

# *Introduction to GPU Computing and Programming*

**Andreas W. Götz**  
**San Diego Supercomputer Center**

**SDSC Summer Institute 2016**  
**3 August 2016**

# *Course Overview*

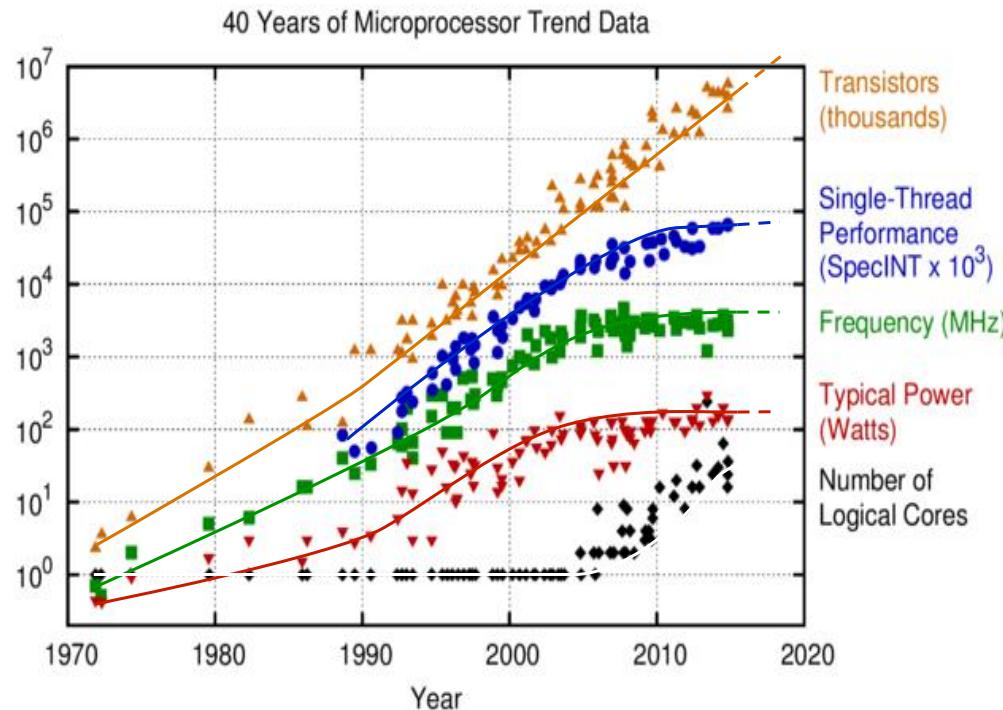
- GPU history / hardware overview
- GPU accelerated software examples
- CUDA C programming basics
- GPU enabled libraries
- OpenAcc introduction
- Exercises on SDSC Comet

# *What is a GPU?*

partly based on material by  
Ross Walker - SDSC

# CPU design trends

- Cannot increase single processor performance by increasing number of transistors
  - Cannot further increase clock speed
  - Replicate compute cores



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 2010-2015 by K. Rupp

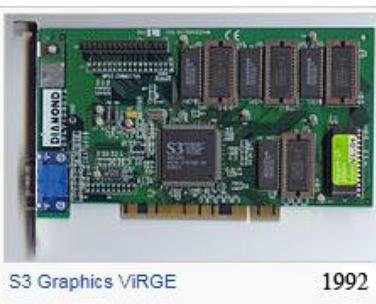
# *What is a GPU?*

- **Graphics Processing Unit**



Tseng Labs ET4000/W32p

1991



S3 Graphics ViRGE

1992



Voodoo3 2000 AGP card

1999



GeForce 6600 GT Personal Cinema

2004



NVIDIA GeForce GTX 280

2008

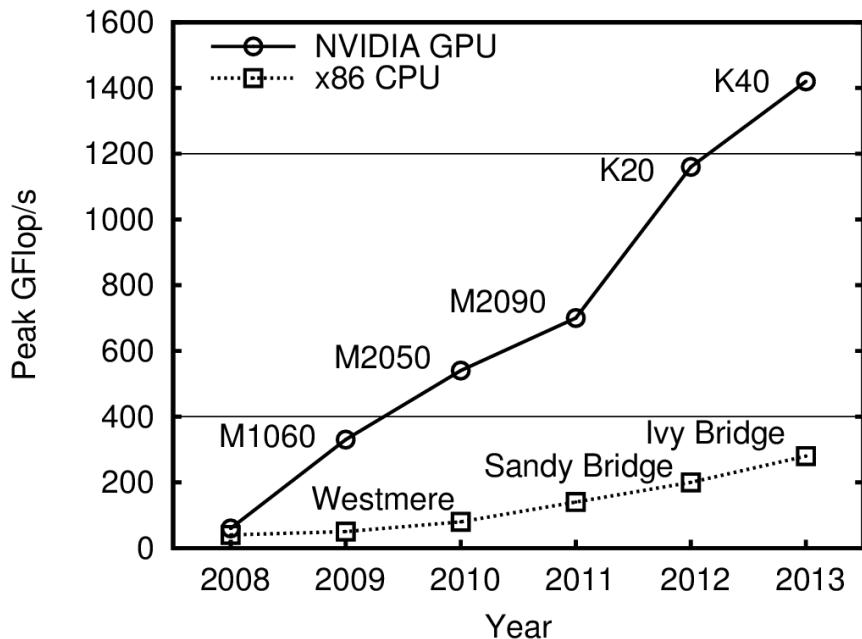
# *What is a GPU?*

- **Modern GPUs are programmable for general purpose computations**
- **GPUs have specialized processing cores**
  - Simplified design compared to CPU cores
    - Larger number on a chip
    - Reduced power consumption
- **Simplified core design**
  - Limited architectural features, e.g. branch caches
  - Partially exposed memory hierarchy

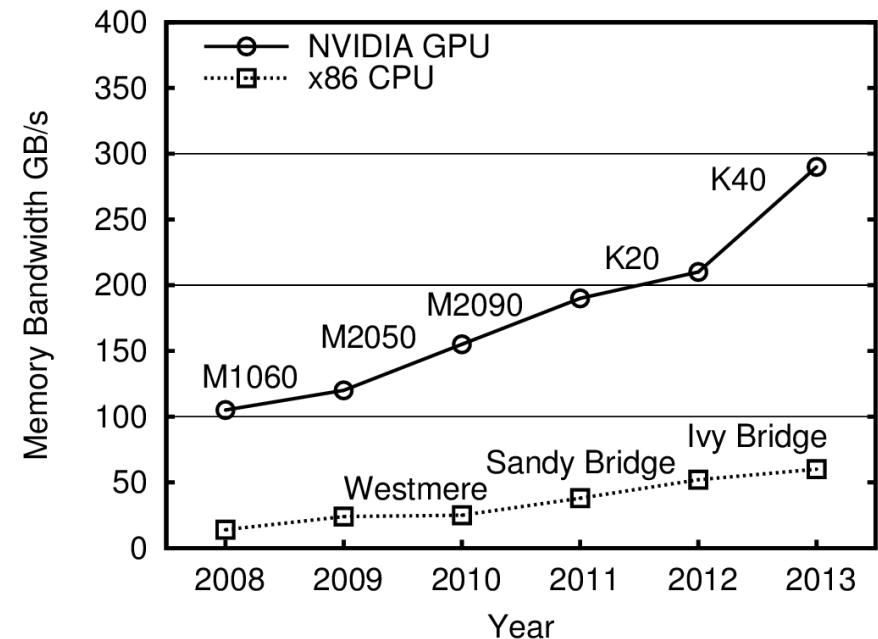
# *Why Interest in GPUs?*

## Performance comparison CPU vs GPU

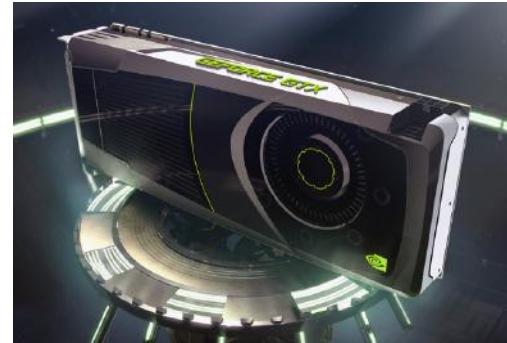
Double precision Flop/s



Memory Bandwidth



# *Why Interest in GPUs?*



## **Dual Socket Intel Haswell E5-2680v3 2.5GHz**

- 12 core
- \$1,900 – each

(approx prices, May 2015)

## **NVIDIA GTX980**

- GM204 GPU
- 4 GB RAM
- \$590 – each

## **NVIDIA Tesla K80**

- 2x GK210 GPU
- 2x 12 GB RAM
- \$4,000 – each

# *Why Interest in GPUs?*



## **ASCI White (LLNL)**

- 12.3 Tflop/s, #1 Top 500, Nov 2001.
- Cost - \$110 million (*in 2001!*)



## **SDSC Comet**

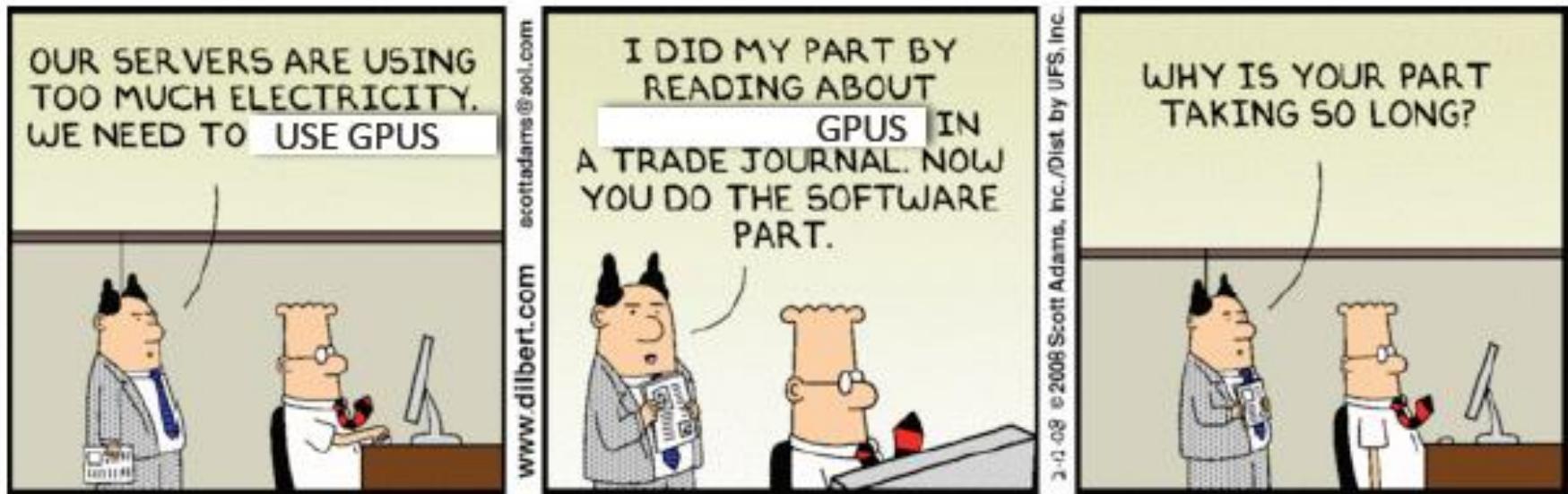
- ~2.0 Pflop/s aggregate
- 1 GPU node (total 36 GPU nodes)  
5.8 Tflop/s DP, 35.0 Tflop/s SP
- Cost - \$12 million



## **DIY 4x GTX980 box**

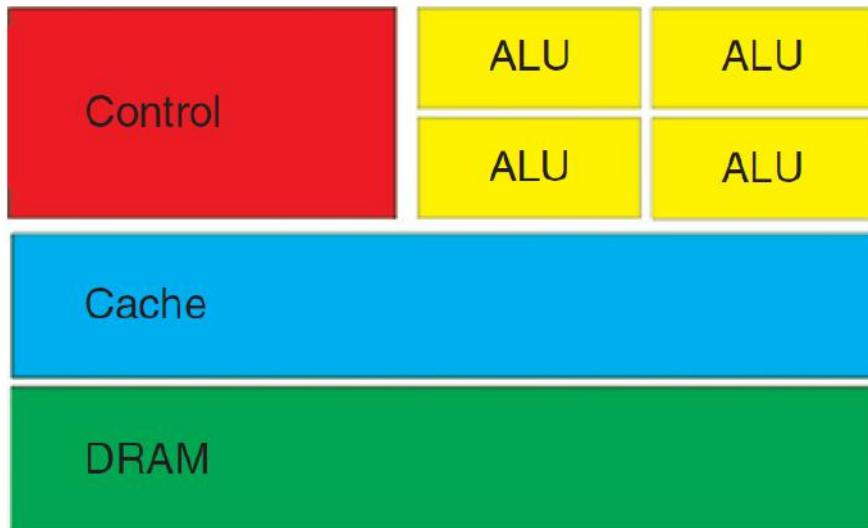
- 0.6 Tflop/s DP, 18.4 Tflop/s SP
- Cost - < \$5 thousand

# *What's the Catch?*

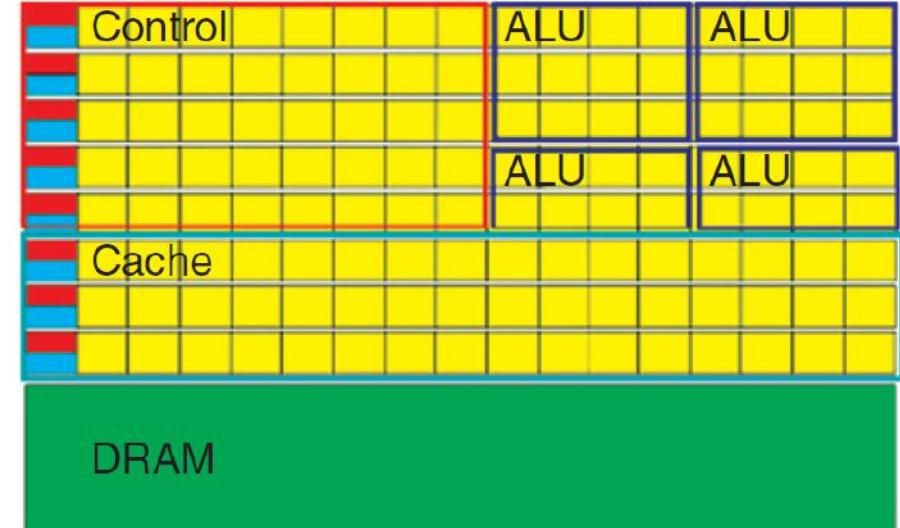


# GPUs Require Rewriting Code

(a) CPU



(b) GPU



- Few processing cores with sophisticated hardware
- Multi-level caching
- Prefetching
- Branch prediction
- Thousands of simplistic compute cores
- Operate in lock-step
- Vectorized loads/stores to memory

# *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
- **CUDA – based on ideas of Brook**
  - Solution to run C seamlessly on GPUs (NVIDIA only)
  - CUDA Toolkit contains compiler, math libraries, debugging and profiling tools
  - Lots of code samples, programming guides and other documentation available

# *GPU Programming 'Languages'*

- **CUDA Fortran**
  - Supports CUDA like extensions for Fortran
  - Supported by Portland Group Compilers
- **OpenACC**
  - OpenMP like compiler directives language for C and Fortran
  - Designed to make porting to GPUs easy and quick
  - Full support by Portland Group Compilers  
(OpenACC Toolkit free for academia)
  - Partial support by GNU compilers  
(experimental since version 5.1)

# ***Hardware Complexities***

- **Hardware characteristics change across GPU models and generations**
  - Single Precision / Double Precision floating point performance
  - Number of compute cores and multiprocessors
  - Number of threads that the hardware can execute
  - Number of Registers
  - Available GPU Memory, Device / Shared
- **Memory hierarchy needs to be explicitly managed**
  - CPU / GPU global, shared, texture, constant
- **Different vendors work in different ways**
  - NVIDIA vs AMD

# Hardware Complexities

## C870 – Nov 2006

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



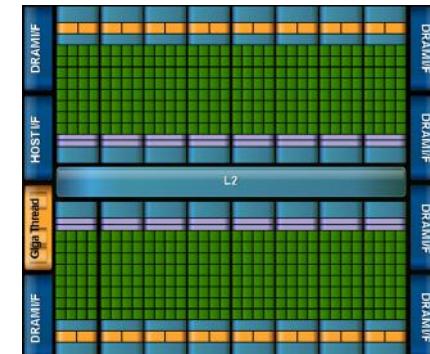
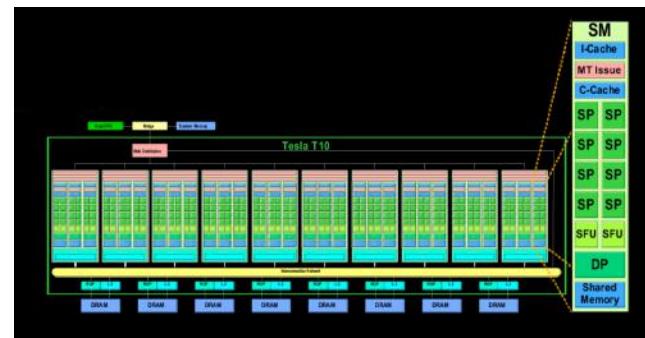
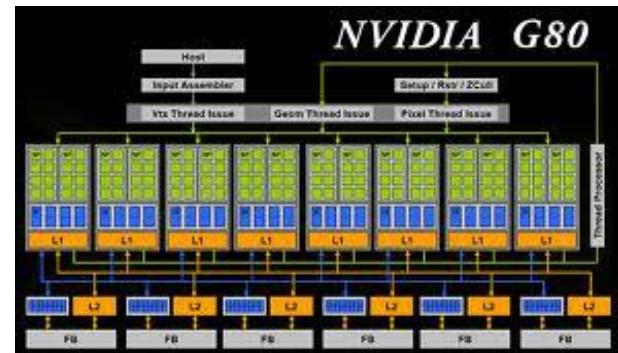
## C1060 – Jun 2008

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



## C2050 – Nov 2010

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



# Hardware Complexities

## K10 – Jun 2012

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



## K20 – Nov 2012

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



## K80 – Nov 2014

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

## K40 – Nov 2013

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

# *Hardware Complexities*

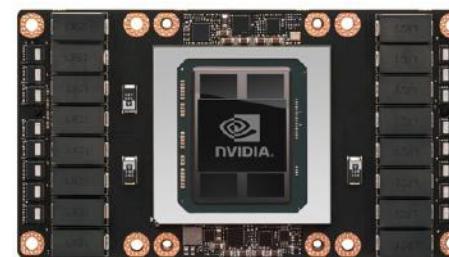
## **M40 – Nov 2015**

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



## **P100 – 2016 (ships Q4)**

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



	C2050	K10	K20	K40	K80	M40	P100
#Multi Proc	14	8 (x2)	13	15	13 (x2)	24	56
SP Cores per MP	32	192	192	192	192	128	64
#Cores	448	1,536 (x2)	2,496	2,880	2,496 (x2)	3,072	3,584
Warp Size	32	32	32	32	32	32	32
DP Gflop/s	515	95 (x2)	1,170	1,680	1,455 (x2)	213	5,304

# *Compute Capability*

Compute capability (version)	GPUs	Cards
1.0	G80, G92, G92b, G94, G94b	GeForce 8800GTX/Ultra, 9400GT, 9600GT, 9800GT, Tesla C/D/S870, FX4/5600, 360M, GT 420
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b	GeForce 8400GS/GT, 8600GT/GTS, 8800GT/GTS, 9600 GSO, 9800GTX/GX2, GTS 250, GT 120/30/40, FX 4/570, 3/580, 17/18/3700, 4700x2, 1xxM, 32/370M, 3/5/770M, 16/17/27/28/36/37/3800M, NVS420/50
1.2	GT218, GT216, GT215	GeForce 210, GT 220/40, FX380 LP, 1800M, 370/380M, NVS 2/3100M
1.3	GT200, GT200b	GeForce GTX 260, GTX 275, GTX 280, GTX 285, GTX 295, Tesla C/M1060, S1070, Quadro CX, FX 3/4/5800
2.0	GF100, GF110	GeForce (GF100) GTX 465, GTX 470, GTX 480, Tesla C2050, C2070, S/M2050/70, Quadro Plex 7000, Quadro 4000, 5000, 6000, GeForce (GF110) GTX 560 Ti 448, GTX570, GTX580, GTX590
2.1	GF104, GF114, GF116, GF108, GF106	GeForce 610M, GT 430, GT 440, GTS 450, GTX 460, GT 545, GTX 550 Ti, GTX 560, GTX 560 Ti, 500M, Quadro 600, 2000
3.0	GK104, GK106, GK107	GeForce GTX 690, GTX 680, GTX 670, GTX 660 Ti, GTX 660, GTX 650 Ti BOOST, GTX 650 Ti, GTX 650, GT 640, GeForce GTX 680MX, GeForce GTX 680M, GeForce GTX 675MX, GeForce GTX 675M, GeForce GTX 670MX, GTX 660M, GeForce GT 650M, GeForce GT 645M, GeForce GT 640M, Quadro K600, Quadro K2000, Quadro K4000, Quadro K5000
3.5	GK110	Tesla K20X, K20, GeForce GTX TITAN

- Double Precision Support Begins at V1.3
- Latest compute capability is V6.1  
(for Pascal architecture, May 2016)

See: <http://en.wikipedia.org/wiki/CUDA>

Feature support (unlisted features are supported for all compute capabilities)	Compute capability (version)																			
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2										
Integer atomic functions operating on 32-bit words in global memory	No	Yes																		
atomicExch() operating on 32-bit floating point values in global memory		Yes																		
Integer atomic functions operating on 32-bit words in shared memory	No	Yes																		
atomicExch() operating on 32-bit floating point values in shared memory		Yes																		
Integer atomic functions operating on 64-bit words in global memory	No	Yes																		
Warp vote functions		Yes																		
Double-precision floating-point operations	No	Yes																		
Atomic functions operating on 64-bit integer values in shared memory	No	Yes																		
Floating-point atomic addition operating on 32-bit words in global and shared memory		Yes																		
_ballot()	No	Yes																		
_threadfence_system()		Yes																		
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()	No	Yes																		
Surface functions		Yes																		
3D grid of thread block	No	Yes																		
Warp shuffle functions		Yes																		
Funnel shift	No	Yes																		
Dynamic parallelism		Yes																		
Feature support (unlisted features are supported for all compute capabilities)	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2										
Compute capability (version)																				
Technical specifications											Compute capability (version)									
Maximum dimensionality of grid of thread blocks						2					3									
Maximum x-, y-, or z-dimension of a grid of thread blocks						65535					2 <sup>31</sup> -1									
Maximum dimensionality of thread block											3									
Maximum x- or y-dimension of a block						512					1024									
Maximum z-dimension of a block											64									
Maximum number of threads per block						512					1024									
Warp size											32									
Maximum number of resident blocks per multiprocessor						8					16									
Maximum number of resident warps per multiprocessor	24					32	48					64								
Maximum number of resident threads per multiprocessor	768					1024	1536					2048								
Number of 32-bit registers per multiprocessor	8 K					16 K	32 K					64 K								
Maximum number of 32-bit registers per thread						128						63								
Maximum amount of shared memory per multiprocessor						16 KB						48 KB								
Number of shared memory banks						16						32								
Amount of local memory per thread						16 KB						512 KB								
Constant memory size												64 KB								
Cache working set per multiprocessor for constant memory												8 KB								
Cache working set per multiprocessor for texture memory												Device dependent, between 6 KB and 8 KB								
Maximum width for 1D texture reference bound to a CUDA array						8192						65536								
Maximum width for 1D texture reference bound to linear memory												2 <sup>27</sup>								
Maximum width and number of layers for a 1D layered texture reference						8192 x 512						16384 x 2048								
Maximum width and height for 2D texture reference bound to a CUDA array						65536 x 32768						65536 x 65535								
Maximum width and height for 2D texture reference bound to a linear memory						65000 x 65000						65000 x 65000								
Maximum width and height for 2D texture reference bound to a CUDA array supporting texture gather						N/A						16384 x 16384								
Maximum width, height, and number of layers for a 2D layered texture reference						8192 x 8192 x 512						16384 x 16384 x 2048								
Maximum width, height and depth for a 3D texture reference bound to linear memory or a CUDA array						2048 x 2048 x 2048						4096 x 4096 x 4096								

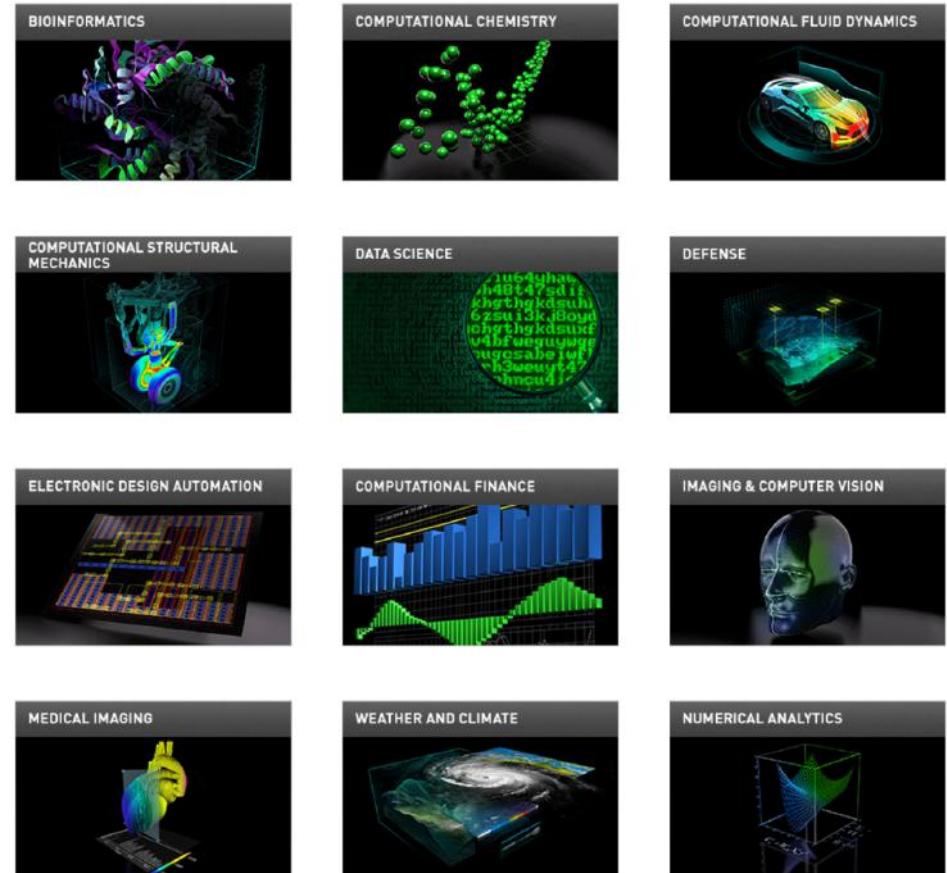
# ***What this all means***

- Never write your code with any assumption for how many threads it will use.
- Use CUDA calls to determine the specs of the GPU you are running on.
- Avoid using double precision where not specifically needed.

# ***GPU Accelerated Software Examples***

# *What GPU Codes Exist?*

- **Exhaustive list on:**  
<http://www.nvidia.com/object/gpu-applications.html>
- **Examples available from almost all fields**
  - Chemistry
  - Life Sciences
  - Astrophysics
  - Finance
  - Medical Imaging
  - Natural Language Processing
  - Social Sciences
  - etc



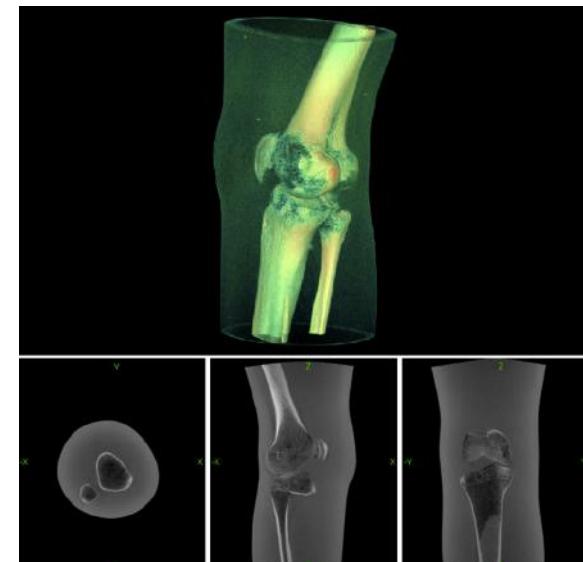
# *Medical Imaging*

- One of the earliest applications to take advantage of GPU acceleration
- E.g. real-time solutions for time-critical imaging



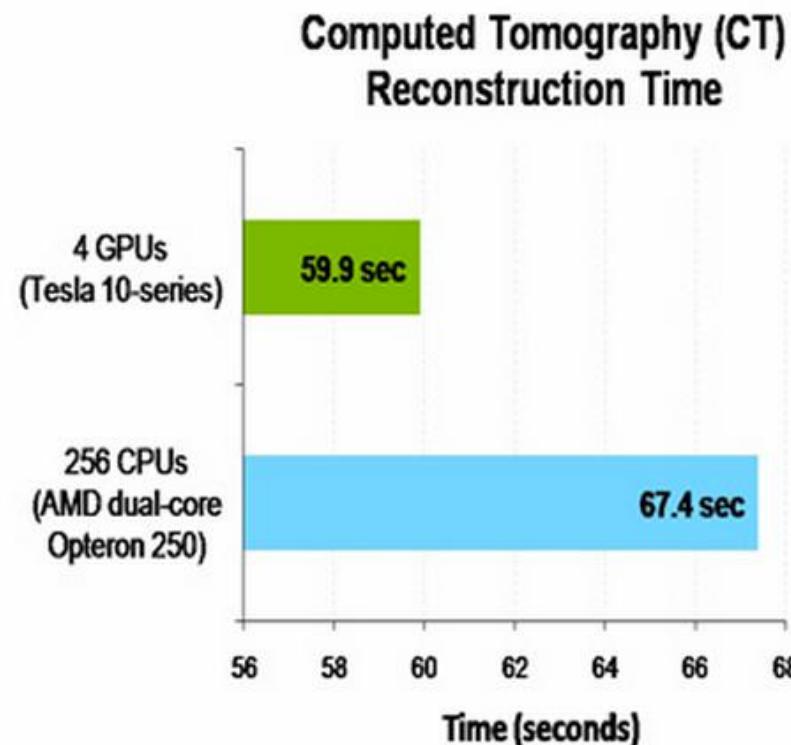
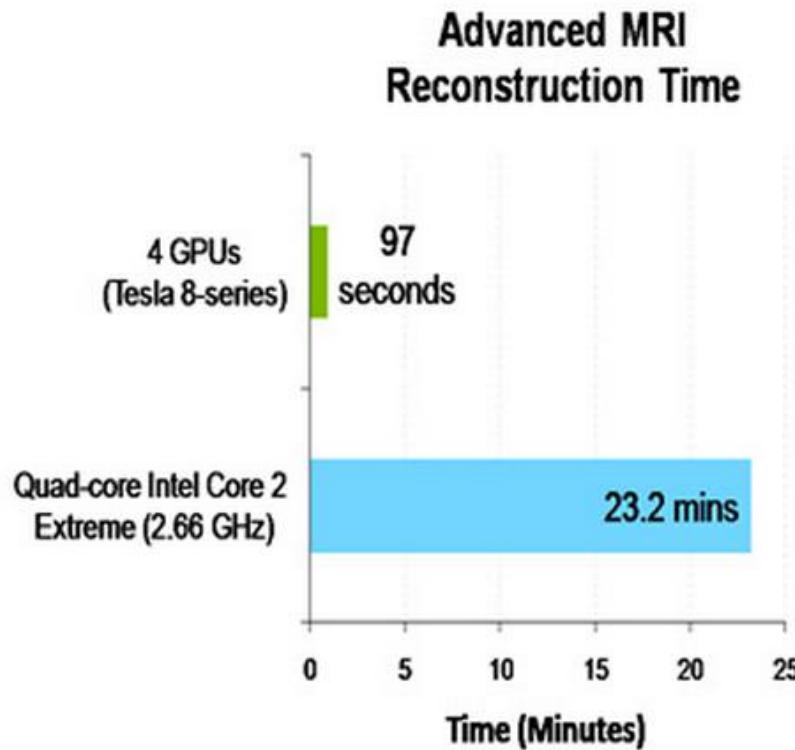
$$p(\mathbf{r}) = \int_{\Lambda} d\lambda \omega^2(\lambda, \mathbf{r}) g_F(u(\lambda, \mathbf{r}), v(\lambda, \mathbf{r})), \quad \mathbf{r}(x, y, z) \in R^3,$$

$$\omega[u, \quad v, \quad 1]^T = \quad \mathbf{M}(\lambda) \cdot [x, \quad y, \quad z, \quad 1]^T.$$

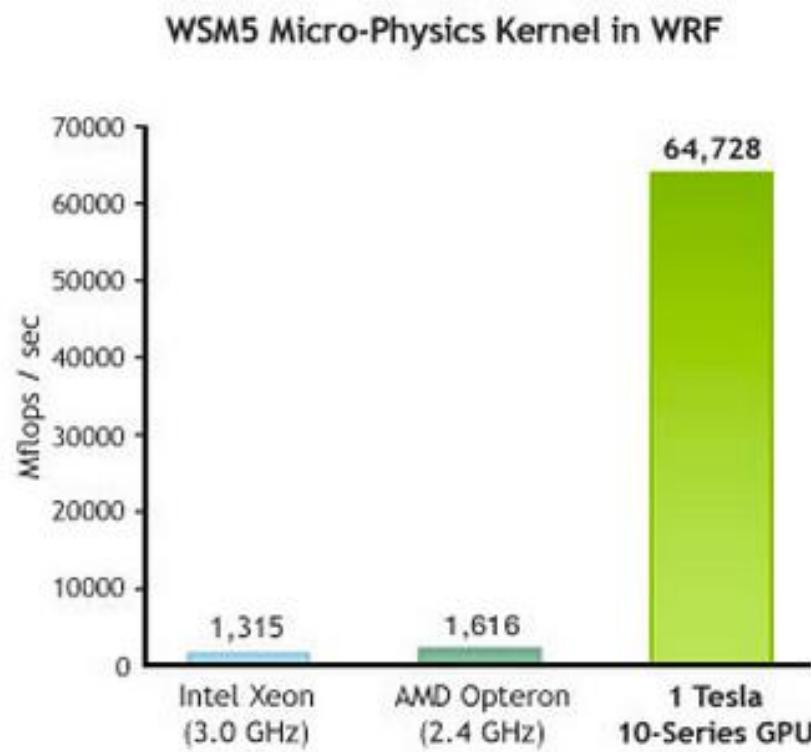


# *Medical Imaging*

- Advanced MRI, Computed Tomography (CT) etc
- Time to solution CPU vs GPU implementations

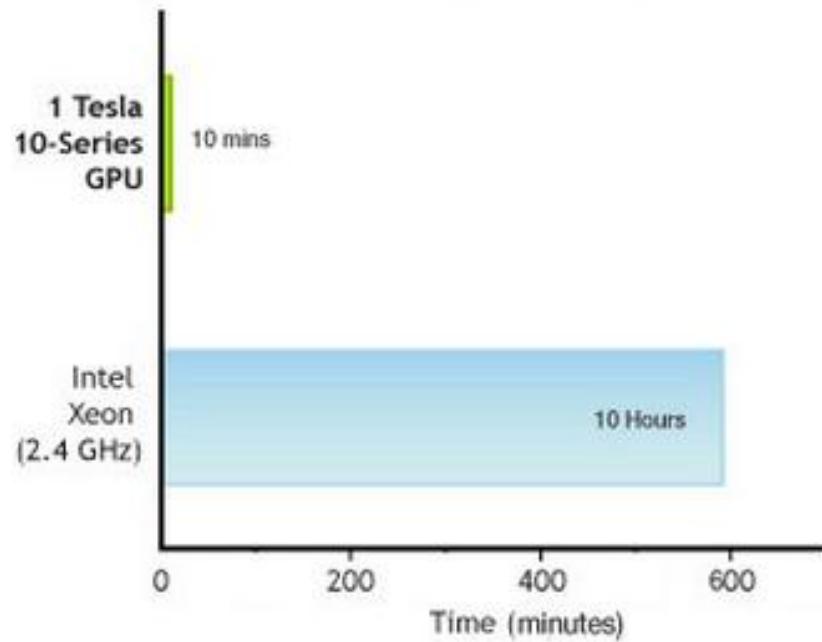


# *Weather, atmospheric, ocean modeling*



Weather forecasting

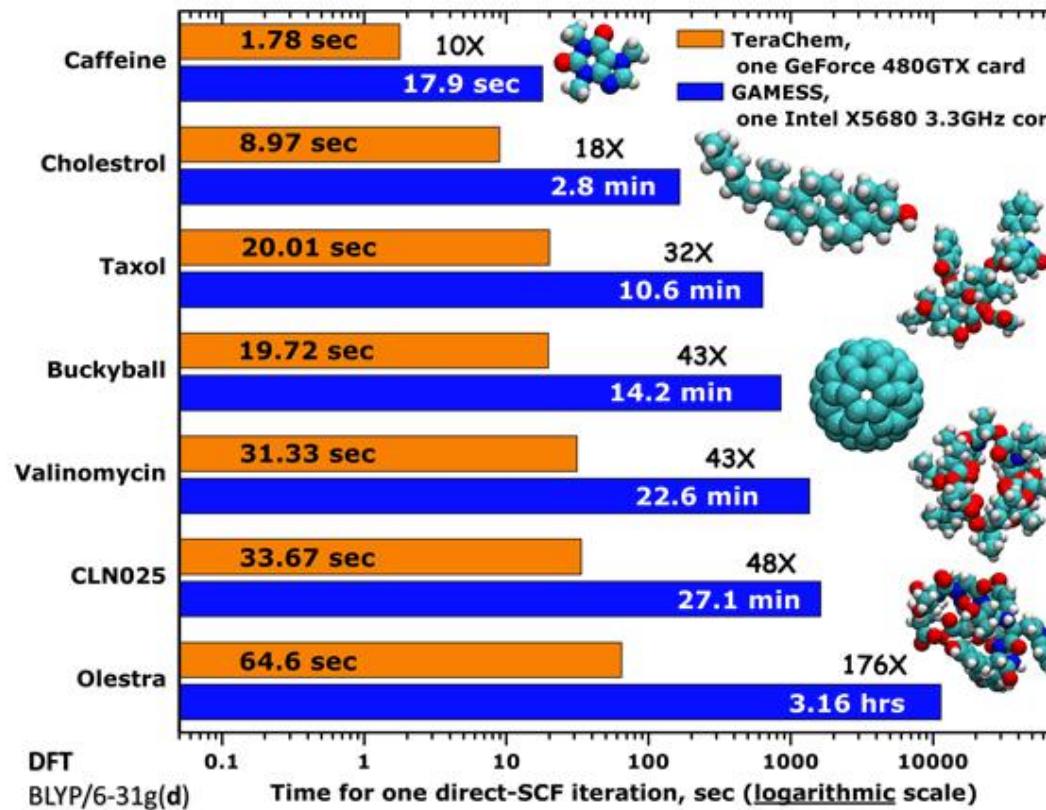
Tsunami Simulation  
3000km x 3000km (500m mesh)



Tsunami modeling

# *Quantum Chemistry*

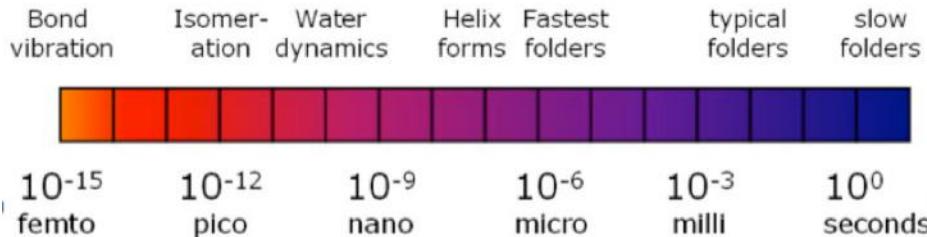
- Prediction of molecular properties from Quantum Mechanics



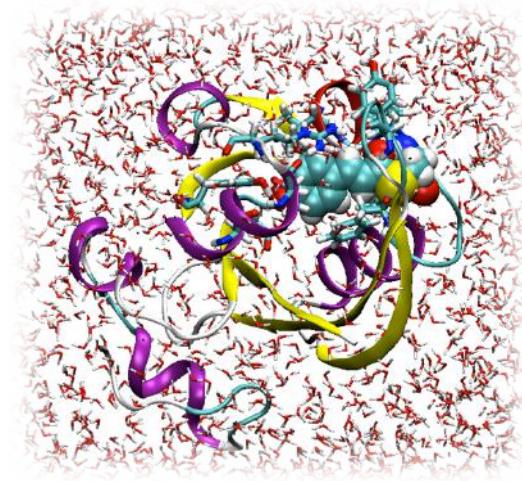
# Molecular Dynamics

- Drug design, biocatalysis
- Atomistic simulation of condensed phase systems
- Classical pairwise potentials
- Time evolution
- Simulation time step 2 fs

## Relevant timescales



$$V_{\text{AMBER}} = \sum_i^{n_{\text{bonds}}} b_i(r_i - r_{i,\text{eq}})^2 + \sum_i^{n_{\text{angles}}} a_i(\theta_i - \theta_{i,\text{eq}})^2 \\ + \sum_i^{n_{\text{dihedrals}}} \sum_n^{n_{i,\text{max}}} (V_{i,n}/2)[1 + \cos(n\phi_i - \gamma_{i,n})] \\ + \sum_{i < j}^{n_{\text{atoms}}} \left( \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} \right) + \sum_{i < j}^{n_{\text{atoms}}} \frac{q_i q_j}{4\pi \epsilon_0 r_{ij}}$$

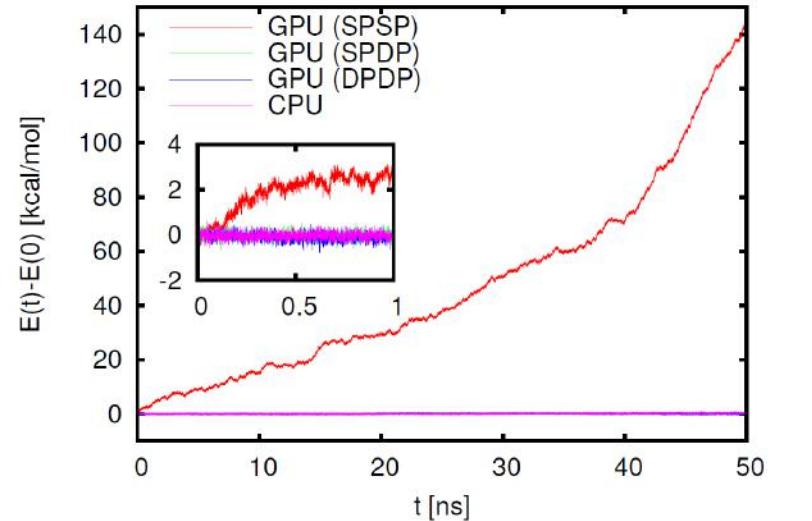


- 500 million steps to reach 1  $\mu$ s  
=> MD step must take < 5 ms to simulate 1  $\mu$ s in 1 week

# Molecular Dynamics

## 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*



Q24.40 for Forces, Q34.30 for Energies / Virials

## AMBER MD package

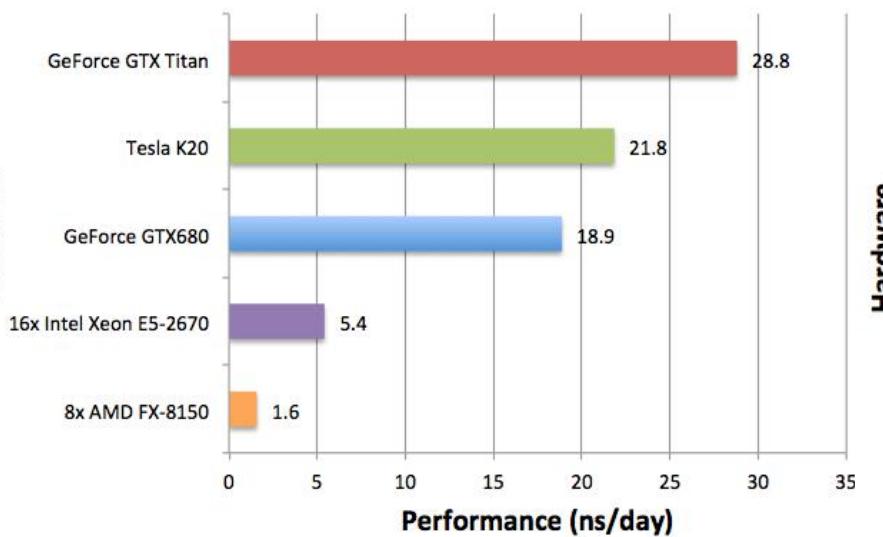
Götz, Williamson, Xu, Poole, Le Grand, Walker, *JCTC* **8** (2012) 1542.

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

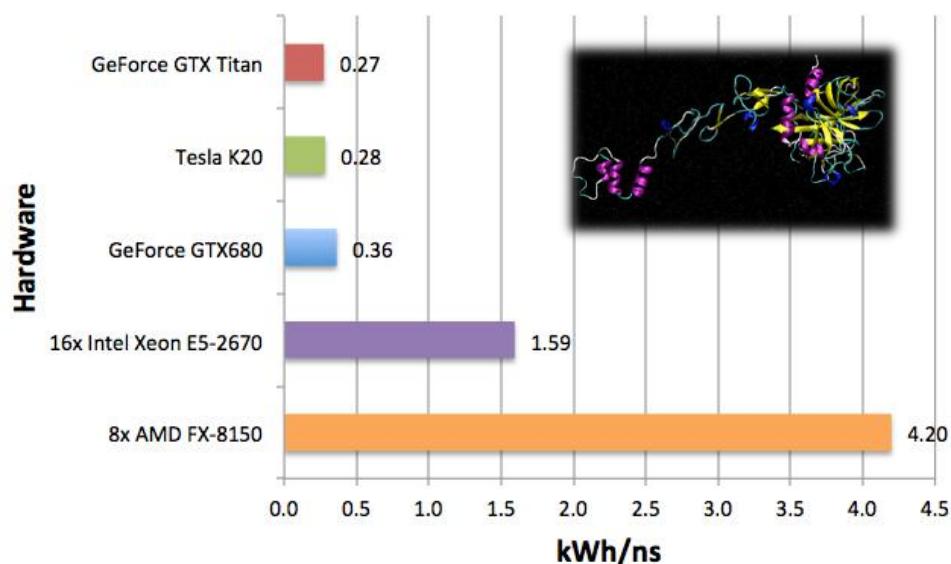
Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *JCTC* **9** (2013) 3878.

# Molecular Dynamics

Factor IX (90k atoms, NVE, 2 fs time step)



Factor IX (90k atoms, NVE, 2 fs time step)



- 5x speedup

- 5x power savings

vs single, state-of-the-art CPU node

# ***CUDA Parallel Computing Platform***

partly based on material by  
Will Ramey - NVIDIA corporation

# *CUDA Toolkit*

- <http://nvidia.com/getcuda>
- **Compiler (nvcc)**
- **Development tools**
  - Debuggers (CUDA-gdb, CUDA-memcheck)
  - Profilers (nvprof, nvvp)
  - Nsight IDE for Eclipse and Visual Studio
- **Libraries**
  - cuBLAS, cuFFT, cuRAND, cuSPARSE, cuSolver, NPP, Thrust, CUDA Math Library
- **CUDA code samples**

# *3 Ways to Use GPUs*

## Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility

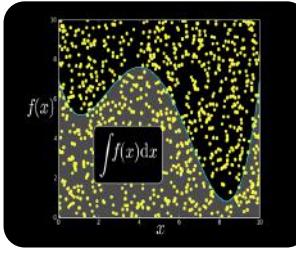
# *Libraries*

- **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 matmul)**

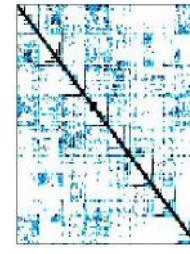
# Some GPU accelerated libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



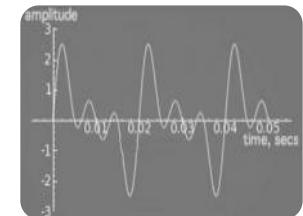
Vector Signal  
Image Processing



GPU Accelerated  
Linear Algebra



Matrix Algebra  
on GPU and  
Multicore



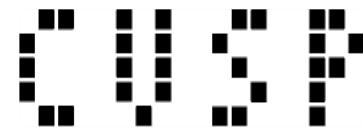
NVIDIA cuFFT



ROGUE WAVE  
SOFTWARE  
IMSL Library



ArrayFire Matrix  
Computations



Sparse Linear  
Algebra



C++ STL  
Features for  
CUDA



# ***3 Steps to Using Libraries***

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

saxpy ( ... )       $\xrightarrow{\hspace{1cm}}$       cublasSaxpy ( ... )

- **Step 2:** Manage data locality

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

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

nvcc myobj.o -l cublas

# **CUBLAS Library example**

```
int N = 1 << 20;  
  
// Perform SAXPY on 1M elements: y[ ]=a*x[ ]+y[ ]  
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

# **CUBLAS Library Example**

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

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

Add “cublas” prefix  
and use device  
variables

# CUBLAS Library Example

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



Initialize CUBLAS

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

```
cublasDestroy(handle);
```



Shut down CUBLAS

# CUBLAS Library Example

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

Allocate device  
vectors

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

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

Deallocate device  
vectors

# CUBLAS Library Example

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

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

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

# *Exercises on SDSC Comet*

# *Hands-on exercises*

- CUDA source code examples are available in the SI2016 github repository

<https://github.com/sdsc/sdsc-summer-institute-2016>

directory: hpc0\_gpu

- You are encouraged to compile/execute/modify/play with these examples
- We will use GPU nodes on SDSC Comet

# ***SDSC Comet Supercomputer***

- Contains a GPU Partition
- 36 GPU nodes
- 2 x K80 GPUs per node.
- Each K80 = 2 GPUs
  - 4 'physical' GPUs per node



```
export CUDA_VISIBLE_DEVICES=0 [for GPU 0]  
export CUDA_VISIBLE_DEVICES=2,3 [for GPU 2 & 3]
```

# Comet GPU nodes

```
andi@flash-4:~>ssh agoetz@comet.sdsc.edu
Last login: Wed Jul 27 14:24:58 2016 from gecko.sdsc.edu
Rocks 6.2 (SideWinder)
Profile built 16:45 08-Feb-2016
```

Kickstarted 17:27 08-Feb-2016

WELCOME TO



```
[agoetz@comet-ln3:~>qstat -q
Queue          Memory CPU Time Walltime Node Run Que Lm State
-----  -----
compute        --    -- 48:00:00   72  383 245 -- E R
debug          --    -- 00:30:00    4   0  0 -- E R
shared          --    -- 48:00:00    1  429 65 -- E R
gpu            --    -- 48:00:00    4   17 125 -- E R
gpu-shared      --    -- 48:00:00    1   25  6 -- E R
large-shared    --    -- 48:00:00    1    5  2 -- E R
monitor         --    --    --    --   0  0 -- E R
maint           --    --    --    --   0  0 -- E R
-----  -----
                                         859  443
```

login

Available queue

We will use  
the queue  
gpu-shared

# *Comet GPU nodes*

## Get interactive access to GPU node

```
[agoetz@comet-ln3:~>srun --partition=gpu-shared --nodes=1 --ntasks-per-node=6 \
[>                                --gres=gpu:1 -t 00:10:00 --pty --wait=0 --export=ALL /bin/bash
srun: job 3643874 queued and waiting for resources
srun: job 3643874 has been allocated resources
[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
[agoetz@comet-30-03 ~]$ module load pgi
[agoetz@comet-30-03 ~]$ pgcc --version
pgcc 15.7-0 64-bit target on x86-64 Linux -tp haswell
The Portland Group - PGI Compilers and Tools
Copyright (c) 2015, NVIDIA CORPORATION. All rights reserved.
```

**Load CUDA module**

**Check Nvidia  
CUDA C compiler**

**Load PGI module**

**Check PGI C compiler**

# Comet GPU nodes

```
agoetz@comet-31-13:~>nvidia-smi
```

```
Wed May 6 20:45:15 2015
```

Check available GPUs

NVIDIA-SMI 346.46      Driver Version: 346.46						
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 K80	Off	0000:05:00.0	Off	0%	Default
N/A	41C	P0	56W / 149W	55MiB / 11519MiB	0%	Default
1	Tesla K80	Off	0000:06:00.0	Off	0%	Default
N/A	34C	P0	71W / 149W	55MiB / 11519MiB	0%	Default
2	Tesla K80	Off	0000:84:00.0	Off	0%	Default
N/A	44C	P0	55W / 149W	55MiB / 11519MiB	0%	Default
3	Tesla K80	Off	0000:85:00.0	Off	98%	Default
N/A	36C	P0	74W / 149W	55MiB / 11519MiB		
Processes:						
GPU	PID	Type	Process name	GPU Memory Usage		
No running processes found						

# *Comet GPU nodes*

- Other jobs may already be running on the nodes  
(we are using shared nodes)

Processes:				GPU Memory Usage
GPU	PID	Type	Process name	
0	79513	C	pmemd.cuda	68MiB

- The node has been configured for the CUDA runtime to use only one of the GPUs
- Check environment variable `CUDA_VISIBLE_DEVICES` for the GPU that has been assigned to you

# CUDA Toolkit code samples

```
agoetz@comet-30-11:~/cuda-install-samples-7.0.sh ./  
Copying samples to ./NVIDIA_CUDA-7.0_Samples now...  
Finished copying samples.  
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples/  
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples>ls  
0_Simple  2_Graphics  4_Finance  6_Advanced  common  Makefile  
1_Utils  3_Imaging  5_Simulations  7_CUDALibraries  EULA.txt  
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples>make  
make[1]: Entering directory `/home/agoetz/NVIDIA_CUDA-7.0_Samples/0_Simple/simpleSurfaceWrite'  
"/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 simpleSurfaceWrite.o -c simpleSurfaceWrite.cu
```

Copy and compile CUDA code samples

- **Compilation takes a while**  
(executables will be in sub dir `bin/x86_64/linux/release/`)
- **Compile individual examples or compile faster with**  
`make -j 6`
- **Have a look at some of the code samples**

# CUDA Toolkit code samples

```
agoetz@comet-30-11:~/NVIDIA_CUDA-7.0_Samples/1_Utils.../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	7.0 / 7.0
CUDA Capability Major/Minor version number:	3.7
Total amount of global memory:	11520 MBytes (12079136768 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP:	2496 CUDA Cores
GPU Max Clock rate:	824 MHz (0.82 GHz)
Memory Clock rate:	2505 MHz
Memory Bus Width:	384-bit
L2 Cache Size:	1572864 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes

Run CUDA code samples

- **deviceQuery gives information on available GPUs**

# *CUDA Toolkit code samples*

- **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
```

# ***CUDA C Basics***

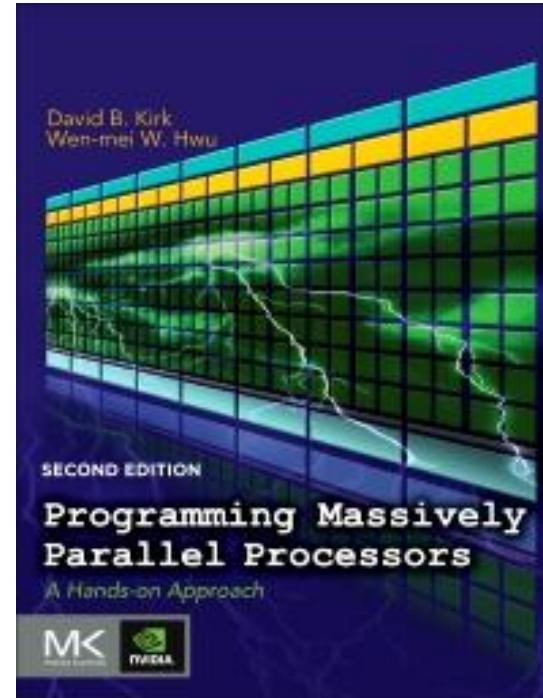
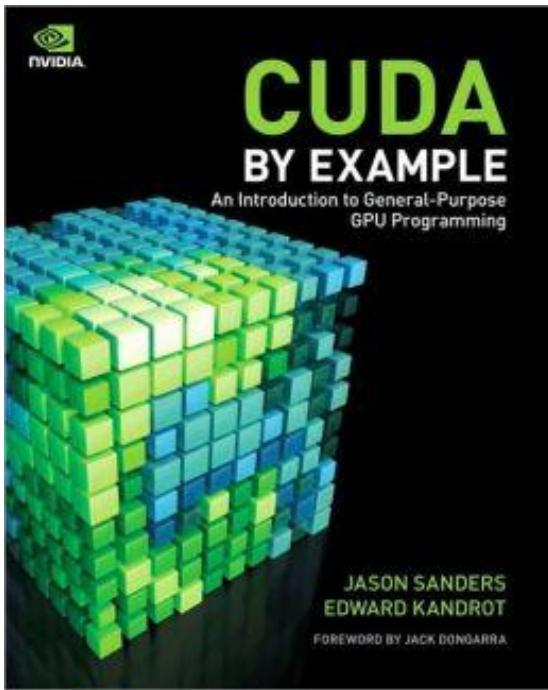
partly based on material by  
Mark Harris - NVIDIA corporation

# **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

# *Recommended Reading*



- CUDA Programming Guide

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

# *Code samples for this course*

- In SI2016 github repository

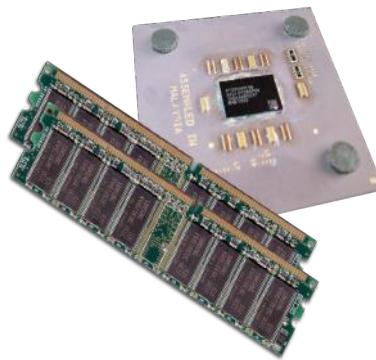
<https://github.com/sdsc/sdsc-summer-institute-2016>

directory: `hpc0_gpu/cuda-samples`

- “Hello world”
- Addition, vector addition
- Squaring matrix elements
- 1D stencil
- We will discuss these simple code samples

# *Heterogeneous Computing*

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

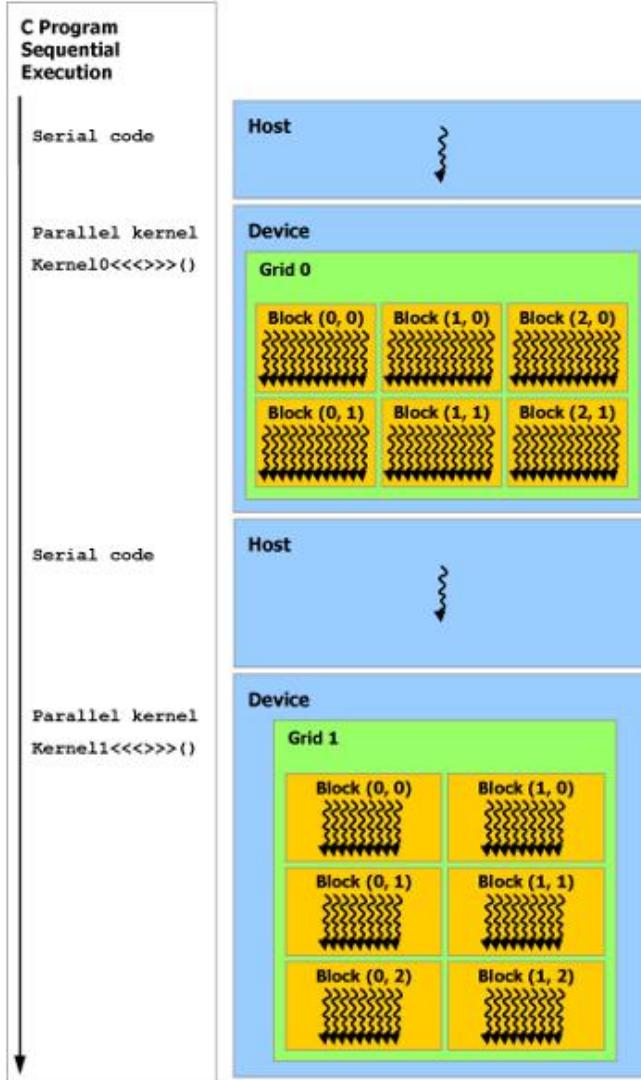


Host

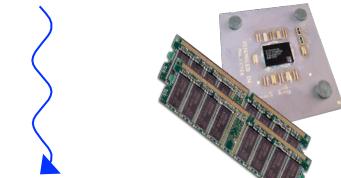


Device

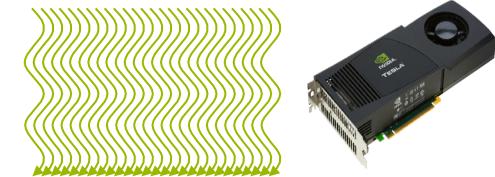
# Heterogeneous Computing



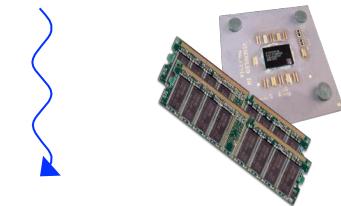
serial code



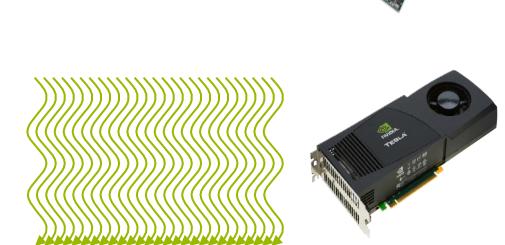
parallel code



serial code



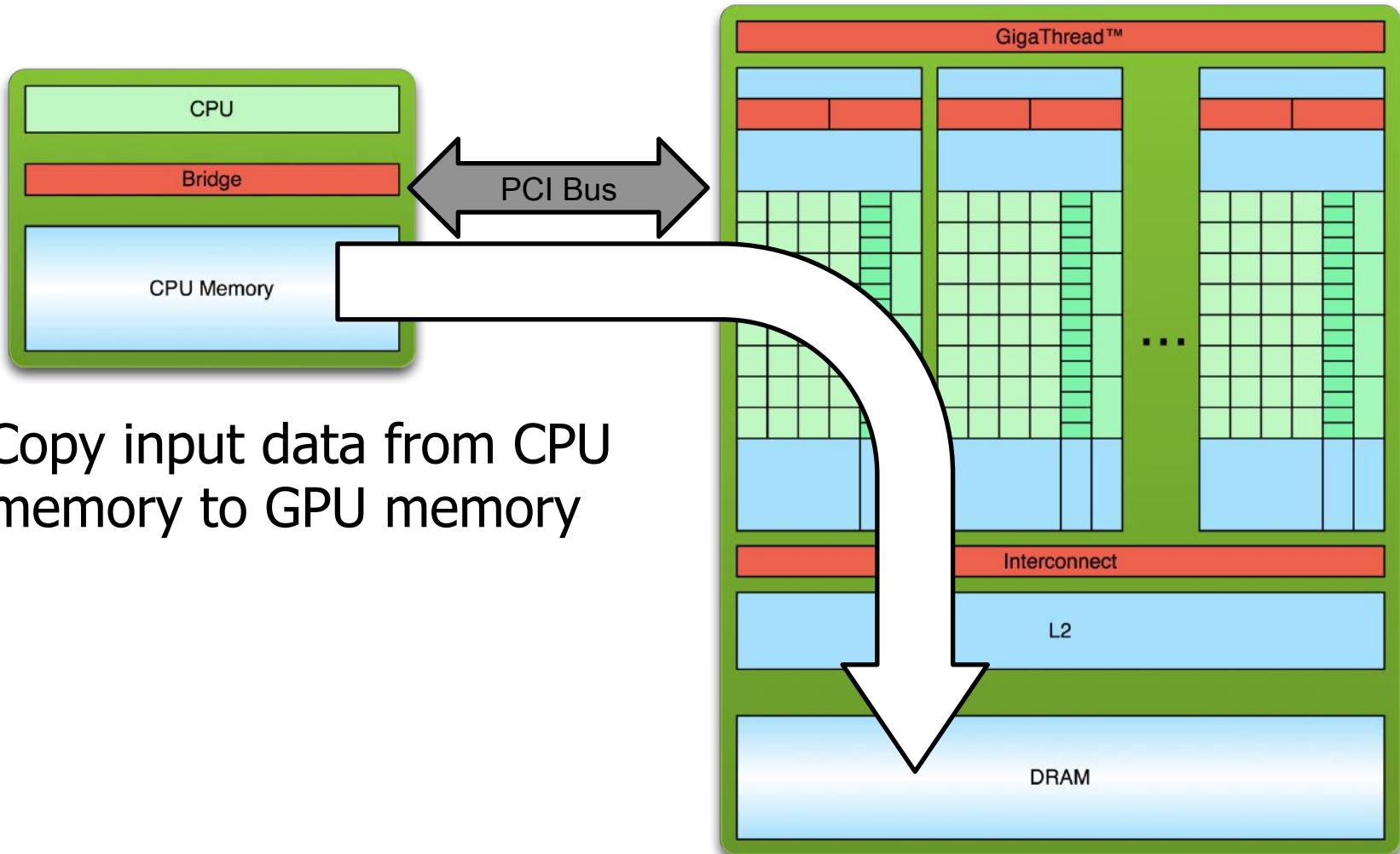
parallel code



# Processing Flow

Host

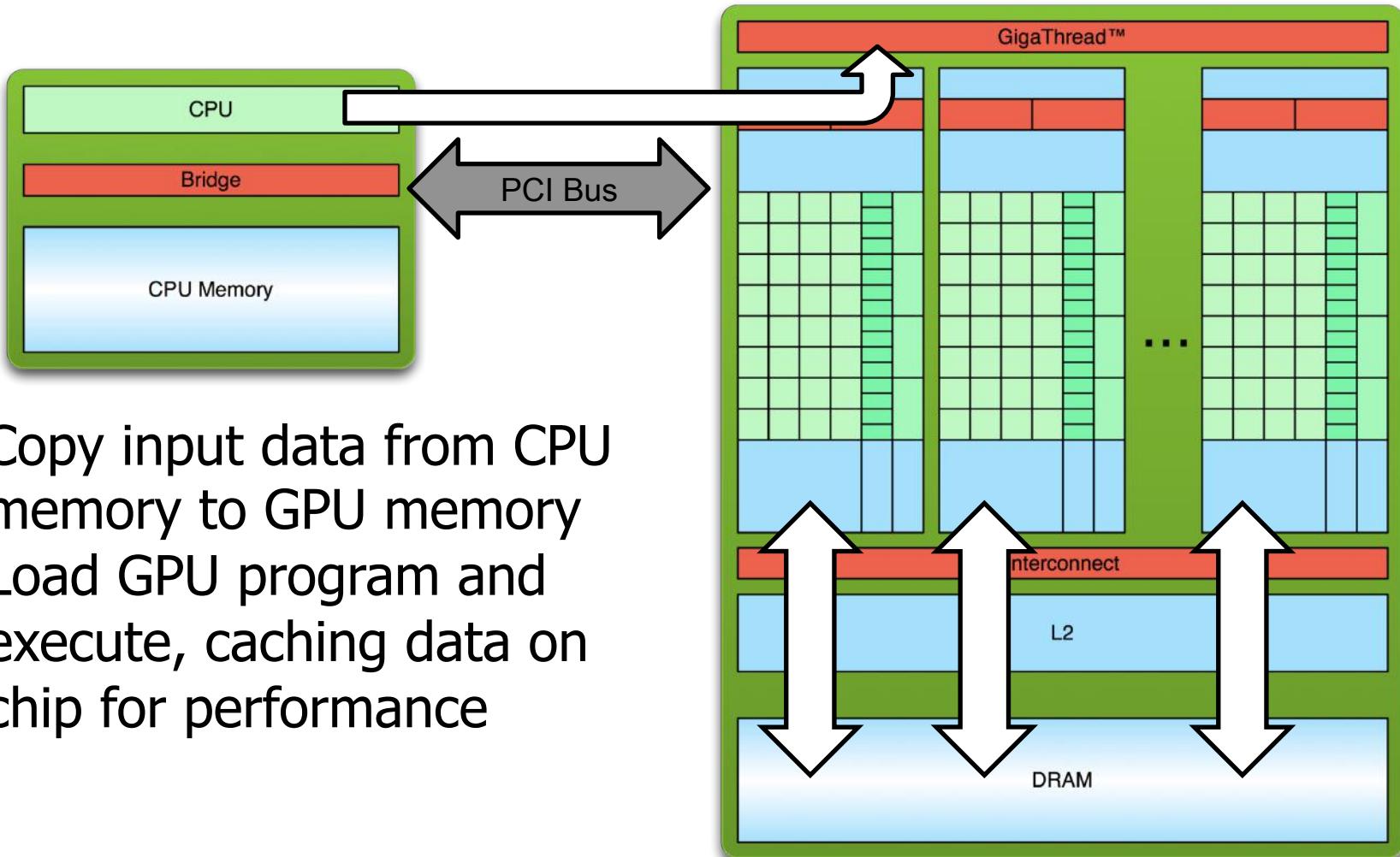
Device



# Processing Flow

Host

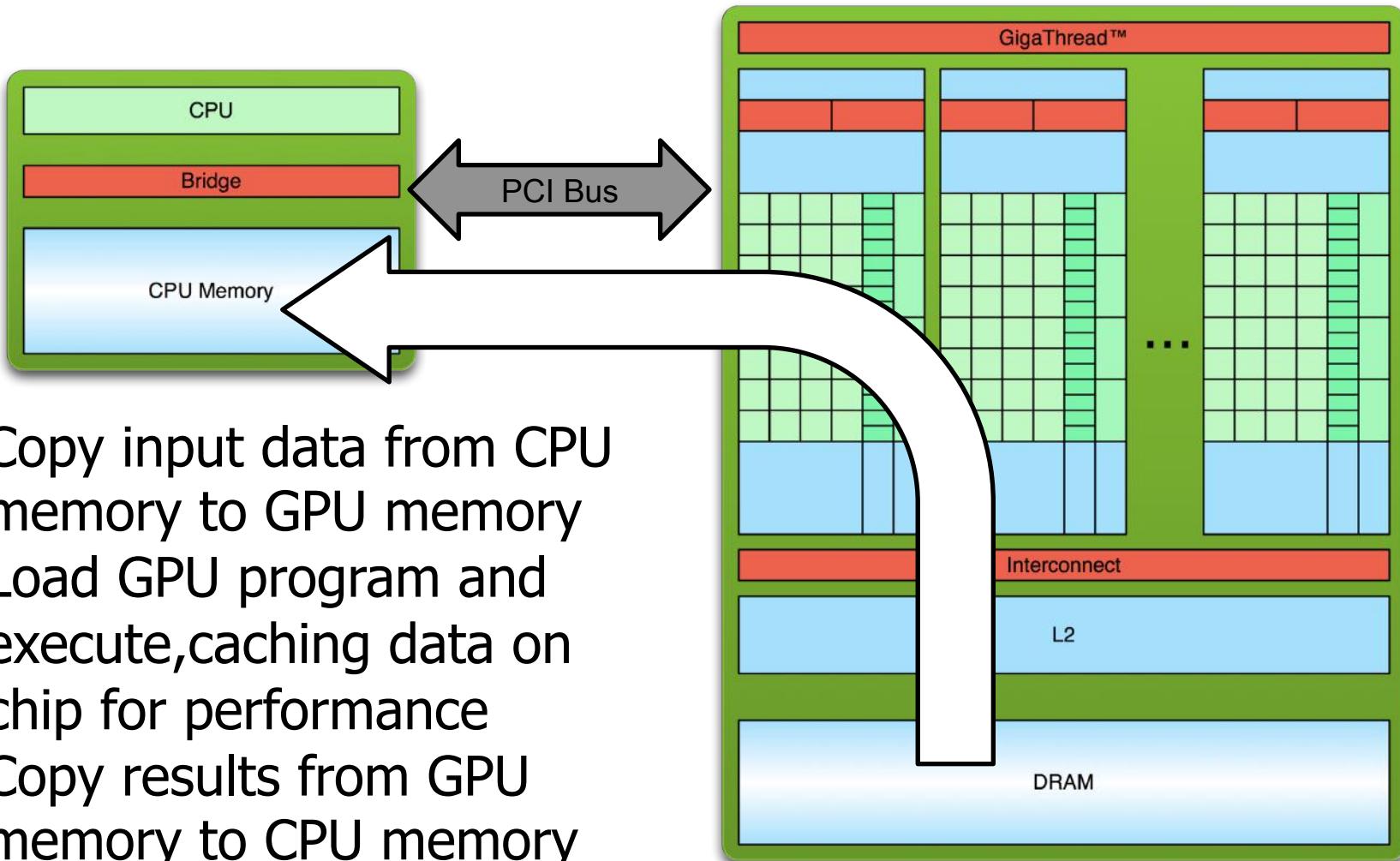
Device



# Processing Flow

Host

Device



# *Hello World! CPU C code*

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C code that runs on the host
- NVIDIA compiler (nvcc) can be used to compile

```
$> nvcc hello_world_cpu.cu
$> ./a.out
Hello World!
```

# *Hello World! with Device Code*

```
__global__ void my_kernel(void){  
}  
  
int main(void) {  
    my_kernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Contains code that is executed on the device (though doing nothing)
- Two new syntactic elements...

# *Hello World! with Device Code*

- CUDA C keyword `__global__` indicates a function that
  - runs on the device (and must return `void`)
  - is 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

# *Hello World! with Device Code*

- Triple angle brackets mark a call from host code to device code
  - called kernel launch
  - Parameters in brackets explained later

```
my_kernel<<<1,1>>>( );
```

- It's that simple to execute a function on the GPU

# *Hello World! with Device Code*

```
__global__ void my_kernel(void){  
}  
  
int main(void) {  
    my_kernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- `my_kernel` does nothing...
- So let's begin making the GPU do some work...

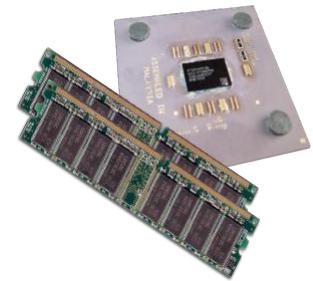
# *Addition on the device*

```
__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;
```

# Addition on the device

```
// 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;
}
```

# ***Addition on the device***

- OK, so we know the basic workflow:
  1. Allocate memory on the device
  2. Copy input data to the device
  3. Launch a kernel on the device
  4. Copy results back to the host
  5. Deallocate memory on the device
- Let's move on to parallel computing on the GPU using CUDA

# *Parallelization with CUDA using blocks*

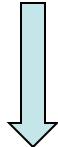
# *Parallelization with CUDA*

- 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
```

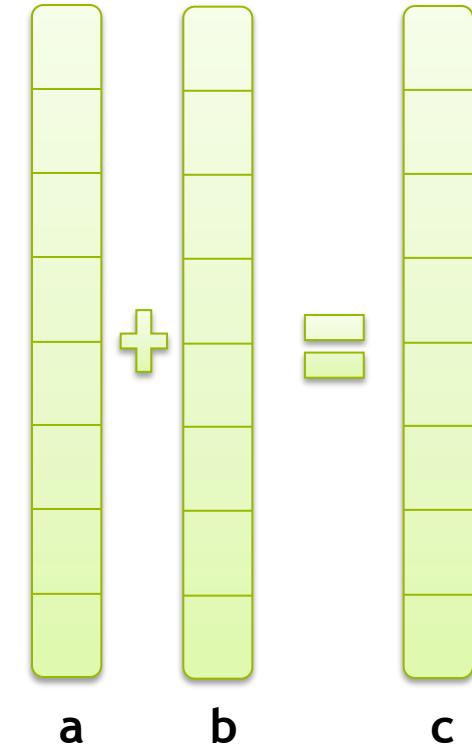
- The GPU is good at
  - efficiently launching lots of threads
  - running lots of threads in parallel  
(many more than processors on the device)
- But why launch N identical copies?

# *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];
    }
}
```

- GPUs can do 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 of available compute cores

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
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

# Review

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Kernels are launched by CPU in parallel
  - Launch N blocks (copies) of `add()` with  
`add<<<N,1>>>(...);`
  - Use `blockIdx.x` to access block index

# *Parallelization with CUDA using blocks and threads*

# **CUDA threads**

- Blocks can be split into parallel **threads**
- 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 needs to be consistent

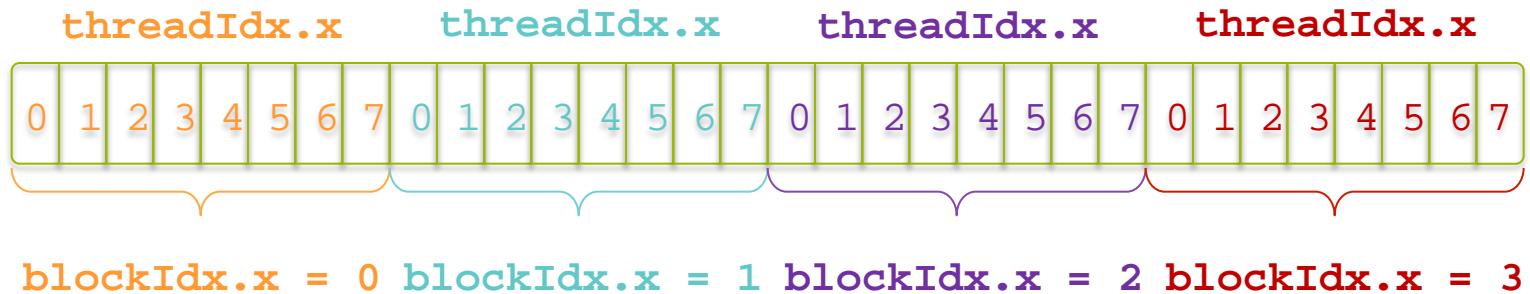
```
// launch N copies  
add<<<1,N>>>(d_a, d_b, d_c);
```

# *CUDA blocks and threads*

- We can use blocks or threads
  - Many blocks with one thread
  - One block with many threads
- Number of allowed threads per block is limited by hardware (typically 1024)
- We can combine blocks and threads
  - need to take care with indexing
- CUDA supports 3D blocks and threads  
(more later)

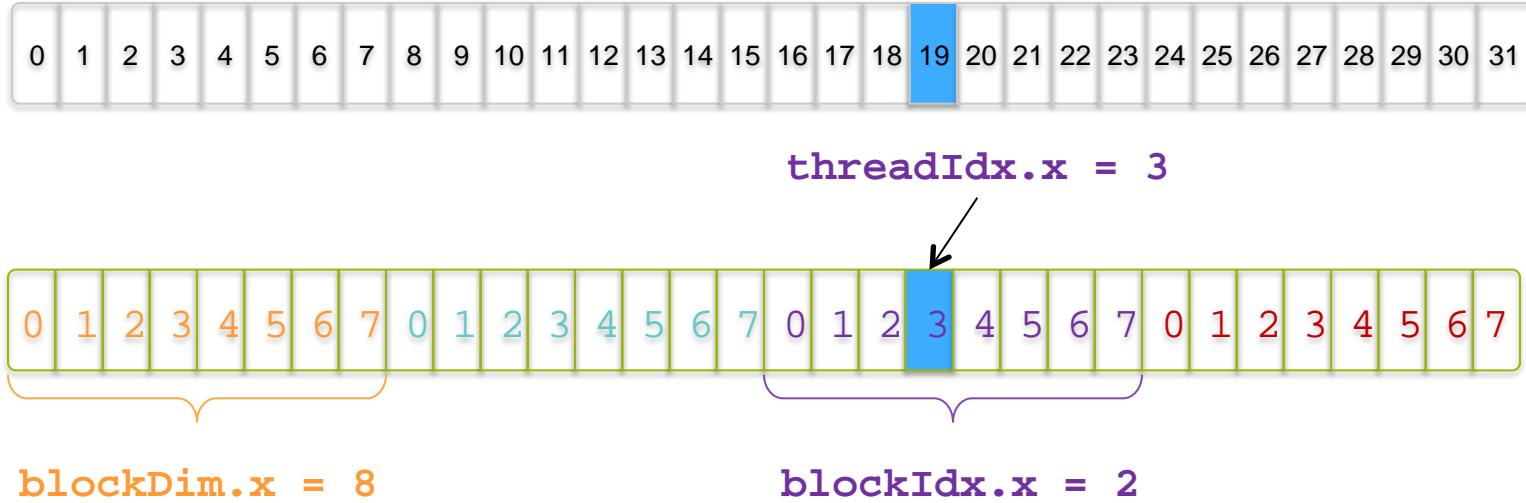
# *Indexing with blocks and threads*

- Example to compute offset into an array
  - 1 element per thread
  - 8 threads per block



- Built-in variable **blockDim.x** knows number of threads per block (block dimension)
  - 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);
```

# *Handle 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];
}
```

# *Why blocks and threads?*

- Combination of threads and blocks seems to add complexity. Reason is mapping to hardware.
- Threads within a block can
  - communicate
  - synchronize
- The number of threads per block is limited by the hardware
- Note: Thread block size should be a multiple of 32 for performance reasons since kernels issue instructions in warps (32 threads)

# 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

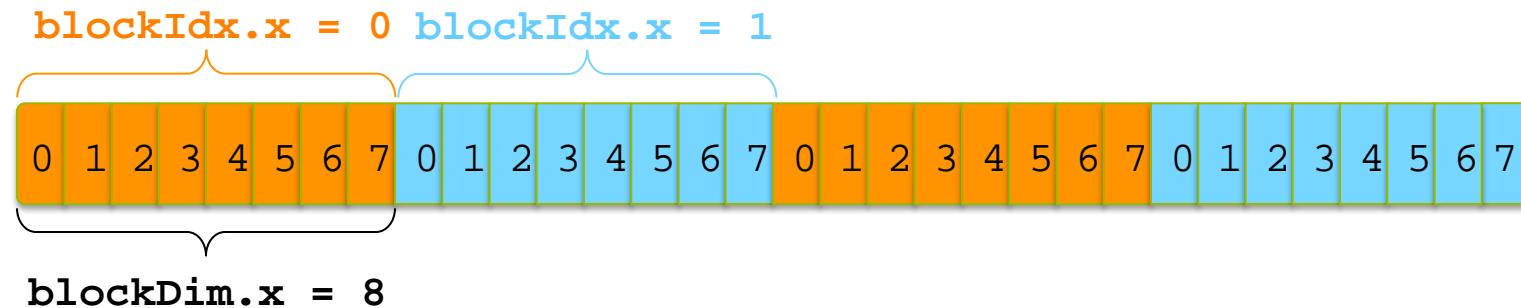
```
tid = threadIdx.x + blockIdx.x * blockDim.x
```

# *Handle arbitrary vector sizes (2)*

- Maximum grid size is limited (usually 65,535)
- Maximum block size is limited (usually 1,024)
- 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** knows number of blocks in grid
- For many kernels, performance will be optimal for a GPU-hardware dependent combination of grid / block size

# Handle arbitrary vector sizes (2)

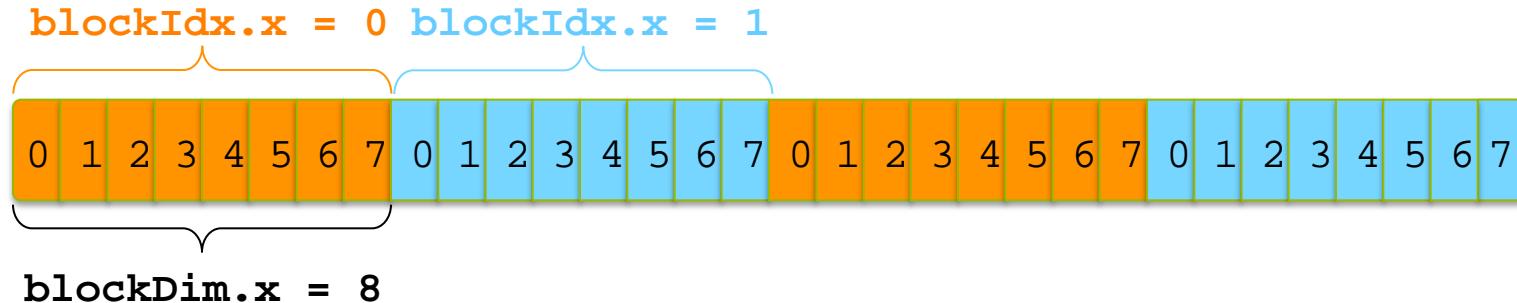
- Example: If we launch 2 blocks with 8 threads each, then kernels with
  - `blockIdx.x=0` need to work on blocks 0 and 2
  - `blockIdx.x=1` need to work on blocks 1 and 3



- 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);
```

# *Vector Add Exercise*

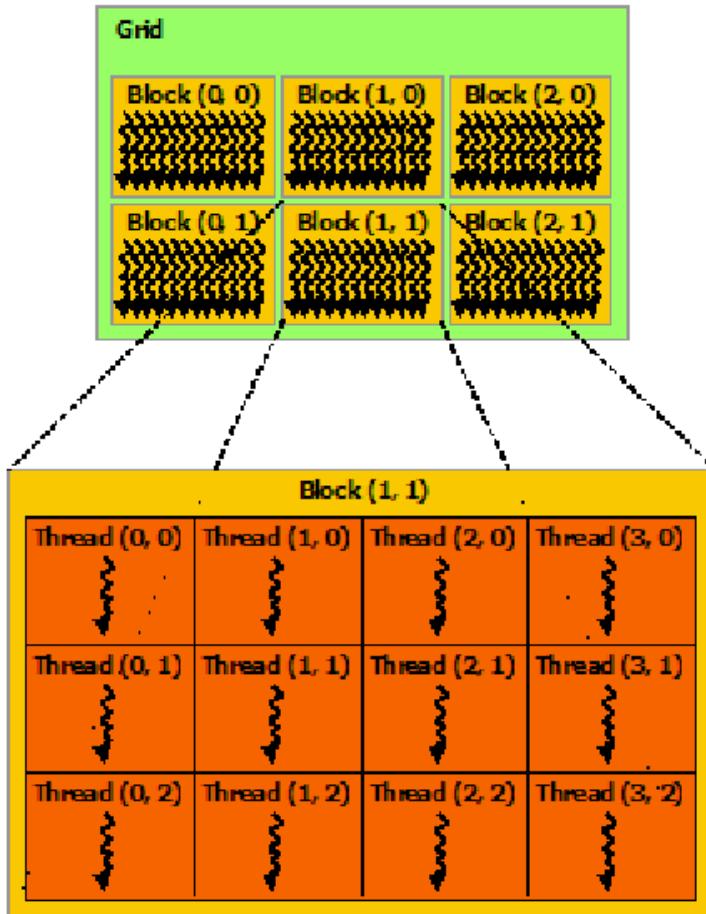
- Compile and run the code in `cuda-samples/addition`
- Go to subdirectory `exercise` and try to complete the code
- (look for `/* FIXME */` comments)

# *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
- 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*



- 2D grid with 2D blocks

```
kernel<<<dim3( 2 , 3 ) ,  
           dim3( 3 , 4 ) >>> ( );
```

- Launches many threads,  
72 in this example

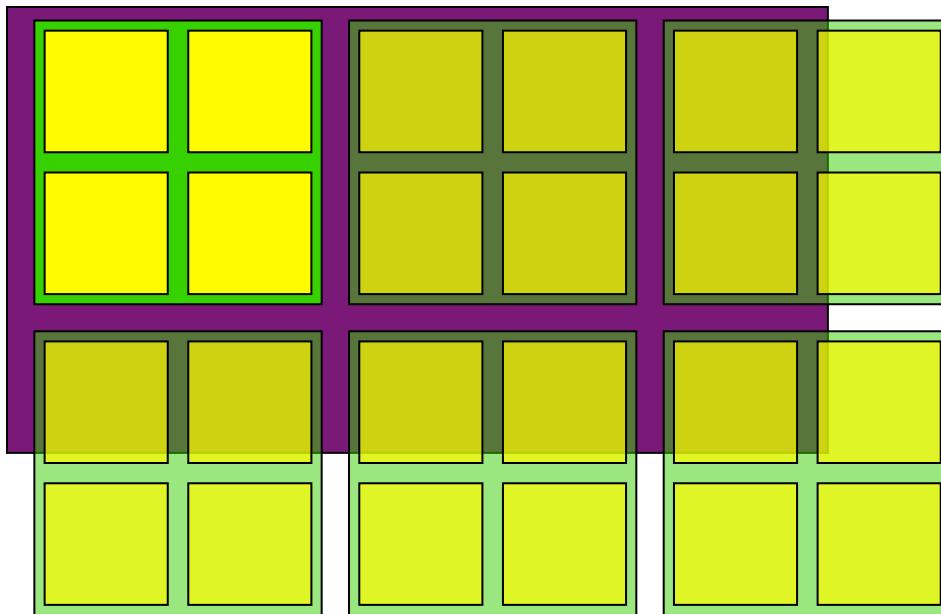
```
gridDim.x * gridDim.y  
* blockDim.x * blockDim.y
```

# *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:

```
__global__ void square(int *arr, int maxrow, int maxcol){  
    // indices and strides  
    int row = 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"  
    while (row < maxrow) {  
        int col = colinit;  
        while (col < maxcol) {  
            pos = row * maxcol + col  
            arr[pos] *= arr[pos]  
            col += colstride  
        }  
        row += rowstride  
    }  
}
```

# *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  $256 \times 256 = 65,536$  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);
```

# *Example: 2D array*

```
// 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;
}
```

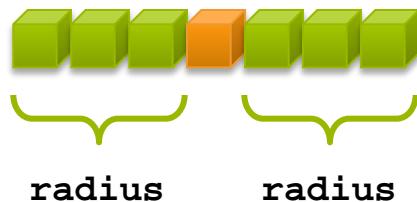
# *Square Array Exercise*

- Compile and run the code in  
`cuda-samples/square_array`
- Go to subdirectory `exercise` and try to complete the code
- (look for `/* FIXME */` comments)

# *Communication among threads – shared memory*

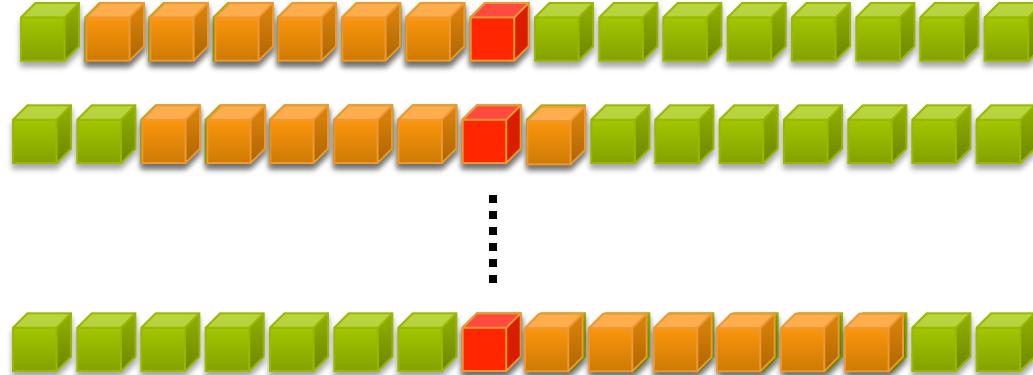
# *Example – 1D stencil*

- 1D stencil for 1D array:
  - Each output elements 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:



# *1D stencil – using blocks*

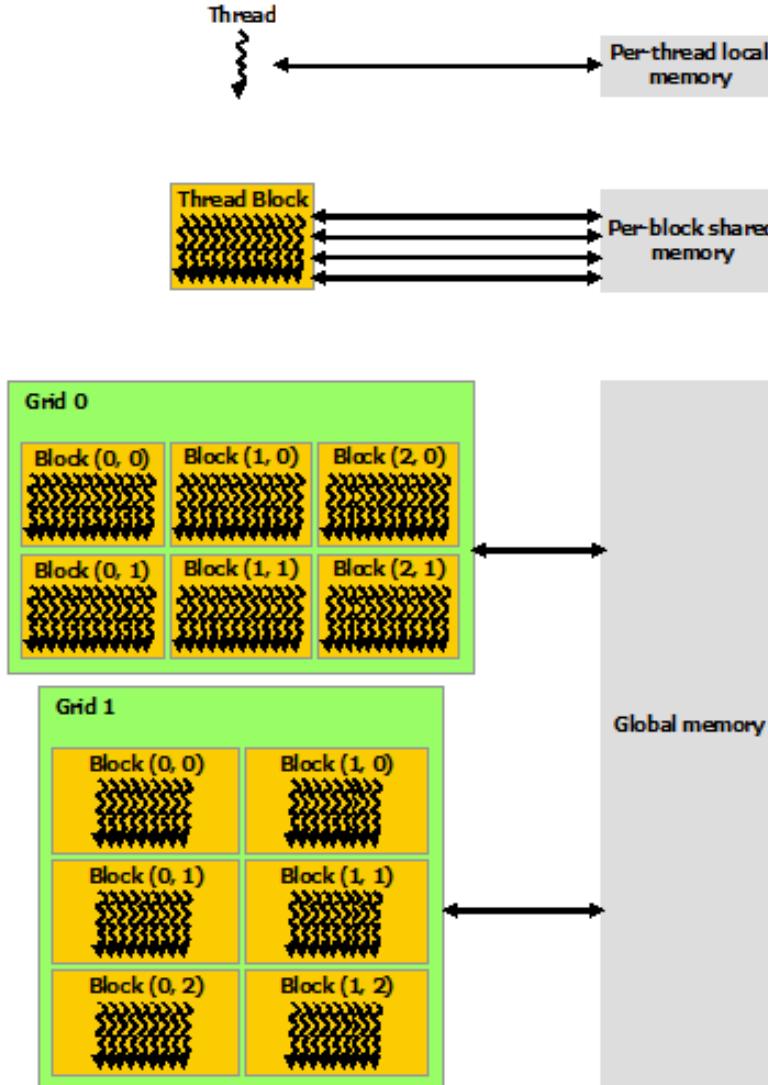
- Each thread processes one output element
  - `blockDim.x` elements are processed per block
- As a consequence, input elements have to be read several times from slow global memory
  - with radius 3, each input element is read 7 times!



# *Sharing data between threads*

- 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

# GPU memory hierarchy

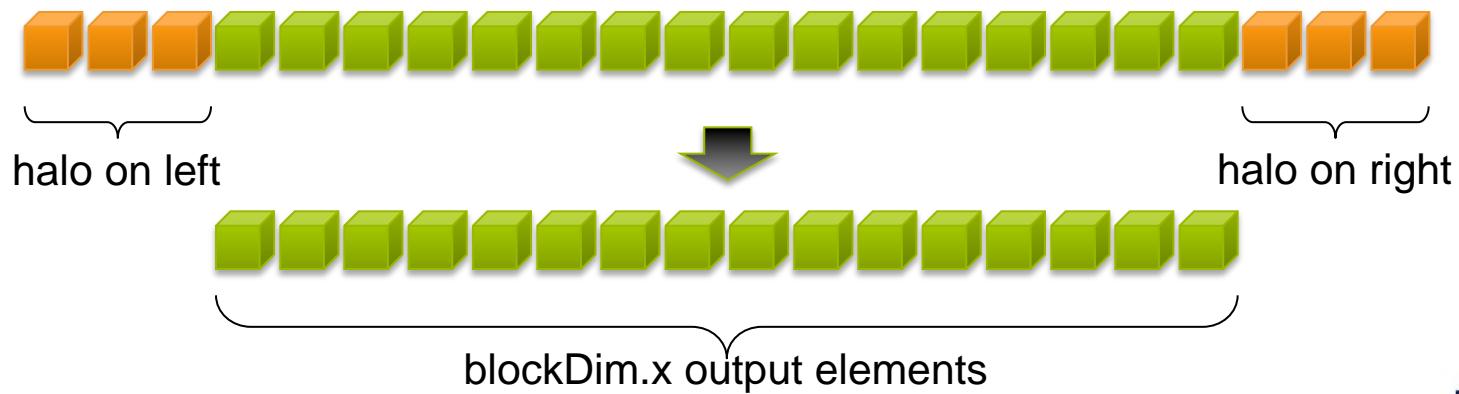


Memory	Latency (cycles)	Cached	Privacy
Global	100s	Yes	Application
Local	100s	Yes	Thread
Constant	1s-100s	Yes	Application
Texture	1s-100s	Yes	Application
Shared	1	–	Block
Register	1	–	Thread

- **On-chip**
  - Registers, shared mem
- **Off-chip**
  - Constant, texture cache (R/O)
  - Local, global memory

# *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 kernel

```
__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];  
          
    }  
}
```

# *1D stencil kernel*

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

// Store the result
out[gindex] = result;
}
```

- Unfortunately the code as it is will not work
- What is the problem?

# 1D stencil kernel

- We have a data race!
- Suppose thread 15 reads the halo in position 19 before thread 0 has fetched it:

```
// Read input elements into shared memory
temp[lindex] = in[gindex]; // store at temp[18]
  
  
if (tid < RADIUS) { // skipped, 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]

```

# ***Synchronizing threads 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

```
__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  
    __syncthreads();
```

# *1D stencil kernel*

```
// Now it is safe to 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 and avoids race conditions by synchronizing the threads within a block

# *1D Stencil Exercise*

- Compile and run the code in  
`cuda-samples/1d_stencil`
- Go to subdirectory exercise and try to complete the code
- (look for `/* FIXME */` comments)

# *Review*

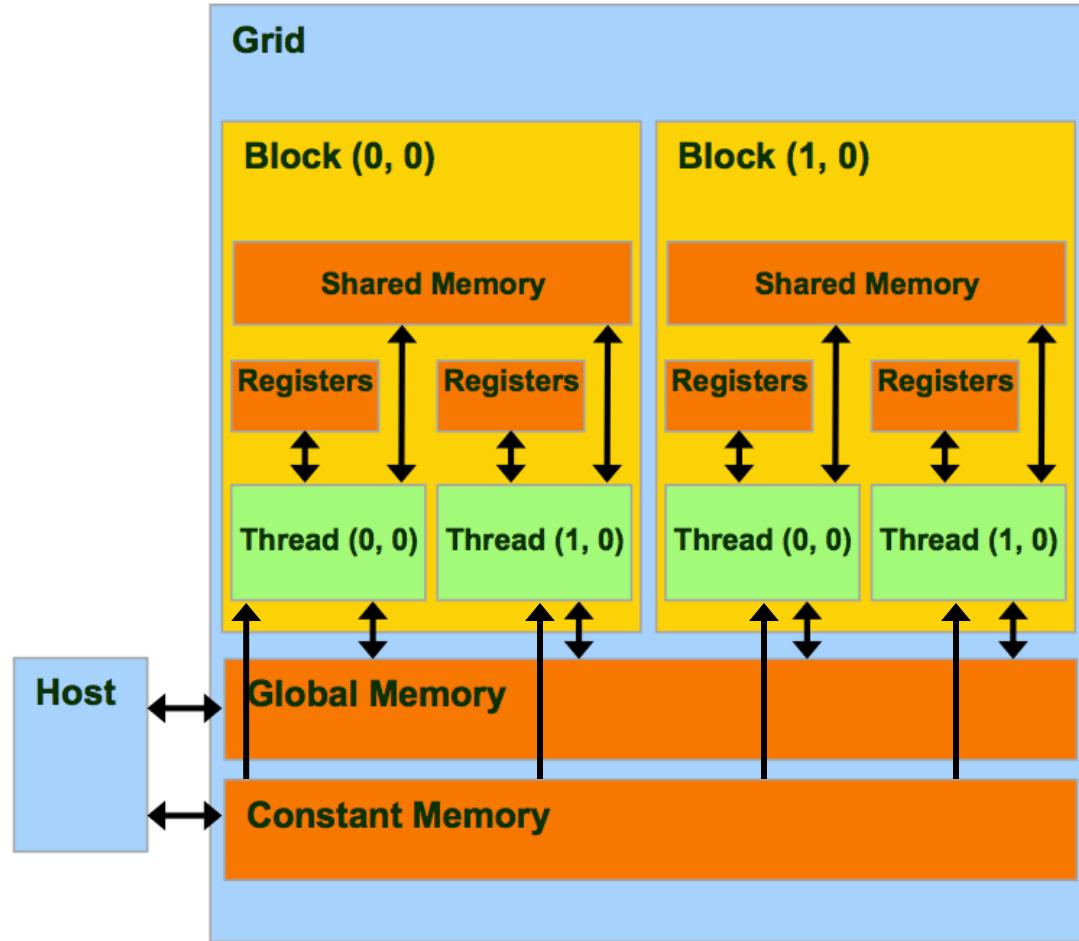
- Use **`__shared__`** to declare a variable / array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use **`__syncthreads()`** as a barrier
  - Required to prevent data hazards / race conditions

# **Constant memory**

- 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 memories

- Each thread can
  - read/write per-thread **registers**
  - read/write per-thread **local memory**
  - read/write per-block **shared memory**
  - read/write per-grid **global memory**
  - read-only per-grid **constant memory**



# *General programming strategy*

- Global memory resides in device memory (DRAM)
  - much slower access than shared memory
- 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 can access data multiple times)
  - copy results from shared memory to global memory

# ***General programming strategy***

- Use constant memory for 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
- Finally, there is texture memory (skipped here)

# *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?

# *Atomic operations*

- 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 (256) possible value
- 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 on a GPU*

- 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[i]]), 1 );
        tid += stride;
    }
}
```

- This will work – but with terrible performance

# *Histogram on a GPU*

- This kernel will have a terrible performance compared to the CPU because
  - the kernel does very little work
  - thousands of threads access few (256) memory locations with atomic add operations
- We can improve the 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 on a GPU*

- 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
```

# *Histogram on a GPU*

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

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

# *Histogram on a GPU*

- Our main program

```
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);
```

# *Histogram on a GPU*

```
// 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;
}
```

# *Device management etc*

# *Coordinating host & 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",
           cudaGetString(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 )
```

# *Query device properties*

- 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;  
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 is 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);
```

# CUDA streams

```
// 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);

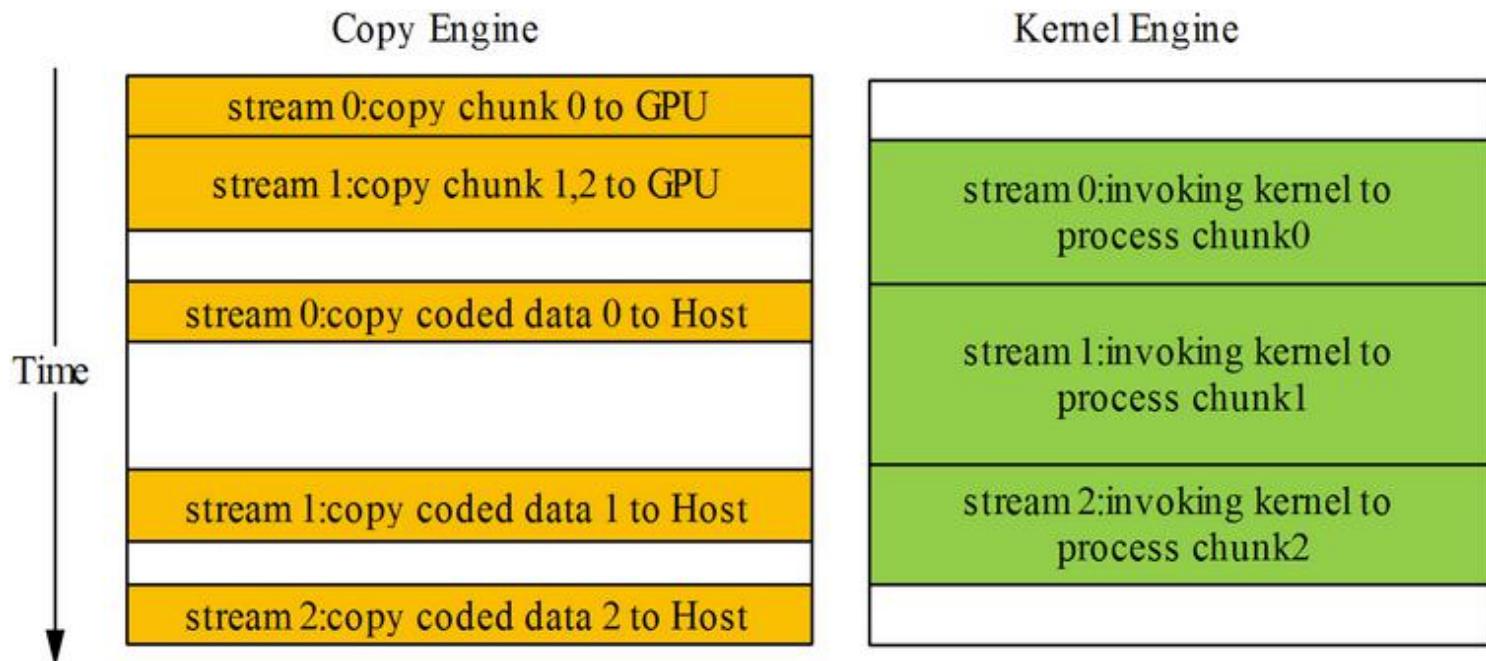
// host does other stuff
do_some_work();

// 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 and overlap memory copies in one stream with computation in a second stream



# 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);
```

# CUDA streams

```
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
```

# *GPU Computing with OpenACC Directives*

based on material by  
Mark Harris - NVIDIA corporation

# A Very Simple Exercise: SAXPY

## SAXPY in C

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

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

## SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy
...
!
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

# *Directive Syntax*

- Fortran

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

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

```
!$acc end directive
```

- C

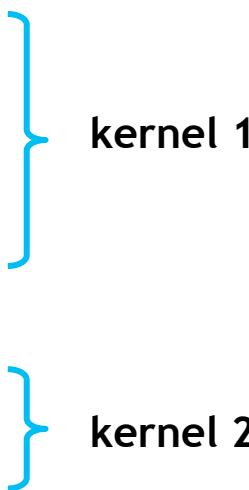
```
#pragma acc directive [clause [,] clause] ...]
```

Often followed by a structured code block

# *kernel*s: Your first OpenACC Directive

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:**  
A parallel  
function that runs  
on the GPU

# *kernel*s Construct

Fortran

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

C

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

Clauses

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

Also, any data clause (more later)

# Complete SAXPY example 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*

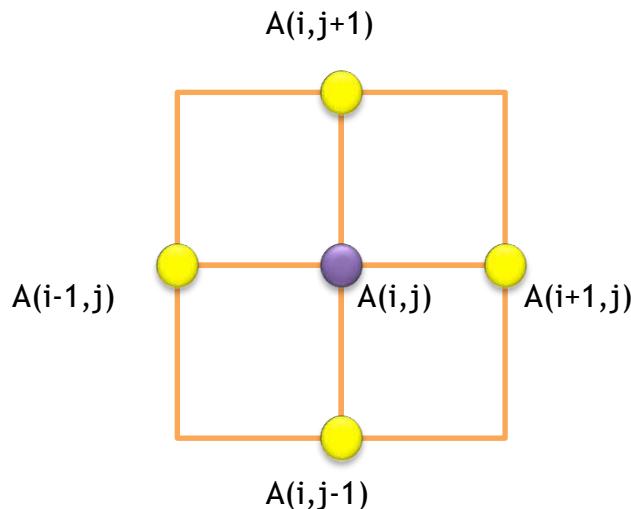
- C:  
`pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c`
- Fortran:  
`pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90`
- Compiler output:

```
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 */
```

# *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:  $\Delta\varphi(x, y) = 0$



# Jacobi Iteration C Code

```
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

# Jacobi Iteration Fortran Code

```
do while ( err > tol .and. iter < iter_max )  
    err=0._fp_kind
```



Iterate until converged

```
do j=1,m  
    do i=1,n
```



Iterate across matrix elements

```
    Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &  
                                A(i , j-1) + A(i , j+1))
```



Calculate new value from neighbors

```
    err = max(err, Anew(i,j) - A(i,j))  
end do  
end do
```



Compute max error for convergence

```
do j=1,m-2  
    do i=1,n-2  
        A(i,j) = Anew(i,j)  
    end do  
end do
```



Swap input/output arrays

```
iter = iter +1  
end do
```

# OpenMP C Code

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
  
#pragma omp parallel for shared(m, n, Anew, A)  
    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 omp parallel for shared(m, n, Anew, A)  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

Parallelize loop across  
CPU threads

Parallelize loop across  
CPU threads

# OpenMP Fortran Code

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j    ) + A(i-1, j    ) + &
                                  A(i    , j-1) + A(i    , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do

!$omp parallel do shared(m,n,Anew,A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

  iter = iter +1
end do
```

Parallelize loop across  
CPU threads

Parallelize loop across  
CPU threads

# GPU startup overhead

- If no other GPU process running, GPU driver may be swapped out
  - Linux specific
  - Starting it up can take 1-2 seconds
- Two options
  - Run `nvidia-smi -pm 1` in persistence mode (requires root permissions)
  - Run “`nvidia-smi -q -l 30`” in the background
  - Any well configured supercomputer should be running the driver in persistence mode.
- If your running time is off by ~2 seconds from results in these slides, suspect this
  - Nvidia-smi should be running in persistent mode for these exercises

# First Attempt: OpenACC C

```
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

# First Attempt: OpenACC Fortran

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

 !$acc kernels
 do j=1,m
   do i=1,n

     Anew(i,j) = .25_fp_kind * (A(i+1, j    ) + A(i-1, j    ) + &
                                 A(i    , j-1) + A(i    , j+1))

     err = max(err, Anew(i,j) - A(i,j))
   end do
 end do
 !$acc end kernels

 !$acc kernels
 do j=1,m-2
   do i=1,n-2
     A(i,j) = Anew(i,j)
   end do
 end do
 !$acc end kernels
 iter = iter +1
end do
```

Generate GPU kernel  
for loop nest



Generate GPU kernel  
for loop nest



# *First Attempt: Compiler output (C)*

```
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
```

# *First Attempt: Performance*

**SDSC Comet**

CPU: Intel Xeon X5680  
6 Cores @ 3.33GHz

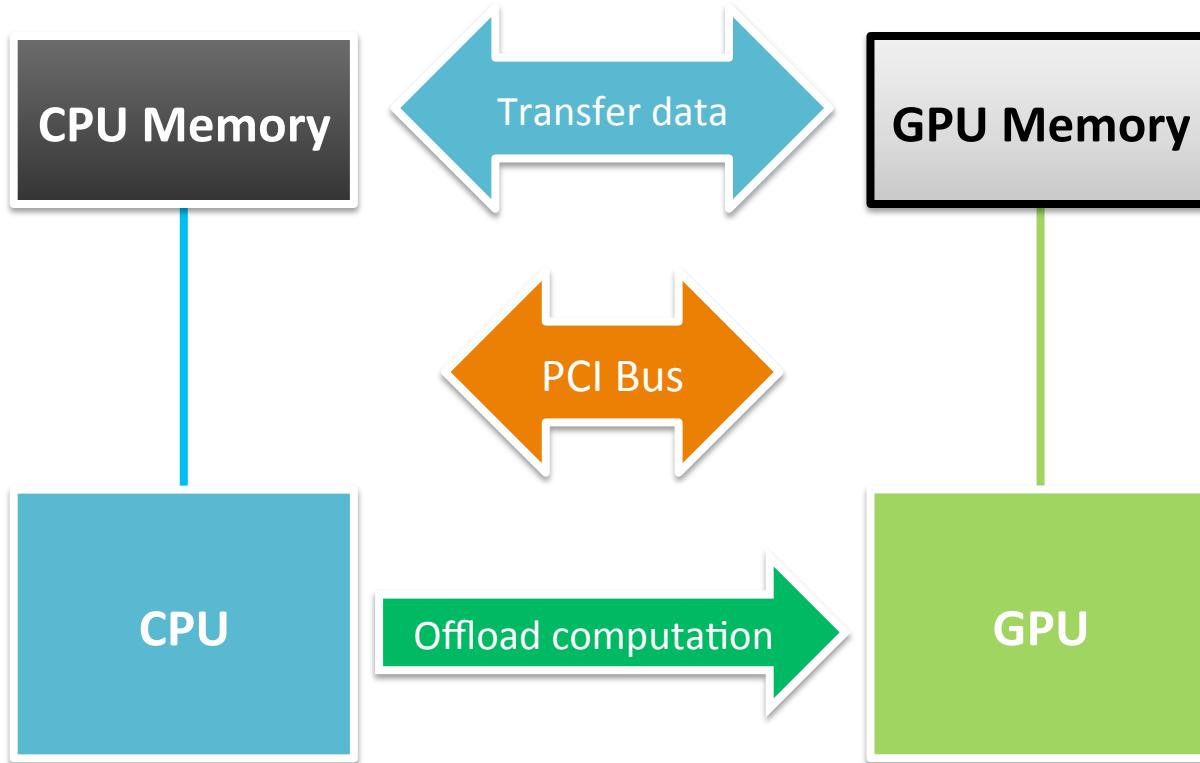
GPU: NVIDIA Tesla K80  
(using single GPU)

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

**Speedup vs.  
1 CPU core**

**Speedup vs.  
6 CPU cores**

# *Basic Concepts*



For efficiency, decouple data movement and compute off-load

# Excessive Data Transfers

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

A, Anew resident on host

#pragma acc kernels

Copy

A, Anew resident on accelerator

These copies  
happen every  
iteration of the  
outer while loop!\*

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

Copy

A, Anew resident on accelerator

A, Anew resident on host

...

\*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!

# ***DATA MANAGEMENT***

# DATA CONSTRUCT

Fortran

```
!$acc data [clause ...]  
    structured block  
!$acc end data
```

C

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

General Clauses

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

Manage data movement. Data regions may be nested.

# 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`.

# ARRAY SHAPING

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”
- C

```
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```
- Fortran

```
!$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```
- Note: data clauses can be used on data, kernels or parallel

# UPDATE CONSTRUCT

Fortran

```
!$acc update [clause ...]
```

C

```
#pragma acc update [clause ...]
```

Clauses

*host( list )*

*if( expression )*

*device( list )*

*async( expression )*

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.  
Data movement can be conditional, and asynchronous.

# Second Attempt: OpenACC C

```
#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



# Second Attempt: OpenACC Fortran

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
    err=0._fp_kind

 !$acc kernels
 do j=1,m
    do i=1,n

        Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                     A(i , j-1) + A(i , j+1))

        err = max(err, Anew(i,j) - A(i,j))
    end do
 end do
 !$acc end kernels

 ...
iter = iter +1
end do
 !$acc end data
```

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

# *Second Attempt: Performance*

**SDSC Comet**

CPU: Intel Xeon X5680  
6 Cores @ 3.33GHz

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**

# *Further speedups*

- OpenACC gives us more detailed control over parallelization
  - Via **gang**, **worker**, and **vector** clauses
- 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

# *Finding Parallelism in your code*

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

# *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

# *OpenACC Learning Resources*

- OpenACC info, specification, FAQ, samples, and more
  - <http://openacc.org>
- PGI OpenACC resources
  - <http://www.pgroup.com/resources/accel.htm>

***COMPLETE OPENACC API***

# Kernels Construct

## Fortran

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

## C

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

## Clauses

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

Also any data clause

# 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 } kernel 1

do i=1,n
    a(i) = b(i) + c(i)
end do } kernel 2
!$acc end kernels
```

# Parallel Construct

## Fortran

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

## Clauses

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

## C

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

```
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 parallel loop [clause  
!$acc kernels loop [clause ...] ...]  
                           !$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 be 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