Introduction to CUDA

HEROES Lab

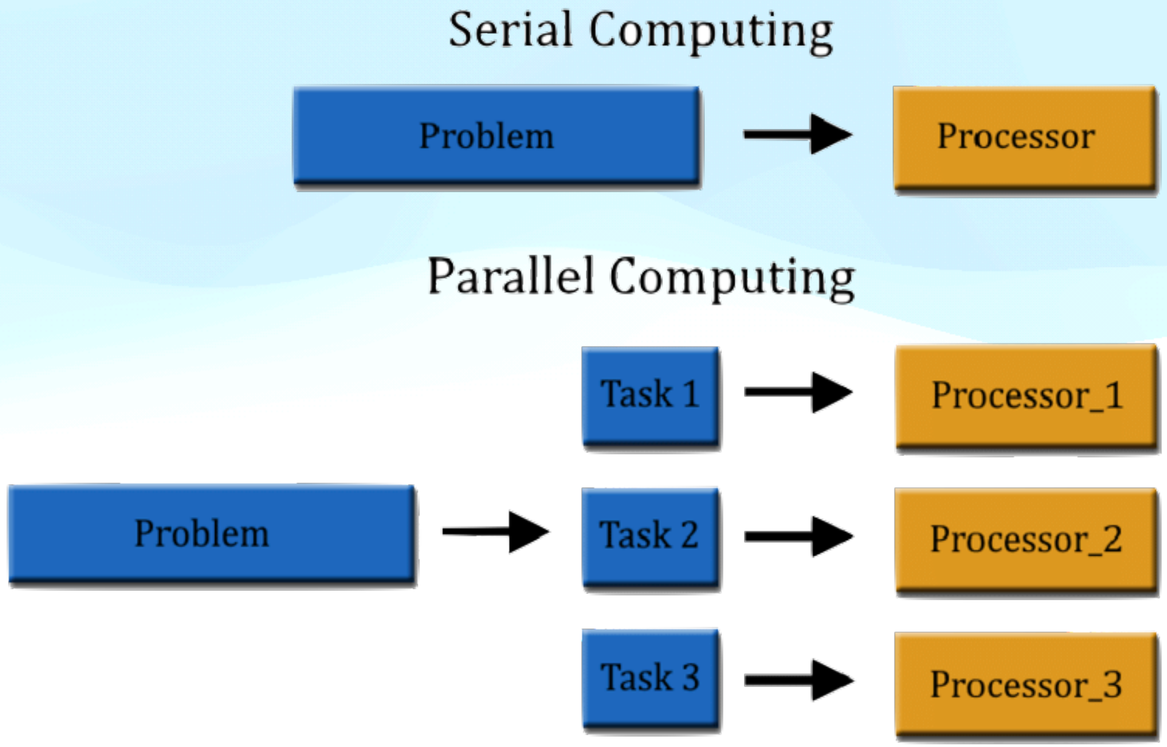
University of Mississippi

June 15th, 2023

THE UNIVERSITY of MISSISSIPPI

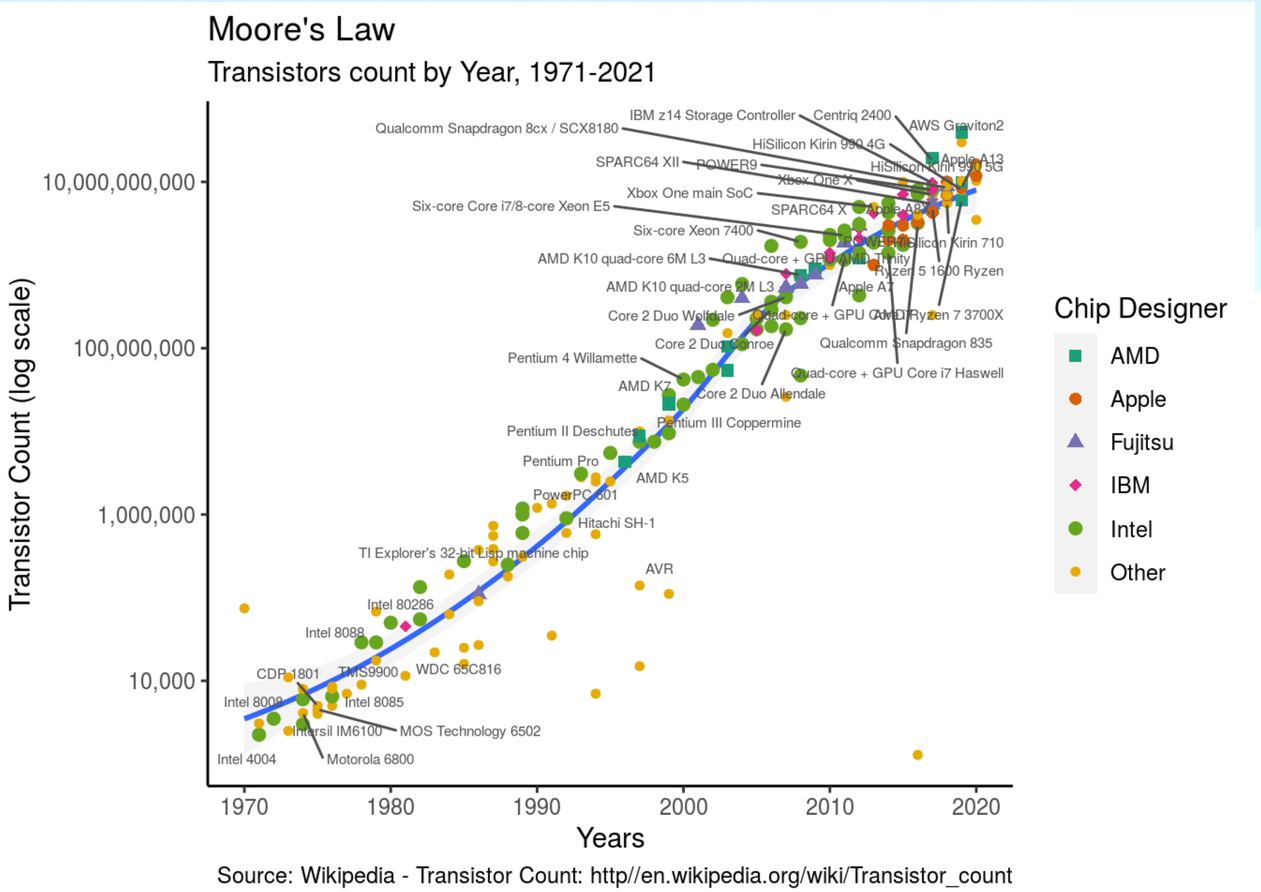
What is Parallel Computing?

- One piece of hardware, with multiple cores, performing several operations at once
- Models the real world
- Saves time with complex or larger problems



What is Parallel Computing?

- With the amount of data we have these days, we need ways to speed things up
- Moore’s Law: the number of transistors in a dense integrated circuit (IC) doubles about every two years
- Need to leverage parallelism to make more progress



Source: Wikipedia - Transistor Count: http//en.wikipedia.org/wiki/Transistor_count

Usage of GPUs

- GPUs were originally designed for image and 3D rendering
  - Compute how the geometric mesh contributes to appearance of pixels

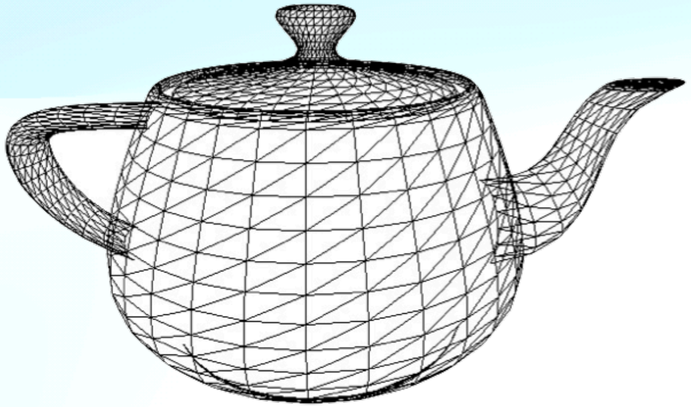


Image credit: Henrik Wann Jensen

Input: description of a scene:

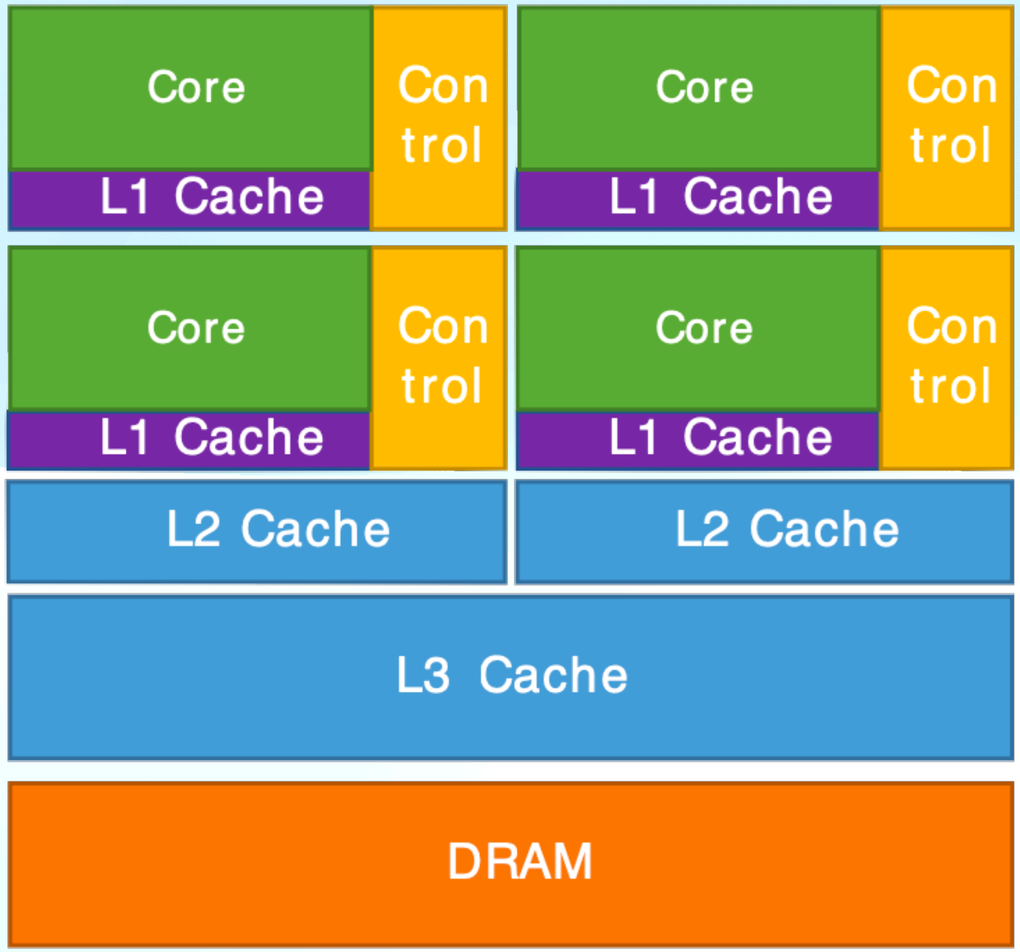
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

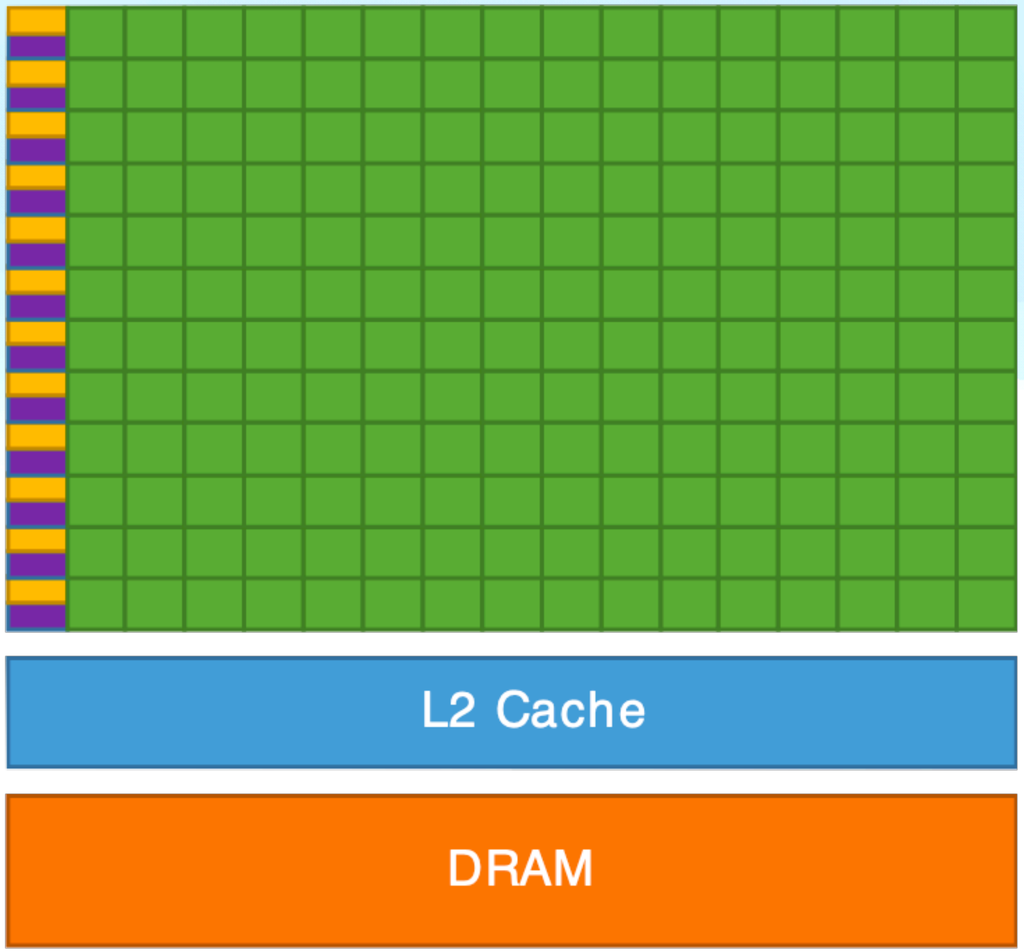
Usage of GPUs

- GPUs support a **Turing Complete** programming model
  - **Any computation can be run** given enough time and resources
- Used nowadays for photo/video editing, machine learning, crypto mining, etc.

GPU Architecture



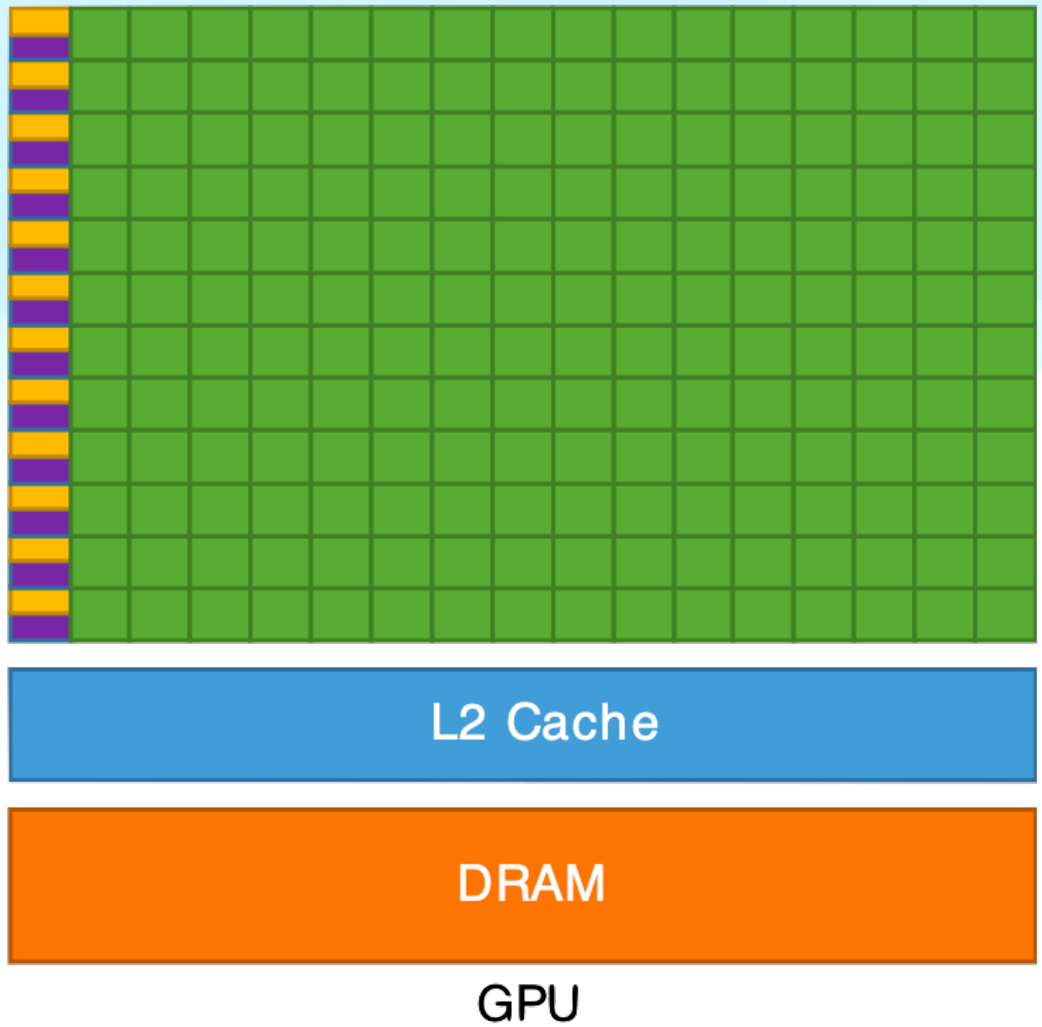
CPU



GPU

GPU Architecture

- **Streaming Multiprocessor**
- **CUDA Core**
- **Warp**: 32 threads
- **SIMD** - Single Instruction Multiple Data
- L1 cache is shared — GPU makes up for it by hiding the memory access overhead by having many threads



CUDA Programming Language

Compute **U**nified **D**evice **A**rchitecture

- Language like C to create programs that can run on GPUs
- Low-level
- Only runs on NVIDIA GPUs (OpenCL for AMD and others)
- Better documented than OpenCL

NVIDIA
CUDA

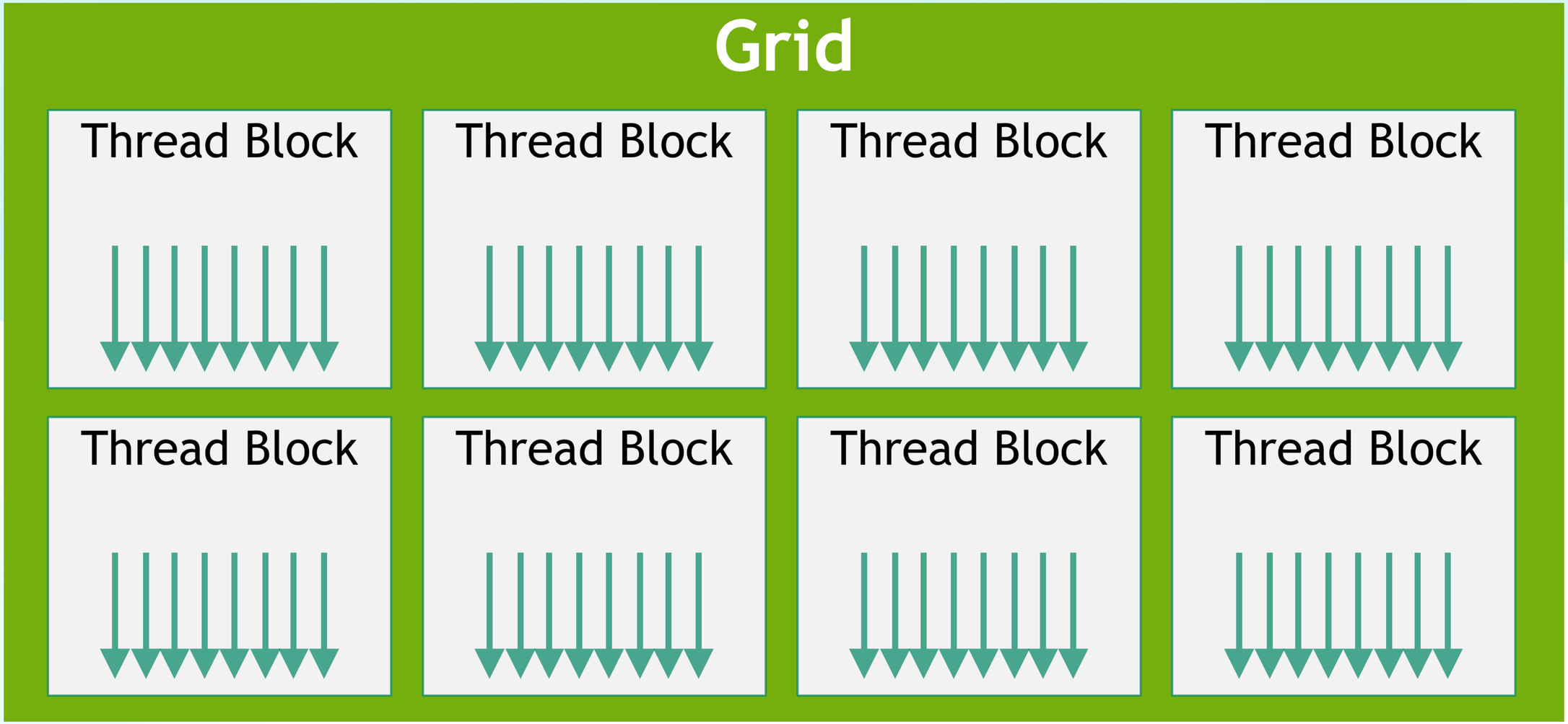
Getting started with CUDA

- Need a machine with an NVIDIA GPU
- Need the GPU drivers installed, as well as the CUDA Toolkit
- NVIDIA has amazing CUDA documentation to dive deeper into CUDA
- CUDA programs look very similar to C/C++
- A file containing CUDA Kernels must have the *.cu* extension
- A CUDA program can be run from the terminal through the *nvcc* command

Thread Hierarchy

- Threads run in an organized manner - allows for scalability
- 3 important terms - **Grid, Thread Block, Threads**
- **Grid -** contains multiple thread blocks
- **Thread block -** contains multiple threads
- Each thread block occupies a Streaming Multiprocessor

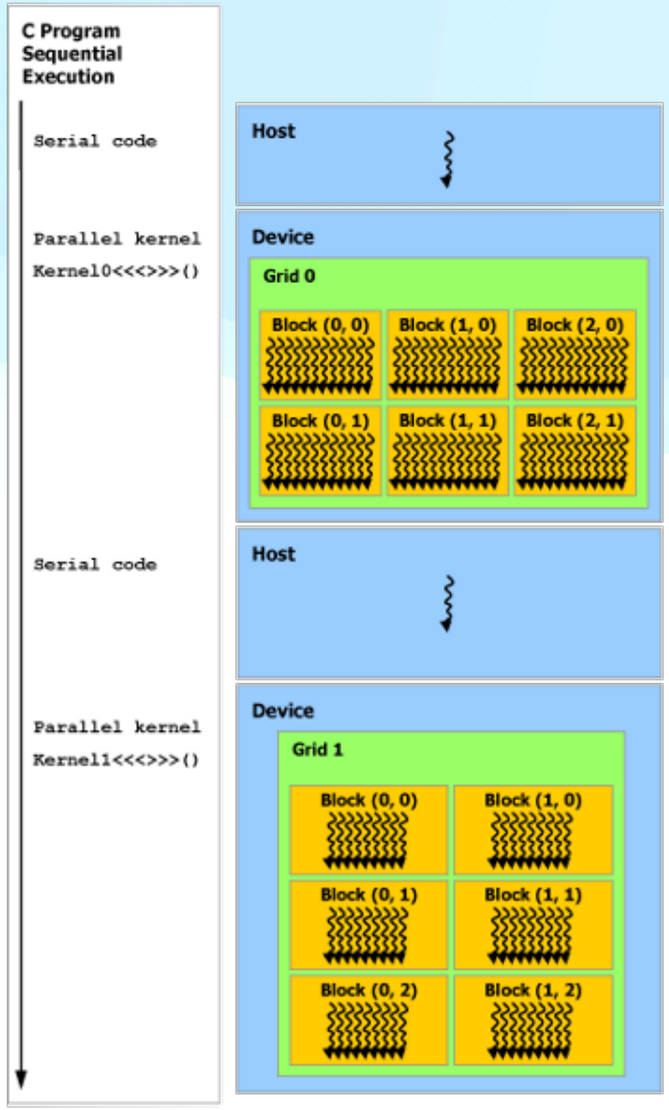
Thread Hierarchy



CUDA Program Execution Flow

- CPU - host, GPU - device

1. Host allocates memory and transfers data to device
2. Host calls the kernel to run code on the device
3. Device runs kernel in parallel
4. Host retrieves resultant data from device



CUDA Kernels

- Special CUDA-defined C++ functions
- Function called by the CPU to run code on the GPU
- Kernel is defined by the **“__*global*__”** keyword
- CPU calls the kernel with the grid and thread block dimensions
- Syntax: *kernelName<<<gridSize, blockSize>>>(arguments);*

CUDA Kernel Example

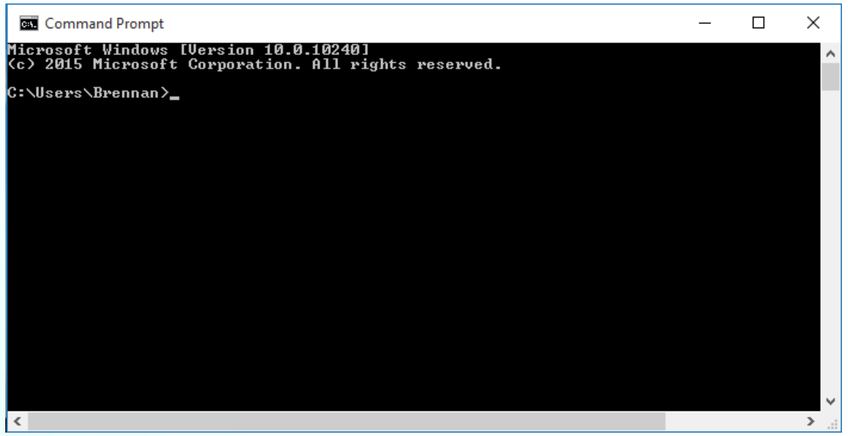
```cpp
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

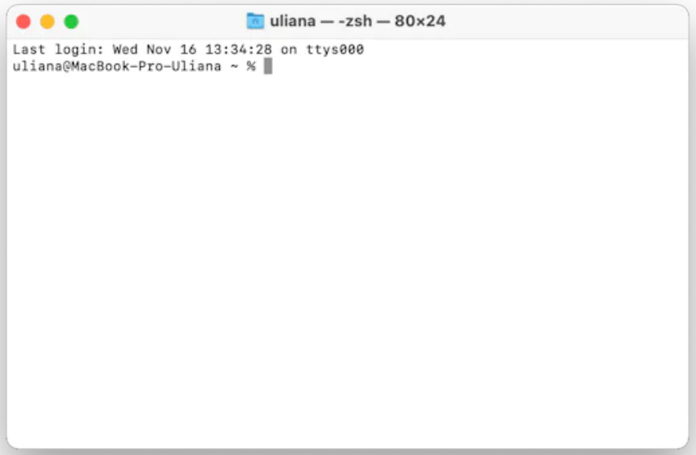
CUDA Device Synchronize

- A kernel launch is *asynchronous*
- Control is returned to CPU right after kernel startup
- What does the CPU do next? Exit the program :(
- It is important to know when computation is finally done on the GPU
- With a function called cudaDeviceSynchronize(), the kernel is guaranteed to finish before program exits

Terminal

- Most important tool for programmers
- For us, we will use it to run our programs, and connect to a computer that has a GPU
- Windows: *Command Prompt*, MacOS/Linux: *Terminal*
- Important commands to remember: *cd, g++, nvcc*





SSH

- You can access other computers (open to public) through ‘SSH’
- SSH - Secure Shell
- Need hostname/IP address, and a user account on the computer
  - All of you have an account on my computer with a GPU
- Format of ssh command:

  ssh username@130.74.96.14

  Password: *HEROESsummer2023*

Running a CUDA Program

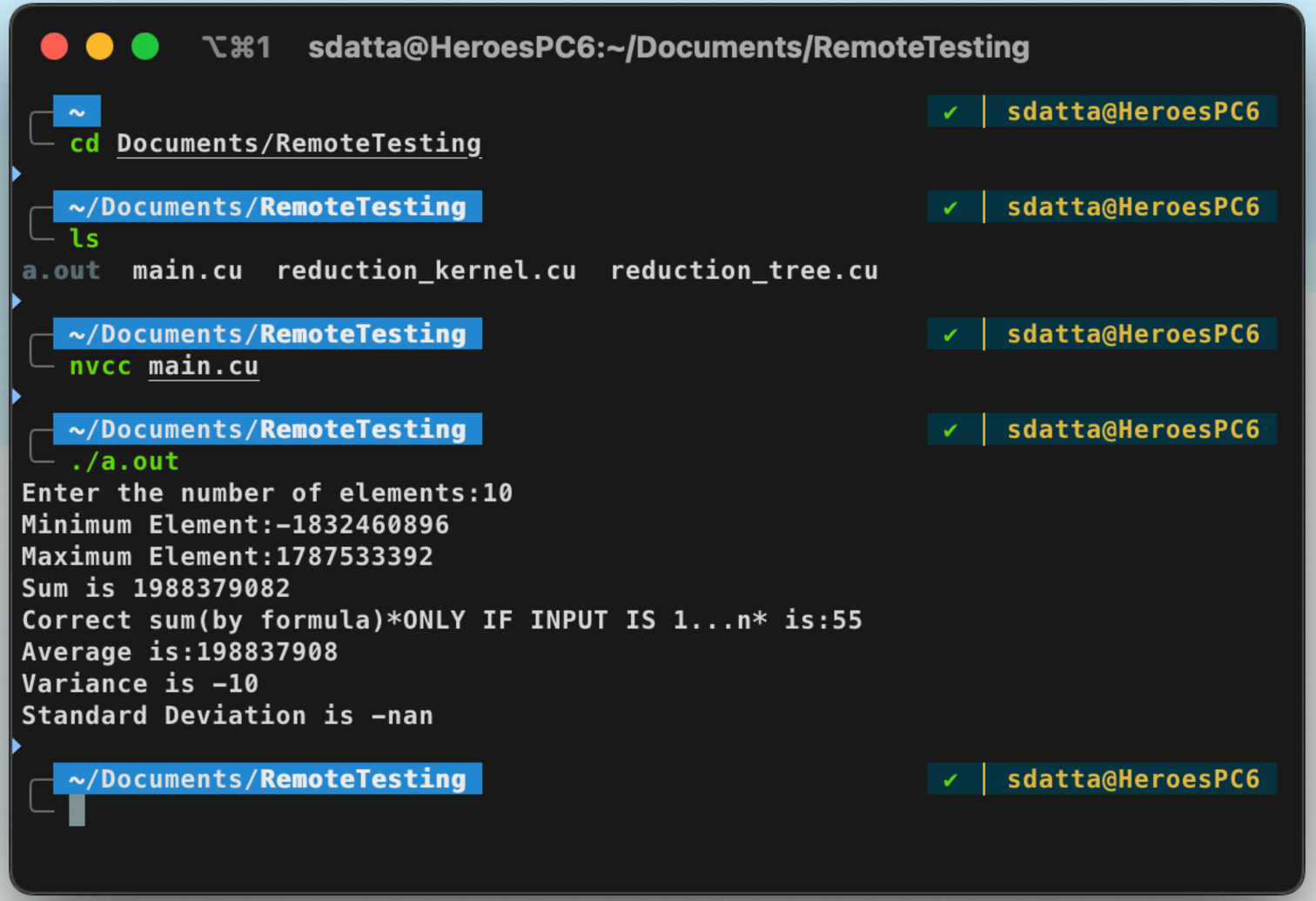
- Terminal must be in the working directory containing the *.cu* CUDA program
  - Use the *cd* command to change the working directory
  - Use *dir*(windows) or *ls*(MacOS/Linux) to check what files are in the directory
- Use the *nvcc* command to compile the program

  nvcc programName.cu
- Run the program by executing the ./a.out file

Alternative way —

- Use Google Colab!
- Free access forever
- Downside — not great for research level performance, but good enough to see noticeable performance improvements
- Another downside — made for Python code, will not detect errors in C++ code
- Template & Instructions:
    - github.com/soumildatta/GoogleColabCUDA-Template

Running a CUDA Program from Terminal



Let’s try connecting to the computer with our accounts!

Username: firstnameLastInitial

Password: HEROESsummer2023

Time to download FileZilla

Username: firstnameLastInitial

Password: HEROESsummer2023

Let’s look at a live example

CUDA Kernels

- Important to know which thread is running the kernel — built-in variables!
- ***threadIdx, blockIdx, blockDim***
- Threads in each block start from index 0
- Simple arithmetic to figure out the **relative** and **local** index of thread:
  - **Relative**: blockIdx * blockDim + threadIdx
  - **Local**: threadIdx

Device Memory

- Kernels work only with Device (GPU) memory
- Must transfer data between CPU and GPU — **2 step process:**
    1. Allocate device memory — ***cudaMalloc(…)***
    2. Transfer data to allocated memory — ***cudaMemcpy(…)***
- Two types of *cudaMemcpy():*
    3. *cudaMemcpyHostToDevice*
    4. *cudaMemcpyDeviceToHost*

Device Memory Code Sample

Host memory allocation

Device memory allocation

Transfer data from host to device

```
int N = ...;
size_t size = N * sizeof(float);

// Allocate input vectors h_A and h_B in host memory
float* h_A = (float*)malloc(size);
float* h_B = (float*)malloc(size);
float* h_C = (float*)malloc(size);

// Initialize input vectors
...

// Allocate vectors in device memory
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Device Memory Code Sample

Kernel invocation

Transfer from device to host

Free memory

```
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// Free host memory
...
```

Global vs. Shared Memory

- Shared memory is on-chip (VRAM)— much faster than Global memory (DRAM off chip)
- 100x+ lower latency on average
- Global memory can be accessed by any thread in any block
- Shared memory is only local to a block. Other blocks cannot access another block’s shared memory
- Shared memory is useful to be used as a scratch-pad: Load items from global memory, and use shared memory instead of making calls to global memory again

Introduction to CUDA

HEROES Lab

University of Mississippi

June 16th, 2023

THE UNIVERSITY of MISSISSIPPI

Thread Synchronization

- Lots of things can go wrong in parallel programming
- Real life example: Group members do not communicate on a project
- Threads must communicate about what they are working on
- Bank account example:

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

What if things went wrong?

- Cannot control how instructions are ordered in parallel program
- 1 is lost from balance because of A and B reading the balance at the same time
- This is called a **race condition**

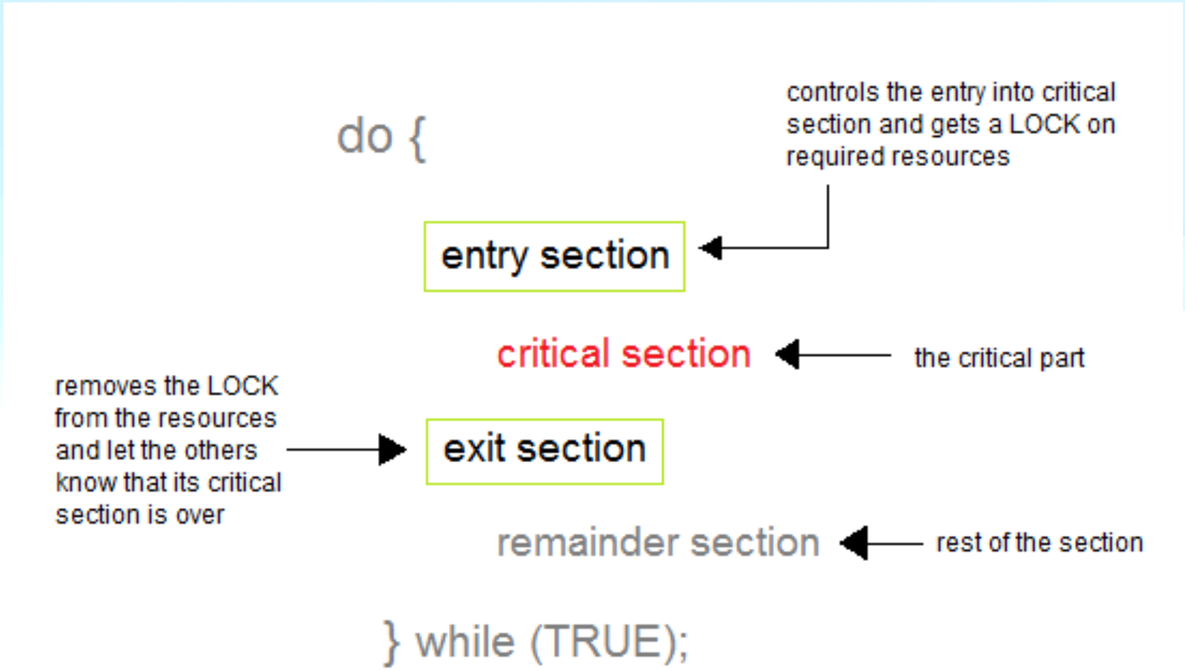
A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

How to fix it?

- There are several tools developed to help solve these problems
- Common tools used: Lock synchronization, Atomic synchronization
- Accesses to shared data should be synchronized
- **Critical section** - part of the code that causes problems without synchronization

Lock

- Lock the critical section — only the thread working on it currently gets access
- Threads try to acquire lock
- Other threads wait for the lock to be lifted
- Once working thread lifts lock, other thread acquires lock



Lock Example

- Pseudocode:

```
global var sum;

arraySum()

{

    arrayPart = array[threadBegin to threadEnd]

    localSum = sum(arrayPart)

    sum += localSum  <-------------------- Critical Section

}
```

Lock Example

```
global var sum;

arraySum()
{    …
     localSum = sum(arrayPart)
     lock()
     sum += localSum
     unlock()
}
```

Critical Section

Lock Downsides

- All other threads have nothing to do while one thread works on critical section
- OS-dependent — might work different on windows and linux
- Context-switching overhead: threads are suspended, need time to restart the thread

Atomic Operations

- No other thread can interfere with an atomic operation
- Utilizes hardware mechanisms instead of software like locks — performance
- Atomic add, atomic compare and swap, etc…
- Threads keep trying the atomic operation until it goes through — no context-switching overhead
- Mutex locks have worse performance on GPUs due to the concept of warps — can cause **deadlocks**

Atomic Operations in CUDA

- GPU has dedicated machine instructions for atomic operations
- CUDA has simple and useful atomic tools:
  - int atomicAdd(variableAddress, 1)
  - int atomicCAS(varAddress, currentValOfVar, valueToSwapWith)

```cpp
#include <iostream>

__global__  void testAdd(float *a)
{
    for (int i = 0; i < 100 ; i++)
    {
        atomicAdd(&a[i], 1.0f);
    }
}
void cuTestAtomicAdd(float *a)
{
    testAdd<<<1, 10>>>(a);
}

int main(){
    float *d_data, *h_data;
    h_data=(float *) malloc(100*sizeof(float));

    cudaMalloc((void **)&d_data, 100*sizeof(float));
    cudaMemset(d_data, 0, 100*sizeof(float));
    cuTestAtomicAdd(d_data);
    cudaMemcpy(h_data, d_data, 100*sizeof(float), cudaMemcpyDeviceToHost);

    for (int i = 0; i < 100; i++)
        if (h_data[i] != 10.0f) {printf("mismatch at %d, was %f, should be %f\n", i, h_data[i], 10.0f); return 1;}
    printf("Success\n");
    return 0;
}
```

Let’s code together

Performance Metrics

- Analyzing the performance of a program is important
- Different types of performance metrics — execution time, throughput, scalability, etc
- Execution time — time taken for a program/kernel to finish calculations
- Throughput — Operations per second (operations / execution time)
- Scalability — Throughput over different thread block configurations

Performance Metrics

- Getting execution time is therefore quite important
- Get time in C++ — std::chrono
- Chrono only gives the current time — get time before and after execution and subtract to get execution time
- Lets dive into a quick example!

Chrono example code

```cpp
#include <chrono>
#include <iostream>
using std::cout; using std::endl;

int main()
{
    std::chrono::high_resolution_clock::time_point startTime = std::chrono::high_resolution_clock::now();

    for(int i = 0; i < 10000; ++i)
    {
        cout << i << endl;
    }

    std::chrono::duration<double> duration = std::chrono::high_resolution_clock::now() - startTime;
    double totalTime = duration.count();
    cout << "Time taken: " << totalTime << " seconds" << endl;
}
```

2D array / Matrix

- 1D arrays can be accessed using a single index. Ex: arr[1], arr[9], etc…
- 2D array is an array of arrays:

  [[1, 2, 3, 4],

  [2, 5, 7, 2],

  [3, 1, 7, 3]]

- Accessed with two indices: arr[1][1], arr[2][0], etc…

2D array / Matrix

- Memory on a GPU is limited, so we must make good use of it
- 2D arrays can be converted into 1D arrays
  - Simply change access patterns / index calculation to treat it like a 2D array
- **row * rowDim + index**

2D array / Matrix

[[1, 2, 3, 4],

[2, 5, 7, 2], = [1, 2, 3, 4, 2, 5, 7, 2, 3, 1, 7, 3]

[3, 1, 7, 3]]

- array[0 * 4 + 0] = 1
- array[1 * 4 + 0] = 2
- array[2 * 4 + 3] = 3

We are ready to create some GPU programs!! 🎉