Introduction to CUDA Programming

Jian Tao
jtao@tamu.edu
Spring 2023 HPRC Short Course
2/28/2023
Introduction to CUDA Programming

Part I. Getting Started with FASTER (~10 mins)
Part II. GPU as an Accelerator (~40 mins)
Part III. Running CUDA Code on FASTER (~30 mins)
Part IV. CUDA C/C++ Basics (~50 mins)
Q&A and Break (10 mins)
Part I. Getting Started with FASTER

TAMU HPRC Short Course: Getting Started with FASTER and ACES
# FASTER Cluster

[Link to FASTER Cluster wiki](https://hprc.tamu.edu/wiki/FASTER:Intro)

<table>
<thead>
<tr>
<th>Resources</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-core login nodes</td>
<td>4 (3 for TAMU, 1 for ACCESS)</td>
</tr>
<tr>
<td>64-core compute nodes (256GB RAM each)</td>
<td>180 (11,520 cores)</td>
</tr>
<tr>
<td>Composable GPUs</td>
<td>200 T4 16GB</td>
</tr>
<tr>
<td></td>
<td>40 A100 40GB</td>
</tr>
<tr>
<td></td>
<td>10 A10 24GB</td>
</tr>
<tr>
<td></td>
<td>4 A30 24GB</td>
</tr>
<tr>
<td></td>
<td>8 A40 48GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Mellanox HDR100 InfiniBand (MPI and storage)</td>
</tr>
<tr>
<td></td>
<td>Liqid PCIe Gen4 (GPU composability)</td>
</tr>
<tr>
<td>Global Disk</td>
<td>5PB DDN Lustre appliances</td>
</tr>
</tbody>
</table>

FASTER (Fostering Accelerated Sciences Transformation Education and Research) is a 180-node Intel cluster from Dell featuring the Intel Ice Lake processor.
Composability at the Hardware Level

Traditional Server Configuration

Composable Resources

Liqid Fabric

Server pool

Accelerator pool (GPUs, FPGA, etc.)

Storage pool (SSDs)

Composable Server Configuration

server Accelerator SSDs

hprc.tamu.edu/wiki/FASTER: Intro
# ACES - Accelerating Computing for Emerging Sciences (Phase I)

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphcore IPU</td>
<td>16</td>
<td>16 Colossus GC200 IPUs and dual AMD Rome CPU server on a 100 GbE RoCE fabric</td>
</tr>
<tr>
<td>Intel FPGA PAC D5005</td>
<td>2</td>
<td>FPGA SOC with Intel Stratix 10 SX FPGAs, 64 bit quad-core Arm Cortex-A53 processors, and 32GB DDR4</td>
</tr>
<tr>
<td>Intel Optane SSDs</td>
<td>8</td>
<td>3 TB of Intel Optane SSDs addressable as memory using MemVerge Memory Machine.</td>
</tr>
</tbody>
</table>

ACES Phase I components are available through [FASTER](#).
Accessing the HPRC Portal

- HPRC webpage: [hprc.tamu.edu](https://hprc.tamu.edu), Portal dropdown menu
Accessing FASTER via the HPRC Portal (TAMU)

Log-in using your TAMU NetID credentials.
Accessing FASTER via the HPRC Portal (ACCESS)

Log-in using your ACCESS credentials.

Select the Identity Provider appropriate for your account.
Login HPRC Portal - FASTER/FASTER(ACCESS)
FASTER Shell Access - Portal

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.
- Authorized users must also adhere to ALL policies at: https://hprc.tamu.edu/policies

!! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!
FASTER Shell Access - Shell

Password: 

UPDATE (12:11a 02/20/2023): There was another storage incident between 11:18-11:30p. We are still investigating this new incident since the indicators observed so far were not related to the previous user's jobs.

UPDATE (10:16p 02/19/2023): We may have isolated and removed the batch jobs that were impacting the Grace and FASTER shared storage. We are continuing to monitor the storage for any further issues.

Original announcement (7:13p 02/10/2023): Both the FASTER and Grace clusters are currently having issues with their shared storage since about 5:15p February 10th. The root cause and recovery options are under investigation.

[02feb2023] FASTER hardware update: 05085 FPGAs are currently unavailable.

To see these messages again, run the motd command.

Your current disk quotas are:

<table>
<thead>
<tr>
<th>Disk</th>
<th>Usage</th>
<th>Limit</th>
<th>File Usage</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/jtao</td>
<td>4KB</td>
<td>10.0G</td>
<td>5</td>
<td>10000</td>
</tr>
<tr>
<td>/scratch/user/jtao</td>
<td>21.5G</td>
<td>1.0T</td>
<td>156632</td>
<td>250000</td>
</tr>
<tr>
<td>/scratch/group/hpcc</td>
<td>4.0T</td>
<td>0.0T</td>
<td>628476</td>
<td>1000000</td>
</tr>
</tbody>
</table>

* Quota increase for /scratch/group/hpcc will expire on Dec 31, 2026

Type 'showquota' to view these quotas again.
Commands to copy the materials

- Navigate to your personal scratch directory
  
  ```
  $ cd $SCRATCH
  ```

- Files for this course are located at
  
  `/scratch/training/cuda.exercise.tgz`

  Make a copy in your personal scratch directory
  
  ```
  $ cp /scratch/training/cuda.exercise.tgz $SCRATCH/
  ```

- Extract the files
  
  ```
  $ tar -zxvf cuda.exercise.tgz
  ```

- Enter this directory (your local copy)
  
  ```
  $ cd CUDA
  ```
Load CUDA Module, Compile, and Run

```
[jtao@faster1 hello_world]$ ml CUDA
[jtao@faster1 hello_world]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Cuda compilation tools, release 12.0, V12.0.76
Build cuda_12.0.r12.0/compiler.31968024_0
[jtao@faster1 hello_world]$ nvcc ./hello_world_device.cu
[jtao@faster1 hello_world]$ ./a.out
Hello World!
[jtao@faster1 hello_world]$ 
```
Part II. GPU as an Accelerator
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?

Traditional CPUs are not economically feasible

2.3 PFlops

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks

7000 homes

GPU-accelerated computing started a new era

7.0 Megawatts

7.0 Megawatts

7.0 Megawatts
Add GPUs: Accelerate Science Applications

Application Code

Compute-Intensive Functions

Rest of Sequential CPU Code

Use GPU to Parallelize

CPU

GPU
HPC - Distributed Heterogeneous System

Programming Models: MPI + (CUDA, OpenCL, OpenMP, OpenACC, etc.)
Amdahl's Law

\[ S_{latency}(s) = \frac{1}{(1 - p) + \frac{p}{s}} \]

- \( S_{latency} \) 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


## 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

- Libraries: "Drop-in" Acceleration
- OpenACC Directives: Easily Accelerate Applications
- Programming Languages: Maximum Flexibility
3 Ways to Accelerate 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

CUDA-accelerated Application with Libraries

- **Step 1:** Substitute library calls with equivalent CUDA library calls
  \[
  \text{saxpy } ( \ldots ) \quad \Rightarrow \quad \text{cublasSaxpy } ( \ldots )
  \]

- **Step 2:** Manage data locality
  - with CUDA: \texttt{cudaMalloc()}, \texttt{cudaMemcpy()}, etc.
  - with CUBLAS: \texttt{cublasAlloc()}, \texttt{cublasSetVector()}, etc.

- **Step 3:** Rebuild and link the CUDA-accelerated library
  \[
  \text{"nvcc myobj.o -l cublas"
  \]
Explore the CUDA (Libraries) Ecosystem

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

NVIDIA CUDA Tools & Ecosystem
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - 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

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

- Libraries: “Drop-in” Acceleration
- OpenACC Directives: Easily Accelerate Applications
- Programming Languages: Maximum Flexibility
## GPU Programming Languages

<table>
<thead>
<tr>
<th>Category</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical analytics</td>
<td>MATLAB, Mathematica, LabVIEW</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenACC, CUDA Fortran</td>
</tr>
<tr>
<td>C</td>
<td>OpenACC, CUDA C, OpenCL</td>
</tr>
<tr>
<td>C++</td>
<td>Thrust, CUDA C++, OpenCL</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA, PyOpenCL, CuPy</td>
</tr>
<tr>
<td>Julia / Java</td>
<td>JuliaGPU/CUDA.jl, jcuda</td>
</tr>
</tbody>
</table>
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

```cpp
// 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++

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

Mathematica
Part III. Running CUDA Code on FASTER
# load CUDA module
$ml CUDA

# copy sample code to your scratch space
$tar -zxvf cuda.exercise.tgz

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

# edit job script & submit your GPU job
$sbatch faster_cuda_run.sh
Part IV. 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 11.5 (as of Nov 2021).
Heterogeneous Computing

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

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
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!

```c
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:

```bash
$ nvcc hello_world.cu
$ ./a.out
$ Hello World!
```
Hello World! with Device Code

```c
__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

```c
__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

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

• mykernel() does nothing!

Output:

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

```c
__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

```c
__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: \texttt{add()} \\

• Returning to our \texttt{add()} kernel \\

\begin{verbatim}
    __global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
    }
\end{verbatim}

• Let’s take a look at main()…
Addition on the Device: `main()`

```c
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
 cudaMemcpy(d_a); cudaMemcpy(d_b); cudaMemcpy(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`

```c
__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

```c
__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

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
</table>
Vector Addition on the Device: add()

• Returning to our parallelized `add()` kernel

```c
__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()…
#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()`

```c
// 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

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

int main() {
    int *ret;
    cudaMemcpy(&ret, 1000 * sizeof(int));
    VecAdd<<<1000, 1>>>(ret, 10, 100);
    cudaMemcpy(&ret, 1000 * sizeof(int));
    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

```c
__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;
}
```
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.
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

```c
__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

```c
__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

  cudaMemcpyDeviceSynchronize() 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)
Acknowledgments

- Educational materials from NVIDIA Deep Learning Institute via its University Ambassador Program.
- Support 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).
- Support from NSF OAC Award #2019129 - MRI: Acquisition of FASTER - Fostering Accelerated Sciences Transformation Education and Research
- Support from NSF OAC Award #2112356 - Category II: ACES - Accelerating Computing for Emerging Sciences
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