COSC 407
Intro to Parallel Computing

Topic 12: Intro to CUDA

1

Outline

Today's topics:

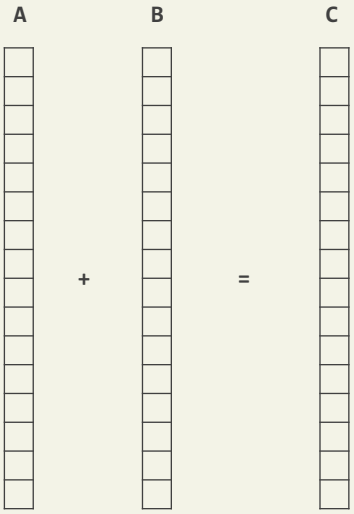
- Intro to GPU programming
  - CPU vs GPU programming
  - Latency vs. Throughput
- CUDA basics: the hardware layout
- CUDA basics: program structure
- Kernel Launch
- Useful Built-in CUDA functions
- Function Declarations (global, device, host)
- Simple examples

Next Lecture:

- Error Handling
- cudaDeviceSynchronize
- Thread organization

2

Serial vs Parallel on **CPU** / GPU



Serial code:

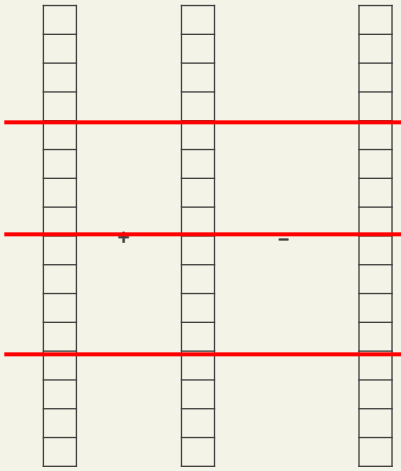
one thread does all the work

```
for(i=0; i<100; i++)
   C[i] = A[i] + B[i];
```

Thread 0

3

Serial vs Parallel on **CPU** / GPU



Parallel on CPU:

Divide the work among Threads

```
for(i=0; i<25; i++)
   C[i] = A[i] + B[i];
```

Thread 0

```
for(i=25; i<50; i++)
   C[i] = A[i] + B[i];
```

Thread 1

```
for(i=50; i<75; i++)
   C[i] = A[i] + B[i];
```

Thread 2

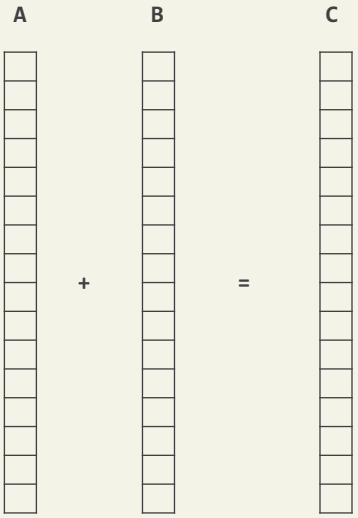
```
for(i=75; i<100; i++)
   C[i] = A[i] + B[i];
```

Thread 3

(Assuming static scheduling)

4

Serial vs Parallel on CPU / **GPU**



On GPU: many threads do the work (one thread/element)

```
C[0] = A[0] + B[0];      Thread 0
C[1] = A[1] + B[2];      Thread 1
C[2] = A[2] + B[2];      Thread 2
C[3] = A[3] + B[3];      Thread 3
        .
        .
        .
C[97] = A[97] + B[97];   Thread 97
C[98] = A[98] + B[98];   Thread 98
C[99] = A[99] + B[99];   Thread 99
```


5

Keep in Mind…

Amdahl's law :

$$\text{Max speedup} = \frac{1}{1-r}$$

Limitation: applies to situations where **problem size is fixed**.

Gustafson's Law:

Max speedup = p (with large enough problem sizes)

- With more cores, larger problem sizes (datasets) can be solved within the same time

Limitation: doesn't apply to problems which do not have fundamentally large datasets.

6

Latency vs. Throughput

Latency: time to finish one task (in seconds)

Throughput: number of tasks finished per unit of time (items/sec)

Latency and Throughput are not necessarily aligned:

- You are standing in a long line at a supermarket
  - **YOU** are aiming for less latency (to finish as soon as possible).
  - **Supermarket Manager** is optimizing for throughput (to serve as many customers as possible per hour)

Option1: 1 fast employee

Option2: 10 slow employees

7

Latency vs. Throughput: Another Example

Required: move rocks from point A to point B

- Mission 1: 2 big rocks.
- Mission 2: 200 small rocks

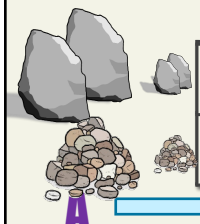
Option1: 2 strong workers **Option2**: 200 weak workers.

- Worker can only carry one rock at a time. A rock can be carried by one worker.
- Time taken to move one rock is shown below (the latency)

Q1: Which option would you choose to finish each mission?

Q2: What if we have a 3rd mission for moving both big and small rocks?

		Option1	Option2
Mission 1: 2 Big rocks	Latency	10 min	100 min
	Overall time	**10 min**	100 min
Mission 2: 200 Small rocks	Latency	2 min	10 min
	Overall time	200 min	**10 min**

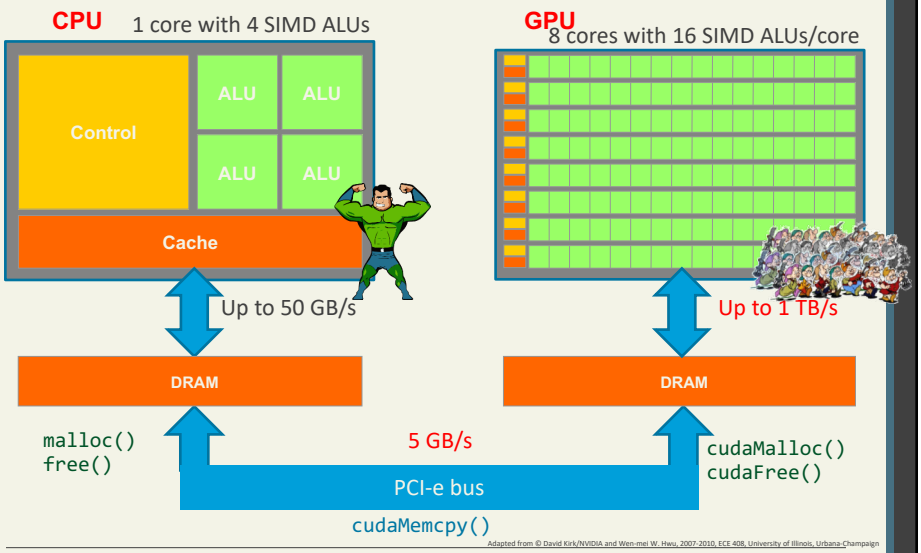


A B

8

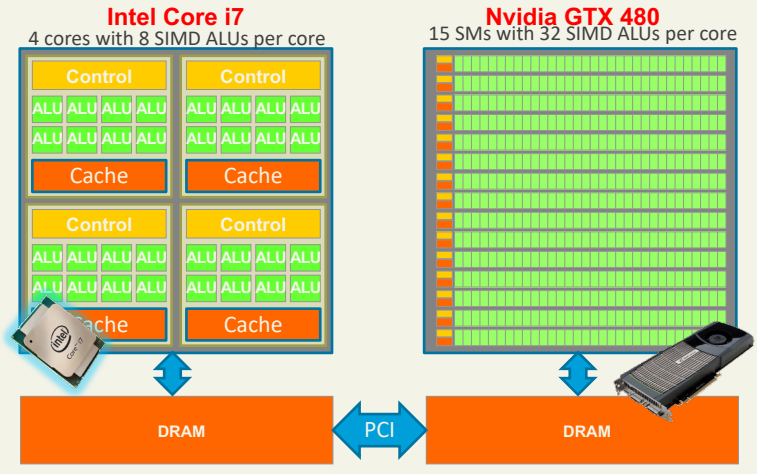
CPU vs. GPU

CPUs and GPUs have fundamentally different design philosophies



9

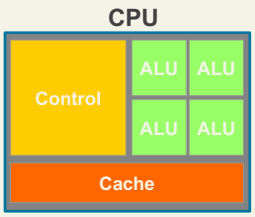
CPU vs. GPU



10

CPUs:
Latency Oriented Design

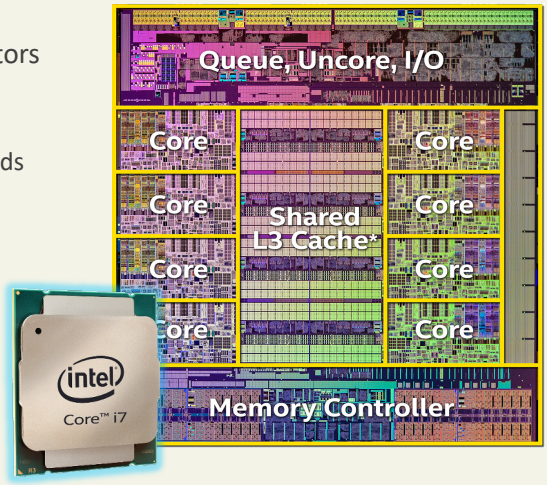
- Optimized for sequential code performance
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Minimize operation **latency**
  - Using ***Powerful ALU***
- Large caches
  - Reduce latency cache accesses



11

Core i7-5960X

- 2.6 billion transistors
- 3 GHz clock rate
- 8 cores
  - 16 hyper threads
  - SIMD

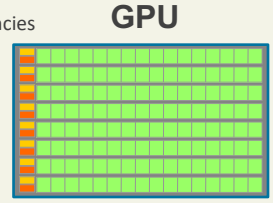


http://techgage.com/article/core-i7-5960x-extreme-edition-review-intels-overdue-desktop-8-core-is-here/

12

GPUs:
Throughput Oriented Design

- Maximize **throughput**
  - **long latency** but heavily pipelined for high throughput
  - Require **massive number** of **threads** to tolerate latencies
- Less space for control logic
  - Trade simple control for more compute
    - No branch prediction
    - No data forwarding
- Large number of simple ALUs per core
  - SIMD execution for each core
    - Many ALUs performing the same instruction
    - You have to design your code to make use of these units
- Memory optimized for bandwidth
  - Large bandwidth allows for serving many ALUs simultaneously



13

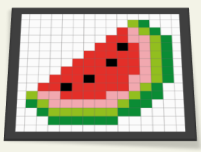
GPUs:
Throughput Oriented Design

Using GPUs would only be efficient if we have **massive number of threads** working on large amount of data

- Is this Amdahl's or Gustafson's law?

Example: In image processing, we need to process many pixels per unit of time. It is ok for each pixel taking more time (higher latency) as long as we maximize the throughput

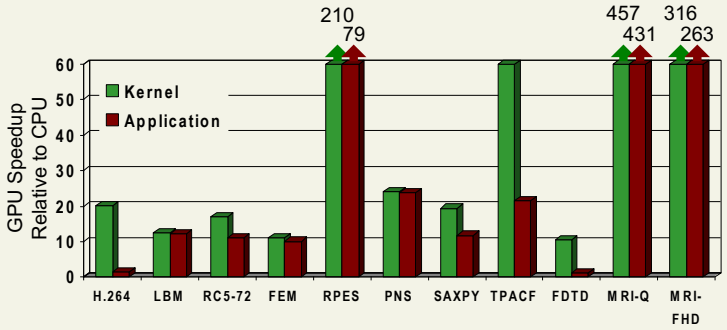
- Remember that GPUs have 100s or 1000s of cores which can finish more work jobs even if each job takes more time.
- Each thread may be assigned to only one pixel



14

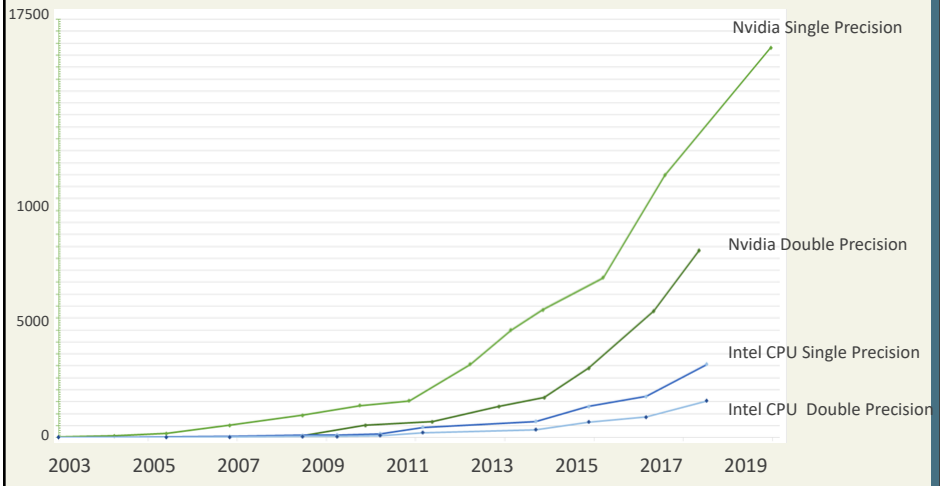
Speedup of Applications

GeForce 8800 GTX vs. 2.2GHz AMD Opteron 248 CPU



- 10× speedup in a kernel (i.e. running on GPU) is typical, as long as the kernel can **occupy enough parallel threads**
- 25× to 400× speedup if the function's data requirements and control flow suit the GPU and the application is optimized

FLOPS for CPUs vs GPUs



Source: Nvidia

GPU Computing

Winning Applications Use BOTH CPU and GPU

- CPUs for sequential parts where **latency** matters
  - CPUs can be 10+ X faster than GPUs for sequential code
- GPUs for parallel parts where **throughput** wins
  - GPUs can be 10+ X faster than CPUs for parallel code
  - GPUs are efficient in launching many threads in parallel
    - Using threads is not expensive
    - If you are not launching many threads, you should probably run your code on the CPU.

17

GPGPU

- GPGPU: General-Purpose computation on GPUs
  - aka GPU Computing
- Before 2007, GPUs were
  - Specially designed for computer graphics and difficult to program
  - Restrictive (e.g., you can only write/read "pixels" data)
- After 2007
  - General Purpose computation
  - Using industry-standard languages such as C
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

18

Differences Between GPU and CPU threads

- GPU threads are extremely lightweight
  - Very little creation overhead
- Remember that GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few

19

What is CUDA?

- CUDA = Compute Unified Device Architecture (almost!)
- A parallel computing platform and API developed by NVIDIA
  - Released in June 2007
- Programmers can use CUDA-enabled **GPU**s for **G**eneral-**P**urpose processing (**GPGPU**)
  - Programmer kicks off batches of threads on the GPU
    - GPU: dedicated super-threaded, massively data parallel co-processor
  - Can significantly increase computing performance by harnessing the power of the GPU.
- CUDA works with programming languages such as C, C++ and Fortran.

Source: http://www.nvidia.ca/object/cuda_home_new.html

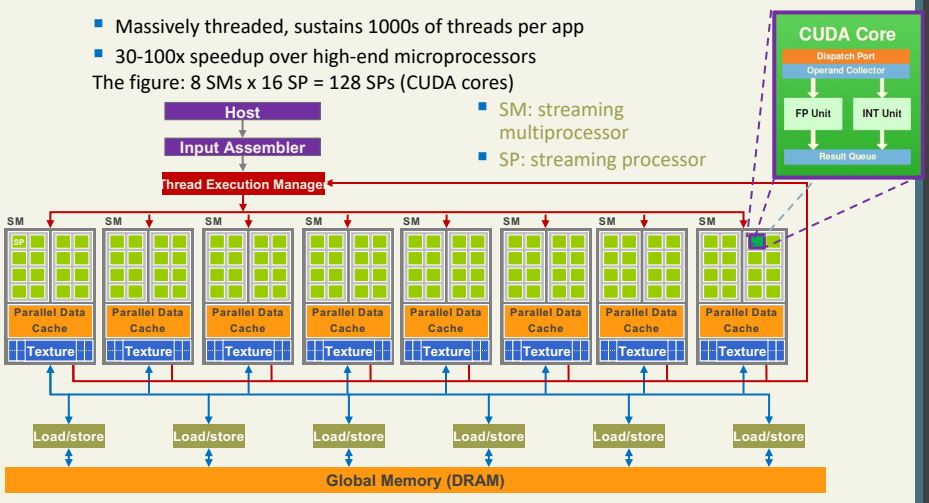
20

GPU Design

- Massively threaded, sustains 1000s of threads per app
- 30-100x speedup over high-end microprocessors

The figure: 8 SMs x 16 SP = 128 SPs (CUDA cores)

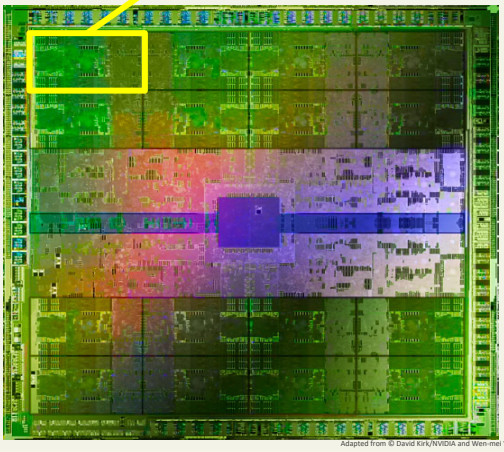
- SM: streaming multiprocessor
- SP: streaming processor



*Note: other components might exit on the GPU (e.g. **Tensor Cores**), but this is outside the scope of this course.*

NVIDIA GeForce GTX 580

512 CUDA cores (16 **SMs** x 32 SPs)



Adapted from © David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign

Comparing Cards

GTX 480

- **480** CUDA Cores
  - **15 SMs**
  - Each SM features **32 SPs** (**CUDA cores**)
    - CUDA core = one ALU (INT) + one FPU
    - # of cores determine # of calculations performed per clock cycle
  - SM executes threads in groups of 32 called *warps*
- Memory Interface: **384-bit**
- Memory Bandwidth **177.4 GB/s**
- Compute capability: 2.1

RTX 3090

- CUDA cores: **10,496**
- Memory Interface: **384-bit**
- Memory Bandwidth **936 GB/s**
- $$$$

Running CUDA Remotely

- On Google Colabs
  - Google has made available an online environment to work with CUDA C/C++ on their GPUs for free.
  - The steps on how to do that are detailed here:
    - https://www.wikihow.com/Run-CUDA-C-or-C%2B%2B-on-Jupyter-(Google-Colab)

Great Reference

NVIDIA official **CUDA Toolkit Documentation v10.2.89**
https://docs.nvidia.com/cuda/

Specially:

- Installation Guides
- CUDA C++ Programming Guide
  https://docs.nvidia.com/cuda/cuda-c-programming-guide
- Best Practices
  https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- Samples
  https://docs.nvidia.com/cuda/cuda-samples/index.html

25

CUDA Terminology

Host: the **CPU** (e.g., intel Core i3)

- ***Host code***: the code that runs on the CPU

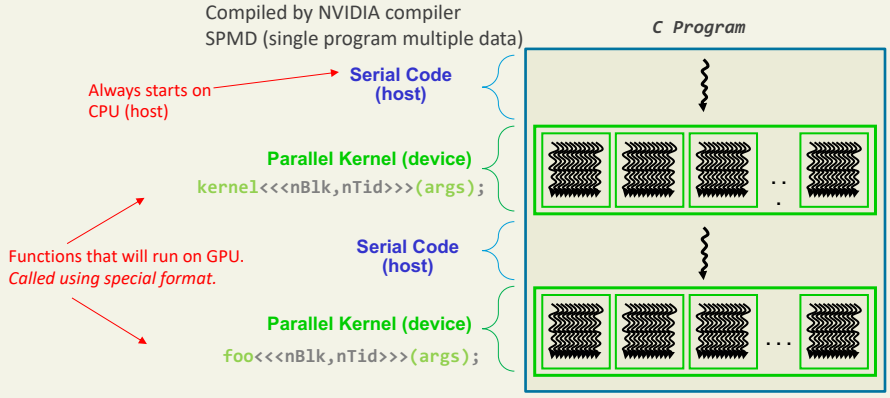
Device: a coprocessor to the CPU

- Typically a **GPU**
  - Can also be other types of parallel processing devices
- Has its own DRAM (***device memory***)
- Runs many threads in parallel
  - ***Thread*** is the concurrent code executed on the CUDA device in parallel with other threads.
- ***Kernel code*** is the data-parallel portions of an application which run on the ***device***

26

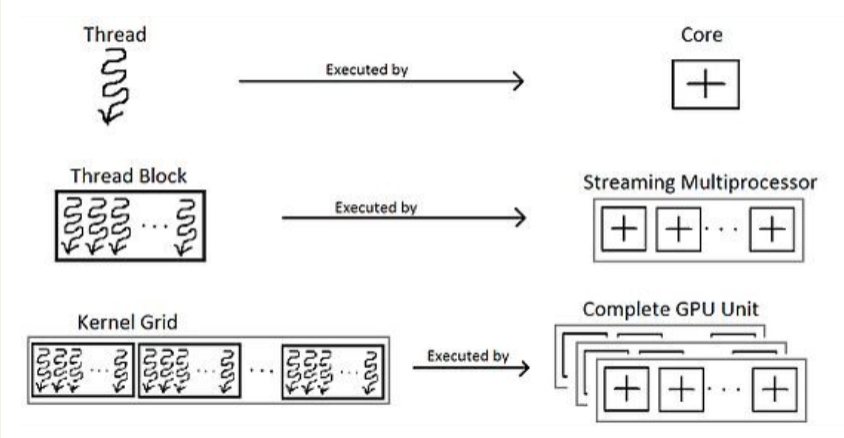
Program Structure

C Program = **host code** (runs on CPU) + **device code** (runs on GPU)
- **Serial or modestly parallel parts → in host code**
  - Compiled by host standard compiler
- **Highly parallel parts → in device SPMD kernel code**
  - Compiled by NVIDIA compiler
  - SPMD (single program multiple data)



27

Threads, Blocks and Grids

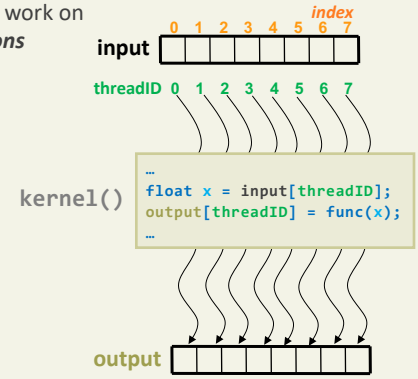


https://medium.com/analytics-vidhya/cuda-compute-unified-device-architecture-part-2-f3841c25375e

28

Arrays of Parallel Threads

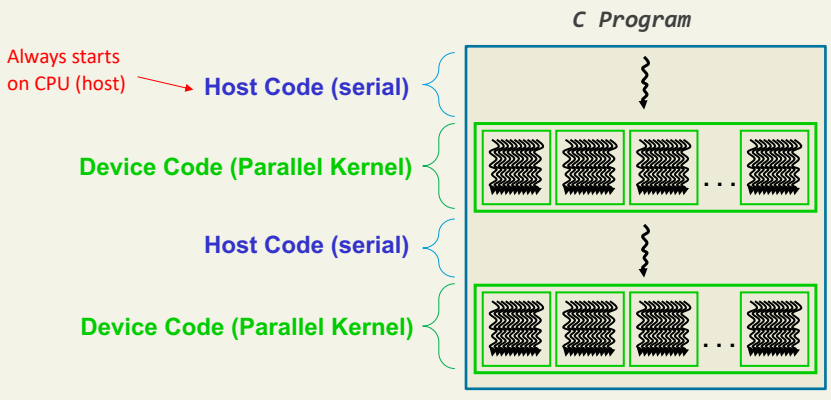
- A CUDA kernel is executed by an array of threads
  - All threads **run the same code** (SPMD)
    - SPMD: Single Program, Multiple Data
  - **Thread *IDs*** are used to
    - Decide what ***data*** to work on
    - Make ***control decisions***



29

How to Write a CUDA Program

C Program = **host code** (runs on CPU) + **device code** (runs on GPU)



30

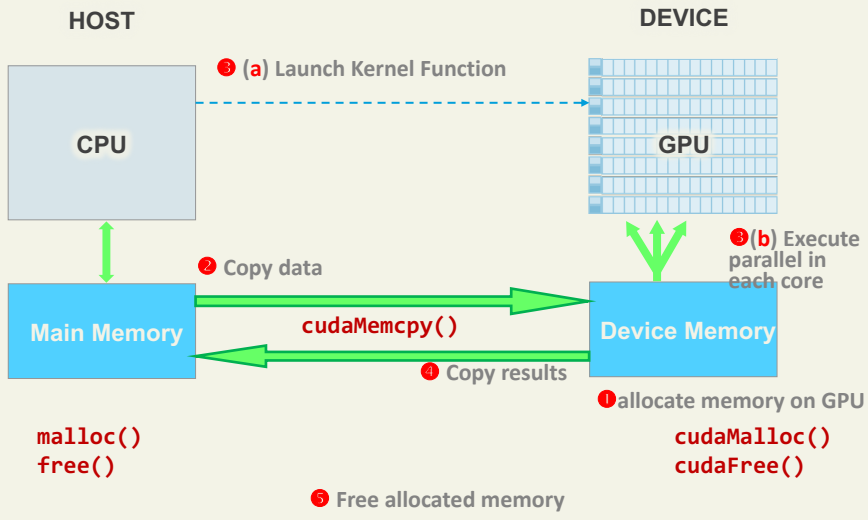
The Host Code

Step #1

1. Allocate space on the GPU using `cudaMalloc`
2. Copy CPU data to the GPU memory using `cudaMemcpy`
   - CPU accesses its own memory, and GPU accesses its own memory
3. Launch the ***kernel function(s)*** on the GPU
   - Might need to define the ***launch-configuration***
   - **Pass reference** to allocated GPU memory **+** any arguments.
   - Results are stored on GPU memory.
4. Copy the results from GPU to CPU using `cudaMemcpy`
   - Must have memory allocated on CPU ram = the size of the results.
5. Free GPU memory using `cudaFree`

31

Processing Flow on CUDA



32

A HOST Example

```
int *a = 0, *d_a = 0, num_bytes = ...;

// Prepare data on CPU
a = malloc(num_bytes);//could also have other data b,c,d,...
    ... // put into a the data that need to be processed on GPU

// Allocate memory on GPU + Copy CPU data to GPU
cudaMalloc(&d_a, num_bytes ); //allocate memory on GPU before copy
cudaMemcpy(d_a, a, num_bytes, cudaMemcpyHostToDevice );

// Process data on GPU
...// invoke the kernel function on the GPU here

// Copy results from GPU to CPU
cudaMemcpy(a, d_a, num_bytes, cudaMemcpyDeviceToHost );

// Free CPU & GPU memory
cudaFree(d_a);
free(a);
```


The Kernel Code

Step 2

- Write the kernel function **as if it will run on a single thread**
  - Use IDs to identify which **piece of data** is processed by this thread.
    - The kernel function will run on only one piece of the data
    - Remember that this SAME kernel function is executed by many threads
- This means **parallelism of threads is expressed in the host code**.
- Kernel functions must be declared with a qualifier, either `__global__` or `__device__`

```
__global__ void foo(int a,int b){...}
```

Launching the Kernel

To launch a CUDA kernel from the host, we need to

- Specify the block dimension
- Specify the Grid dimension

- Recall
  - Threads are organized into blocks
  - Blocks are organized into grids
- When launching the Kernel we will pass into it (more later…)
  - Number of blocks in the grid to process
    - Based on your data size
    - Data needs be covered -> need to make sure that you have one thread for each element of data
  - Number of threads in block (check card stats)

```
foo<<<1,1024>>>(a, b);  //run N threads on 1 block
```

Grid size (number of blocks) Block size (threads per block)

35

Built-in CUDA Functions

```
cudaError_t cudaMalloc(void** d_ptr, size_t n)
```

- Allocates *n* bytes of linear memory on the device and returns in *d_ptr a pointer to the allocated memory.
- *d_ptr*: address of a pointer to the allocated device memory
  - We need to pass this pointer by-reference (i.e., pass its address) so that cudaMalloc is allowed to modify its value and store the pointer to the allocated memory
- *n*: size of requested memory in bytes.

```
cudaError_t cudaFree(void* d_Ptr)
```

- Frees memory on device pointed at by d_ptr

Note that CUDA functions return an error code if anything goes wrong.

More about this later

36

Built-in CUDA Functions

`cudaError_t cudaMemcpy(void *dst, void *src,size_t n, dir)`

- Copies data between host / device.
  - *dst*/*src* are pointers to destination/source memory segments
  - *n* is number of bytes to copy
  - *dir* is kind of transfer, and it might have different values including:
    - cudaMemcpyDefault
      - (recommended, but not supported on all machines)
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice
    - cudaMemcpyHostToDevice
- Starts copying **after** previous CUDA calls complete
- CPU thread is blocked until copy is complete after which method returns.

37

Built-in CUDA Functions

`cudaError_t cudaMemset(void* d_ptr, int value, size_t n)`

- Fills the first *n* bytes of the memory area pointed to by *d_ptr* with a constant value *value*
  - Note that we are filling the first ***n*** bytes, not ***n*** integers or any other type!

38

Function Declarations

	Executed on	callable from
`__global__ void KernelFunc()`	device (GPU)	host (CPU) **
`__device__ float DeviceFunc()`	device (GPU)	device (GPU)
`__host__ float HostFunc()`	host (CPU)	host (CPU)

`__global__`
Called from CPU (**). It must **return void**

`__device__`
Called and executed by GPU. Cannot be called from CPU
i.e. called from other `__global__` and `__device__` functions

`__host__`
called and executed by CPU

`__device__` and `__host__` can be used together

```
__host__ __device__ int max(int a, int b){return (a>b)?a:b;}
```

This means both CPU and GPU can call *max*

(**) *Dynamic parallelism allows calling kernels from within other kernels* on cards of compute 3.5 or higher. This is outside the scope of this course, and we will always call kernels from host code.

Limitations

For functions executed on the GPU:

- No recursion
- No static variable declarations inside the function
- No variable number of arguments
- Only pointers to GPU memory can be dereferenced

Add Two Numbers

Example 1: Serial on CPU

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

int main (void) {                   // runs on the host
        int c;                      // c is on the host DRAM


        // execute add on the host
        add (2, 7, &c);   // save results in c on the host DRAM


        printf( "2 + 7 = %d\n", c);
        return 0;
}
```


41

Add Two Numbers

Example 1: On the GPU (Parallel)

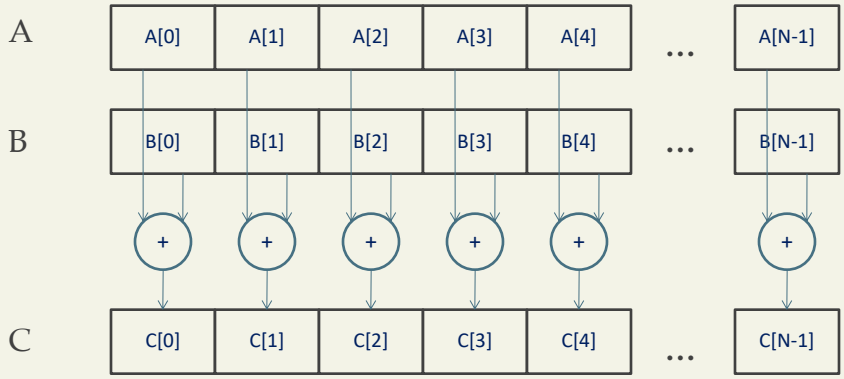
```
__global__ void add(int a,int b,int *c){  //runs on the device
        *c = a + b;
}

int main (void) {                   // runs on the host
        int c;                      // c is on the host DRAM
        // allocate memory on the device (no data to copy to the device for now)
        int *d_c;
        cudaMalloc(&d_c, sizeof(int) );

        // execute add on the device (also pass data to device memory as arguments)
        add<<<1,1>>>(2, 7, d_c);      //save results on device DRAM
        // copy results back from the device memory to the host memory
        cudaMemcpy(&c, d_c, sizeof(int),cudaMemcpyDeviceToHost);
        // free the device memory and process the results from the device
        cudaFree(d_c);
        printf( "2 + 7 = %d\n", c);
        return 0;
}
```


42

Vector Addition: Conceptual View



43

Vector Addition

Example 2: Serial

```
#define N 1024
void vectorAdd(int* a, int* b, int* c, int n) {
   int i;
   for (i = 0; i < n; i++)
         c[i] = a[i] + b[i];
}

int main() {
   int *a, *b, *c;
   a = malloc(N * sizeof(int));       //create three arrays
   b = malloc(N * sizeof(int));
   c = malloc(N * sizeof(int));

   for(int i = 0; i < N; i++)         //intialize a,b (for testing)
         a[i] = b[i] = i;

   vectorAdd(a, b, c, N);             // vector addition

   for(int i = 0; i < 10; i++)        // print first 10 elements(for testing)
         printf("c[%d] = %d\n", i, c[i]);

   free(a);free(b);free(c);           // free memory taken by a, b, c
   return 0;
}
```


44

Vector Addition

Example 2: Parallel on GPU

Step1: parallelize the function

- __global__ to the function header
  - Tells the compiler that the function is executable on the GPU and callable from the host
- We need a way to divide the data among threads.
  - The same function is going to be executed by many threads
  - So we need a way to divide the data based on the thread ID
- Make sure you don't run on invalid data range

```
__global__ void vectorAdd(int* a, int* b, int* c, int n) {
    int i = threadIdx.x;                    //threadIdx.x is read only variable
   if(i<n)        //in case we have more threads than the number of elements
        c[i] = a[i] + b[i];
}
```


45

Vector Addition

Example 2: Parallel on GPU

Step2: modify your code to run the function on the device

- Allocate memory on the GPU ***and*** copy the data to the GPU memory
- Launch the kernel function (using a specific number of threads)
- Copy the results back from GPU to CPU memory to print the results.

```
int main() {
   int *a, *b, *c;                          //pointers to host memory

   int *d_A, *d_B, *d_C;          //pointers to device memory
    a = malloc(N * sizeof(int));  //allocate space on host
    b = malloc(N * sizeof(int));
    c = malloc(N * sizeof(int));

    //allocate space on device
    cudaMalloc(&d_A, N * sizeof(int));
    cudaMalloc(&d_B, N * sizeof(int));
    cudaMalloc(&d_C, N * sizeof(int));

    for(int i = 0; i < N; i++)     //initialize a, b (for testing)
         a[i] = b[i] = i;
```


46

Vector Addition

Example 2: Parallel on GPU

```
    //copy data from host to device
    cudaMemcpy(d_A, a, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, b, N * sizeof(int), cudaMemcpyHostToDevice);

    //launch the kernel (with pointers to device memory)
    vectorAdd<<<1,N>>>(d_A, d_B, d_C, N);//run N threads on 1 block

    //copy results from device to host
    cudaMemcpy(c, d_C, N * sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0; i<10; i++)       // print first 10 elements(for testing)
         printf("c[%d] = %d\n", i, c[i]);

    free(a);free(b);free(c); // free host memory taken by a, b, c

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

    return 0;
}
```


Example 3

- Following is a simple C program that:
  - Creates 2 arrays A,B on the host and 2 arrays d_A, d_B on the GPU device (all float)
  - Initialize A to any float values
  - Copy A to d_A
  - Copy d_A to d_B
  - Copy d_B to B
  - Make sure A is equal to B (use assert)
  - Free memory taken by all four arrays
- Let's trace the code and see how the data moves

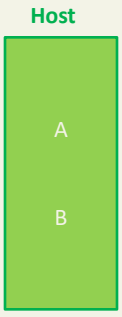
Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```


49

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

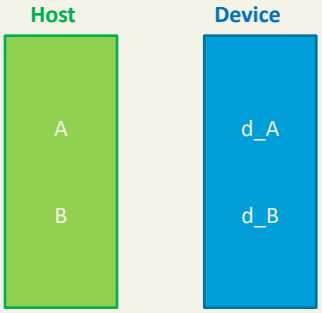


Creates 2 arrays A,B on the host.

50

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

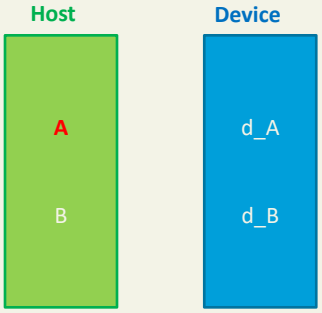


Creates 2 arrays A,B on the host and 2 arrays d_A, d_B on the GPU device (all float)

51

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

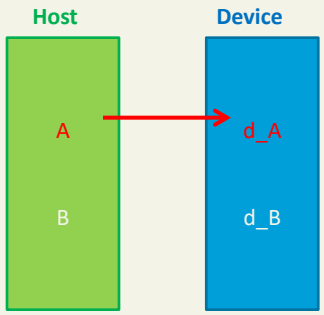


Initialize A to any float values

52

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

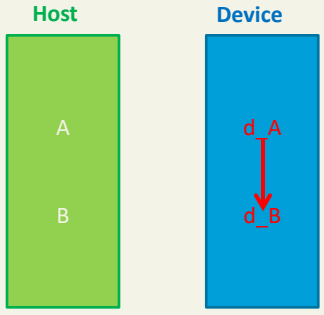


Copy A to d_A
Copy d_A to d_B
Copy d_B to B

53

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

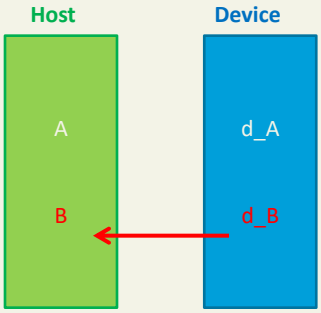


Copy A to d_A
Copy d_A to d_B
Copy d_B to B

54

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

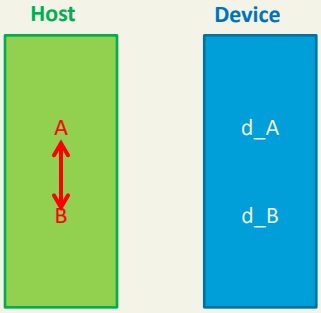


Copy A to d_A
Copy d_A to d_B
Copy d_B to B

55

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```

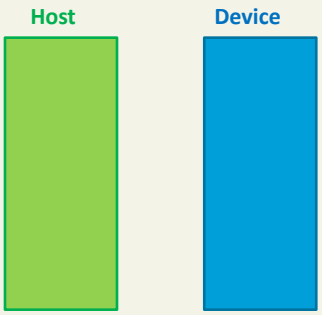


Make sure A is equal to B (use assert)

56

Example 3

```
int main(){
   float *A, *B;  // host data
   float *d_A, *d_B;  // device data
   int N = 14, i ;
   int nBytes = N * sizeof(float);
   A = (float*)malloc(nBytes);
   B = (float*)malloc(nBytes);
   cudaMalloc(&d_A, nBytes);
   cudaMalloc(&d_B, nBytes);
   for (i=0; i<N; i++) A[i] = 100.f + i;
   cudaMemcpy(d_A, A, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, d_A, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(B, d_B, nBytes, cudaMemcpyDeviceToHost);
   for (i=0; i< N; i++) assert( A[i] == B[i] );
   free(A); free(B); cudaFree(d_A); cudaFree(d_B);
   return 0;
}
```



Free memory taken by all four arrays

57

What We Covered

Today's topics:

- Intro to GPU programming
  - CPU vs GPU programming
  - Latency vs. Throughput
- CUDA basics: the hardware layout
- CUDA basics: program structure
- Kernel Launch
- Useful Built-in CUDA functions
- Function Declarations (global, device, host)
- Simple examples

Next Lecture:

- Error Handling
- cudaDeviceSynchronize
- Thread organization

58