

ACES: GPU Programming

Introduction to CUDA

Jian Tao

jtao@tamu.edu

Spring 2026 HPRC Short Course

3/17/2026



High Performance
Research Computing
DIVISION OF RESEARCH



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

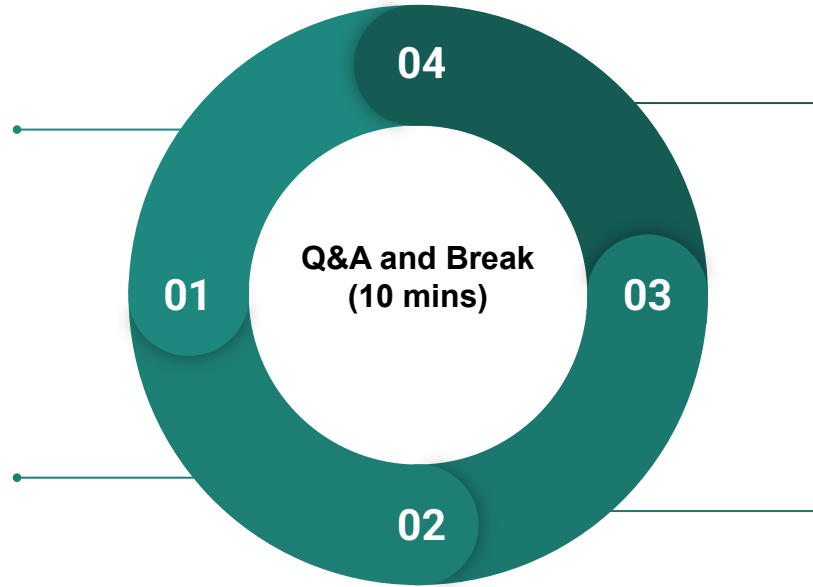
Introduction to CUDA Programming

Part I. Getting Started with ACES (~30 mins)

Part II. GPU as an Accelerator (~30 mins)

Part IV. CUDA C/C++ Basics (~50 mins)

Part III. Running CUDA Code on ACES (~30 mins)



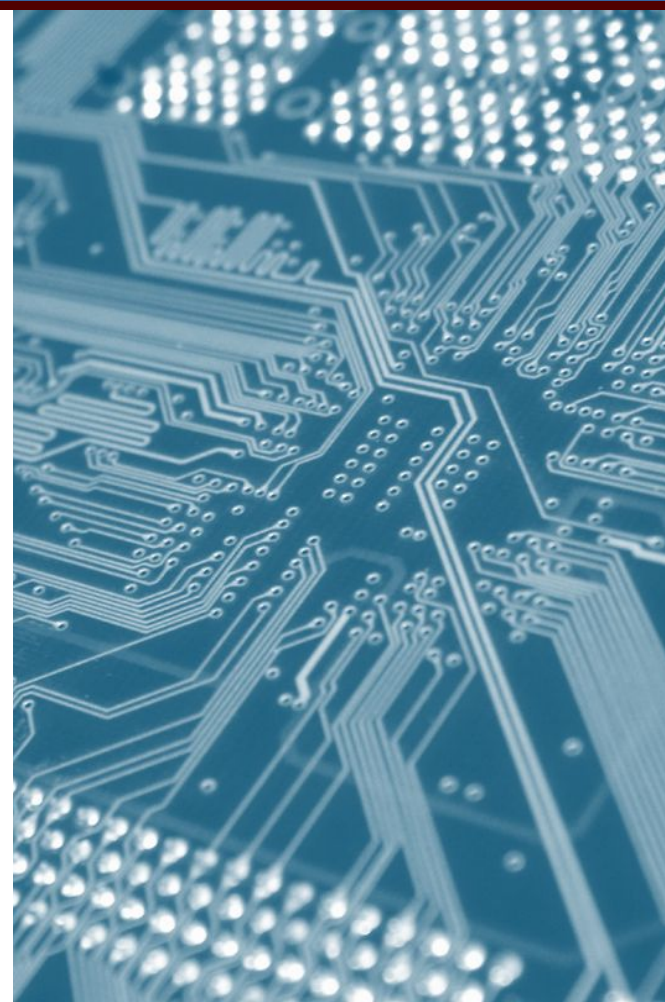
Part I. Get Started with ACES



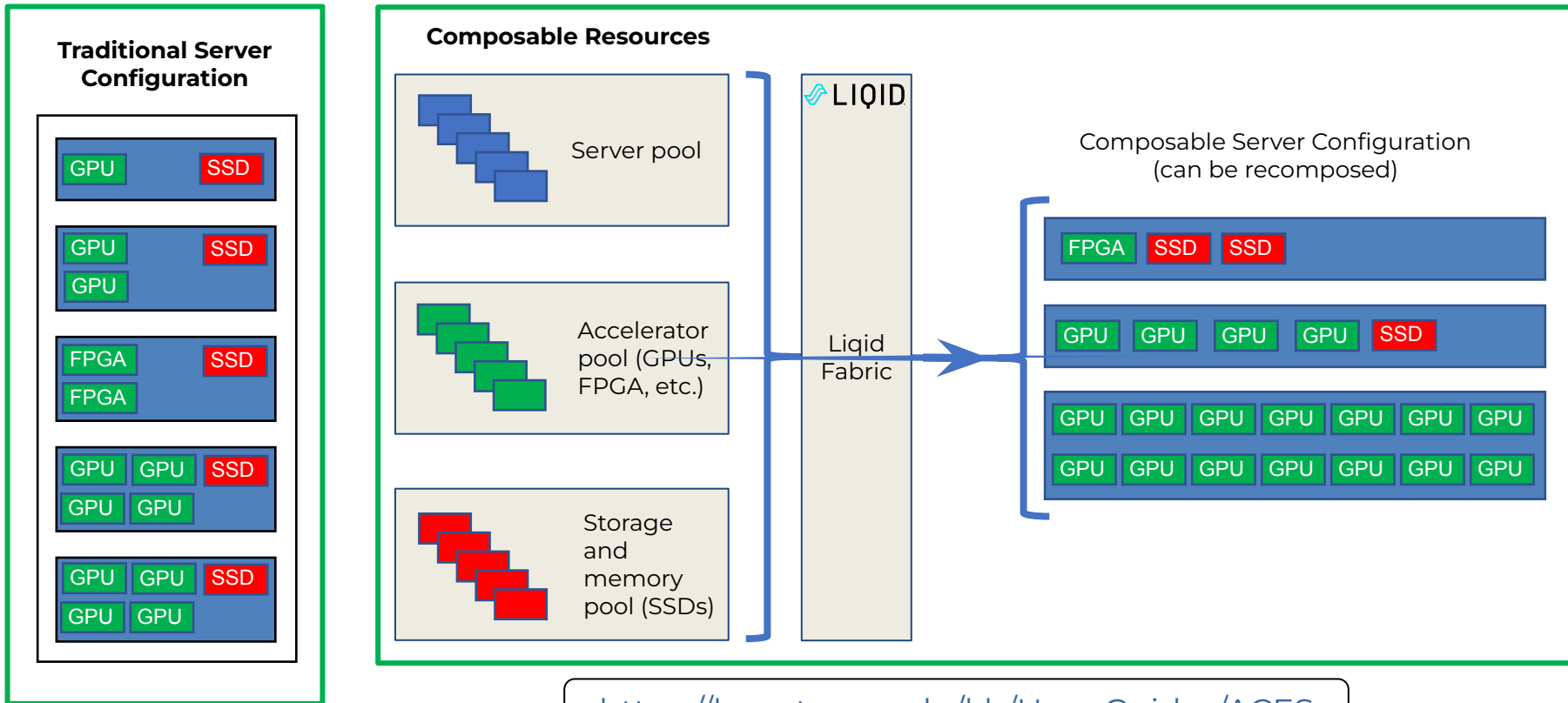
HPRC's Composable Clusters

- **FASTER** – First large-scale composable CPU/GPU system
- **ACES** – Composability for mixed-resource workflows

Focusing on ACES today

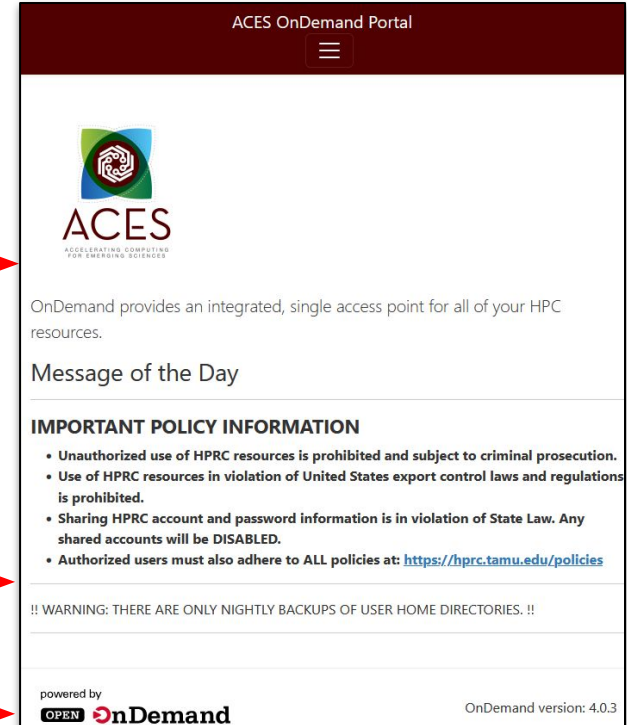
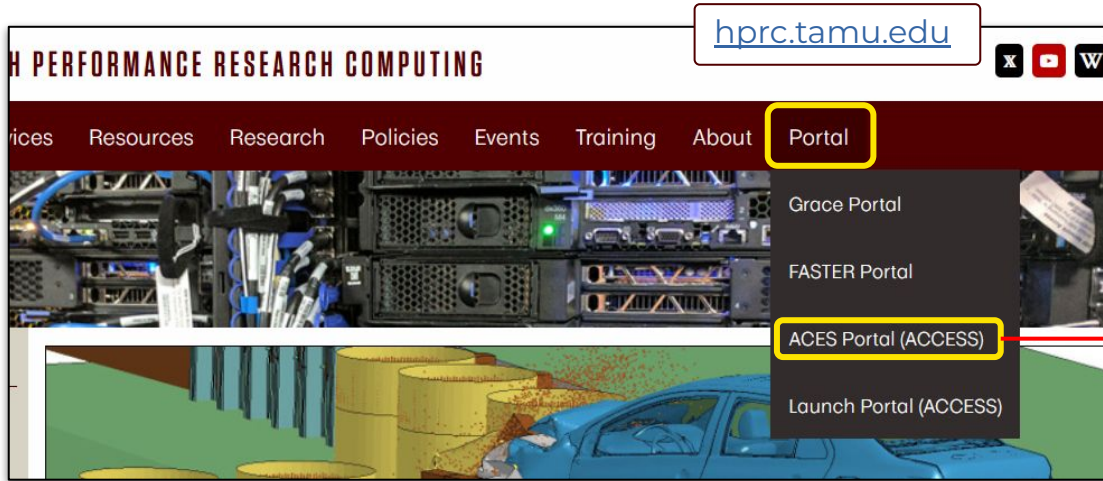


Composability



<https://hprc.tamu.edu/kb/User-Guides/ACES>

Login to ACES Portal

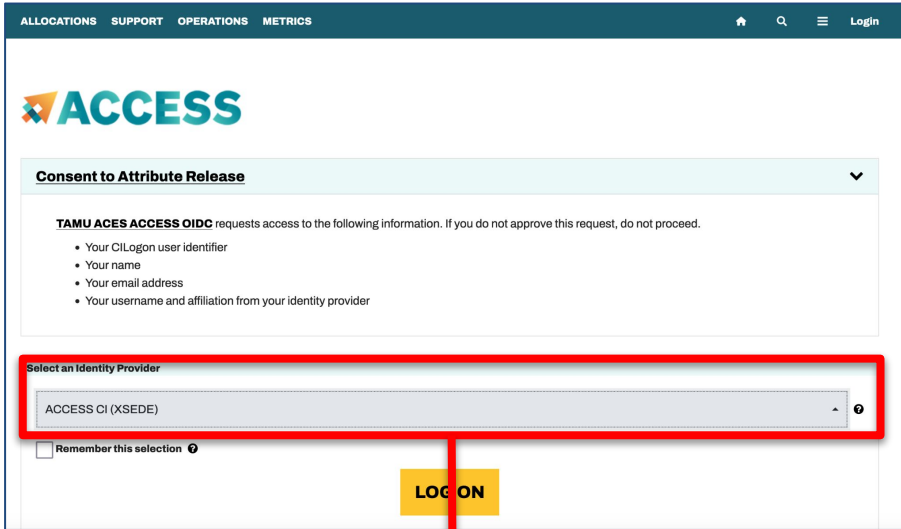


(You will have to log in with ACCESS if you haven't already)

ACES Portal portal-aces.hprc.tamu.edu
is the web-based user interface for the ACES cluster

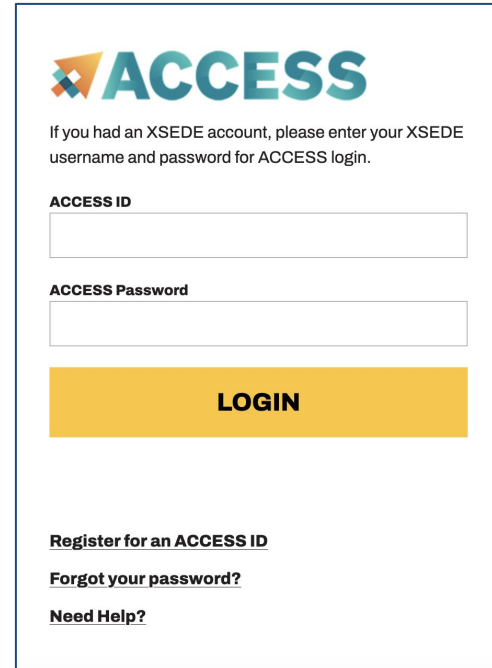
Open OnDemand (OOD) is an advanced web-based
graphical interface framework for HPC users

Accessing ACES via the Portal (ACCESS)



The screenshot shows the ACCESS portal interface. At the top, there is a navigation bar with links for ALLOCATIONS, SUPPORT, OPERATIONS, METRICS, and a Login button. Below the navigation bar is the ACCESS logo. A section titled "Consent to Attribute Release" contains a message from TAMU ACES ACCESS OIDC and a list of requested information: CILogon user identifier, name, email address, and affiliation. Below this is a "Select an Identity Provider" dropdown menu with "ACCESS CI (XSEDE)" selected. A red box highlights this dropdown menu. Below the dropdown is a "Remember this selection" checkbox and a yellow "LOG ON" button.

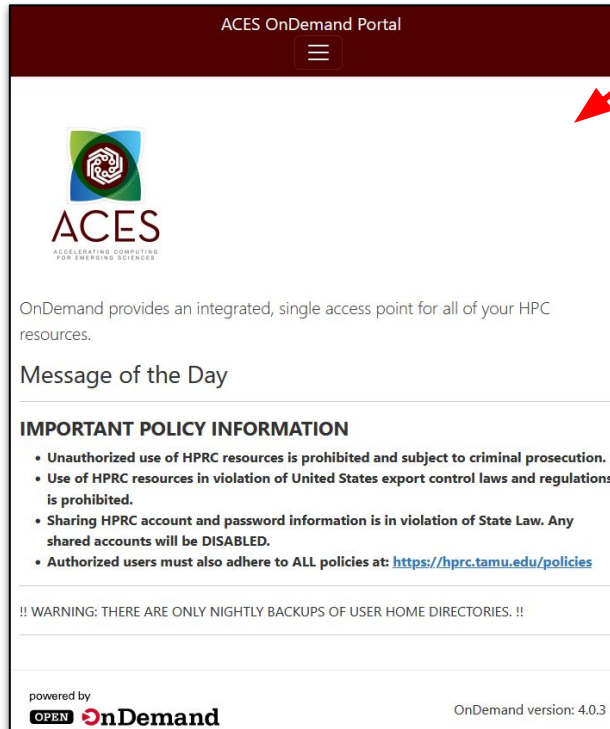
Select the Identity Provider appropriate for your account.



The screenshot shows the ACCESS login form. At the top is the ACCESS logo. Below it is a message: "If you had an XSEDE account, please enter your XSEDE username and password for ACCESS login." There are two input fields: "ACCESS ID" and "ACCESS Password". Below these fields is a yellow "LOGIN" button. At the bottom, there are links for "Register for an ACCESS ID", "Forgot your password?", and "Need Help?".

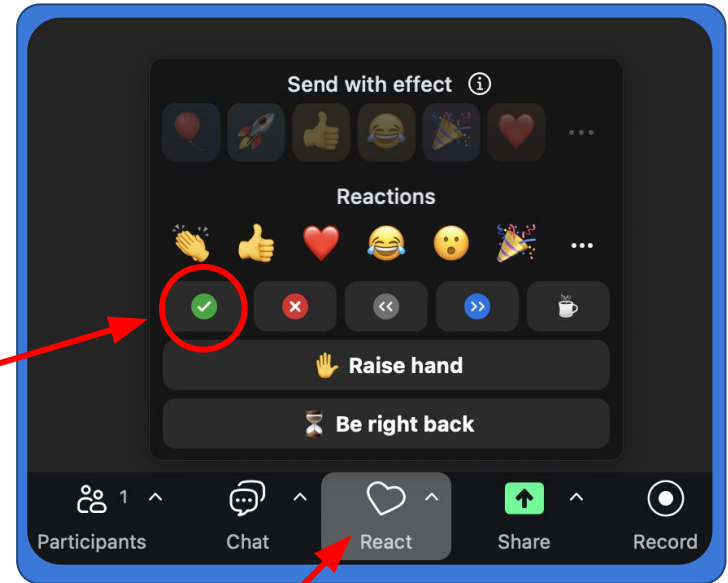
Log-in using your ACCESS or institutional credentials.

Login to ACES Portal



Once you're on this screen...

...Click the green checkmark reaction in Zoom



(find it here)

Shell Access via the Portal

ACES OnDemand Portal Files Jobs Clusters Interactive Apps Affinity Groups Dashboard

>_aces Shell Access

Get a shell terminal right in your browser

ACES

ACCELERATING COMPUTING FOR EMERGING SCIENCES

```
Host: login.aces Theme: Default
Warning: Permanently added 'login.aces,10.71.1.13' (ECDSA) to the list of known hosts.
*****
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.
*****

Last login: Mon Feb 12 13:11:13 2024 from 10.71.1.6

=====
Texas A&M University High Performance Research Computing

Website:          https://hprc.tamu.edu
Consulting:       help@hprc.tamu.edu (preferred) or (979) 845-0219
ACES Documentation: https://hprc.tamu.edu/kb/User-Guides/ACES
FASTER Documentation: https://hprc.tamu.edu/kb/User-Guides/FASTER
Grace Documentation: https://hprc.tamu.edu/kb/User-Guides/Grace
Terra Documentation: https://hprc.tamu.edu/kb/User-Guides/Terra
YouTube Channel:  https://www.youtube.com/texasamhprc
=====

*****
===== 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
*   Texas State Law. Any shared accounts will be DISABLED.
* - Authorized users must also adhere to ALL policies at:
*   https://hprc.tamu.edu/policies/
*
*****

*** ACES Partial Availability, February 12 ***

We are still troubleshooting issues for various compute nodes that were
reconfigured for PCIe fabric connectivity to the H100 and PVCs.

!! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!

Please restrict usage to 8 CORES across ALL login nodes.
Users found in violation of this policy will be SUSPENDED.

To see these messages again, run the motd command.
Your current disk quotas are:
Disk          Disk Usage    Limit   File Usage    Limit
/home/u..jw123527  16M          499     10000
/scratch/user/u..jw123527  28.1G       1.0T    102472  250000
Type 'showquota' to view these quotas again.
[u..jw123527@aces-login3 ~]$ !
```

ACES In Action



ACES System Description

Component	Quantity	Description
Sapphire Rapids Nodes: Compute Nodes Data Transfer Nodes Login & Management Nodes	110 nodes 2 nodes 5 nodes	96 cores per node, dual Intel Xeon 8468 processors 512 GB DDR5 memory 1.6 TB NVMe storage Compute: NVIDIA Mellanox NDR 200 Gbps InfiniBand adapter DTNs & Login & Management nodes: 100 Gbps Ethernet adapter
Ice Lake Login & Management Nodes	2 nodes	64 cores per node, dual Intel Xeon 8352Y processors 512 GB DDR4 memory 1.6 TB NVMe storage NVIDIA Mellanox NDR 200 Gbps InfiniBand adapter
PCIe Gen4 Composable Infrastructure	50 SPR nodes	Dynamically reconfigurable infrastructure that allows up to 20 PCIe cards (GPU, FPGA, etc.) per compute node
PCIe Gen5 Composable Infrastructure	60 SPR nodes	Dynamically reconfigurable infrastructure that allows up to 16 H100s or 14 PVCs per compute node
NVIDIA InfiniBand (IB) Interconnect	110 nodes	Two leaf and two spine switches in a 2:1 fat tree topology
DDN Lustre Storage	2.5 PB usable	HDR IB connected flash and disk storage for Lustre file systems

ACES Accelerators

Component	Quantity	Description
Graphcore IPU	32	16 Colossus GC200 IPU, 16 Bow IPU. Each IPU group hosted with a CPU server as a POD16 on a 100 GbE RoCE fabric
<i>FPGAs:</i> BittWare IA-840F	3	Accelerator with Agilex AGF027 FPGA and 64 GB of DDR4
NextSilicon Coprocessor	2	Reconfigurable accelerator with an optimizer continuously evaluating application behavior.
NEC Vector Engine	8	Vector computing card (8 cores and HBM2 memory)
Intel Optane SSD	48	18 TB of SSDs addressable as memory w/ MemVerge Memory Machine.
<i>NVIDIA GPUs:</i> H100	30	For HPC, DL Training, AI Inference
A30	4	For AI Inference and Mainstream Compute
Intel PVC GPUs	120	Intel GPUs for HPC, DL Training, AI Inference

Refer to our Knowledge Base for more:

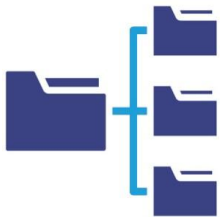
<https://hprc.tamu.edu/kb/User-Guides/ACES/Hardware/>

Accelerator Access Summary

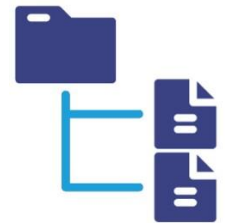
Component	Access	node or partition
BittWare IA-840F FPGA	Slurm	--partition=bittware
Intel GPU Max 1100 (PVC)	Slurm	--partition=pvc
Intel Optane SSD	Slurm	--partition=memverge
NextSilicon Coprocessor	Slurm	--partition=nextsilicon
NVIDIA A30 GPUs and H100 GPUs	Slurm	--partition=gpu --partition=gpu_debug
NEC Vector Engine	Slurm	--partition=nec
Kubernetes & Development Cluster	Interactive	upon request (pilot)
Graphcore Bow IPUs	Interactive	ssh poplar2
Graphcore Colossus IPUs	Interactive	ssh poplar1

File Systems and User Directories

Directory	Environment Variable	Space Limit	File Limit	Intended Use
/home/\$USER	\$HOME	10 GB	10,000	Small to modest amounts of processing. Backed up.
/scratch/user/\$USER	\$SCRATCH	1 TB	250,000	Temporary storage of large files for on-going computations. Not intended to be a long-term storage area. Not backed up.
/scratch/group/PROJECTID	\$PROJECT	5 TB	500,000	High performance storage for specific storage allocation requests. Not purged while allocation is active.



Do not share your Home or Scratch areas!
Use Project to collaborate and share files!



Job Scripts on ACES: Slurm

```
#!/bin/bash
#NECESSARY JOB SPECIFICATIONS
#SBATCH --job-name=my_job
#SBATCH --time=2-00:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=96
#SBATCH --mem=488G
#SBATCH --partition=gpu
#SBATCH --gres=gpu:h100:2
#SBATCH --output=stdout.%x.%j
#SBATCH --error=stderr.%x.%j

# load required module(s)
module purge
module load CUDA/12.3.2

./a.out
```

These parameters describe the resources needed for your program to the job scheduler (Slurm)

Most of the ACES accelerators will be specified with either a partition or gres argument

Script to execute
(In this case, set up environment and launch an executable)

Part II. GPU as an Accelerator



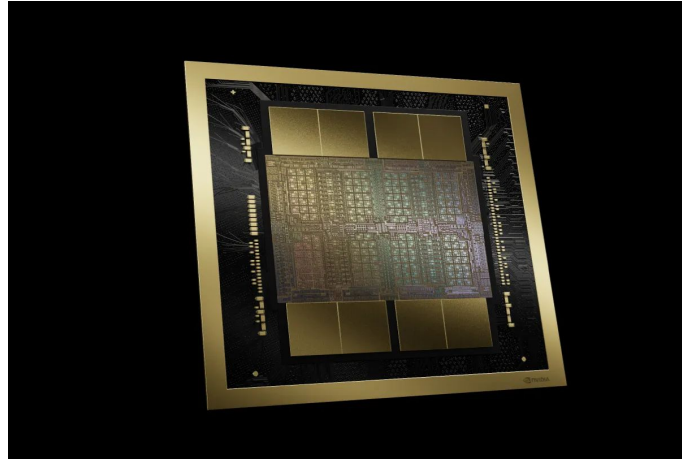
CPU



GPU Accelerator

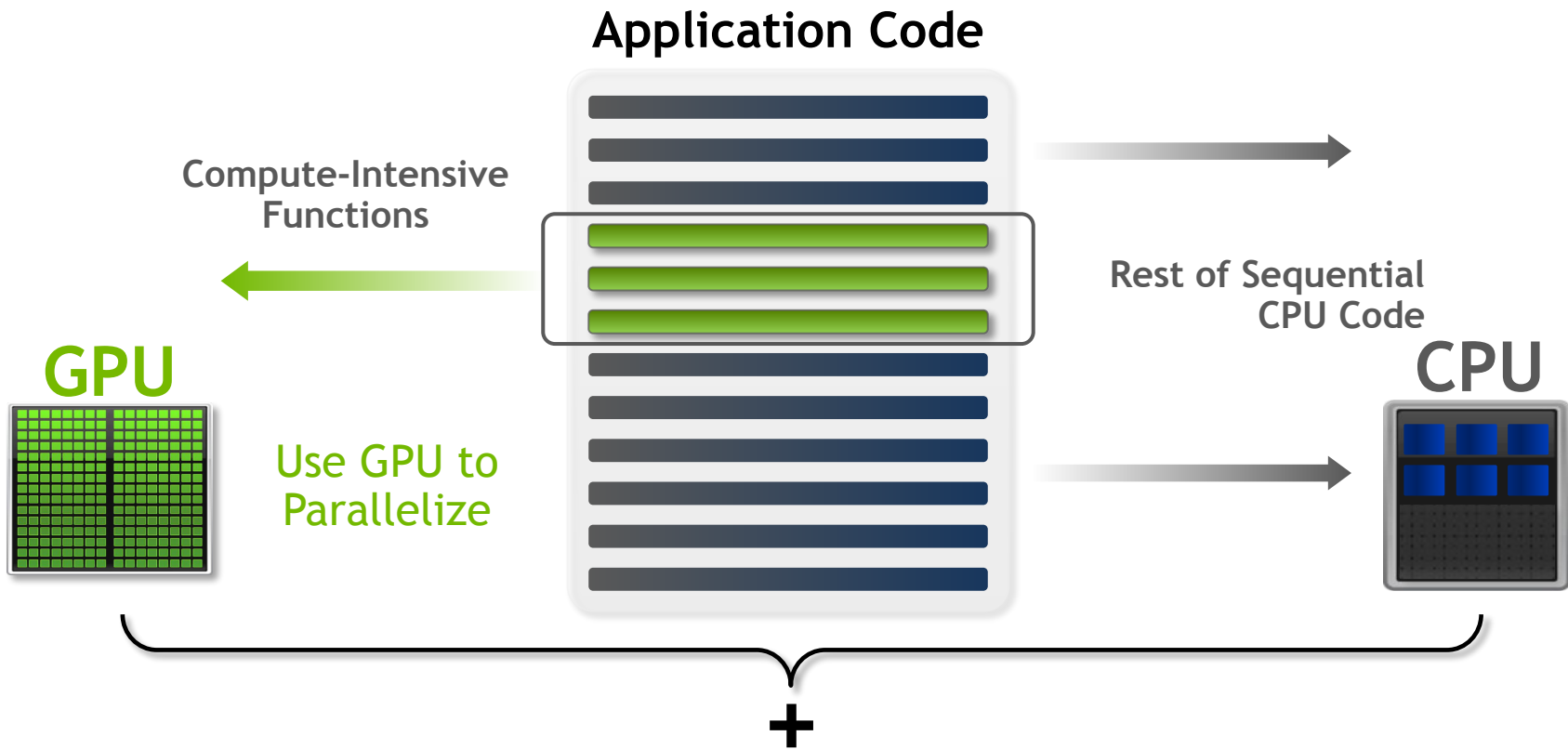


NVIDIA B200 with 208 Billion Transistors



Announced and released in early 2025 was the Blackwell-based B200 accelerator. Built on TSMC's 4NP process, the B200 features 208 billion transistors and delivers up to 90 teraflops of FP64 performance, 20 petaflops of AI inference with FP4 precision, and incorporates NVIDIA's 6th-generation Tensor Cores. It includes 192GB of HBM3e memory with an impressive 8TB/s memory bandwidth.

Add GPUs: Accelerate Science Applications



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

CUDA Math Libraries

GPU-accelerated math libraries lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational fluid dynamics, computational chemistry, medical imaging, and seismic exploration.



cuBLAS

GPU-accelerated basic linear algebra (BLAS) library.

[Learn More >](#)



cuFFT

GPU-accelerated library for Fast Fourier Transform implementations.

[Learn More >](#)



cuRAND

GPU-accelerated random number generation.

[Learn More >](#)



cuSOLVER

GPU-accelerated dense and sparse direct solvers.

[Learn More >](#)



cuSPARSE

GPU-accelerated BLAS for sparse matrices.

[Learn More >](#)



cuTENSOR

GPU-accelerated tensor linear algebra library.

[Learn More >](#)



cuDSS

GPU-accelerated direct sparse solver library.

[Learn More >](#)



CUDA Math API

GPU-accelerated standard mathematical function APIs.

[Learn More >](#)



AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods.

[Learn More >](#)

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.

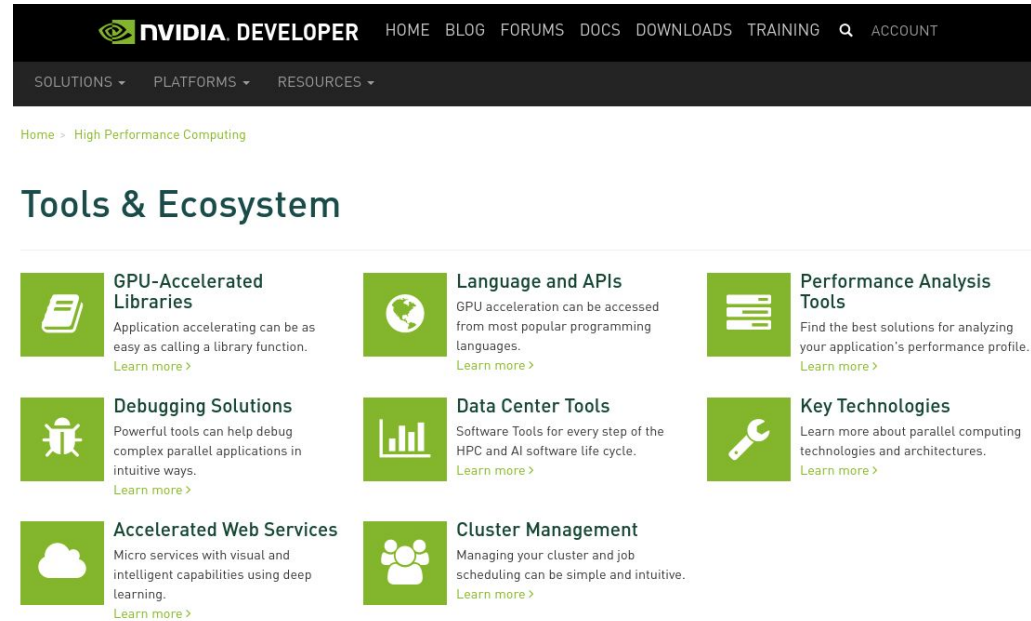
- with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.

- **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, and ACCOUNT. Below the navigation bar, there are dropdown menus for SOLUTIONS, PLATFORMS, and RESOURCES. The main content area is titled 'Tools & Ecosystem' and features a grid of nine cards, each representing a different tool or ecosystem component. Each card includes 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

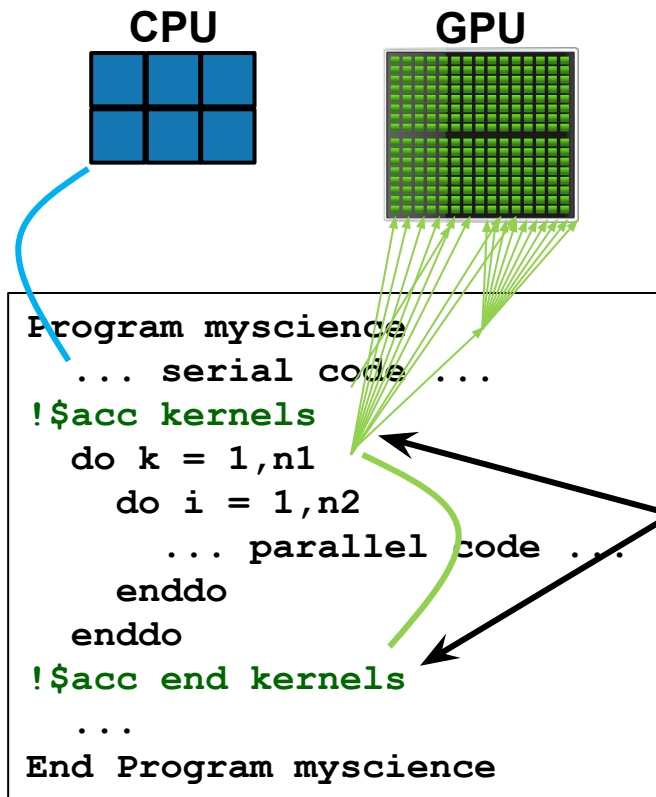
OpenACC
Directives

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

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

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>

CuPy (Python)

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

Thrust C++ Template Library

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

MATLAB

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

CUDA Fortran

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

Mathematica

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

Part III. Running CUDA Code on ACES



Running CUDA Code on ACES

```
# load CUDA module
$m1 CUDA/12.3.2

# copy sample code to your scratch space
cp /scratch/training/cuda/cuda.exercise.tgz $SCRATCH
cd $SCRATCH
$tar -zxvf cuda.exercise.tgz

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

# edit job script & submit your GPU job
$SBATCH aces_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 13.0 Update 2 (as of Nov 2025).

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

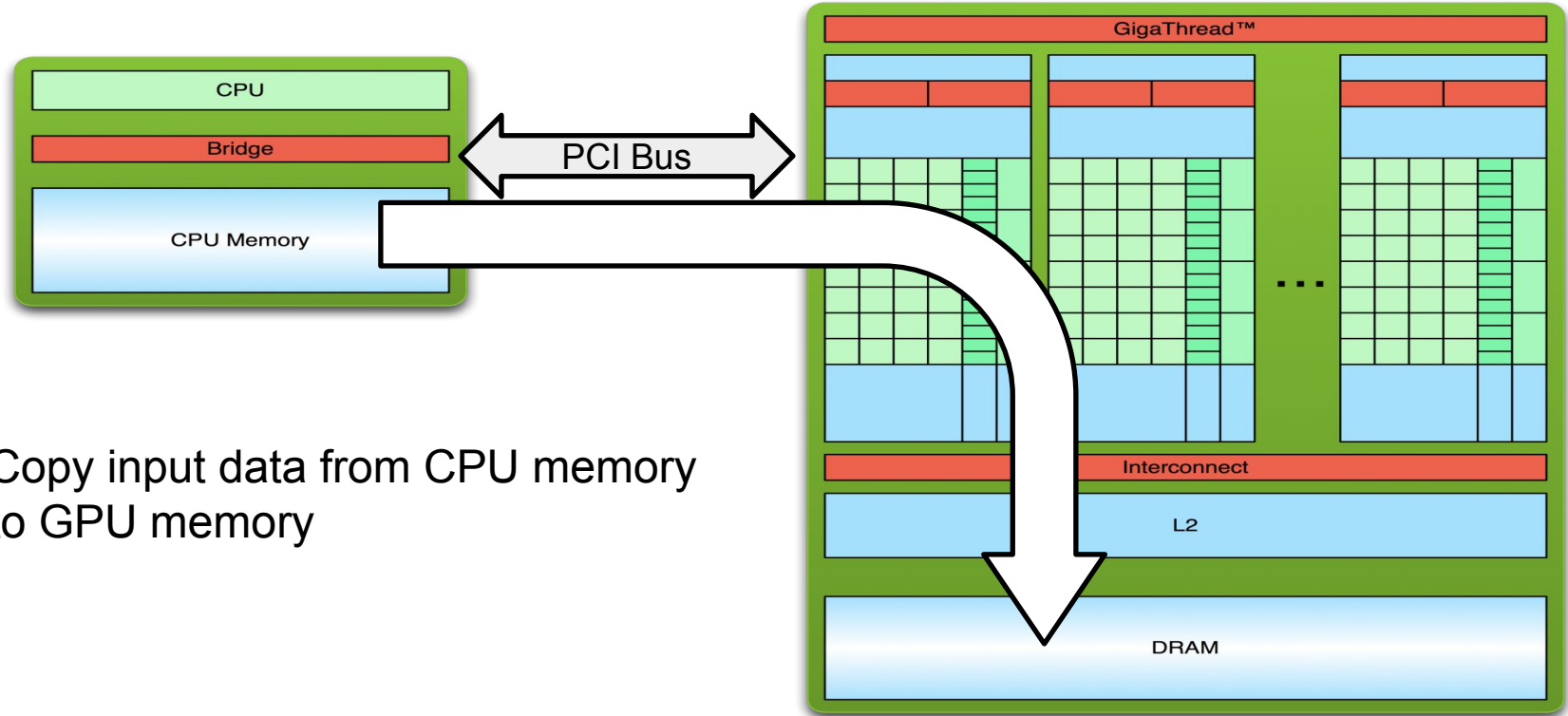
serial code

parallel code

serial code

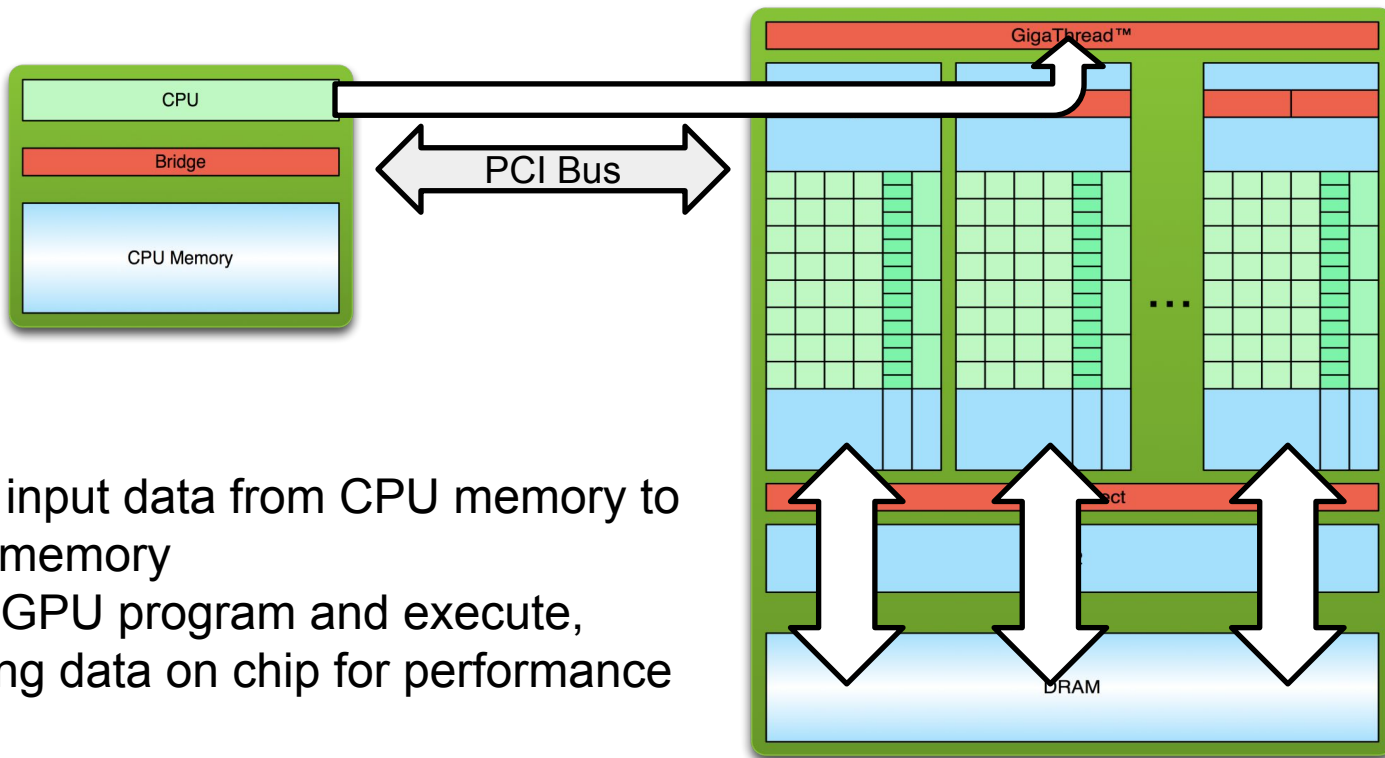


Simple Processing Flow



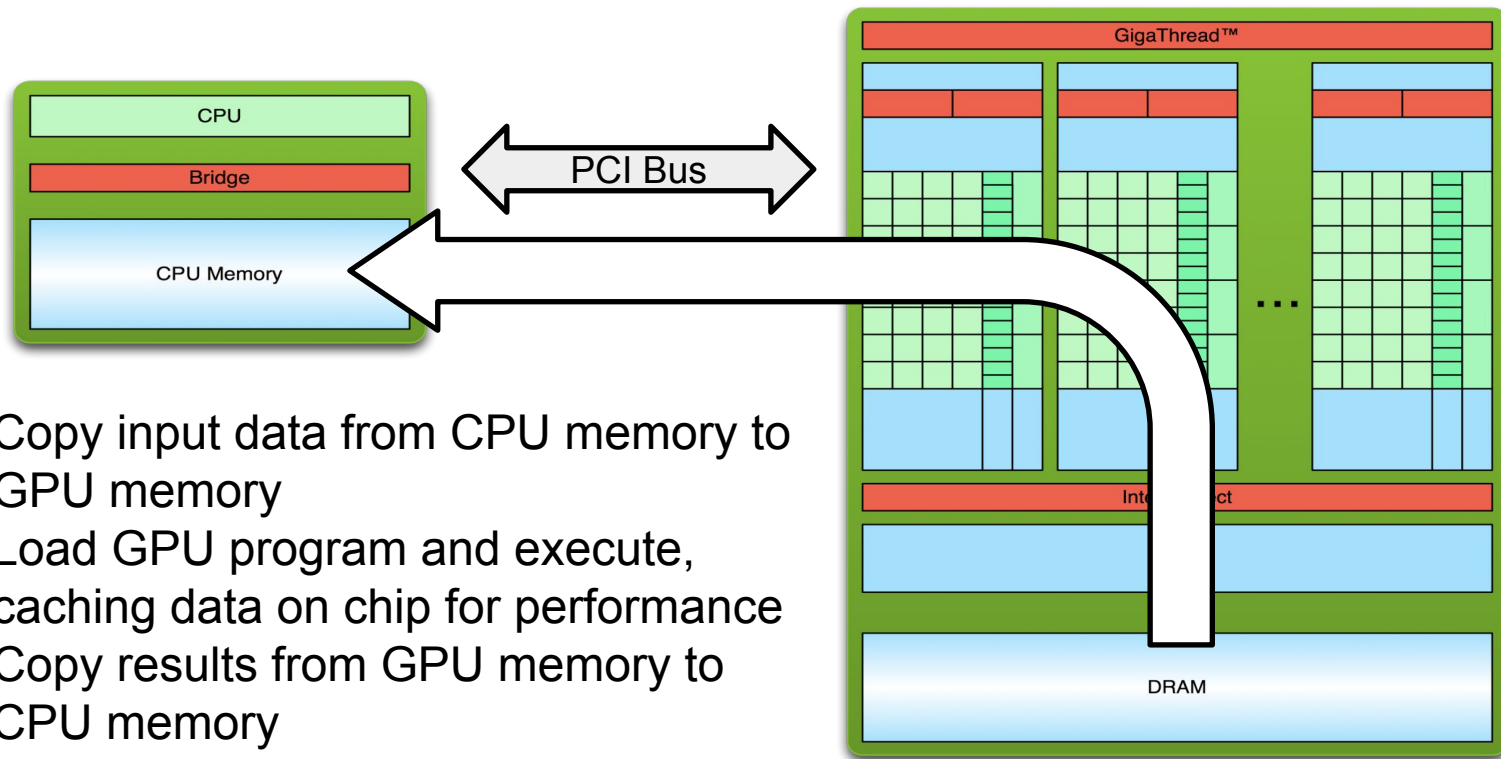
1. Copy input data from CPU memory to GPU memory

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

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;  
}
```

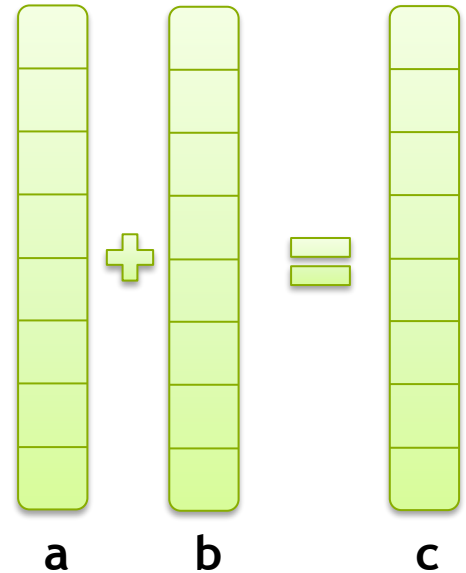
Output:

```
$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

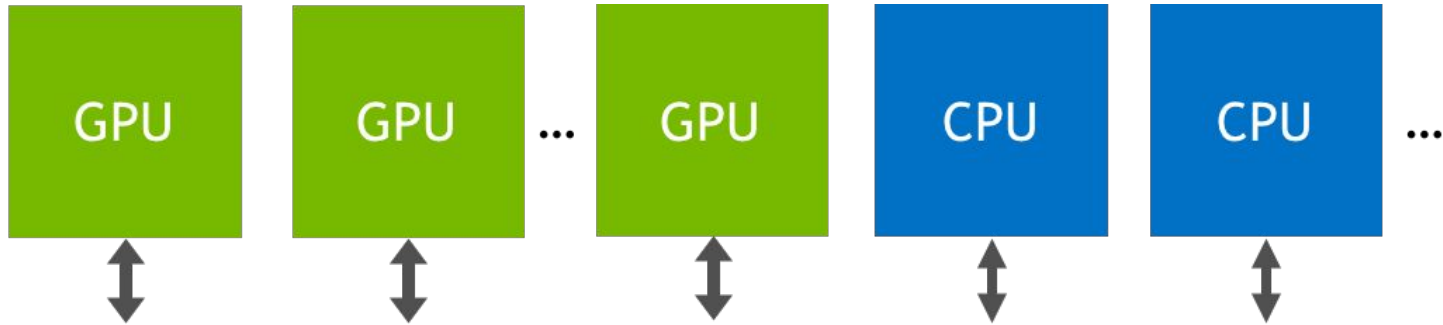
- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API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



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.

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;
}
```

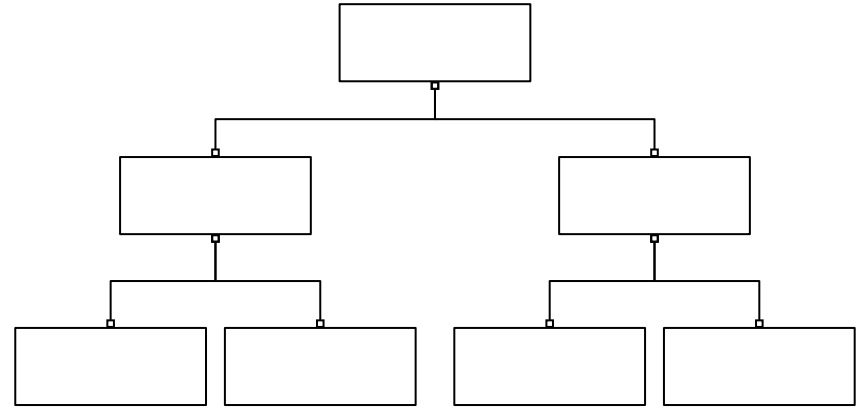
Vector Addition with Managed Global Memory

```
__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;
}
```

Hierarchy of Threads



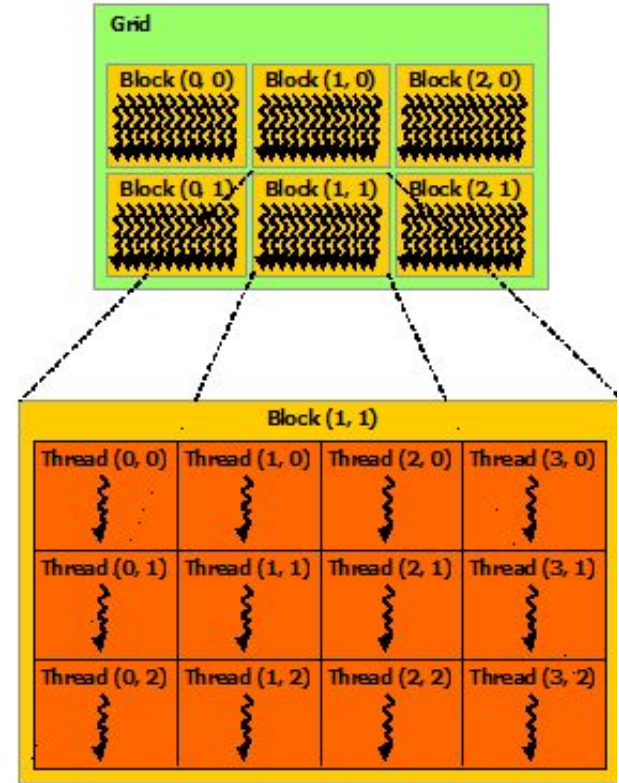
Key Programming Abstractions

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**

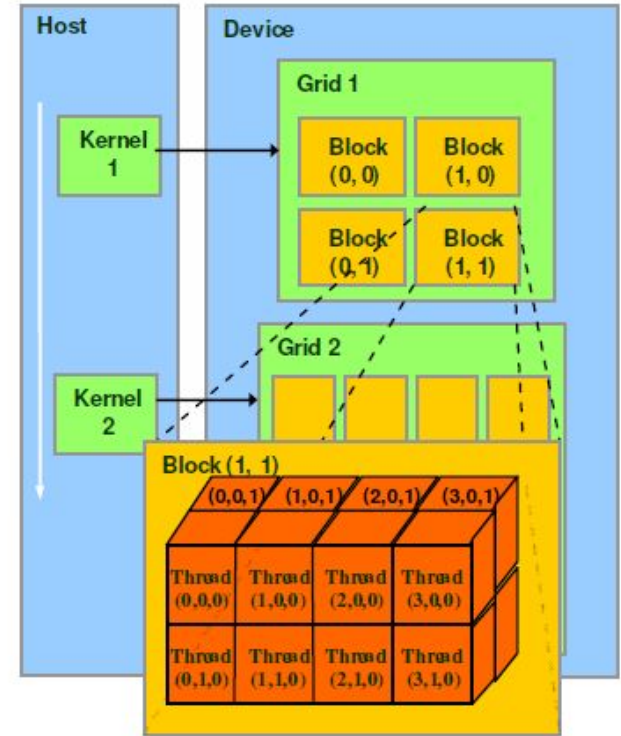
Glossary

- **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.



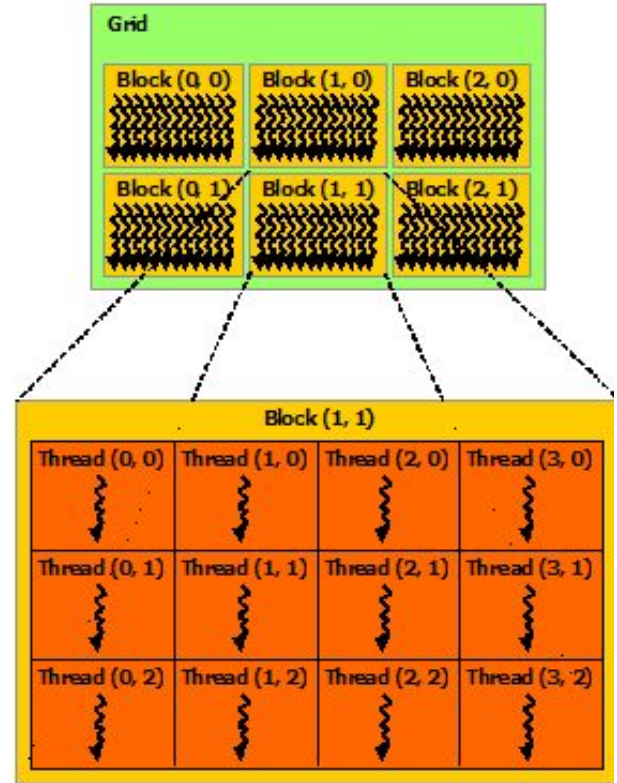
Thread Hierarchy - I

- 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**

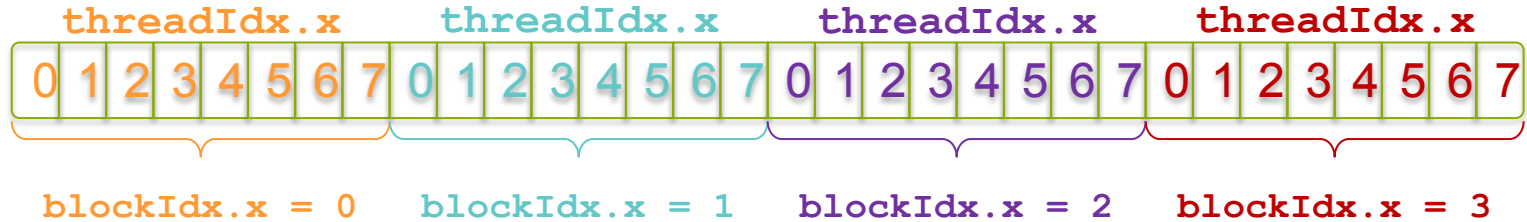
for a two-dimensional block of size $(\text{blockDim.x}, \text{blockDim.y})$, the thread ID of a thread of index (x, y) is $(x + y * \text{blockDim.x})$

- **3D**

for a three-dimensional block of size $(\text{blockDim.x}, \text{blockDim.y}, \text{blockDim.z})$, the thread ID of a thread of index (x, y, z) is $(x + y * \text{blockDim.x} + z * \text{blockDim.x} * \text{blockDim.y})$

Indexing Arrays with Blocks and Threads

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

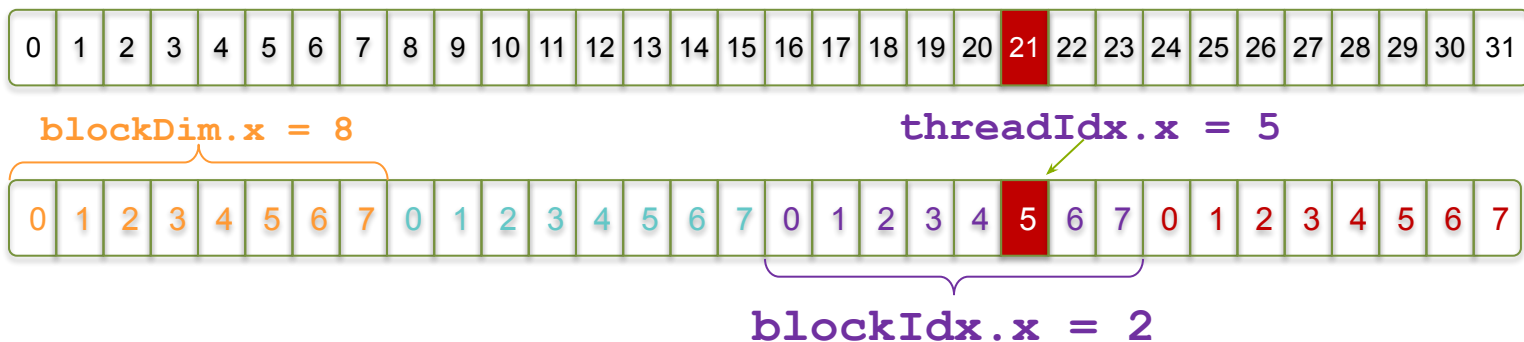


- With `blockDim.x` threads/block, the thread is given by:

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

Indexing Arrays: Example

- Which thread will operate on the red element?



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

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 **shared memory**
- by synchronizing their execution to coordinate memory accesses with `__syncthreads ()`

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(...)`

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)

A more detailed version of this slidedeck can be found at the link below

[Texas A&M Advanced GPU Programming Workshop - Day 1](#)

Acknowledgments

- Educational materials from [NVIDIA Deep Learning Institute via](#) its University Ambassador Program.
- Support from [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

HPRC Survey

https://u.tamu.edu/hprc_shortcourse_survey



HPRC Survey

[HPRC Survey](https://u.tamu.edu/hprc_shortcourse_survey)