GPU Computing and Programming

Andreas W Götz

San Diego Supercomputer Center
University of California, San Diego

SDSC Summer Institute 2019
Thursday, August 8, 2019, 8:30 am to 12:00 pm

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
- Exercises 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

Tseng Labs ET4000/W32p 1991

Voodoo3 2000 AGP card 1999

GeForce 6600 GT Personal Cinema 2004

NVIDIA GeForce GTX 280 2008

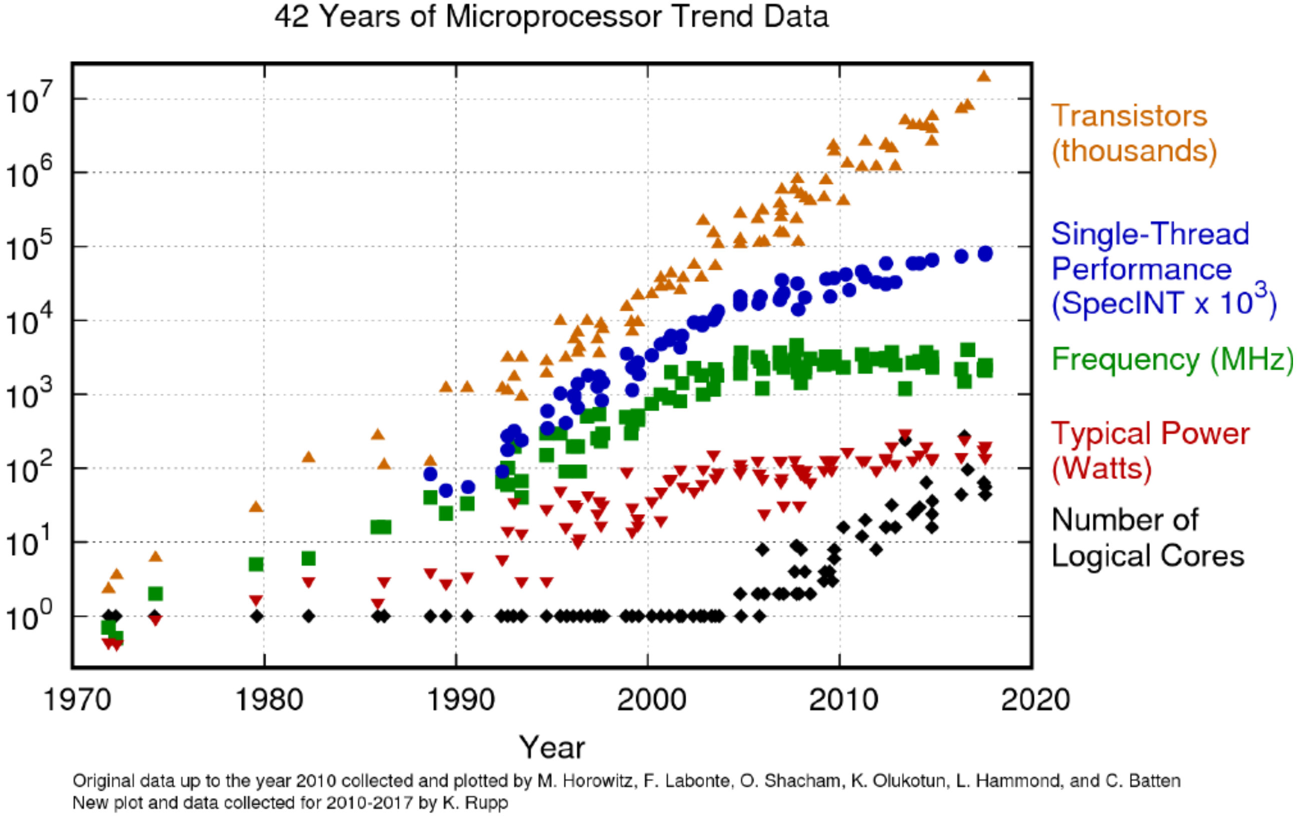
Why is there such an interest in GPUs?

Moore's law

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

Trends since mid 2000s

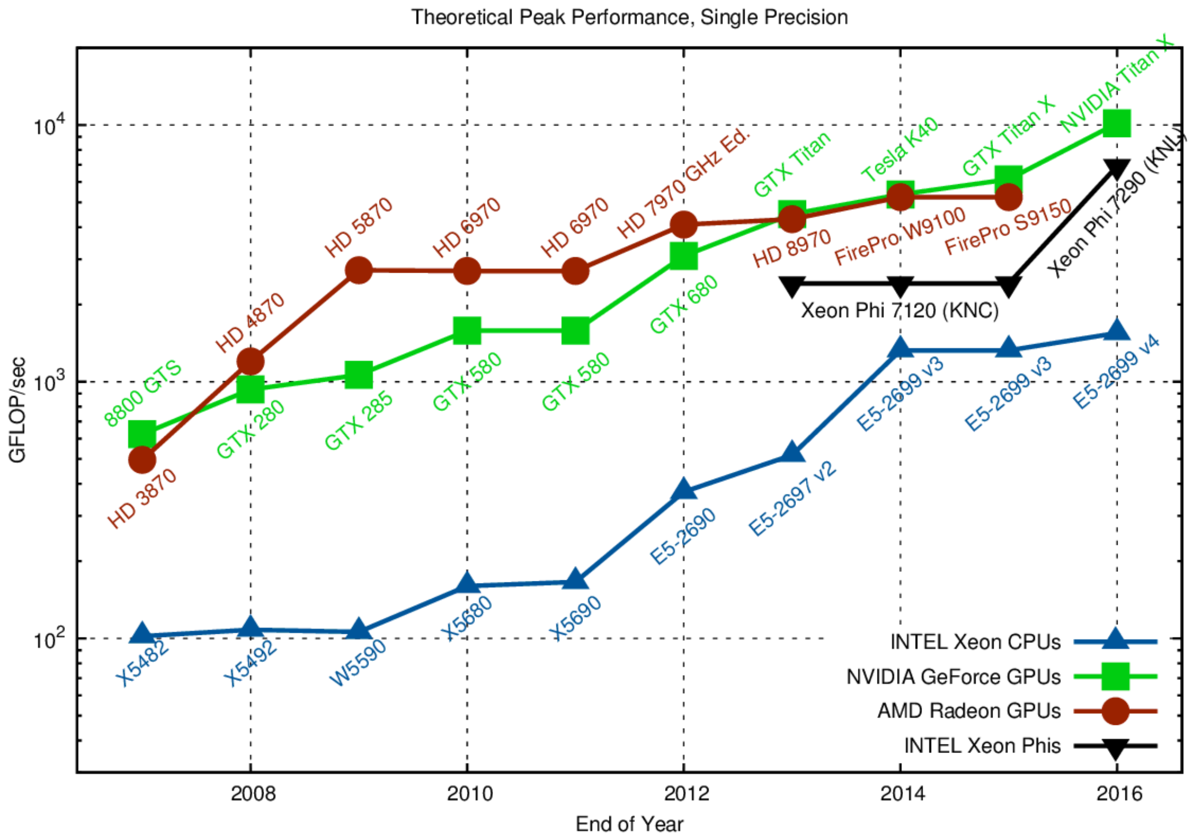
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.

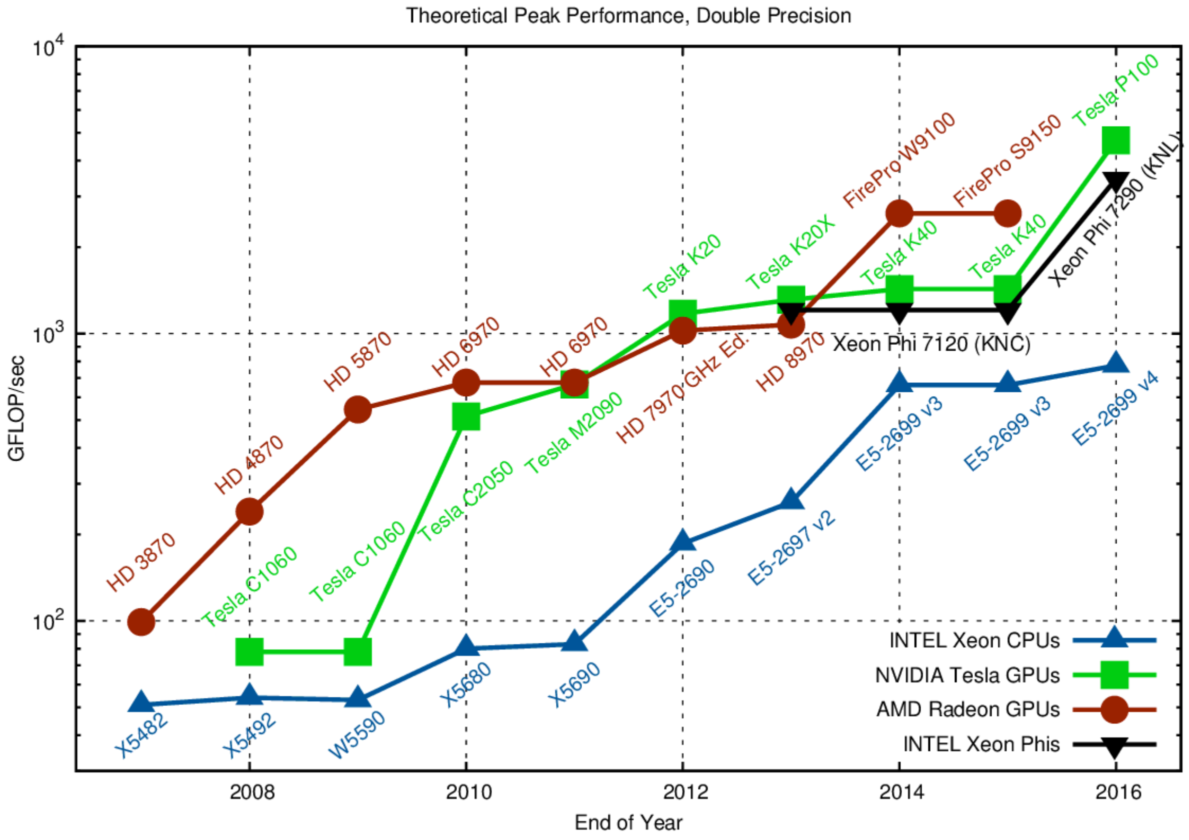


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

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

Why is there such an interest in GPUs?



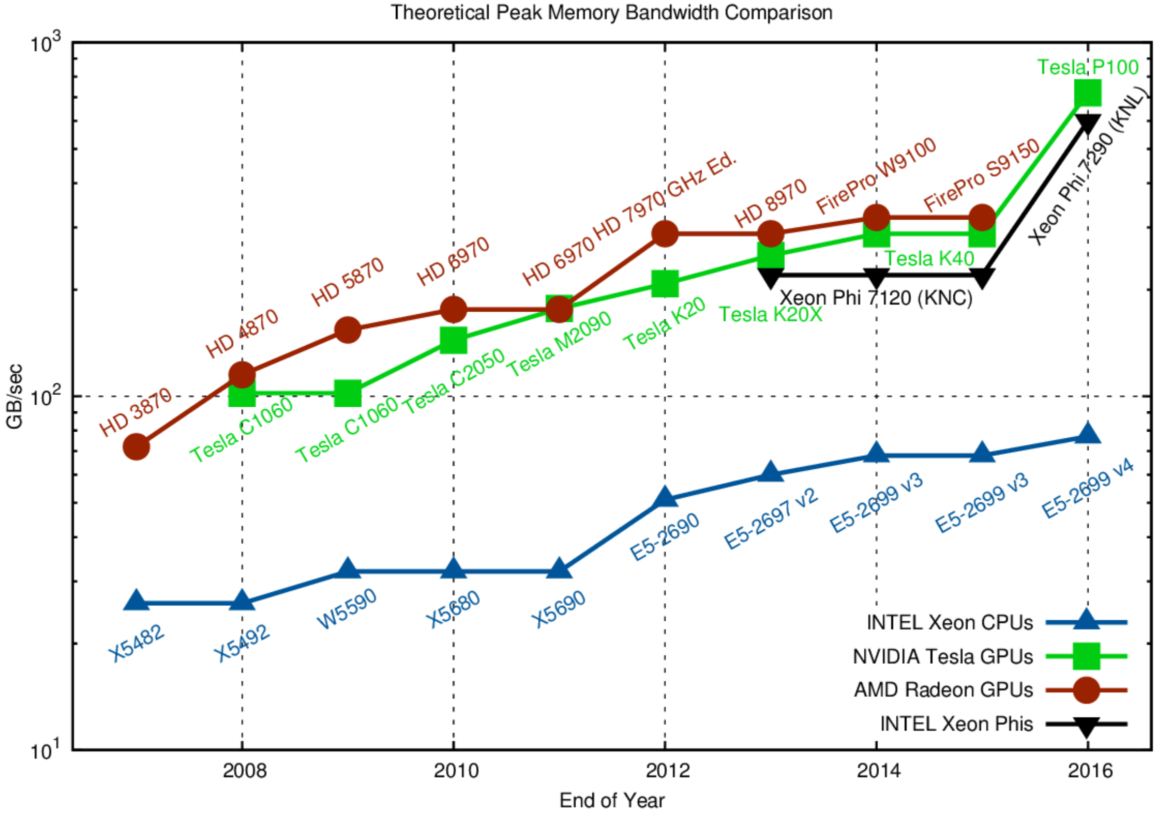


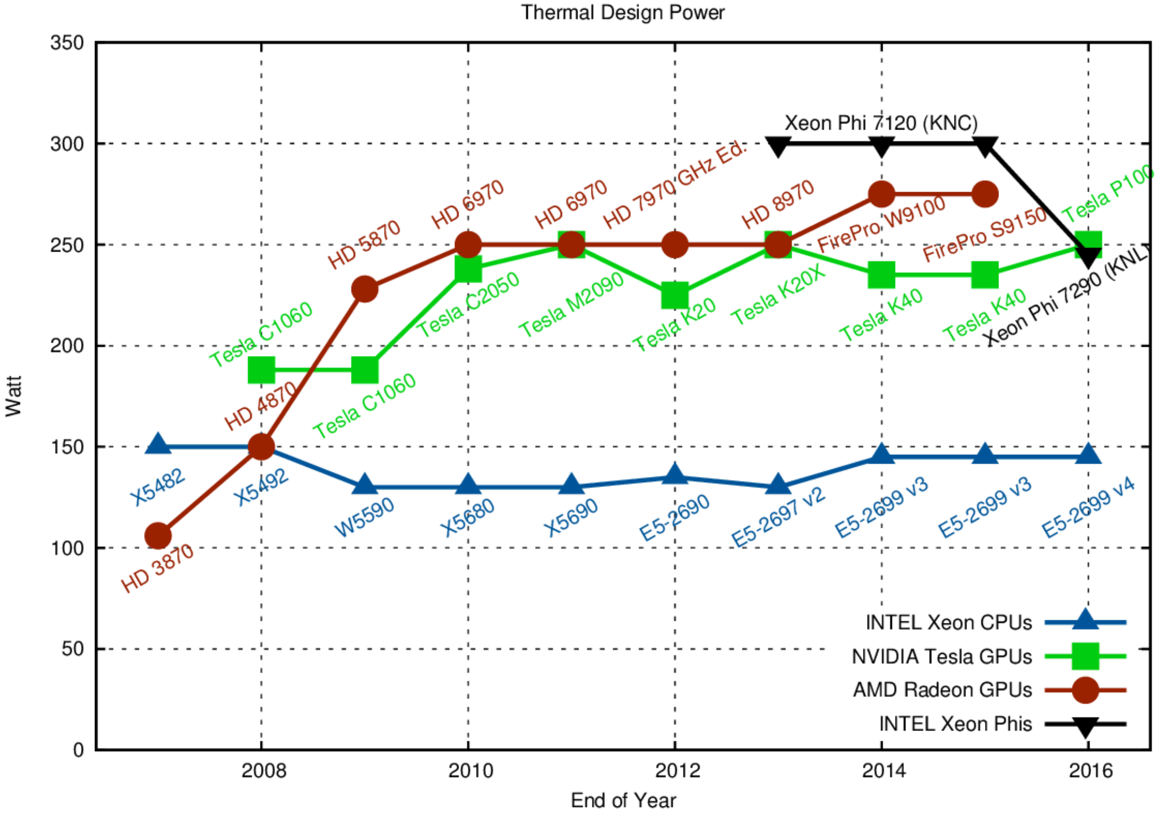
- GPUs offer significantly higher 32-bit floating point performance than CPUs.

- 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

intel
Intel® Xeon® Platinum Processor
intel
Intel® Xeon® Platinum Processor

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)

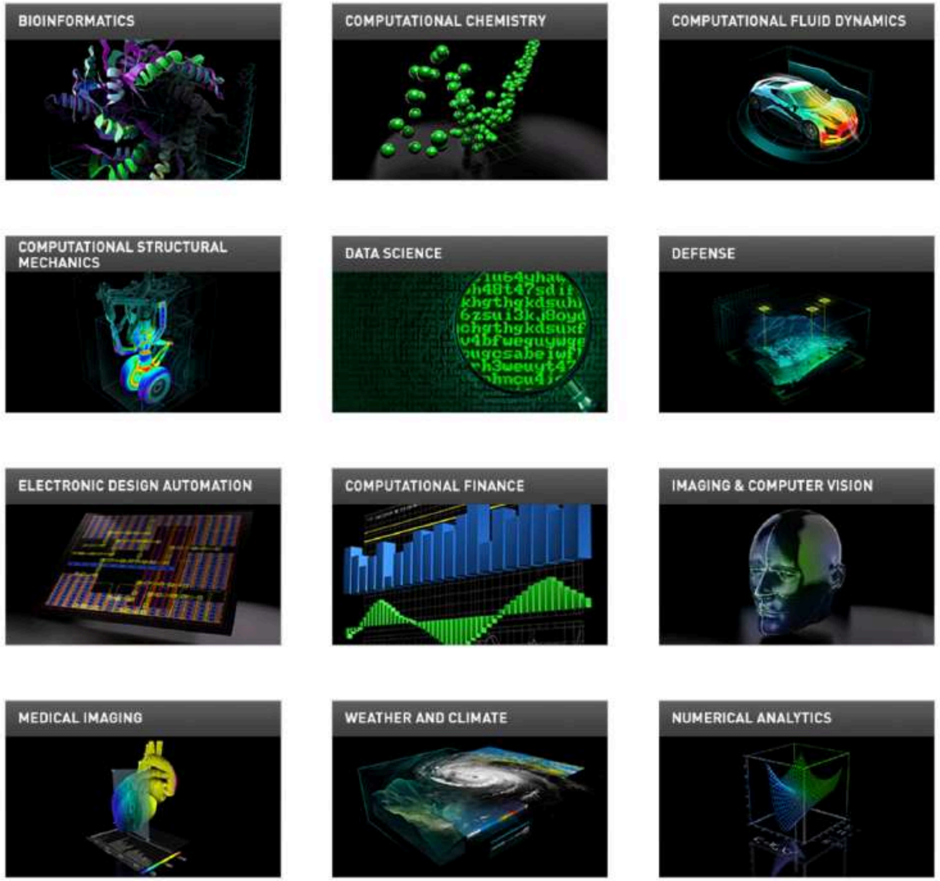
DIY 4 x Nvidia RTX 2080 box

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

GPU accelerated software

Examples from virtually any field

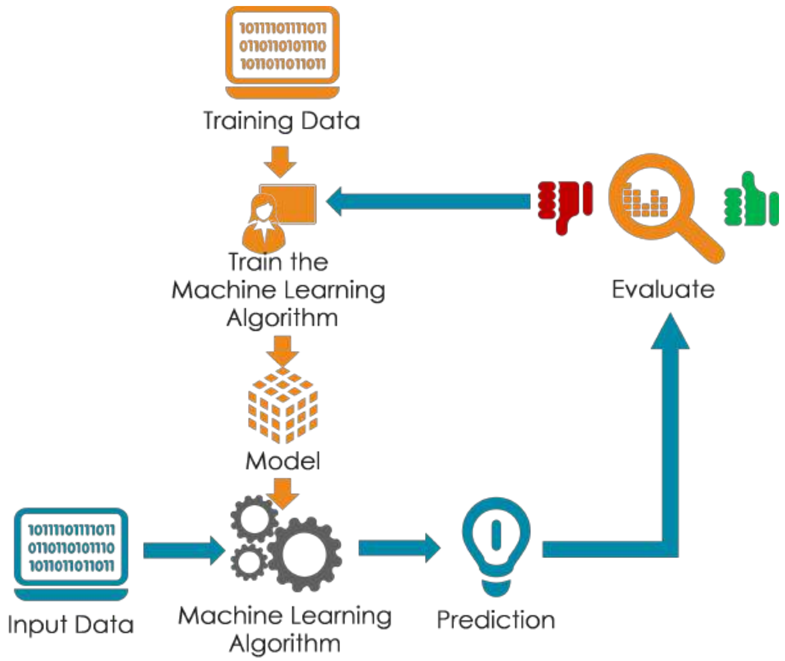
- Exhautive list on https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/
- Chemistry
- Life sciences
- Bioinformatics
- Astrophysics
- Finance
- Medical imaging
- Natural language processing
- Social sciences
- Weather and climate
- Computational fluid dynamics
- Machine learning, of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



Machine learning and GPUs

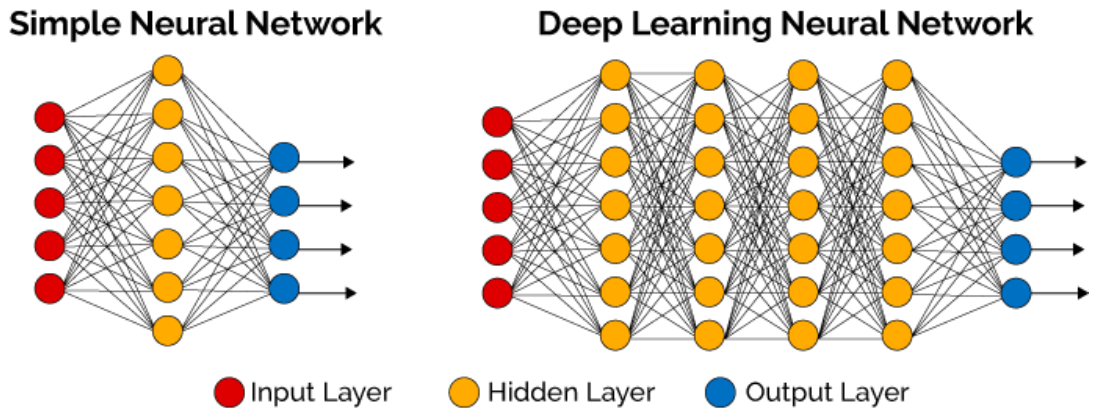
Machine learning

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

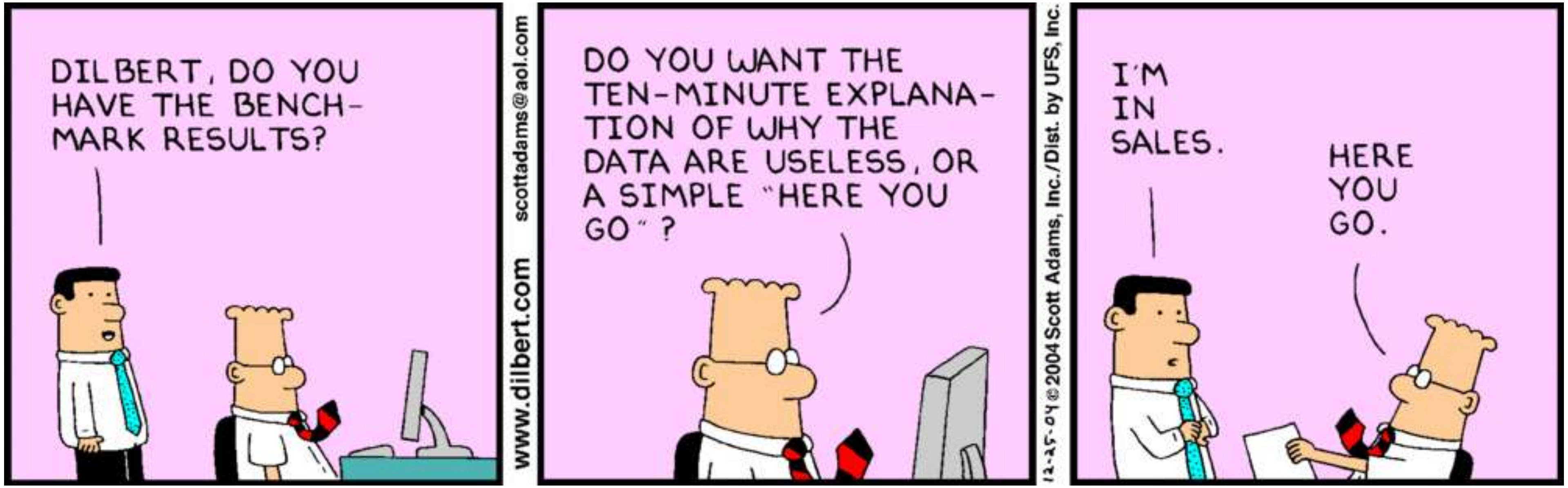

Training Data
Train the Machine Learning Algorithm
Evaluate
Model
Input Data
Machine Learning Algorithm
Prediction

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)



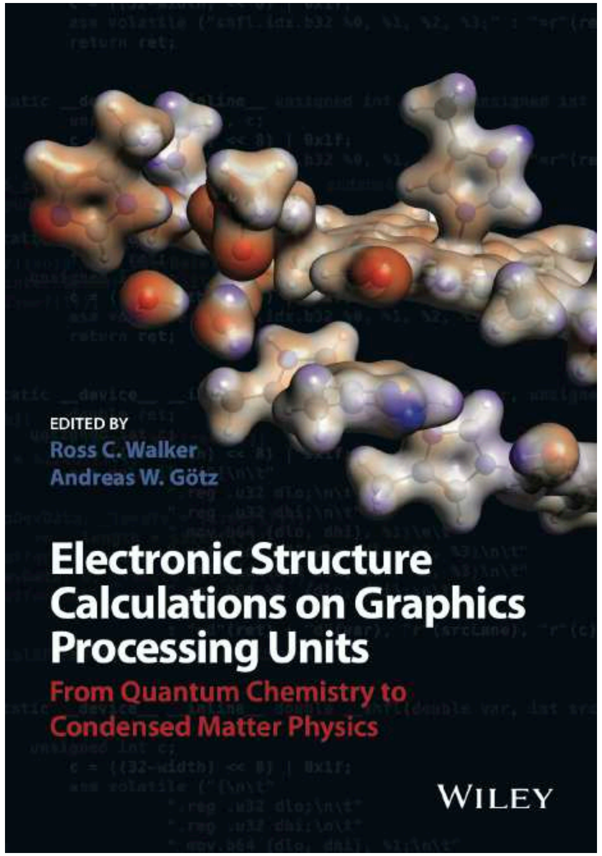
Benchmark examples

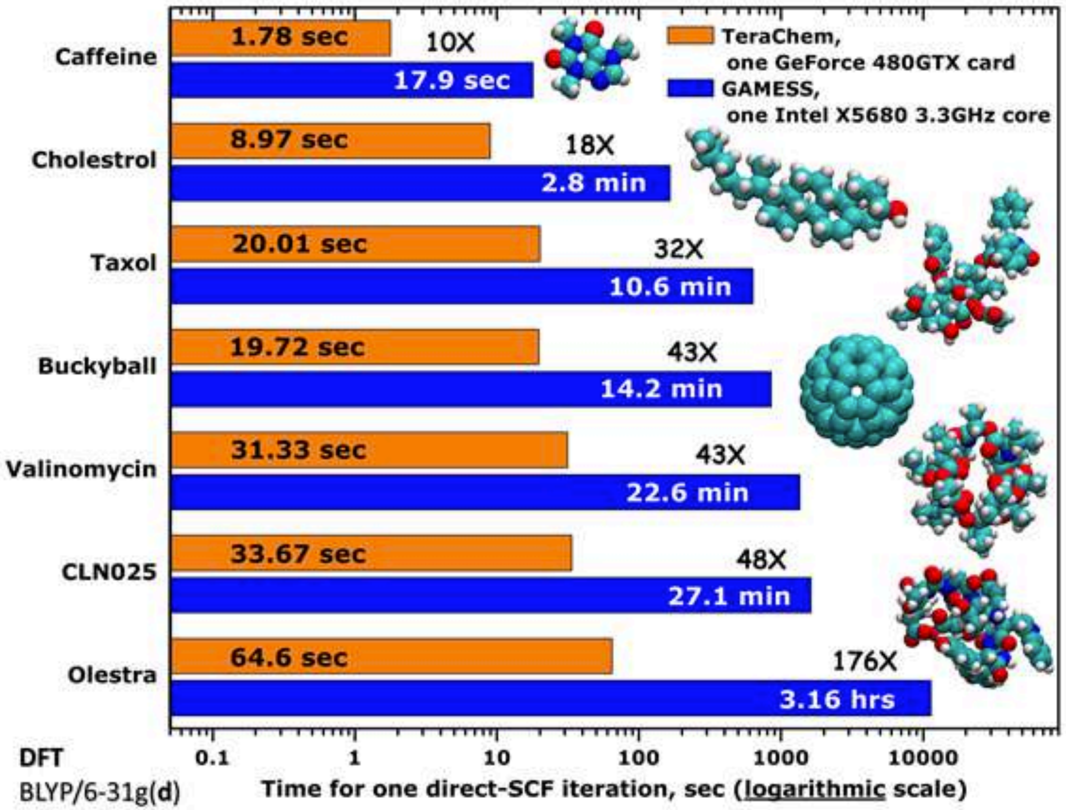

DILBERT, DO YOU HAVE THE BENCHMARK RESULTS?
DO YOU WANT THE TEN-MINUTE EXPLANATION OF WHY THE DATA ARE USELESS, OR A SIMPLE "HERE YOU GO"?
I'M IN SALES.
HERE YOU GO.
scottadams@aol.com
www.dilbert.com
12-25-04 © 2004 Scott Adams, Inc./Dist. by UFS, Inc.

Benchmark examples

Quantum chemistry

- Compute molecular properties from quantum mechanics (TeraChem code)

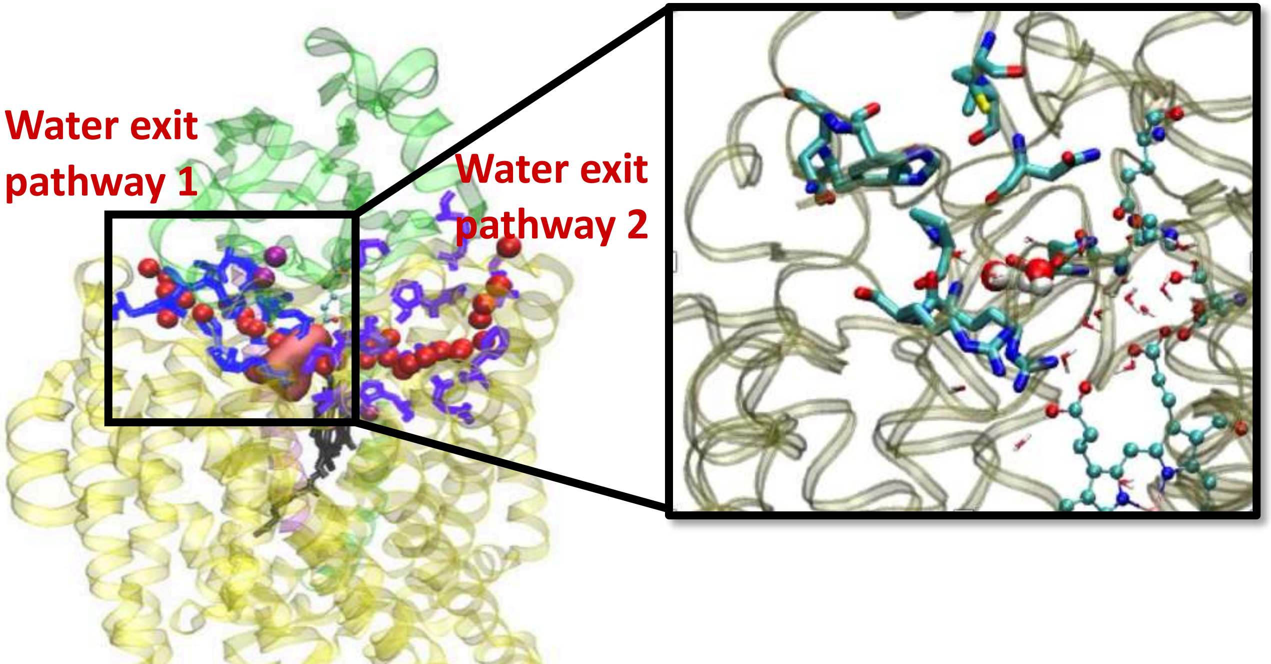

EDITED BY
Ross C. Walker
Andreas W. Götz
Electronic Structure Calculations on Graphics Processing Units
From Quantum Chemistry to Condensed Matter Physics
WILEY



Benchmark examples

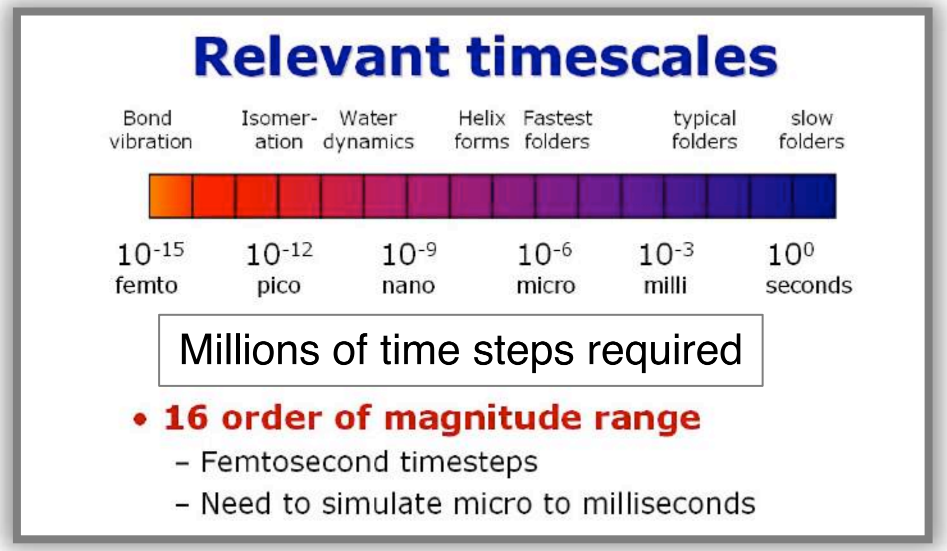
Molecular dynamics

- Amber code: Atomistic simulations of condensed phase biomolecular systems



Cytochrome c oxidase enzyme

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



Relevant timescales

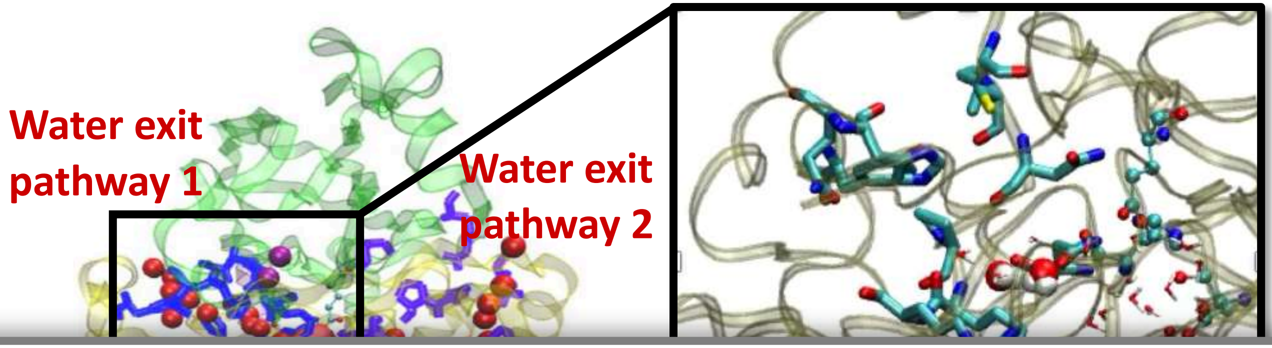
Millions of time steps required

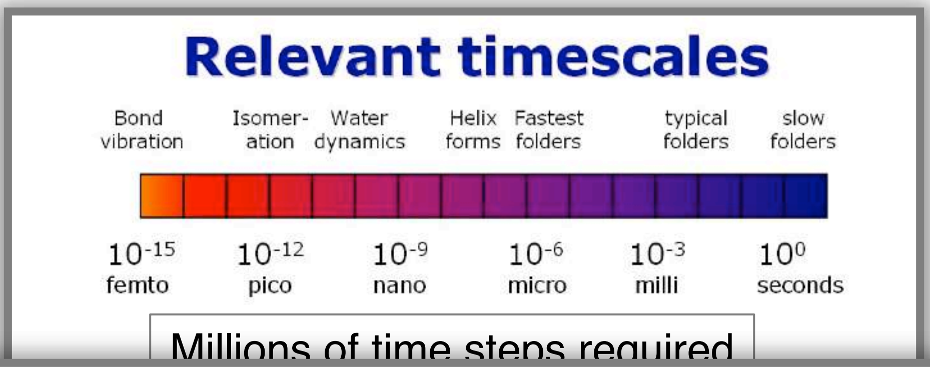
- **16 order of magnitude range**
  - Femtosecond timesteps
  - Need to simulate micro to milliseconds

Benchmark examples

Molecular dynamics

- Amber code: Atomistic simulations of condensed phase biomolecular systems





Relevant timescales

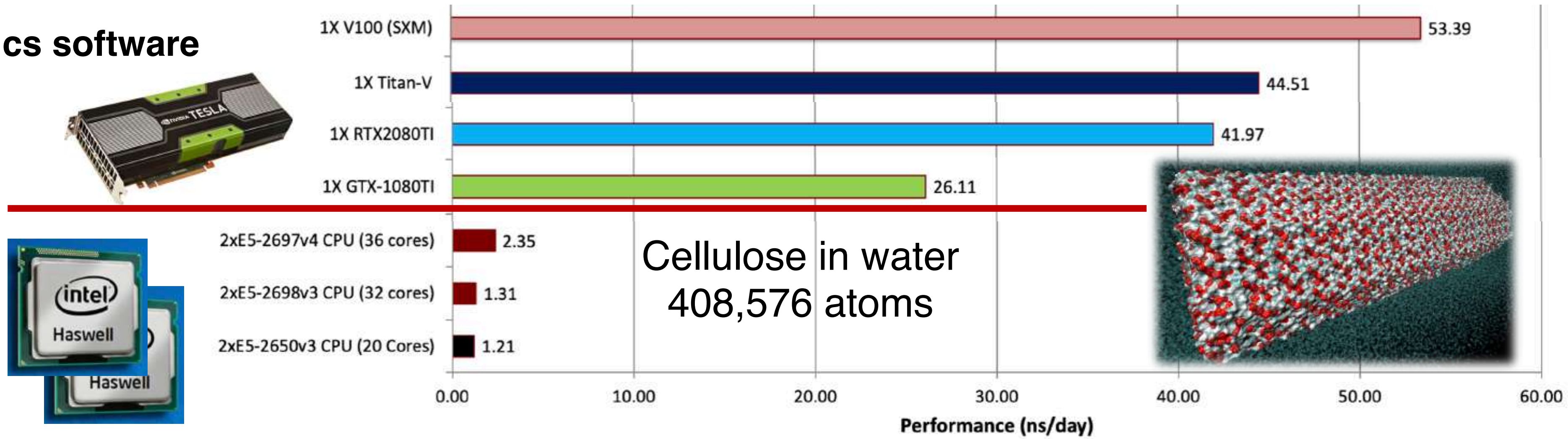
Millions of time steps required

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.

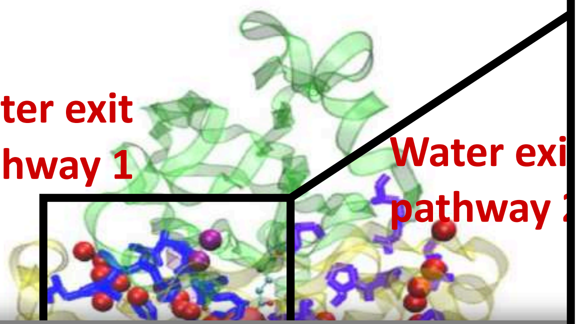


Cellulose in water
408,576 atoms

Benchmark examples

Molecular dynamics

- Amber code: Atomistic

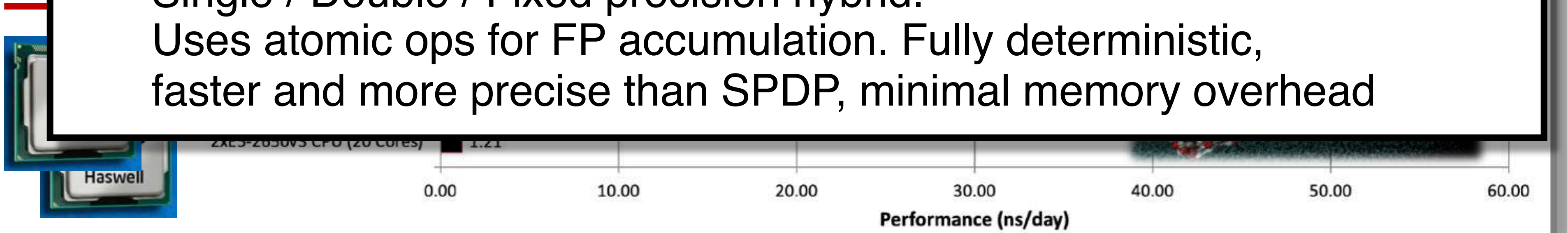


Amber 18 molecular dynamics

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

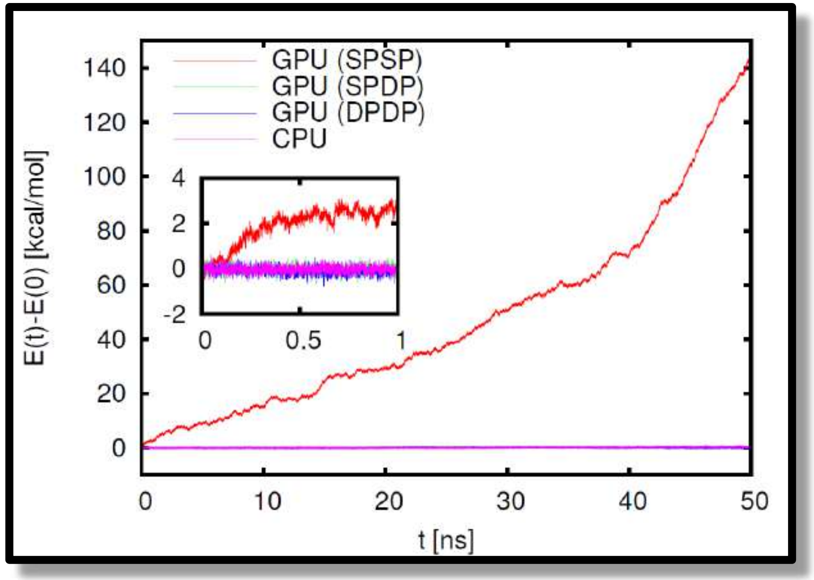
Le Grand, Götz, Walker, *Comput Phys Comm* 2013 (184) 374.

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

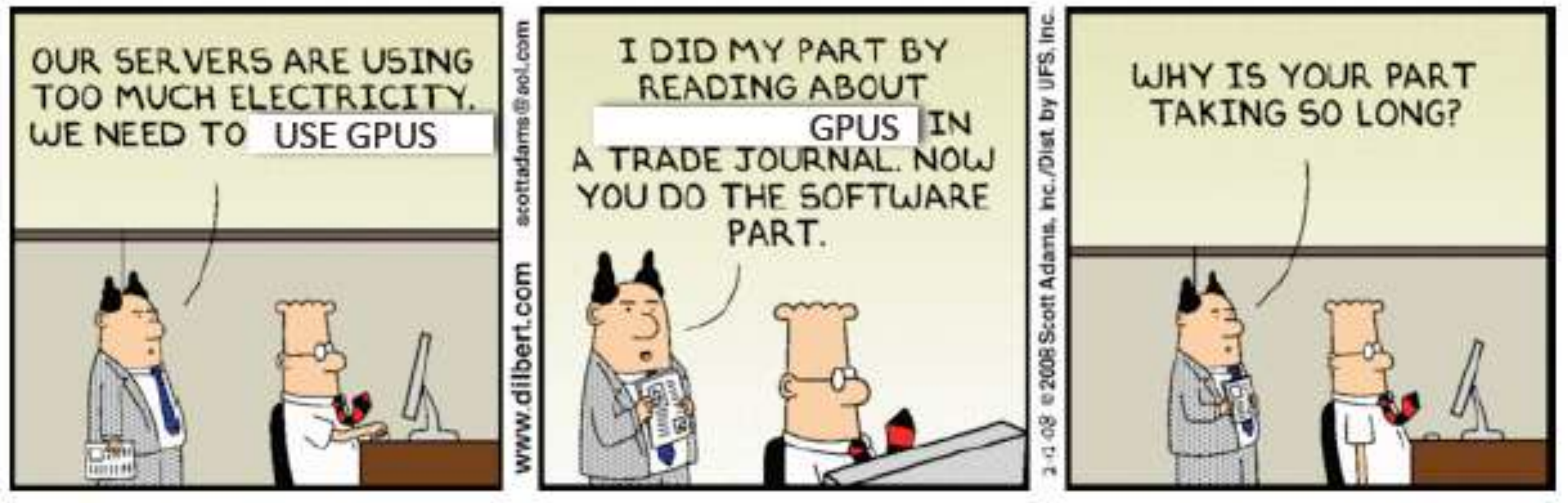


Precision matters

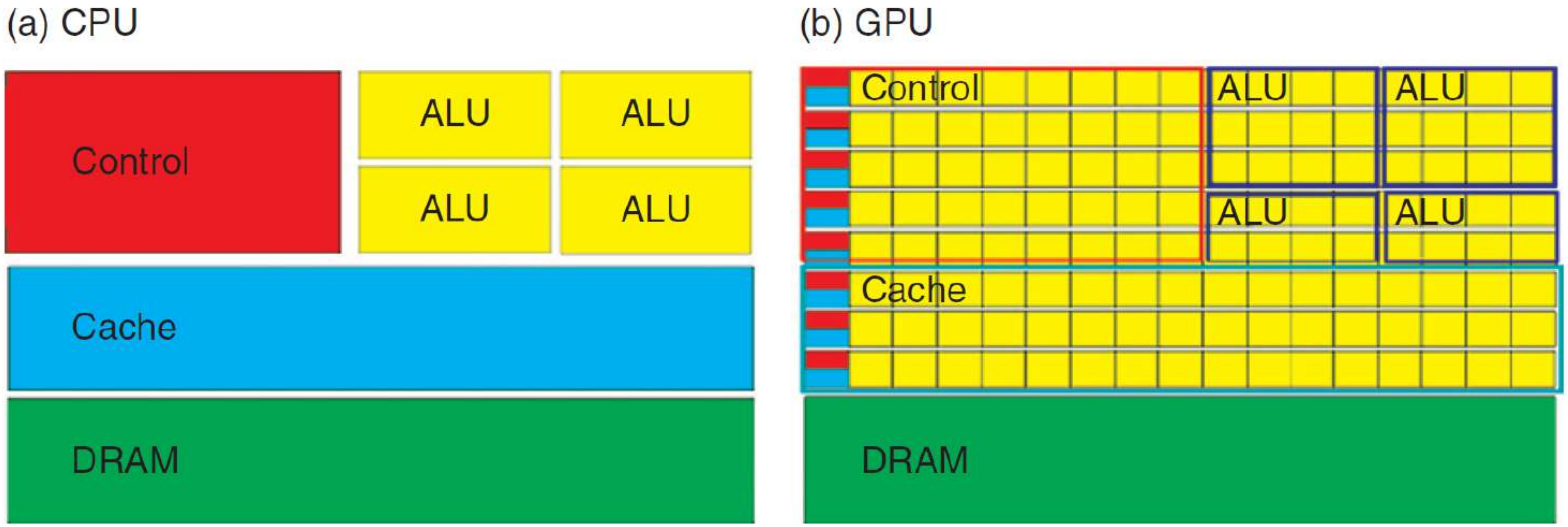
- ***SPSP***
  Only single precision
- ***SPDP***
  Single precision for calculation
  Double precision for accumulation
- ***DPDP***
  Full double precision
- ***SPFP***
  Single / Double / Fixed precision hybrid.
  Uses atomic ops for FP accumulation. Fully deterministic, faster and more precise than SPDP, minimal memory overhead



What’s the catch?


OUR SERVERS ARE USING TOO MUCH ELECTRICITY. WE NEED TO USE GPUS
I DID MY PART BY READING ABOUT GPUS IN A TRADE JOURNAL. NOW YOU DO THE SOFTWARE PART.
WHY IS YOUR PART TAKING SO LONG?
www.dilbert.com scottadams@aol.com
2-11-08 © 2008 Scott Adams, Inc./Dist. by UFS, Inc.

GPU vs CPU architecture



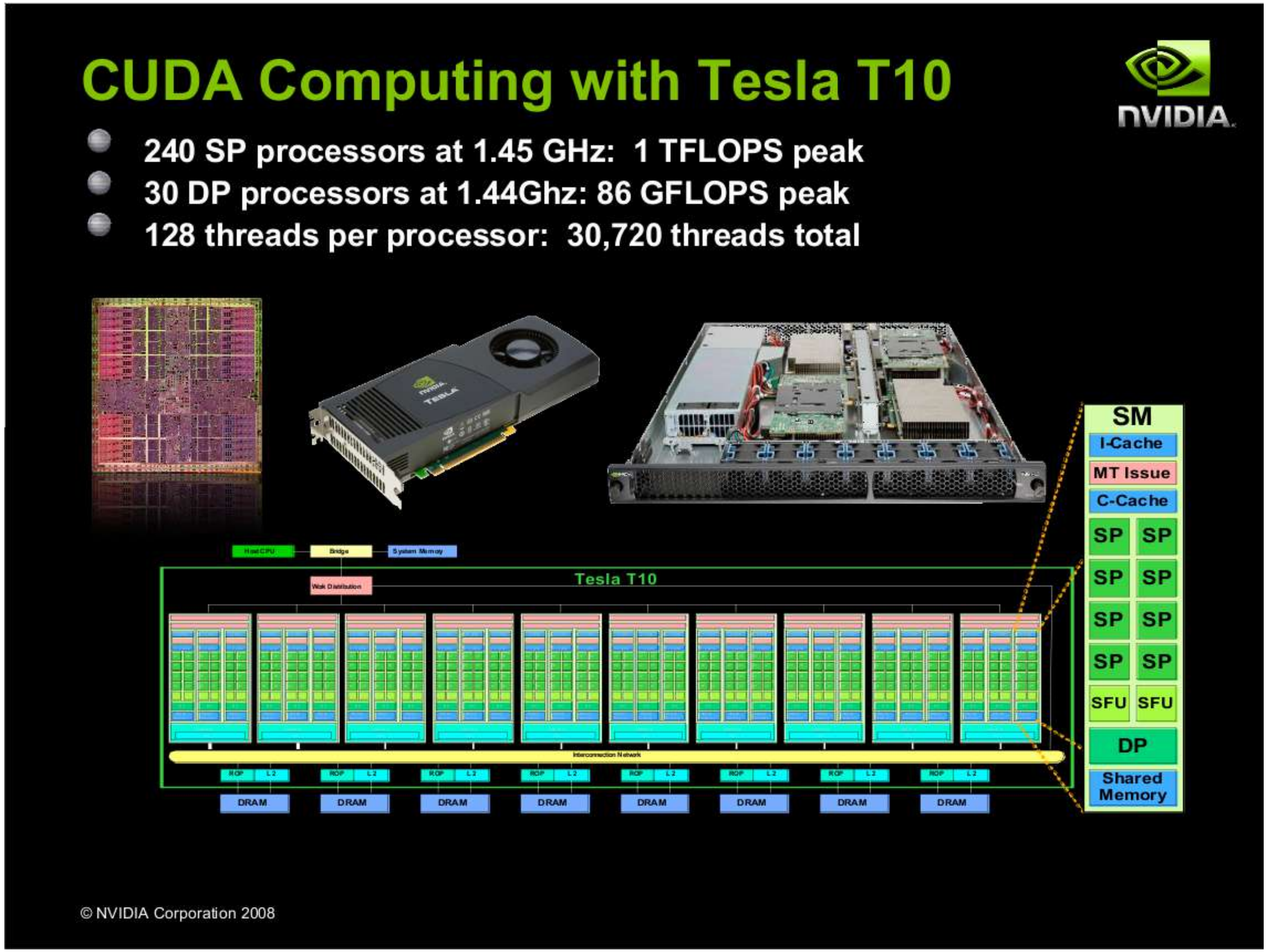
CPU

- 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



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

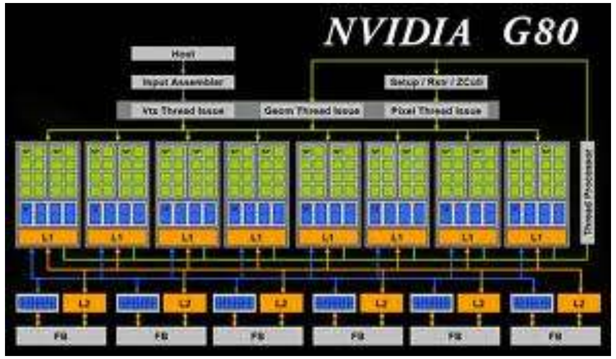
Different hardware vendors work in different ways

- Nvidia vs AMD

Hardware complexities

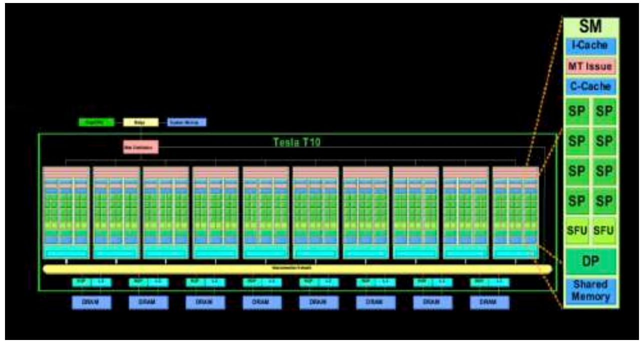
C870 – Nov 2006

- First ‘programmable’ Card
- Single Precision Only
- 1.5GB RAM



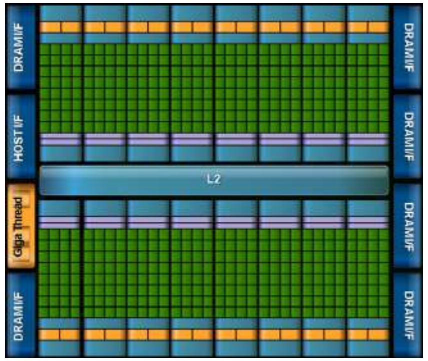
C1060 – Jun 2008

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



C2050 – Nov 2010

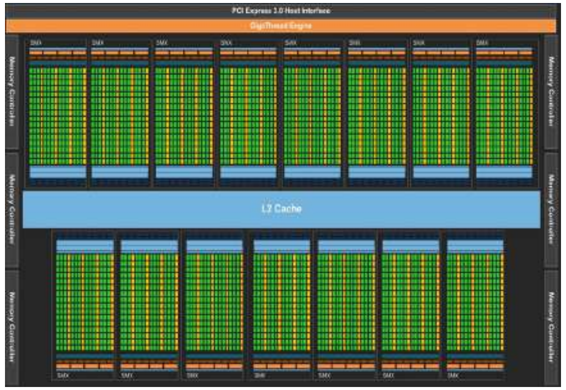
- DP / SP = 1 / 2
- 3GB RAM



Hardware complexities

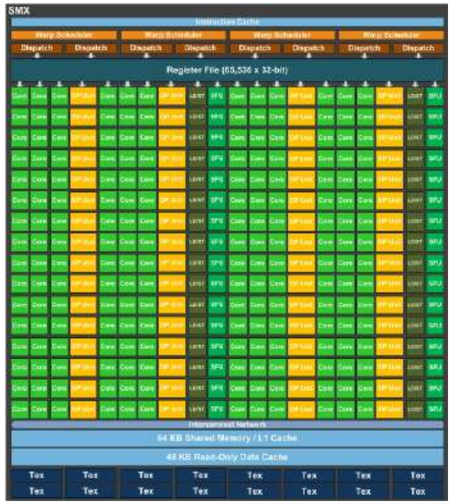
K10 – Jun 2012

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



K20 – Nov 2012

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


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

K80 – Nov 2014

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

K40 – Nov 2013

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

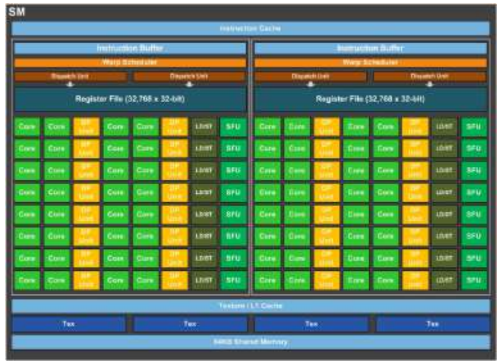
Hardware complexities

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

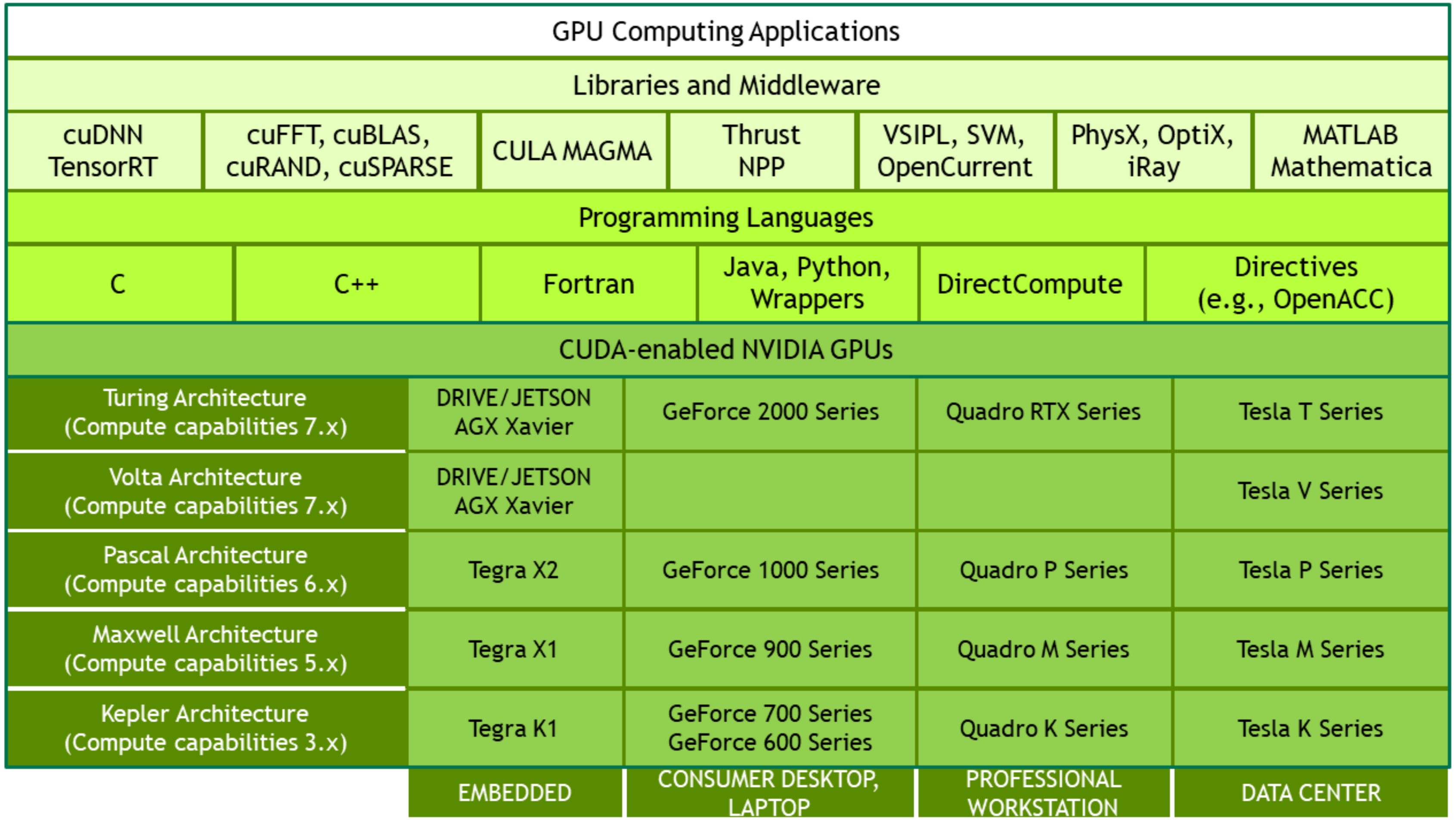
OpenACC

- 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

	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER
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

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

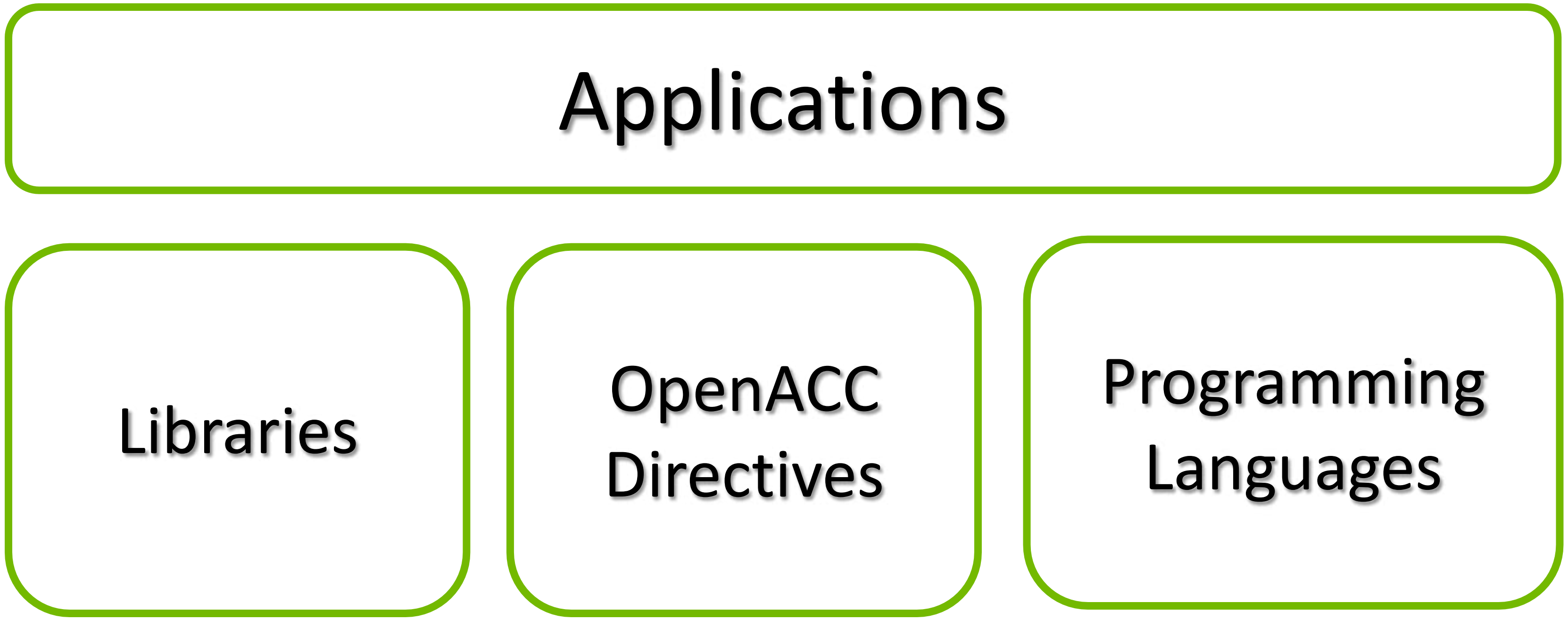
- Debugger (CUDA-gdbm CUDA-memcheck)
- Profiler (nvprof, nvvp)
- Nsight IDE for Eclipse and Visual Studio

Libraries

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

CUDA code samples

3 ways to use GPUs



Applications

Libraries

OpenACC Directives

Programming Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

Using GPU accelerated libraries

GPU accelerated libraries

Ease of use

- GPU acceleration without in-depth knowledge of GPU programming

“Drop-in”

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

Quality

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

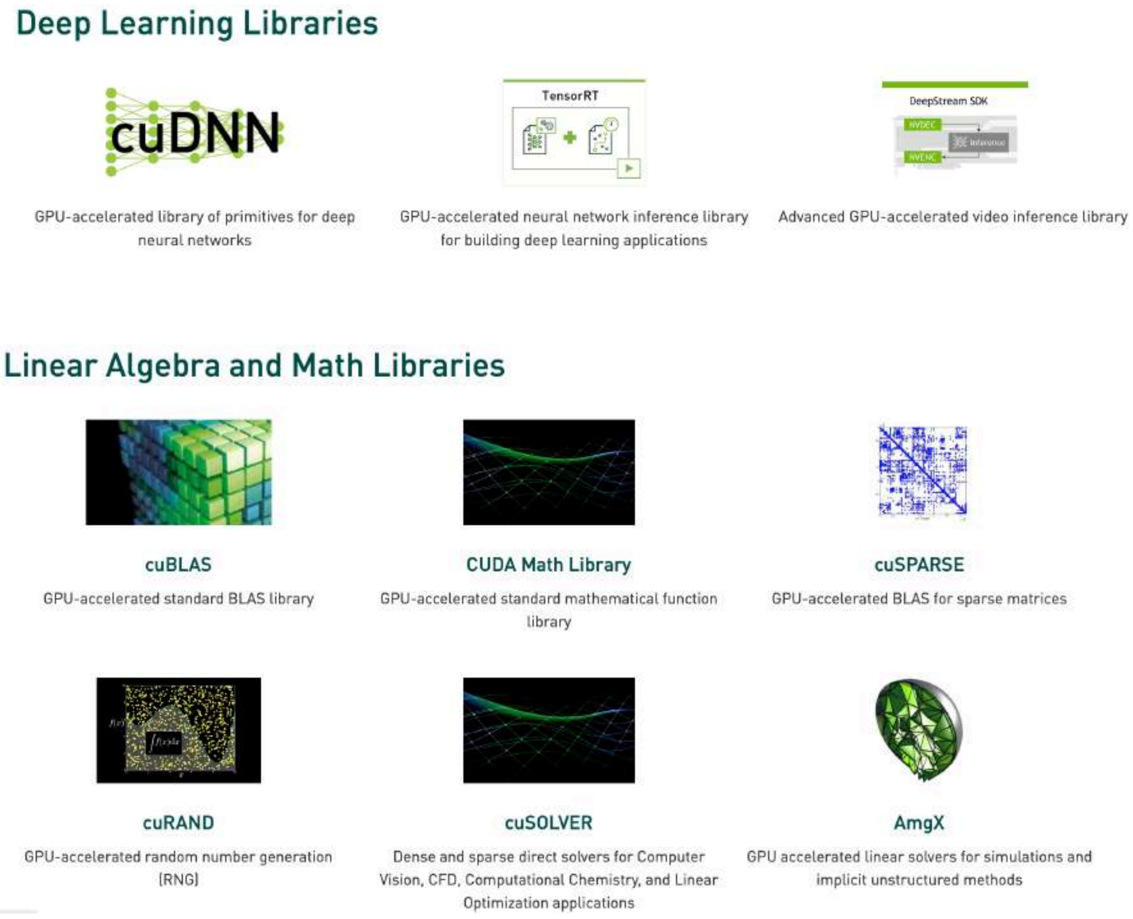
Performance

- Libraries are tuned by experts

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

GPU accelerated libraries

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



Deep Learning Libraries

cuDNN
GPU-accelerated library of primitives for deep neural networks

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

Advanced GPU-accelerated video inference library

Linear Algebra and Math Libraries

cuBLAS
GPU-accelerated standard BLAS library

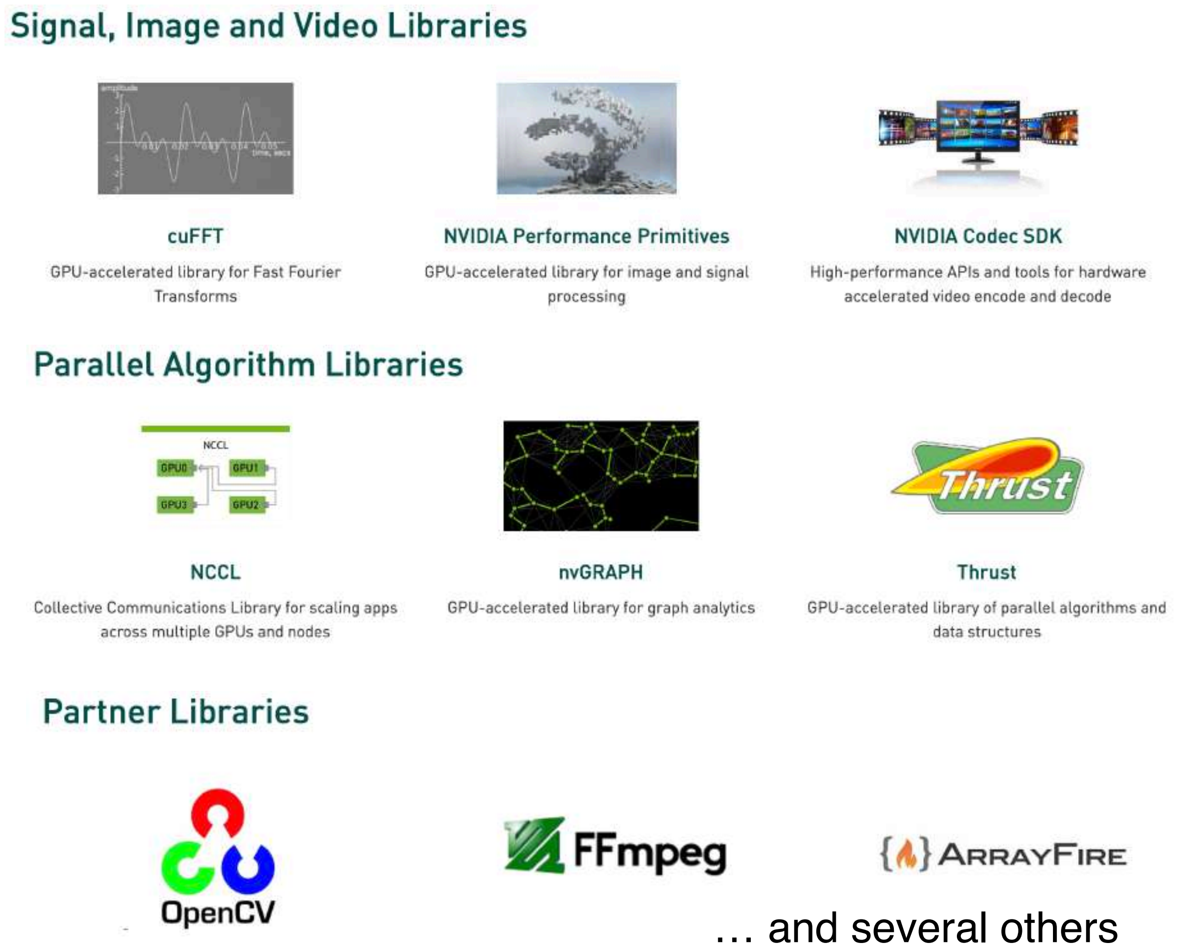
CUDA Math Library
GPU-accelerated standard mathematical function library

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

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, d_x, 1, d_y, 1);
```

saxpy =
single precision
a times **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);
```

Transfer data to GPU

```
// 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);
```

Read data back from GPU

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

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);
```

CUDA C Basics

Nvidia CUDA

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

CUDA C

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

CUDA 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 – Content overview

Learn how to write massively parallel programs for GPUs using CUDA:

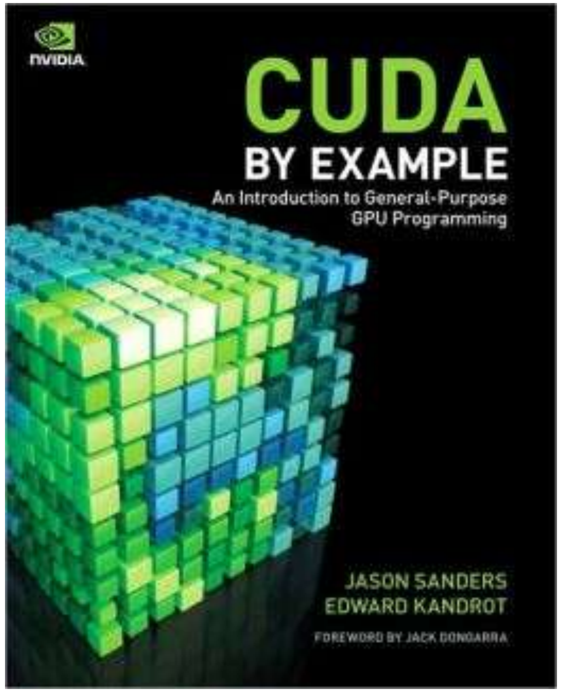
- Start from “Hello World!” program
- Write and launch CUDA C kernels
- Manage GPU memory
- Manage communication and synchronization
- Query device properties
- Error handling
- Asynchronous operations, CUDA streams

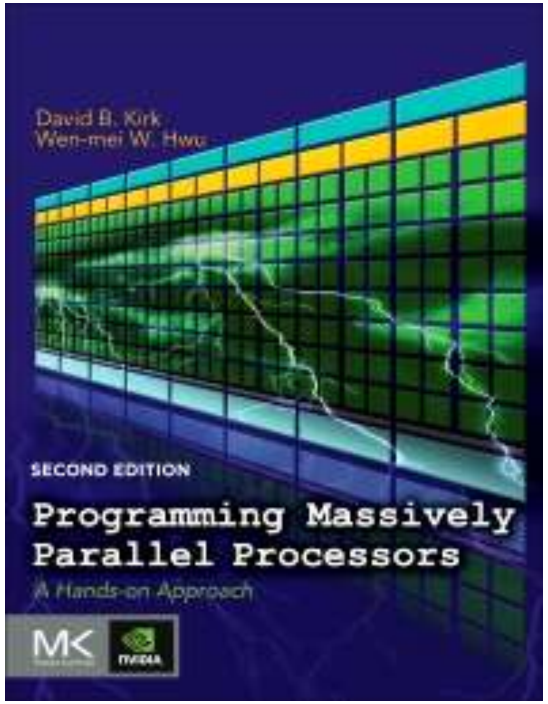
Nvidia CUDA C basics

CUDA programming guide

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

Good books to get started


NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA



Exercises on SDSC Comet – CUDA

Code samples for this course

- In SI2019 Github repository
  https://github.com/sdsc/sdsc-summer-institute-2019
  directory hpc3_gpu_cuda
- Directory **`hpc3_gpu_cuda/cuda-samples`**
  - “Hello world”
  - Addition, vector addition
  - Squaring matrix elements
  - 1D stencil
- We will discuss these simple code samples
- It is also a good idea to look at the code samples in the CUDA Toolkit and the CUDA programming guide

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

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

Accessing GPU nodes for this course

- Use alias `gpu[1234]`
- This will give you interactive access to a node with 1 GPU reserved for 1/2/3/4 hours
- These aliases have been set in your training accounts

This is an alias for

```
srun --pty --nodes=1 --ntasks-per-node=6 -p gpu-shared --gres=gpu:k80:1 \
     -t 1:00:00 --reservation=SI2019DAY4 --wait 0 /bin/bash
```

Please try to get access to a Comet GPU node now

SDSC Comet GPU nodes

- 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

- 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

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_Utilities  3_Imaging   5_Simulations  7_CUDALibraries  EULA.txt
```

SDSC Comet GPU nodes

CUDA Toolkit Samples

- 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
```

- 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_Utilities/deviceQuery
[agoetz@comet-31-16 deviceQuery]$ make
/usr/local/cuda-7.0/bin/nvcc -ccbin g++ -I../../common/inc  -m64    -gencode arch=com
```

SDSC Comet GPU nodes

CUDA Toolkit Samples

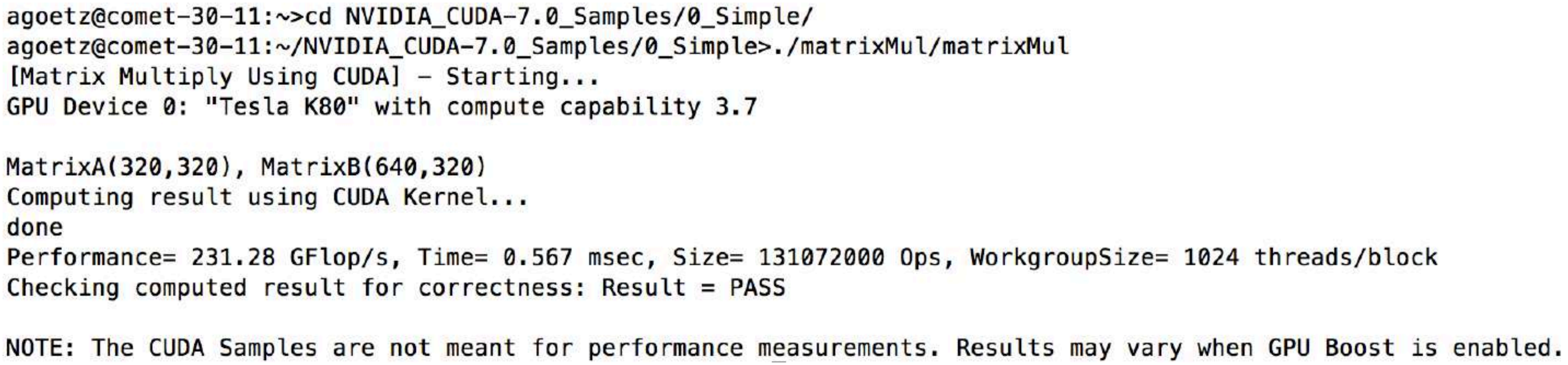
- After compiling **`deviceQuery`**, execute the program to obtain details about available CUDA devices

```
[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

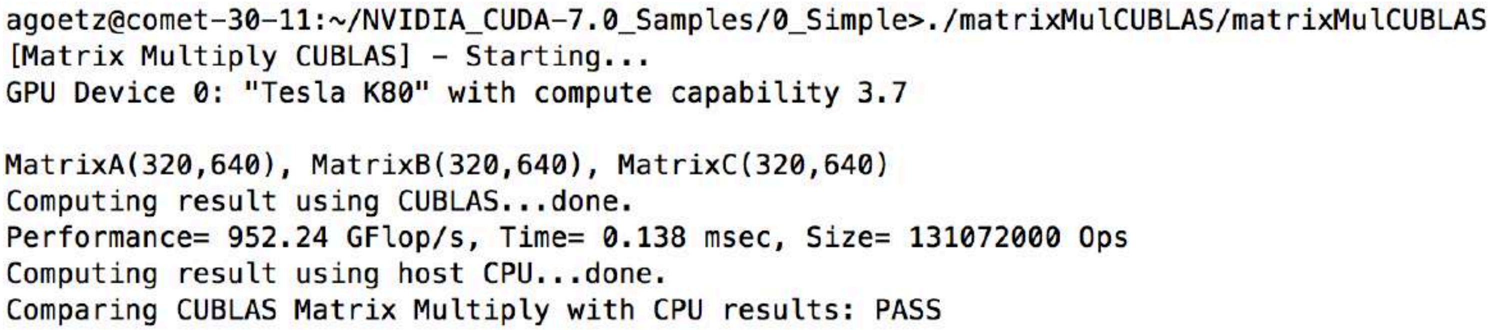
CUDA Toolkit

- 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

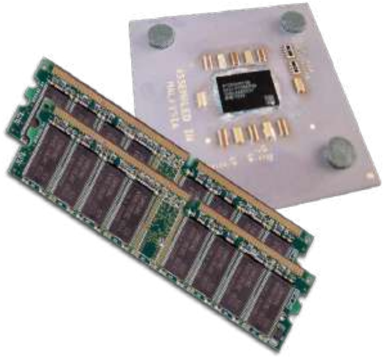


```
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples/0_Simple>./matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla K80" with compute capability 3.7

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
```

Now for real: CUDA C Basics

Heterogeneous Computing

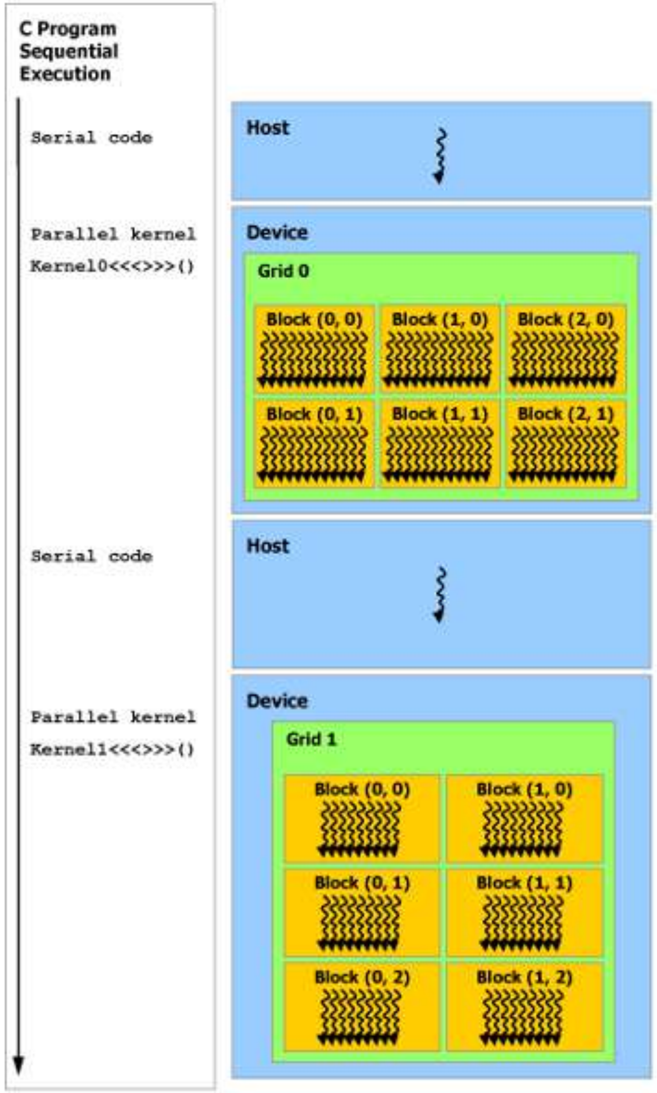
- Host The CPU and its memory
- Device The GPU and its memory
- Device code is launched from Host code



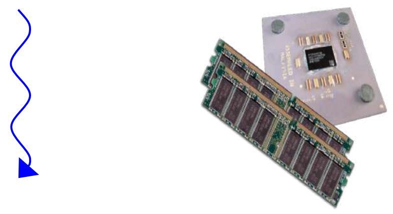
Host

Device

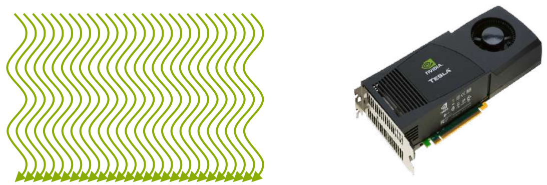
Heterogeneous Computing



serial code

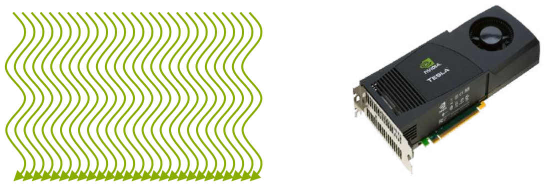


parallel code

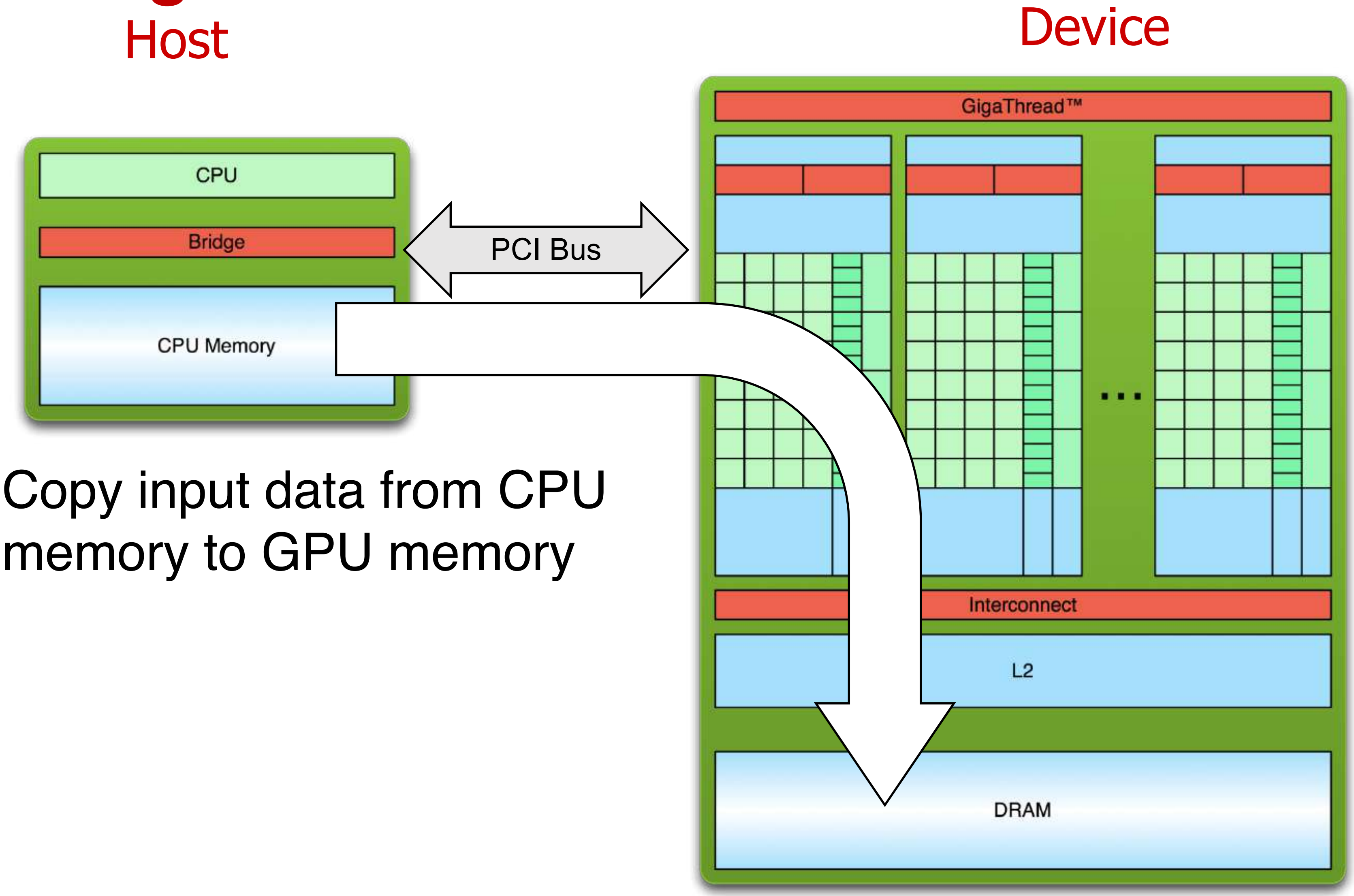


serial code

parallel code

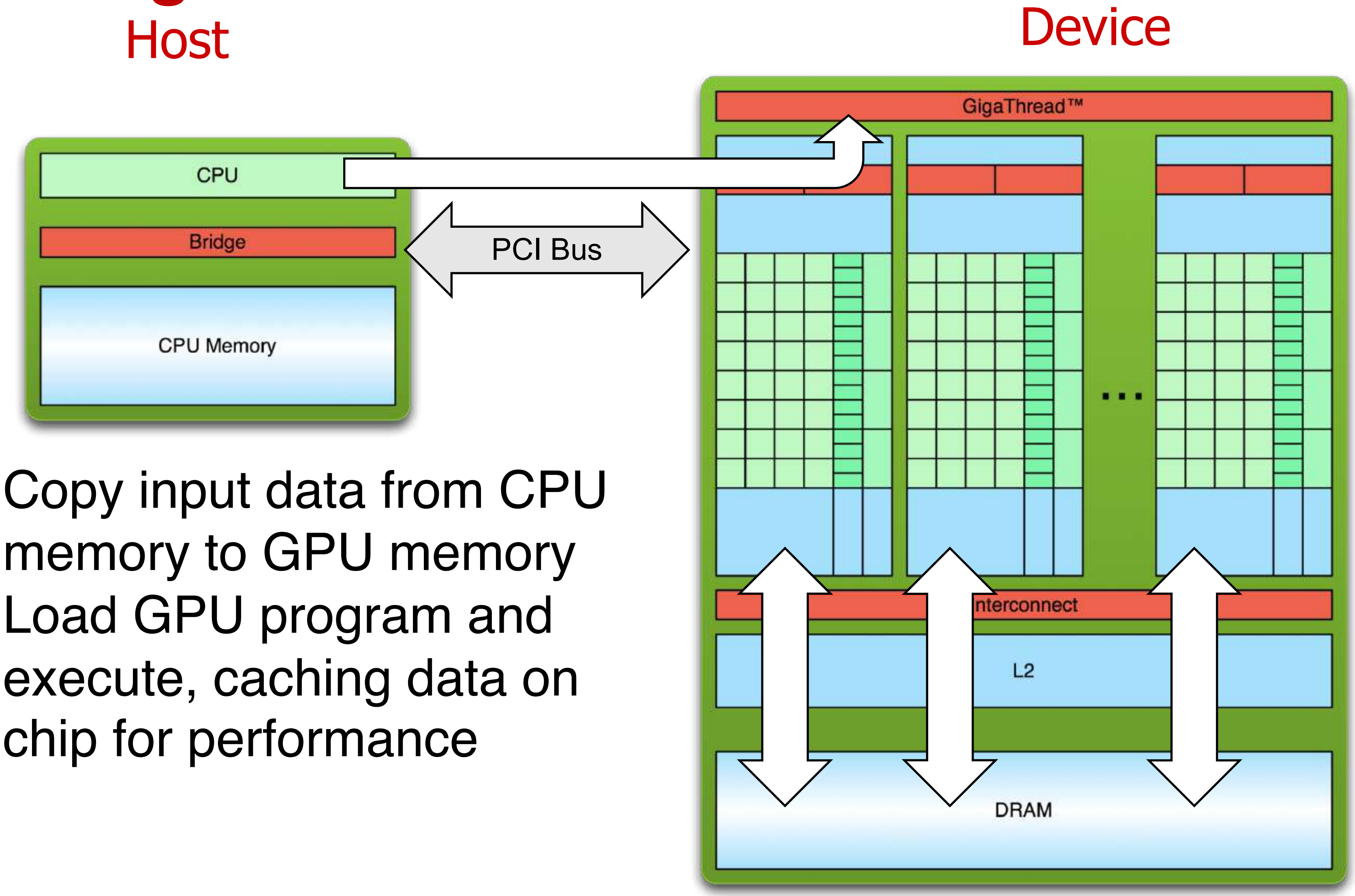


Processing Flow



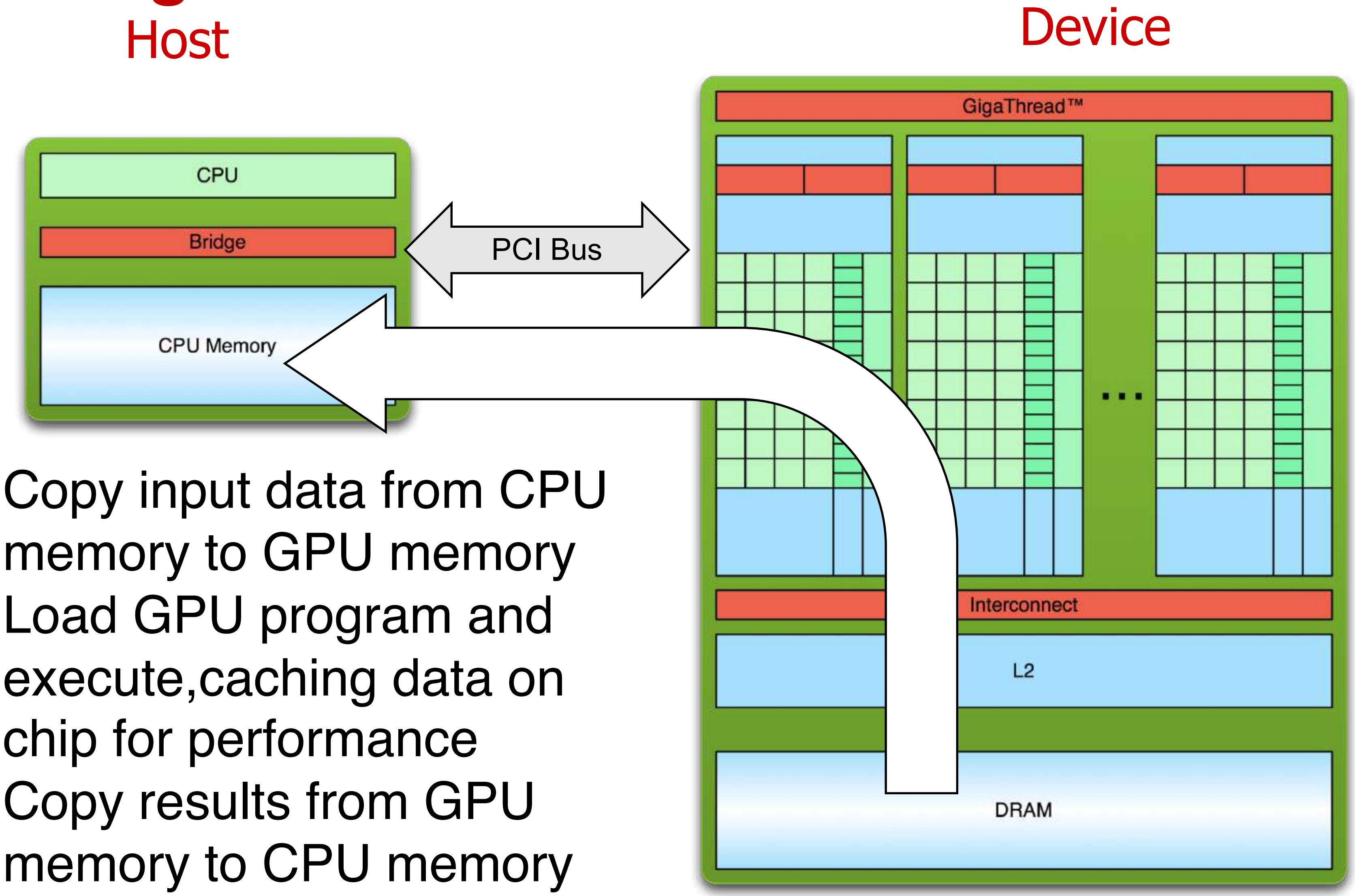
1. Copy input data from CPU memory to GPU memory

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

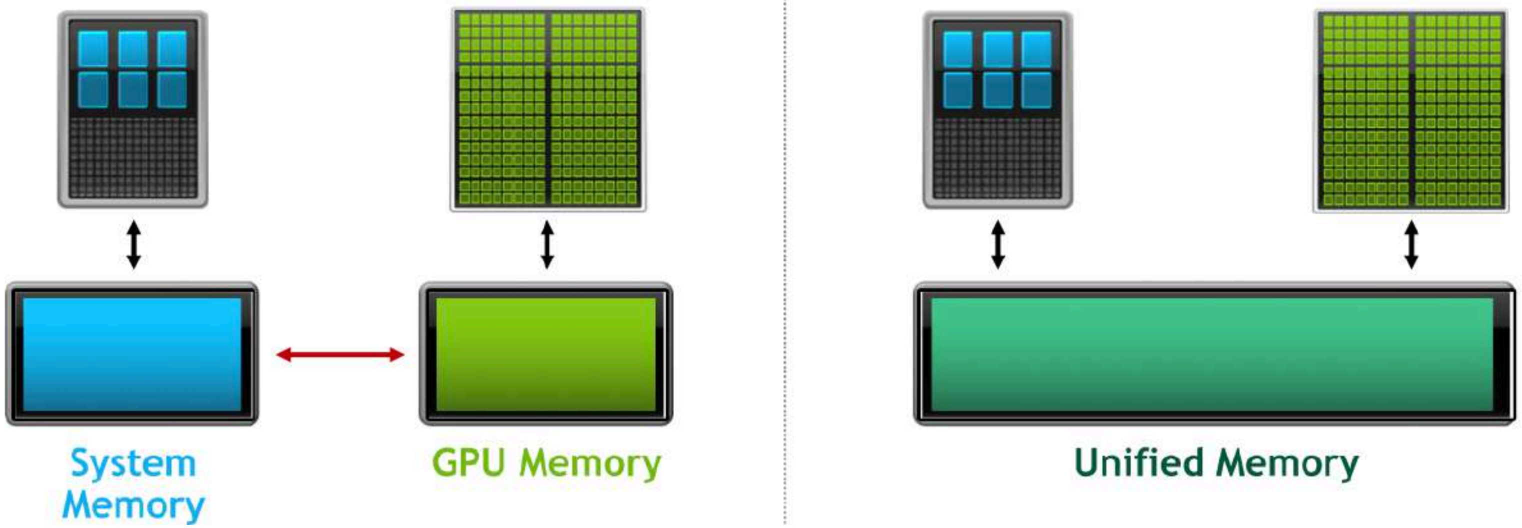


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,caching data on chip for performance
3. Copy results from GPU memory to CPU memory

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

Hello World!

- CUDA C keyword `__global__` indicates a function that
  - runs on the device (and must return **`void`**)
  - can be called from the host code

```
__global__ void my_kernel(void){
}
```

- **`nvcc`** separates source code into host and device components
  - device functions processed by **`nvcc`**
  - host functions processed by standard C compiler, e.g. **`gcc`**

Hello World!

- Triple angle brackets mark a call from host code to device code
  - Called kernel launch
  - Parameters in brackets are kernel launch configuration (explained later)

```
__global__ void my_kernel(void){
}

int main(void) {
        my_kernel<<<16,32>>>();
        printf(“Hello World!\n”);
        return 0;
}
```

- The kernel **`my_kernel()`** does nothing …
- Let’s look at writing code to be executed on the GPU

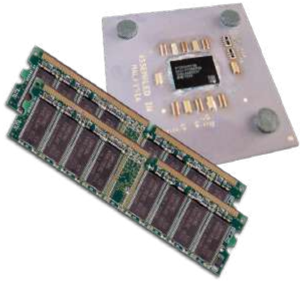
Addition on the device

```
__global__ void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

- Remember: `__global__` is a CUDA C keyword, thus
  - **`add()`** will execute on the device
  - **`add()`** will be called from the host
- Thus **`a`**, **`b`** and **`c`** must point to device memory

Memory management

- Host and device memory are separate
  - Host pointers point to CPU memory
    - Can be passed to/from device code
    - Cannot be dereferenced in device code!
  - Device pointers point to GPU memory
    - Can be passed to/from host code
    - Cannot be dereferenced in host code!
- CUDA API handles device memory
  - `cudaMalloc(), cudaFree(), cudaMemcpy()`
  - equivalent to C `malloc(), free(), memcpy()`



Addition on the device

```
__global__ void add(int *a, int *b, int *c){
        *c = *a + *b;
}
```

- How do we reserve memory on the device?
- How do we transfer data from the host to the device?
- This happens in the host C code that launches the kernel. Let’s see how this works…

Addition on the device

```
int main(void){
    int h_a, h_b, h_c;     // host copies
    int *d_a, *d_b, *d_c; // device copies
    int size = sizeof(int);

    // Allocate memory on device
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    h_a = 5;
    h_b = 7;

    // Copy input data to device
    cudaMemcpy(d_a, &h_a, size,
                   cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &h_b, size,
                   cudaMemcpyHostToDevice);

    // Launch add() kernel
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy results back to host
    cudaMemcpy(&h_c, d_c, size,
                    cudaMemcpyDeviceToHost);

    // Deallocate memory
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    printf(“%d + %d = %d”, h_a, h_b, h_c);
    return 0;
}
```

Basic CUDA workflow

- Allocate memory on the device
- Copy input data to the device
- Launch CUDA kernel on the device
- Copy results back to the host
- Deallocate memory on the device

Let’s move on to parallel computing on the GPU using CUDA

Parallelization with CUDA

- GPU computing is about massive parallelism
    - How do we run code in parallel using CUDA?
- Instead of executing the kernel add() once, we execute it N times in parallel
- The central idea defining GPU computing:
    - Kernels look like serial programs
    - Write programs as if they run on a single thread
    - The GPU will run that program on many threads

Parallelization with CUDA

- Executing **`add()`** N times

```
add<<<1,1>>>();   // launch 1 copy
     ↓
add<<<N,1>>>();   // launch N copies
```

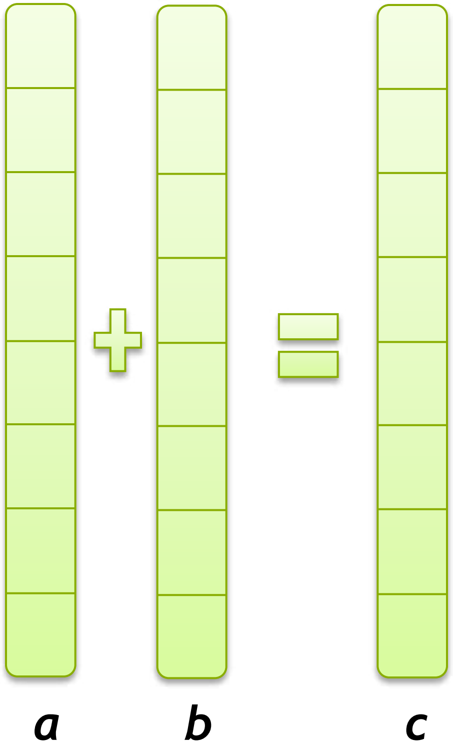
- The GPU is good at
  - efficiently launching lots of threads
  - running lots of threads in parallel
    (many more than processors on the device)
- But why launch N identical copies?

Vector addition

- CPU code (serial) uses a loop

```
#define N 512
void vadd_cpu(int *a, int *b, int *c){
        int i;
        for (i=0; i<N; i++){
                c[i] = a[i] + b[i];
        }
}
```

- The GPU does this in parallel by running N copies of the `add()` kernel, each copy working on a different vector element



Parallel vector addition (1)

- Parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c){
        int tid = blockIdx.x;
        c[tid] = a[tid] + b[tid];
}
```

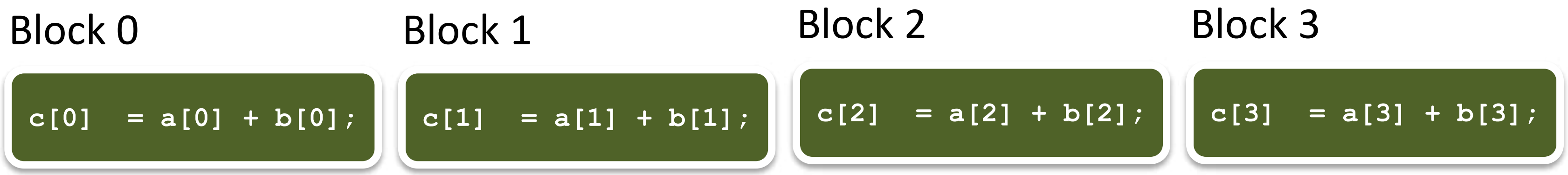
- Each parallel invocation of **`add()`** is called a block. The set of blocks is called a grid
- Each kernel instance knows its block index
- By using its index, each block operates on different data

Parallel vector addition (1)

- We launch N blocks of the **`add()`** kernel

```
// launch N copies
add<<<N,1>>>(d_a, d_b, d_c);
```

- Kernel call needs to be consistent with kernel implementation
- On the device, each block can execute in parallel, depending on the number and type of available multiprocessors



```
c[0]   = a[0] + b[0];
c[1]   = a[1] + b[1];
c[2]   = a[2] + b[2];
c[3]   = a[3] + b[3];
```

Parallel vector addition (1)

```
// CUDA kernel for vector addition
__global__ void add(int *a, int *b, int *c){
        int tid = blockIdx.x;
        c[tid] = a[tid] + b[tid];
}

#define N 512
int main(void){
        int h_a[N], h_b[N], h_c[N]; // host copies
        int *d_a, *d_b, *d_c;        // device copies
        int size = N * sizeof(int);

        // Allocate memory on device
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
```

Parallel vector addition (1)

```
    // Setup input values
    get_input_vectors(h_a, h_b);

    // Copy input data to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch N blocks of the add() kernel
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy results back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Deallocate memory
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}
```

Review

- Distinguish host and device code
  - host = CPU
  - device = GPU
- CUDA keyword `__global__` declares functions as device code
  - execute on device
  - called from host
- Parameters can be passed from host code to device function

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Kernels are launched by CPU and execute in parallel
  - Launch N blocks (copies) of add() with

    `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

CUDA blocks and threads

- Blocks contain threads executing in parallel
- Parallelized **`add()`** kernel using threads

```
__global__ void add(int *a, int *b, int *c){
        int tid = threadIdx.x;
        c[tid] = a[tid] + b[tid];
}
```

- Each kernel instance knows its index
- Parallel kernel call

```
// launch N copies
add<<<1,N>>>(d_a, d_b, d_c);
```

CUDA blocks and threads

- Blocks contain threads executing in parallel
- Parallelized **`add()`** kernel using threads

```
__global__ void add(int *a, int *b, int *c){
        int tid = threadIdx.x;
        c[tid] = a[tid] + b[tid];
}
```

- Each kernel instance knows its index
- Parallel kernel call

```
// launch N copies
add<<<1,N>>>(d_a, d_b, d_c);
```

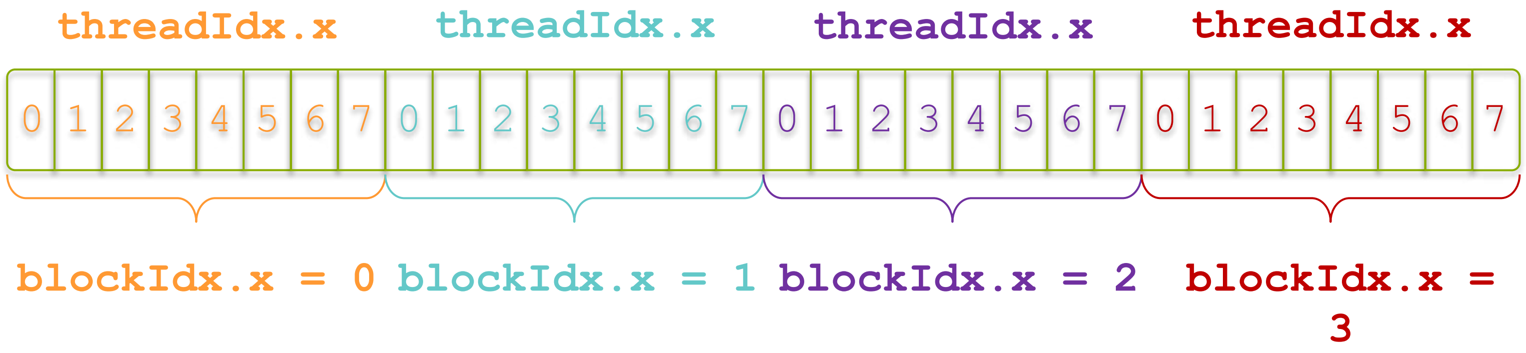
- We can combine blocks and threads
  - need to take care with indexing

- CUDA supports 3D blocks and threads (more later)
- Number of allowed threads per block and concurrent blocks is limited by hardware (up to 2048 threads per block on current GPUs)
- Threads and blocks map to the underlying hardware
- All threads in a block are guaranteed to execute on the same streaming multiprocessor (SM), thus sharing resources
- Threads within a block can communicate and synchronize
- **Note:** Thread block size should be a multiple of warp size (32) for performance reasons since kernels issue instructions in warps

Indexing with blocks and threads

Example to compute offset into an array

- 1 element per thread
- 8 threads per block



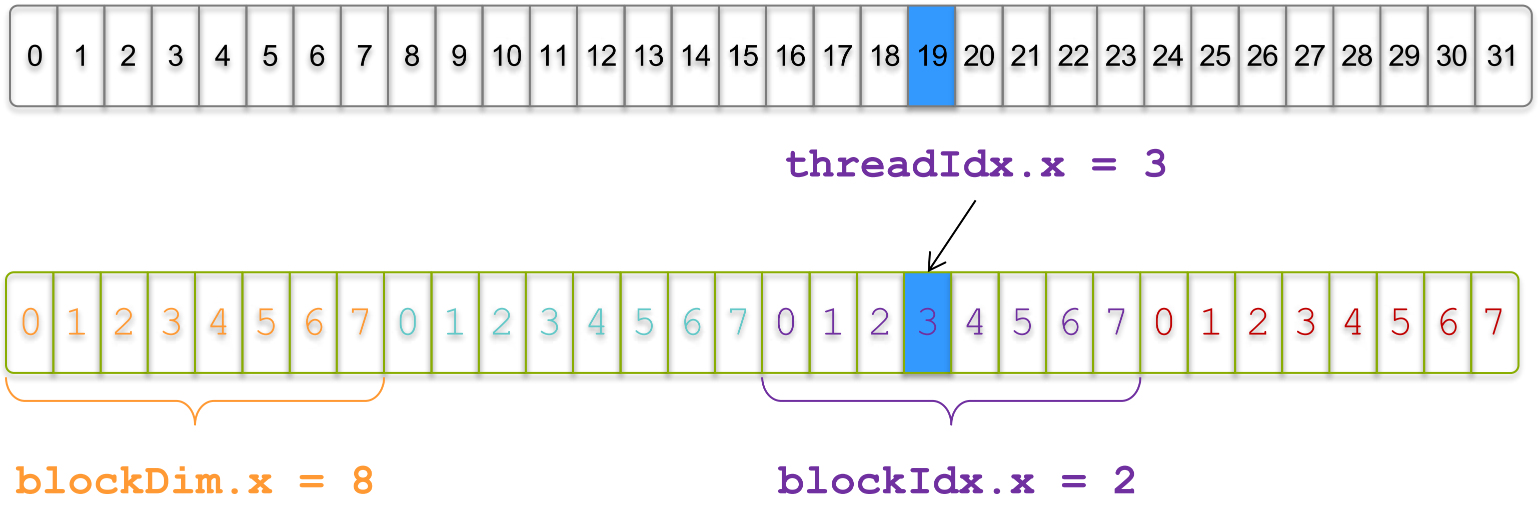
`threadIdx.x` `threadIdx.x` `threadIdx.x` `threadIdx.x`

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7

`blockIdx.x = 0` `blockIdx.x = 1` `blockIdx.x = 2` `blockIdx.x = 3`

- Built-in variable **`blockDim.x`** contains the number of threads per block
  - this makes it possible to calculate a unique index (e.g. offset into an array) for each kernel

Indexing with blocks and threads



- Calculate a unique index (Example: blue field above has index 19)

```
int tid = threadIdx.x + blockIdx.x * blockDim.x
        = 3           + 2          * 8
        = 19
```

Parallel vector addition (2)

- Parallelized **add()** using blocks and threads

```
__global__ void add(int *a, int *b, int *c){
        int tid = threadIdx.x + blockDim.x * blockIdx.x;
        c[tid] = a[tid] + b[tid];
}
```

- Corresponding kernel call

```
// launch N/TPB copies with
// TPB threads per block
add<<<N/TPB,TPB>>>(d_a, d_b, d_c);
```

Handling arbitrary vector sizes (1)

- Problem size **N** is typically **not** a multiple of our chosen **blockDim.x**
- Launch a sufficient number of kernels

```
// launch at least N/TBP copies with
// TPB threads per block
add<<<(N+TBP-1)/TPB,TPB>>>(d_a, d_b, d_c, N);
```

- Ensure to stay within array boundaries

```
__global__ void add(int *a, int *b, int *c, int n){
        int tid = threadIdx.x + blockDim.x * blockIdx.x;
        if (tid < n)
             c[tid] = a[tid] + b[tid];
}
```

- If n is not a multiple of the number of threads per block TBP, a few threads will do no-ops

Review

- We write a kernel that looks like it runs on one thread
- We can launch that kernel on any number of threads
  - Use **`kernel<<<(N+TPB-1)/TPB, TPB>>>()`**
- Each thread knows its index in the block and grid
  - use **`blockIdx.x`** to get the block index
  - use **`blockDim.x`** to get the block size
  - use **`threadIdx.x`** to get the thread index

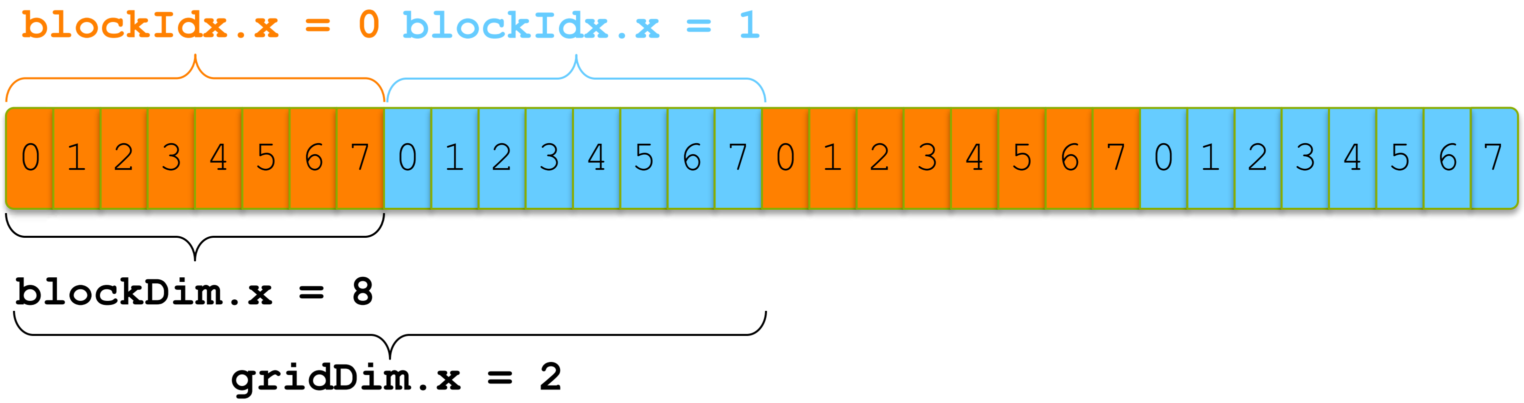
```
tid = threadIdx.x + blockIdx.x * blockDim.x
```

Handle arbitrary vector sizes (2)

- Maximum grid and block size is limited by hardware
- We thus need to
  - launch a fixed number of blocks and threads
  - rewrite our kernel for fixed grid and block size
- Built-in variable **`gridDim.x`** contains number of blocks in grid
- We can use this to compute strides
- For many kernels, performance will be optimal for a GPU-hardware dependent combination of grid / block size
  - Query hardware with CUDA calls to determine optimal kernel launch configuration at runtime

Handle arbitrary vector sizes (2)

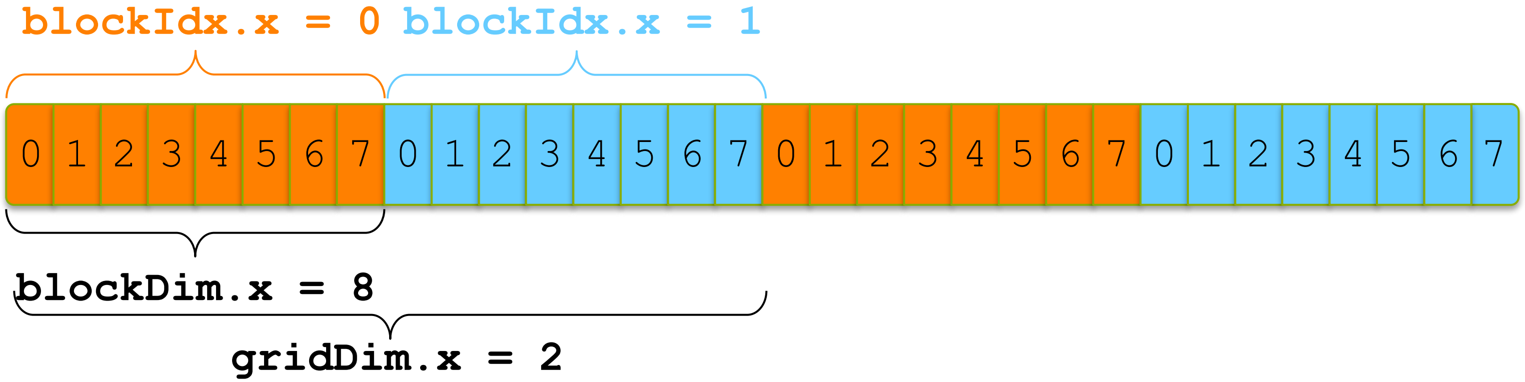
- Example: If we launch 2 blocks with 8 threads each, then kernels with
  - `blockIdx.x=0` need to work on blocks 0 and 2
  - `blockIdx.x=1` need to work on blocks 1 and 3



- Each kernel needs to know the stride required for accessing its data elements

```
int stride = blockDim.x * gridDim.x
```

Handle arbitrary vector sizes (2)



```
__global__ void add(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
        }
}
```

- We can now launch a fixed number of kernels:

```
add<<<NBL,TPB>>>(d_a, d_b, d_c, N);
```

Multi-dimensional indexing

- CUDA supports
  - 3D grids of blocks and
  - 3D blocks of threads
- Convenient for mapping multi-dimensional problems (bitmaps, matrix operations etc)
- `grid3` data type, dimensions default to 1
- Example: launch grid of `(bx*by*bz)` blocks of `(tx*ty*tz)` threads with

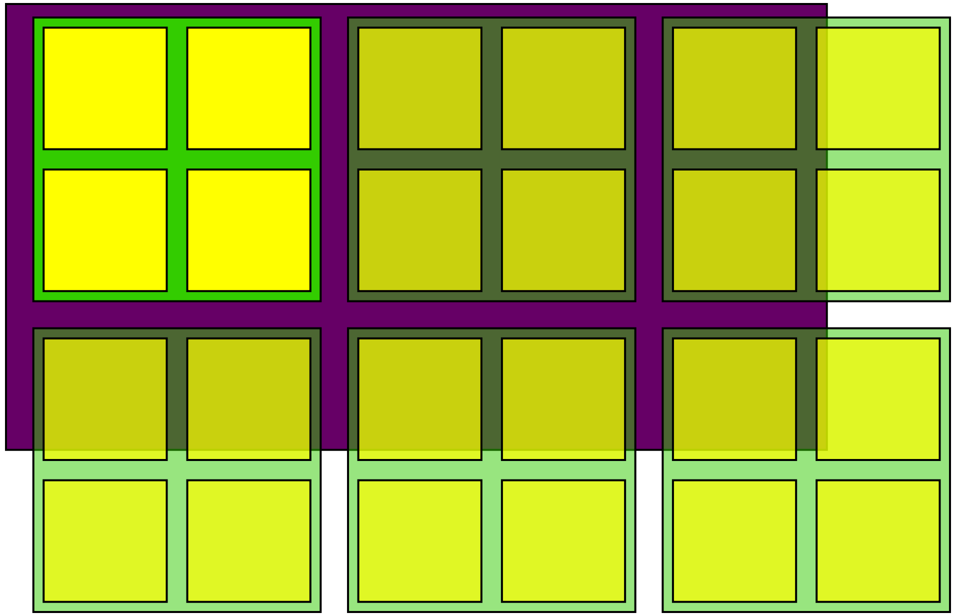
```
kernel<<<dim3(bx,by,bz),dim3(tx,ty,tz)>>>();
```

Multi-dimensional indexing

- Get grid dimension from
  `gridDim.x, gridDim.y, gridDim.z`
- Get block index in grid from
  `blockIdx.x, blockIdx.y, blockIdx.z`
- Get block dimension from
  `blockDim.x, blockDim.y, blockDim.z`
- Get thread index in block from
  `threadIdx.x, threadIdx.y, threadIdx.z`
- Use these to determine data access offsets and strides

Example: 2D array

- Example: Kernel that squares a 2D array using a 2D grid of 2D blocks:



- Matrix (purple)
- 2D grid (green, 2x2)
- 2D blocks (yellow)
- (threads not shown)

Example: 2D array

- Example: Kernel that squares a 2D array using a 2D grid of 2D blocks, assuming linear storage in memory:

```
__global__ void square(int *arr, int maxrow, int maxcol){
   // indices and strides
   int rowinit = threadIdx.x + blockDim.x * blockIdx.x;
   int colinit = threadIdx.y + blockDim.y * blockIdx.y;
   int rowstride = gridDim.x * blockDim.x;
   int colstride = gridDim.y * blockDim.y;

   // operate on all 2D “submatrices”
   for (int row = rowinit; row < maxrow; row += rowstride) {
      for (int col = colint; col < maxcol; col += colstride) {
         pos = row * maxcol + col
         arr[pos] *= arr[pos]
      }
   }
}
```

Example: 2D array

- We can launch the kernel for example with a grid of (16*16) blocks of (16*16) threads, i.e. a total of 256*256 = 65,536 concurrent threads

```
#define NROW 2048
#define NCOL 512
int main(void){
    int h_a[NROW][NCOL];    // host copy
    int *d_a;               // device copy
    int size = NROW * NCOL * sizeof(int);

    // Allocate memory on device
    cudaMalloc((void **)&d_a, size);

    // Setup input values
    get_input_array(h_a);

    // Copy input data to device
    cudaMemcpy(d_a, h_a, size,
                cudaMemcpyHostToDevice);

    // Launch square() kernel
    dim3 gridSize(16,16);
    dim3 blockSize(16,16);
    square<<<gridSize,blockSize>>>(d_a, NROW, NCOL);

    // Copy results back to host
    cudaMemcpy(h_a, d_a, size,
                cudaMemcpyDeviceToHost);

    // Deallocate memory
    cudaFree(d_a);

    return 0;
}
```

Unified memory

- Great to get started, simplifies programming

```
cudaMallocManaged(…);
```

- CUDA keeps track of memory location and migrates data from device to host and vice versa as required
- Developer needs to ensure that no race conditions are caused by simultaneous access to host/device memory
    - Pre-Pascal architecture will segfault; Pascal give wrong results
    - Therefore synchronize CPU/GPU (wait for kernel to finish):

```
cudaDeviceSynchronize();
```

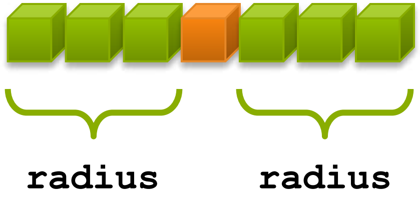
Example: 2D array with unified memory

```
#define NROW 2048
#define NCOL 512
int main(void){

    int size = NROW * NCOL * sizeof(int);
    int *array

    // Allocate managed memory, get data
    cudaMallocManaged(&array, size);
    get_input_array(array);

    // Launch square() kernel
    dim3 gridSize(16,16); dim3 blockSize(16,16);
    square<<<gridSize,blockSize>>>(array, NROW, NCOL);

    // Wait for kernel to finish before accessing data
    cudaDeviceSynchronize();
    print_results(array);
    cudaFree(array)
}
```

Communication among threads – shared memory

Example: 1D stencil

- 1D stencil for 1D array:
  - Each output element is the sum of input elements within a given radius
- If the radius is 3, then each output element is the sum of 7 input elements:

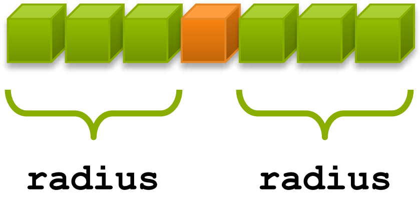


$$y_i' = y_i + \sum_{j=1}^{3} y_{i-j} + \sum_{j=1}^{3} y_{i+j}$$

Communication among threads – shared memory

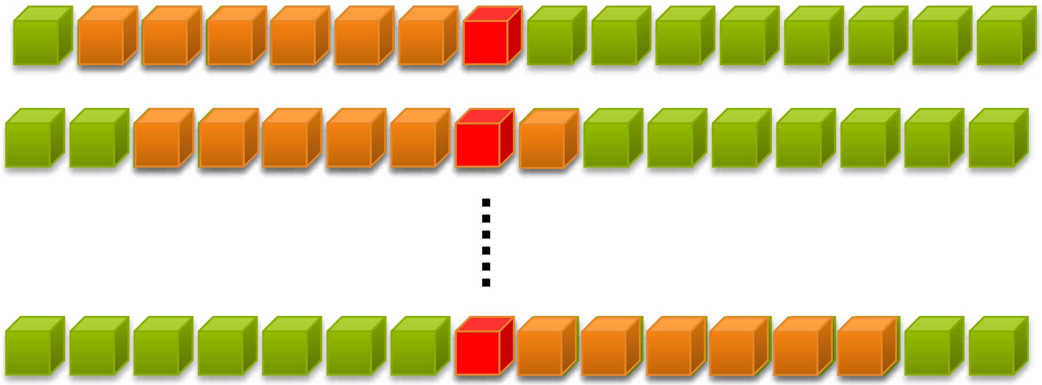
Example: 1D stencil

- 1D stencil for 1D array:
  - Each output element is the sum of input elements within a given radius
- If the radius is 3, then each output element is the sum of 7 input elements:



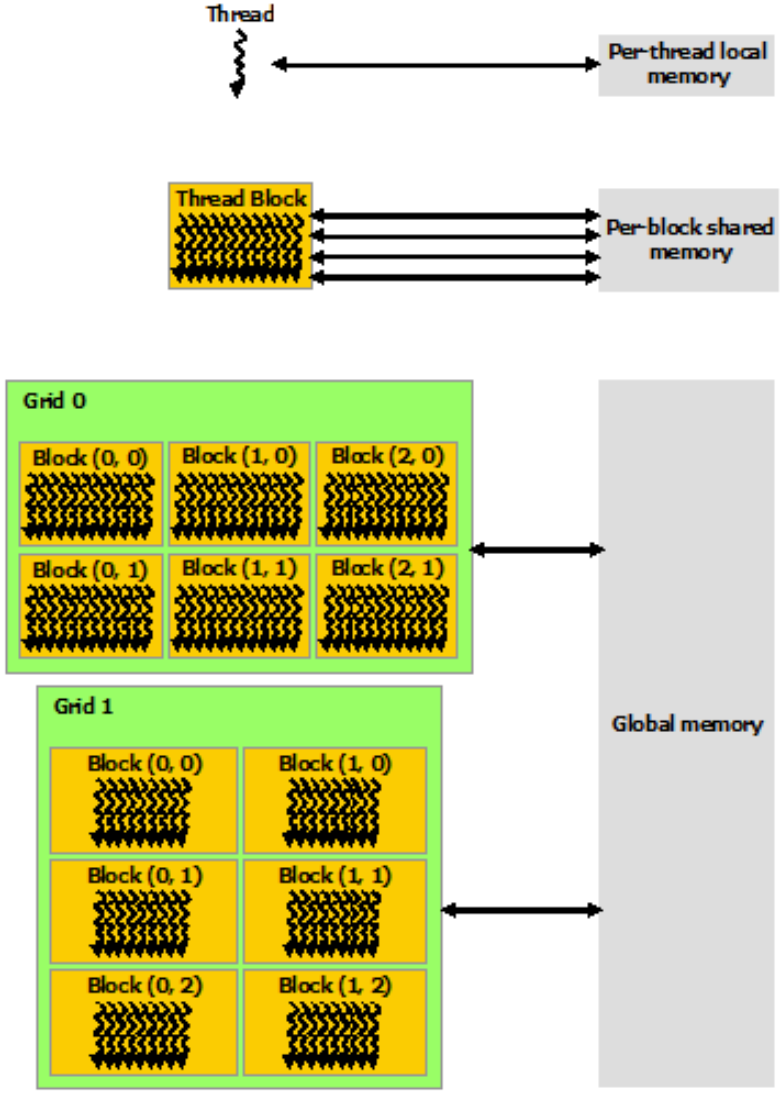
$$y_i' = y_i + \sum_{j=1}^{3} y_{i-j} + \sum_{j=1}^{3} y_{i+j}$$

- Each thread processes one output element
  - `blockDim.x` elements are processed per block
- As a consequence, input elements have to be read several times from slow global memory
  - with radius 3, each input element is read 7 times!



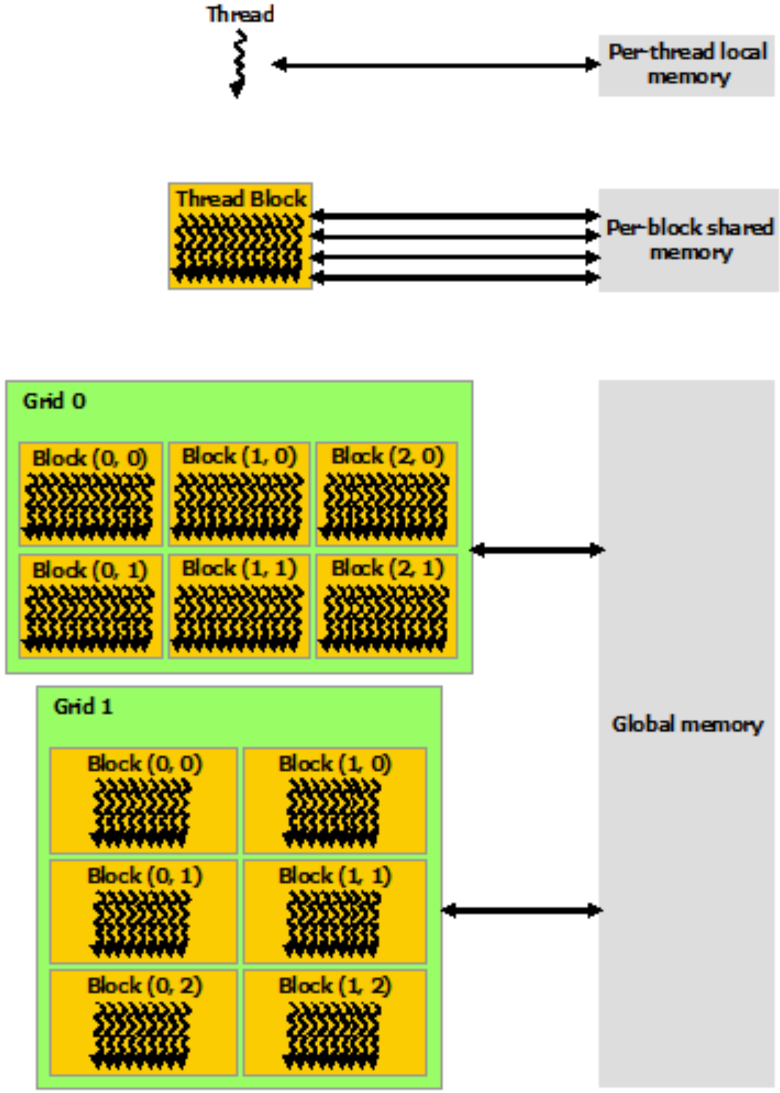
Communication among threads – shared memory

- Within a block, threads can share data via shared memory
- This is very fast on-chip memory
- Shared memory is user-managed
- Declare as `__shared__`, will be allocated per block
- Data in shared memory is not visible to other blocks



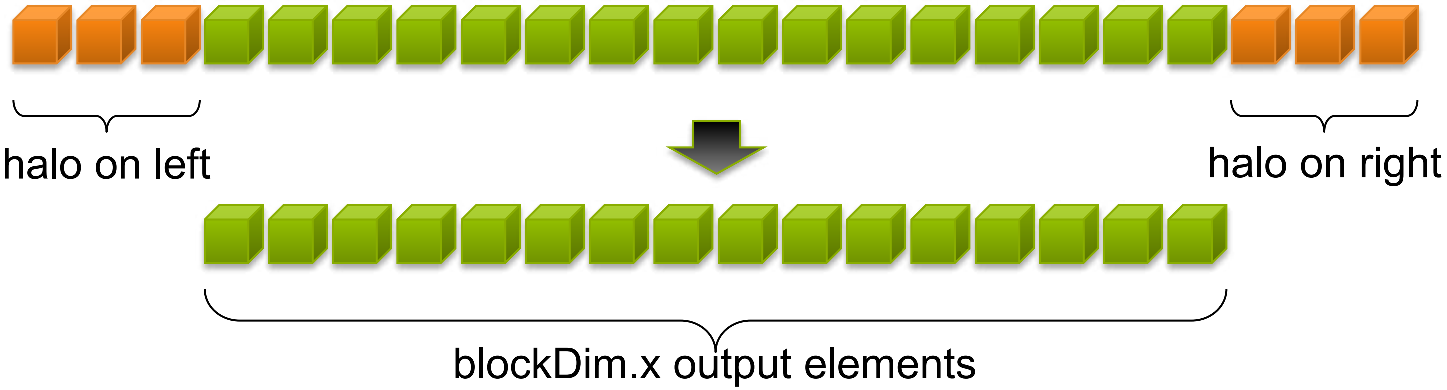
CUDA memory hierarchy

Memory	Latency (cycles)	Cached	Privacy
Global	100s	Yes	Application
Local	100s	Yes	Thread
Constant	1s-100s	Yes	Application
Texture	1s-100s	Yes	Application
Shared	1	–	Block
Register	1	–	Thread



1D stencil using shared memory

- Cache data for use by different threads in shared memory
  - Read **`(blockDim.x + 2 * radius)`** input elements from global to shared memory
  - Compute **`blockDim.x`** output elements
  - Write **`blockDim.x`** output elements to global memory
  - Each block needs a “halo” of **`radius`** elements at each boundary



1D stencil using shared memory

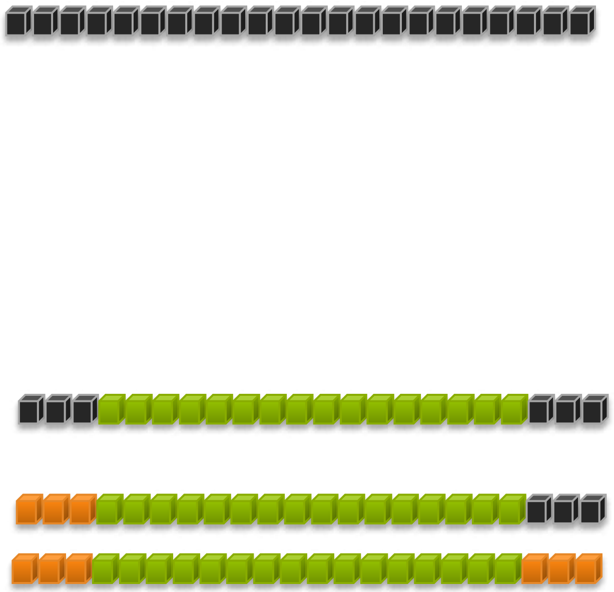
```
__global__ void stencil_1D(int *in, int *out){

    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];

    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS
    int tid = threadIdx.x







}
```

Data in shared memory

1D stencil using shared memory

```
__global__ void stencil_1D(int *in, int *out){

    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];

    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS
    int tid = threadIdx.x

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (tid < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }







}
```

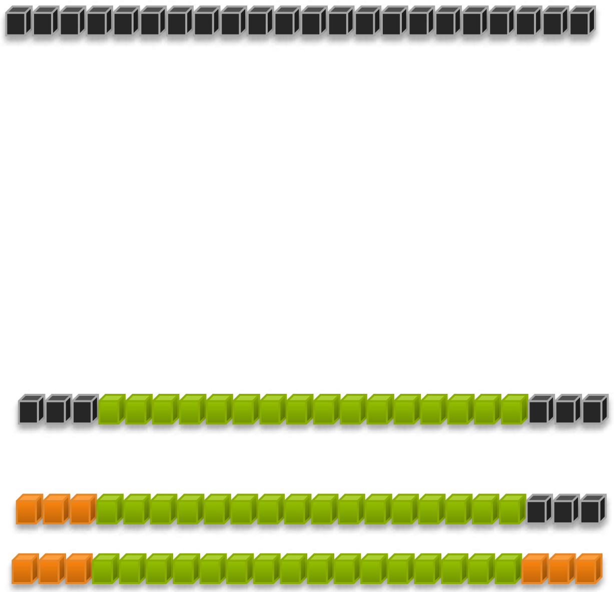
Data in shared memory



1D stencil using shared memory

```
__global__ void stencil_1D(int *in, int *out){

    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];

    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS
    int tid = threadIdx.x

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (tid < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }
    // Store the result
    out[gindex] = result;
}
```

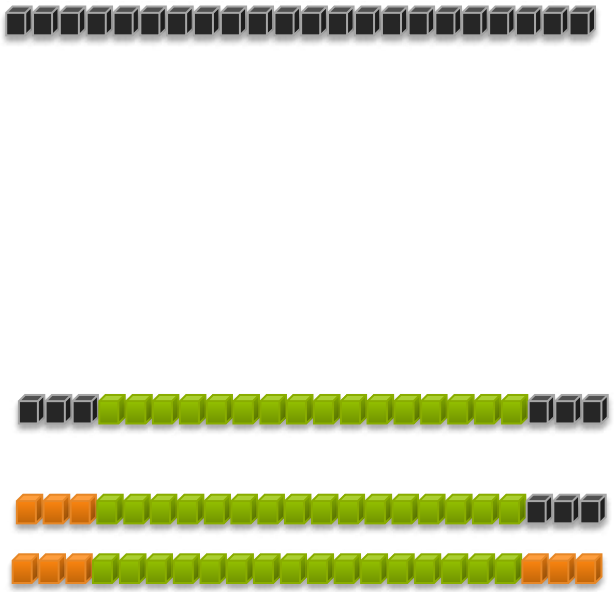
Data in shared memory



1D stencil using shared memory

```
__global__ void stencil_1D(int *in, int *out){

    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];

    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS
    int tid = threadIdx.x

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (tid < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }
    // Store the result
    out[gindex] = result;
}
```

Data in shared memory

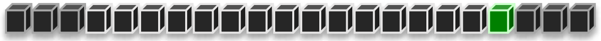


Unfortunately the code as it is will not work – Why ?

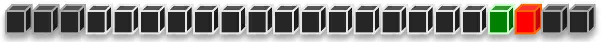
1D stencil using shared memory

- We have a data race!
- Suppose thread 15 reads the halo before thread 0 has fetched it:

```
// Read input elements into shared memory
temp[lindex] = in[gindex]; // store at temp[18]
```



```
if (tid < RADIUS) {         // skipped by thread 15 (tid > RADIUS)
        temp[lindex – RADIUS] = in[gindex – RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex+1];  // load from temp[19]
```



Thread synchronization in a block

- To avoid race conditions we need to synchronize our threads in the block
    - used to prevent RAW / WAR / WAW hazards
- `void __syncthreads();`
- All threads in the block must reach the barrier
    - Similar to **`MPI_Barrier()`**
    - In conditional code, the condition must be uniform across the block to avoid deadlocks

1D stencil kernel using shared memory

```
__global__ void stencil_1D(int *in, int *out){

    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];

    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS
    int tid = threadIdx.x

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (tid < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize to ensure that all data is available
    __synthreads();
```

1D stencil kernel using shared memory

```
    // Synchronize to ensure that all data is available
    __synthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }
    // Store the result
    out[gindex] = result;
}
```

This kernel

- improves performance by using shared memory
- avoids race conditions by synchronizing the threads within a block

1D stencil kernel using shared memory

```
    // Synchronize to ensure that all data is available
    __synthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }
    // Store the result
    out[gindex] = result;
}
```

This kernel

- improves performance by using shared memory
- avoids race conditions by synchronizing the threads within a block

- Use **`__shared__`** to declare a variable / array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use **`__syncthreads()`** as a barrier
  - Required to prevent data hazards / race conditions

Constant memory

In addition to shared memory, there is constant memory

- Read-only during kernel execution
- Located off-chip in global memory but accessed via dedicated hardware
  - broadcasts to all threads in a half-warp (16 threads), saving bandwidth
  - cached
- Define constant memory
  - `__constant__ int c_a[dimension];`
- Copy data into constant memory
  - `cudaMemcpyToSymbol(c_a, h_a, size);`

CUDA basics summary

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.

CUDA basics summary

Threads, blocks, grids, warps

Grids

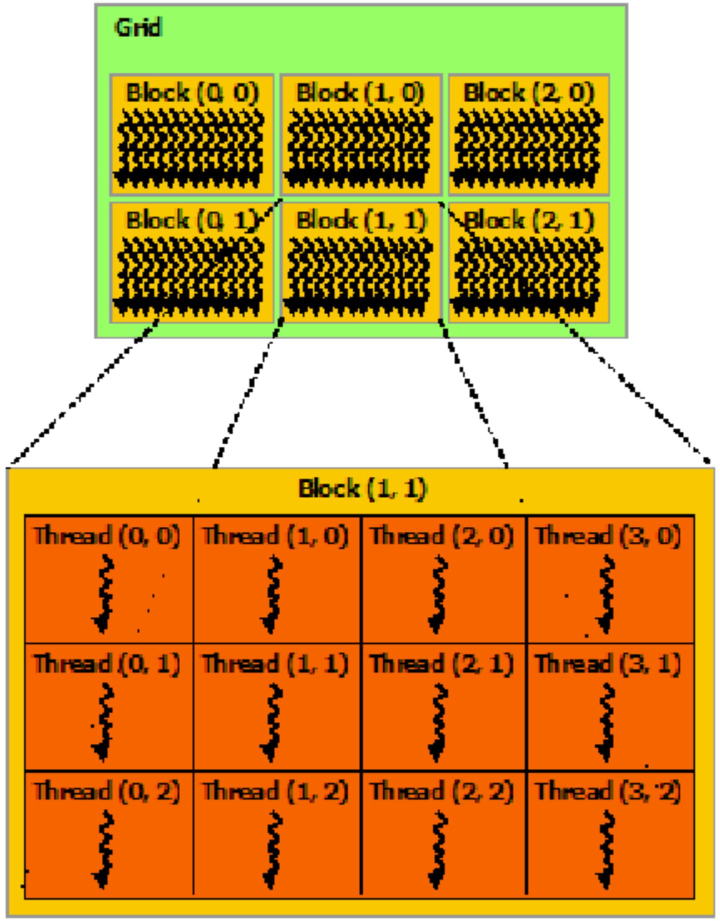
- Grids map to GPUs

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).



CUDA basics summary

CUDA built-in variables

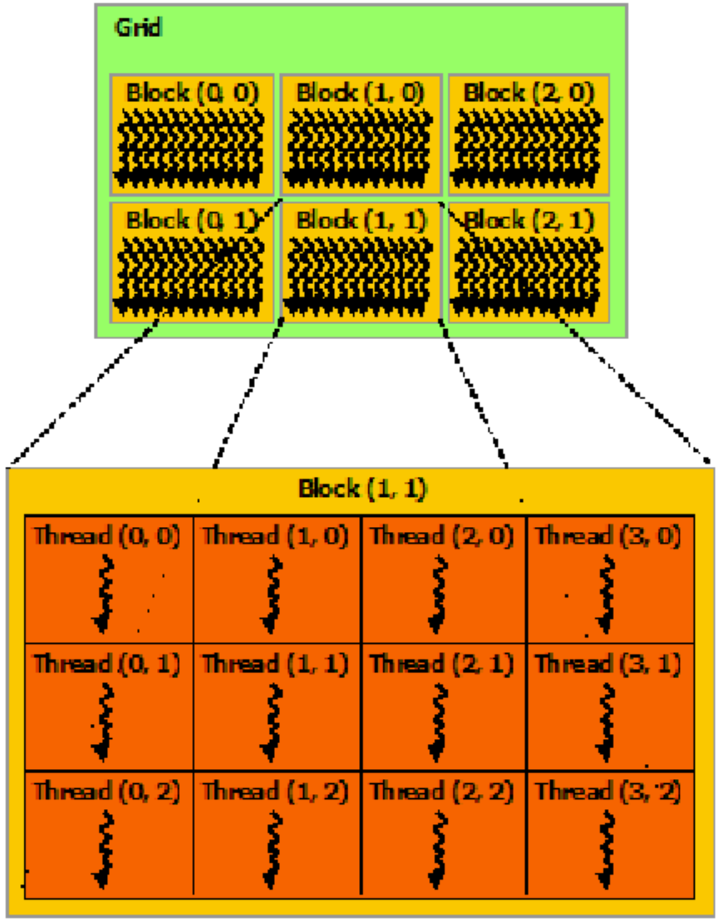
- Following variables allow to compute the ID of each individual thread that is executing in a grid block.

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

CUDA basics summary

__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);
```

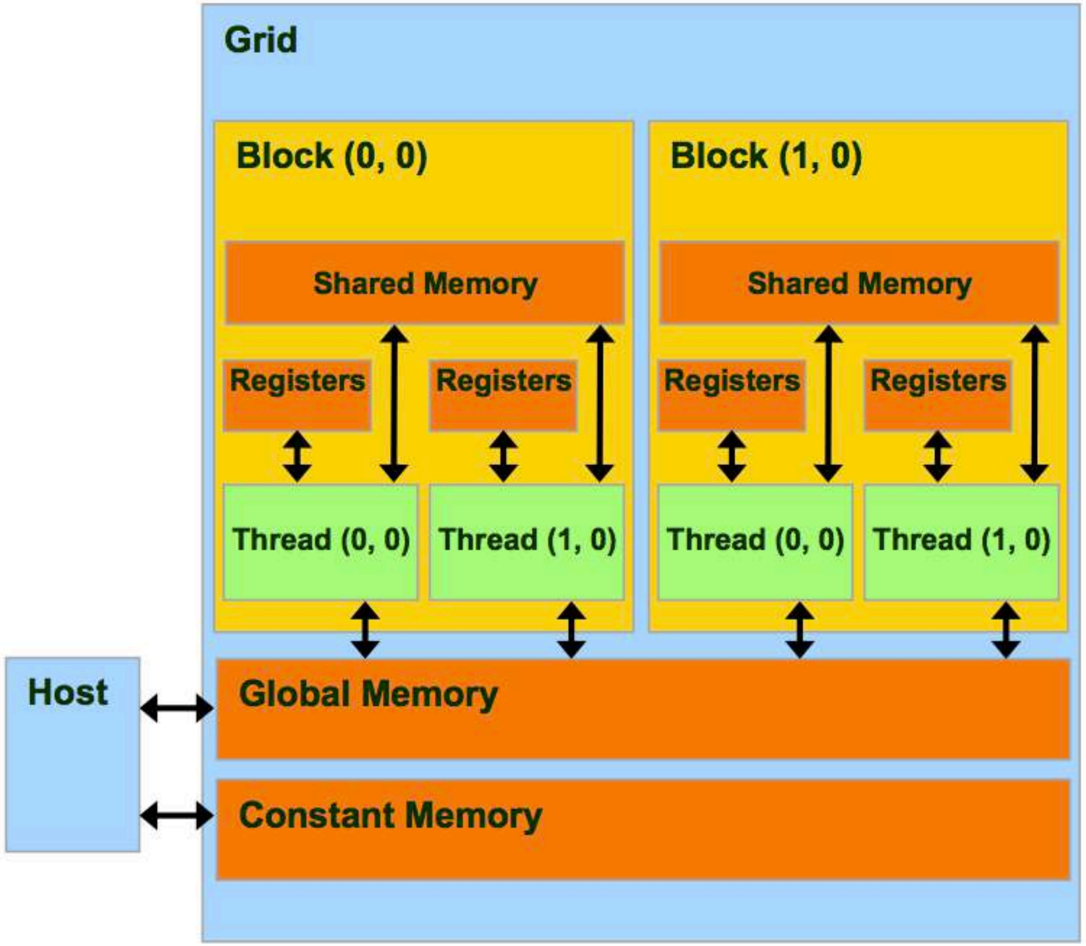
CUDA basics: Memory overview

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

General CUDA programming strategy

Use of constant memory for data that is constant during run time

- Data that is accessed by all threads within a block (half-warp, actually) at the same time
  - this reduces memory bandwidth requirements
- Do not use constant memory if threads access different elements of the data
  - half-warps can place only a single read-request at a time
  - if threads need different data from constant memory, these reads get serialized

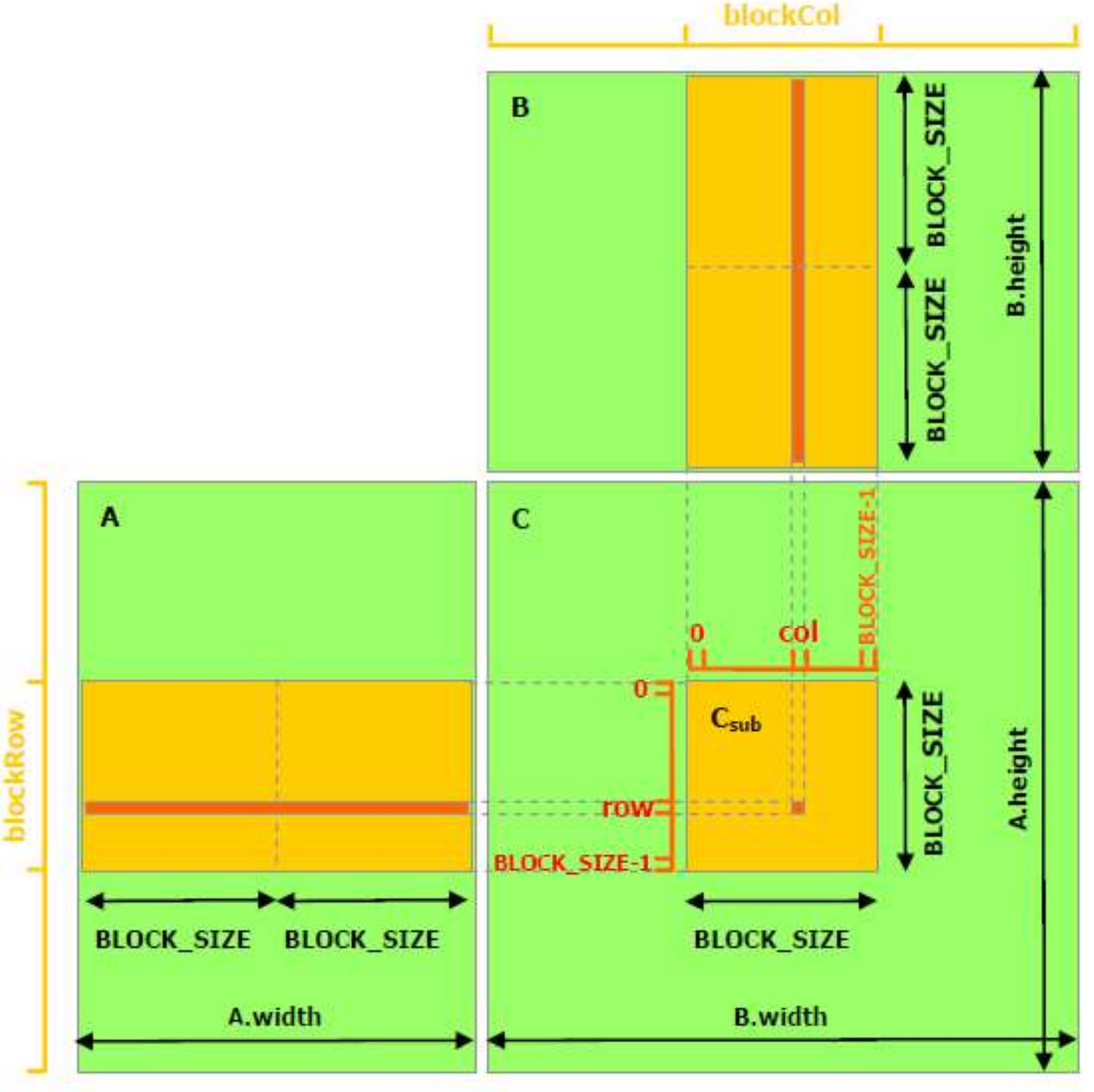
CUDA Example: Matrix-matrix multiply

```
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

Avoiding race conditions with atomic operations

Atomic operations

- An atomic operation cannot be divided into several operations
- What happens when we increment a counter?
  ```
  i++;
  ```
- This consists of three steps
  - 1) Read the value stored at the address of **`i`**
  - 2) Add 1 to the value read in step 1)
  - 3) Write the result back to the address of **`i`**
- What happens if multiple threads increment the counter?

Avoiding race conditions with atomic operations

Atomic operations

- An atomic operation cannot be divided into several operations
- What happens when we increment a counter?
  `i++;`
- This consists of three steps
  - 1) Read the value stored at the address of **`i`**
  - 2) Add 1 to the value read in step 1)
  - 3) Write the result back to the address of **`i`**
- What happens if multiple threads increment the counter?

- If multiple threads modify an address, the result is unpredictable
- Atomic operations are performed without interference from other threads
- Atomic operations are available on newer GPUs (efficient since Kepler chips) and can operate on global or shared memory
  - **`atomicAdd(), atomicSub(), atomicMin()`**, …
  - etc. see programming guide for full list
- Use wisely – code execution will be serialized

Example: Histogram

- Assume data set of 8-bit (1 byte) values
- Compute occurrence of each of the 256 possible values
- Serial CPU code:

```
#define SIZE (100*1024*1024)
int main(void){
    unsigned char buffer[SIZE];
    unsigned int histo[256];

    get_data(buffer,SIZE);          //read data

    for (int i=0; i<256; i++)       //initialize to zero
        histo[i] = 0;

    for (int i=0; i<SIZE; i++)      //compute histogram
        histo[buffer[i]]++;

    return 0;
}
```

Histogram CUDA code

- Using atomic add operation to avoid data races

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                    int size, unsigned int *histo){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x *gridDim.x;

    while (tid < size) {
        atomicAdd( &(histo[buffer[tid]]), 1 );
        tid += stride;
    }
}
```

- This will work – but with terrible performance – why?

Histogram CUDA code

- Using atomic add operation to avoid data races

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                    int size, unsigned int *histo){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x *gridDim.x;

    while (tid < size) {
        atomicAdd( &(histo[buffer[tid]]), 1 );
        tid += stride;
    }
}
```

- This will work – but with terrible performance – why?

Bad performance because

- the kernel does very little work
- thousands of threads access few (256) memory locations with atomic add operations

Improve performance by using shared memory

- write operations to shared memory are fast
- fewer threads will try to access the same memory locations with the atomic add operation

Histogram CUDA code

Histogram kernel with shared memory

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                  int size, unsigned int *histo){

    // shared memory,
    // assume 256 threads per block
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x *gridDim.x;
```

```
    // compute histogram in shared memory
    // for this block
    while (tid < size) {
        atomicAdd( &(temp[buffer[tid]]), 1 );
        tid += stride;
    }
    __syncthreads();

    // now merge each block's histogram
    // into global memory
    // again assuming 256 threads per block
    atomicAdd( &(histo[threadIdx.x]),
                         temp[threadIdx.x] );
}
```

Histogram CUDA code

Histogram host code

```
int main(void) {

    // host and device memory / pointers
    unsigned char h_buffer[SIZE], *d_buffer;
    unsigned int h_histo[256], *d_histo;

    // get our input data
    get_data(buffer, SIZE);

    // allocate device memory
    cudaMalloc( (void**)&d_buffer, SIZE);
    cudaMalloc( (void**)&d_histo,
                     256 * sizeof(int) );

    // copy data to device
    cudaMemcpy( d_buffer, h_buffer, SIZE,
                    cudaMemcpyHostToDevice);

    // initialize histogram to zero
    cudaMemset(d_histo, 0, 256);

    // launch kernel
    histo_kernel<<<32,256>>>(d_buffer,
                             SIZE, d_histo);

    // copy results back
    cudaMemcpy( h_histo, d_histo,
      256*sizeof(int), cudaMemcpyDeviceToHost);

    // free memory
    cudaFree(d_buffer); cudaFree(d_histo);

    // print our histogram
    print_histogram(h_histo);

    return 0;
}
```

Coordinating host and device

Kernel launches are asynchronous

- Control returns to CPU immediately
- This is great since the CPU can do work while the GPU is busy

CPU needs to synchronize before using results

- **`cudaMemcpy()`**
  Blocks the CPU until the copy is complete;
  copy begins when all preceding CUDA calls have completed
- **`cudaMemcpyAsync()`**
  Asynchronous, does not block the CPU
- **`cudaDeviceSynchronize()`**
  Blocks CPU until all preceding CUDA calls completed

Error management

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get error code for last error and handle error

```
// do some stuff on the GPU, now check status
cudaError_t error = cudaGetLastError();

// and handle error
if (error != cudaSuccess){
        // print CUDA error message and exit
        printf(“CUDA error: %s\n”),
               cudaGetErrorString(error));
        exit(MYERRCODE);
}
```

Query device properties

Application can query and select GPUs

- How many GPUs do we have
  `cudaGetDeviceCount(int *count)`
- Choose device for execution
  `cudaSetDevice(int device)`
- Which device am I currently running on?
  `cudaGetDevice(int *device)`
- Get hardware information
  ```
  cudaGetDeviceProperties(
       cudaDeviceProp *prop,
                  int device)
  ```
  (see also deviceQuery from CUDA Toolkit samples)

Examples of CUDA device properties

- `char name[256]`
- `int major, int minor`
- `size_t totalGlobalMem, size_t totalConstMem`
- `int maxThreadsDim[3], int maxGridSize[3]`
- `int multiProcessorCount`

E.g. launch number of blocks depending on HW

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);
int blocks = 4 * prop.multiProcessorCount;
int threads = 8 * prop.warpSize;
my_kernel<<<blocks,threads>>>();
```

Measuring performance

CUDA events can record GPU time stamps

```
// event timers
cudaEvent_t start, stop;
float elapsedTime;

// create start and stop events
cudaEventCreate(&start); cudaEventCreate(&stop);
// get start time stamp
cudaEventRecord(start, 0);

// do some work on the GPU (call a kernel)

// get stop time stamp and synchronize
cudaEventRecord(stop, 0); cudaEventSynchronize(stop);
// get elapsed time, in milliseconds
cudaEventElapsedTime(&elapsedTime, start, stop);

// destroy events
cudaEventDestroy(start); cudaEventDestroy(stop);
```

Page locked host memory

Allocate page-locked (pinned) host memory

- will never be paged out to disk
- GPU can use DMA to copy data (bypass CPU)
- DMA copies are usually faster than regular memory copies (depends on PCIE, FSB speed)
- don’t overuse – RAM needs to be available

```
int *h_a;

cudaHostAlloc( (void**)&a, size, cudaHostAllocDefault);

cudaFreeHost( a );
```

CUDA streams

- Up to now we have exploited data parallelism
- CUDA streams are about task parallelism,
  - that is executing different operations simultaneously
- CUDA streams represent a queue of GPU operations that get executed in a specific order
  - kernel launches
  - memory copies
  - event starts / stops

CUDA streams

Create a stream and add work to its queue

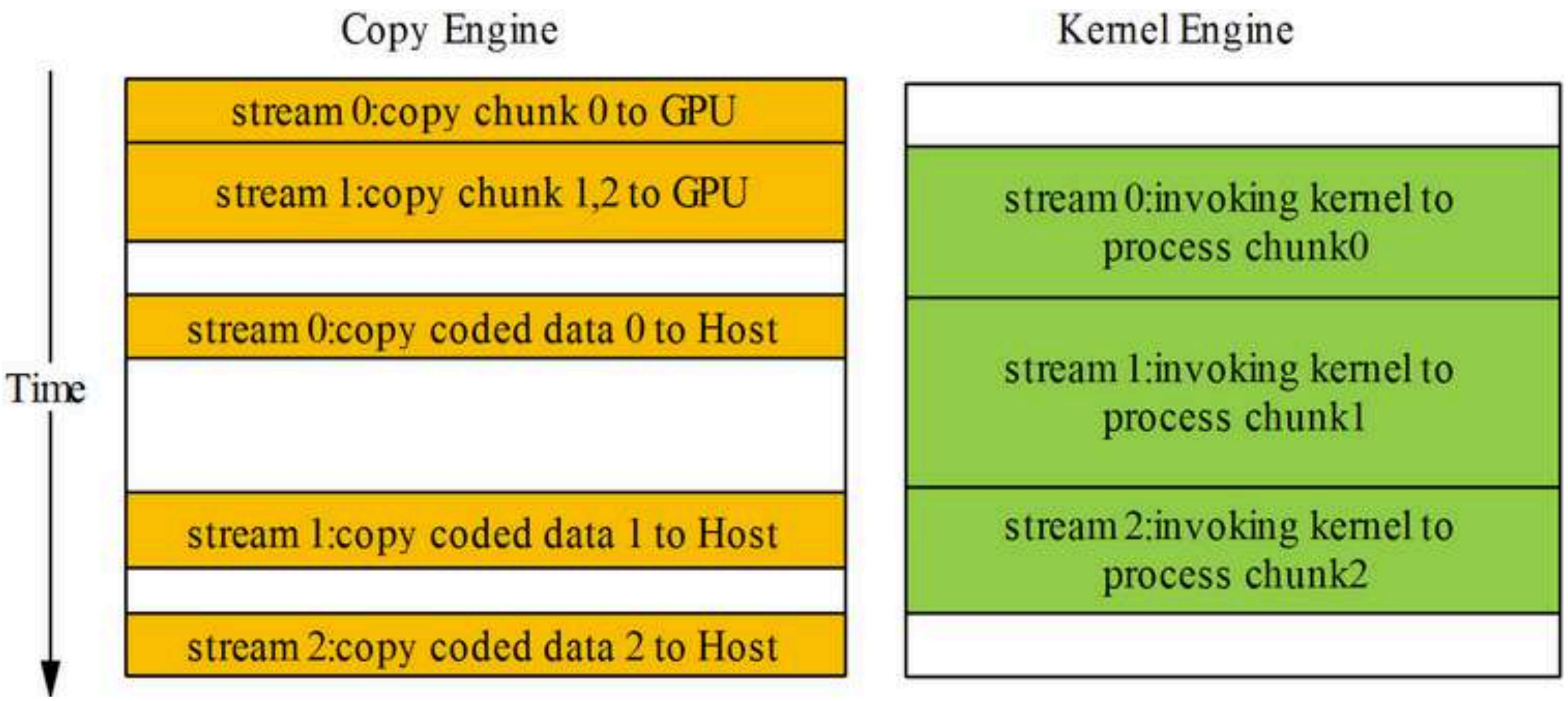
```
int *d_a, *h_a;

// initialize the stream
cudaStream_t stream;
cudaStreamCreate( &stream );

// allocate memory
cudaHostAlloc((void**)&h_a, SIZE,
              cudaHostAllocDefault);
cudaMalloc((void**)&d_a, SIZE);

// asynchronous memory copy
// (needs pinned memory)
cudaMemcpyAsync(d_a, h_a, SIZE,
      cudaMemcpyHostToDevice, stream);
```

```
// launch a kernel for this stream
my_kernel<<<grid,block,stream>>>(d_a,SIZE);

// asynchronous memory copy
// (needs pinned memory)
cudaMemcpyAsync(h_a, d_a, SIZE,
            cudaMemcpyDeviceToHost, stream);

// host does other stuff
do_some_work();

// synchronize with host, destroy stream
cudaStreamSynchronize(stream);
cudaStreamDestroy(stream);

// now it's safe to do some stuff with h_a
do_more_work(h_a);
```

CUDA streams

Using multiple CUDA streams

- for example divide problem into small chunks
- overlap memory copies in one stream with computation in a second stream



Copy Engine
Kernel Engine
Time
stream 0:copy chunk 0 to GPU
stream 1:copy chunk 1,2 to GPU
stream 0:copy coded data 0 to Host
stream 1:copy coded data 1 to Host
stream 2:copy coded data 2 to Host
stream 0:invoking kernel to process chunk0
stream 1:invoking kernel to process chunk1
stream 2:invoking kernel to process chunk2

CUDA streams

Using two streams

```
int *d_a0, *d_a1, *h_a;

// initialize the stream
cudaStream_t stream0;
cudaStream_t stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

// allocate memory
cudaHostAlloc((void**)&h_a, SIZE,
              cudaHostAllocDefault);
cudaMalloc((void**)&d_a0, SIZE/2);
cudaMalloc((void**)&d_a1, SIZE/2);

// asynchronous memory copy
//  (needs pinned memory)
cudaMemcpyAsync(d_a0, h_a, SIZE/2,
     cudaMemcpyHostToDevice, stream0);
```

```
cudaMemcpyAsync(d_a1, h_a+SIZE/2, SIZE/2,
              cudaMemcpyHostToDevice, stream1);

// launch kernels for both streams
my_kernel<<<grid,block,stream0>>>(d_a0,SIZE/2);
my_kernel<<<grid,block,stream1>>>(d_a1,SIZE/2);

// asynchronous memory copy
//  (needs pinned memory)
cudaMemcpyAsync(h_a, d_a0, SIZE/2,
              cudaMemcpyDeviceToHost, stream0);
cudaMemcpyAsync(h_a+SIZE/2, d_a1, SIZE/2,
              cudaMemcpyDeviceToHost, stream1);

// do stuff on the host, synchronize streams,
// destroy streams etc
```

Exercises on SDSC Comet – CUDA

Code samples for this course

- In SI2019 Github repository
  https://github.com/sdsc/sdsc-summer-institute-2019

- Directory `hpc3_gpu_cuda/cuda-samples`
  - “Hello world”
  - Addition, vector addition
  - Squaring matrix elements
  - 1D stencil

If you have not done so, please clone the repository now

- Check the README files
- Compile and run CUDA examples
- Try the exercises (look for **FIXME** comments and try to replace with correct code)

Directive based GPU programming with OpenACC

Partially based on material by
Mark Harris (Nvidia)

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 hhttps://developer.nvidia.com/openacc-toolkit
- Community Edition is free
- PGI Accelerator Fortran / C / C++ compilers
- PGI 2018 supports
  - OpenACC 2.6 for Nvidia GPIs
  - 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

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

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

SAXPY in Fortran

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

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

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`.

Complete SAXPY code

Trivial first example

- Apply a loop directive
- Learn compiler commands

```
#include <stdlib.h>

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

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


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


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


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

  return 0;
}
```

Compile and run SAXPY OpenACC code

- C:

```
pgcc –acc –ta=nvidia –Minfo=accel –o saxpy_acc saxpy.c
```

- Fortran:

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

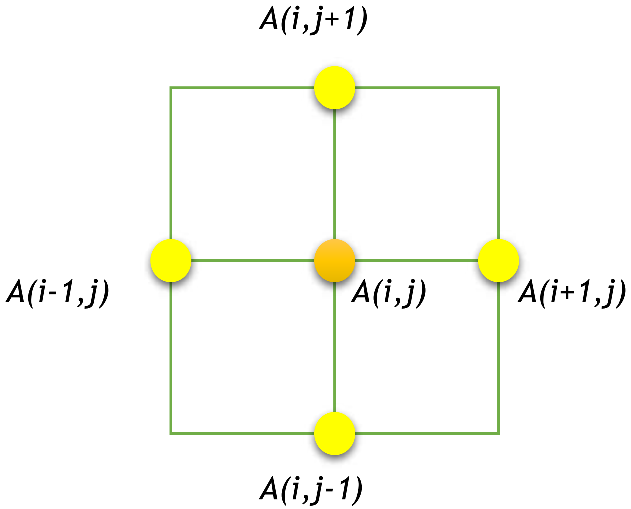
- Compiler output:

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

OpenACC 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 -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	69	--
CPU 2 OpenMP threads	36	1.92x
CPU 4 OpenMP threads	22	3.14x
CPU 6 OpenMP threads	17	4.06x
OpenACC GPU	501	0.03x FAIL

Compiler:
pgcc 17.5-0

CPU flags:
-fastsse -O3 -mp [-Minfo=mp]

GPU flags:
-acc [-Minfo=accel]

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

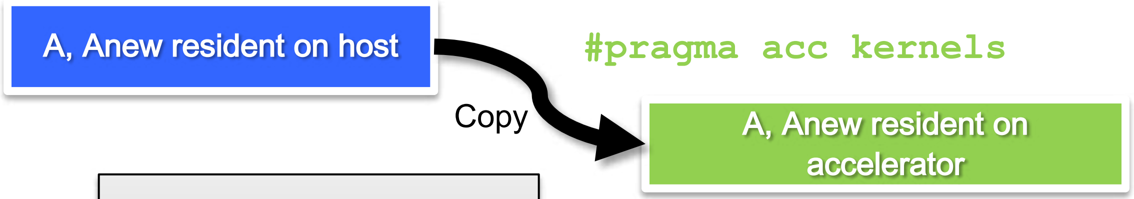
What went wrong?

- 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

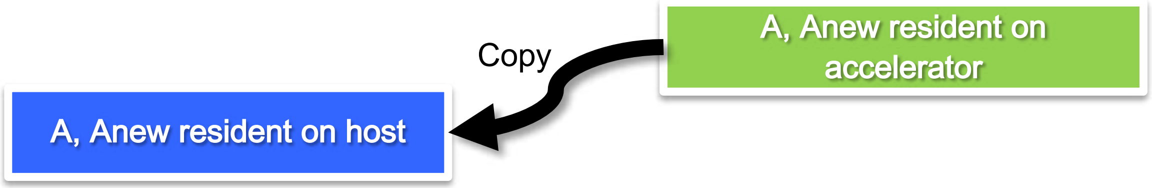
Copy

A, Anew resident on accelerator

```
                                    #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]);
                                        }
                                      }
```

These copies happen every iteration of the outer while loop!


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

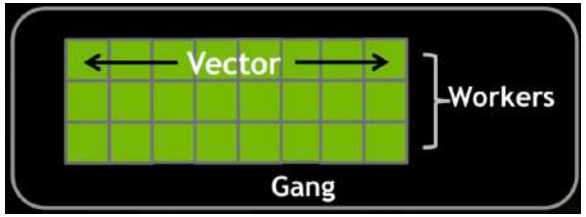
SDSC Comet CPU: Intel Xeon E5-2680 v3 GPU: NVIDIA Tesla K80 (using single GPU)

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 <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
  `-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

Exercises on SDSC Comet – OpenACC

Code samples for this course

- In SI2019 Github repository
  https://github.com/sdsc/sdsc-summer-institute-2019

- Directory **`hpc3_gpu_cuda/openacc-samples`**
  - saxpy
  - laplace-2d

If you have not done so, please clone the repository now

- Check the README files
- Compile and run OpenACC examples
- Check timings of Jacobi iterations on CPU in serial, wit OpenMP, and on GPU with OpenACC

Complete OpenACC API

Kernels Construct

Fortran

```
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

Clauses

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

Also any data clause

C

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

Kernels Construct

Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
    do i=1,n
       a(i) = 0.0
       b(i) = 1.0
       c(i) = 2.0
    end do

    do i=1,n
       a(i) = b(i) + c(i)
    end do
!$acc end kernels
```

kernel 1

kernel 2

Parallel Construct

Fortran

```
!$acc parallel [clause …]
    structured block
!$acc end parallel
```

C

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

Clauses

```
if( condition )
async( expression )
num_gangs( expression )
num_workers( expression )
vector_length( expression )
```

```
private( list )
firstprivate( list )
reduction( operator:list )
```

Also any data clause

Parallel Clauses

`num_gangs (` *`expression`* `)`	Controls how many parallel gangs are created (CUDA `gridDim`).
`num_workers ( expression )`	Controls how many workers are created in each gang (CUDA `blockDim`).
`vector_length ( list )`	Controls vector length of each worker (SIMD execution)
`private( list )`	A copy of each variable in list is allocated to each gang
`firstprivate ( list )`	`private` variables initialized from host
`reduction( operator:list )`	`private` variables combined across gangs

Loop Construct

Fortran

```
!$acc loop [clause …]
    loop
!$acc end loop
```

C

```
#pragma acc loop [clause …]
    { loop }
```

Combined directives

```
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
```

```
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
```

Detailed control of the parallel execution of the following loop.

Loop Clauses

`collapse( n )`

Applies directive to the following `n` nested loops.

`seq`

Executes the loop sequentially on the GPU.

`private( list )`

A copy of each variable in list is created for each iteration of the loop.

`reduction( operator:list )`

`private` variables combined across iterations.

Loop Clauses Inside parallel Region

`gang`	Shares iterations across the gangs of the parallel region.
`worker`	Shares iterations across the workers of the gang.
`vector`	Execute the iterations in SIMD mode.

Loop Clauses Inside kernels Region

`gang [( num_gangs )]`

Shares iterations across across at most `num_gangs` gangs.

`worker [( num_workers )]`

Shares iterations across at most `num_workers` of a single gang.

`vector [( vector_length )]`

Execute the iterations in SIMD mode with maximum `vector_length`.

`independent`

Specify that the loop iterations are independent.

OTHER SYNTAX

Other Directives

`cache` construct	Cache data in software managed data cache (CUDA shared memory).
`host_data` construct	Makes the address of device data available on the host.
`wait` directive	Waits for asynchronous GPU activity to complete.
`declare` directive	Specify that data is to allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.

Runtime Library Routines

Fortran

```
use openacc
#include "openacc_lib.h"

acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all
```

C

```
#include "openacc.h"


acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

Environment and Conditional Compilation

`ACC_DEVICE` *`device`*	Specifies which device type to connect to.
`ACC_DEVICE_NUM num`	Specifies which device number to connect to.
`_OPENACC`	Preprocessor directive for conditional compilation. Set to OpenACC version

Questions?