JuliaCon 2019

Parallel Computing Workshop

Welcome!

Parallel Computing workshop

- Download JuliaPro version 1.1.1.1
    - https://juliacomputing.com/products/juliapro
- Install it!
- Download the workshop materials
    - github.com/mbauman/ParallelWorkshop2019
- Open Julia Pro, go to File -> Add Project Folder… and select the downloaded folder

Troubleshooting tips

- Ensure running Julia 1.1.

- Running on JuliaPro works best

- Make sure you're in the "Project folder" specifically for the downloaded folder (not one level higher)

- If running on vanilla Juno + Julia, run

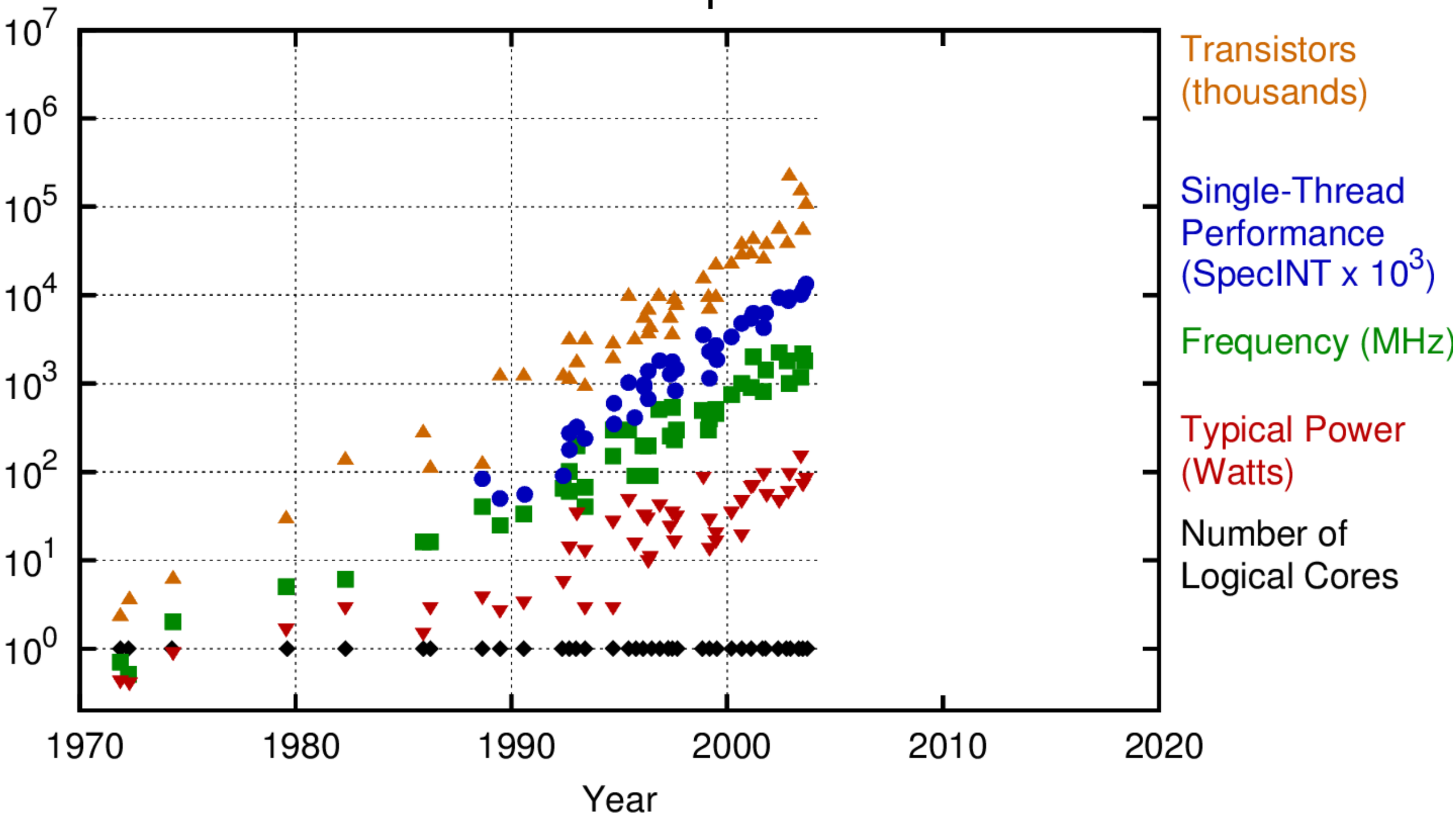
```
]activate
]rm CredentialsHandler
]instantiate
```

Goals

- Understand modern parallel architectures
- Write and run a multithreaded algorithm on your own computer
- Learn how to structure your program to avoid race conditions
- Treat your computer like a cluster and run multi-process code on it

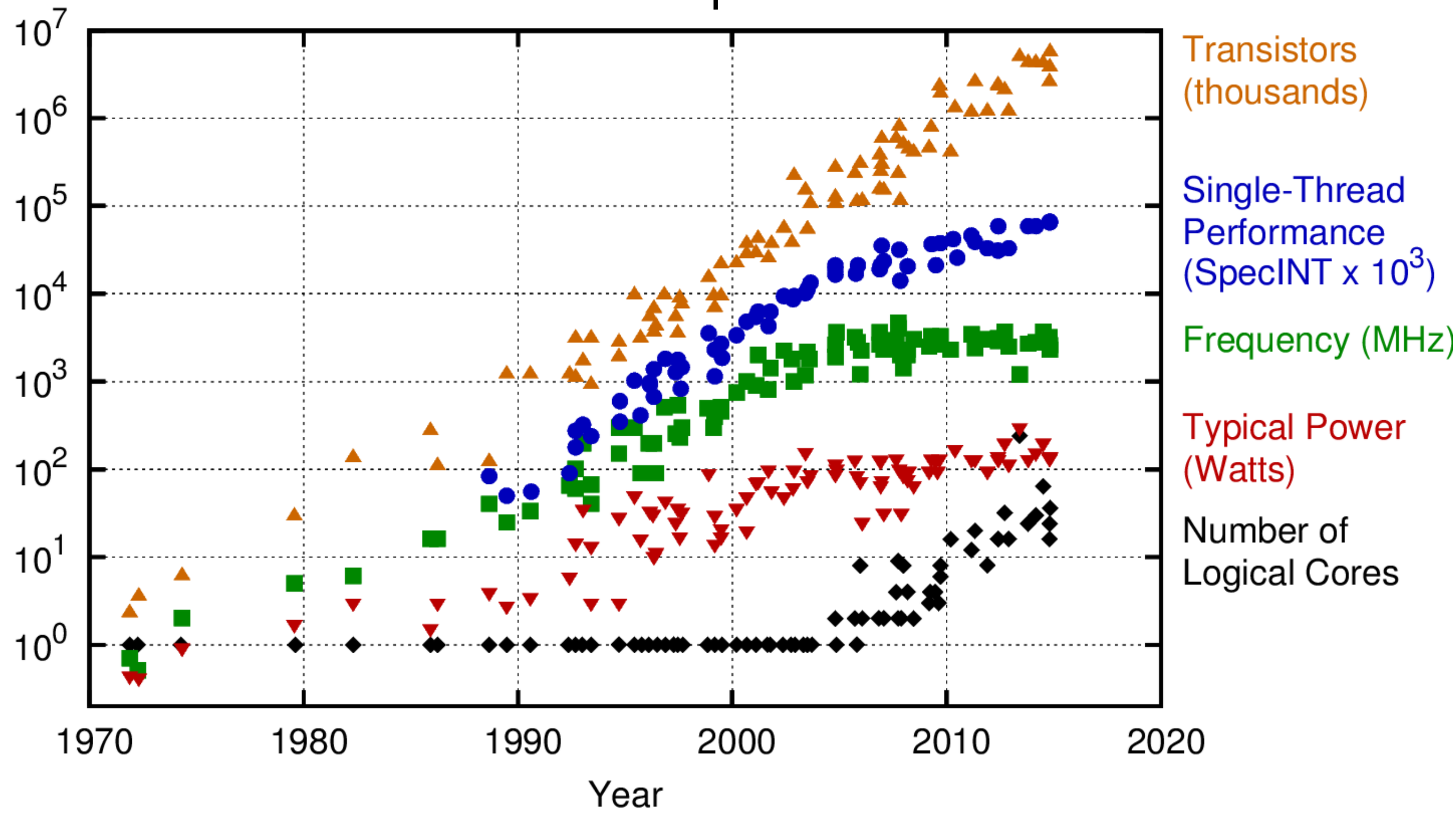
What is happening to our computers?

Historical microprocessor trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Historical microprocessor trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Parallelism is required for best performance

Computers today:

- Have single processor cores that can do more than one thing at once
- Have multiple cores/processors (up to 50 cores on a chip, multiple chips)
- Can network multiple computers together
- May have specialized compute hardware (GPU, TPU)

Fastest supercomputers in the world

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371

Fastest supercomputers in the world

Rank	Site	System				
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power Sys AC922, IBM POWER9 22C NVIDIA Volta GV100, Dual Mellanox EDR Infiniband IBM				
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power Syste S922LC, IBM POWER9 22 NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox				
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371

4,608 nodes
2 processors per node
22 cores per processor
6 GPUs per node

Parallel Challenges

- Chasing peak performance
  Need a fast language for biggest benefit

- We learn to reason about & write programs serially
  Need to safely express parallelism

- Often dovetails with large datasets
  Need to consider data movement costs

- Vast array of parallel hardware configurations
  Need to pick the strategy that fits your system

Parallel Strategies

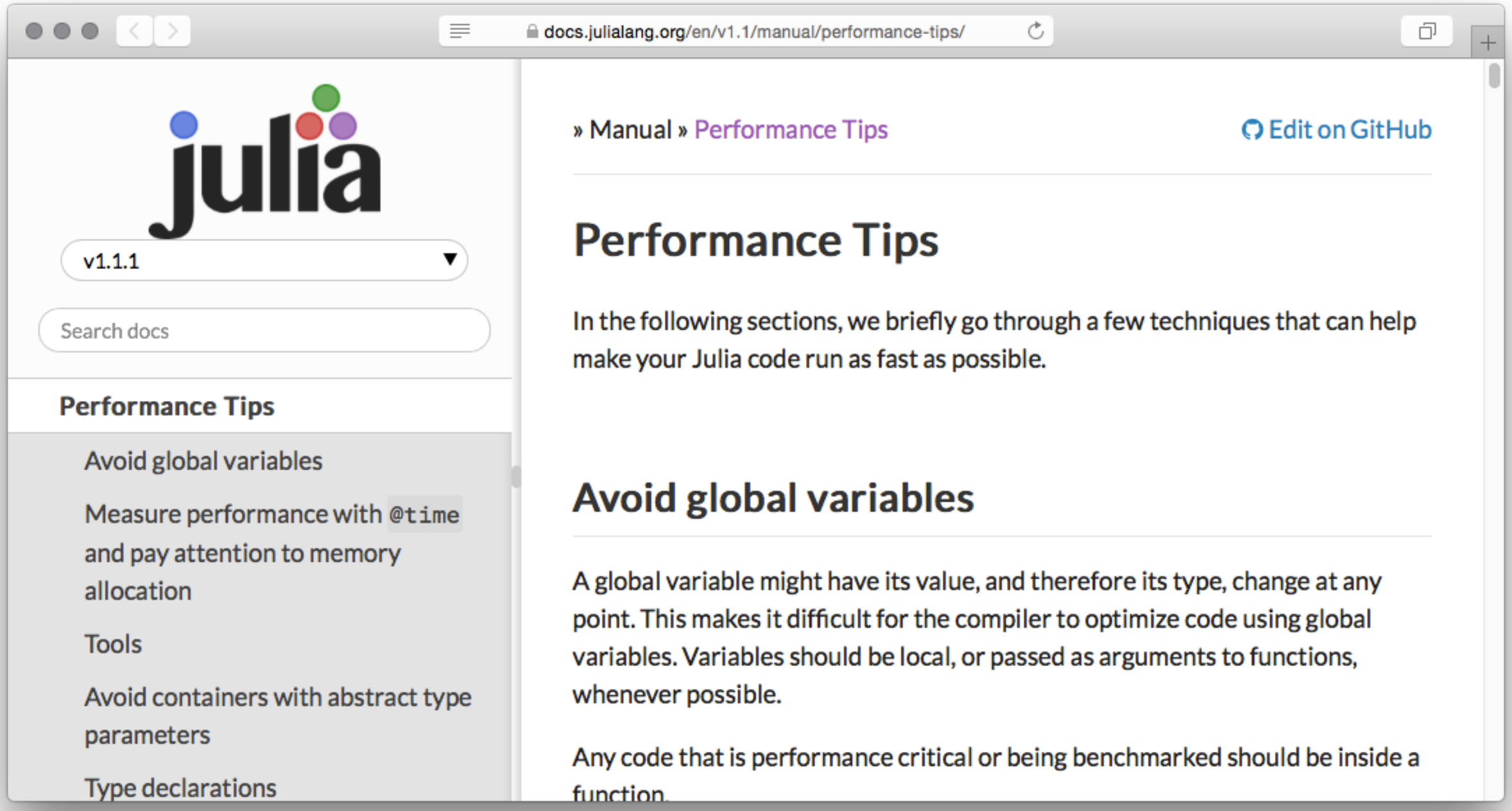
- Making the most of one core:
  high performance Julia and SIMD

- Making the most of one computer:
  multithreading

- Making the most of multiple computers/clusters:
  distributed computing

- Enabling external accelerators:
  GPUs and TPUs

High Performance Julia



High Performance Julia

Single-instruction, multiple-data (SIMD)

$$
\begin{aligned}
x_1 + y_1 &\rightarrow z_1 \\
x_2 + y_2 &\rightarrow z_2 \\
x_3 + y_3 &\rightarrow z_3 \\
x_4 + y_4 &\rightarrow z_4
\end{aligned}
$$

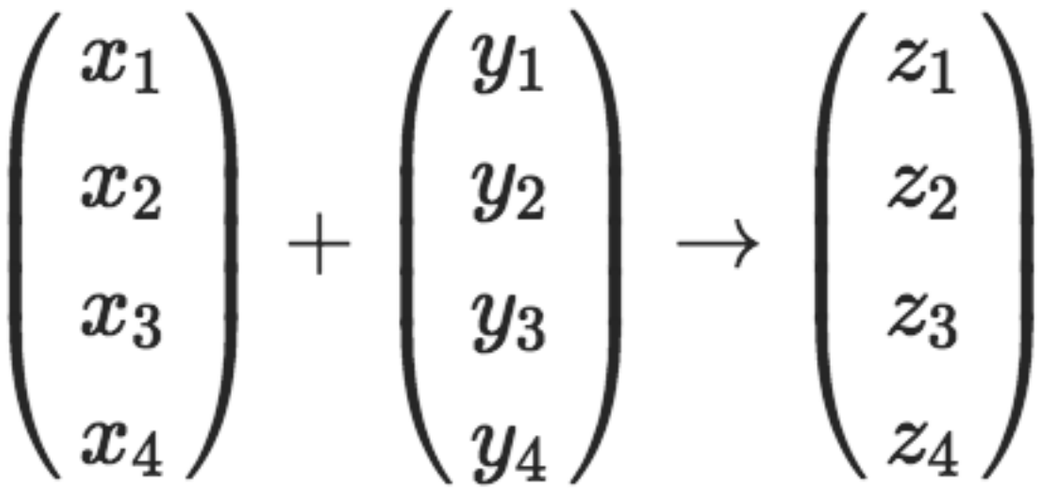
four instructions

$$
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \rightarrow \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}
$$

one instruction!

High Performance Julia

Single-instruction, multiple-data (SIMD)

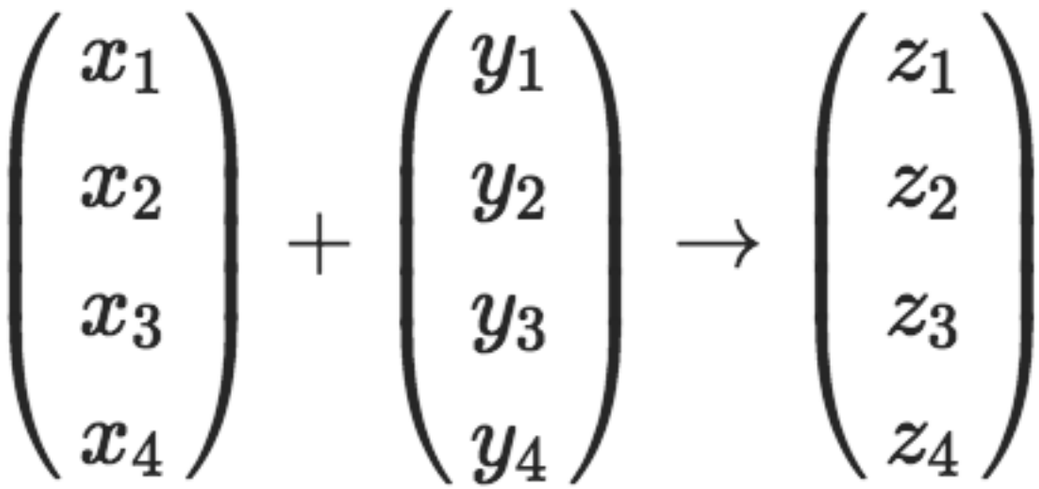



```
1 for x in xs
2     # ...
3 end
```

- Happens automatically!

High Performance Julia

Single-instruction, multiple-data (SIMD)

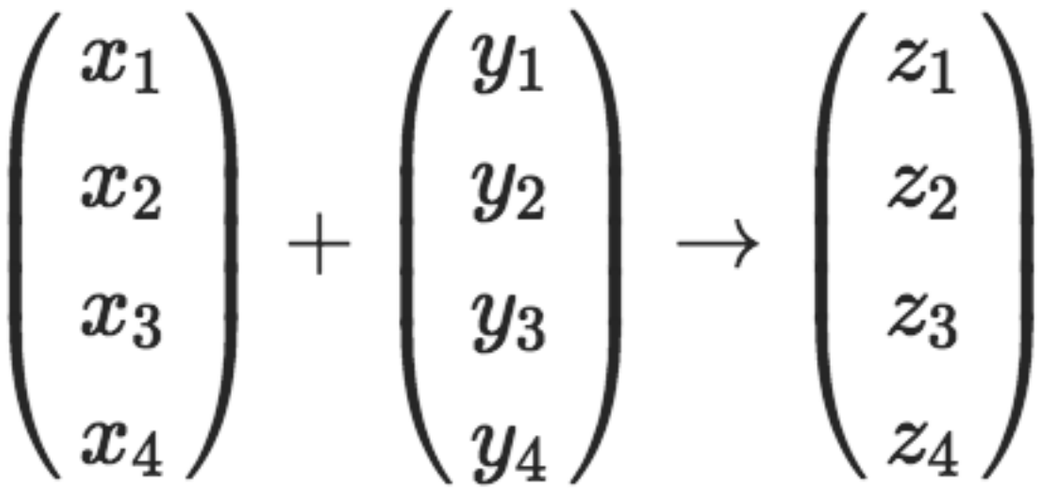



```
1 for x in xs
2     # ...
3 end
```

- Happens automatically (if it's safe)!
  - Requires "straight line" code: no if statements, `@inbounds` array accesses, no breaks, only call "small" functions, etc.
  - Limited to basic processor instructions (simple arithmetic)

High Performance Julia

Single-instruction, multiple-data (SIMD)




```
1 @simd for x in xs
2     # ...
3 end
```

- `@simd` can help it happen more frequently
  - Allows floating point re-associativity
- Using smaller data types can increase parallelism
  - Up to 512 bits can process 32 Float32s at a time!

High Performance Julia

- Understanding data movement costs

	Actual cost	Scaled "human" cost
One CPU Cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Main memory access	~100 ns	4 min
NVMe SSD I/O	~25 µs	17 hours
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet call: SF to NYC	65 ms	5 years

https://www.prowesscorp.com/computer-latency-at-a-human-scale

Multithreading

Hands on!

Parallel Algorithm Design

Demo

Tasks

Quick demo

Distributed Computation

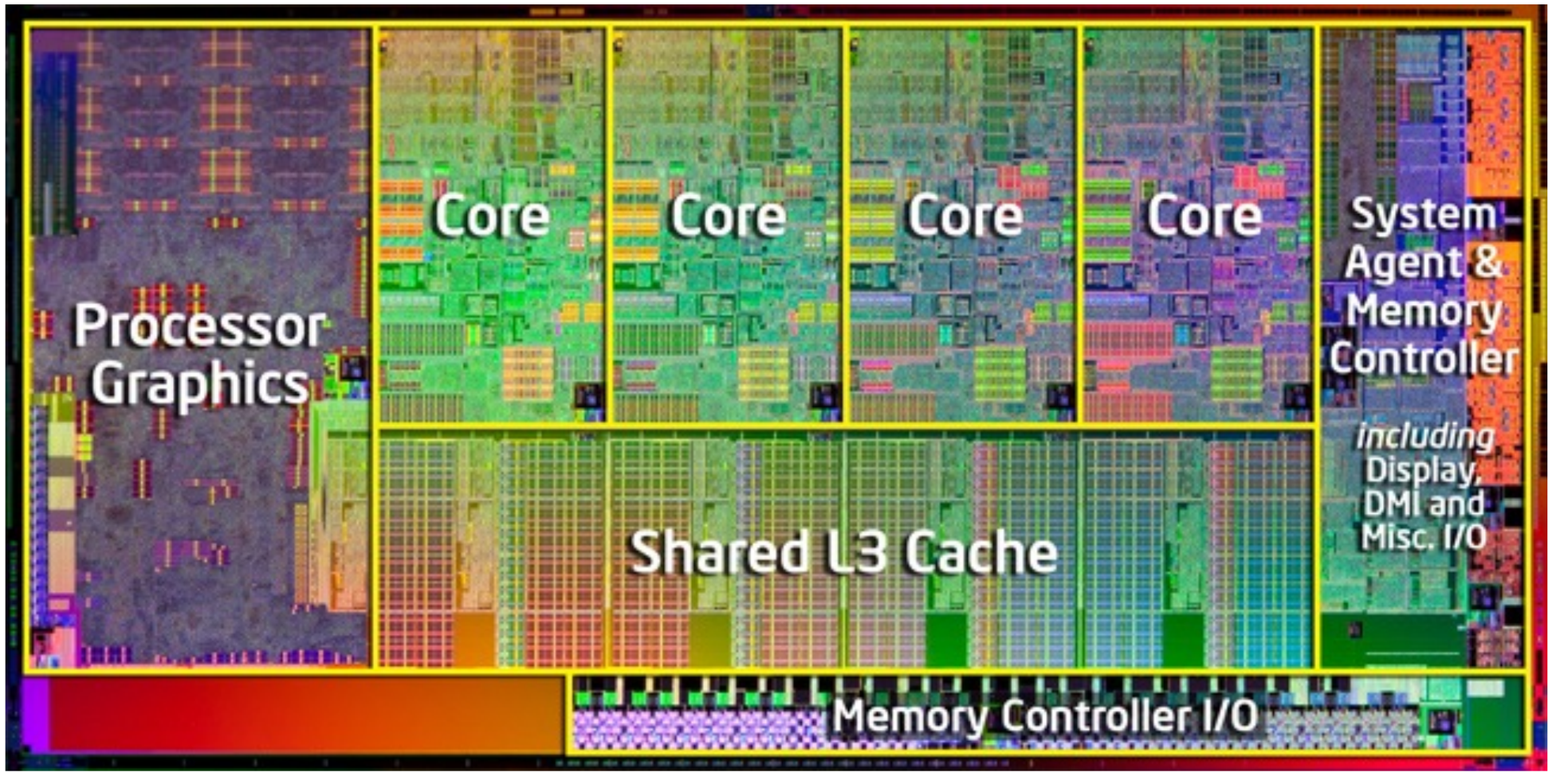
Towards clusters!

Mental model

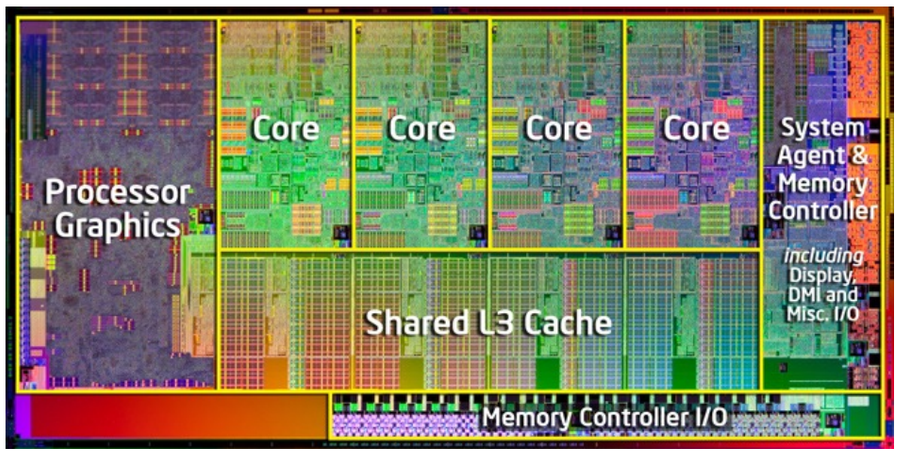
Distributed demo!

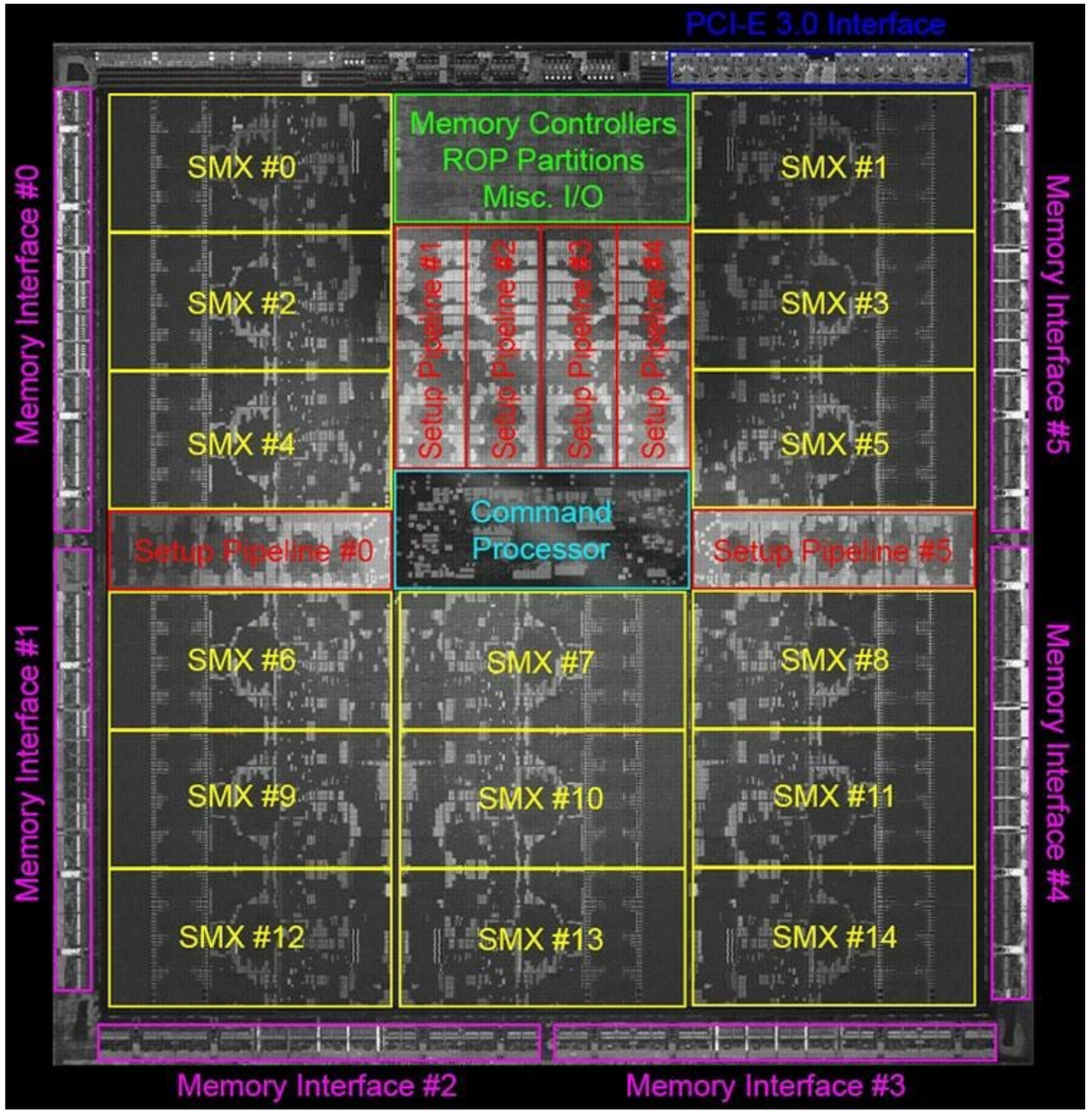
GPUs and more

What's a CPU look like?



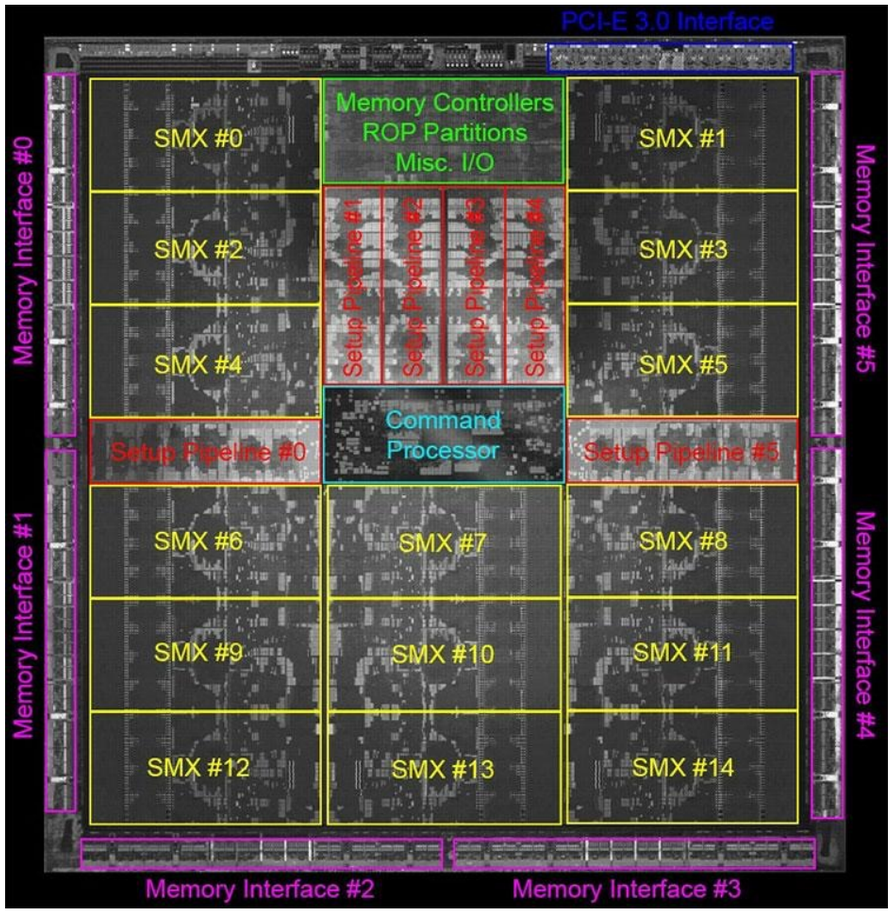
What's a GPU look like?





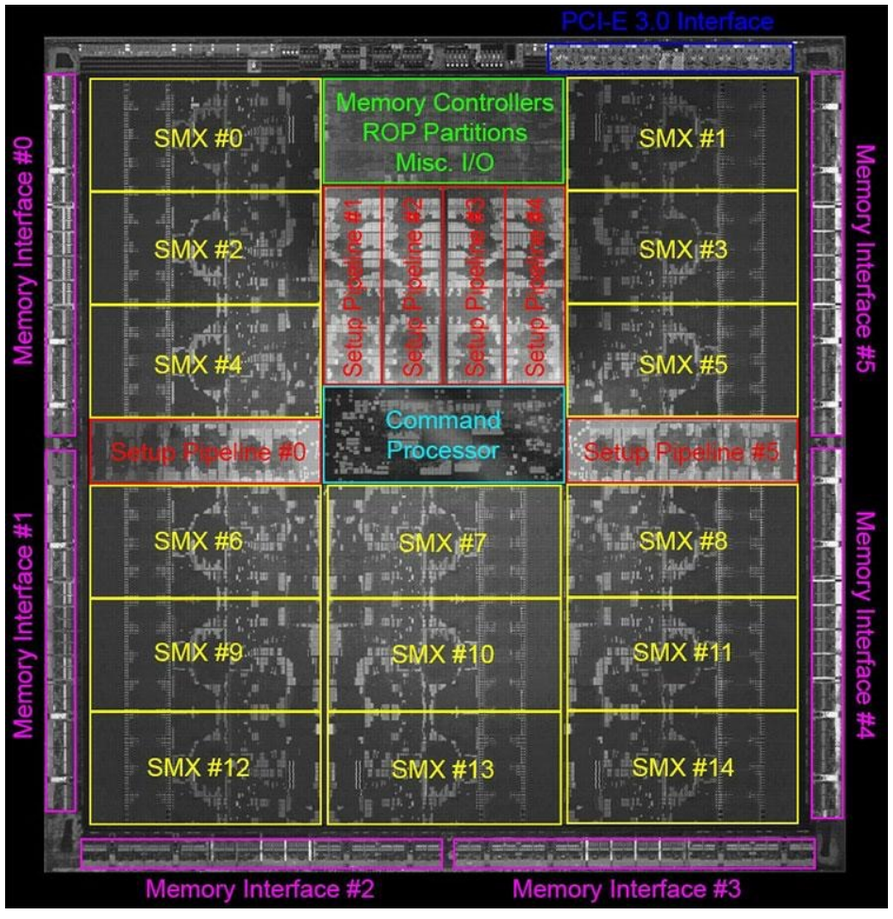
NVidia GK110

What's a GPU look like?

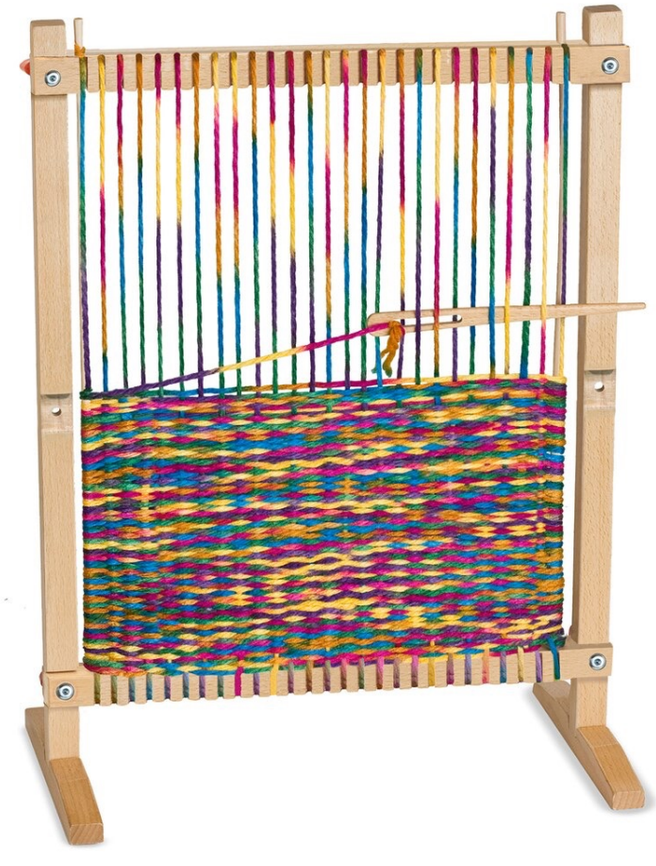


- 15 "multiprocessors"
- Up to 6 warps per multiprocessor

What's a GPU look like?

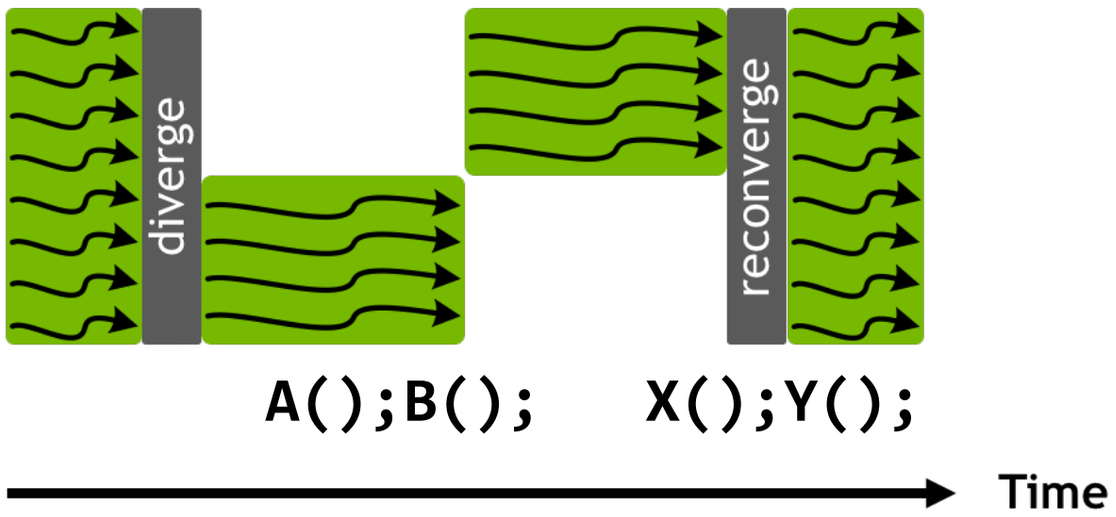


- 15 "multiprocessors"
- Up to 6 warps per multiprocessor
- Each warp is 32 threads



Warp programming

```
1  if threadid() <= 4
2      A()
3      B()
4  else
5      X()
6      Y()
7  end
8  Z()
```



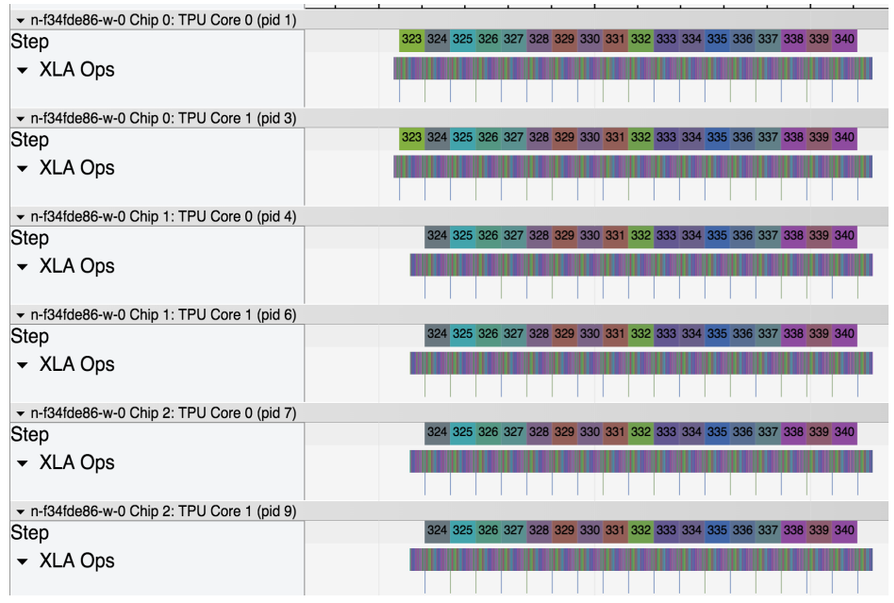
GPUs: Using JuliaGPU to make it easy

```
julia> cu(rand(100))
100-element CuArray{Float32,1}:
 0.5351649
 0.93891734
 0.6757371
 0.4985685
 0.9094223
 0.8695512
 0.0144788
 0.77135414
 0.612888
```

- Move data to the GPU
- CuArray is specialized to operate on the GPU and favors Float32s
- In particular, you can create your own “GPU kernels” with broadcast fusion
- You can also manually write CUDA code

TPU: the Tensor Programming Unit

Scales to pods (512 TPU cores - 4.3 PF_{16}/s on ResNet50)



Fischer et al. Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs (arXiv:1810.09868)

Julia on TPUs

- Unchanged Language Semantics on TPUs
  - Control Flow
  - Multiple Dispatch
  - Data Structures
- Share code between CPU/GPU/TPU
- Integration with native distributed computing facilities (WIP)
- Goal: Make TPU pods and traditional HPC systems look the same to the programmer. Seamless retargeting.

```
julia> X = randn(1000, 1000)
[...]
julia> X*X                    # Runs on CPU
1000×1000 Array{Float64,2}:
[...]
```

```
julia> X = randn(1000, 1000) |> gpu
[…]
julia> X*X                    # Runs on GPU
1000×1000 CuArray{Float64,2}:
[...]
```

```
julia> X = randn(1000, 1000) |> tpu
[…]
julia> X*X                    # Runs on TPU
1000×1000 XRTArray{Float64,(1000, 1000),2}:
[...]
```

Happy parallel computing
Happy parallel computing
Happy parallel computing
Happy parallel computing