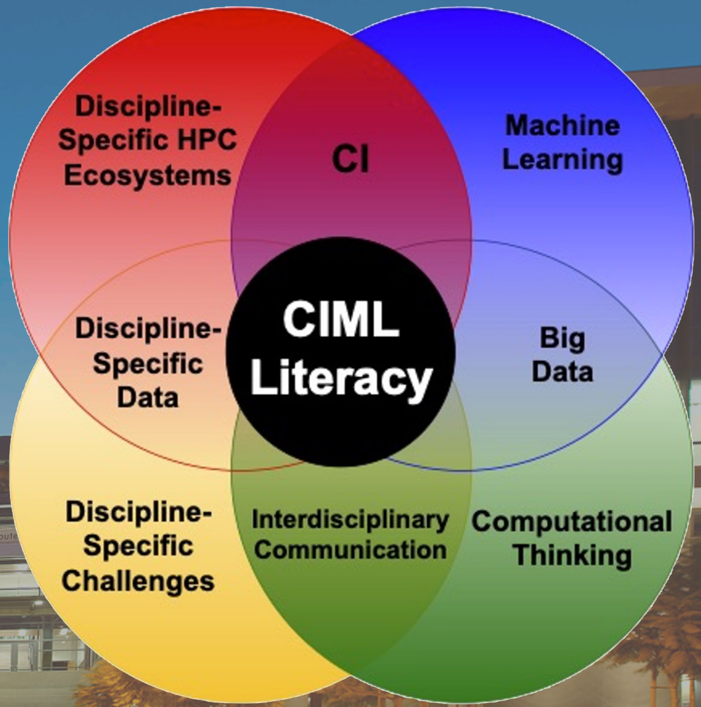
CIML Summer Institute:
GPU Computing – Hardware architecture and software infrastructure



Andreas W Götz

San Diego Supercomputer Center
University of California, San Diego

Tuesday, June 27, 2023, 3:00 pm to 4:30 pm, PDT

Outline

We will cover the following topics

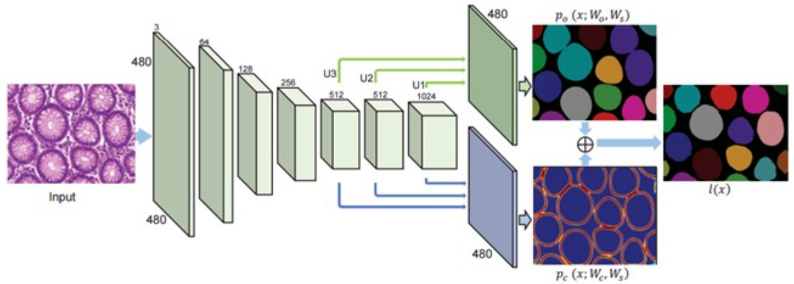
- Brief Intro on current ML trends
- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
  - GPU accelerated libraries
  - CUDA intro
  - OpenACC intro
- SDSC Expanse GPU nodes
  - Accessing GPU nodes
  - Running GPU jobs
  - Developing GPU software

Machine Learning in Computational Sciences

AI and ML have been around for a long time

Why the current interest?

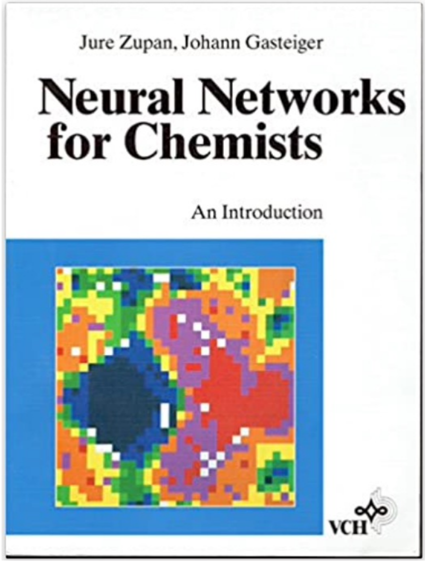
- Flood of available data (in particular with internet)
- Breakthroughs in theory and algorithms (deep learning, generative AI, …)
- Increasing interest and support from industry
- **Increase in computational power**

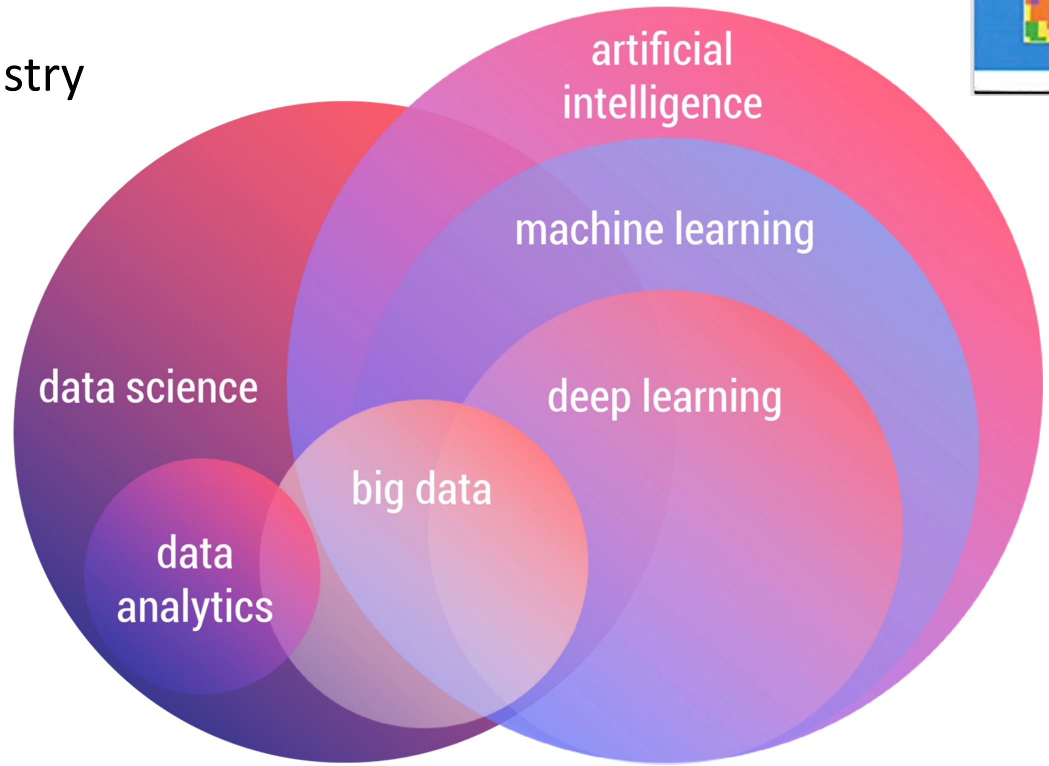


Adoption of machine learning

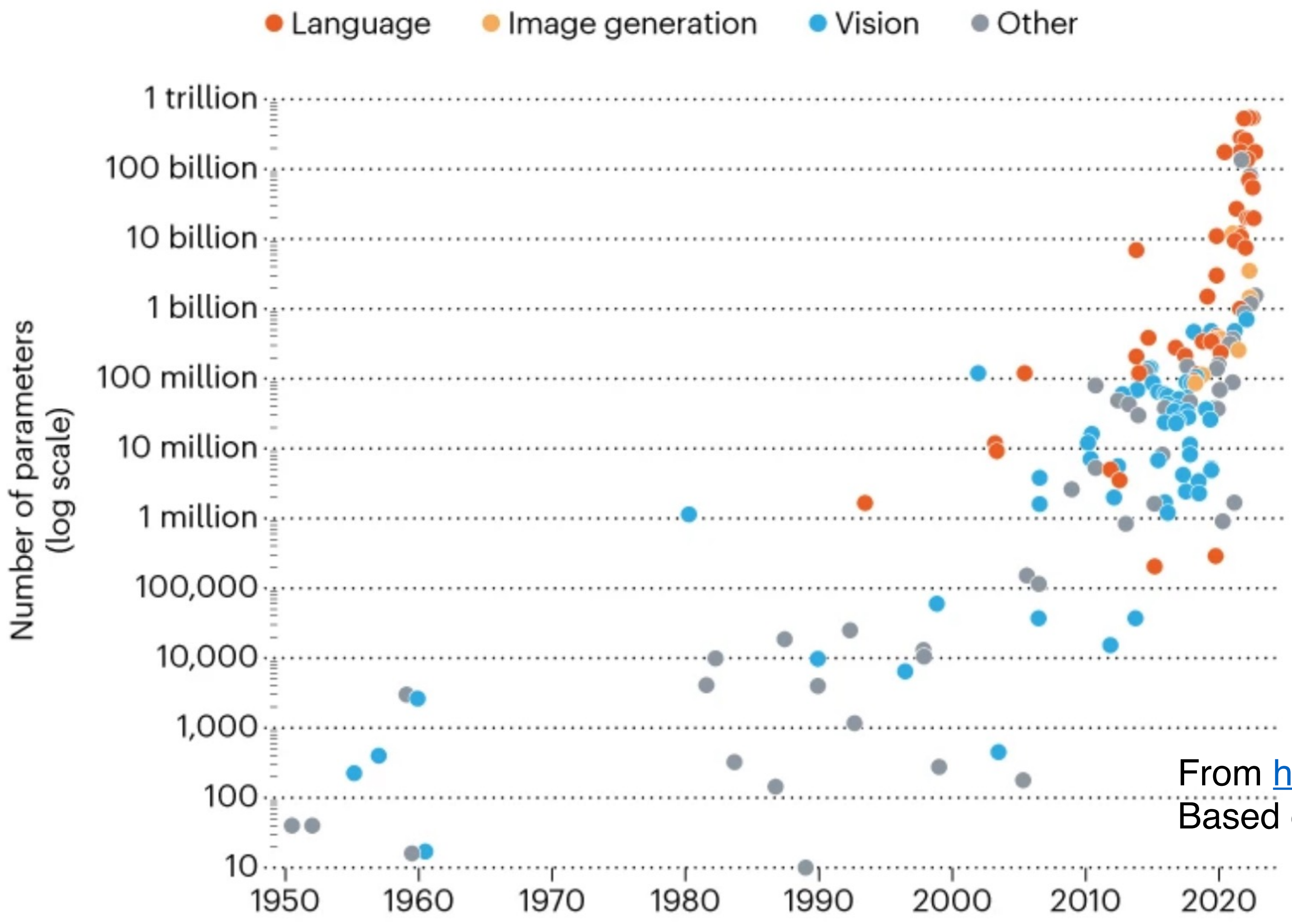
- Practically all science fields
- Augment modeling and simulation

Neural Networks for Chemists, published in 1993





Drive to Bigger AI Models

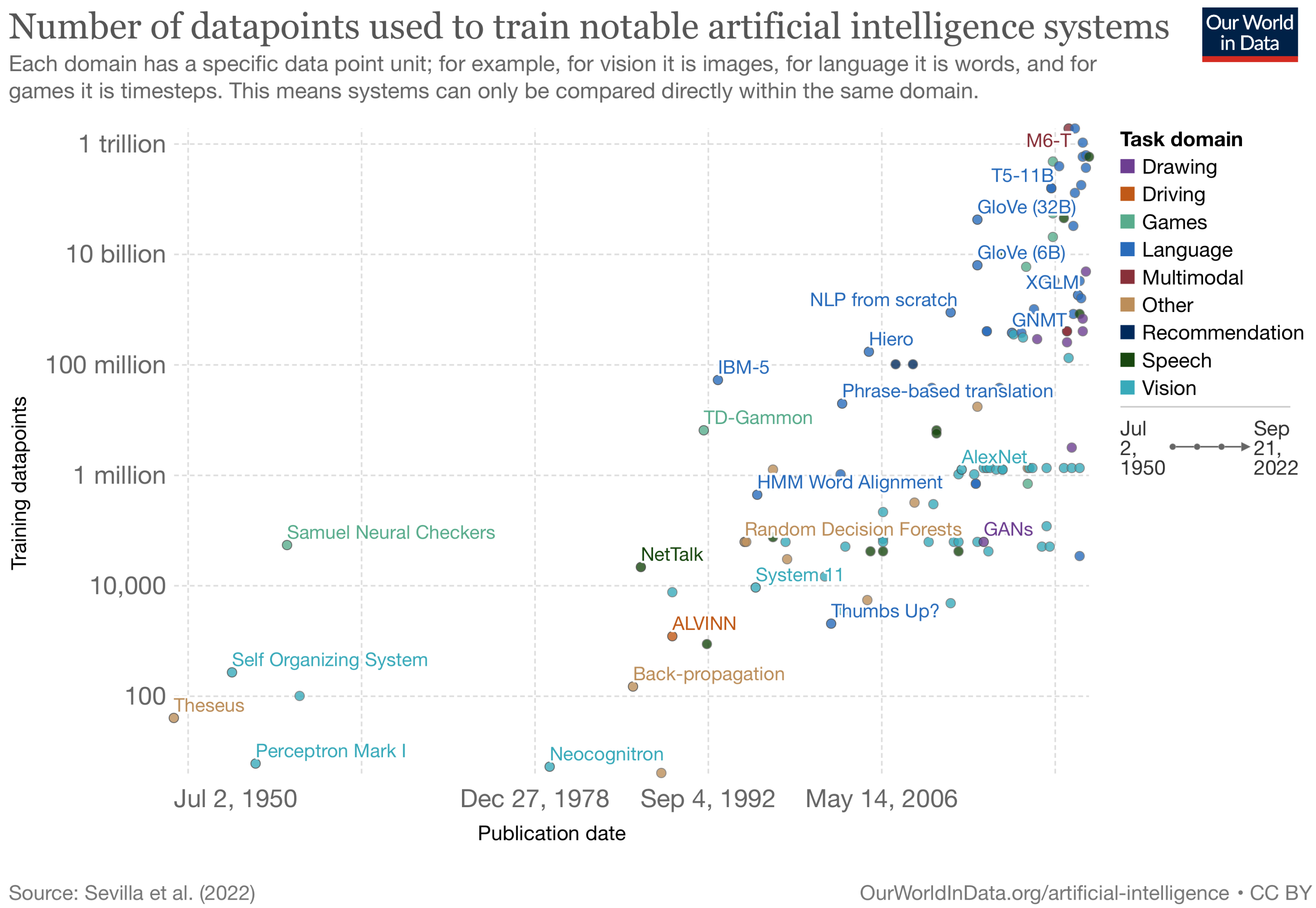


Scale of AI NN models is growing exponentially

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

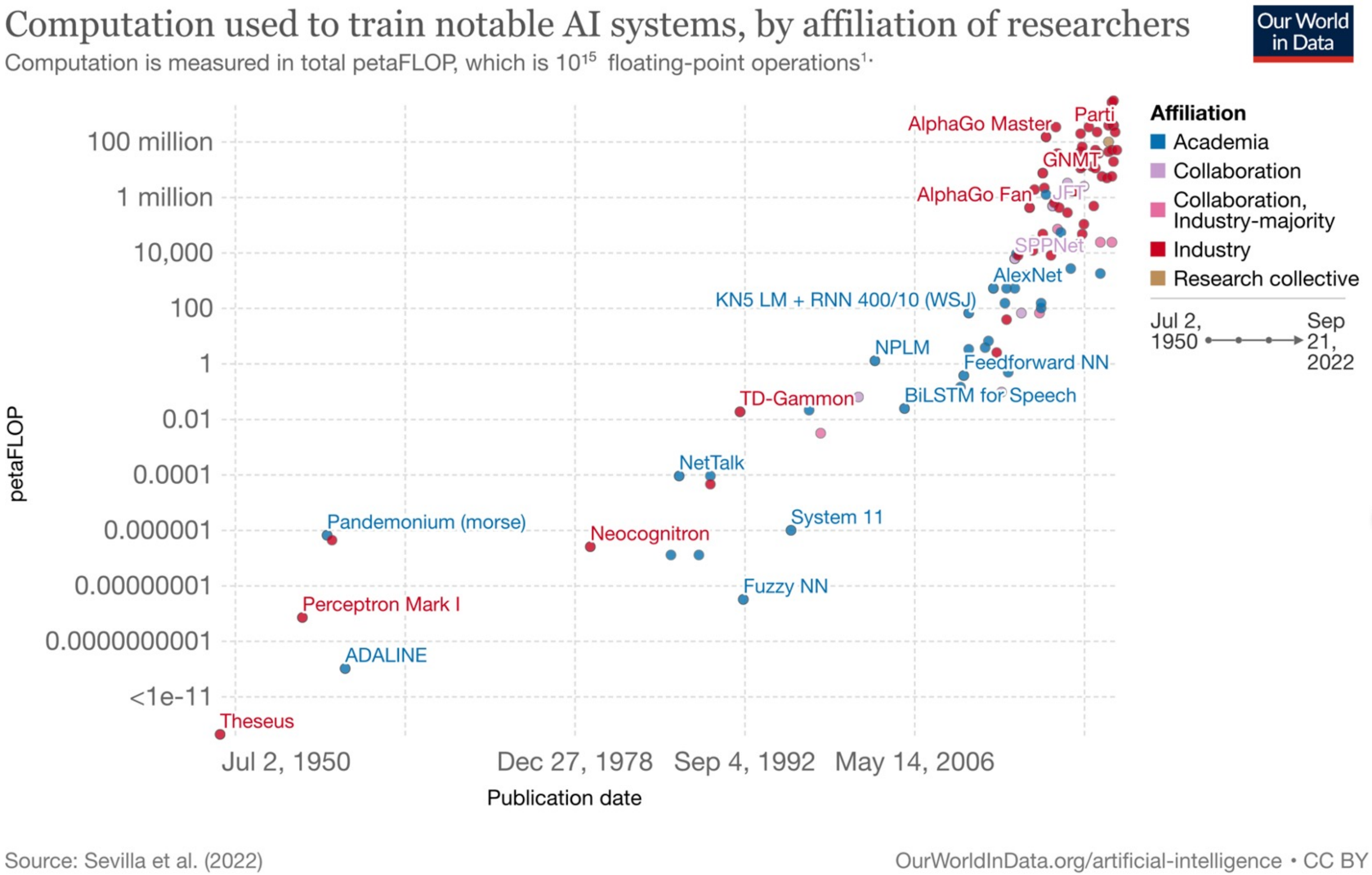
From https://www.nature.com/articles/d41586-023-00777-9
Based on data from Sevilla et al. arxiv 2202.05924

Drive to Bigger AI Models



Large AI NN models Require massive amounts of data for training

Drive to Bigger AI Models



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.

Outline

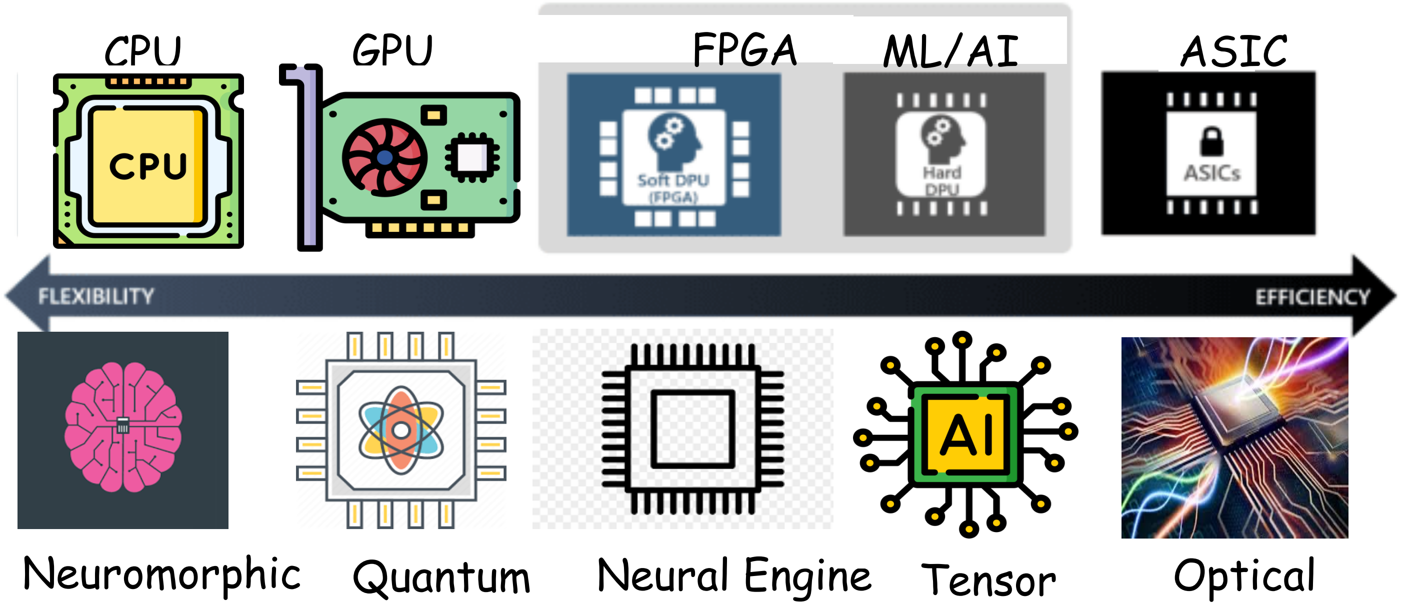
We will cover the following topics

- Brief Intro on current ML trends
- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
    - GPU accelerated libraries
    - 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?

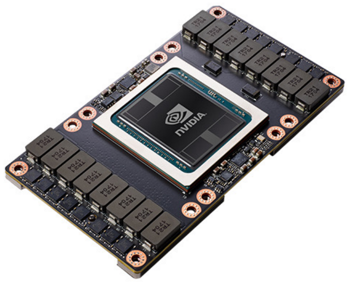
Graphics processing unit

- "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

Tseng Labs ET4000/W32p 1991

Voodoo3 2000 AGP card 1999

NVIDIA GeForce GTX 280 2008



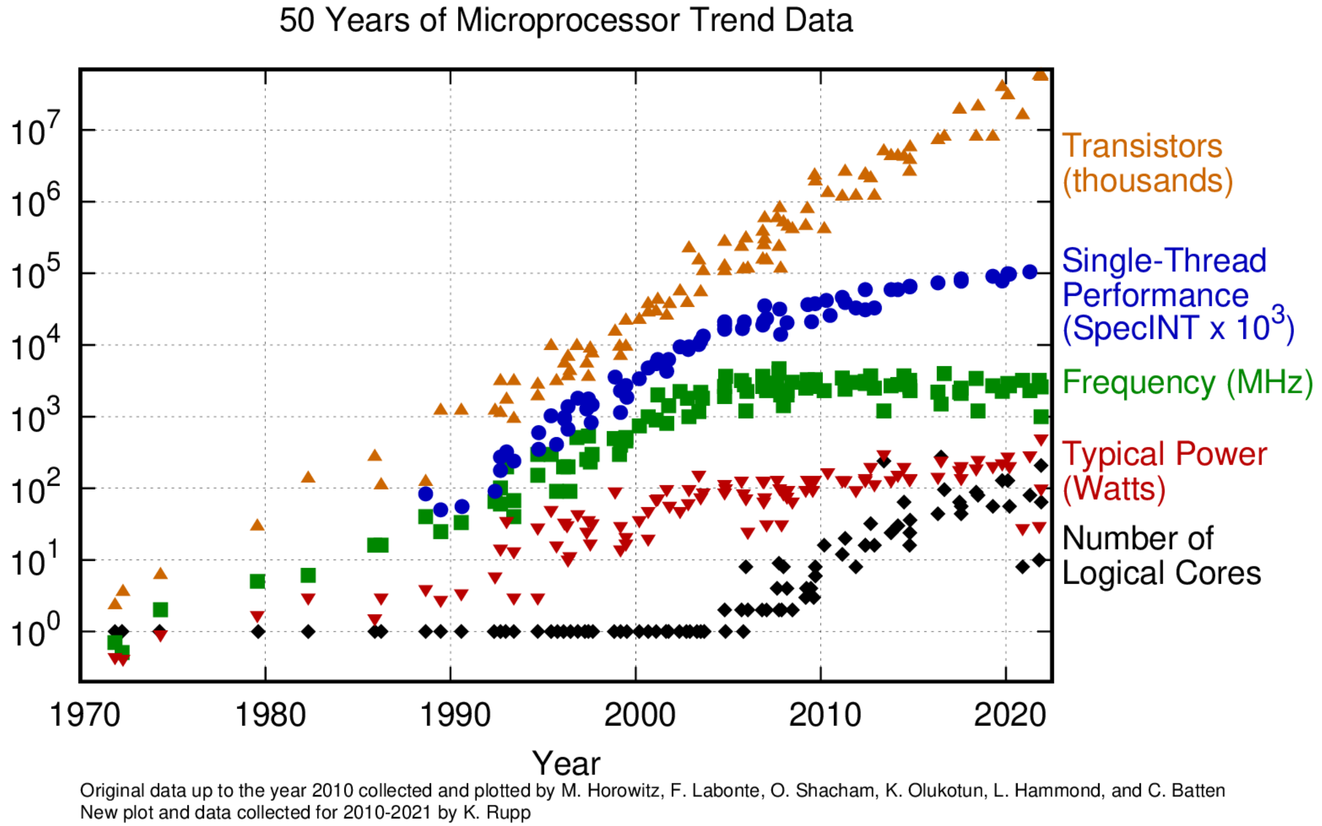
Why is there such an interest in GPUs?

Moore's law

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

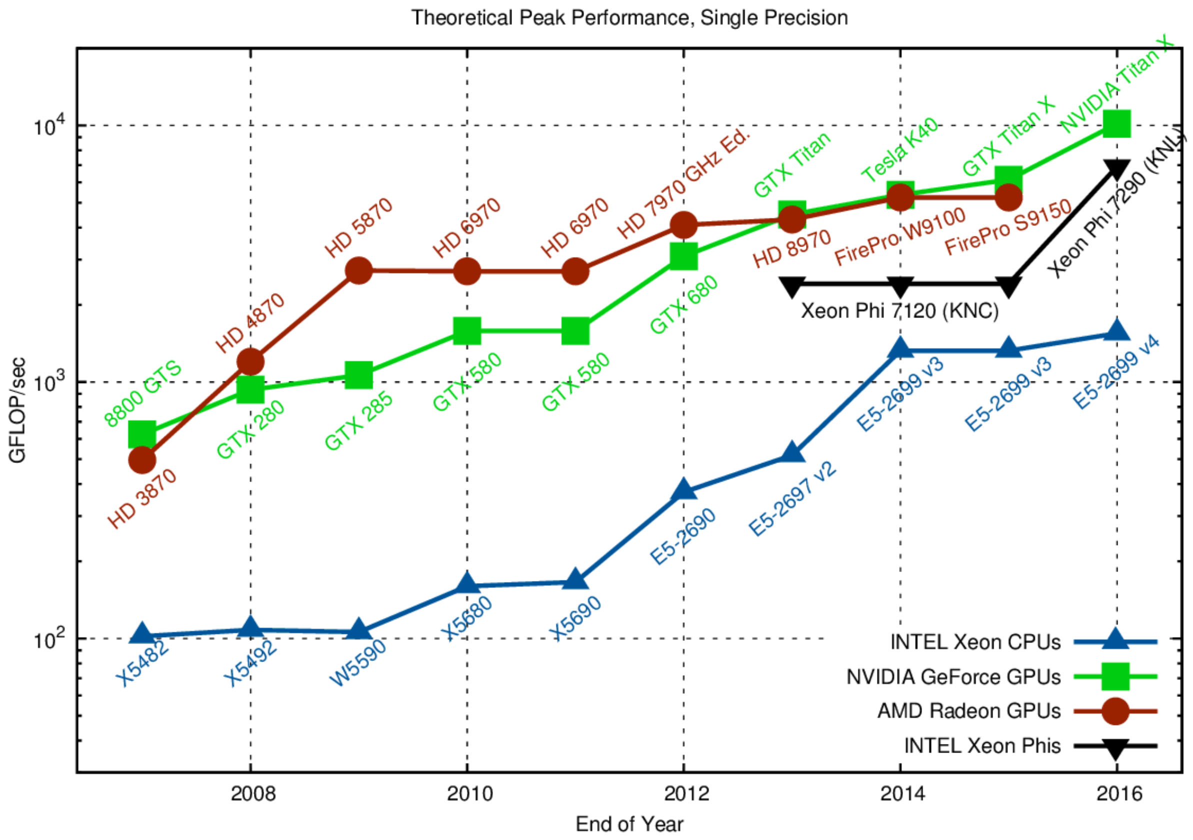
Trends since mid 2000s

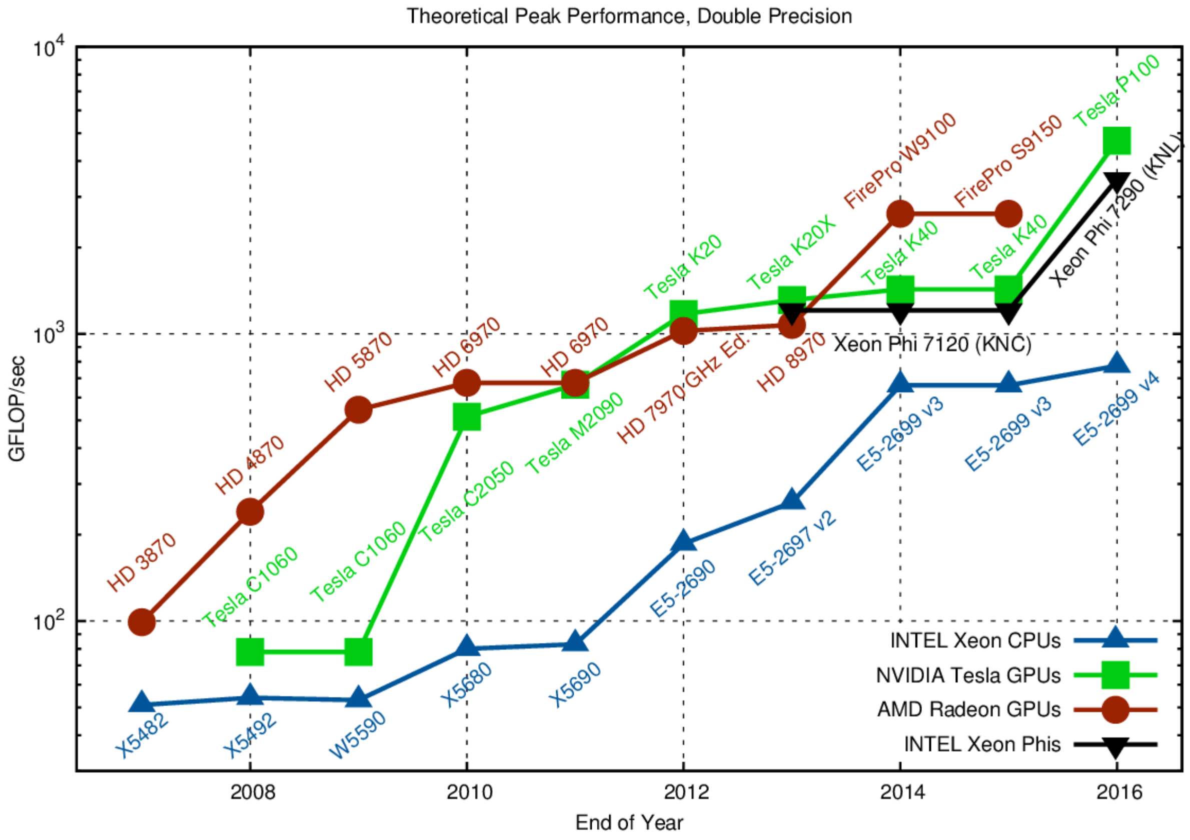
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.



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

Why is there such an interest in GPUs?

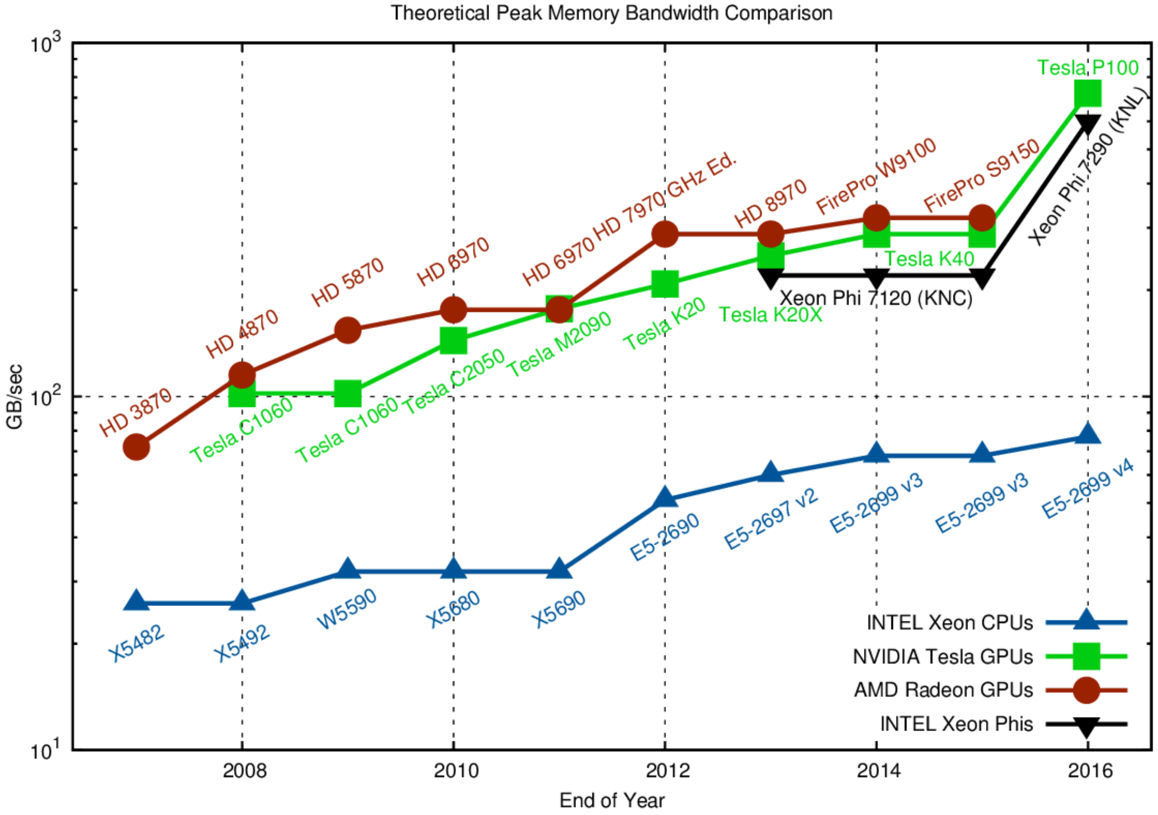


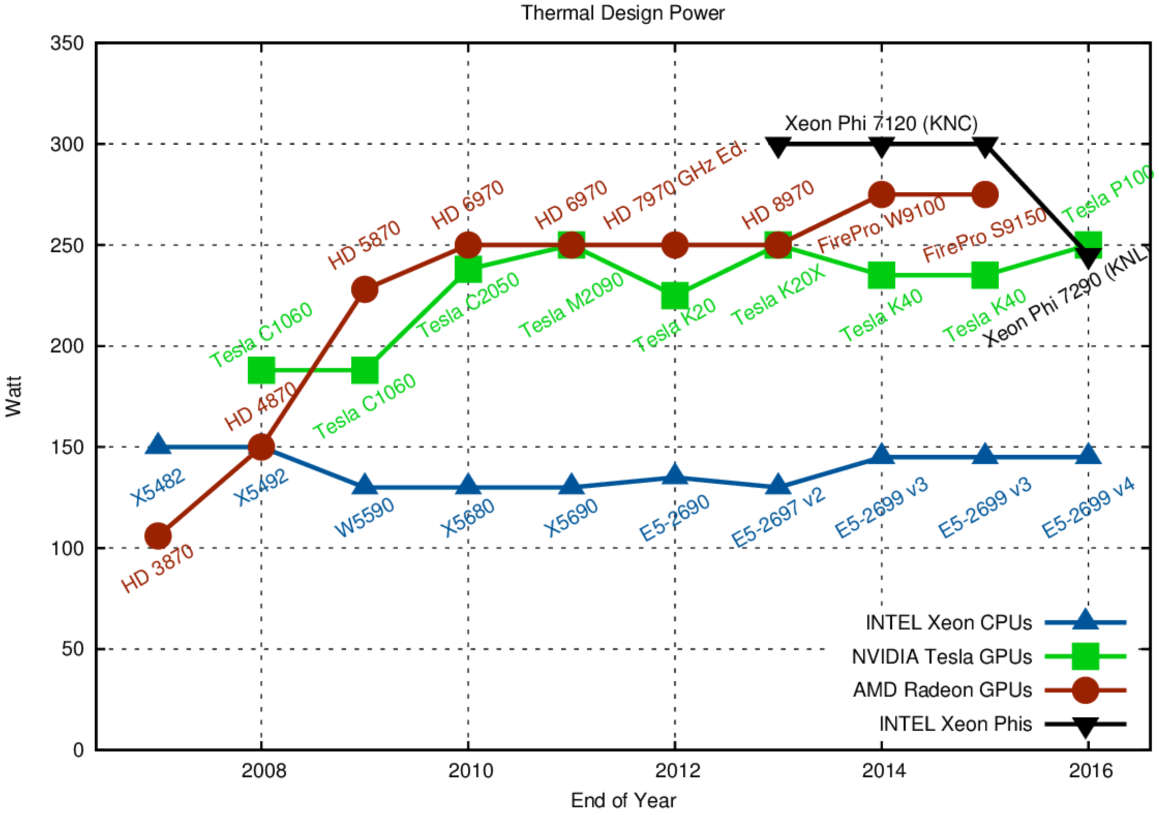


- GPUs offer significantly higher 32-bit floating point performance than CPUs.
- Datacenter GPUs also offer significantly higher 64-bit floating point performance than CPUs.

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

Why is there such an interest in GPUs?

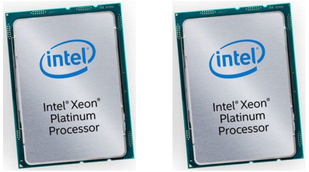




- GPUs have significantly higher memory bandwidth than CPUs.
- Given power consumption, a fair comparison would be a single GPU to 1- to 2-socket CPU server.

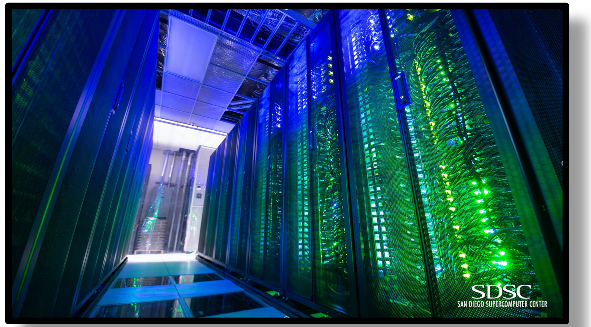
Figures source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

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



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

A supercomputer in a desktop?



ASCI White (LLNL)

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

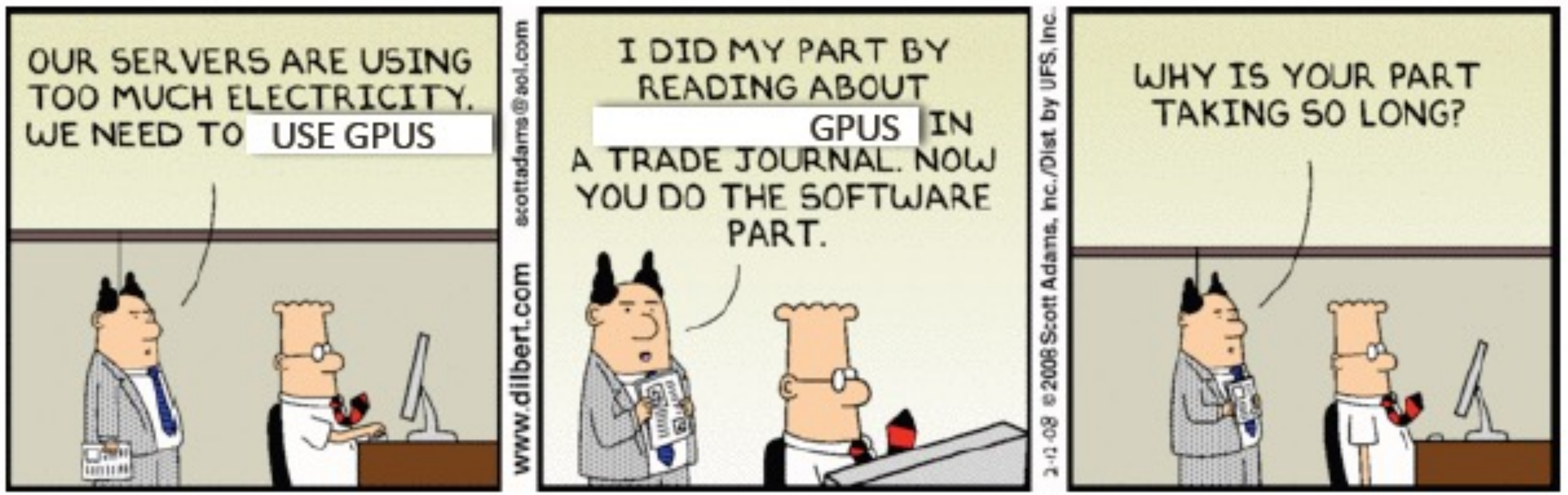
SDSC Expanse

- 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

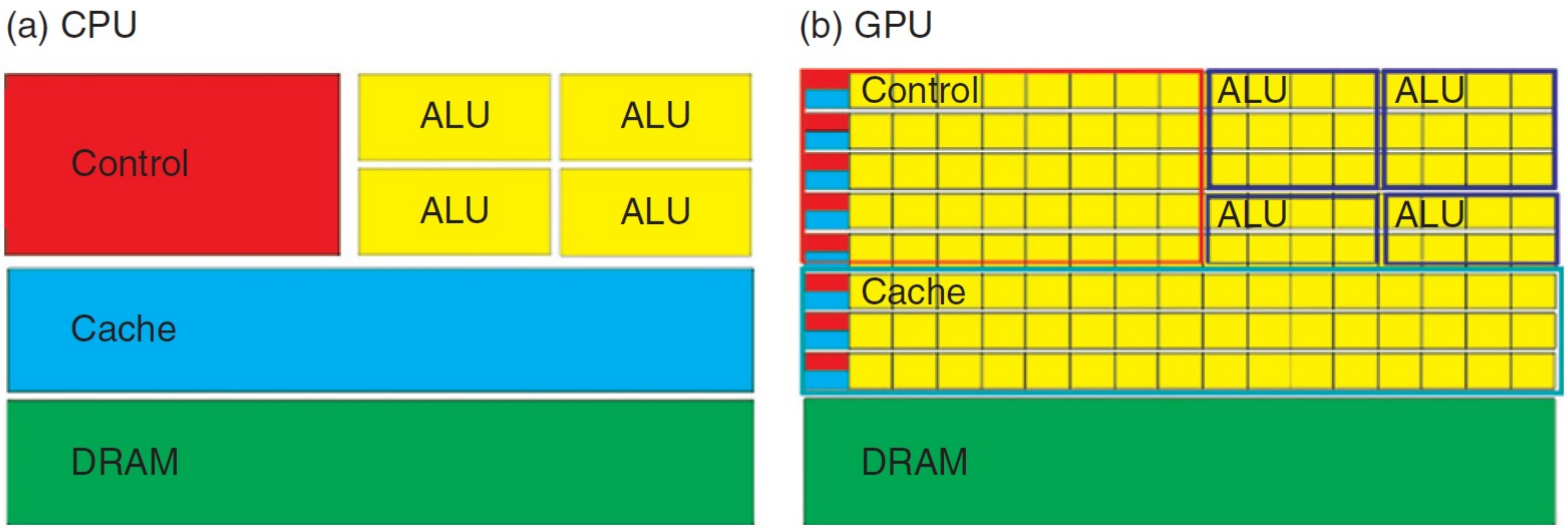
DIY 4 x Nvidia RTX 3080 box (2020) (Ampere arch)

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

What’s the catch?



GPU vs CPU architecture



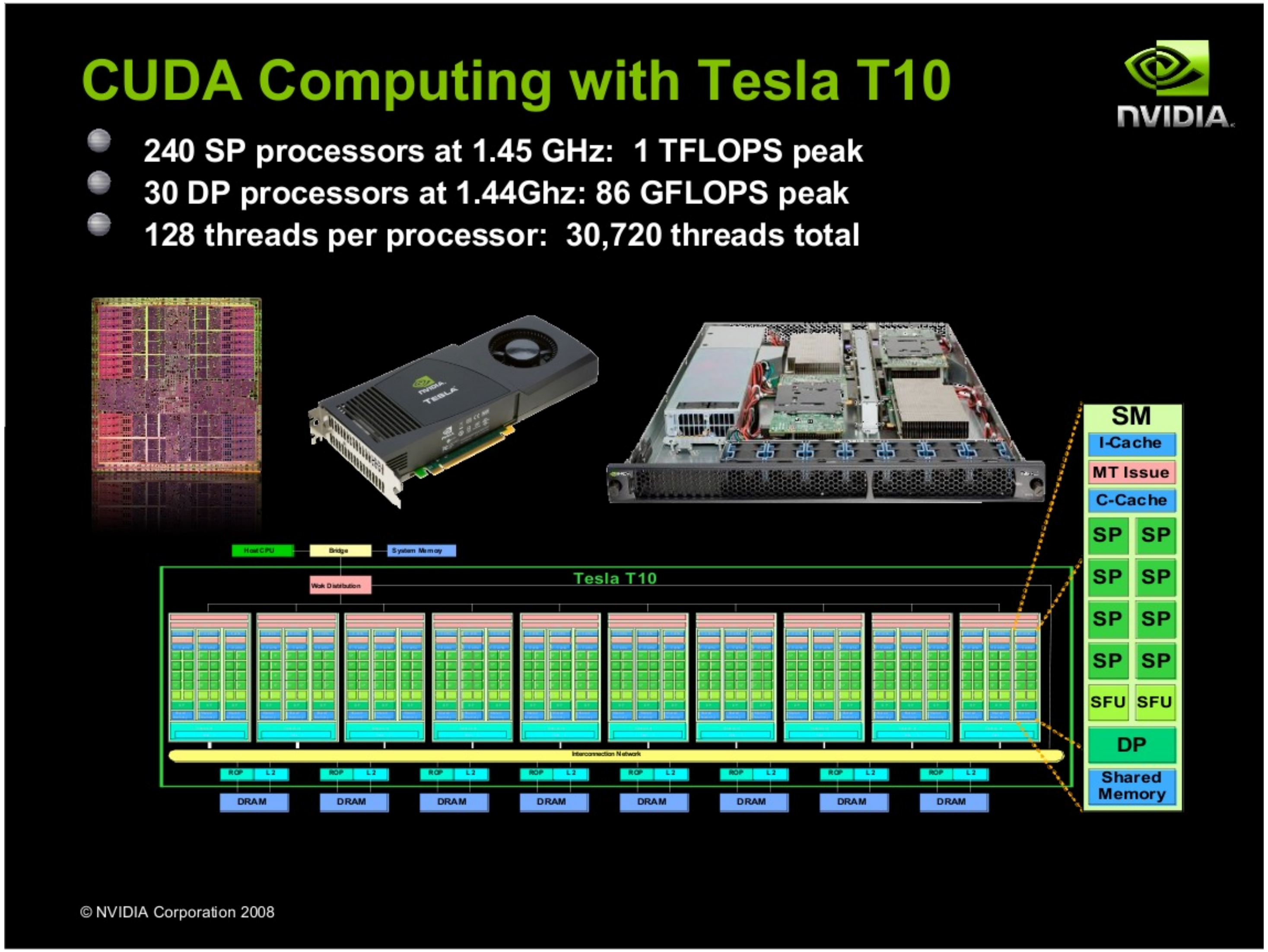
CPU

- 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



Nvidia GPU architecture in 2009

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

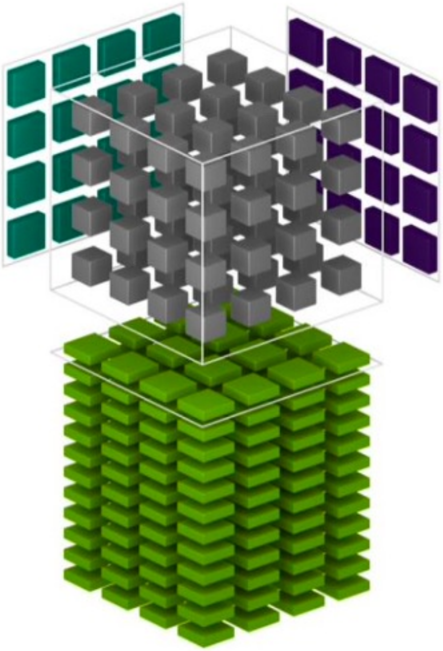
Multiprocessor

- 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 Tensore 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

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

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 harware

Outline

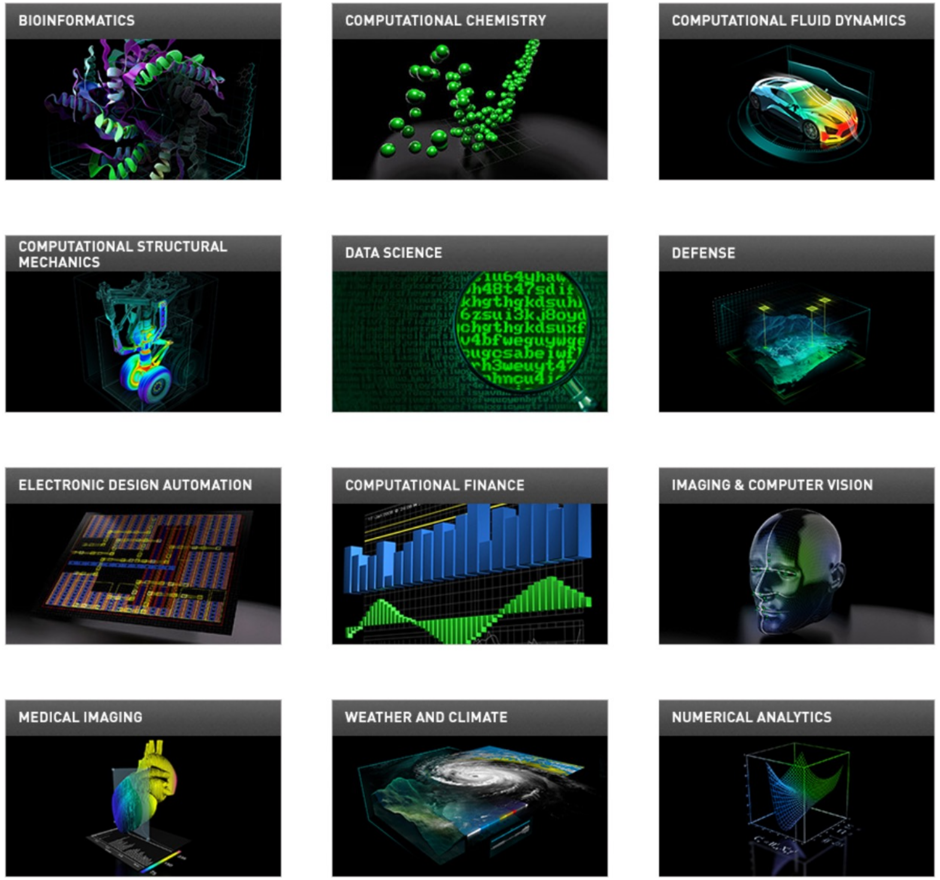
We will cover the following topics

- 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

GPU accelerated software

Examples from virtually any field

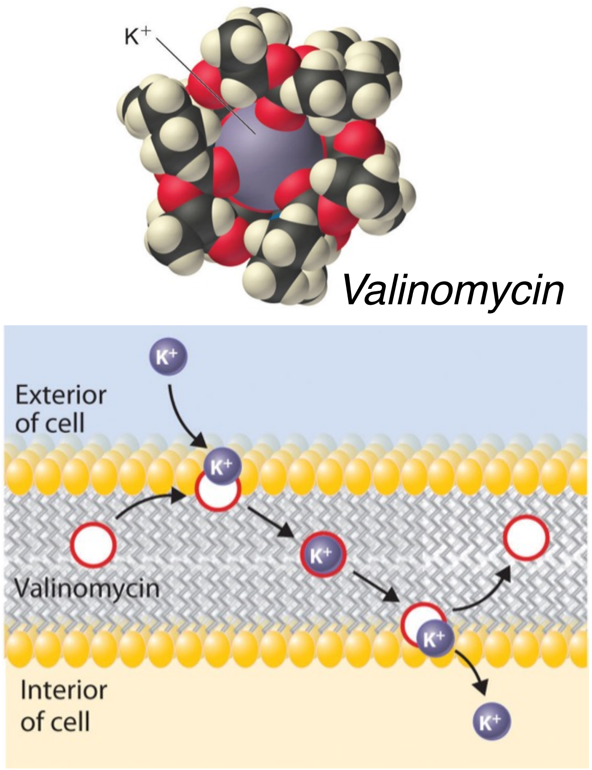
- Exhautive list on https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/
- Chemistry
- Life sciences
- Bioinformatics
- Astrophysics
- Finance
- Medical imaging
- Natural language processing
- Social sciences
- Weather and climate
- Computational fluid dynamics
- **Machine learning**,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

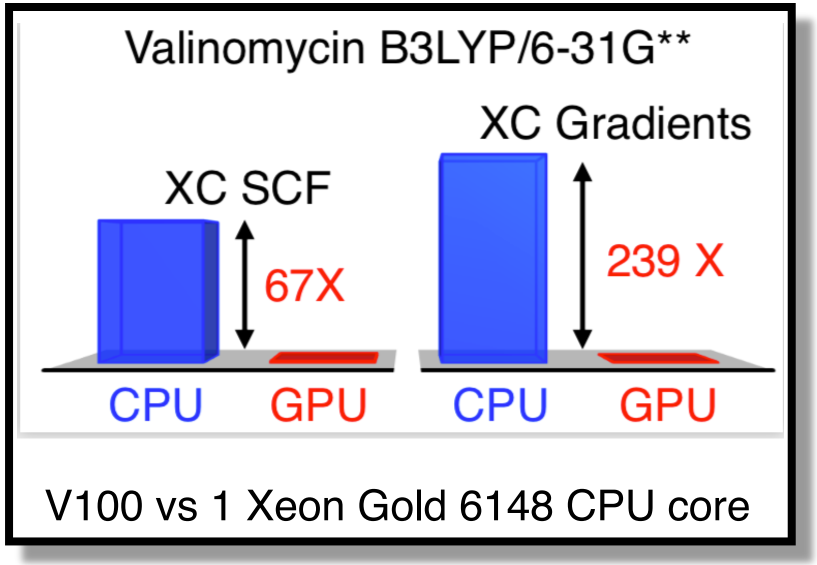


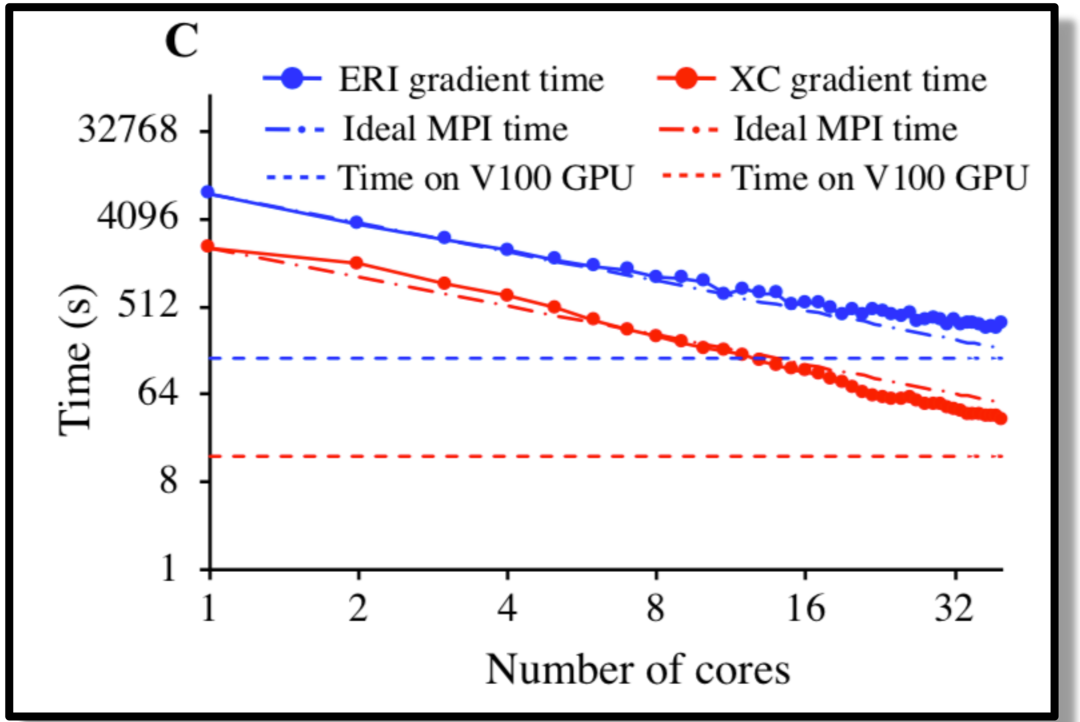
Quantum Chemistry

Example: Open source code QUICK

- 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 (3rd 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

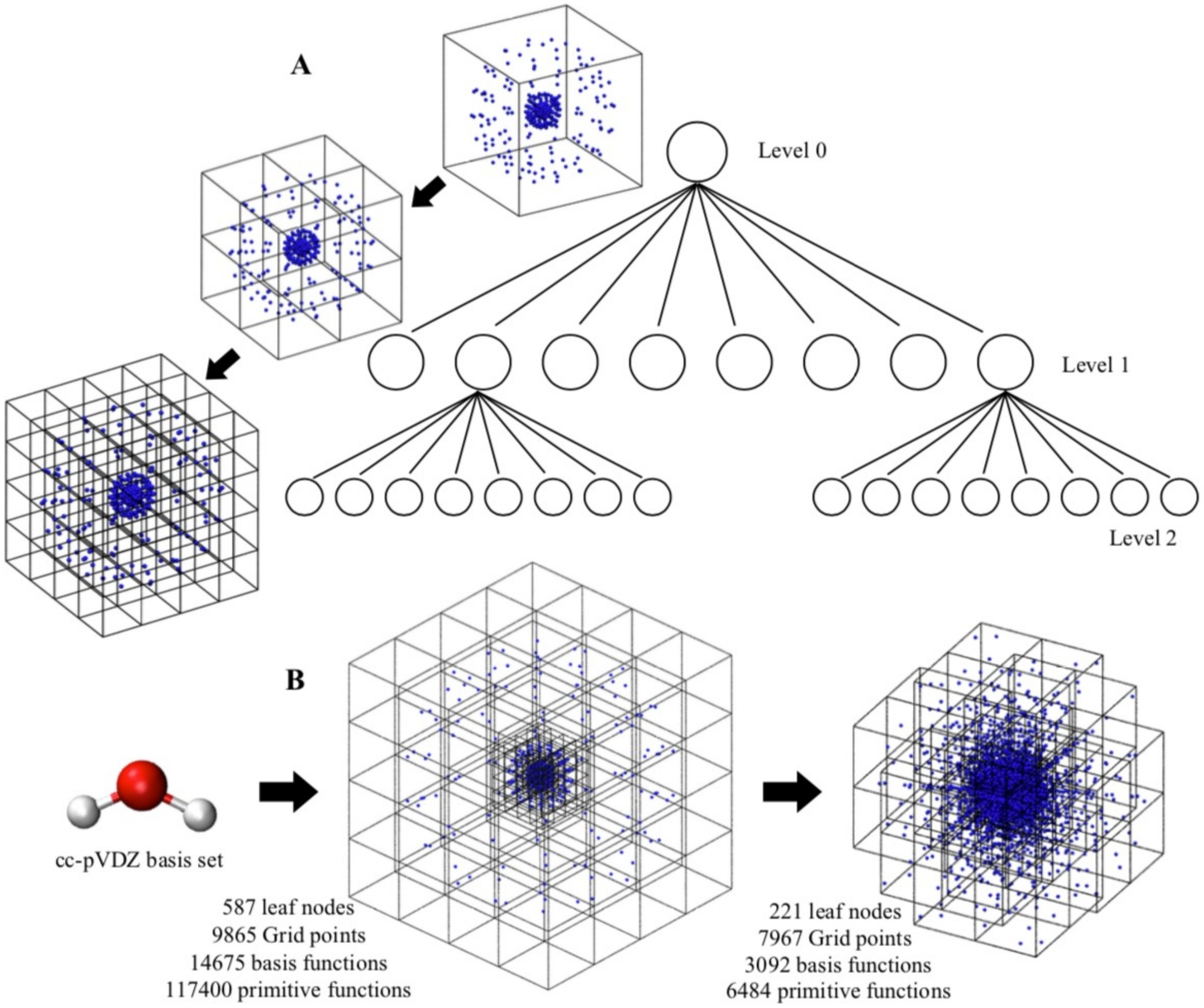
Relevant publications

- **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.

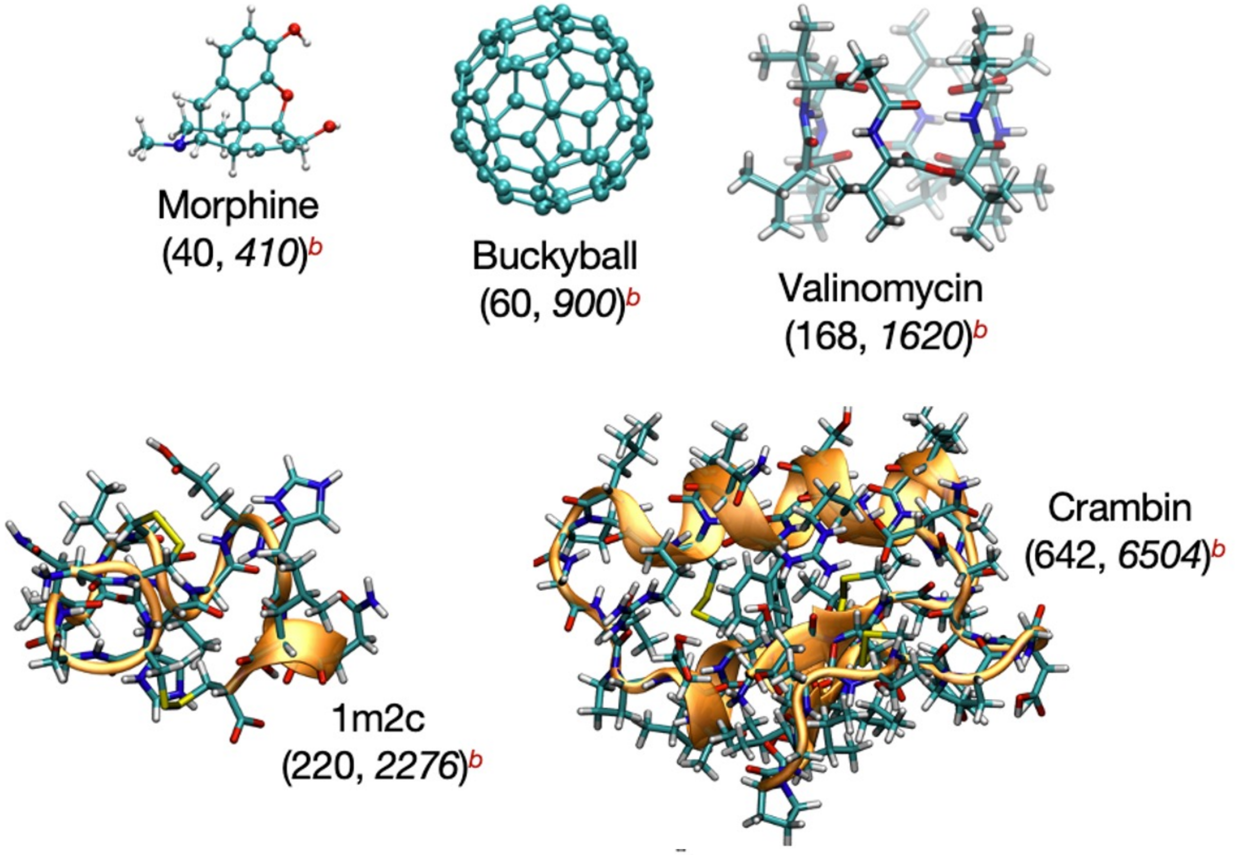
Quantum Chemistry

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

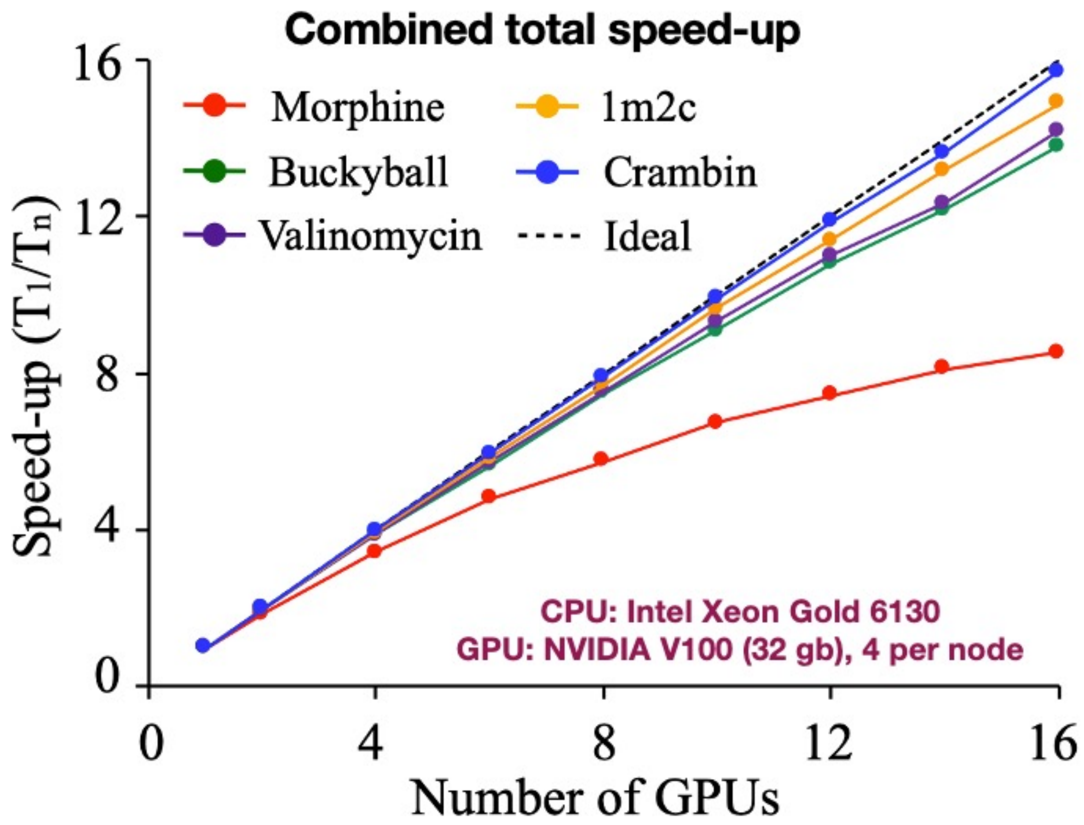


b. The number of atoms and number of basis functions are given in brackets.

Benchmarks: QUICK 20.06

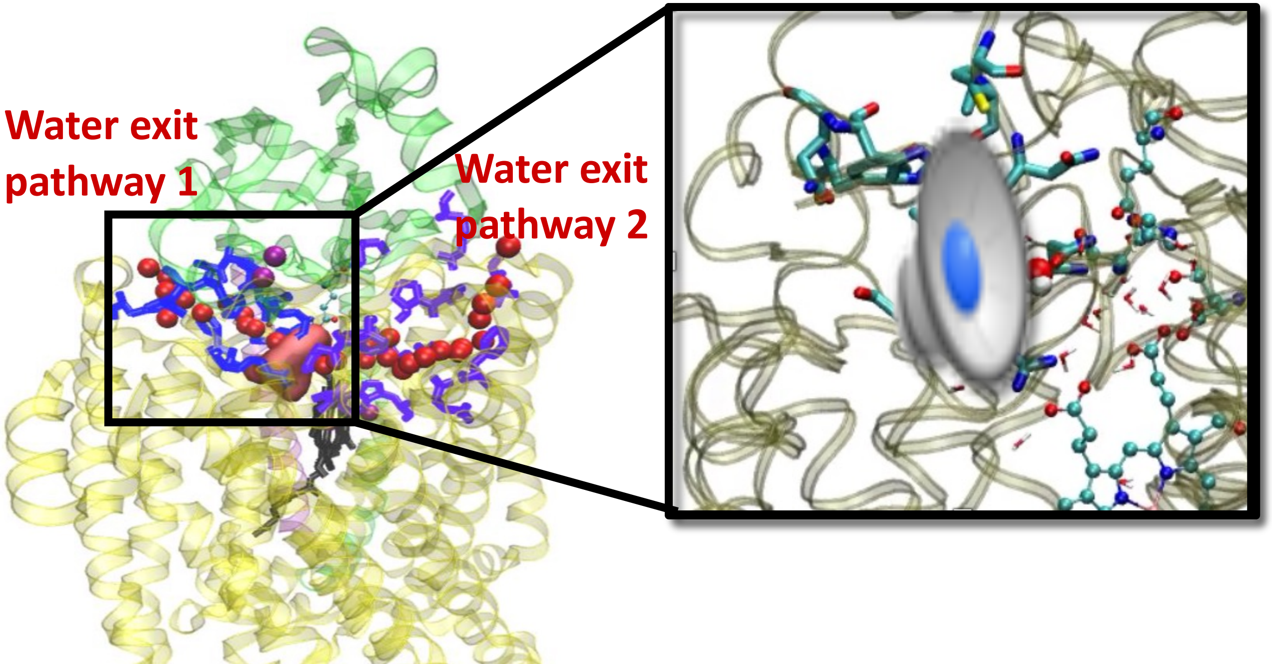
Manathunga *et al.*, *JCTC* **17** (2021) 3955.

B3LYP/6-31G** gradient calculations



Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems

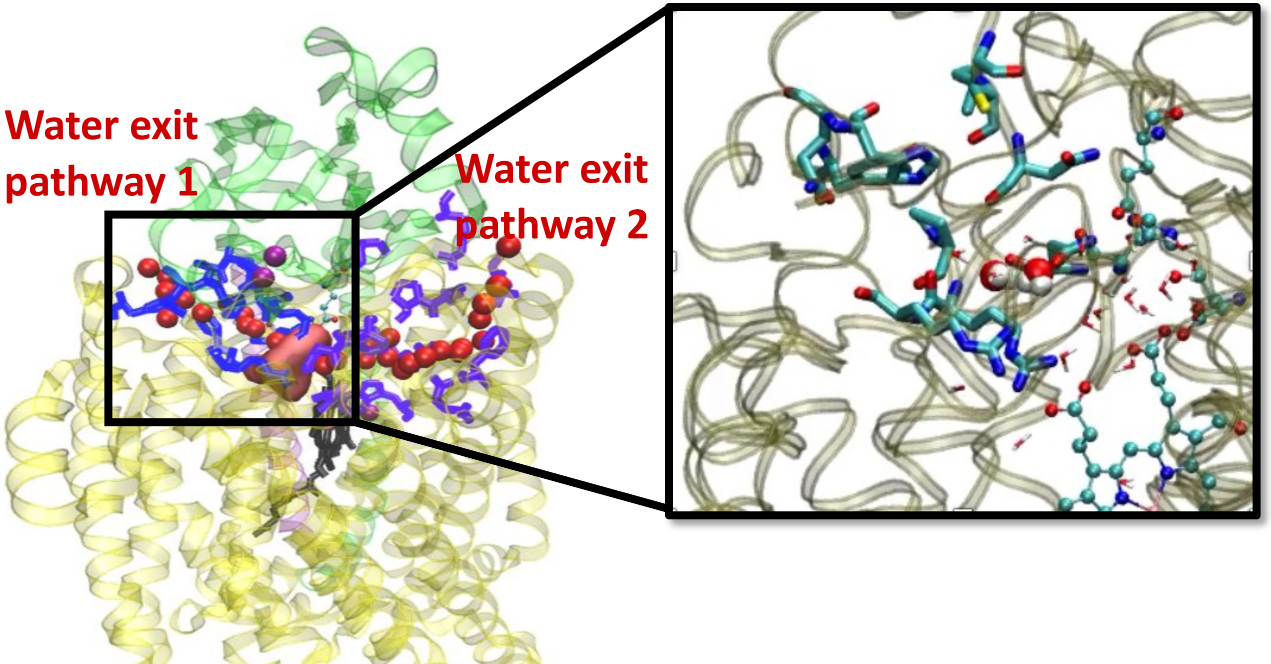


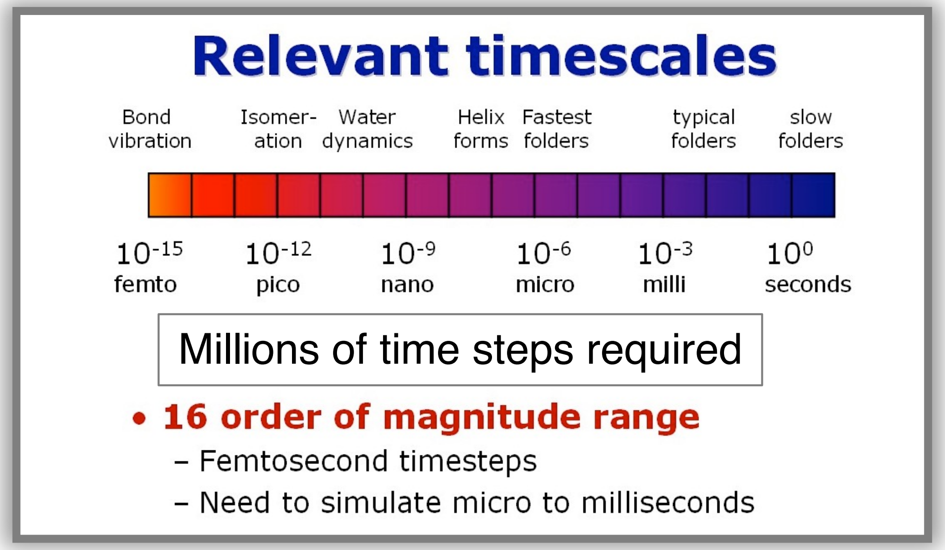
Cytochrome c oxidase enzyme

Yang, Skjevik, Han Du, Noodleman, Walker, Götz,
BBA Bioenergetics 2016 (1857) 1594.

Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems



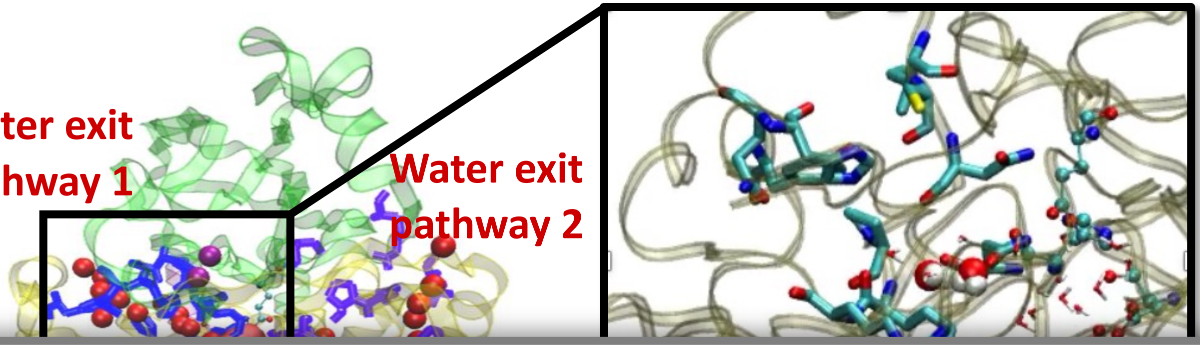


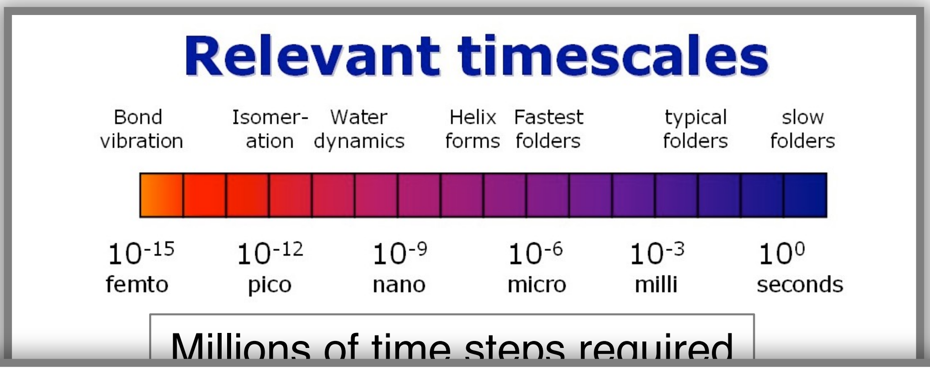
Cytochrome c oxidase enzyme

Yang, Skjevik, Han Du, Noodleman, Walker, Götz, *BBA Bioenergetics* 2016 (1857) 1594.

Molecular dynamics

Amber code: Atomistic simulations of condensed phase biomolecular systems



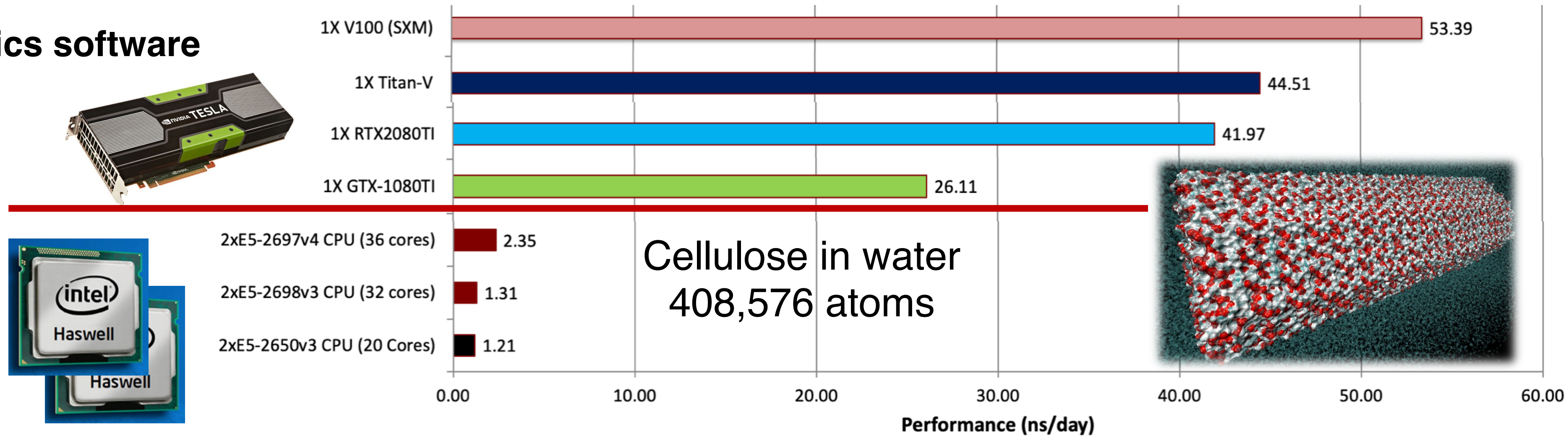


Amber 18 molecular dynamics software

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

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

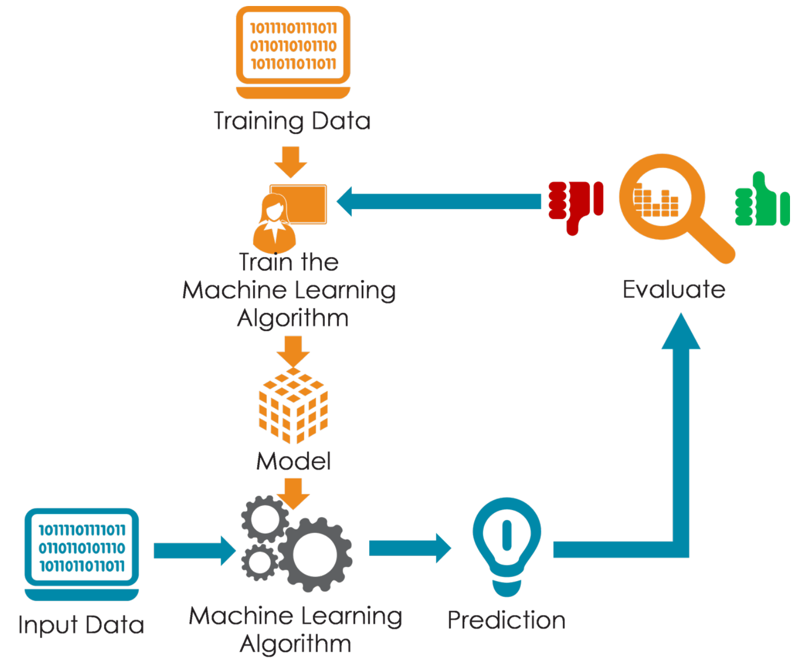
Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.



Machine learning and GPUs

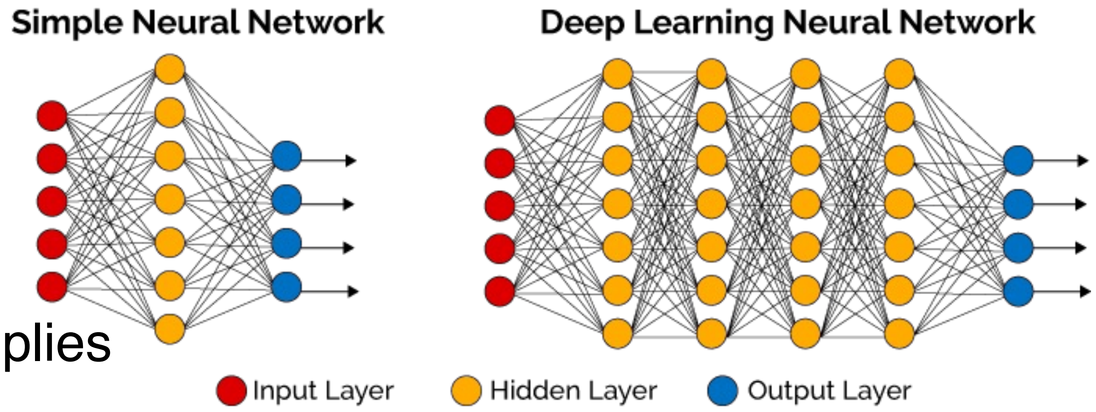
Machine learning

- 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)



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

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)

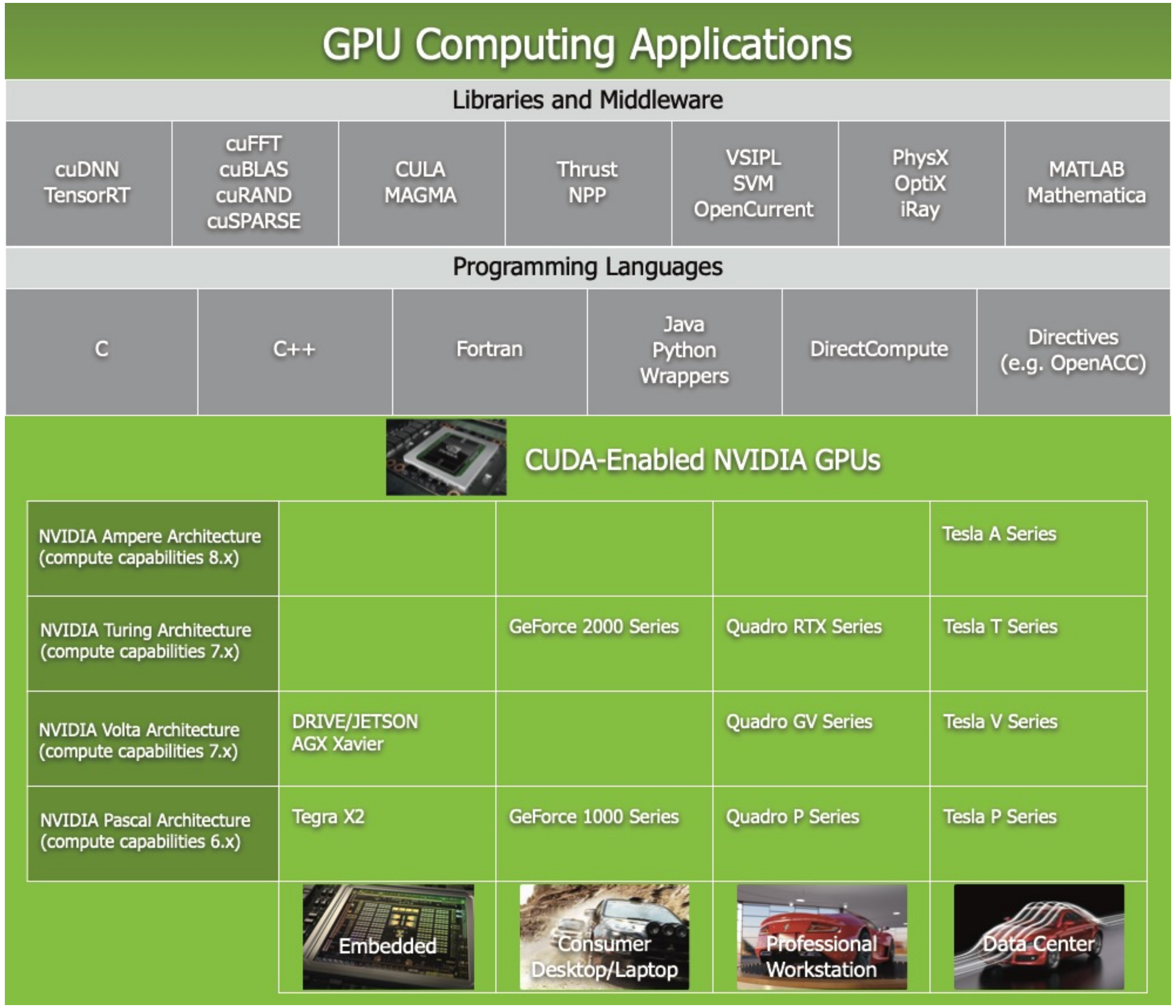
OpenACC

- 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

Nvidia GPU computing universe



Source: CUDA C programming guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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 (now available at https://github.com/NVIDIA/cuda-samples)
- https://developer.nvidia.com/cuda-zone
- https://nvidia.com/getcuda

Activate on Expanse GPU nodes

```
$> module purge
$> module reset
$> module load cuda11.7/toolkit
```

3 ways to use GPUs

Applications

Libraries	OpenACC Directives	Programming Languages
“Drop-in” Acceleration	Easily Accelerate Applications	Maximum Flexibility

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

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

Using GPU accelerated libraries

GPU accelerated libraries

Ease of use

- GPU acceleration without in-depth knowledge of GPU programming

“Drop-in”

- Many GPU accelerated libraries follow standard APIs
- Minimal code changes required

Quality

- High-quality implementations of functions encountered in a broad range of applications

Performance

- Libraries are tuned by experts

=> Use if you can – (do not write your own matrix multiplication)

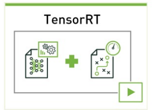
GPU accelerated libraries

See https://developer.nvidia.com/gpu-accelerated-libraries

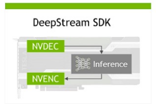
Deep Learning Libraries

cuDNN

GPU-accelerated library of primitives for deep neural networks

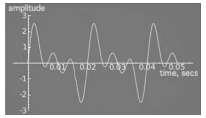


GPU-accelerated neural network inference library for building deep learning applications



Advanced GPU-accelerated video inference library

Signal, Image and Video Libraries



cuFFT

GPU-accelerated library for Fast Fourier Transforms

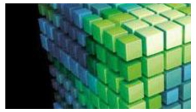
NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

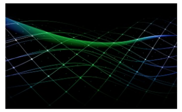
High-performance APIs and tools for hardware accelerated video encode and decode

Linear Algebra and Math Libraries



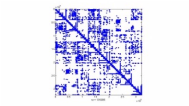
cuBLAS

GPU-accelerated standard BLAS library



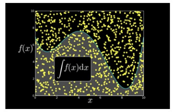
CUDA Math Library

GPU-accelerated standard mathematical function library



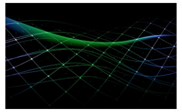
cuSPARSE

GPU-accelerated BLAS for sparse matrices



cuRAND

GPU-accelerated random number generation (RNG)



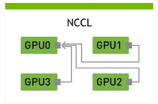
cuSOLVER

Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

AmgX

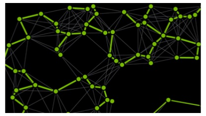
GPU accelerated linear solvers for simulations and implicit unstructured methods

Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics

Thrust

GPU-accelerated library of parallel algorithms and data structures

Partner Libraries

OpenCV

FFmpeg

ArrayFire

… and several others

GPU accelerated libraries

3 steps to using libraries

- Step 1: Substitute library calls with equivalent CUDA library calls

  `saxpy ( … )` → `cublasSaxpy ( … )`

- Step 2: Manage data locality

  - with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
  - with CUBLAS: `cublasSetVector(), cublasGetVector() etc.`

- Step 3: Rebuild and link the CUDA-accelerated library

```
nvcc myobj.o –l cublas
```

CUBLAS library example

```
int N = 1 << 20;
```

saxpy =
single precision
a time **x** **p**lus **y**

$$y = a * x + y$$

```
// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);
```

CUBLAS library example

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix and use device variables

CUBLAS library example

```
int N = 1 << 20;
cublasCreate(&handle);
```

Initialize CUBLAS

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

```
cublasDestroy(handle);
```

Shut down CUBLAS

CUBLAS library example

```
int N = 1 << 20;
cublasCreate(&handle);
cudaMalloc((void**)&d_x, N*sizeof(float));
cudaMalloc((void**)&d_y, N*sizeof(float));
```

Allocate device vectors

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

```
cudaFree(d_x);
cudaFree(d_y);
cublasDestroy(handle);
```

Deallocate device vectors

CUBLAS library example

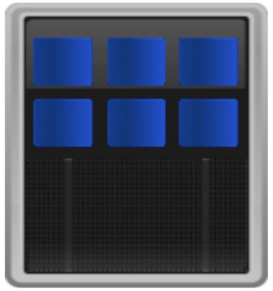
```
int N = 1 << 20;
cublasCreate(&handle);
cudaMalloc((void**)&d_x, N*sizeof(float));
cudaMalloc((void**)&d_y, N*sizeof(float));

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasDestroy(handle);
```

Transfer data to GPU

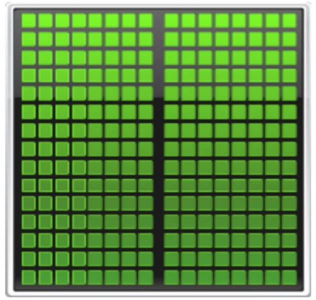
Read data back from GPU

NUMERICAL COMPUTING IN PYTHON

NumPy



CuPy



- 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.lingalg.qr(A)
```

AFTER

```
import cupy as np

size = 4096
A = np.random.randn(size,size)

Q, R = np.lingalg.qr(A)
```

NumPy

52x Speedup!

CuPy

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

Recommended Reading

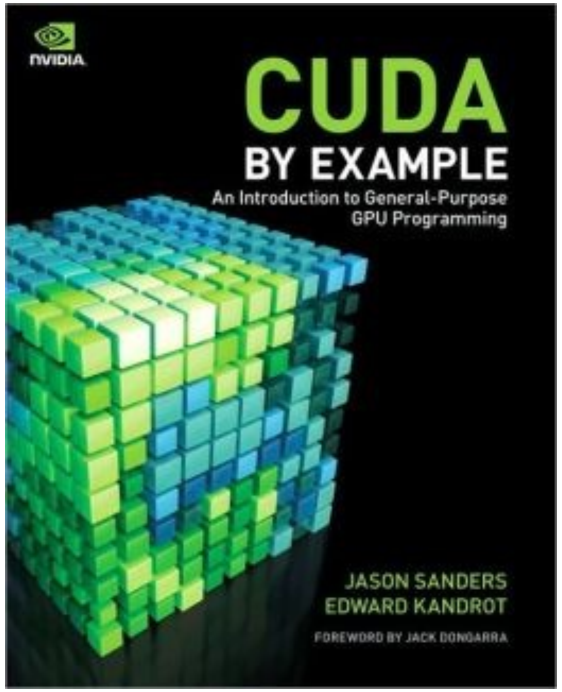
NVIDIA HPC SDK: https://docs.nvidia.com/hpc-sdk/index.html

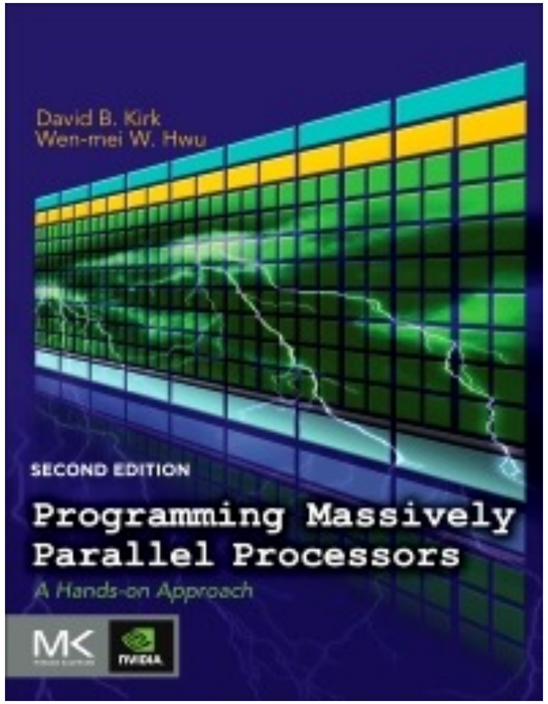
CUDA C: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

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

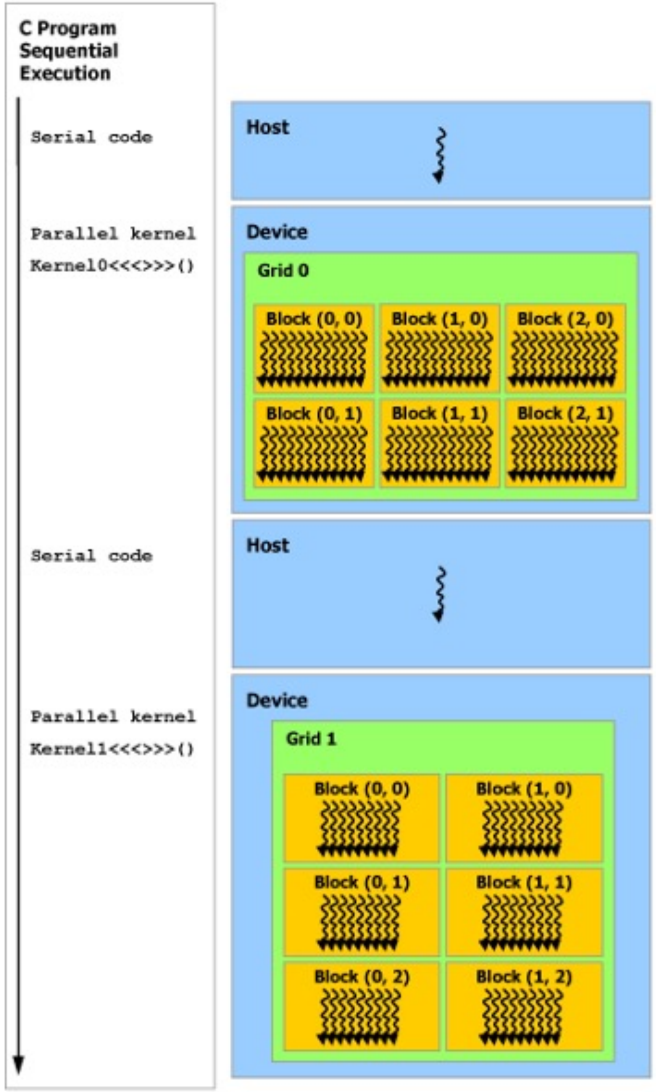
Many resources here: https://www.gpuhackathons.org/technical-resources

Good books to get started





Heterogeneous Computing

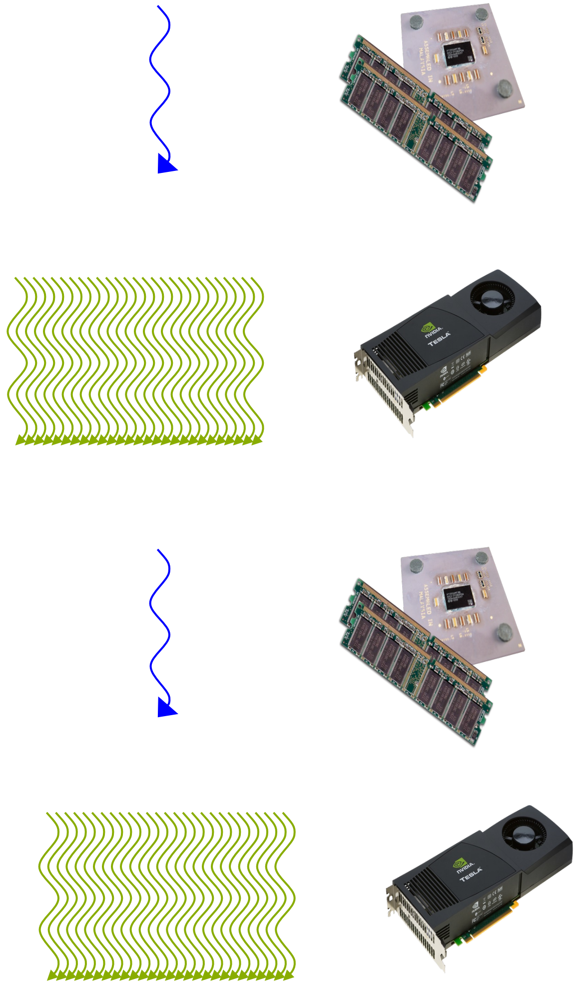


serial code

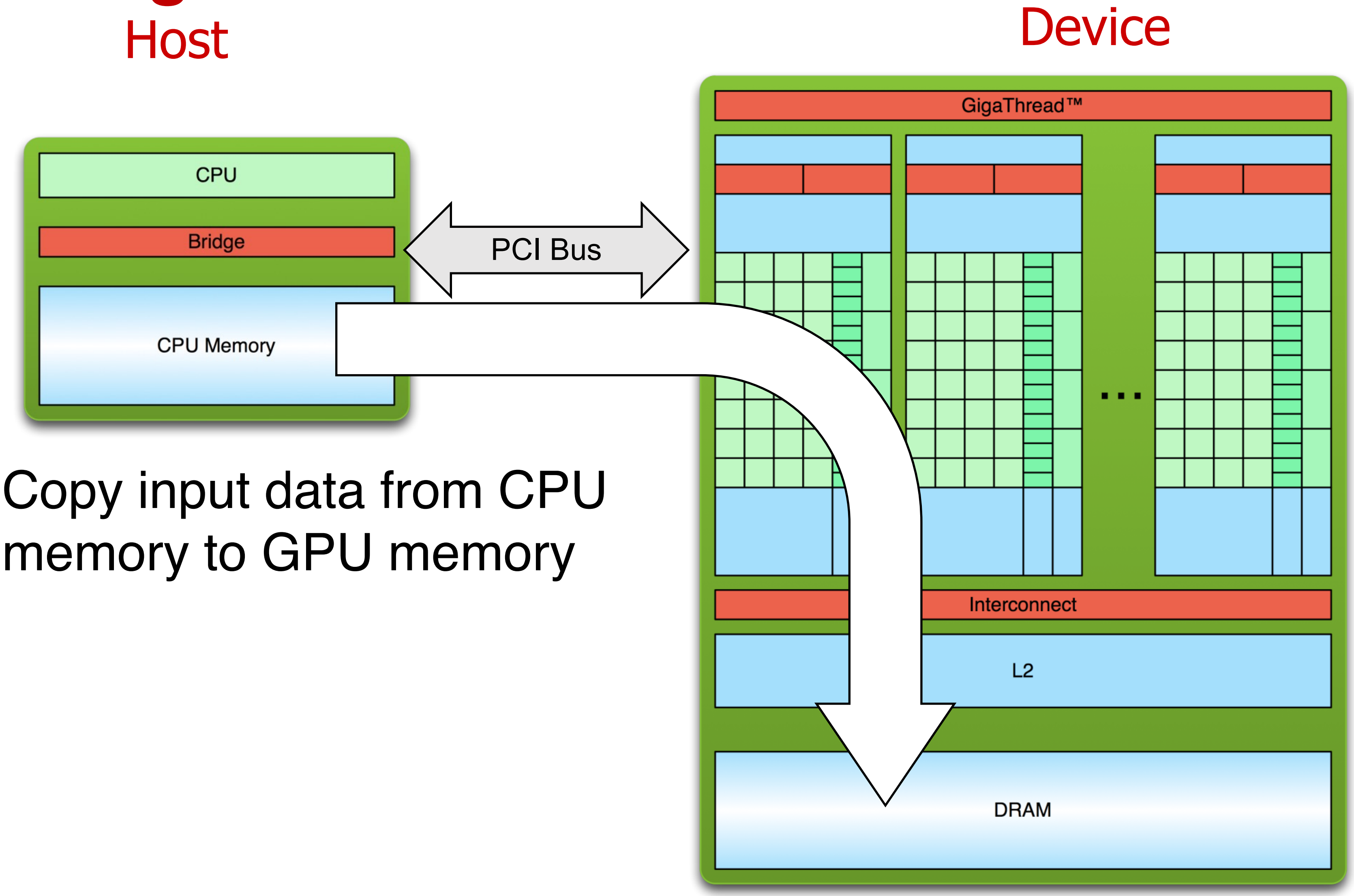
parallel code

serial code

parallel code

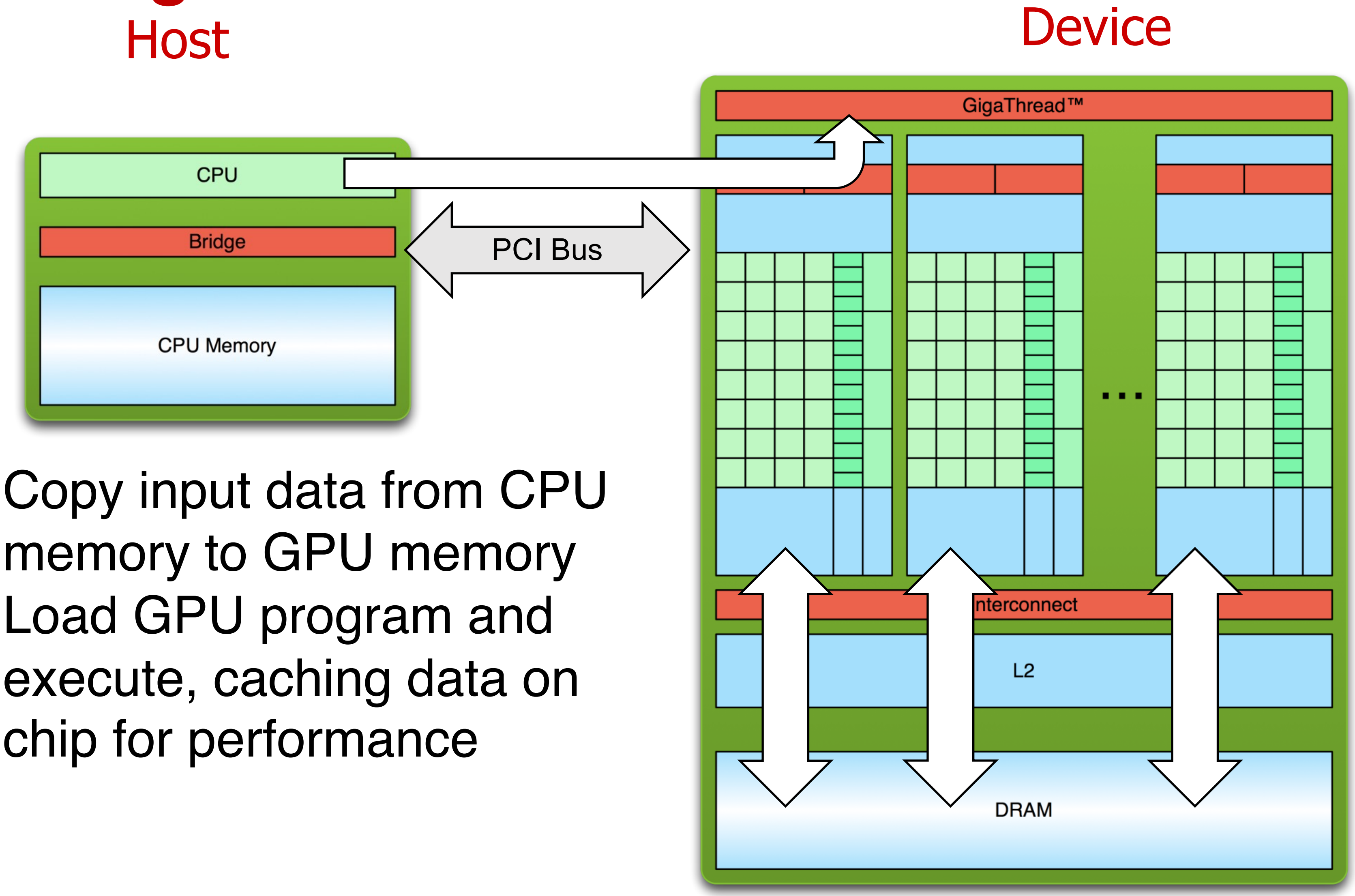


Processing Flow



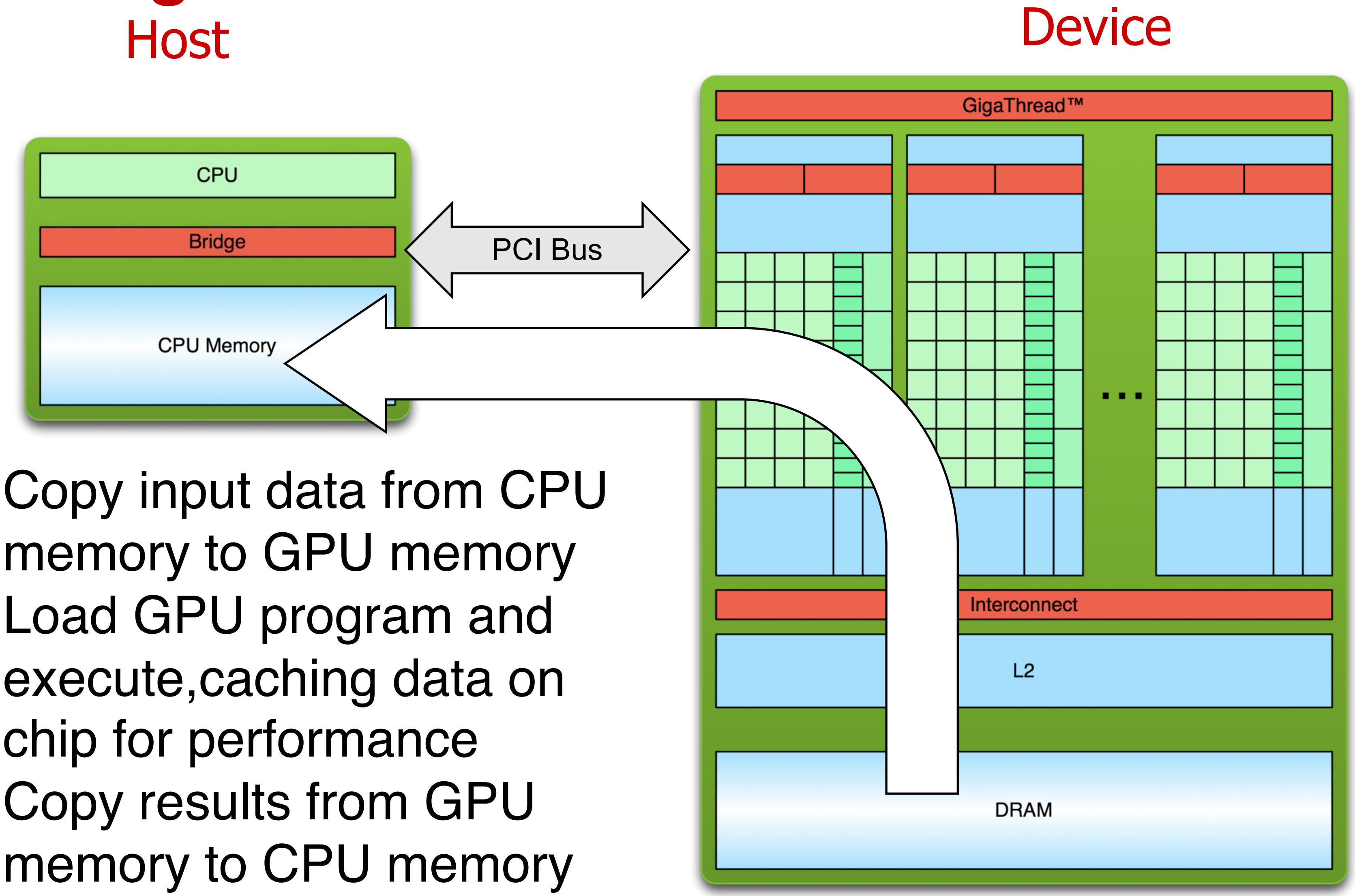
1. Copy input data from CPU memory to GPU memory

Processing Flow



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

Processing Flow



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)
- SIMT – 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

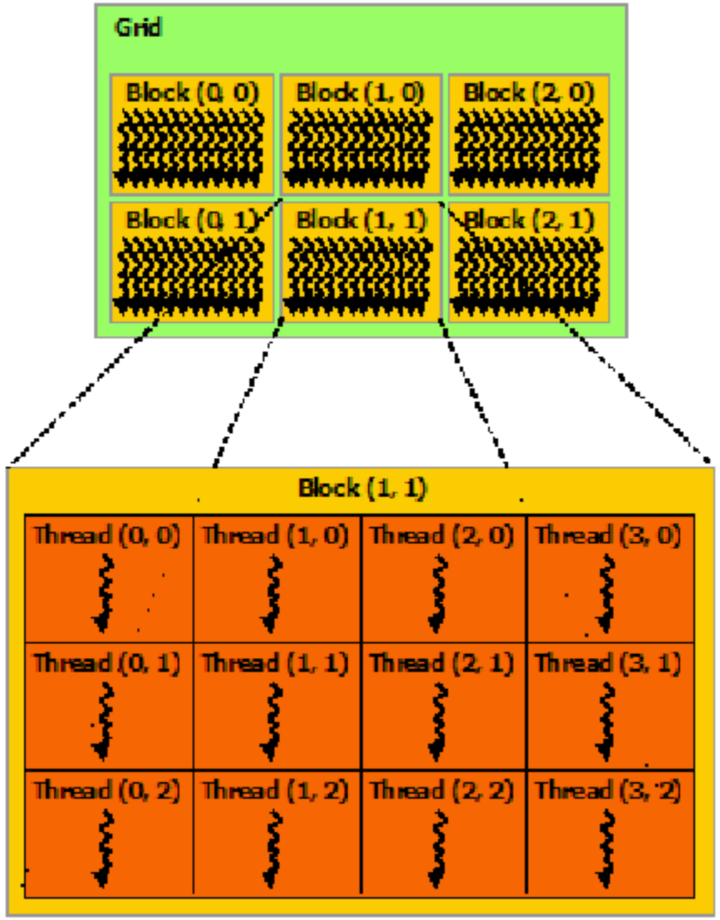
- Grids map to GPUs

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

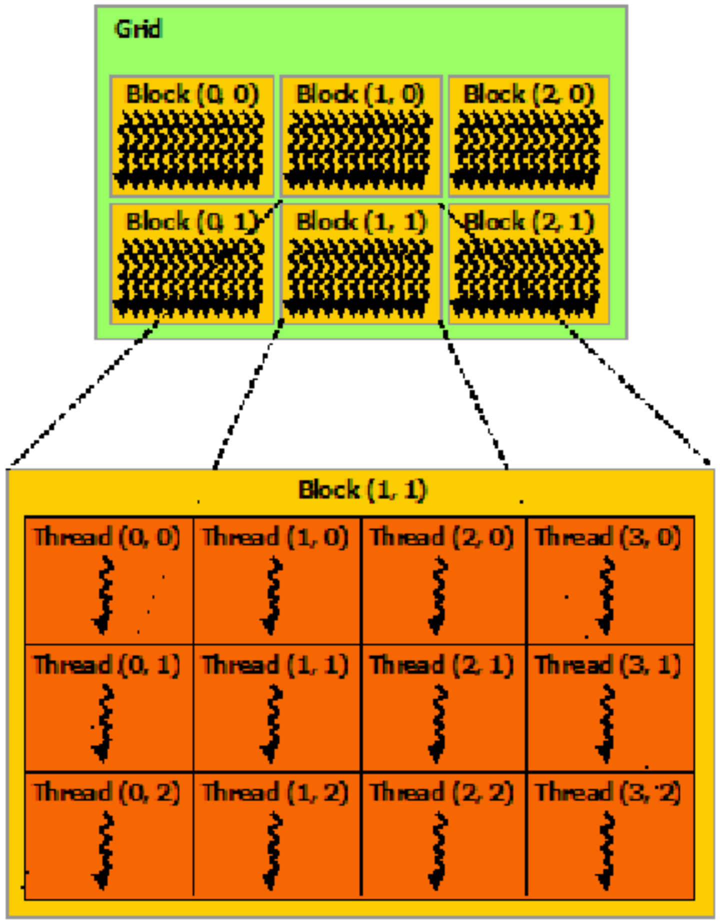
- Following variables allow to compute the ID of each individual thread that is executing in a grid block.

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);
```

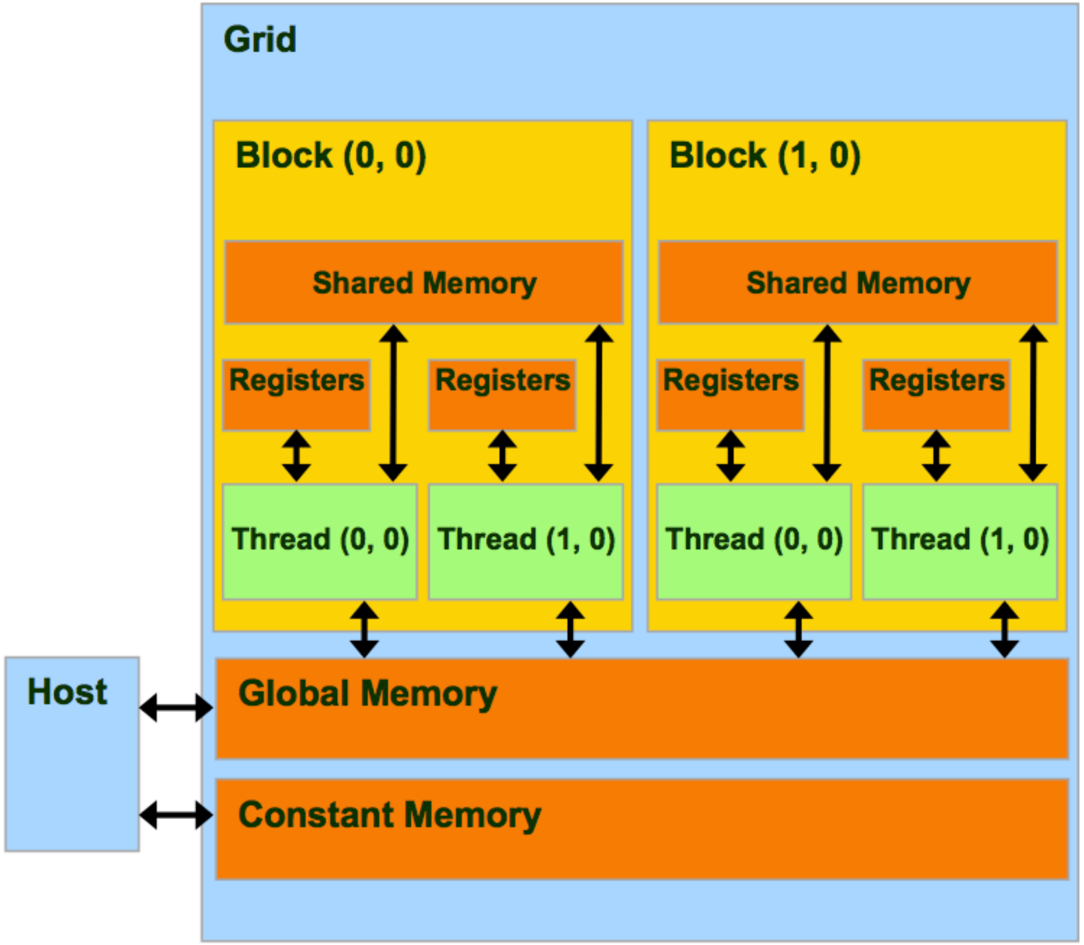
Some CUDA basics

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

```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

```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
```

C

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

Often followed by a structured code block

`kernels` construct

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

Clauses

```
if( condition )
async( expression )
```

or data clauses

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

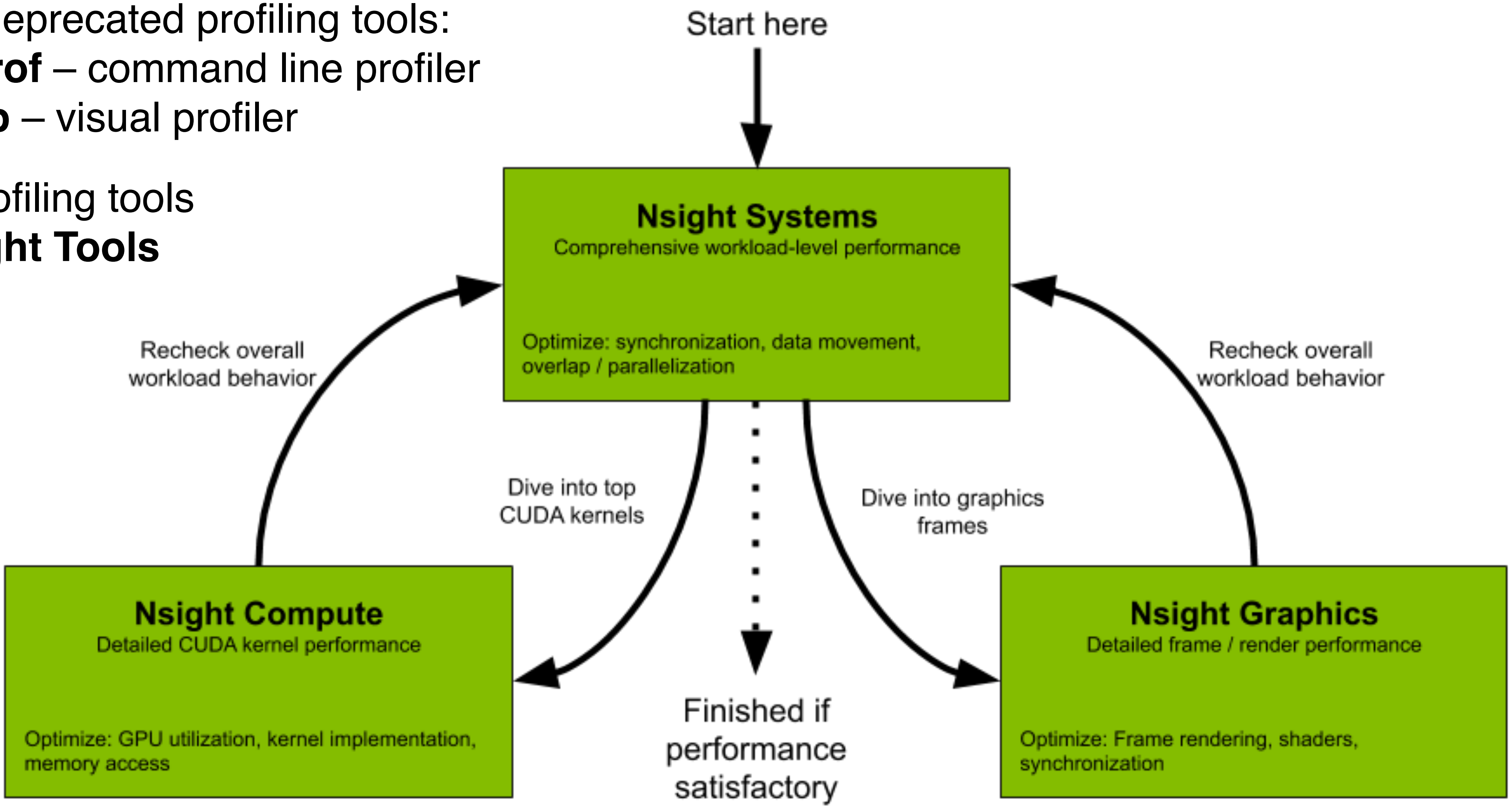
Nvidia profiling tools

Older, deprecated profiling tools:

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

New profiling tools

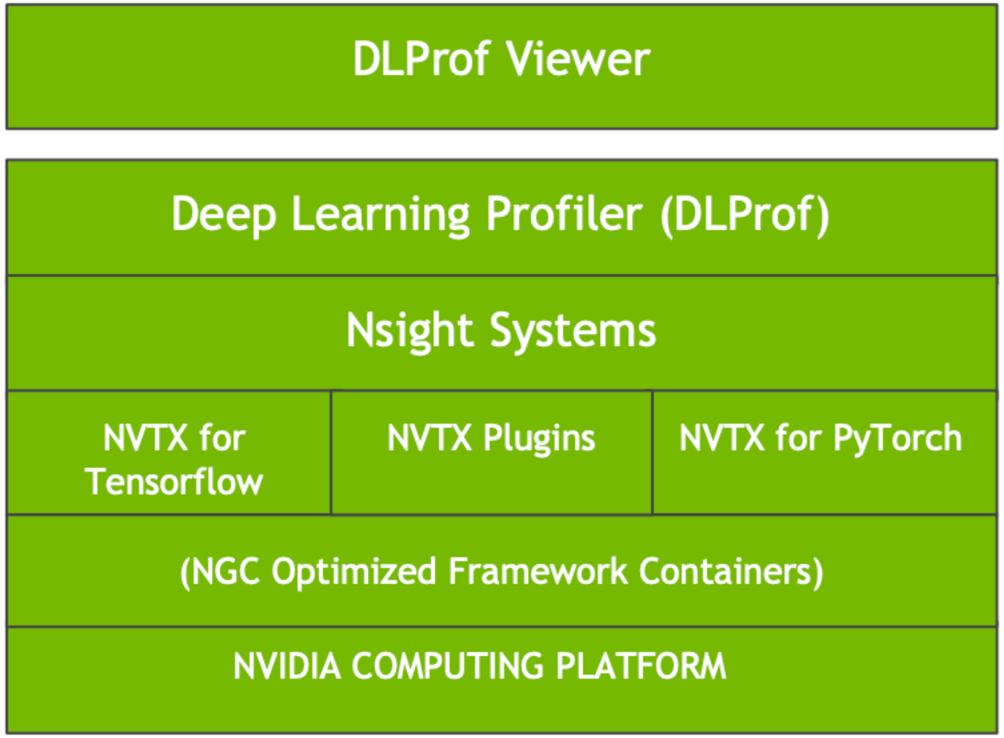
- **Nsight Tools**



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 optimized by Nvidia (nvidia-tensorflow) contains 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

TensorFlow TensorRT

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

Profile → Visualize in viewer

PyTorch

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

Add few lines of code → Profile → Visualize in viewer

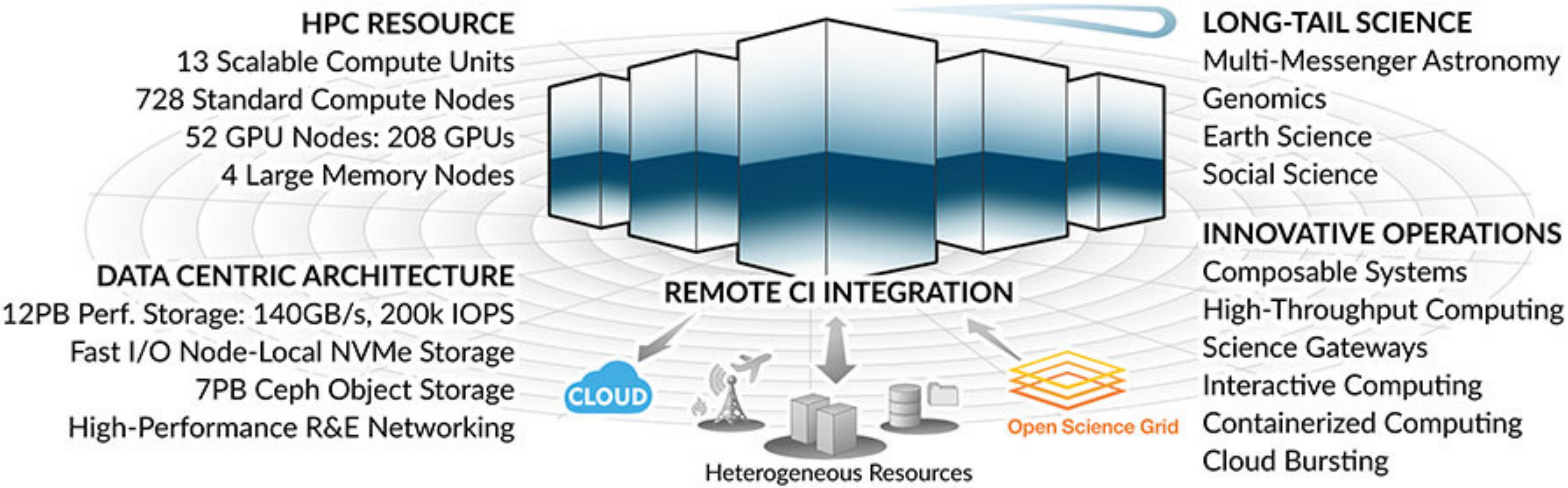
Outline

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

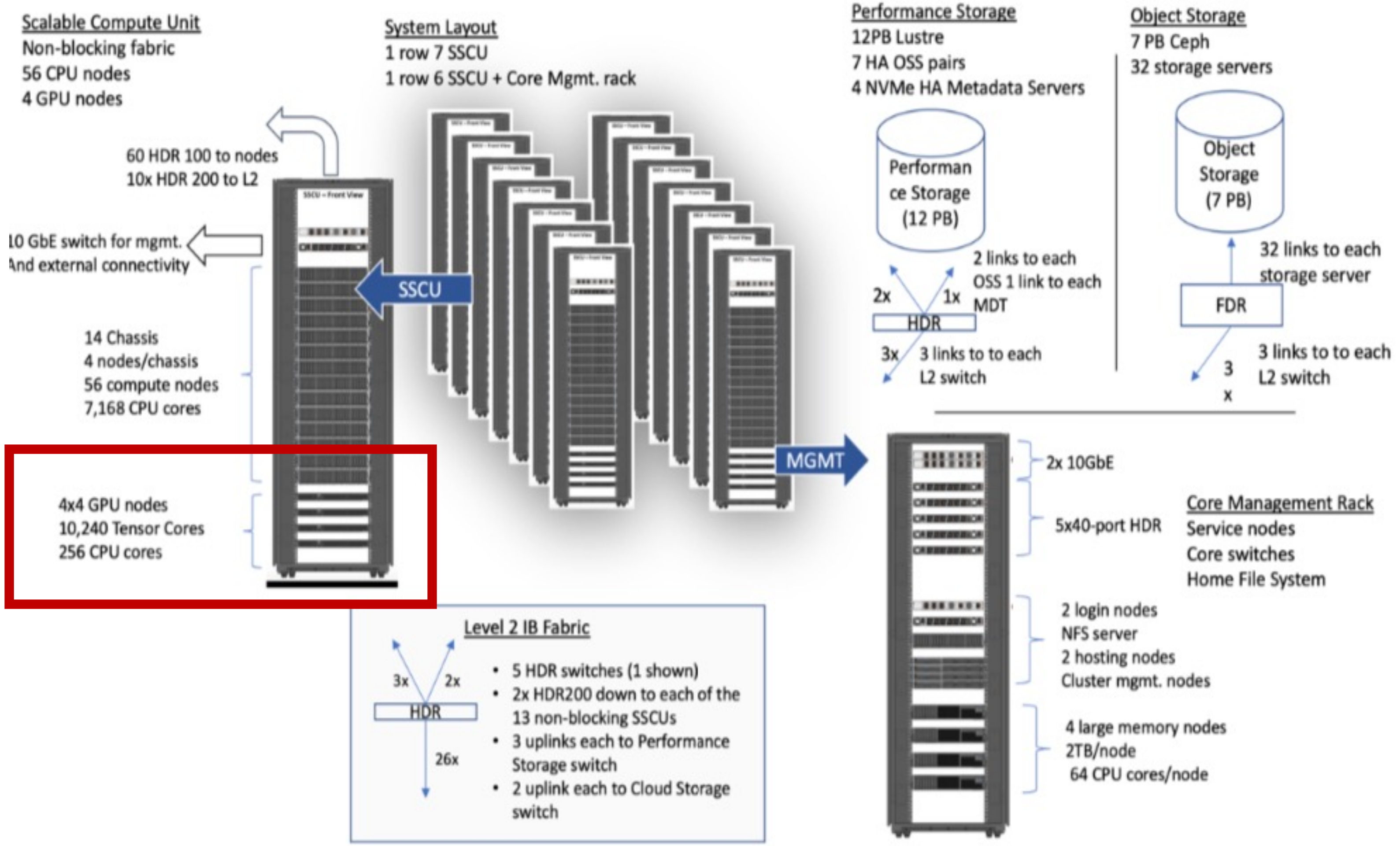
Launched in Fall 2020



Expanse Heterogeneous Architecture

System Summary

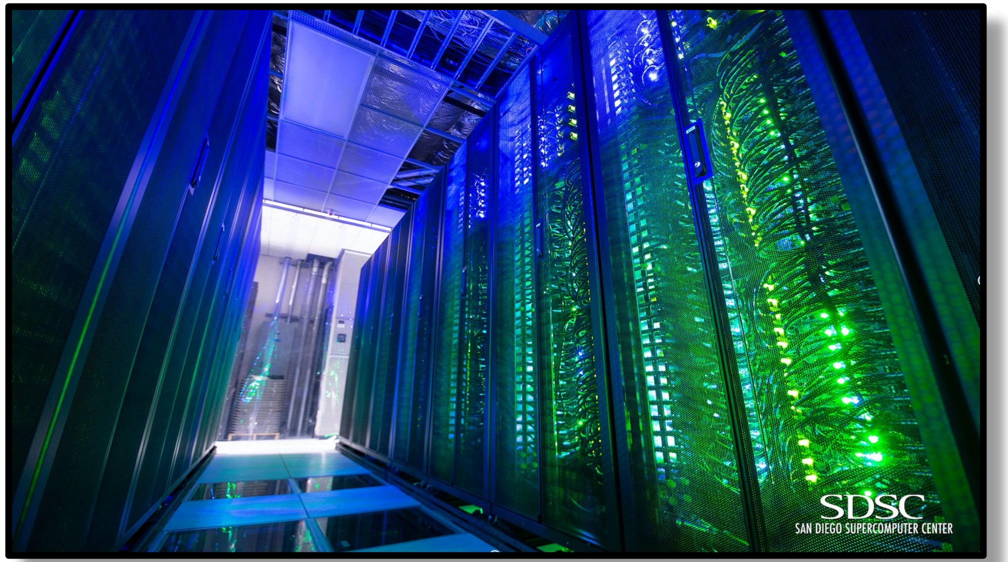
- 13 SDSC Scalable Compute Units (SSCU)
- 728 Standard Compute Nodes
- 93,184 Compute Cores
- 200 TB DDR4 Memory
- 52x 4-way GPU Nodes w/NVLINK
- **208 V100 GPUs**
- 4x 2TB Large Memory Nodes
- HDR 100 non-blocking Fabric
- 12 PB Lustre High Performance Storage
- 7 PB Ceph Object Storage
- 1.2 PB on-node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

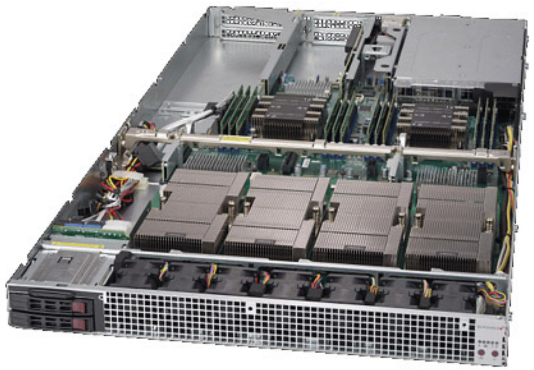


SDSC Expanse GPU nodes with Nvidia V100 SXM2

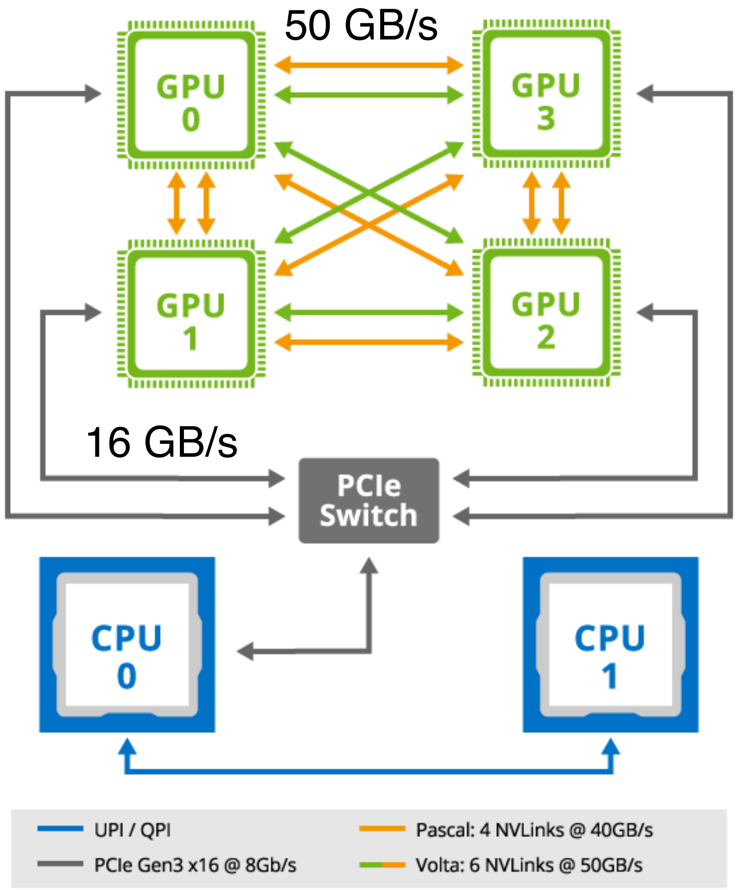
52 GPU nodes

- 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



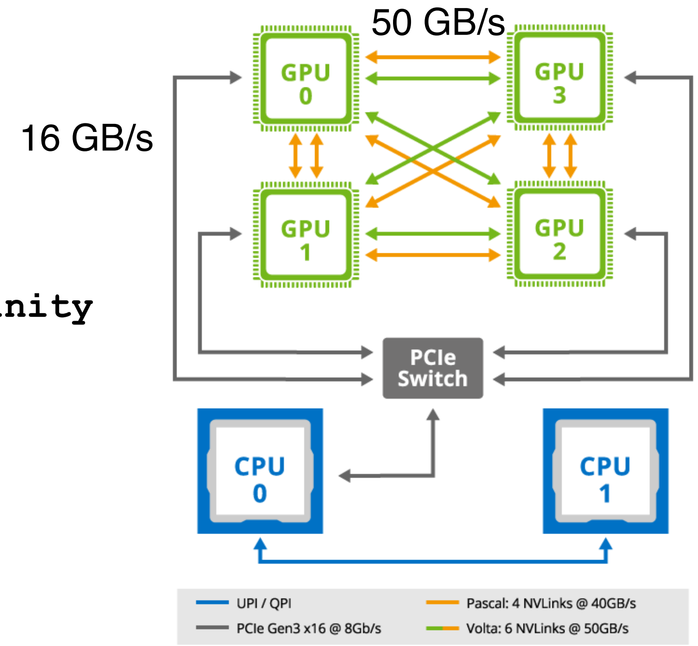
SDSC Expanse GPU nodes with Nvidia V100 SXM2

Node topology

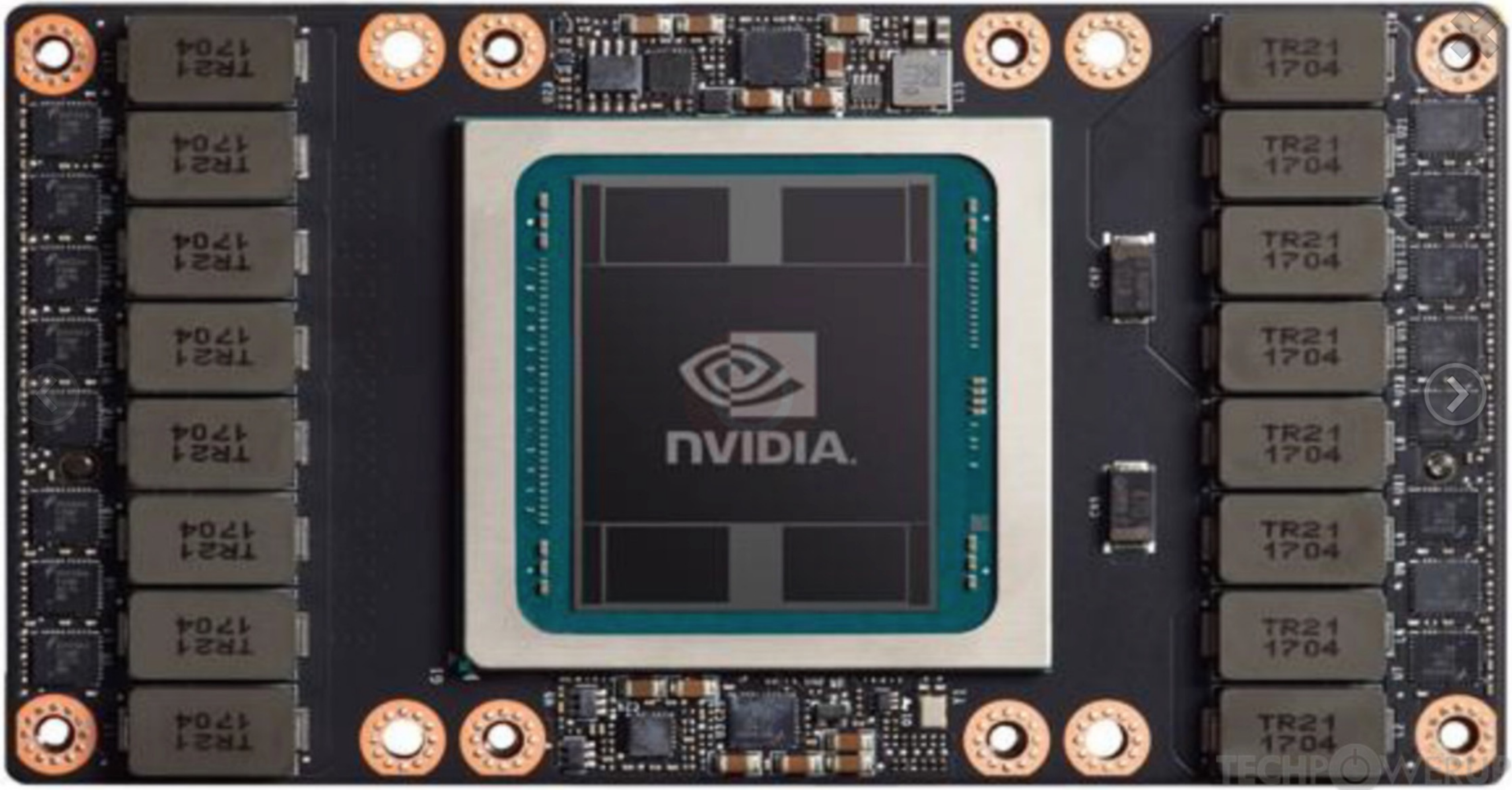
	GPU0	GPU1	GPU2	GPU3	mlx5_0	CPU Affinity	NUMA Affinity
GPU0	X	NV2	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU1	NV2	X	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU2	NV2	NV2	X	NV2	SYS	1,3,5,7,9,11	1
GPU3	NV2	NV2	NV2	X	SYS	1,3,5,7,9,11	1
mlx5_0	NODE	NODE	SYS	SYS	X		

Legend:

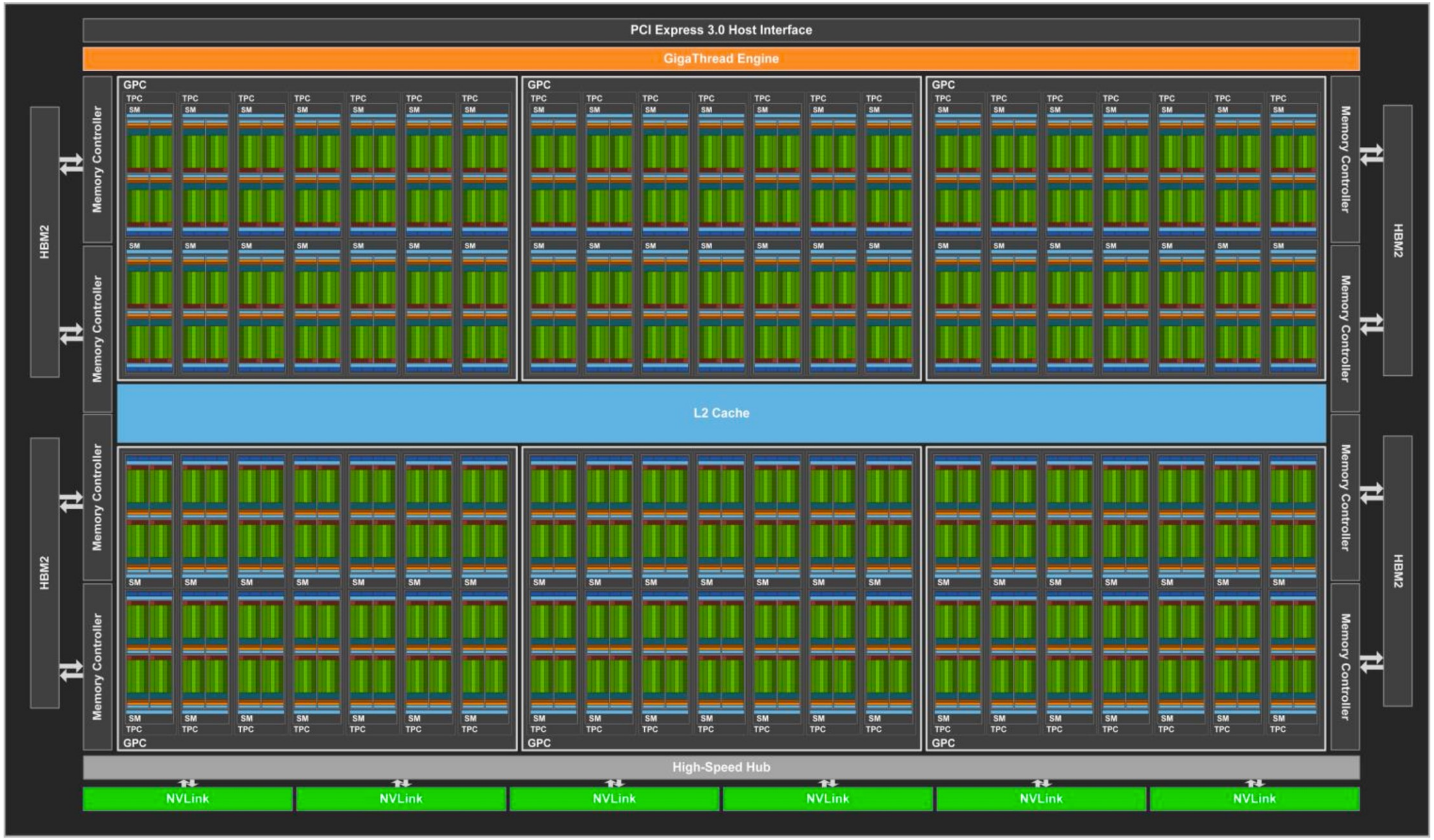
```
X    = Self
SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the 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



SDSC Expanse – V100 SXM2 GPUs



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 agoetz@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
[agoetz@login01 ~]$
```

SDSC Expanse GPU nodes

- The GPU nodes can be accessed via two different partitions
  "gpu" (entire nodes with 4 GPUs) and "gpu-shared" (individual GPUs).

```
#SBATCH --partition=gpu
```

or

```
#SBATCH --partition=gpu-shared
```

- In addition to the partition name (required), the number of GPUs must be specified.

```
#SBATCH --gpus=n
```

- For example, to obtain access to a single GPU for 30 minutes in an interactive session

```
srun --partition=gpu-shared --nodes=1 --gpus=1 --ntasks-per-node=1 --cpus-per-task=10 \
     --mem=92G --time=00:30:00 --wait=0 --pty /bin/bash

srun: job 2210010 queued and waiting for resources
srun: job 2210010 has been allocated resources
[agoetz@exp-8-59 ~]$
```

SDSC Expanse GPU nodes

- Note to make requests proportional to the number of available resources.
  - 4 x V100 GPUs
  - 40 CPU cores
  - 374 GB RAM
- Do not request more than 10 CPU cores and 92GB RAM per GPU, otherwise you will be charged for proportionally more time.
- Purge, then load GPU related modules

```
module purge
module reset
```

```
# Load CUDA Toolkit
module load cuda11.7/toolkit

# Load Nvidia HPC SDK
module load nvhpc
```

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

- Interactive access to GPU nodes

```
[agoetz@login01 ~] srun-shared  # This is an alias (check alias for full command)
```

- 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                                                   Usage      |
|=============================================================================|
|  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 ~]$ tar xvf $CIML_DATA_DIR/cuda-samples-v11.6.tar.gz ~/
```

- Explore CUDA Toolkit samples – great resource!

```
[agoetz@exp-8-59 ~]$ cd cuda-samples-11.6
[agoetz@exp-8-59 cuda-samples]$ ls Samples
0_Introduction  2_Concepts_and_Techniques  4_CUDA_Libraries   6_Performance
1_Utilities     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`, which prints information on available GPUs

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

Thank you!