CPSC 4240/5240: Parallel Programming Techniques

Lecture 21: Programming on GPUs

Lecturer: Quanquan C. Liu

Spring 2026

Debugging using CUDA Tools

Debugging Using Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need for any device or CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (`gdb` breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- **Results of floating-point computations** will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

Run-time functions and macros

In CUDA run-time services,

cudaGetDeviceProperties(deviceProp &dp, d);

check number, type and whether device present

In libcutil.a of Software Developers' Kit,

cutComparef (float *ref, float *data, unsigned len);

compare output with reference from CPU implementation

In cutil.h of Software Developers' Kit (with #define _DEBUG or –D_DEBUG compile flag),

CUDA_SAFE_CALL(f(<args>)), CUT_SAFE_CALL(f(<args>))

check for error in run-time call and exit if error detected

CUT_SAFE_MALLOC(cudaMalloc(<args>));

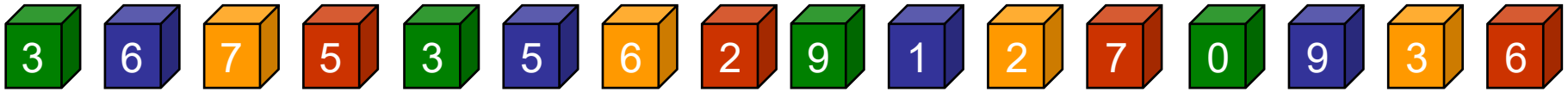
similar to above, but for malloc calls

CUT_CHECK_ERROR("error message goes here");

check for error immediately following kernel execution and if detected, exit with error message

Simple working code example

- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid



threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with “6”

Compute local result

DEVICE FUNCTION:

Compare current element and “6”

Return 1 if same, else 0

Reminder: Gathering and Reporting Results

- Global, device functions and excerpts from host, main

```
__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}

__global__
   *d_out) {

  d_out[thre

  for (i=0; i<

     int val = d_in[i*BLOCKSIZE +
           threadIdx.x];

     d_out[threadIdx.x] +=
                  compare(val, 6);
   }
}
```

Compute individual results for each thread

Serialize final results gathering on host

```
int __host__ void outer_compute
   (int *h_in_array, int *h_out_array) {
   …
   compute<<<1,BLOCKSIZE,msize)>>>
     (d_in_array, d_out_array);

 cudaMemcpy(h_out_array,
d_out_array,
   BLOCKSIZE*sizeof(int),
   cudaMemcpyDeviceToHost);

main(int argc, char **argv) {
…
  for (int i=0; i<BLOCKSIZE; i++)
  {  sum+=out_array[i]; }
  printf (”Result = %d\n",sum);
}
```

Gathering Results on GPU: Synchronization

```
void __syncthreads();
```

- Functionality: Synchronizes all threads in a block
  - Each thread waits at the point of this call until all other threads have reached it
  - Once all threads have reached this point, execution resumes normally
- Why is this needed?
  - A thread can freely read the shared memory of its thread block or the global memory of either its block or grid.
  - Allows the program to guarantee partial ordering of these accesses to prevent incorrect orderings.
- Watch out!
  - Potential for deadlock when it appears in conditionals

Gathering Results on GPU for “Count 6”

```
__global__ void compute(int *d_in, int
   *d_out) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

      int val = d_in[i*BLOCKSIZE +
            threadIdx.x];

      d_out[threadIdx.x] +=
                  compare(val, 6);

   }
}
```

```
__global__ void compute(int *d_in, int
   *d_out, int *d_sum) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

      int val = d_in[i*BLOCKSIZE +
            threadIdx.x];

      d_out[threadIdx.x] +=
                  compare(val, 6);

   }
   __synthreads();
   if (threadIdx.x == 0) {
     for 0..BKSIZE-1 {
     *d_sum += d_out[i];
   }
}
```

Thread Naming (Again)

Naming Threads

- Grid → blocks → threads
- Each thread has threadIdx
- Each block has blockIdx
- Each block has blockDim
- Typical 1D index:

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

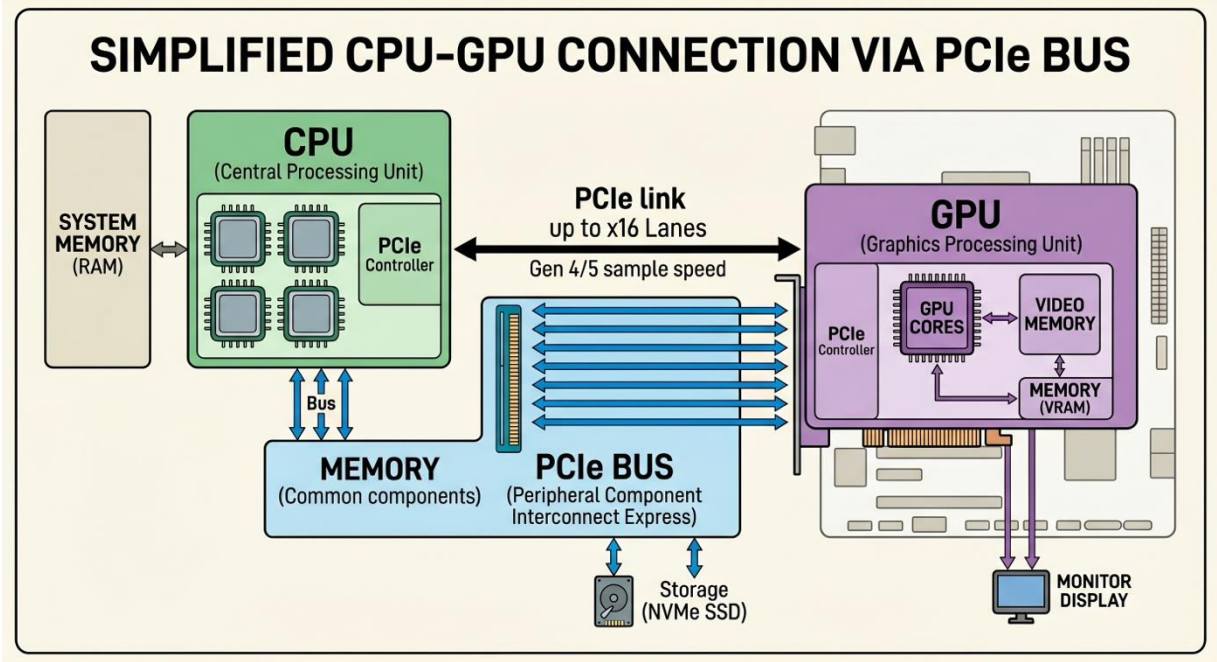
```
// THE KERNEL (Runs on the GPU)
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements) {
    // 1. The Golden Formula: Calculate the global thread ID
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. The Boundary Check: Ensure we don't read/write past the array
    if (i < numElements) {
        C[i] = A[i] + B[i];
    }
}
```

```
// THE HOST LAUNCH (Runs on the CPU)
void launchKernel(float *d_A, float *d_B, float *d_C, int numElements) {
    // Define the execution hierarchy
    int threadsPerBlock = 256;

    // Calculate total blocks needed (ceiling division)
    int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;

    // Launch the kernel with the Grid and Block dimensions
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
}
```

More Details about GPU Architecture

PCIe: Peripheral Component Interconnect Express

- Standard physical connection that allows **CPU communication with GPUs**



PCIe: Peripheral Component Interconnect Express

- Standard physical connection that allows **CPU communication with GPUs**
- **PCIe Bus:**
  - Physical multi-lane memory bus that transfers data from CPU to GPU
  - CUDA **manually coordinates transferring data across PCIe bus**
  - We've seen the syntax for moving data from the CPU to GPU before

PCIe: Peripheral Component Interconnect Express

- CUDA Programming Syntax:
  - **Allocation (cudaMalloc):** The CPU sends a message over PCIe telling the GPU to clear out space in its local VRAM
  - **The Transfer In (cudaMemcpy HostToDevice):** CPU packs up raw data transfers across **PCIe** to the GPU
  - **The Compute (myKernel<<<...>>>()):** GPU computes using data; does not use the PCIe bus
  - **The Transfer Out (cudaMemcpy DeviceToHost):** GPU takes final results and ships back across the **PCIe bus** to the CPU

PCIe: Peripheral Component Interconnect Express

Data transfer across PCIe 50 to 100 times slower than GPU memory speeds

- CUDA Programming Syntax:
  - **Allocation (cudaMalloc):** The CPU sends a message over PCIe telling the GPU to clear out space in its local VRAM
  - **The Transfer In (cudaMemcpy HostToDevice):** CPU packs up raw data transfers across **PCIe** to the GPU
  - **The Compute (myKernel<<<...>>>()):** GPU computes using data; does not use the PCIe bus
  - **The Transfer Out (cudaMemcpy DeviceToHost):** GPU takes final results and ships back across the **PCIe bus** to the CPU

PCIe: Peripheral Component Interconnect Express

Slowest part of your GPU program! Minimize your memory transfers!

- CUDA Programming Syntax:
  - **Allocation (cudaMalloc):** The CPU sends a message over PCIe telling the GPU to clear out space in its local VRAM
  - **The Transfer In (cudaMemcpy HostToDevice):** CPU packs up raw data transfers across **PCIe** to the GPU
  - **The Compute (myKernel<<<...>>>()):** GPU computes using data; does not use the PCIe bus
  - **The Transfer Out (cudaMemcpy DeviceToHost):** GPU takes final results and ships back across the **PCIe bus** to the CPU

PCIe: Peripheral Component Interconnect Express

- Simple Skeleton:

```
cudaMalloc(&d_x, n * sizeof(float));
cudaMemcpy(d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice);
kernel<<<blocks, threads>>>(d_x, n);
cudaMemcpy(h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost);
```

CUDA Memory Management

CUDA Memory Management API

- Prefixes are programmers' convention; GPU (device) pointers are prepended with d_ and CPU (host) pointers are prepended with h_
- **Segmentation fault** occurs when you try to dereference a device pointer from host and vice versa
  - E.g. Setting values in arrays from the wrong location will also result in segmentation fault

Standard malloc vs. cudaMalloc

- **Standard C malloc returns an address**: void* malloc(size_t size);
- **cudaMalloc returns a status code**:
  - cudaError_t cudaMalloc(void** devPtr, size_t size);
- **The C-Compatibility Constraint**: C functions can only return a single value.
- **The Design Choice**: NVIDIA uses the return value strictly for success/failure checks (cudaError_t).
- **The Outcome**: The newly allocated VRAM address must be passed back through the function arguments.

Using Double Pointers

- **The Host Variable:** int* arr; is declared and lives in CPU memory.
- **The Pass-by-Value Problem:** Passing arr directly only provides a copy of a null or uninitialized address.
- **Passing the Address:** To modify the actual pointer, the API needs its location in memory: &arr.
- **The Generic Cast:** Since &arr is an int** and the API requires a generic pointer, it is cast to (void**).
- **Final Syntax:** cudaMalloc((void**)&arr, size); allows the host pointer to be successfully updated with the device address.

More Architectural Notes

SIMT Architecture

- **What the programmer sees**: CUDA allows you to write a kernel as if it is executing on a single, independent thread (e.g., int tid = threadIdx.x;).
- **The Physical Reality (SIMT):** The GPU hardware does not schedule or execute individual threads. It groups threads into rigid bundles of 32, called **Warps**.
- **Single Instruction, Multiple Threads**: Every thread within a warp shares a single instruction.
- **The Lockstep Rule**: All 32 threads *must* execute the exact same instruction at the exact same time.
- **Analogy**: Think of a warp not as 32 independent motorcycles, but as a 32-person rowboat. Everyone must row at the exact same tempo.

Warp Divergence

- **The Question**: What happens if the code has an if/else statement, and the threads in a warp disagree on which path to take?
- **The Answer: Serialization.**
  - The GPU **cannot split the warp**.
  - Instead, it executes the if branch first. Threads that evaluated to false are physically disabled (masked out) and sit idle.
  - Then, it executes the else branch. The threads that just finished the if branch are put to sleep, and the false threads wake up to do their work.

Warp Divergence

Very Bad!

```
// threadIdx.x goes from 0 to 31 inside the warp
if (threadIdx.x % 2 == 0) {
    do_even_work(); // 16 threads active, 16 asleep
} else {
    do_odd_work();  // 16 threads asleep, 16 active
}
```

Warp Divergence

It's OK

```
// Branching based on the Block ID rather than the Thread ID
if (blockIdx.x == 0) {
    do_block_zero_work();
} else {
    do_other_work();
}
```

Another Example: Adding Matrices

Another Example: Adding Two Matrices

CPU C program

```
void add_matrix_cpu(float *a, float *b,
   float *c, int N)
{
int i, j, index;
for (i=0;i<N;i++) {
            for (j=0;j<N;j++) {
      index =i+j*N;
      c[index]=a[index]+b[index];
    }
  }
}

void main() {
   .....
   add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu(float *a, float *b, float *c, intN)
{
   int i =blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   int index =i+j*N;
   if( i <N && j <N)
     c[index]=a[index]+b[index];
}

void main() {
   dim3 dimBlock(blocksize,blocksize);
   dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- Let blocksize = 2 and N = 4

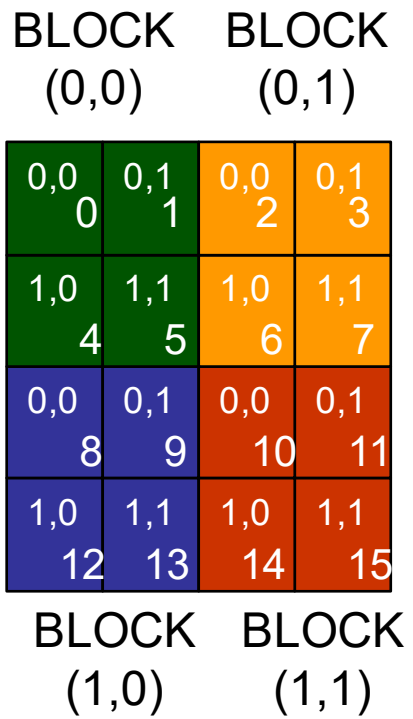
Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what single element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```



- Let blocksize = 2 and N = 4

Data Distribution to Threads

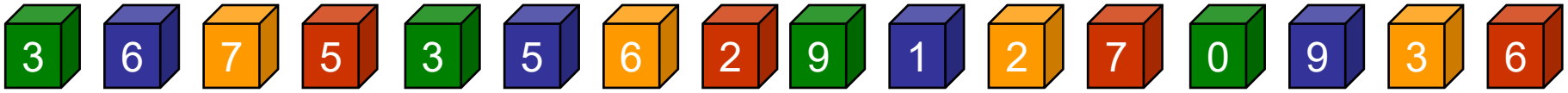
- Many data parallel programming languages have mechanisms for expressing **how a distributed data structure is mapped to threads**
  - That is, the **portion of the data a thread accesses** (and usually stores locally)
- Reasons for selecting a particular distribution
  - Sharing patterns: group together according to “data reuse” and communication needs
  - Affinity with other data
  - Locality: transfer size and capacity limit on shared memory
  - Access patterns relative to computation
  - Minimizing overhead: Cost of copying to host and between global and shared memory

Common Data Distributions

- Consider a 1-Dimensional array

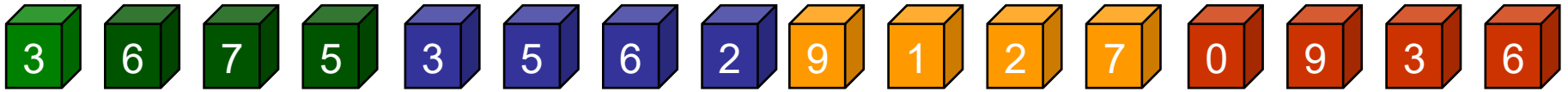
CYCLIC:

```
for (i = 0; i<BLOCKSIZE; i++)
        … d_in [i*BLOCKSIZE + threadIdx.x];
```



BLOCK:

```
for (i=threadIdx.x*BLOCKSIZE; i<(threadIdx.x+1)
    *BLOCKSIZE; i++)
    … d_in[i];
```



Which one is better?

- **Block distribution** is simpler and often more friendly to CUDA's memory system
  - Naturally aligns well with local reuse patterns.
- However, **cyclic distribution** sometimes brings these benefits:
  - **Better load balancing**
  - **Avoiding hot spots**
    - Contiguous chunks of data might cause too many memory conflicts or high load on particular parts of memory
  - **Periodic access patterns**
    - For workloads where data is processed in fixed strides, a cyclic distribution lines up more naturally with strides