

A photograph of the San Diego Supercomputer Center building at night. The building is a modern, multi-story structure with a prominent cantilevered upper floor. It is illuminated from within, and the surrounding area is lit by streetlights. The sky is a deep blue.

# GPU Computing and Programming

Andreas W Götz

San Diego Supercomputer Center  
University of California, San Diego

Friday, January 24, 2020, 1:00 pm to 3:00 pm, PDT

# Training overview

## We will cover the following topics

- GPU hardware overview
- GPU accelerated software examples
- GPU enabled libraries
- CUDA C programming basics
- OpenACC introduction
- Accessing GPU nodes and running GPU jobs on SDSC Comet

# 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





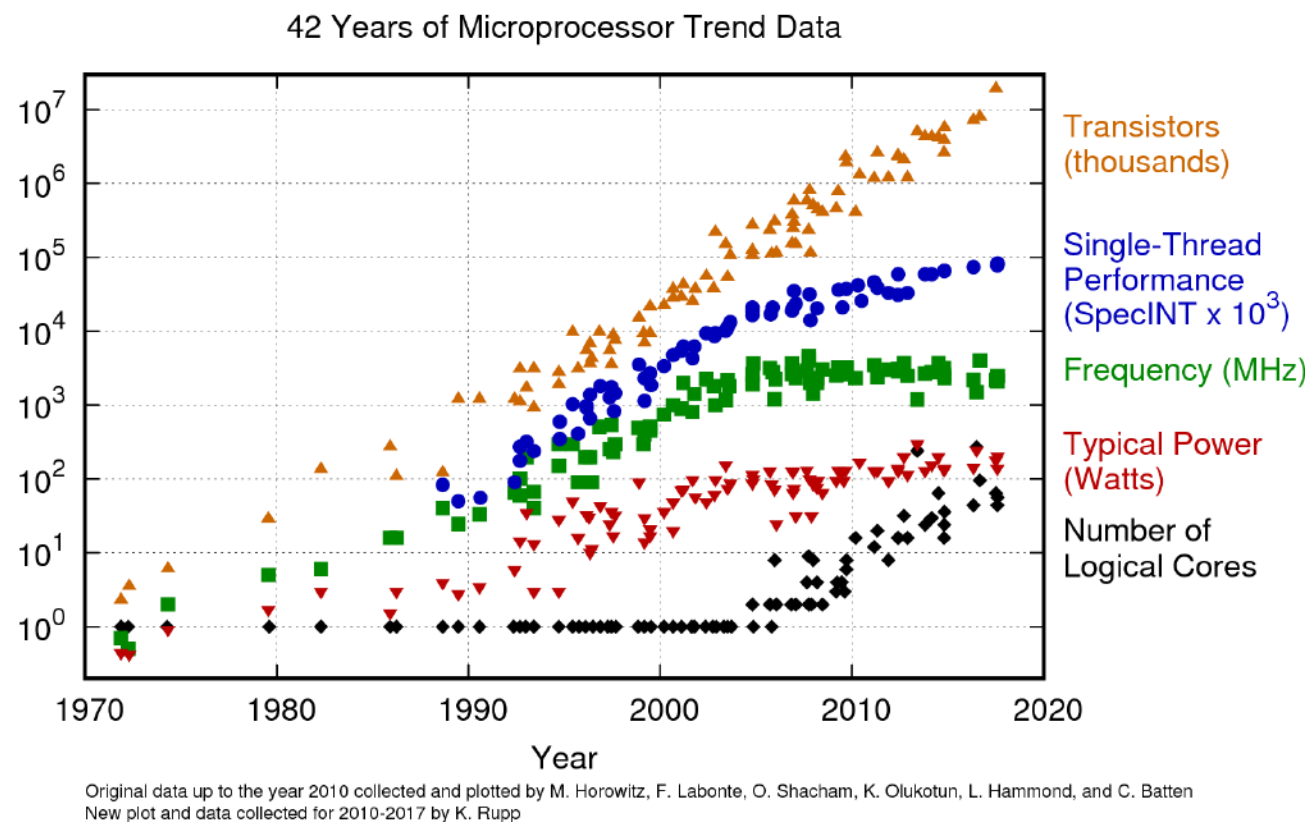
# Why is there such an interest in GPUs?

## Moore's law

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

## Trends since mid 2000s

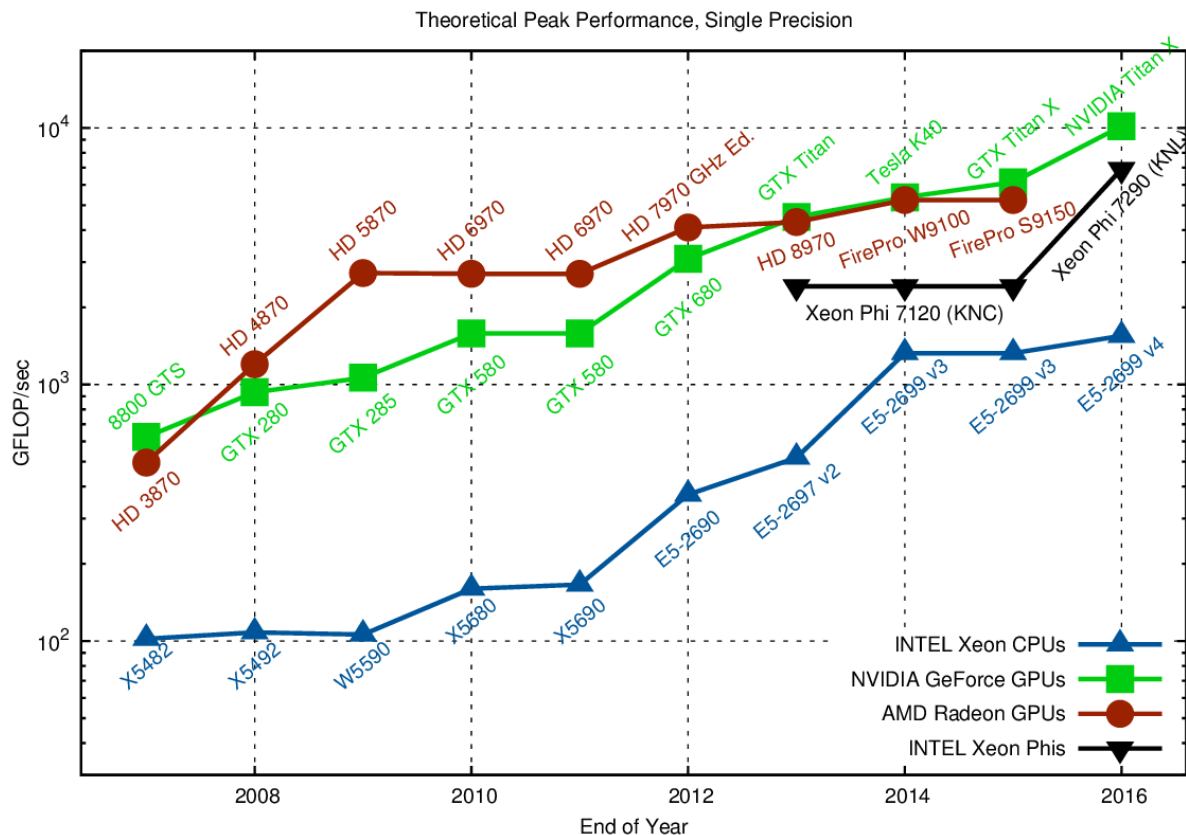
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.



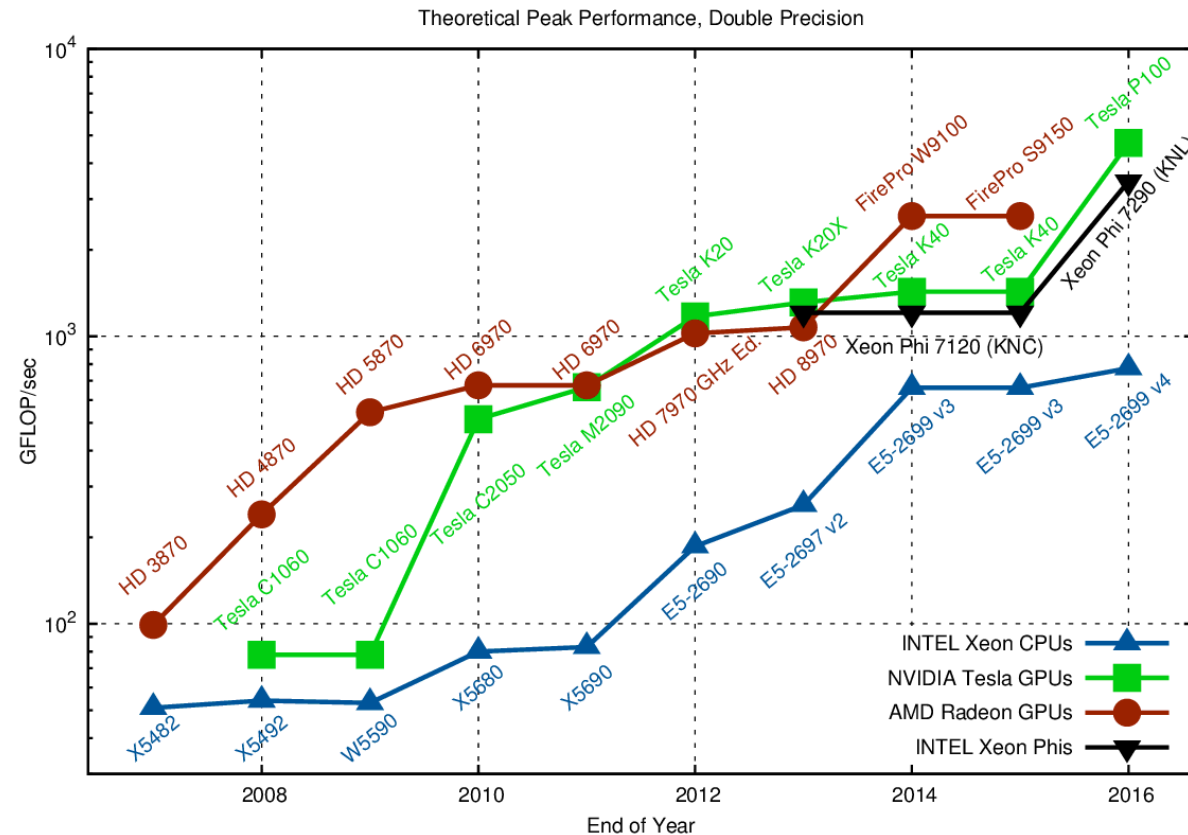
Source:

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

# Why is there such an interest in GPUs?



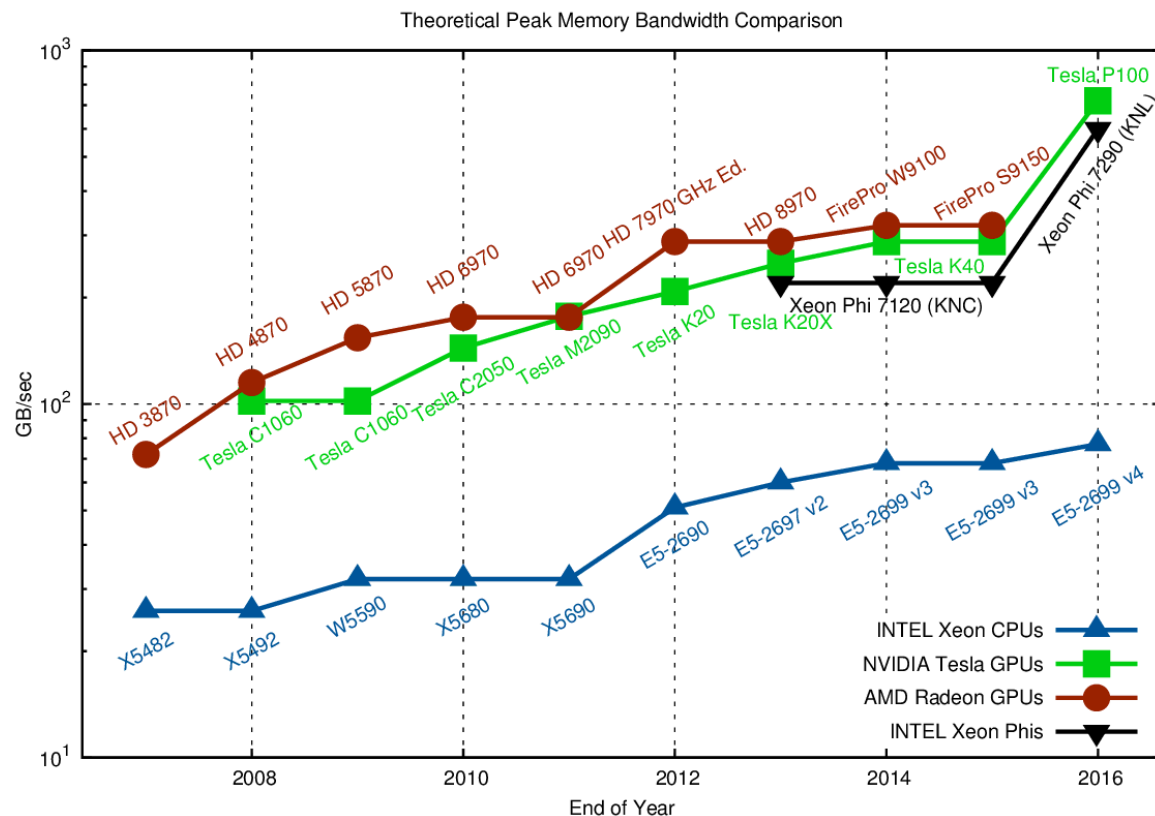
- GPUs offer significantly higher 32-bit floating point performance than CPUs.



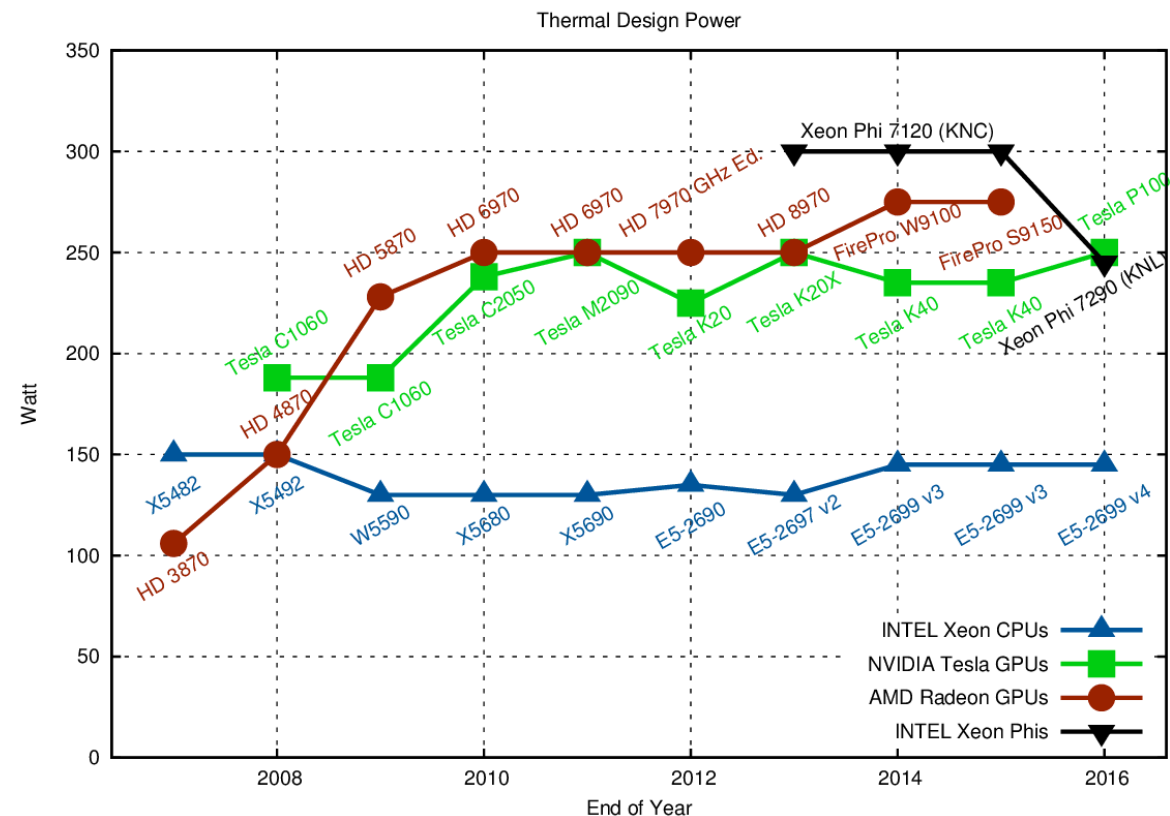
- Datacenter GPUs also offer significantly higher 64-bit floating point performance than CPUs.

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 Comet

- 2.8 PFLOP/sec aggregate
- 36 nodes 2 x Nvidia K80  
5.5 TFLOP/sec DP, 16.4 TFLOP/sec SP (each node)
- 36 nodes 4 x Nvidia P100  
18.8 TFLOP/sec DP, 37.2 TFLOP/sec SP (each node)
- Cost – \$25 Million USD (\$14 Million Hardware)



## DIY 4 x Nvidia RTX 2080 box

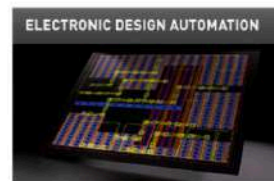
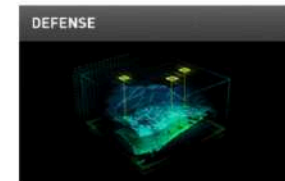
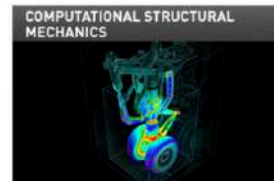
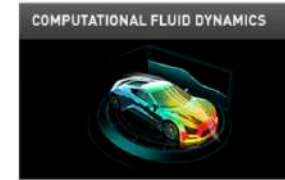
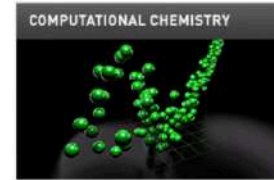
- 1.3 TFLOP/sec DP
- 40.0 TFLOP/sec SP
- Cost – ~ \$5 Thousand USD



# GPU accelerated software

## Examples from virtually any field

- Exhaustive 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...



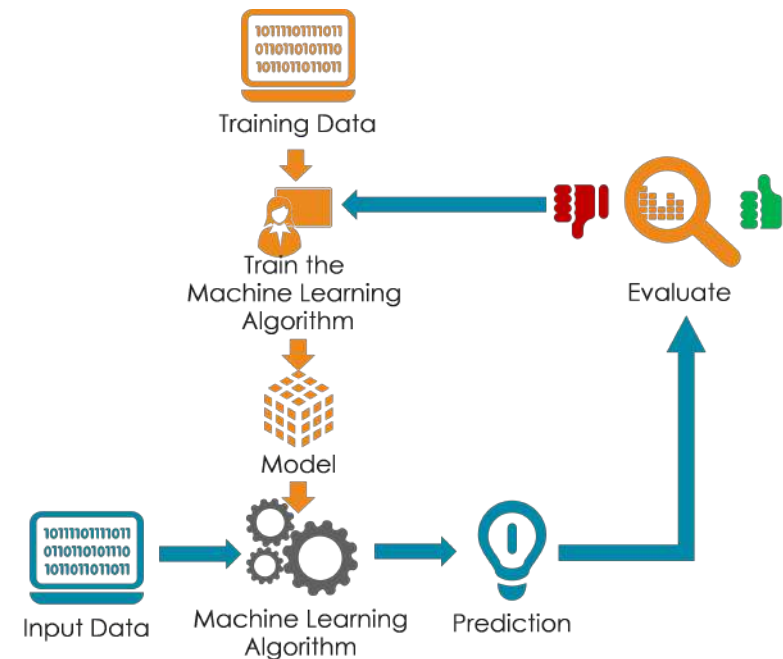
# Machine learning and GPUs

## Machine learning

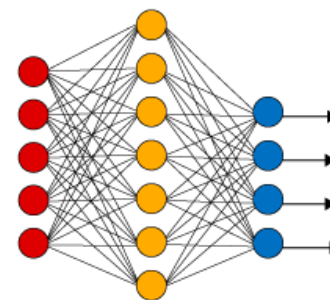
- Estimate / predictive model based on reference data.
- Many different methods and algorithms.
- GPUs are particularly well suited for deep learning workloads

## Deep learning

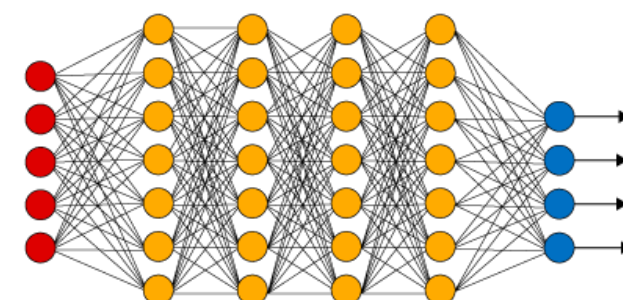
- Neural networks with many hidden layers.
- Tensor operations (matrix multiplications).
- GPUs are very efficient at these (4x4 matrix algebra is used in 3D graphics)
- Half-precision arithmetic can be used for many ML applications, at least for inference.
- ML frameworks provide GPU support (E.g. PyTorch,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



Simple Neural Network

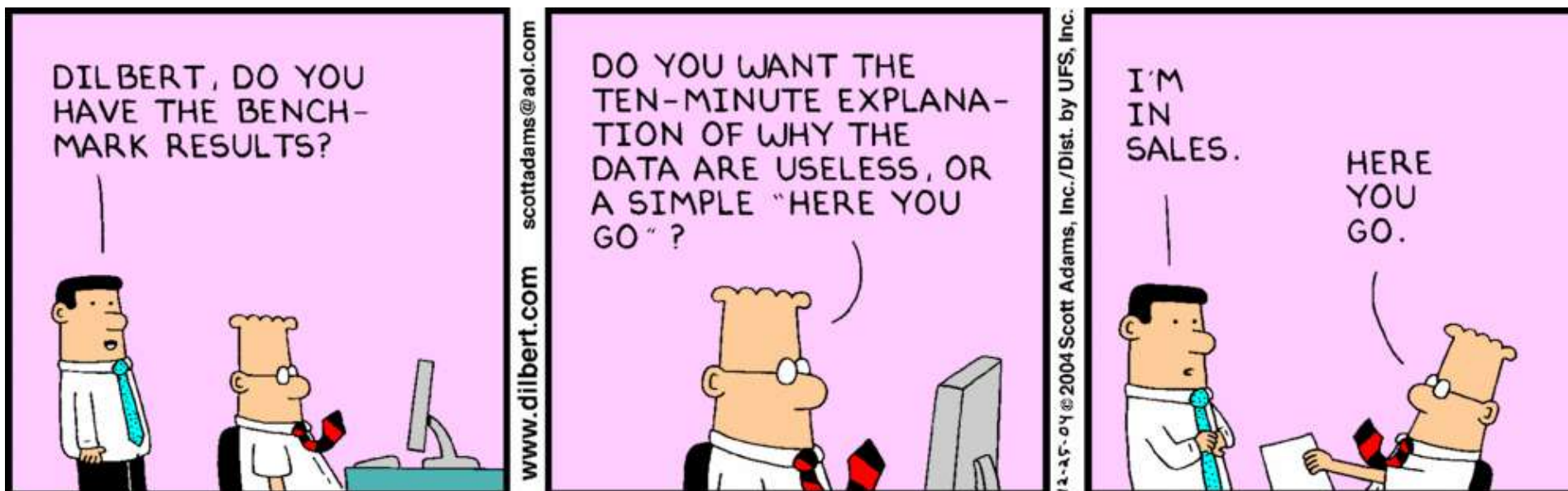


Deep Learning Neural Network



● Input Layer    ● Hidden Layer    ● Output Layer

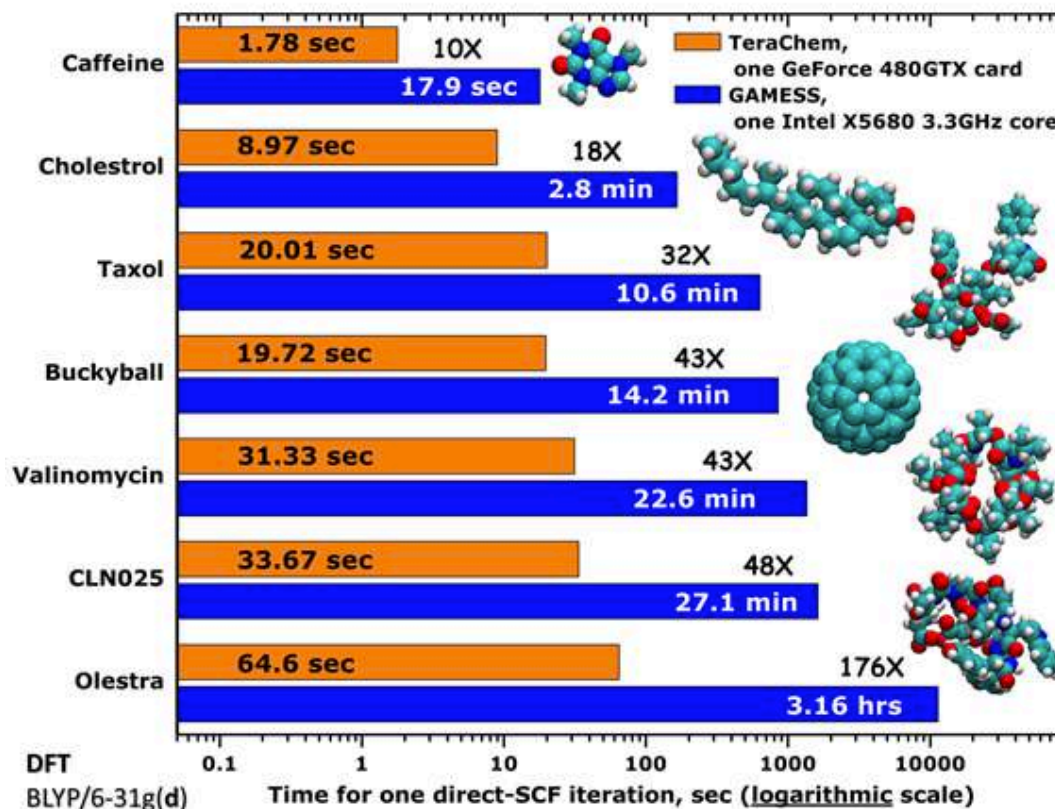
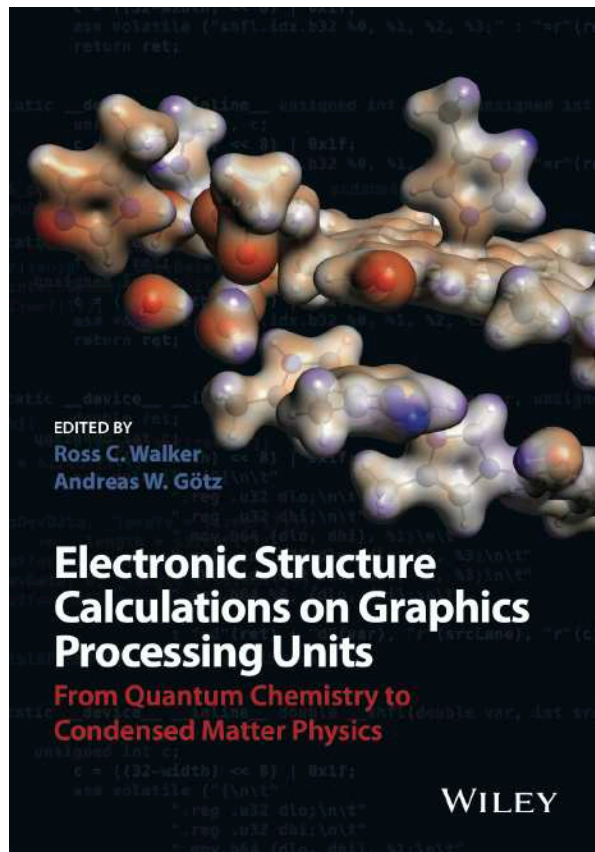
# Benchmark examples



# Benchmark examples

## Quantum chemistry

- Compute molecular properties from quantum mechanics (TeraChem code)

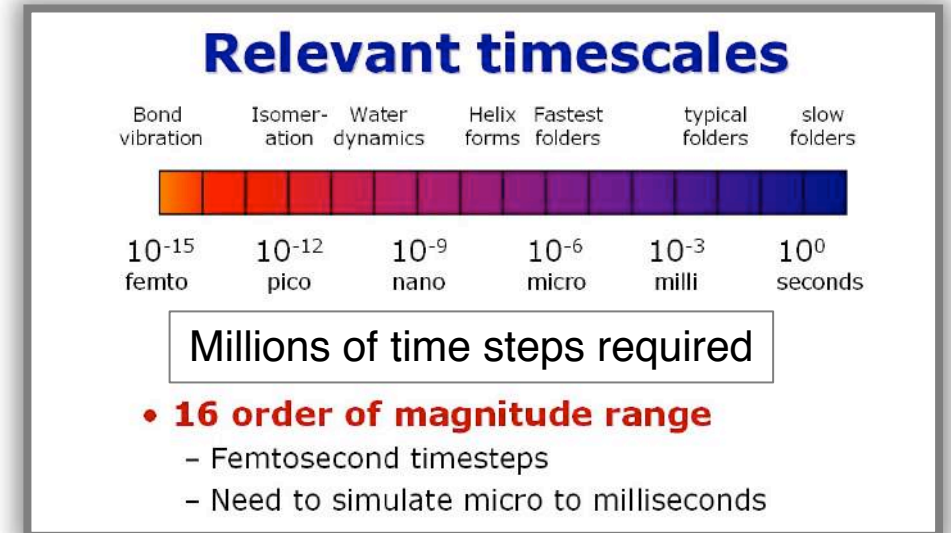
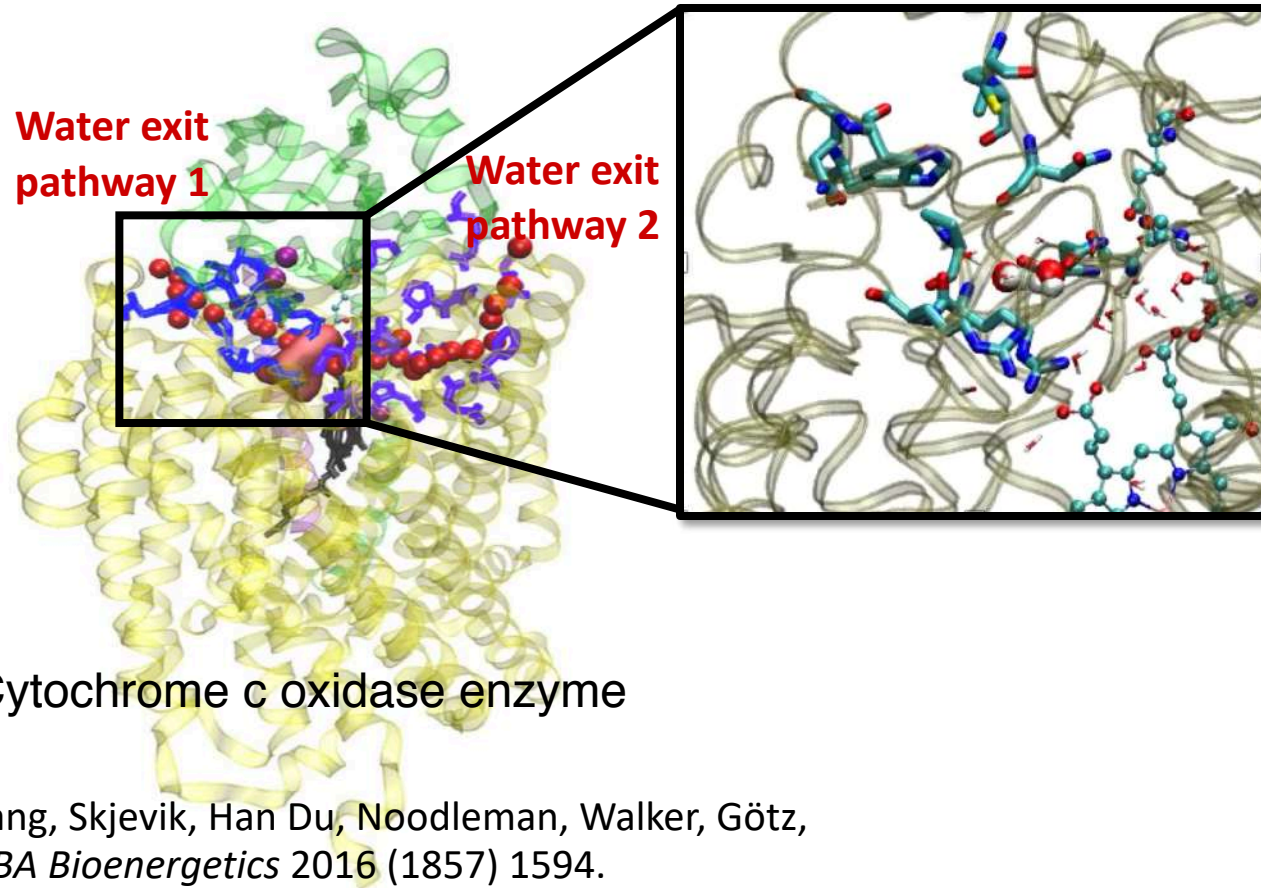




# Benchmark examples

## Molecular dynamics

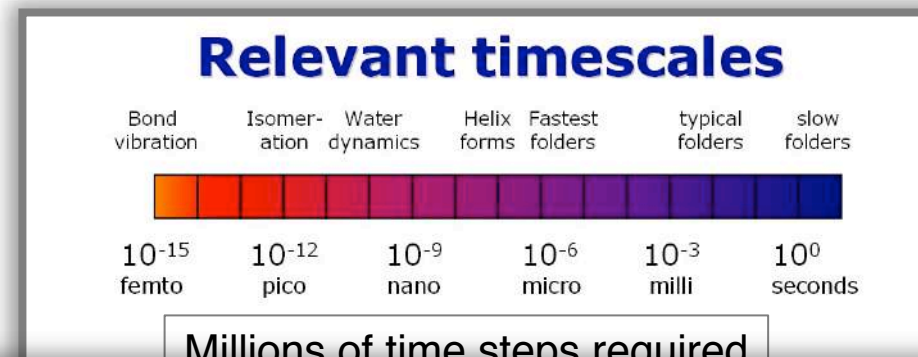
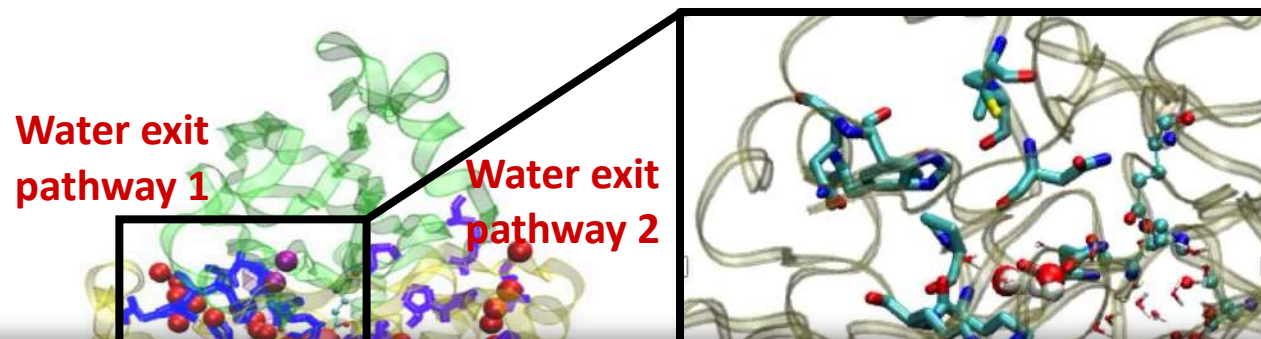
- Amber code: Atomistic simulations of condensed phase biomolecular systems



# Benchmark examples

## Molecular dynamics

- Amber code: Atomistic simulations of condensed phase biomolecular systems

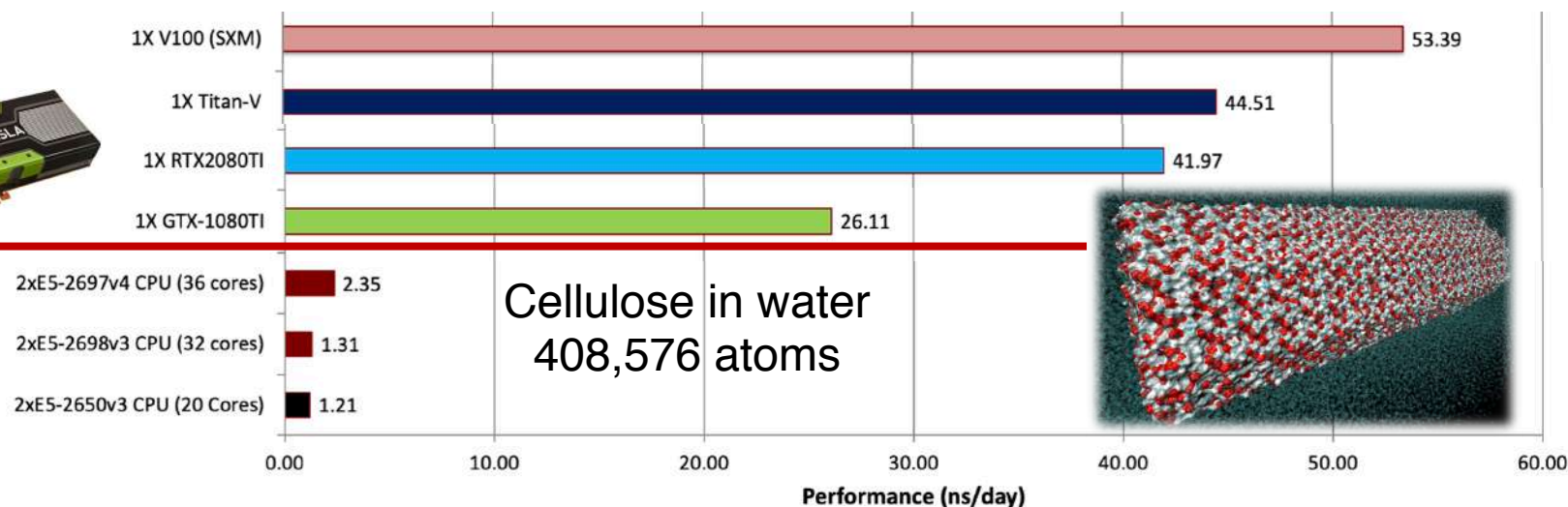


## Amber 18 molecular dynamics software

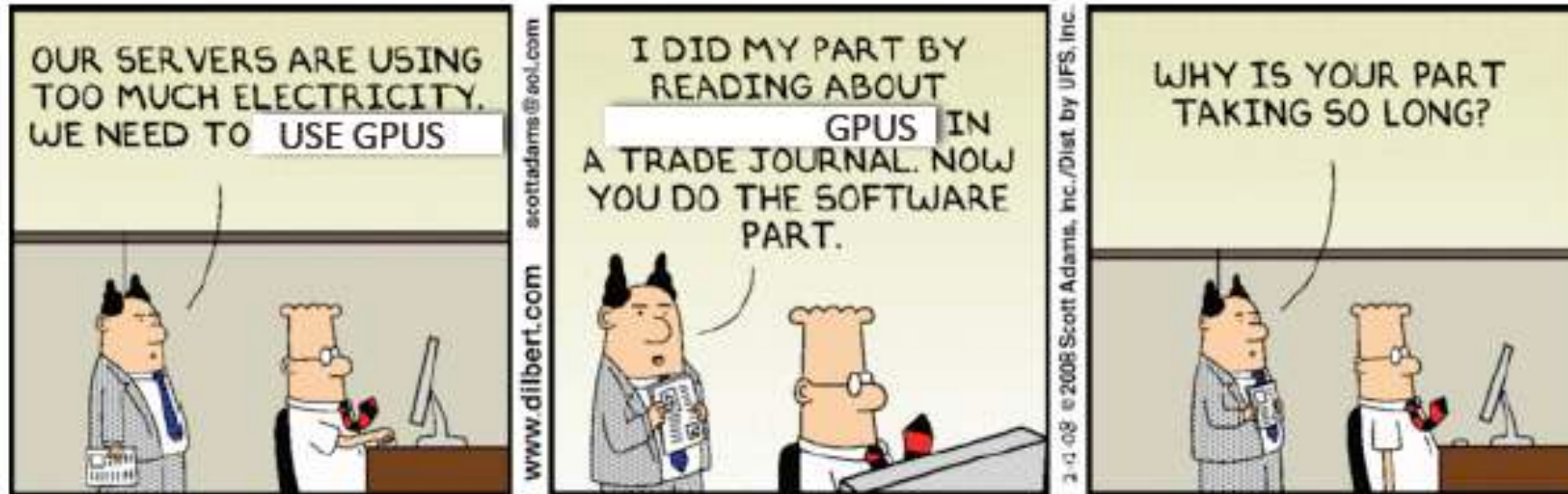
Götz, Williamson, Xu, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

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

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



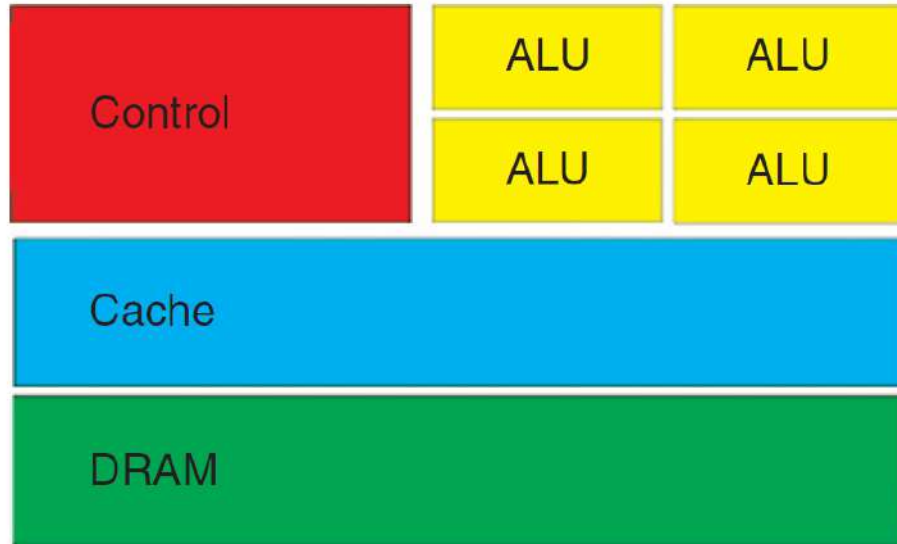
# What's the catch?



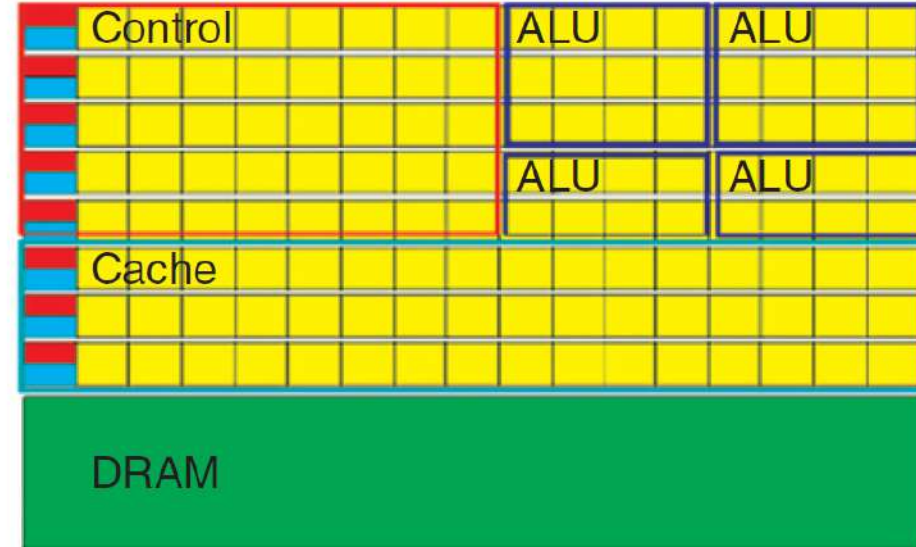


# GPU vs CPU architecture

(a) CPU



(b) GPU



## CPU

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

## GPU

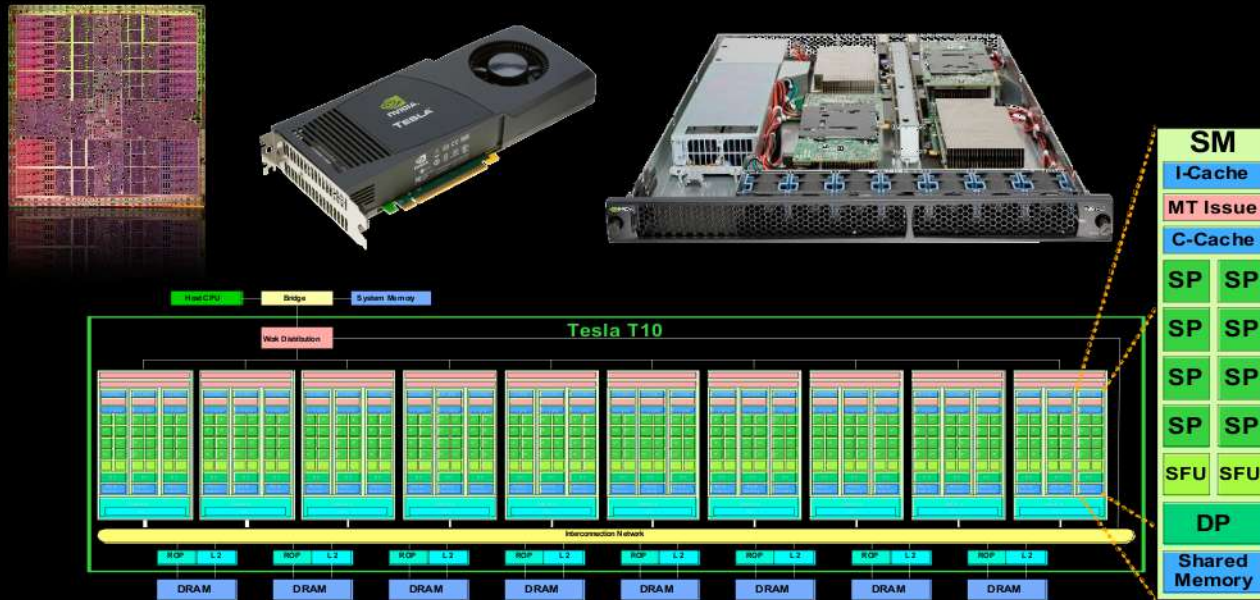
- Thousands of simplistic compute cores (packaged into a few multiprocessors)
- Operate in lock-step
- Vectorized loads/stores to memory
- Need to manage memory hierarchy



# GPU architecture

## CUDA Computing with Tesla T10

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



© NVIDIA Corporation 2008

## 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

# A brief history of GPU computing

- **2003 - First attempts to use GPUs for general computing.**
  - Programmed as graphics primitives (heroic)
  - problems had to be expressed in terms of vertex coordinates, textures and shader programs.
  - Hardware lacking certain ‘features’ – No random reads or writes etc.
- **2004 – ‘Brook’ programming language for GPUs.**
- **2007 – NVIDIA announce CUDA at SC07**
  - Release GPUs with specific ‘computational’ features.
- **2008 – OpenCL language ratified.**
  - Mainly aimed at embedded devices but has features for GPU computation.
- **2010 – CUDA Fortran language defined.**
- **2011 – OpenACC compiler directive language ratified.**
  - Provides OpenMP like directives for use with GPUs.

# GPU programming ‘languages’

## Brook

- First widely adopted programming model for general purpose GPU (GPGPU) programming
- Extends C with data-parallel constructs
- Concepts such as streams, kernels, reduction operators
- Easier to write AND faster than hand-tuned GPU code

## CUDA – based on ideas of Brook

- Solution to run C seamlessly on GPUs (Proprietary, NVIDIA GPUs only)
- CUDA Toolkit contains compiler, math libraries, debugging and profiling tools
- Lots of code samples, programming guides and other documentation available
- De facto standard for high-performance code

## OpenCL

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

# GPU programming ‘languages’

## CUDA Fortran

- Supports CUDA extensions for Fortran.
- Implemented in Portland Group Compilers.

## OpenACC

- OpenMP-like compiler directives language for C/C++ and Fortran.
- Designed to make porting to GPUs easy and quick.
- Full support by 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

- Version 4.x includes accelerator and vectorization directives
- Works well with Intel Xeon Phi (and AVX512), not mature for GPUs, will not discuss here



# Common GPU programming

- GPU Kernels – executed on GPU but controlled from host CPU.
- Memory can be copied to and from CPU and GPU memory (synchronization is important)

```
__global__ void kernel( int *a, int dimx, int dimy )  
{  
    int ix    = blockIdx.x*blockDim.x + threadIdx.x;  
    int iy    = blockIdx.y*blockDim.y + threadIdx.y;  
    int idx = iy*dimx + ix;  
  
    a[idx]    = a[idx]+1;  
}
```

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix    = blockIdx.x*blockDim.x + threadIdx.x;
    int iy    = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]    = a[idx]+1;
}
```



## CUDA kernel

## C program, calls CUDA kernel

```
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
}
```

```
cudaMemset( d_a, 0, num_bytes );

dim3 grid, block;
block.x = 4;
block.y = 4;
grid.x  = dimx / block.x;
grid.y  = dimy / block.y;

kernel<<<grid, block>>>( d_a, dimx, dimy );

cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

for(int row=0; row<dimy; row++)
{
    for(int col=0; col<dimx; col++)
        printf("%d ", h_a[row*dimx+col] );
    printf("\n");
}

free( h_a );
cudaFree( d_a );

return 0;
}
```

# 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

## Memory hierarchy needs to be explicitly managed

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

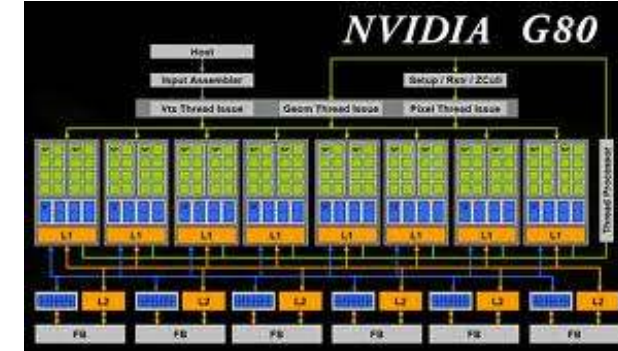
## Different hardware vendors work in different ways

- Nvidia vs AMD

# Hardware complexities

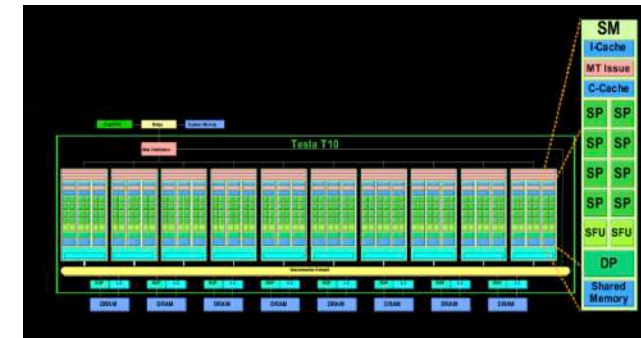
## C870 – Nov 2006

- First 'programmable' Card
- Single Precision Only
- 1.5GB RAM



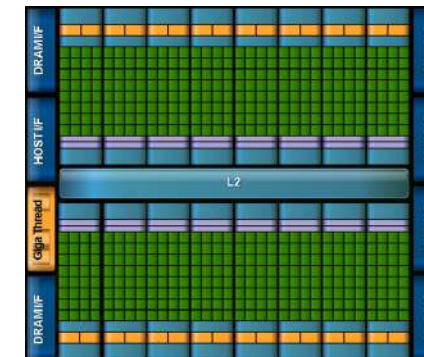
## C1060 – Jun 2008

- DP / SP = 1 / 8
- 4GB RAM



## C2050 – Nov 2010

- DP / SP = 1 / 2
- 3GB RAM





# Hardware complexities

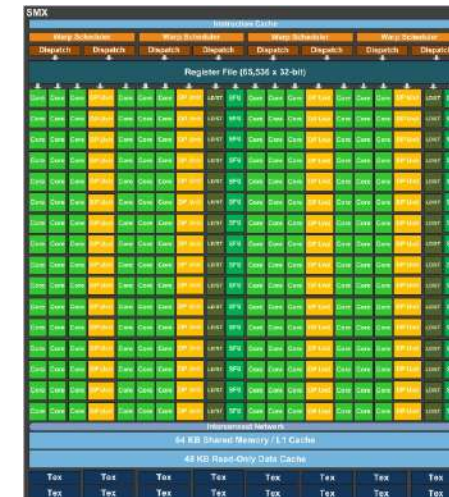
## K10 – Jun 2012

- DP / SP = 1 / 24
- 2 GPUs on 1 board
- 4GB RAM per GPU



## K20 – Nov 2012

- DP / SP = 1 / 3
- 5GB RAM



## K80 – Nov 2014

- DP / SP = 1 / 3
- 2 GPUs per board
- 2 x 12GB RAM

## K40 – Nov 2013

- DP / SP = 1 / 3
- 12GB RAM

# Hardware complexities

## M40 – Nov 2015

- DP / SP = 1 / 3
- 12 or 24GB RAM



## P100 – Q2 2016

- DP / SP = 1 / 2
- 16GB RAM



## V100 – Q2 2017

- DP / SP = 1 / 2
- 16 or 32GB RAM

	C2050	K10	K20	K40	K80	M40	P100	V100
#Multi Proc	14	8 (x2)	13	15	13 (x2)	24	56	80
SP Cores per MP	32	192	192	192	192	128	64	64
#Cores	448	1,536 (x2)	2,496	2,880	2,496 (x2)	3,072	3,584	5,120
Warp Size	32	32	32	32	32	32	32	32
DP Gflop/s	515	95 (x2)	1,170	1,680	1,455 (x2)	213	4,763	7,066

# Nvidia GPU models

## Nvidia compute capabilities determine features available on Nvidia GPUs

- E.g. double precision support since version 1.3

### Hardware Version 3.0 / 3.5 (Kepler I / Kepler II)

- Tesla K20 / K20X / K40 / K80
- Tesla K10 / K8
- GTX-Titan / Titan-Black / Titan-Z
- GTX770 / 780 / 780Ti
- GTX670 / 680 / 690
- Quadro cards supporting SM3.0 or 3.5

### Hardware Version 2.0 (Fermi)

- Tesla M2090
- Tesla C2050/C2070/C2075 (and M variants)
- GTX560 / 570 / 580 / 590
- GTX465 / 470 / 480
- Quadro cards supporting SM2.0

### Hardware Version 7.0 (Volta V100)

- Titan-V
- V100

### Hardware Version 6.1 (Pascal GP102/104)

- Titan-XP [aka Pascal Titan-X]
- GTX-1080TI / 1080 / 1070 / 1060
- Quadro P6000 / P5000
- P4 / P40

### Hardware Version 6.0 (Pascal P100/DGX-1)

- Quadro GP100 (with optional NVLink)
- P100 12GB / P100 16GB / DGX-1

### Hardware Version 5.0 / 5.5 (Maxwell)

- M4, M40, M60
- GTX-Titan-X
- GTX970 / 980 / 980 Ti
- Quadro cards supporting SM5.0 or 5.5

# 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 where not specifically needed.

# 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

## OpenACC

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

## OpenMP

- Version 4.x includes accelerator and vectorization directives
- Works well with Intel Xeon Phi (and AVX512), not mature for GPUs



# Nvidia GPU computing universe

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series		Quadro RTX Series	Tesla T Series	
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier				Tesla V Series	
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series		Quadro P Series	Tesla P Series	
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series		Quadro M Series	Tesla M Series	
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series		Quadro K Series	Tesla K Series	
	EMBEDDED	CONSUMER DESKTOP, LAPTOP		PROFESSIONAL WORKSTATION	DATA CENTER	

Source: CUDA C programming guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

# Nvidia CUDA Toolkit

Obtain from <https://nvidia.com/getcuda>

## Compiler

- CUDA compiler (nvcc)

## Development Tools

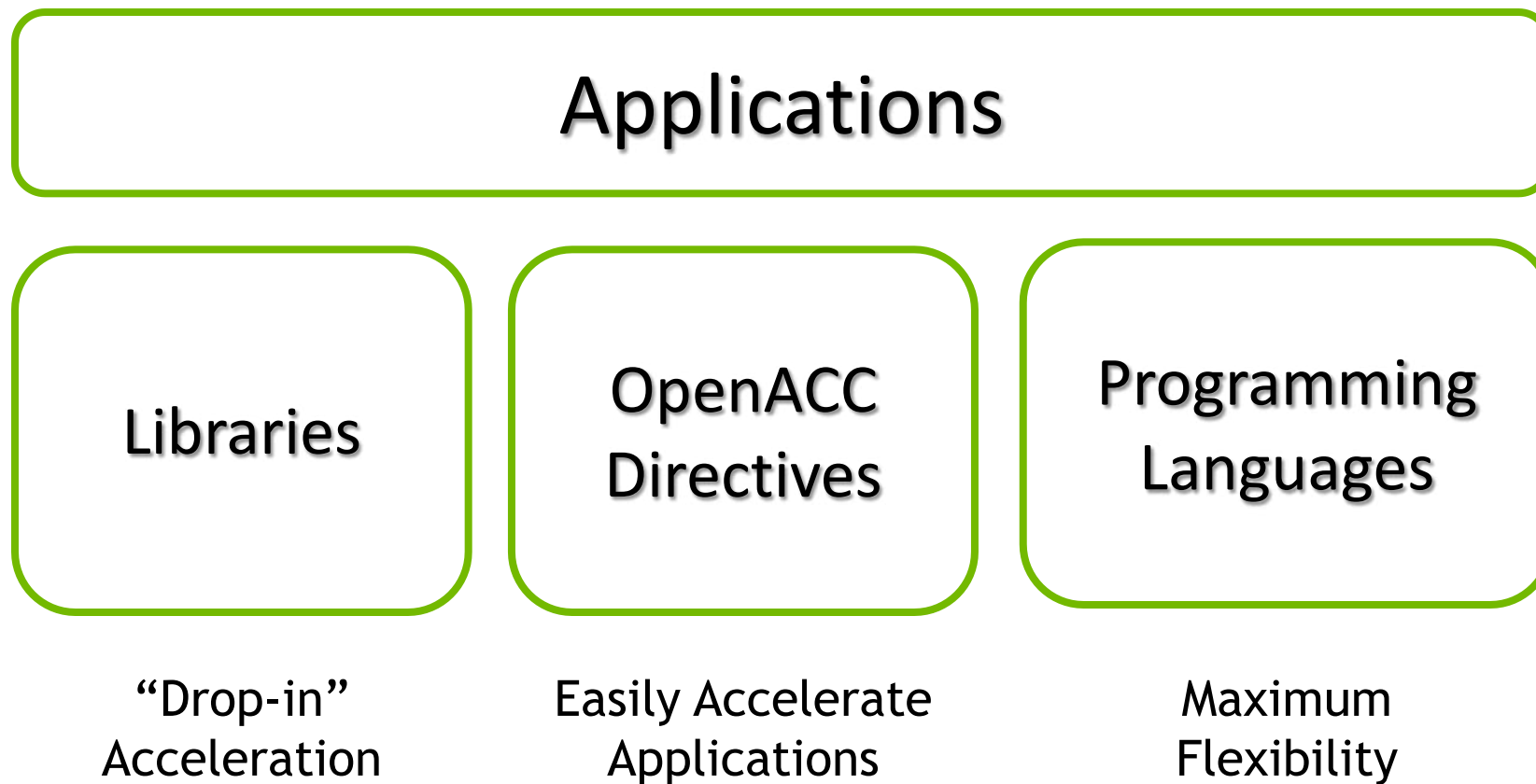
- Debugger (CUDA-gdbm CUDA-memcheck)
- Profiler (nvprof, nvvp)
- Nsight IDE for Eclipse and Visual Studio

## Libraries

- cuBLAS, cuFFT, cuRAND, cuSPARSE, cuSolver, NPP, cuDNN, Thrust, CUDA Math Library, cuDNN

## CUDA code samples

# 3 ways to use GPUs



# 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)



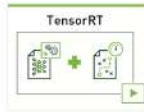
# GPU accelerated libraries

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

## Deep Learning Libraries



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



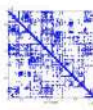
cuBLAS

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

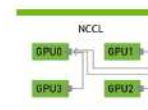
GPU-accelerated library for image and signal processing



NVIDIA Codec SDK

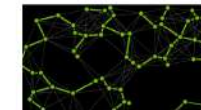
High-performance APIs and tools for hardware accelerated video encode and decode

## Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics



Thrust

GPU-accelerated library of parallel algorithms and data structures

## Partner Libraries



... 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, d_x, 1, d_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

# 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 Toolkit (free)

- Contains CUDA C compiler, math libraries, debugging and profiling tools

## CUDA Fortran

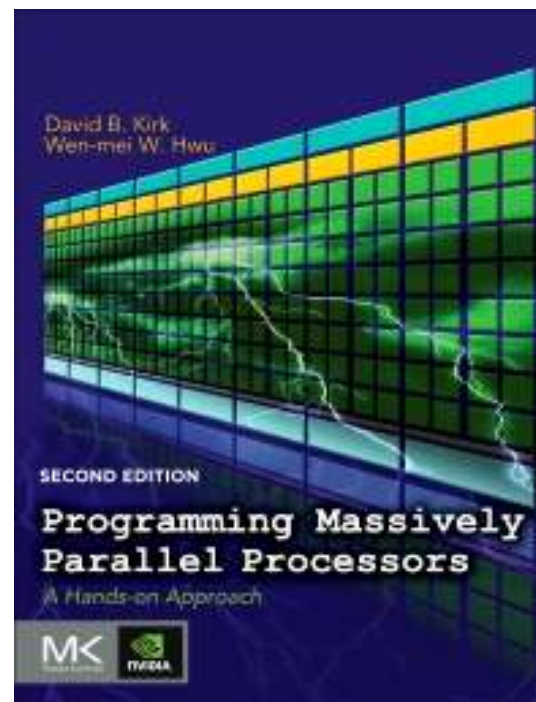
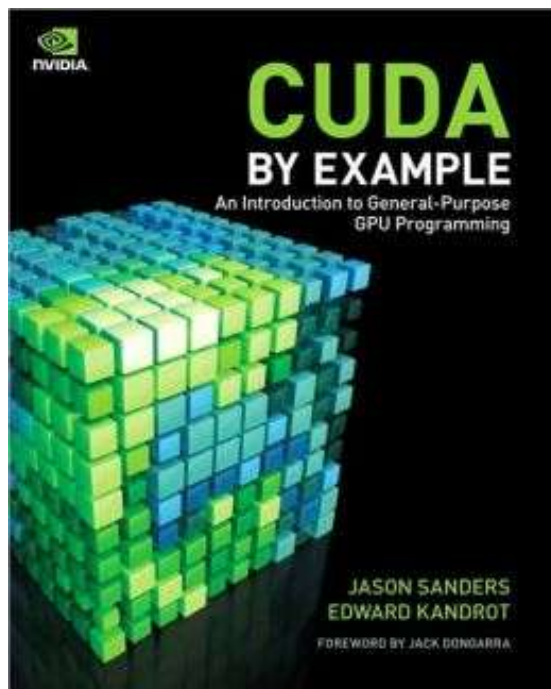
- Supports CUDA extensions in Fortran, developed by Portland Group Inc (PGI)
- Available in the PGI Fortran Compiler
- PGI is now part of Nvidia

# Nvidia CUDA C basics

## CUDA programming guide

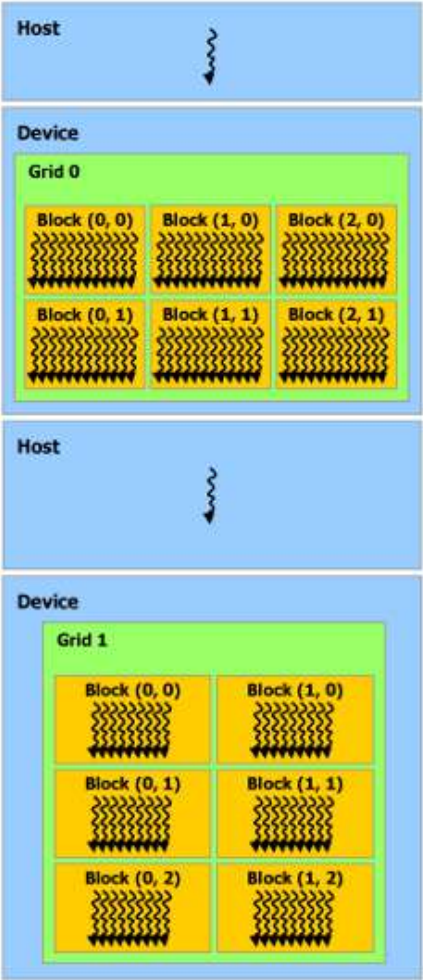
- See <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

## Good books to get started





# Heterogeneous Computing

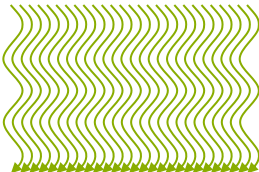
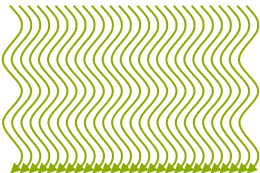


*serial code*

*parallel code*

*serial code*

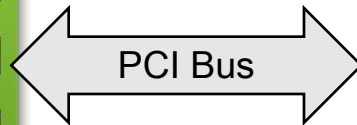
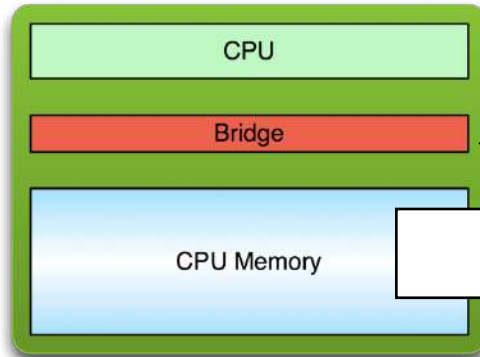
*parallel code*



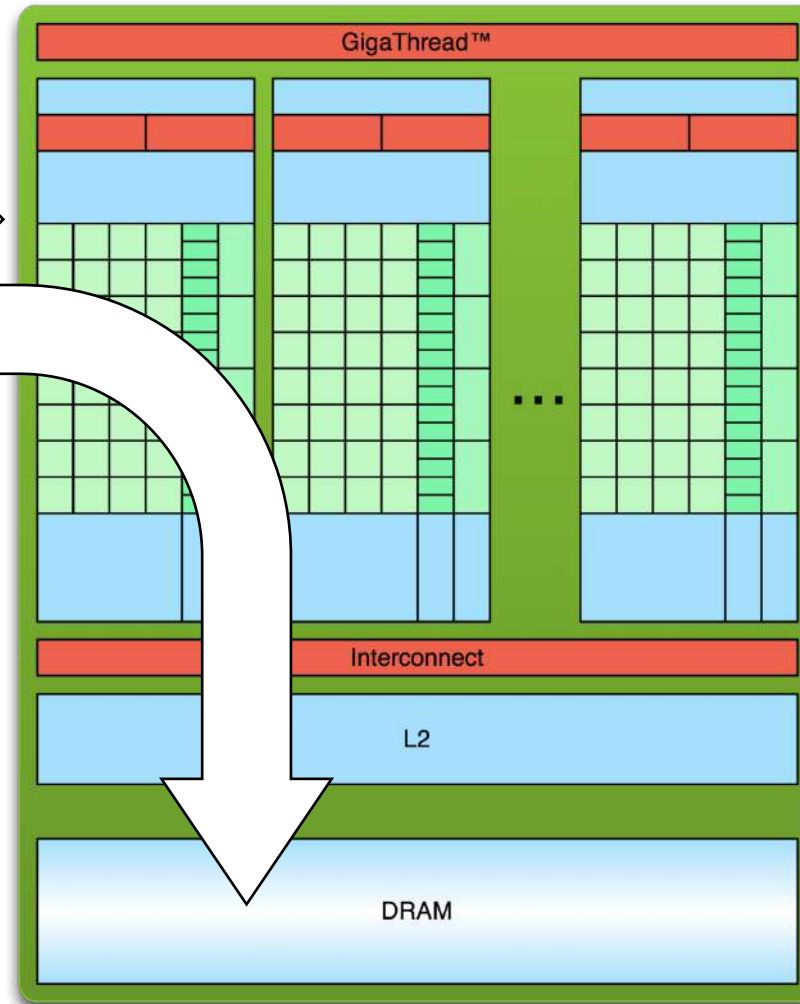
# Processing Flow

Host

Device

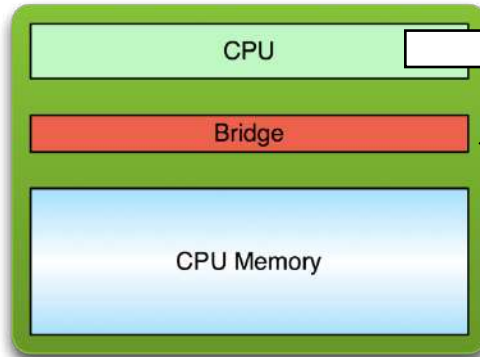


1. Copy input data from CPU memory to GPU memory



# Processing Flow

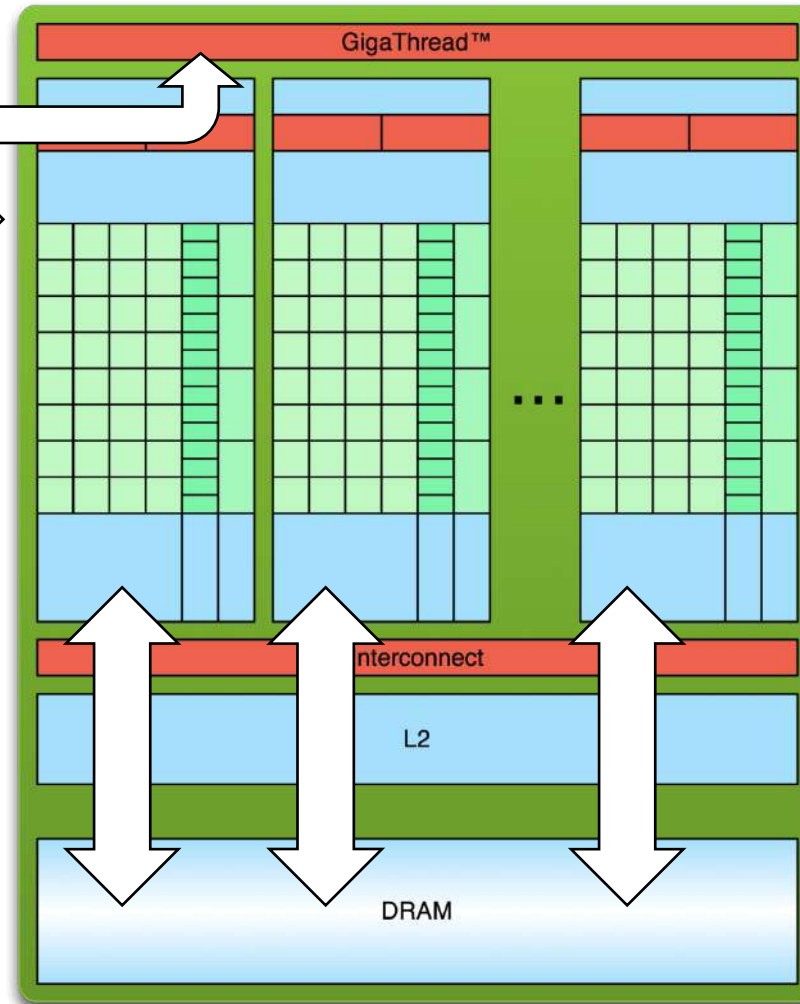
Host



PCI Bus

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

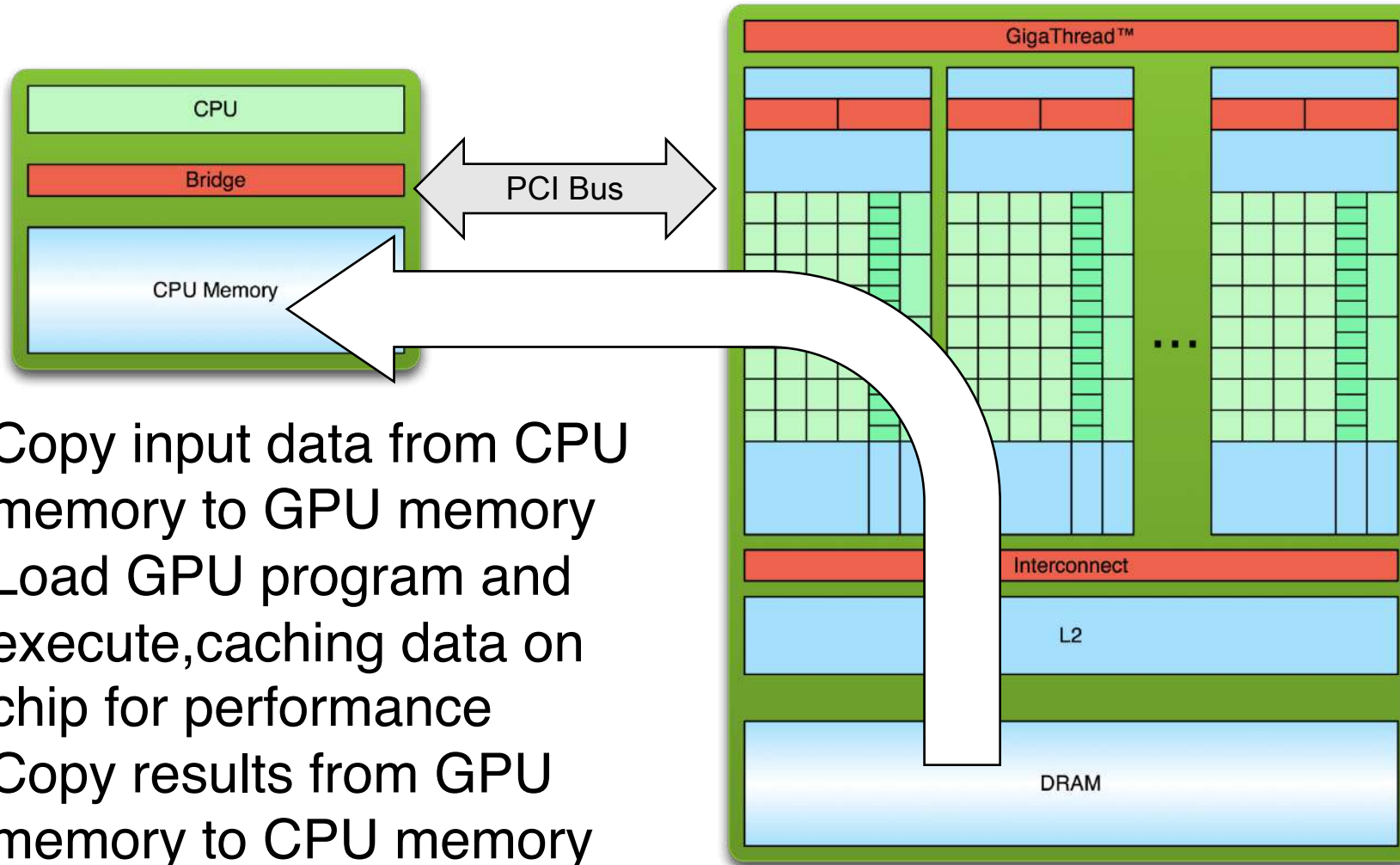
Device



# Processing Flow

Host

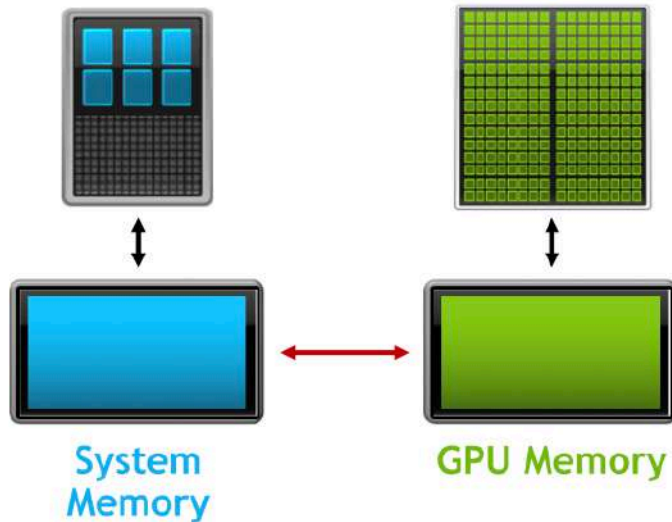
Device



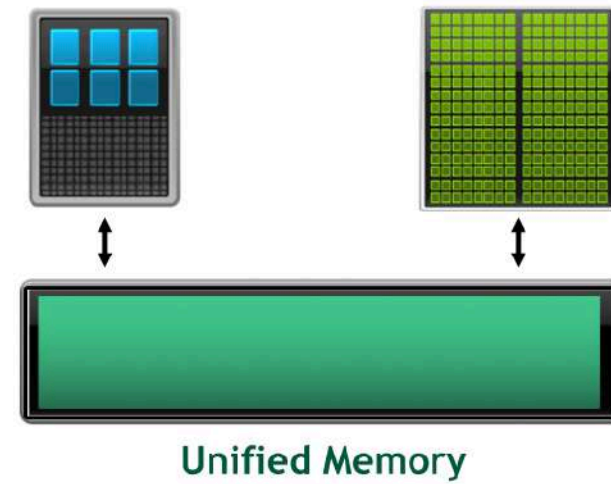
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

# 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
- Available since CUDA 6 on Kepler architecture
- Page fault mechanisms supported since Pascal architecture



# 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)

## 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

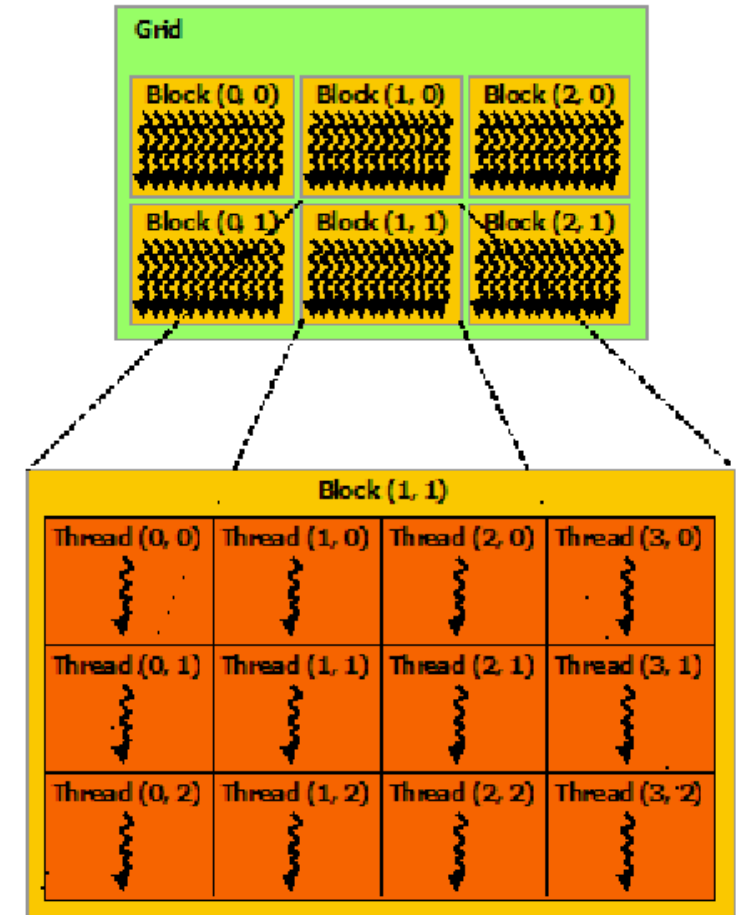
- Grids map to GPUs

### 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

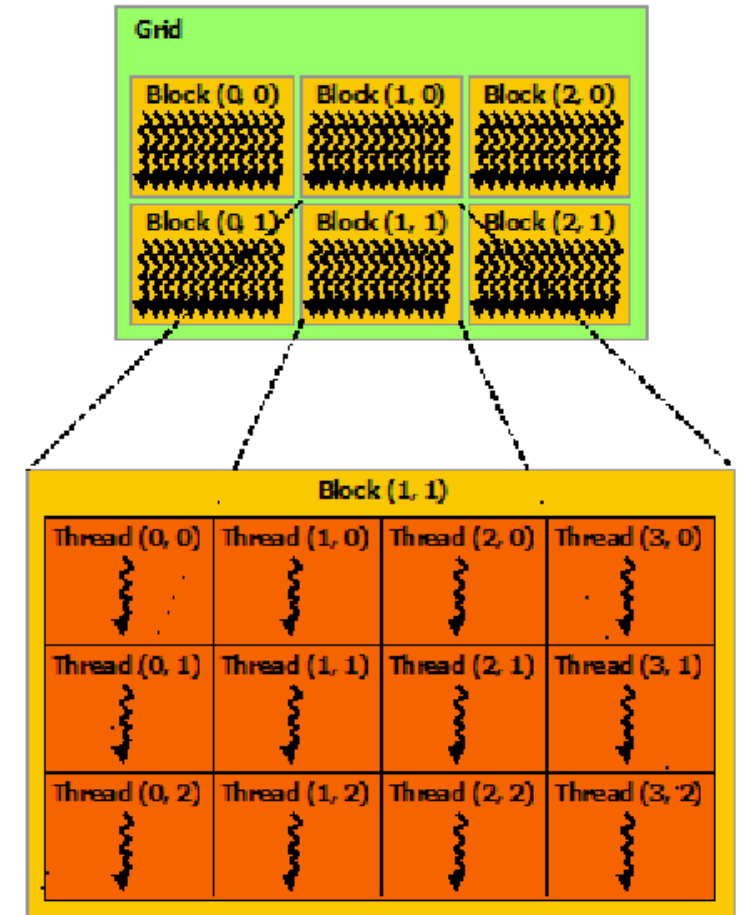
- `gridDim.x`, `gridDim.y`, `gridDim.z` (unused)
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- Variables that return the grid dimension (number of blocks) and block ID in the x-, y-, and z-axis.

## 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);
```

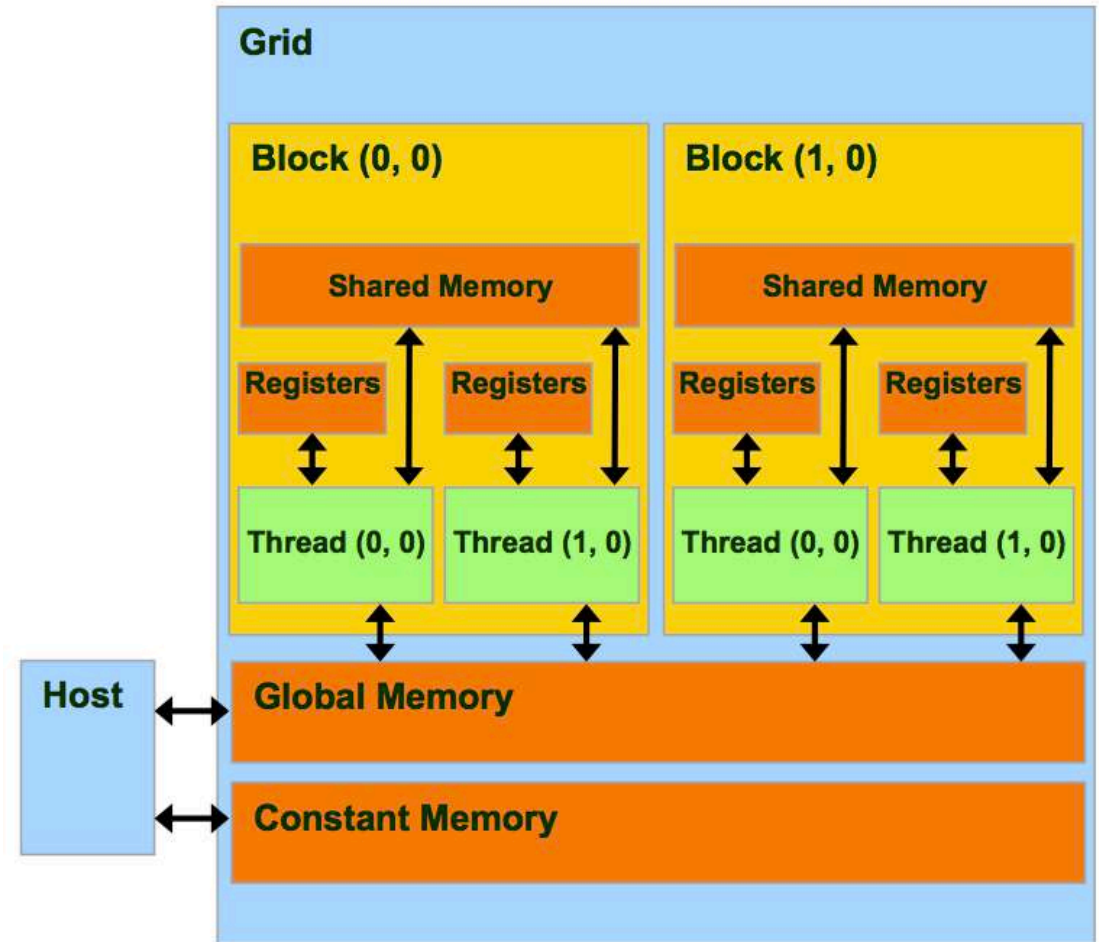
# Some CUDA basics

## 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

```
float* host_A, host_B, host_C;
float* device_A, device_B, device_C;

// Allocate host memory
host_A = (float*) malloc(mem_size_A);
host_B = (float*) malloc(mem_size_B);
host_C = (float*) malloc(mem_size_C);

// Allocate device memory
cudaMalloc((void**) &device_A, mem_size_A);
cudaMalloc((void**) &device_B, mem_size_B);
cudaMalloc((void**) &device_C, mem_size_C);

// Set up the initial values of A and B here.
...
```

# CUDA Example: Matrix-matrix multiply - 2

```
// copy host memory to device
cudaMemcpy(device_A, host_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, mem_size_B, cudaMemcpyHostToDevice);

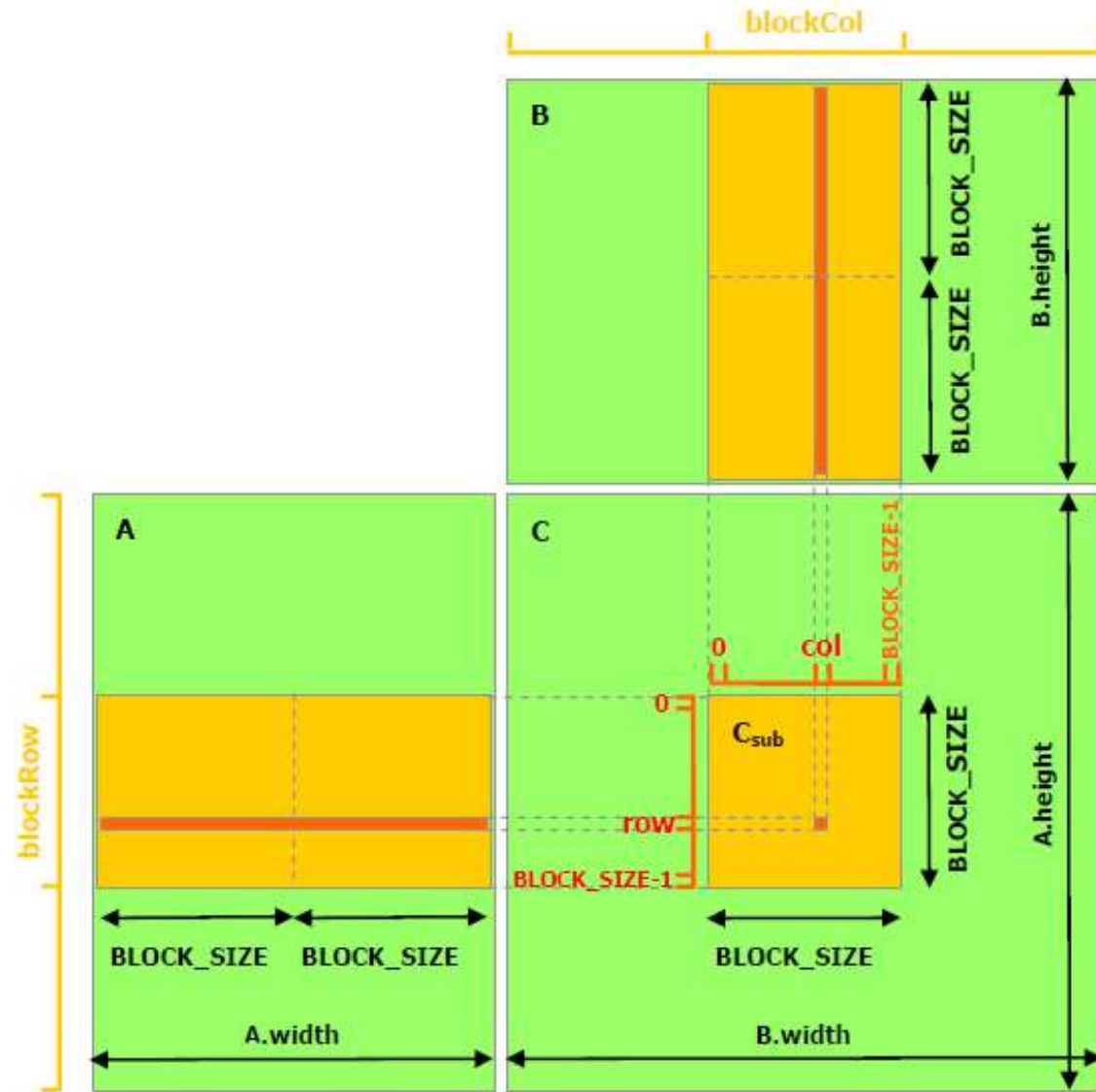
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

// execute the kernel
matrixMul<<< grid, threads >>>(device_C, device_A, device_B, WA, WB);

// copy result from device to host
cudaMemcpy(host_C, device_C, mem_size_C, cudaMemcpyDeviceToHost);

// Free host and device memory
...
```

# CUDA Example: Matrix-matrix multiply kernel



# CUDA Example: Matrix-matrix multiply kernel

```
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;
    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;
```

# CUDA Example: Matrix-matrix multiply kernel – 2

```
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {
    // Declaration of the shared memory array As
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    // Declaration of the shared memory array Bs
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    // Synchronize to make sure the matrices are loaded
    __syncthreads();
}
```

# CUDA Example: Matrix-matrix multiply kernel – 3

```
// Multiply the two matrices together;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);
// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}
// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}
```



# CUDA Example: Matrix-matrix multiply summary

## Summary

- We made use of a variety of CUDA features including
- 2D grids and blocks
- Shared memory
- Thread synchronization

## Note

- In reality we would not write a matrix-matrix multiplication function
- The CUDA implementation of BLAS is highly optimized for GPUs

# 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 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

## PGI Community Edition

- See <https://developer.nvidia.com/openacc-toolkit>
- Community Edition is free
- PGI Accelerator Fortran / C / C++ compilers
- PGI 2018 supports
  - OpenACC 2.6 for Nvidia GPUs
  - OpenACC 2.6, CUDA Fortran, OpenMP 4.5 for Multicore CPUs
- Pgprof performance profiler
- GPU-enabled libraries
- OpenACC code samples

# 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

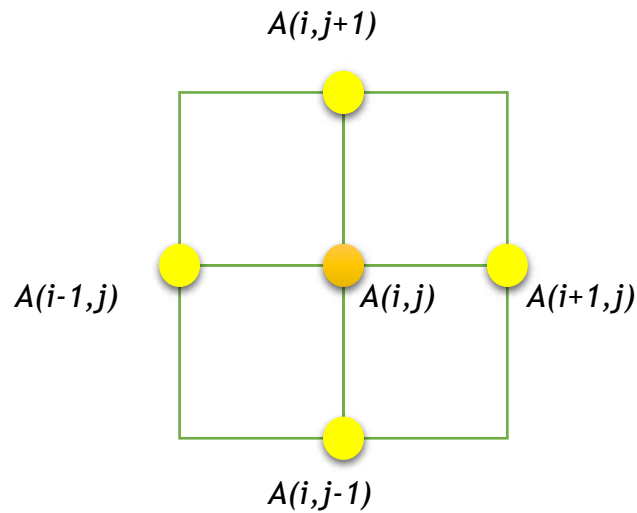
- `copy ( list )` Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
  - `copyin ( list )` Allocates memory on GPU and copies data from host to GPU when entering region.
  - `copyout ( list )` Allocates memory on GPU and copies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



Iterate until converged

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



Iterate across matrix  
elements

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



Calculate new value  
from neighbors

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



Compute max error for  
convergence

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



Swap input/output  
arrays

```
    iter++;  
}
```

# 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 -ta=nvidia -Minfo=accel -o jacobi-pgf90-acc-v1.x jacobi-acc-v1.f90
```

```
laplace:
```

```
44, Generating copyout(anew(1:4094,1:4094))
    Generating copyin(a(0:4095,0:4095))
45, Loop is parallelizable
46, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
45, !$acc loop gang ! blockidx%y
46, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
49, Max reduction generated for error
57, Generating copyin(anew(1:4094,1:4094))
    Generating copyout(a(1:4094,1:4094))
58, Loop is parallelizable
59, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
58, !$acc loop gang ! blockidx%y
59, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

# OpenACC example: Jacobi iteration – first attempt

**SDSC Comet**    CPU: Intel Xeon E5-2680 v3    GPU: NVIDIA Tesla K80  
(using single GPU)

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	71	--
CPU 2 OpenMP threads	41	1.73x
CPU 4 OpenMP threads	26	2.73x
CPU 6 OpenMP threads	24	2.96x
OpenACC GPU	501	0.05x FAIL

Speedup vs.  
1 CPU core

Speedup vs.  
6 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

#pragma acc kernels

Copy

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 Comet**

CPU: Intel Xeon E5-2680 v3

GPU: NVIDIA Tesla K80  
(using single GPU)

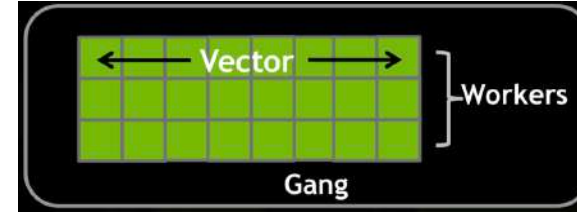
Execution	Time (s)	Speedup
CPU 1 OpenMP thread	71	--
CPU 2 OpenMP threads	41	1.73x
CPU 4 OpenMP threads	26	2.73x
CPU 6 OpenMP threads	24	2.96x
OpenACC GPU	5	4.8x

**CPU Speedup  
vs.  
1 CPU core**

**GPU Speedup  
vs.  
6 CPU cores**

# More OpenACC

- OpenACC gives us more detailed control over parallelization
  - Via **gang**, **worker**, and **vector** clauses
  - Gang corresponds to block, shares resources such as cache, streaming multiprocessor etc)
  - Vector threads work in lockstep (warp)
  - Workers compute a vector, correspond to threads
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance



# 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 independent 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  
`-ta=nvidia,time`
- 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 Comet GPU nodes

## 36 Nvidia K80 GPU nodes

- 2 x 12-core Intel Xeon E5-2680 v3 (Haswell) CPUs
- 128 GB RAM
- 2 x K80 GPUs on each node
- Each K80 = 2 GPUs => 4 GPUs per node
- 12 GB RAM per GPU

## 36 Nvidia P100 GPU nodes

- 2 x 14-core Intel Xeon E5-2680 v4 (Broadwell) CPUs
- 128 GB RAM
- 4 x P100 GPUs on each node
- 16 GB RAM per GPU

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





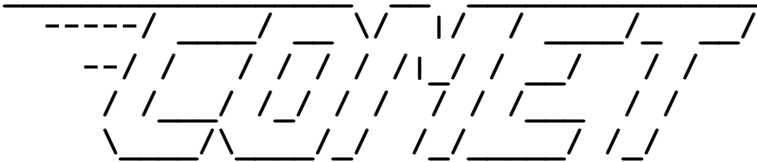
# SDSC Comet GPU nodes

## Login

```
$> ssh agoetz@comet.sdsc.edu
Last login: Tue Aug  2 15:45:49 2016 from 137.110.219.183
Rocks 6.2 (SideWinder)
Profile built 16:44 08-Feb-2016

Kickstarted 17:18 08-Feb-2016
```

WELCOME TO



## Checking available queues

```
agoetz@comet-ln2:~> qstat -q
Queue           Memory CPU Time  Walltime Node  Run Que  Lm  State
-----
compute         --      --    48:00:00   72   387 404  --  E R
debug           --      --    00:30:00    4     0  0  --  E R
shared          --      --    48:00:00    1   381  65  --  E R
gpu             --      --    48:00:00    4    18 239  --  E R
gpu-shared      --      --    48:00:00    1    28  13  --  E R
large-shared    --      --    48:00:00    1     8   4  --  E R
monitor         --      --      --      --     0   0  --  E R
maint           --      --      --      --     0   0  --  E R
-----
                        822   725
```

## GPU queues

- gpu  
(entire nodes with 4 GPUs)
- gpu-shared  
(individual GPUs)

# SDSC Comet GPU nodes

- The GPU nodes can be accessed via either the "gpu" or the "gpu-shared" partitions.

```
#SBATCH -p gpu
```

Or

```
#SBATCH -p gpu-shared
```

- In addition to the partition name (required), the type of gpu (optional) and the individual GPUs are scheduled as a resource.

```
#SBATCH --gres=gpu[:type]:n
```

- GPUs will be allocated on a first available, first schedule basis, unless specified with the [type] option, where type can be k80 or p100 (type is case sensitive).

```
#SBATCH --gres=gpu:4 #first available gpu node
```

```
#SBATCH --gres=gpu:k80:4 #only k80 nodes
```

```
#SBATCH --gres=gpu:p100:4 #only p100 nodes
```

# SDSC Comet GPU nodes

- For example, on the "gpu" partition, the following lines are needed to utilize all 4 p100 GPUs:

```
#SBATCH -p gpu
```

```
#SBATCH --gres=gpu:p100:4
```

- Users should always set `--ntasks-per-node` equal to 6 x [number of GPUs] requested on all k80 "gpu-shared" jobs, and 7 x [number of GPUs] requested on all p100 "gpu-shared" jobs".  
For instance, to request 2 x P100 GPUs:

```
#SBATCH -p gpu-shared
```

```
#SBATCH --ntasks-per-node=14
```

```
#SBATCH --gres=gpu:p100:2
```

- Example job submission scripts are in `/share/apps/examples/GPU`

## Charging SUs

- GPU SUs = [(Number of K80 GPUs) + (Number of P100 GPUS)\*1.5] x (wallclock time)

# SDSC Comet GPU nodes

- Load CUDA module and check Nvidia CUDA C compiler (available CUDA versions: 6.5, 7.0 (default), 7.5, 8.0, 9.2)

```
[agoetz@comet-30-03 ~]$ module load cuda
[agoetz@comet-30-03 ~]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2015 NVIDIA Corporation
Built on Mon_Feb_16_22:59:02_CST_2015
Cuda compilation tools, release 7.0, V7.0.27
```

- Load PGI module and check PGI C compiler

```
[agoetz@comet-30-03 ~]$ module load pgi
[agoetz@comet-30-03 ~]$ pgcc --version

pgcc 17.5-0 64-bit target on x86-64 Linux -tp haswell
PGI Compilers and Tools
Copyright (c) 2017, NVIDIA CORPORATION. All rights reserved.
```

# SDSC Comet GPU nodes

This reservation is active  
on 01/24/2020  
from 1 pm to 6 pm

- Interactive access to GPU nodes

```
agoetz@comet-ln2:~> srun --partition=gpu-shared --reservation=gputraining \  
--nodes=1 --ntasks-per-node=6 --gres=gpu:k80:1 \  
-t 00:10:00 --pty --wait=0 --export=ALL /bin/bash
```

- Check available GPUs using Nvidia system management interface

```
[agoetz@comet-33-02 ~]$ nvidia-smi  
Tue Apr  9 00:41:26 2019
```

```
+-----+  
| NVIDIA-SMI 396.26                  Driver Version: 396.26                  |  
+-----+-----+-----+-----+  
| GPU   Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |  
| Fan   Temp   Perf   Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |  
+-----+-----+-----+-----+  
|    0   Tesla P100-PCIE...    On      | 00000000:04:00.0 Off |             0      |  
| N/A    31C    P0      30W / 250W |      0MiB / 16280MiB |      0%      Default |  
+-----+-----+-----+-----+  
|    1   Tesla P100-PCIE...    On      | 00000000:05:00.0 Off |             0      |  
| N/A    57C    P0     135W / 250W |     523MiB / 16280MiB |     96%      Default |  
+-----+-----+-----+-----+  
...
```

# SDSC Comet GPU nodes

- Other jobs may already be running on shared GPU nodes.

```
...
+-----+-----+-----+
|  3  Tesla P100-PCIE...  On  | 00000000:86:00.0 Off |           0 |
| N/A   62C   P0   156W / 250W |  1047MiB / 16280MiB |       95%   Default |
+-----+-----+-----+

+-----+-----+-----+
| Processes:                                     GPU Memory |
| GPU      PID    Type   Process name                      Usage          |
|=====|=====|=====|=====|
|    1    181582    C    /opt/amber/16/bin/pmemd.cuda        513MiB         |
|    2     65784    C    pmemd.cuda                        1037MiB         |
|    3     67447    C    pmemd.cuda                        1037MiB         |
+-----+-----+-----+
```

- The nodes of the shared GPU queue are configured for the CUDA runtime to use only the requested number of GPUs.
- Check environment variable `CUDA_VISIBLE_DEVICES` for the GPU that has been assigned to you.

# SDSC Comet GPU nodes

## CUDA Toolkit Samples

- Install CUDA Toolkit code samples (does not require GPU node access)

```
[agoetz@comet-31-16 ~]$ cuda-install-samples-7.0.sh ./  
Copying samples to ./NVIDIA_CUDA-7.0_Samples now...  
Finished copying samples.
```

- Explore CUDA Toolkit samples – great resource!

```
[agoetz@comet-31-16 ~]$ cd NVIDIA_CUDA-7.0_Samples/  
[agoetz@comet-31-16 NVIDIA_CUDA-7.0_Samples]$ ls  
0_Simple      2_Graphics    4_Finance      6_Advanced      common      Makefile  
1_Uutilities  3_Imaging     5_Simulations  7_CUDALibraries EULA.txt
```

- Compile CUDA Toolkit samples

```
[agoetz@comet-31-16 NVIDIA_CUDA-7.0_Samples]$ make -j 6  
make[1]: Entering directory `/home/agoetz/NVIDIA_CUDA-  
7.0_Samples/0_Simple/simpleMultiCopy'  
/usr/local/cuda-7.0/bin/nvcc -ccbin g++ -I../common/inc -m64 -gencode  
arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode  
arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode  
arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -gencode  
arch=compute_52,code=compute_52 -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@comet-31-16 NVIDIA_CUDA-7.0_Samples]$ cd 1_Uutilities/deviceQuery
[agoetz@comet-31-16 deviceQuery]$ make
/usr/local/cuda-7.0/bin/nvcc -ccbin g++ -I.././common/inc -m64 -gencode arch=com
...
[agoetz@comet-31-16 deviceQuery]$ ./deviceQuery
./deviceQuery Starting...
  CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
```

Device 0: "Tesla K80"

CUDA Driver Version / Runtime Version	8.0 / 7.0
CUDA Capability Major/Minor version number:	3.7
Total amount of global memory:	11440 MBytes (11995578368 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP:	2496 CUDA Cores
GPU Max Clock rate:	824 MHz (0.82 GHz)
Memory Clock rate:	2505 Mhz

# SDSC Comet GPU nodes

## CUDA Toolkit

- Matrix multiplication example

```
agoetz@comet-30-11:~>cd NVIDIA_CUDA-7.0_Samples/0_Simple/  
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples/0_Simple>./matrixMul/matrixMul  
[Matrix Multiply Using CUDA] - Starting...  
GPU Device 0: "Tesla K80" with compute capability 3.7
```

```
MatrixA(320,320), MatrixB(640,320)  
Computing result using CUDA Kernel...  
done  
Performance= 231.28 GFlop/s, Time= 0.567 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block  
Checking computed result for correctness: Result = PASS
```

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.

- Matrix multiplication example with CUBLAS

```
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples/0_Simple>./matrixMulCUBLAS/matrixMulCUBLAS  
[Matrix Multiply CUBLAS] - Starting...  
GPU Device 0: "Tesla K80" with compute capability 3.7
```

```
MatrixA(320,640), MatrixB(320,640), MatrixC(320,640)  
Computing result using CUBLAS...done.  
Performance= 952.24 GFlop/s, Time= 0.138 msec, Size= 131072000 Ops  
Computing result using host CPU...done.  
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```



# Questions?