Introduction to CUDA® Programming

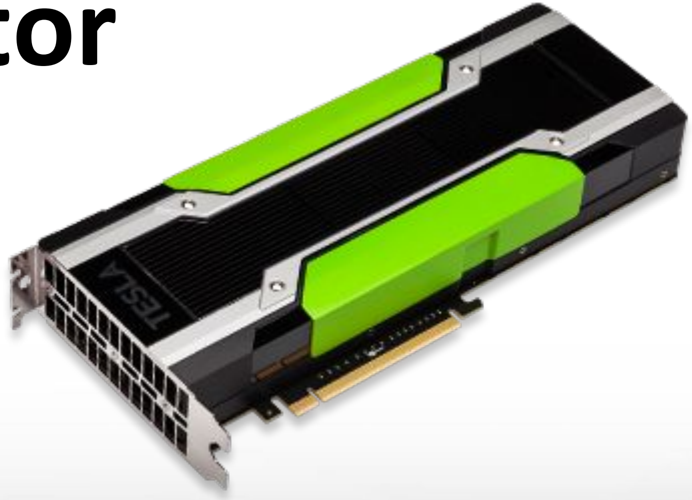
Jian Tao

jtao@tamu.edu

Spring 2018 HPRC Short Course

03/09/2018

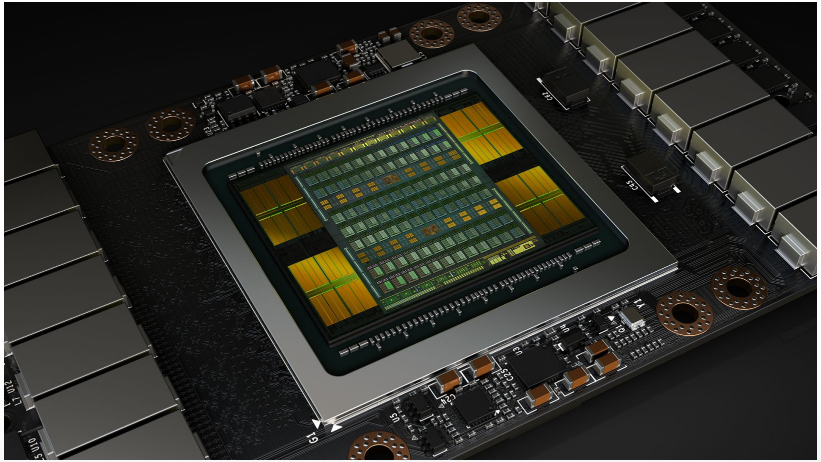
GPU as an Accelerator



CPU

GPU Accelerator

NVIDIA Tesla V100 with 21.1 Billion Transistors

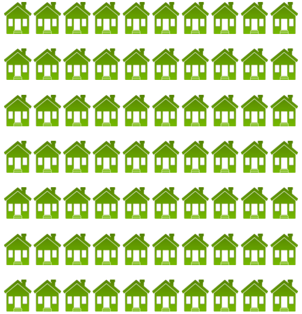


Why Computing Perf/Watt Matters?

2.3 PFlops

7.0
Megawatts

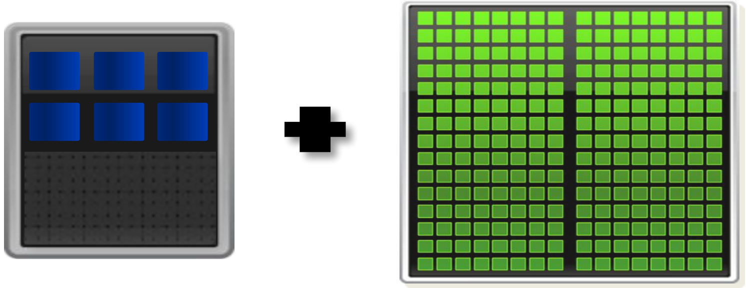
7000 homes



7.0
Megawatts

CPU
Optimized for Serial Tasks

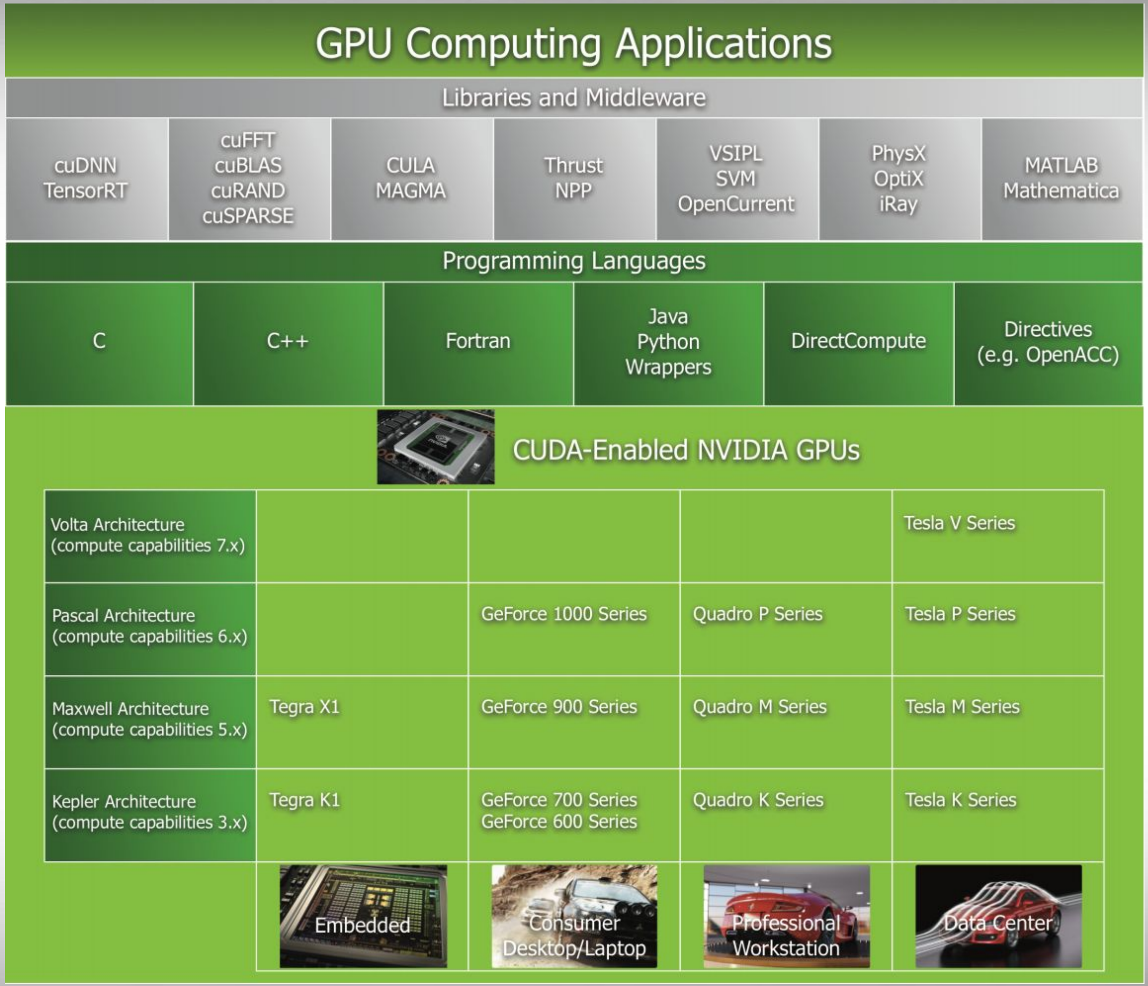
GPU Accelerator
Optimized for Many Parallel Tasks



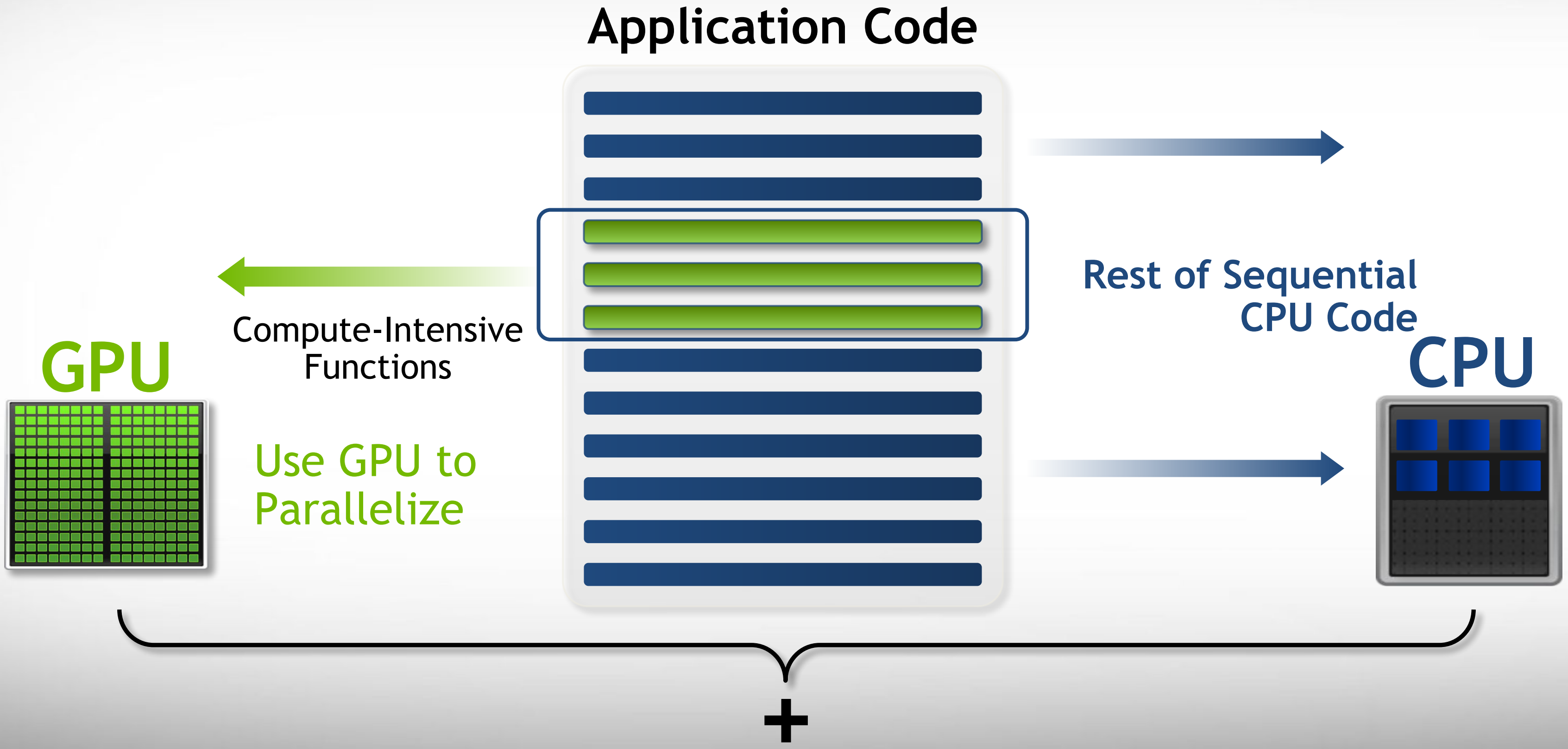
Traditional CPUs are not economically feasible

GPU-accelerated computing started a new era

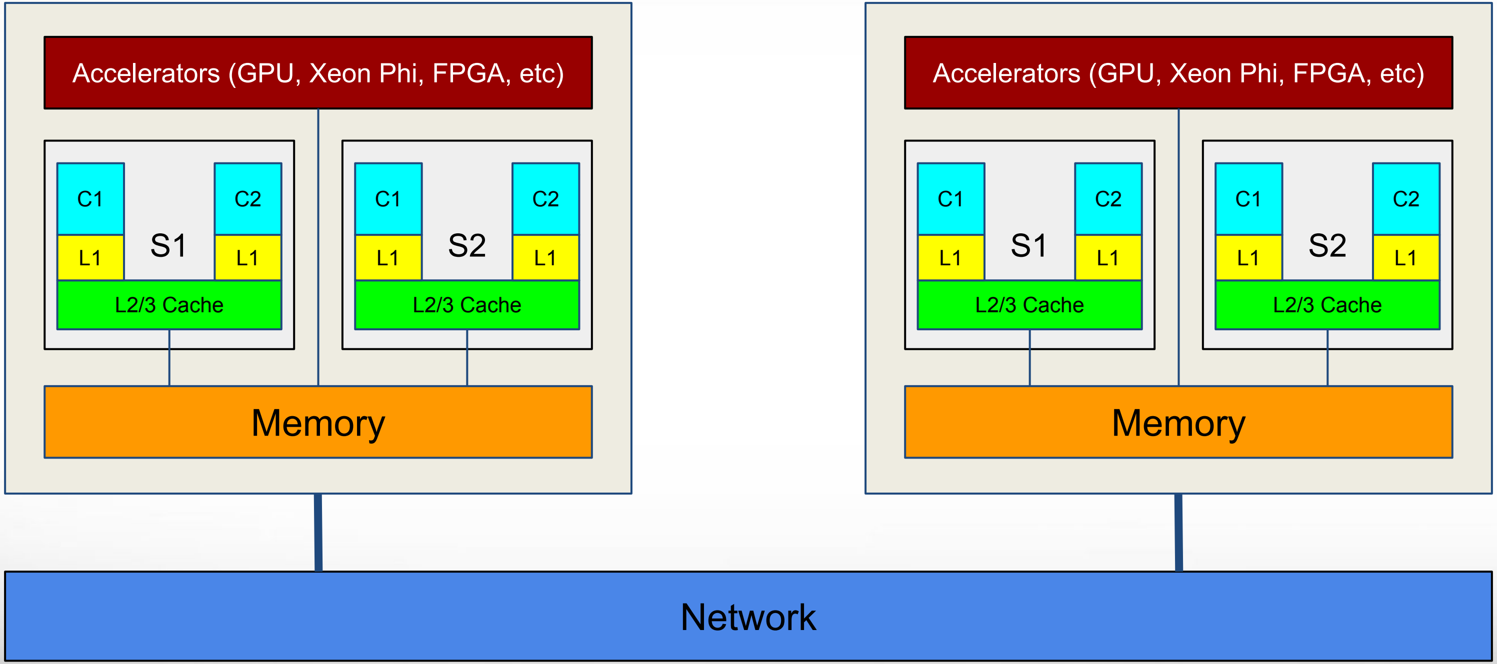
GPU Computing Applications



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

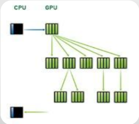
LLVM COMPILER INFRASTRUCTURE

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

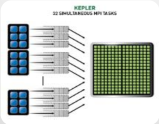
Hardware Capabilities

SMX

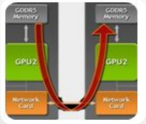
Dynamic Parallelism



HyperQ



GPUDirect



3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

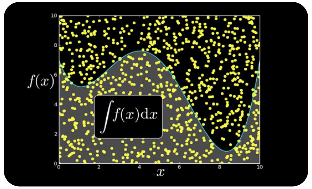
Libraries: Easy, High-Quality Acceleration

- **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

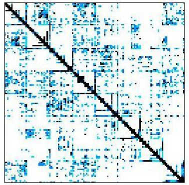
Some GPU-accelerated Libraries

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

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

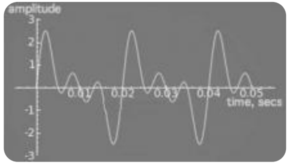
CULA tools

GPU Accelerated
Linear Algebra

MAGMA

Matrix Algebra on GPU
and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix
Computations

CUSP

Sparse Linear
Algebra

open source initiative

Thrust

C++ STL Features
for CUDA

open source initiative

CUDA-accelerated Application with Libraries

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

  `saxpy ( … )` ► **`cublasSaxpy ( … )`**

- **Step 2:** Manage data locality

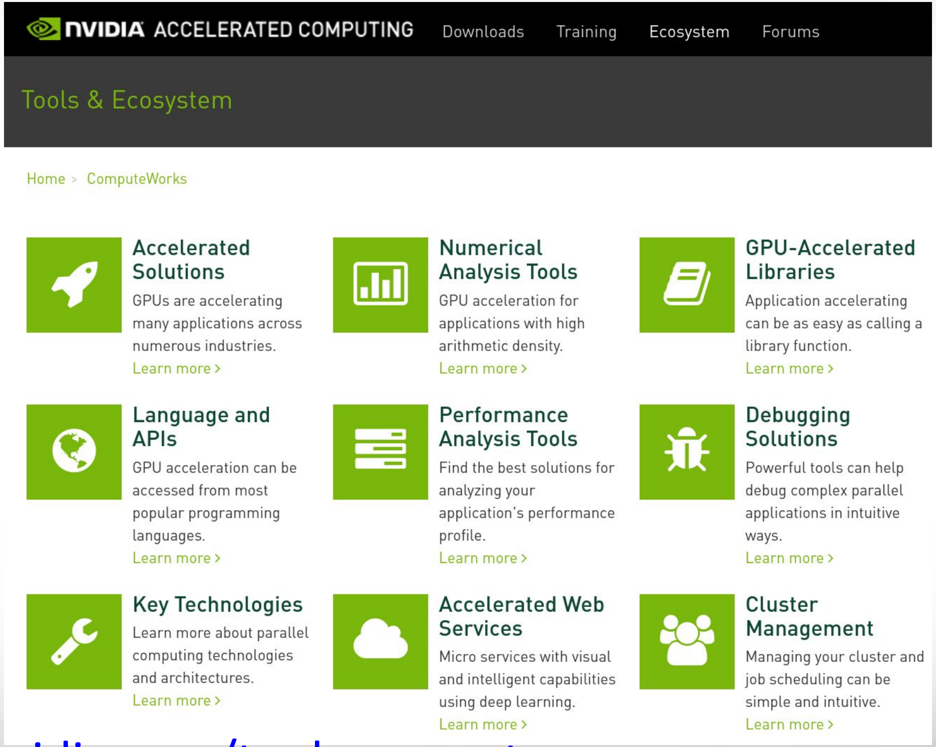
  - with CUDA: **`cudaMalloc()`**, **`cudaMemcpy()`**, etc.
  - with CUBLAS: **`cublasAlloc()`**, **`cublasSetVector()`**, etc.

- **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.



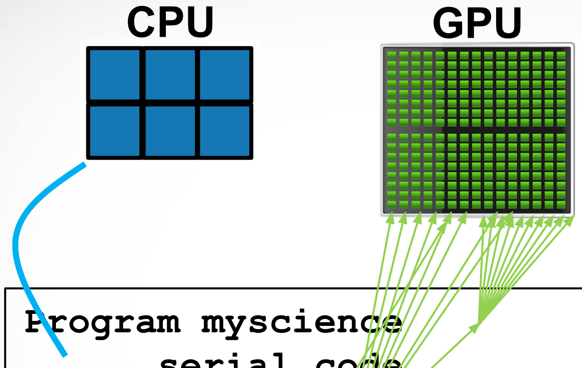
https://developer.nvidia.com/tools-ecosystem

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

OpenACC Directives




```
Program myscience
   ... serial code ...
!$acc kernels
   do k = 1,n1
      do i = 1,n2
         ... parallel code ...
      enddo
   enddo
!$acc end kernels
   ...
End Program myscience
```

OpenACC compiler Hint

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC

More Science, Less Programming

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

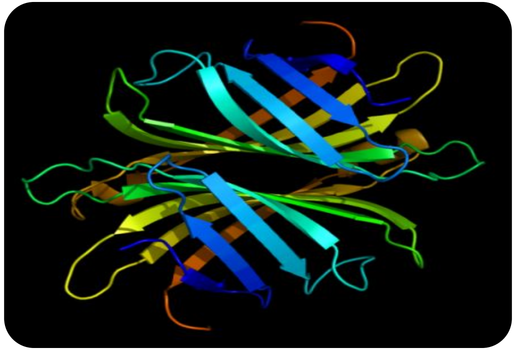
Valuation of Stock Portfolios using Monte Carlo

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	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	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, Copperhead
Java / F#	JCuda / Alea GPU

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

Thrust

```
// 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

Thrust C++ Template Library
http://developer.nvidia.com/thrust

CUDA Fortran
https://developer.nvidia.com/cuda-fortran

PyCUDA (Python)
https://developer.nvidia.com/pycuda

Alea GPU
http://www.aleagpu.com

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

Mathematica
http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/

CUDA C/C++ BASICS

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 8.0 (as of Sept. 2017).
- CUDA 9 Release Candidate is available.

Heterogeneous Computing

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

Host

Device

Heterogeneous Computing

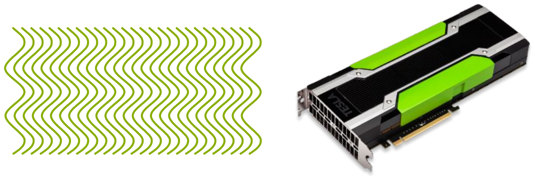
```
#include <iostream>
#include <algorithm>

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

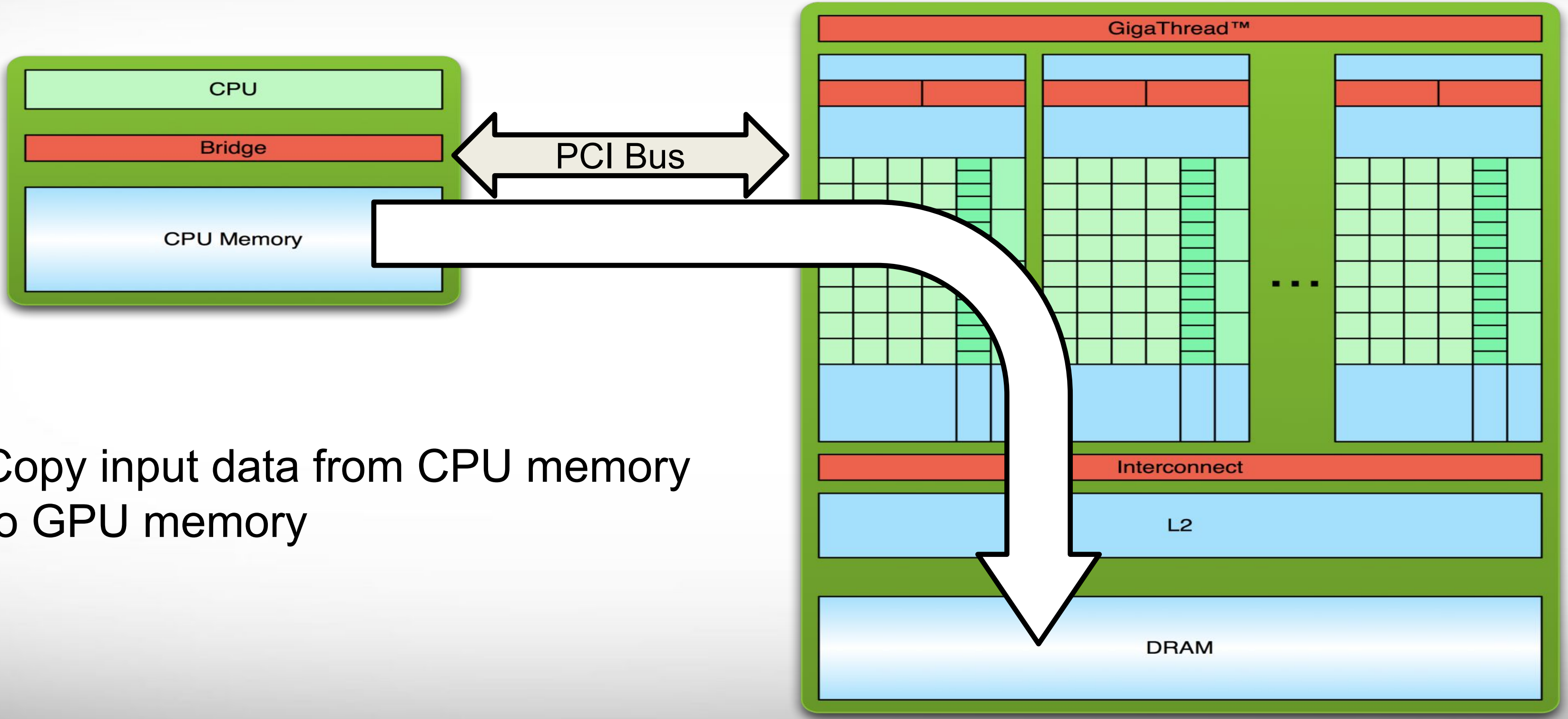
serial code

parallel code

serial code

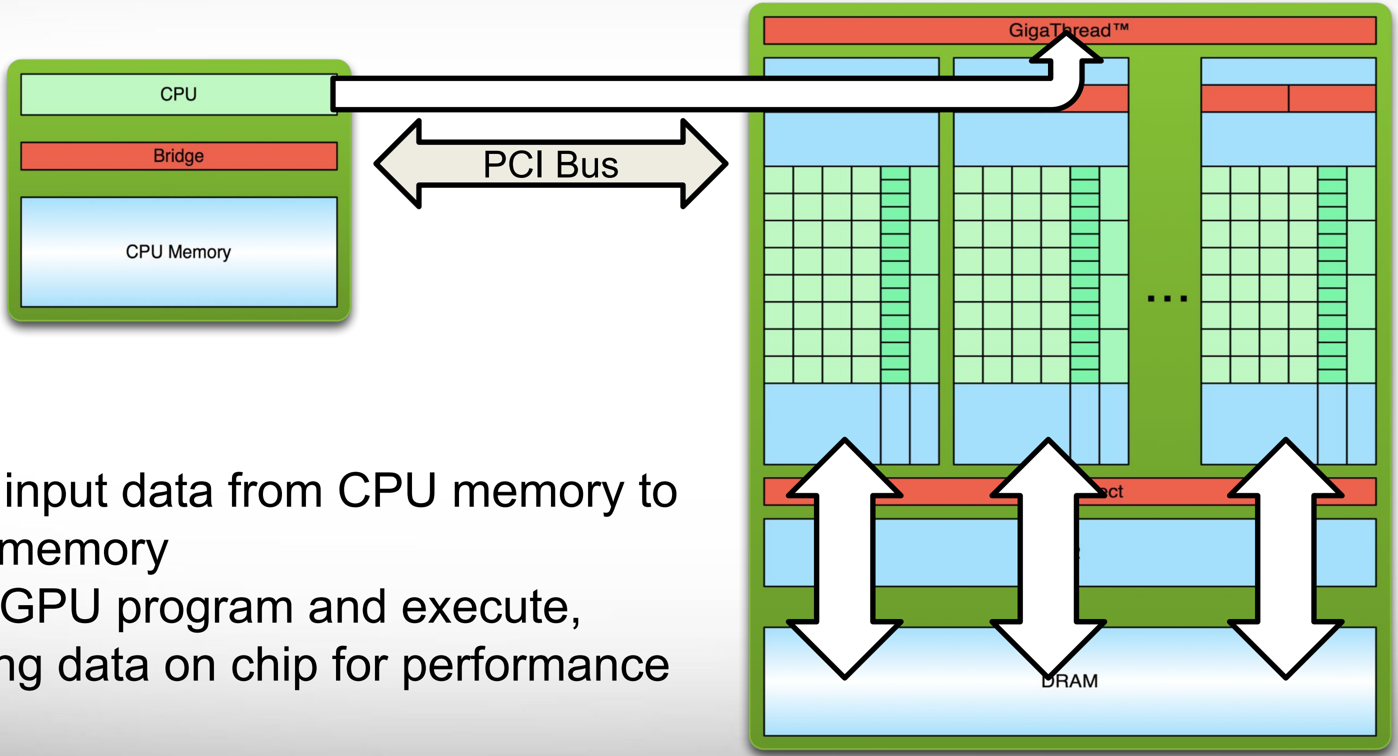


Simple Processing Flow



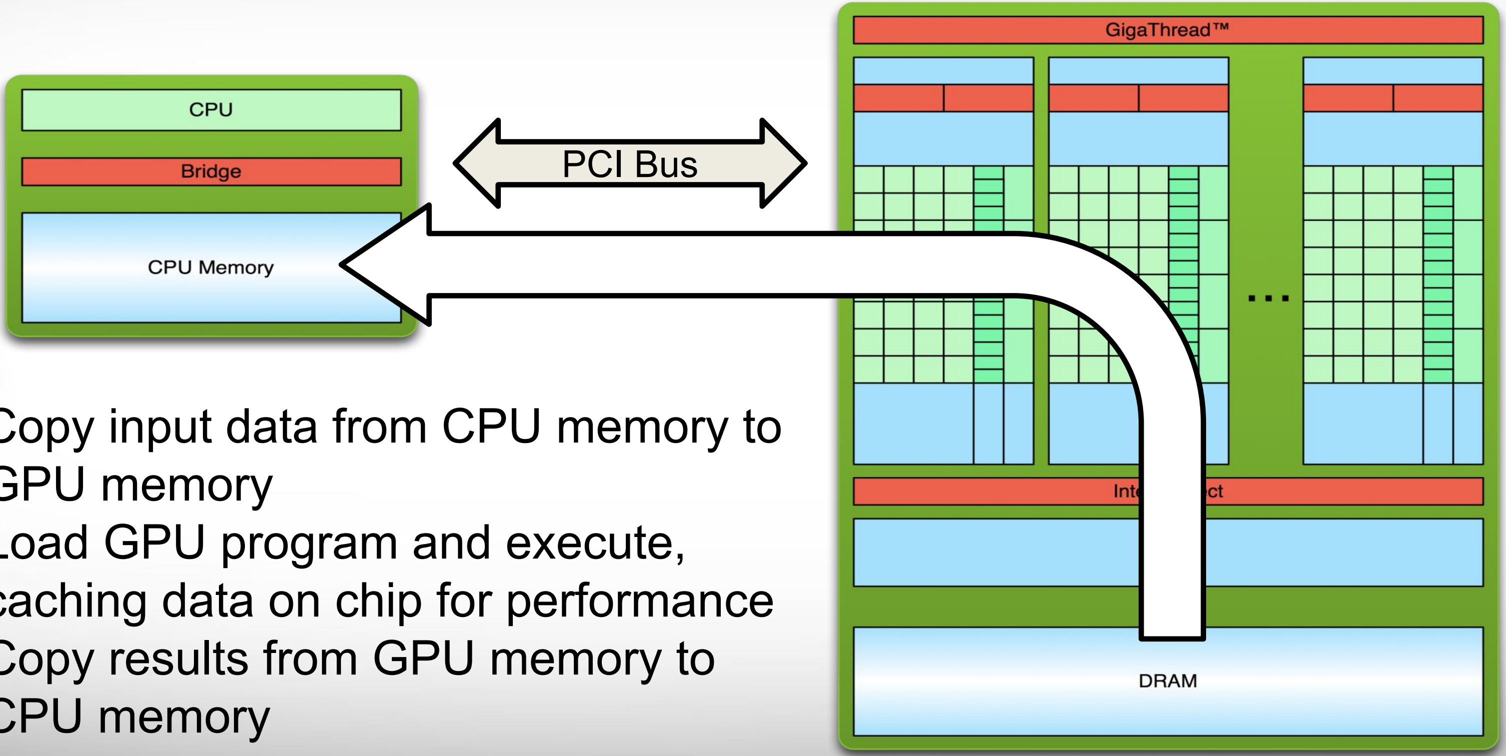
1. Copy input data from CPU memory to GPU memory

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

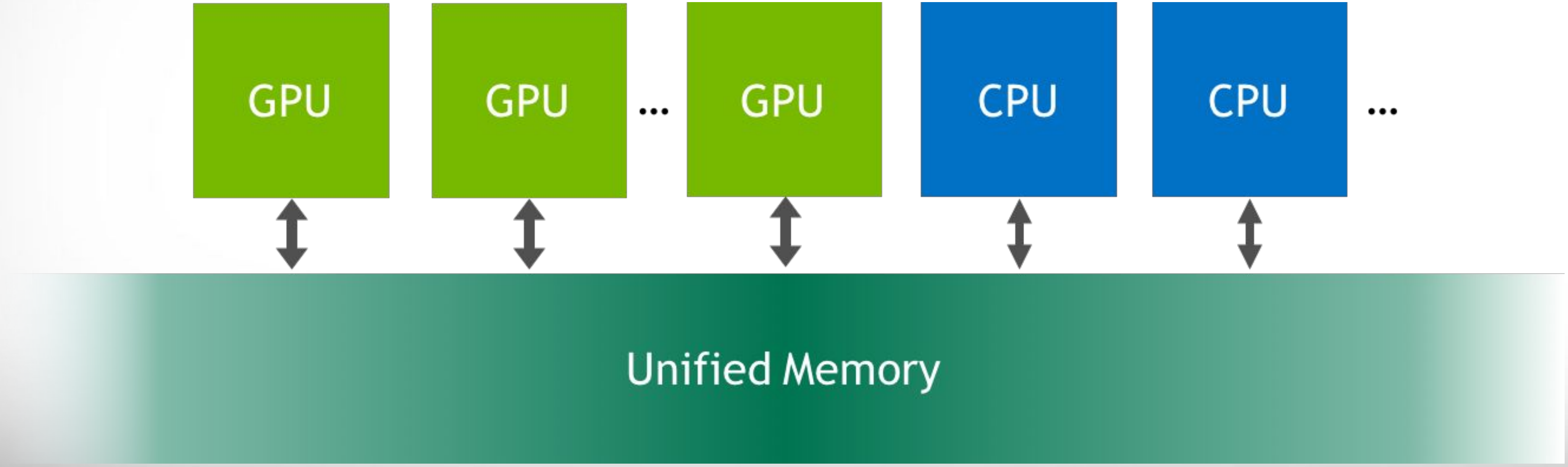


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Unified Memory

- 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;
}
```

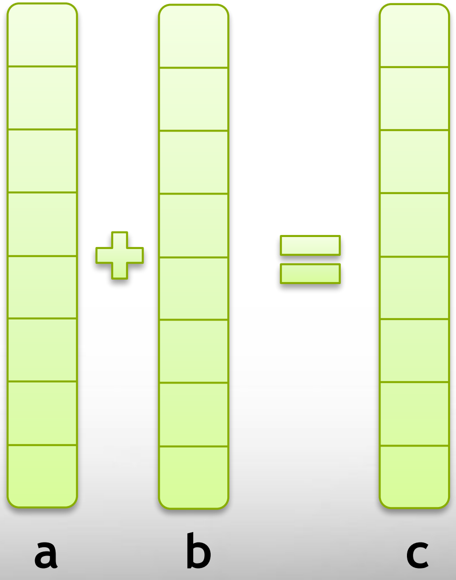
Output:

```
$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

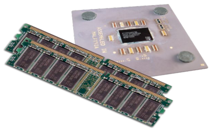
- Note that we use pointers for the variables

```
__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 index

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

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;
}
```

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

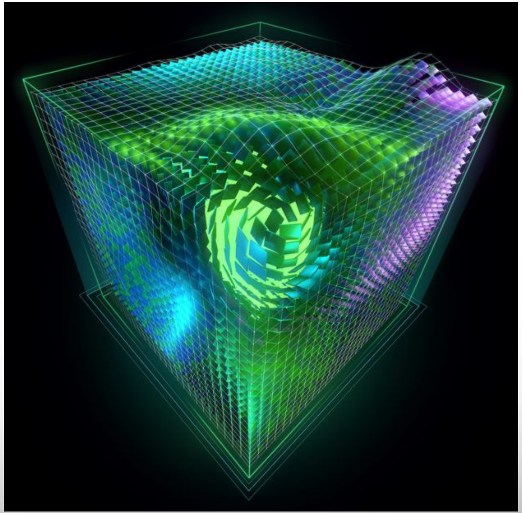
You can learn more about the details at

- 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

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);
```

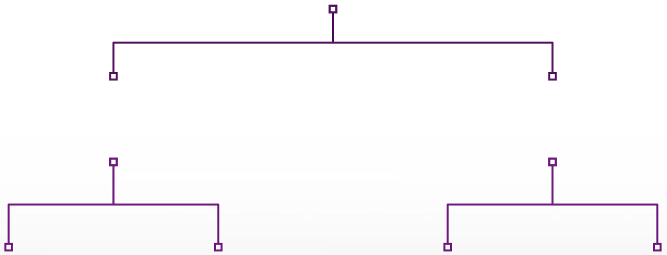
Example 1 - Kernel Definition

```
// 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

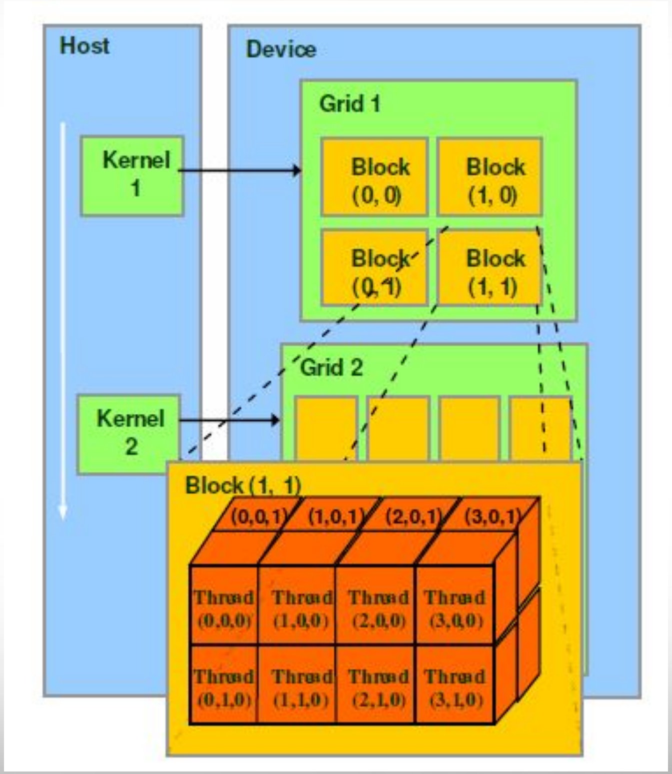
```
// Kernel invocation
int main()
{
   ...
// Call kernel with one block of N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Hierarchy of Threads



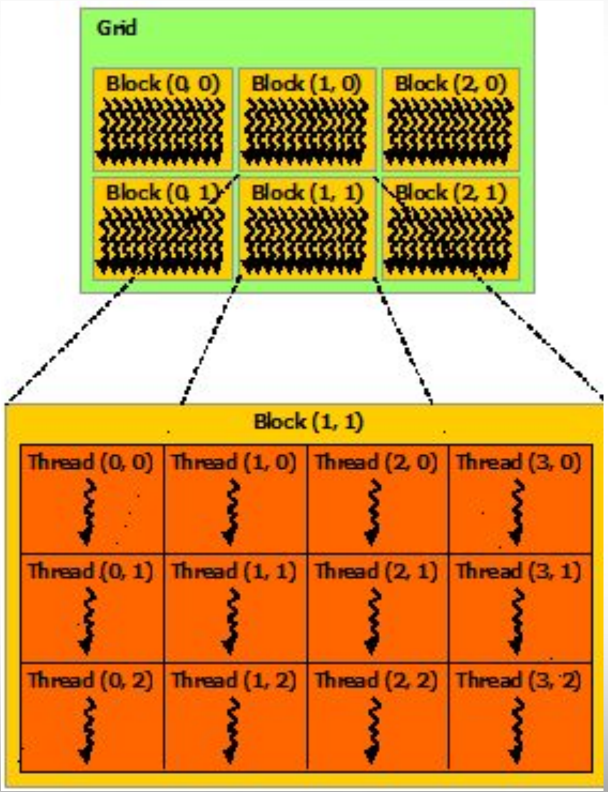
Thread Hierarchy - I

- 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`**.

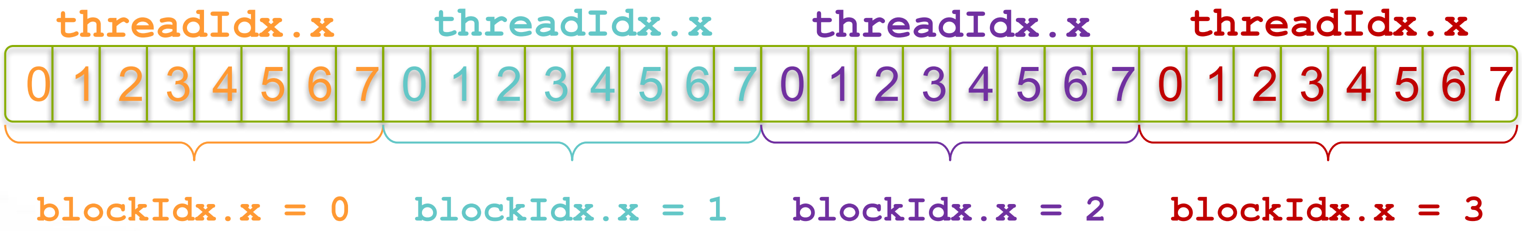


Thread Index and Thread ID

- **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?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

`blockDim.x = 8`

`threadIdx.x = 5`

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

`blockIdx.x = 2`

```
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 = blockDim.x**

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

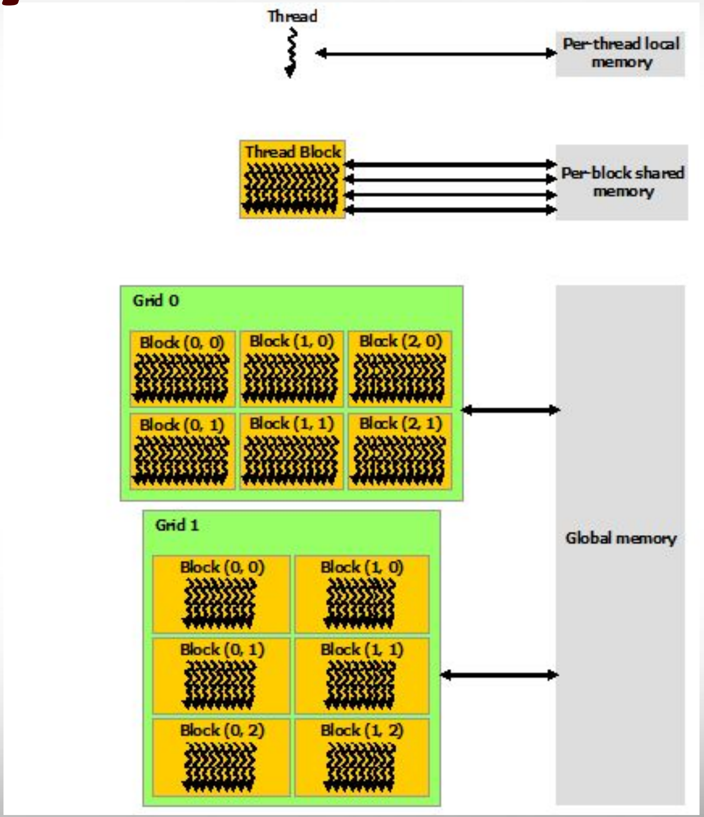
Why Bother with Threads?

- 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.

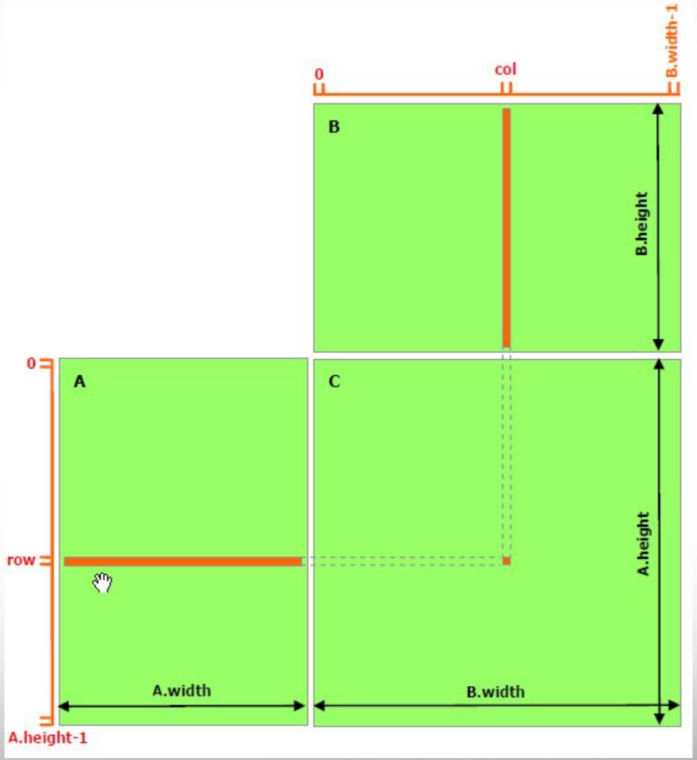
```
__shared__ float A[BLOCK_SIZE][BLOCK_SIZE];
```

- 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.

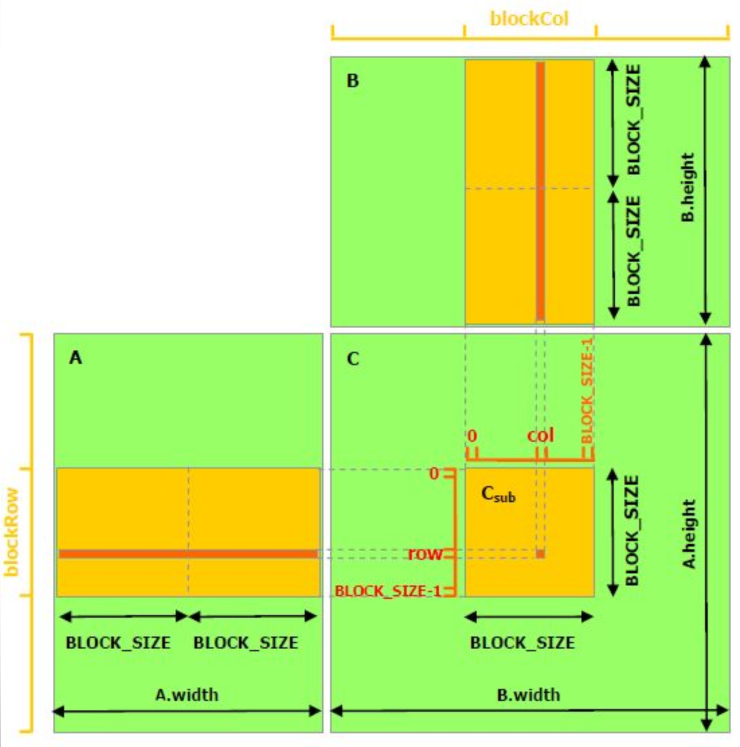
```
__global__ void MatMulKernel(Matrix A, Matrix B,
                                   Matrix C)
{
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]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

```
...
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

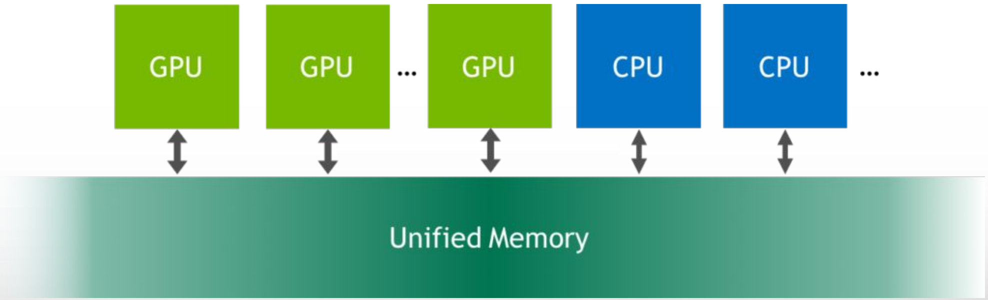
- Launching parallel threads
  - Launch `N` blocks with `blockDim.x` threads per block with `kernel<<<N, blockDim.x>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
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

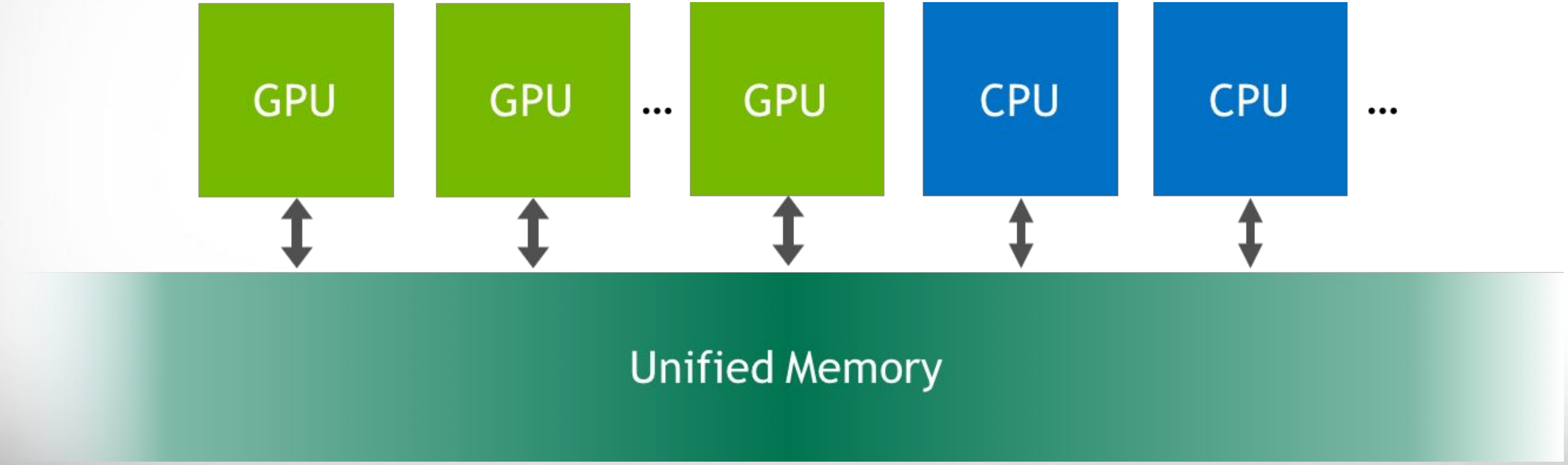
Unified Memory Programming



Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Unified Memory

- 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

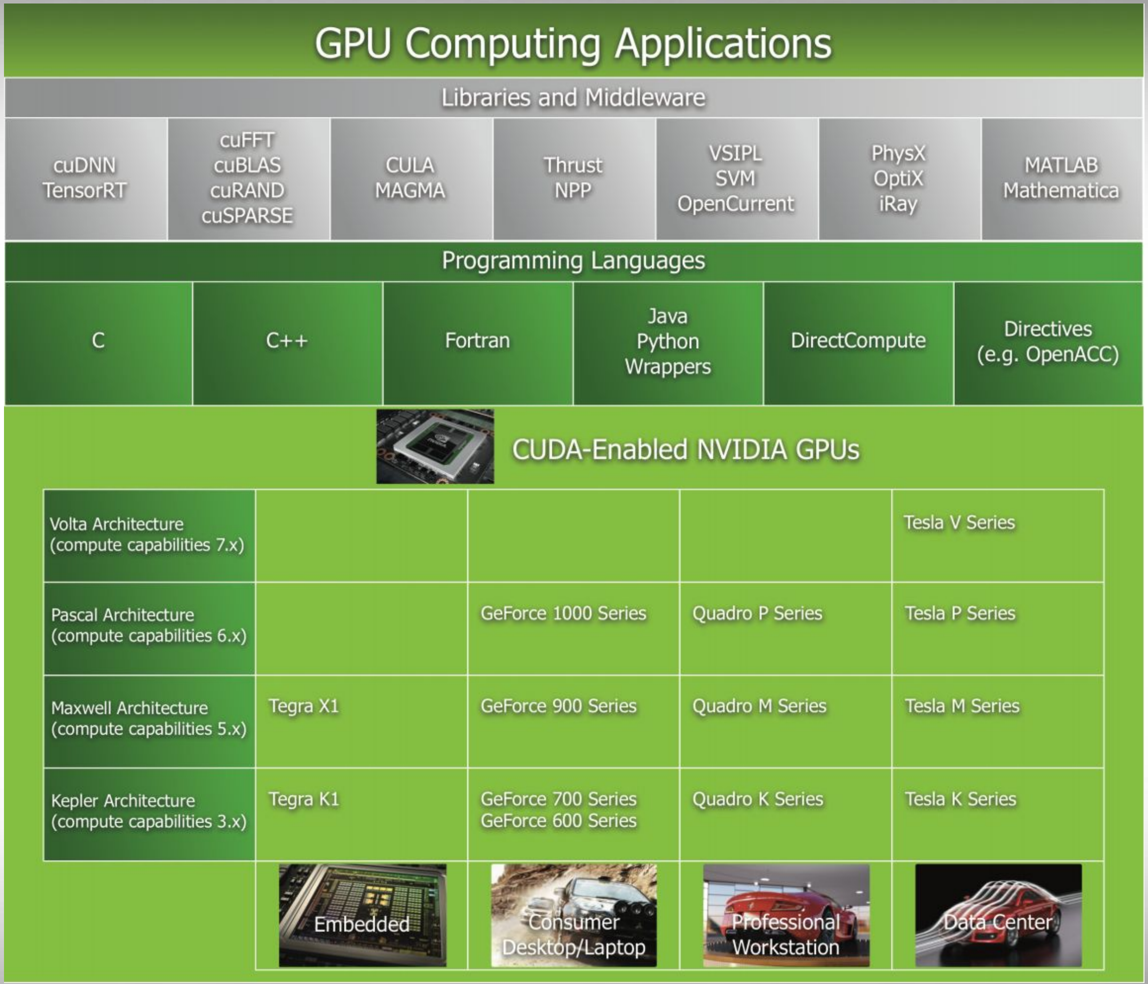
  Select current device: `cudaSetDevice(i)`

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

† requires OS and device support

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.



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 via its Academic Programs.
- Supports from Texas A&M Engineering Experiment Station (TEES) and 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;
}
```

Running CUDA Code on Ada

https://github.com/jtao/coehpc

```
# load CUDA module
$ml CUDA/9.1.85

# copy sample code to your scratch space
$cd $SCRATCH
$cp -r /scratch/training/CUDA .

# compile CUDA code
$cd CUDA
$nvcc hello_world_host.cu -o hello_world

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