ACES: GPU Programming

Introduction to CUDA

Jian Tao
jtao@tamu.edu
Spring 2024 HPRC Short Course
02/20/2024

TEXAS A&M UNIVERSITY
School of Performance, Visualization & Fine Arts

High Performance Research Computing
DIVISION OF RESEARCH

TEXAS A&M Institute of Data Science
Introduction to CUDA Programming

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

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

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

Q&A and Break (10 mins)

Part IV. CUDA C/C++ Basics (~50 mins)
Part I. Getting Started with ACES

TAMU HPRC Short Course: Getting Started with FASTER and ACES
Composable HPC Architectures for AI

**Common HPC**
- Built on Converged Hardware
- Static Hardware Design
- Fixed GPU/Accelerator
- Fixed Memory
- Storage: SATA and SAS
- Vendor Lock

**HPC for AI**
- Built on Disaggregated Hardware
- Composable Hardware Platform
- Composable GPU/Accelerator
- Composable Memory - Optane
- Modern Storage: NVMe-oF
- Open Platform

Next Generation HPC/AI Platform Supports Composable Accelerators and Memory
Common HPC System

Programming Models: MPI + (CUDA, OpenCL, OpenMP, OpenACC, etc.)
Composable HPC for AI

Traditional Server Configuration

- GPU
- SSD
- GPU
- SSD
- FPGA
- SSD
- GPU
- GPU
- SSD
- GPU
- SSD

Composable Resources

- Server pool
- Accelerator pool (GPUs, FPGA, etc.)
- Storage and memory pool (SSDs)

Composable Server Configuration (can be recomposed)

- FPGA
- SSD
- SSD
- GPU
- GPU
- GPU
- GPU
- SSD
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU
- GPU

https://hprc.tamu.edu/kb/User-Guides/ACES
NSF ACES
Accelerating Computing for Emerging Sciences

Our Mission:
- NSF ACSS CI test-bed
- Offer an accelerator testbed for numerical simulations and AI/ML workloads
- Provide consulting, technical guidance, and training to researchers
- Collaborate on computational and data-enabled research.
# ACES System Description

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-centric computing with variable memory requirements</td>
<td>Dual Intel Sapphire Rapids 2.1 GHz 96 cores per node, 512 GB memory, 1.6 TB NVMe storage (PCIe 5.0), NVIDIA Mellanox NDR 200 Gbps InfiniBand</td>
</tr>
<tr>
<td>Composable infrastructure</td>
<td>Reconfigurable infrastructure that allows up to 20 PCIe cards (GPU, FPGA, VE, etc.) per compute node</td>
</tr>
<tr>
<td>Data transfer nodes</td>
<td>100 Gbps network adapter</td>
</tr>
</tbody>
</table>
# ACES Accelerators

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphcore IPU</td>
<td>32</td>
<td>16 Colossus GC200 IPUs; 16 Bow IPUs. Each IPU group hosted with a CPU server as a POD16 on a 100 GbE RoCE fabric</td>
</tr>
<tr>
<td>Intel PAC D5005 FPGA</td>
<td>2</td>
<td>Accelerator with Intel Stratix 10 GX FPGA and 32 GB DDR4</td>
</tr>
<tr>
<td>BittWare IA-840F FPGA</td>
<td>2</td>
<td>Accelerator with Agilex AGF027 FPGA and 64 GB of DDR4</td>
</tr>
<tr>
<td>NextSilicon Coprocessor</td>
<td>2</td>
<td>Reconfigurable accelerator with an optimizer continuously evaluating application behavior.</td>
</tr>
<tr>
<td>NEC Vector Engine</td>
<td>8</td>
<td>Vector computing card (8 cores and HBM2 memory)</td>
</tr>
<tr>
<td>Intel Optane SSD</td>
<td>48</td>
<td>18 TB of Intel Optane SSDs addressable as memory w/ MemVerge Memory Machine.</td>
</tr>
<tr>
<td>NVIDIA H100 + A30</td>
<td>30 + 4</td>
<td>NVIDIA GPUs for HPC, DL Training, AI Inference</td>
</tr>
<tr>
<td>Intel GPU Max 1100 (PVC)</td>
<td>120</td>
<td>Intel GPUs for HPC, DL Training, AI Inference</td>
</tr>
</tbody>
</table>
# Research Workflows - Accelerators

<table>
<thead>
<tr>
<th>Hardware Profile</th>
<th>Applications Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEC Vector Engines</td>
<td>• AI/ML (Statistical Machine Learning, Data Frame)</td>
</tr>
<tr>
<td></td>
<td>• Chemistry (VASP, Quantum ESPRESSO)</td>
</tr>
<tr>
<td></td>
<td>• Earth Sciences</td>
</tr>
<tr>
<td></td>
<td>• NumPy Acceleration</td>
</tr>
<tr>
<td></td>
<td>• Oil &amp; Gas (Seismic Imaging, Reservoir Simulation)</td>
</tr>
<tr>
<td></td>
<td>• Plasma Simulation</td>
</tr>
<tr>
<td></td>
<td>• Weather/Climate Simulation</td>
</tr>
<tr>
<td>Graphcore IPUs</td>
<td>• Graph Data</td>
</tr>
<tr>
<td></td>
<td>• LSTM Neural Networks</td>
</tr>
<tr>
<td></td>
<td>• Markov Chain Monte Carlo</td>
</tr>
<tr>
<td></td>
<td>• Natural Language Processing (Deep Learning)</td>
</tr>
<tr>
<td>Intel/Bittware FPGA</td>
<td>• AI Models for Embedded Use Cases</td>
</tr>
<tr>
<td></td>
<td>• Big Data</td>
</tr>
<tr>
<td></td>
<td>• CXL Memory Interface</td>
</tr>
<tr>
<td></td>
<td>• Deep Learning Inference</td>
</tr>
<tr>
<td></td>
<td>• Genomics</td>
</tr>
<tr>
<td></td>
<td>• MD Codes</td>
</tr>
<tr>
<td></td>
<td>• Microcontroller Emulation for Autonomy Simulations</td>
</tr>
<tr>
<td></td>
<td>• Streaming Data Analysis</td>
</tr>
<tr>
<td>Intel Optane SSDs</td>
<td>• Bioinformatics</td>
</tr>
<tr>
<td></td>
<td>• Computational Fluid Dynamics (OpenFOAM)</td>
</tr>
<tr>
<td></td>
<td>• MD Codes</td>
</tr>
<tr>
<td></td>
<td>• R</td>
</tr>
<tr>
<td></td>
<td>• WRF</td>
</tr>
<tr>
<td>NextSilicon</td>
<td>• Biosciences (BLAST)</td>
</tr>
<tr>
<td></td>
<td>• Computational Fluid Dynamics (OpenFOAM)</td>
</tr>
<tr>
<td></td>
<td>• Cosmology (HACC)</td>
</tr>
<tr>
<td></td>
<td>• Graph Search (Pathfinder)</td>
</tr>
<tr>
<td></td>
<td>• Molecular Dynamics (NAMD, AMBER, LAMMPS)</td>
</tr>
<tr>
<td></td>
<td>• Quantum ChromoDynamics (MILC)</td>
</tr>
<tr>
<td></td>
<td>• Weather/Environment modeling (WRF)</td>
</tr>
</tbody>
</table>
ACES Configuration - Feb 2024

- **Ice Lake Nodes (15)**
  - Kubernetes Host
  - Lustre System 2.54 PB usable
  - Management Nodes (4)
  - Data Transfer Nodes (2)
  - Login Nodes (3)
  - NDR InfiniBand
  - SPR Nodes (12)
    - Optane SSD
    - Intel FPGA
    - NextSilicon
  - SPR Nodes (13)
    - Optane SSD
    - Intel FPGA
  - SPR Nodes (12)
    - NVIDIA A30
  - SPR Nodes (13)
    - Optane SSD
    - BittWare FPGA
  - SPR Nodes (15)
    - Intel PVC
  - SPR Nodes (15)
    - Intel PVC
  - SPR Nodes (15)
    - Intel PVC
  - SPR Nodes (15)
    - NVIDIA H100

- **SPR Nodes (13)**
  - Optane SSD
  - Intel FPGA

- **SPR Nodes (12)**
  - Optane SSD
  - Intel FPGA
  - NextSilicon

- **SPR Nodes (15)**
  - Intel PVC

- **SPR Nodes (15)**
  - Intel PVC

- **SPR Nodes (15)**
  - Intel PVC

- **SPR Nodes (15)**
  - NVIDIA H100

- **NEC VE Node**
  - Graphcore Bow POD16
  - Graphcore Colossus POD16

- **Intel PVC**
  - Ice Lake Nodes (15)
  - Kubernetes Host

- **Intel PVC**
  - NEC VE Node

- **Intel PVC**
  - Graphcore Bow POD16

- **Intel PVC**
  - Graphcore Colossus POD16

planned
Getting on ACES

- You must have an ACCESS account!
- Application for ACES is available through ACCESS: https://allocations.access-ci.org
- Email us at help@hprc.tamu.edu for questions, comments, and concerns.

PIs can apply for an account and sponsor accounts for their researchers.

(Grad students may also apply directly with a letter of collaboration from their PI)
Batch Computing on Clusters

Workflow on a cluster:

- Interact via **your own machine**
- Log in to the cluster’s **portal** (and/or the **login nodes**) and write instructions
- Send instructions to **compute nodes** to do the heavy-lifting
Accessing the HPRC Portal

- HPRC webpage: hprc.tamu.edu, Portal dropdown menu
Accessing ACES via the HPRC Portal (ACCESS)

Log-in using your ACCESS credentials.

Select the Identity Provider appropriate for your account.
Shell Access via the Portal

Get a shell terminal right in your browser

Warning: Permanently added 'login.aces,18.71.1.13' (ECDSA) to the list of known hosts.
This computer system and the data herein are available only for authorized purposes by authorized users. Use by 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 25.81.300.62 Accessible use for more information.

Last login: Mon Feb 12 13:13:13 2024 from 18.71.1.6

Website: https://hprc.tamu.edu
Consulting: help@hprc.tamu.edu (preferred) or (979) 845-8219
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/texasacoct

----- ACCESS CONTROL: SECURITY POLICY INFORMATION

- Unauthorized use of HPRC resources is prohibited and subject to disciplinary action.
- 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: http://www.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 P4X.

!! 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 mail command.

Your current disk quotas are:

Disk Usage Limit File Usage Limit
/home/su,jw123527 1568 10.8G 469 10000
/jscratch/user/su,jw123527 189.1G 3.8T 182472 250000

Type 'showquotas' to view these quotas again.

su,jw123527@aces-logins-15
ACES Shell Access - Shell

---------- 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:

<table>
<thead>
<tr>
<th>Disk</th>
<th>Disk Usage</th>
<th>Limit</th>
<th>File Usage</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/u.jt1630</td>
<td>7.9G</td>
<td>10.0G</td>
<td>145</td>
<td>10000</td>
</tr>
<tr>
<td>/scratch user/u.jt1630</td>
<td>58.1G</td>
<td>1.0T</td>
<td>138298</td>
<td>250000</td>
</tr>
</tbody>
</table>

Type 'showquota' to view these quotas again.

[u.jt1630@aces-login1 ~]$
Commands to Copy Examples

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

- Download the files for this course
  
  ```
  wget https://hprc.tamu.edu/files/training/2024/Spring/cuda.exercise.tgz
  ```

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

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

```
[u.jt1630@aces-login3 hello_world]$ ml purge
[u.jt1630@aces-login3 hello_world]$ ml CUDA
[u.jt1630@aces-login3 hello_world]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Wed Nov 22 10:17:15 PST 2023
Cuda compilation tools, release 12.3, V12.3.107
Build cuda_12.3.r12.3/compiler.33567101_0
[u.jt1630@aces-login3 hello_world]$ nvcc ./hello_world_device.cu
[u.jt1630@aces-login3 hello_world]$ ./a.out
Hello World!
[u.jt1630@aces-login3 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

GPU-accelerated computing started a new era

2.3 PFlops

7000 homes

7.0 Megawatts

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks

7.0 Megawatts
GPU Computing Applications

Add GPUs: Accelerate Science Applications

Use GPU to Parallelize Compute-Intensive Functions

Rest of Sequential CPU Code

Application Code
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
  
  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.
3 Ways to Accelerate 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

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

Valuation of Stock Portfolios using Monte Carlo
Global Technology Consulting Company

Interaction of Solvents and Biomolecules
University of Texas at San Antonio

5x in 40 Hours  2x in 4 Hours  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

- **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

```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)

- Thrust C++ Template Library

- MATLAB

- CUDA Fortran

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

# copy sample code to your scratch space
$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 12.0 (as of Feb 2023).
Heterogeneous Computing

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

### Serial Code

```cpp
#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 lindex = threadIdx.x + blockIdx.x * blockDim.x;
    int gindex = lindex;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}
void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}
int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);
    // Alloc space for host copies and setup values
    in  = (int*)malloc(size); fill_ints(in,  N + 2*RADIUS);
    out = (int*)malloc(size); fill_ints(out, N + 2*RADIUS);
    // Alloc space for device copies
    cudaMalloc((void**)&d_in,  size);
    cudaMalloc((void**)&d_out, size);
    // Copy to device
    cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    // Cleanup
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

### Parallel Function

```cpp
__global__
void stencil_1d(int *in, int *out) {
    __shared__
    int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x;
    int index = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[index] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];
    // Store the result
    out[index] = result;
}
```
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

- 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.
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

```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

\[ \text{mykernel} \lll 1,1 \ggg (); \]

- Triple angle brackets mark a call from \textit{host} code to \textit{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;
}

• 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: add( )

• Returning to our add() kernel

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

• Let’s take a look at 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 \texttt{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.
```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:

<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;
    int size = 1000;
    cudaMallocManaged(&ret, size * 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

```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;
}
```
• 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 \texttt{add()} \ with \texttt{add<<<N,1>>>(...)}.
  – Use \texttt{blockIdx.x} to access block index.
  – Use \texttt{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.
__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

```
__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 (\texttt{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:
  \begin{verbatim}
  cudaError_t cudaGetLastError(void)
  \end{verbatim}

• Get a string to describe the error:
  \begin{verbatim}
  char *cudaGetErrorString(cudaError_t)
  printf("%s\n", cudaGetErrorString(cudaGetLastError()));
  \end{verbatim}
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 Texas A&M Engineering Experiment Station (TEES), 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