

Introduction to CUDA Programming

Jian Tao

jtao@tamu.edu

Spring 2021 HPRC Short Course

04/23/2021



Texas A&M Engineering
Experiment Station

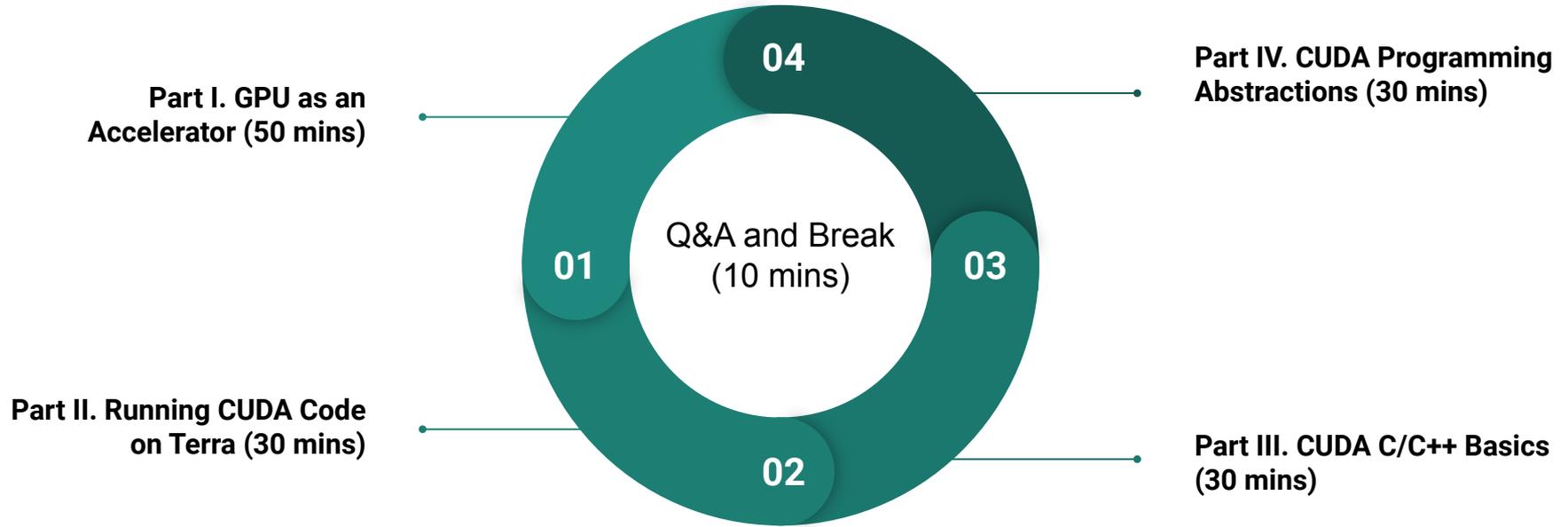


High Performance
Research Computing
DIVISION OF RESEARCH



TEXAS A&M
Institute of
Data Science

Introduction to CUDA Programming



Connecting to HPRC Portal

High Performance Research Computing

TEXAS A&M HIGH PERFORMANCE RESEARCH COMPUTING

Home User Services Resources Research Policies Events About Portal

Ada Portal
Terra Portal

Quick Links

- New User Information
- Accounts
 - Apply for Accounts
 - Manage Accounts
- User Consulting
- Training
- Documentation
- Software
- FAQ

User Guides

- Ada

Molecular Jump-Rope: Multiringed Metal-Complexes That Really Know How To Jump

"These platinum complexes can undergo a 'triple-jump rope' mechanism rendering the three methylene chains of their ligands equivalent, a motion that is unheard of and reminiscent of Olympic traditions such as the triple-Axel or the triple jump."
-- Dr. John Gladysz, Department of Chemistry

HPRC Portal

* VPN is required for off-campus users.

Login HPRC Portal (Terra)

High Performance Research x +

hprc.tamu.edu

ATM TEXAS A&M HIGH PERFORMANCE RESEARCH COMPUTING

Home User Services Resources Research Policies Events About Portal

Ada Portal

Terra Portal

Quick Links

- New User Information
- Accounts
 - Apply for Accounts
 - Manage Accounts
- User Consulting
- Training
- Documentation
- Software
- FAQ

User Guides

- Ada

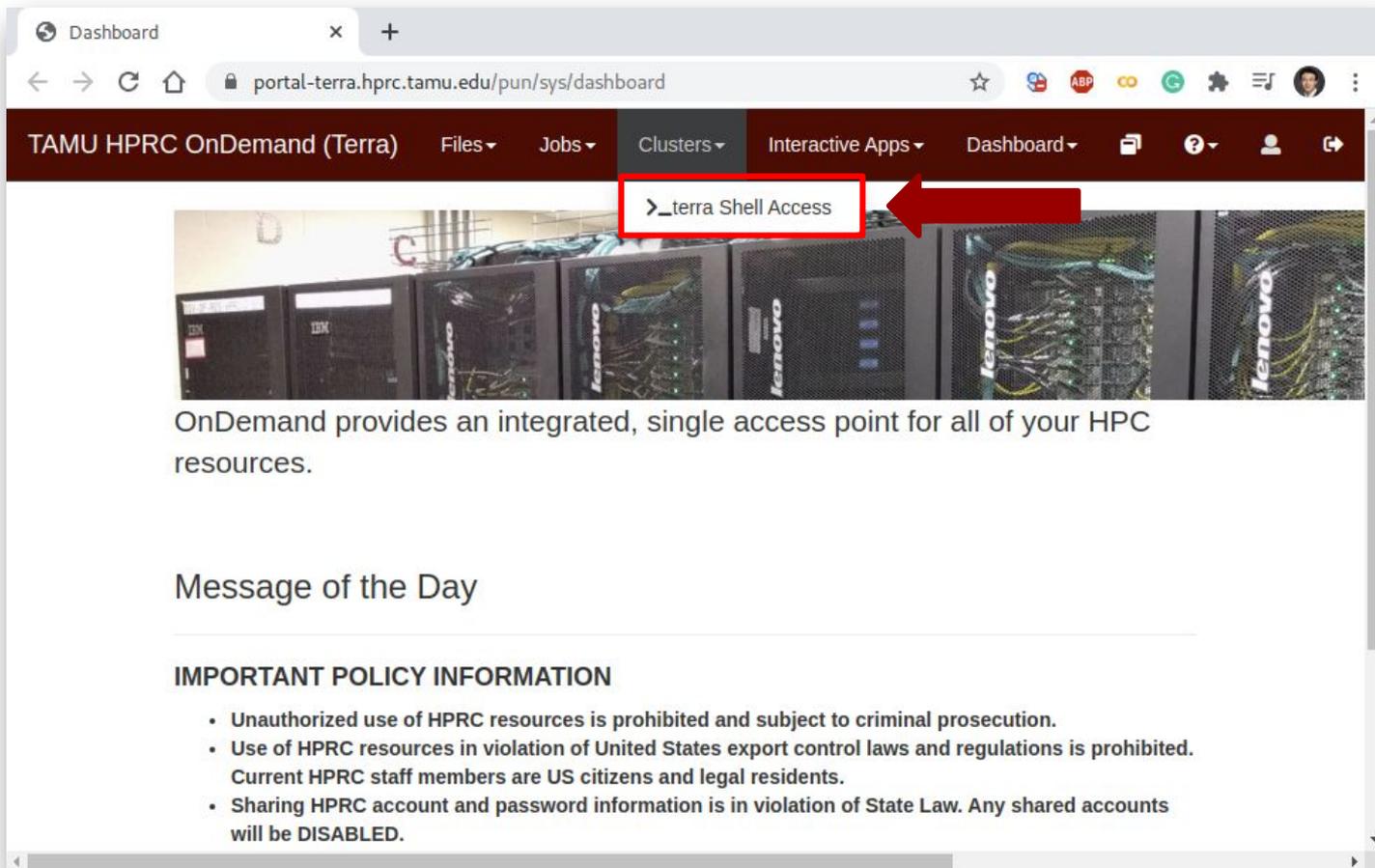
Molecular Jump-Rope: Multiringed Metal-Complexes That Really Know How To Jump

"These platinum complexes can undergo a 'triple-jump rope' mechanism rendering the three methylene chains of their ligands equivalent, a motion that is unheard of and reminiscent of Olympic traditions such as the triple-Axel or the triple jump."
-- Dr. John Gladysz, Department of Chemistry

6 examples

https://portal-terra.hprc.tamu.edu

Terra Shell Access - I



The screenshot shows a web browser window with the URL `portal-terra.hprc.tamu.edu/pun/sys/dashboard`. The navigation bar includes links for "TAMU HPRC OnDemand (Terra)", "Files", "Jobs", "Clusters", "Interactive Apps", and "Dashboard". A red box highlights the "_terra Shell Access" link in the "Clusters" dropdown menu, with a red arrow pointing to it from the right.

OnDemand provides an integrated, single access point for all of your HPC resources.

Message of the Day

IMPORTANT POLICY INFORMATION

- Unauthorized use of HPRC resources is prohibited and subject to criminal prosecution.
- Use of HPRC resources in violation of United States export control laws and regulations is prohibited. Current HPRC staff members are US citizens and legal residents.
- Sharing HPRC account and password information is in violation of State Law. Any shared accounts will be DISABLED.

Terra Shell Access - II

```
*****
This computer system and the data herein are available only for authorized
purposes by authorized users: use for any other purpose is prohibited and may
result in administrative/disciplinary actions or criminal prosecution against
the user. Usage may be subject to security testing and monitoring to ensure
compliance with the policies of Texas A&M University, Texas A&M University
System, and the State of Texas. There is no expectation of privacy on this
system except as otherwise provided by applicable privacy laws. Users should
refer to Texas A&M University Standard Administrative Procedure 29.01.03.M0.02,
Rules for Responsible Computing, for guidance on the appropriate use of Texas
A&M University information resources.
*****

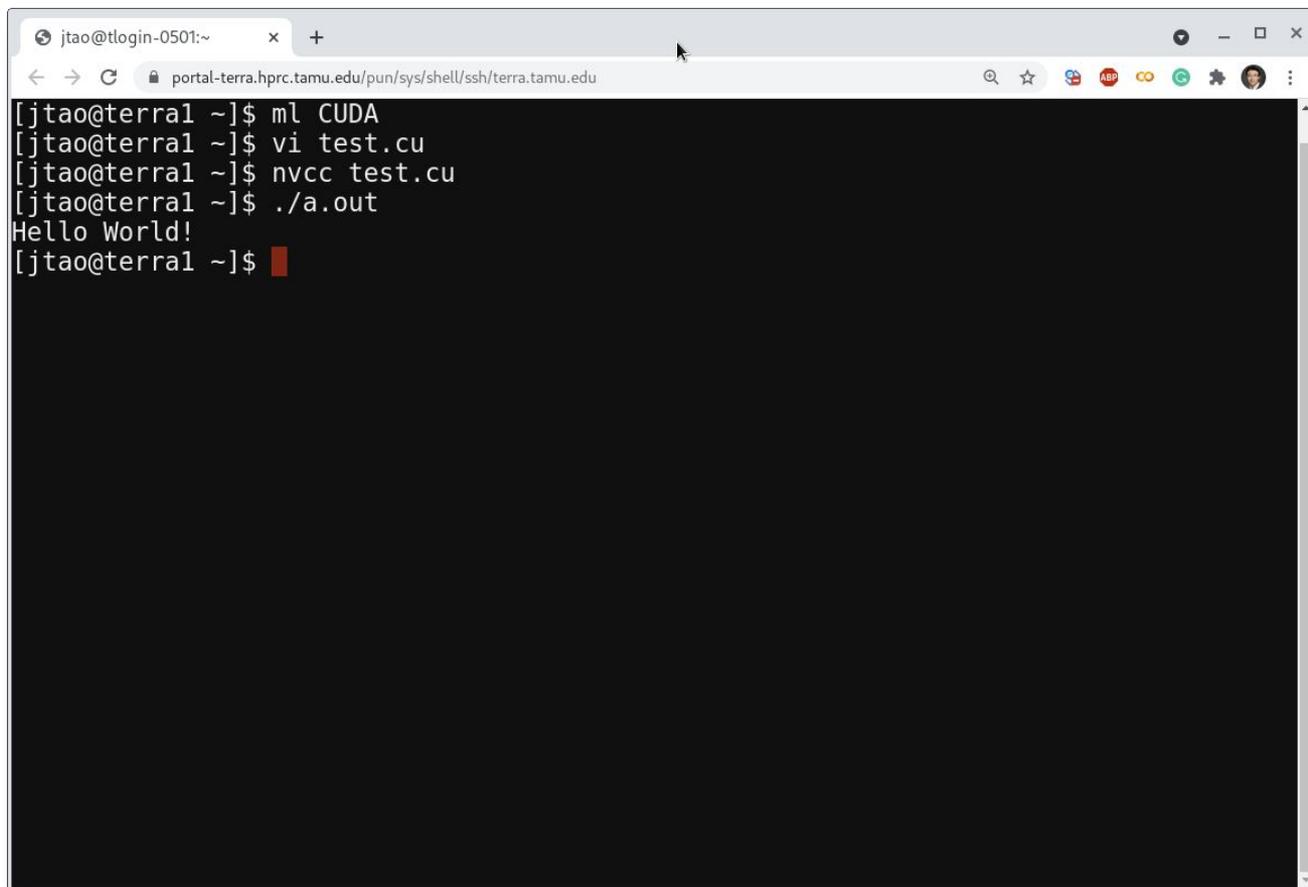
Password:
Duo two-factor login for jtao

Enter a passcode or select one of the following options:

  1. Duo Push to iPhone (iOS)
  2. Duo Push to iPad (iOS)

Passcode or option (1-2): 1
█
```

Load CUDA Module



```
jtao@tlogin-0501:~  
portal-terra.hprc.tamu.edu/pun/sys/shell/ssh/terra.tamu.edu  
[jtao@terra1 ~]$ ml CUDA  
[jtao@terra1 ~]$ vi test.cu  
[jtao@terra1 ~]$ nvcc test.cu  
[jtao@terra1 ~]$ ./a.out  
Hello World!  
[jtao@terra1 ~]$
```

Part I. GPU as an Accelerator



CPU



GPU Accelerator



NVIDIA Tesla A100 with 54 Billion Transistors



Announced and released on May 14, 2020 was the Ampere-based A100 accelerator. With 7nm technologies, the A100 has 54 billion transistors and features 19.5 teraflops of FP32 performance, 6912 CUDA cores, 40GB of graphics memory, and 1.6TB/s of graphics memory bandwidth. The A100 80GB model announced in Nov 2020, has 2.0TB/s graphics memory bandwidth

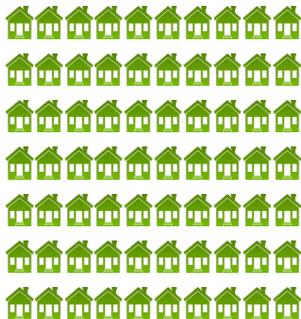
Why Computing Perf/Watt Matters?

2.3 PFlops



7.0
Megawatts

7000 homes

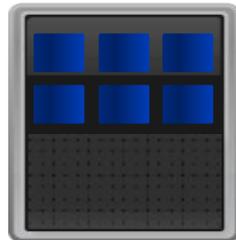


7.0
Megawatts

Traditional CPUs are
not economically feasible

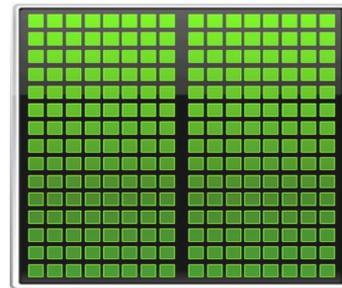
CPU

Optimized for
Serial Tasks



GPU Accelerator

Optimized for Many
Parallel Tasks

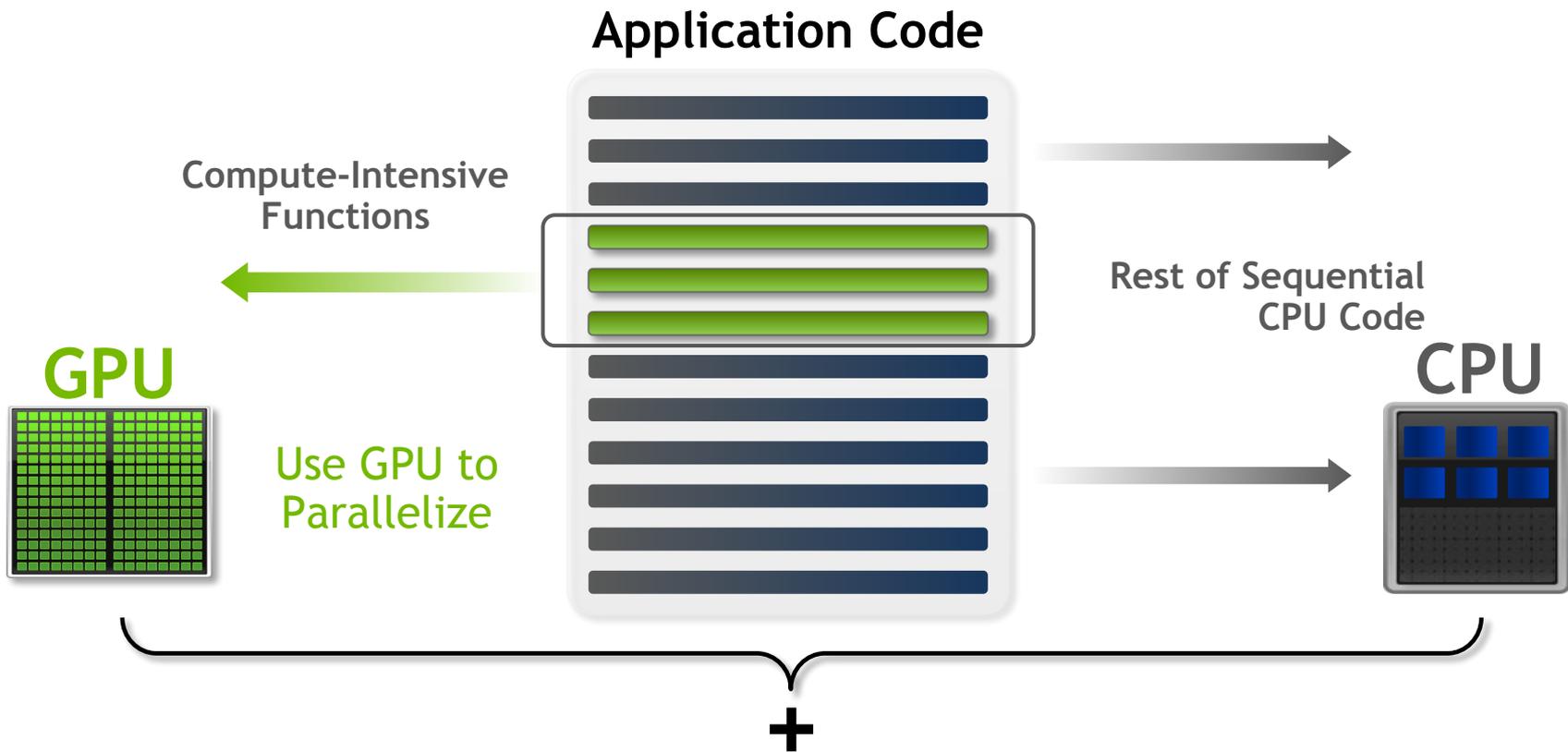


GPU-accelerated computing
started a new era

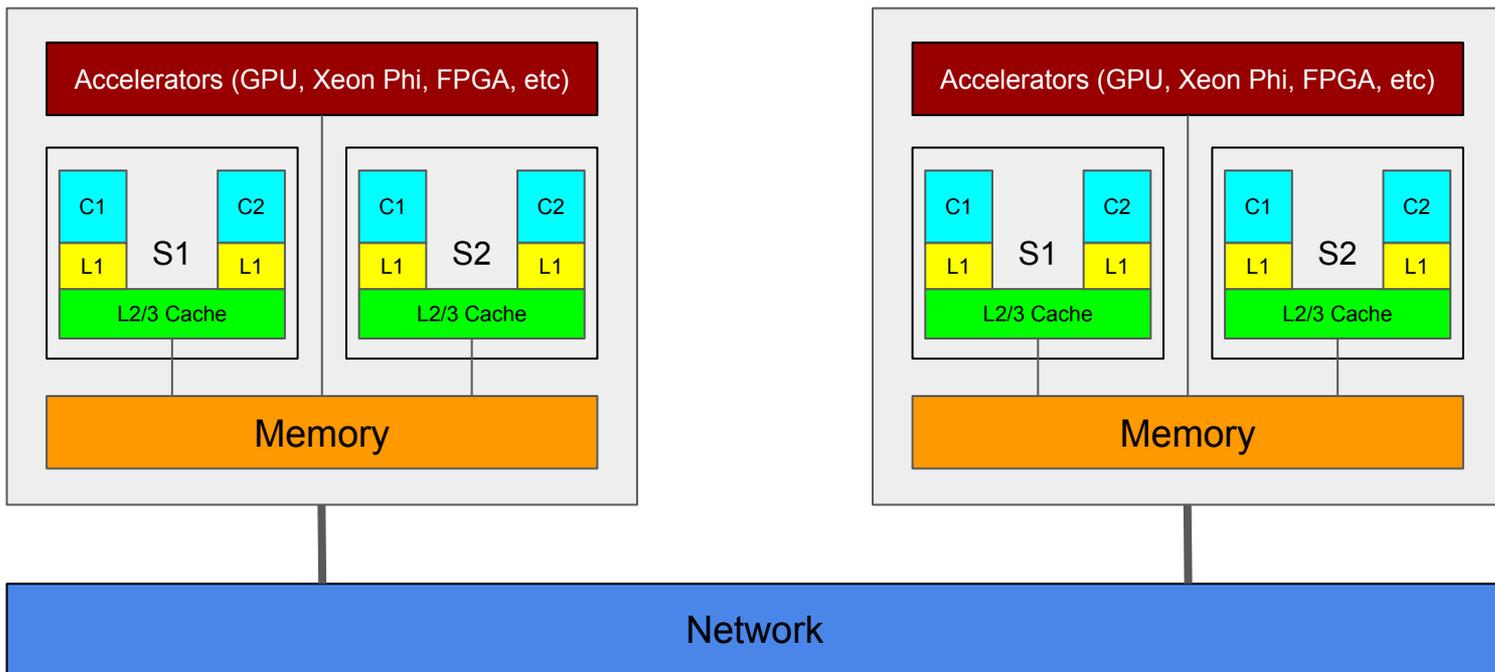
GPU Computing Applications

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			
NVIDIA Ampere Architecture (compute capabilities 8.x)						Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)			GeForce 2000 Series	Quadro RTX Series		Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Quadro GV Series		Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2		GeForce 1000 Series	Quadro P Series		Tesla P Series
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		

Add GPUs: Accelerate Science Applications



HPC - Distributed Heterogeneous System



Programming Models: MPI + (CUDA, OpenCL, OpenMP, OpenACC, etc.)

Amdahl's Law



$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

S is the theoretical speedup of the execution of the whole task;
 s is the speedup of the part of the task that benefits from improved system resources; p is the proportion of execution time that the part benefiting from improved resources originally occupied.

CUDA Parallel Computing Platform

<https://developer.nvidia.com/cuda-toolkit>

Programming
Approaches

Libraries

“Drop-in” Acceleration

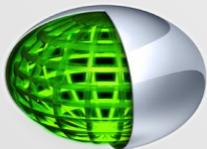
OpenACC
Directives

Easily Accelerate Apps

Programming
Languages

Maximum Flexibility

Development
Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler
Tool Chain



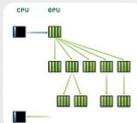
Enables compiling new languages to CUDA platform, and
CUDA languages to other architectures

Hardware
Capabilities

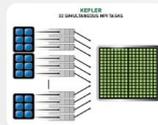
SMX



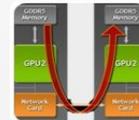
Dynamic Parallelism



HyperQ



GPUDirect



3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

Libraries: Easy, High-Quality Acceleration

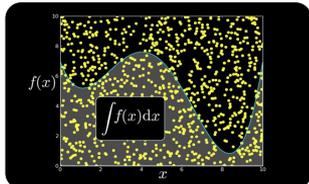
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

NVIDIA CUDA-X GPU-Accelerated Libraries

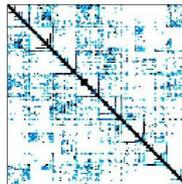
<https://developer.nvidia.com/gpu-accelerated-libraries>



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



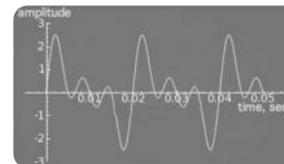
Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



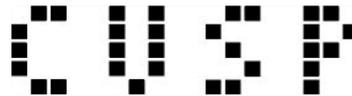
Matrix Algebra on GPI
and Multicore



NVIDIA cuFFT



ArrayFire Matrix
Computations



Sparse Linear
Algebra



C++ STL Features
for CUDA



CUDA-accelerated Application with Libraries

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

`saxpy (...)` ► `cublasSaxpy (...)`

- **Step 2:** Manage data locality

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

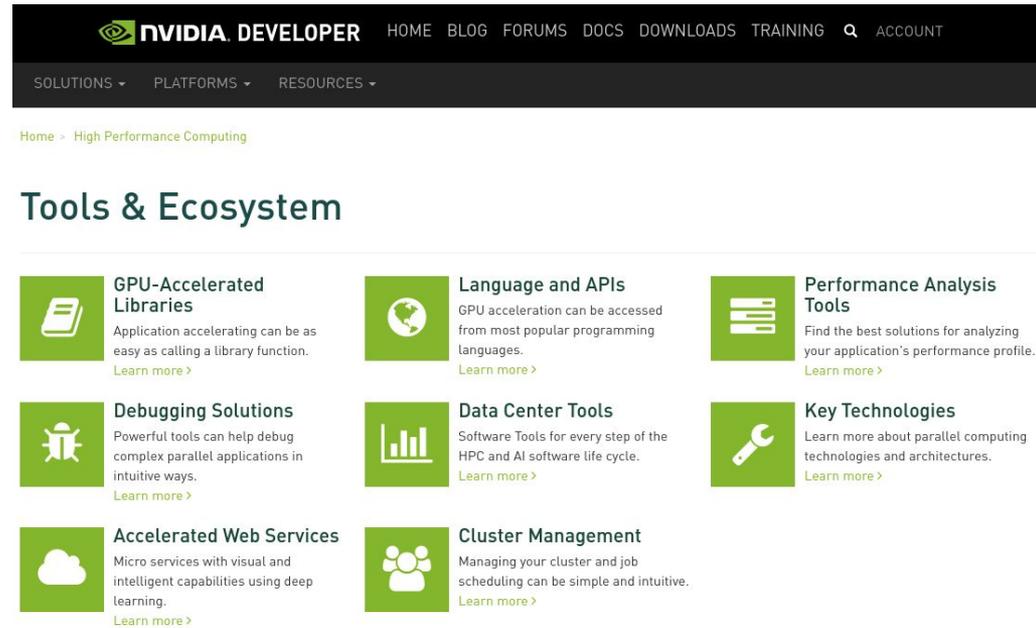
- with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.

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

```
$nvcc myobj.o -l cublas
```

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone.



The screenshot shows the NVIDIA Developer Zone website. The top navigation bar includes the NVIDIA Developer logo and links for HOME, BLOG, FORUMS, DOCS, DOWNLOADS, TRAINING, a search icon, and ACCOUNT. Below the navigation bar, there are dropdown menus for SOLUTIONS, PLATFORMS, and RESOURCES. The main content area is titled 'Tools & Ecosystem' and features a grid of nine cards, each with an icon, a title, a brief description, and a 'Learn more' link.

Home > High Performance Computing

Tools & Ecosystem

- GPU-Accelerated Libraries**
Application accelerating can be as easy as calling a library function.
[Learn more >](#)
- Language and APIs**
GPU acceleration can be accessed from most popular programming languages.
[Learn more >](#)
- Performance Analysis Tools**
Find the best solutions for analyzing your application's performance profile.
[Learn more >](#)
- Debugging Solutions**
Powerful tools can help debug complex parallel applications in intuitive ways.
[Learn more >](#)
- Data Center Tools**
Software Tools for every step of the HPC and AI software life cycle.
[Learn more >](#)
- Key Technologies**
Learn more about parallel computing technologies and architectures.
[Learn more >](#)
- Accelerated Web Services**
Micro services with visual and intelligent capabilities using deep learning.
[Learn more >](#)
- Cluster Management**
Managing your cluster and job scheduling can be simple and intuitive.
[Learn more >](#)

[NVIDIA CUDA Tools & Ecosystem](#)

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

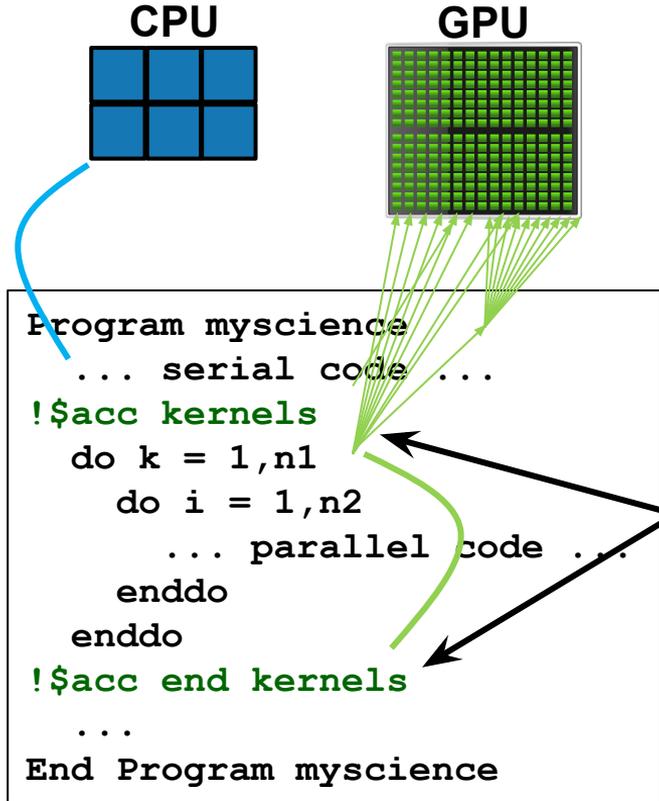
OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

OpenACC Directives



Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC



The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

Directives: Easy & Powerful

Real-Time Object
Detection

Global Manufacturer of
Navigation Systems



5x in 40 Hours

Valuation of Stock Portfolios
using Monte Carlo

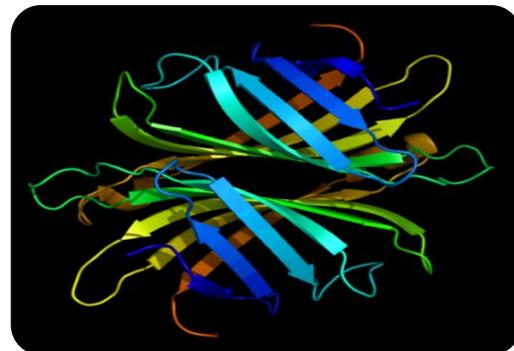
Global Technology Consulting
Company



2x in 4 Hours

Interaction of Solvents and
Biomolecules

University of Texas at San Antonio



5x in 8 Hours

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

GPU Programming Languages

Numerical analytics ▶

MATLAB, Mathematica, LabVIEW

Fortran ▶

OpenACC, CUDA Fortran

C ▶

OpenACC, CUDA C, OpenCL

C++ ▶

Thrust, CUDA C++, OpenCL

Python ▶

PyCUDA, PyOpenCL, CuPy

Julia / Java ▶

JuliaGPU/CUDA.jl, jcuda

Rapid Parallel C++ Development



- Resembles C++ STL
- High-level interface
 - Enhances developer productivity
 - Enables performance portability between GPUs and multicore CPUs
- Flexible
 - CUDA, OpenMP, and TBB backends
 - Extensible and customizable
 - Integrates with existing software
- Open source

```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                h_vec.end(),
                rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
            d_vec.end(),
            h_vec.begin());
```

<https://thrust.github.io/>

Learn More

These languages are supported on all CUDA-capable GPUs.

You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++

<http://developer.nvidia.com/cuda-toolkit>

PyCUDA (Python)

<https://developer.nvidia.com/pycuda>

Thrust C++ Template Library

<http://developer.nvidia.com/thrust>

MATLAB

<http://www.mathworks.com/discovery/matlab-gpu.html>

CUDA Fortran

<https://developer.nvidia.com/cuda-fortran>

Mathematica

<http://www.wolfram.com/mathematica/new-in-8/cuda-and-openssl-support/>

Part II. Running CUDA Code on Terra



Login to Terra3 - ssh terra3

```
jtao@tlogin-0301:~$ ssh terra3
portal-terra.hprc.tamu.edu/pun/sys/shell/ssh/terra.tamu.edu
[jtao@terra3 ~]$ nvidia-smi
Mon Apr 19 18:50:03 2021

+-----+
| NVIDIA-SMI 460.32.03      Driver Version: 460.32.03      CUDA Version: 11.2     |
+-----+-----+-----+
| 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          0n          | 00000000:83:00.0 Off  |      0%      Default |
| N/A   27C    P8         26W / 149W | 0MiB / 12206MiB |              MIG M. |
+-----+-----+-----+
| 1   Tesla K80          0n          | 00000000:84:00.0 Off  |      0%      Default |
| N/A   36C    P8         30W / 160W | 0MiB / 12206MiB |              MIG M. |
+-----+-----+-----+

+-----+
| Processes: |
| GPU   GI    CI          PID   Type   Process name                      GPU Memory |
|   ID   ID   ID             |              | Usage     |
+-----+-----+-----+
|          |          |          |          |          |          |          |
+-----+-----+-----+
| No running processes found |
+-----+

[jtao@terra3 ~]$
```

Running CUDA Code on Terra

```
# load CUDA module
$m1 CUDA/9.1.85-GCC-6.4.0-2.28

# copy sample code to your scratch space
$cd $SCRATCH
$wget https://hprc.tamu.edu/files/training/2021/Spring/cuda.exercise.tgz
$tar -zxvf cuda.exercise.tgz

# compile CUDA code
$cd CUDA
$nvcc hello_world_host.cu
$nprof ./a.out

# edit job script & submit your first GPU job
$bsub < terra_cuda_run.sh
```

Part III. CUDA C/C++ BASICS



nVIDIA®

CUDA

What is CUDA?

- CUDA Architecture
 - Used to mean “Compute Unified Device Architecture”
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.

A Brief History of CUDA

- Researchers used OpenGL APIs for general purpose computing on GPUs before CUDA.
- In 2007, NVIDIA released first generation of Tesla GPU for general computing together their proprietary CUDA development framework.
- Current stable version of CUDA is 10.2 (as of Mar. 2020).

Heterogeneous Computing

- Terminology:
 - *Host* The CPU and its memory (host memory)
 - *Device* The GPU and its memory (device memory)



Host



Device

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE * 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

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

    // Synchronize (ensure all the data is available)
    __syncthreads();

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

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

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(d_in + RADIUS, d_out +
RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel function

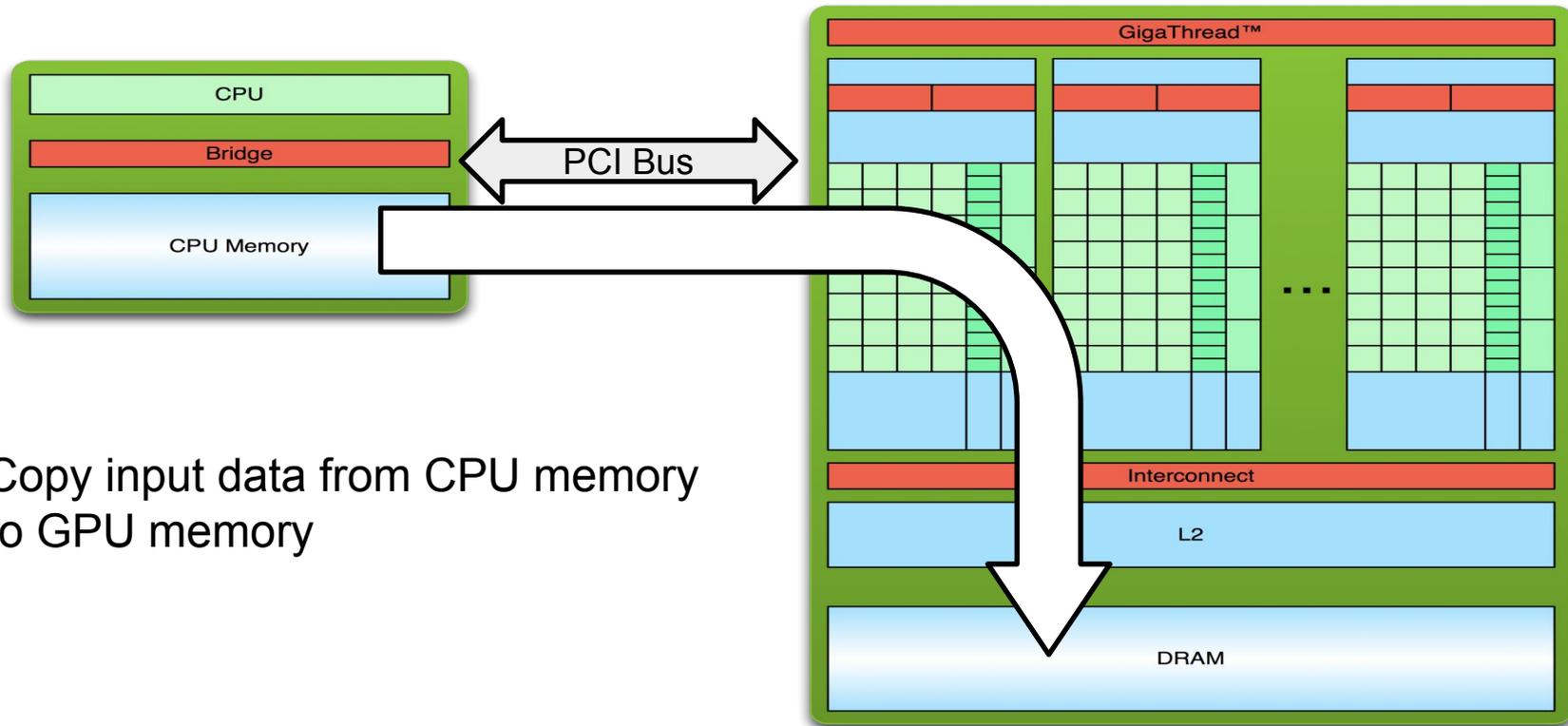
serial code

parallel code

serial code

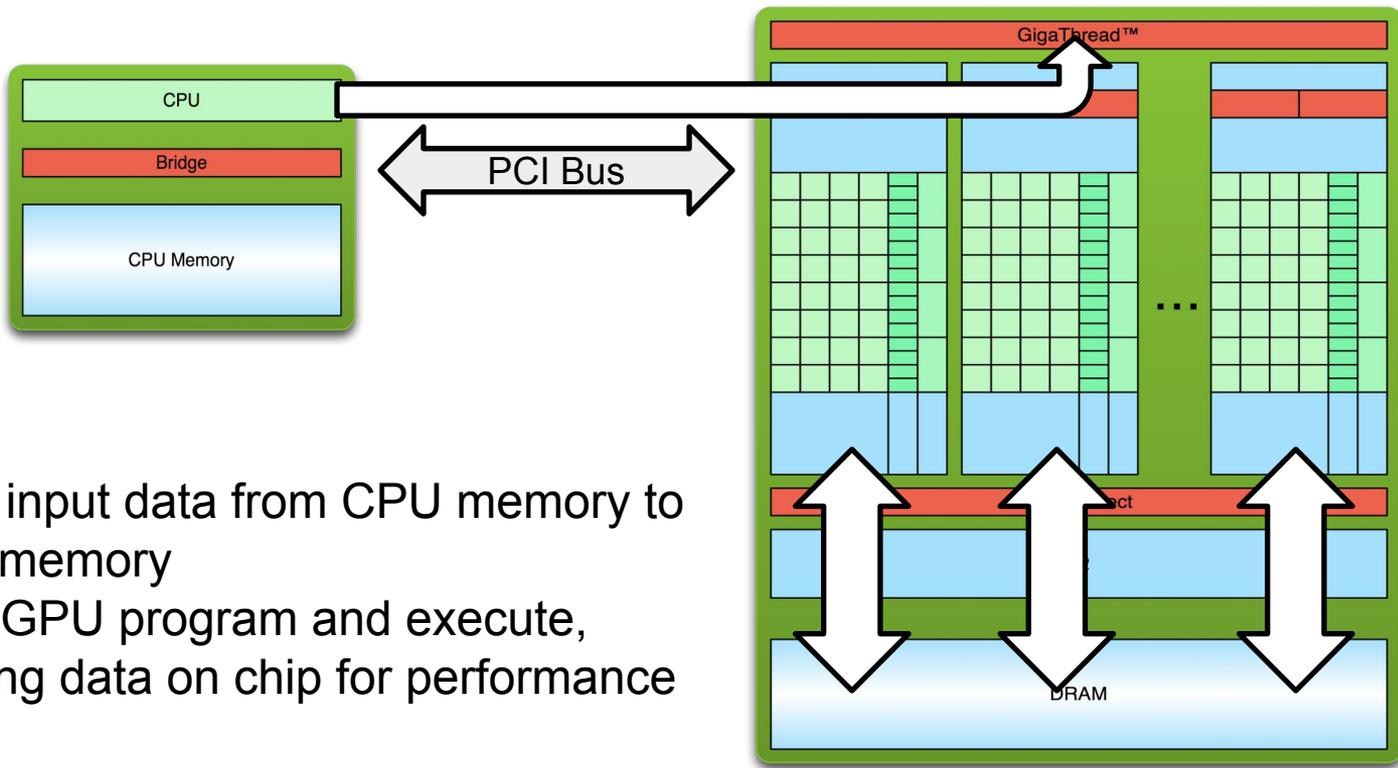


Simple Processing Flow



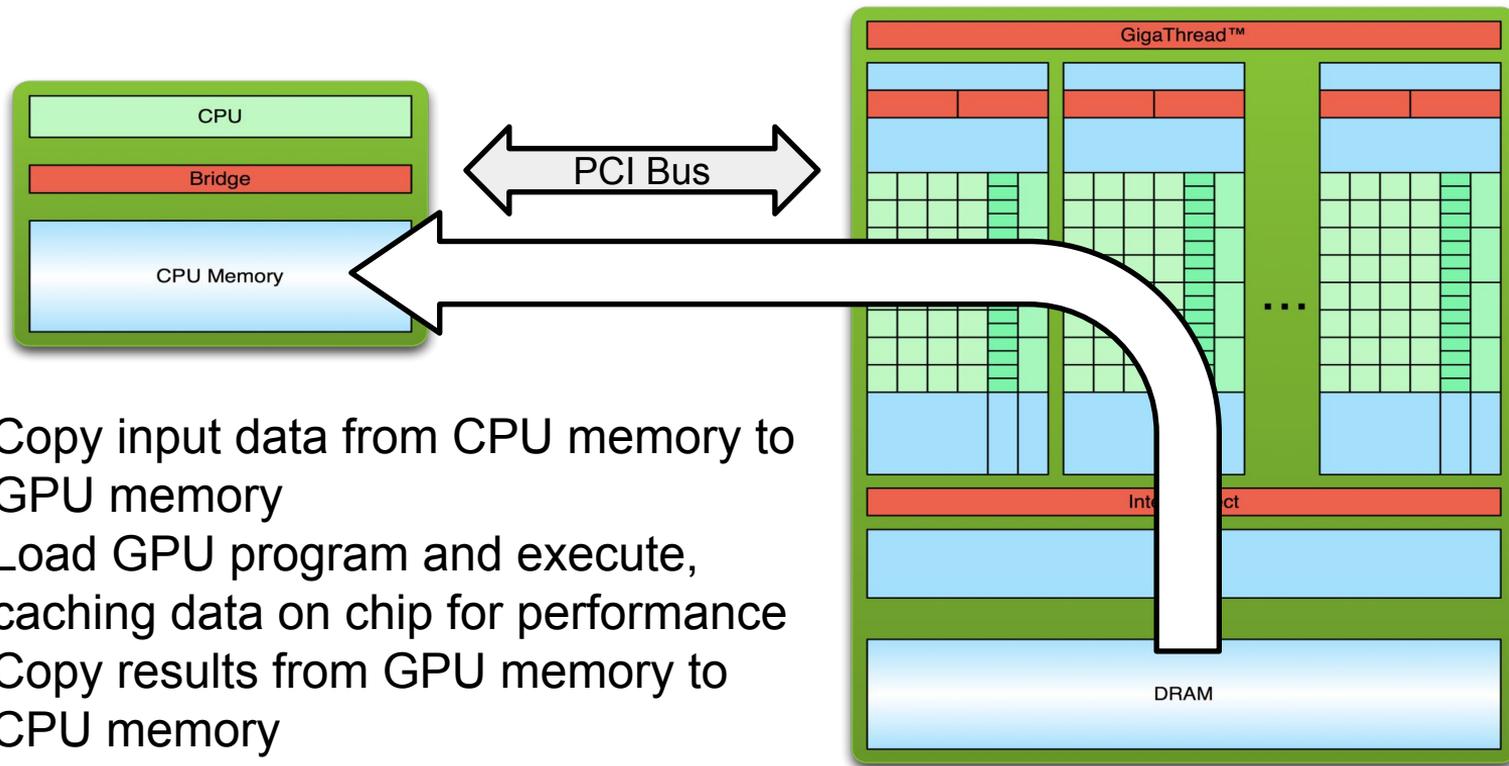
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing Flow

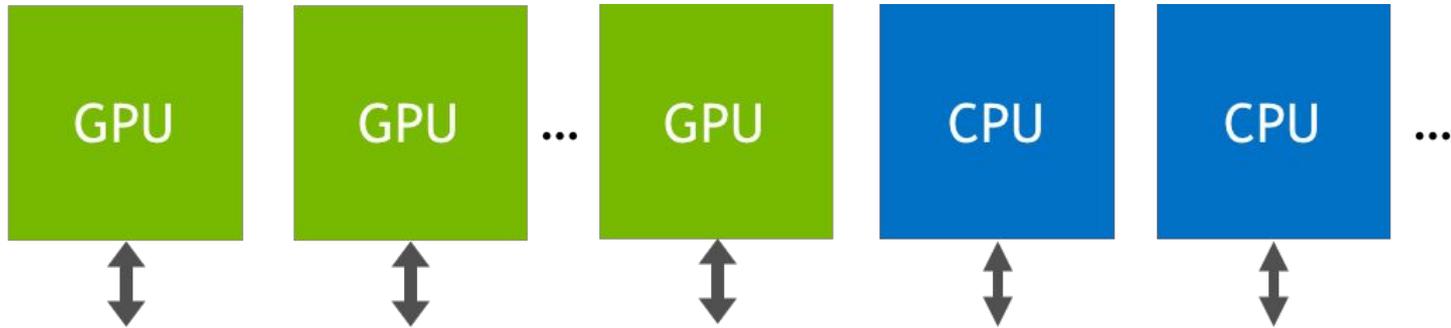


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Unified Memory

Unified Memory

- A managed memory space where all processors see a single coherent memory image with a common address space.
- Memory allocation with `cudaMallocManaged()`.
- Synchronization with `cudaDeviceSynchronize()`.
- Eliminates the need for `cudaMemcpy()`.
- Enables simpler code.
- Hardware support since Pascal GPU.

Hello World!

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

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu  
$ ./a.out  
$ Hello World!
```

Hello World! with Device Code

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

- Two new syntactic elements...

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- `nvcc` separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc`, `icc`, etc.

Hello World! with Device Code

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

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1, 1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

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

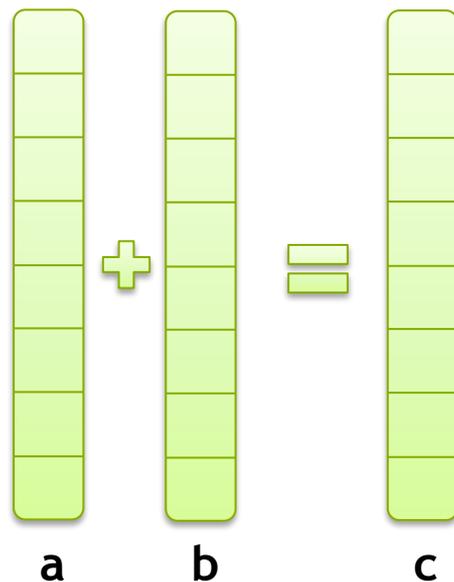
Output:

```
$nvcc hello.cu  
$./a.out  
Hello World!
```

- `mykernel()` does nothing!

Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host

Addition on the Device

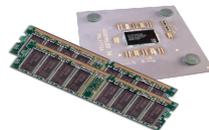
- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b`, and `c` must point to device memory
- We need to allocate memory on the GPU.

Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



Addition on the Device: `add()`

- Returning to our `add()` kernel

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

- Let's take a look at `main()`...

Addition on the Device: `main()`

```
int main(void) {  
    int a, b, c;           // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

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

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

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



```
add<<< N, 1 >>> ();
```

- Instead of executing `add ()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different element of the array.

Vector Addition on the Device

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

- On the device, each block can execute in parallel:

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];`

Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

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

- Let's take a look at `main()`...

Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and set up input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Vector Addition with Unified Memory

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

int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    VecAdd<<< 1000, 1 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

Vector Addition with Managed Global Memory

```
__device__ __managed__ int ret[1000];

__global__ void VecAdd(int *ret, int a, int b) {
    ret[blockIdx.x] = a + b + blockIdx.x;
}

int main() {
    VecAdd<<< 1000, 1 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Profiling with nvprof

```
$nvprof add_parallel
```

```
==28491== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
43.45%	4.3520us	1	4.3520us	4.3520us	4.3520us	add(int*, int*, int*)
30.35%	3.0400us	2	1.5200us	1.3120us	1.7280us	[CUDA memcpy HtoD]
26.20%	2.6240us	1	2.6240us	2.6240us	2.6240us	[CUDA memcpy DtoH]

```
==28491== API calls:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
99.34%	231.73ms	3	77.242ms	6.1990us	231.71ms	cudaMalloc
0.33%	766.63us	182	4.2120us	171ns	143.74us	cuDeviceGetAttribute
0.15%	357.72us	2	178.86us	173.06us	184.67us	cuDeviceTotalMem
0.08%	175.05us	3	58.351us	6.6470us	147.94us	cudaFree
0.03%	75.722us	1	75.722us	75.722us	75.722us	cudaLaunch
0.03%	74.091us	3	24.697us	10.865us	35.014us	cudaMemcpy
0.03%	65.073us	2	32.536us	30.391us	34.682us	cuDeviceGetName
0.00%	4.6390us	3	1.5460us	221ns	3.9590us	cudaSetupArgument
0.00%	4.4490us	3	1.4830us	434ns	3.3590us	cuDeviceGetCount
0.00%	2.7070us	6	451ns	196ns	777ns	cuDeviceGet
0.00%	1.9940us	1	1.9940us	1.9940us	1.9940us	cudaConfigureCall

Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function

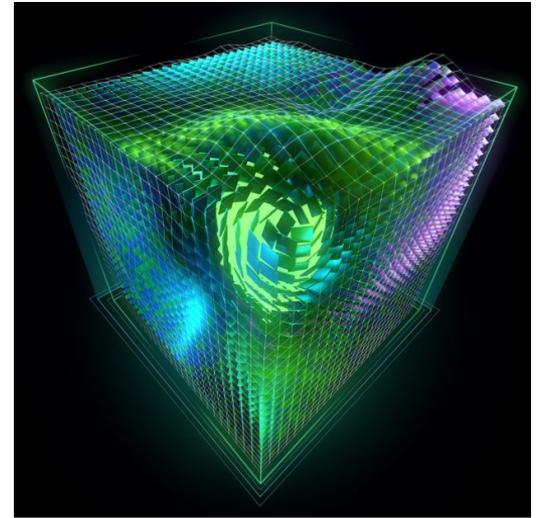
Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch **N** copies of `add()` with `add<<<N,1>>>(...)`.
 - Use `blockIdx.x` to access block index.
 - Use `nvprof` for collecting & viewing profiling data.

More Resources

- You can learn more about the details at
 - CUDA Programming Guide (docs.nvidia.com/cuda)
 - CUDA Zone – tools, training, etc. (developer.nvidia.com/cuda-zone)
 - Download CUDA Toolkit & SDK (www.nvidia.com/getcuda)
 - Nsight IDE (Eclipse or Visual Studio) (www.nvidia.com/nsight)
- Intermediate CUDA Programming Short Course
 - GPU memory management and unified memory
 - Parallel kernels in CUDA C
 - Parallel communication and synchronization
 - Running a CUDA code on Ada
 - Profiling and performance evaluation

Part IV. CUDA Programming Abstractions



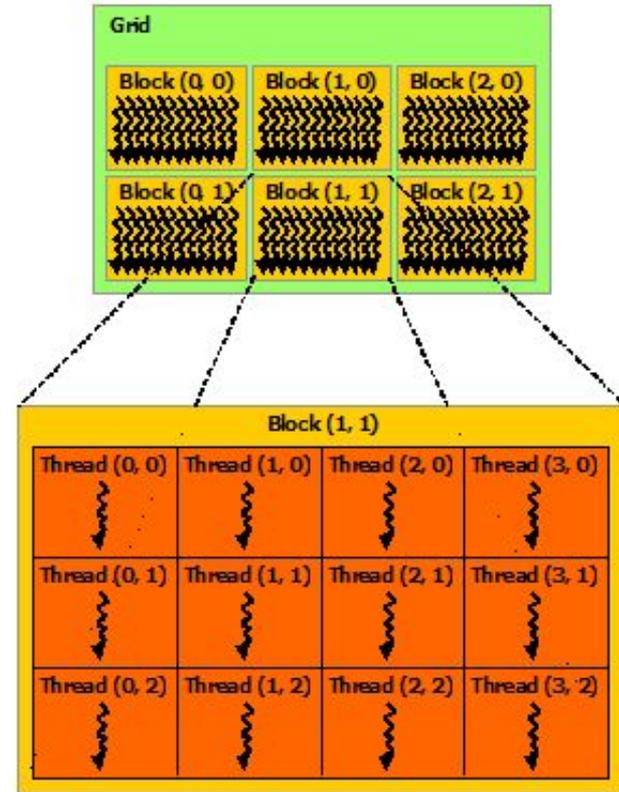
Key Programming Abstractions

Three key abstractions that are exposed to CUDA programmers as a minimal set of language extensions:

- **a hierarchy of thread groups**
- **shared memories**
- **barrier synchronization**

Glossary

- **Thread** is an abstract entity that represents the execution of the kernel, which is a small program or a function.
- **Grid** is a collection of Threads. Threads in a Grid execute a Kernel Function and are divided into Thread Blocks.
- **Thread Block** is a group of threads which execute on the same multiprocessor (SMX). Threads within a Thread Block have access to shared memory and can be explicitly synchronized.



CUDA Kernels

- CUDA kernels are C functions that, when called, are executed N times in parallel by N different CUDA threads.
- A kernel is defined with `__global__` declaration specifier.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Kernel Invocation

- The number of CUDA threads that execute a kernel is specified using a new `<<<. . .>>>` execution configuration syntax.
- Each thread that executes the kernel is given a unique **thread ID** that is accessible within the kernel through the built-in 3-component vector **threadIdx**.

```
// Kernel Invocation with N threads  
VecAdd<<<1, N>>>(A, B, C);
```

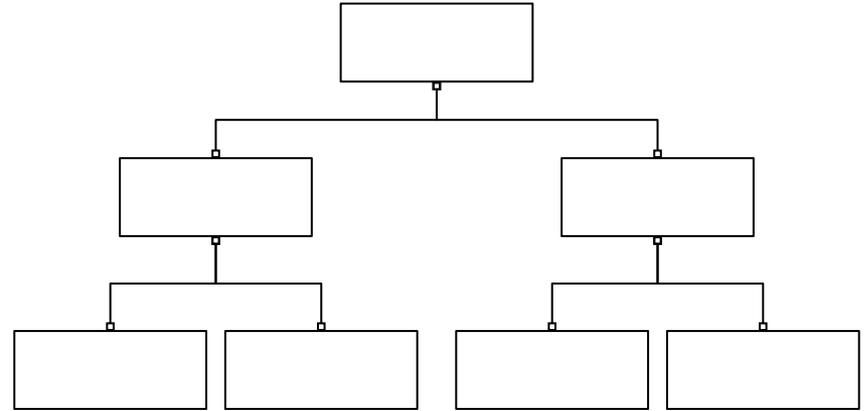
Example 1 - Kernel Definition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

Example 1 - Kernel Invocation

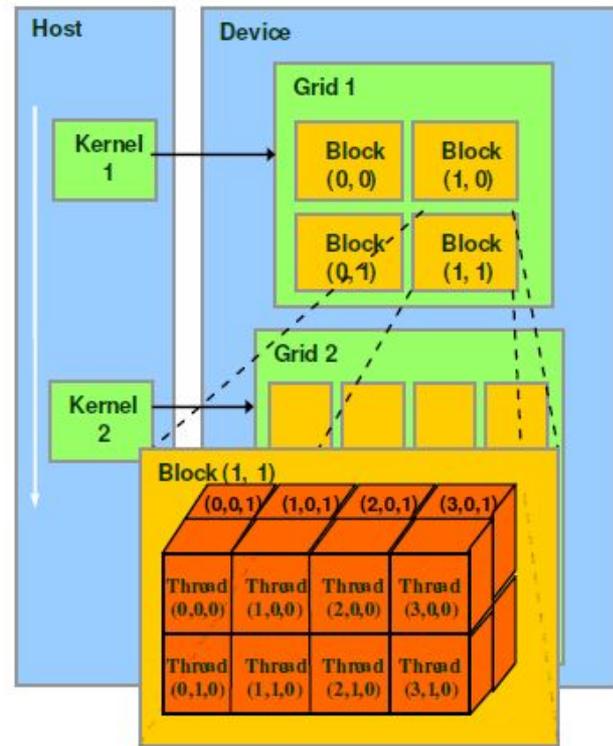
```
// Kernel invocation
int main()
{
    ...
    // Call kernel with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Hierarchy of Threads



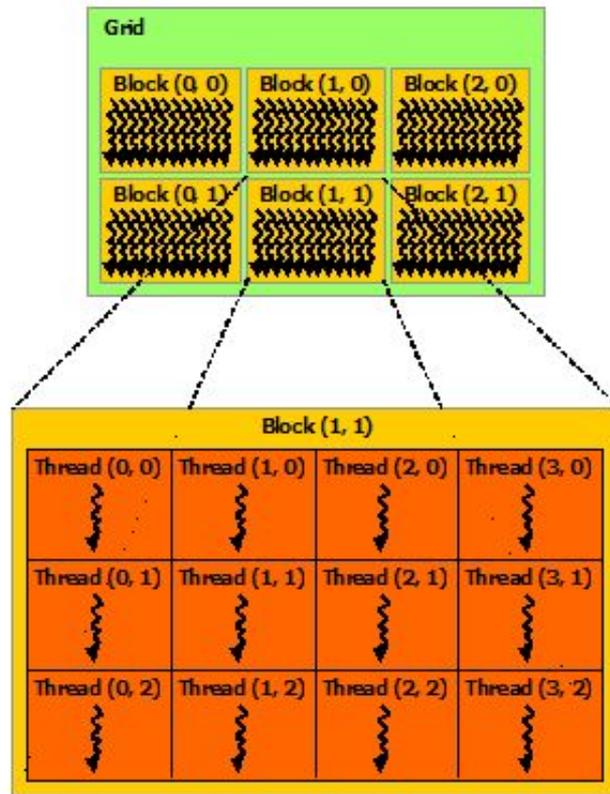
Thread Hierarchy - I

- 1D, 2D, or 3D threads can form 1D, 2D, or 3D **thread blocks**.
- 1D, 2D, or 3D blocks can form 1D, 2D, or 3D **grid of thread blocks**
- The number of threads per block and the number of blocks per grid are specified in the `<<< . . . >>>` syntax.



Thread Hierarchy - II

- Each block within the grid can be identified by an index accessible within the kernel through the built-in 3-component vector **blockIdx**.
- The dimension of the thread block is accessible within the kernel through the built-in 3-component vector **blockDim**.



Thread Index and Thread ID

- **1D**

thread ID is the same as the index of a thread

- **2D**

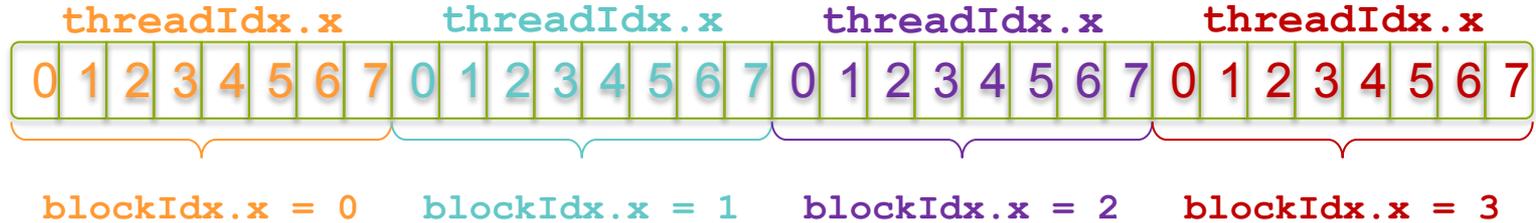
for a two-dimensional block of size `(blockDim.x, blockDim.y)`, the thread ID of a thread of index `(x, y)` is `(x + y * blockDim.x)`

- **3D**

for a three-dimensional block of size `(blockDim.x, blockDim.y, blockDim.z)`, the thread ID of a thread of index `(x, y, z)` is `(x + y * blockDim.x + z * blockDim.x * blockDim.y)`

Indexing Arrays with Blocks and Threads

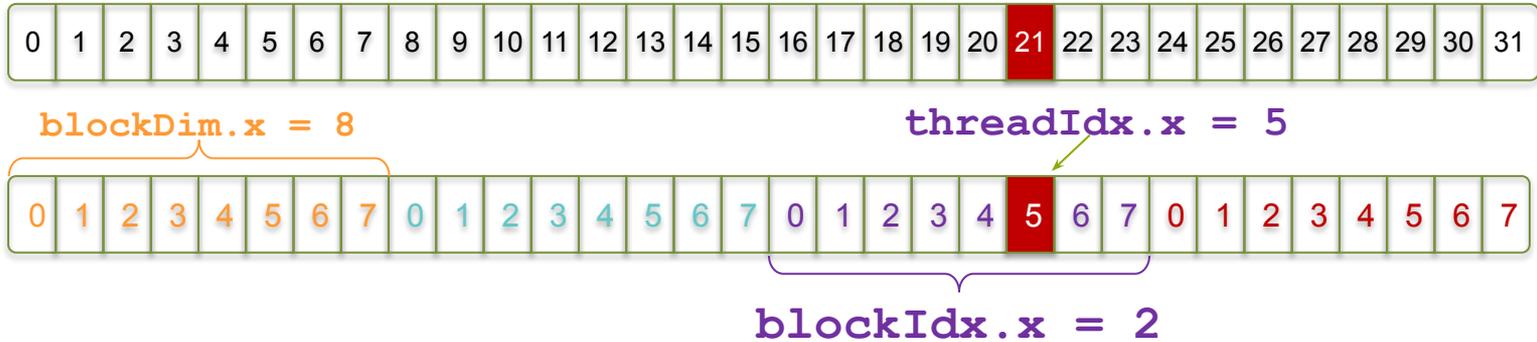
- Consider indexing an array with one element per thread (8 threads/block)



- With `blockDim.x` threads/block, the thread is given by:
`int index = threadIdx.x + blockIdx.x * blockDim.x;`

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * blockDim.x;  
          =           5           +           2           *           8  
          = 21
```

Example 2 - Kernel Definition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

Example 2 - Kernel Invocation

```
// Kernel invocation
int main()
{
    ...
    // run kernel with multiple blocks of 16*16*1 threads
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N /
threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void VecAdd(int *A, int *B, int *C, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        C[index] = A[index] + B[index];  
}
```

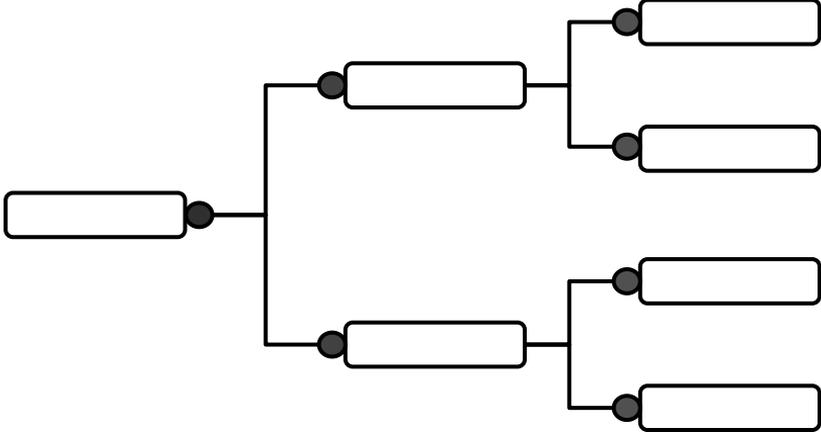
Update the kernel launch: $M = \text{blockDim.x}$

```
VecAdd<<< (N + M - 1) / M, M >>>(A, B, C, N);
```

Why Bother with Threads?

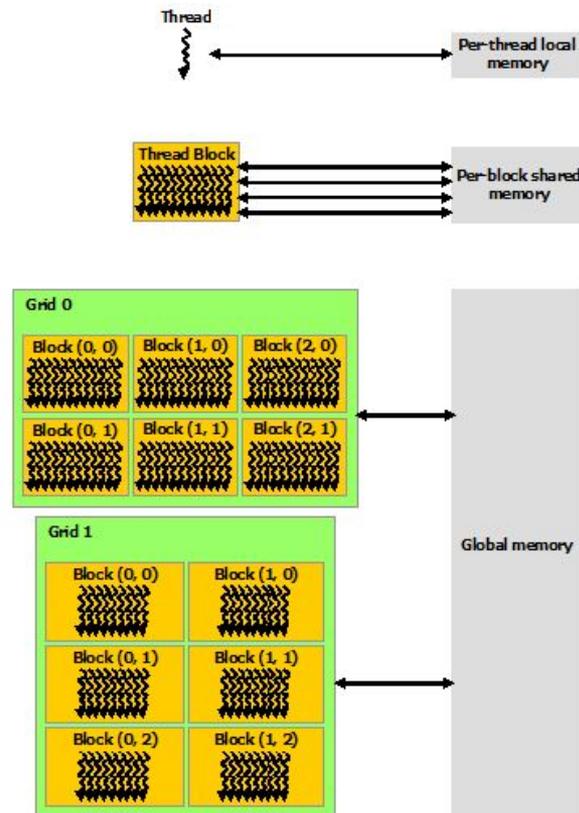
- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Threads within a block can cooperate by sharing data through some shared memory
- by synchronizing their execution to coordinate memory accesses with `__syncthreads ()`

Memory Hierarchy



Hierarchical Memory Structure

- Each thread has access to **registers** and **private local memory**.
- Each thread block has **shared memory** visible to all threads of the block and with the same lifetime as the block.
- All threads have access to **global memory**



Memory Spaces

- **Register, local, shared, global, constant (read only), and texture (read only)** memory are the memory spaces available.
- Only register and shared memory reside on GPU.
- The **global, constant, and texture memory spaces** are cached and persistent across kernel launches by the same application.

Memory: Scope and Performance

- Data in **register memory** is visible only to the thread and lasts only for the lifetime of that thread.
- **Local memory** has the same scope rules as register memory, but performs slower.
- Data stored in **shared memory** is visible to all threads within that block and lasts for the duration of the block.
- Data stored in **global memory** is visible to all threads within the application (including the host), and lasts for the duration of the host allocation.
- **Constant memory** is used for data that will not change over the course of a kernel execution and is read only.
- **Texture memory** is another variety of read-only memory on the device.

Using Global Memory

- Linear memory is typically allocated using **cudaMalloc()** and freed using **cudaFree()** and data transfer between host and device is done using **cudaMemcpy()**.
- Linear memory can also be allocated through **cudaMallocPitch()** and **cudaMalloc3D()** and transferred using **cudaMemcpy2D()** and **cudaMemcpy3D()** with better memory alignment.

Using Shared Memory

- Much faster than global memory.
- Allocated using the `__shared__` memory space specifier.

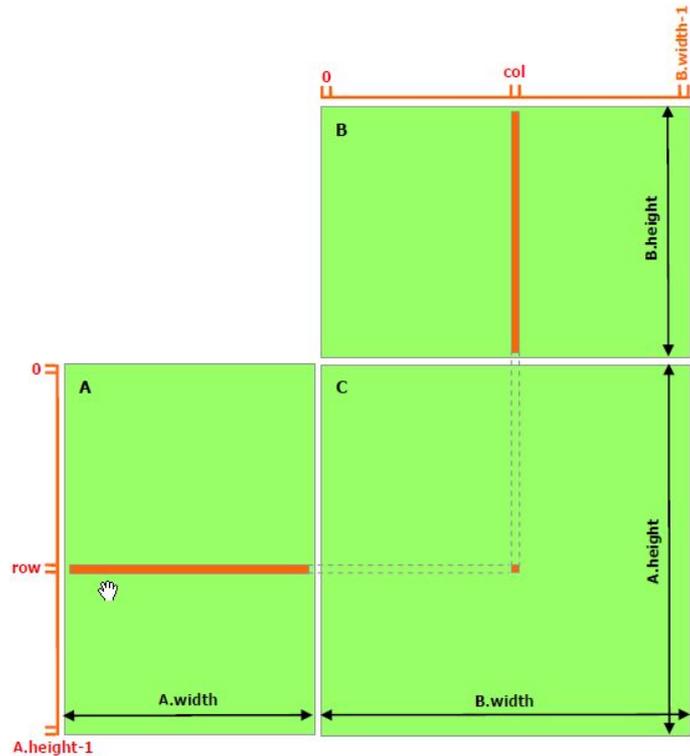
```
__shared__ float A[BLOCK_SIZE][BLOCK_SIZE];
```

- Shared memory shall be used as a cache for global memory to exploit locality of the code.

Example 3 - Matrix Multiplication w/o SM

Each thread computes one element of C by accumulating results into Cvalue.

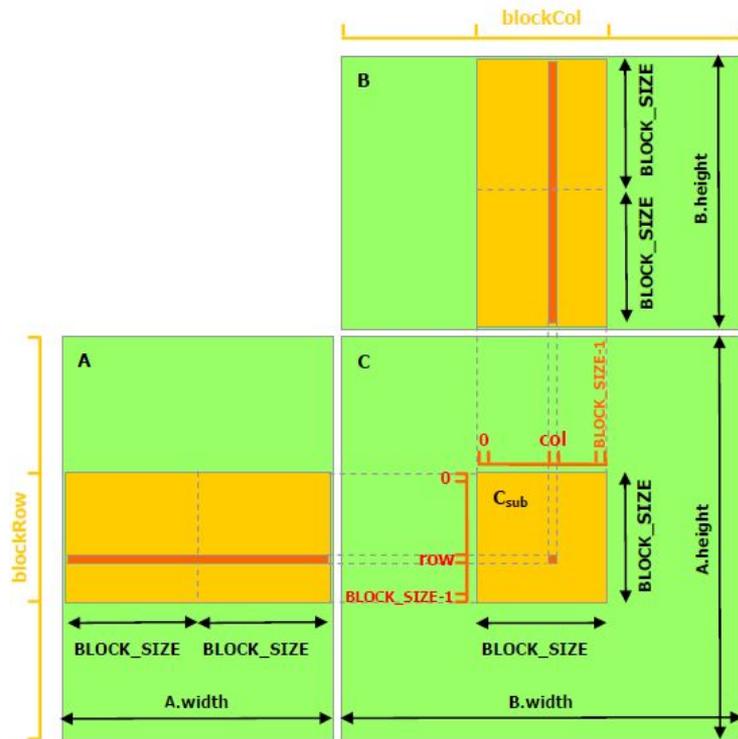
```
__global__ void MatMulKernel(Matrix A, Matrix B,  
                             Matrix C)  
{  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e] *  
                 B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Example 4 - Matrix Multiplication with SM

Each thread computes one element of C_{sub} // by accumulating results into C_{value}

```
...  
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {  
    Matrix Asub = GetSubMatrix(A, blockRow, m);  
    Matrix Bsub = GetSubMatrix(B, m, blockCol);  
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
    As[row][col] = GetElement(Asub, row, col);  
    Bs[row][col] = GetElement(Bsub, row, col);  
    __syncthreads();  
    for (int e = 0; e < BLOCK_SIZE; ++e)  
        Cvalue += As[row][e] * Bs[e][col];  
    __syncthreads();  
}
```



Review - 1

- Launching parallel kernels
 - Launch N copies of `add()` with `add<<<N/M,M>>>(...)` ;
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

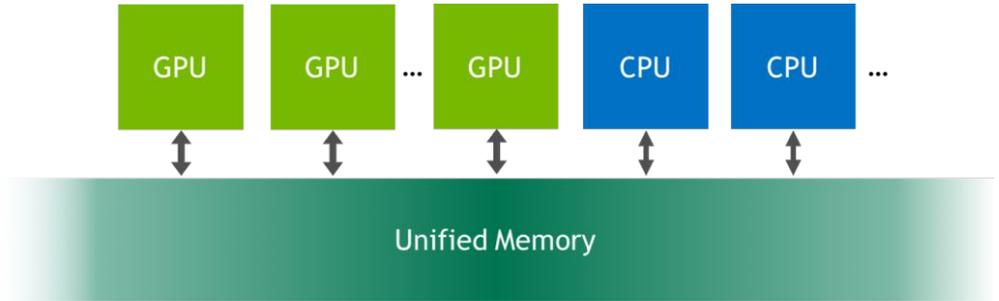
Review - 2

- Launching parallel threads
 - Launch N blocks with `blockDim.x` threads per block with
`kernel<<<N, blockDim.x>>>(...);`
 - Use `blockIdx.x` to access block index within grid
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:
`int index = threadIdx.x + blockIdx.x * blockDim.x;`

Review - 3

- 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
 - Use to prevent data hazards

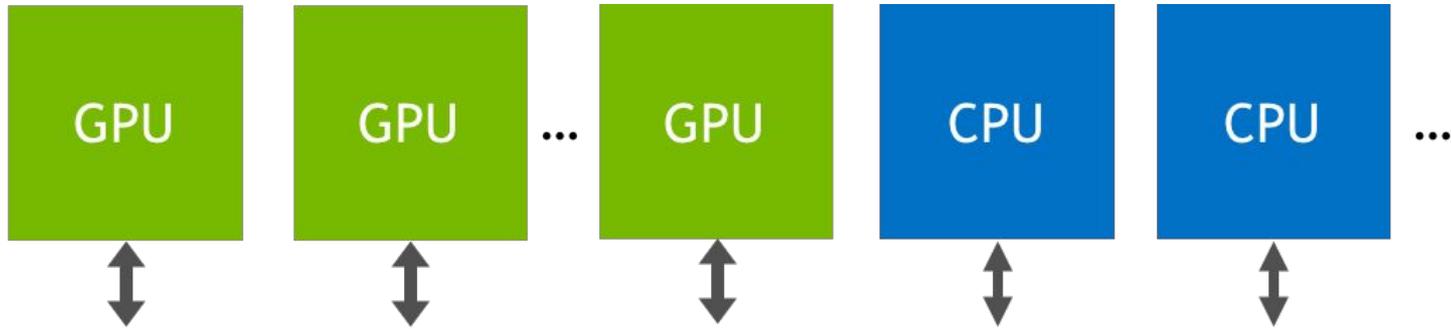
Unified Memory Programming



Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Unified Memory

Unified Memory

- A managed memory space where all processors see a single coherent memory image with a common address space.
- Eliminates the need for `cudaMemcpy ()`.
- Enables simpler code.
- Equipped with hardware support since Pascal.

Example 5 - Vector Addition w/o UM

```
__global__ void VecAdd( int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

Example 6 - Vector Addition with UM

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

int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

Example 7 - Vector Addition with Managed Global Memory

```
__device__ __managed__ int ret[1000];

__global__ void VecAdd(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Managing Devices



Coordinating Host & Device

- Kernel launches are asynchronous
 - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the 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 the CPU until all preceding CUDA calls have completed

Reporting Errors

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

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()))
);
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
```

```
cudaSetDevice(int device)
```

```
cudaGetDevice(int *device)
```

```
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

Select current device: `cudaSetDevice(i)`

For peer-to-peer copies: `cudaMemcpy(...)`

GPU Computing Capability

The compute capability of a device is represented by a version number that identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

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						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)			GeForce 2000 Series	Quadro RTX Series	Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Quadro GV Series	Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2		GeForce 1000 Series	Quadro P Series	Tesla P Series	
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		

More Resources

You can learn more about CUDA at

- CUDA Programming Guide (docs.nvidia.com/cuda)
- CUDA Zone – tools, training, etc.
(developer.nvidia.com/cuda-zone)
- Download CUDA Toolkit & SDK
(www.nvidia.com/getcuda)
- Nsight IDE (Eclipse or Visual Studio)
(www.nvidia.com/nsight)

Acknowledgements

- Educational materials from NVIDIA Deep Learning Institute via its University Ambassador Program.
- Supports from the Texas A&M Engineering Experiment Station (TEES), the Texas A&M Institute of Data Science (TAMIDS), and Texas A&M High Performance Research Computing (HPRC).

Appendix

1D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_1D_1D ()  
{  
    return blockIdx.x * blockDim.x + threadIdx.x;  
}
```

```
__device__ int getGlobalIdx_1D_2D ()  
{  
    return blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x +  
        threadIdx.x;  
}
```

```
__device__ int getGlobalIdx_1D_3D ()  
{  
    return blockIdx.x * blockDim.x * blockDim.y * blockDim.z  
        + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x +  
        threadIdx.x;  
}
```

2D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_2D_1D ()
```

```
{  
    int blockId = blockIdx.y * gridDim.x + blockIdx.x;  
    int threadId = blockId * blockDim.x + threadIdx.x;  
    return threadId;  
}
```

```
__device__ int getGlobalIdx_2D_2D ()
```

```
{  
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;  
    int threadId =  
        blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;  
    return threadId;  
}
```

```
__device__ int getGlobalIdx_2D_3D ()
```

```
{  
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;  
    int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)  
        + (threadIdx.z * (blockDim.x * blockDim.y))  
        + (threadIdx.y * blockDim.x) + threadIdx.x;  
    return threadId;  
}
```

3D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_3D_1D ()
{
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
    int threadId = blockId * blockDim.x + threadIdx.x;
    return threadId;
}
```

```
__device__ int getGlobalIdx_3D_2D ()
{
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
    int threadId = blockId * (blockDim.x * blockDim.y)
        + (threadIdx.y * blockDim.x) + threadIdx.x;
    return threadId;
}
```

```
__device__ int getGlobalIdx_3D_3D ()
{
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
    int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
        + (threadIdx.z * (blockDim.x * blockDim.y))
        + (threadIdx.y * blockDim.x) + threadIdx.x;
    return threadId;
}
```

Tesla V100 GPU Node

Device 0: "Tesla V100-PCIE-32GB"

CUDA Driver Version / Runtime Version	10.1 / 9.0
CUDA Capability Major/Minor version number:	7.0
Total amount of global memory:	32480 MBytes (34058272768 bytes)
(80) Multiprocessors, (64) CUDA Cores/MP:	5120 CUDA Cores
GPU Max Clock rate:	1380 MHz (1.38 GHz)
Memory Clock rate:	877 Mhz
Memory Bus Width:	4096-bit
L2 Cache Size:	6291456 bytes
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)
Concurrent copy and kernel execution:	Yes with 7 copy engine(s)
Run time limit on kernels:	No
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Supports Cooperative Kernel Launch:	Yes

Tesla A100 GPU Node

Device 0: "A100-PCIE-40GB"

CUDA Driver Version / Runtime Version	11.2 / 11.0
CUDA Capability Major/Minor version number:	8.0
Total amount of global memory:	40536 MBytes (42505273344 bytes)
(108) Multiprocessors, (64) CUDA Cores/MP:	6912 CUDA Cores
GPU Max Clock rate:	1410 MHz (1.41 GHz)
Memory Clock rate:	1215 Mhz
Memory Bus Width:	5120-bit
L2 Cache Size:	41943040 bytes
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)
Concurrent copy and kernel execution:	Yes with 3 copy engine(s)
Run time limit on kernels:	No
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Supports Cooperative Kernel Launch:	Yes