

# Introduction to CUDA Programming

**Jian Tao**

[jtao@tamu.edu](mailto:jtao@tamu.edu)

Spring 2021 HPRC Short Course

11/12/2021



TEXAS A&M UNIVERSITY

Visualization



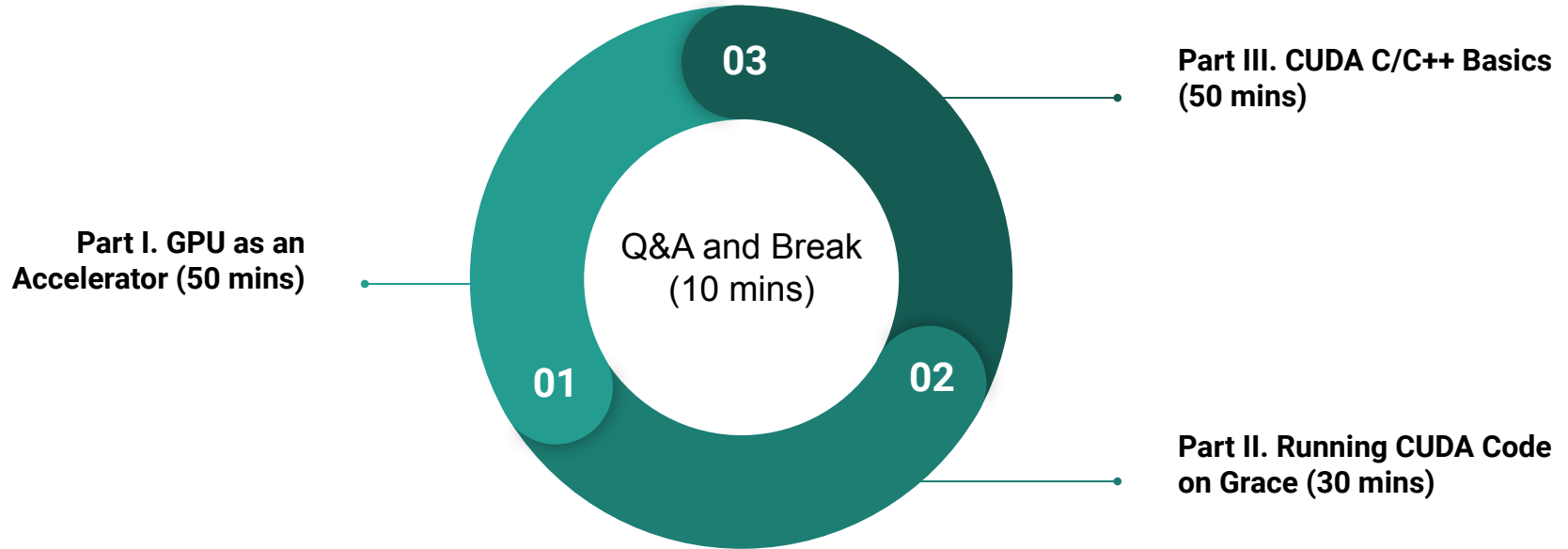
High Performance  
Research Computing  
DIVISION OF RESEARCH



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

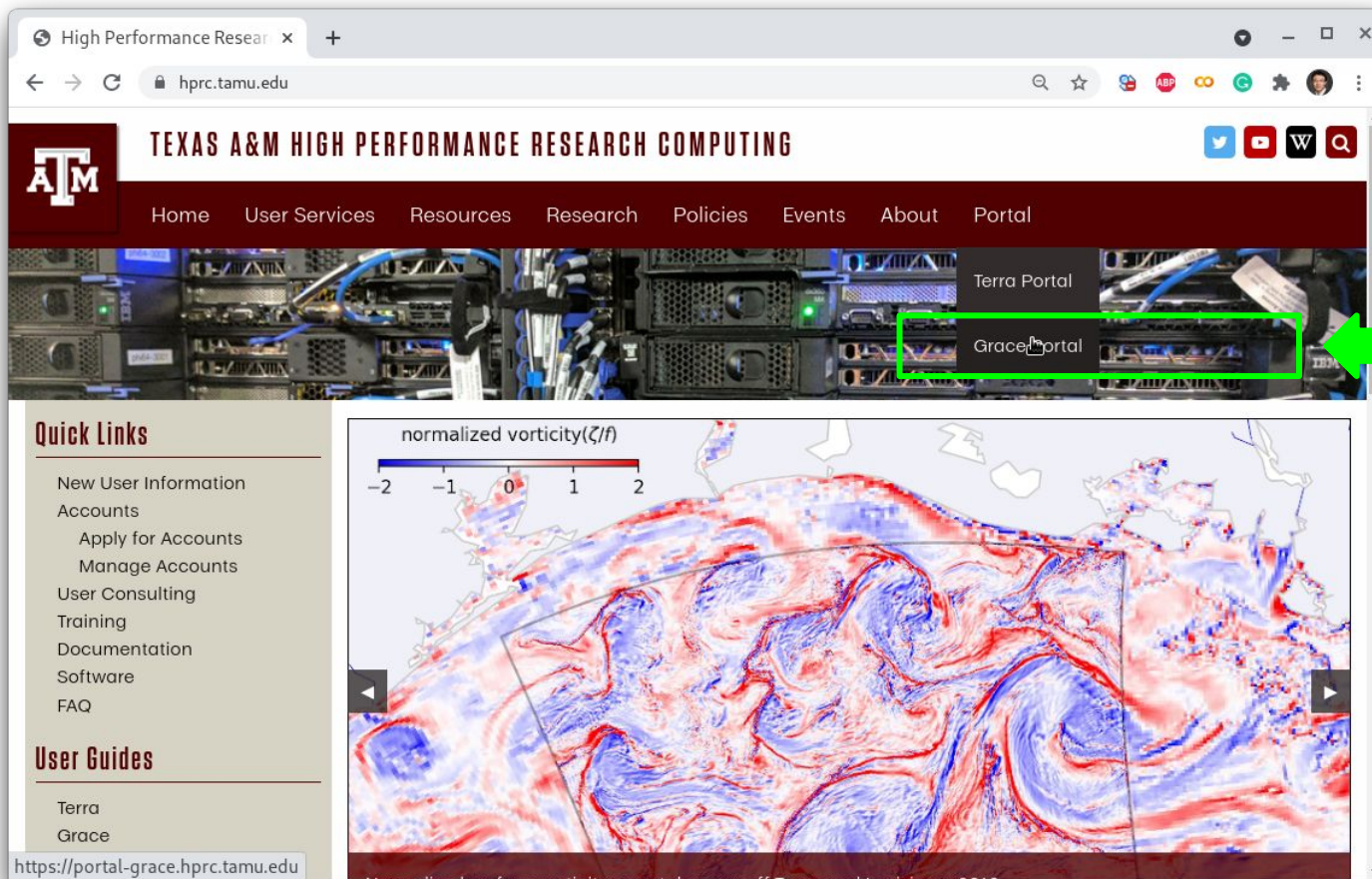
**Molecular Jump-Rope: Multiringed Metal-Complexes That Really Know How To Jump**

"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 (Grace)



The screenshot shows a web browser window with the URL `hprc.tamu.edu`. The page header includes the Texas A&M University logo and the text "TEXAS A&M HIGH PERFORMANCE RESEARCH COMPUTING". A navigation menu contains links for Home, User Services, Resources, Research, Policies, Events, About, and Portal. Below the menu is a banner image of server racks with a dropdown menu showing "Terra Portal" and "Grace Portal". A green box highlights the "Grace Portal" link, and a green arrow points to it from the right. On the left side, there is a "Quick Links" section with various user service links and a "User Guides" section with links for Terra and Grace. The main content area features a scientific visualization titled "normalized vorticity( $\zeta/f$ )" showing a map of the North Pacific with swirling patterns in red and blue.

High Performance Resear x +

hprc.tamu.edu

ATM TEXAS A&M HIGH PERFORMANCE RESEARCH COMPUTING

Home User Services Resources Research Policies Events About Portal

Terra Portal

Grace Portal

Quick Links

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

User Guides

- Terra
- Grace

normalized vorticity( $\zeta/f$ )

`https://portal-grace.hprc.tamu.edu`

# Grace Shell Access - Portal

High Performance Resear x Dashboard - TAMU HPRC x +

portal-grace.hprc.tamu.edu/pun/sys/dashboard

TAMU HPRC OnDemand (Grace) Files Jobs Clusters Interactive Apps Dashboard

>\_gracc Shell Access Develop Help Log Out

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

Message of the Day

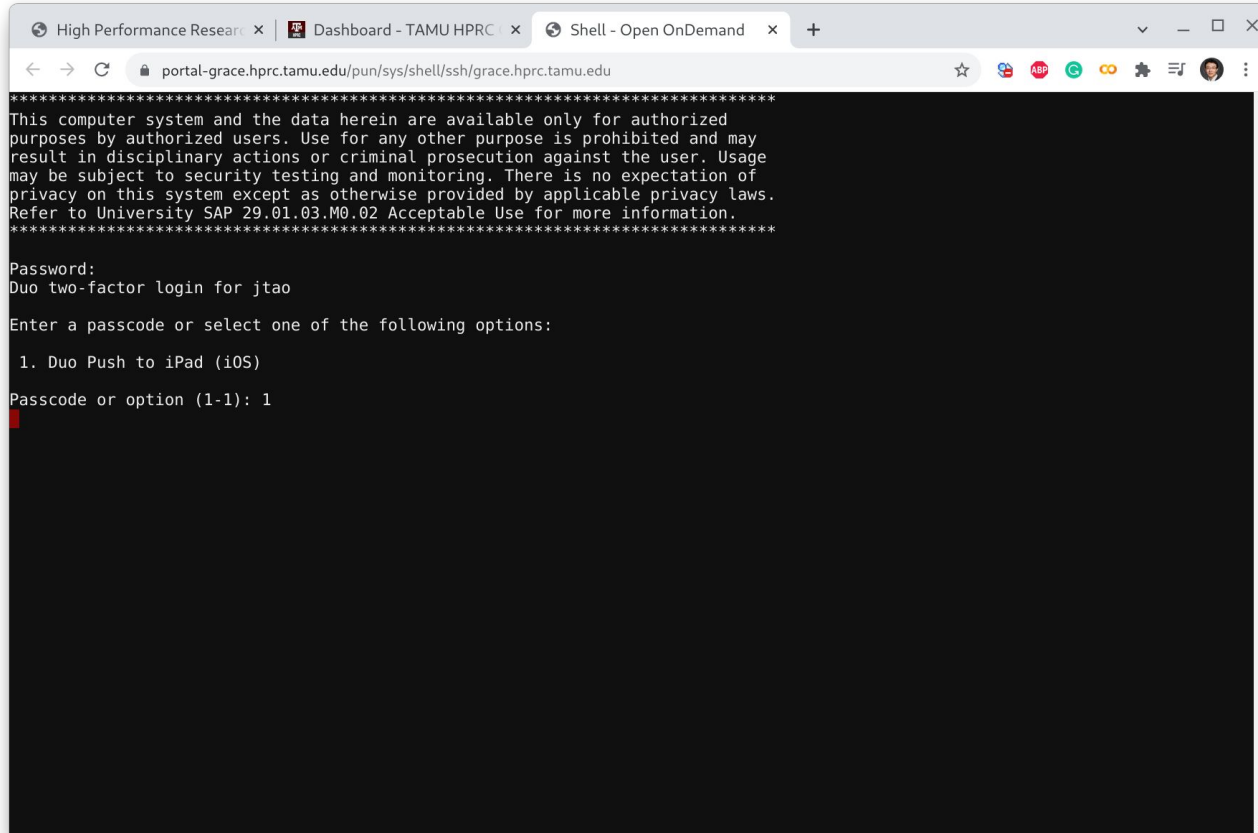
**IMPORTANT POLICY INFORMATION**

- 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. !!

portal-grace.hprc.tamu.edu/.../grace.hprc.tamu.edu

# Grace Shell Access - Shell



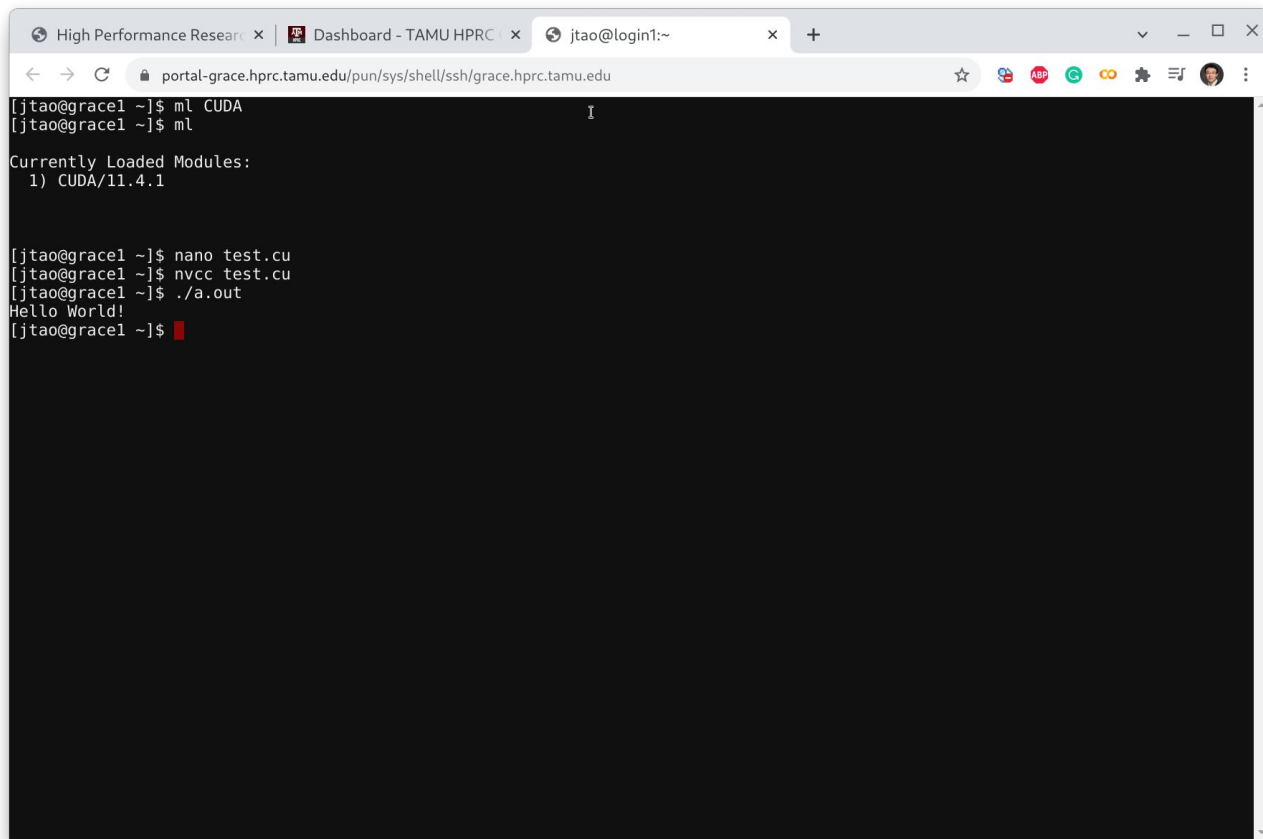
```
High Performance Research x Dashboard - TAMU HPRC x Shell - Open OnDemand x +
portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu
*****
This computer system and the data herein are available only for authorized
purposes by authorized users. Use for any other purpose is prohibited and may
result in disciplinary actions or criminal prosecution against the user. Usage
may be subject to security testing and monitoring. There is no expectation of
privacy on this system except as otherwise provided by applicable privacy laws.
Refer to University SAP 29.01.03.M0.02 Acceptable Use for more information.
*****
Password:
Duo two-factor login for jtao

Enter a passcode or select one of the following options:

1. Duo Push to iPad (iOS)

Passcode or option (1-1): 1
```

# Load CUDA Module, Compile, and Run



```
[jtao@grace1 ~]$ m1 CUDA
[jtao@grace1 ~]$ m1

Currently Loaded Modules:
 1) CUDA/11.4.1

[jtao@grace1 ~]$ nano test.cu
[jtao@grace1 ~]$ nvcc test.cu
[jtao@grace1 ~]$ ./a.out
Hello World!
[jtao@grace1 ~]$
```

# 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.

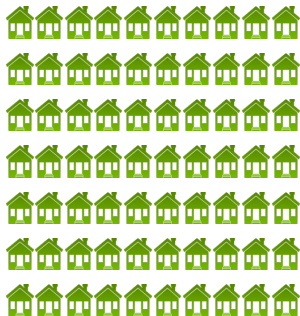
# Why Computing Perf/Watt Matters?

2.3 PFlops



7.0  
Megawatts

7000 homes

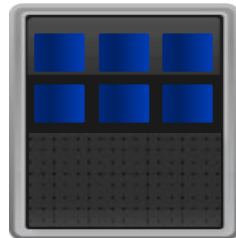


7.0  
Megawatts

Traditional CPUs are  
not economically feasible

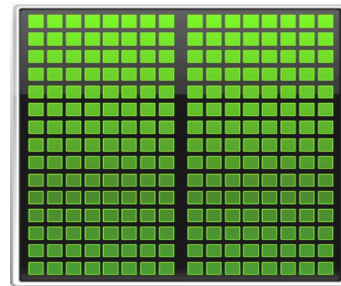
CPU

Optimized for  
Serial Tasks



GPU Accelerator






Optimized for Many  
Parallel Tasks



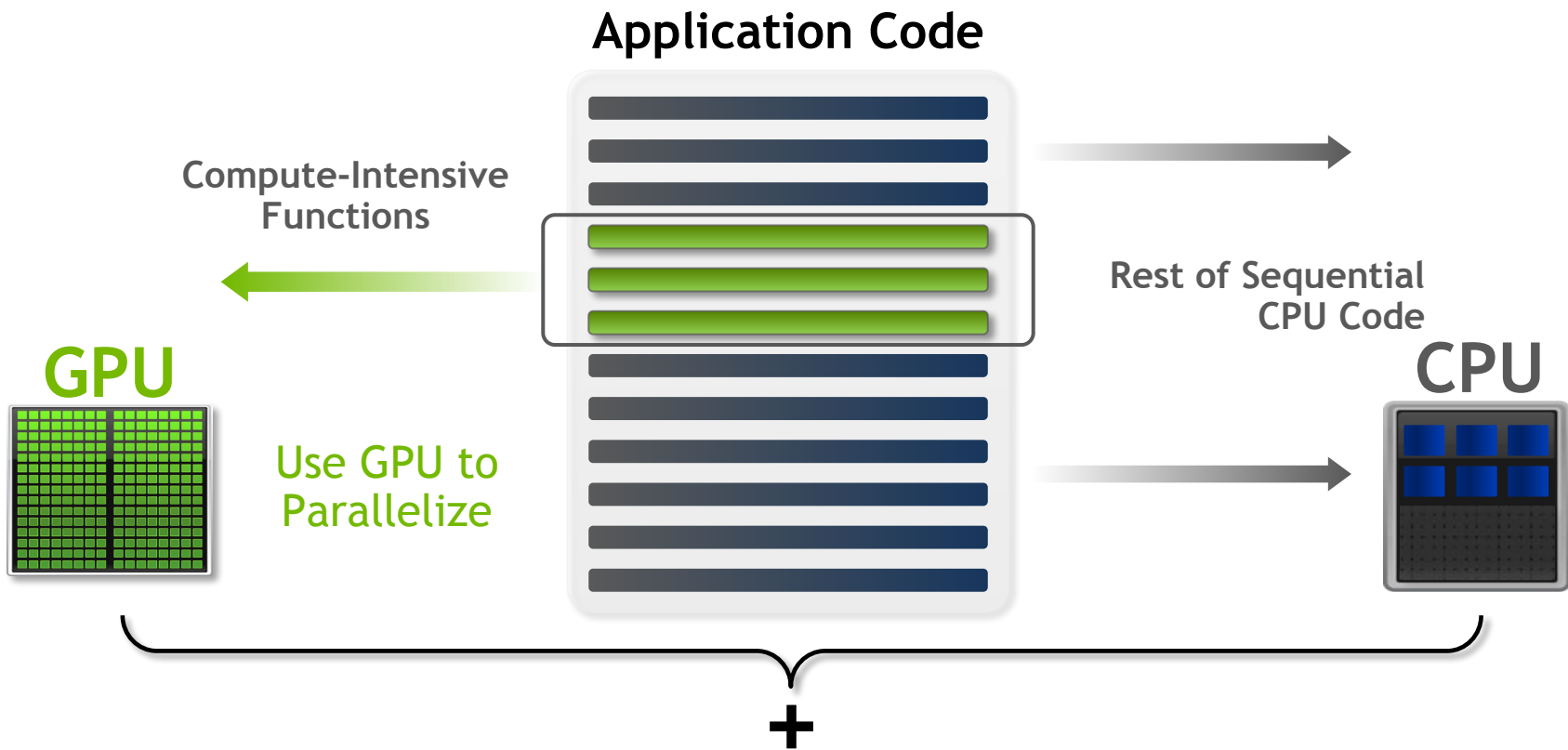
GPU-accelerated computing  
started a new era

# GPU Computing Applications

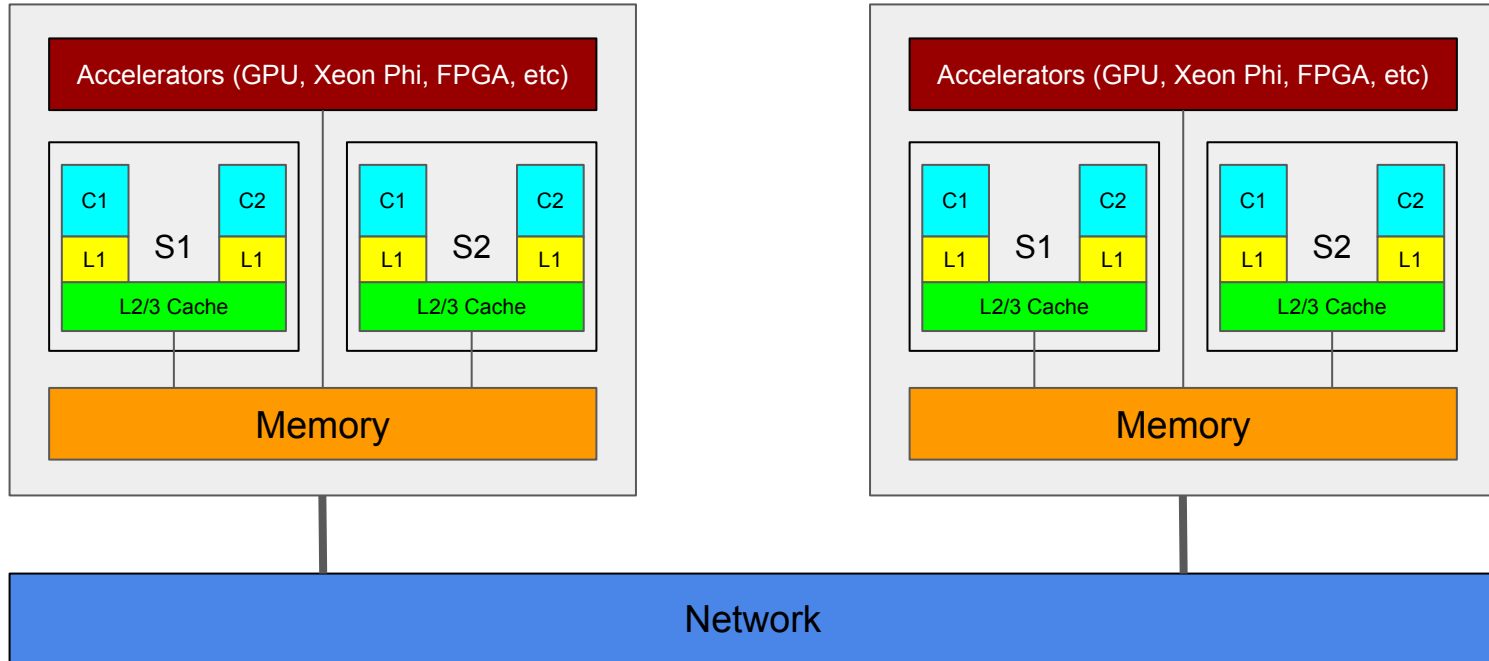
A catalog of GPU-accelerated applications can be found at <https://www.nvidia.com/en-us/gpu-accelerate-d-applications/>.

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)	
			 <b>CUDA-Enabled NVIDIA GPUs</b>			
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



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

- $S_{latency}$  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

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

Programming  
Approaches

Libraries

“Drop-in” Acceleration

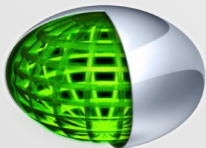
OpenACC  
Directives

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



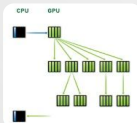
Enables compiling new languages to CUDA platform, and  
CUDA languages to other architectures

Hardware  
Capabilities

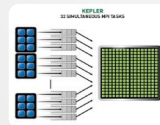
SMX



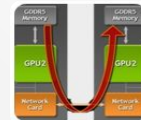
Dynamic Parallelism



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

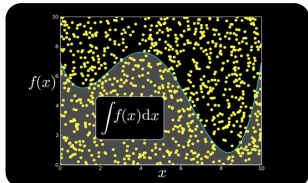
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions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

# NVIDIA CUDA-X GPU-Accelerated Libraries

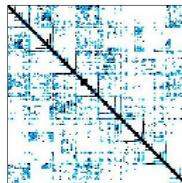
<https://developer.nvidia.com/gpu-accelerated-libraries>



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



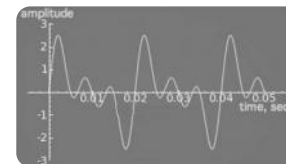
Vector Signal  
Image Processing



GPU Accelerated  
Linear Algebra



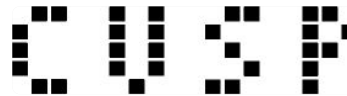
Matrix Algebra on GPI  
and Multicore



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.

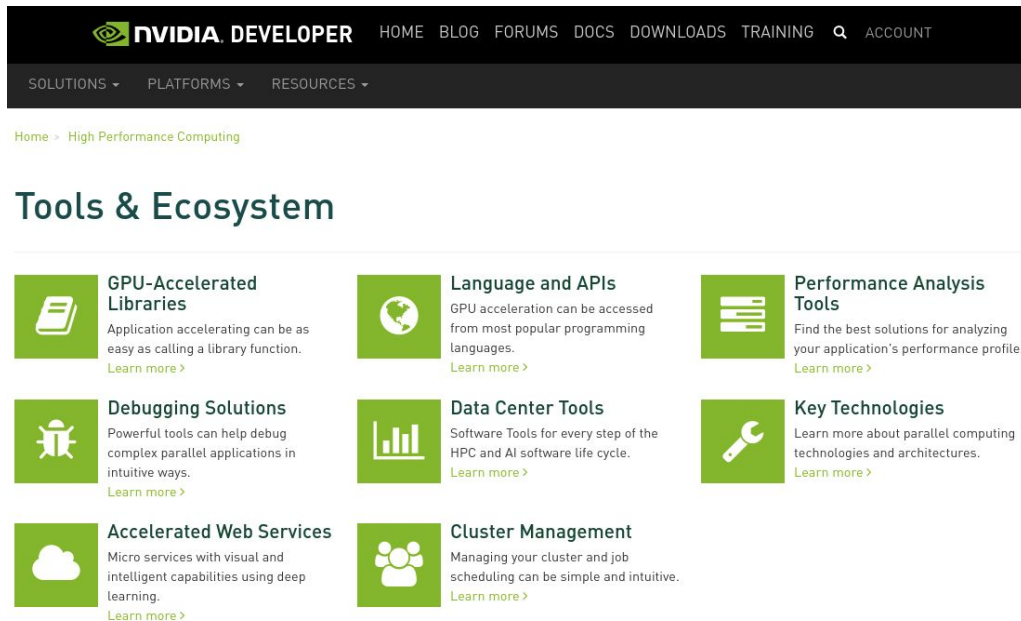
- with CUBLAS:      `cublasAlloc()`, `cublasSetVector()`, etc.

- **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 categories, 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

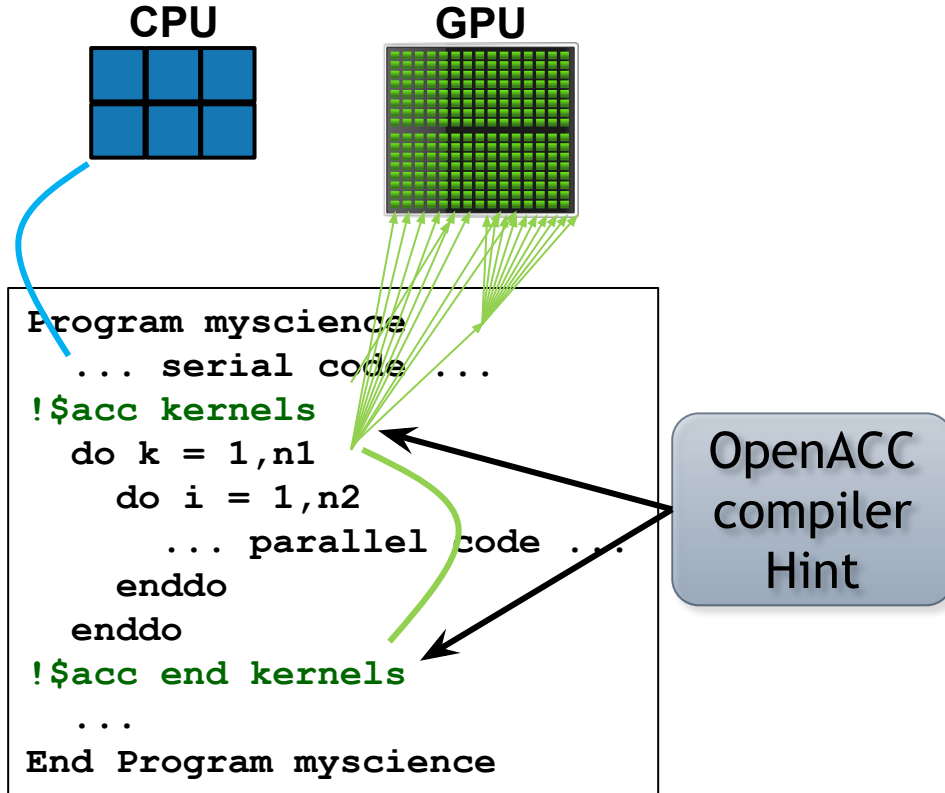
OpenACC  
Directives

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

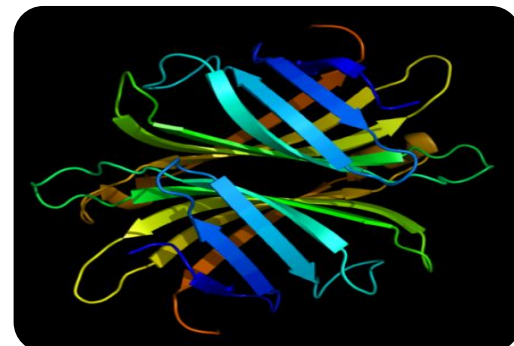
Global Technology Consulting  
Company



**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>

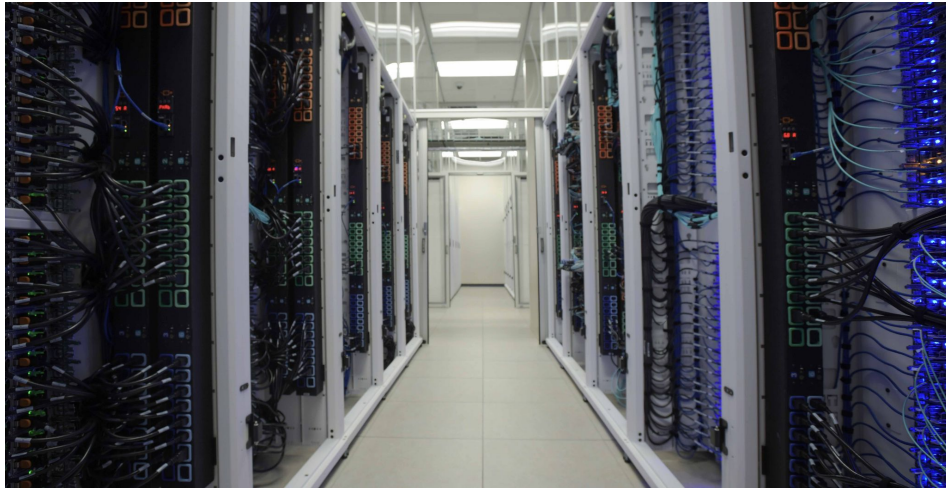
CUDA Fortran

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

Mathematica

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

# Part II. Running CUDA Code on Grace

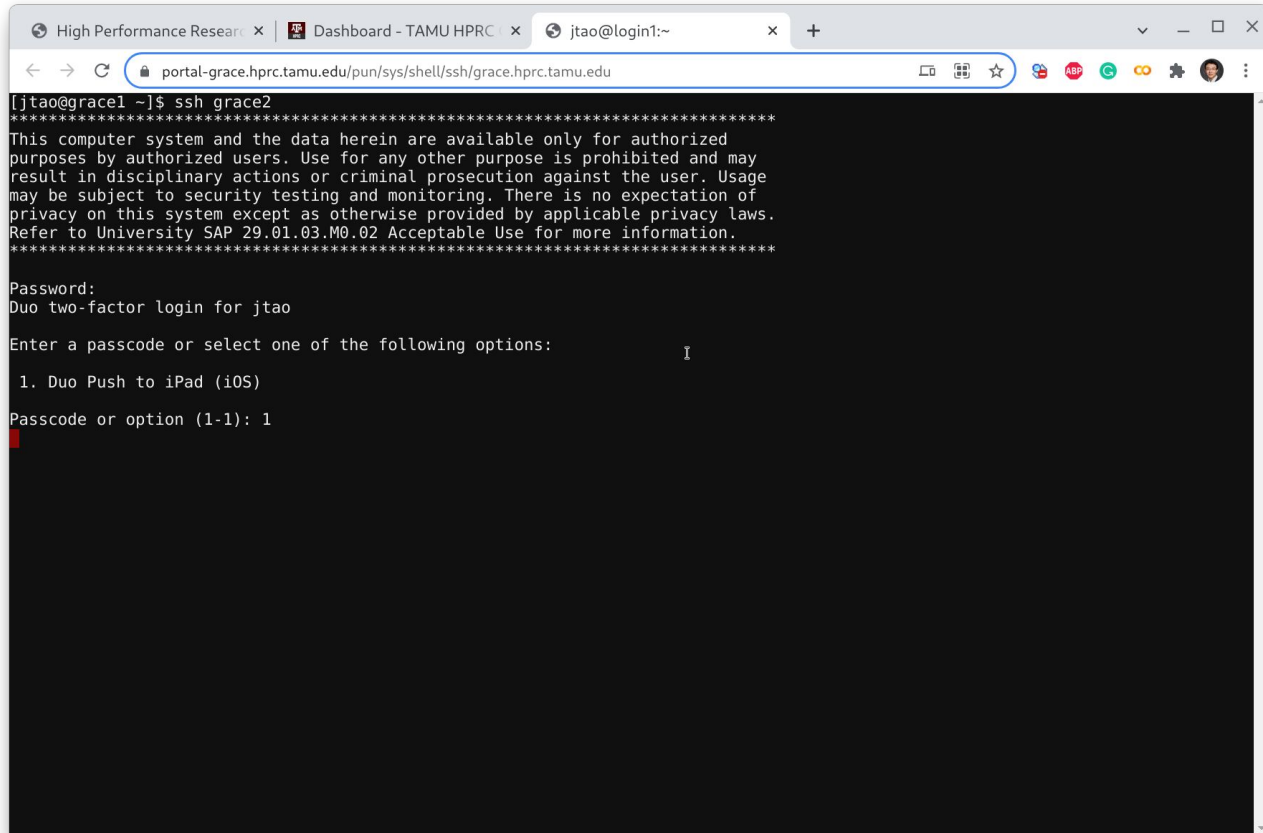


# Grace Login Nodes

	NVIDIA A100 GPU	NVIDIA RTX 6000 GPU	NVIDIA T4 GPU	No GPU
<b>Hostnames</b>	grace1.hprc.tamu.edu	grace2.hprc.tamu.edu	grace3.hprc.tamu.edu	grace4.hprc.tamu.edu grace5.hprc.tamu.edu
<b>Processor Type</b>	Intel Xeon 6248R 3.0GHz 24-core			
<b>Memory</b>	384GB DDR4 3200 MHz			
<b>Total Nodes</b>	1	1	1	2
<b>Cores/Node</b>	48			
<b>Interconnect</b>	Mellanox HDR 100 InfiniBand			
<b>Local Disk Space</b>	per node: two 480 GB SSD drives, 1.6 TB NVMe			



# Login to Other Nodes



```
[jtao@grace1 ~]$ ssh grace2
*****
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 disciplinary actions or criminal prosecution against the user. Usage
may be subject to security testing and monitoring. There is no expectation of
privacy on this system except as otherwise provided by applicable privacy laws.
Refer to University SAP 29.01.03.M0.02 Acceptable Use for more information.
*****

Password:
Duo two-factor login for jtao

Enter a passcode or select one of the following options:

1. Duo Push to iPad (iOS)

Passcode or option (1-1): 1
```

# nvidia-smi on Grace1

```
High Performance Research x | Dashboard - TAMU HPRC x | jtiao@login1:~ x +
portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu
[jtiao@grace1 ~]$ nvidia-smi
Thu Nov 11 04:56:24 2021
-----+-----
NVIDIA-SMI 495.29.05      Driver Version: 495.29.05      CUDA Version: 11.5
-----+-----
GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC
Fan  Temp   Perf      Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M.
                                           |              | GPU-Util  Compute M.
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
0   NVIDIA A100-PCI...    On          | 00000000:3B:00:0  Off |          0MiB / 40536MiB |          0%      Default
N/A  25C    P0       30W / 250W  |              |          0%      Disabled
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
Processes:
GPU  GI   CI      PID  Type  Process name                        GPU Memory
ID   ID   ID                                     Usage
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
No running processes found
-----+-----
[jtiao@grace1 ~]$
```

# nvidia-smi on Grace2

```
High Performance Research x Dashboard - TAMU HPRC x jtiao@login2:~ x +
portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu
[jtiao@grace2 ~]$ nvidia-smi
Thu Nov 11 04:55:04 2021
-----+-----+-----+-----+-----+-----+-----+-----+
NVIDIA-SMI 495.29.05      Driver Version: 495.29.05      CUDA Version: 11.5
-----+-----+-----+-----+-----+-----+-----+-----+
GPU  Name           Persistence-M  Bus-Id        Disp.A    Volatile Uncorr. ECC
Fan  Temp    Perf   Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute M.
                                           |              |   GPU-Util  Compute M.
-----+-----+-----+-----+-----+-----+-----+-----+
  0   Quadro RTX 6000      On          00000000:3B:00:00 Off      0MiB / 22698MiB |    0%      Default
N/A   22C     P8     12W / 250W |              |              |
                                           |              |   GPU-Util  Compute M.
-----+-----+-----+-----+-----+-----+-----+-----+
  1   Quadro RTX 6000      On          00000000:D8:00:00 Off      0MiB / 22698MiB |    0%      Default
N/A   23C     P8     13W / 250W |              |              |
                                           |              |   GPU-Util  Compute M.
-----+-----+-----+-----+-----+-----+-----+-----+

Processes:
GPU  GI   CI      PID  Type  Process name                      GPU Memory
ID  ID  ID                                     Usage
-----+-----+-----+-----+-----+-----+-----+
No running processes found
-----+-----+-----+-----+-----+-----+-----+-----+
[jtiao@grace2 ~]$
```

# nvidia-smi on Grace3

```
High Performance Research | Dashboard - TAMU HPRC | jtao@login3:~ | +
portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu

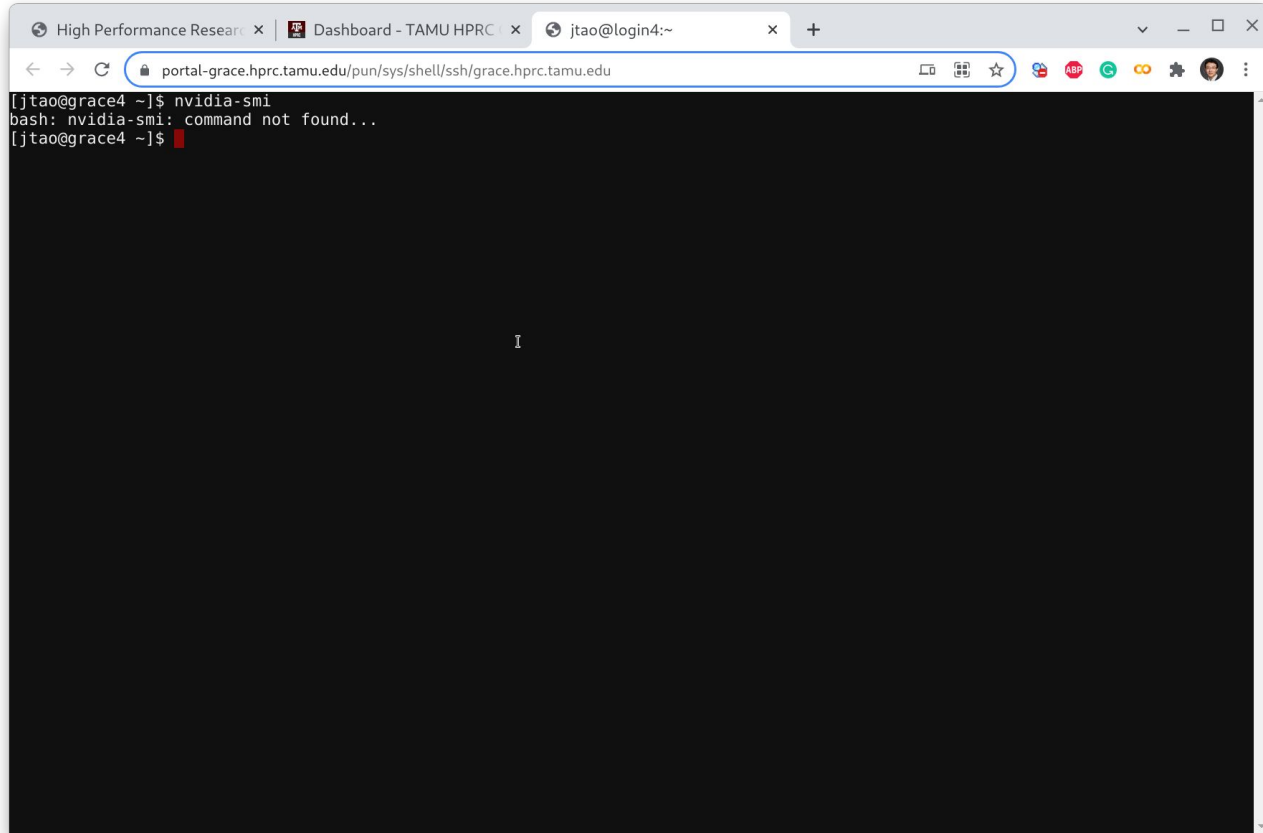
[jtao@grace3 ~]$ nvidia-smi
Thu Nov 11 04:59:20 2021

+-----+
| NVIDIA-SMI 495.29.05   | Driver Version: 495.29.05   | CUDA Version: 11.5   |
+-----+-----+
| 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 T4              On          | 00000000:3B:00:00 Off |   0%      Default  |
| N/A   26C            P8             11W /  70W |  0MiB / 15109MiB |           |    MIG M. |
+-----+-----+
|  1   Tesla T4              On          | 00000000:AF:00:00 Off |   0%      Default  |
| N/A   29C            P8             9W /  70W |  0MiB / 15109MiB |           |    MIG M. |
+-----+-----+

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

[jtao@grace3 ~]$
```

# nvidia-smi on Grace4/Grace5



The image shows a terminal window within a web browser. The browser's address bar displays the URL `portal-grace.hprc.tamu.edu/pun/sys/shell/ssh/grace.hprc.tamu.edu`. The terminal prompt is `[jtao@grace4 ~]$`. The user has entered the command `nvidia-smi`, which has resulted in the error message `bash: nvidia-smi: command not found...`. The terminal prompt is now `[jtao@grace4 ~]$` with a red cursor. The rest of the terminal area is black.

```
[jtao@grace4 ~]$ nvidia-smi
bash: nvidia-smi: command not found...
[jtao@grace4 ~]$
```

# Running CUDA Code on Grace

```
# load CUDA module
$m1 CUDA

# copy sample code to your scratch space
$cd $SCRATCH
$wget https://hprc.tamu.edu/files/training/2021/Fall/cuda.exercise.tgz
$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 grace_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 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)



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

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

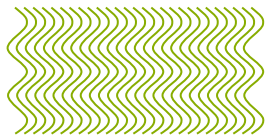
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel function

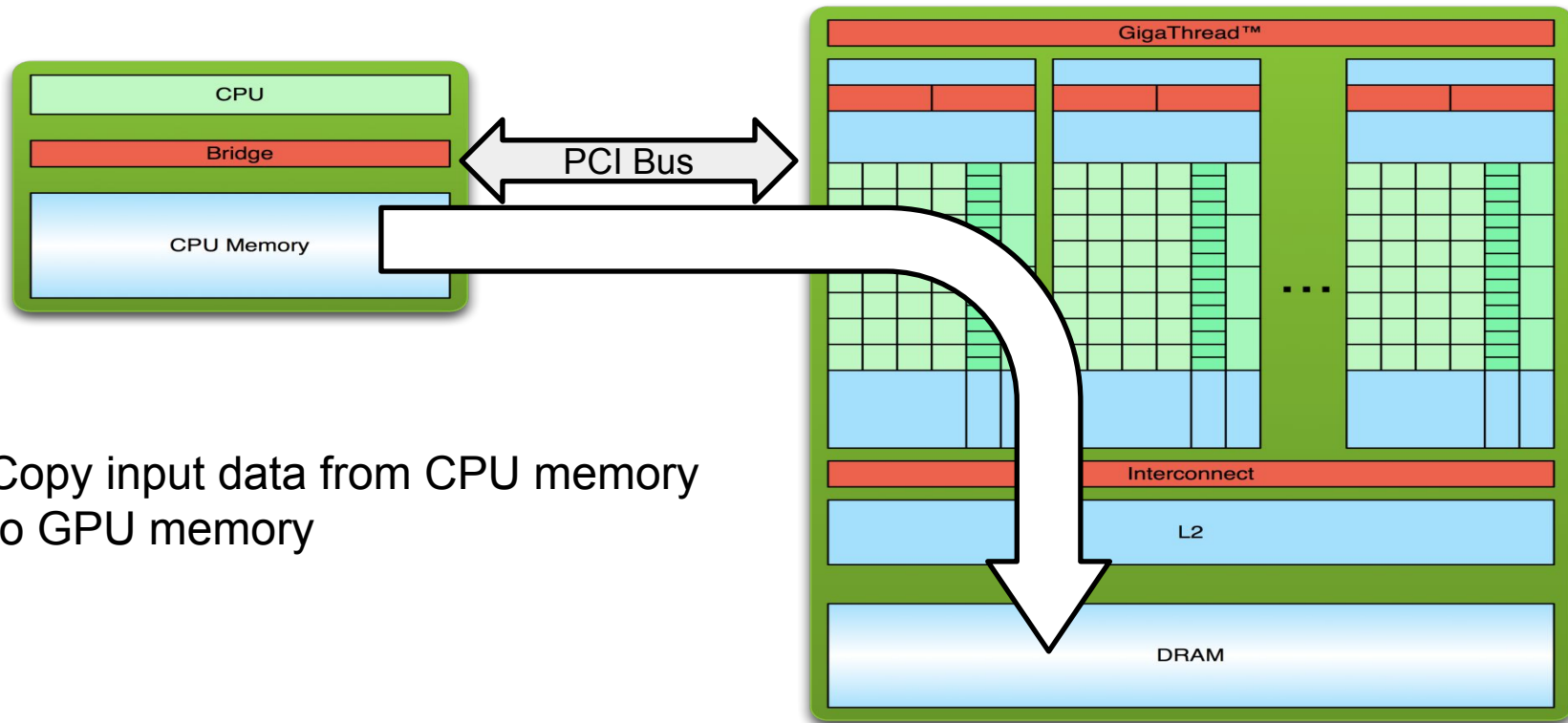
serial code

parallel  
code

serial code

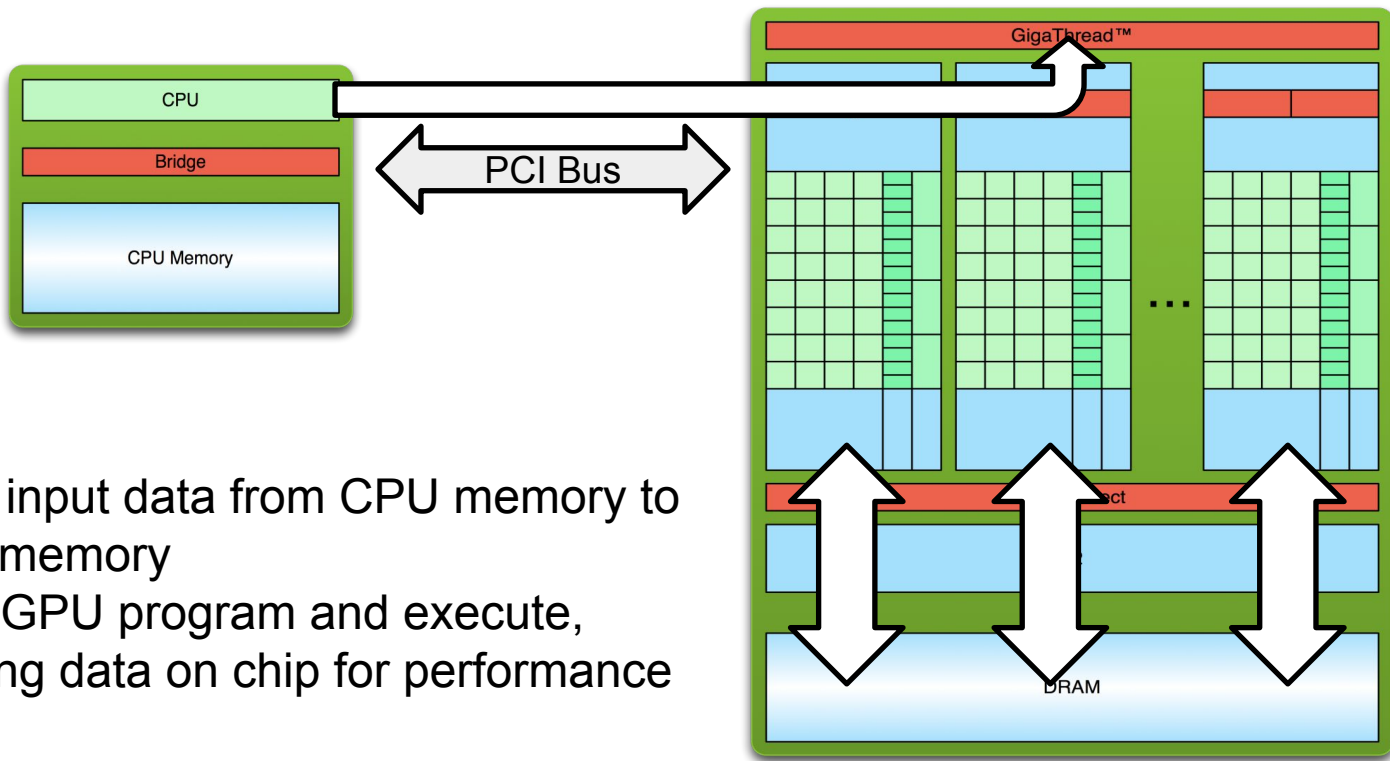


# Simple Processing Flow



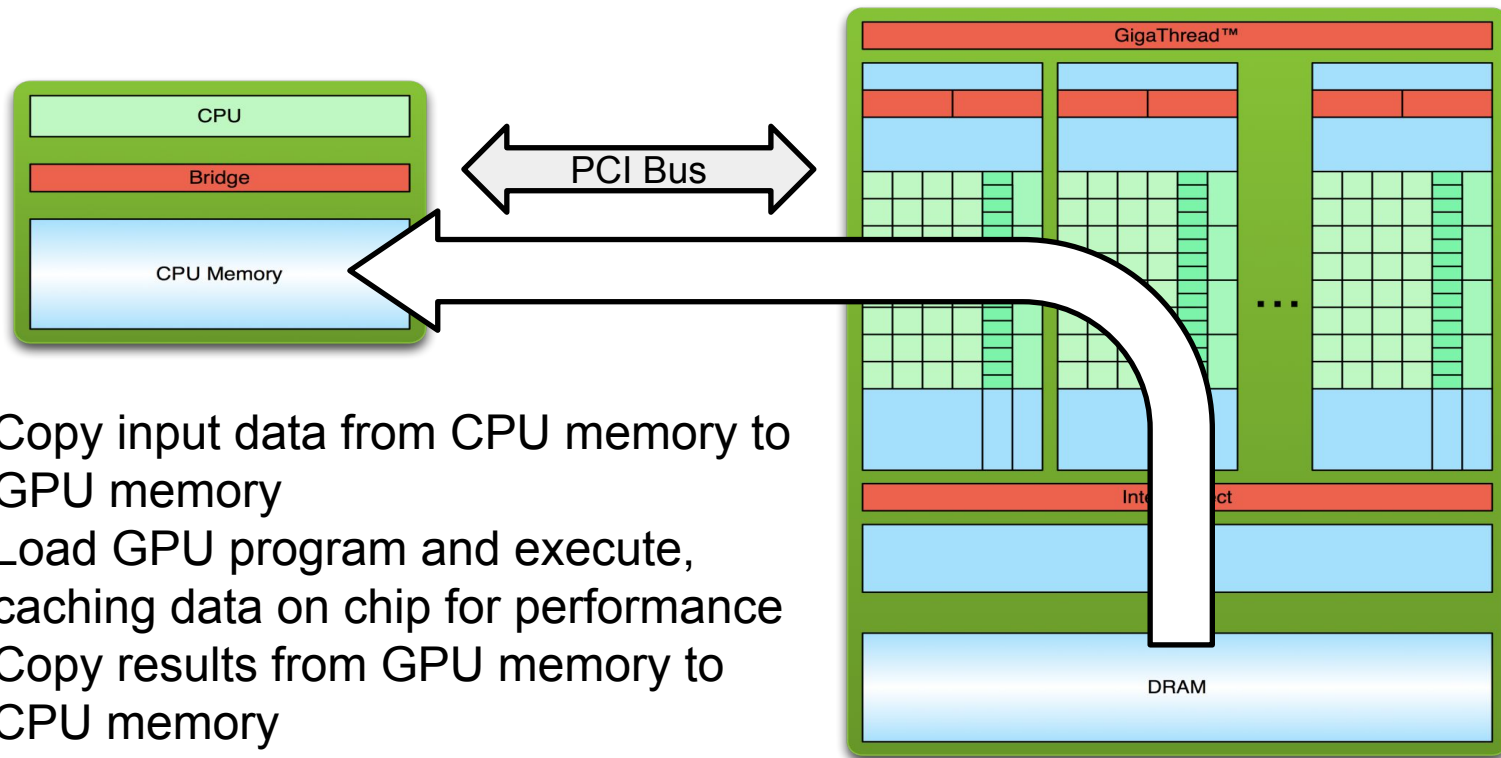
1. Copy input data from CPU memory to GPU memory

# 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

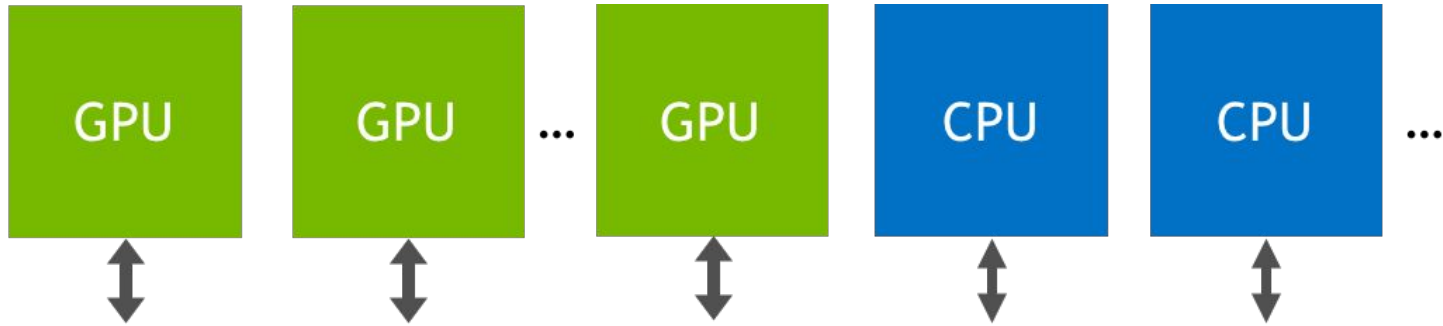


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

# 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;  
}
```

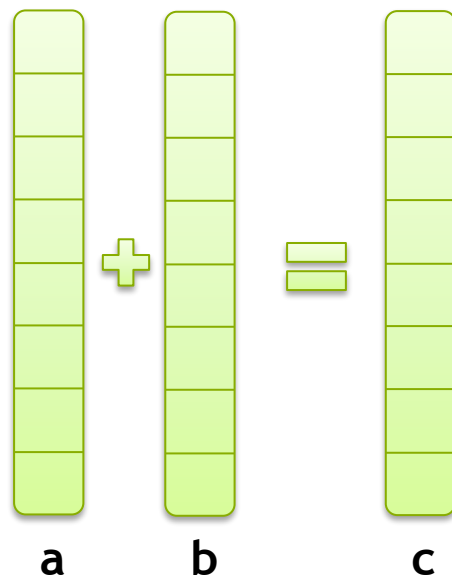
## Output:

```
$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

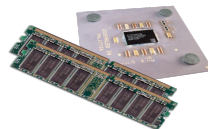
```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `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;
}
```

# 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

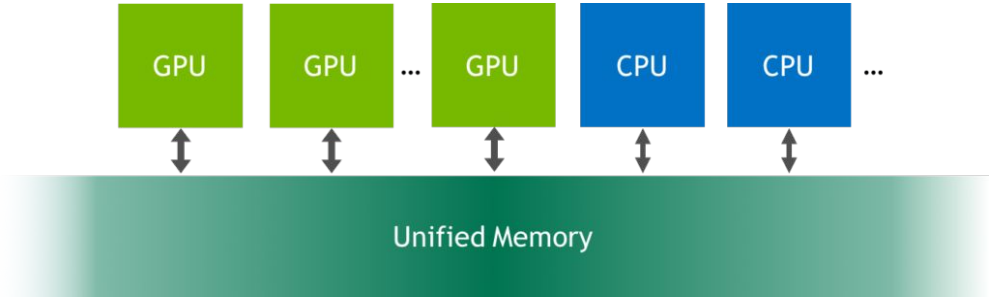
# Review (2 of 2)

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

# Unified Memory Programming

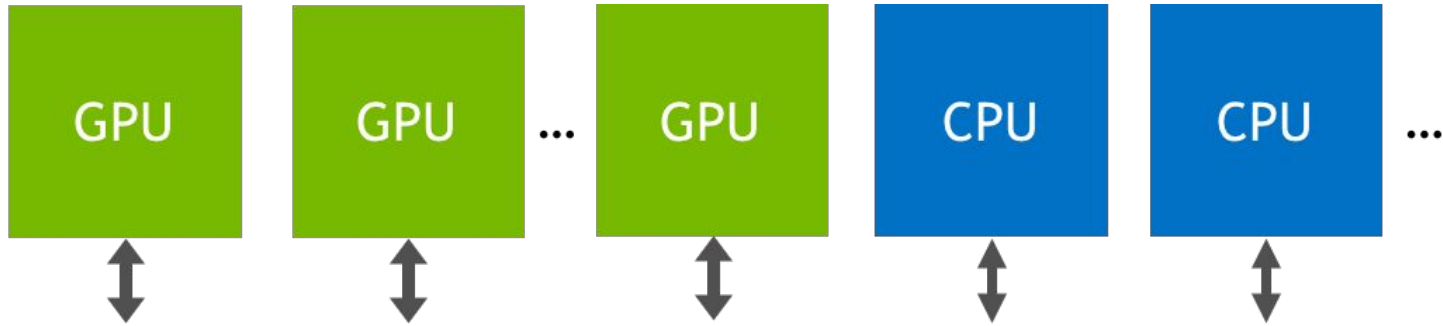




# Unified Memory

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)	
 <b>CUDA-Enabled NVIDIA GPUs</b>						
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](https://docs.nvidia.com/cuda))
- CUDA Zone – tools, training, etc.  
([developer.nvidia.com/cuda-zone](https://developer.nvidia.com/cuda-zone))
- Download CUDA Toolkit & SDK  
([www.nvidia.com/getcuda](https://www.nvidia.com/getcuda))
- Nsight IDE (Eclipse or Visual Studio)  
([www.nvidia.com/nsight](https://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).

# 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