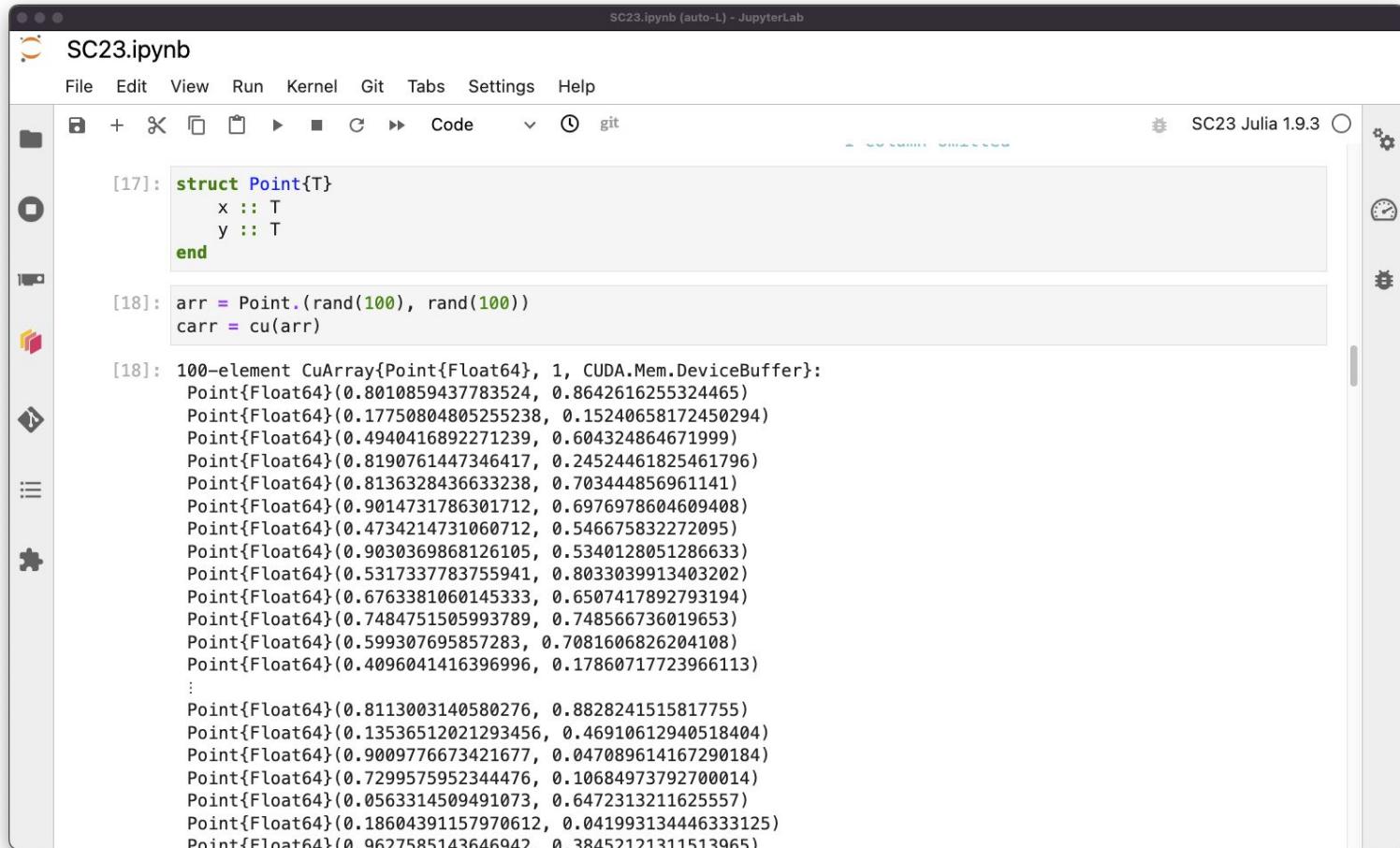
1 column omitted

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

Time (%)	Time	Calls	Avg time	Min time	Max time	Name	...
7.09%	158.07 µs	1	158.07 µs	158.07 µs	158.07 µs	ampere_sg	...
0.10%	2.15 µs	1	2.15 µs	2.15 µs	2.15 µs	[set devic...]	...

1 column omitted

CUDA.jl is compatible with Structs



The screenshot shows a JupyterLab interface with the title "SC23.ipynb". The code cell [17] defines a struct Point{T} with fields x and y. The code cell [18] creates an array arr of Point{Float64} objects and converts it to a CuArray. The output of cell [18] is a 100-element CuArray of Point{Float64} objects.

```
struct Point{T}
    x :: T
    y :: T
end

arr = Point.(rand(100), rand(100))
carr = cu(arr)

100-element CuArray{Point{Float64}, 1, CUDA.Mem.DeviceBuffer}:
 Point{Float64}(0.8010859437783524, 0.8642616255324465)
 Point{Float64}(0.17750804805255238, 0.15240658172450294)
 Point{Float64}(0.4940416892271239, 0.604324864671999)
 Point{Float64}(0.8190761447346417, 0.24524461825461796)
 Point{Float64}(0.8136328436633238, 0.703444856961141)
 Point{Float64}(0.9014731786301712, 0.6976978604609408)
 Point{Float64}(0.4734214731060712, 0.546675832272095)
 Point{Float64}(0.9030369868126105, 0.5340128051286633)
 Point{Float64}(0.5317337783755941, 0.8033039913403202)
 Point{Float64}(0.6763381060145333, 0.6507417892793194)
 Point{Float64}(0.7484751505993789, 0.748566736019653)
 Point{Float64}(0.599307695857283, 0.7081606826204108)
 Point{Float64}(0.4096041416396996, 0.17860717723966113)
 :
 Point{Float64}(0.8113003140580276, 0.8828241515817755)
 Point{Float64}(0.13536512021293456, 0.46910612940518404)
 Point{Float64}(0.9009776673421677, 0.047089614167290184)
 Point{Float64}(0.7299575952344476, 0.10684973792700014)
 Point{Float64}(0.0563314509491073, 0.6472313211625557)
 Point{Float64}(0.18604391157970612, 0.041993134446333125)
 Point{Float64}(0.9627585143646942, 0.38452121311513965)
```

CUDA.jl is compatible with Structs

Julia converts struct to
cuda-compatible type

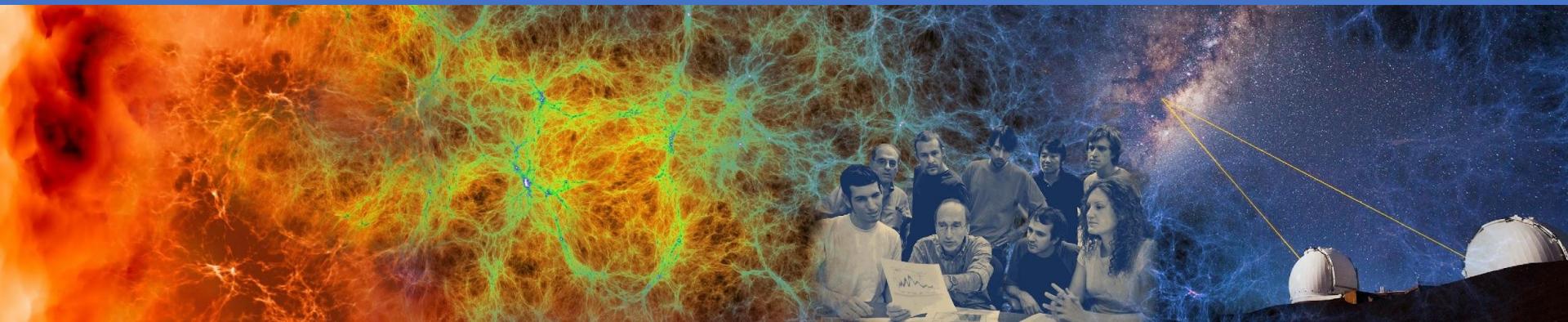
```
[17]: struct Point{T}
    x :: T
    y :: T
end

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

The screenshot shows a JupyterLab interface with a notebook titled "SC23.ipynb". In cell [17], a Point struct is defined with fields x and y of type T. In cell [18], an array arr is created by applying the Point constructor to two arrays of 100 random numbers, and then carr is created as a CUDA (cu) version of arr. Below the code, the first 20 elements of the carr array are listed, all of type Point{Float64}.

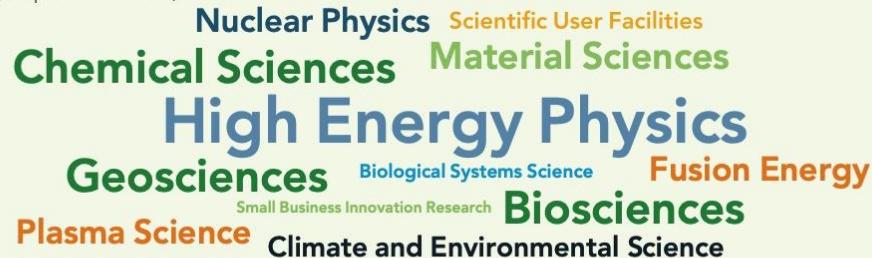
```
Point{Float64}(0.801085943783524, 0.8642616255324465)
Point{Float64}(0.17750804805255238, 0.15240658172450294)
Point{Float64}(0.4940416892271239, 0.604324864671999)
Point{Float64}(0.8190761447346417, 0.24524461825461796)
Point{Float64}(0.8136328436633238, 0.703444856961141)
Point{Float64}(0.9014731786301712, 0.6976978604609408)
Point{Float64}(0.4734214731060712, 0.546675832272095)
Point{Float64}(0.9030369868126105, 0.5340128051286633)
Point{Float64}(0.5317337783755941, 0.8033039913403202)
Point{Float64}(0.6763381060145333, 0.6507417892793194)
Point{Float64}(0.7484751505993789, 0.748566736019653)
Point{Float64}(0.599307695857283, 0.7081606826204108)
Point{Float64}(0.4096041416396996, 0.17860717723966113)
:
Point{Float64}(0.8113003140580276, 0.8828241515817755)
Point{Float64}(0.13536512021293456, 0.46910612940518404)
Point{Float64}(0.9009776673421677, 0.047089614167290184)
Point{Float64}(0.7299575952344476, 0.10684973792700014)
Point{Float64}(0.0563314509491073, 0.6472313211625557)
Point{Float64}(0.18604391157970612, 0.041993134446333125)
Point{Float64}(0.9627585143646942, 0.38452121311513965)
```

Why does NERSC care about Julia?



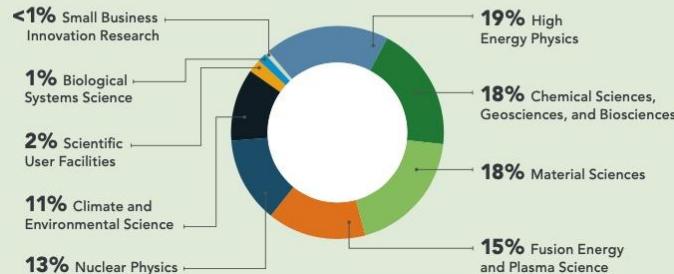
NERSC is the mission HPC and data facility for the U.S Department of Energy Office of Science

Top Science Disciplines (By computational hours used)



~1,000 Projects

Breakdown of Compute Used by DOE Program



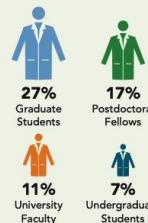
9

2021 NERSC USERS ACROSS US AND WORLD

50 States + Washington D.C. and Puerto Rico
46 Countries



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



>2,000

Scientific Journal Articles per Year



62



BERKELEY LAB
Bringing Science Solutions to the World



U.S. DEPARTMENT OF
ENERGY

Office of
Science

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)

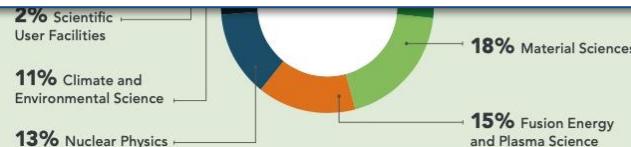
Chem

Geo

Plasma S



- 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

NERSC BY THE NUMBERS

