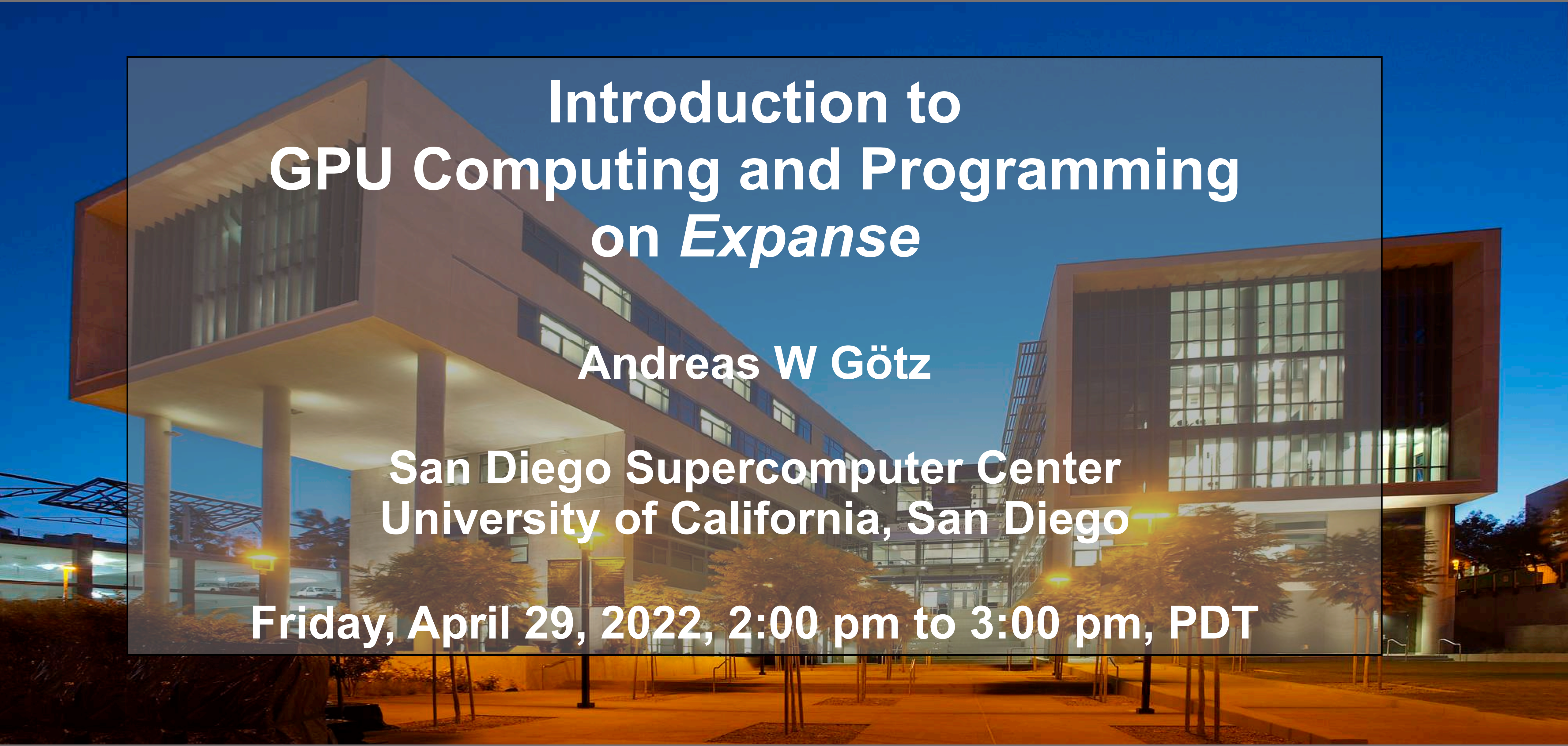


Introduction to GPU Computing and Programming on *Expanse*

Andreas W Götz

San Diego Supercomputer Center
University of California, San Diego

Friday, April 29, 2022, 2:00 pm to 3:00 pm, PDT

SDSC SAN DIEGO SUPERCOMPUTER CENTER

UC San Diego

Webinar overview

We will cover the following topics

- GPU hardware overview
- GPU accelerated software examples
- Programming GPUs
    - GPU enabled libraries
    - CUDA C programming basics
    - OpenACC introduction
- 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. More recently, Field Programmable Gate Arrays (FPGAs).

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


GeForce 6600 GT Personal Cinema 2004


NVIDIA GeForce GTX 280 2008

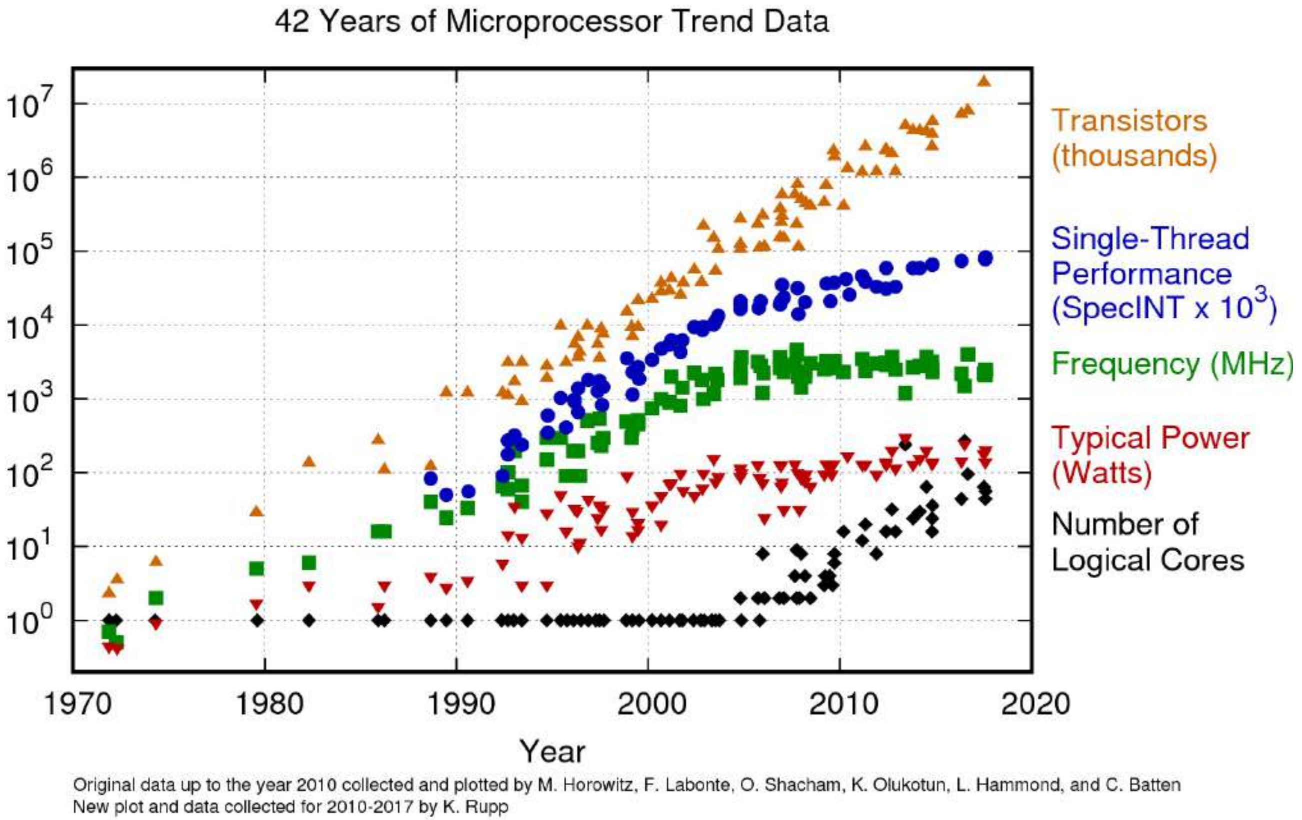
Why is there such an interest in GPUs?

Moore's law

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

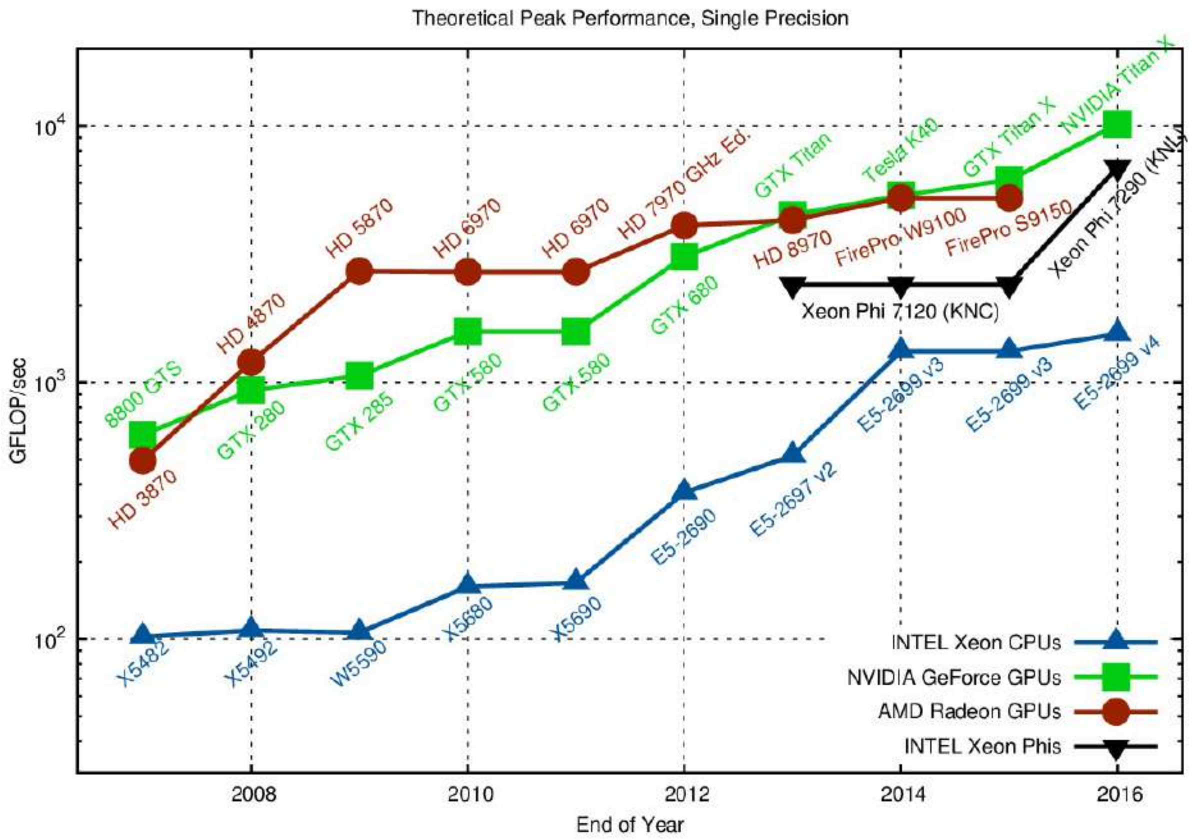
Trends since mid 2000s

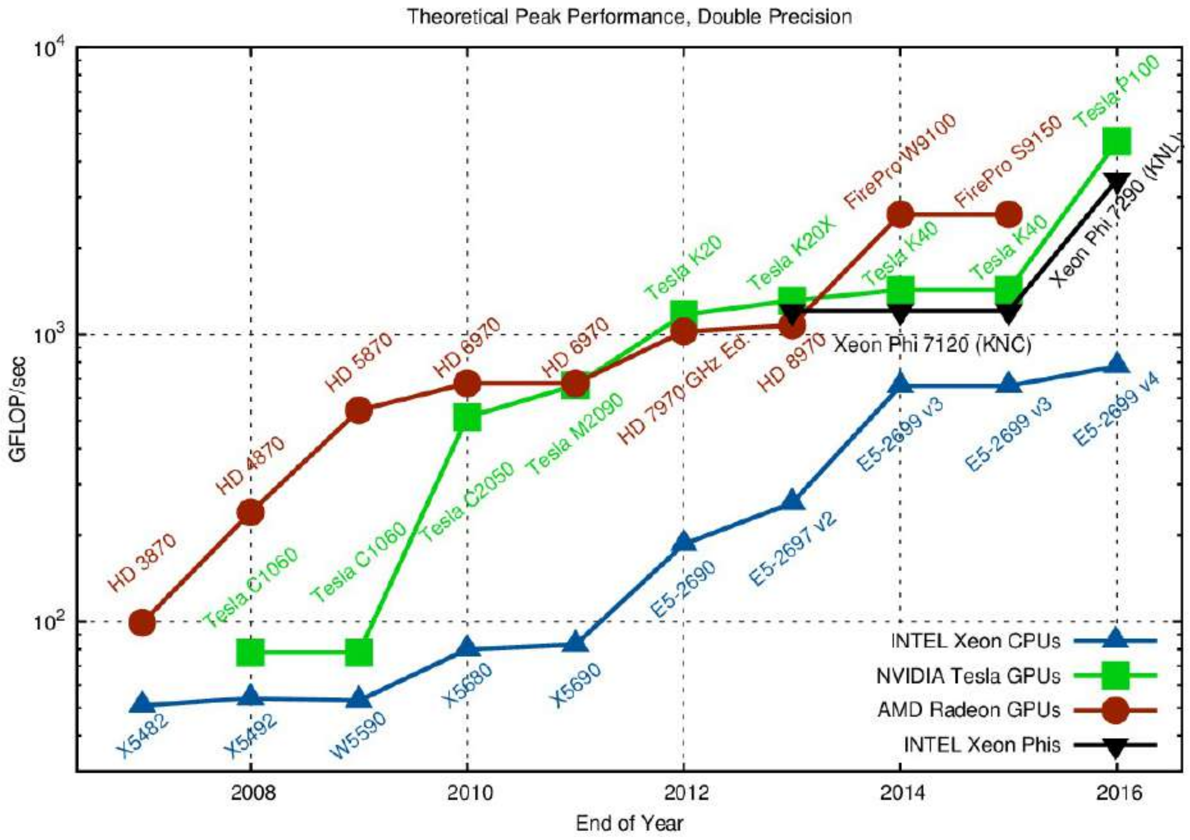
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.



Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Why is there such an interest in GPUs?

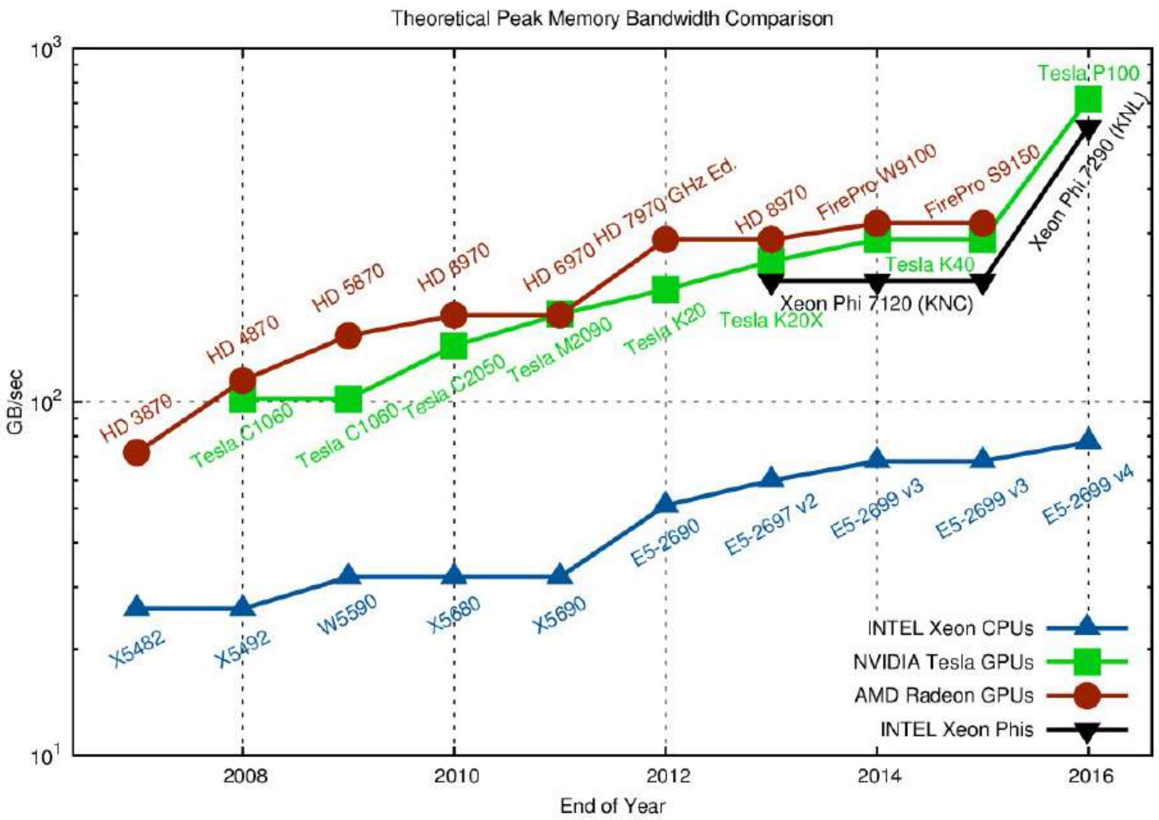


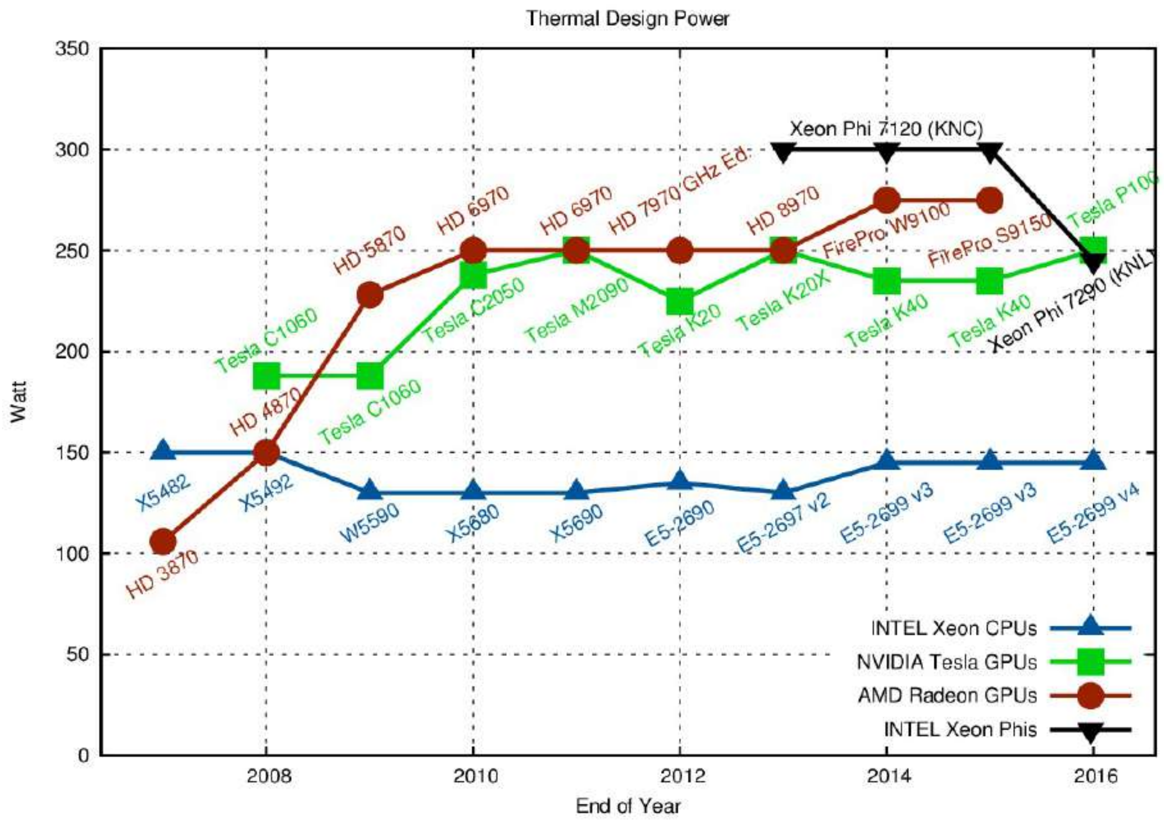


- GPUs offer significantly higher 32-bit floating point performance than CPUs.
- Datacenter GPUs also offer significantly higher 64-bit floating point performance than CPUs.

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

Why is there such an interest in GPUs?

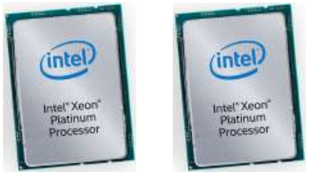




- GPUs have significantly higher memory bandwidth than CPUs.
- Given power consumption, a fair comparison would be a single GPU to 2-socket CPU server.

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

Comparison of top X86 CPU vs Nvidia V100 GPU

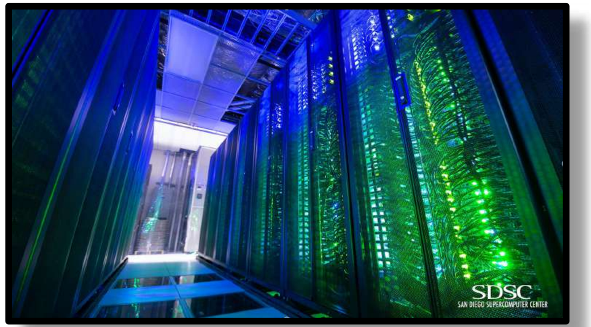


Aggregate performance numbers (FLOPs, BW)	**Dual socket Intel 8180 28-core (56 cores per node)**	**Nvidia Tesla V100, dual cards in an x86 server**
Peak DP FLOPs	4 TFLOPs	14 TFLOPs (3.5x)
Peak SP FLOPs	8 TFLOPs	28 TFLOPs (3.5x)
Peak HP FLOPs	N/A	224 TFLOPs
Peak RAM BW	~ 200 GB/sec	~ 1,800 GB/sec (9x)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

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
  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

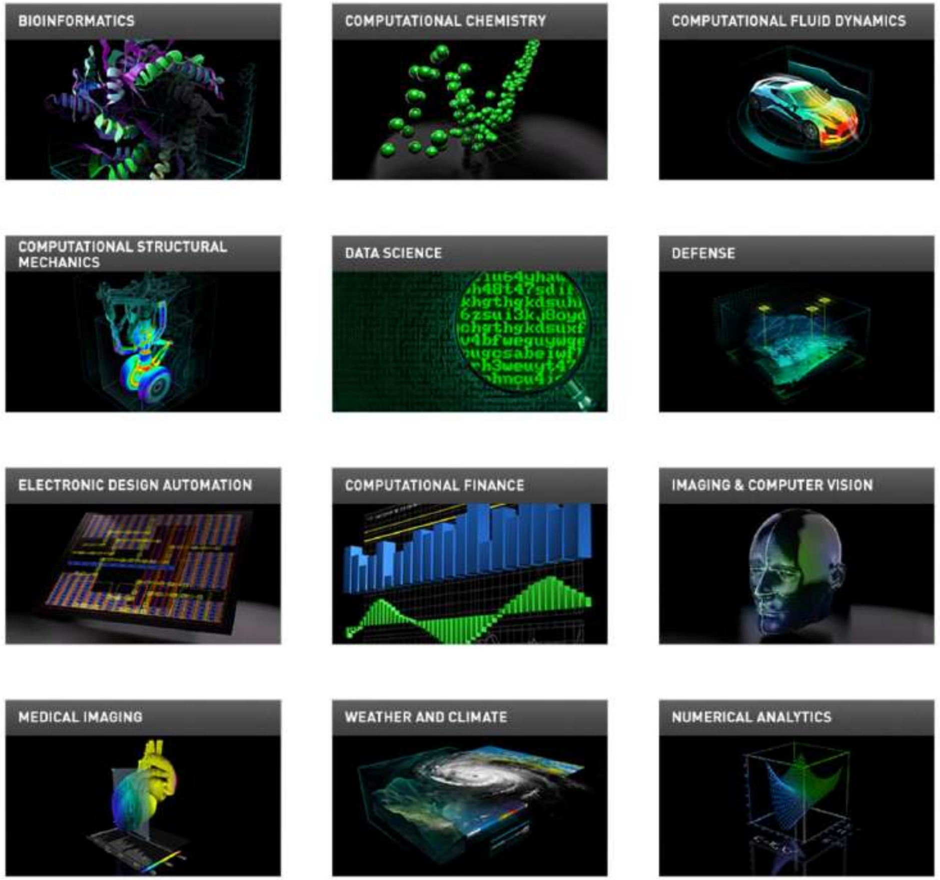
DIY 4 x Nvidia RTX 3080 box (2020)

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

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...



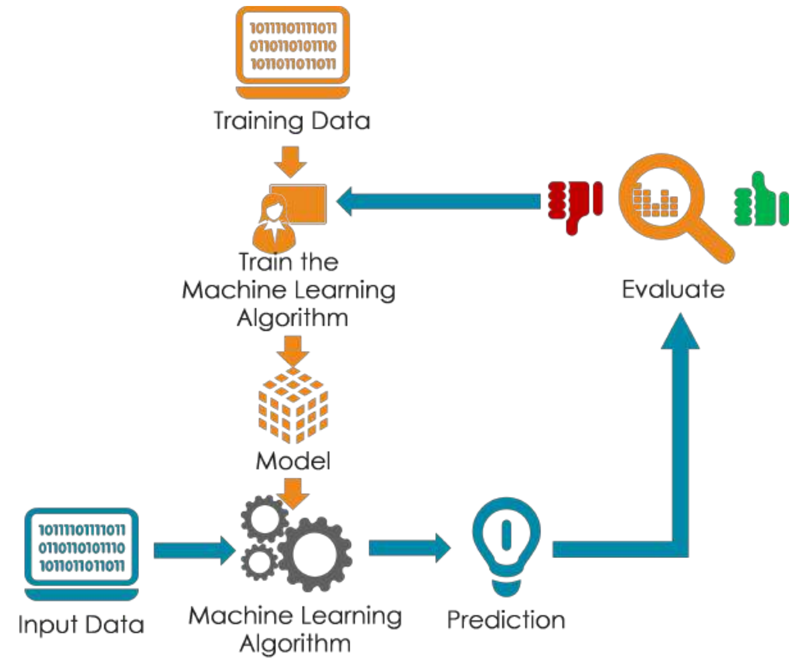
Applications: Deep learning

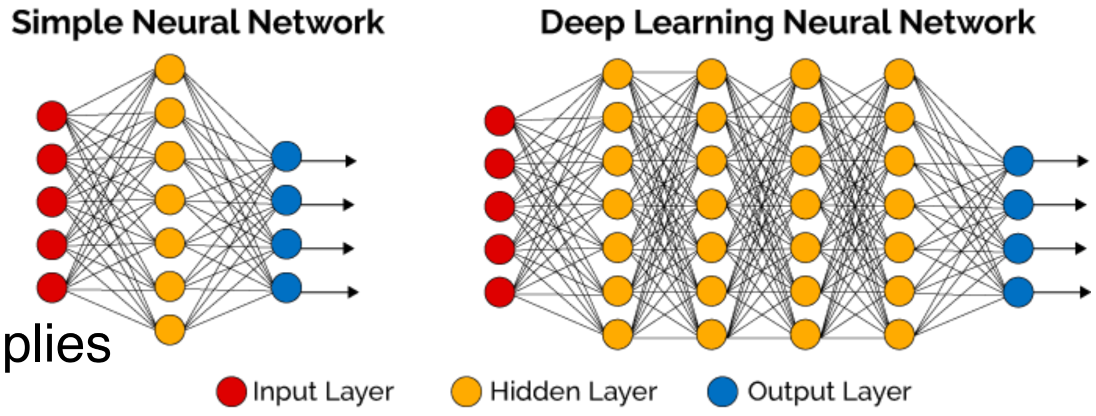
Machine learning

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

Deep learning

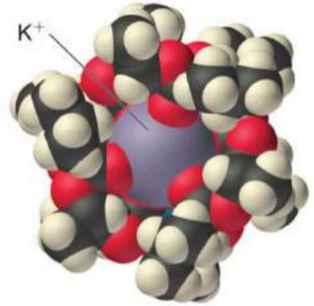
- Neural networks with many hidden layers.
- Tensor operations (matrix multiplications).
- GPUs are very efficient at these (4x4 matrix algebra is used in 3D graphics)
- Half-precision arithmetic can be used for many ML applications, at least for inference.
- Nvidia Volta architecture introduced tensor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

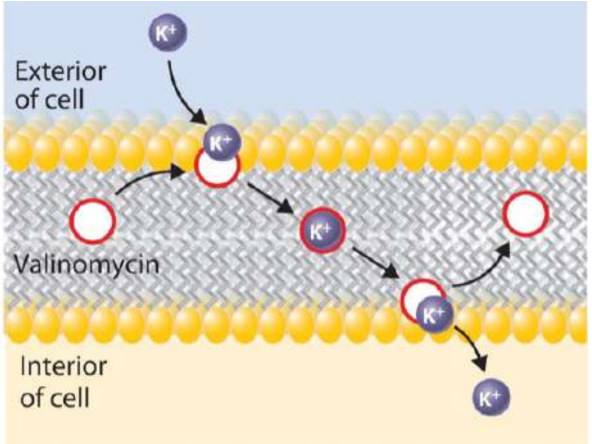




Applications: Quantum Chemistry

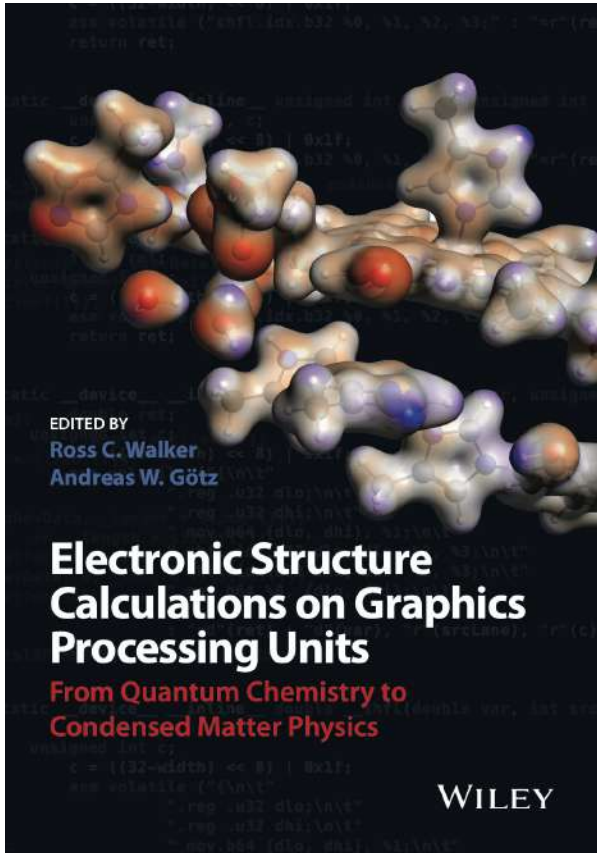
Valinomycin

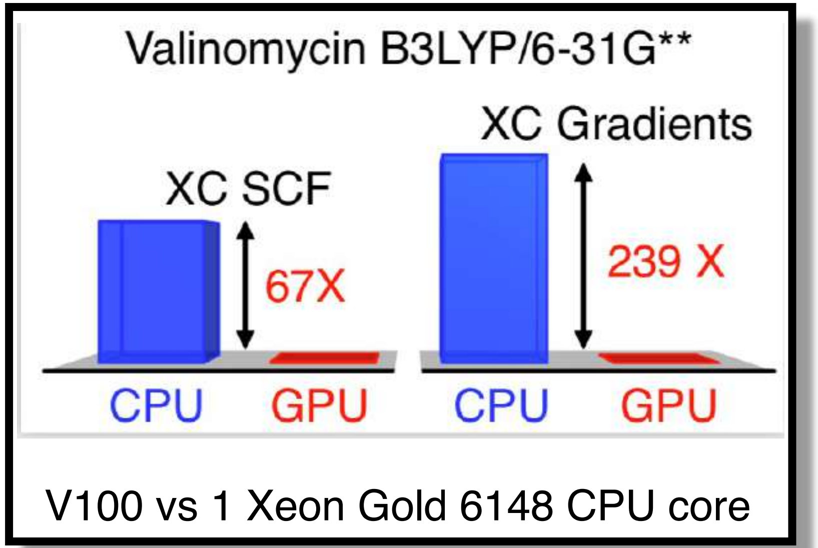


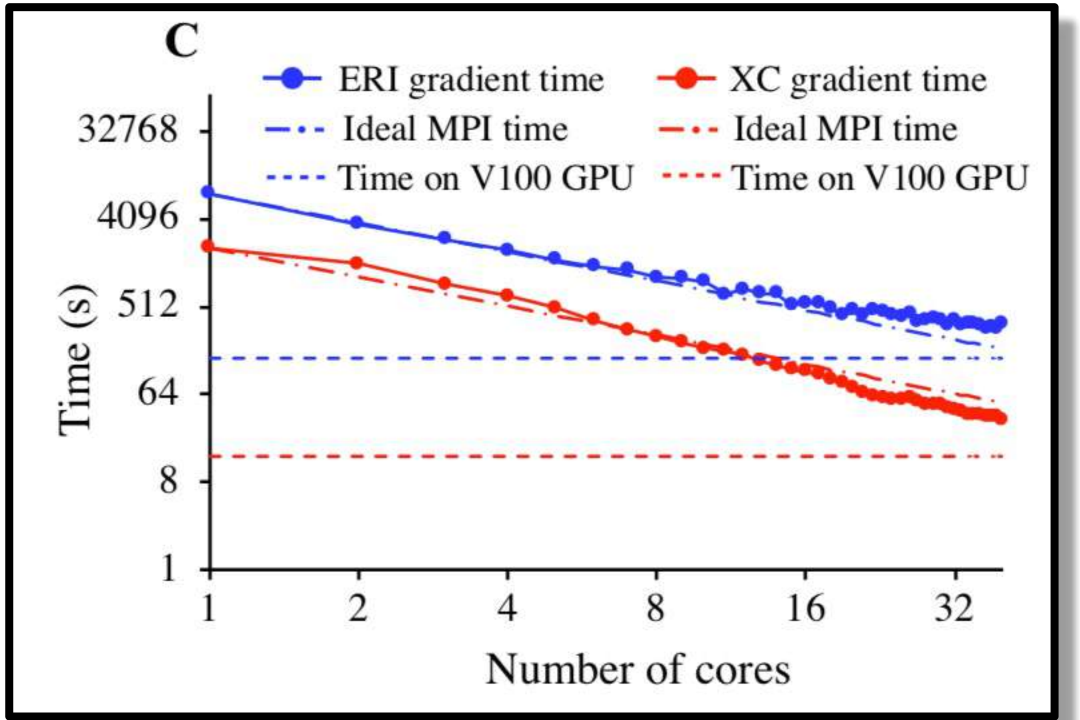


Example: Open source QUICK code

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





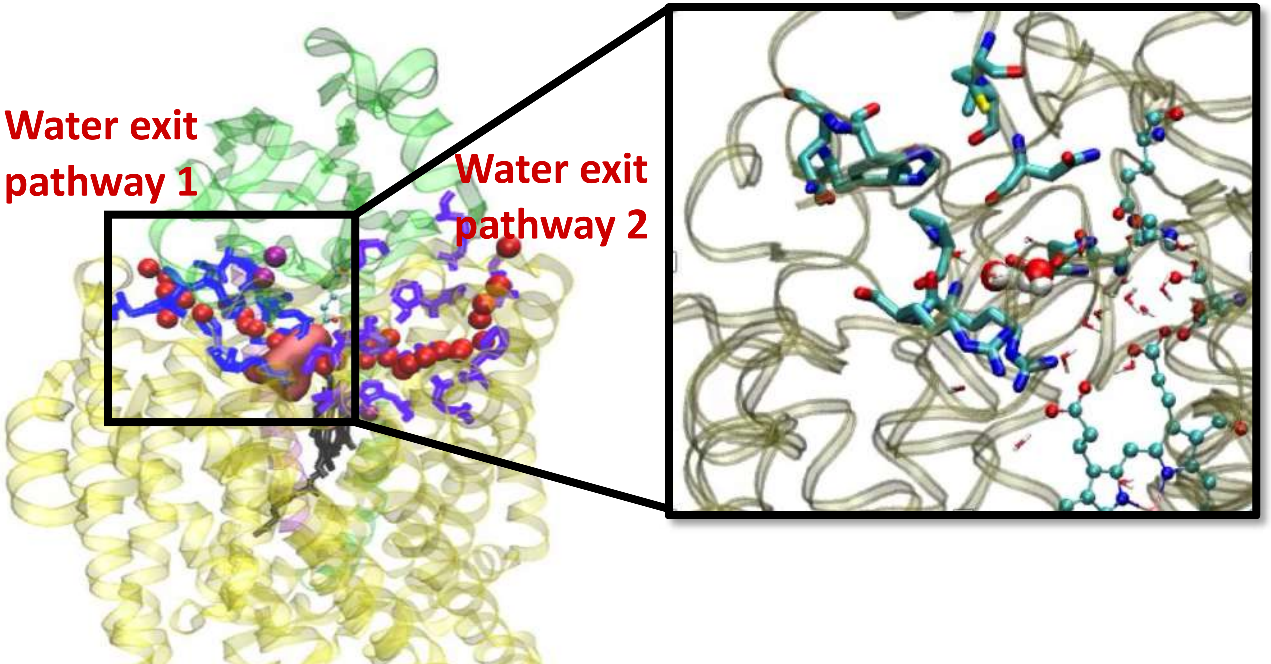


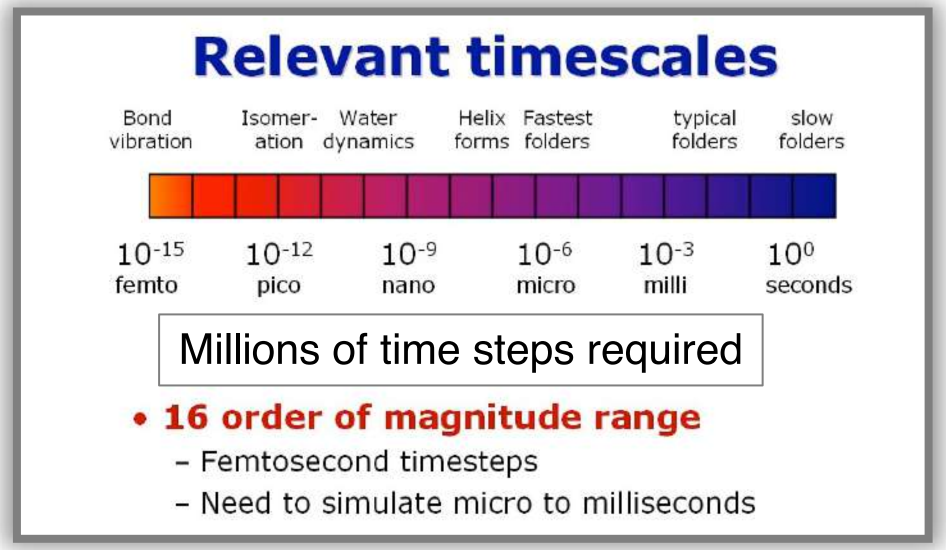
See *J. Chem. Theory Comput.* **16**, 4315-4326 (2020) https://dx.doi.org/10.1021/acs.jctc.0c00290

Applications: Molecular Dynamics

Example: Amber MD code

- Atomistic simulations of condensed phase biomolecular systems





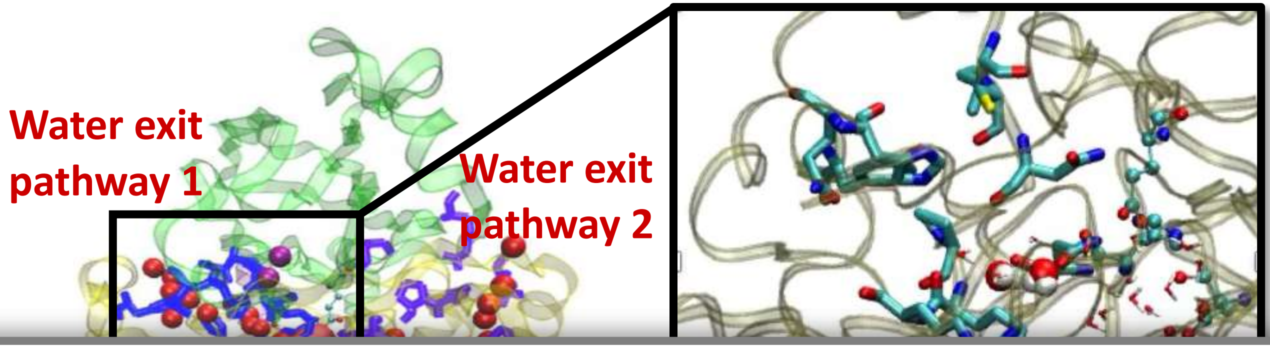
Cytochrome c oxidase enzyme

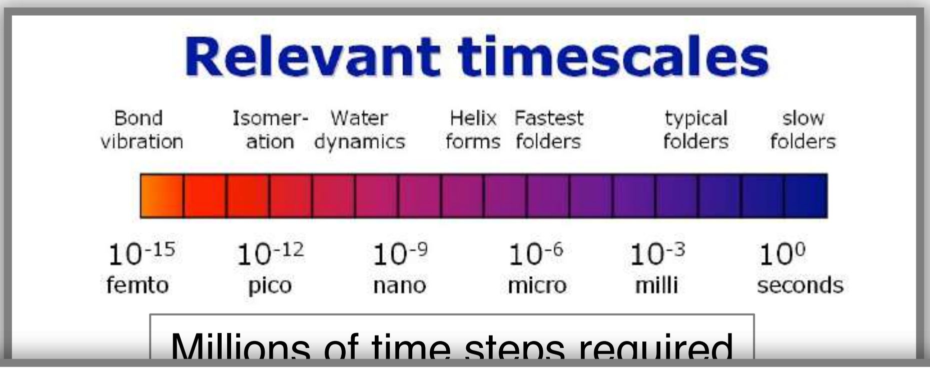
Yang, Skjevik, Han Du, Noodleman, Walker, Götz, *BBA Bioenergetics* 2016 (1857) 1594.

Applications: Molecular Dynamics

Example: Amber MD code

- Atomistic simulations of condensed phase biomolecular systems



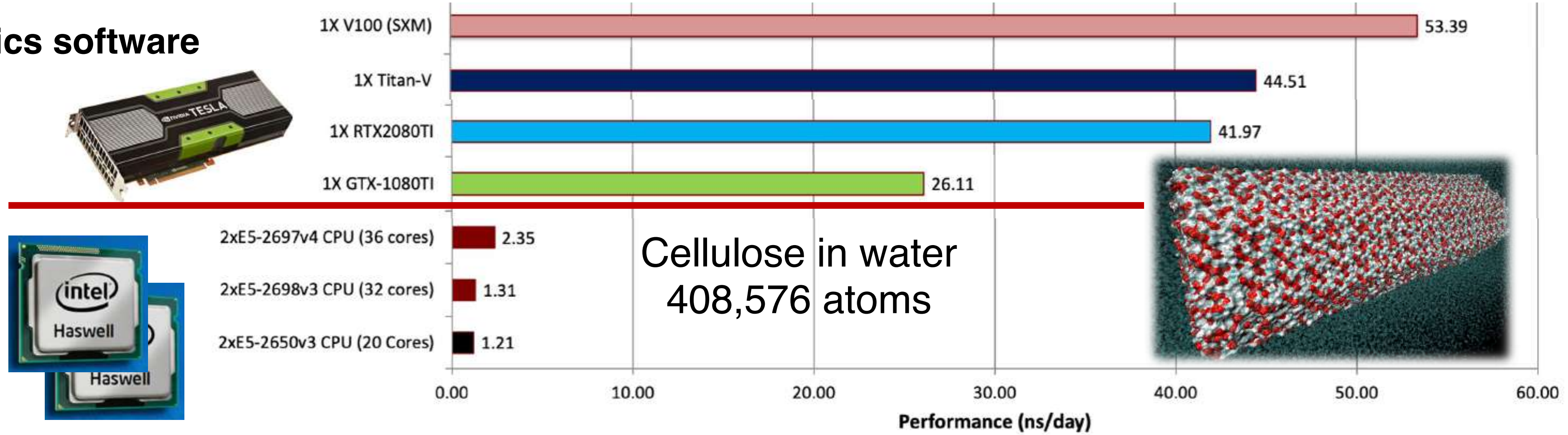


Amber 18 molecular dynamics software

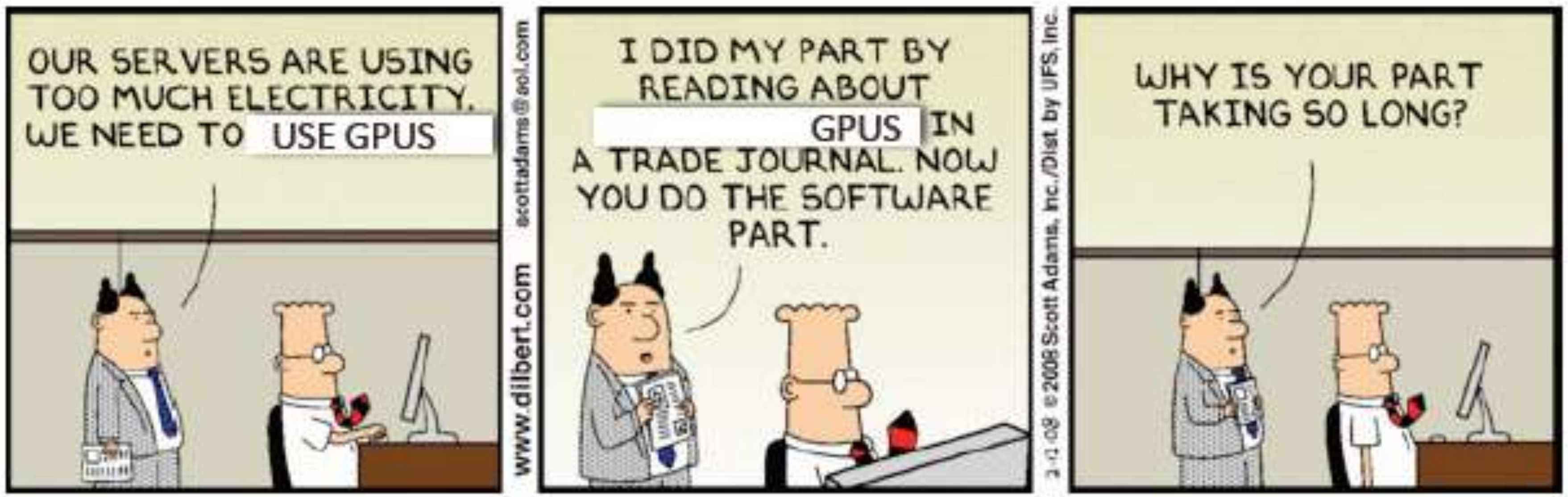
Götz, Williamson, Xu, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

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

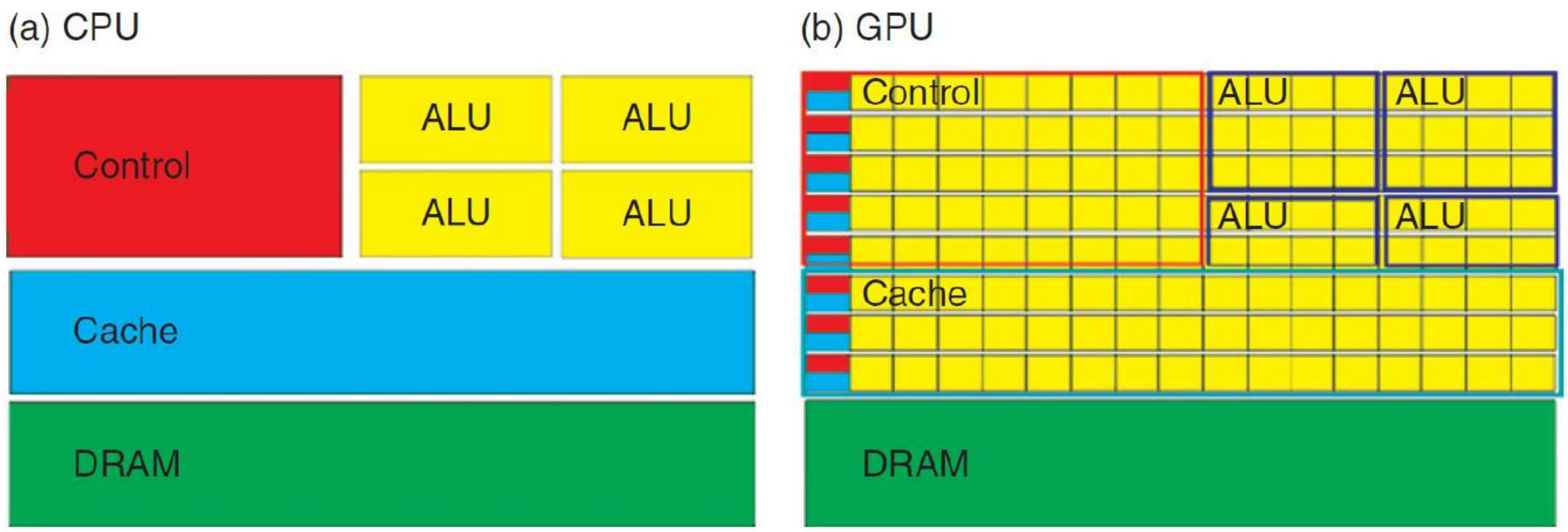
Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.



What’s the catch?



GPU vs CPU architecture



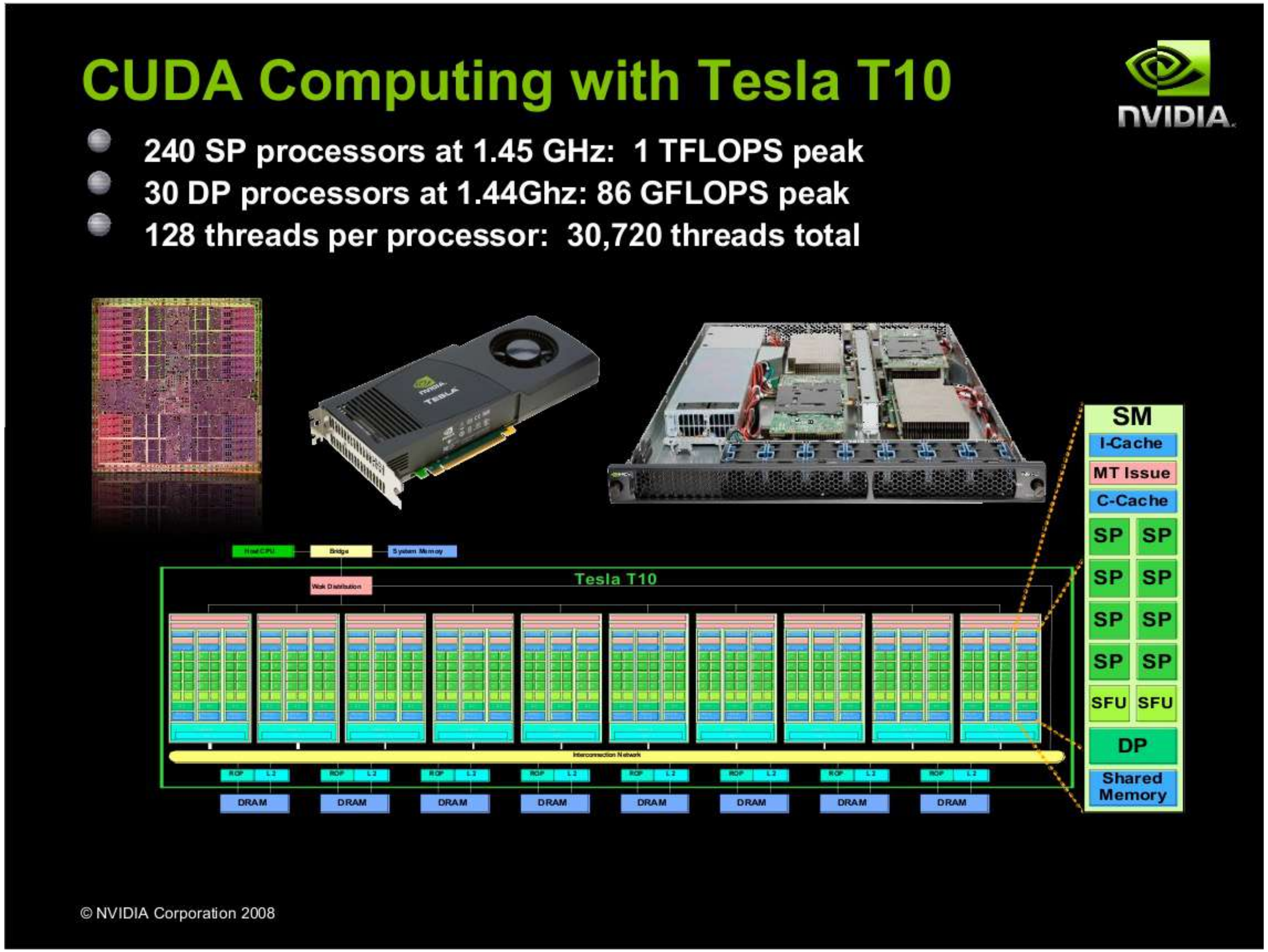
CPU

- 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

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

Hardware complexities

Hardware characteristics change across GPU models and generations

- Single precision / double precision 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

What this means for your program

Threads

- Never write code with any assumption for how many threads it will use.
- Use functions (CUDA calls) to query the hardware configuration at runtime.
- Launch many more threads than processing cores.

Data types

- Avoid using double precision (64-bit floats, FP64) where not specifically needed.
- Instead, use single precision (32-bit floats, FP32)
- Machine learning applications often work with half-precision (16-bit floats, FP16) or special data types like bfloat16 or Nvidia's TensorFloat

But be careful to make sure that reduced precision does not lead to reduced accuracy in your simulations.

GPU programming languages

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

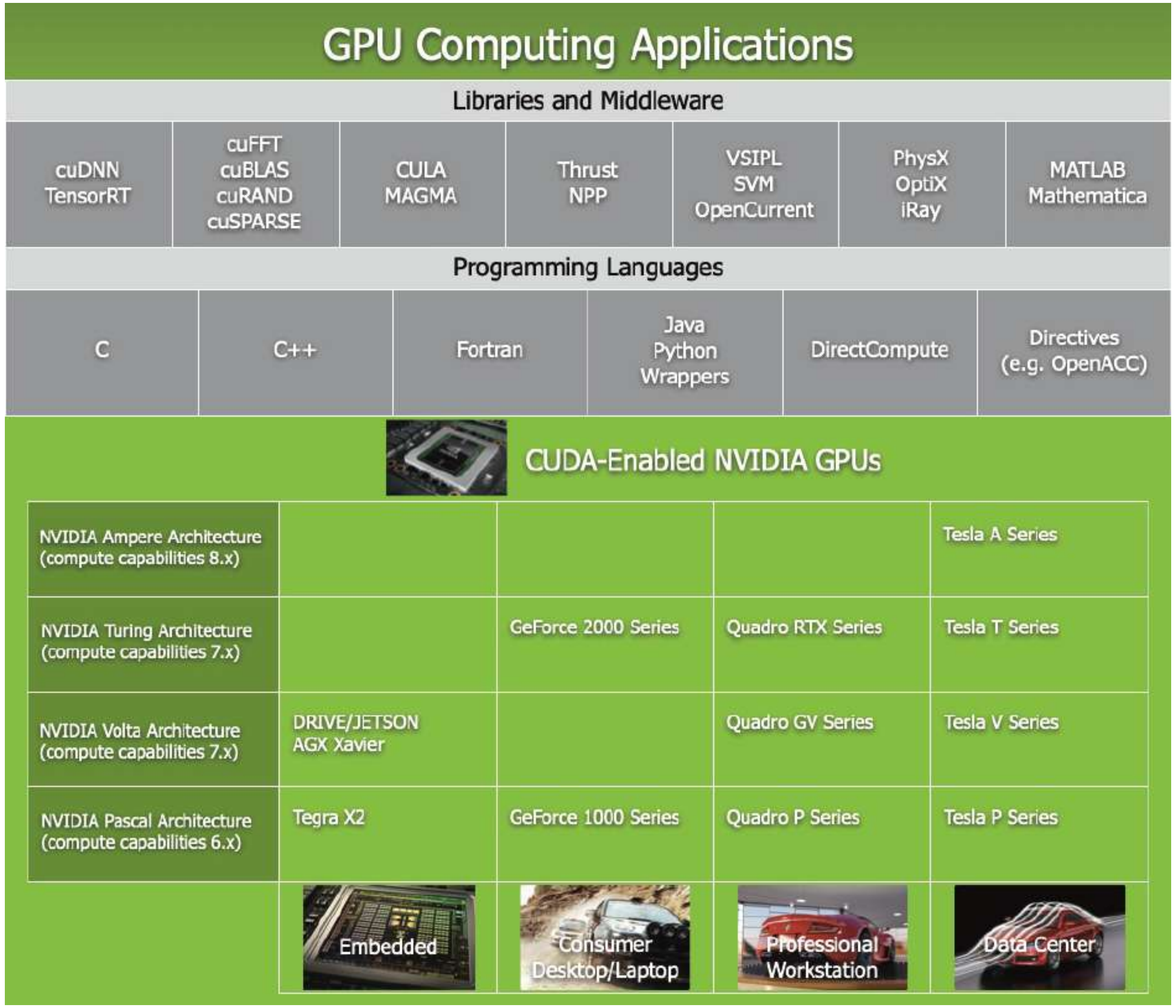
CUDA
- Proprietary, works only for Nvidia GPUs
- De-facto standard for high-performance code, C/C++, Fortran

HIP/ROCm
- AMD’s C++ solution, works for Nvidia (via CUDA) and AMD GPUs
- Syntax very similar to CUDA

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
- https://developer.nvidia.com/cuda-zone
- https://nvidia.com/getcuda

Activate on Expanse GPU nodes

```
$> module purge
$> module reset
$> module load cuda
```

- Currently loads CUDA 11.0.2
- CUDA 9.2 and 10.2 also available

Nvidia CUDA development tools

Nvidia HPC SDK (free)

- Replacement for the CUDA Toolkit
- Contains most of CUDA Toolkit including CUDA compiler nvcc, libraries, debuggers, profiler
- Nvidia C/C++, Fortran compiler (nvfortran, nvc, nvc++) (formerly PGI compilers)
- https://developer.nvidia.com/hpc-sdk

Activate on Expanse GPU nodes

```
$> module purge
$> module reset
$> module load nvhpc
```

- Currently loads NVHPC 22.2
- NVHPC 20.9 and 21.7 also available

3 ways to use GPUs

Applications

Libraries

OpenACC Directives

Programming Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

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)

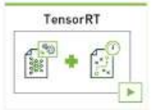
GPU accelerated libraries

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

Deep Learning Libraries

cuDNN

GPU-accelerated library of primitives for deep neural networks



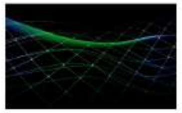
GPU-accelerated neural network inference library for building deep learning applications

Advanced GPU-accelerated video inference library

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

Signal, Image and Video Libraries

cuFFT

GPU-accelerated library for Fast Fourier Transforms

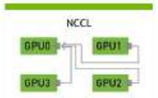
NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

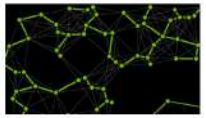
High-performance APIs and tools for hardware accelerated video encode and decode

Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics

Thrust

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;

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

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

$$y = a * x + y$$

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

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);
```

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

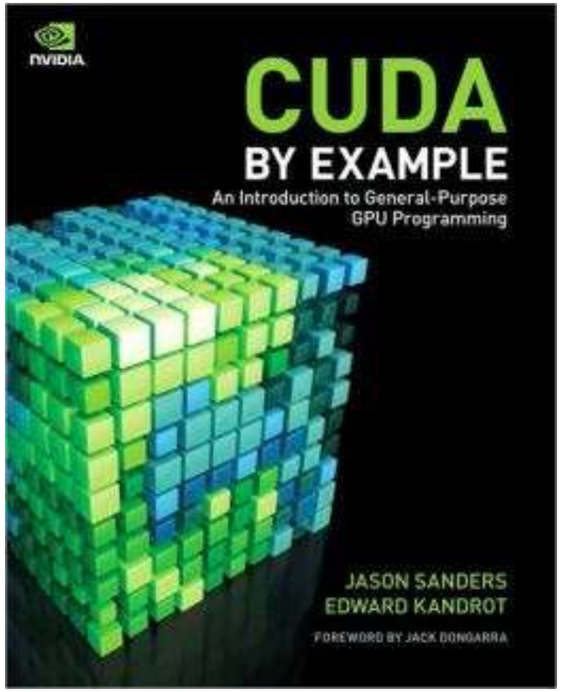
NVIDIA HPC SDK: https://docs.nvidia.com/hpc-sdk/index.html

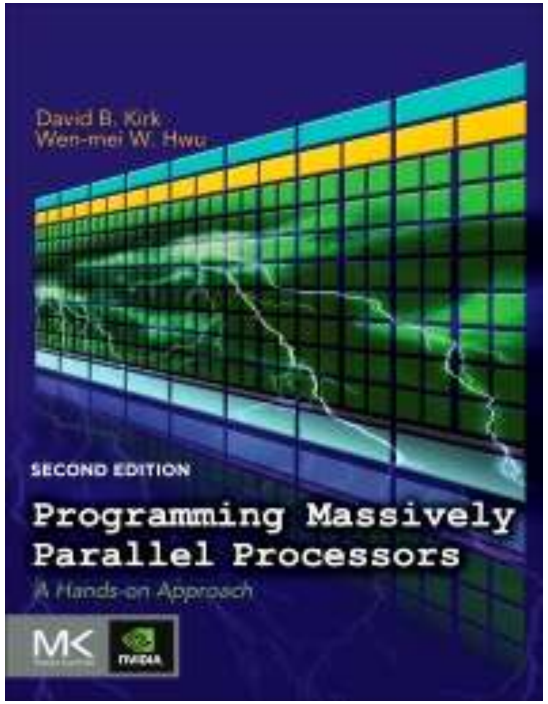
CUDA C: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

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

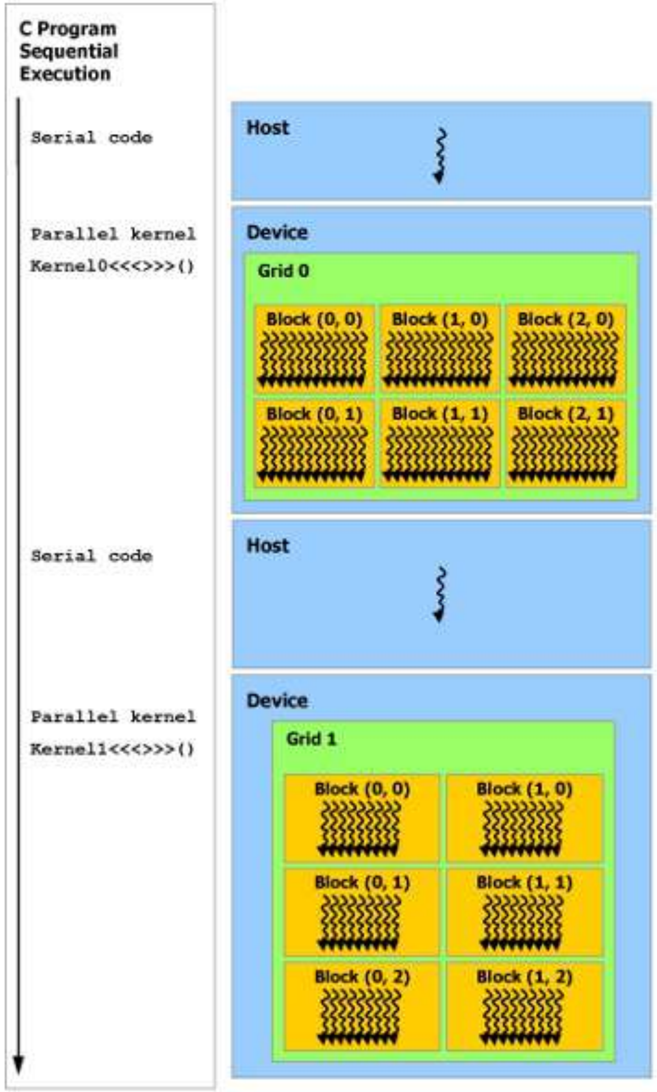
Many resources here: https://www.gpuhackathons.org/technical-resources

Good books to get started

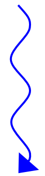




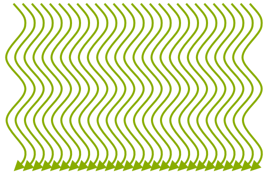
Heterogeneous Computing



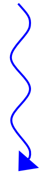
serial code



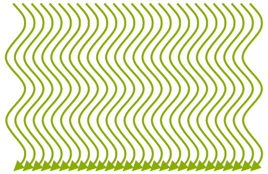
parallel code



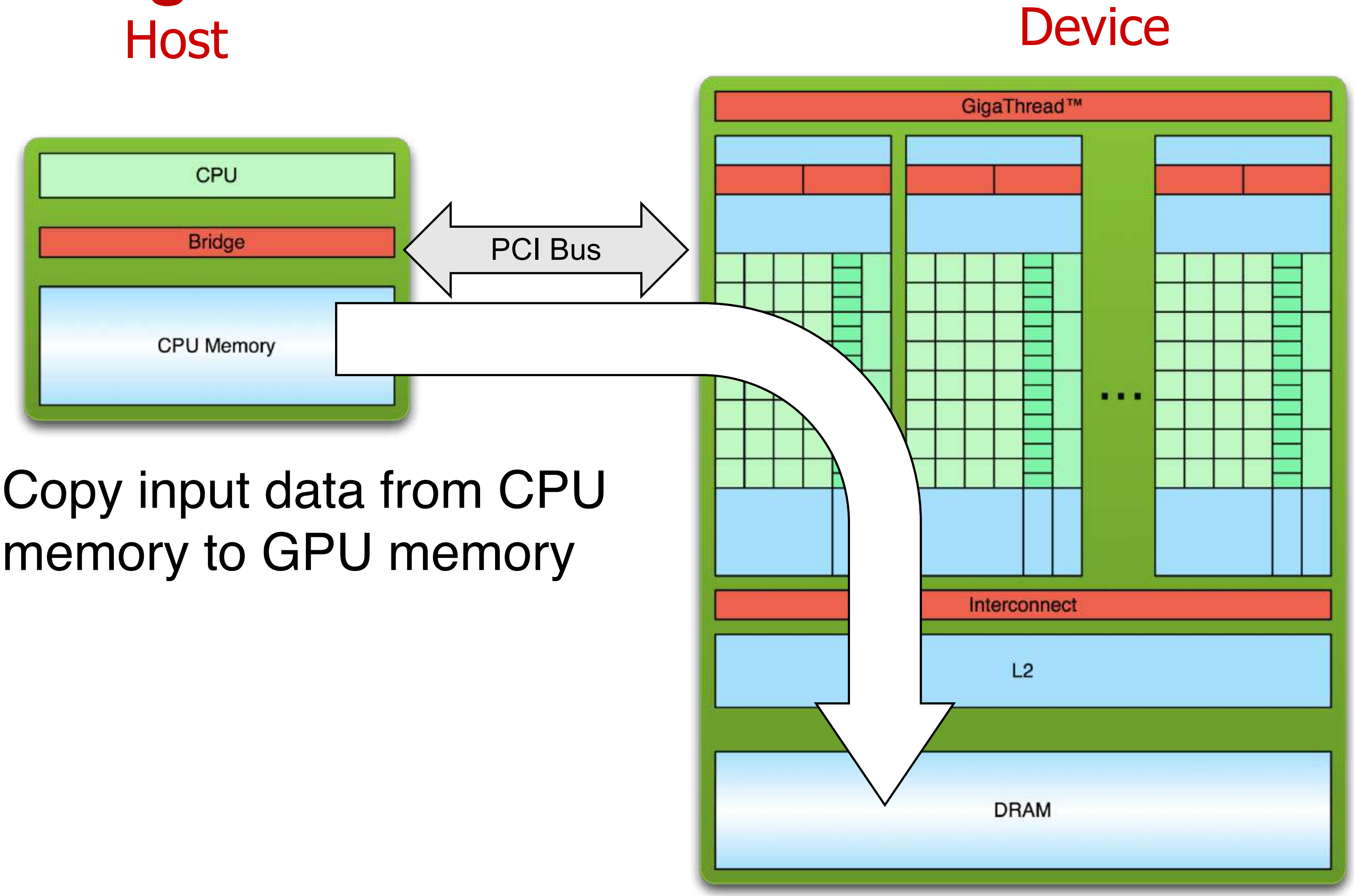
serial code



parallel code

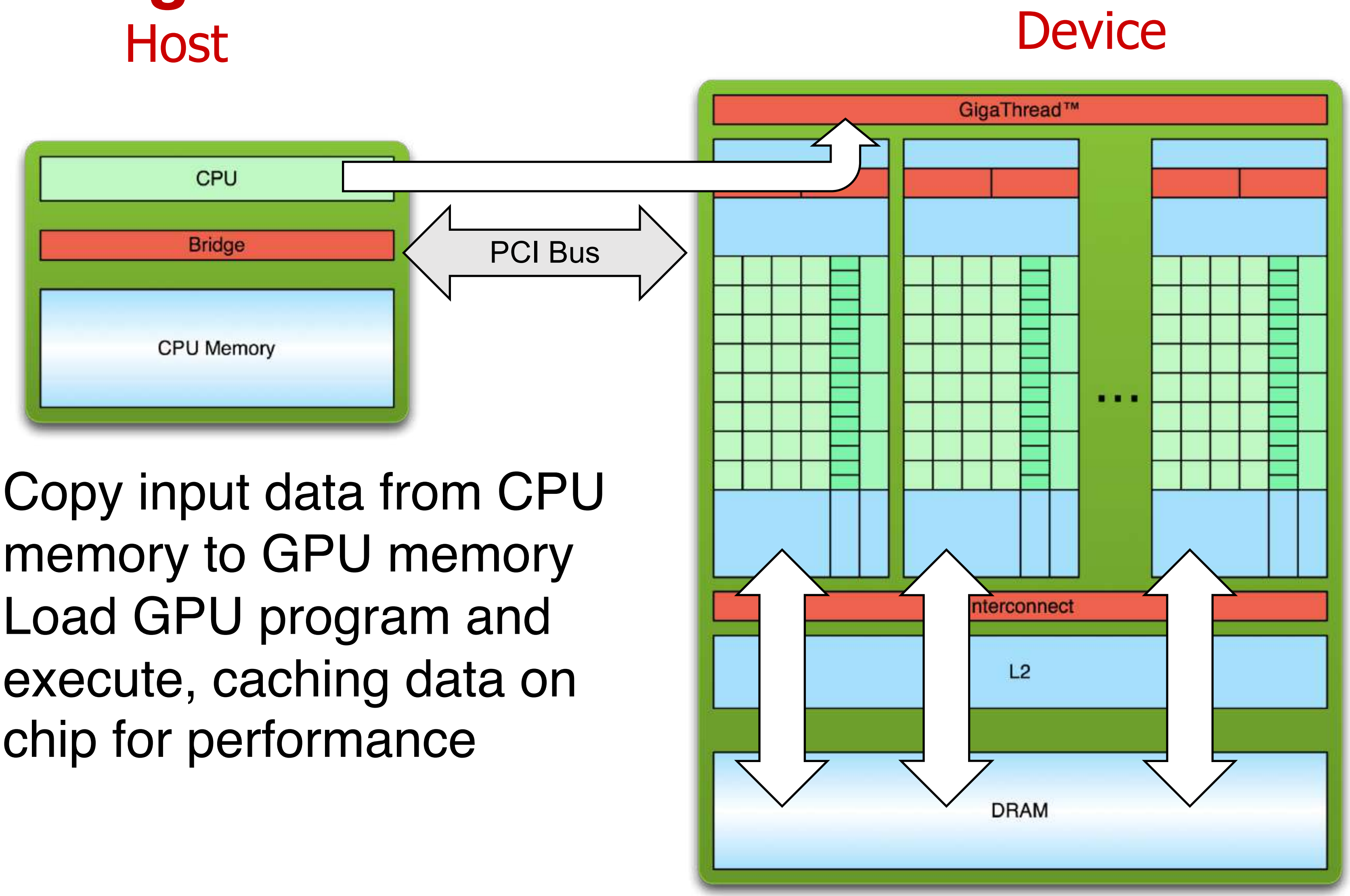


Processing Flow



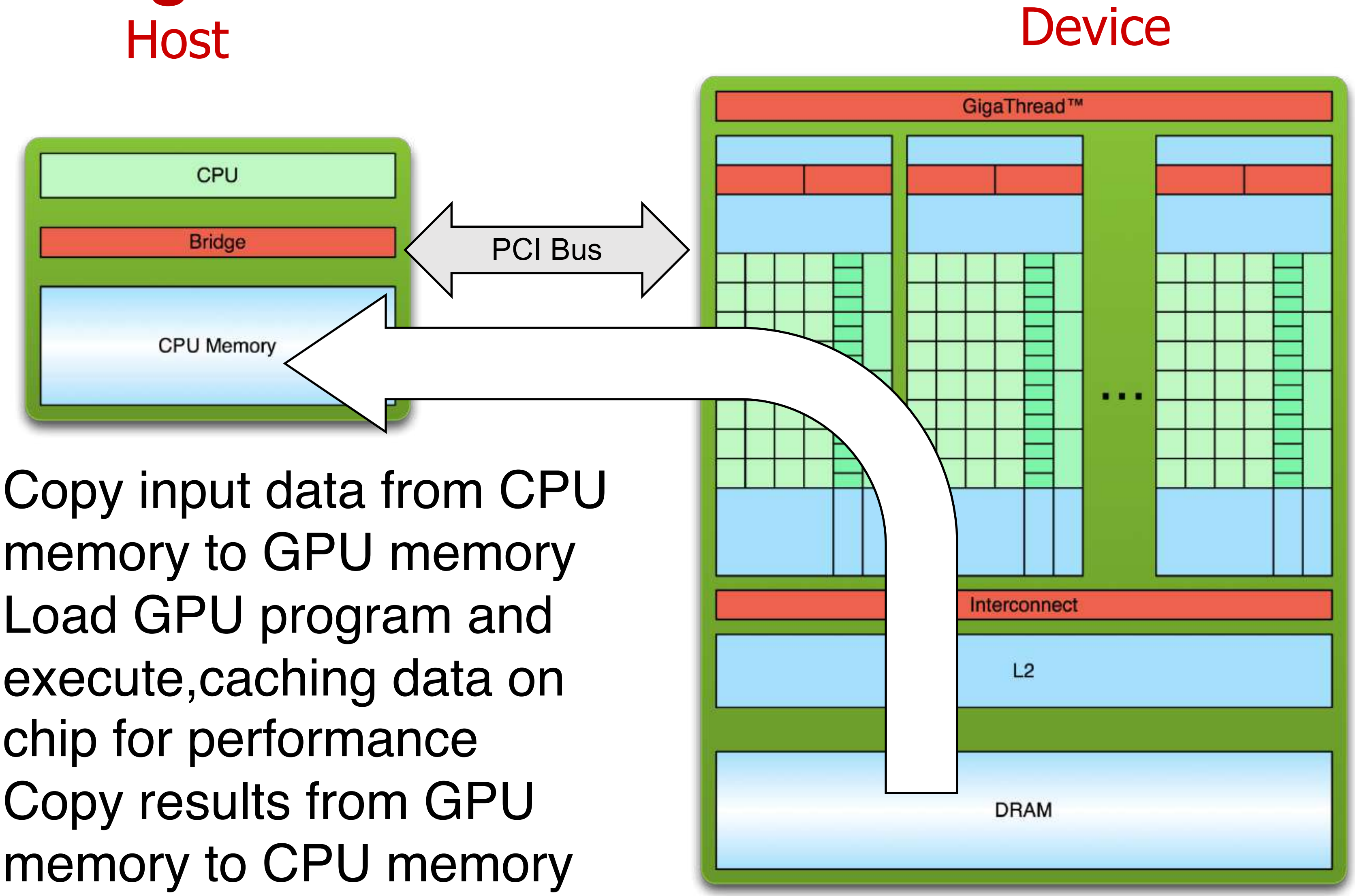
1. Copy input data from CPU memory to GPU memory

Processing Flow



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

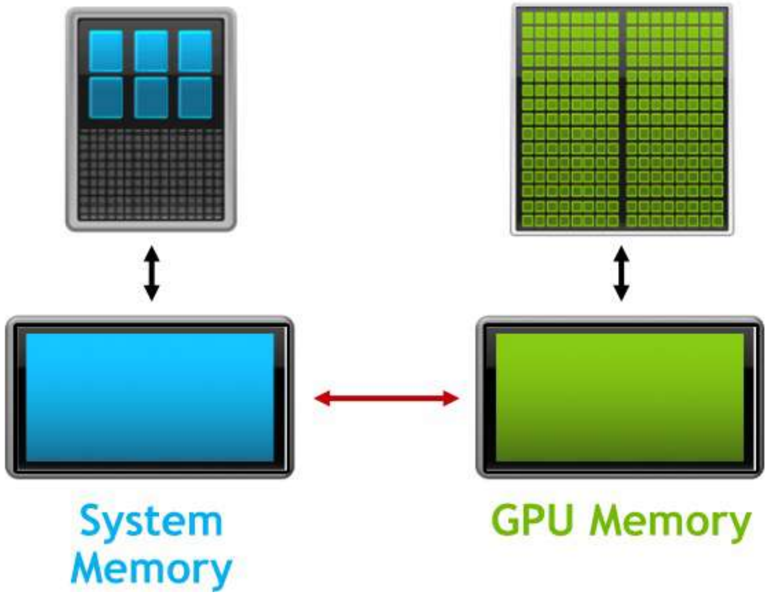
Processing Flow



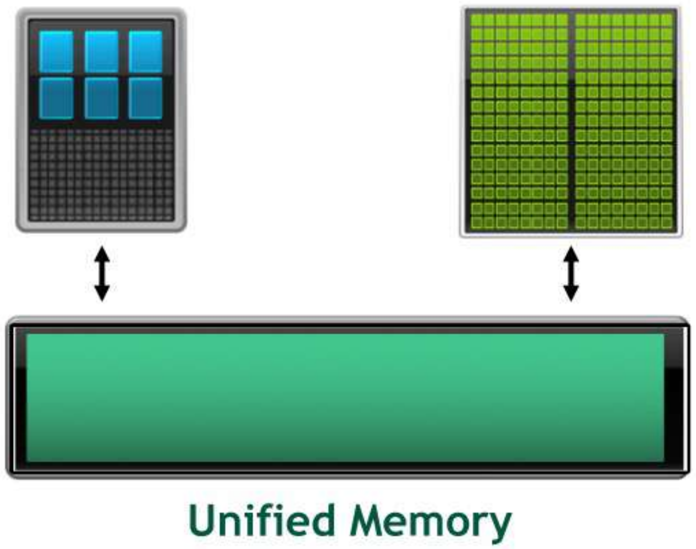
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Unified memory

Developer view so far



Developer view with CUDA Unified Memory



- Pool of managed memory that is shared between host and device
- Primarily productivity feature
- Memory copies still happen under the hood

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

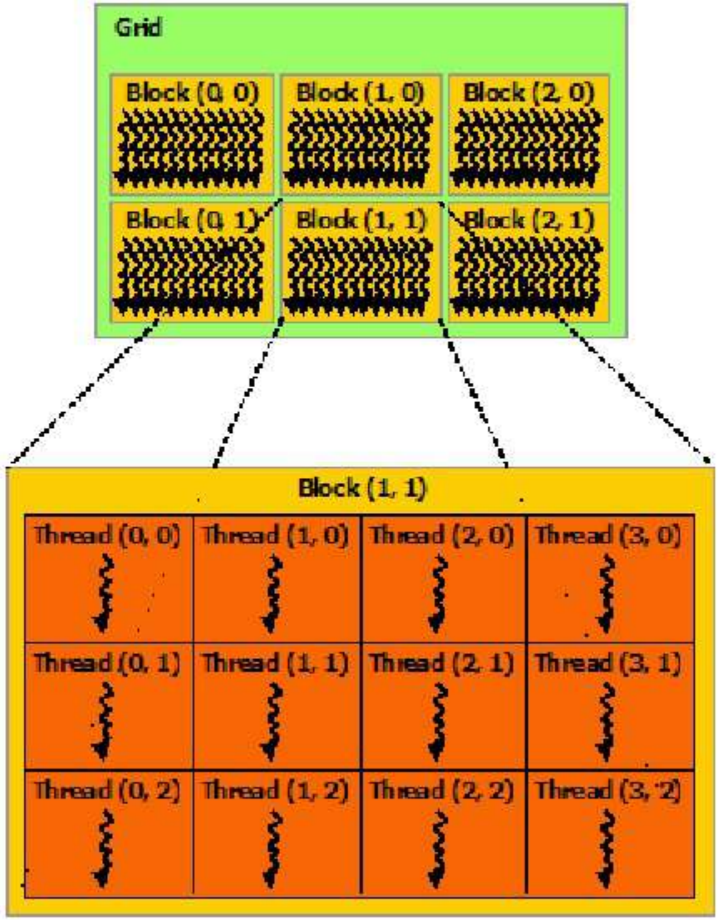
- Grids map to GPUs

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

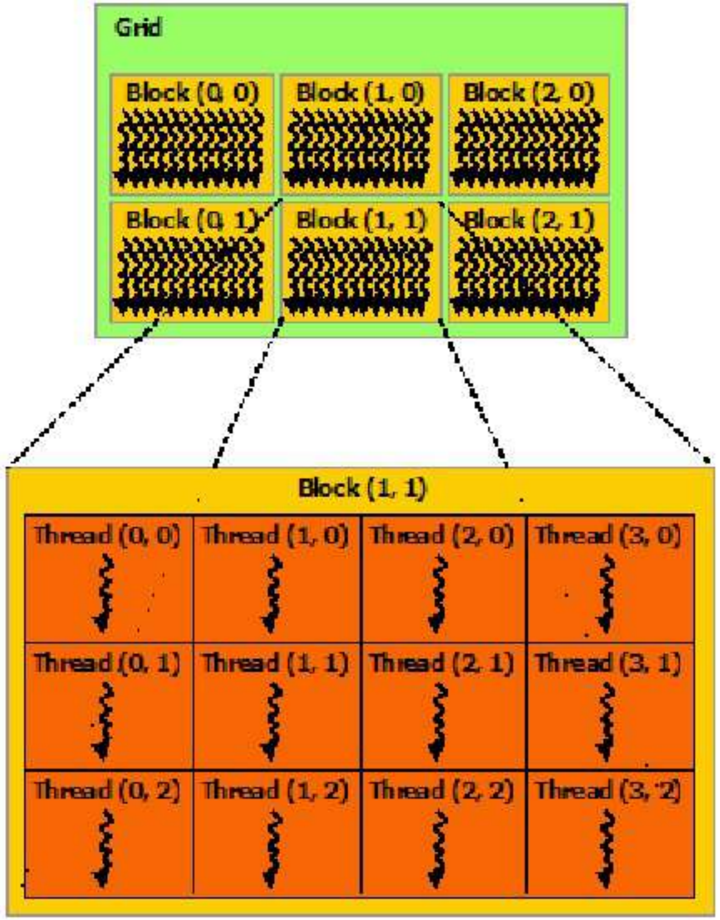
- `gridDim.x, gridDim.y, gridDim.z (unused)`
- `blockIdx.x, blockIdx.y, blockIdx.z`
- Variables that return the grid dimension (number of blocks) and block ID in the x-, y-, and z-axis.

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);
```

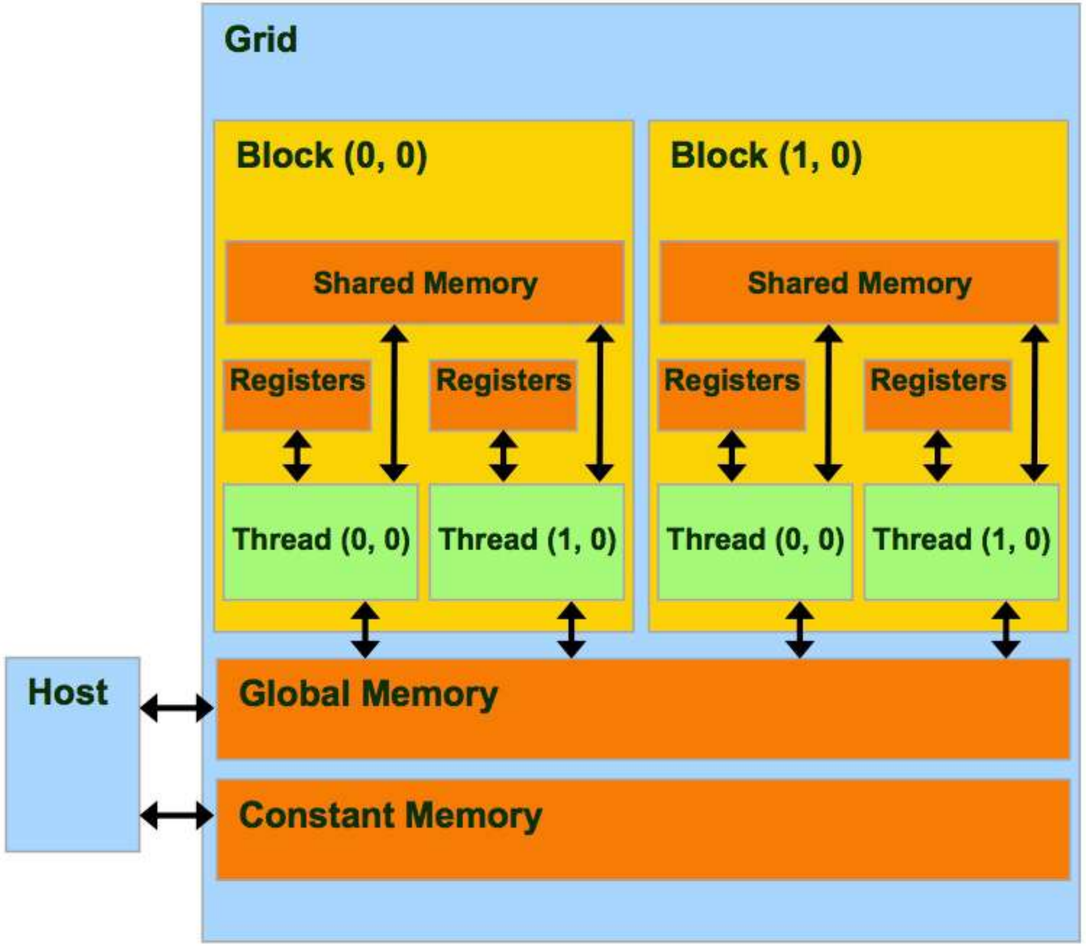
Some CUDA basics

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

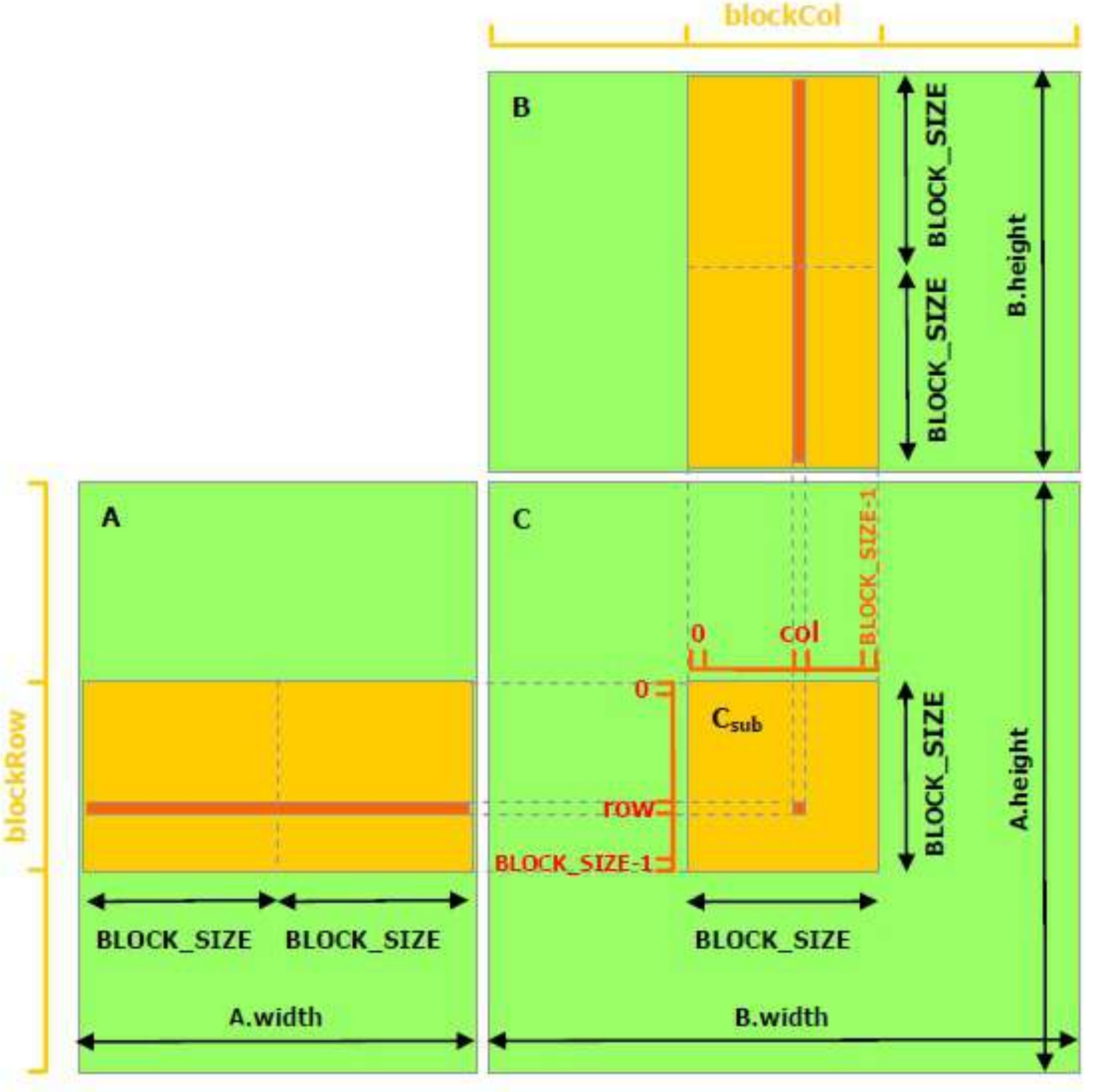
Minimize access to global memory

- 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

CUDA Example: Matrix-matrix multiply kernel



How could we implement a CUDA kernel for this matrix-matrix multiplication?

Simplifications for this example:

- Width and height of result matrix C is multiple of BLOCK_SIZE
- We launch one thread per element C_{ij} of the result matrix
- Each thread computes one dot-product of a row-vector of A with a column-vector of B

Performance critical optimization

- Cache blocking
- Load sub-blocks of arrays A and B into shared memory to avoid multiple reads from slow global memory

Note

- In reality we would use the highly optimized cuBLAS GPU library

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/C++
- Full support by Nvidia (formerly PGI compilers) and Cray compilers on Crays

OpenMP

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

Directive based programming

PGI Community Edition

- See https://developer.nvidia.com/openacc-toolkit
- Community Edition is free
- PGI Fortran / C / C++ compilers
- Support for OpenMP and OpenACC
- pgprof performance profiler
- GPU-enabled libraries
- OpenACC code samples

Note: Can also use Nvidia HPC SDK

Activate on Expanse GPU nodes

```
$> module purge
$> module reset
$> module load pgi
```

- Currently loads version 20.4 by default
- Versions 19.7 and 18.10 also available

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)
...
```

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

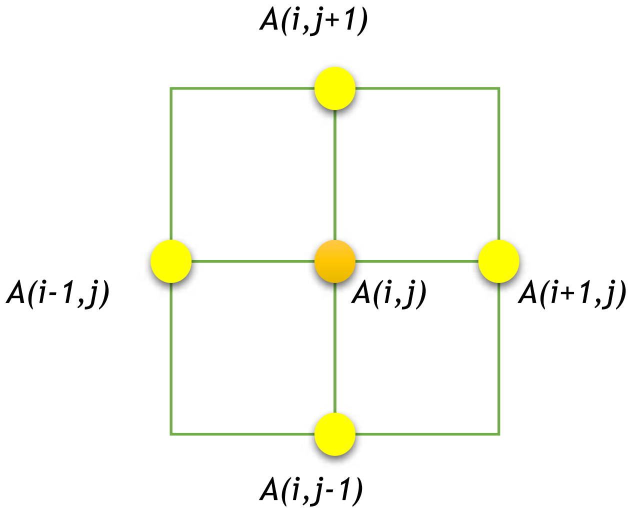
`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to 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`.

OpenACC example: Jacobi iteration

Iteratively converges to correct value (e.g. Temperature),
by computing new values at each point from the average of neighboring points.

- Common, useful algorithm
- Example: Solve Laplace equation in 2D: $\Delta\varphi(x,y)=0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

OpenACC example: Jacobi iteration

```
while ( error > tol && iter < iter_max )
{
  error=0.0;


  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

OpenACC example: Jacobi iteration – first attempt

```
while ( error > tol && iter < iter_max )
{
  error=0.0;

  #pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  #pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

OpenACC example: Jacobi iteration – first attempt

Compiler output

```
pgf90 -acc -Minfo=accel -o jacobi-pgf90-acc-v1.x jacobi-acc-v1.f90
laplace:
     44, Generating implicit copyin(a(0:4095,0:4095)) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(anew(1:4094,1:4094)) [if not already present]
     45, Loop is parallelizable
     46, Loop is parallelizable
         Generating Tesla code
         45, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
             Generating implicit reduction(max:error)
         46, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
     57, Generating implicit copyout(a(1:4094,1:4094)) [if not already present]
         Generating implicit copyin(anew(1:4094,1:4094)) [if not already present]
     58, Loop is parallelizable
     59, Loop is parallelizable
         Generating Tesla code
         58, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         59, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

OpenACC example: Jacobi iteration – first attempt

SDSC Expanse GPU node

CPU: Intel Xeon Gold 6248
(2 x 20 core)

GPU: Nvidia V100
(4 GPUs, using single GPU)

Matrix dimension 4096 x 4096

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	42.7	--
CPU 2 OpenMP threads	21.8	1.96x
CPU 4 OpenMP threads	11.4	3.75x
CPU 8 OpenMP threads	7.4	5.77x
OpenACC GPU	104.9	0.07x FAIL

- Compiler:
  pgf90 20.4-0
- CPU flags:
  -fast -mp [-Minfo=mp]
- GPU flags:
  -acc [-Minfo=accel]

Speedup vs. 1 CPU core

Speedup vs. 8 CPU cores

OpenACC example: Jacobi iteration – first attempt

```
export PGI_ACC_TIME=1    ! Activate profiling, then run again
```

```
Accelerator Kernel Timing data
/server-home1/agoetz/UCSD_Phys244/2017/openacc-samples/laplace-2d/jacobi-acc-v1.f90
  laplace  NVIDIA  devicenum=0
    time(us): 89,612,134
 ..... <snip – some lines cut>
44: data region reached 2000 times
        44: data copyin transfers: 8000
             device time(us): total=22,587,486 max=2,898 min=2,799 avg=2,823
        52: data copyout transfers: 8000
             device time(us): total=20,278,262 max=2,612 min=2,497 avg=2,534
    57: compute region reached 1000 times
        59: kernel launched 1000 times
            grid: [128x1024]  block: [32x4]
             device time(us): total=1,456,273 max=1,465 min=1,452 avg=1,456
            elapsed time(us): total=1,498,877 max=1,524 min=1,492 avg=1,498
    57: data region reached 2000 times
        57: data copyin transfers: 8000
             device time(us): total=22,664,227 max=2,902 min=2,802 avg=2,833
        63: data copyout transfers: 8000
             device time(us): total=20,278,000 max=2,618 min=2,498 avg=2,534
```

22.5 seconds

1.5 seconds

What went wrong?

- We spent all the time with data transfers between host and device

OpenACC example: Jacobi iteration – first attempt

Excessive data transfers

```
while ( error > tol && iter < iter_max )
{
  error=0.0;
```

A, Anew resident on host

Copy

```
#pragma acc kernels
```

A, Anew resident on accelerator

These copies happen every iteration of the outer while loop!

```
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++) {
        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                             A[j-1][i] + A[j+1][i]);
        error = max(error, abs(Anew[j][i] - A[j][i]);
      }
    }
```

A, Anew resident on accelerator

Copy

A, Anew resident on host

```
    ...
}
```

OpenACC example: Jacobi iteration – second attempt

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

OpenACC example: Jacobi iteration – second attempt

SDSC Expanse GPU node

CPU: Intel Xeon Gold 6248
(2 x 20 core)

GPU: Nvidia V100
(4 GPUs, using single GPU)

Matrix dimension 4096 x 4096

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	42.7	--
CPU 2 OpenMP threads	21.8	1.96x
CPU 4 OpenMP threads	11.4	3.75x
CPU 8 OpenMP threads	7.4	5.77x
OpenACC GPU	1.1	6.73x

- Compiler:
  pgf90 20.4-0
- CPU flags:
  -fast -mp [-Minfo=mp]
- GPU flags:
  -acc [-Minfo=accel]

Speedup vs. 1 CPU core

Speedup vs. 8 CPU cores

More OpenACC

Finding and exploiting parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be <u>independent</u> of each other
  - To help compiler: **`restrict`** keyword (C), **`independent`** clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.

More OpenACC

Tips and Tricks

- (PGI) Use time option to learn where time is being spent

  Compiler flag: `-ta=nvidia,time`
  Environment variable: `PGI_ACC_TIME=1`

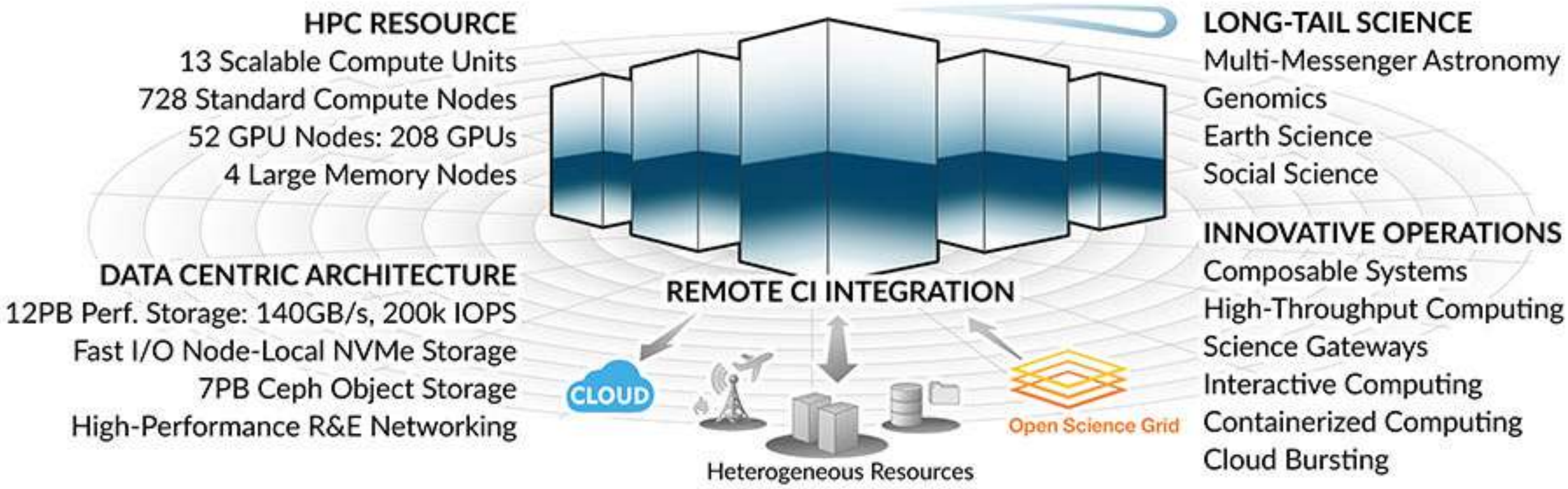
- Eliminate pointer arithmetic
- Inline function calls in directives regions

  (PGI): `-Minline` or `-Minline=levels:N`

- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro

SDSC Expanse

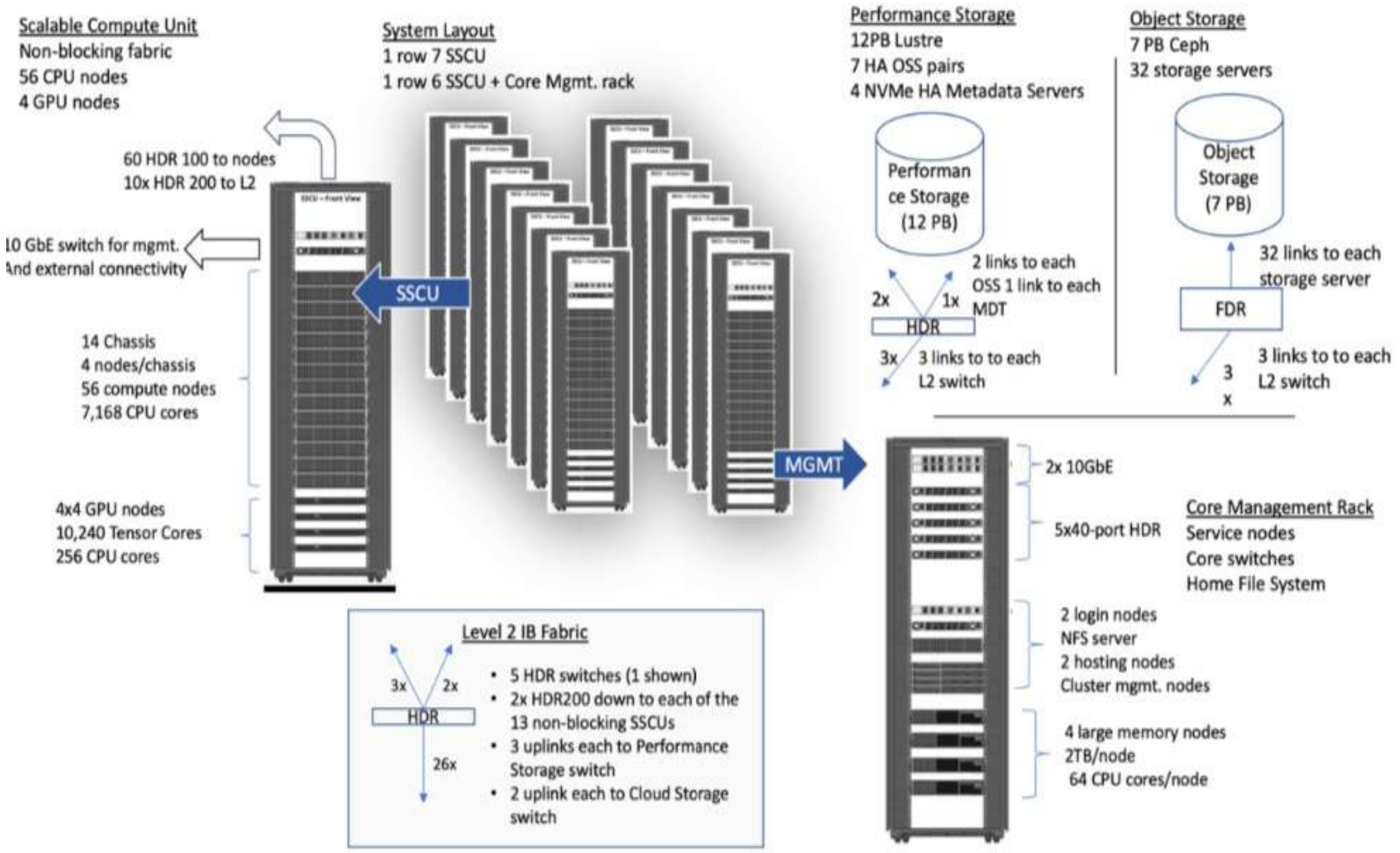
Launched in Fall 2020



Expanse Heterogeneous Architecture

System Summary

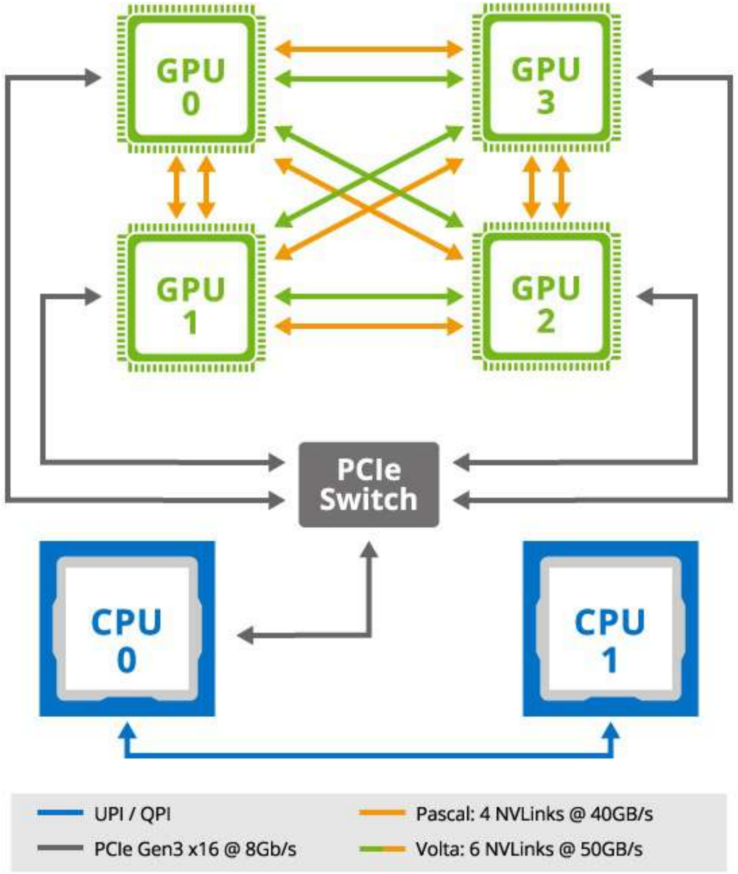
- 13 SDSC Scalable Compute Units (SSCU)
- 728 Standard Compute Nodes
- 93,184 Compute Cores
- 200 TB DDR4 Memory
- 52x 4-way GPU Nodes w/NVLINK
- 208 V100 GPUs
- 4x 2TB Large Memory Nodes
- HDR 100 non-blocking Fabric
- 12 PB Lustre High Performance Storage
- 7 PB Ceph Object Storage
- 1.2 PB on-node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



SDSC Expanse GPU nodes

52 GPU nodes

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



User guide: https://www.sdsc.edu/support/user_guides/expanse.html

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=80G --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 93GB RAM per GPU, otherwise you will be charged for proportionally more time.

- Purge, then load GPU related modules

```
module purge
module reset
module load sdsc
```

```
# Either Load CUDA Toolkit and PGI compiler
module load cuda
module load pgi

# Or load Nvidia HPC SDK
module load nvhpc

# Note: CUDA samples on Expanse available
#       only for CUDA Toolkit 10.2
```

SDSC Comet 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 (output is for NVHPC)

```
[agoetz@exp-8-59 ~]$ pgcc --version

pgcc (aka nvc) 22.2-0 64-bit target on x86-64 Linux -tp skylake-avx512
PGI Compilers and Tools
Copyright (c) 2022, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.
```

SDSC Expanse GPU nodes

- Interactive access to GPU nodes

```
[agoetz@login01 ~] srun --partition=gpu-shared --nodes=1 --gpus=1 \
                          --ntasks-per-node=1 --cpus-per-task=10 --mem=80GB \
                          --time=00:30:00 --pty --wait=0 /bin/bash
```

- 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 Comet 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 other jobs running on the GPU
- 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 ~]$ module load cuda10.2/toolkit
[agoetz@exp-8-59 ~]$ cp -r /cm/shared/apps/cuda10.2/sdk/10.2.89 ./CUDA_samples
```

- Explore CUDA Toolkit samples – great resource!

```
[agoetz@exp-8-59 ~]$ cd CUDA_samples/
[agoetz@exp-8-59 CUDA_samples]$ ls
0_Simple     3_Imaging      6_Advanced        common     opencl
1_Utilities  4_Finance      7_CUDALibraries   EULA.txt   verify_cuda10.2.sh
2_Graphics   5_Simulations  bin               Makefile   verify_opencl.sh
```

- 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 Comet 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_examples]$ cd 1_Utilities/deviceQuery
[agoetz@exp-1-57 CUDA_examples]$ 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/0_Simple/
[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 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
```

Questions?