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

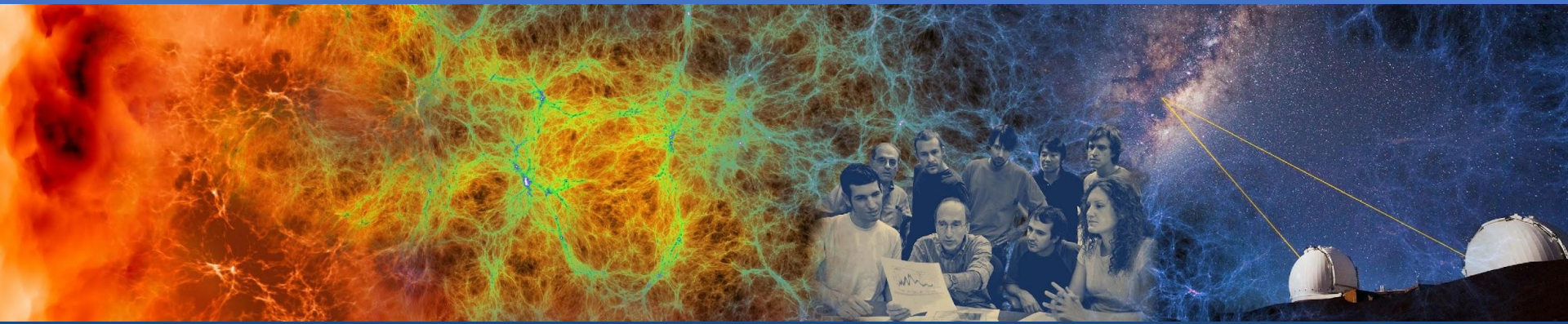
None of this would be possible without:

- 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

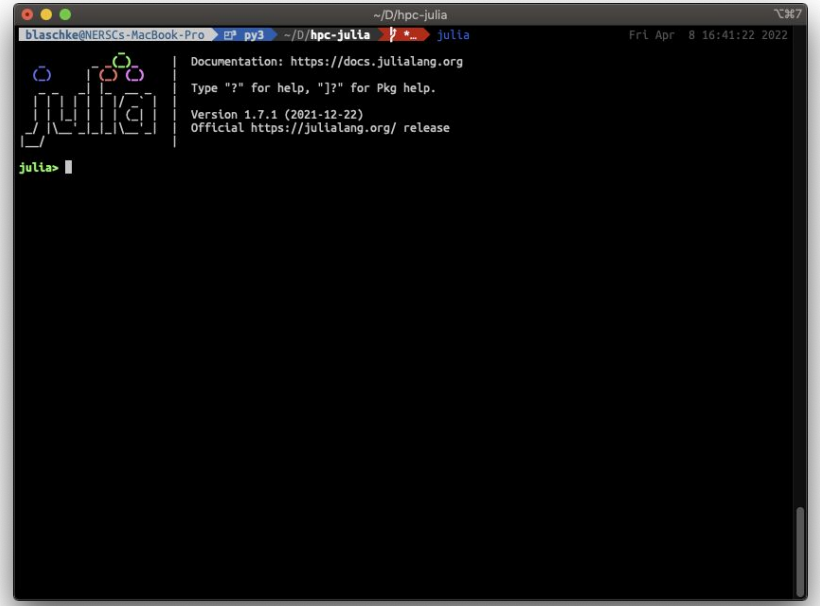


Julia is a High-Productivity Language

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

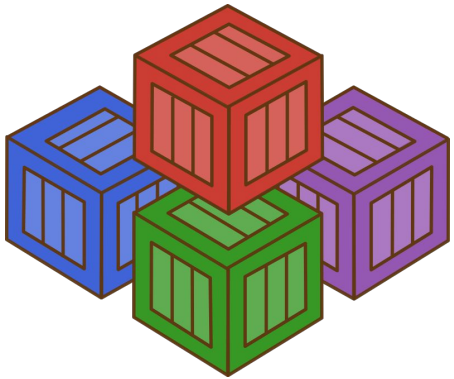
Julia is a High-Productivity Language

- 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)



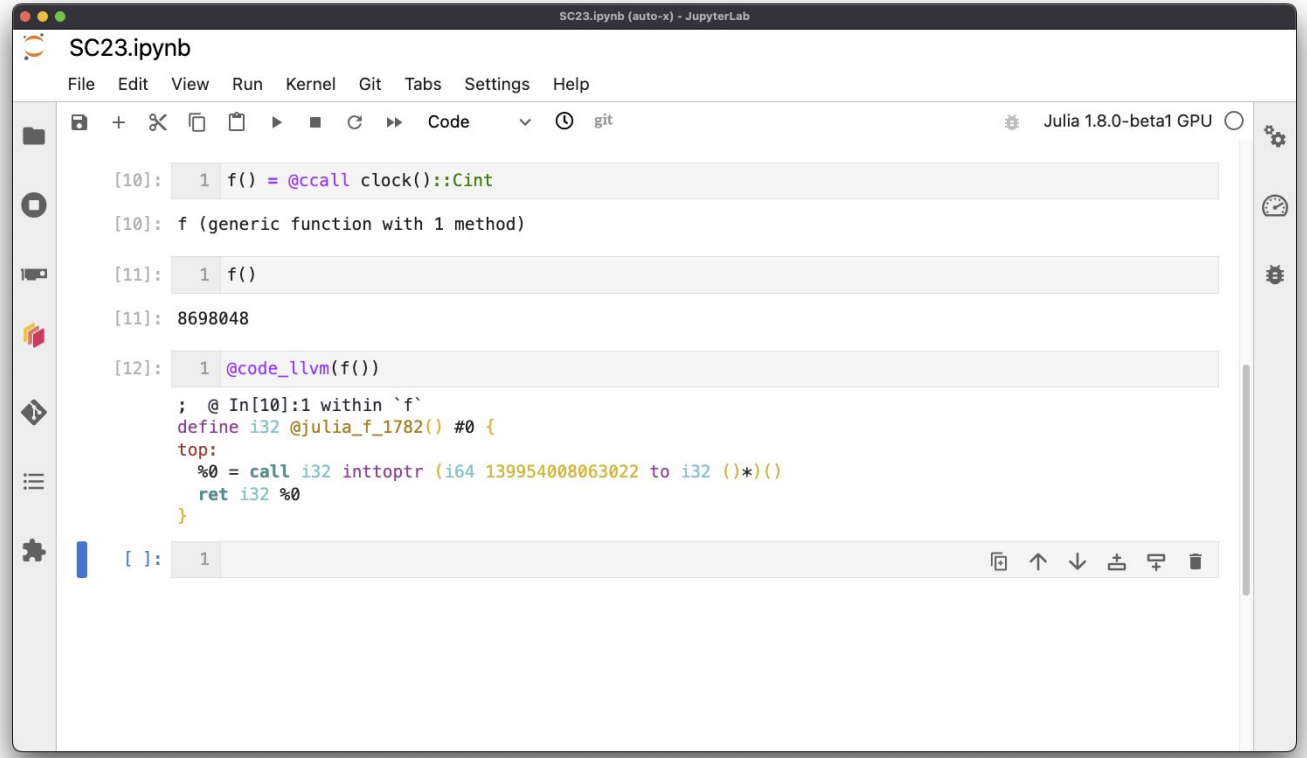
`Project.toml`:

```
[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", "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"
```

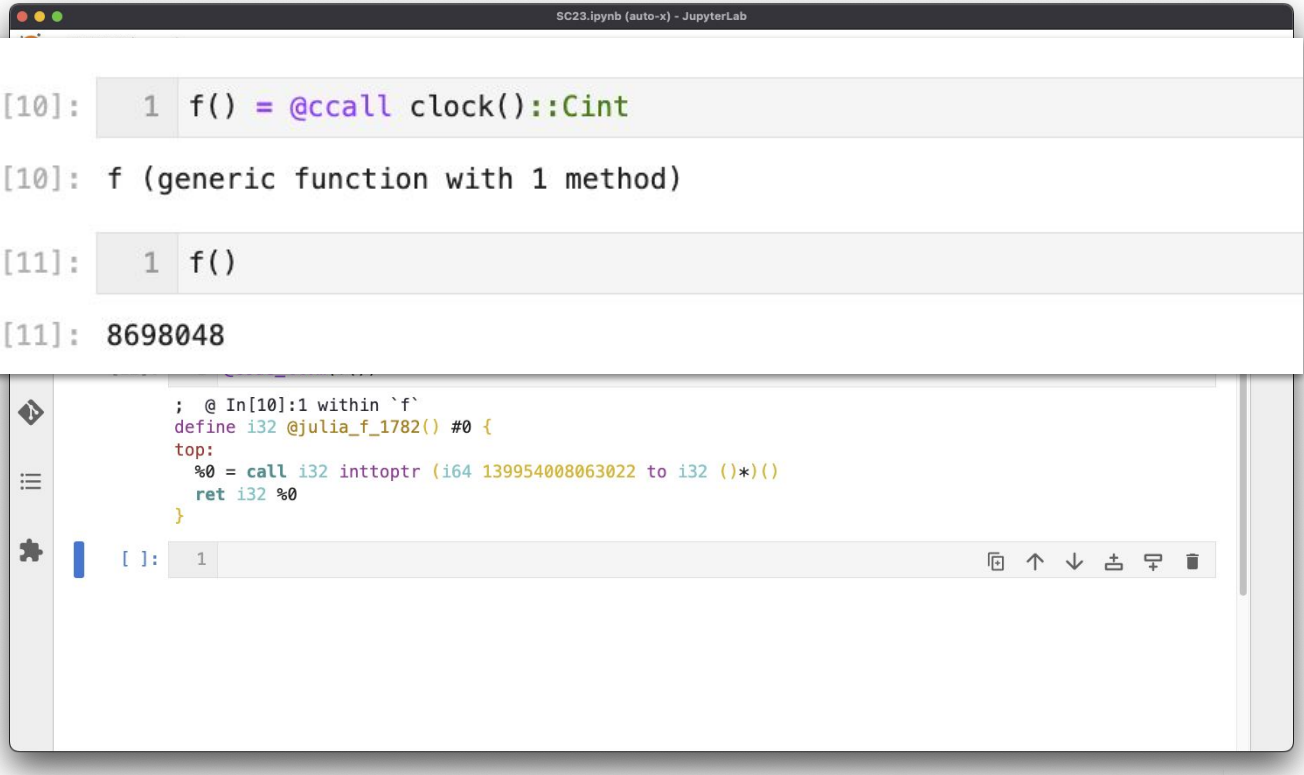
Julia has LLVM under the Hood



Julia has LLVM under the Hood

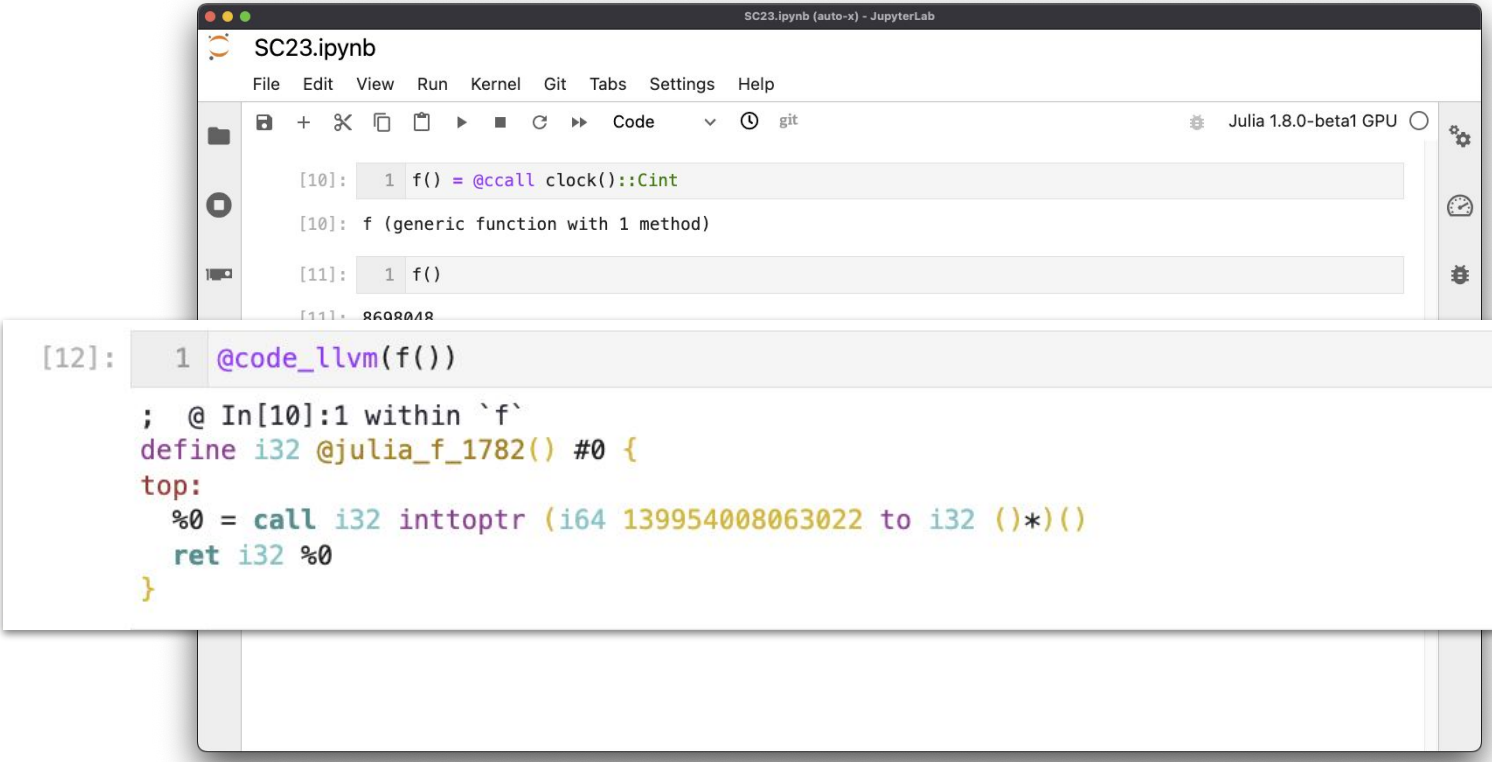
Julia data types are binary-compatible with C

`@ccall` equivalent to c function call

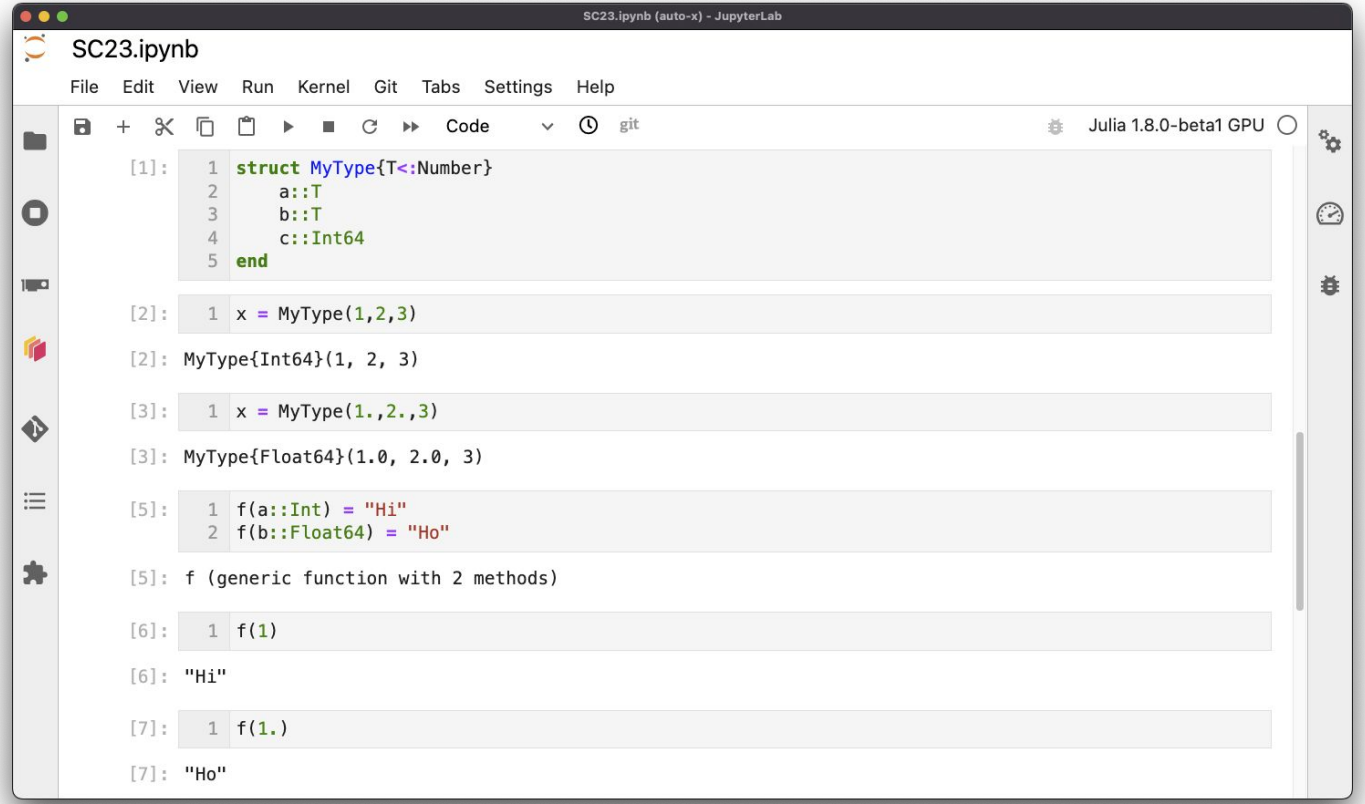


Julia has LLVM under the Hood

`@code_llvm` exposes the LLVM IR for debug purposes



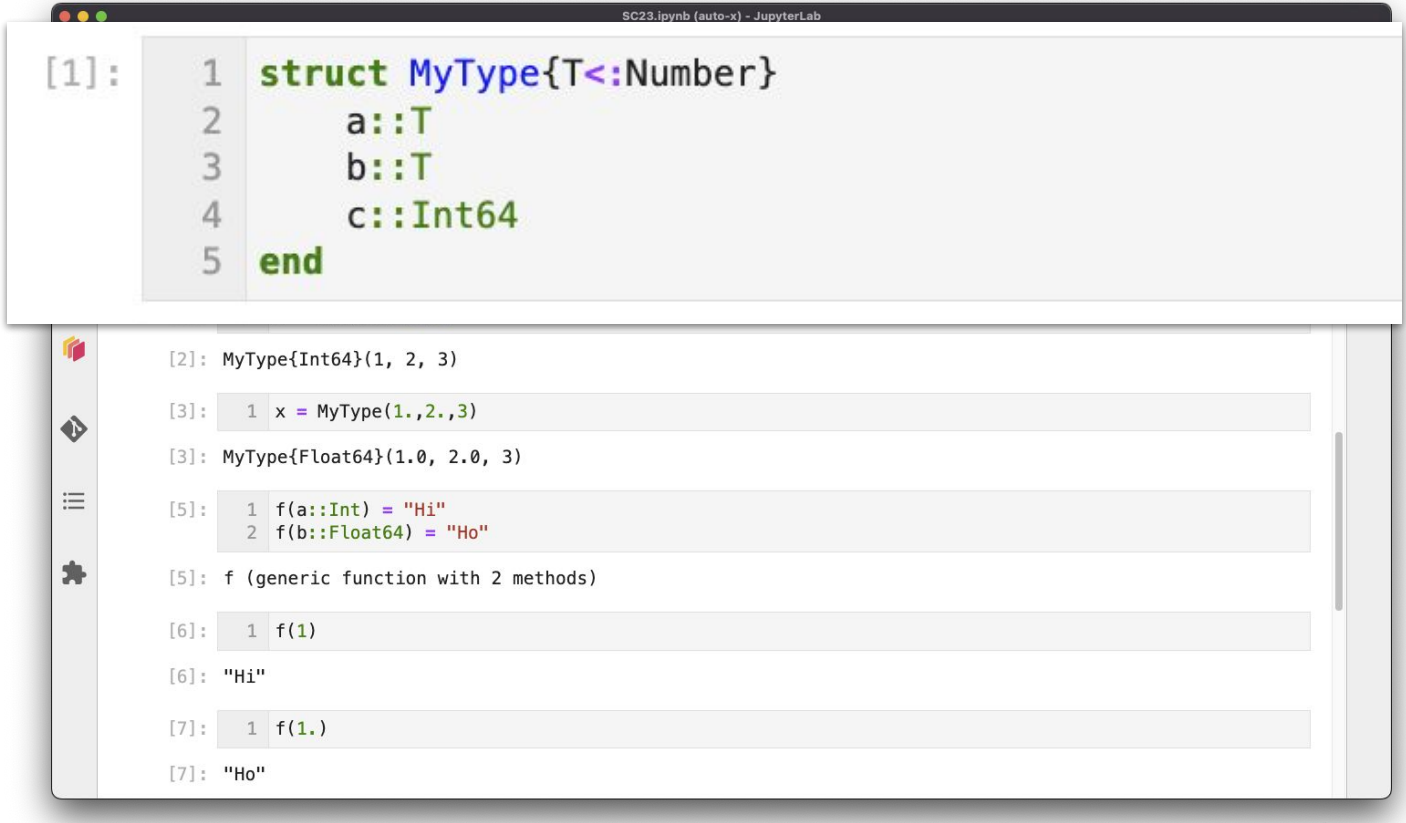
Julia has a Powerful Type System



Julia has a Powerful Type System

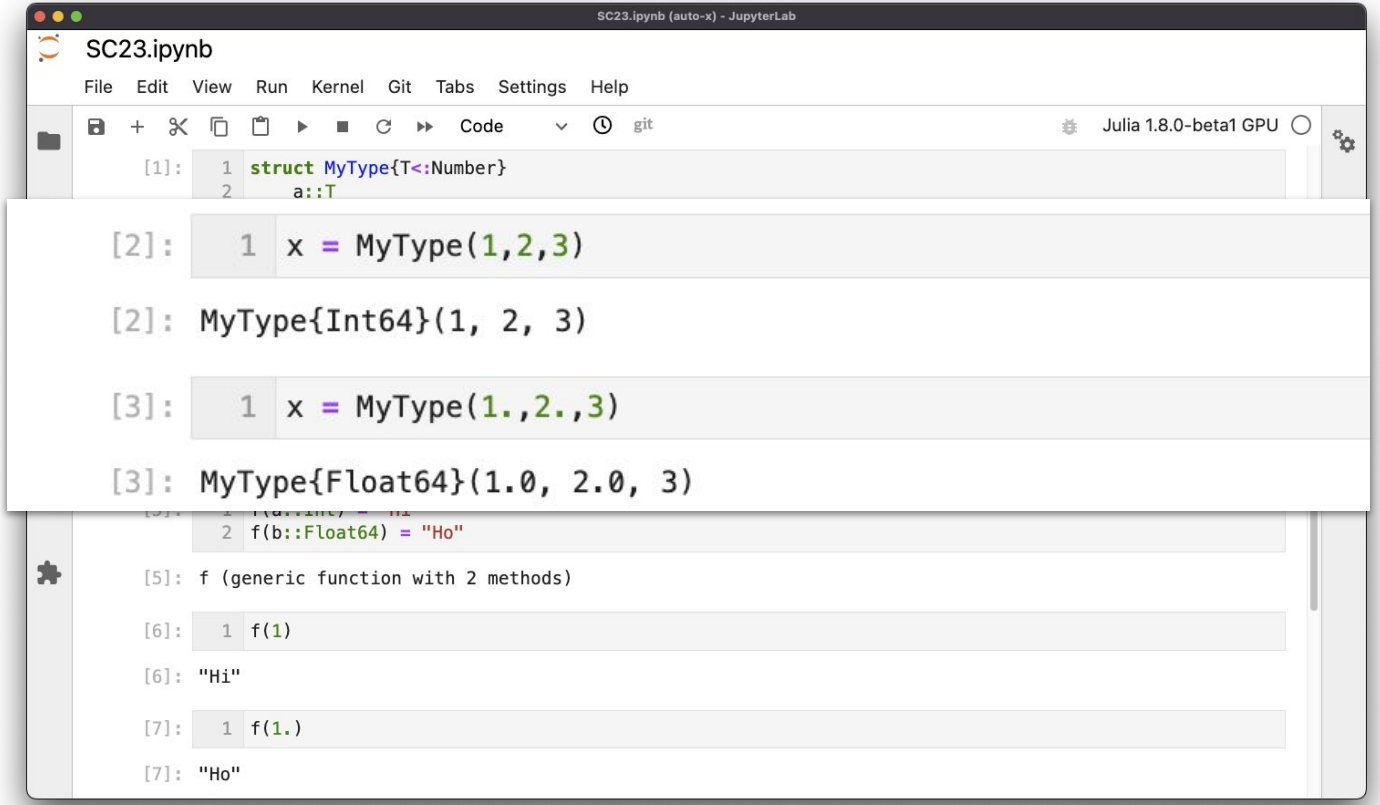
Structured data types are also compatible with C

`{T<:Number}` represents a type template for all types inheriting from `Number`

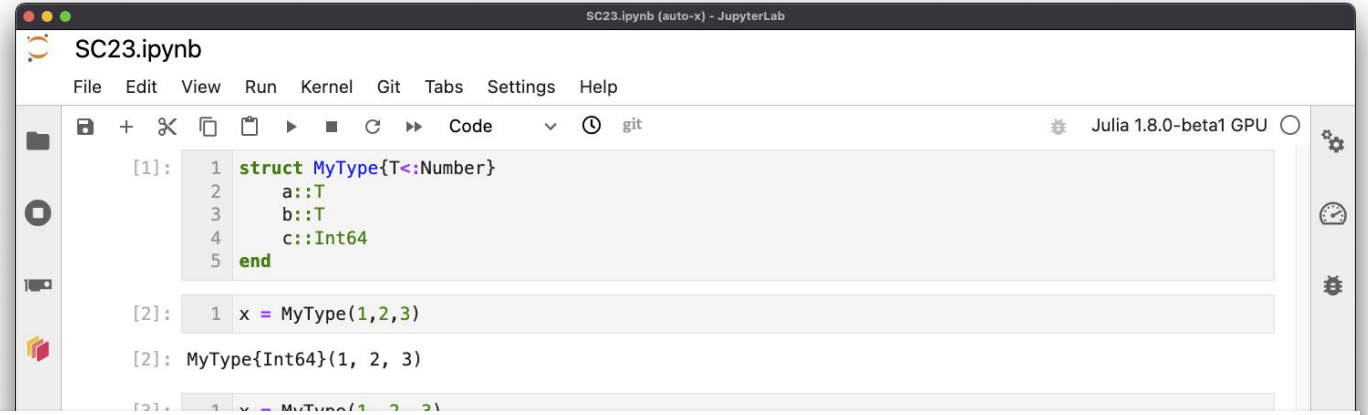


Julia has a Powerful Type System

`{T<:Number}` represents a type template for all types inheriting from `Number`



Julia has a Powerful Type System

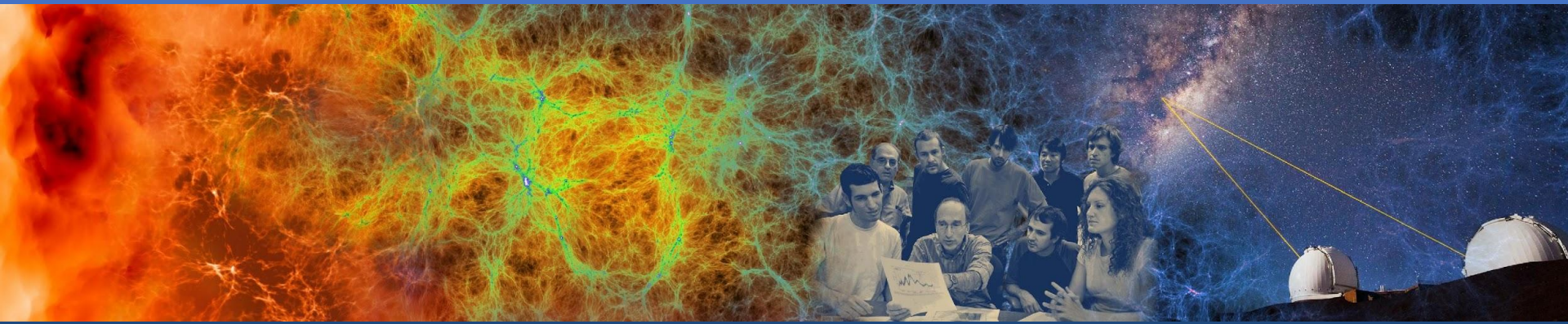


Julia has multiple dispatch: a function can have several implementations (methods) depending on the input types

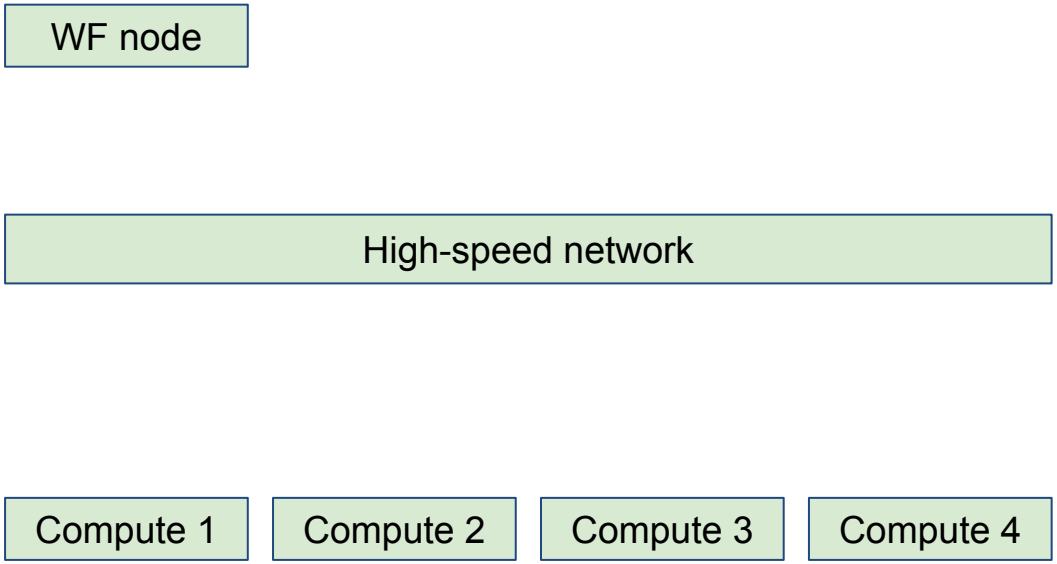
```
[5]:  1 f(a::Int) = "Hi"
      2 f(b::Float64) = "Ho"

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

[6]:  1 f(1)

[6]: "Hi"

[7]:  1 f(1.)

[7]: "Ho"
```

NERSC

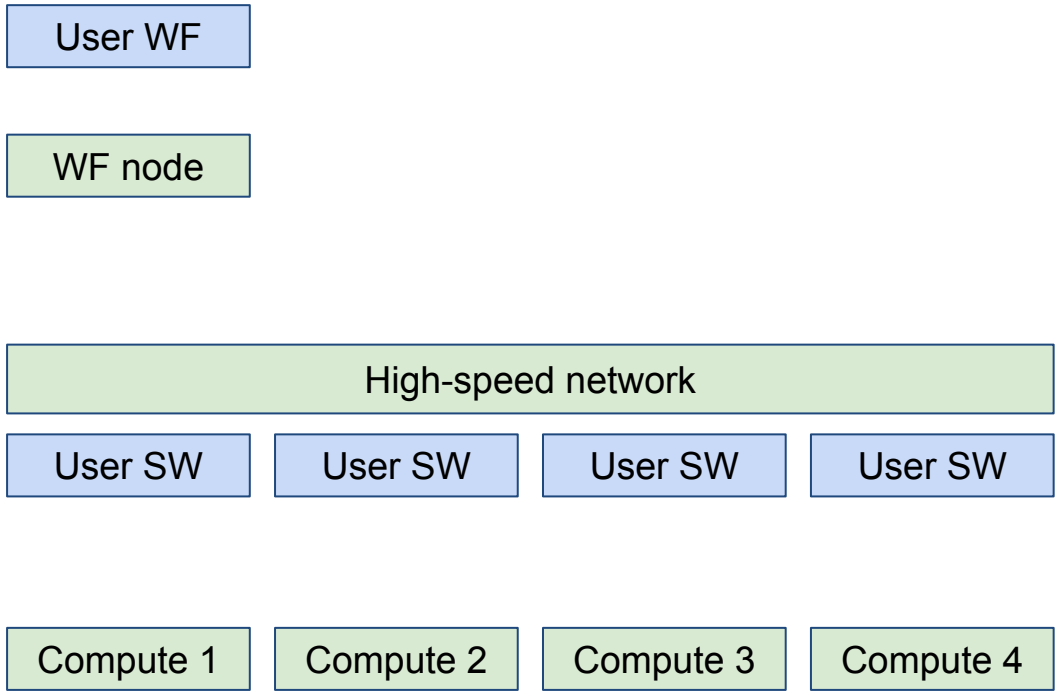
Jupyter as a workflow engine



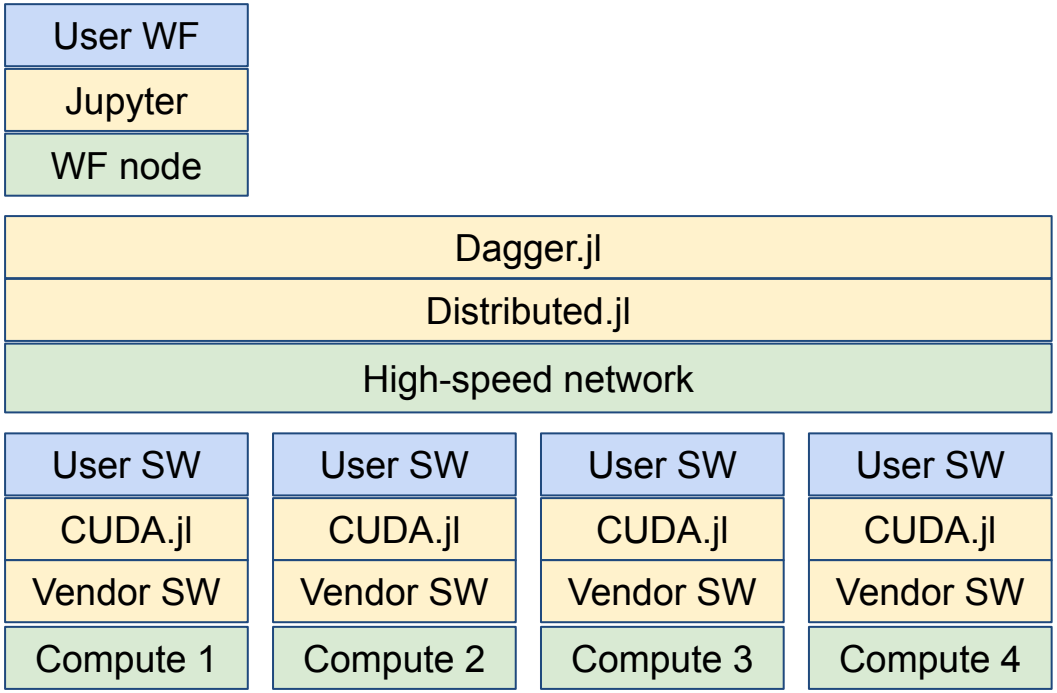
Building a Distributed Julia Application (without MPI)



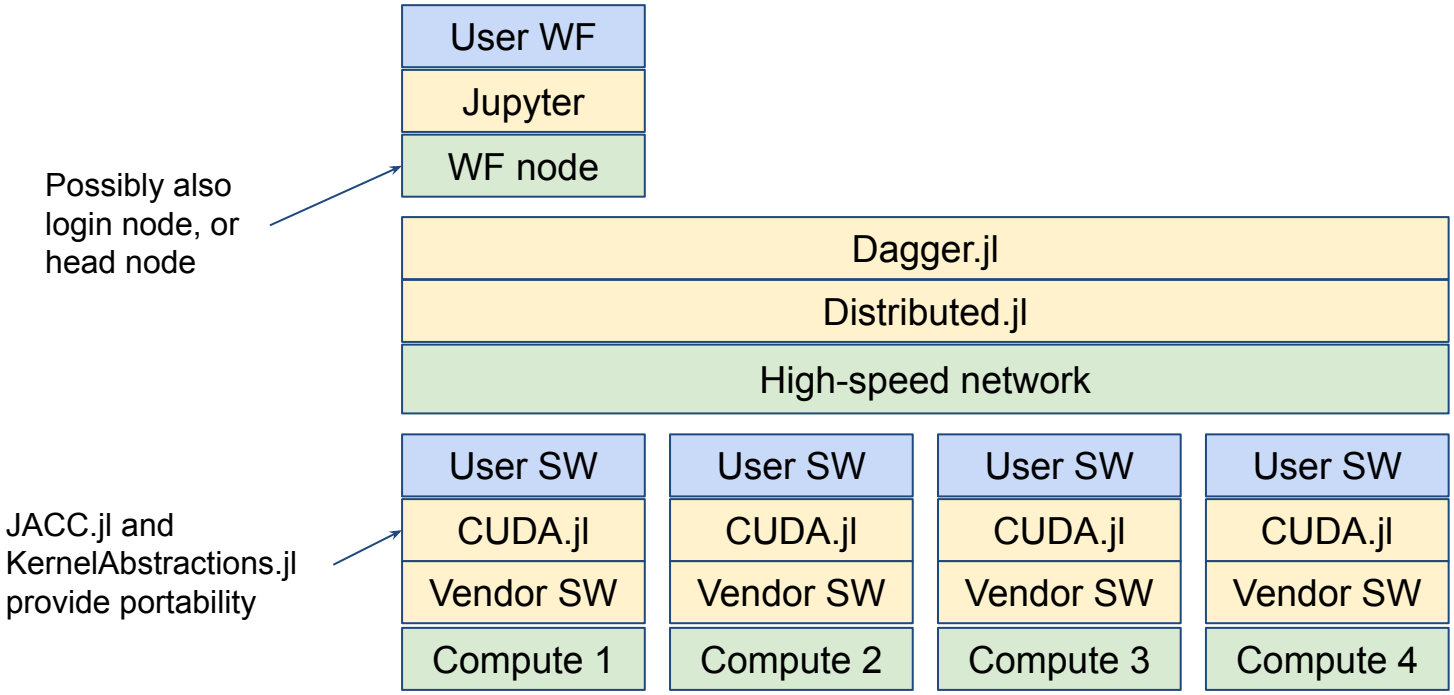
Building a Distributed Julia Application (without MPI)



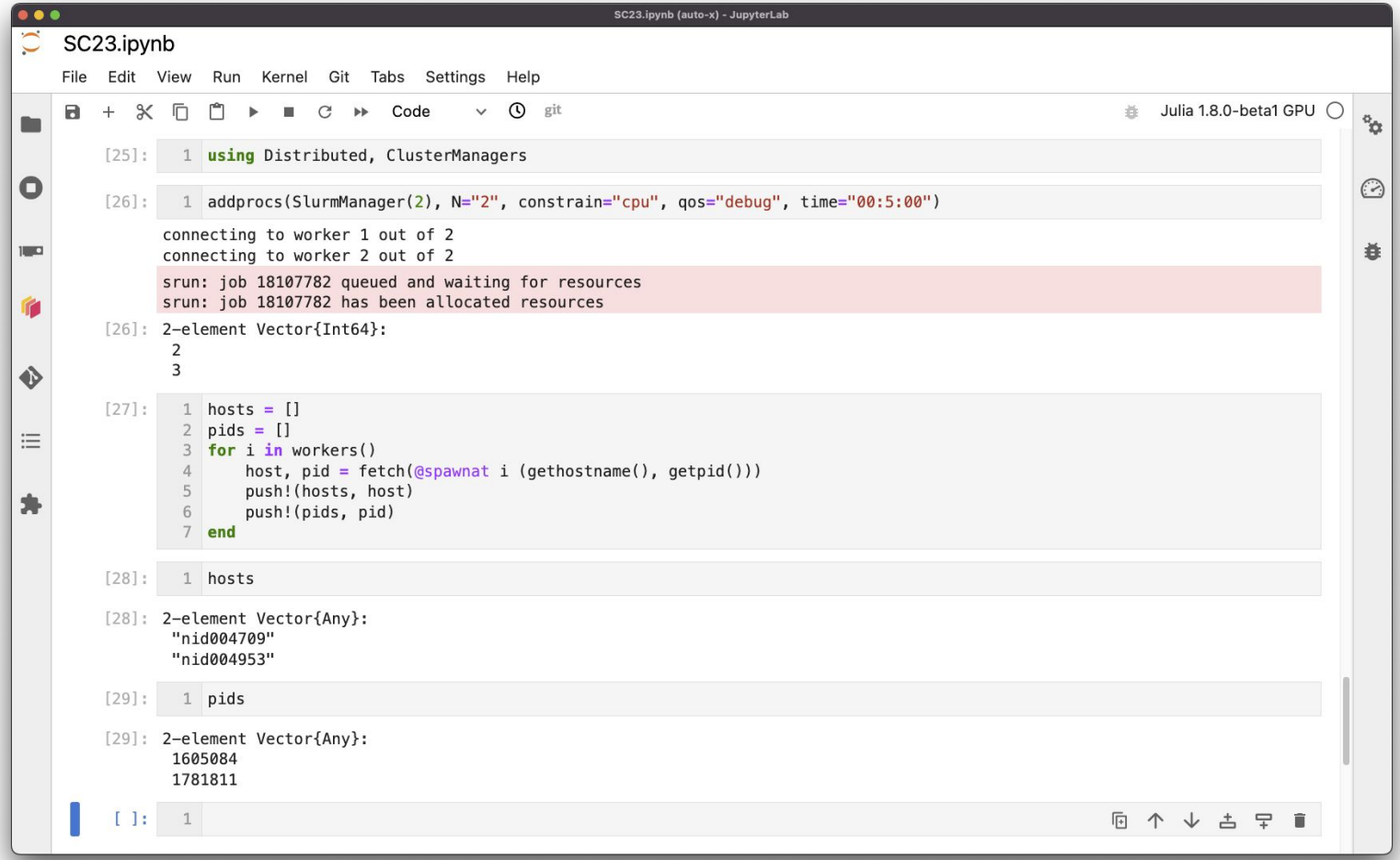
Building a Distributed Julia Application (without MPI)



Building a Distributed Julia Application (without MPI)



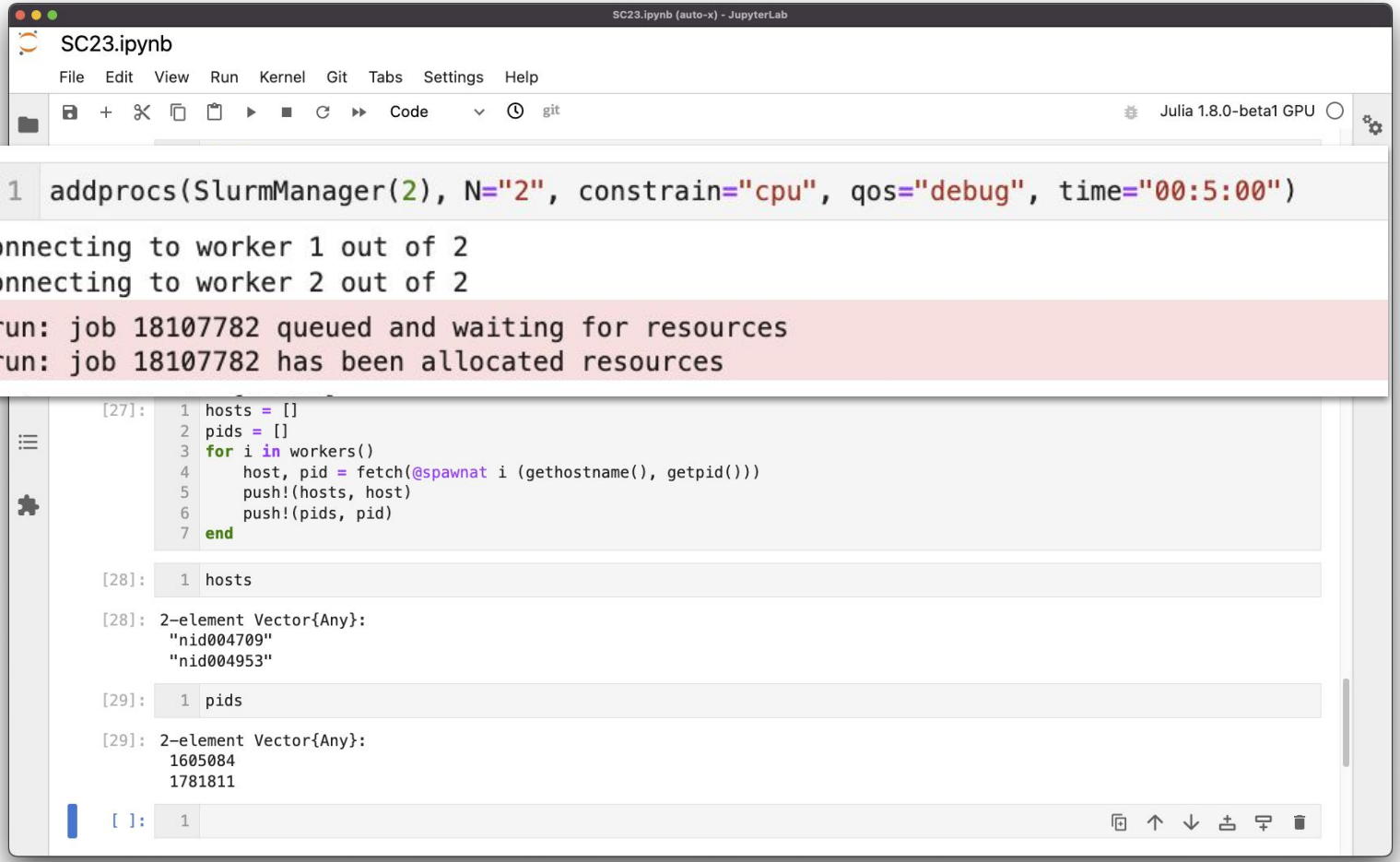
Cluster Managers: Interaction with Slurm



Cluster Managers: Interaction with Slurm

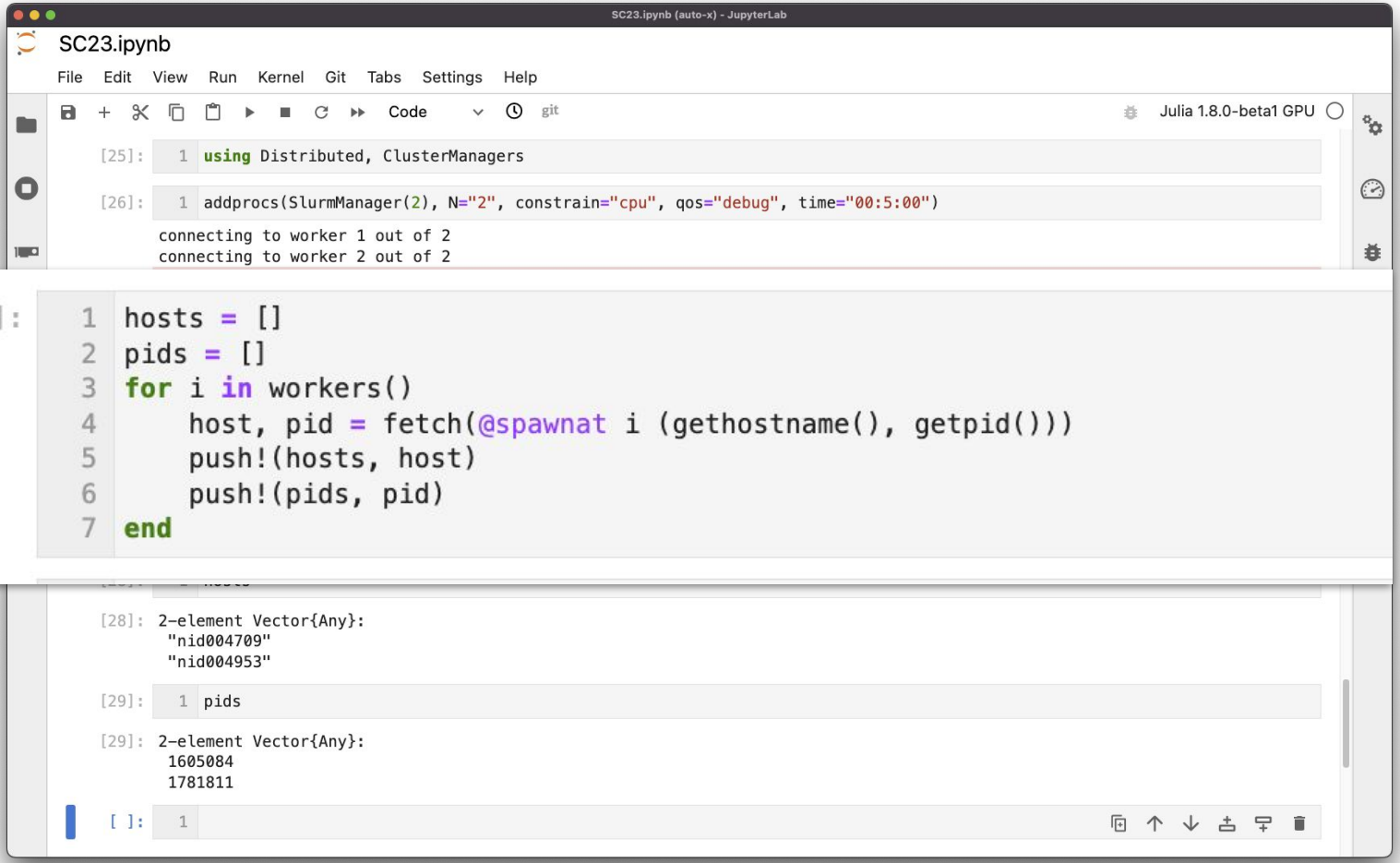
Request two cpu nodes

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

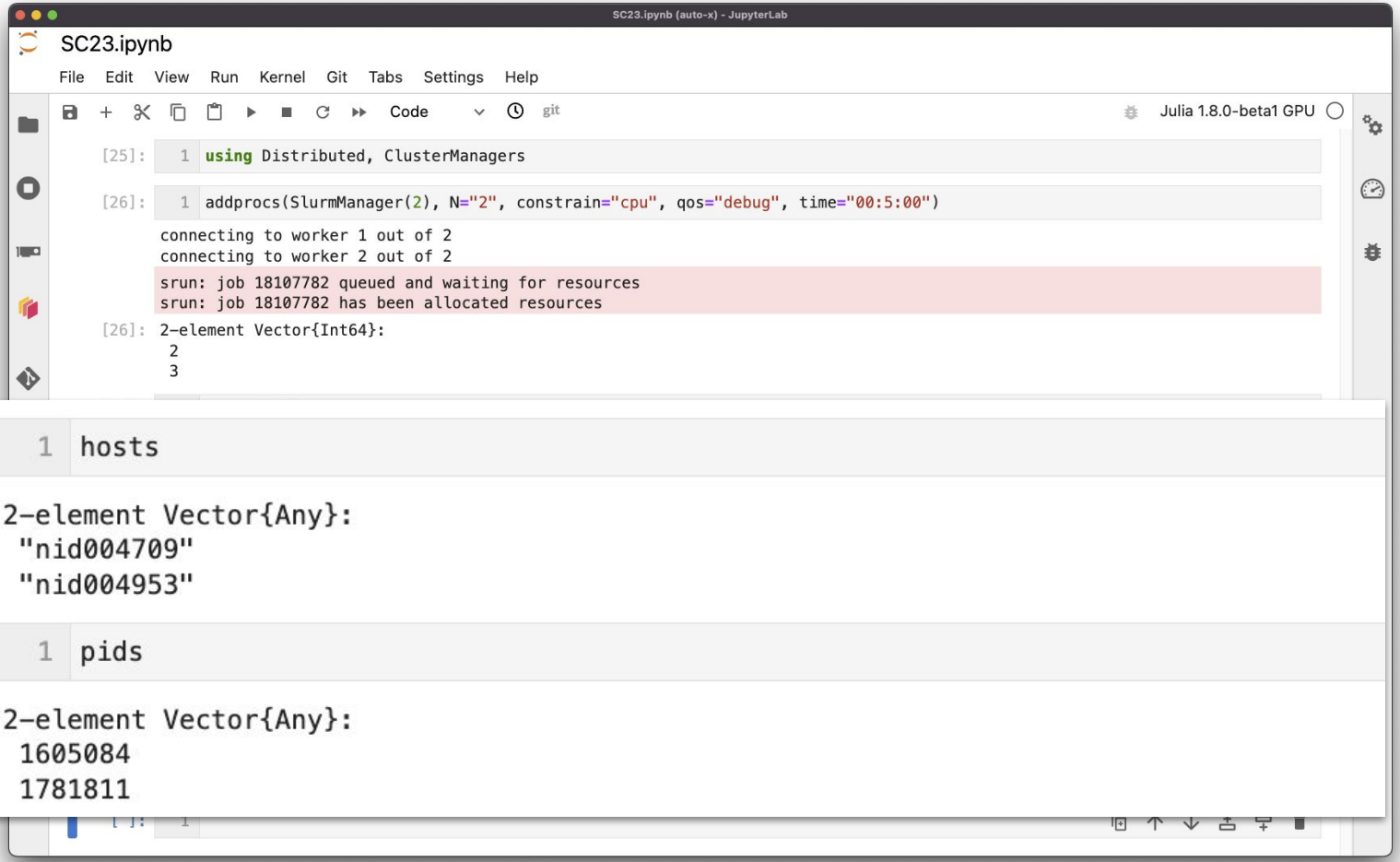


Cluster Managers: Interaction with Slurm

Get hostnames and pids from each worker



Cluster Managers: Interaction with Slurm



`Distributed.jl` supports basic workflows

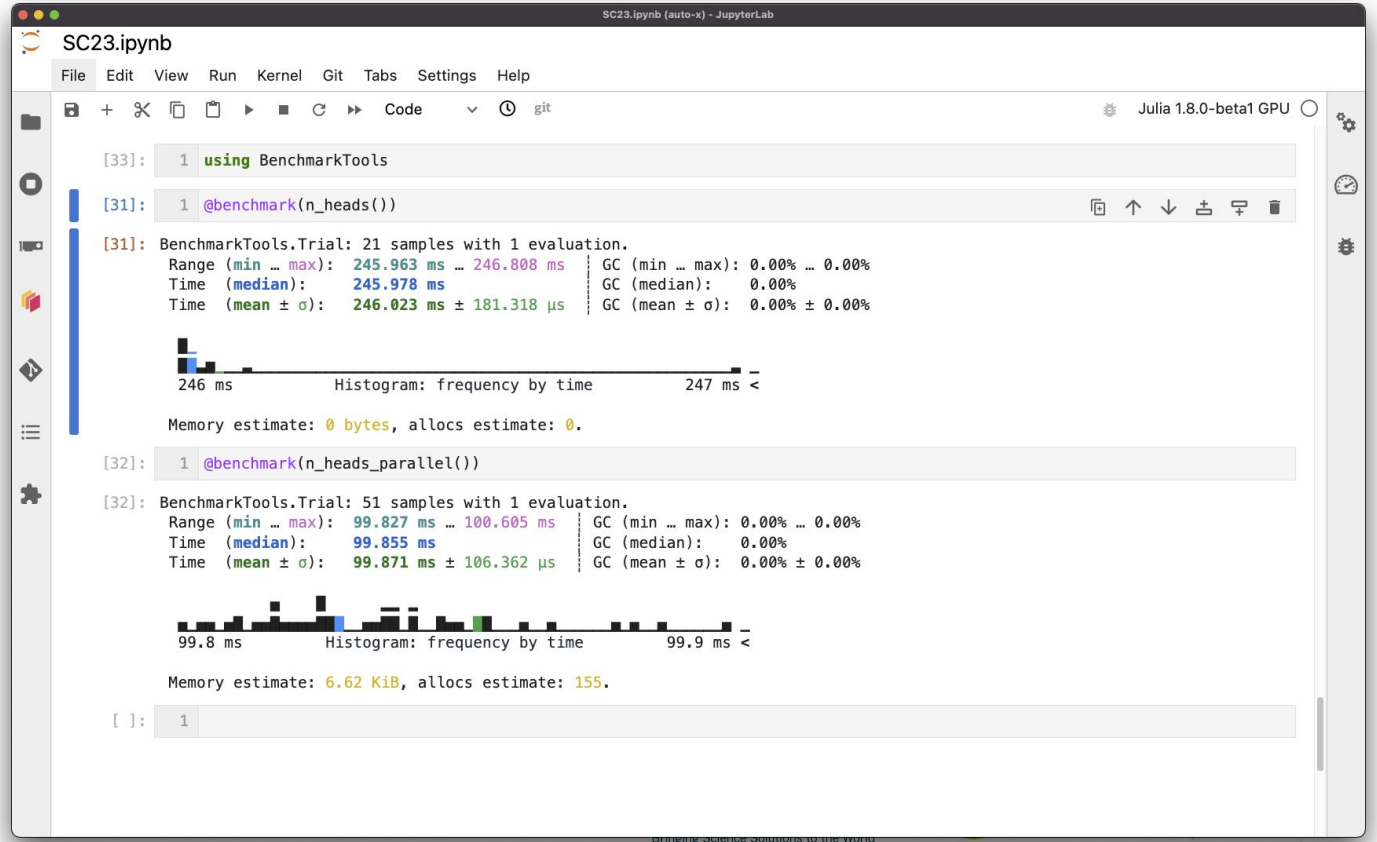
Basic task: count number of heads from 2x10^8 fair coin tosses.

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

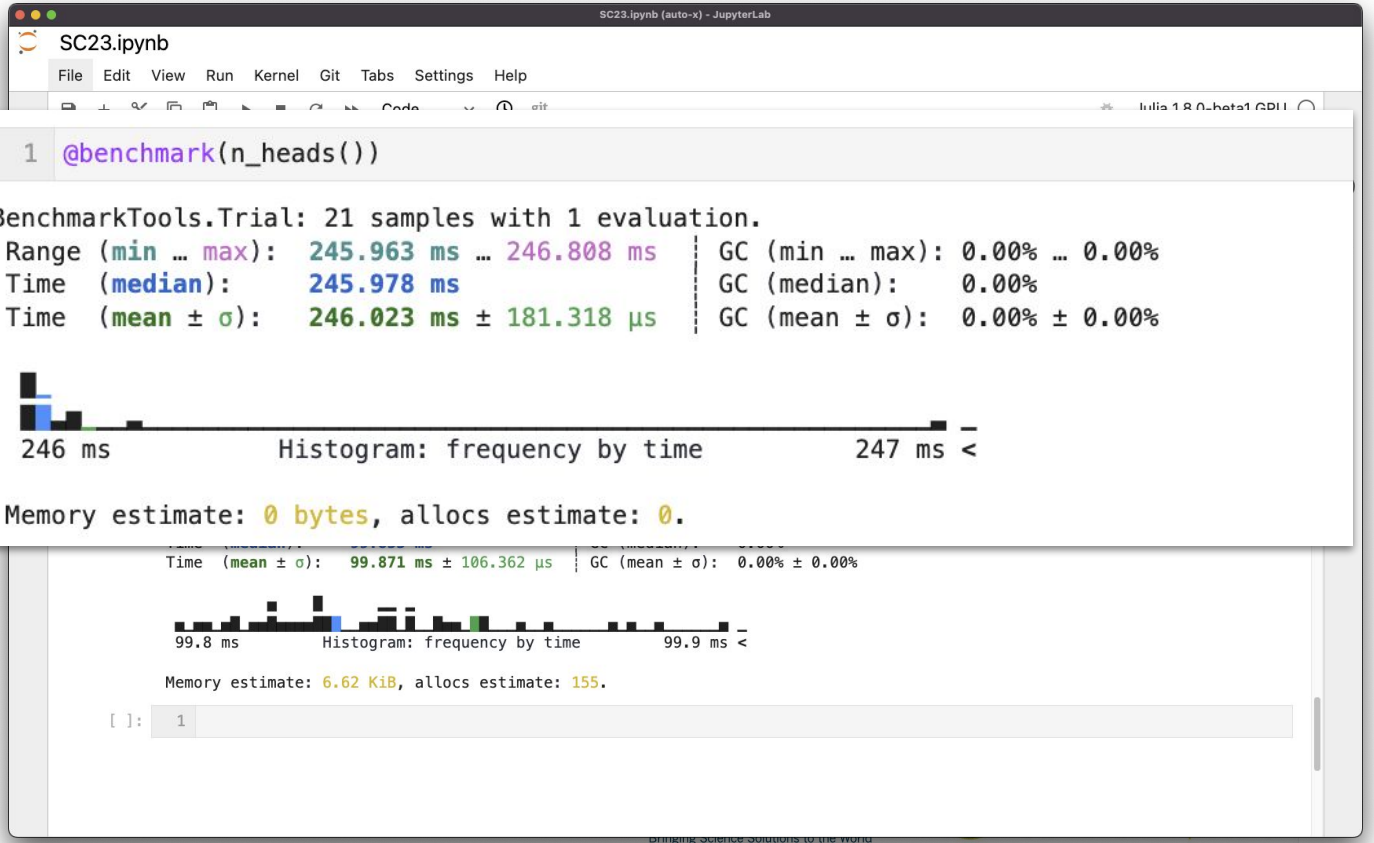
- Most Julia data types are trivial to serialize and communicate over network (the users doesn't have to do anything "special" to enable this)

```
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

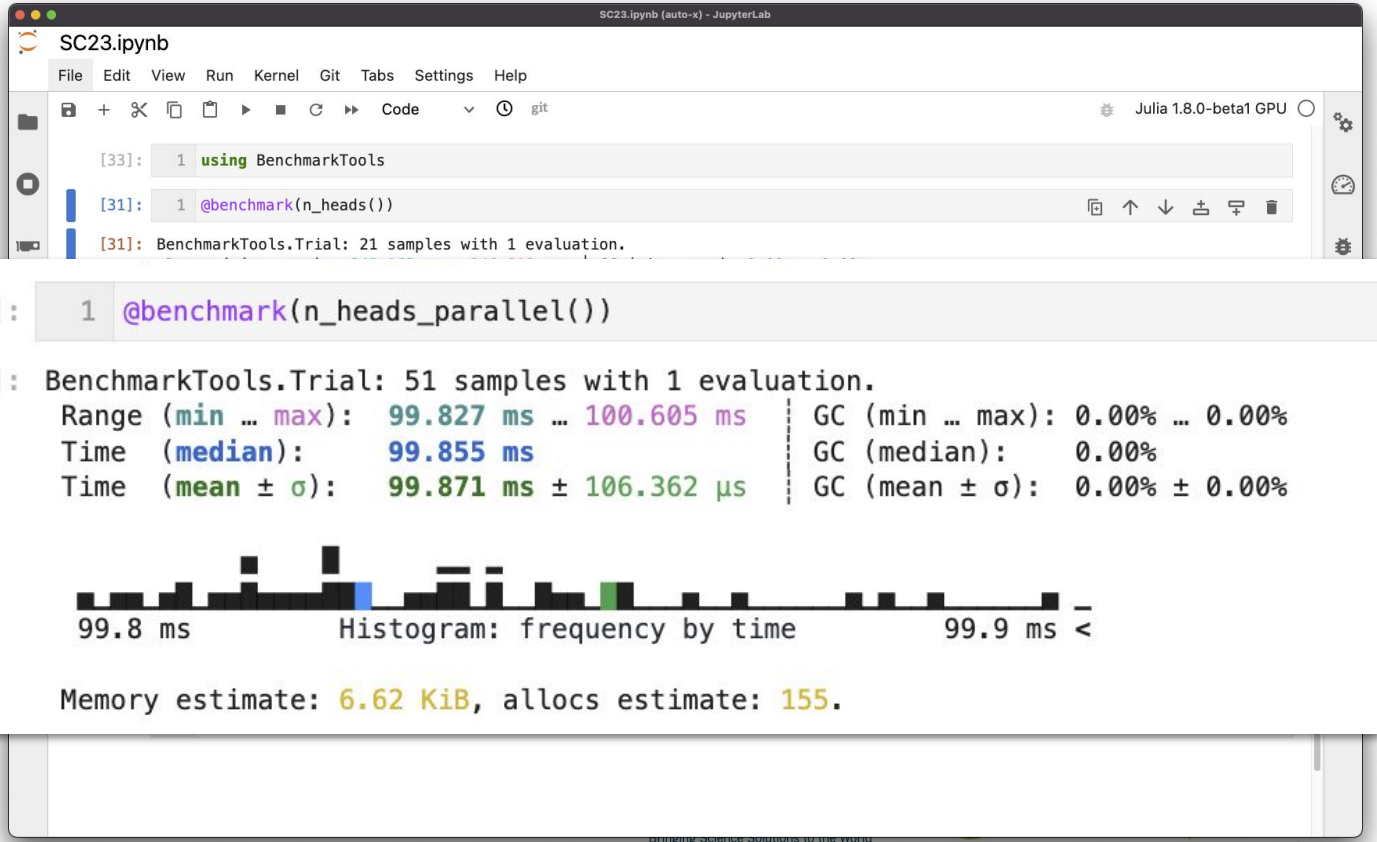


Performance Gains from Distributing Work

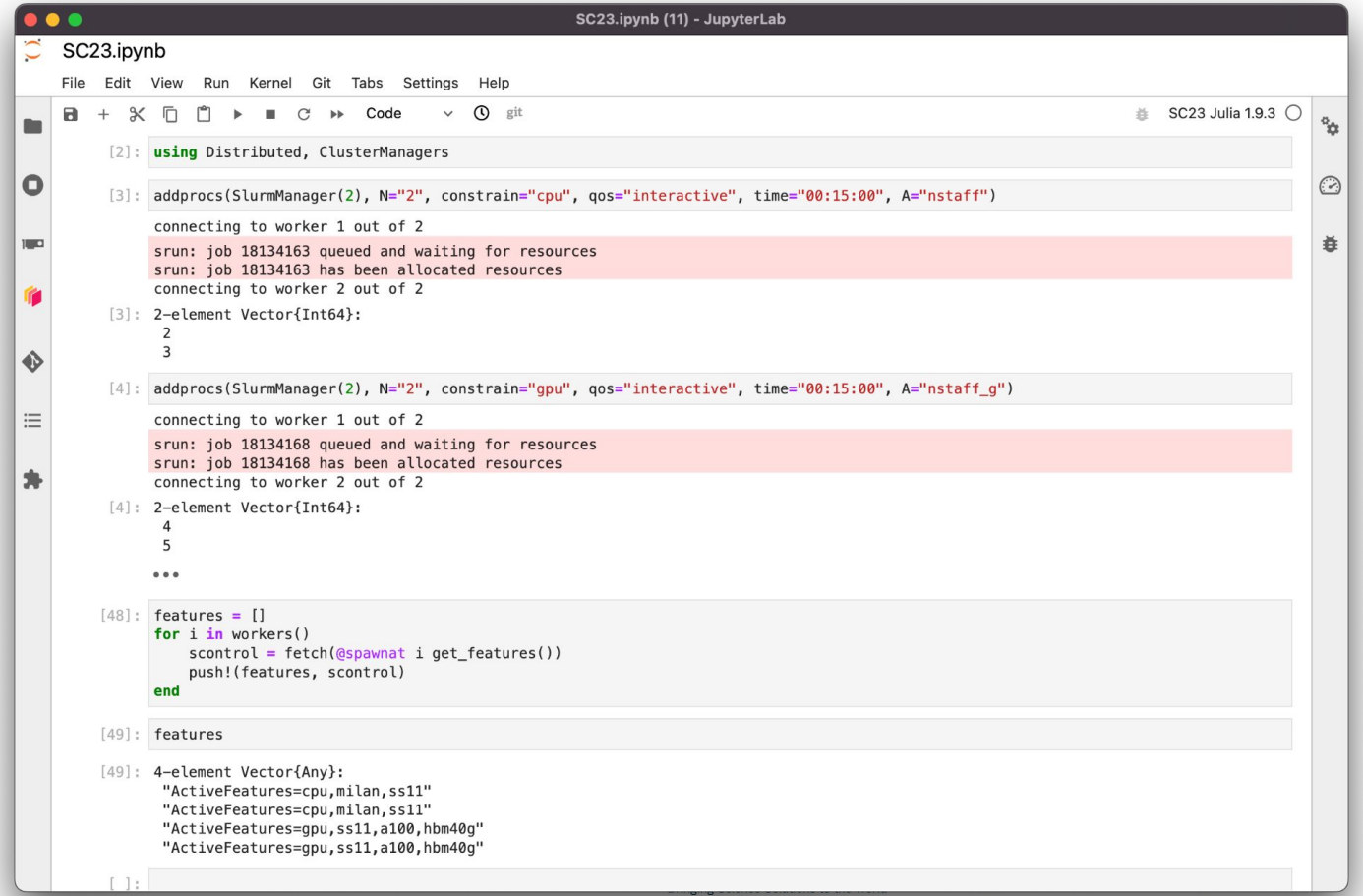


Performance Gains from Distributing Work

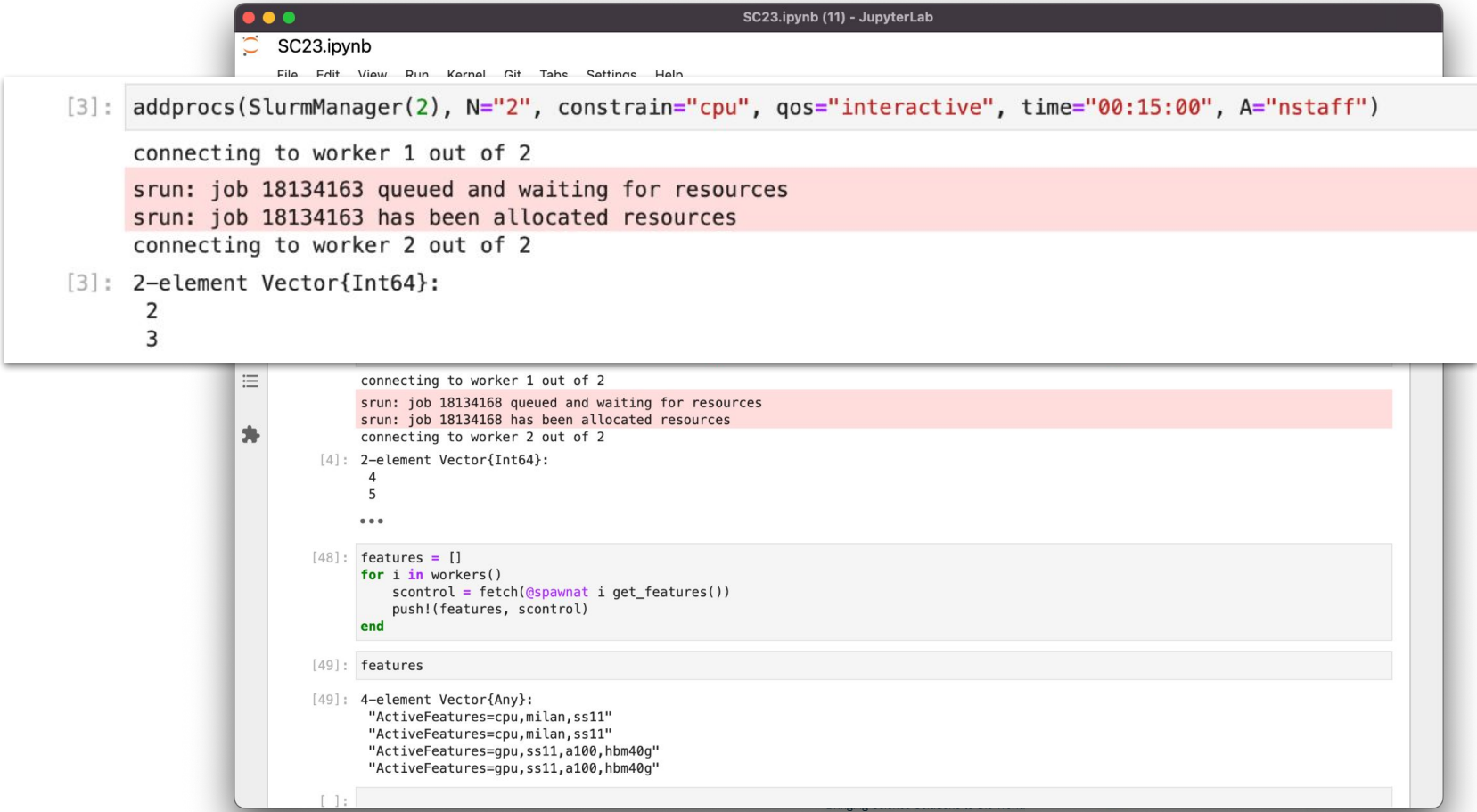
Distributing work over 2 nodes results in a 2x performance increase



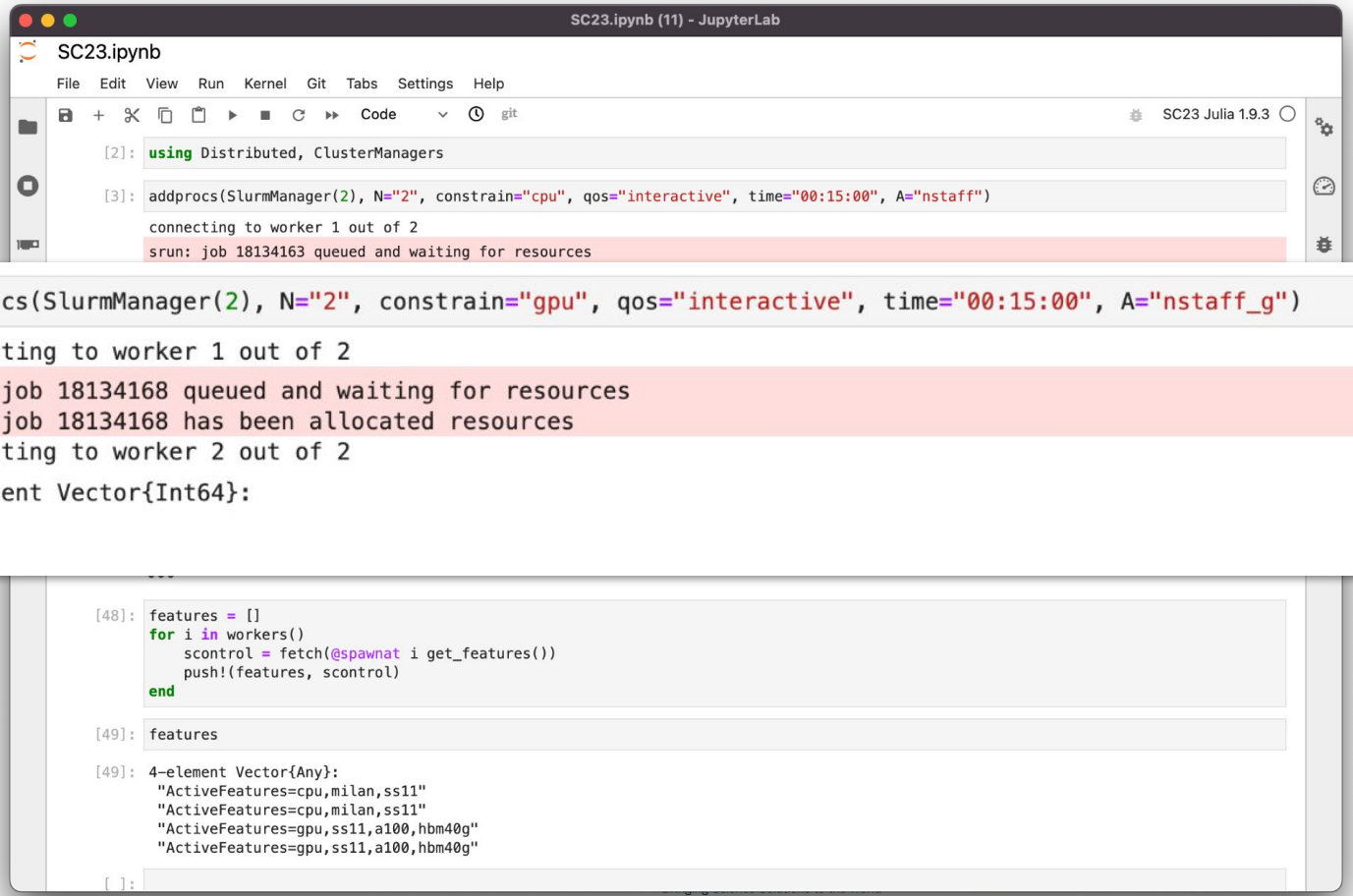
Tangent: Hybrid CPU/GPU Jobs



Tangent: Hybrid CPU/GPU Jobs

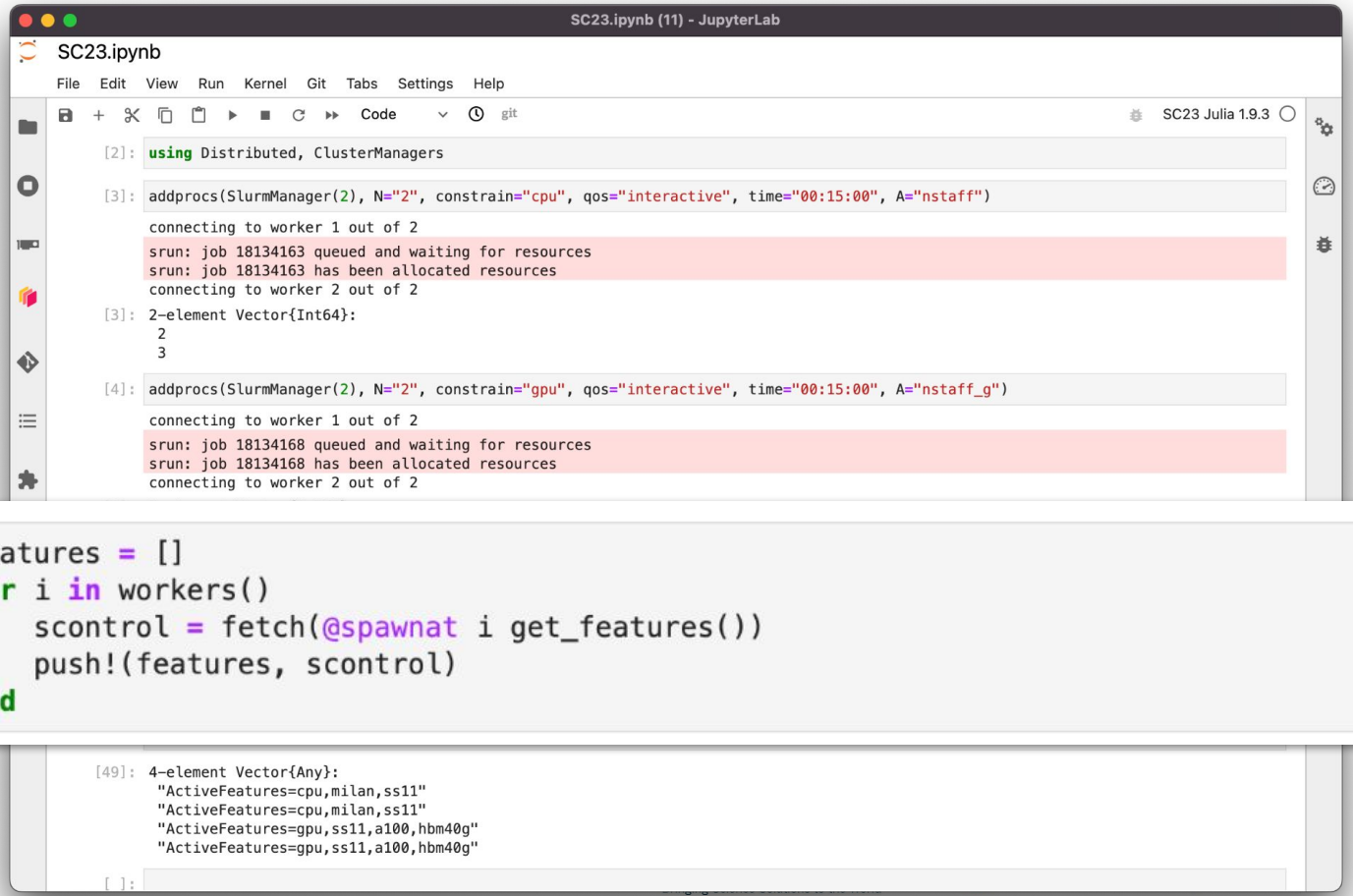


Tangent: Hybrid CPU/GPU Jobs

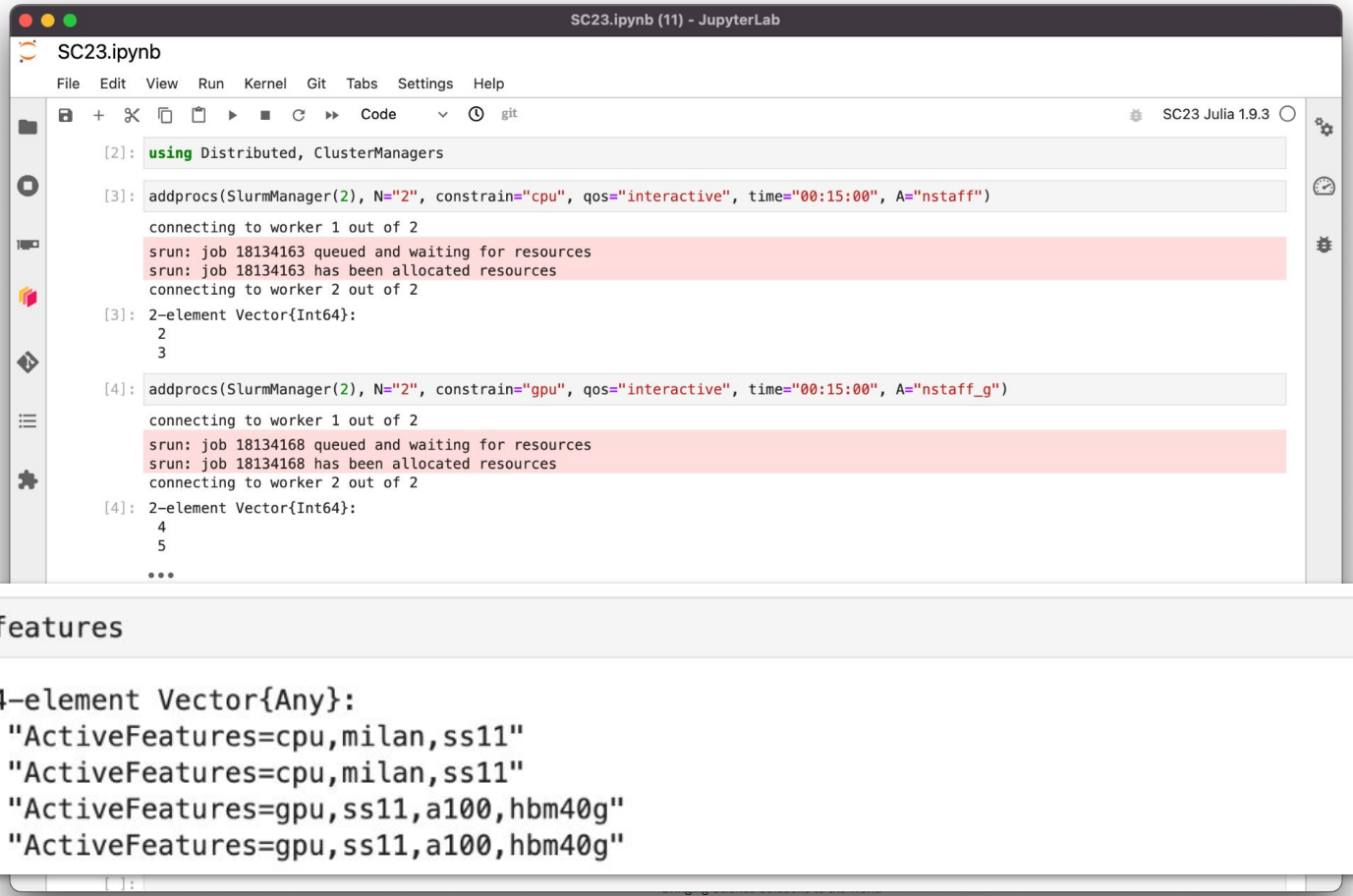


NERSC

Tangent: Hybrid CPU/GPU Jobs



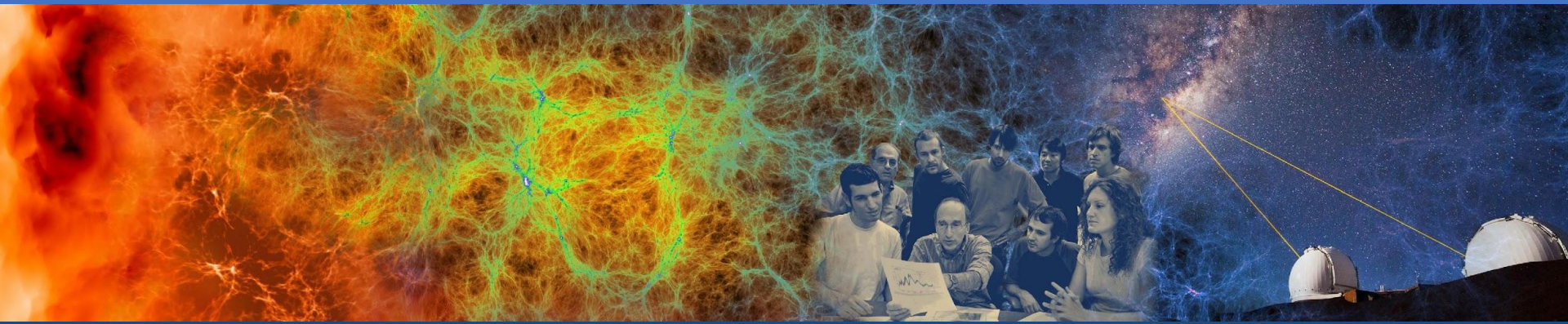
Tangent: Hybrid CPU/GPU Jobs




```
[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"
```

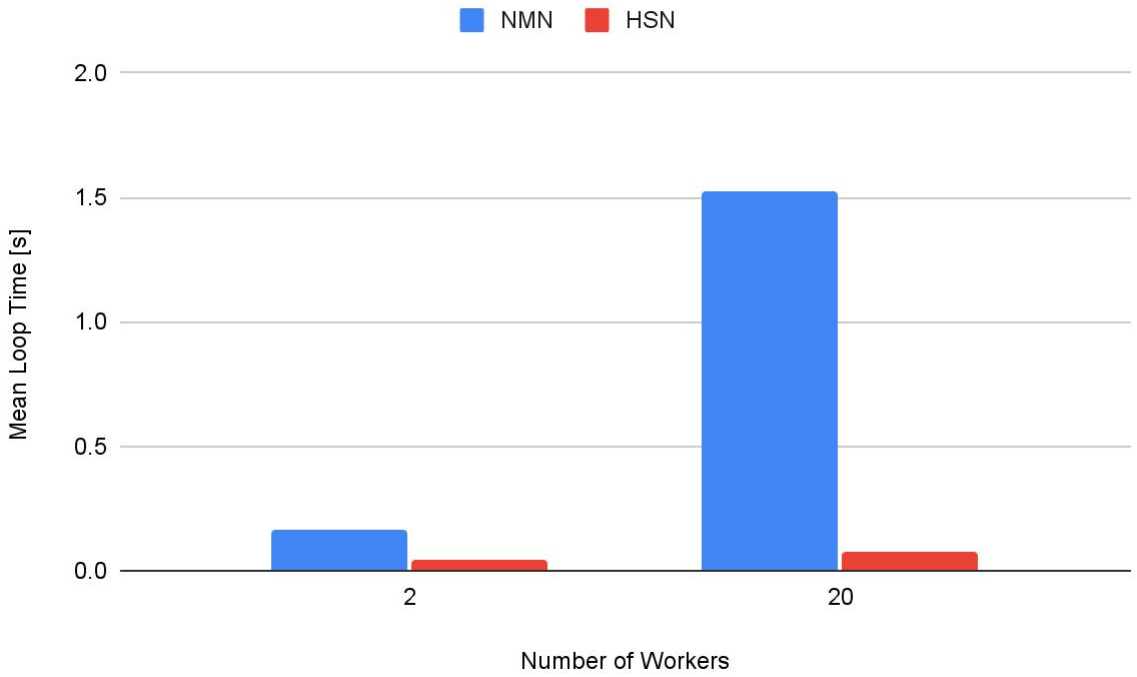
NERSC

Tangent: Network Discovery



Workflow Support Story

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

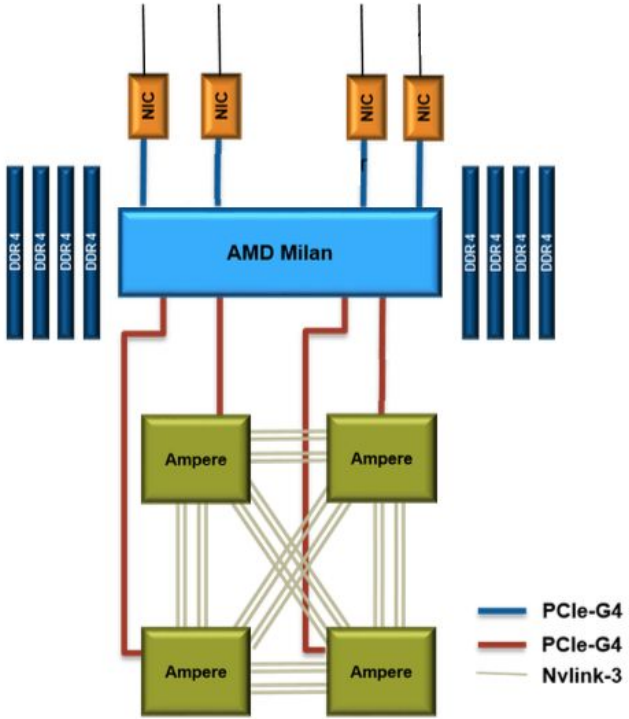


Perlmutter is a Heterogeneous System

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

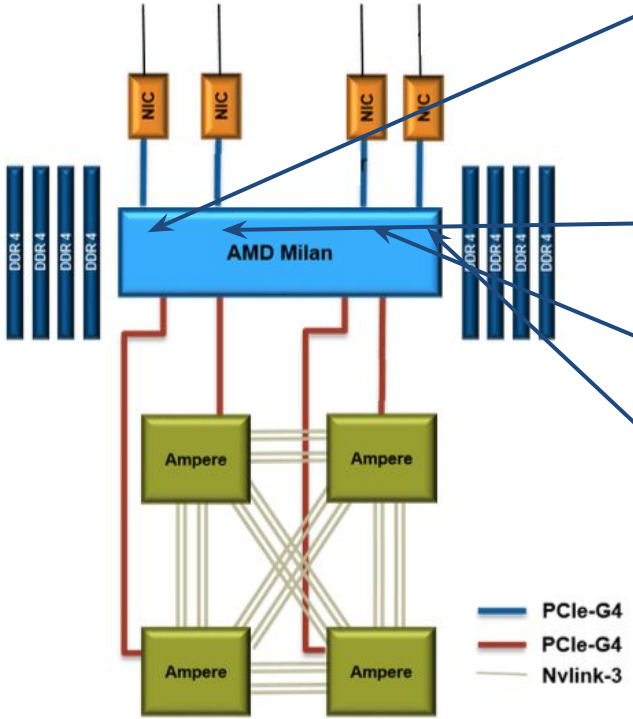
- Each GPU node has 4 NICs
  - 1 NIC and 1 GPU per host bridge
- Each CPU node has 1 NIC

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"]
```

NERSC

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
```

This one! 

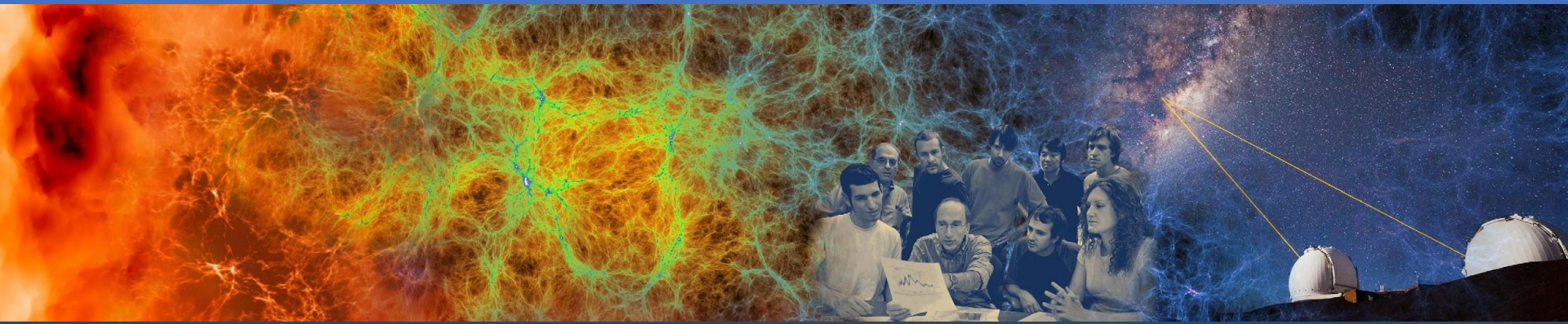
```
213: Hwloc.Object: PCI_Device [c2:00.0 (Ethernet)]
└─ Hwloc.Object: OS_Device [Net "hsn0"]
213: Hwloc.Object: PCI_Device [c3:00.0 (Ethernet)]
└─ Hwloc.Object: OS_Device [Net "nmn0"]
47: Hwloc.Object: PCI_Device [81:00.0 (Ethernet)]
└─ Hwloc.Object: OS_Device [Net "hsn1"]
379: Hwloc.Object: PCI_Device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

NERSC

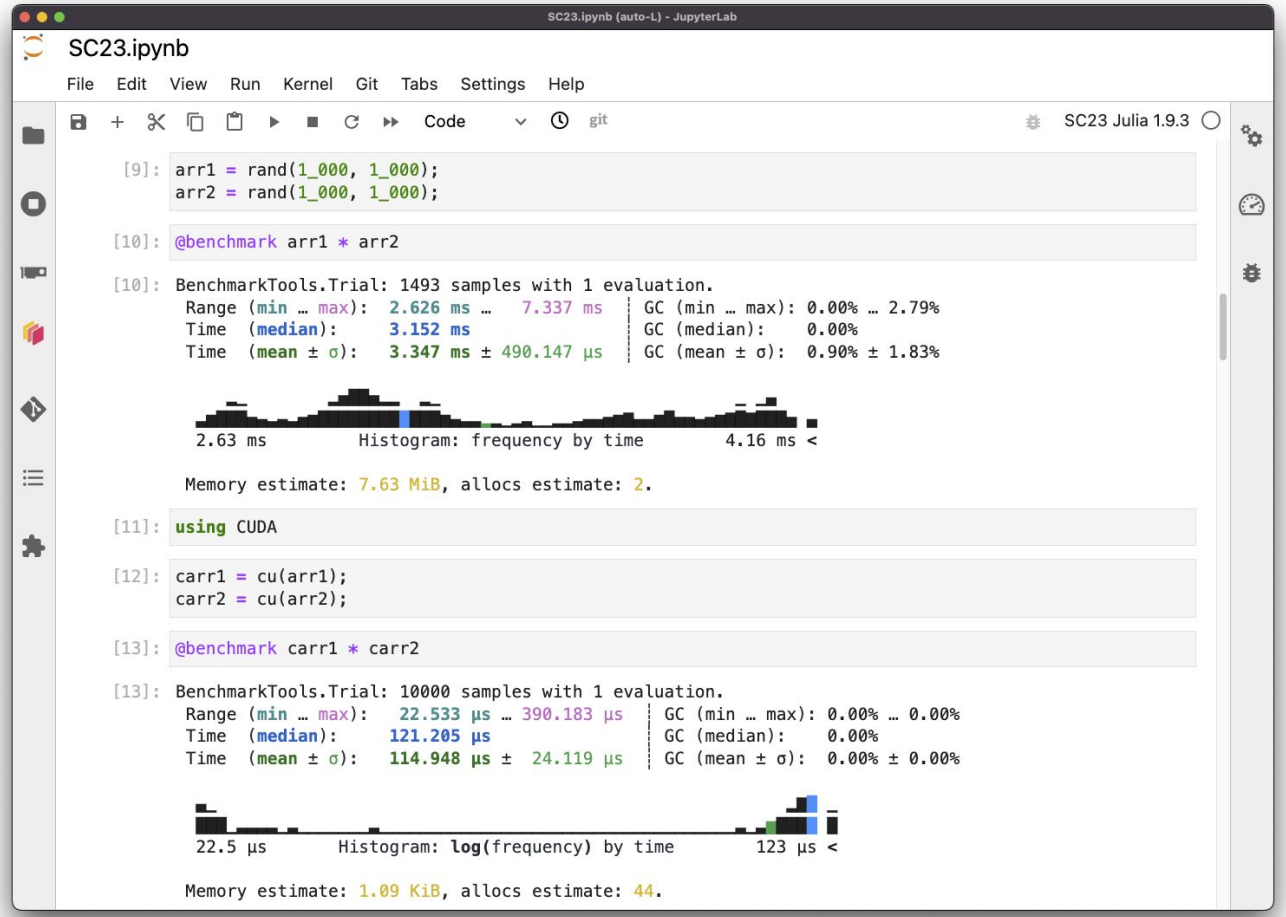
Future

- 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



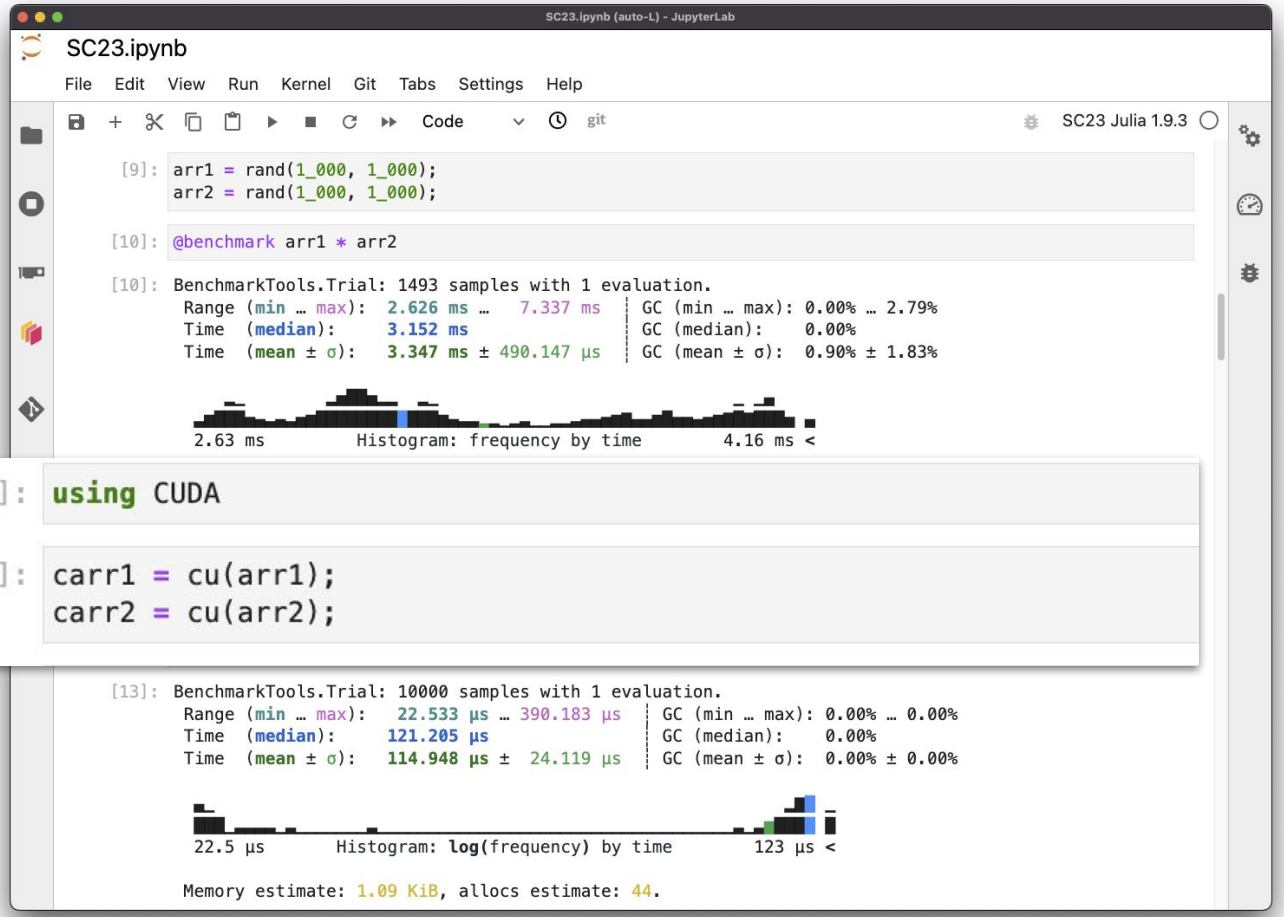
NERSC

CUDA.jl: Interfacing with Nvidia GPUs

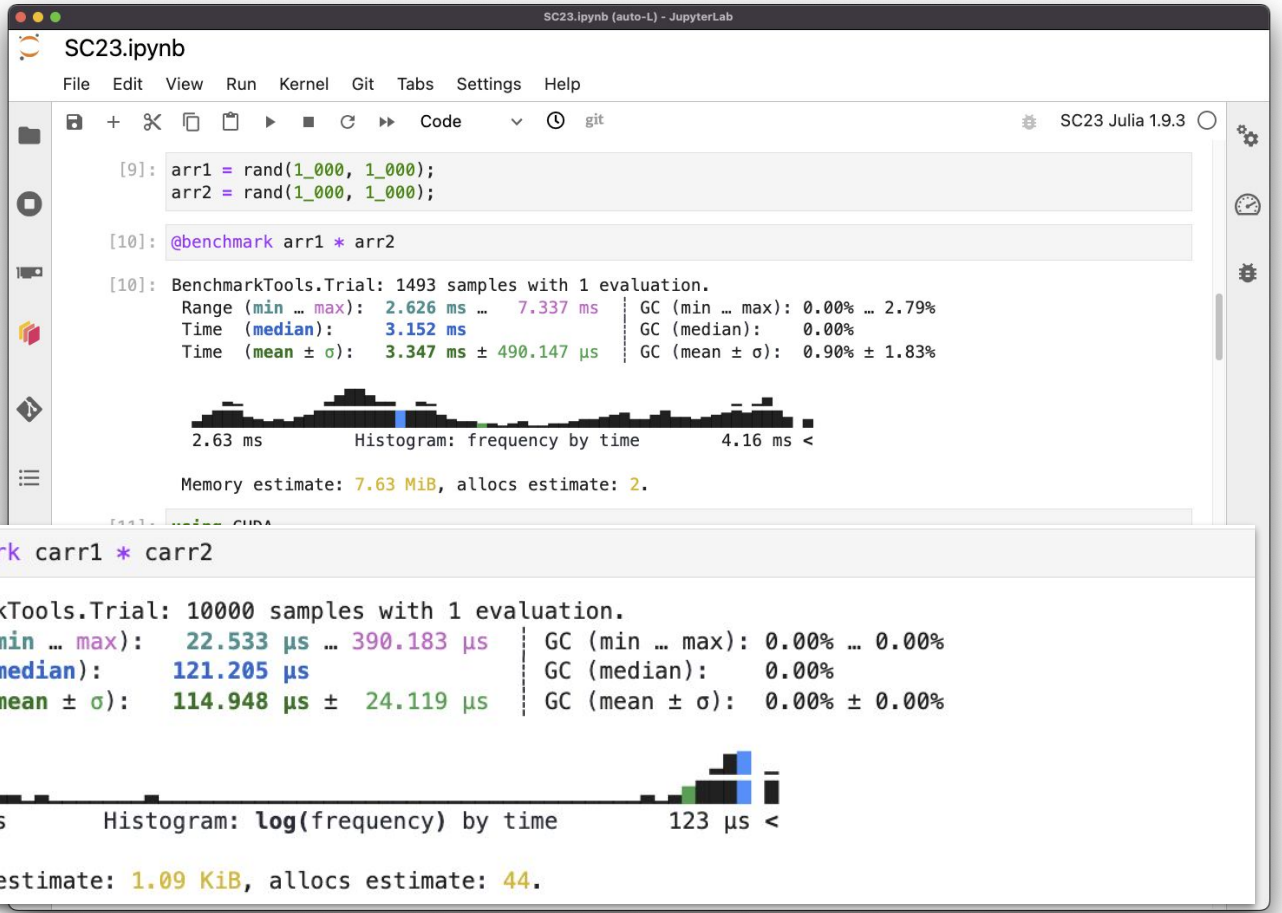
Basic example:
1000x1000 matmul
using OpenBLAS

CUDA.jl: Interfacing with Nvidia GPUs

Copy arrays to device



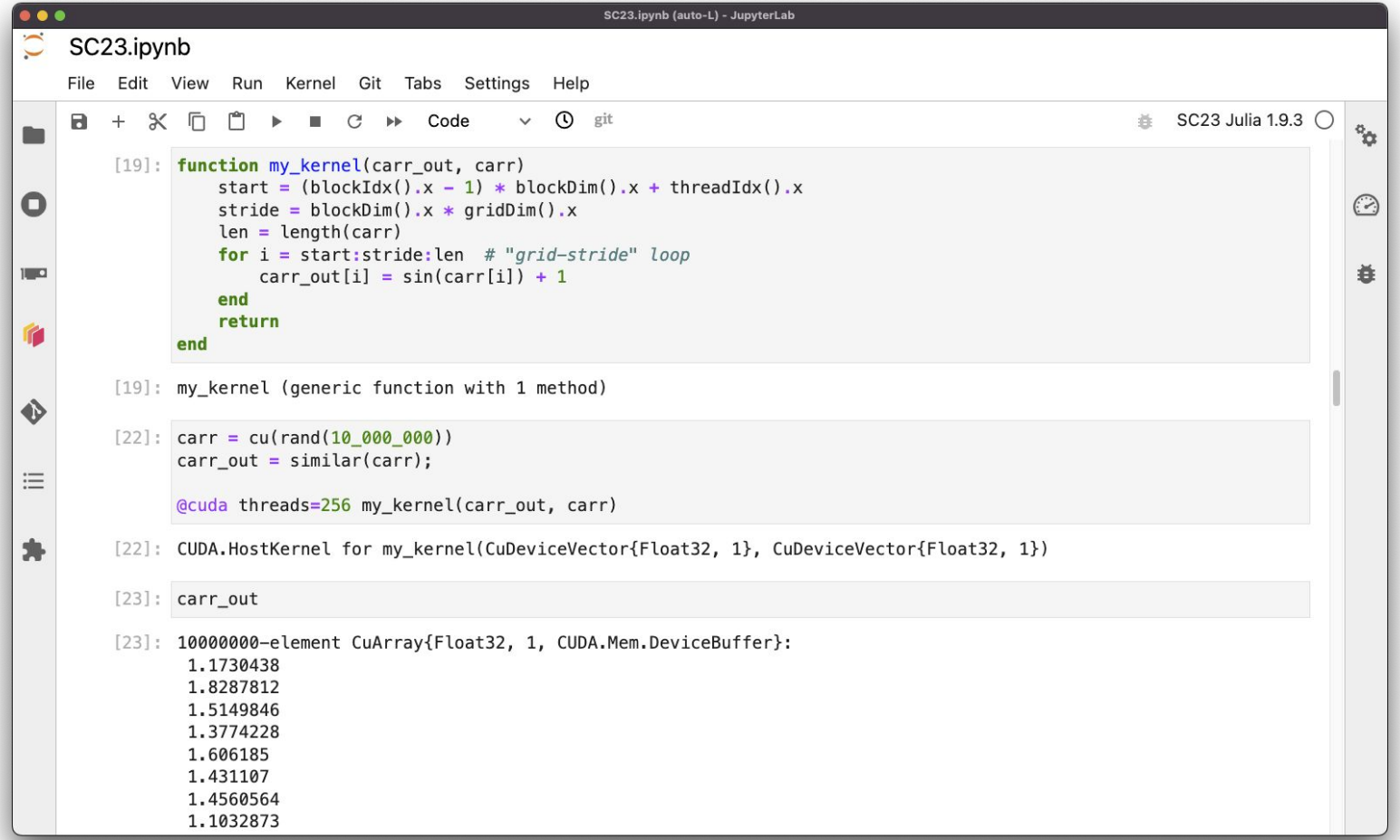
CUDA.jl: Interfacing with Nvidia GPUs



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

NERSC

Write Your Own CUDA Kernels in Julia



NERSC

Write Your Own CUDA Kernels in Julia

Define kernels using Julia functions

Write Your Own CUDA Kernels in Julia

SC23.ipynb (auto-L) - JupyterLab

SC23.ipynb

File Edit View Run Kernel Git Tabs Settings Help

Code git SC23 Julia 1.9.3

```
[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)
```

Launch kernel using the `@cuda` macro

```
[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]: 10000000-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
       1.1730438
       1.8287812
       1.5149846
       1.3774228
       1.606185
       1.431107
       1.4560564
       1.1032873
```

NERSC

(advanced) LLVM + Julia

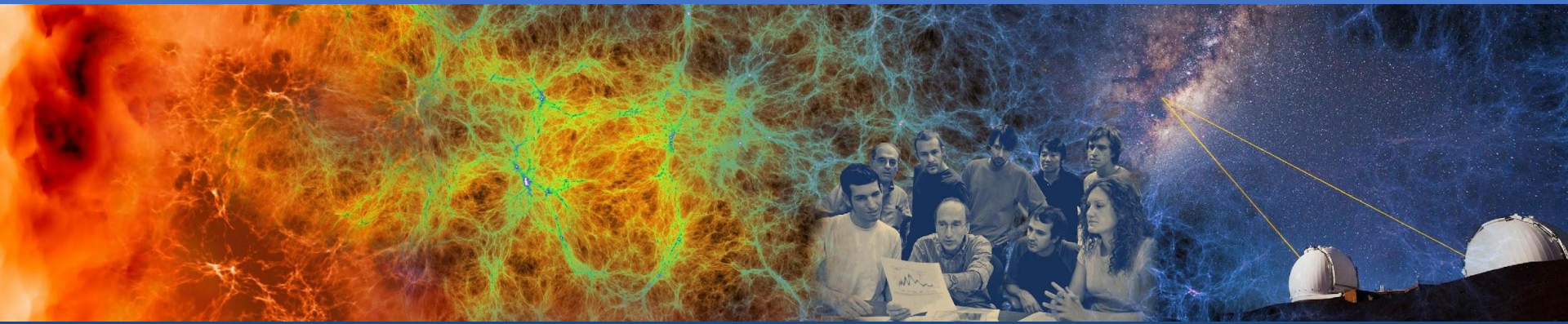
Julia provides interfaces to the LLVM backend.

Eg.:

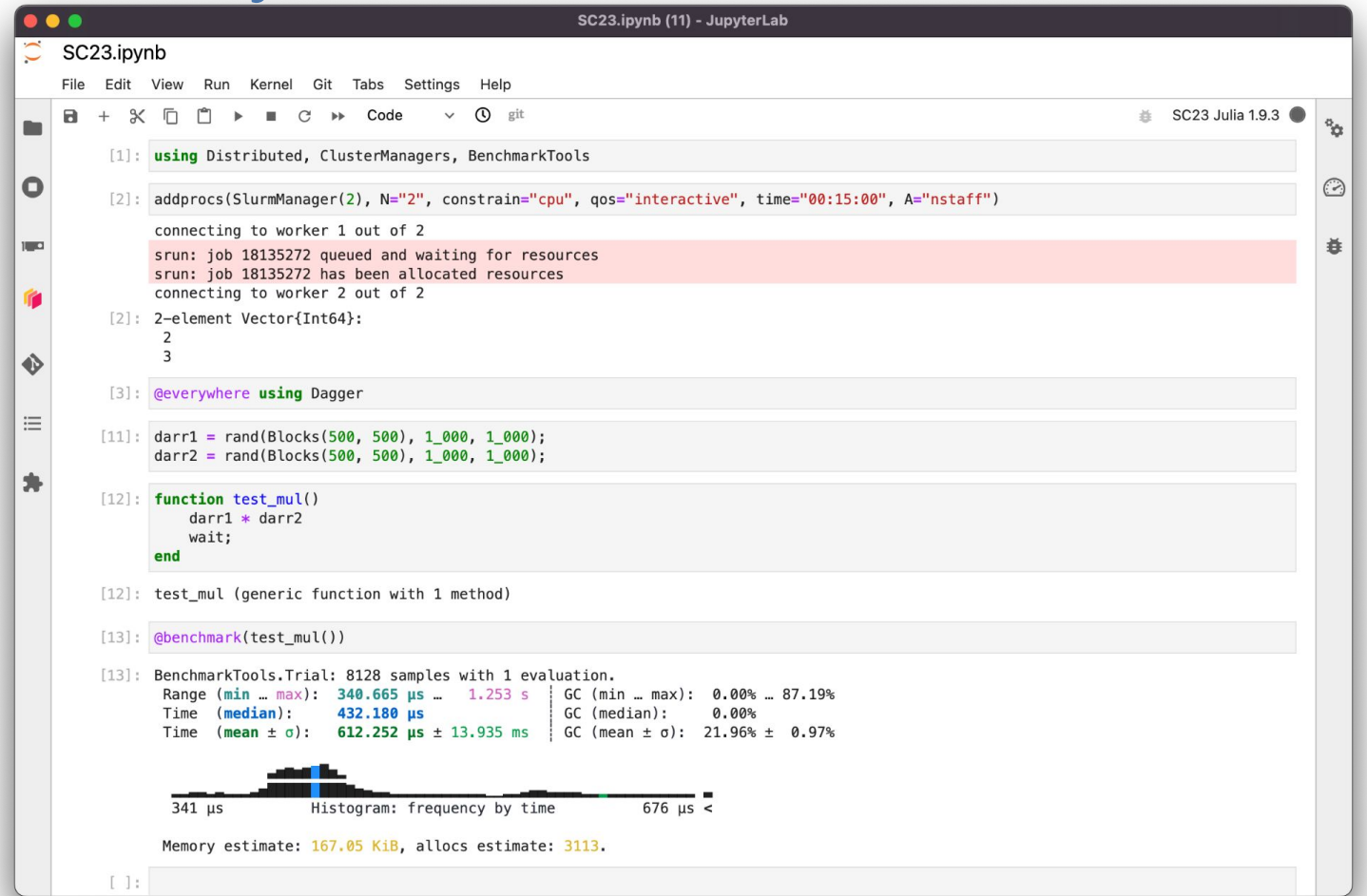
- `loopinfo`
- `llvmcall`

[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::$T)
                Base.llvmcall($ir, $T, Tuple{$T, $T}, a, b)
            end
        end
    end
    @eval function $jlf(args...)
        Base.$jlf(args...)
    end
end
```

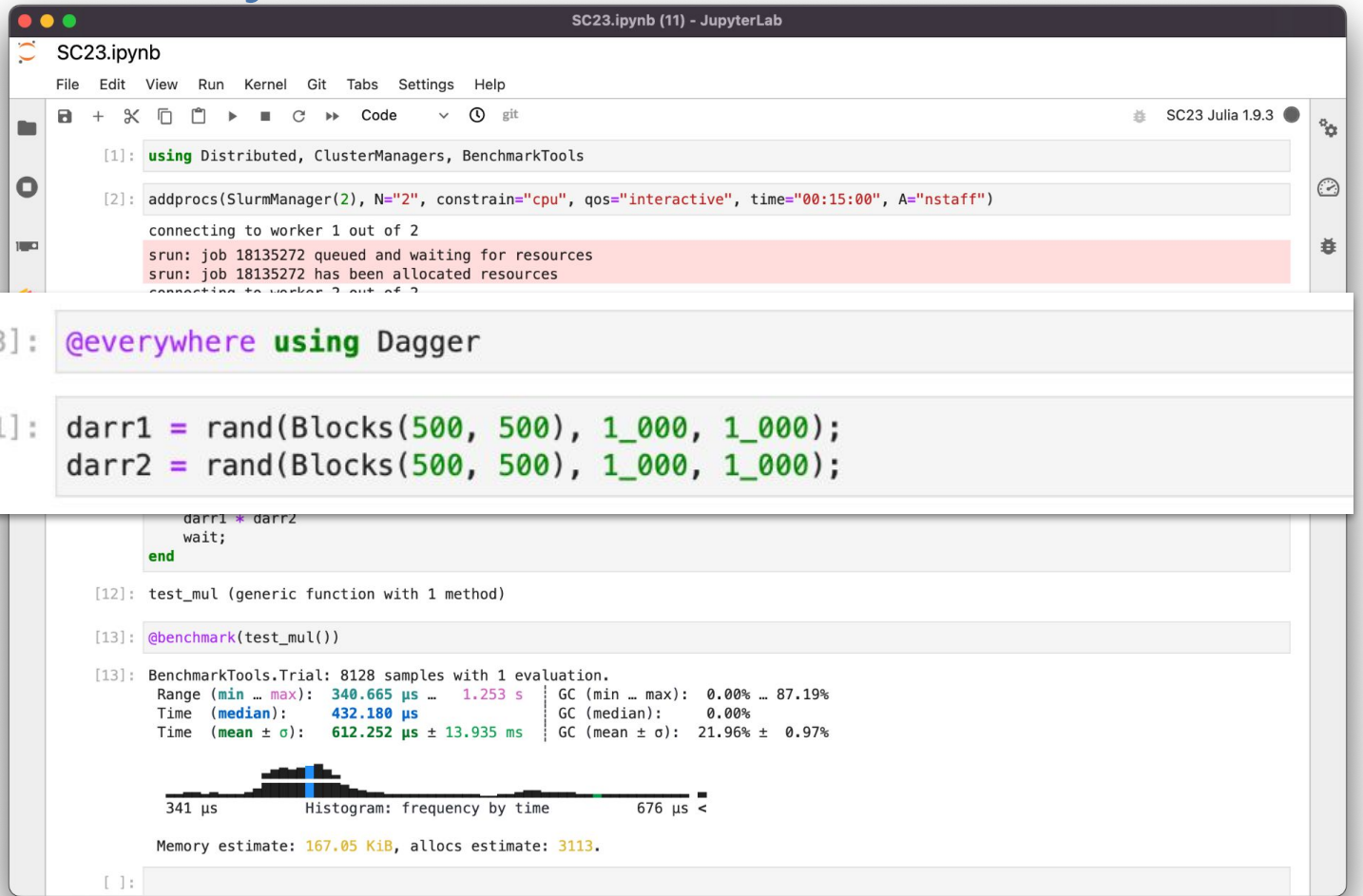
`Distributed.jl` is nice, but…



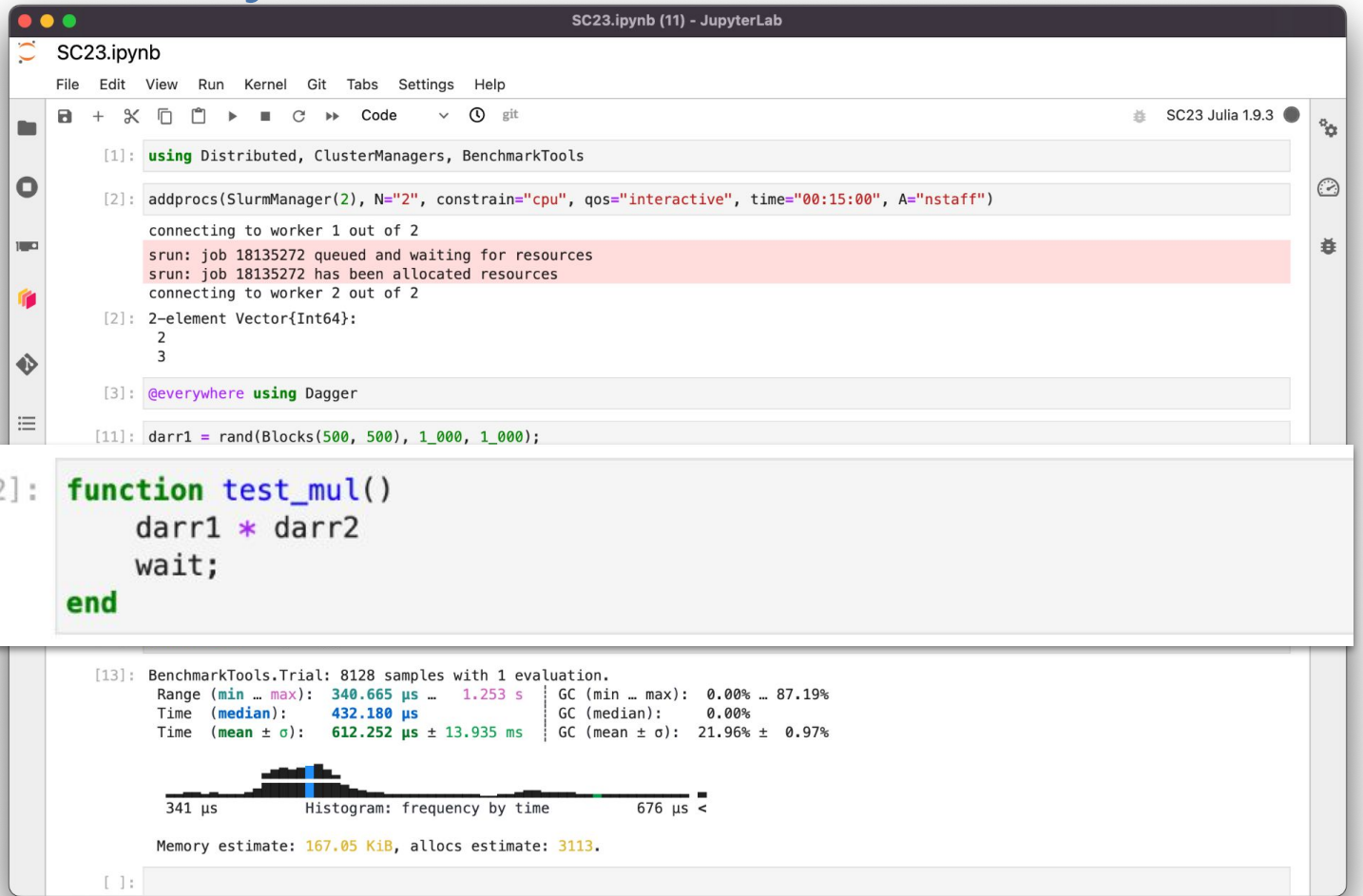
Dagger.jl: Easy work distribution



Dagger.jl: Easy work distribution



Dagger.jl: Easy work distribution



Dagger.jl: Easy work distribution

SC23.ipynb (11) - JupyterLab

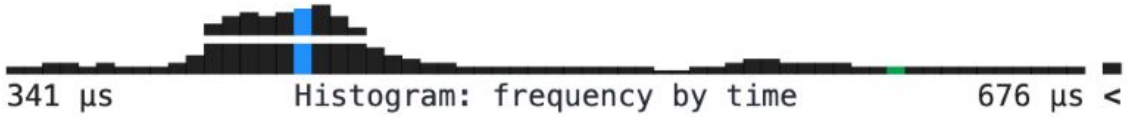
SC23.ipynb

File Edit View Run Kernel Git Tabs Settings Help

Code git 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);
```

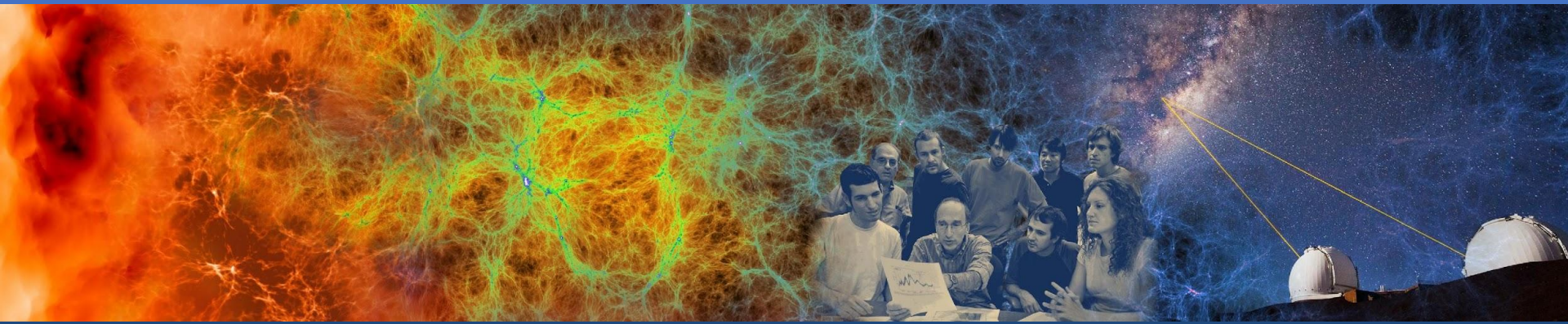
```
[13]: @benchmark(test_mul())

[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

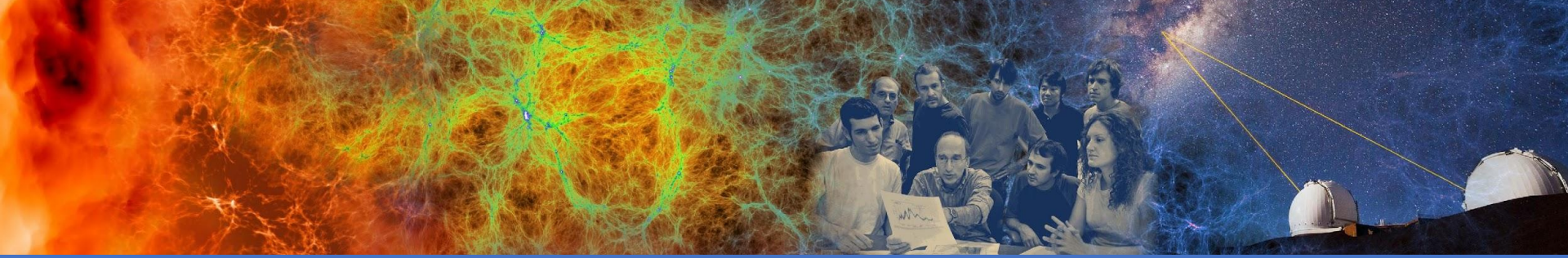
NERSC

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

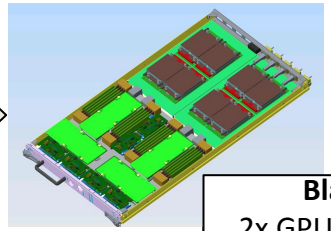


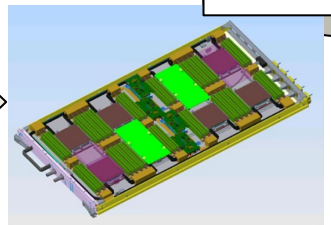
Extra Slides

Perlmutter system configuration

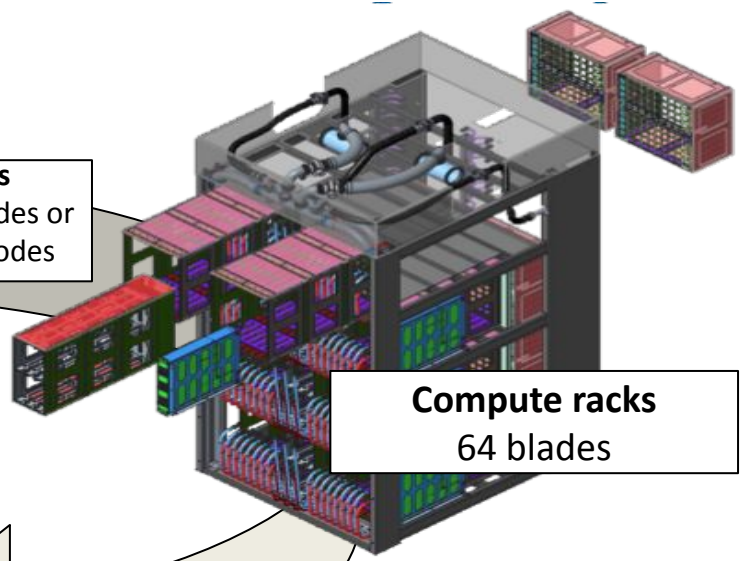
NVIDIA "Ampere" GPU Nodes
4x GPU + 1x CPU
40 GiB HBM + 256 GiB DDR
4x 200G "Slingshot" NICs

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





Blades
2x GPU nodes or
4x CPU nodes

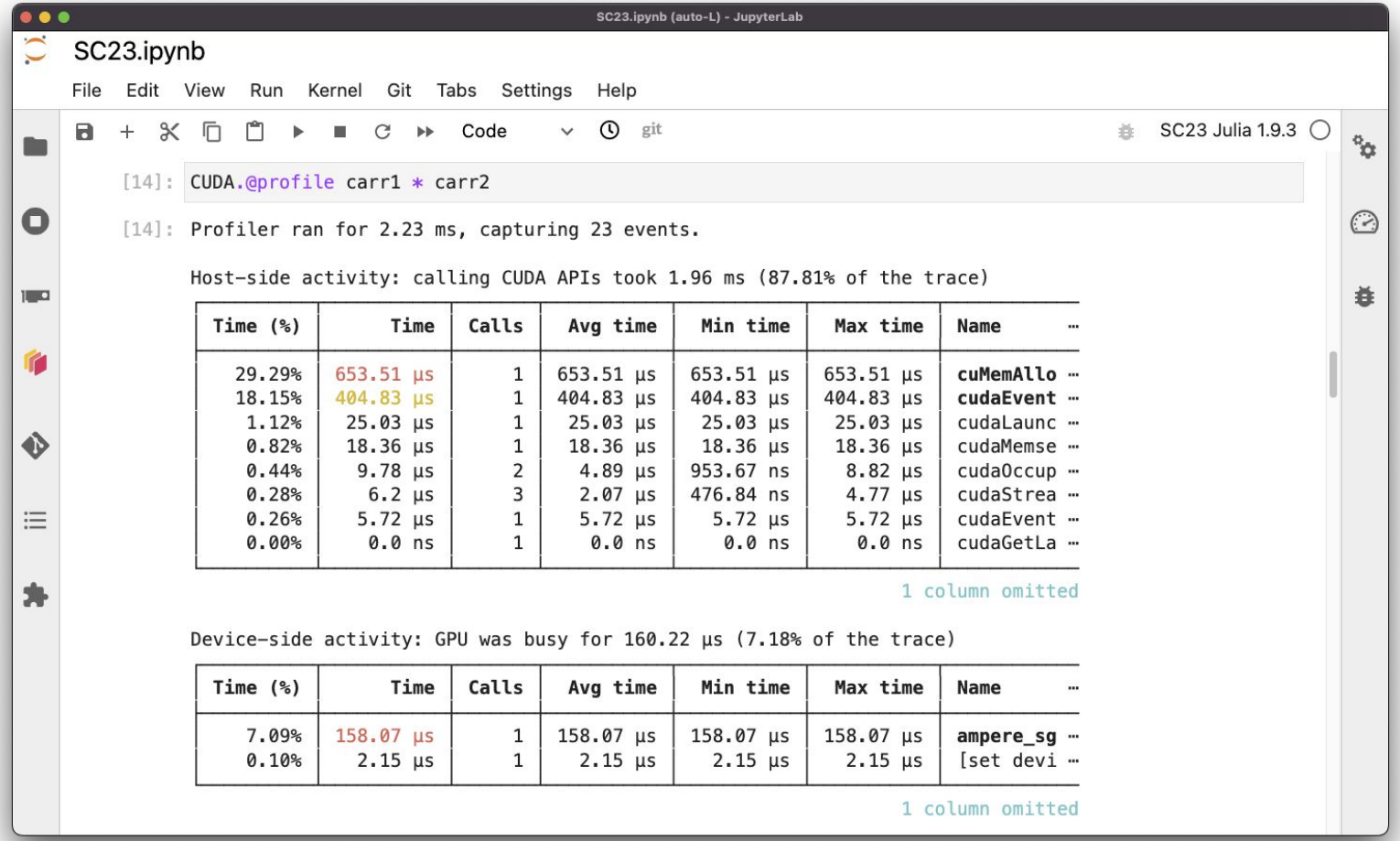


Compute racks
64 blades

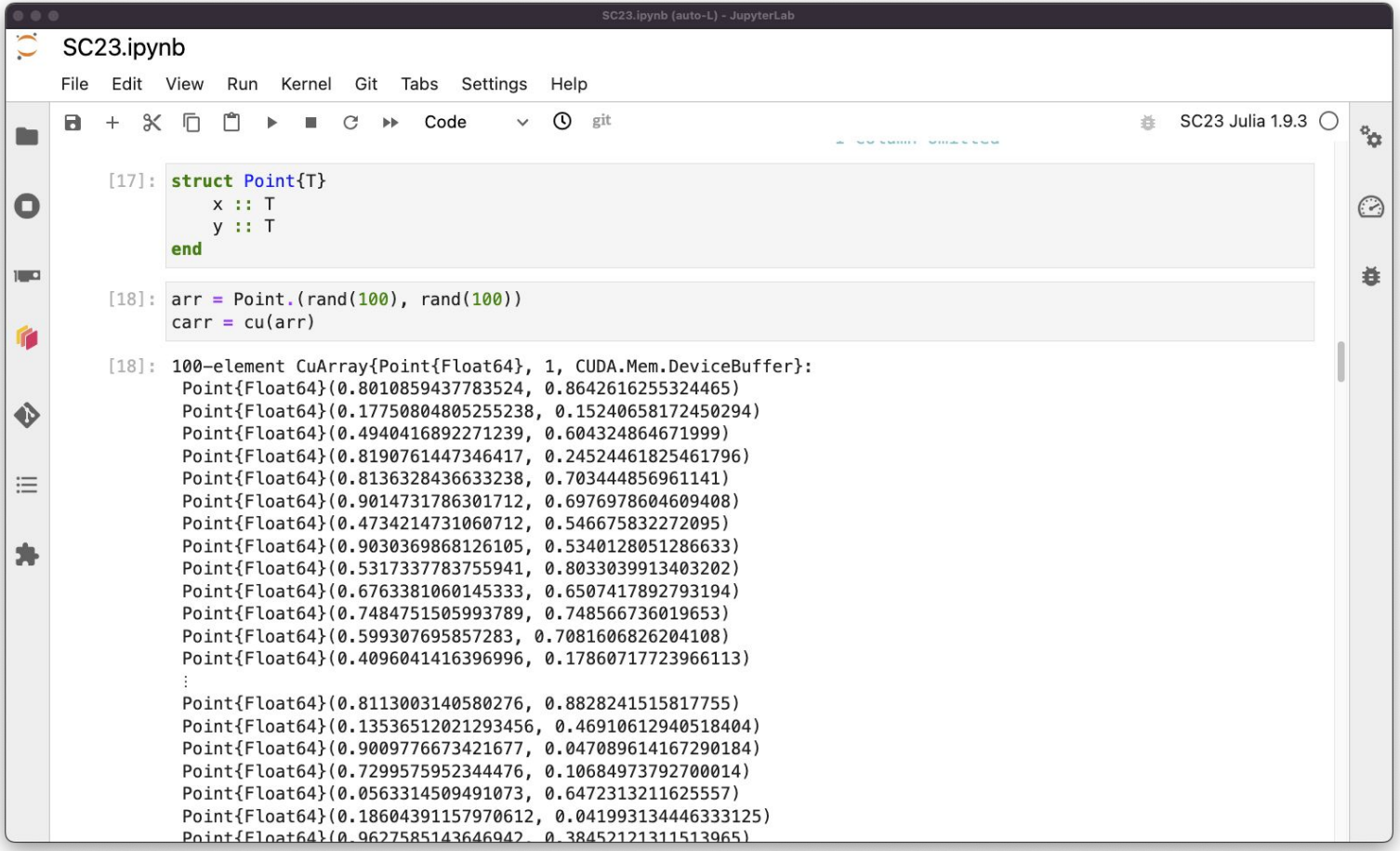
Centers of Excellence
Network
Storage
App. Readiness
System SW

Perlmutter system
GPU racks
CPU racks
~6 MW

CUDA.jl provides detailed profiling interface

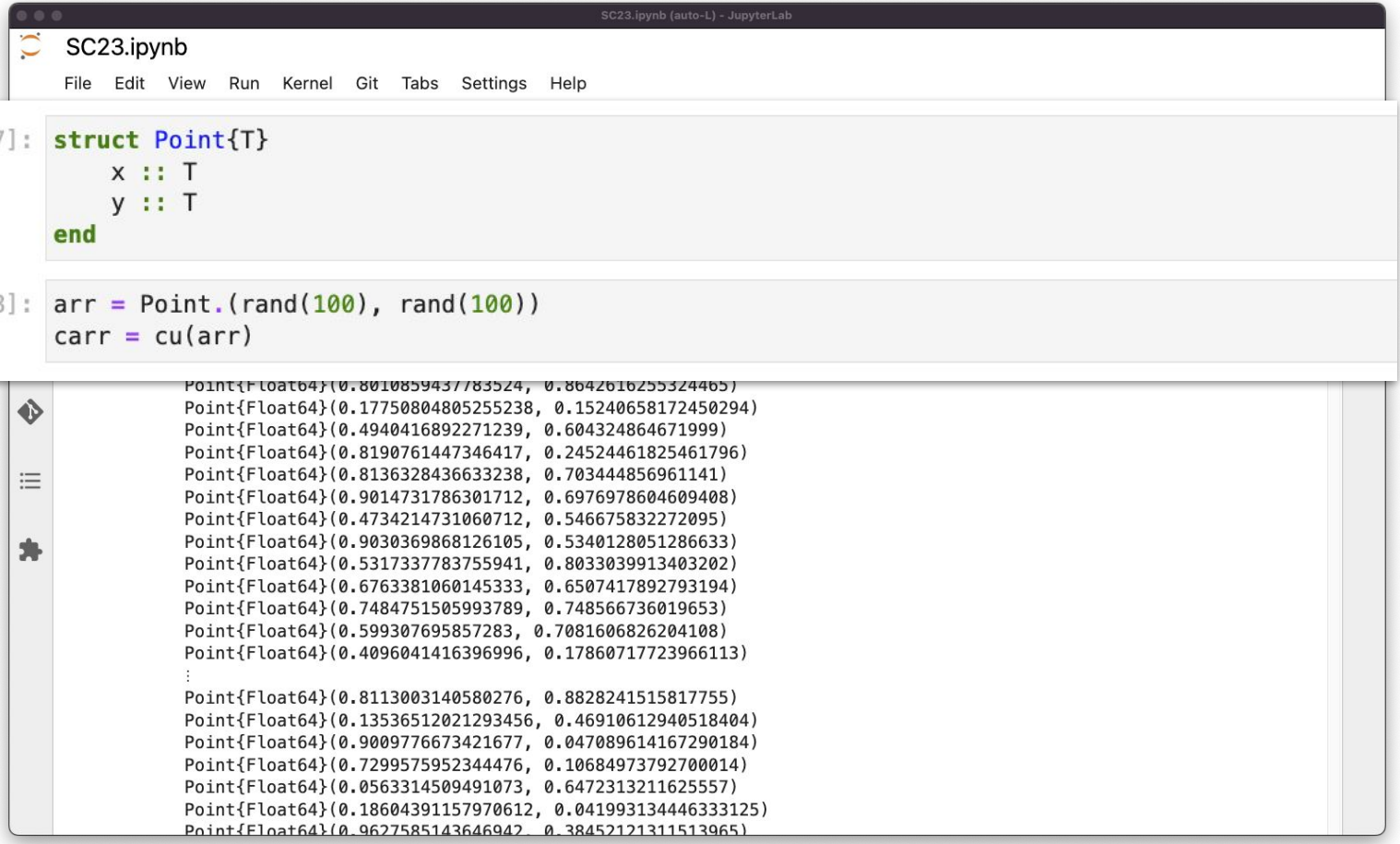


CUDA.jl is compatible with Structs

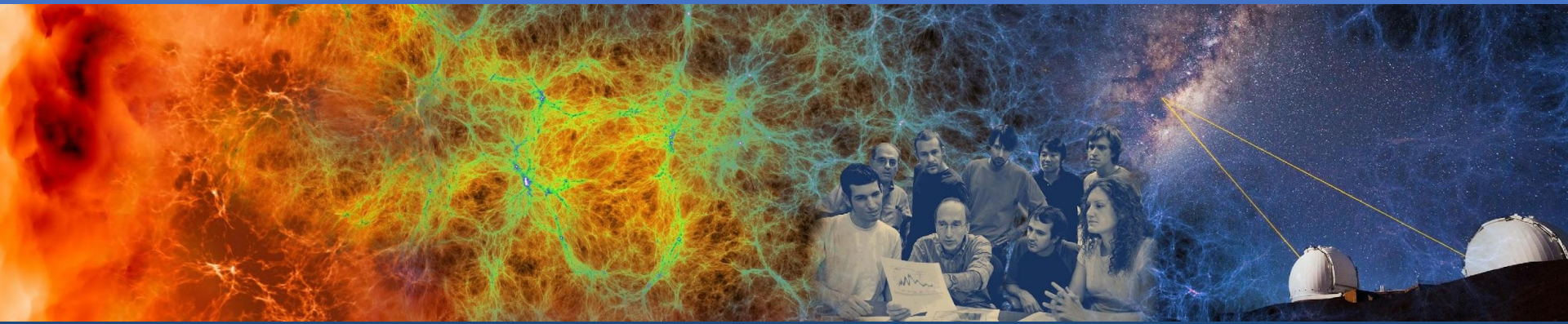


CUDA.jl is compatible with Structs

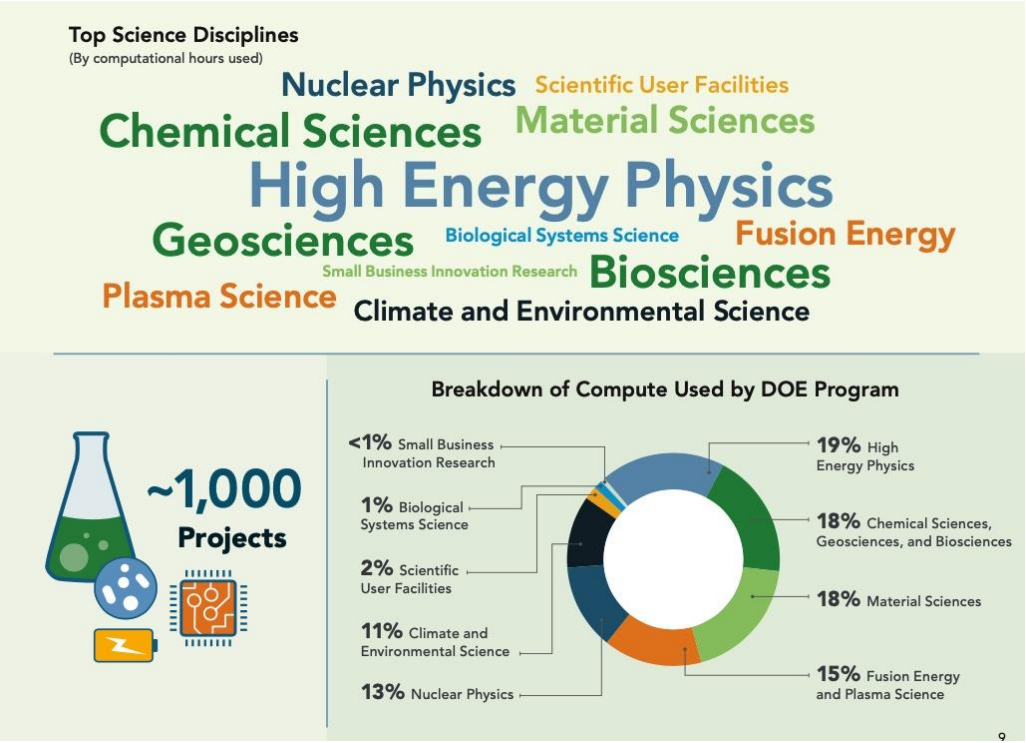
Julia converts struct to cuda-compatible type

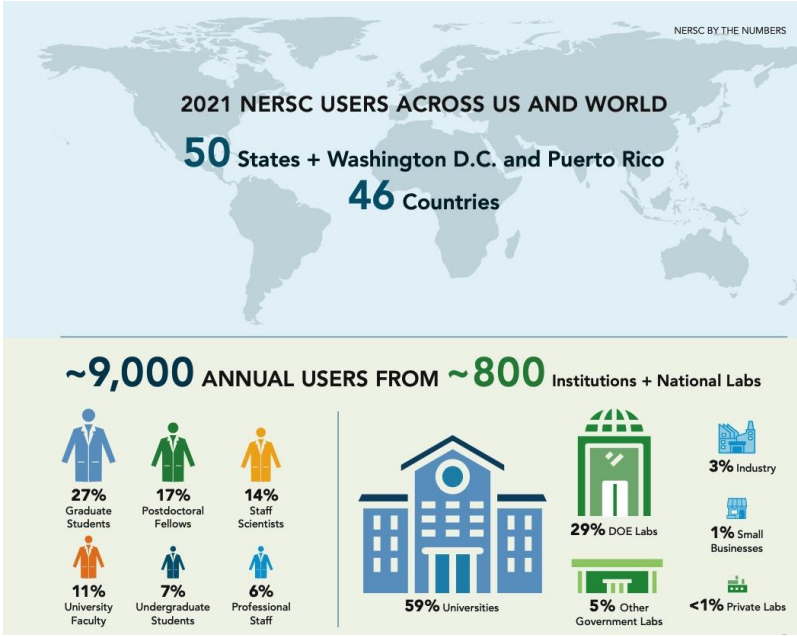


Why does NERSC care about Julia?



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

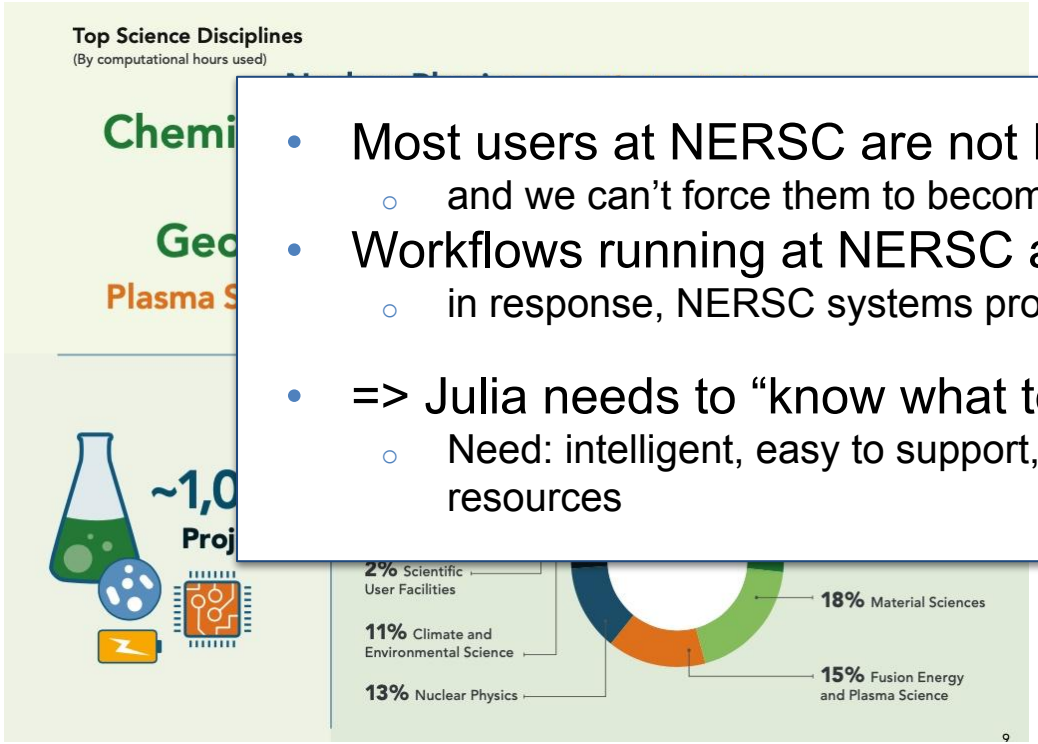




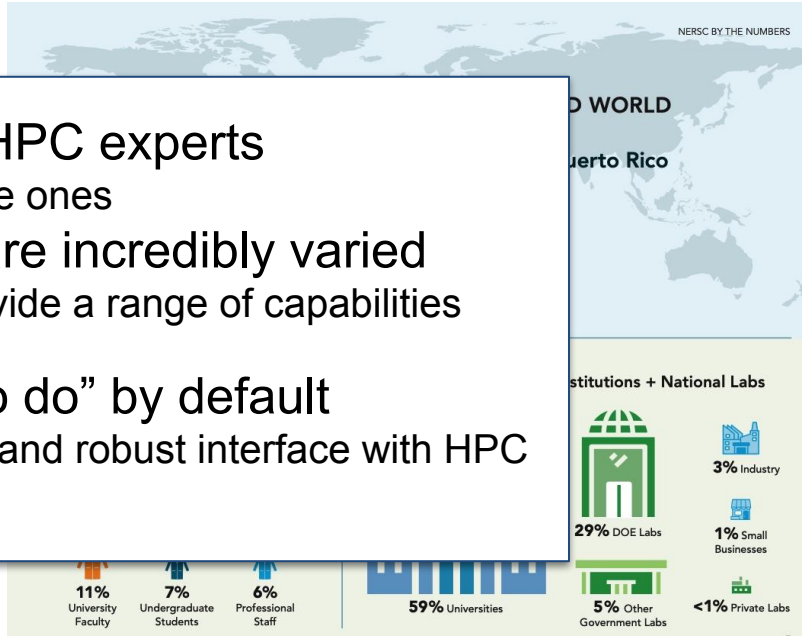
>2,000

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