High Performance Computing and GPU Programming

Lecture 1: Introduction

Objectives

C++/CPU Review

GPU Intro

Programming Model

Objectives

Objectives

- Before we begin…a little motivation

Intel Xeon 2.67GHz
8 Cores
0.85 secs/iteration

Tesla C1070
240 CUDA cores
0.057 secs/iteration

Objectives

Intel Xeon 2.67GHz	Tesla C2070
4 CPUs – 32 Cores	4 GPUs – 1,792 Cores
~145 days	~6 days

Objectives

- I have three goals
  - Develop good programming skills and practices
  - Understand GPU programming and architecture
  - Achieve peak performance through code optimization

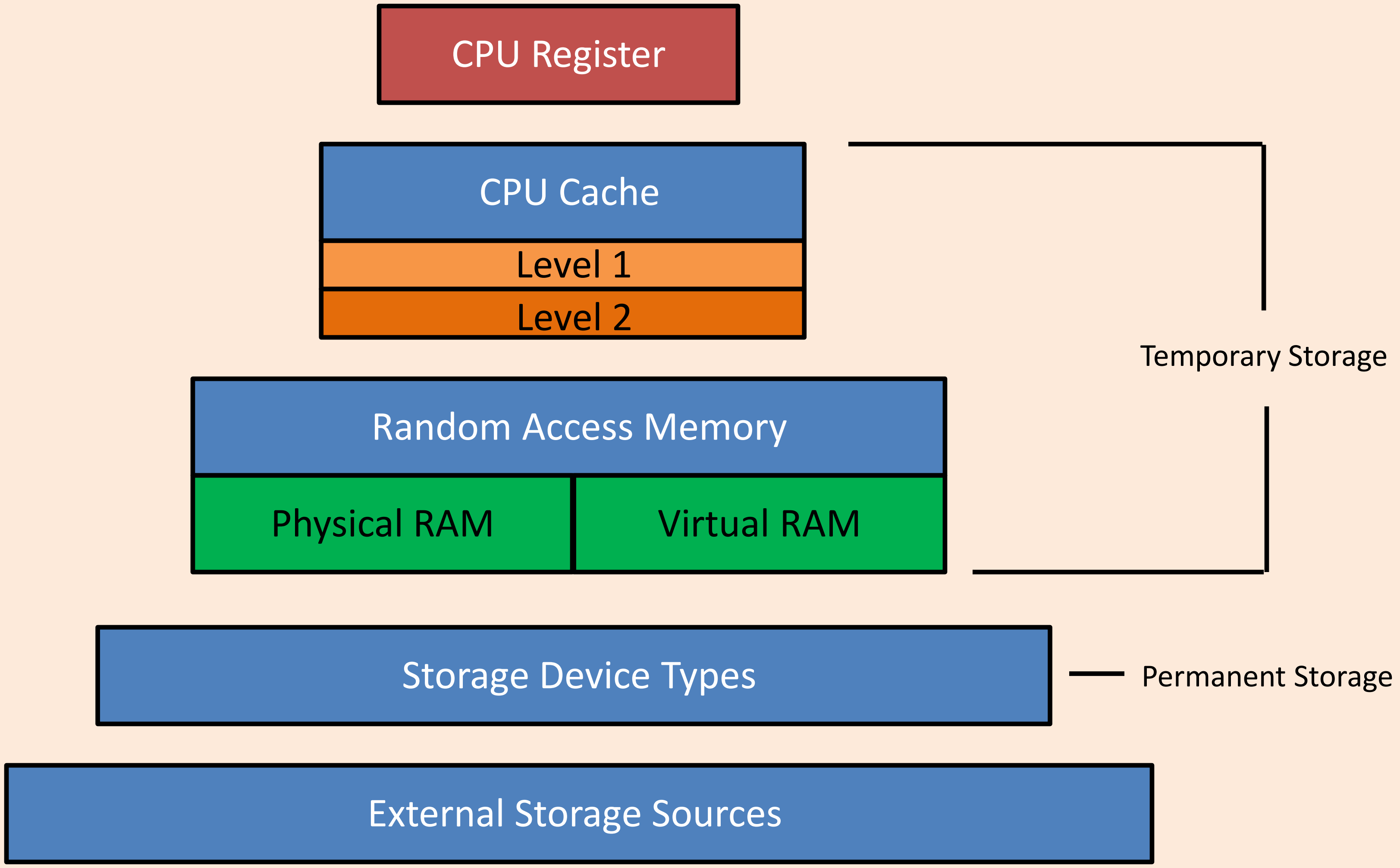
Objectives

- To reach these goals you need
  - Understanding of pointers and computer memory
  - Knowledge of some computer language
    - C, C++, FORTRAN – I prefer C++
  - Patience
    - Learning GPU computing can be VERY frustrating

C++/CPU Review

C++/CPU Review

- How does computer memory work?



C++/CPU Review

- Why is this important?
  - Your computer takes time to access data
  - Your computer takes time to operate of data

Intel Processor	
Instruction	**Latency (clocks)**
FABS	3
FADD	6
FSUB	6
FMULT	8
FDIV (S)	30
FDIV (D)	44

C++/CPU Review

- Processor stalling
  - Idling occurs when a processor cannot execute the next instruction due to dependency

Intel Nehalem Processor	
Memory	Latency (clocks)
Main memory	~100
L1	~4
L2	~10
L3	~40

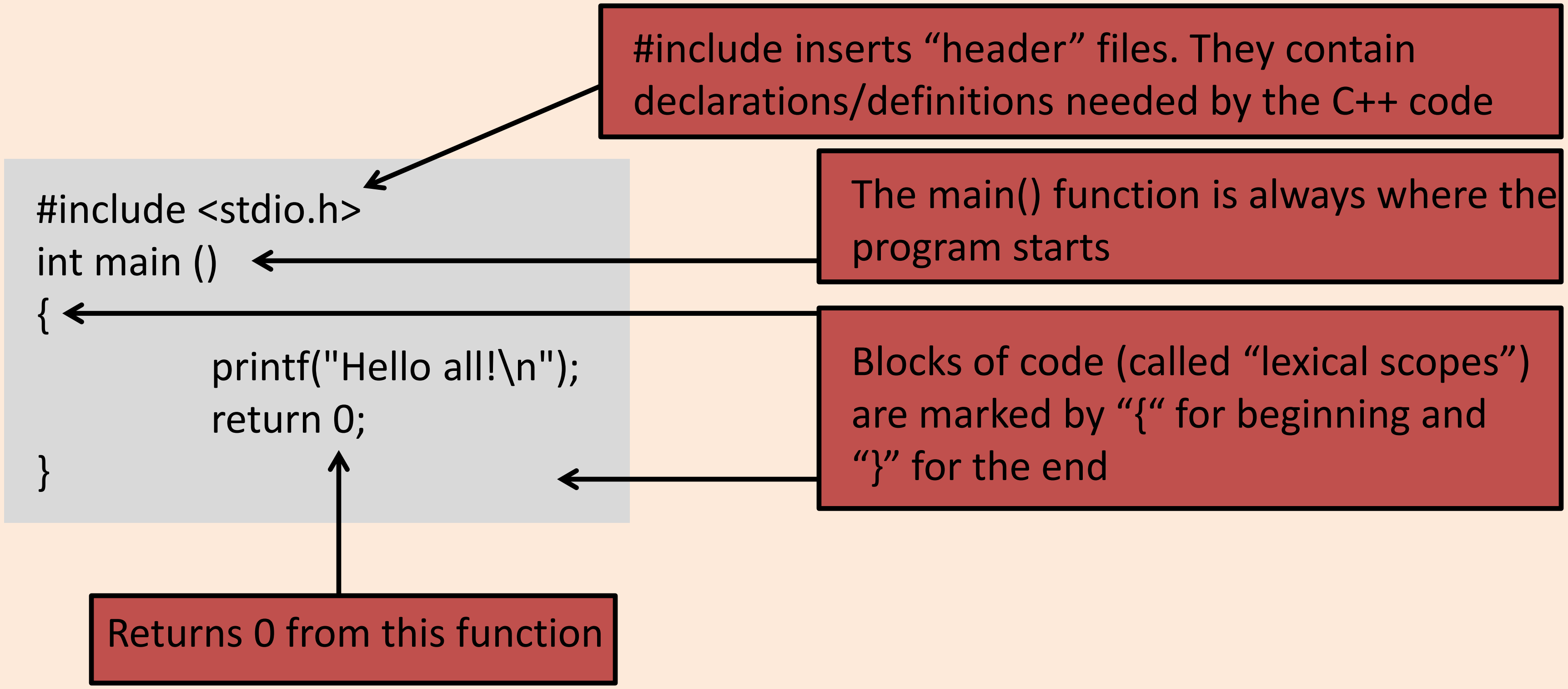
  - CPU’s try to “hide” latency in caches
    - Cache – store subset of data closer to processing elements for fast access
    - Latency – you are hungry, place an order, you are served 30 mins later. The latency is 30 minutes.

C++/CPU Review

- Latency depends on where the food is
  - If stored in your mini-fridge by your desk – fast access (this is on-chip storage)
  - If you want a burger at the hub, you got to go get it (this is off-chip storage)

- Once you have the food
  - Bandwidth
    - How much can you scarf down in 1 second?
    - Depends on food organization in storage and what food you have
    - Also – How are you eating your food?

C++/CPU Review



#include inserts "header" files. They contain declarations/definitions needed by the C++ code

```
#include <stdio.h>
int main ()
{
        printf("Hello all!\n");
        return 0;
}
```

The main() function is always where the program starts

Blocks of code (called "lexical scopes") are marked by "{" for beginning and "}" for the end

Returns 0 from this function

- Very basic C++ code
- Remember – C++ starts at 0 … NOT 1!
  - First element in array A is A[0], NOT A[1]

C++/CPU Review

- Pointers
  - Why do we need them?
    - Memory management
    - GPU computing REQUIRES them

```
#define n 16
int main() {
    //Define the pointer type
    double *a, **b;
    //Allocate them
    a = new double[n];
    b = new double*[n];
    for (int i=0; i<n; i++) b[i] = new double[n]

    //Operate with them
    for (int i=0; i<n; i++) {
        a[i] = ...
        b[0][i] = ...
        b[1][i] = ...
    }
}
```

C++/CPU Review

- Pointers
  - What do they do?
    - Point to memory location where information can be found
    - Allow dynamic allocation – On the fly memory management
    - Can freely pass into functions an operate on them
  - In the case of large data storage
    - Array to function
      - Pass ALL data into function – Takes time
    - Pointer to function
      - Passes only memory LOCATION to function
      - Much faster!

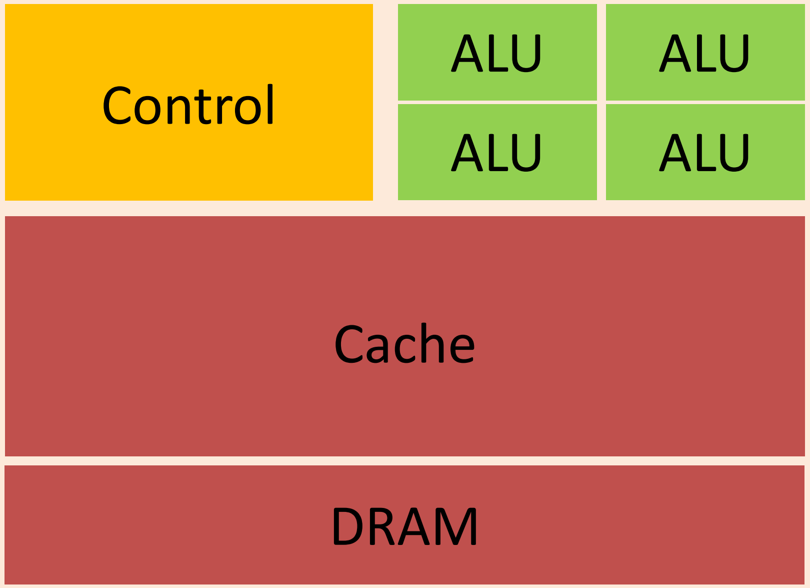
C++/CPU Review

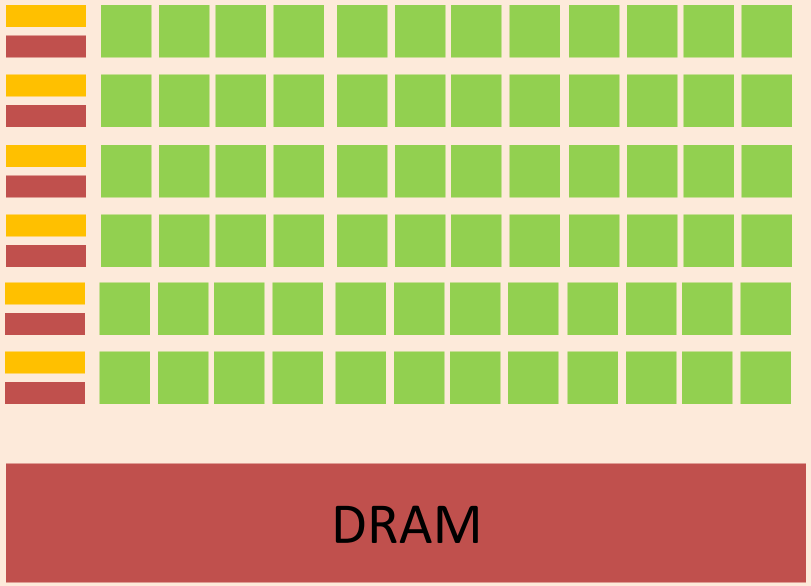
- Storage
  - 1-D storage of all arrays
  - Multi-dimensional arrays are more convenience
  - With GPU’s, multi-dimensional arrays are:
    - More hassle
    - Slower
  - Conversion is simple

GPU Intro

GPU Intro

- CPU vs. GPU



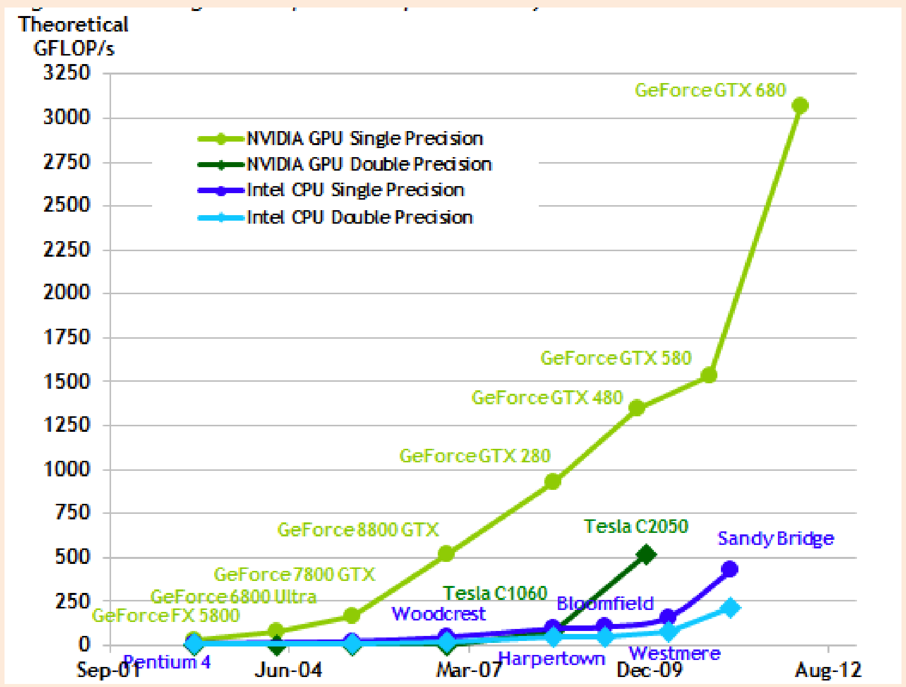


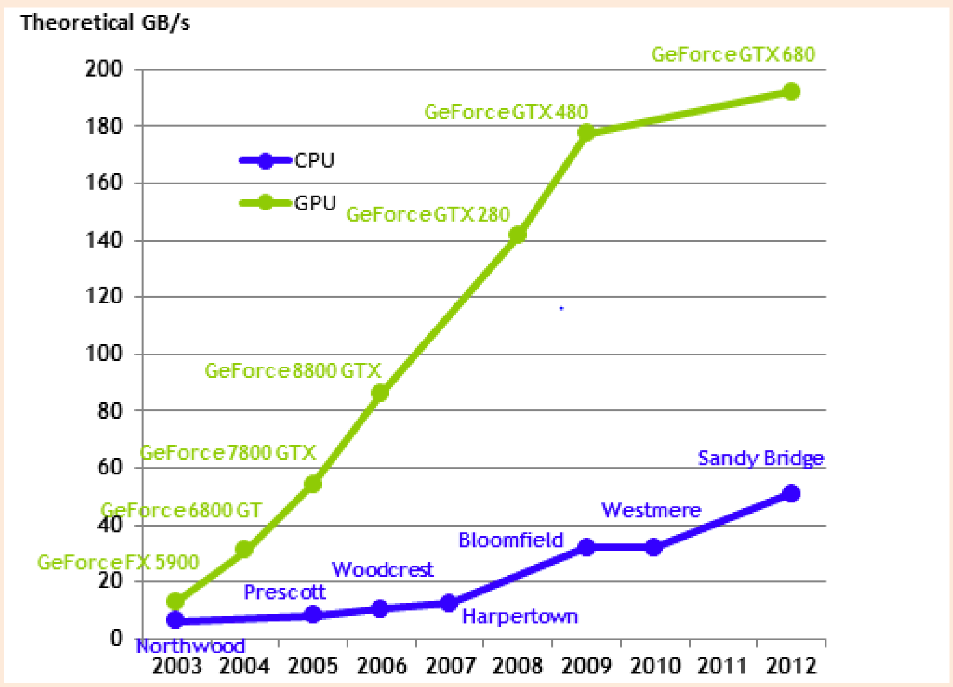
CPU
SIMD – Single instruction multiple data vector units

GPU
SIMT – Single instruction multiple threads

GPU Intro

- CPU vs. GPU





Floating point operations per second

Memory Bandwidth

GPU Intro

- CPU vs. GPU

	GPU – Tesla K20	CPU – Intel I7
Cores	Main memory	4 (8)
Memory	5 GB	32 KB L1 cache / core 256 KB L2 cache / core 8 MB L3 shared
Clock Speed	2.6 GHz	3.2 GHz
Bandwidth	208 GB/s	25.6 GB/s
FLOPS	1.17 x 10^12	70 x 10^9

GPU Intro

- Why the difference?
  - GPU specialized for compute intensive highly parallel computation – graphics rendering
  - More devoted to data processing rather than caching
- For CPU parallelism – Rely on:
  - Parallel libraries – Convenient
  - Compiler – Very hard
- For GPU parallelism
  - Language extension requiring human interaction
  - YOU must generate the parallel executable

GPU Intro

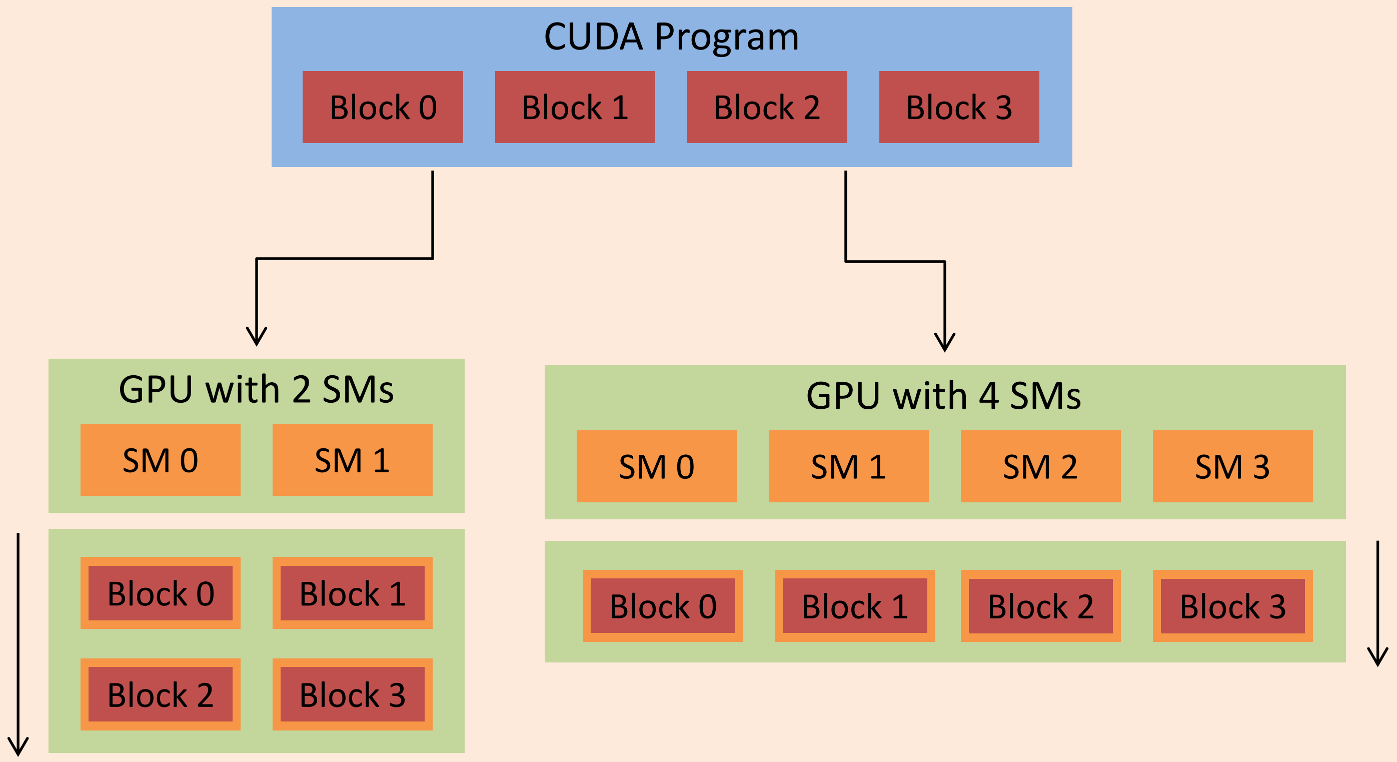
- Another fact:
  - Everything costs $$$
  - AFRL supercomputer
    - > $100,000
    - Plus a ridiculous amount of power consumption
  - GPU Tesla K20
    - ~ $5,000
    - Put it in your desktop – with a large power supply
  - GPU GTX 660
    - ~ $200
    - Cheap AND fast – Not a lot to work with however

Programming Model

Programming Model

- Compute Unified Device Architecture
  - NVIDIA CUDA GPU Programming
  - Not exactly High Performance Computing (HPC) – Shares same aspect
    - Combine multiple GPUs together – HPC GPU cluster
  - Data is executed over many threads in parallel
  - Controlled with:
    - Grid
    - Blocks
    - Threads

Programming Model

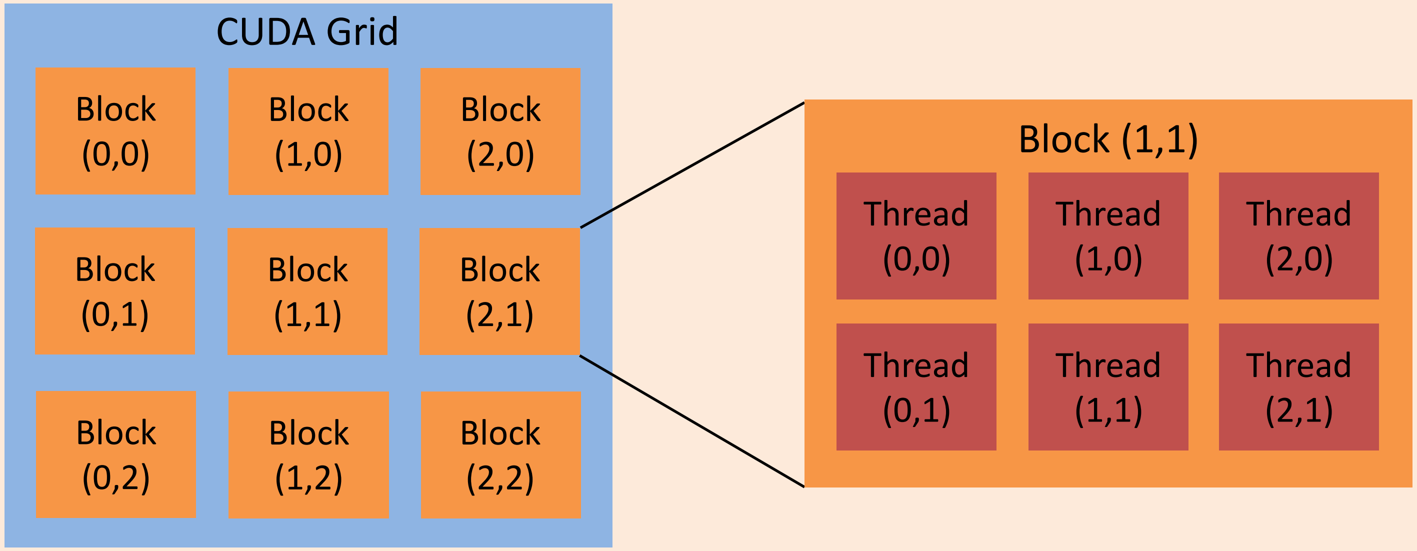


- SM creates, manages, schedules, and executes threads in groups of parallel threads - Warps

Programming Model

- Warps are not easy
  - Warp size is 32 threads on current GPUs
  - Threads in a warp start together
  - When an SM has a task or block:
    - The threads are split into warps
    - A sort of “scheduling” is done
- Most of the time we have to ignore this
  - Not all problems fit into a multiple of 32!
  - Many papers claim 500x speed-up for matrix operations
    - The cases are for sizes of 32x32, 256x256, 512x512, ect…

Programming Model



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Programming Model

- Memory must be allocated

```
// Host code
int main()
{
    // Allocate input vectors in host memory
    float* A_h = (float*)malloc(N * sizeof(float));
    float* B_h = (float*)malloc(N * sizeof(float));

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* A_d, B_d, C_d;
    cudaMalloc(&A_d, N * sizeof(float));
    cudaMalloc(&B_d, N * sizeof(float));
    cudaMalloc(&C_d, N * sizeof(float));

    // Copy vectors from host memory to device memory
    cudaMemcpy(A_d, A_h, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, N * sizeof(float), cudaMemcpyHostToDevice);

    // Invoke kernel
    VecAdd<<<1, N>>>(A_d, B_d, C_d, N);

    // Copy result from device memory to host memory
    cudaMemcpy(C_h, C_d, N * sizeof(float), cudaMemcpyDeviceToHost);

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

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Programming Model

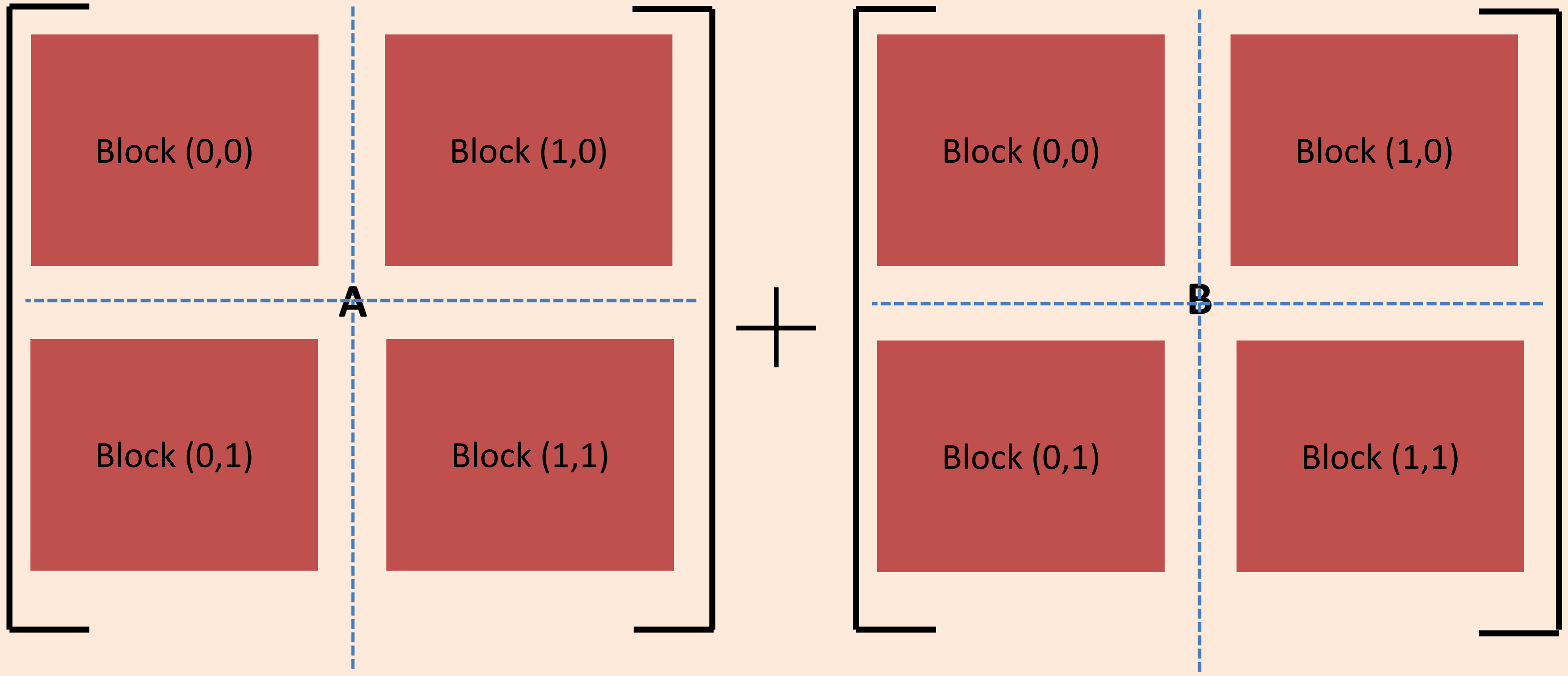
- Thread Hierarchy
  - Controlled by “dim3” declaration
  - Threads have a limit!

```
// 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);
    ...
}
```

Programming Model

- Now consider a multi-block multi-thread problem

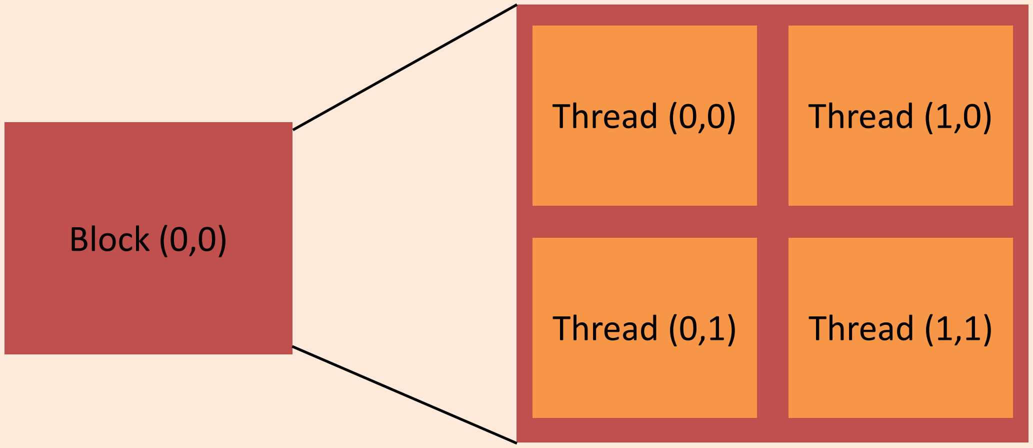


- Break it up!

Programming Model

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

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(2, 2);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



	Threads	Blocks	Grid
Idx.x	0,1	0,1	-
Idx.y	0,1	0,1	-
Dim.x	-	2	2
Dim.y	-	2	2

Wrap Up

- Next time…
  - CUDA for you
    - What you need, where to get it, how to install it
  - Thread index mapping
    - 2-D or 3-D to 1-D
  - Introduce CUDA memory types
    - Texture, local, global, shared
  - Program interpolation function (if time permits it)
    - CPU vs. GPU implementation