

# Numerical Computations

# GPU Programming

Present By:

Armin Ahmadzadeh

---

Hamid Sarbazi-Azad &  
Samira Hossein Ghorban

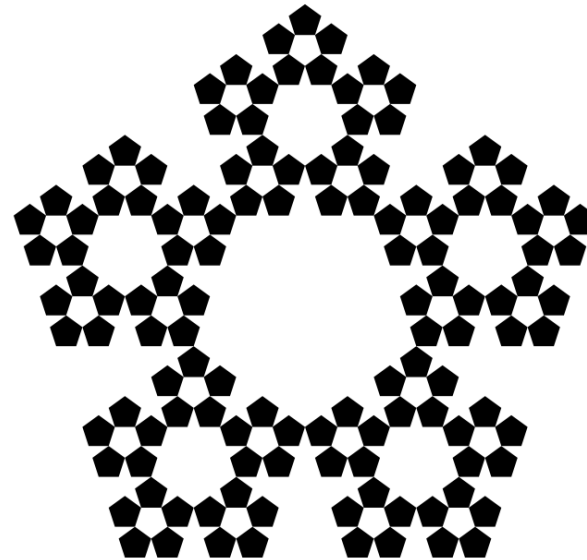
Department of Computer Engineering  
Sharif University of Technology (SUT)  
Tehran, Iran



# Outline

2

- GPU Short History
  - ▣ Massively Parallel Processing
- GPU Architecture
- GPU programming models
  - ▣ Memory Model
  - ▣ Processing Model
- CUDA Programming
- GPU and AI





# INTRODUCTION TO GPU & CUDA

# GPU Architecture



# Graphic Cards/ History

5

- 1980's – No GPUs. Just VGA controller
- 1990's – Add more function into VGA controller
- 1997 – 3D acceleration functions:
  - Hardware for triangle setup and rasterization
  - Texture mapping
  - Shading



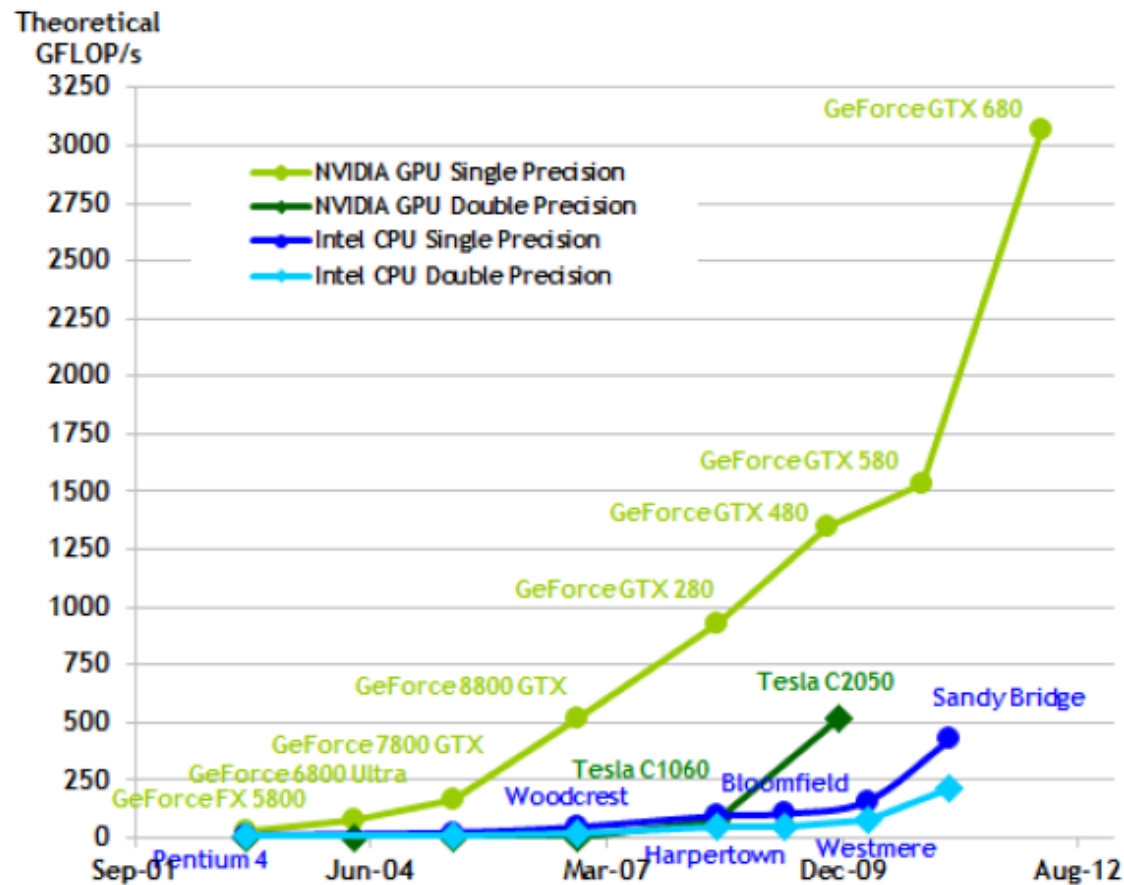
# GPU History

6

- 2000 – A single chip graphics processor
  - beginning of GPU term
- 2005 – Massively parallel programmable processors
- 2007 – CUDA (Compute Unified Device Architecture)
  - Nvidia initiated
  - C/C++ extention
- 2008 – OpenCL
  - Apple initiated
  - Based on C99 standard

# CPU vs. GPU

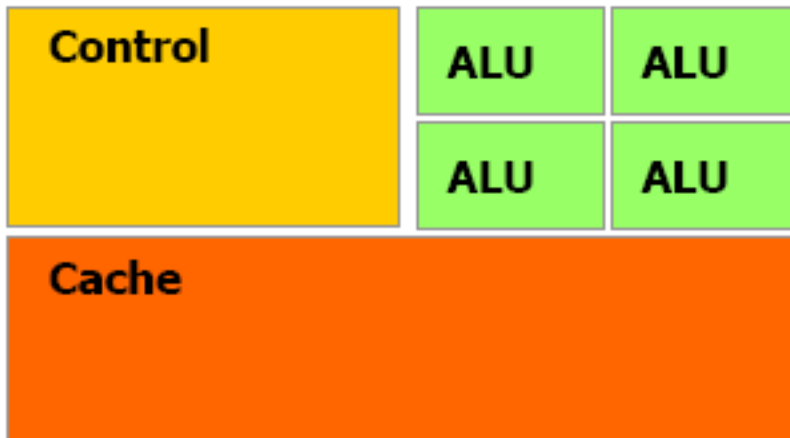
7



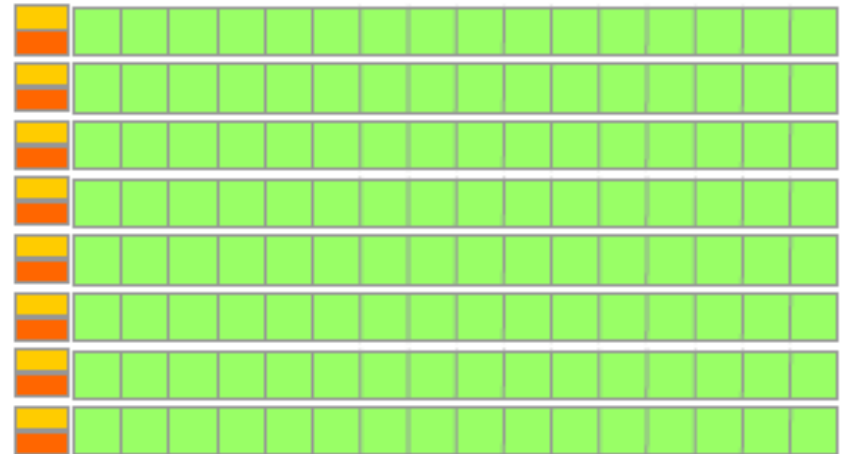
# CPU vs. GPU (continued)

8

## ➤ CPU



## ➤ GPU





# CPU vs. GPU (continued)

9

## ➤ CPU

- Latency oriented Cores
- Large caches
  - Lessen latency
- Sophisticated control
  - Branch prediction
  - Data forwarding
- Powerful ALUs
  - Reduce Latency

## ➤ GPU

- Throughput Oriented Cores
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding

# CPU vs. GPU (continued)

10

## ➤ CPU

- Sequential parts where latency matters
  - 10+X faster than GPU for sequential codes

## ➤ GPU

- Heavily pipelined
- Require massive number of threads to tolerate latencies

# Case Study

11

## ➤ CPU

### ➤ Intel Core i7:960

- 4-core, 3.2 GHz
- 2-way multi-threading
- 4-way SIMD
- L1 32KB, L2 256KB, L3 3MB
- 32 GB/sec

## ➤ GPU

### ➤ NVIDIA GTX 280

- 30 core, 1.3GHz
- 1024-way multi-threading
- 8-way SIMD
- 16KB software managed cache (shared memory)
- 141 GB/sec

# Number of Cores

12

- It is all about the core complexity:
  - The common goal: Improving pipeline efficiency
  - CPU goal: Single-thread performance
    - Exploiting ILP
    - Sophisticated branch predictor
    - Multiple issue logics
  - GPU goal: Throughput
    - Interleaving hundreds of threads

# Cache Size

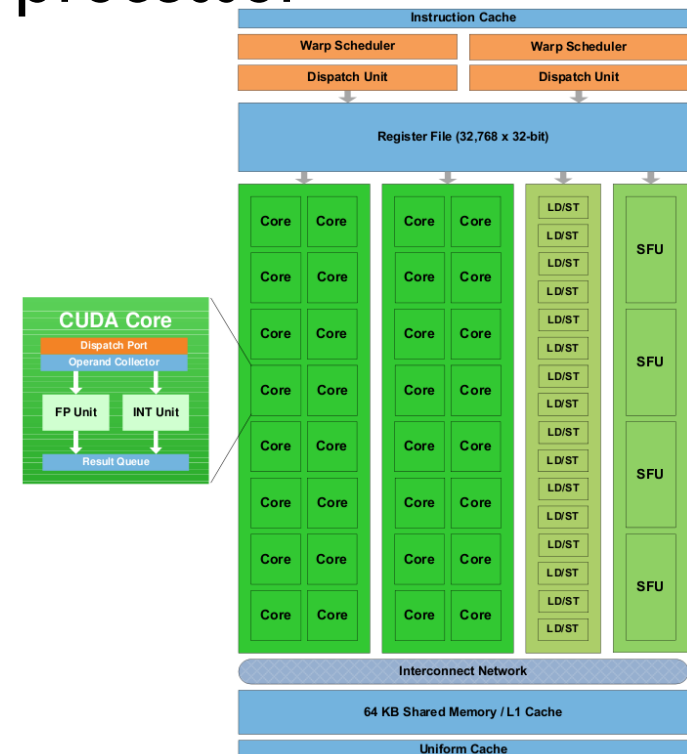
13

- CPU goal: reducing memory latency
  - Programmer-transparent data caching
    - Increasing the cache size to capture the working set
  - Prefetching (HW/SW)
- GPU goal: hiding memory latency
  - Interleave the execution of hundreds of threads to hide the latency of each other
- Notice:
  - CPU uses multi-threading for latency hiding
  - GPU uses software controlled caching (shared memory) for reducing memory latency

# Streaming Multiprocessor (SM)

14

- Pipeline deep-multithreaded SIMD processor
- Multiple SIMD groups
- Resources shared among threads
  - Shared memory
  - Register file
  - L1 Cache

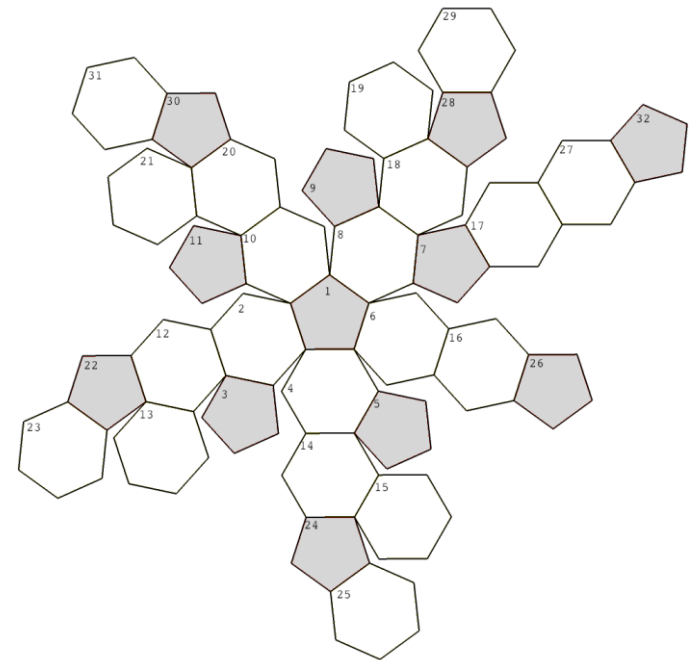


# Compute Capability (Nvidia)

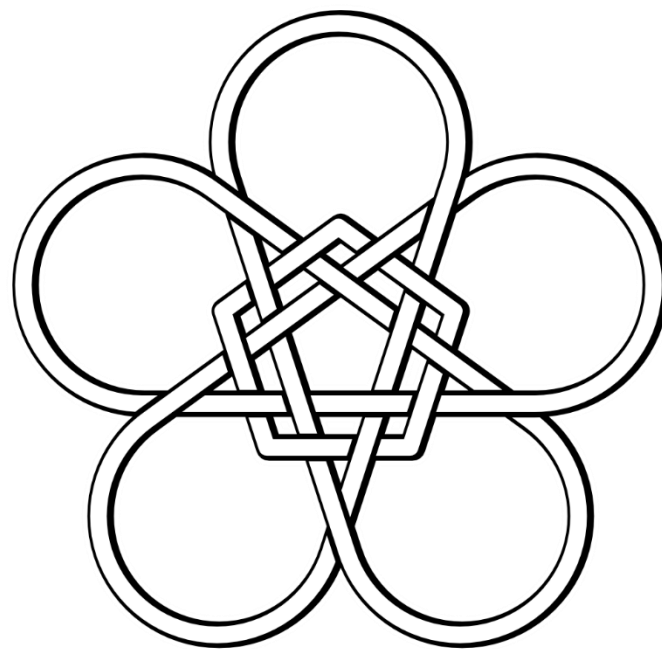
15

## ➤ Denotes the Capability of the GPU

- Major . Minor
- 1.0 and 1.1
- 1.2
  - Atomic operation on shared memory
- 1.3
  - Double-precision operations
- 2.x and 3.x
  - Informative `__syncthreads`
  - 64-bit atomic
  - 3D grid
  - Tensor Cores
  - Mix Lib (Curand, CuFFT, .. )



# GPU Programming Model





# What Is CUDA

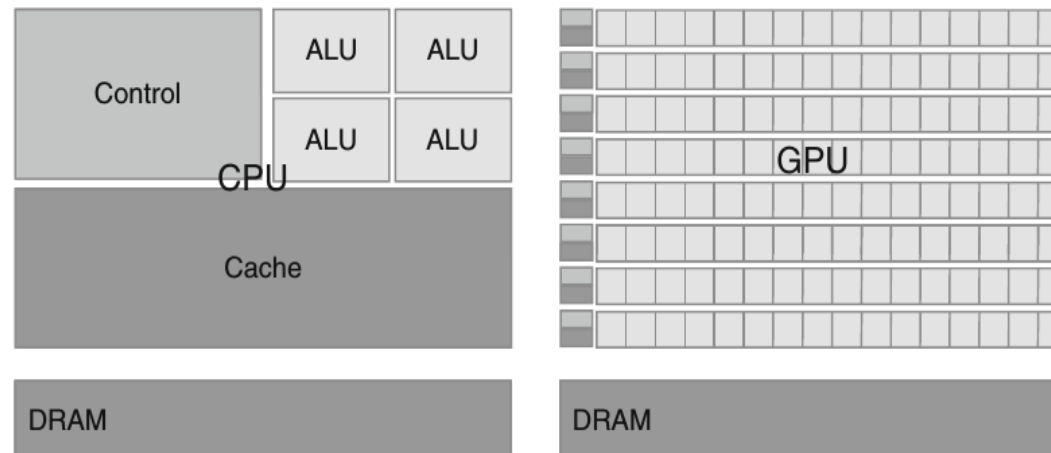
17

- Compute Unified Device Architecture
- Based on industry-standard C
- A handful of language extensions to allow heterogeneous programs
- Straightforward APIs to manage devices, memory, etc

# Why CUDA?

18

- Massive parallel computing power
- Maximize throughput of all threads
- Using hundreds of ALU inside a GPU

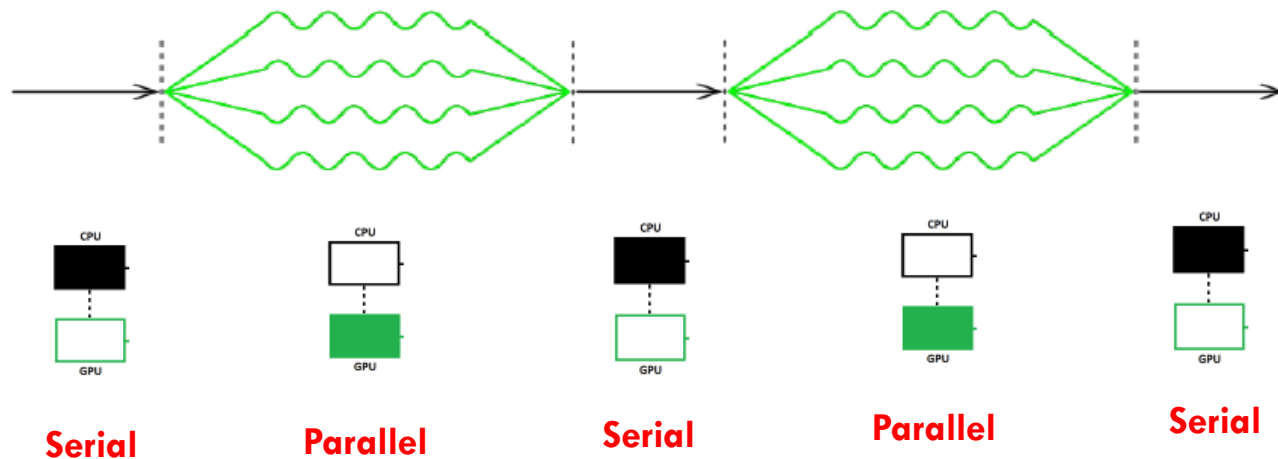


# CUDA Programming Model

19

## □ Heterogeneous Programming

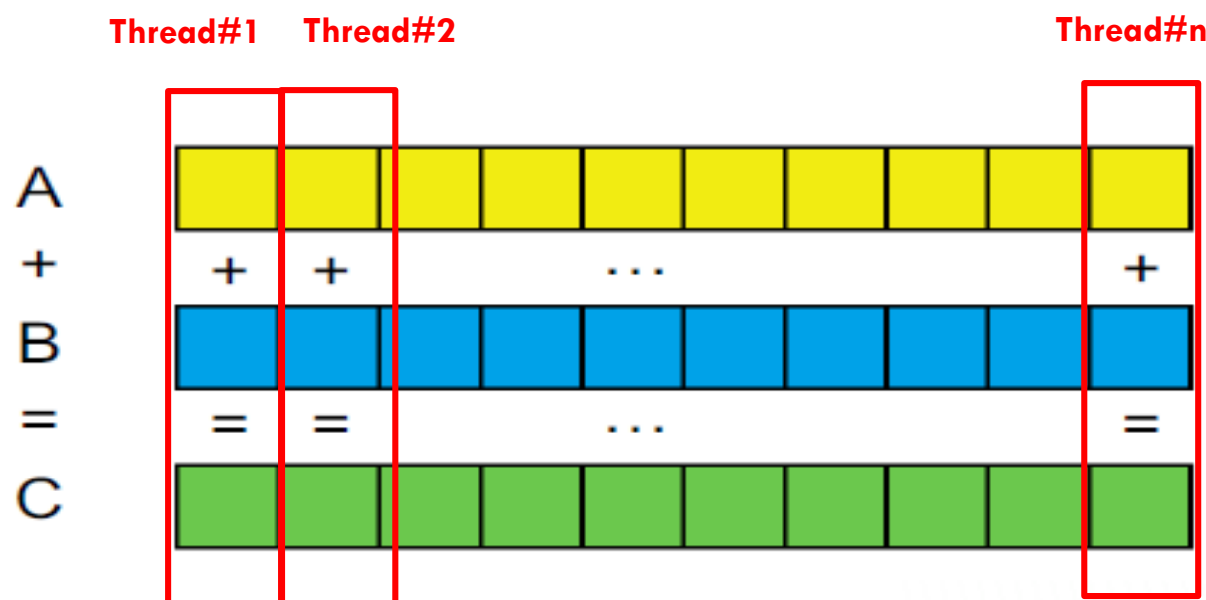
- program separated into serial regions (**run on CPU**) & parallel regions (**run on GPU**)



# CUDA Programming Model

20

- Parallel regions consist of many calculations that can be executed independently
  - ▣ Data Parallelism (e.g. vector addition)

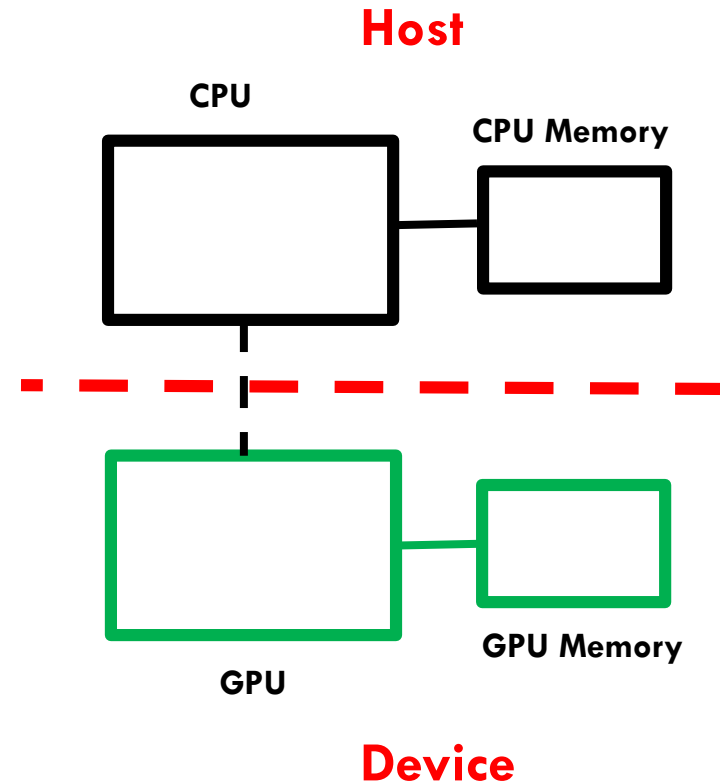


# A Basic CUDA Program Outline

21

```
Int main() {  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory }  

```

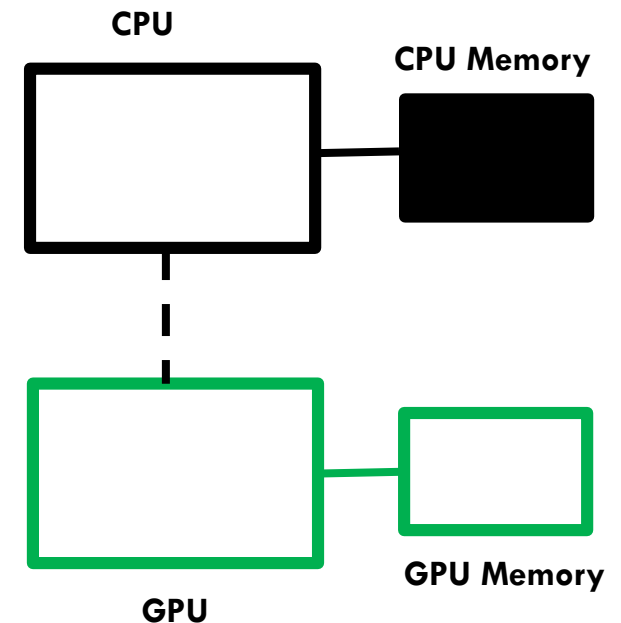


# A Basic CUDA Program Outline

22

```
Int main() {  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory }  

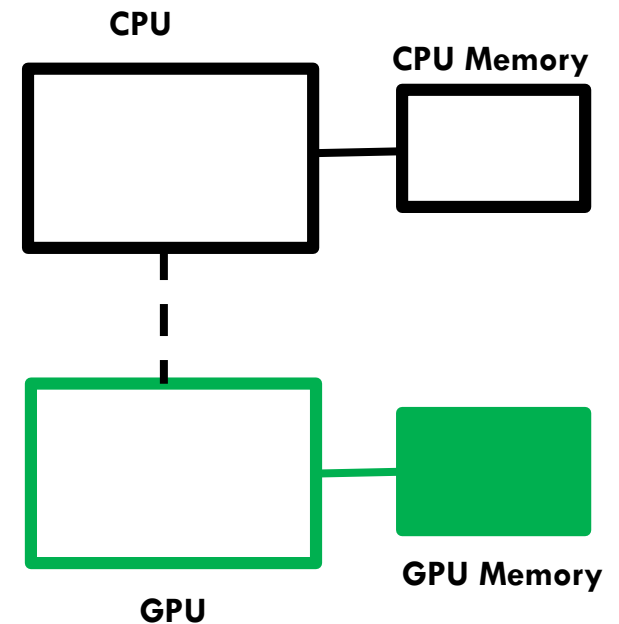
```



# A Basic CUDA Program Outline

23

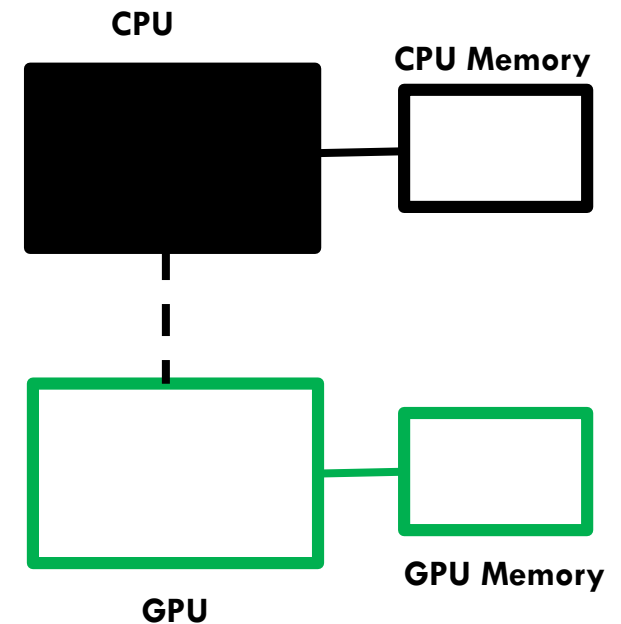
```
Int main() {  
  
  // Allocate memory for array on host  
  
  // Allocate memory for array on device  
  
  // Fill array on host  
  
  // Copy data from host array to device array  
  
  // Do something on device (e.g. vector addition)  
  
  // Copy data from device array to host array  
  
  // Check data for correctness  
  
  // Free Host Memory  
  
  // Free Device Memory }  
}
```



# A Basic CUDA Program Outline

24

```
Int main() {  
  
  // Allocate memory for array on host  
  
  // Allocate memory for array on device  
  
  // Fill array on host  
  
  // Copy data from host array to device array  
  
  // Do something on device (e.g. vector addition)  
  
  // Copy data from device array to host array  
  
  // Check data for correctness  
  
  // Free Host Memory  
  
  // Free Device Memory }  
}
```



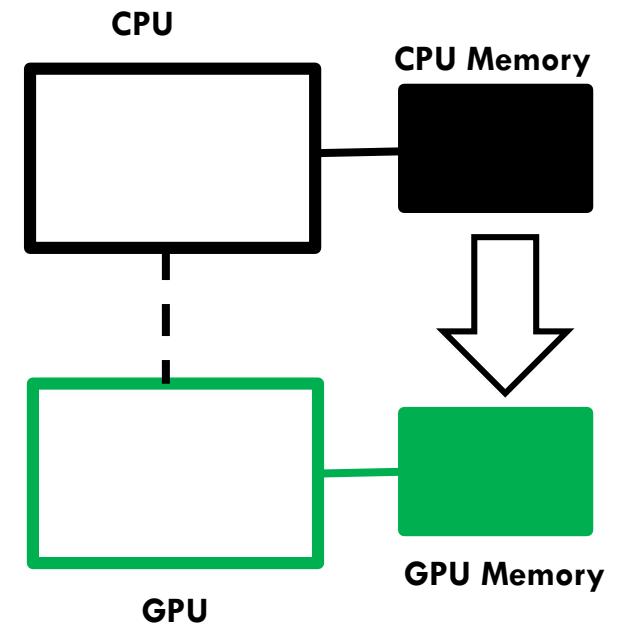


# A Basic CUDA Program Outline

25

```
Int main() {  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory }  

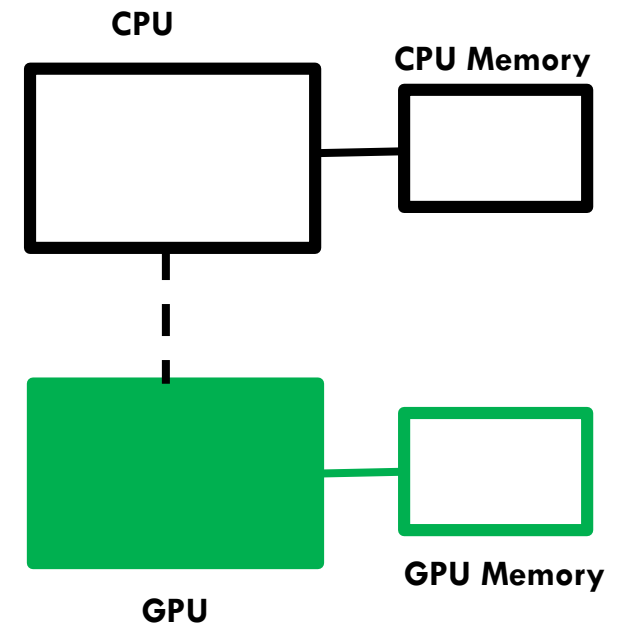
```



# A Basic CUDA Program Outline

26

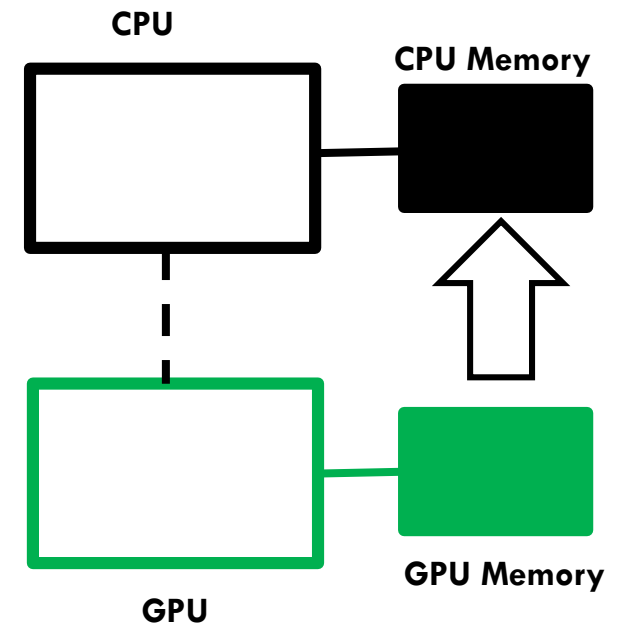
```
Int main() {  
  
// Allocate memory for array on host  
  
// Allocate memory for array on device  
  
// Fill array on host  
  
// Copy data from host array to device array  
  
// Do something on device (e.g. vector addition)  
  
// Copy data from device array to host array  
  
// Check data for correctness  
  
// Free Host Memory  
  
// Free Device Memory }  
}
```



# A Basic CUDA Program Outline

27

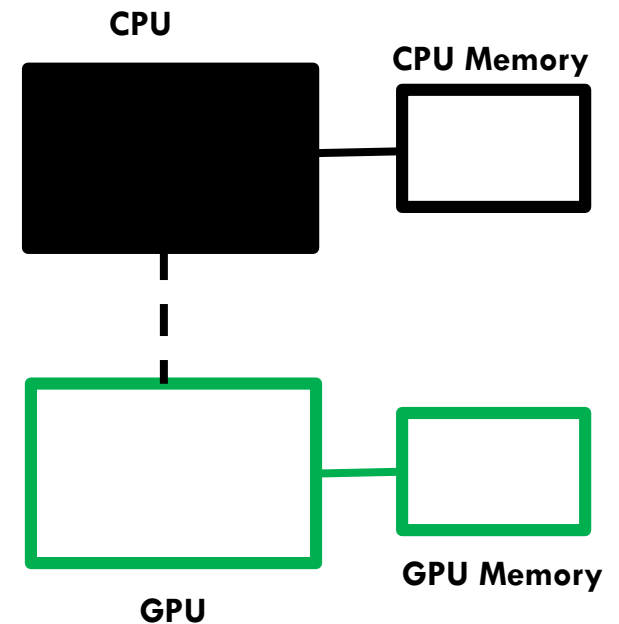
```
Int main() {  
  
// Allocate memory for array on host  
  
// Allocate memory for array on device  
  
// Fill array on host  
  
// Copy data from host array to device array  
  
// Do something on device (e.g. vector addition)  
  
// Copy data from device array to host array  
  
// Check data for correctness  
  
// Free Host Memory  
  
// Free Device Memory }  
}
```



# A Basic CUDA Program Outline

28

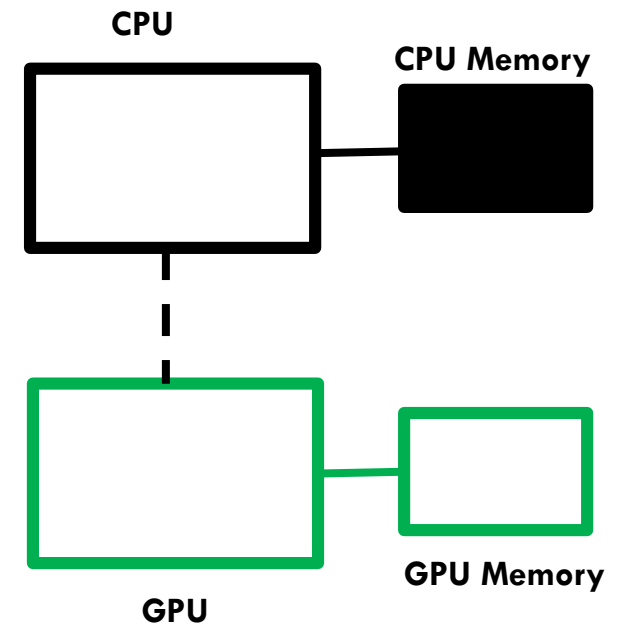
```
Int main() {  
  
// Allocate memory for array on host  
  
// Allocate memory for array on device  
  
// Fill array on host  
  
// Copy data from host array to device array  
  
// Do something on device (e.g. vector addition)  
  
// Copy data from device array to host array  
  
// Check data for correctness  
  
// Free Host Memory  
  
// Free Device Memory }  
}
```



# A Basic CUDA Program Outline

29

```
Int main() {  
  
  // Allocate memory for array on host  
  
  // Allocate memory for array on device  
  
  // Fill array on host  
  
  // Copy data from host array to device array  
  
  // Do something on device (e.g. vector addition)  
  
  // Copy data from device array to host array  
  
  // Check data for correctness  
  
  // Free Host Memory  
  
  // Free Device Memory }  
}
```

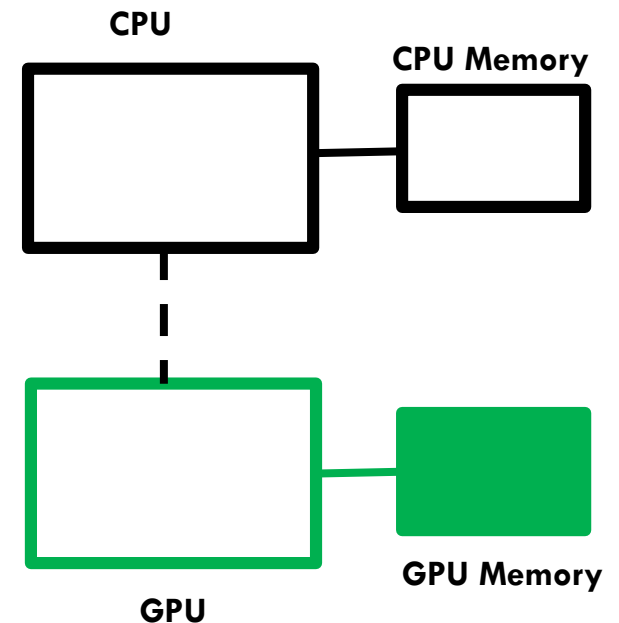


# A Basic CUDA Program Outline

30

```
Int main() {  
  
    // Allocate memory for array on host  
  
    // Allocate memory for array on device  
  
    // Fill array on host  
  
    // Copy data from host array to device array  
  
    // Do something on device (e.g. vector addition)  
  
    // Copy data from device array to host array  
  
    // Check data for correctness  
  
    // Free Host Memory  
  
    // Free Device Memory }  

```



# A Basic CUDA Program Outline

31

```
Int main() {
```

```
    // Allocate memory for array on host
```

```
    size_t bytes = N*sizeof(int);
```

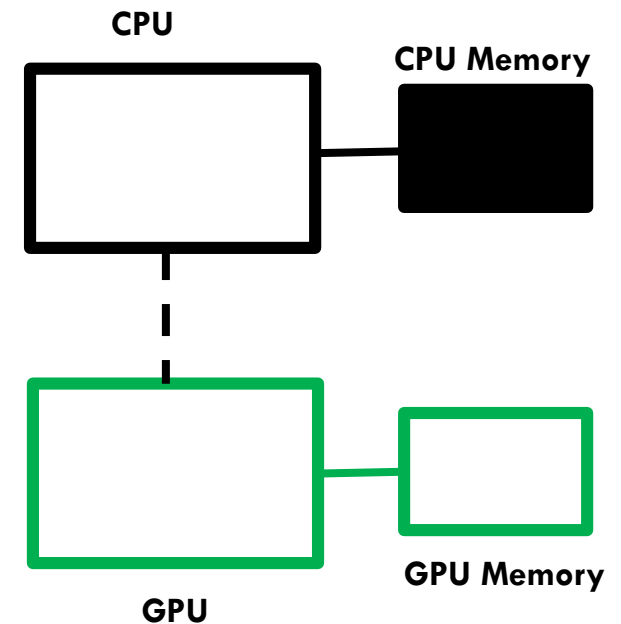
```
    int *A = (int*)malloc(bytes);
```

```
    int *B = (int*)malloc(bytes);
```

```
    int *C = (int*)malloc(bytes);
```

```
    ...
```

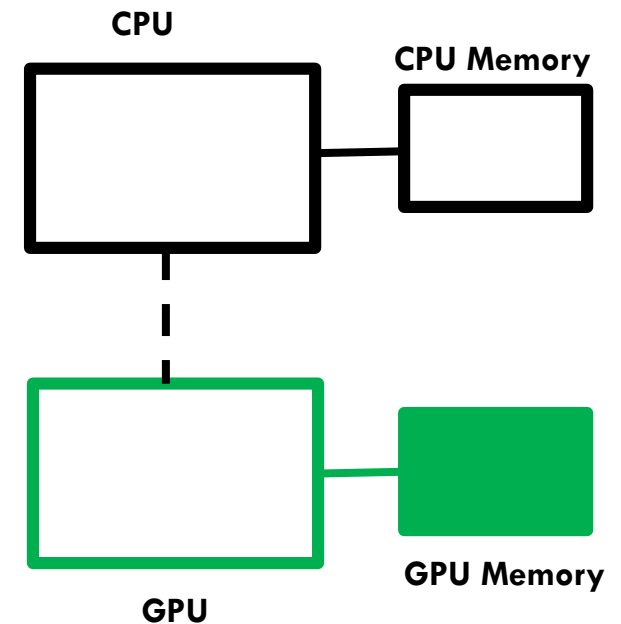
```
}
```



# A Basic CUDA Program Outline

32

```
Int main() {  
  
    ...  
  
    // Allocate memory for array on device  
  
    int *d_A, *d_B, *d_C;  
  
    cudaMalloc(&d_A, bytes);  
  
    cudaMalloc(&d_B, bytes);  
  
    cudaMalloc(&d_C, bytes);  
  
    ...  
  
}
```

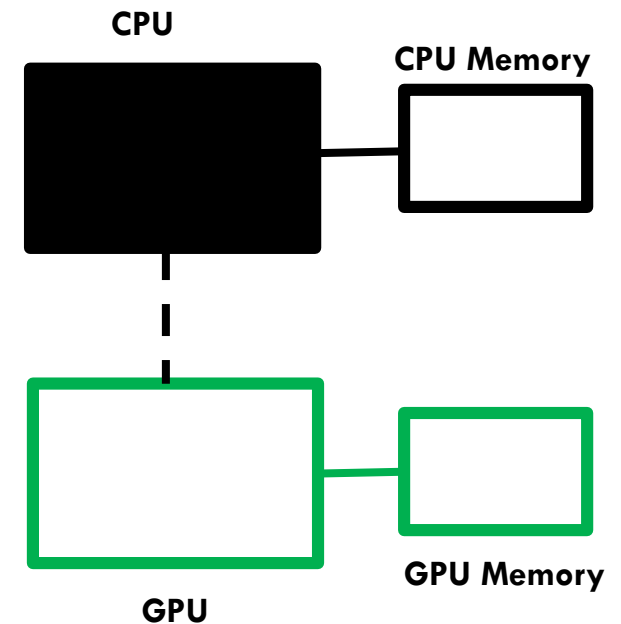




# A Basic CUDA Program Outline

33

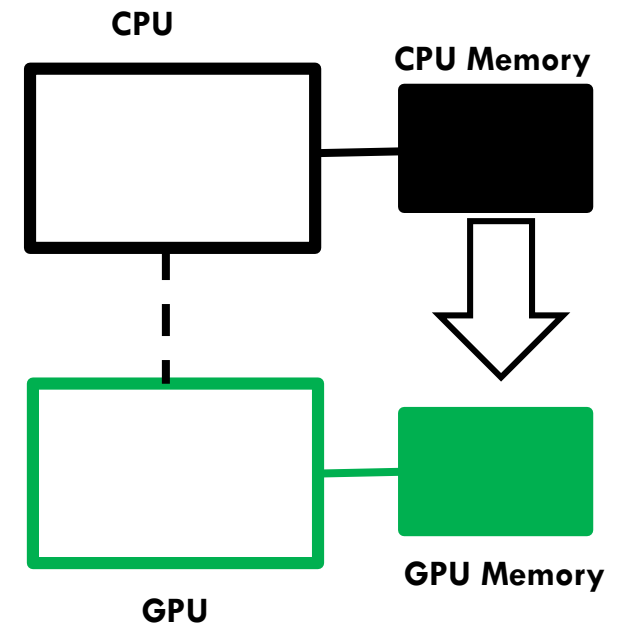
```
Int main() {  
  
    ...  
  
    // Fill array on host  
  
    for(int i=0; i<N; i++)  
    {  
        A[i] = 1;  
        B[i] = 2;  
        C[i] = 0;  
    }  
    ...  
}
```



# A Basic CUDA Program Outline

34

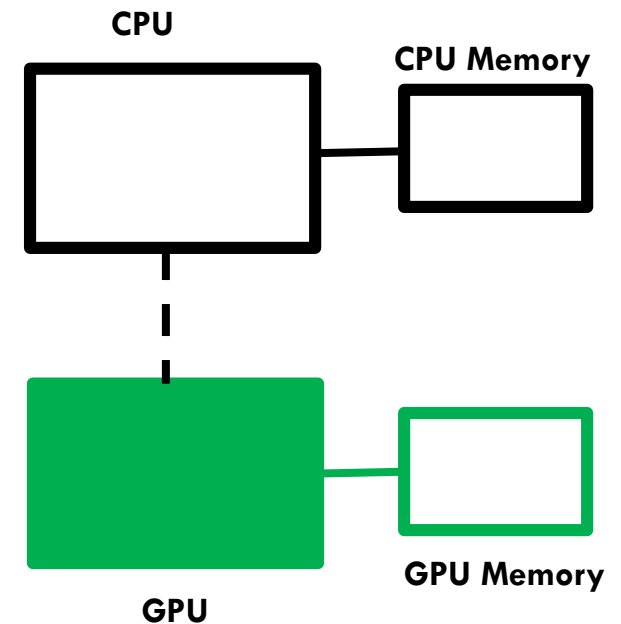
```
Int main() {  
    ...  
  
    // Copy data from host array to device array  
  
    cudaMemcpy(d_A, A, bytes,  
               cudaMemcpyHostToDevice);  
  
    cudaMemcpy(d_B, B, bytes,  
               cudaMemcpyHostToDevice);  
  
    ...  
}
```



# A Basic CUDA Program Outline

35

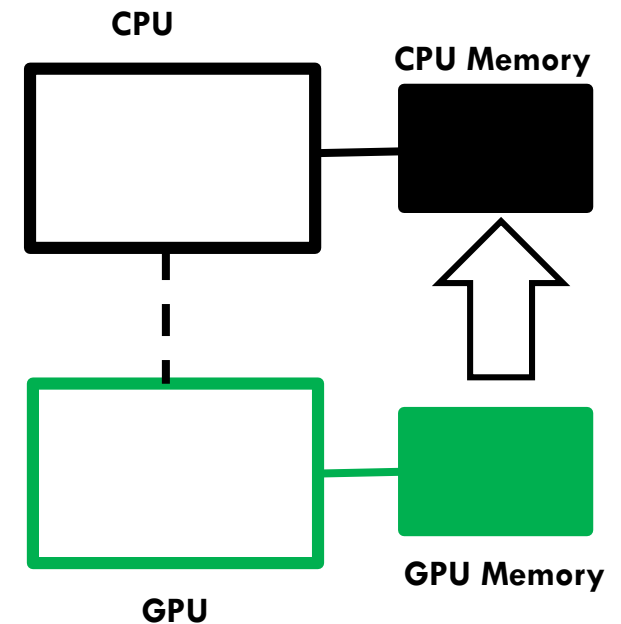
```
Int main() {  
  
    ...  
  
    // Do something on device (e.g. vector addition)  
  
    // We'll come back to this soon  
  
    ...  
  
}
```



# A Basic CUDA Program Outline

36

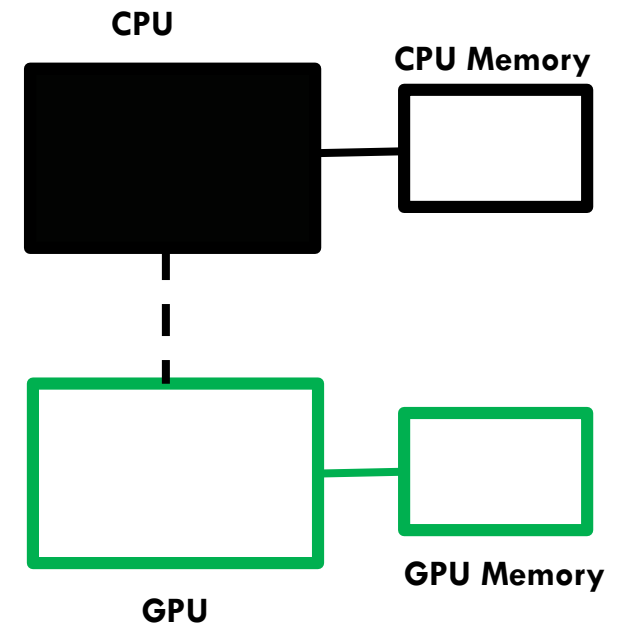
```
Int main() {  
    ...  
  
    // Copy data from device array to host array  
    cudaMemcpy(C, d_C, bytes,  
               cudaMemcpyDeviceToHost);  
    ...  
}
```



# A Basic CUDA Program Outline

37

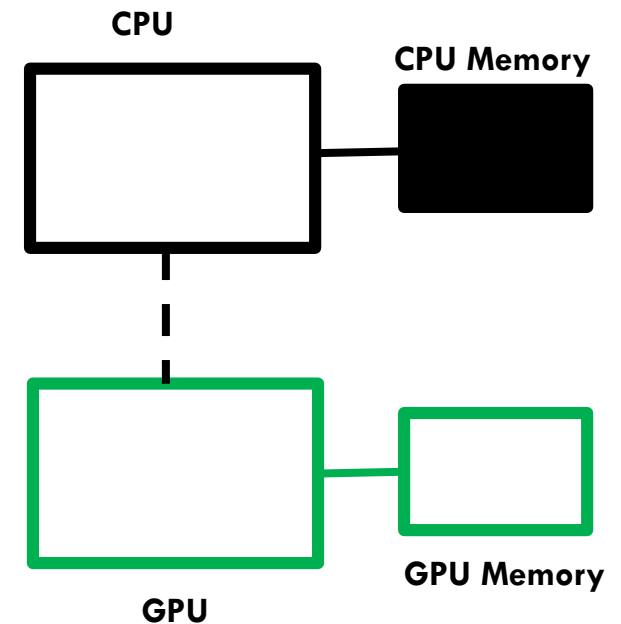
```
Int main() {  
  
    ...  
  
    // Check data for correctness  
  
    for (int i=0; i<N; i++)  
    {  
        if(C[i] != 3)  
        {  
            // Error – value of C[i] is not correct!  
        }  
    }  
  
    ...  
}
```



# A Basic CUDA Program Outline

38

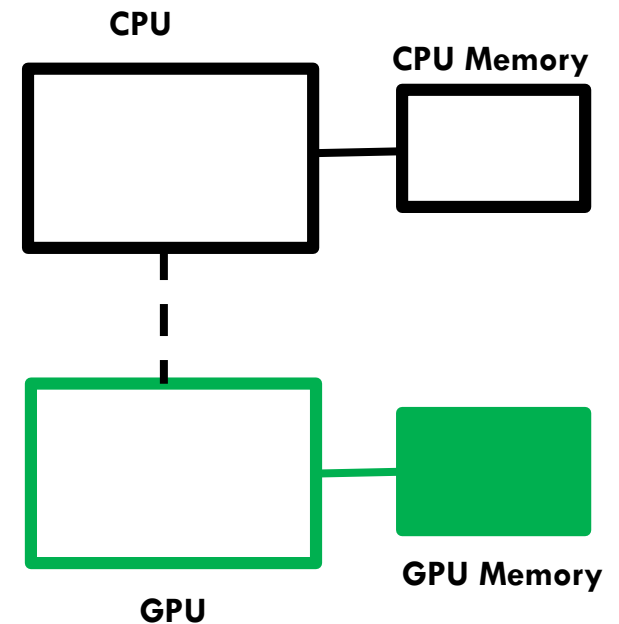
```
Int main() {  
    ...  
  
    // Free Host Memory  
  
    free(A);  
  
    free(B);  
  
    free(C);  
  
    ...  
}
```



# A Basic CUDA Program Outline

39

```
Int main() {  
  
    ...  
  
    // Free Device Memory  
  
    cudaFree(d_A);  
  
    cudaFree(d_B);  
  
    cudaFree(d_C);  
  
}
```



# CUDA Kernels

40

- What is difference between serial and parallel implementation?

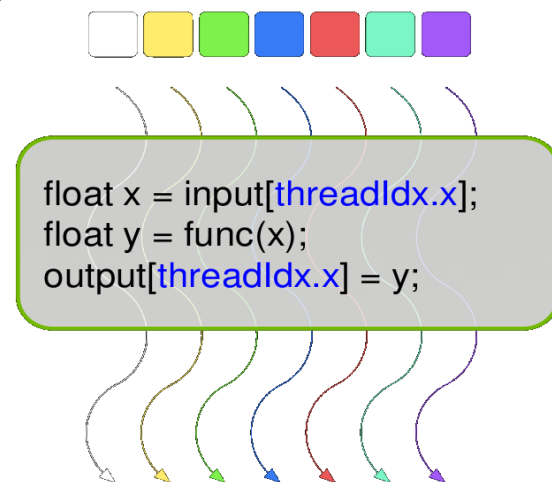
## Serial – CPU

```
for (int i=0; i<N; i++){  
    C[i] = A[i] + B[i];  
}
```

## Parallel – GPU

```
C[i] = A[i] + B[i];
```

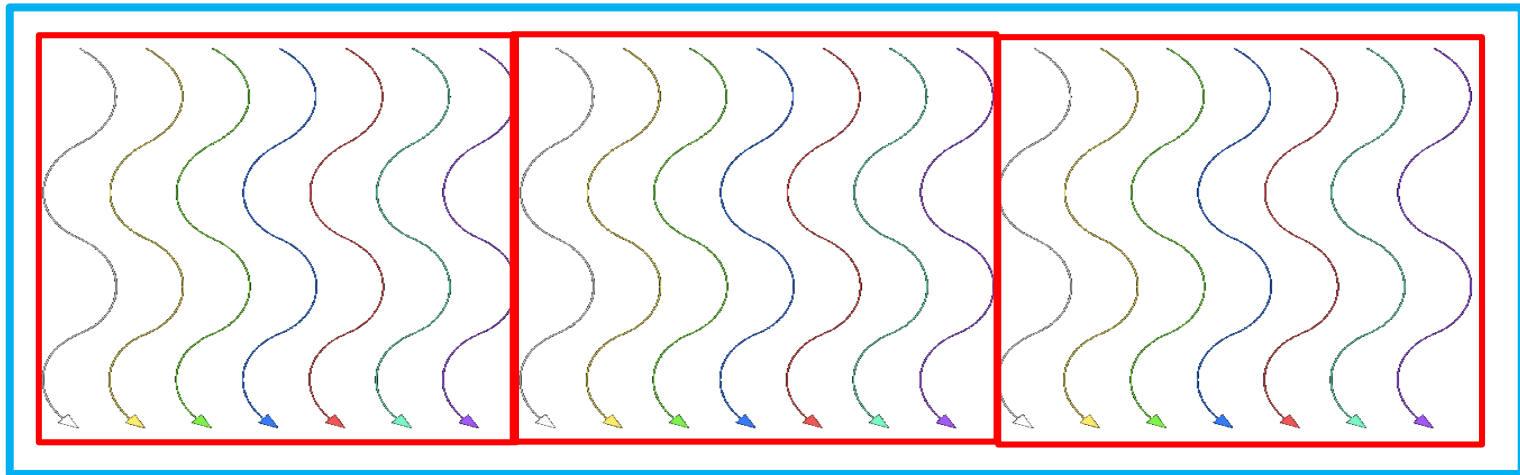
- A kernel is a function executed on the GPU as an array of threads in parallel
- Same code is executed by all threads
  - ▣ Single-Program Multiple-Data (SPMD)
- Each thread has:
  - ▣ thread ID
  - ▣ inputs, and output results





# CUDA Kernels

41



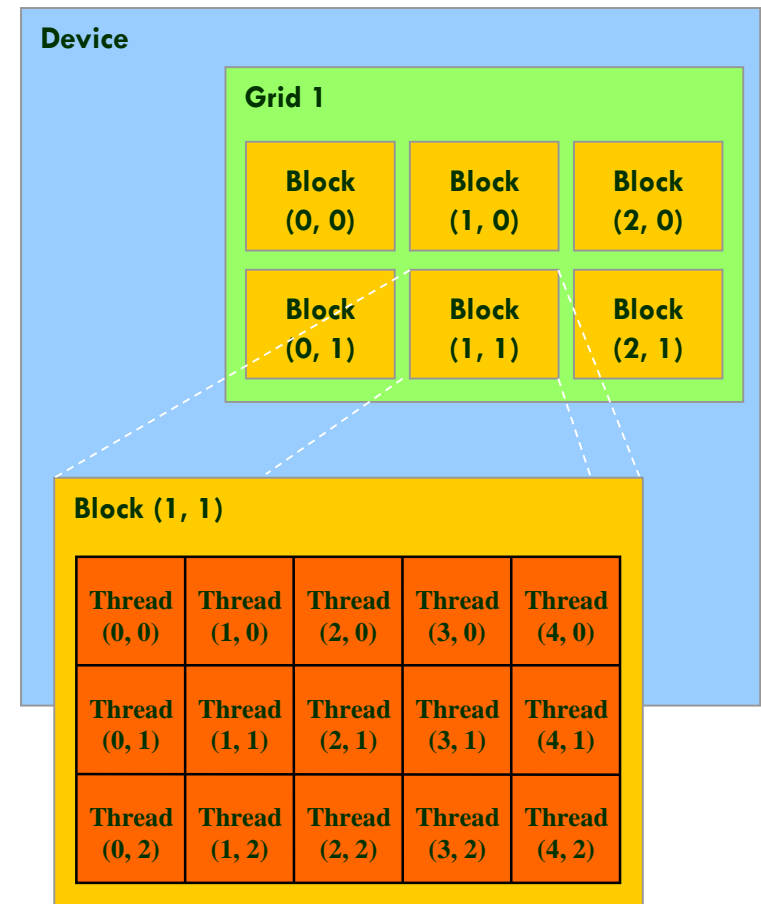
- Threads are grouped into blocks
- Blocks are grouped into a grid

**Kernel is executed as a grid of blocks of threads**

# Built-in Variables

42

- **gridDim:** Grid dimension
- **blockDim:** Block dimension
- **blockIdx:** Block index
- **threadIdx:** Thread index



# Execution Configuration

43

## □ 1D grid / 1D blocks

**gridDim.x = 1024**    **blockDim.x = 64**  
**gridDim.y = 1**      **blockDim.y = 1**  
                         **blockDim.z = 1**

```
dim3 gd(1024)
dim3 bd(64)
akernel<<<gd, bd>>>(...)
```

## □ 2D grid / 3D blocks

**gridDim.x = 4**            **blockDim.x = 64**  
**gridDim.y = 128**        **blockDim.y = 16**  
                         **blockDim.z = 4**

```
dim3 gd(4, 128)
dim3 bd(64, 16, 4)
akernel<<<gd, bd>>>(...)
```

# CUDA Function Declarations

44

Functions	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
  - ▣ Must return void
  - ▣ Can only call `__device__` functions

# Vector Addition Kernel (1)

45

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c)
{
    int i = blockIdx.x ;
    c[i] = a[i] + b[i];
}
```

## □ Host code

```
Int main(){
    ...
    vector_addition <<< N, 1 >>> (d_A, d_B, d_C);
}
```

**N blocks**

**Each block has one thread**

# Vector Addition Kernel (2)

46

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c )  
{  
  
    int i = threadIdx.x;  
  
    c[i] = a[i] + b[i];  
}
```

What is wrong  
with this code?

## □ Host code

```
Int main(){  
    ...  
    vector_addition <<< 1 , N >>> (d_A, d_B, d_C);  
}
```

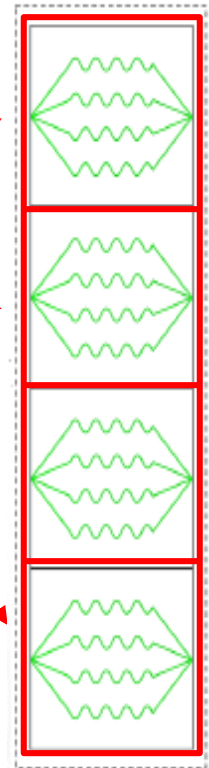
One block with N threads

# Vector Addition Kernel (3)

47

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c )  
{  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i<N)  
        c[i] = a[i] + b[i];  
}
```

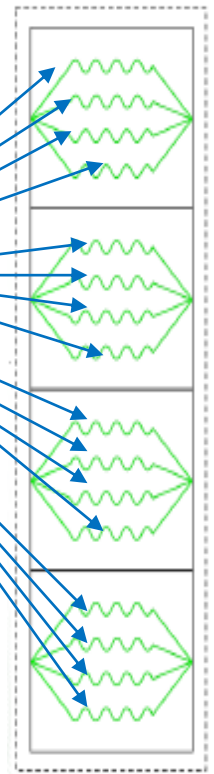
**Blocks**

# Vector Addition Kernel (3)

48

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c )  
{  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < N)  
        c[i] = a[i] + b[i];  
}
```

**Threads**

**This defines a unique thread id among all threads in a grid**

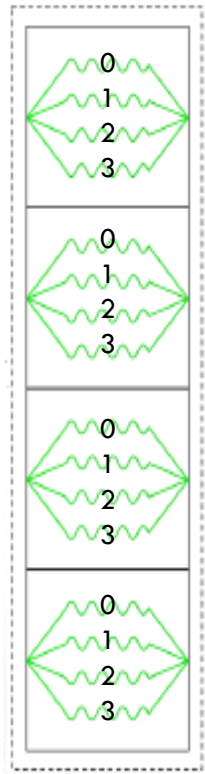


# Vector Addition Kernel (3)

49

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c )  
{  
    (4)      (0-3)      (0-3)  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < N)  
        c[i] = a[i] + b[i];  
}
```

**0****1****2****3**

# Vector Addition Kernel (3)

50

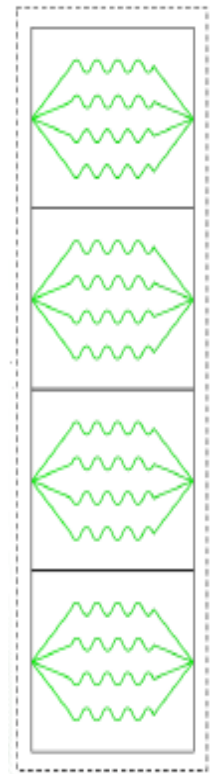
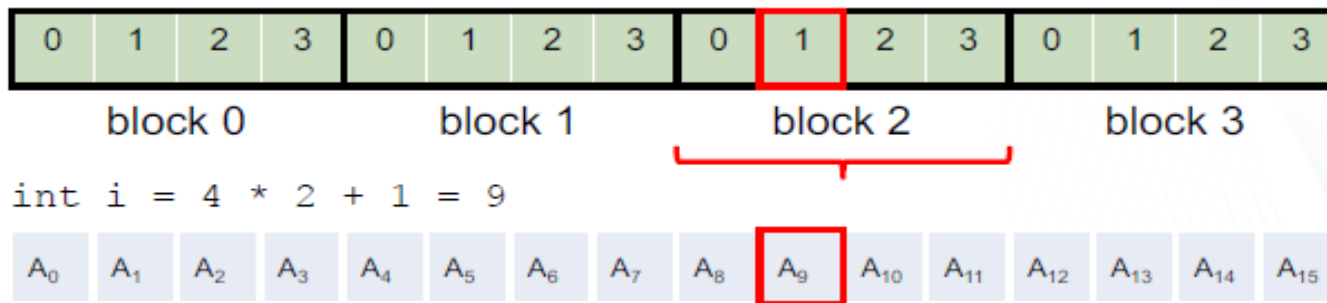
## □ Device code

```

__global__ vector_addition (int *a, int *b, int *c )
{
    (4)      (2)      (1)
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < N)
        c[i] = a[i] + b[i];
}

```



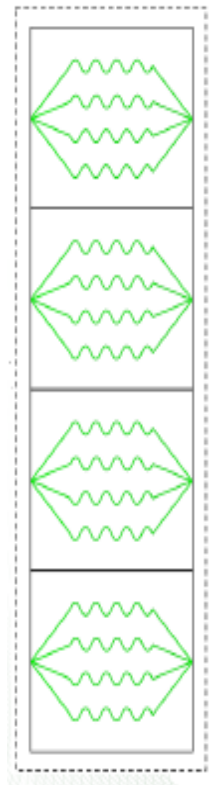
# Vector Addition Kernel (3)

51

## □ Device code

```
__global__ vector_addition (int *a, int *b, int *c )  
{  
  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i<N)  
        c[i] = a[i] + b[i];  
}
```

**Number of threads in the grid might be larger than  
number of elements in array**



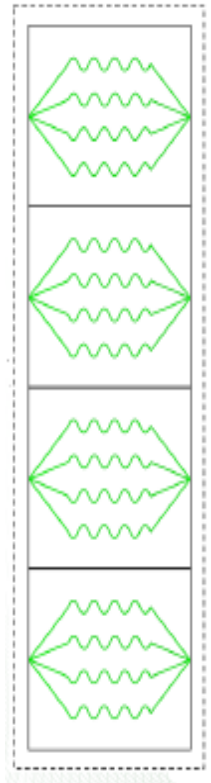
# Vector Addition Kernel (3)

52

## □ Device code

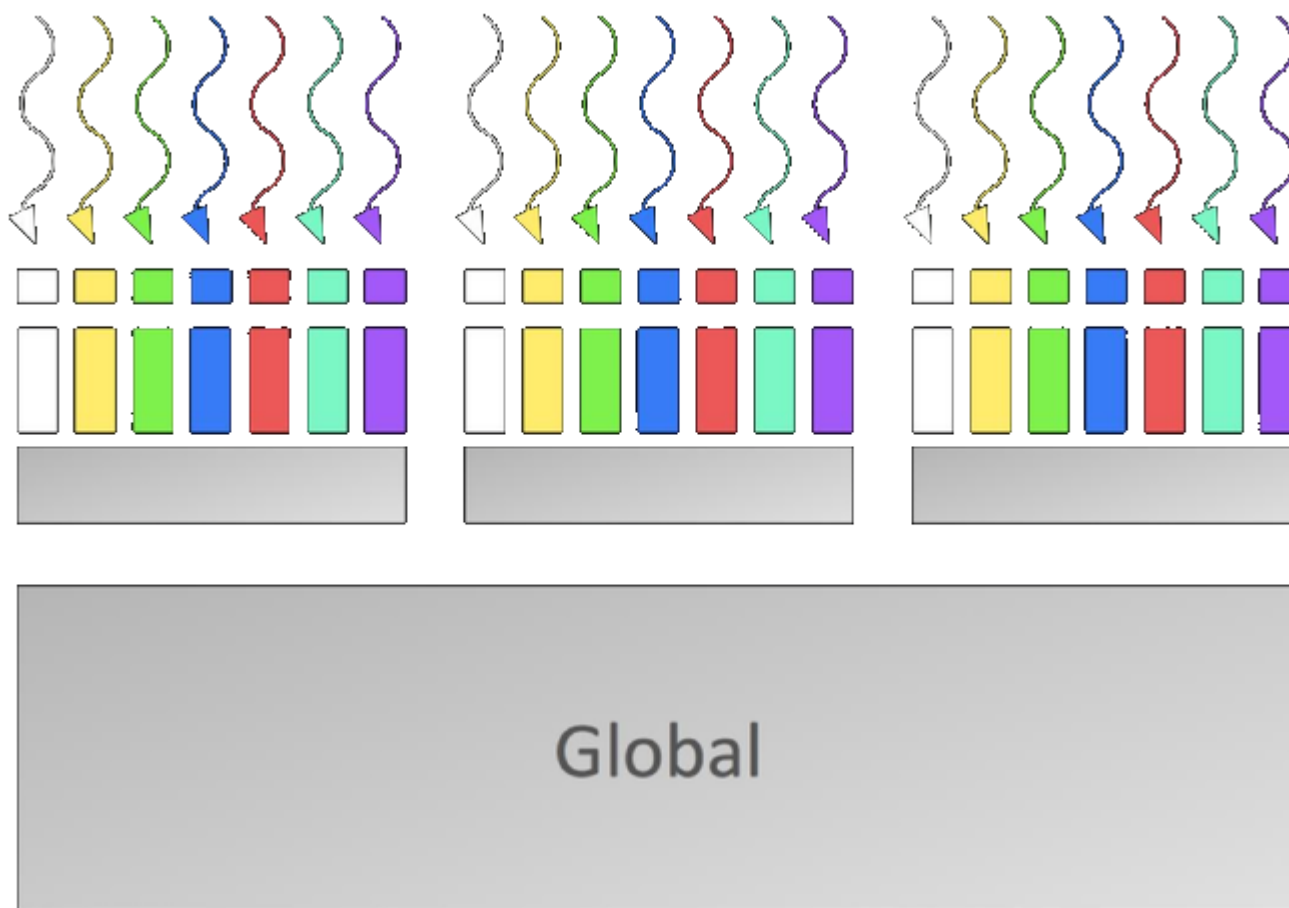
```
__global__ vector_addition (int *a, int *b, int *c )  
{  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if (i < N)  
        c[i] = a[i] + b[i];  
}
```

**Local variables are private to each thread.  
The loop was replaced by a grid of threads**



# Memory Model

53



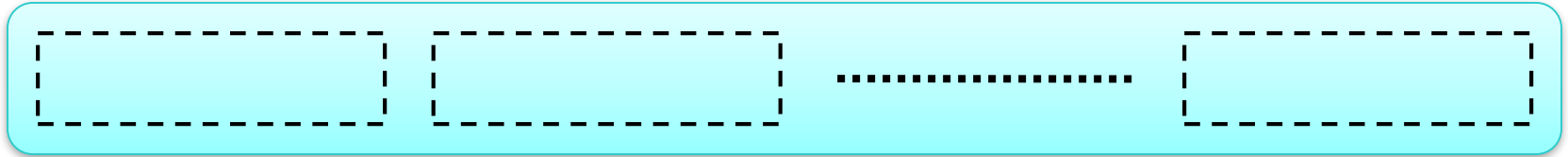
# Shared Memory

54

- Very fast on-chip memory
- Allocated per thread block
  - ▣ Allows data sharing between threads in the same block
  - ▣ Declared with `__shared__` specifier
- Limited amount
- Must take care to avoid race conditions. For example...
  - ▣ Say, each thread writes the value 1 to one element of an array element
  - ▣ Then one thread sums up the elements of the array
  - ▣ Synchronize with `__syncthreads()`

# Using Shared Memory

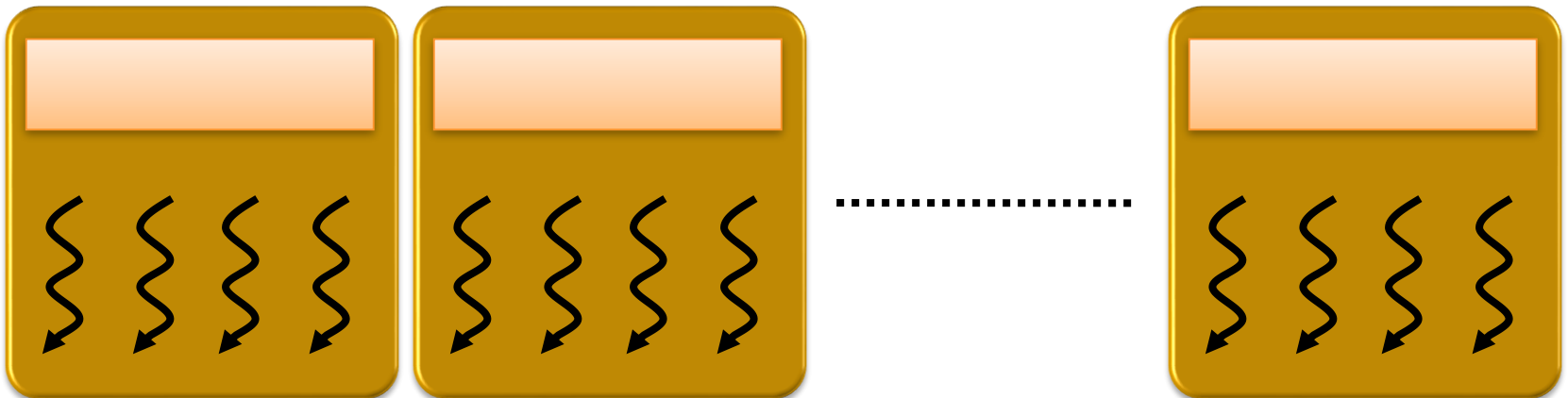
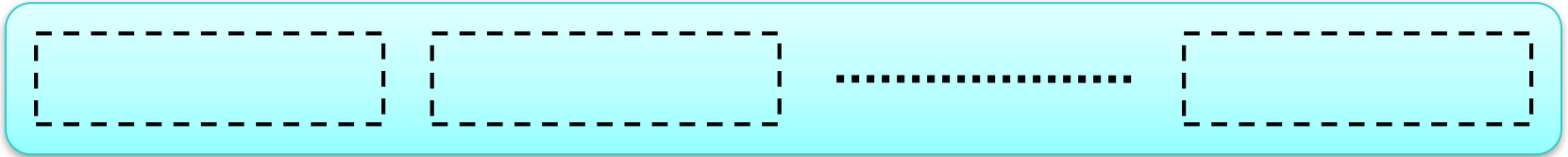
55



- Partition data into subsets that fit into **shared memory**

# Using Shared Memory

56

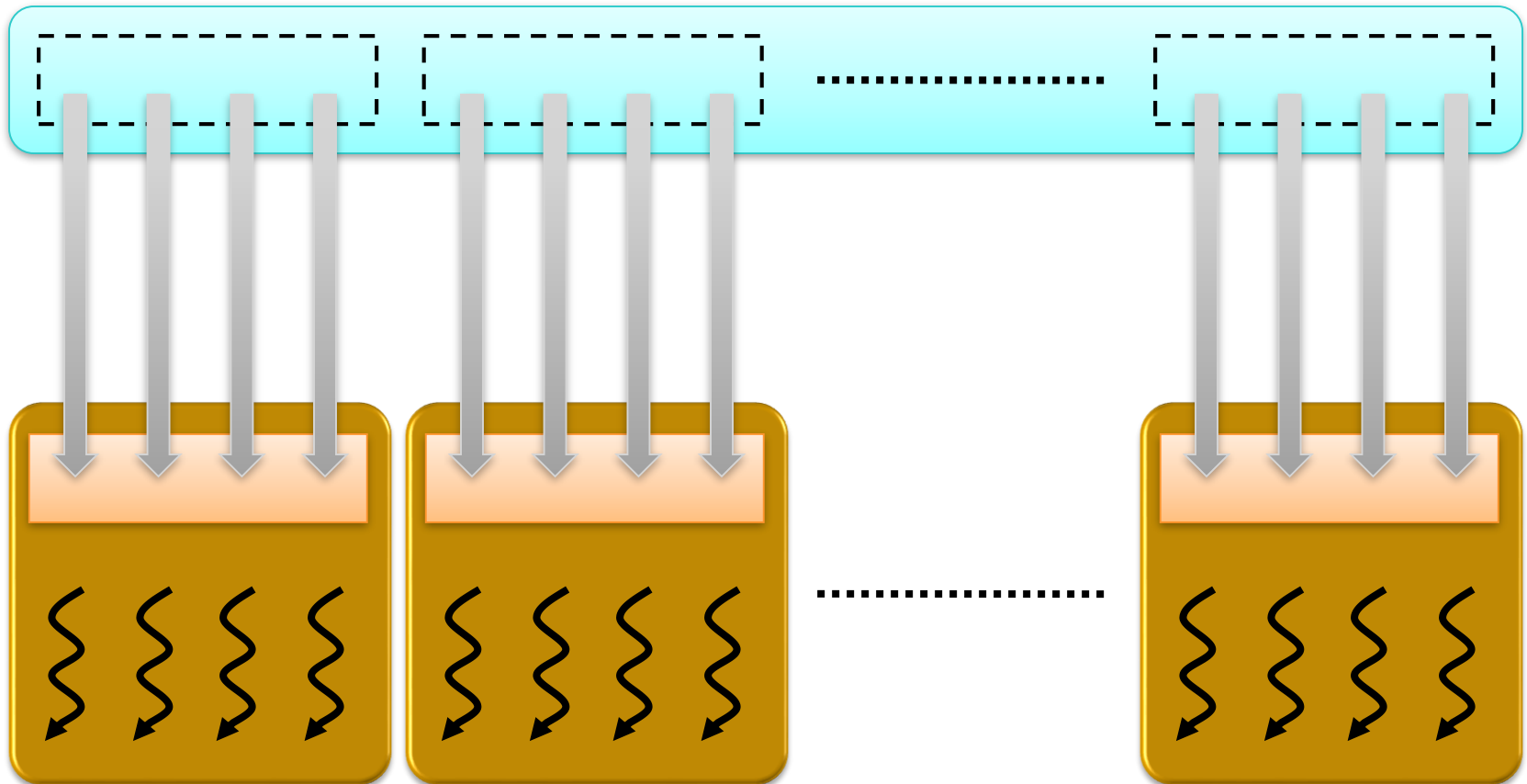


- Handle each data subset with one thread block



# Using Shared Memory

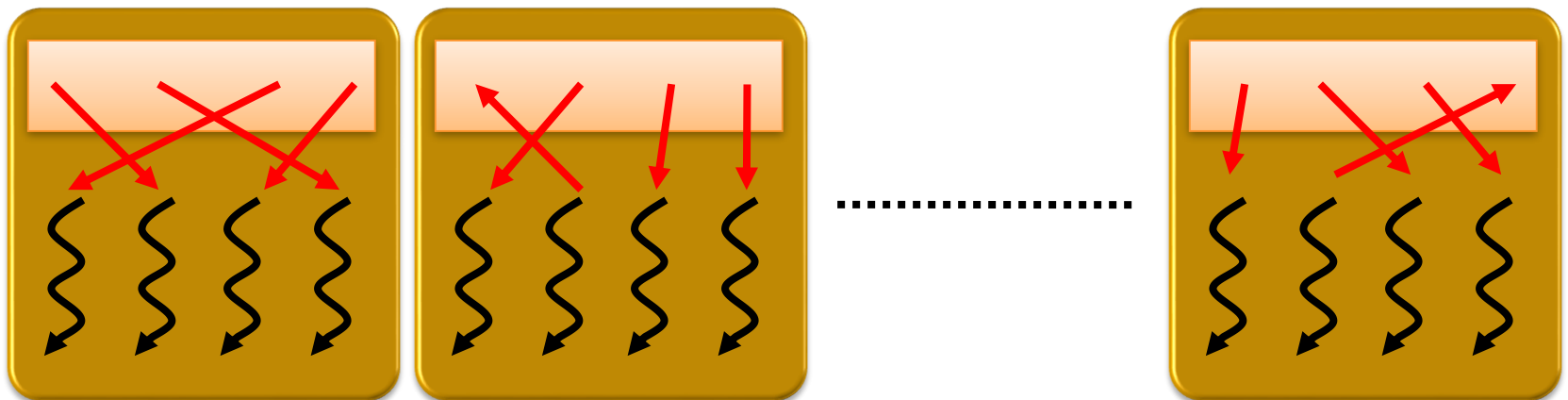
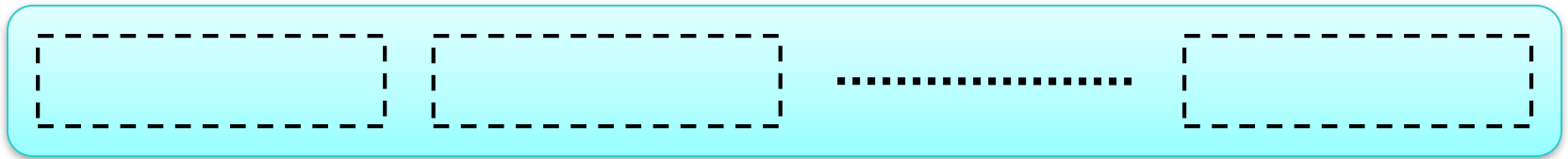
57



- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

# Using Shared Memory

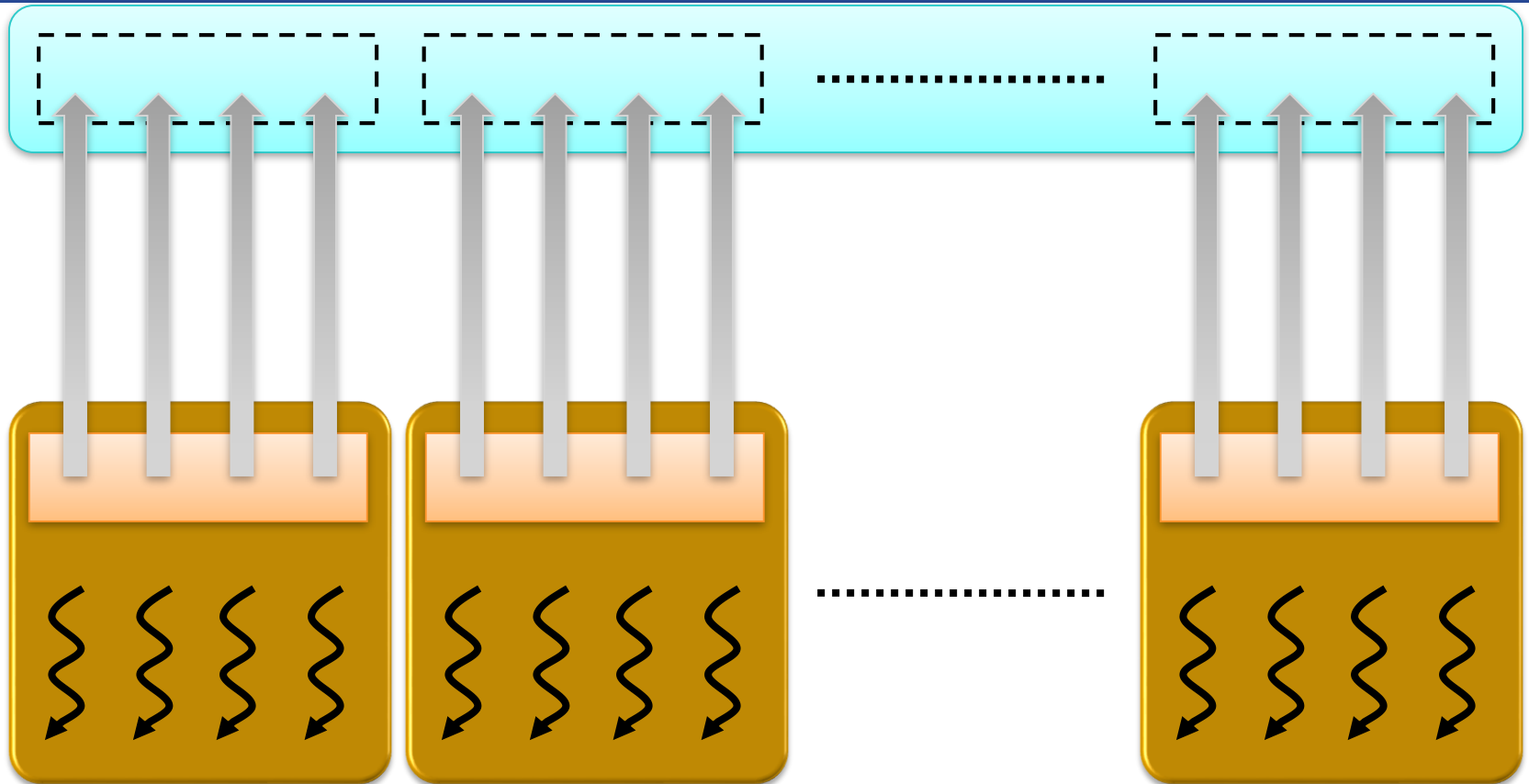
58



- Perform the computation on the subset from **shared memory**

# Using Shared Memory

59

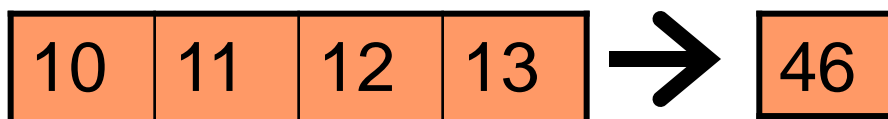


- Copy the result from **shared memory** back to global memory

# Reduction Operations

60

- Multiple values are reduced into a single value
  - ▣ ADD, MUL, AND, OR, ....



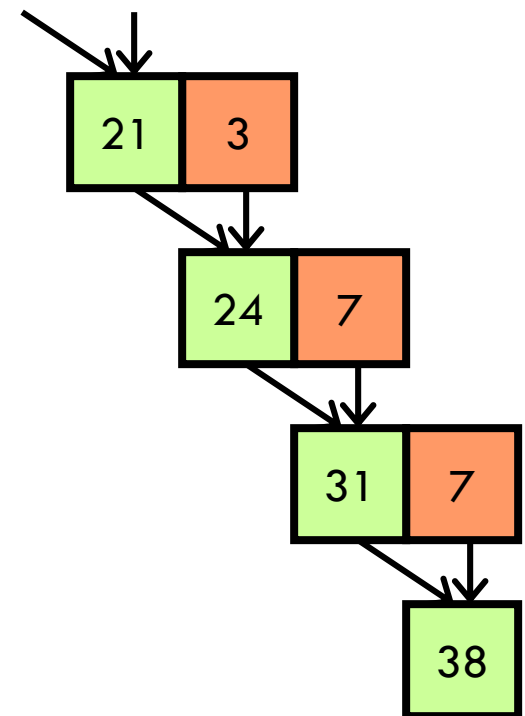
- Useful primitive in parallel computing
  - ▣ Many tasks generating intermediate results
  - ▣ Reduction must be done to combine the intermediate results into final results
- Easy enough to allow us to focus on optimization techniques

# Sequential Reduction

61

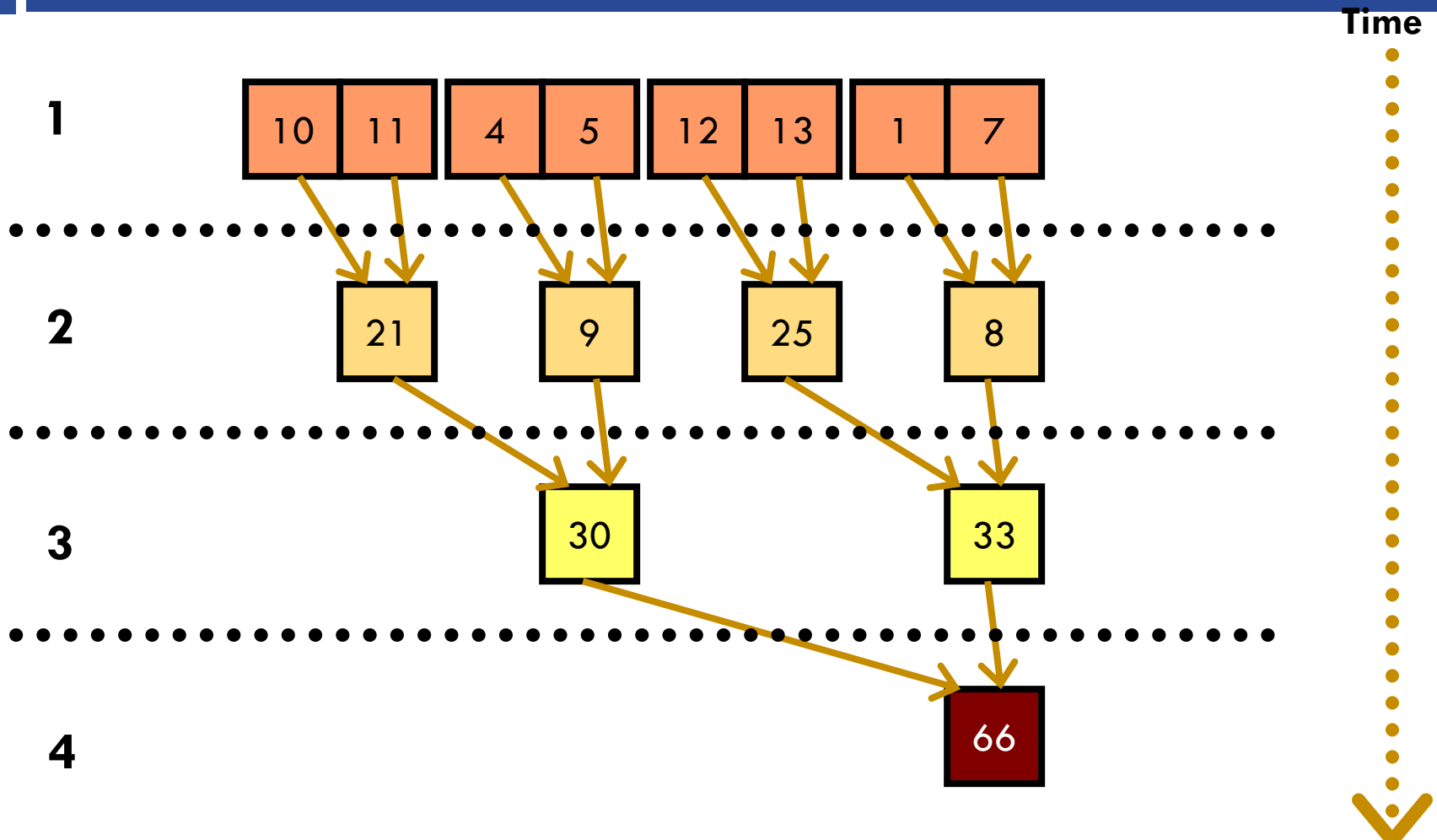
```
sum = array[0];  
for (k = 1; k < N; k++) {  
    sum += array[i];  
}
```

- Start with the first two elements --> partial result
- Process the next element
- $O(N)$



# Parallel Reduction

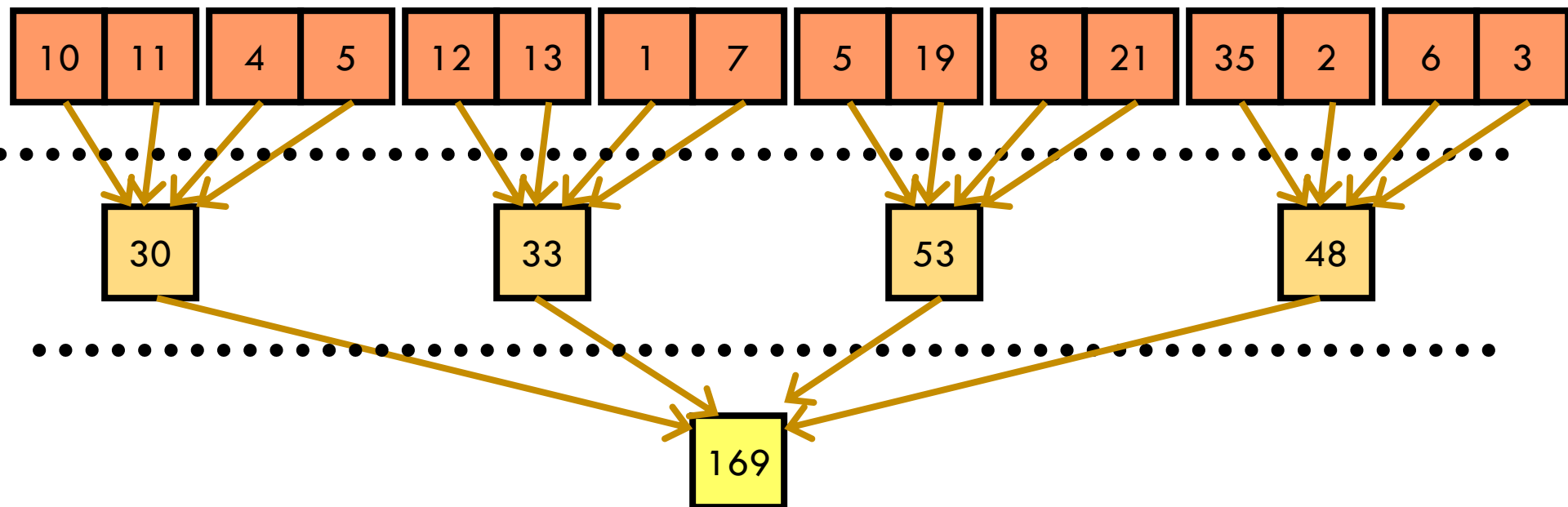
62



$\log_2(N)$ , where  $N$  is the number of elements

# Trees with Larger Degrees

63



$\log_4(N)$ , where  $N$  is the number of elements

# Single Thread Block

64

```
__global__ void
reduce(float *g_idata, float *g_odata, int n)
{
    int tid = threadIdx.x;

    // copy input data into output data
    g_odata[tid] = g_idata[tid];

    for (int s = 1; s < blockDim.x; s *= 2) {
        if ((tid % (2*s)) == 0) {
            g_odata[tid] += g_odata[tid + s];
        }
        __syncthreads();
    }
}
```

Why do we need this?

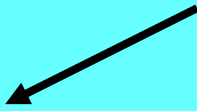


# Using Shared Memory

65

```
__global__ void  
reduce(float *g_idata, float *g_odata, int n)  
{  
    int tid = threadIdx.x;  
    __shared__ float s_data[BLOCK_DIM];
```

Shared memory



```
    // copy input data into output data
```

```
    s_data[tid] = g_idata[tid];
```

```
    __syncthreads();
```

```
    for (int s = 1; s < blockDim.x; s *= 2) {
```

```
        if ((tid % (2*s)) == 0) {
```

```
            s_data[tid] += s_data[tid + s];
```

```
        }
```

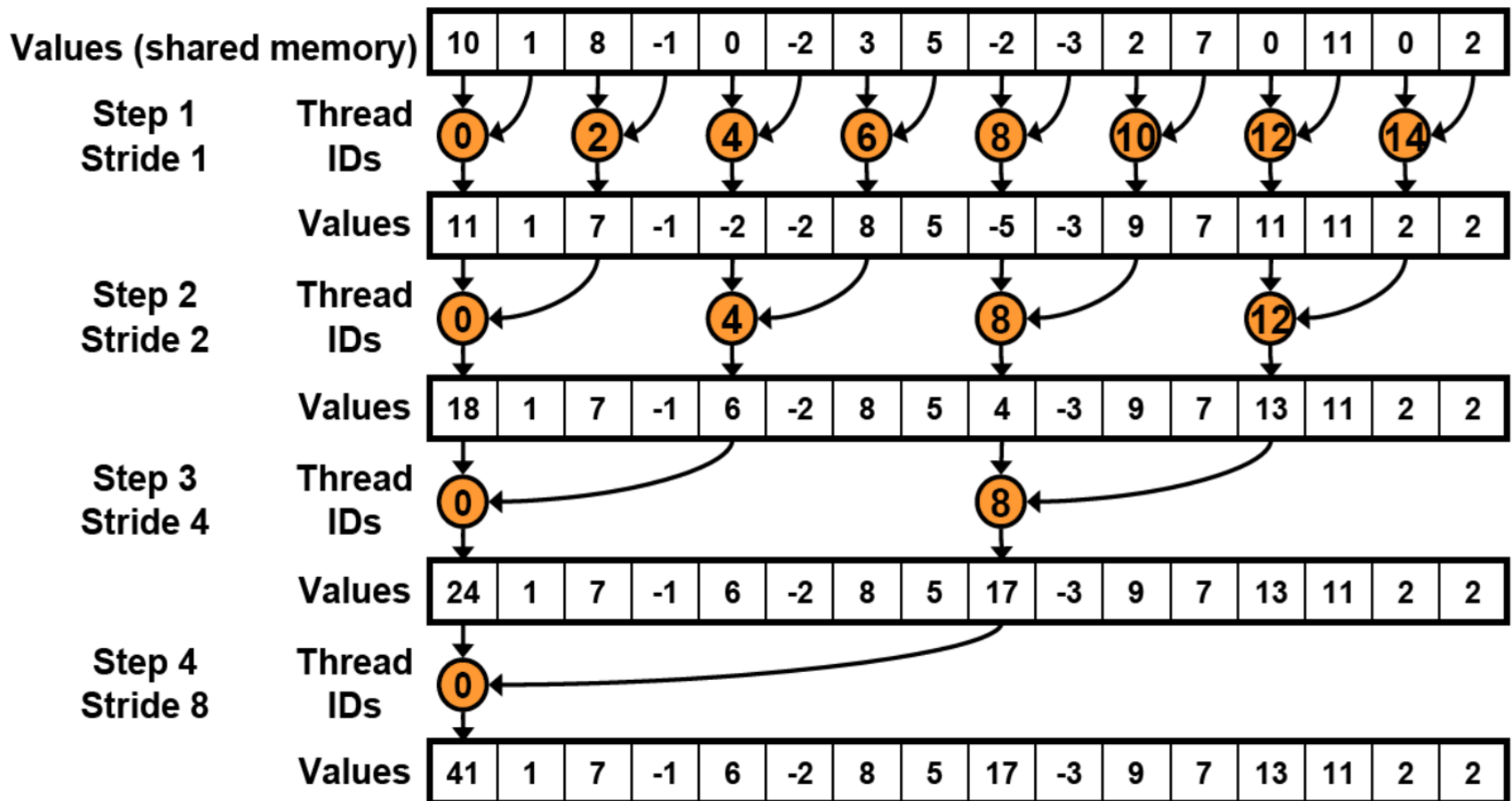
```
        __syncthreads();
```

```
    }
```

```
    g_odata[tid] = s_data[tid]; }
```

# Reduction Steps

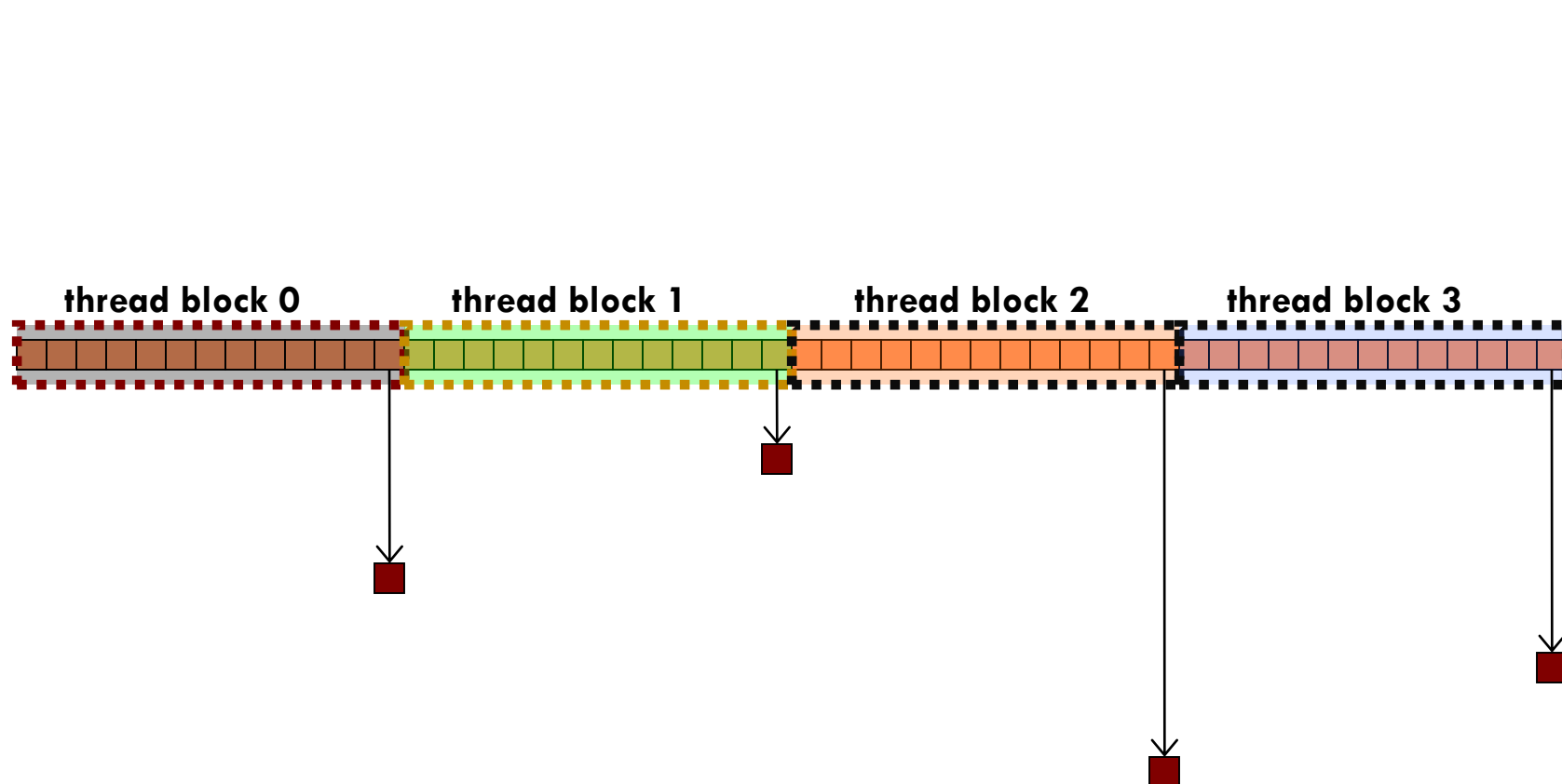
66



# How about Multiple Thread Blocks

67

□ How do we communicate results across blocks? Time



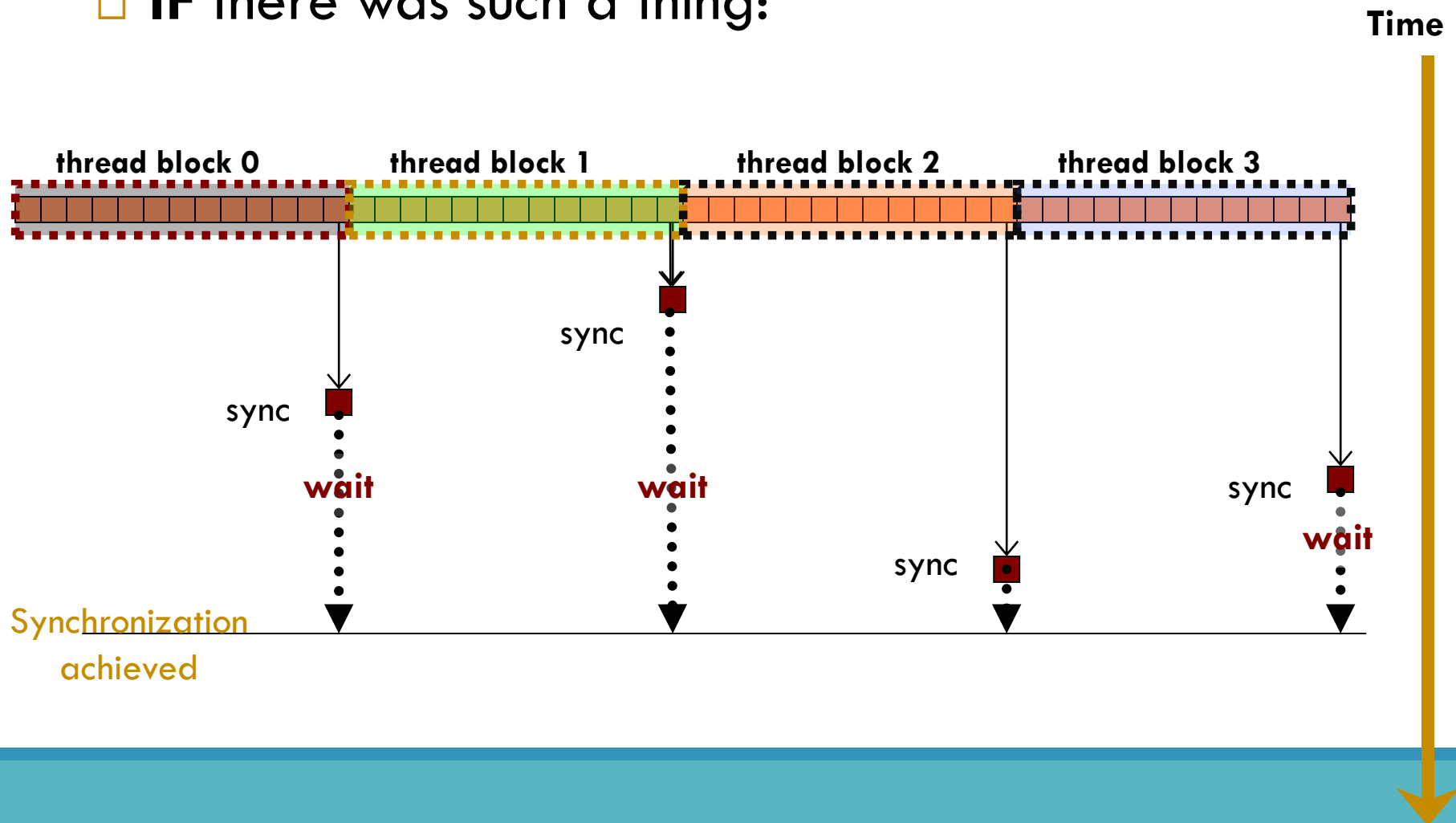
□ The key problem is **synchronization**:

▣ How do we know that each block has finished?

# Global Synchronization

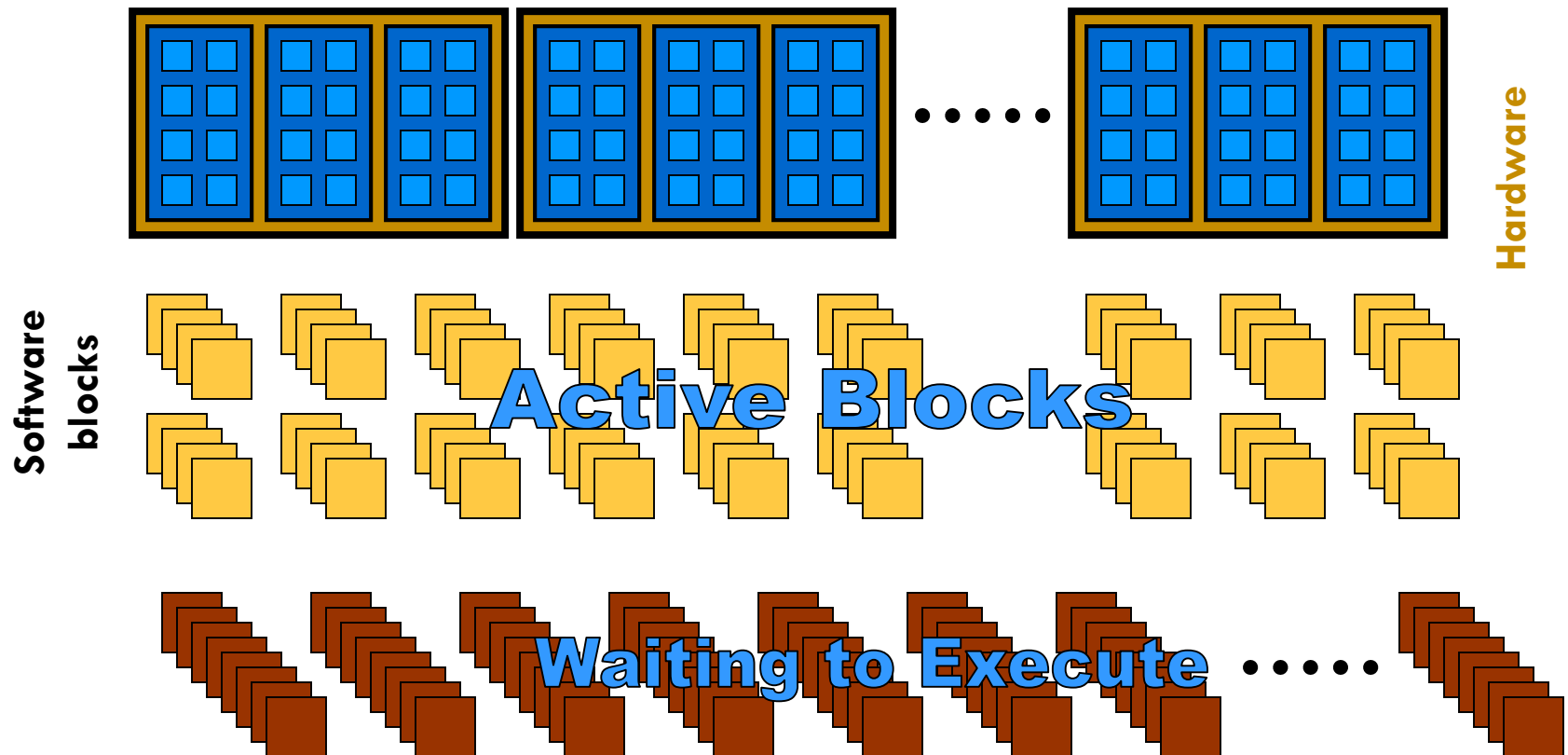
68

□ **IF** there was such a thing:



# Global Synchronization

69



70

atomics

# The Problem

71

- How do you do global communication?
- Finish a grid and start a new one

# Global Communication

72

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);  
// The system ensures that all  
// writes from step1 complete.  
step2<<<grid2,blk2>>>(...);
```



# Global Communication

73

- Would need to decompose kernels into before and after parts

# Race Conditions

74

- Or, write to a predefined memory location
  - ▣ Race condition! Updates can be lost

# Race Conditions

75

```
threadId:0
```

```
// vector[0] was equal to 0
```

```
vector[0] += 5;
```

```
...
```

```
a = vector[0];
```

```
threadId:1917
```

```
vector[0] += 1;
```

```
...
```

```
a = vector[0];
```

- What is the value of *a* in thread 0?
- What is the value of *a* in thread 1917?

# Race Conditions

76

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

# Race Conditions

77

- Answer: not defined by the programming model, can be arbitrary

# Atomics

78

- CUDA provides **atomic** operations to deal with this problem

# Atomics

79

- An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in fermi

# Example: Histogram

80

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                          int* buckets)
{
    int i = threadIdx.x
          + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```



# Example: Workqueue

81

```
// For algorithms where the amount of work per item  
// is highly non-uniform, it often makes sense for  
// to continuously grab work from a queue
```

```
__global__
```

```
void workq(int* work_q, int* q_counter,  
          int* output, int queue_max)
```

```
{
```

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

```
    int q_index =  
        atomicInc(q_counter, queue_max);
```

```
    int result = do_work(work_q[q_index]);  
    output[i] = result;
```

```
}
```

# Atomics

82

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on G80!

# Example: Global Min/Max (Naive)

83

```
// If you require the maximum across all threads  
// in a grid, you could do it with a single global  
// maximum value, but it will be VERY slow
```

```
__global__
```

```
void global_max(int* values, int* gl_max)
```

```
{
```

```
    int i = threadIdx.x
```

```
        + blockDim.x * blockIdx.x;
```

```
    int val = values[i];
```

```
    atomicMax(gl_max, val);
```

```
}
```

# Example: Global Min/Max (Better)

84

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__
void global_max(int* values, int* max,
               int *regional_maxes,
               int num_regions)
{
    // i and val as before ...
    int region = i % num_regions;
    if(atomicMax(&reg_max[region], val) < val)
    {
        atomicMax(max, val);
    }
}
```

# Global Min/Max

85

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously
- See next lecture for even better version!

# Summary

86

- Can't use normal load/store for inter-thread communication because of **race conditions**
- Use atomic instructions for sparse and/or unpredictable global communication
  - ▣ See next lectures for shared memory and scan for other communication patterns
- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism

# Questions?

88

# SM Execution & Divergence



# How an SM executes threads

89

- Overview of how a Stream Multiprocessor works
- SIMT Execution
- Divergence

# Scheduling Blocks onto SMs

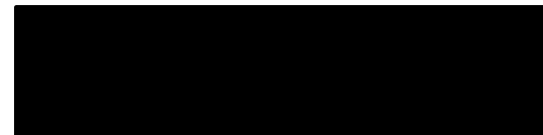
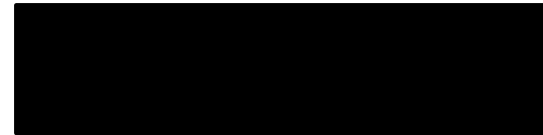
90

## Streaming Multiprocessor



Thread Block 5

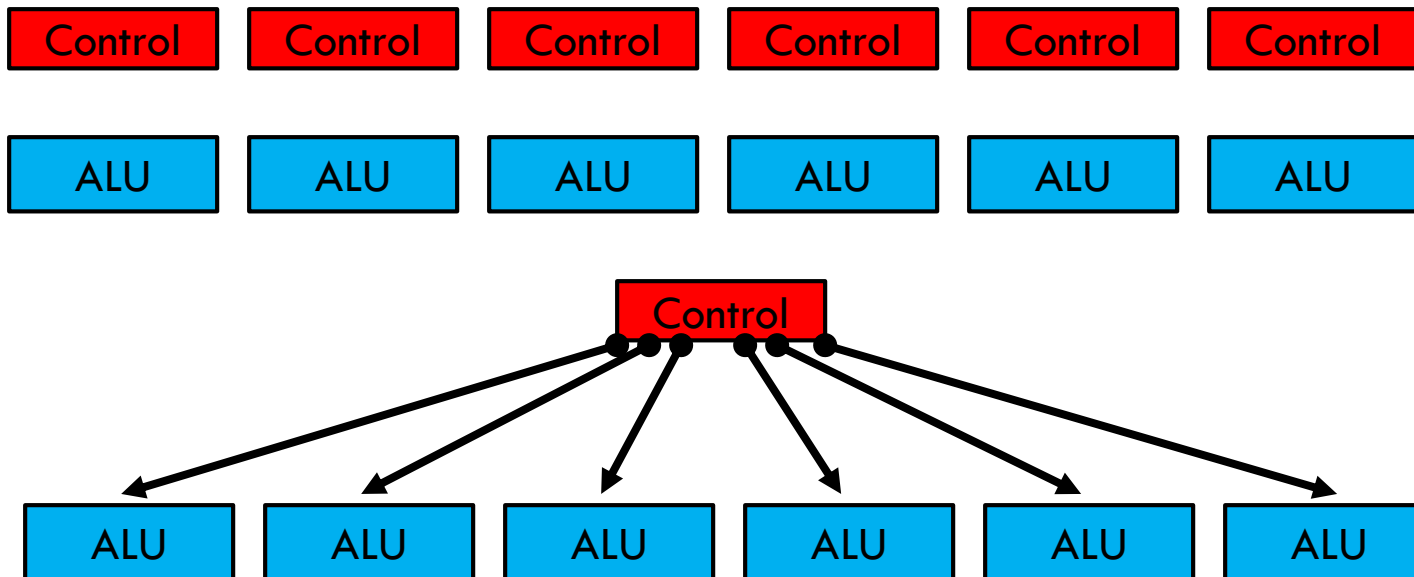
Thread Block 27



- **HW Schedules thread blocks onto available SMs**
  - No guarantee of ordering among thread blocks
  - HW will schedule thread blocks as soon as a previous thread block finishes

# Warps

91

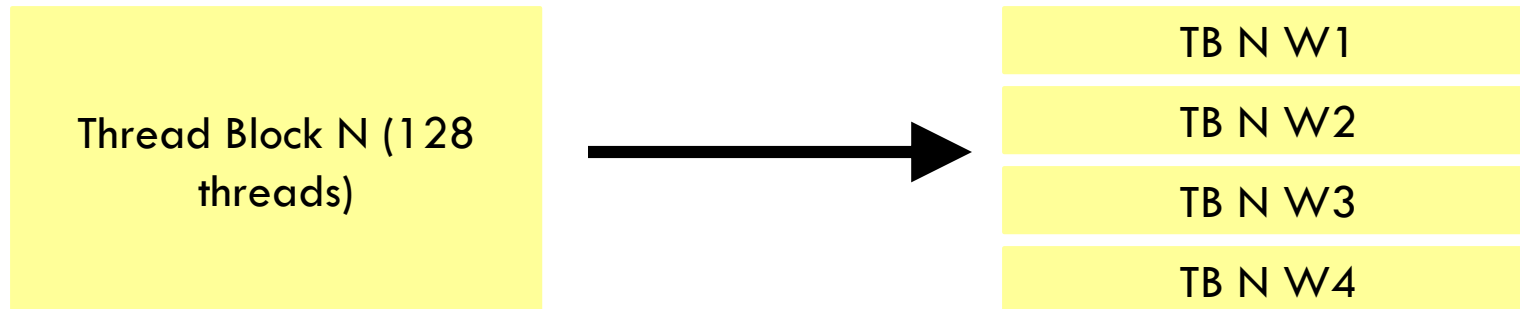


- A **warp** = 32 threads launched together
- Usually, execute together as well

# Mapping of Thread Blocks

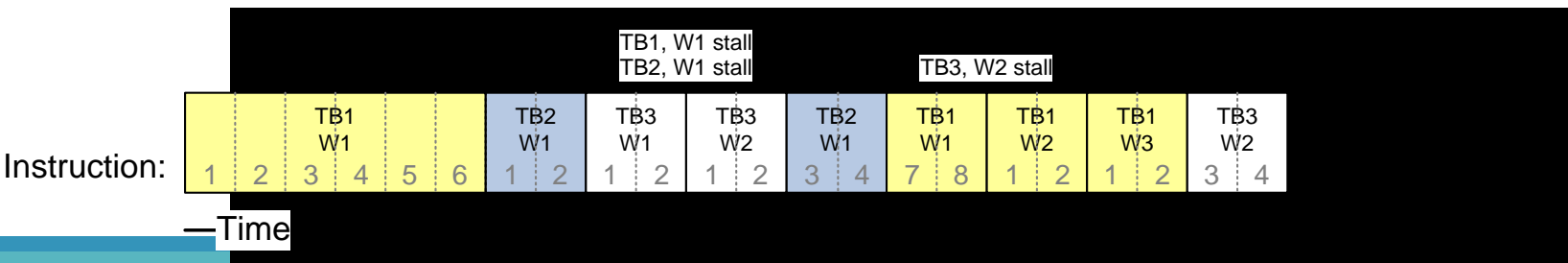
92

- Each thread block is mapped to one or more warps
- The hardware schedules each warp independently



# THREAD SCHEDULING EXAMPLE

- SM implements zero-overhead warp scheduling
  - ▣ At any time, only one of the warps is executed by SM \*
  - ▣ Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - ▣ Eligible Warps are selected for execution on a prioritized scheduling policy
  - ▣ All threads in a warp execute the same instruction when selected



# Control Flow Divergence

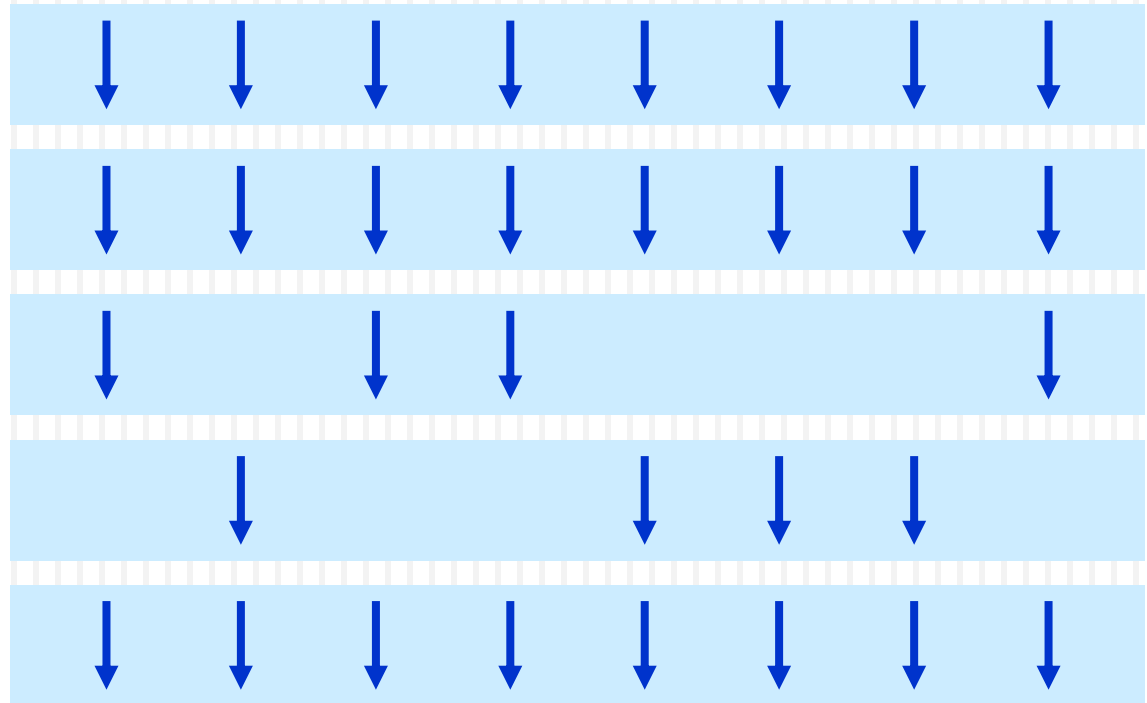
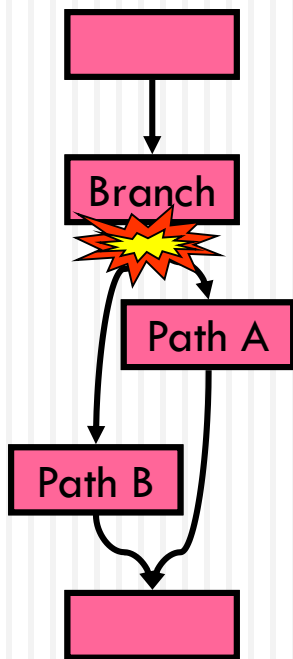
94

□ What happens if you have the following code?

```
if (foo (threadIdx.x) )  
{  
    do_A () ;  
}  
else  
{  
    do_B () ;  
}
```

# Control Flow Divergence

95



# Control Flow Divergence

96

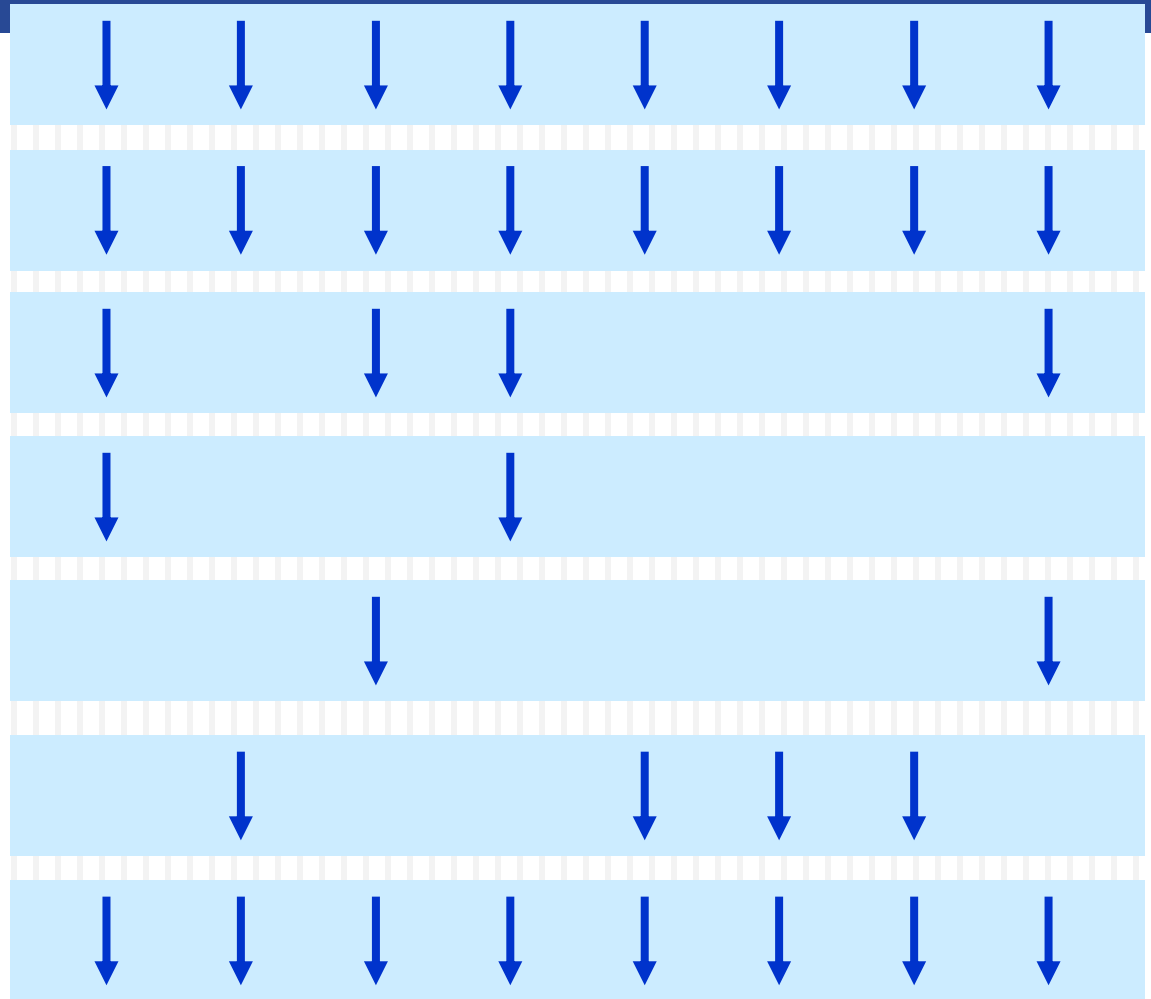
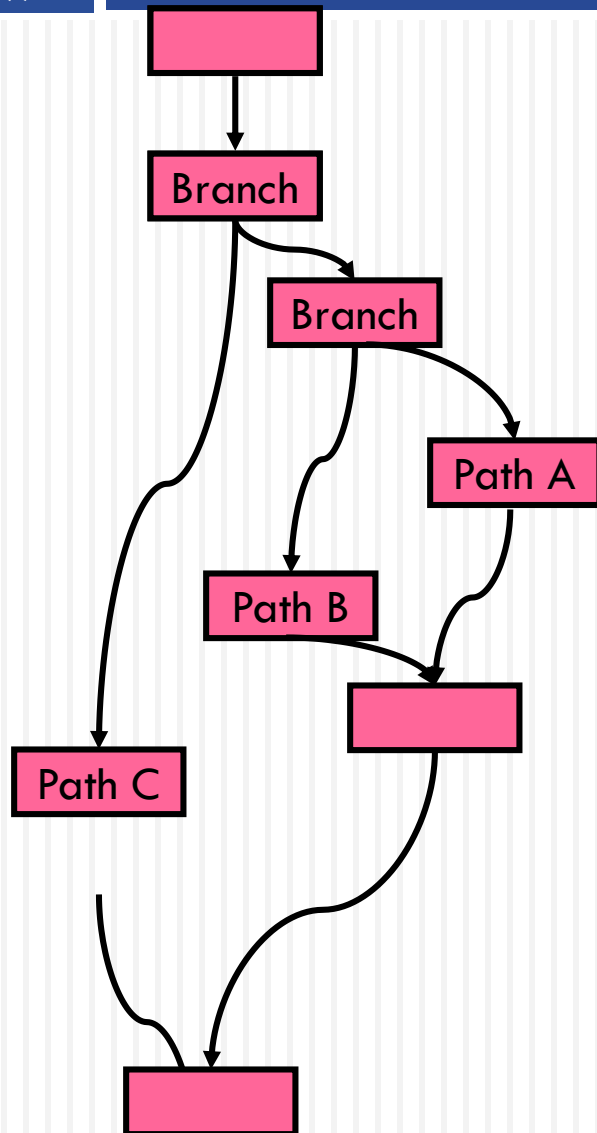
- Nested branches are handled as well

```
if (foo (threadIdx.x) )  
{  
    if (bar (threadIdx.x) )  
        do_A () ;  
    else  
        do_B () ;  
}  
else  
    do_C () ;
```



# Control Flow Divergence

97



# Control Flow Divergence

98

- You don't have to worry about divergence for correctness (\*)
- You might have to think about it for performance
  - ▣ Depends on your branch conditions

# Control Flow Divergence

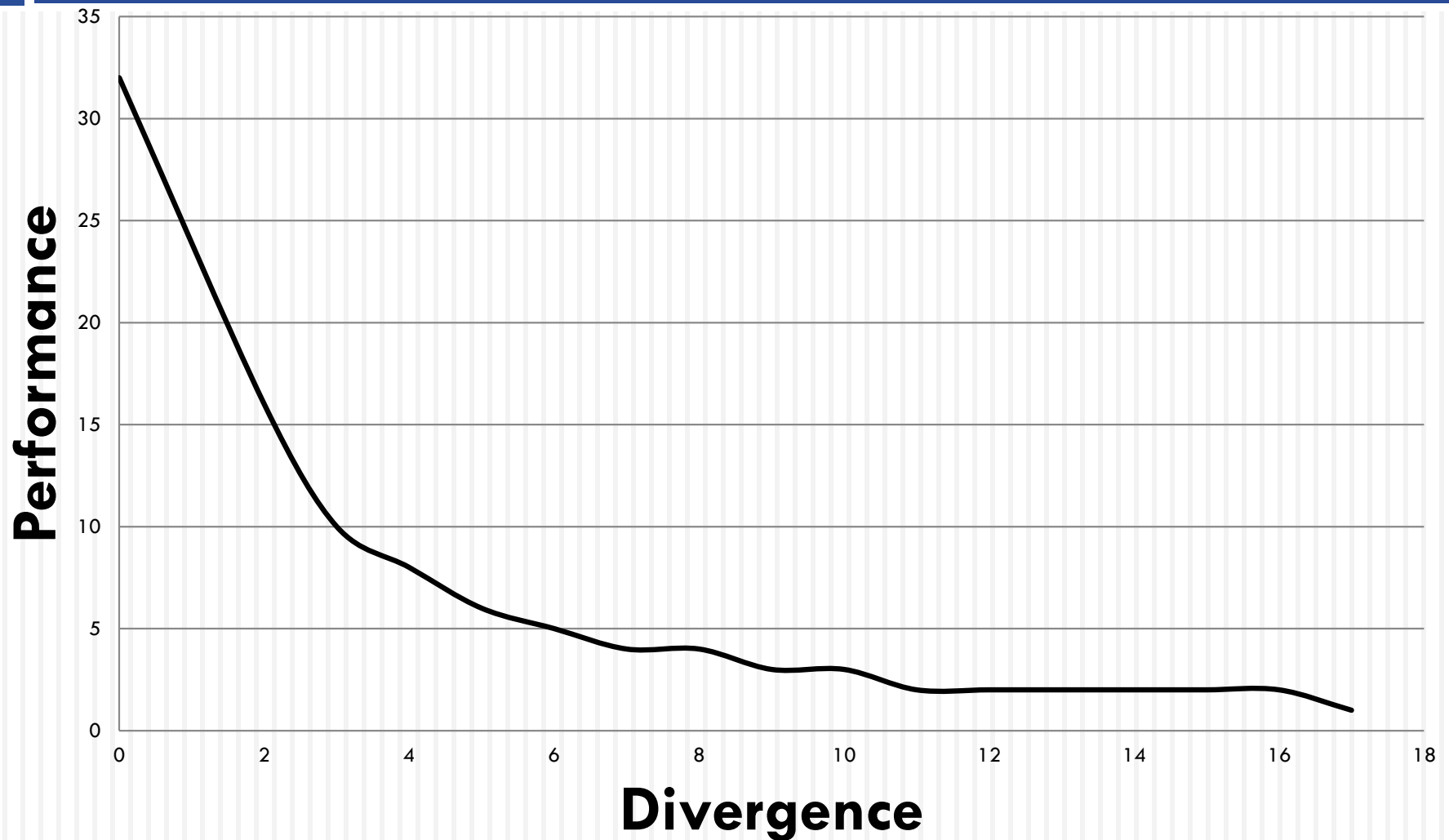
99

- Performance drops off with the degree of divergence

```
switch (threadIdx.x % N)
{
    case 0:
        ...
    case 1:
        ...
}
```

# Divergence

100



# Atomics

101

- `atomicAdd` returns the previous value at a certain address
- Useful for grabbing variable amounts of data from a list

# GPU Performance



# But First!

103

- Always measure where your time is going!
  - ▣ Even if you think you know where it is going
  
- Keep in mind Amdahl's Law when optimizing any part of your code
  - ▣ Don't continue to optimize once a part is only a small fraction of overall execution time

# Performance Considerations

104

- Memory Coalescing
- Shared Memory Bank Conflicts
- Control-Flow Divergence
- Occupancy
- Kernel Launch Overheads



105

# Memory Coalescing

# Memory Coalescing

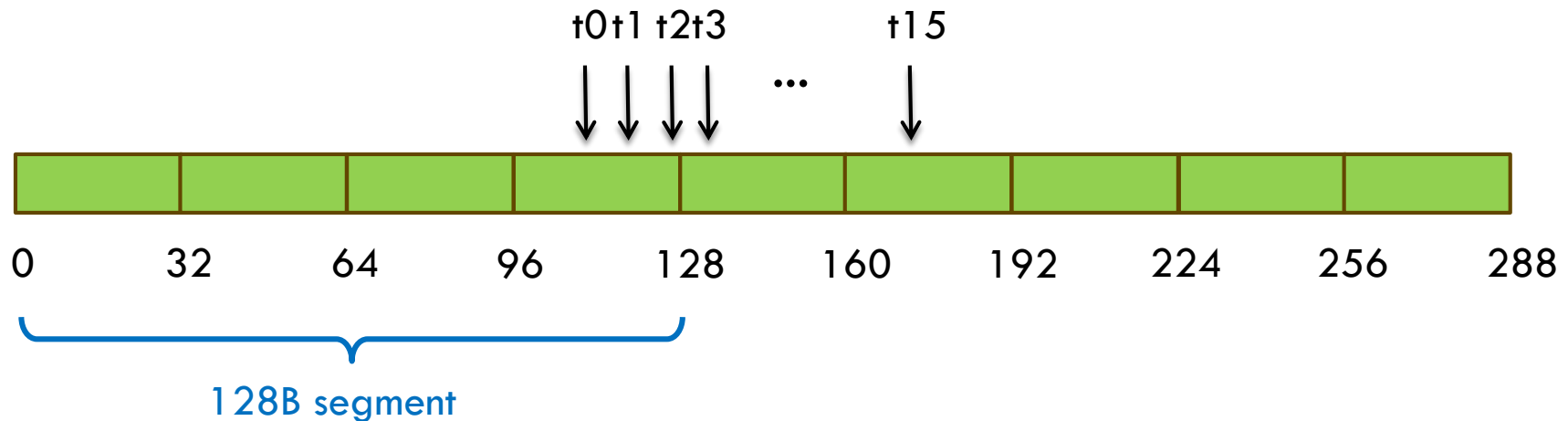
106

- Off-chip memory is accessed in chunks
  - ▣ Even if you read only a single word
  - ▣ If you don't use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32/64/128 bytes
  - ▣ Unaligned accesses will cost more

# Threads 0-15 access 4-byte words at addresses 116-176

107

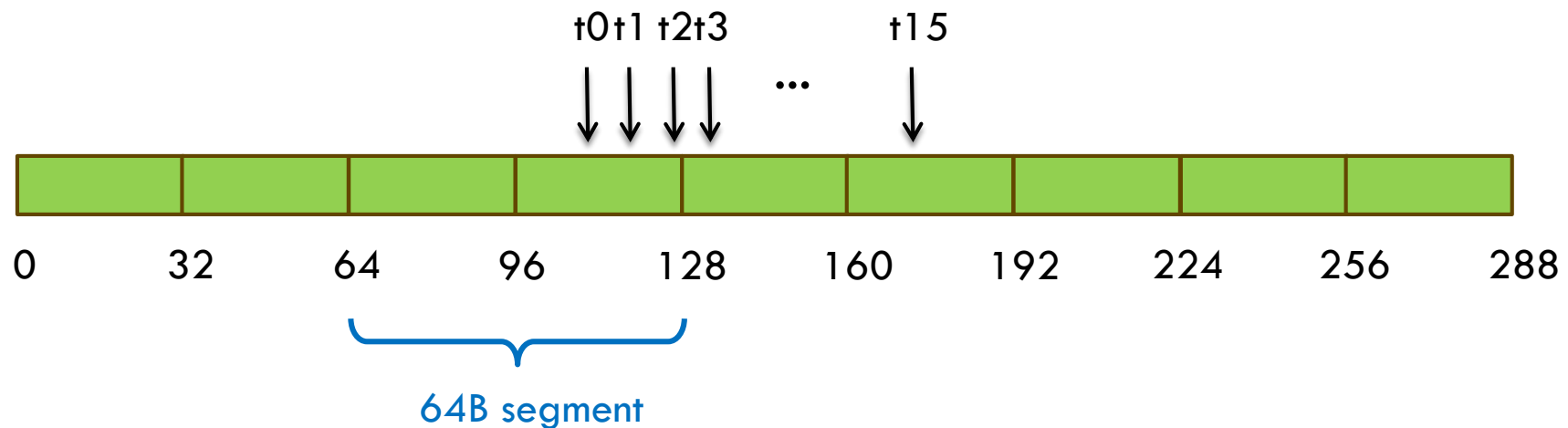
- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127



# Threads 0-15 access 4-byte words at addresses 116-176

108

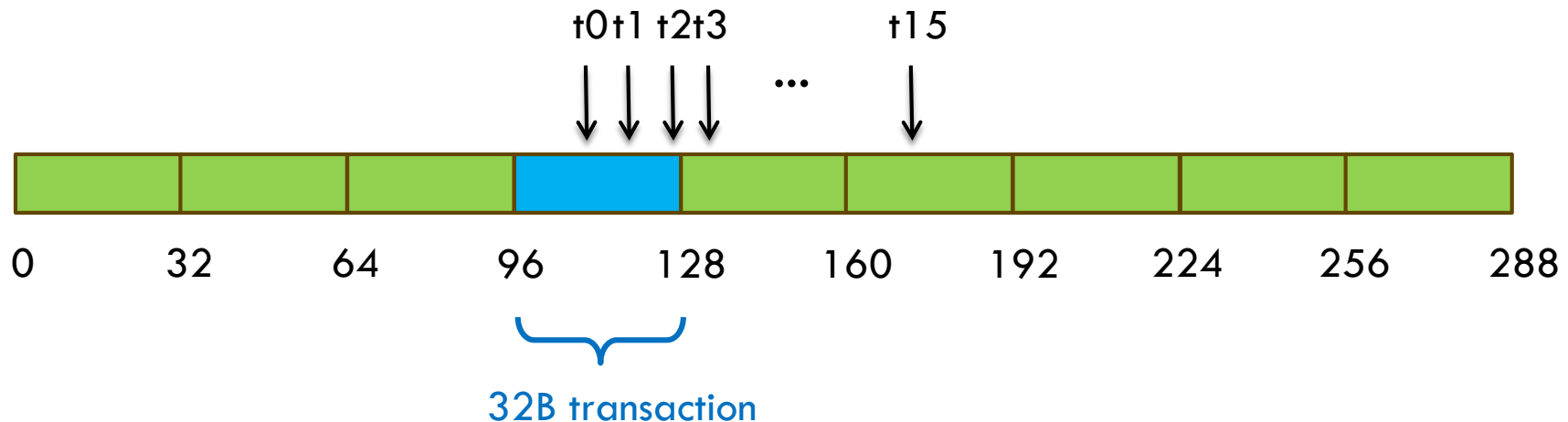
- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (**reduce to 64B**)



# Threads 0-15 access 4-byte words at addresses 116-176

109

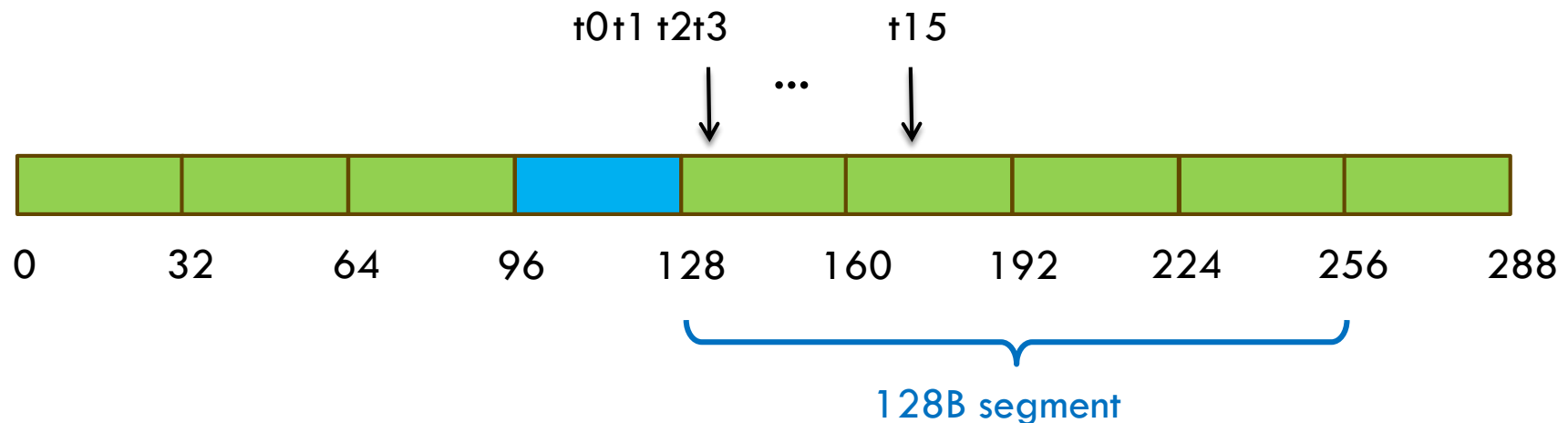
- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (**reduce to 32B**)



# Threads 0-15 access 4-byte words at addresses 116-176

110

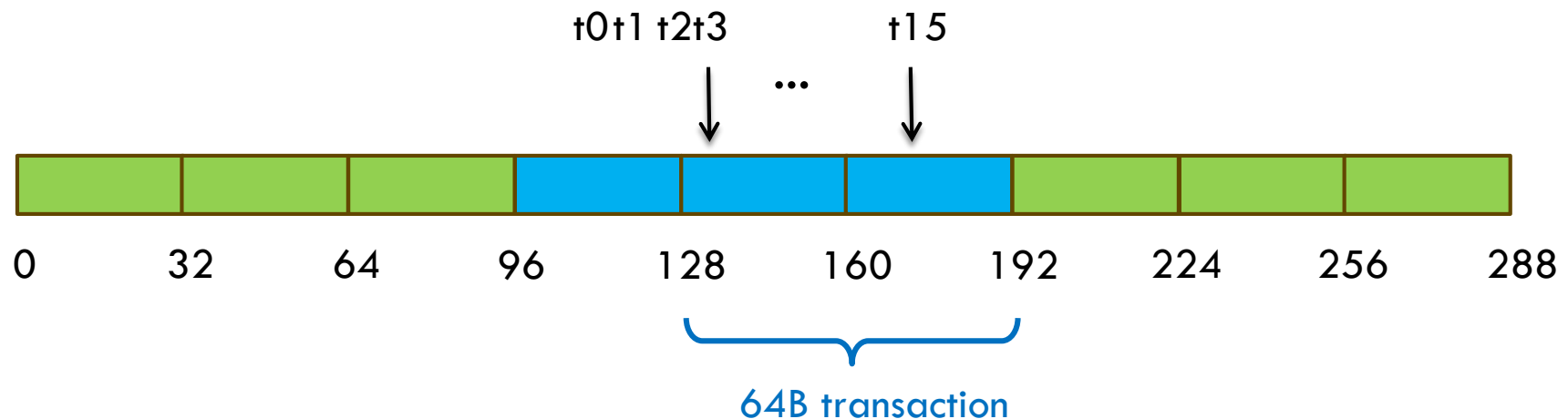
- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255



# Threads 0-15 access 4-byte words at addresses 116-176

111

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 (**reduce to 64B**)



# Consider the stride of your accesses

112

```
__global__ void foo(int* input,
                    float3* input2)
{
    int i = blockDim.x * blockIdx.x
          + threadIdx.x;
    // Stride 1
    int a = input[i];
    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];
    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```



# Example: Array of Structures (AoS)

113

```
struct record
{
    int key;
    int value;
    int flag;
};
```

```
record    *d_records;
cudaMalloc((void**) &d_records, ...);
```

# Example: Structure of Arrays (SoA)

114

```
struct SoA
{
    int * keys;
    int * values;
    int * flags;
};
```

```
SoA d_SoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.values, ...);
cudaMalloc((void**) &d_SoA_data.flags, ...);
```

# Example: SoA vs. AoS

115

```
__global__ void bar(record *AoS_data,  
                    SoA SoA_data)  
{  
    int i = blockDim.x * blockIdx.x  
          + threadIdx.x;  
    // AoS wastes bandwidth  
    int key = AoS_data[i].key;  
    // SoA efficient use of bandwidth  
    int key_better = SoA_data.keys[i];  
}
```

# Memory Coalescing

116

- Structure of array is often better than array of structures
  - ▣ Very clear win on regular, stride 1 access patterns
  - ▣ Unpredictable or irregular access patterns are case-by-case

117

# Shared memory Bank Conflicts

# Shared Memory

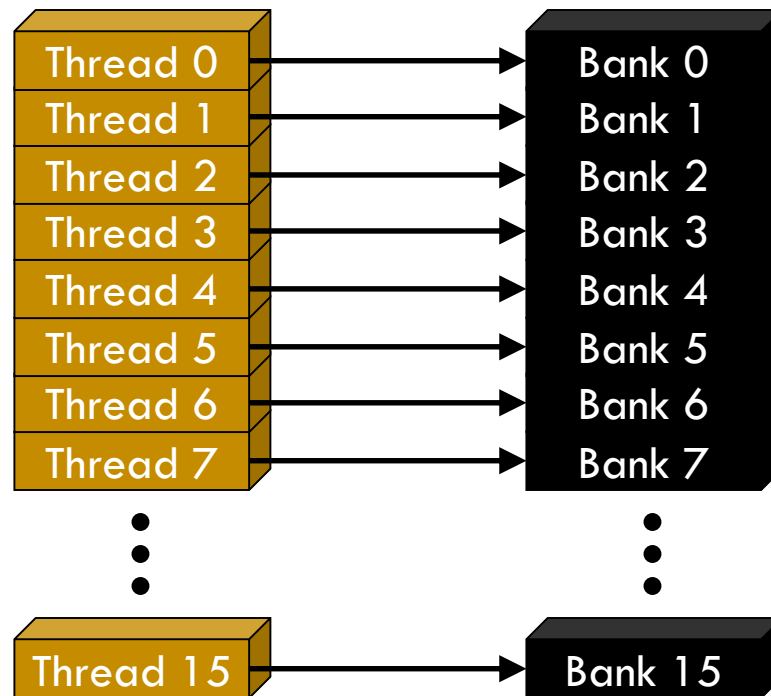
118

- Shared memory is banked
  - ▣ Only matters for threads within a warp
  - ▣ Full performance with some restrictions
  - ▣ Threads can each access different banks
  - ▣ Or can all access the same value
- Consecutive words are in different banks
- If two or more threads access the same bank but different value, get bank conflicts

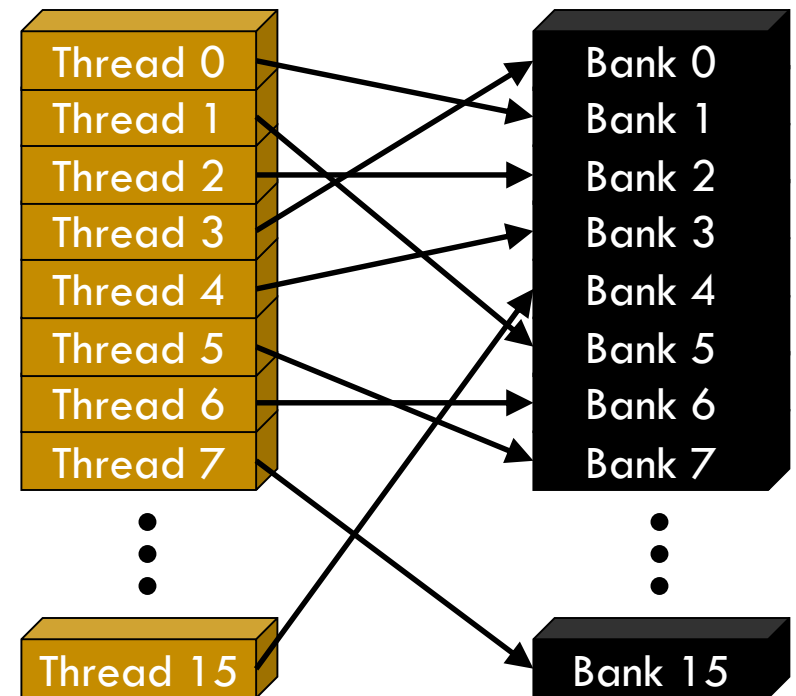
# Bank Addressing Examples

119

□ No Bank Conflicts



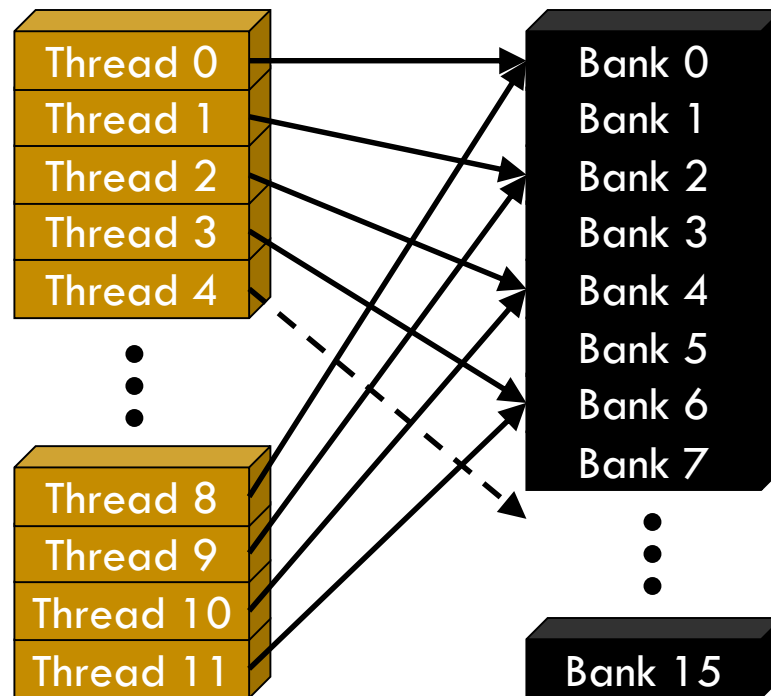
□ No Bank Conflicts



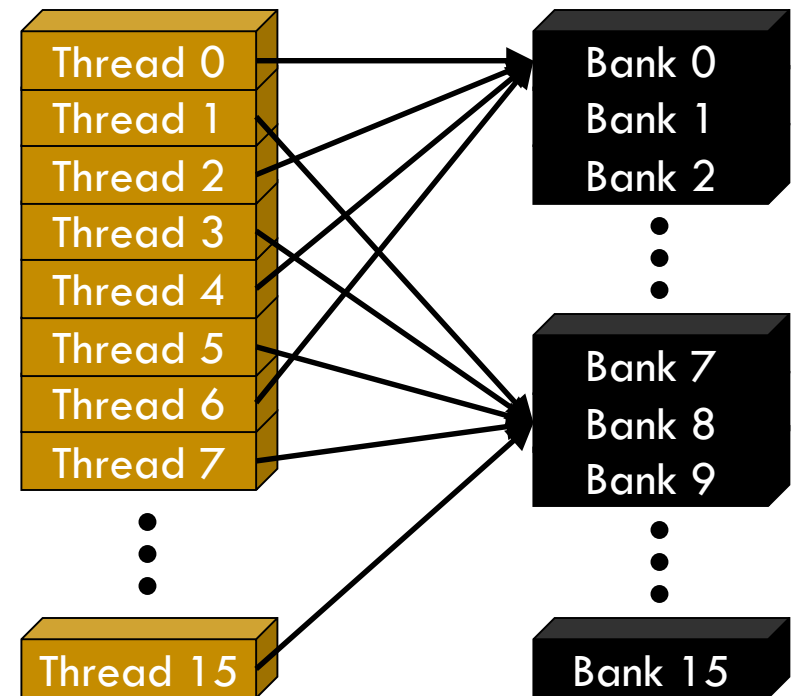
# Bank Addressing Examples

120

## 2-way Bank Conflicts



## 8-way Bank Conflicts





# Trick to Assess Impact On Performance

121

- Change all SMEM reads to the same value
  - ▣ All broadcasts = no conflicts
  - ▣ Will show how much performance could be improved by eliminating bank conflicts
  
- The same doesn't work for SMEM writes
  - ▣ So, replace SMEM array indices with `threadIdx.x`
  - ▣ Can also be done to the reads

# Additional “memories”

122

- `texture` and `__constant__`
- Read-only
- Data resides in global memory
- Different read path:
  - ▣ includes specialized caches

# Constant Memory

123

- Data stored in global memory, read through a constant-cache path
  - ▣ `__constant__` qualifier in declarations
  - ▣ Can only be read by GPU kernels
  - ▣ Limited to **64KB**
- To be used when all threads in a warp read the same address
  - ▣ Serializes otherwise
- Throughput:
  - ▣ 32 bits per warp per clock per multiprocessor

124

# Control Flow divergence

# Control Flow

125

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - ▣ Threads within a single warp take different paths
    - `if-else, ...`
  - ▣ Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

# Control Flow

126

## □ Avoid diverging within a warp

### ▣ Example with divergence:

```
if (threadIdx.x > 2) { ... }  
else { ... }
```

Branch granularity < warp size

### ▣ Example without divergence:

```
if (threadIdx.x / WARP_SIZE > 2)  
{ ... }  
else { ... }
```

Branch granularity is a whole multiple of warp size

# Example: Divergent Iteration

127

```
__global__ void per_thread_sum(int *indices,
                               float *data,
                               float *sums)
{
    ...
    // number of loop iterations is data
    // dependent
    for(int j=indices[i]; j<indices[i+1]; j++)
    {
        sum += data[j];
    }
    sums[i] = sum;
}
```

# Iteration Divergence

128

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
- If data is unpredictable, try to flatten peaks by letting threads work on multiple data items

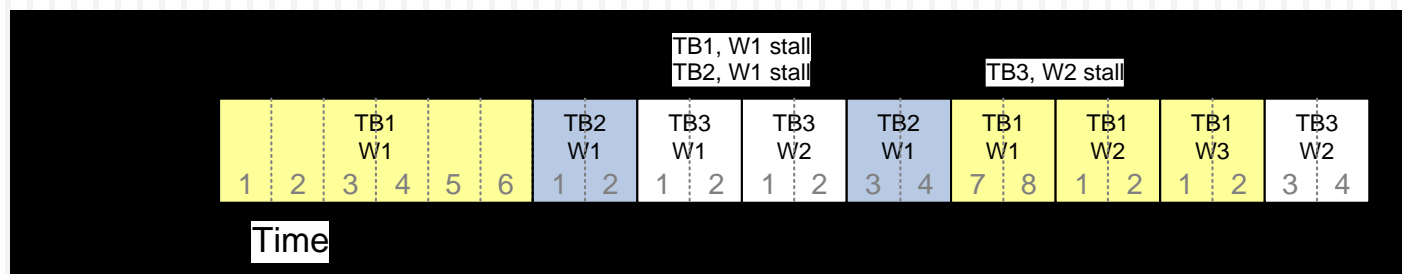


129

# Occupancy

# Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - ▣ At any time, only one of the warps is executed by SM \*
  - ▣ Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - ▣ Eligible Warps are selected for execution on a prioritized scheduling policy
  - ▣ All threads in a warp execute the same instruction when selected



# Thread Scheduling

131

- What happens if all warps are stalled?
  - ▣ No instruction issued → performance lost
  
- Most common reason for stalling?
  - ▣ Waiting on global memory
  
- If your code reads global memory every couple of instructions
  - ▣ You should try to maximize occupancy

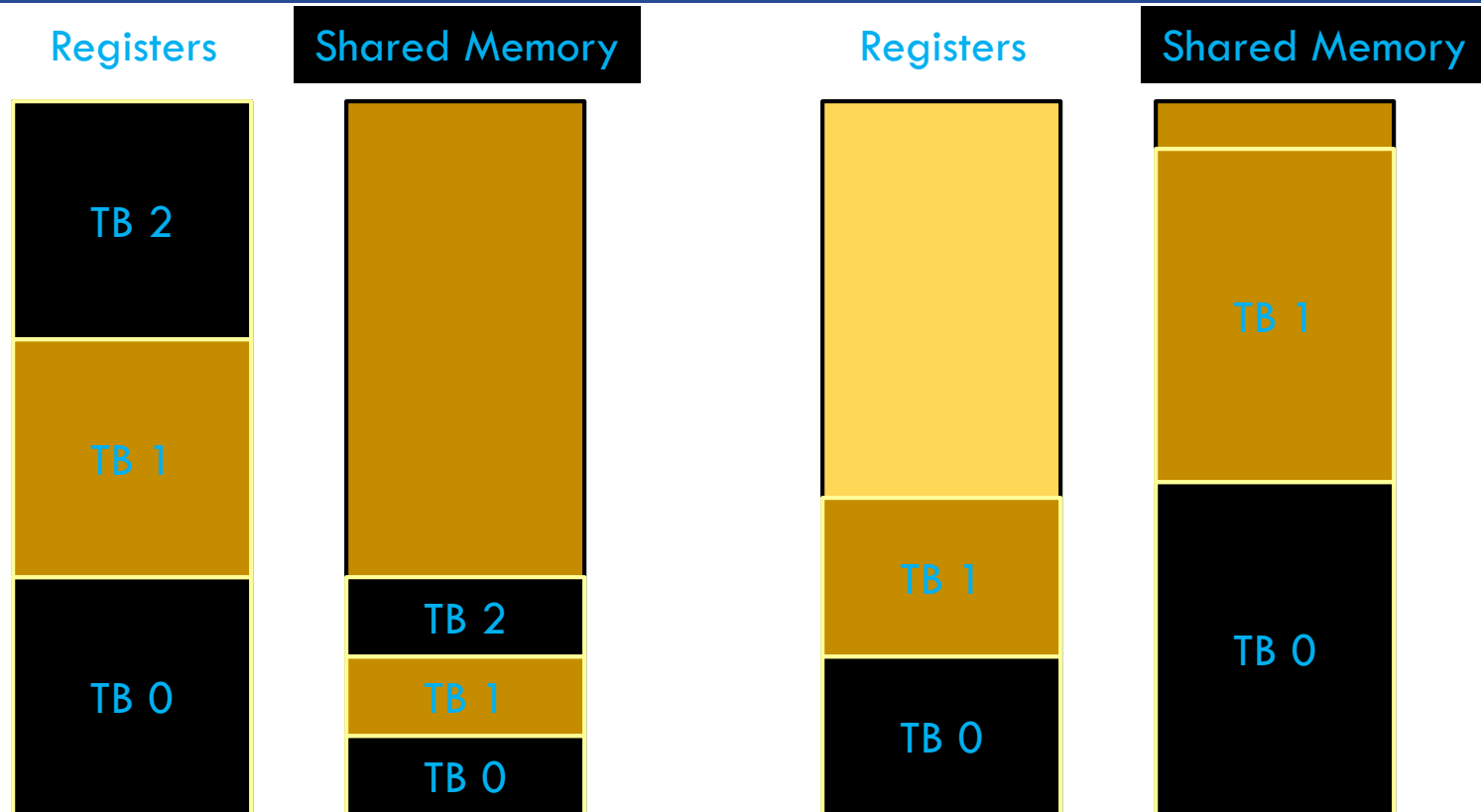
# What determines occupancy?

132

- Register usage per thread & shared memory per thread block

# Resource Limits (1)

133



**Pool of registers and shared memory per SM**

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized -> no more thread blocks

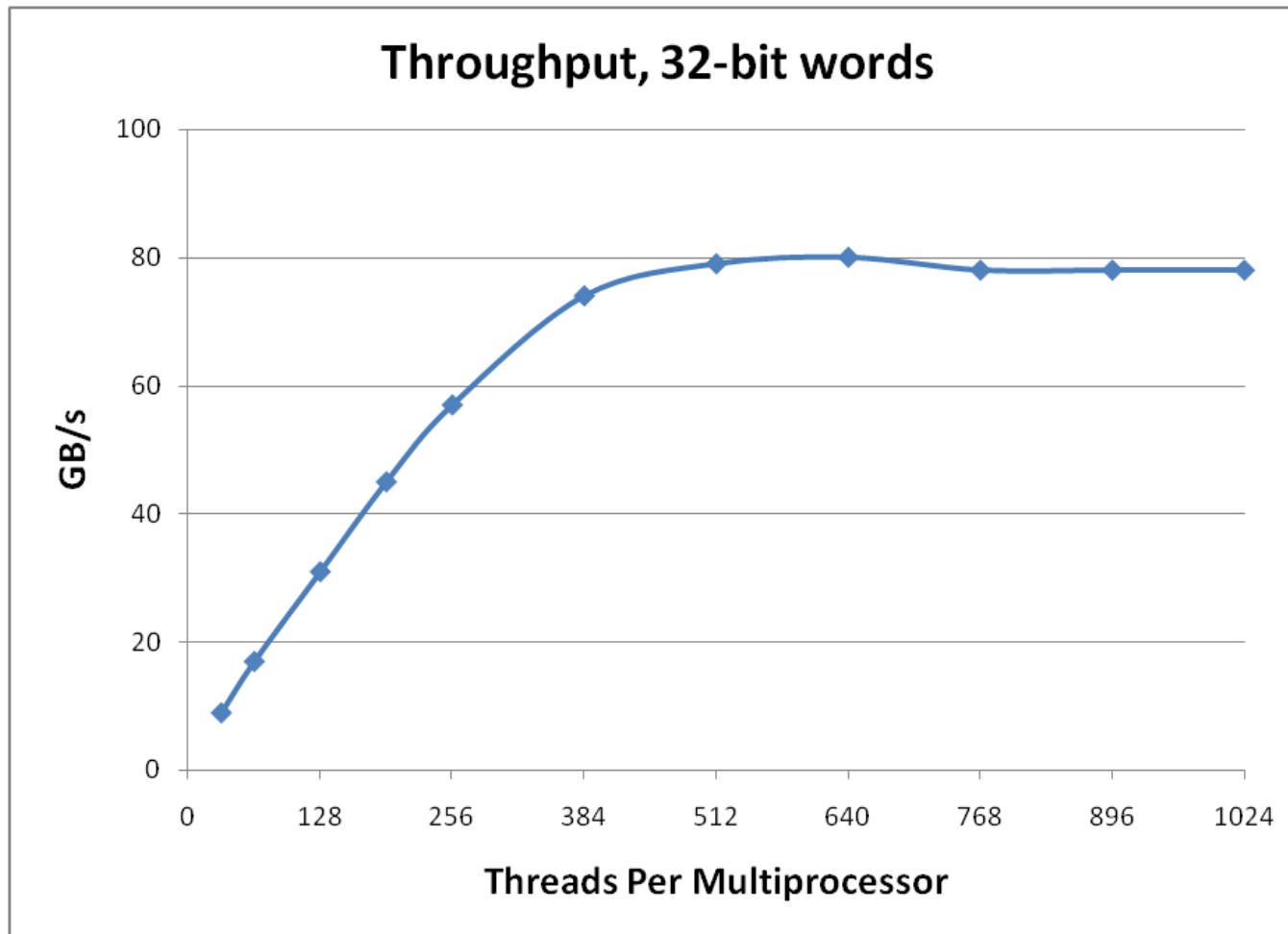
# Resource Limits (2)

134

- **Can only have 8 thread blocks per SM**
  - **If they're too small, can't fill up the SM**
  - **Need 128 threads / TB (gt200), 192 thread/ TB (gf100)**
- **Higher occupancy has diminishing returns for hiding latency**

# Hiding Latency with more threads

135



# How do you know what you're using?

136

- Use `nvcc -Xptxas -v` to get register and shared memory usage
- Plug those numbers into CUDA Occupancy Calculator



# How to influence how many registers you use

137

- Pass option `-maxrregcount=X` to `nvcc`
- This isn't magic, won't get occupancy for free
- Use this very carefully when you are right on the edge

138

# Kernel Launch Overhead

# Kernel Launch Overhead

139

- Kernel launches aren't free
  - ▣ A null kernel launch will take non-trivial time
  - ▣ Actual number changes with HW generations and driver software, so I can't give you one number
- Independent kernel launches are cheaper than dependent kernel launches
  - ▣ Dependent launch: Some readback to the cpu
- If you are launching lots of small grids you will lose substantial performance due to this effect

# Kernel Launch Overheads

140

- If you are reading back data to the cpu for control decisions, consider doing it on the GPU
- Even though the GPU is slow at serial tasks, can do surprising amounts of work before you used up kernel launch overhead

# Performance Considerations

141

- Measure, measure, then measure some more!
- Once you identify bottlenecks, apply judicious tuning
  - ▣ What is most important depends on your program
  - ▣ You'll often have a series of bottlenecks, where each optimization gives a smaller boost than expected

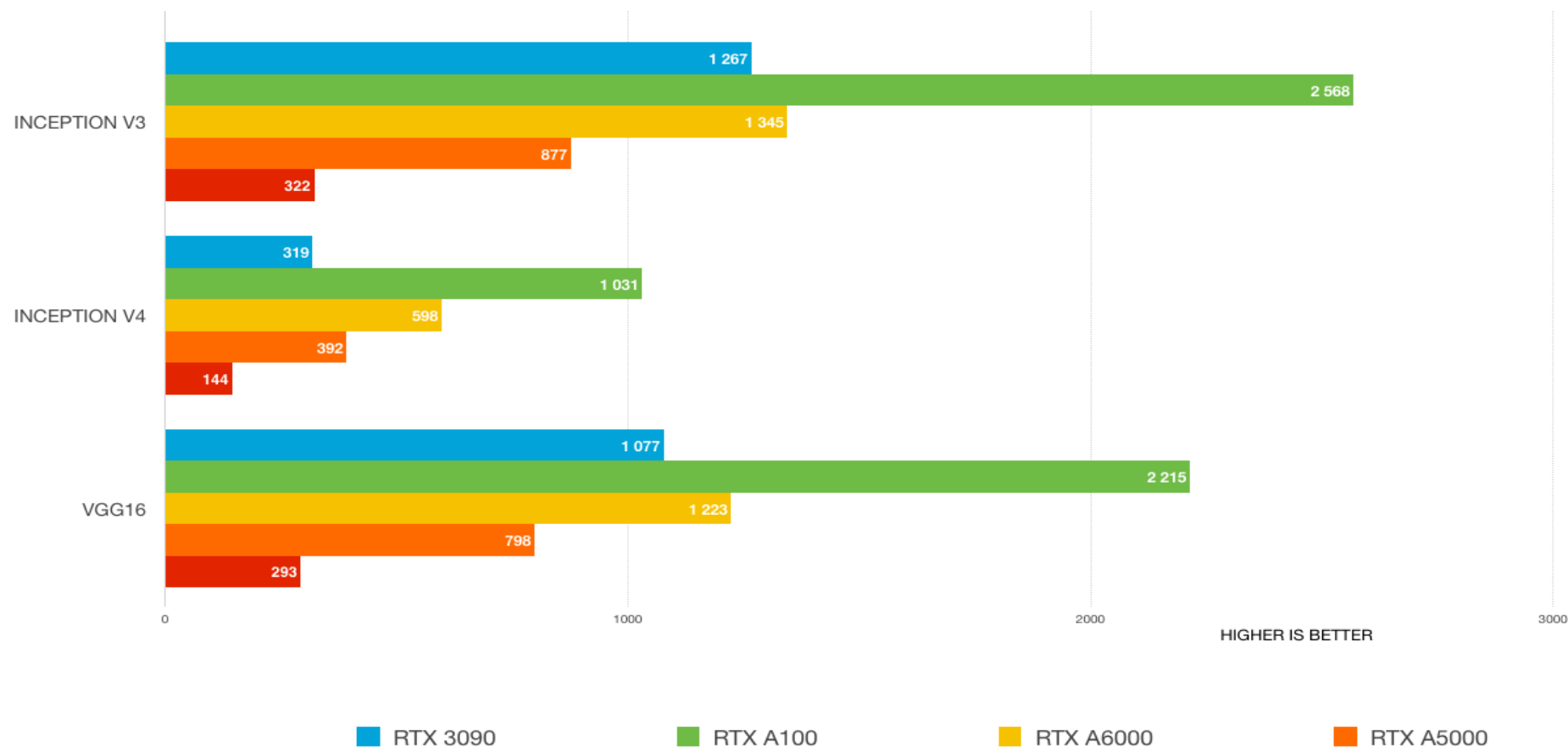
# GPU Performance

142

GPU Model	Memory	Core	Performance (TFLOPS)	Price (\$)
GTX 980 Ti	6 GB	2816	5.6	230
GTX 1080 Ti	11 GB	3584	11	700
P100	16GB	3584	9	1000
RTX 2080 Ti	11 GB	4352	11.7	1000
V100	32GB	5120	14	3900
RTX 3080 TI	12 GB	10240	28	1200
RTX 3090	24 GB	10496	29.2	1500
A100	40 GB	6912	19.5	23000
H100	80 GB	14592	48	36000

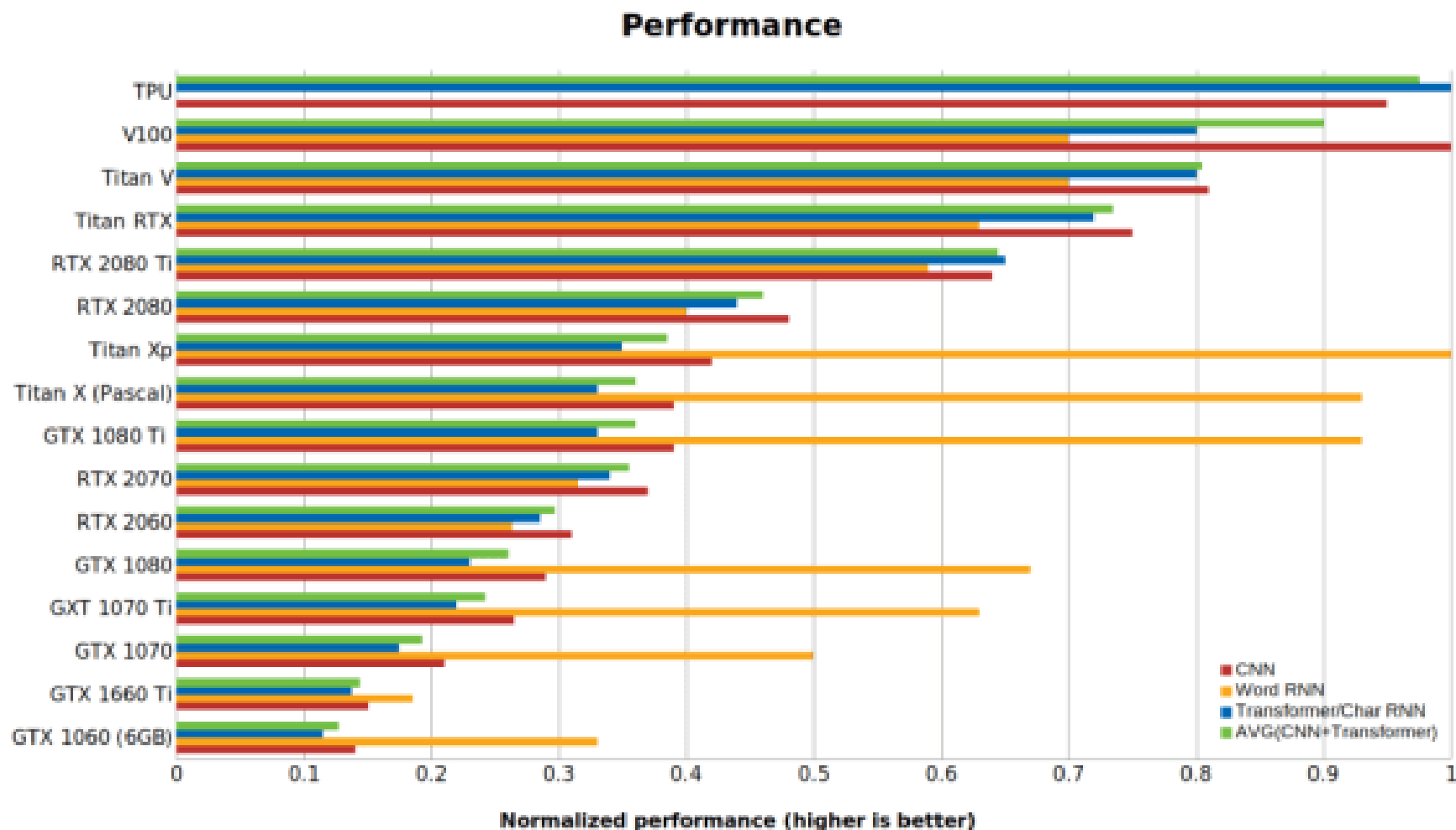
# GPU Performance

143



# GPU Performance

144





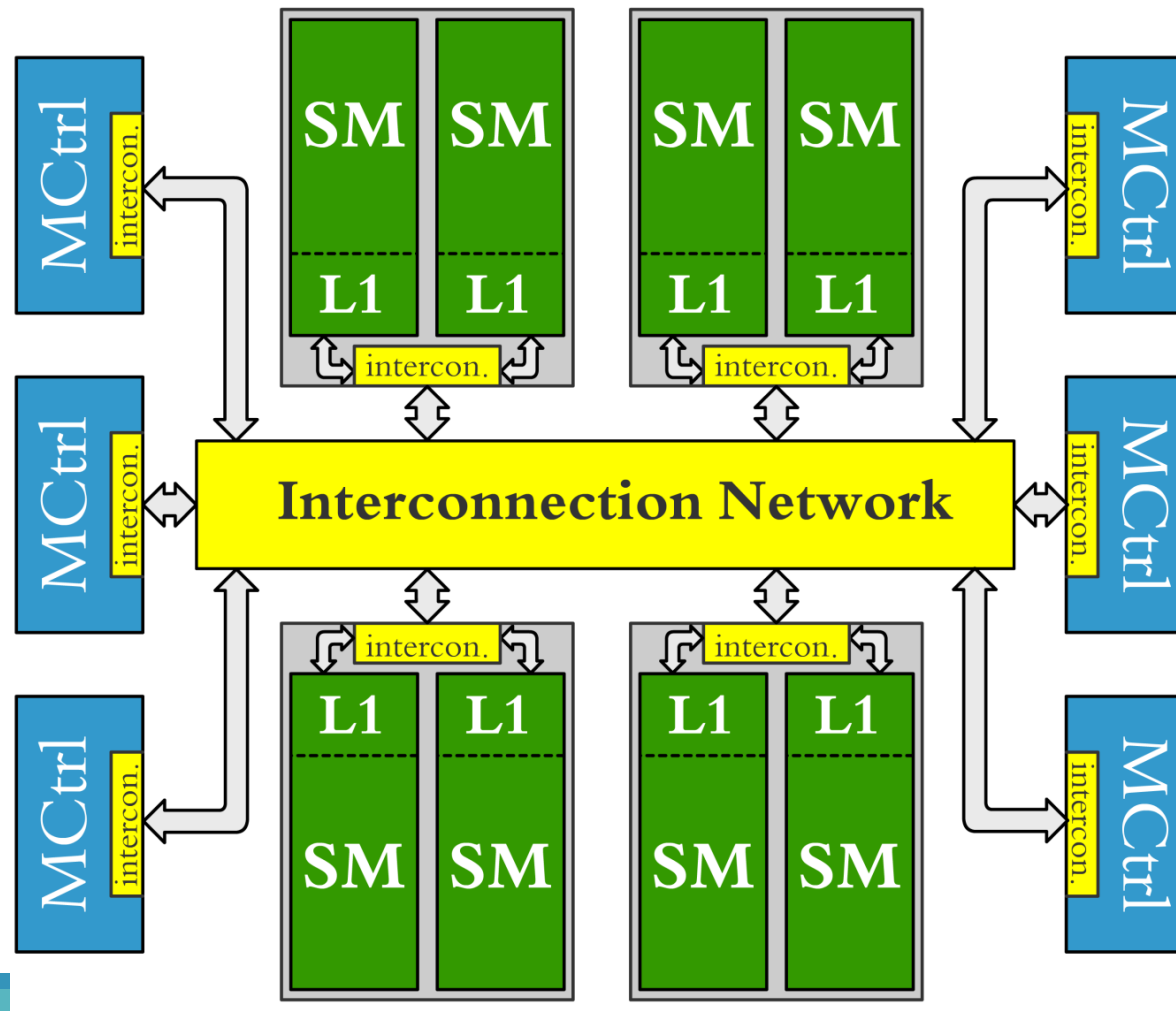


THANK YOU!

□ Questions?

# GPU High-level Design

146



# Bandwidth difference

151

- Bandwidth versus latency
- CPU goal: single thread performance
  - Workloads do not demand for many memory accesses
  - Bring the data as soon as possible
- GPU goal: throughput
  - There are lots of memory accesses, provide the good bandwidth
  - No matter the latency, core will hide it!