INTRODUCTION TO

PARALLEL PROGRAMMING

Davood Mohajerani, Amirali Sharifian

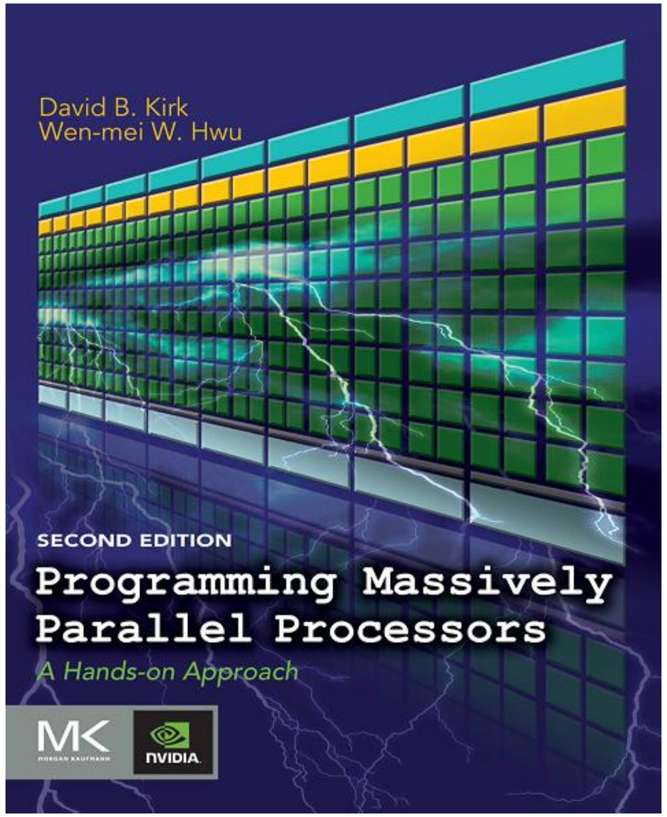
Isfahan University of Technology

Course Goals

- Learn how to program a heterogeneous parallel platform
- Parallel Programming API, tools and techniques
- Principles and patterns of parallel algorithms
- Processor Architecture features and constraints

Recommended books

- **"Programming Massively Parallel Processors - A Hands-on Approach"**
  2nd Edition, D. Kirk and W. Hwu
  Morgan Kaufman Publisher, 2013



Recommended books

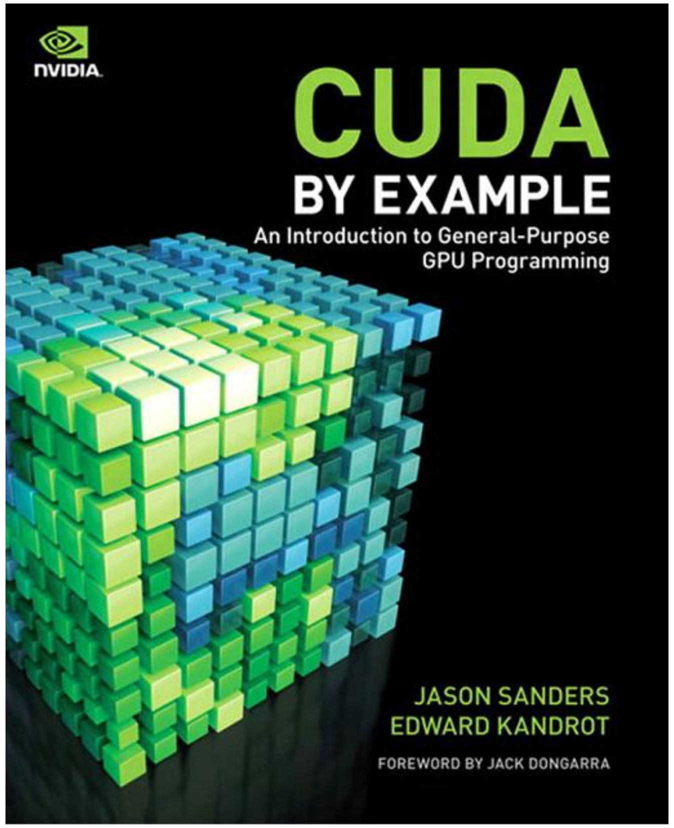
- **“NVidia CUDA C Programming Guide, version 5.0**
  NVidia's (reference book)

Recommended books

- **“CUDA by Example: An Introduction to General-Purpose GPU Programming”**

  Jason Sanders and Edward Kandro

  Addison-Wesley Professional, 2010


NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA

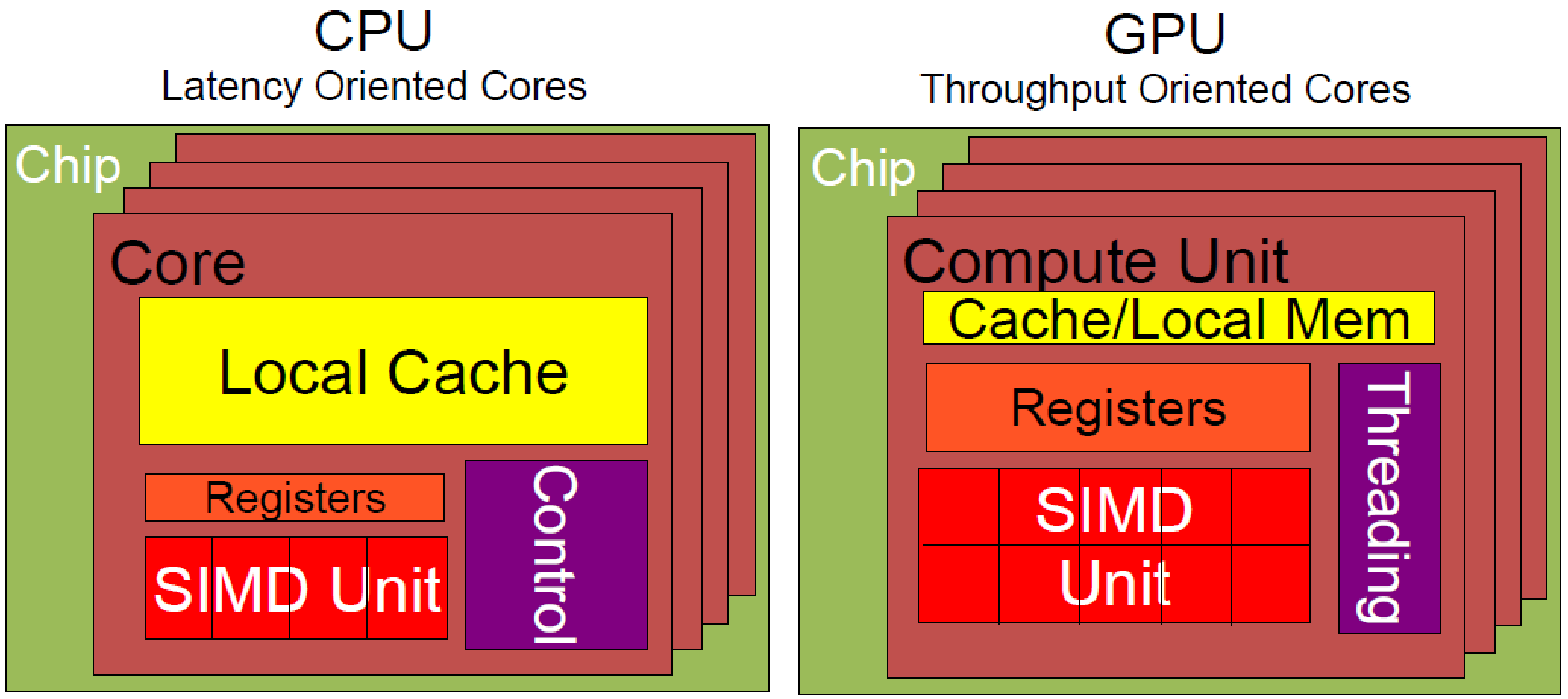
Part1

Overview

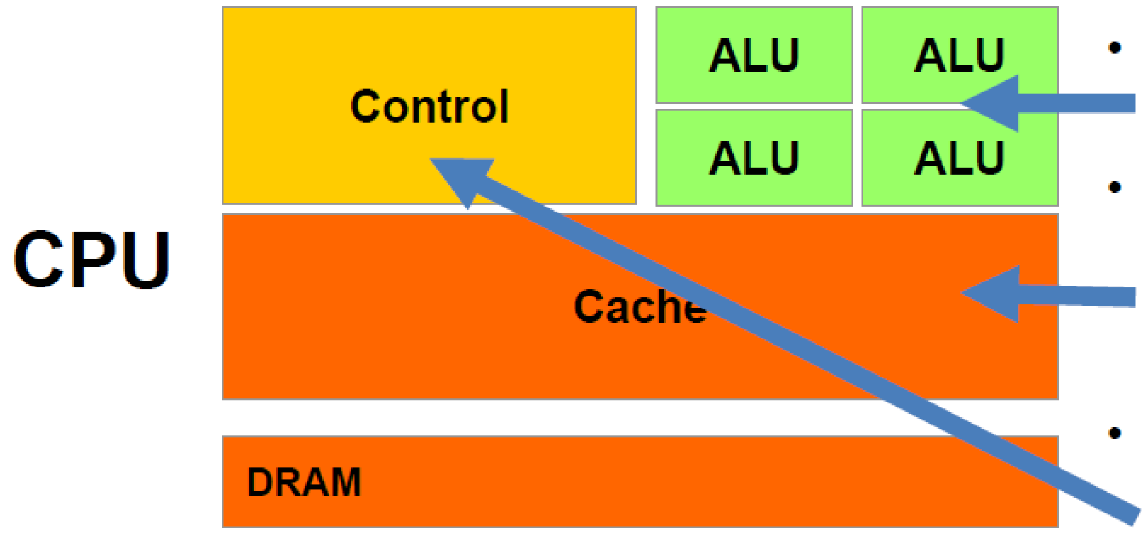
Objective

- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

CPU and GPU are designed very differently

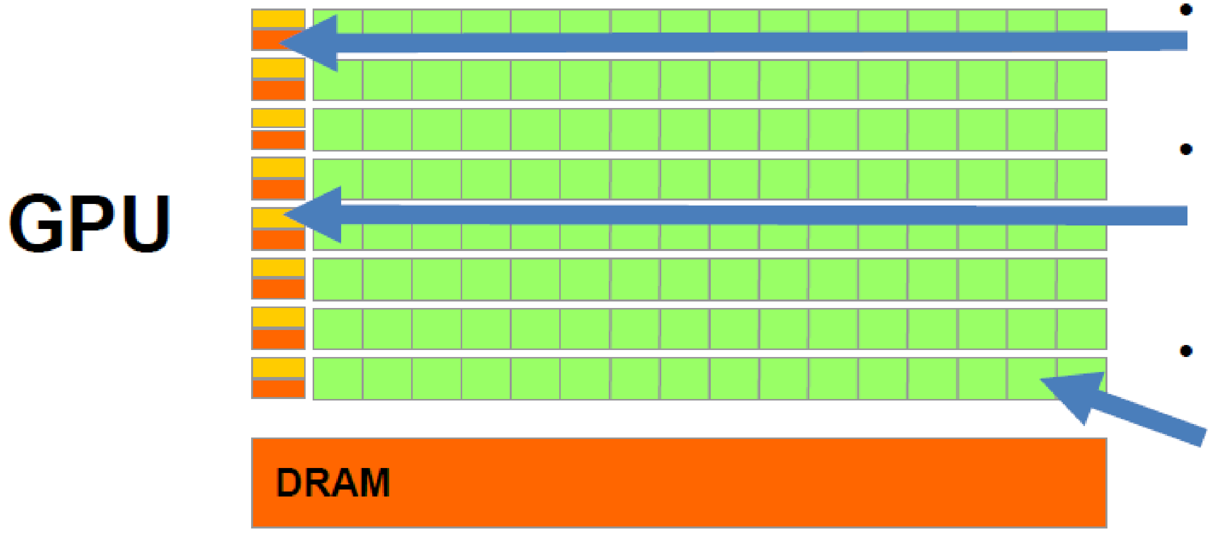


CPU: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies

Winning applications use both CPU & 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
  - PUGs can be 10+X faster than CPUs for parallel code

Heterogeneous parallel computing is

- Financial Analysis
- Scientific Simulation
- Engineering Simulation
- Data Intensive Analytics
- Medical Imaging
- Digital Audio Processing
- Digital Video Processing
- Computer Vision
- Biomedical Informatics
- Electronic Design Automation
- Statistical Modeling
- Numerical Methods
- Ray Tracing Rendering
- Interactive Physics

Part2

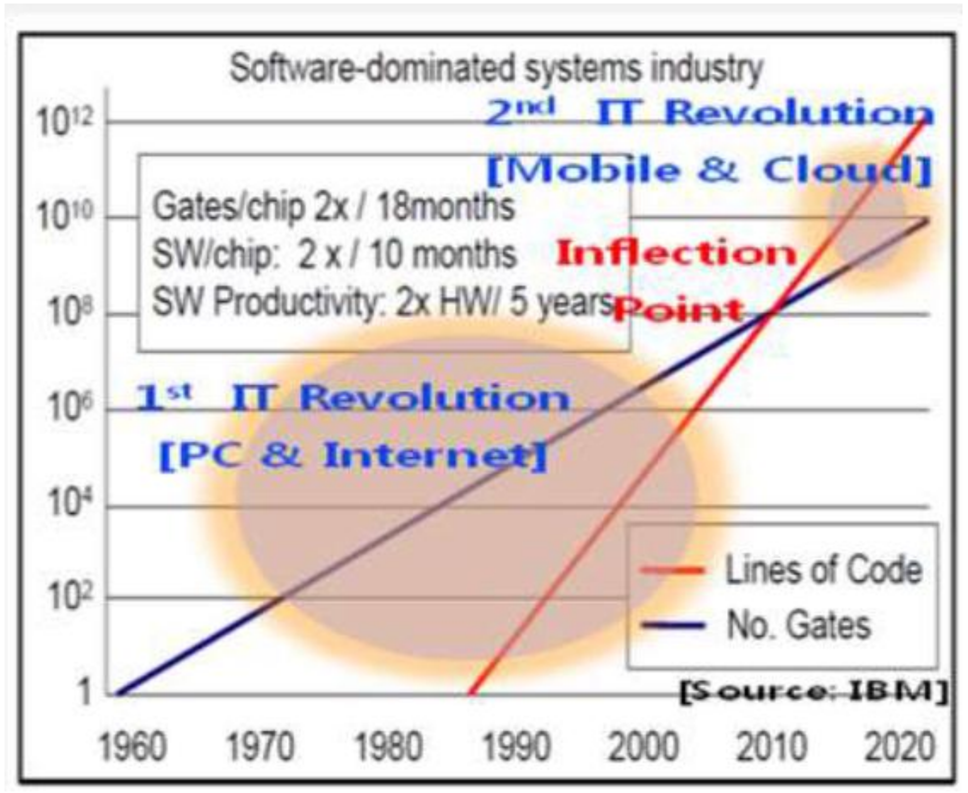
Portability And Scalability

Objectives

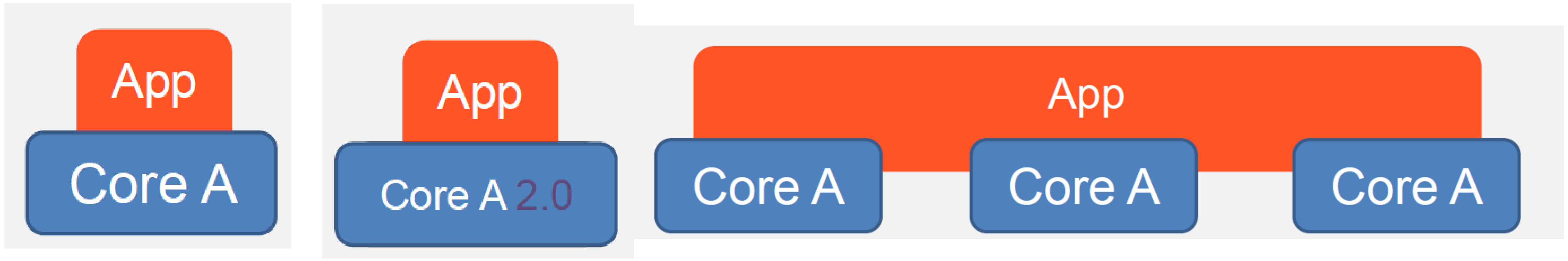
- To understand the importance and nature of scalability and portability in parallel programming

Software Dominates system cost

- SW lines per chip increases at 2x/10 months
- HW gates per chip increases at 2X/18 months(Moore's law)
- Future system must minimize software redevelopment



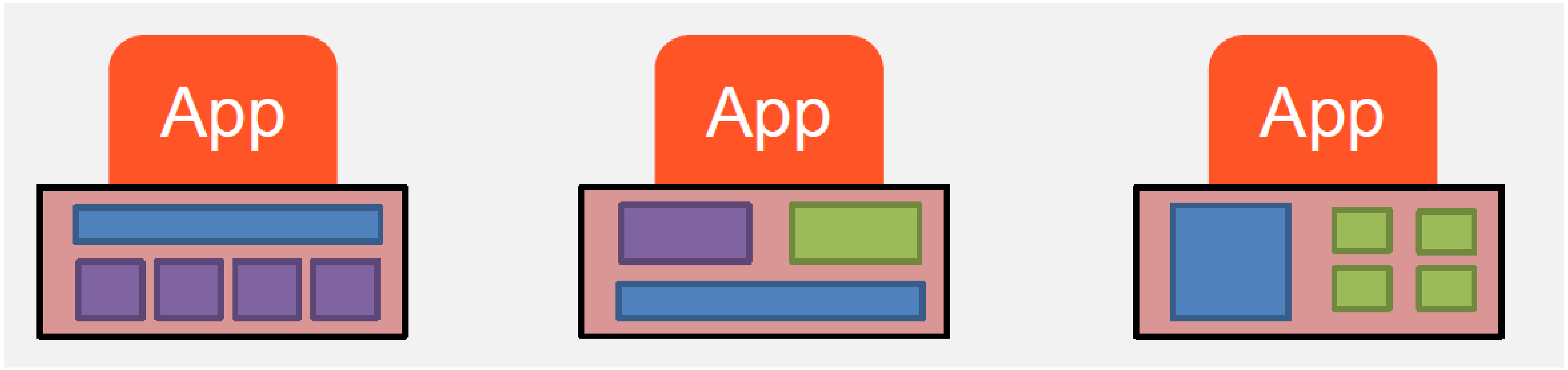
Keys to software cost control



Scalability

- The same application runs efficiently on new generations of cores
- The same application runs efficiently on more of the cores

Keys to software cost control



Portability

- The same application runs efficiently on different types of cores
- The same application runs efficiently on systems with different organizations and interfaces

Part3

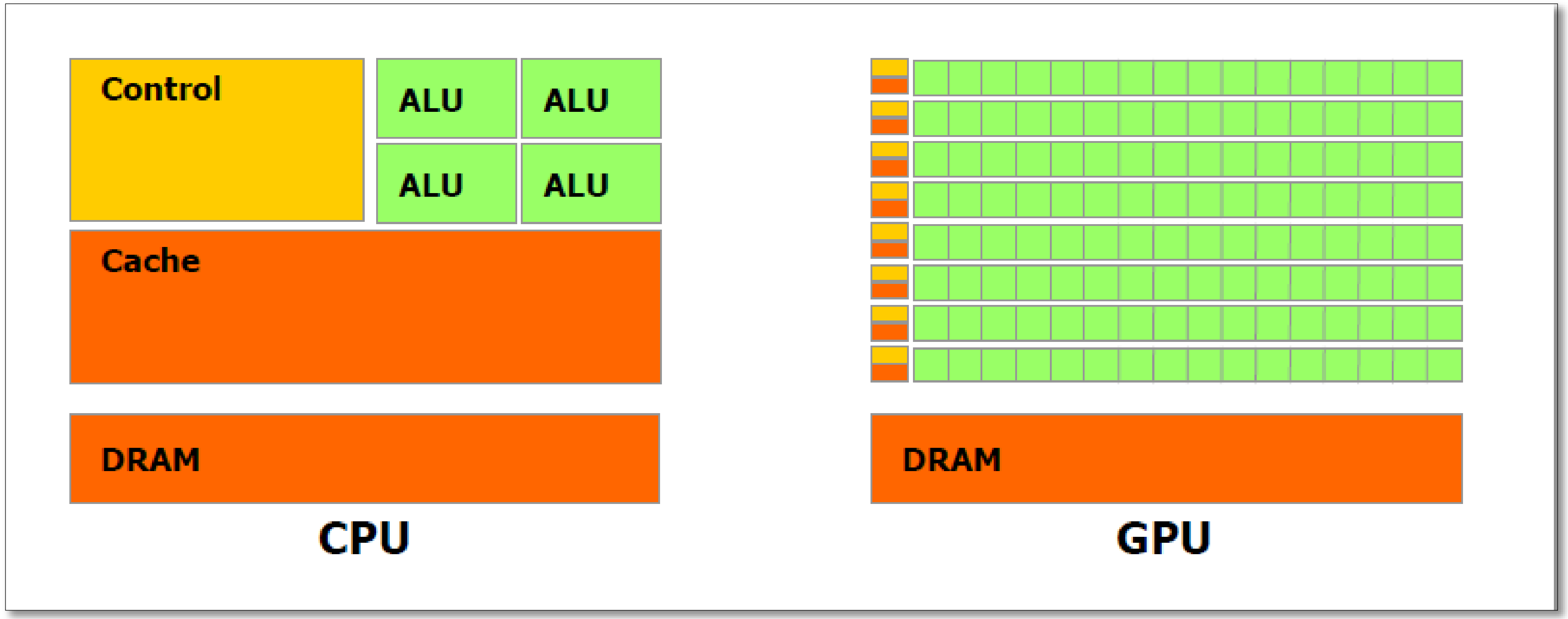
CUDA Programming Model

Outline

- **From CPU to GPU to GPGPU**
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- Block
- Grid
- Warp
- Steps of Writing a Program in CUDA

From CPU to GPU

- GPU is specialized for **compute-intensive** tasks
- Compared with CPU, more transistors devoted to **Data Processing** vs. **Caching**



From GPU to GPGPU

- GPU’s have evolved into **Many-core Multi-processors** with lots of processing power and an affordable cost
- What if you can program your GPU and convert into your own personal mini-supercomputer?
- NVIDIA CUDA enables you to do this without lots of PAIN ☺

From GPU to GPGPU

Programming your multithreaded GPU to perform general purpose tasks

is called

General Purpose GPU Computing or GPGPU.

Outline

- From CPU to GPU to GPGPU
- **Streaming Multiprocessor**
- CUDA Architecture
- Thread
- Kernel
- Block
- Grid
- Warp
- Steps of Writing a Program in CUDA

Streaming Multiprocessor

- NVIDIA CUDA Architecture is structured around an array of multithreaded multiprocessors

- Each Multiprocessor called a **“Streaming Multiprocessor”** or **“SM”**

- Number of Cores in SM varies depending on device **“Compute Capability”**

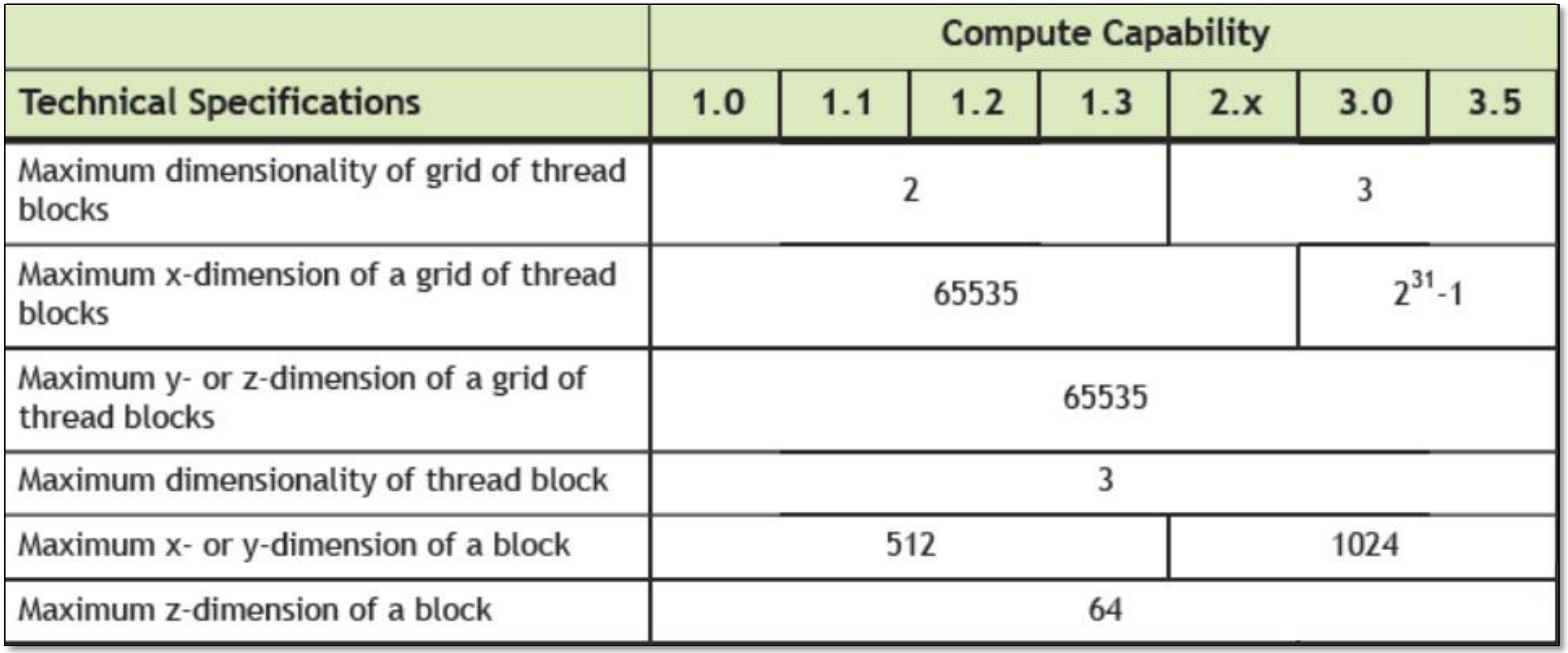
- **“Compute Capability”** : indicator of device architecture and specifications

Compute Capability

- Compute Capability = Major . Minor
- Major : Category of Core Architecture
- Minor : Features of Core Architecture

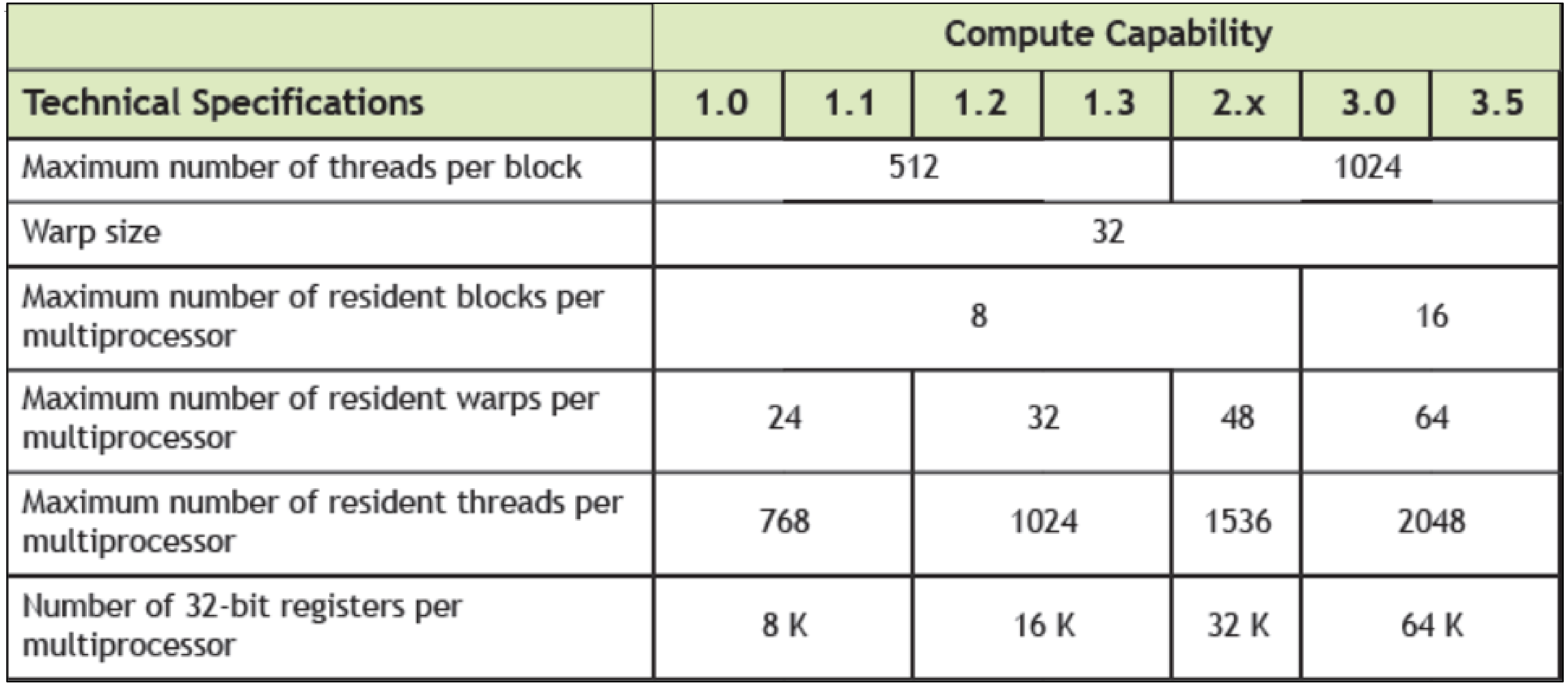
Compute Capability	Number of Cores in SM
1.X	8
2.0	32
2.1	48
3.X	192

Compute Capability



	Compute Capability						
Technical Specifications	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum dimensionality of grid of thread blocks	2	2	2	2	3	3	3
Maximum x-dimension of a grid of thread blocks	65535	65535	65535	65535	65535	$2^{31}-1$	$2^{31}-1$
Maximum y- or z-dimension of a grid of thread blocks	65535	65535	65535	65535	65535	65535	65535
Maximum dimensionality of thread block	3	3	3	3	3	3	3
Maximum x- or y-dimension of a block	512	512	512	512	1024	1024	1024
Maximum z-dimension of a block	64	64	64	64	64	64	64

Compute Capability



	Compute Capability						
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**
Maximum number of threads per block	512	512	512	512	1024	1024	1024
Warp size	32	32	32	32	32	32	32
Maximum number of resident blocks per multiprocessor	8	8	8	8	8	16	16
Maximum number of resident warps per multiprocessor	24	24	32	32	48	64	64
Maximum number of resident threads per multiprocessor	768	768	1024	1024	1536	2048	2048
Number of 32-bit registers per multiprocessor	8 K	8 K	16 K	16 K	32 K	64 K	64 K

Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- **CUDA Architecture**
- Thread
- Kernel
- Block
- Grid
- Warp
- Steps of Writing a Program in CUDA

CUDA C/C++:

- Set of extensions added for C/C++ programming language
- Enables Programmer to use GPU for General Purpose Computing (GPGPU)
- In other words High Performance Computing with GPU
- Architecture of NVIDIA CUDA devices is **“SIMT”**

SIMD and SIMT

SIMD : Single Instruction Multiple Data

- Execution scope is a process or a single thread

SIMT : Single Instruction Multiple Thread

- Extends SIMD to thread level execution
- Executing same instruction for all threads
- With Proper coordination programmer can realize **Data-Parallel** algorithms

Outline

Problem to Solve : Adding two vectors

- There are two vectors A, B each of size N
- Want to compute C = A+B

Sequential Version

```
• for ( i=0 ; i<N ; i=i+1 )
      c[i] = a[i] + b[i]
```

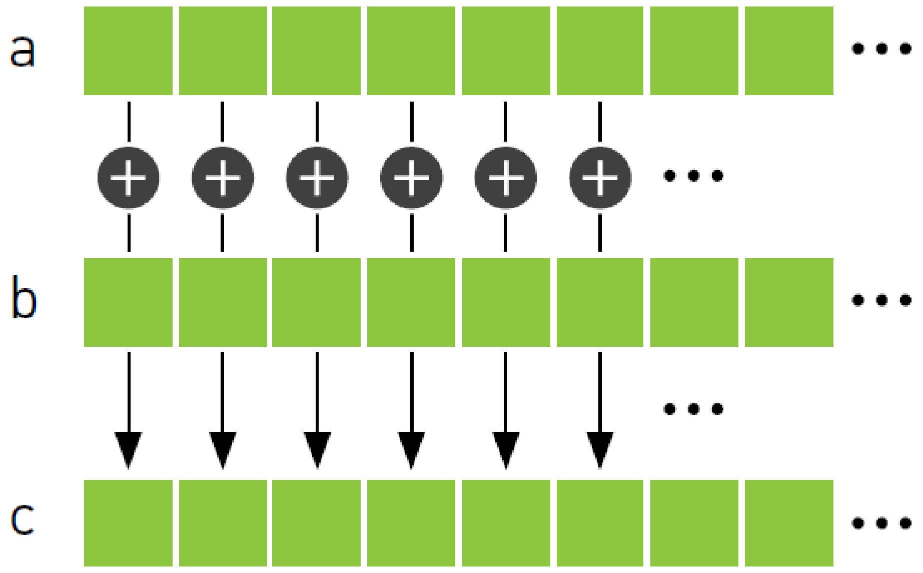
- This loop is executed **N** times !

Problem to Solve : Adding two vectors

- Now add **N** processors to this problem and then solve it
- One Processor for computing each element of C

Parallel Version

```
    c[i] = a [i] + b [i]
```



Now

In **1 cycle** of computation!

Thread

- In CUDA units of execution are called **“thread”**
- Programmer writes a code which consists of a set of threads
- Threads are scheduled and then executed

Thread

- For our example, we assign an index to each thread, **threadIdx**:

```
c[threadIdx] = a[threadIdx] + b[threadIdx]
```

Thread Execution

- In reality each thread is executed on one core of GPU

What About Order of Thread Execution ?

- That's out of our control, warp scheduler defines order of thread execution

What is warp scheduler exactly?

- We'll find out, but not Now !

Thread

- Thread Index = Position of thread in block
- Thread Index has **x, y, z** based on the structure of block
- 1D block only has **x** coordinate
- 2D block has **x, y** coordinates
- 3D block has **x, y, z** coordinates

Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- Block
- Grid
- Warp
- Steps of Writing a Program in CUDA

Kernel

Kernel

- Function that Each thread executes on GPU
- For N threads, kernel will be executed N times

Invoking Kernel

```
Kernel Name <<< gridDim, blockDim >>> (arguments);
```

Kernel

- In our example we need a kernel which has 3 arguments:

  vectors a and b as input, vector c as output

```
vectorAdd ( int * a, int * b, int * c)
{

threadIdx = find thread index ;

c[threadIdx] = a[threadIdx] + b[threadIdx] ;

}
```

Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- **Block**
- Grid
- Warp
- Steps of Writing a Program in CUDA

Block

- Programmer should partition threads in groups named **“Block"**
- Blocks can be 1, 2 or 3 dimensional
- Maximum number of threads per block, depending on device compute capability varies between 512,1024 and 1536

Block

- Maximum number of threads per block is considered for all dimensions:

For 1D Block

- blockDim.x = MAX

For 2D Block

- blockDim.x * blockDim.y = MAX

For 3D Block

- blockDim.x * blockDim.y * blockDim.z = MAX

BlockDim

BlockDim = Stores size of dimensions of a block

- blockDim.x
- blockDim.y
- blockDim.z

BlockIdx = Position of block in grid

- 1D grid only has **x** coordinate
- 2D grid has **x, y** coordinates
- 3D grid has **x, y, z** coordinates

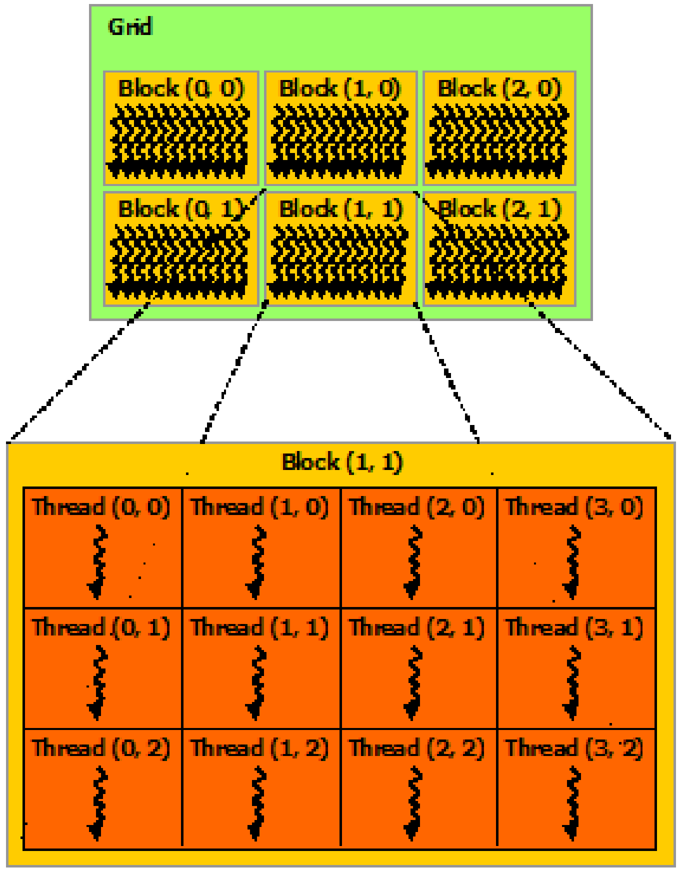
Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- Block
- **Grid**
- Warp
- Steps of Writing a Program in CUDA

Grid

- Programmer partitions blocks in groups named **"grid"**
- Maximum number of blocks per grid is 65535 in each dimension.
- Some devices support only 2 dimensional grids which means 65535*65535 blocks in a grid
- Newer devices support 3 dimensional grid which means 65535*65535*65535 blocks will be placed on a grid

Grid


Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)

Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- Block
- Grid
- **Warp**
- Steps of Writing a Program in CUDA

Warp

- Threads are created, managed and scheduled in groups of 32, each group is named a **warp**
- Threads in one warp begin execution at same point
- But each thread in warp has its own instruction address counter and register state
- It means threads in a warp are free to branch
- Threads in a warp have consecutive increasing numbers, starting from 0

Warp Scheduler

- GPU partitions threads of all blocks into warps
- Block -> Warp -> Warp Scheduler -> Execution Warp after Warp
- Warp/block = BlockSize / WarpSize

Warp Scheduler

Optimization Points:

- Define size of your block as a multiple of 32

❑ **Because** If block size is less than 32, empty places will be padded with empty threads and get executed, so fill your blocks

- Avoid using conditional branches that diverge execution of threads

❑ **Because** branching **decreases** parallelism

Host and Device

- **Device**

  GPU + Device Memory

- **Host**

  CPU + Host Memory

- **Device Memory**

  Memory which resides on GPU streaming multiprocessor

- **Host Memory**

  Memory which resides on host and is accessible by CPU

Outline

- From CPU to GPU to GPGPU
- Streaming Multiprocessor
- CUDA Architecture
- Thread
- Kernel
- Block
- Grid
- Warp
- **Steps of Writing a Program in CUDA**

Steps of Writing a Parallel Program in CUDA

1. Initialize environment, include headers, checking linker problems
2. Initialize input data
3. Allocate memory on device (GPU global memory)
4. Copy input data to device (GPU global memory)
5. Initialize and invoke kernel(s)
6. Copy results back from device memory to host memory (RAM)

cudaMalloc : Allocate Memory On Device

- Allocates memory on device, returns a pointer to the allocated memory
- We can only access the allocated memory via the returned pointer
- Before invoking the function we need to provide a **Pointer** on Host and **Size**

```
int * dev_vectorA ;

int sizeDev_VectorA = SIZE * sizeof (int);

cudaMalloc(dev_vectorA , sizeDev_VectorA)
```

cudaMemCpy : Copy Input Data To Device

- Copy data from our host memory to device allocated memory
- Programmer

  initializes data in a structure

  passes starting address of structure

  passes size of structure to device allocate memory

```
cudaMemCpy ( destination, source, size, kind_of_copy )
```

cudaMemCpyKind : Set Direction of Copy

syntax :

- cudaMemCpyDeviceToDevice
- cudaMemCpyHostToDevice
- cudaMemCpyDeviceToHost
- cudaMemCpyHostToHost

cudaEvent

- `cudaEvent_t event1;`
- `cudaEventCreate ( &event1 )`
- `cudaEventRecord ( &event1 )`

-
  ```
  cudaEventElapsedTime (    float * ms,
                            cudaEvent_t &start,
                            cudaEvent_t & stop    )
  ```

Schynronization

- `cudaEventSynchronize ( cudaEvent_t , cudaStream )`
- `cudaDeviceSynchronize ( cudaEvent_t )`

Example

Objectives:

- Developing a CUDA program for **Adding Two Vectors**, each of size 1MB
- Developing a **Sequential Version** which will be executed on CPU
- Measuring execution time of CUDA program using **cudaEvents**
- Measuring execution time of sequential using **QueryPerformanceCounter**
- Effect of **Critical Variables** and observe the results on performance

Step 0 : Installing CUDA driver and API

- Check Your Device Supports CUDA Programming
- Install **NVIDIA Driver** + **CUDA Toolkit** + **Visual Studio**
- Create An NVIDIA CUDA Project In Visual Studio
- Compile It To Ensure Proper Configuration Of Your Installation

Step1: Initialize Environment + include headers

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
using namespace std;
```

Step2 : Initialize Input Data-I

Where is your input ?

- ❑ Read input data from file
- ❑ Read input data from console
- ❑ Generate input inside program

Step2 : Initialize Input Data-I

- In our example we define a function "initVector" which initializes a vector with a specified value:

```
void initVector (int * vector, const int vectorSize, const int value)
{
	for ( int i=0; i < vectorSize ; i++)
		vector[i]=value;
}
```

Step2 : Initialize Input Data-II

- Define constant variables of your program in pragmas before the main function

```
#define SIZE 1024*1024 //size of vectors
#define BLOCK_SIZE 512
```

Step2 : Initialize Input Data-III

```
void initVector (int * vector, const int vectorSize, const int value)
{
    for ( int i=0; i < vectorSize ; i++)
        vector[i]=value;
}

int main()
{
    int * vecA = new int [SIZE];
    int * vecB = new int [SIZE];
    int * result = new int [SIZE];

    initVector(vecA, SIZE, 1);
    initVector(vecB, SIZE, -2);
    initVector(result, SIZE, 0);

return 0;
}
```

Step2 : Initialize Input Data-IV

```
//main
    int * d_vecA,
        * d_vecB,
        * d_result;

    int size_A,
        size_B,
        size_result;

    size_A = SIZE * sizeof(int);
    size_B = SIZE * sizeof(int);
    size_result = SIZE * sizeof(int);
//main
```

Step3 : Allocating Memory on Device

```
cudaMalloc ( (void **)&d_vecA , size_A);

cudaMalloc ( (void **)&d_vecB , size_B);

cudaMalloc ( (void **)&d_result , size_result);
```

Step4 : Copy Data to Allocated Device Memory

```
//main
    cudaMemcpy ( d_vecA , vecA, size_A , cudaMemcpyHostToDevice );

    cudaMemcpy (  d_vecB ,
                    vecB ,
                  size_B ,
                  cudaMemcpyKind::cudaMemcpyHostToDevice
               );

//main
```

Step5 : Initializing and Invoking Kernel

```
//main
    cudaEventRecord(start,0);

    blockDim.x=BLOCK_SIZE;

    gridDim.x= (SIZE+BLOCK_SIZE-1)/BLOCK_SIZE;

    vectorAdd <<<gridDim,blockDim>>>(d_result, d_vecA, d_vecB);

    cudaDeviceSynchronize();

    cudaEventRecord(stop,0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsed, start, stop);

//main
```

Step6 : Copy Result Data Back from Device Memory

```
//main

    cudaMemcpy ( result, d_result, size_result, cudaMemcpyDeviceToHost);

//main
```

Reference

All Pictures From :

- ❑ CUDA C Programming Guide, V 5.0, Authors
- ❑ CUDA By Example, Authors,

Part4

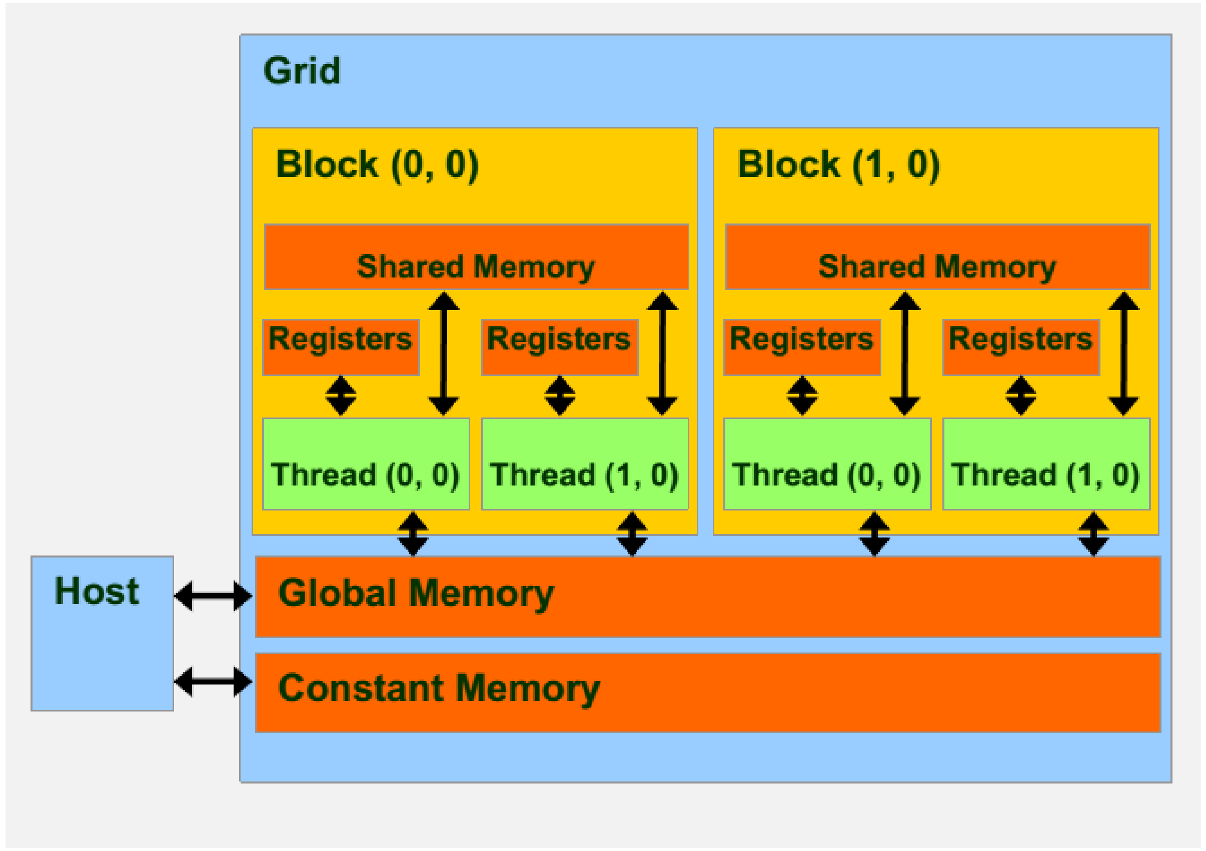
Dot Product

Objectives

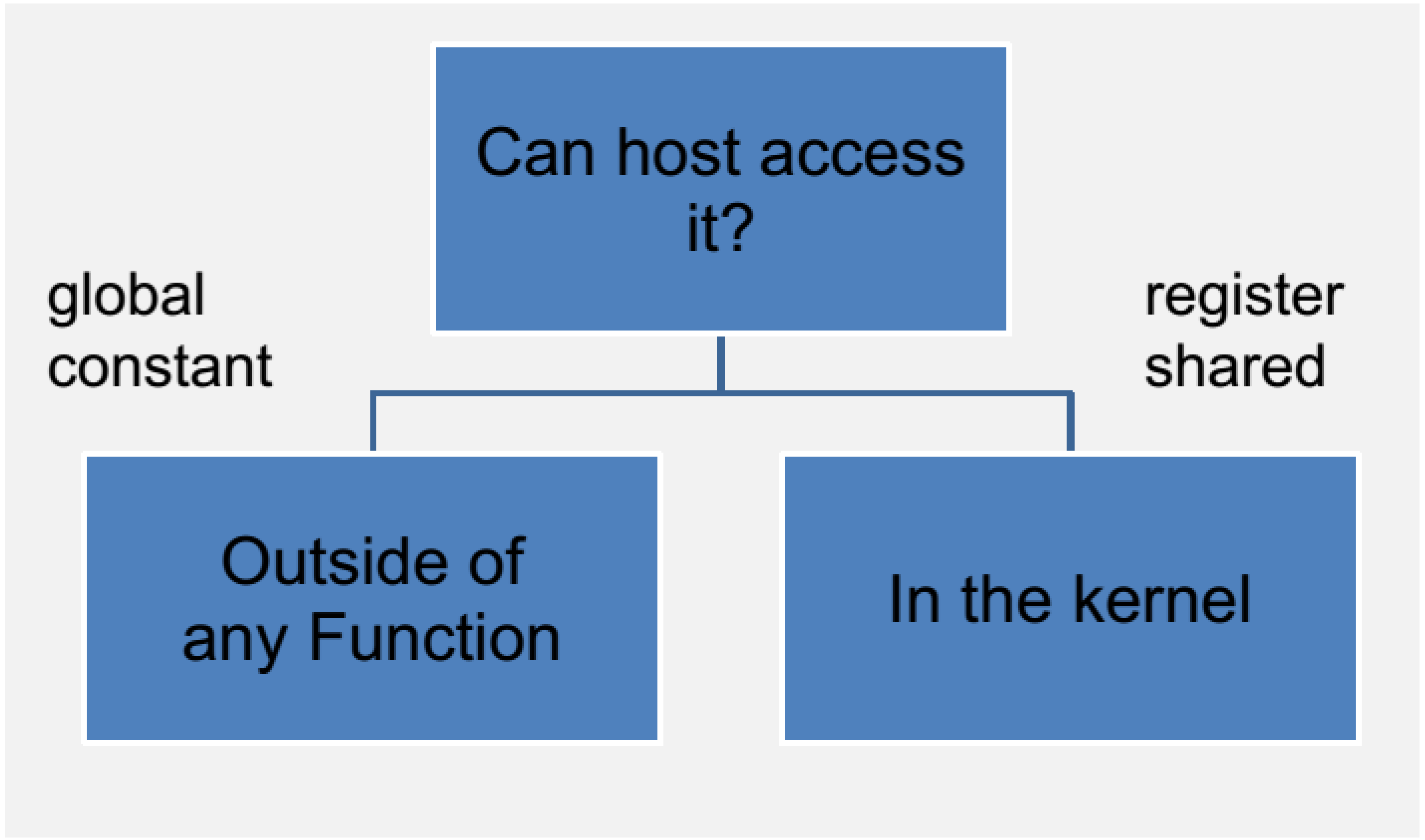
Learn How To Use The Important Levels Of The Memory :

- Registers, Shared, Global Memory
- Learn Dot Product Algorithm

Programmer view of CUDA memories

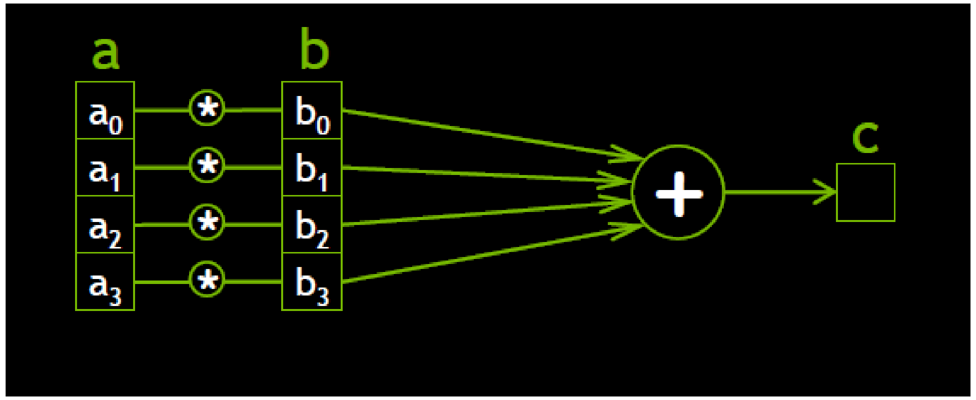


Where to declare variables?



Dot product

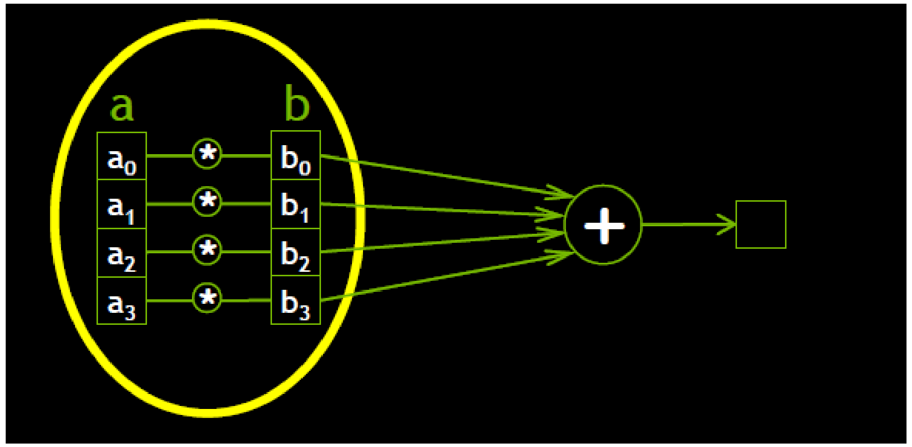
- Unlike vector addition, dot product is *reduction* from vectors to a scalar



- $c = a^{\rightarrow} . b^{\rightarrow}$
  - $= (a_0, a_1, a_2, a_3).(b_0, b_1, b_2, b_3)$
  - $= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$

Dot product

Parallel threads have no problem computing the pairwise products:

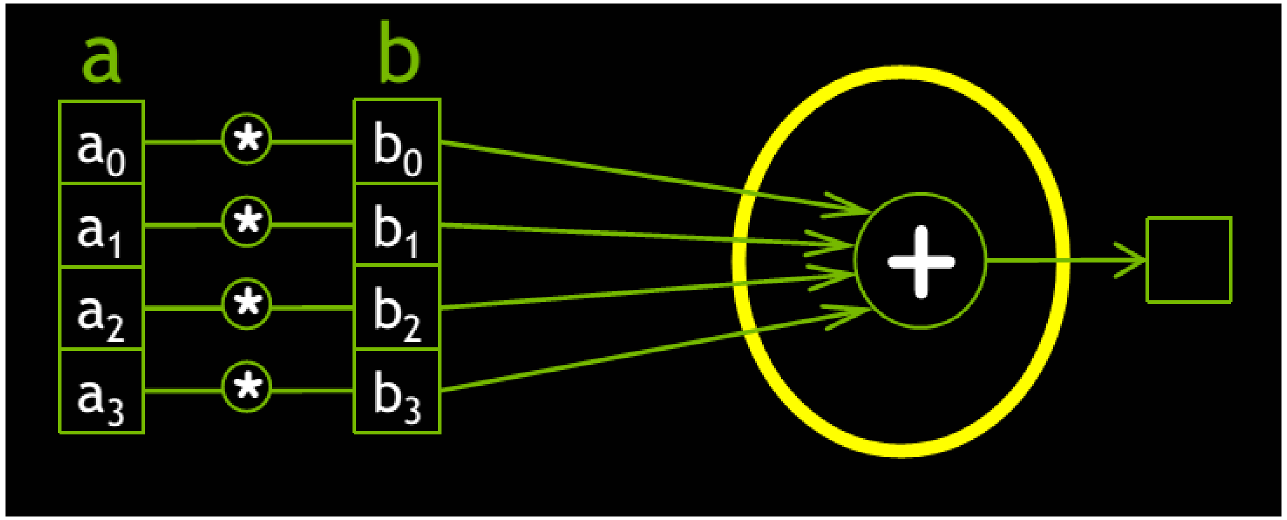


So we can start dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c ) {
       // Each thread computes a pairwiseproduct
       int temp = a[threadIdx.x] * b[threadIdx.x];
```

Dot Product

But we need to share data between threads to compute the final sum:



```
__global__ void dot( int *a, int *b, int *c ) {
    // Each thread computes a pairwiseproduct
    int temp = a[threadIdx.x] * b[threadIdx.x];

    // Can’t compute the final sum
    // Each thread’s copy of ‘temp’ is private
}
```

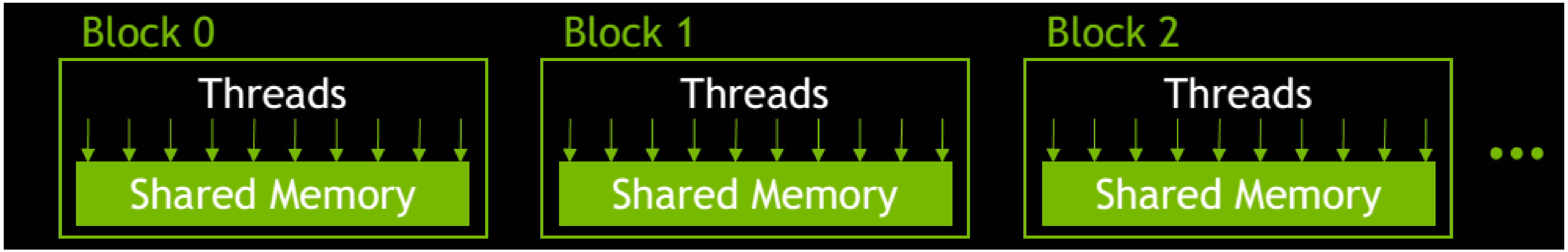
Sharing data between Threads

Terminology: A block of threads shares memory called … *shared memory*

Extremely fast, on chip memory (user-managed cache)

Declared with the `__shared__` CUDA keyword

Not visible to threads in other blocks running in parallel



Parallel Dot Product: **dot()**

We perform parallel multiplication, serial addition:

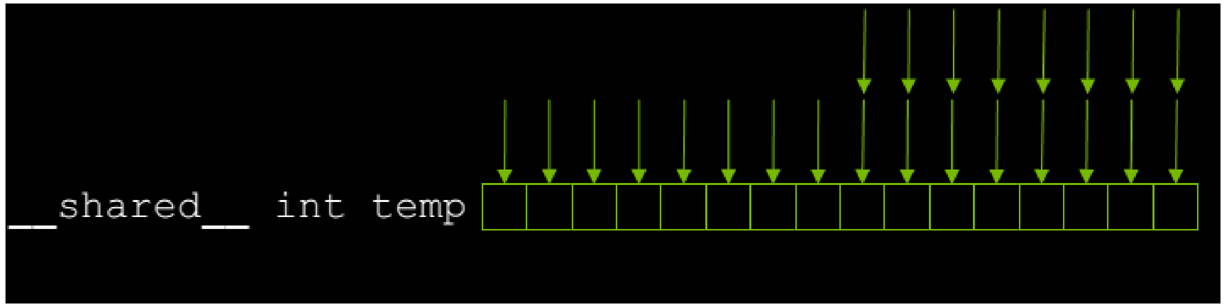
```
#definen 512
__global__ void dot( int *a, int *b, int *c ) {
    // Shared memory for results of multiplication
    __shared__ inttemp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwiseproducts
    if( ThreadIdx.x == 0) {
        Int sum = 0;
        for( int i= 0; i< N; i++ )
        sum += temp[i];
        *c = sum;
    }
}
```

Parallel Dot Product Recap

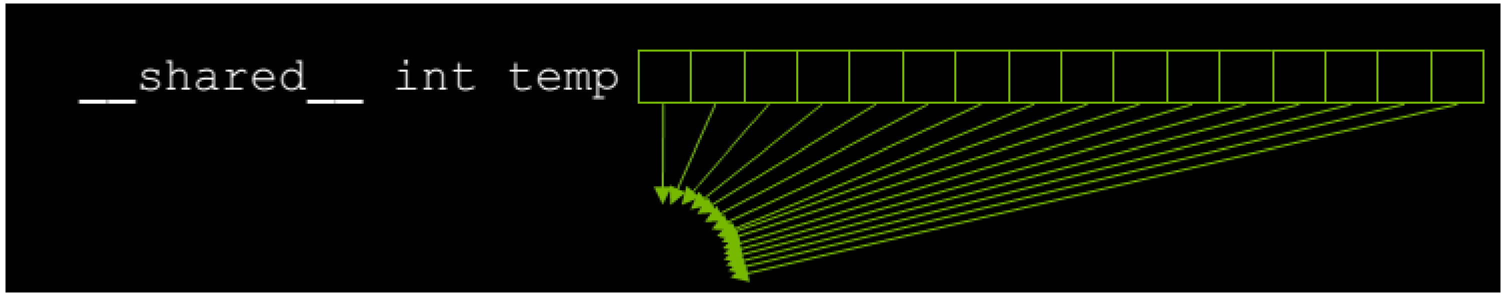
- We perform parallel, pairwise multiplications
- Shared memory stores each thread’s result
- We sum these pairwise products from a single thread
- Sounds good… but we’ve made a huge mistake

Faulty Dot Product Exposed!

- Step 1: In parallel, each thread writes a pairwise product



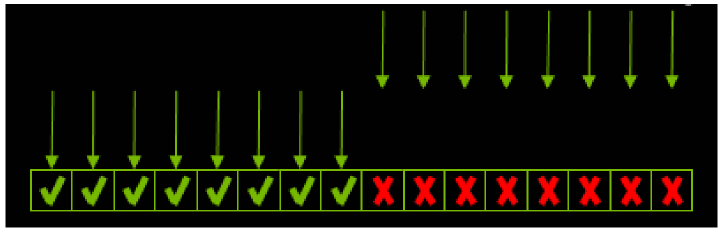
- Step 2: Thread 0 reads and sums the products



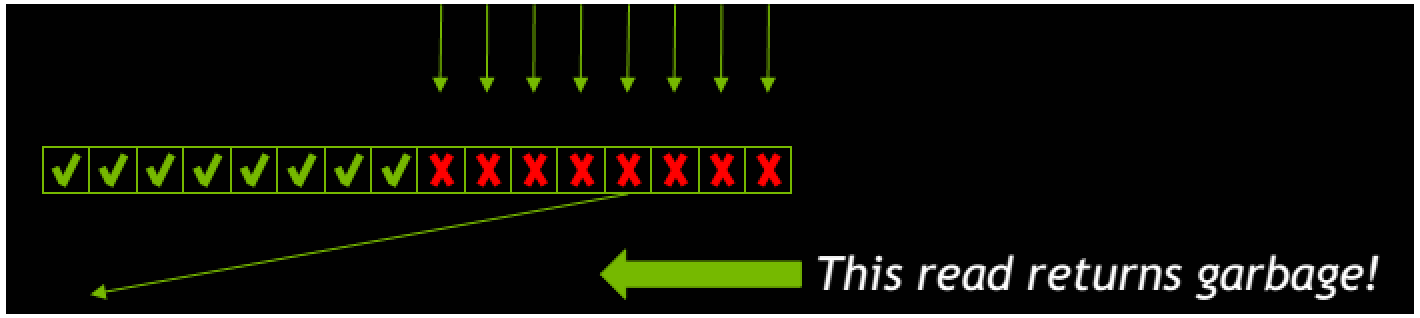
- But there’s an assumption hidden in Step 1…

Read-Before-Write Hazard

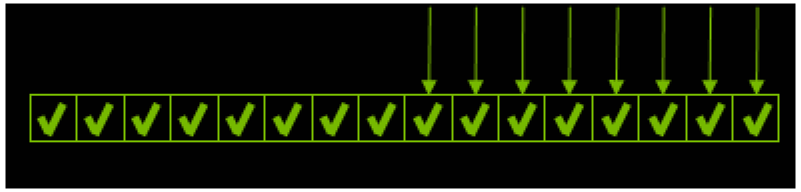
- Suppose thread 0 finishes its write in step 1



- Then thread 0 reads index 12 in step 2



- Before thread 12 writes to index 12 in step 1?



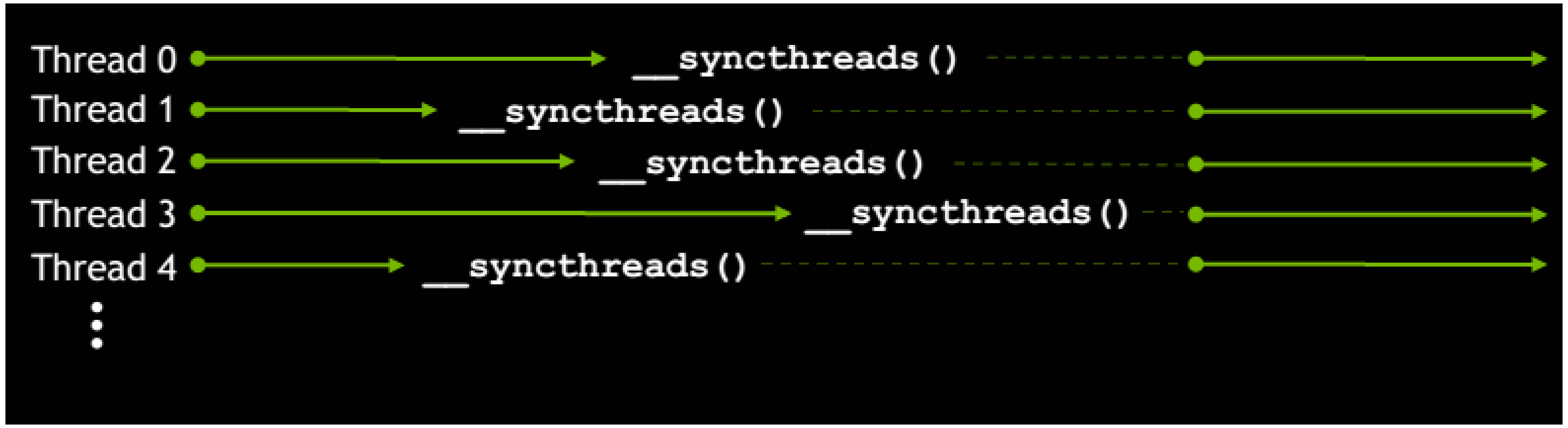
Synchronization

We need threads to wait between the sections of `dot()`:

```
__global__ void dot( int *a, int *b, int *c ) {

    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // * NEED THREADS TO SYNCHRONIZE HERE *
    // No thread can advance until all threads
    // have reached this point in the code
    // Thread 0 sums the pairwise products

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
        sum += temp[i];
        *c = sum;           }
}
```

__syncthreads()

- We can synchronize threads with the function `__syncthreads()`
- Threads in the block wait until *all* threads have hit the `__syncthreads()`



- Threads are *only* synchronized within a block

Parallel Dot Product: **dot()**

```
__global__ void dot( int *a, int *b, int *c ) {

    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
        sum += temp[i];
        *c = sum;
    }
}
```

Parallel Dot Product: `main()`

```
#define N 512
Int main( void ) {

    int *a, *b, *c;  // copies of a, b, c
    int *dev_a, *dev_b, *dev_c;  // device copies of a, b, c
    int size = N * sizeof( int );  // we need space for 512 integers

    // allocate device copies of a, b, c

    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: **`main()`**

```
// copy inputs to device

cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel with 1 block and N threads

dot<<<1, N >>>( dev_a, dev_b, dev_c);

// copy device result back to host copy of c

cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
return0;
}
```

Review

Launching kernels with parallel threads

- Launch `add()` with N threads: `add<<<1,N>>>();`
- Used `threadIdx.x` to access thread's index

Using both blocks and threads

- Used `(threadIdx.x + blockIdx.x*blockDim.x)` to index input/output
- `N/THREADS_PER_BLOCK` blocks and `THREADS_PER_BLOCK` threads gave us N threads total

Review

- Using `__shared` to declare memory as shared memory
  - Data shared among threads in a block
  - Not visible to threads in other parallel blocks

- Using `__syncthreads()` barrier
  - No thread executes instructions after `__synchthreads()` until all threads have reached the `__syncthreads()`
  - Needs to be used to prevent data hazards

Multi Dot Product

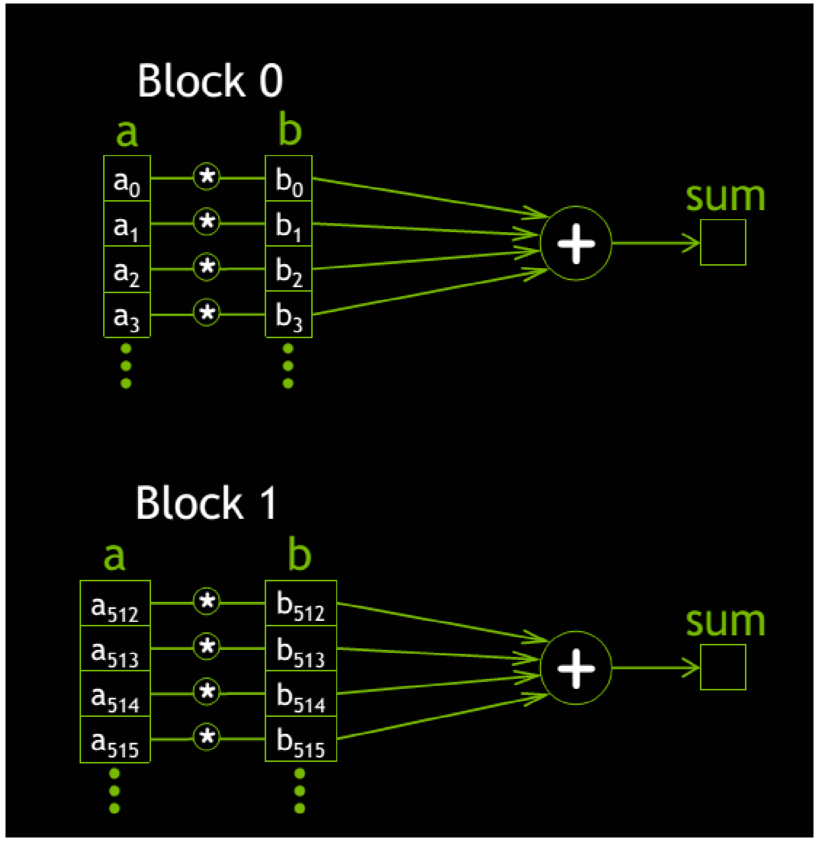
- Recall our dot product launch:

```
// launch dot() kernel with 1 block and N threads
dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

- Launching with one block will not utilize much of the GPU
- Let’s write a multiblockversion of dot product

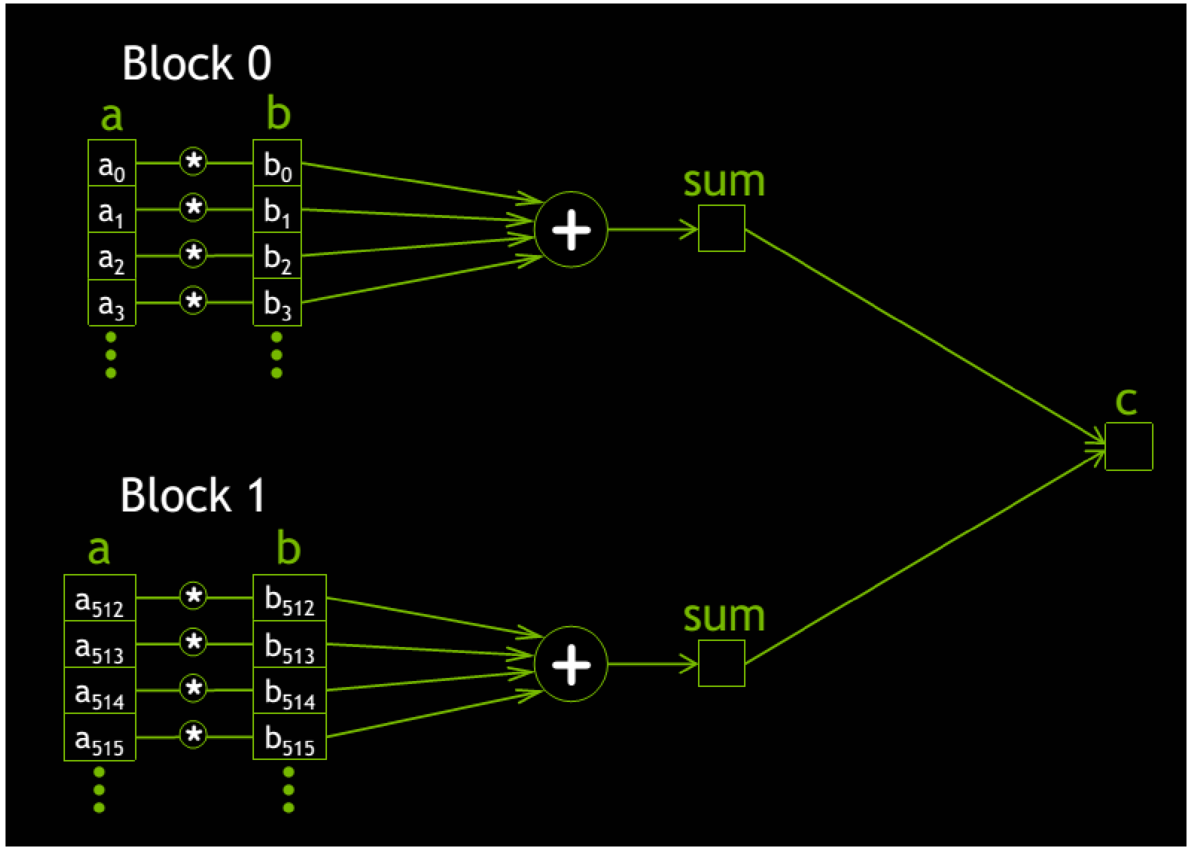
Multiblock Dot Product: Algorithm

Each block computes a sum of its pairwise products like before:



Multiblock Dot Product: Algorithm

And then contributes its sum to the final result:



Multiblock Dot Product: **dot()**

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
    __global__ voiddot( int*a, int*b, int*c ) {
    __shared__ inttemp[THREADS_PER_BLOCK];
    intindex = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        intsum = 0;
        for( inti= 0; i< THREADS_PER_BLOCK; i++ )
            sum += temp[i];

        Atomicadd(c, sum);
    }
}
```

But we have a race condition… !

Race Conditions

Terminology: A *race* condition occurs when program behavior depends upon relative timing of two (or more) event sequences

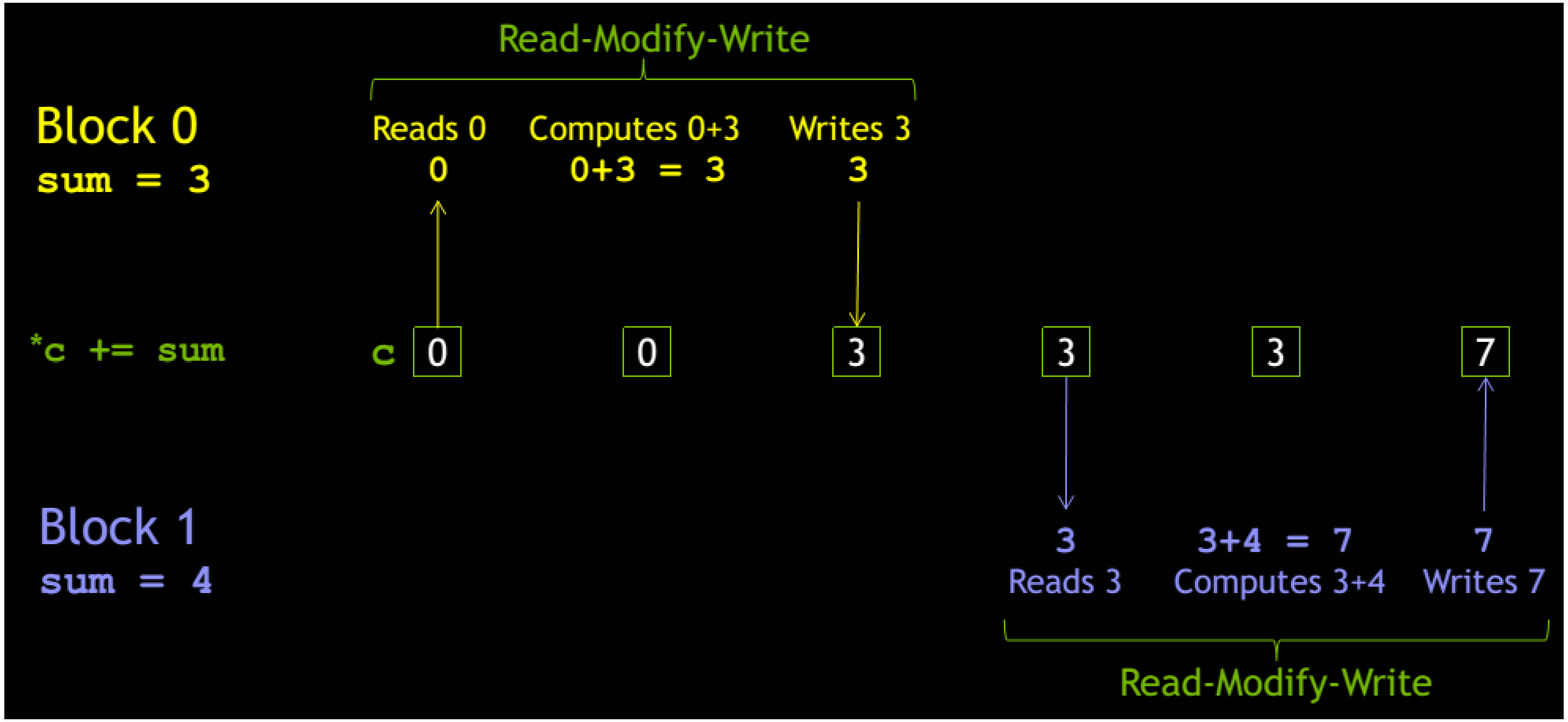
What actually takes place to execute the line in question: `*c += sum;`

- Read value at address c
- Add `sum` to value
- Write result to address c

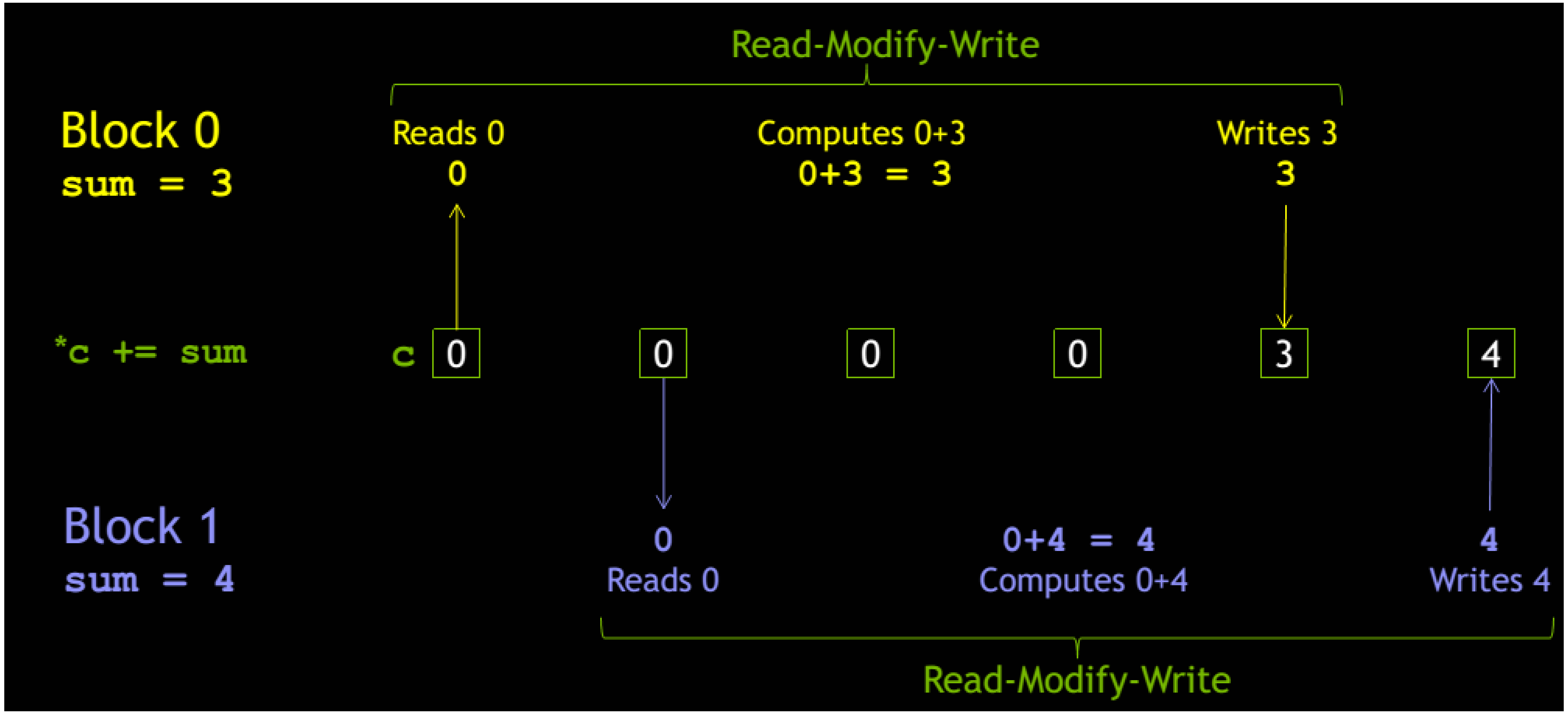
$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ Terminology: *Read-Modify-Write*

What if two threads are trying to do this at the same time?

Global Memory Contention

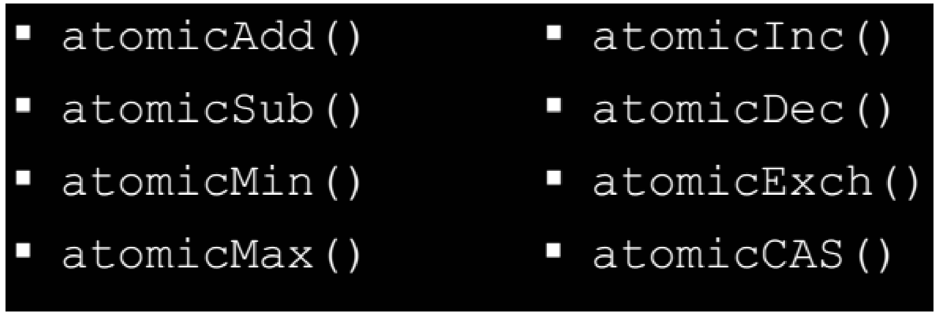


Global Memory Contention



Atomic Operation

- Terminology: Read-modify-write uninterruptible when atomic
- Many atomic operations on memory available with CUDA C



- `atomicAdd()`
- `atomicSub()`
- `atomicMin()`
- `atomicMax()`
- `atomicInc()`
- `atomicDec()`
- `atomicExch()`
- `atomicCAS()`

- Predictable result when simultaneous access to memory required
- We need to atomically add sumto cin our multiblockdot product

Multiblock Dot Product: dot()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
    __global__ voiddot( int*a, int*b, int*c ) {
    __shared__ inttemp[THREADS_PER_BLOCK];
    intindex = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        intsum = 0;
        for( inti= 0; i< THREADS_PER_BLOCK; i++ )
            sum += temp[i];

        Atomicadd(c, sum);
    }
}
```

Parallel Dot Product: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

Int main( void ) {
    int*a, *b, *c;  // host copies of a, b, c
    int*dev_a, *dev_b, *dev_c;  // device copies of a, b, c
    intsize = N * sizeof( int );  // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof(int) );
    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( sizeof(int) );
    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main()

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);

    // launch dot() kernel
    dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>( dev_a, dev_b, dev_c);

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, sizeof(int ) , cudaMemcpyDeviceToHost);

    free( a ); free( b ); free( c );
    cudaFree( dev_a);
    cudaFree( dev_b);
    cudaFree( dev_c);
    return0;
}
```

Review

- Race conditions
  - Behavior depends upon relative timing of multiple event sequences
  - Can occur when an implied read-modify-write is interruptible
- Atomic operations
  - CUDA provides read-modify-write operations guaranteed to be atomic
  - Atomics ensure correct results when multiple threads modify memory