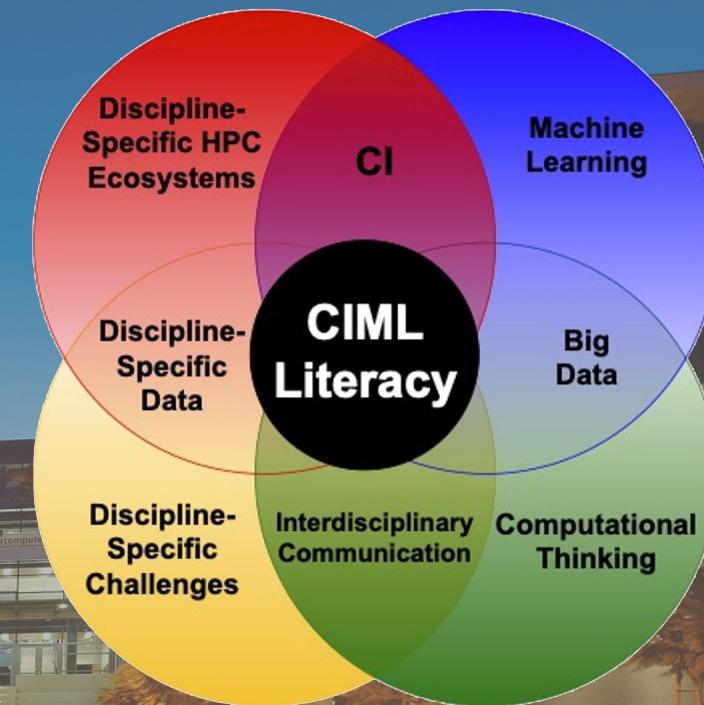


# CIML Summer Institute: GPU Computing – Hardware architecture and software infrastructure

Andreas W Götz

San Diego Supercomputer Center  
University of California, San Diego

Tuesday, June 25, 2024, 1:30 pm to 3:00 pm, PDT



# Outline

## We will cover the following topics

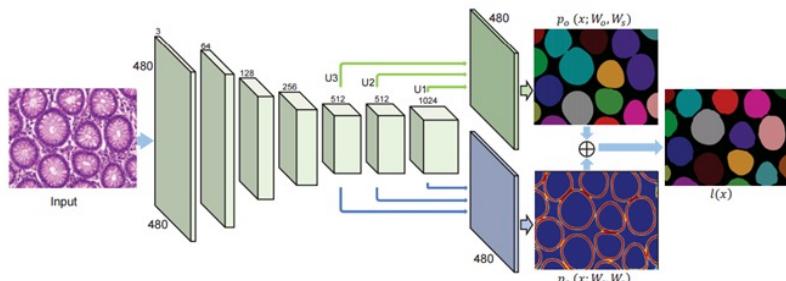
- Brief Intro on current ML trends
- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - CUDA intro
  - OpenACC intro
- SDSC Expanse GPU nodes
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

# Machine Learning in Computational Sciences

AI and ML have been around for a long time

Why the current interest?

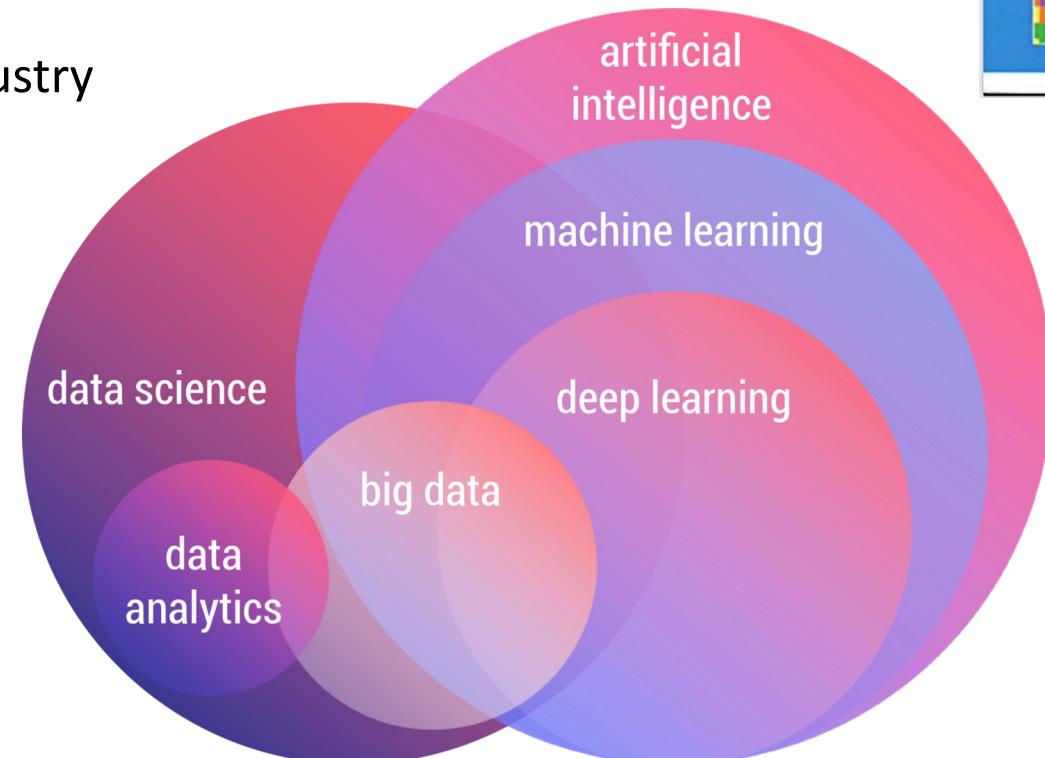
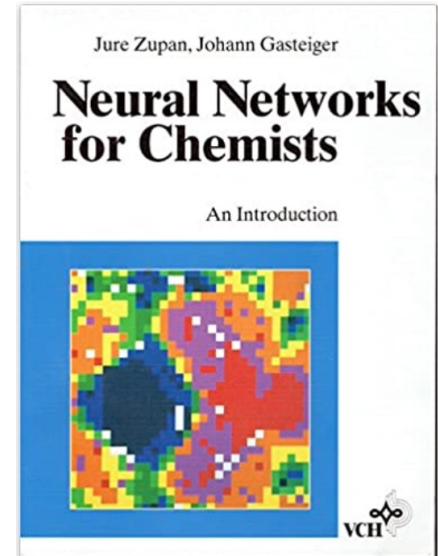
- Flood of available data (in particular with internet)
- Breakthroughs in theory and algorithms (deep learning, generative AI, ...)
- Increasing interest and support from industry
- **Increase in computational power**



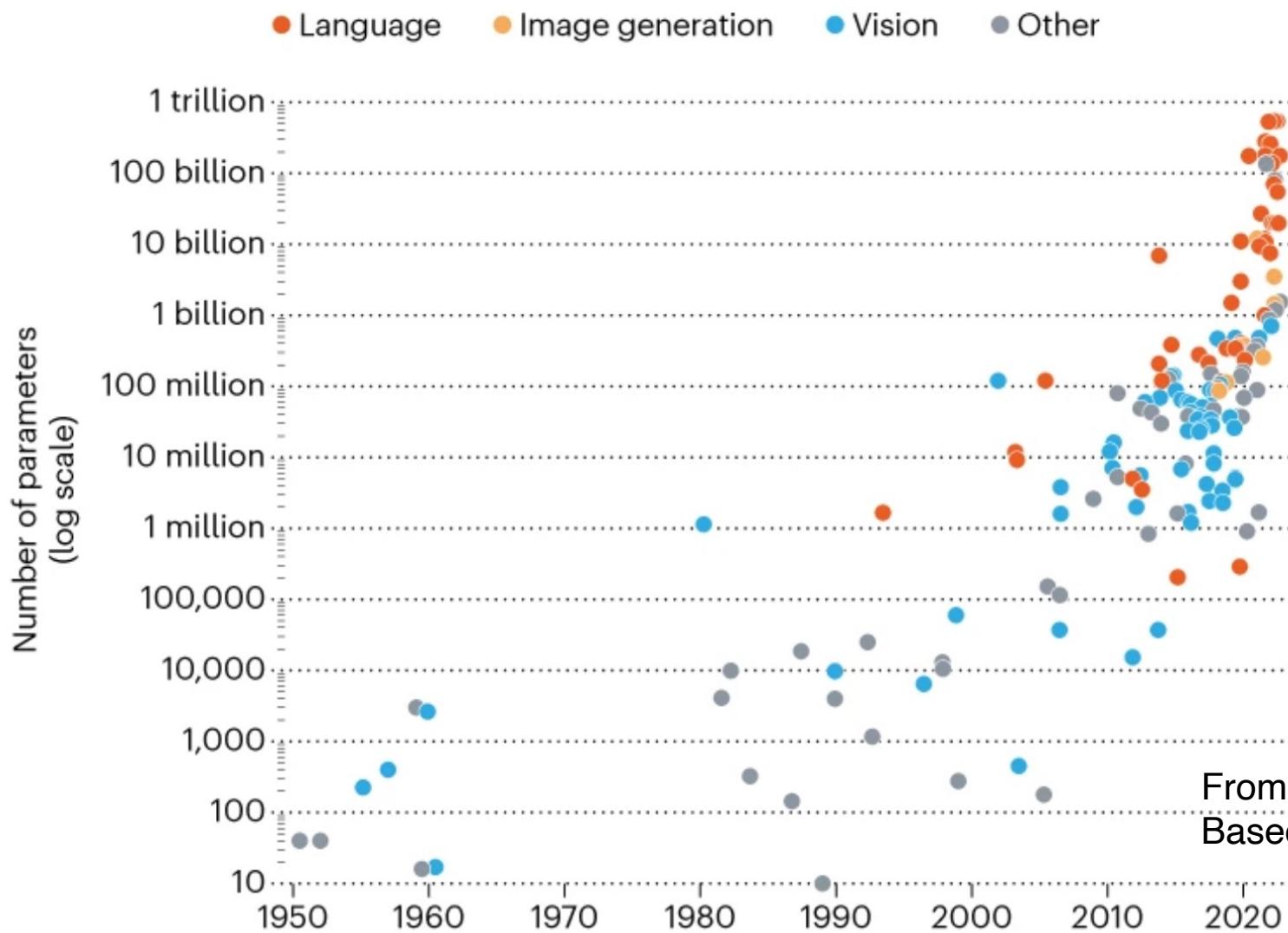
Adoption of machine learning

- Practically all science fields
- Augment modeling and simulation

Neural Networks  
for Chemists,  
published in 1993



# Drive to Bigger AI Models



**Scale of AI NN models is growing exponentially**

As measured by the number of parameters (roughly, the number of connections between their neurons)

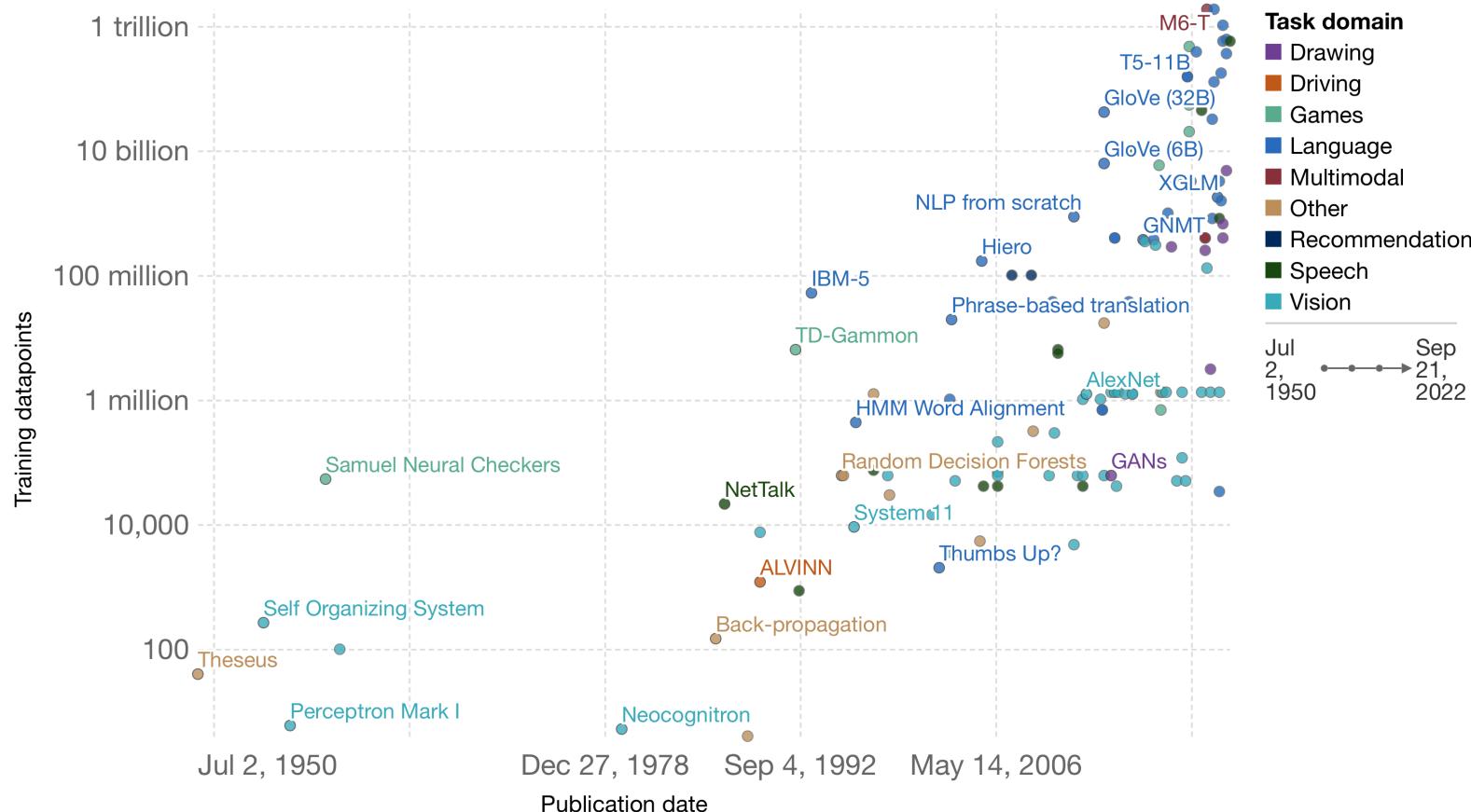
From <https://www.nature.com/articles/d41586-023-00777-9>  
Based on data from Sevilla et al. arxiv 2202.05924

# Drive to Bigger AI Models

Number of datapoints used to train notable artificial intelligence systems

Our World  
in Data

Each domain has a specific data point unit; for example, for vision it is images, for language it is words, and for games it is timesteps. This means systems can only be compared directly within the same domain.



**Large AI NN models  
Require massive amounts  
of data for training**

Source: Sevilla et al. (2022)

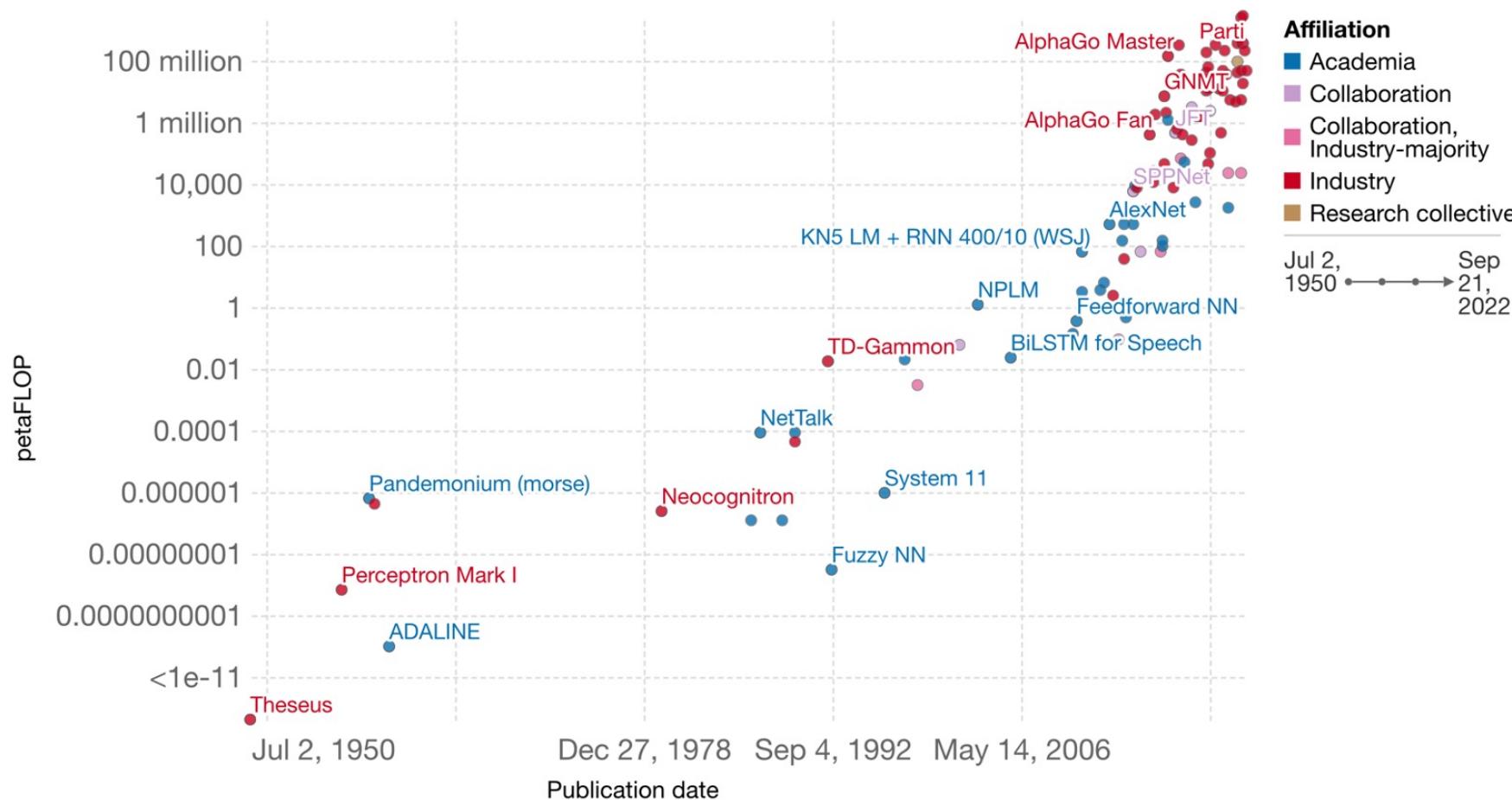
[OurWorldInData.org/artificial-intelligence](https://OurWorldInData.org/artificial-intelligence) • CC BY

# Drive to Bigger AI Models

Computation used to train notable AI systems, by affiliation of researchers

Computation is measured in total petaFLOP, which is  $10^{15}$  floating-point operations<sup>1</sup>.

Our World  
in Data



**Large AI NN models require massive amounts of compute power for training**

Until a few years ago, this was mostly an academic exercise.

The largest AI systems today are built by industry, not academia.

Source: Sevilla et al. (2022)

OurWorldInData.org/artificial-intelligence • CC BY

# Outline

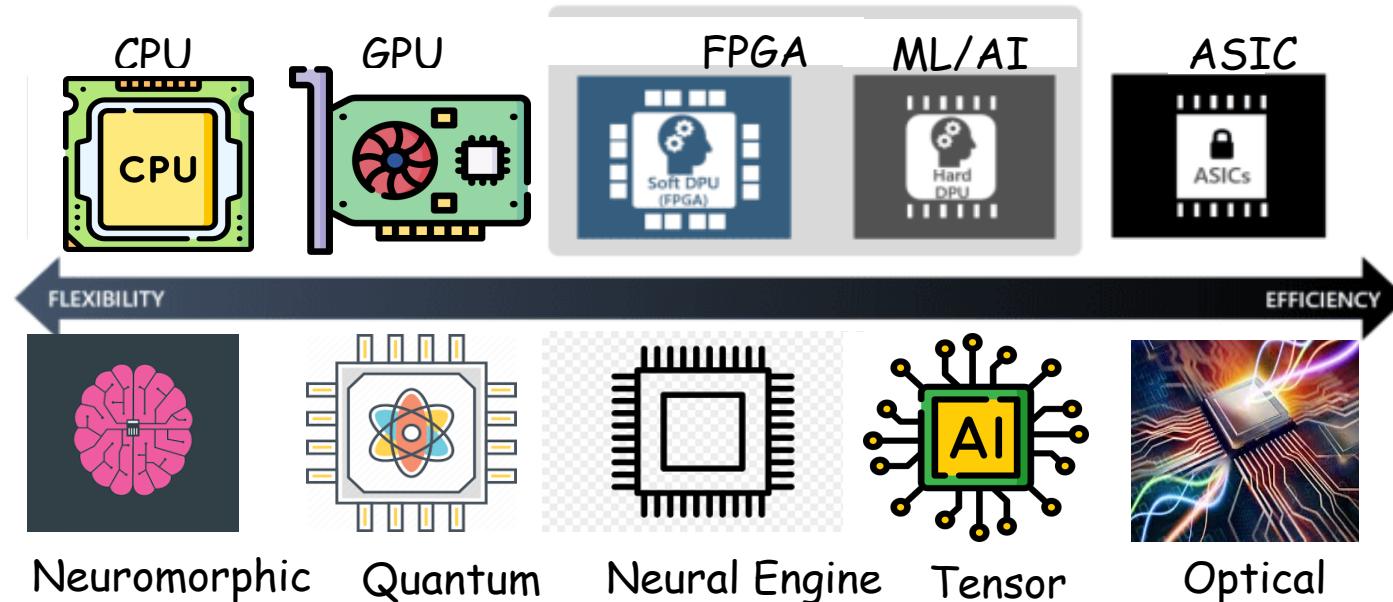
## We will cover the following topics

- Brief Intro on current ML trends
- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - CUDA intro
  - OpenACC intro
- SDSC Expanse GPU nodes
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

# What is a GPU?

## Accelerator

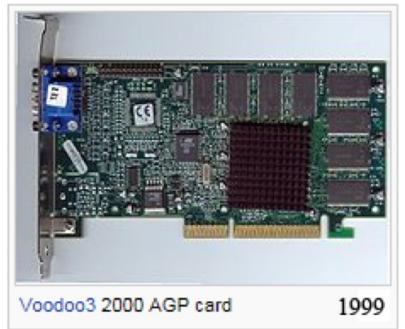
- Specialized hardware component to speed up some aspect of a computing workload.
- Examples include floating point co-processors in older PCs, specialized chips to perform floating point math in hardware rather than software.
- Many types of accelerators, including GPUs, Field Programmable Gate Arrays (FPGAs), ML/AI processors, Quantum processing units (QPUs)
- Different types of accelerators for different types of applications / compute workloads



# What is a GPU?

## Graphics processing unit

- “Specialist” processor to accelerate the rendering of computer graphics.
- Development driven by \$150 billion gaming industry.
- Originally fixed function pipelines.
- Modern GPUs are programmable for general purpose computations (GPGPU).
- Simplified core design compared to CPU
  - Limited architectural features, e.g. branch caches
  - Partially exposed memory hierarchy



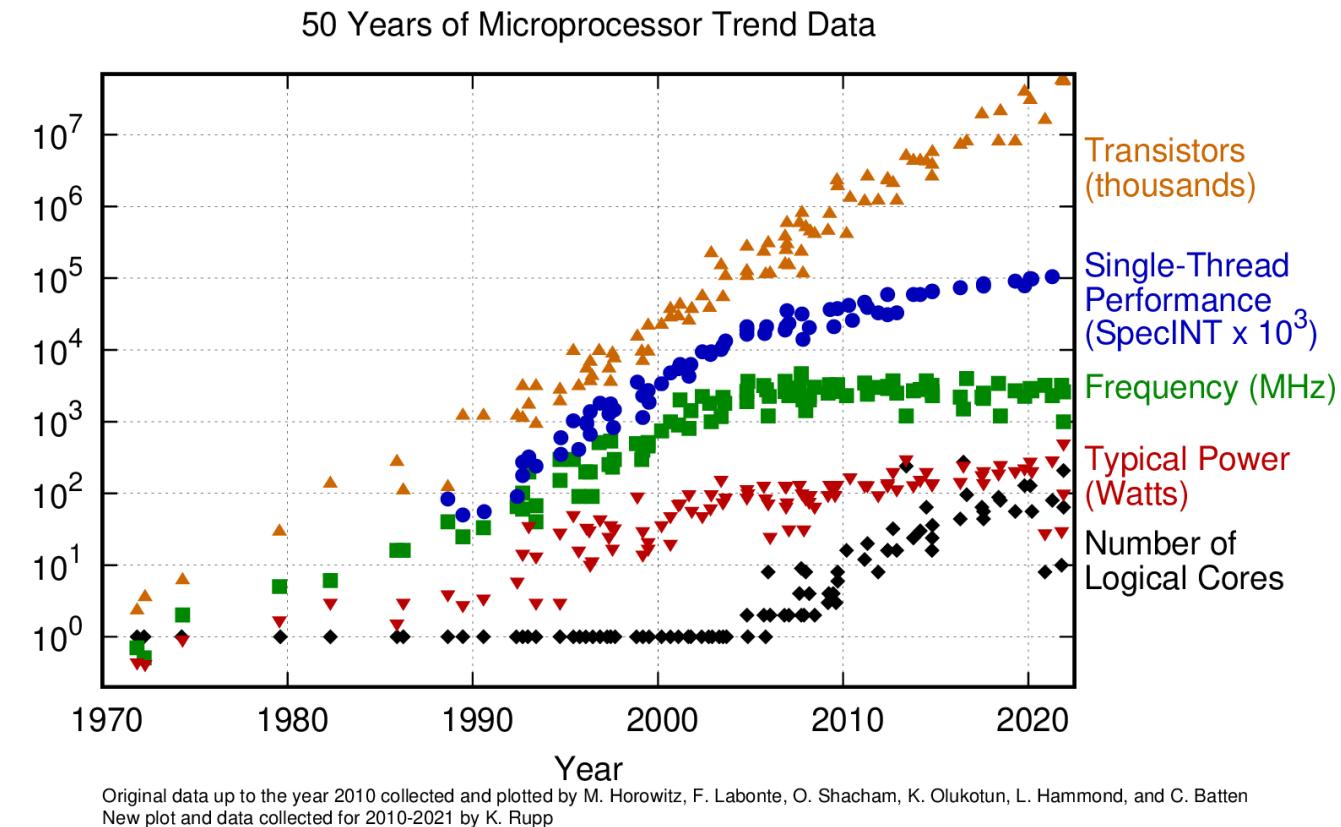
# Why is there such an interest in GPUs?

## Moore's law

- Transistor count in integrated circuits doubles about every two years.
- Exponential growth still holds (see figure).
- However...

## Trends since mid 2000s

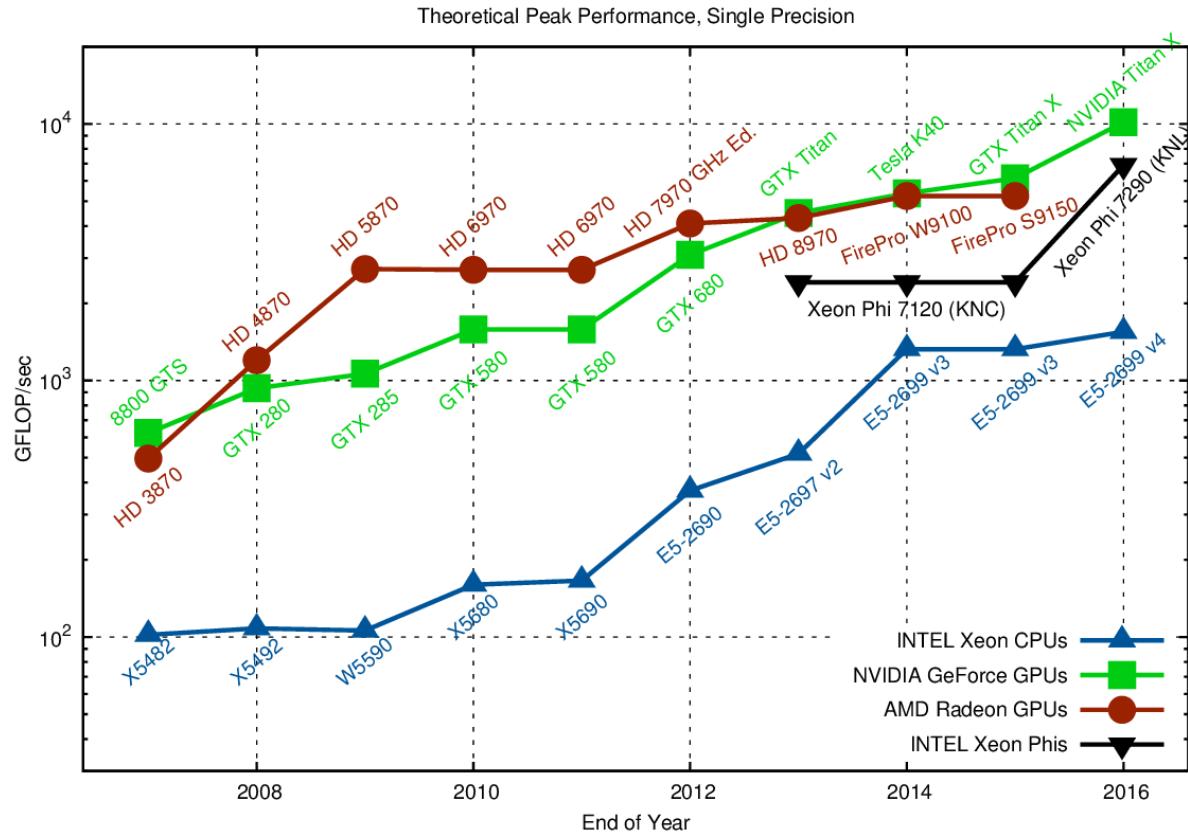
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.



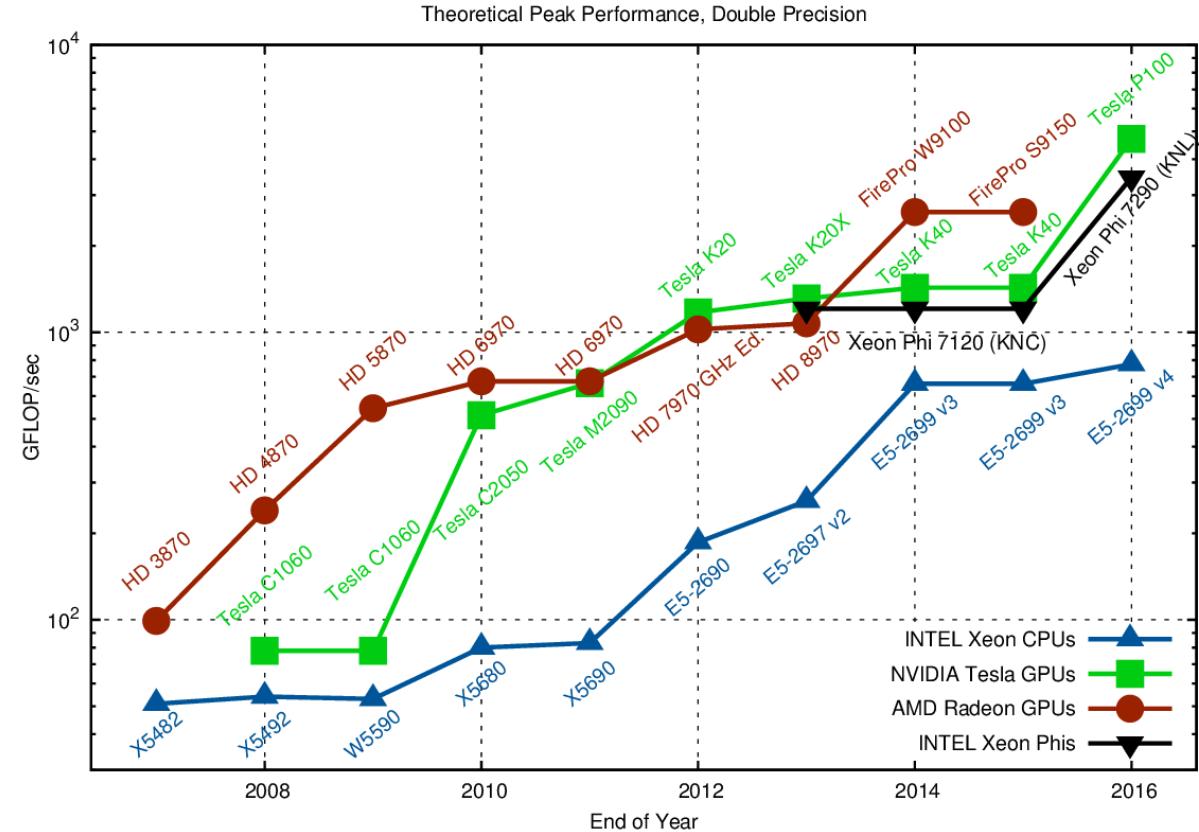
Source:

<https://github.com/karlrupp/microprocessor-trend-data/tree/master/50yrs>

# Why is there such an interest in GPUs?



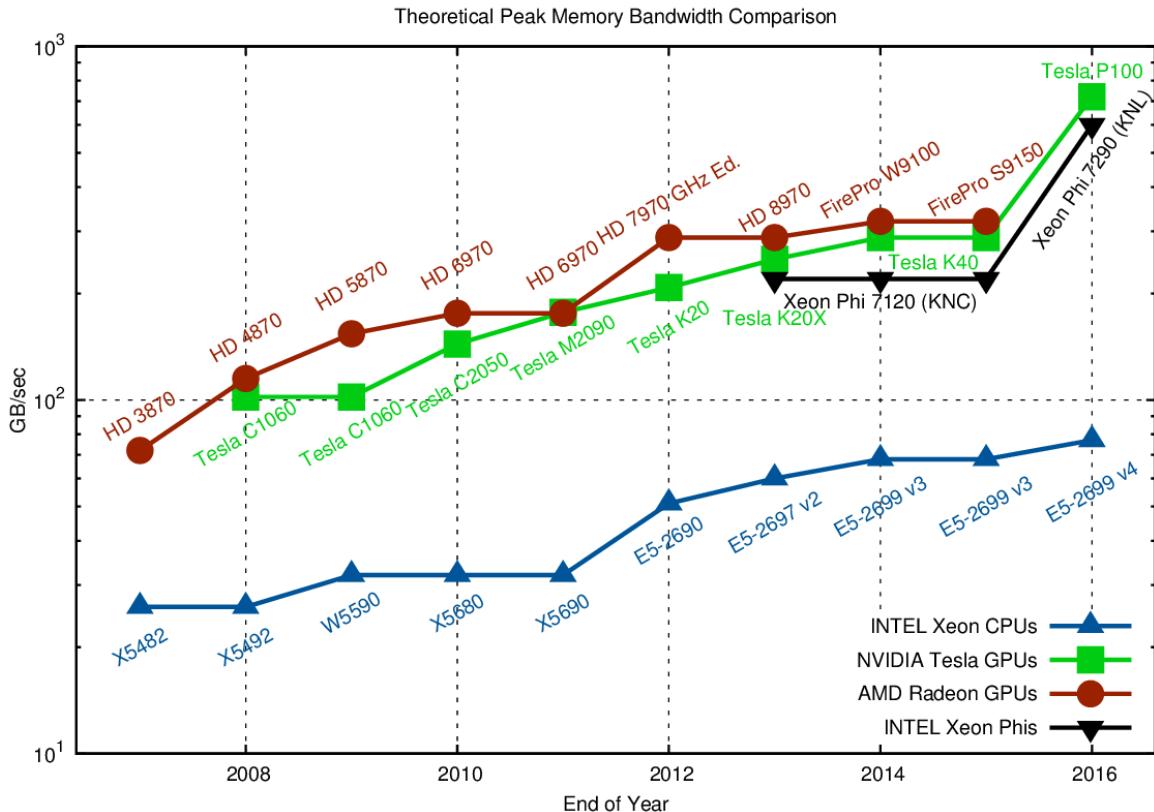
- GPUs offer significantly higher 32-bit floating point performance than CPUs.



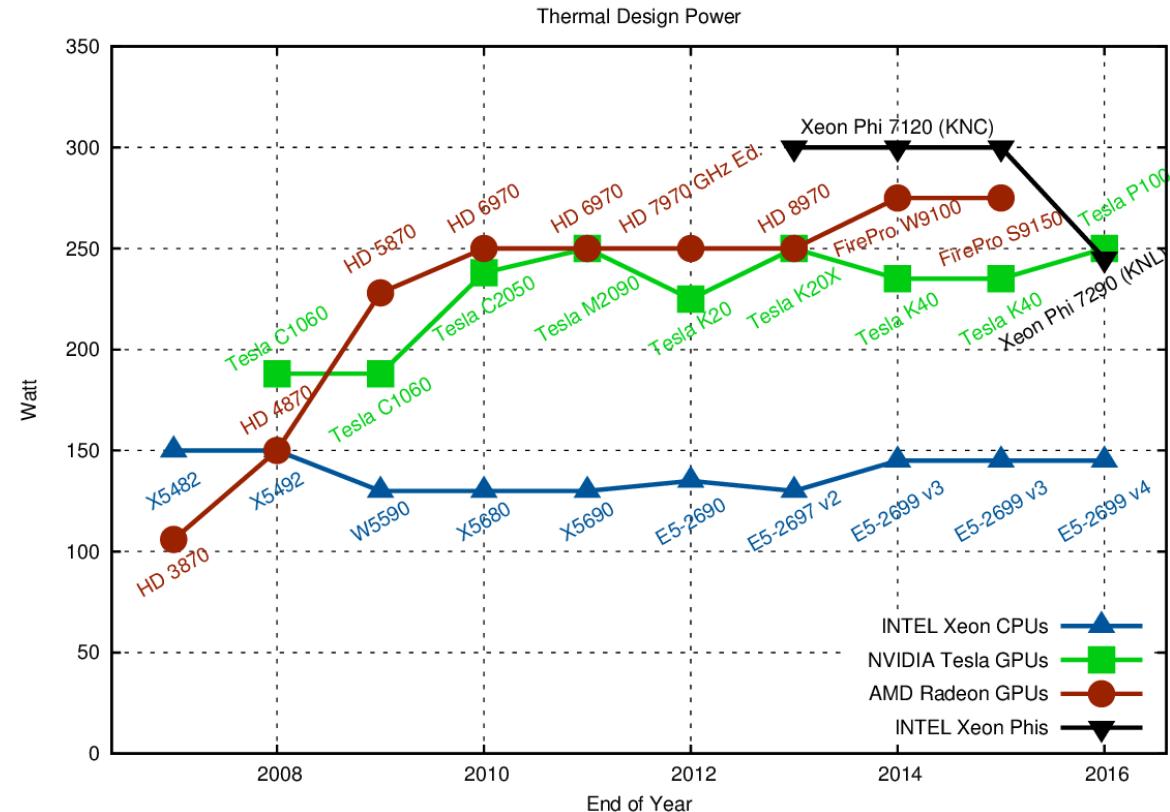
- Datacenter GPUs also offer significantly higher 64-bit floating point performance than CPUs.

Figures source: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

# Why is there such an interest in GPUs?



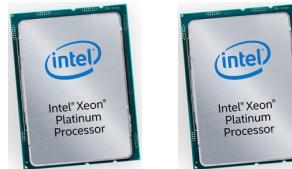
- GPUs have significantly higher memory bandwidth than CPUs.



- Given power consumption, a fair comparison would be a single GPU to 1- to 2-socket CPU server.

Figures source: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

# Comparison of top (in 2018) X86 CPU vs Nvidia V100 GPU



Aggregate performance numbers (FLOPs, BW)	<b>Dual socket Intel 8180 28-core (56 cores per node)</b>	<b>Nvidia Tesla V100, dual cards in an x86 server</b>
<b>Peak DP FLOPs</b>	4 TFLOPs	14 TFLOPs (3.5x)
<b>Peak SP FLOPs</b>	8 TFLOPs	28 TFLOPs (3.5x)
<b>Peak HP FLOPs</b>	N/A	224 TFLOPs
<b>Peak RAM BW</b>	~ 200 GB/sec	~ 1,800 GB/sec (9x)
<b>Peak PCIe BW</b>	N/A	32 GB/sec
<b>Power / Heat</b>	~ 400 W	2 x 250 W (+ ~ 400 W for server) (~ 2.25x)
<b>Purchase cost</b>	\$20,000 USD	\$20,000 USD
<b>Code portable?</b>	Yes	Yes (OpenACC, OpenMP, OpenCL, HIP, SYCL)

# A supercomputer in a desktop?



## ASCI White (LLNL)

- **12.3 TFLOP/sec** – #1 Top 500, November 2001.
- Cost – \$110 Million USD (in 2001!)

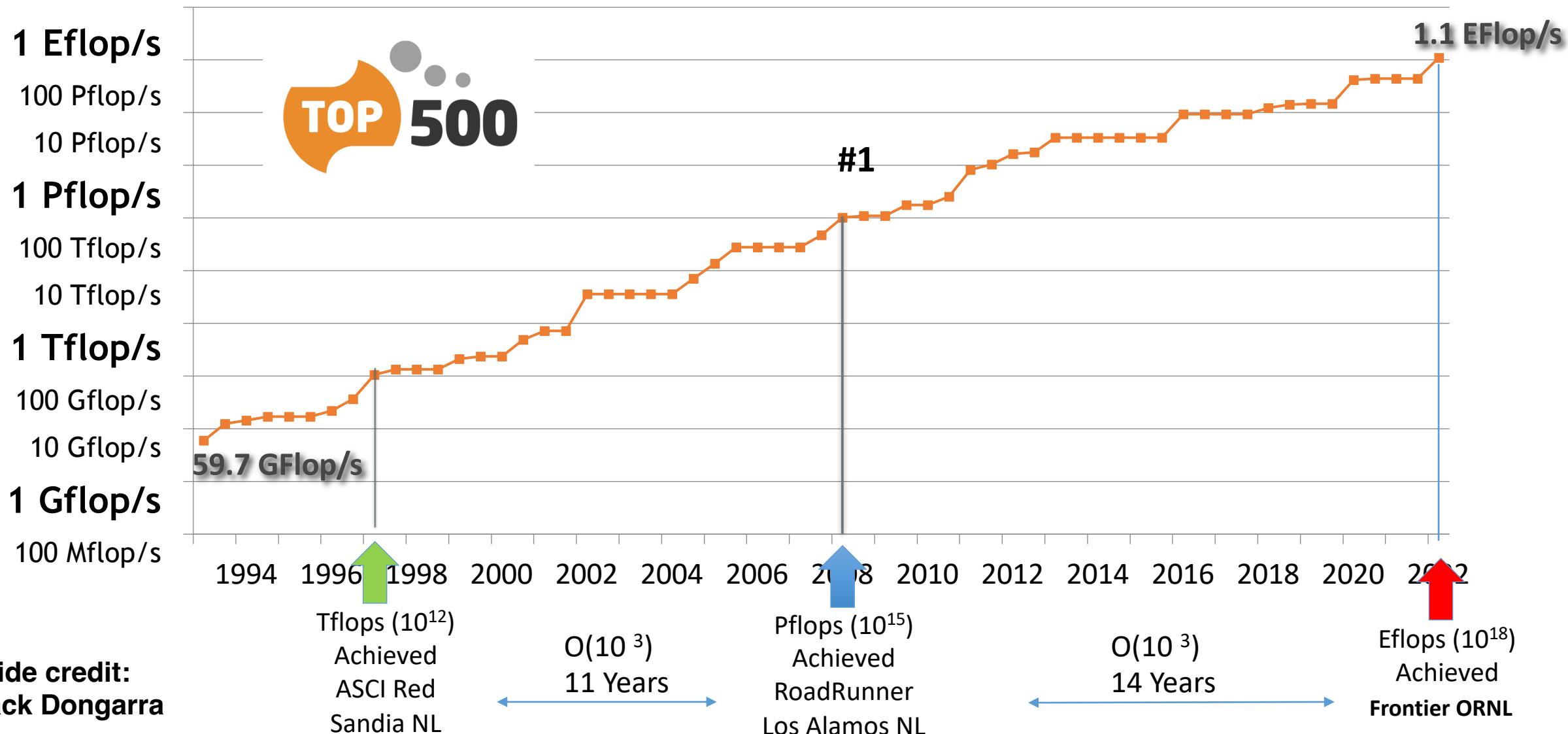
## SDSC Expanse

- 728 CPU nodes with 4.6 TFLOP/sec (each node)  
**3.4 PFLOP/sec (aggregate CPU)**
- 52 GPU nodes 4 x Nvidia V100 (Volta arch)  
31.3 TFLOP/sec DP, 62.7 TFLOP/sec SP (each node)  
**1.6 PFLOP/sec DP, 3.3 PFLOP/sec SP (aggregate GPU)**
- Hardware Cost – \$10 Million USD

## DIY 4 x Nvidia RTX 3080 box (2020) (Ampere arch)

- 1.9 TFLOP/sec DP
- **119.0 TFLOP/sec SP**
- Cost – ~ \$4 Thousand USD

# Performance Development



Slide credit:  
Jack Dongarra

# Current HPC Environment for Scientific Computing

## DOE Exascale Supercomputers



AMD APU Based  
(Planned)



Intel CPU + GPU Based  
(Recently installed)



AMD CPU + GPU Based  
(Up & running)



AMD + Intel CPUs  
Nvidia GPUs

### Highly parallel

- Distributed memory
- MPI + Open-MP programming model

### Heterogeneous

- Commodity processors + GPU accelerators

### Communication

- Special interconnects
- Moving data between parts very expensive compared to floating point operations

### Floating point hardware

- 64, 32, 16, & 8 bit precision

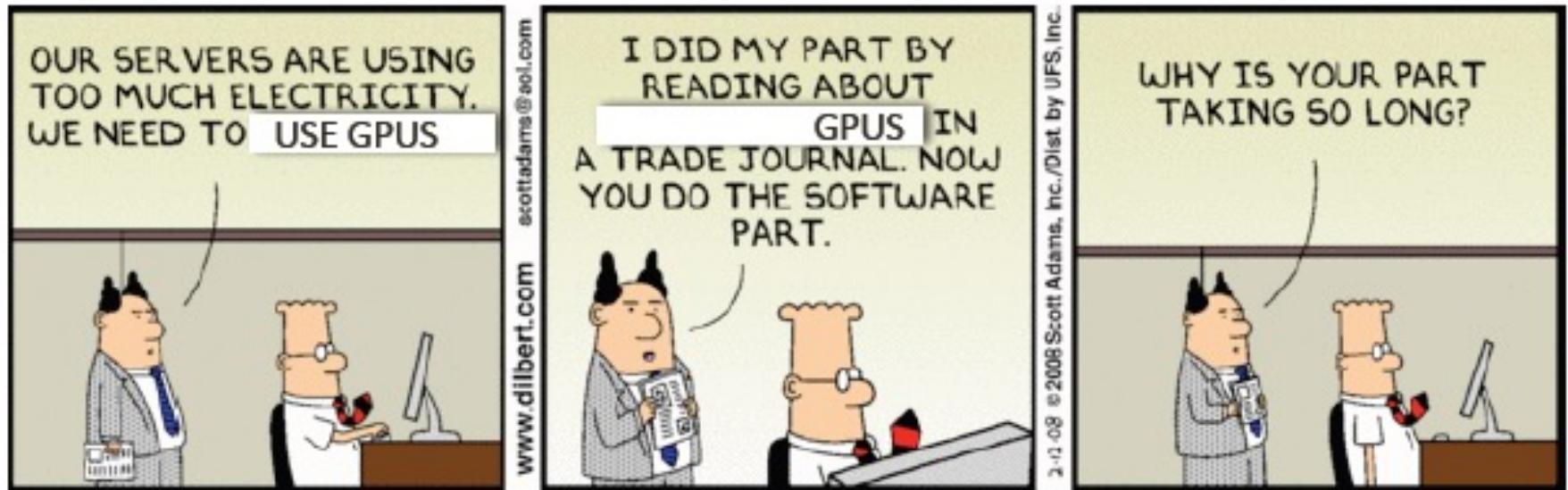
**ORNL Frontier: 2 Eflop/s**  
 $8.8 \times 10^6$  Cores, 9408 nodes, 30 MW  
(node = 1 AMD CPU + 4 AMD GPUs)  
**> 98% of performance from GPUs**



# HPCG Benchmark Top 10, November 2022

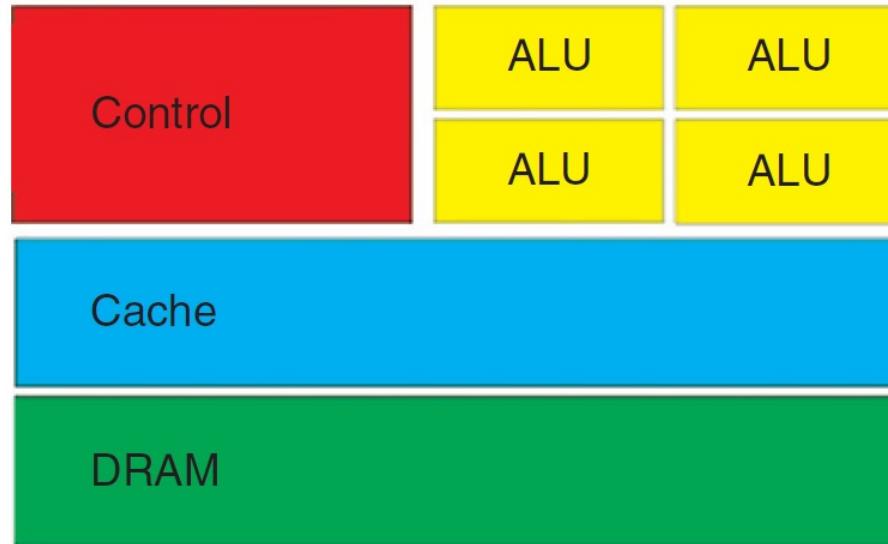
Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science Japan	Fugaku, Fujitsu A64FX 48C 2.2GHz, Tofu D, Fujitsu	7,630,848	442	2	16.0	3.0%
2	DOE/SC/ORNL USA	Frontier, HPE Cray EX235a, AMD 3rd EPYC 64C, 2 GHz, AMD Instinct MI250X, Slingshot 10	8,730,112	1,102	1	14.1	0.8%
3	EuroHPC/CSC Finland	LUMI, HPE Cray EX235a, AMD Zen-3 (Milan) 64C 2GHz, AMD MI250X, Slingshot-11	2,174,976	304	3	3.41	0.8%
4	DOE/SC/ORNL USA	Summit, AC922, IBM POWER9 22C 3.7GHz, Dual-rail Mellanox FDR, NVIDIA Volta V100, IBM	2,414,592	149	5	2.93	1.5%
5	EuroHPC/CINECA Italy	Leonardo, BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 Infiniband	1,463,616	175	4	2.57	1.0%
6	DOE/SC/LBNL USA	Perlmutter, HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	761,856	70.9	8	1.91	2.0%
7	DOE/NNSA/LLNL USA	Sierra, S922LC, IBM POWER9 20C 3.1 GHz, Mellanox EDR, NVIDIA Volta V100, IBM	1,572,480	94.6	6	1.80	1.4%
8	NVIDIA USA	Selene, DGX SuperPOD, AMD EPYC 7742 64C 2.25 GHz, Mellanox HDR, NVIDIA Ampere A100	555,520	63.5	9	1.62	2.0%
9	Forschungszentrum Juelich (FZJ) Germany	JUWELS Booster Module, Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, Mellanox HDR InfiniBand, NVIDIA Ampere A100, Atos	449,280	44.1	12	1.28	1.8%
10	Saudi Aramco Saudi Arabia	Dammam-7, Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, InfiniBand HDR 100, NVIDIA Volta V100, HPE	672,520	22.4	20	0.88	1.6%

# What's the catch?

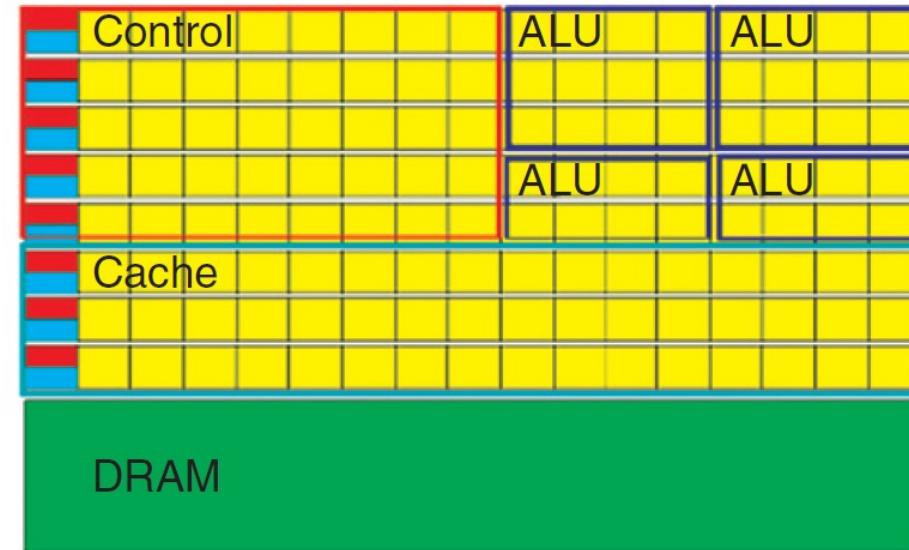


# GPU vs CPU architecture

(a) CPU



(b) GPU



## CPU

- Few processing cores with sophisticated hardware
- Multi-level caching
- Prefetching
- Branch prediction

## GPU

- Thousands of simplistic compute cores (packaged into a few multiprocessors)
- Operate in lock-step
- Vectorized loads/stores to memory
- Need to manage memory hierarchy

# GPU architecture

## CUDA Computing with Tesla T10

- 240 SP processors at 1.45 GHz: 1 TFLOPS peak
- 30 DP processors at 1.44Ghz: 86 GFLOPS peak
- 128 threads per processor: 30,720 threads total

The diagram illustrates the Tesla T10 architecture. It shows the physical components: a silicon die, a graphics card, and a server chassis. Below these is a detailed block diagram of the Tesla T10. The diagram is labeled 'Tesla T10' at the top. It features a 'Host CPU' connected via a 'Bridge' to 'System Memory'. A 'Host Distribution' block feeds into ten 'Multiprocessor' units. Each multiprocessor contains two 'SP' (Single Precision) cores, two 'SFU' (Special Function Unit) cores, one 'DP' (Double Precision) core, and shared memory. Below the multiprocessors is the 'Interconnection Network'. Each multiprocessor is connected to a 'DRAM' (Dynamic Random Access Memory) module. The diagram also shows 'ROP' (Register-to-Register Operations) and 'L2' cache units.

© NVIDIA Corporation 2008



## Nvidia GPU architecture in 2009

- Tesla T10, a server with early C1060 datacenter GPU
- Basic architecture is still the same

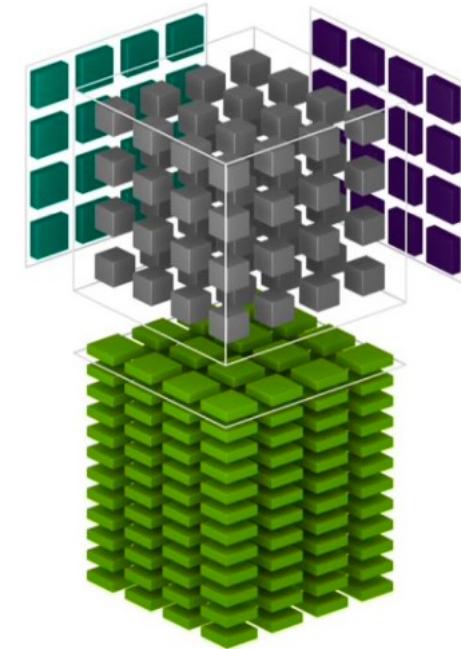
### Multiprocessor

- FP32 compute cores
- FP64 compute core(s)
- Special function units
- Instruction cache
- Shared memory / data cache
- Handles many more threads than processing cores

# Tensor cores

## Accelerating MMA for FP64, TF32, Bfloat16, and FP16

- Tensor Cores are specialized hardware for deep learning that help accelerate matrix multiply and accumulate (MMA) operations
- Nvidia Volta architecture introduced Tensor Cores with FP16 data types
- Nvidia Ampere GPUs introduce Tensor Core support for FP64, TF32, and Bfloat16 data types
- Deep learning operations that benefit from tensor cores are
  - Fully connected / linear / dense layers
  - Convolutional layers
  - Recurrent layers
- Tensor Cores are also used for mixed precision matrix operations (CUBLAS library)



Tensor core 4x4 matrix multiply and accumulate, Volta architecture

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

# GPU architecture

**Hardware characteristics change across GPU models and generations**

- FP32 / FP64 floating point performance
- Memory bandwidth
- Number of compute cores and multiprocessors
- Number of threads that the hardware can execute
- Number of registers and cache size
- Available GPU memory, device / shared

Data Center GPUs	P100	V100	A100
#Multi Proc	56	80	108
SP Cores per MP	64	64	64
#Cores	3,584	5,120	6,912
Warp Size	32	32	32
FP64 Gflop/s	4,763	7,066	9,746

**Memory hierarchy needs to be explicitly managed**

- CPU memory, GPU global / shared / texture / constant memory
- Unified memory helps, but the memory hierarchy still exists

**Different hardware vendors work in different ways**

- Nvidia vs AMD vs Intel GPUs

**You don't have to worry about any of this if you use an ML framework such as PyTorch or Tensorflow, which will automatically compile your ML model for the targeted hardware**



# Outline

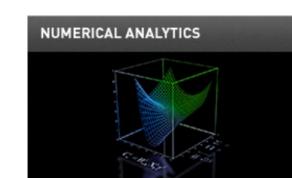
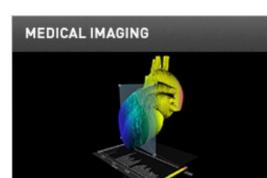
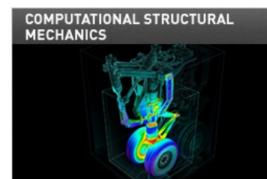
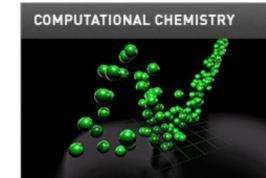
## We will cover the following topics

- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - CUDA intro
  - OpenACC intro
- SDSC Expanse GPU nodes
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

# GPU accelerated software

Examples from virtually any field

- Exhautive list on <https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/>
- Chemistry
- Life sciences
- Bioinformatics
- Astrophysics
- Finance
- Medical imaging
- Natural language processing
- Social sciences
- Weather and climate
- Computational fluid dynamics
- **Machine learning**, of course
- etc...

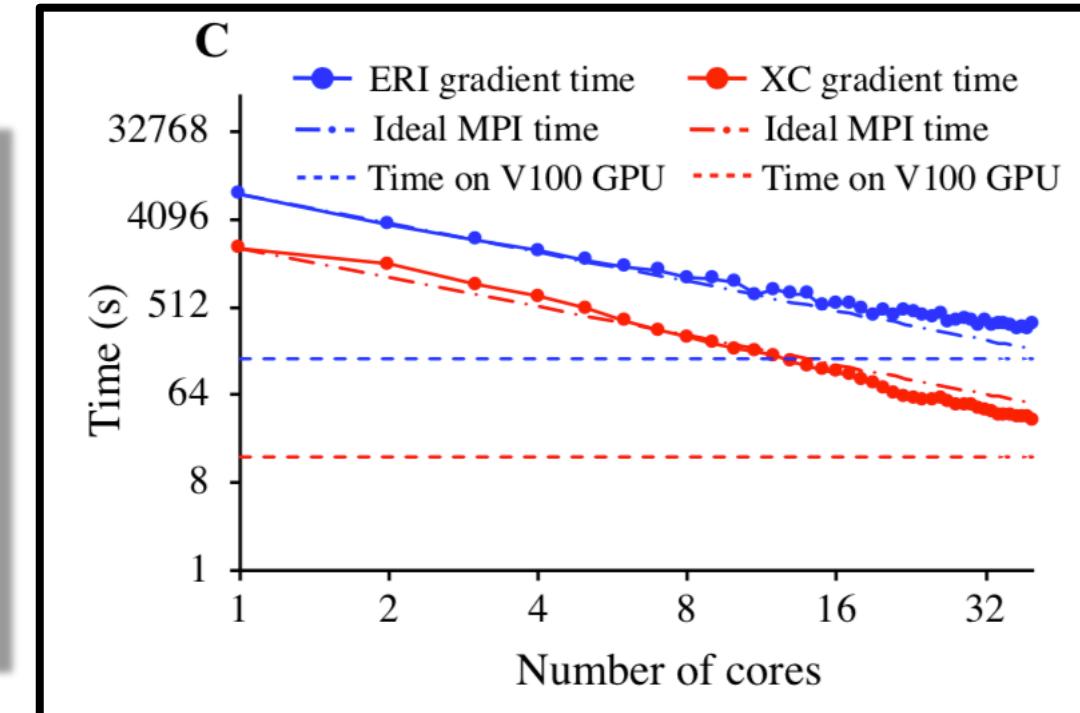
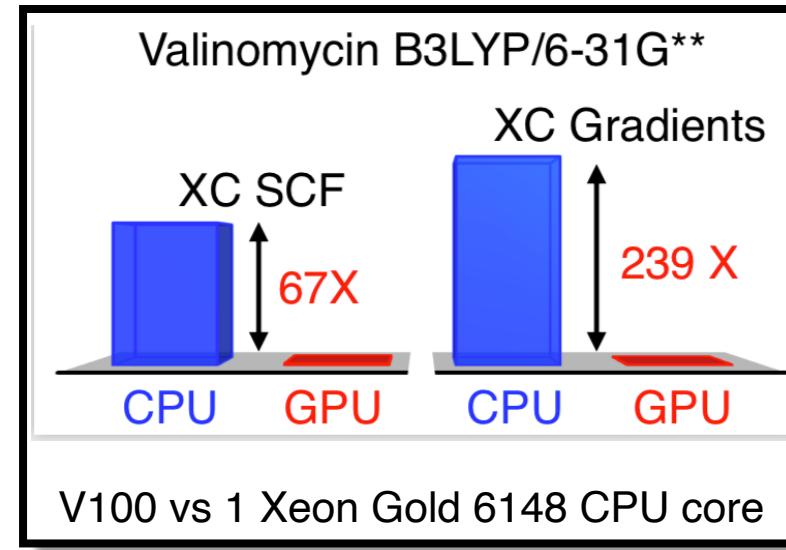
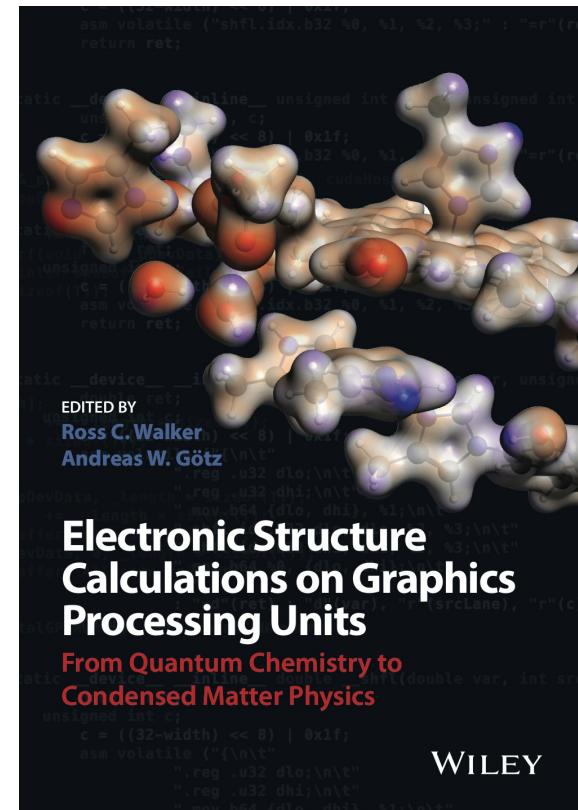
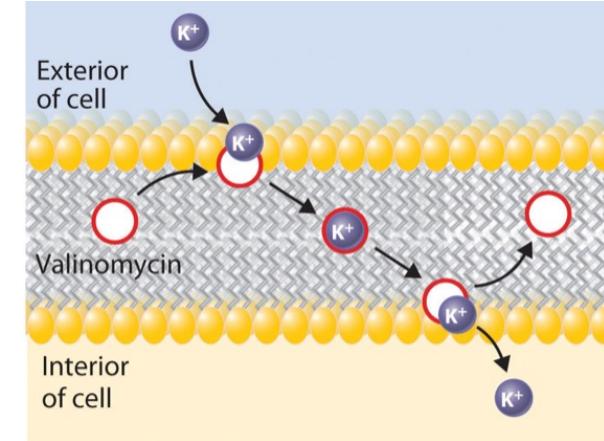
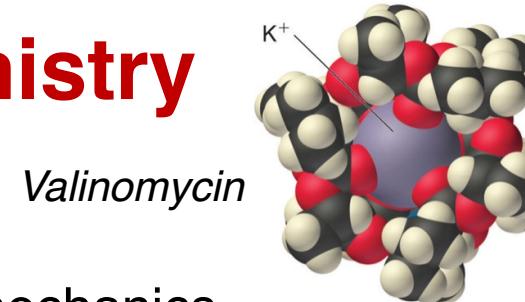


# Applications: Quantum Chemistry

# Quantum Chemistry

# Example: Open source QUICK code

- Compute molecular properties from quantum mechanics
  - <https://github.com/merzlab/QUICK> (developed by Merz and Goetz labs)



See *J. Chem. Theory Comput.* **16**, 4315-4326 (2020) <https://dx.doi.org/10.1021/acs.jctc.0c00290>

# Quantum Chemistry

$$E[\rho] = T_s[\rho] + \int d\mathbf{r} \rho(\mathbf{r}) v_{\text{ext}}(\mathbf{r}) + \frac{1}{2} \int d\mathbf{r} d\mathbf{r}' \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} + E_{\text{xc}}[\rho]$$

## Analytical electron repulsion integrals (ERIs)

- Coulomb interaction between electrons (3<sup>rd</sup> term)
- Exact exchange (not shown in equation above)

## Numerical quadrature of XC potential and energy

- XC energy has complicated form
- Cannot be computed analytically
- Thousands of grid points per atom

$$E^{xc} = \int f(\rho_\alpha, \rho_\beta, \gamma_{\alpha\alpha}, \gamma_{\alpha\beta}, \gamma_{\beta\beta}) dr,$$

$$\int d\mathbf{r} f(\mathbf{r}) \approx \sum_i \omega_i f(\mathbf{r}_i)$$

Get QUICK for free at  
<https://github.com/merzlab/quick>

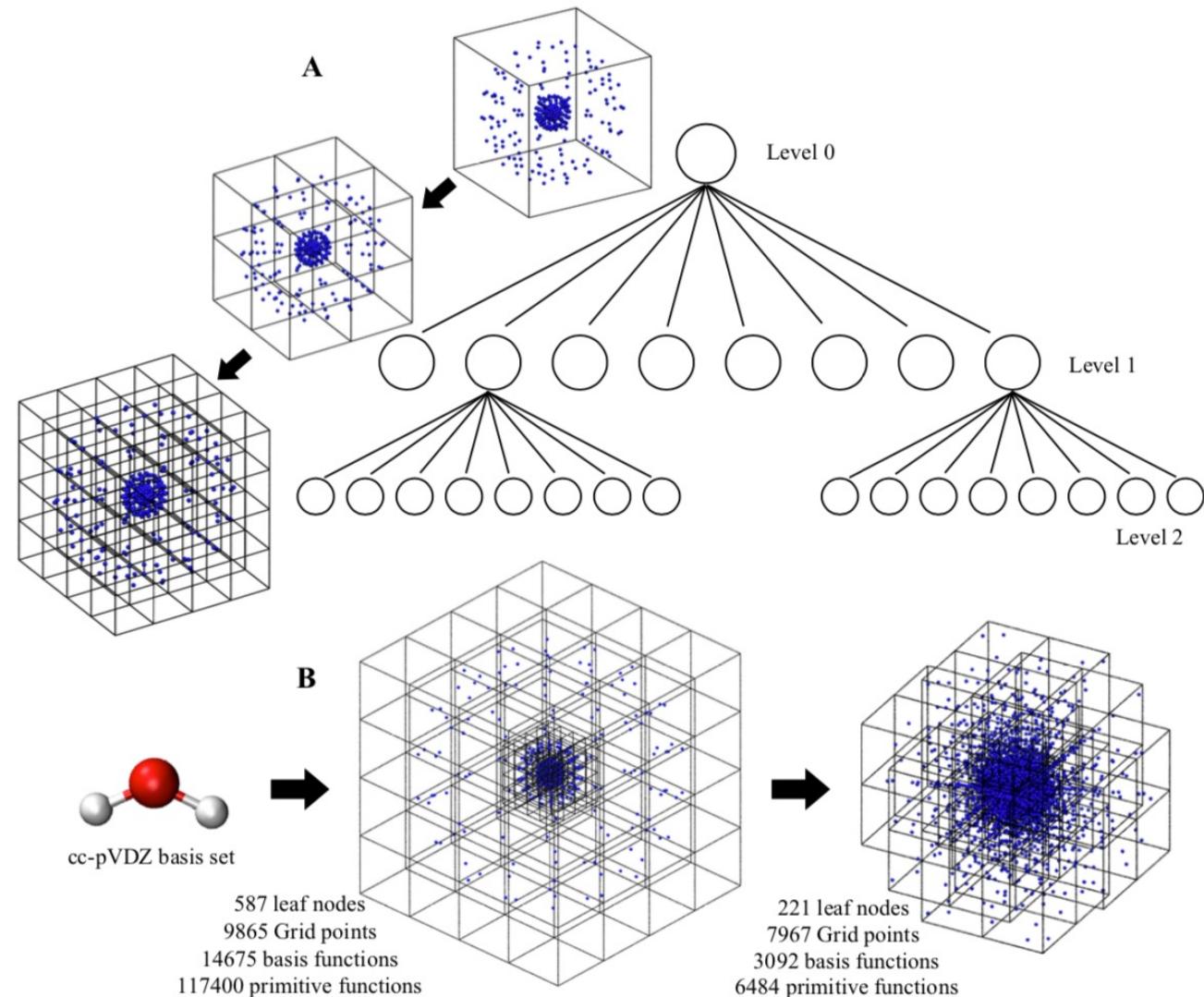
### Relevant publications

- **DFT octree-based quadrature on GPUs**  
Manathunga, Miao, Mu, Götz, Merz, *JCTC* **16** (2020) 4315.
- **Multi-GPU parallelization**  
Manathunga, Jin, Cruzeiro, Miao, Mu, Arumugam, Keipert, Aktulga, Merz, Götz, *JCTC* **17** (2021) 3955.
- **QM/MM implementation**  
Cruzeiro, Manathunga, Merz, Götz, *JCIM* **61** (2021) 2109.
- **Nvidia and AMD GPU QM/MM + optimization**  
Manathunga, Aktulga, Merz, Götz, *JCIM* **63** (2023) 711.

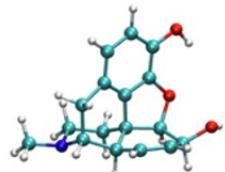
# Quantum Chemistry

## Parallel numerical quadrature in QUICK

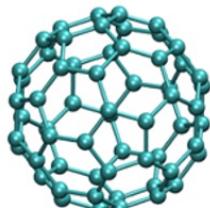
- Octree based partitioning of 3D grid points
- Prescreening of function values on grid point batches leads to linear scaling for large molecules
- Grid point batches are processed in parallel on CPU cores via MPI or GPUs via CUDA.
- See *J. Chem. Theory Comput.* **16**, 4315-4326 (2020)  
<https://dx.doi.org/10.1021/acs.jctc.0c00290>



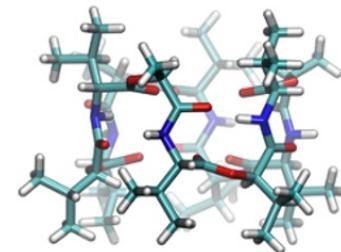
# Quantum Chemistry



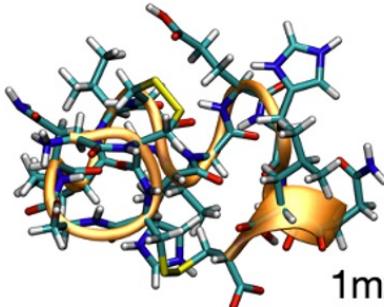
Morphine  
(40, 410)<sup>b</sup>



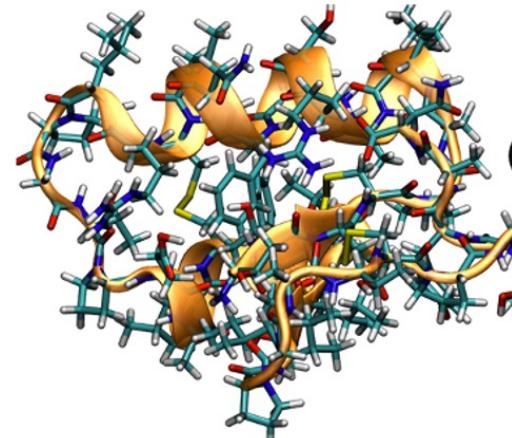
Buckyball  
(60, 900)<sup>b</sup>



Valinomycin  
(168, 1620)<sup>b</sup>



1m2c  
(220, 2276)<sup>b</sup>



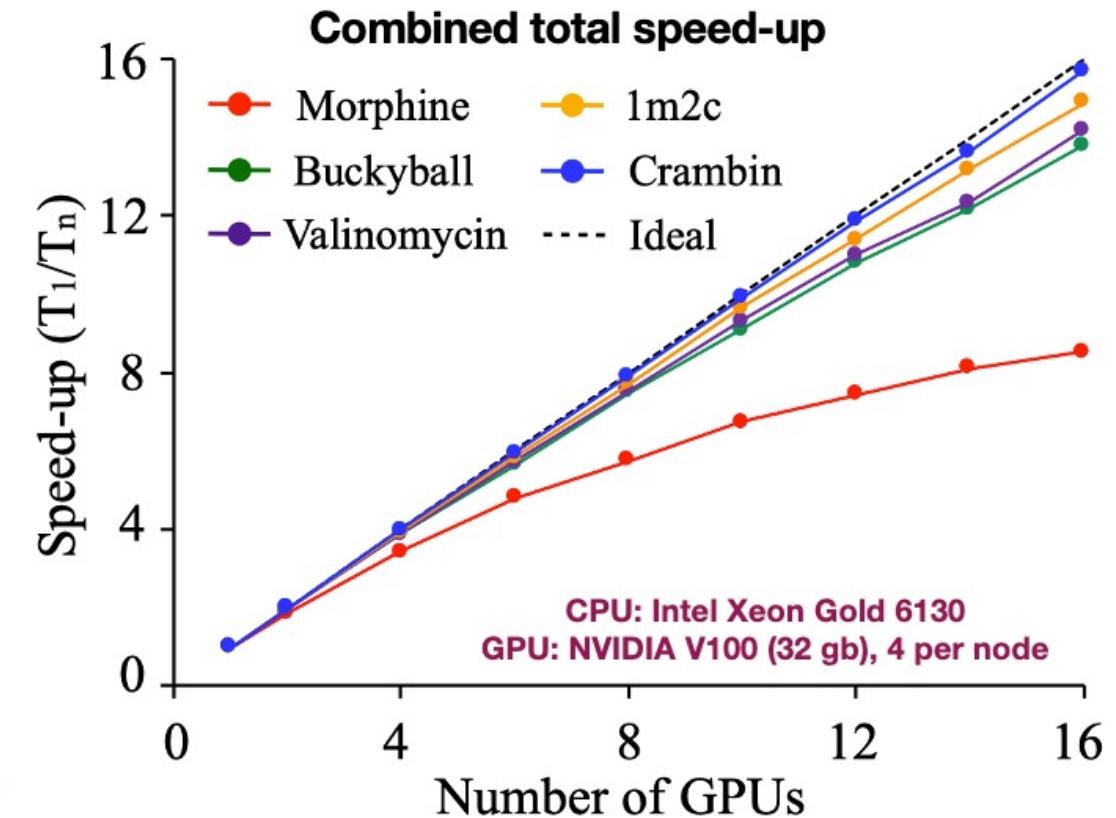
Crambin  
(642, 6504)<sup>b</sup>

*b.* The number of atoms and number of basis functions are given in brackets.

Benchmarks: QUICK 20.06

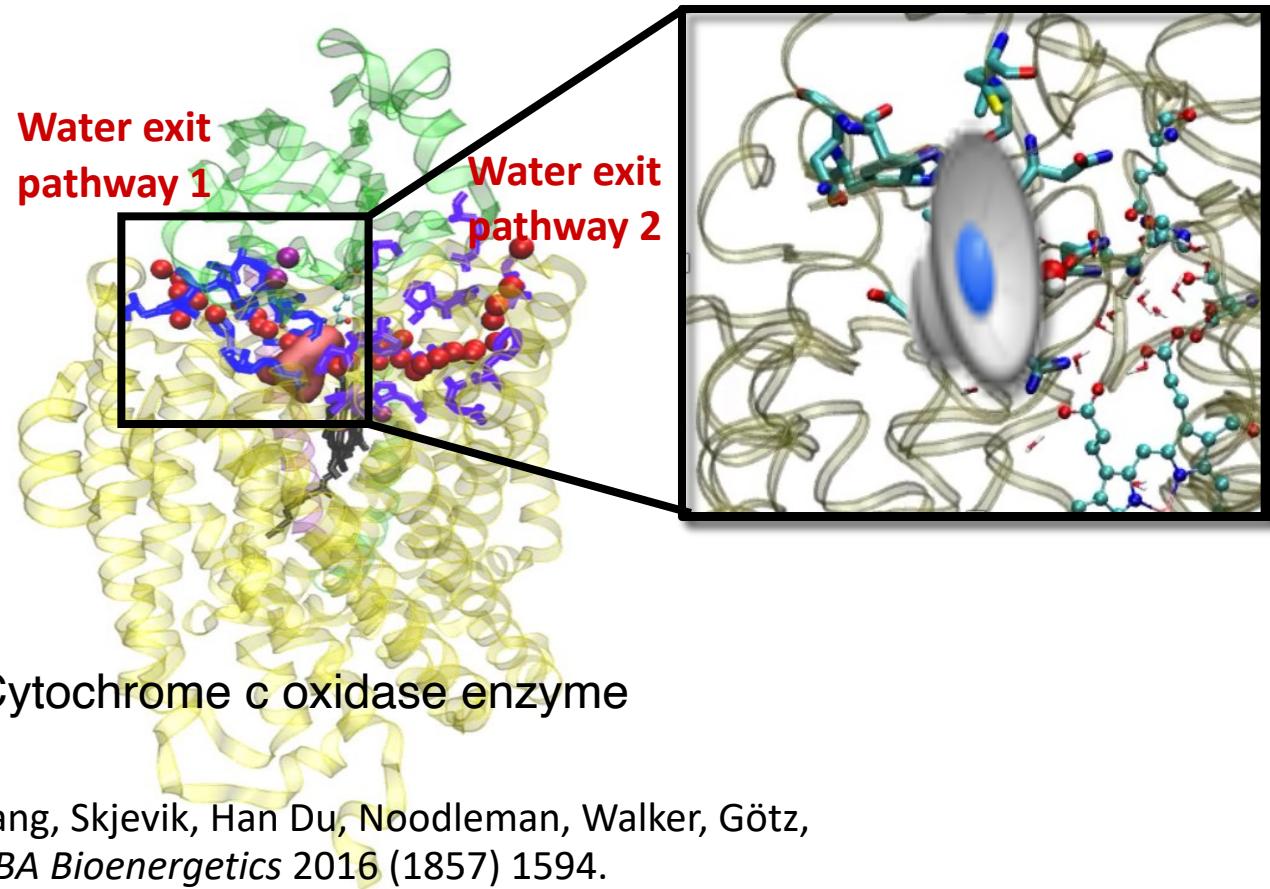
Manathunga *et al.*, JCTC 17 (2021) 3955.

B3LYP/6-31G\*\* gradient calculations



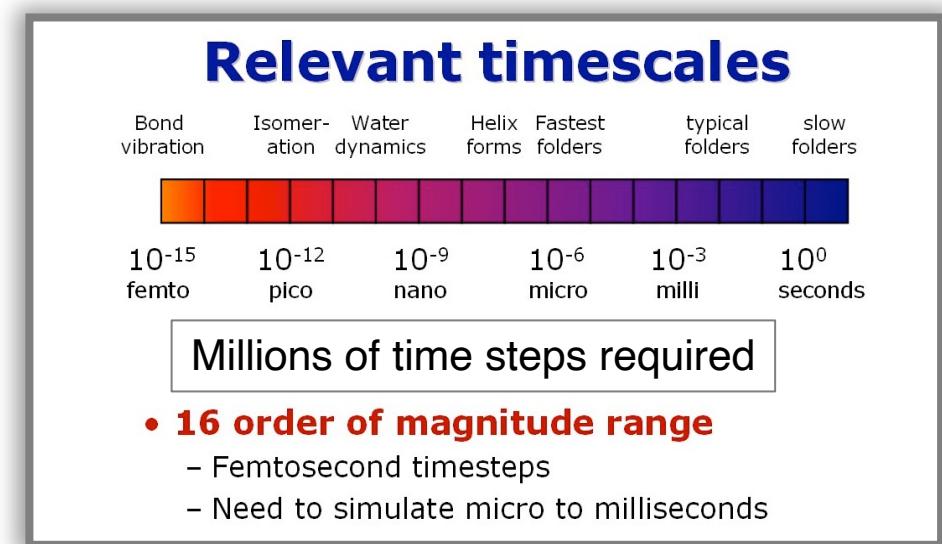
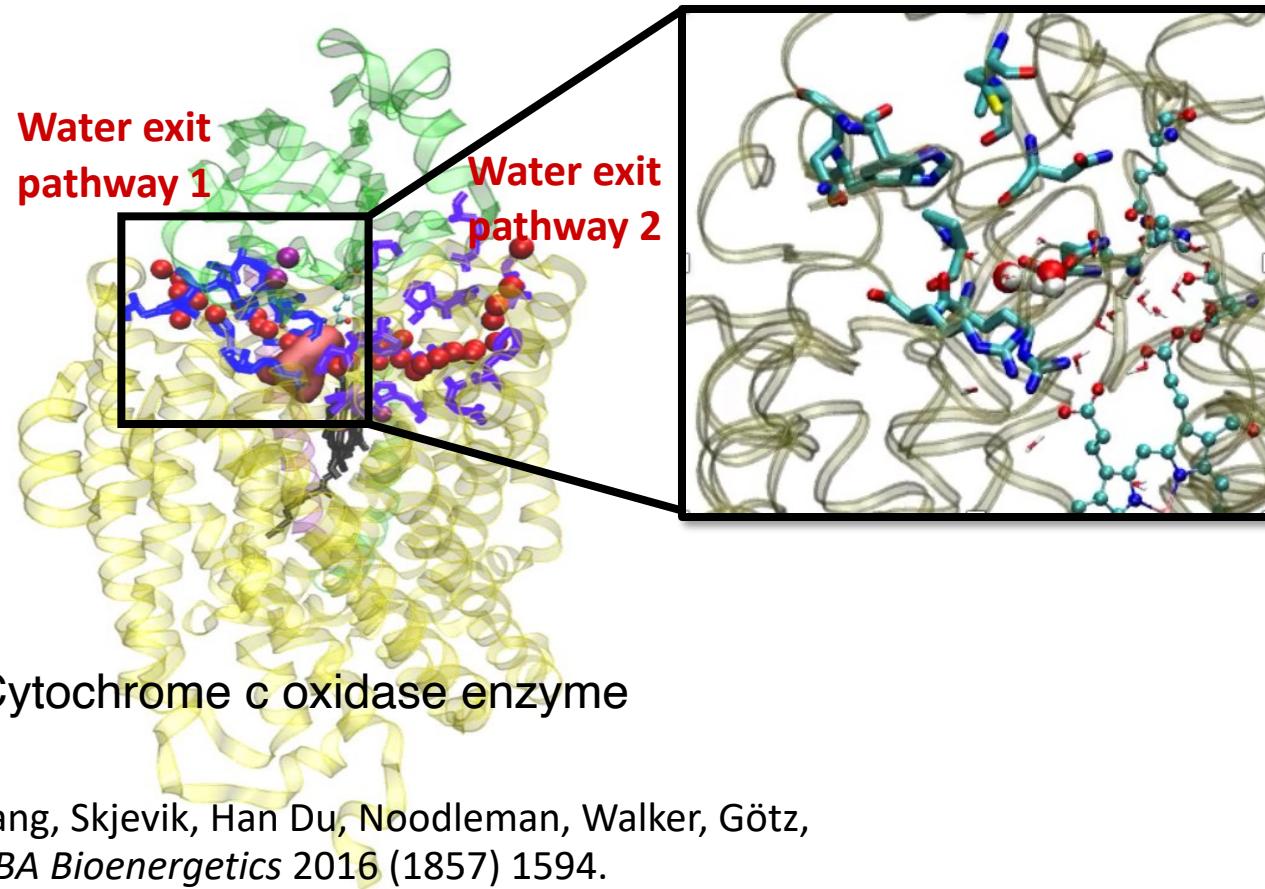
# Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems



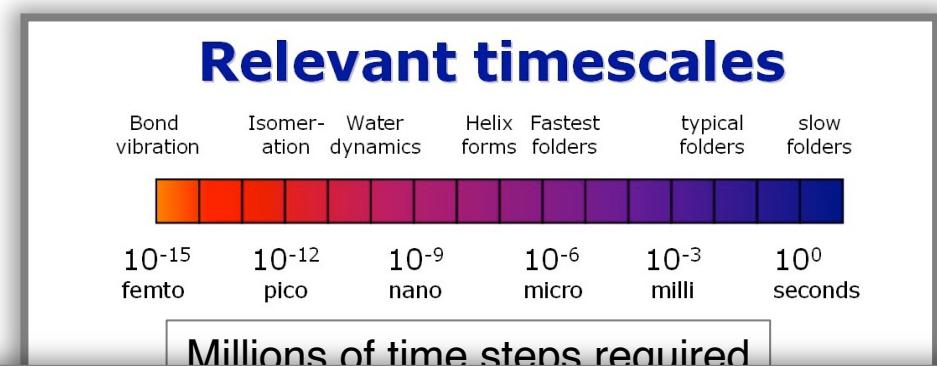
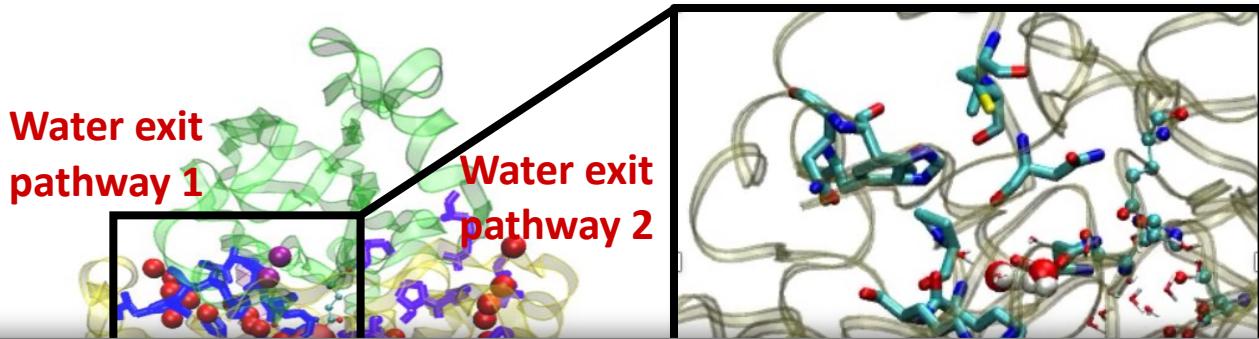
# Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems



# Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems

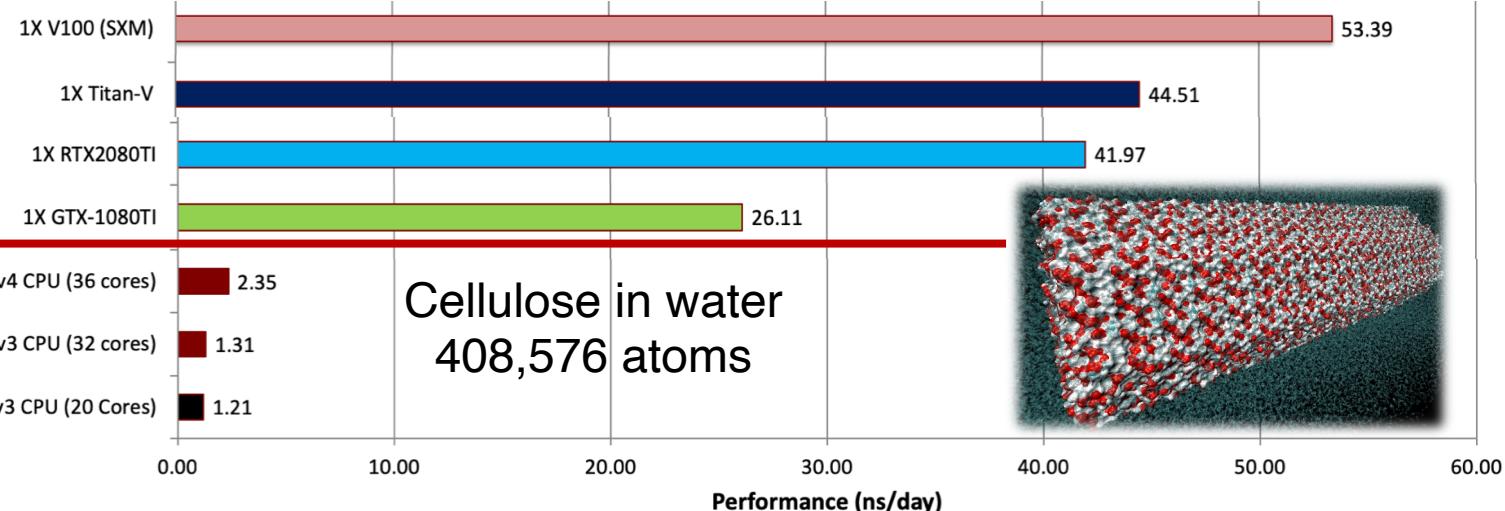


## Amber 18 molecular dynamics software

Götz, Williamson, Xu, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

Le Grand, Götz, Walker, *Comput Phys Comm* 2013 (184) 374.

Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.



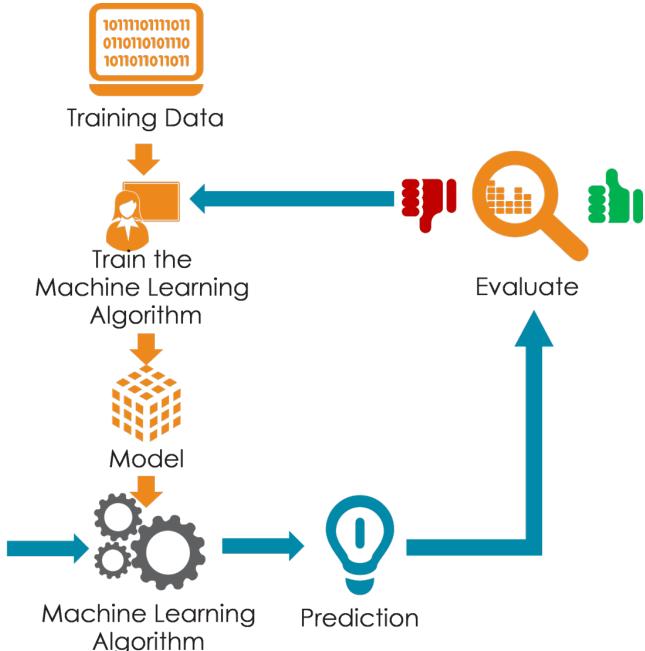
# Machine learning and GPUs

## Machine learning

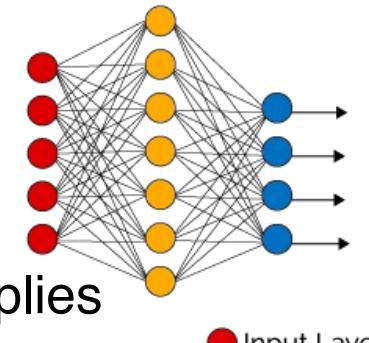
- Estimate / predictive model based on reference data.
- Many different methods and algorithms.
- GPUs are particularly well suited for deep learning workloads

## Deep learning

- Neural networks with many hidden layers.
- Tensor operations (matrix multiplications).
- GPUs are very efficient at these (4x4 matrix algebra is used in 3D graphics)
- Half-precision arithmetic can be used for many ML applications, at least for inference.
- Nvidia Volta architecture introduced tensor cores, dedicated hardware for mixed-precision matrix multiplies
- ML frameworks provide GPU support (E.g. PyTorch, TensorFlow)

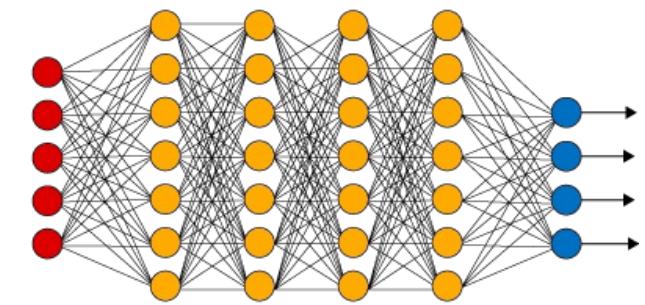


Simple Neural Network



Input Layer

Deep Learning Neural Network



Hidden Layer

Output Layer

# Outline

## We will cover the following topics

- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - CUDA intro
  - OpenACC intro
- SDSC Expanse GPU nodes
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

# GPU programming languages

## CUDA

- Proprietary, works only for Nvidia GPUs
- De-facto standard for high-performance code

## HIP

- Open-source C++ runtime API and kernel language developed by AMD
- Similar to CUDA, works with Nvidia (via CUDA) and AMD (via ROCm)

## OpenCL

- Industry standard, works for Nvidia, AMD and Intel GPUs (and other devices)

## SYCL

- Single-source, high-level, standard C++ programming model
- Can target a range of heterogeneous platforms (CPUs, GPUs, FPGAs)

## OpenACC

- Accelerator directives for Nvidia and AMD
- Works with C/C++ and Fortran

## OpenMP

- Version 4.x includes accelerator and vectorization directives
- Not mature for GPUs

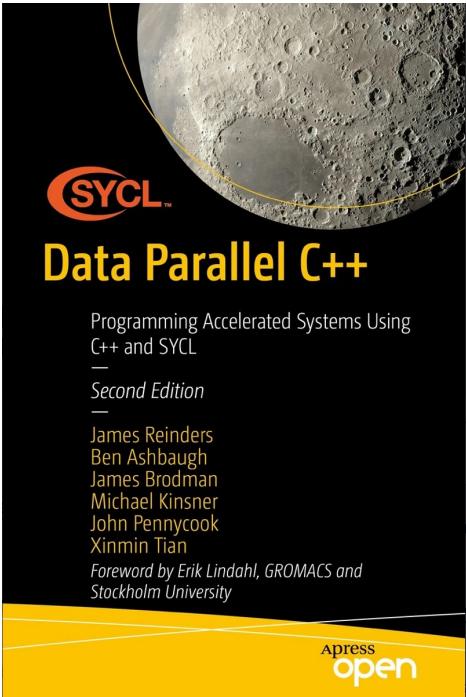
# Recommended Reading

SYCL reference: [https://github.khronos.org/SYCL\\_Reference/](https://github.khronos.org/SYCL_Reference/)

SYCL training material:

<https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/overview.html>

SYCL free ebook: <https://link.springer.com/book/10.1007/978-1-4842-9691-2>



# Recommended Reading

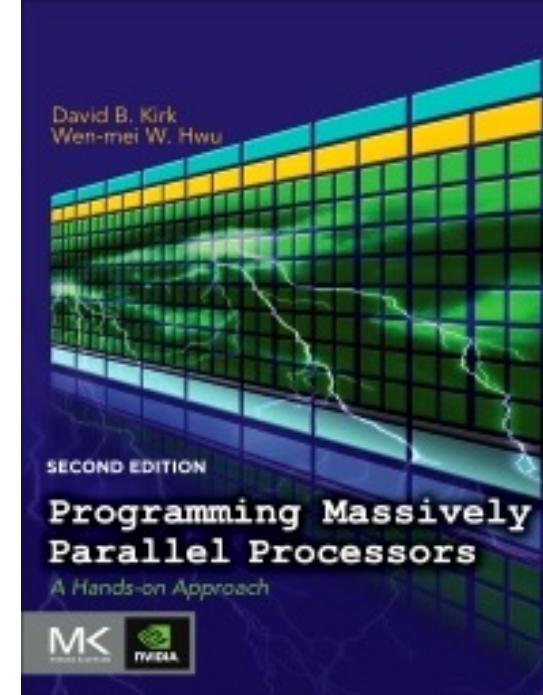
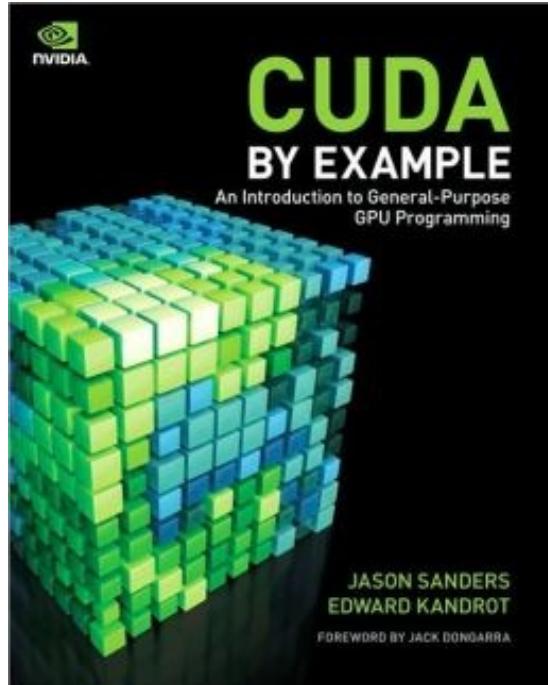
NVIDIA HPC SDK: <https://docs.nvidia.com/hpc-sdk/index.html>

CUDA C: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

CUDA Fortran: <https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/>

Many resources here: <https://www.gphuhackathons.org/technical-resources>

Good books to get started



# Nvidia CUDA development tools

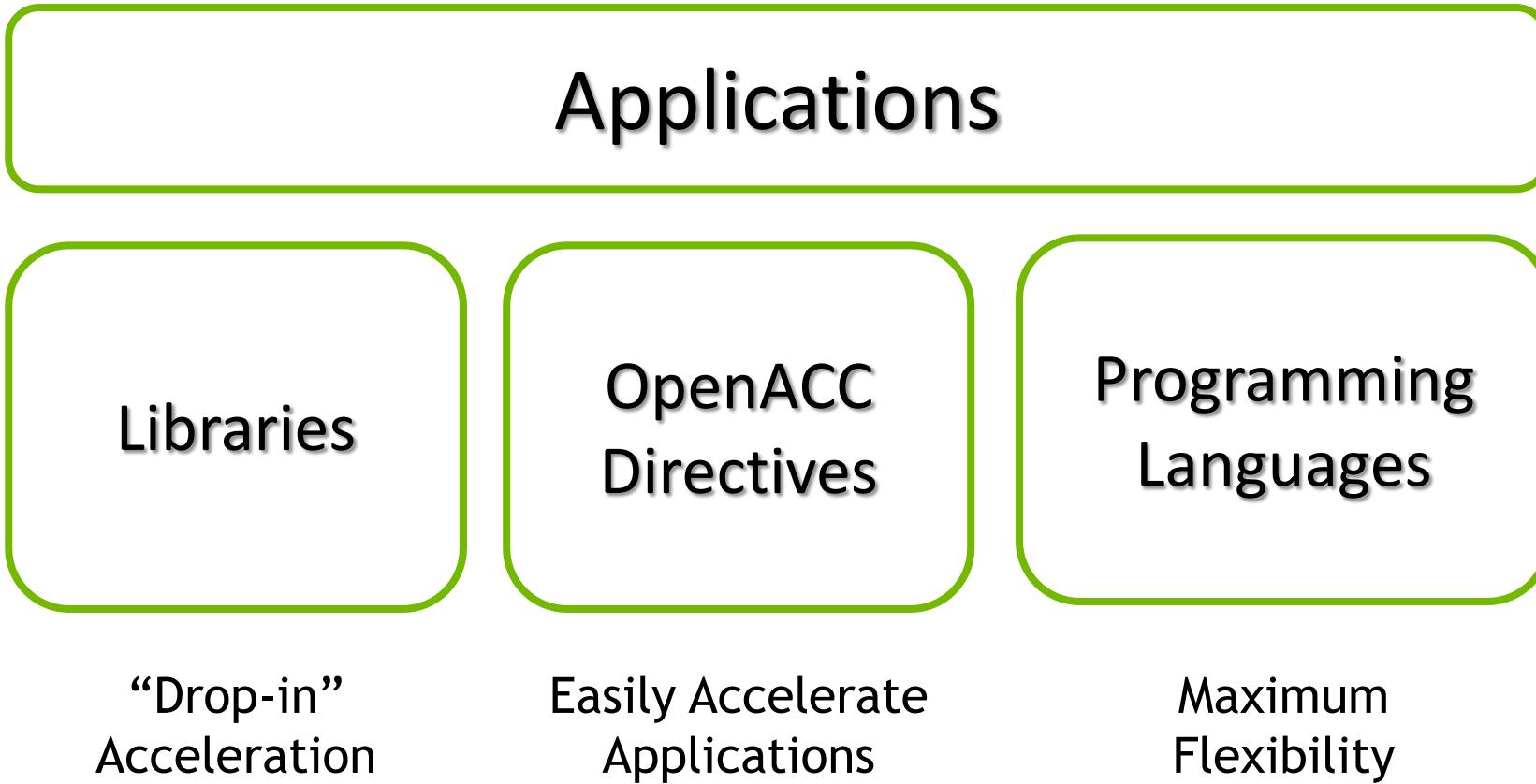
## CUDA Toolkit/SDK (free)

- CUDA C compiler (nvcc)
- Libraries (cuBLAS, cuFFT, cuDNN, cuRAND, cuSPARSE, cuSOLVER, Thrust, CUDA Math lib)
- Debugging tools (CUDA-gdb, CUDA-memcheck)
- Profiling tools (nvprof, nvvp, Nsight Systems/Compute)
- Code samples available at <https://github.com/NVIDIA/cuda-samples>
- <https://developer.nvidia.com/cuda-zone>
- <https://nvidia.com/getcuda>

## Activate on Expanse GPU nodes

```
$> module reset  
$> module load cuda12.2/toolkit
```

# 3 ways to program GPUs



**Note: Deep Learning frameworks like Tensorflow and PyTorch come with built-in GPU support**

- This means you do not have to install or use the CUDA toolkit separately

# PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

## ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; });

```

```
do concurrent (i = 1:n)
    y(i) = y(i) + a*x(i)
enddo

```

```
import cunumeric as np
...
def saxpy(a, x, y):
    y[:] += a*x

```

## INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}

```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}

```

## PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
            float *x, float *y) {
    int i = blockIdx.x*blockDim.x +
            threadIdx.x;
    if (i < n) y[i] += a*x[i];
}

int main(void) {
    ...
    cudaMemcpy(d_x, x, ...);
    cudaMemcpy(d_y, y, ...);

    saxpy<<<(N+255)/256,256>>>(...);

    cudaMemcpy(y, d_y, ...);
}

```

## ACCELERATION LIBRARIES

Core

Math

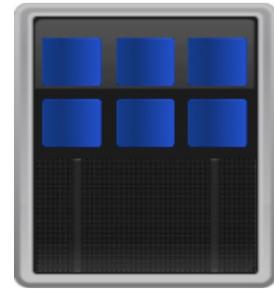
Communication

Data Analytics

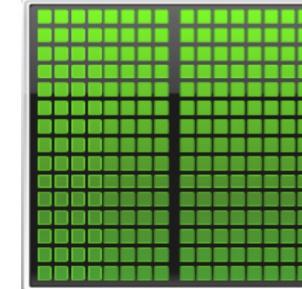
AI

Quantum

# NUMERICAL COMPUTING IN PYTHON



- Mathematical focus
- Operates on arrays of data
  - *ndarray*, holds data of same type
- Many years of development
- Highly tuned for CPUs



- NumPy like interface
- Trivially port code to GPU
- Copy data to GPU
  - CuPy *ndarray*
- Data interoperability with DL frameworks, RAPIDS, and Numba
- Uses high tuned NVIDIA libraries
- Can write custom CUDA functions

# CUPY

A NumPy like interface to GPU-acceleration ND-Array operations

**BEFORE**

```
import numpy as np  
  
size = 4096  
A = np.random.randn(size,size)  
  
Q, R = np.linalg.qr(A)
```

**AFTER**

```
import cupy as np  
  
size = 4096  
A = np.random.randn(size,size)  
  
Q, R = np.linalg.qr(A)
```



*52x Speedup!*



# CUDA C Basics

# Nvidia CUDA

See <https://developer.nvidia.com/cuda-zone>

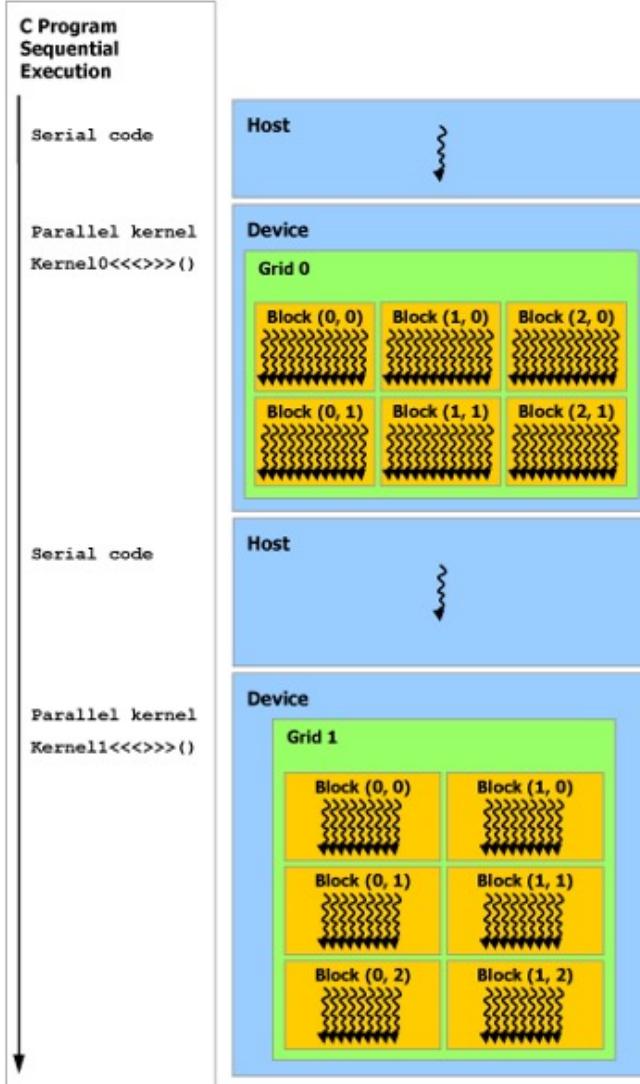
## CUDA C

- Solution to run C seamlessly on GPUs (Nvidia only)
- De-facto standard for high-performance code on Nvidia GPUs
- Nvidia proprietary
- Modest extensions but major rewriting of code

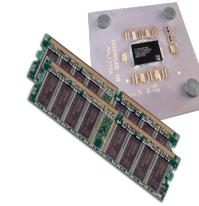
## CUDA Fortran

- Supports CUDA extensions in Fortran, developed by Portland Group Inc (PGI)
- Available in the Nvidia Fortran compiler (formerly PGI Fortran Compiler)
- PGI is now part of Nvidia

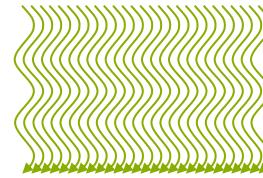
# Heterogeneous Computing



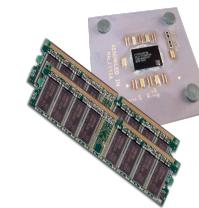
*serial code*



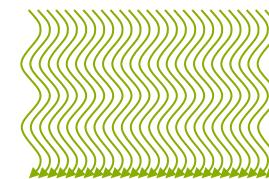
*parallel code*



*serial code*

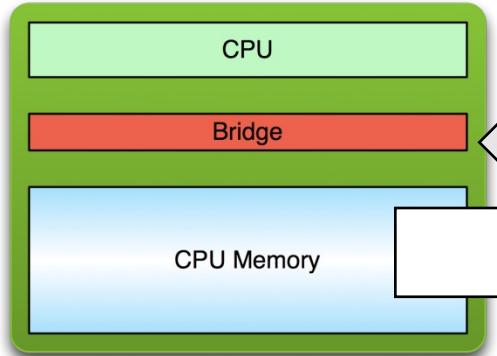


*parallel code*

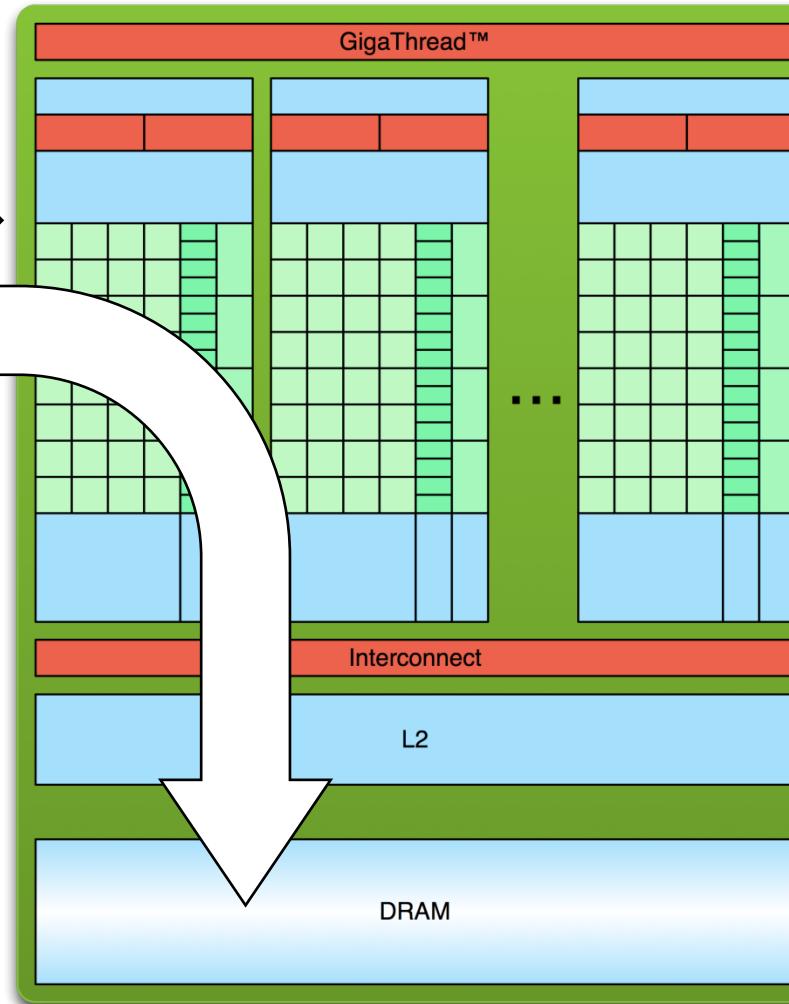


# Processing Flow

Host



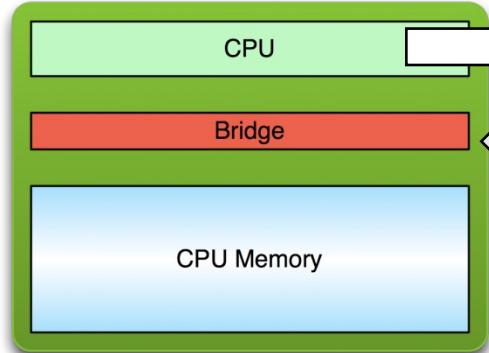
Device



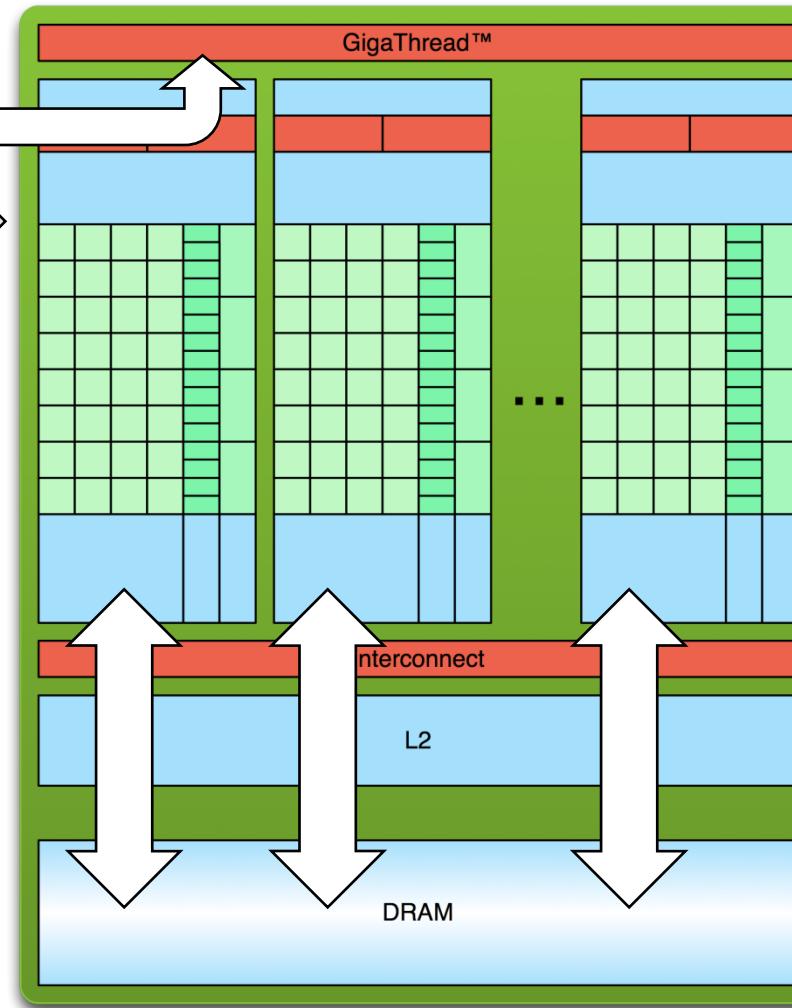
1. Copy input data from CPU memory to GPU memory

# Processing Flow

Host



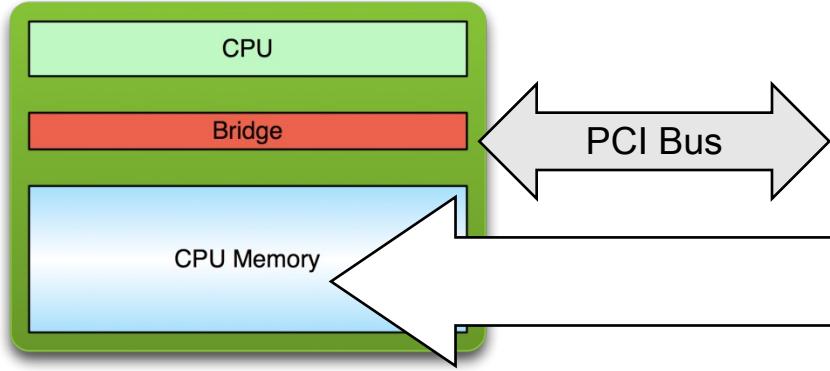
Device



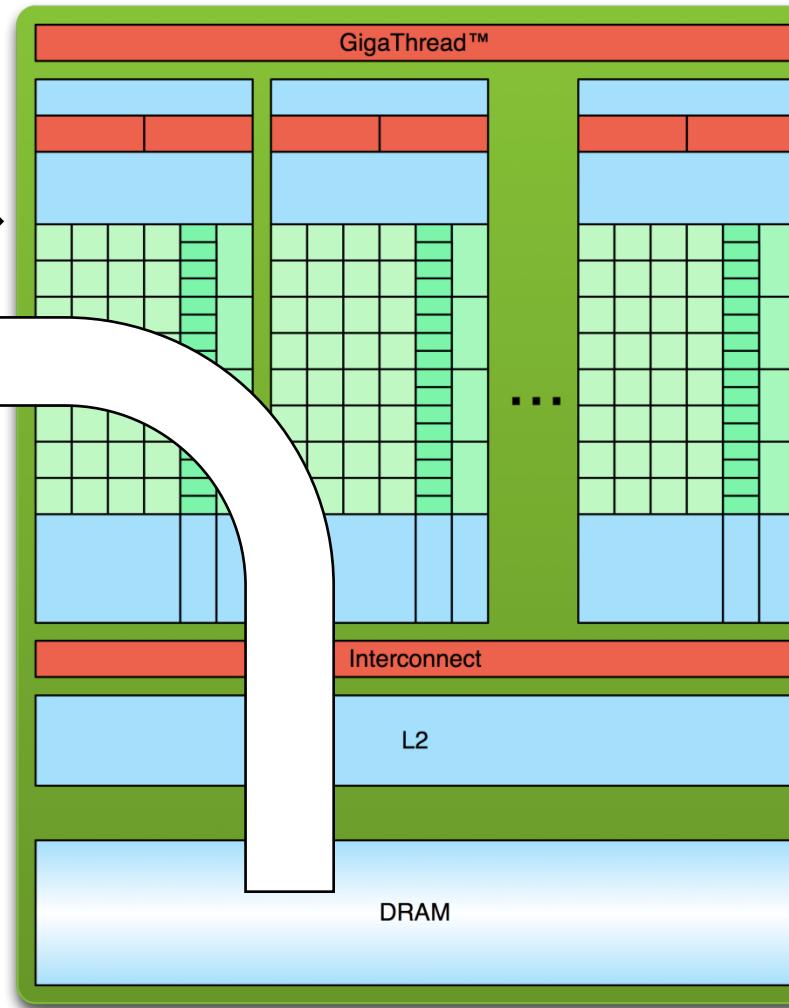
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

# Processing Flow

Host



Device



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

# Some CUDA basics

## Kernel

- In CUDA, a kernel is code (typically a function), that can be executed on the GPU.
- The kernel code operates in lock-step on the multiprocessors of the GPU.  
(In so-called warps, currently consisting of 32 threads)
- SIMD – single instruction multiple threads

## Thread

- A thread is an execution of a kernel with a given index.
- Each thread uses its index to access a subset of data (e.g. array) to operate on.

## Block

- Threads are grouped into blocks, which are guaranteed to execute on the same multiprocessor.
- Threads within a thread block can synchronize and share data

## Grid

- Thread blocks are arranged into a grid of blocks.
- The number of threads per block times the number of blocks gives the total number of running threads.

# Some CUDA basics

## Threads, blocks, grids, warps

### Grids

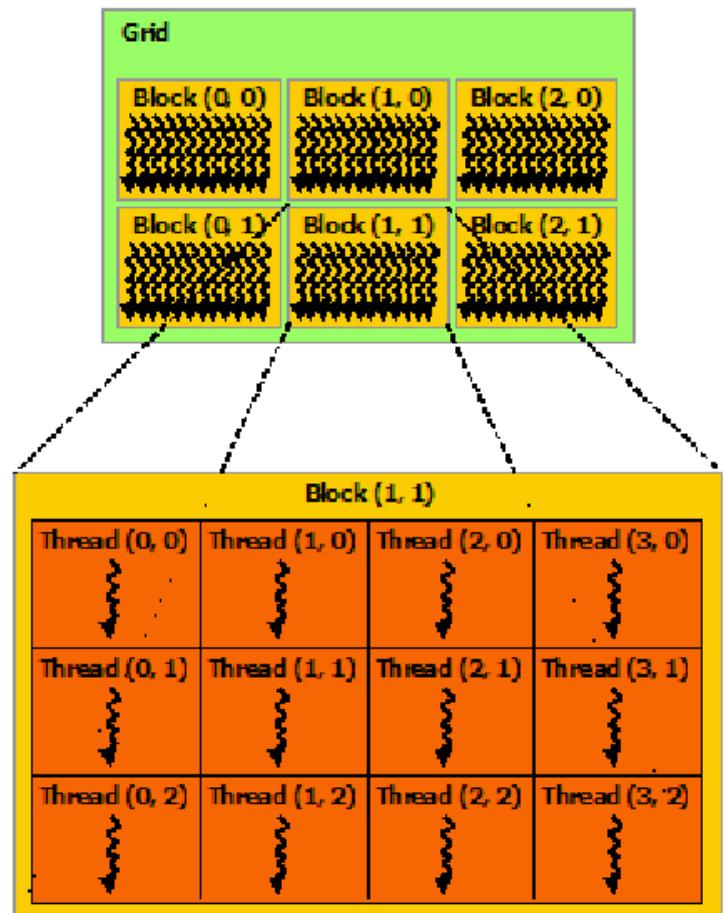
- Grids map to GPUs

### Blocks

- Blocks map to the multiprocessors (MP)
- Blocks are never split across MPs
- Multiple blocks can execute simultaneously on an MP

### Threads

- Threads are executed on stream processors (GPU cores)
- Warps are groups of threads that execute simultaneously, in lock-step (currently 32, not guaranteed to remain fixed).



# Some CUDA basics

## CUDA built-in variables

- Following variables allow to compute the ID of each individual thread that is executing in a grid block.

## Block indexes

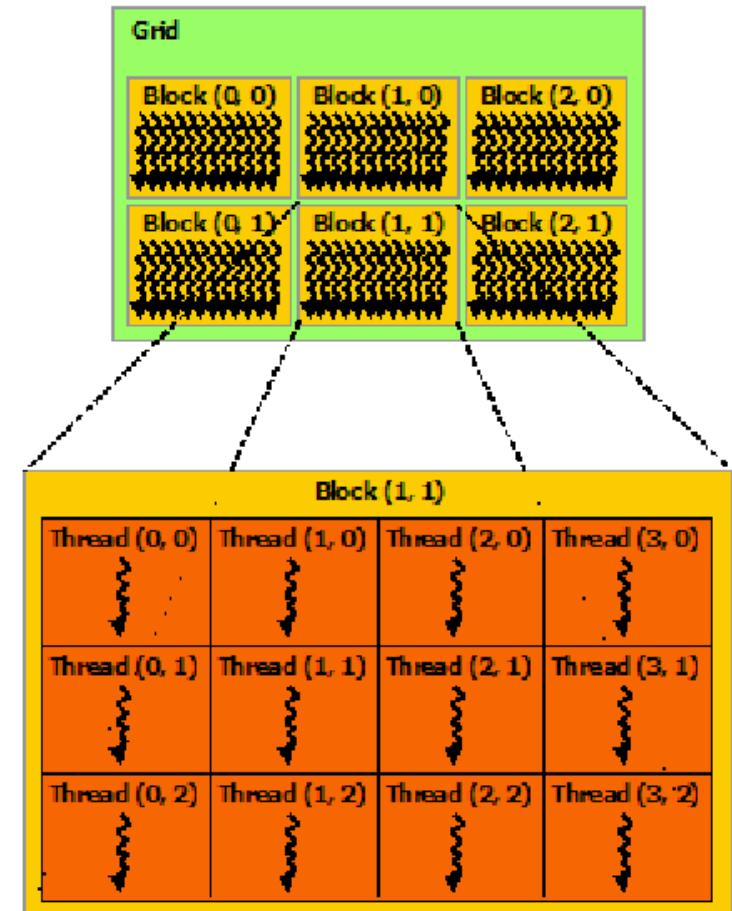
- `gridDim.x`, `gridDim.y`, `gridDim.z` (unused)
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- Variables that return the grid dimension (number of blocks) and block ID in the x-, y-, and z-axis.

## Thread indexes

- `blockDim.x`, `blockDim.y`, `blockDim.z`
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- Variables that return the block dimension (number of threads per block) and thread ID in the x-, y-, and z-axis.

Example in the figure is executing 72 threads

- (3 x 2) blocks = 6 blocks
- (4 x 3) threads per block = 12 threads per block



# Some CUDA basics

## `__global__` keyword

- Function that executes on the device (GPU), must return `void`, and is called from host code.

```
__global__ vector_add_kernel(int *a, int *b, int *c, int n) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (tid < n) {
        c[tid] = a[tid] + b[tid];
        tid += stride;
    }
}
```

## CUDA API handles device memory

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Equivalent to C `malloc()`, `free()`, `memcpy()`
- `cudaMemcpy()` is used to transfer data between CPU and GPU memory.

## CUDA kernel launch specification

- Triple angle bracket determines grid and block size (i.e. total number of threads) for kernel launch:

```
vector_add_kernel<<<dim3(bx,by,bz), dim3(tx,ty,tz)>>>(d_a, d_b, d_c, N);
```

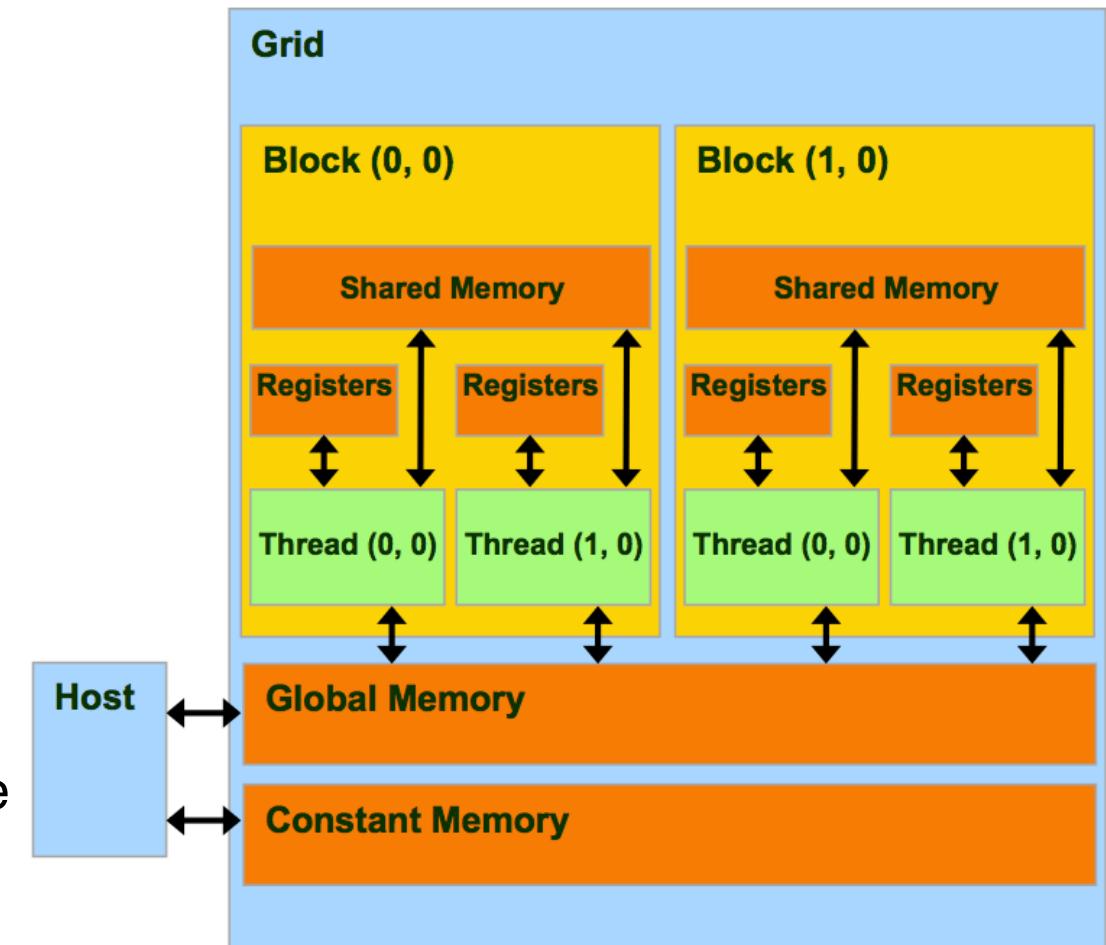
# Some CUDA basics

## CUDA memory hierarchy

- Host memory (x86 server)
- Device memory (GPU)

## Device memory

- **Global memory**  
visible to all threads, slow
- **Shared memory**  
visible to all threads in a block, fast on-chip
- **Registers**  
per-thread memory, fast on-chip
- **Local memory**  
per-thread, slow, stored in Global Memory space
- **Constant memory**  
visible to all threads, read only, off-chip, cached  
broadcast to all threads in a half-warp (16 threads)



# General CUDA programming strategy

## Avoid data transfers between CPU and GPU

- These are slow due to low PCI express bus bandwidth

## Minimize access to global memory

- Hide memory access latency by launching many threads

## Take advantage of fast shared memory by tiling data

- Partition data into subsets that fit into shared memory
- Handle each data subset with one thread block
- Load the subset from global to shared memory using multiple threads to exploit parallelism in memory access
- Perform computation on data subset in shared memory (each thread in thread block can access data multiple times)
- Copy results from shared memory to global memory

# **Directive based GPU programming with OpenACC**

# Directive based programming

## OpenACC

- See <https://www.openacc.org>
- Open standard for expressing accelerator parallelism
- Designed to make porting to GPUs easy, quick, and portable
- OpenMP-like compiler directives language
  - If the compiler does not understand the directives, it will ignore them.
  - Same code can work with or without accelerators.
- Fortran and C
- Full support by Nvidia (formerly PGI compilers) and Cray compilers on Crays
- Partial support by GNU compilers (experimental since version 5.1)
- Also some less commonly used and experimental compilers

## OpenMP

- See <https://www.openmp.org>
- Not mature for GPUs, will not discuss here

# Directive based programming

## Nvidia HPC SDK

- See <https://developer.nvidia.com/hpc-sdk>
- Available on SDSC Expanse
- Nvidia Fortran / C / C++ / CUDA compilers (nvfortran, nvc, nvc++, nvcc)
  - compiler support OpenACC, CUDA Fortran, CUDA C/C++, OpenMP (for multicore CPUs)
- nvprof performance profiler
- cuda-gdb debugger
- GPU-enabled libraries
- OpenACC code samples

## Activate on Expanse GPU nodes

```
$> module load nvhpc/21.9
```

# A simple OpenACC exercise: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

# Complete SAXPY code

## Trivial first example

- Apply a loop directive
- Learn compiler commands

```
#include <stdlib.h>

void saxpy(int n,
           float a,
           float *x,
           float * restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

    float *x = (float*)malloc(N *
sizeof(float));
    float *y = (float*)malloc(N *
sizeof(float));

    for (int i = 0; i < N; ++i)
    {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

# Compile and run SAXPY OpenACC code

- C:

```
pgcc -acc -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

```
pgf90 -acc -Minfo=accel -o saxpy_acc saxpy.f90
```

- Compiler output:

```
pgcc saxpy.c -acc -Minfo=accel -o saxpy-gpu.x
saxpy:
  8, Generating copyin(x[:n])
      Generating copy(y[:n])
  9, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
      9, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
```

# OpenACC directives syntax

## Fortran

```
!$acc directive [clause [, clause] ...]
```

Often paired with a matching end directive  
surrounding a structured code block

```
!$acc end directive
```

## kernels construct

```
!$acc kernels [clause ...]  
    structured code block  
!$acc end kernels
```

## Clauses

```
if( condition )  
async( expression )  
or data clauses
```

## C

```
#pragma acc directive [clause [, clause] ...]
```

Often followed by a structured code block

## kernels construct

```
#pragma acc kernels [clause ...]  
{ structured code block }
```

# OpenACC directives syntax

## Data clauses

- `copy ( list )` Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- `copyin ( list )` Allocates memory on GPU and copies data from host to GPU when entering region.
- `copyout ( list )` Allocates memory on GPU and copies data to the host when exiting region.
- `create ( list )` Allocates memory on GPU but does not copy.
- `present ( list )` Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

# GPU profiling

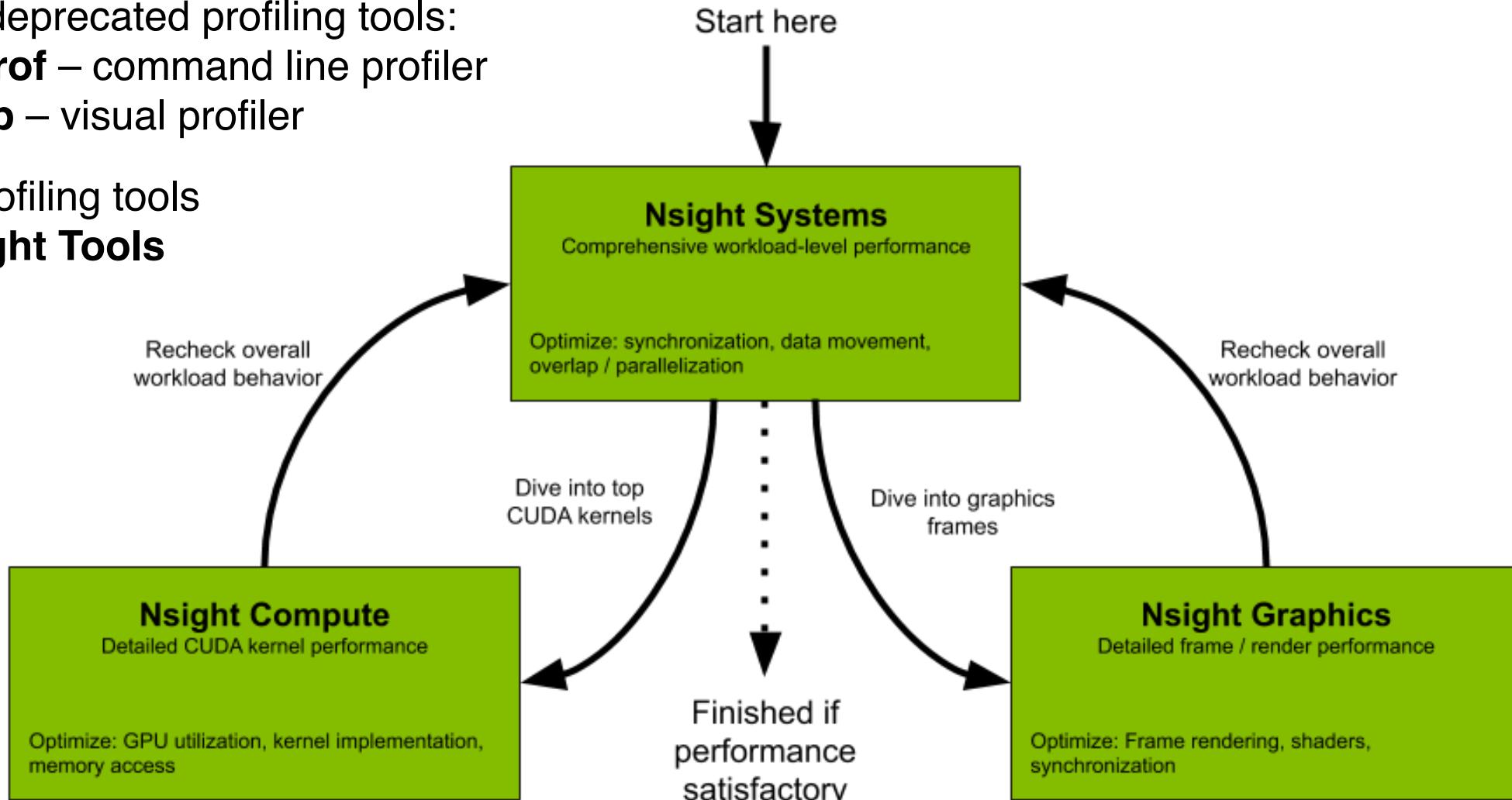
# Nvidia profiling tools

Older, deprecated profiling tools:

- **nvprof** – command line profiler
- **nvvp** – visual profiler

New profiling tools

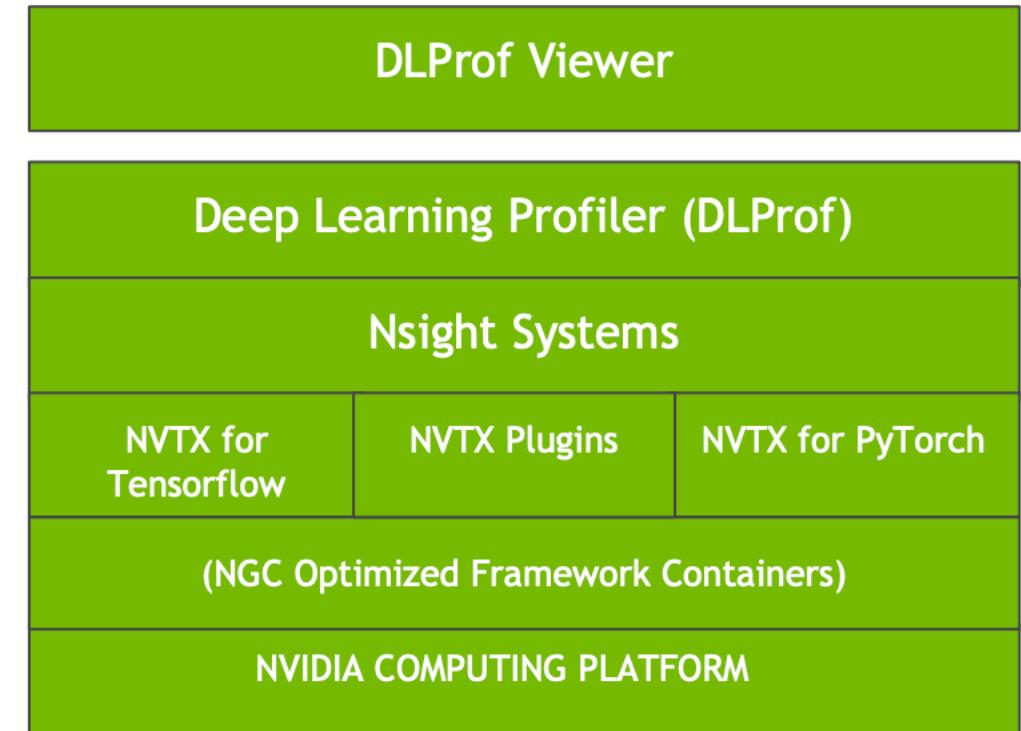
- **Nsight Tools**



# Nvidia profiling tools

## DLProf: Profiler for Deep Learning applications

- Nsight Systems and Nsight Compute have been built using CUDA Profiling Tools Interface (CUPTI)
- NVTX Nvidia Tools Extension Library is a way to annotate source code with markers
- NVTX markers are used to annotate and focus on sections of code important to the user
- TensorFlow and PyTorch optimized by Nvidia contain support for NVTX markers
- NVTX plugins are Python bindings for users to add markers easily
- DLProf calls Nsight systems to collect profile data and correlate with the DL model



# Nvidia profiling tools

**DLProf:** Profiler for Deep Learning applications

- Are my GPUs being utilized?
- Am I using Tensor Cores?
- How can I improve performance?



FW Support: TF1, TF2, PyT, and TRT  
Lib Support: DALI, NCCL



Visualize Analysis and Recommendations

# Nvidia profiling tools

## DLProf: Profiler for Deep Learning applications



1. TensorFlow and TRT require no additional code modification
2. Profile using DLProf CLI - prepend with ***dlprof***
3. Visualize results with DLProf Viewer



1. Add few lines of code to your training script to enable ***nvidia\_dlprof\_pytorch\_nvtx*** module
2. Profile using DLProf CLI - prepend with ***dlprof***
3. Visualize with DLProf Viewer



# Outline

## We will cover the following topics

- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - GPU accelerated libraries
  - CUDA intro
  - OpenACC intro
- **SDSC Expanse GPU nodes**
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

# SDSC Expanse

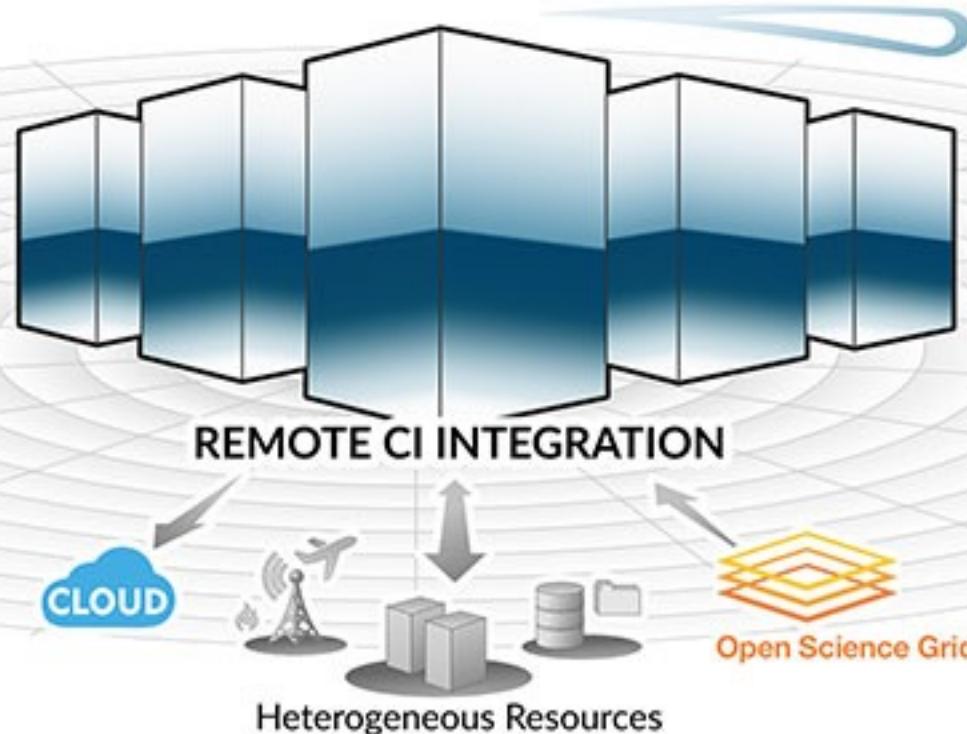
Launched in Fall 2020

## HPC RESOURCE

13 Scalable Compute Units  
728 Standard Compute Nodes  
52 GPU Nodes: 208 GPUs  
4 Large Memory Nodes

## DATA CENTRIC ARCHITECTURE

12PB Perf. Storage: 140GB/s, 200k IOPS  
Fast I/O Node-Local NVMe Storage  
7PB Ceph Object Storage  
High-Performance R&E Networking



## LONG-TAIL SCIENCE

Multi-Messenger Astronomy  
Genomics  
Earth Science  
Social Science

## INNOVATIVE OPERATIONS

Composable Systems  
High-Throughput Computing  
Science Gateways  
Interactive Computing  
Containerized Computing  
Cloud Bursting

# Expanse Heterogeneous Architecture

## System Summary

- 13 SDSC Scalable Compute Units (SSCU)
- 728 Standard Compute Nodes
- 93,184 Compute Cores
- 200 TB DDR4 Memory
- 52x 4-way GPU Nodes w/NVLINK
- **208 V100 GPUs**
- 4x 2TB Large Memory Nodes
- HDR 100 non-blocking Fabric
- 12 PB Lustre High Performance Storage
- 7 PB Ceph Object Storage
- 1.2 PB on-node NVMe
- Dell EMC PowerEdge
- Direct Liquid Cooled

### Scalable Compute Unit

Non-blocking fabric  
56 CPU nodes  
4 GPU nodes

60 HDR 100 to nodes  
10x HDR 200 to L2

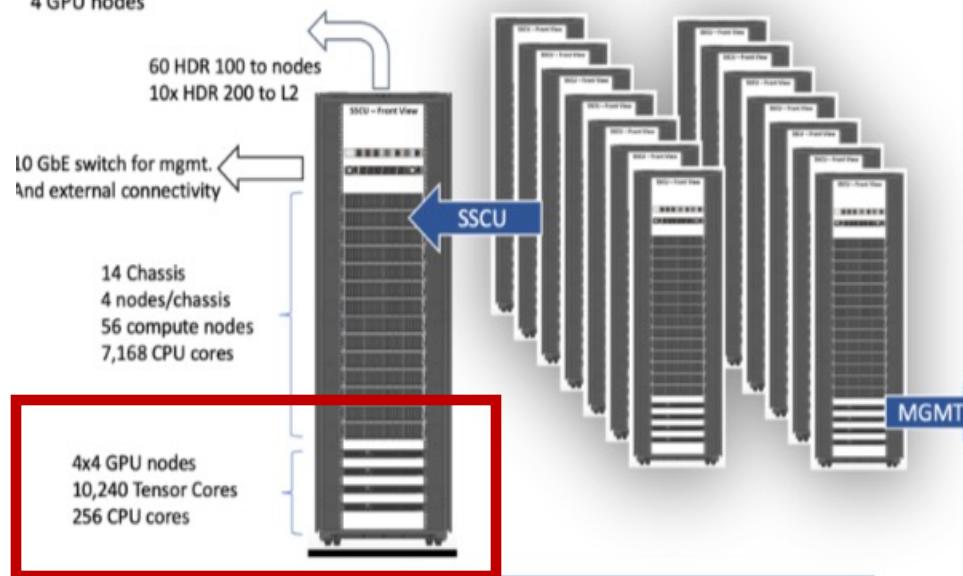
I0 GbE switch for mgmt.  
And external connectivity

14 Chassis  
4 nodes/chassis  
56 compute nodes  
7,168 CPU cores

4x4 GPU nodes  
10,240 Tensor Cores  
256 CPU cores

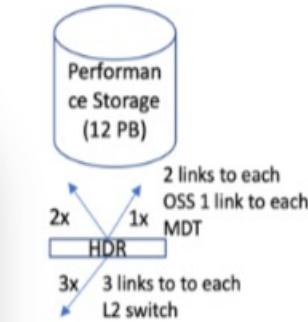
### System Layout

1 row 7 SSCU  
1 row 6 SSCU + Core Mgmt. rack



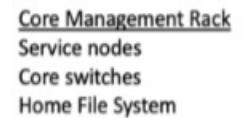
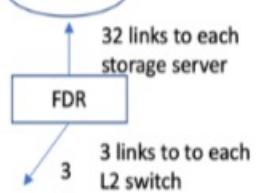
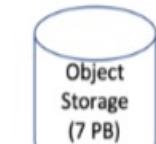
### Performance Storage

12PB Lustre  
7 HA OSS pairs  
4 NVMe HA Metadata Servers



### Object Storage

7 PB Ceph  
32 storage servers



### Core Management Rack

Service nodes

Core switches

Home File System

2 login nodes

NFS server

2 hosting nodes

Cluster mgmt. nodes

4 large memory nodes

2TB/node

64 CPU cores/node

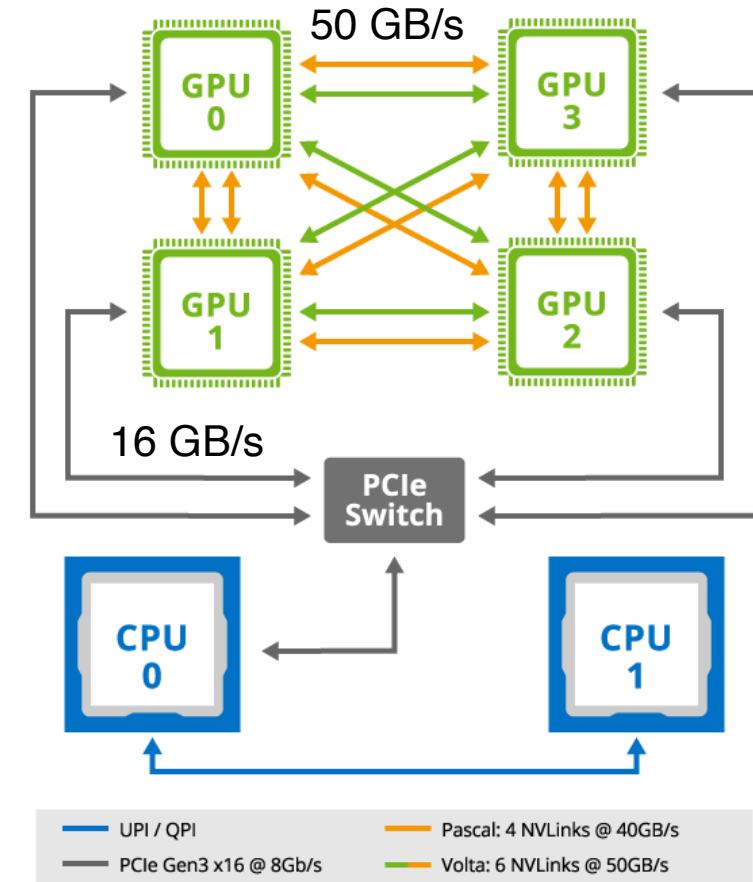
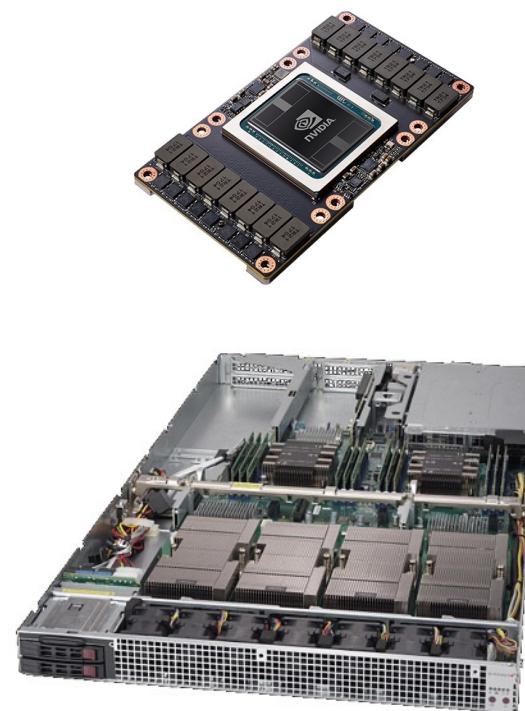
### Level 2 IB Fabric

- 5 HDR switches (1 shown)
- 2x HDR200 down to each of the 13 non-blocking SSCUs
- 3 uplinks each to Performance Storage switch
- 2 uplink each to Cloud Storage switch

# SDSC Expanse GPU nodes with Nvidia V100 SXM2

## 52 GPU nodes

- 2 x 20-core Intel Xeon Gold 6248 (Cascade Lake) CPUs
- 384 GB RAM (131 GB/s)
- 4 x Nvidia V100 SXM2 GPUs
- 32 GB HBM2 RAM per GPU (897 GB/s)
- 1.6 TB NVMe/node



## User guide:

[https://www.sdsc.edu/support/user\\_guides/expanse.html](https://www.sdsc.edu/support/user_guides/expanse.html)

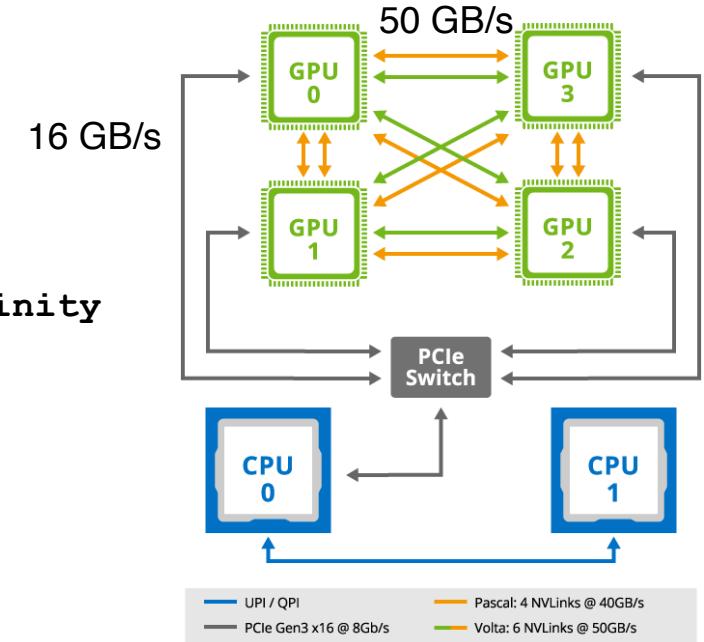
# SDSC Expanse GPU nodes with Nvidia V100 SXM2

## Node topology

	GPU0	GPU1	GPU2	GPU3	mlx5_0	CPU Affinity	NUMA Affinity
GPU0	X	NV2	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU1	NV2	X	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU2	NV2	NV2	X	NV2	SYS	1,3,5,7,9,11	1
GPU3	NV2	NV2	NV2	X	SYS	1,3,5,7,9,11	1
mlx5_0	NODE	NODE	SYS	SYS	X		

### Legend:

- X = Self
- SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
- NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
- PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
- PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
- PIX = Connection traversing at most a single PCIe bridge
- NV# = Connection traversing a bonded set of # NVLinks



# SDSC Expanse – V100 SXM2 GPUs

## Each GPU

- 32 GB HBM2 RAM (897 GB/s)
- 80 SMs (Streaming Multiprocessors)
- 64 FP32 cores / SM (5120 total)
- 32 FP64 cores / SM (2560 total)
- 8 Tensor cores / SM (640 total)
- 300 Watt TDP

## Peak performance

- 7.8 FP64 TFLOPs
- 15.7 FP32 TFLOPs
- 31.3 FP16 TFLOPs
- 125 Tensor TFLOPs



# SDSC Expanse login

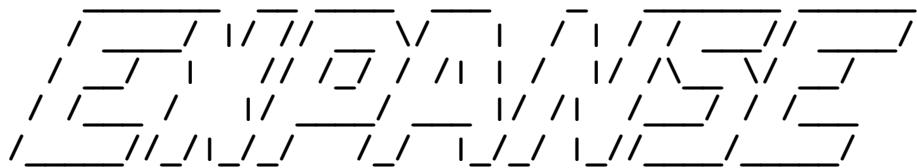
## Login

```
$> ssh train107@expanse.sdsc.edu  
Welcome to Bright release 9.0
```

```
Based on CentOS Linux 8  
ID: #000002
```

---

WELCOME TO



---

Use the following commands to adjust your environment:

```
'module avail'           - show available modules  
'module add <module>' - adds a module to your environment for this session  
'module initadd <module>' - configure module to be loaded at every login
```

---

```
Last login: Tue Apr 27 01:58:53 2021 from 136.26.112.138  
[train107@login01 ~]$
```

# SDSC Expanse GPU nodes

- Obtain access to one GPU on a shared GPU node

```
[train107@login02 ~]$ srun-gpu-shared # This is an alias (check alias for full command)
[train107@exp-16-57 ~]
```

- Reset, then load GPU related modules

```
module reset
# Load CUDA Toolkit
module load cuda12.2/toolkit

# Load Nvidia HPC SDK
module load nvhpc/21.9
```

# SDSC Expanse GPU nodes

- Check Nvidia CUDA C compiler

```
[agoetz@exp-8-59 ~]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, v11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

- Check PGI C compiler / NVHPC C compiler

```
[agoetz@exp-8-59 ~]$ pgcc --version
pgcc (aka nvc) 21.9-0 64-bit target on x86-64 Linux -tp skylake
PGI Compilers and Tools
Copyright (c) 2021, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
```

```
[agoetz@exp-8-59 ~]$ nvc --version
nvc 21.9-0 64-bit target on x86-64 Linux -tp skylake
NVIDIA Compilers and Tools
Copyright (c) 2021, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
```

# SDSC Expanse GPU nodes

- Check available GPUs using Nvidia system management interface

```
[agoetz@exp-8-59 ~]$ nvidia-smi
Tue Apr 27 02:45:26 2021
+-----+
| NVIDIA-SMI 450.51.05      Driver Version: 450.51.05      CUDA Version: 11.0      |
|-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC  | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
|          |          |              |             |          MIG M.   |
|-----+-----+-----+
|  0  Tesla V100-SXM2... On | 00000000:18:00.0 Off |                0 | | |
| N/A   45C     P0    67W / 300W |        0MiB / 32510MiB |      0%     Default |
|          |          |              |             |          N/A   |
+-----+-----+-----+
...
...
```

# SDSC Expanse GPU nodes

- Processes running on the GPU are also listed.

```
...
+-----+
| Processes:
| GPU   GI   CI          PID    Type  Process name           GPU Memory |
|        ID   ID
|=====|
| No running processes found
+-----+
```

- There should be no jobs running on the GPU assigned to you
- The nodes of the shared GPU queue are configured for the CUDA runtime to use only the requested number of GPUs.

# SDSC Expanse GPU nodes

## CUDA Toolkit Samples

- CUDA Toolkit code samples are available for the Toolkit (does not require GPU node access)

```
[agoetz@exp-8-59 ~]$ cd /scratch/$USER/$SLURM_JOBID      # work in local scratch
[agoetz@exp-8-59 ~]$ tar xvf $CIML24_DATA_DIR/cuda-samples-v12.2.tar.gz
[agoetz@exp-8-59 ~]$ ln -s cuda-samples-12.2 cuda-samples
```

- Explore CUDA Toolkit samples – great resource!

```
[agoetz@exp-8-59 ~]$ cd cuda-samples
[agoetz@exp-8-59 cuda-samples]$ ls Samples
0_Introduction  2_Concepts_and_Techniques  4_CUDA_Libraries  6_Performance
1_Utils         3_CUDA_Features          5_Domain_Specific
```

- Compile CUDA Toolkit samples

```
[agoetz@exp-8-59 cuda-samples]$ make -k -j 10
make[1]: Entering directory `/home/agoetz/CUDA_samples/0_Simple/simpleMultiCopy'
/usr/local/cuda-10.2/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode
arch=compute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -gencode
...
arch=compute_75,code=sm_75 -gencode arch=compute_75,code=compute_75 -o simpleMultiCopy.o -c
simpleMultiCopy.cu
```

# SDSC Expanse GPU nodes

## CUDA Toolkit Samples

- Compilation takes a while, executables will reside in sub directory `bin/x86_64/linux/release/`
- Can also compile individual examples, e.g. `deviceQuery` (prints information on available GPUs)

```
[agoetz@exp-1-57 cuda-samples]$ cd Samples/1_Utilsilities/deviceQuery
[agoetz@exp-1-57 deviceQuery]$ make
/usr/local/cuda-10.2/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_70,code=sm_70
...
[agoetz@exp-1-57 deviceQuery]$ ./deviceQuery
./deviceQuery Starting...
```

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-32GB"

CUDA Driver Version / Runtime Version	11.0 / 10.2
CUDA Capability Major/Minor version number:	7.0
Total amount of global memory:	32510 MBytes (34089730048 bytes)
(80) Multiprocessors, ( 64) CUDA Cores/MP:	5120 CUDA Cores

# SDSC Expanse GPU nodes

## CUDA Toolkit

- Matrix multiplication example

```
[agoetz@exp-3-58 ~]$ cd ~/cuda-samples/Samples/0_Introduction/
[agoetz@exp-3-58 0_Simple]$ ./matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Volta" with compute capability 7.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 3274.22 GFlop/s, Time= 0.040 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

- Matrix multiplication example with CUBLAS

```
[agoetz@exp-3-58 ~]$ cd ~/cuda-samples/Samples/4_CUDA_Libraries/
[agoetz@exp-3-58 0_Simple]$ ./matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Volta" with compute capability 7.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 7588.93 GFlop/s, Time= 0.026 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

A wide-angle photograph of a coastal scene at sunset. The sky is a gradient from blue to orange and yellow. In the foreground, there are large, dark rocks on a beach. In the middle ground, a long wooden pier extends from the shore into the ocean. A few people are visible on the beach and rocks. In the background, there's a hillside with houses and trees.

**Thank you!**