



# Multicore Computing

## Lecture21 – GPU



남 범 석

[bnam@skku.edu](mailto:bnam@skku.edu)

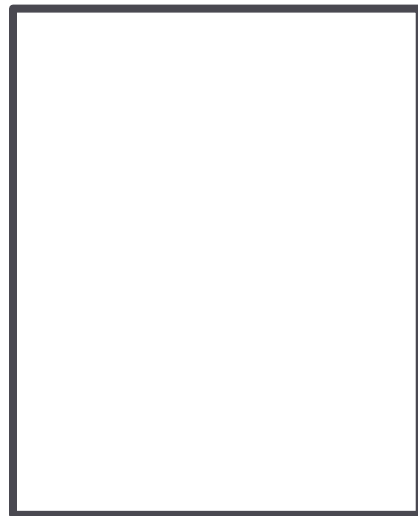


## How a Streaming Multiprocessor (SM) works

- CUDA Threads are grouped in thread blocks
  - All threads of the same thread block are executed in the same SM at the same time
    - SMs have shared memory, then threads within a thread block can communicate
  - The entirety of the threads of a thread block must be executed before there is space to schedule another thread block

## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks



SM

Block 100

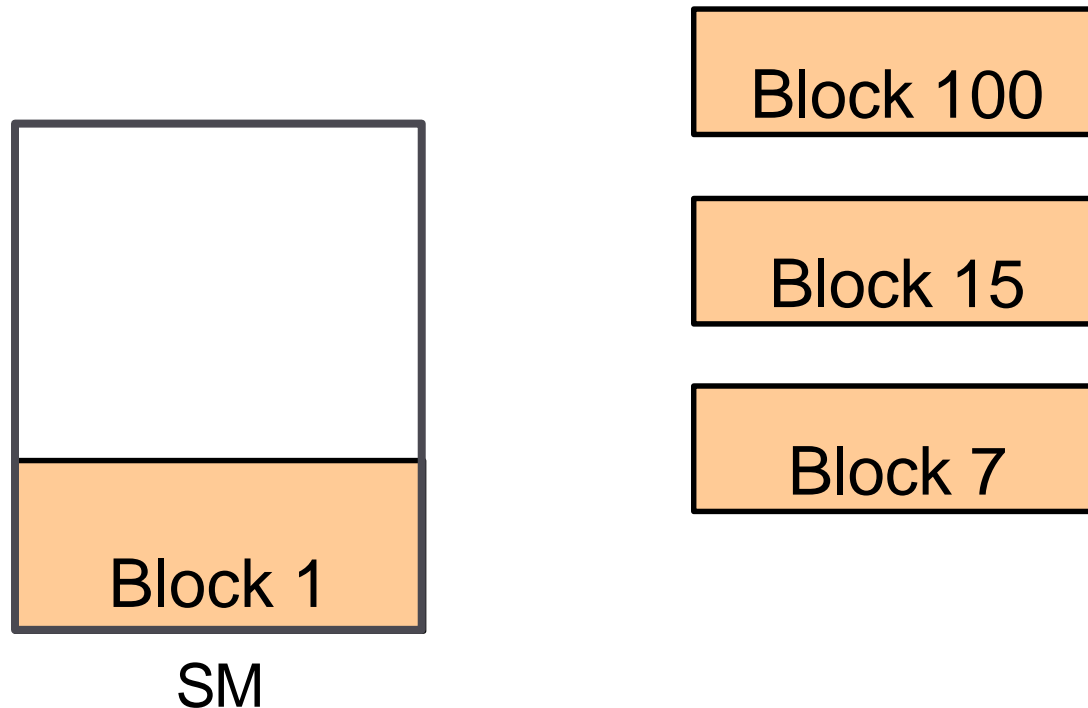
Block 15

Block 7

Block 1

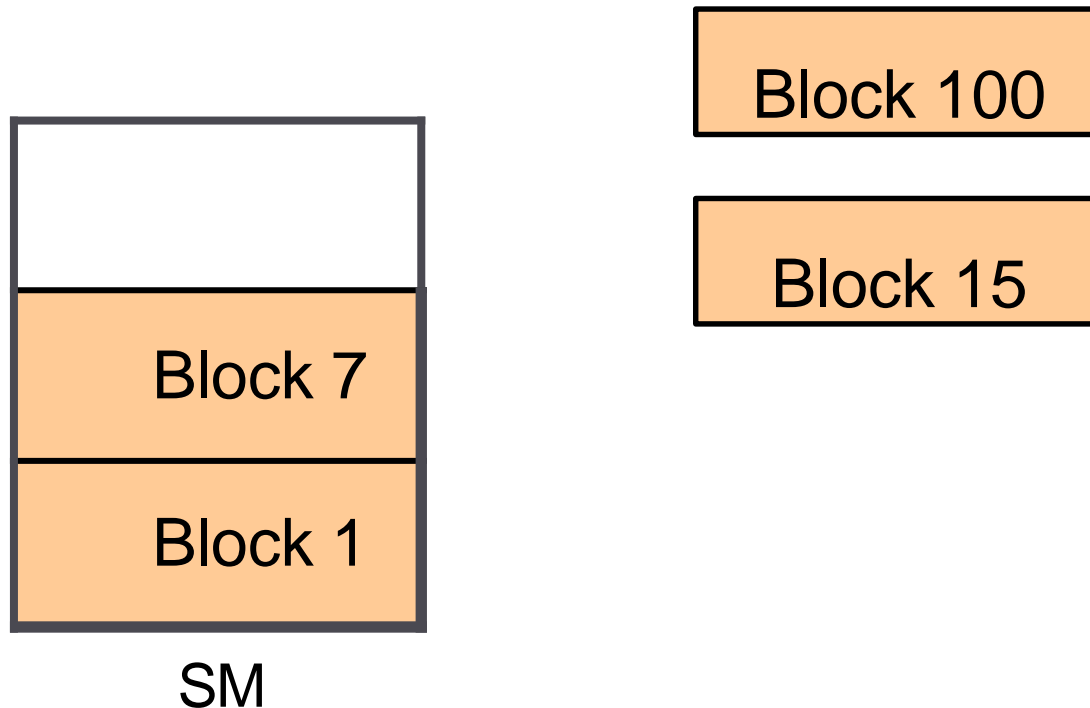
## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks



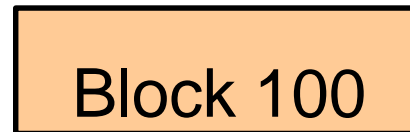
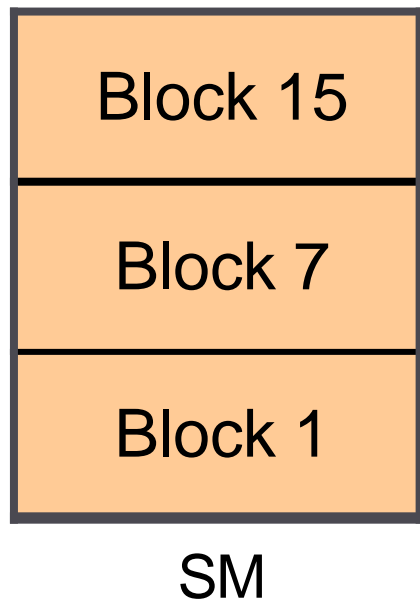
## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks



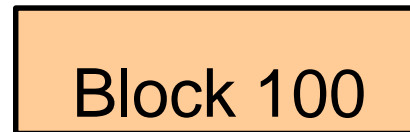
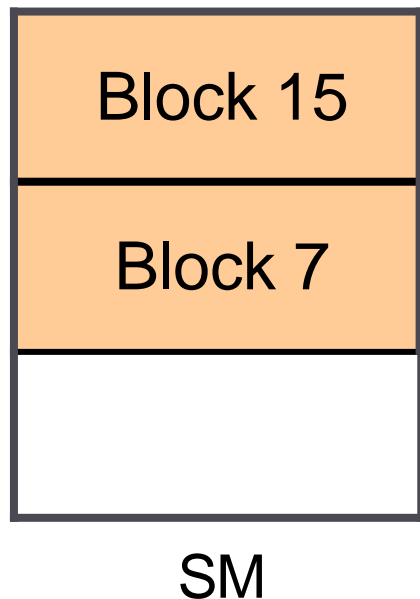
## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks



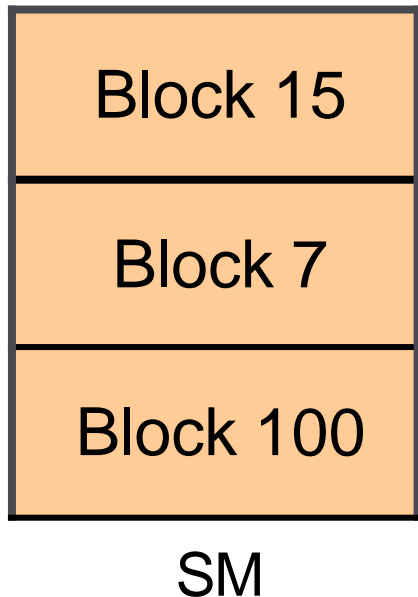
## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks



## How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks





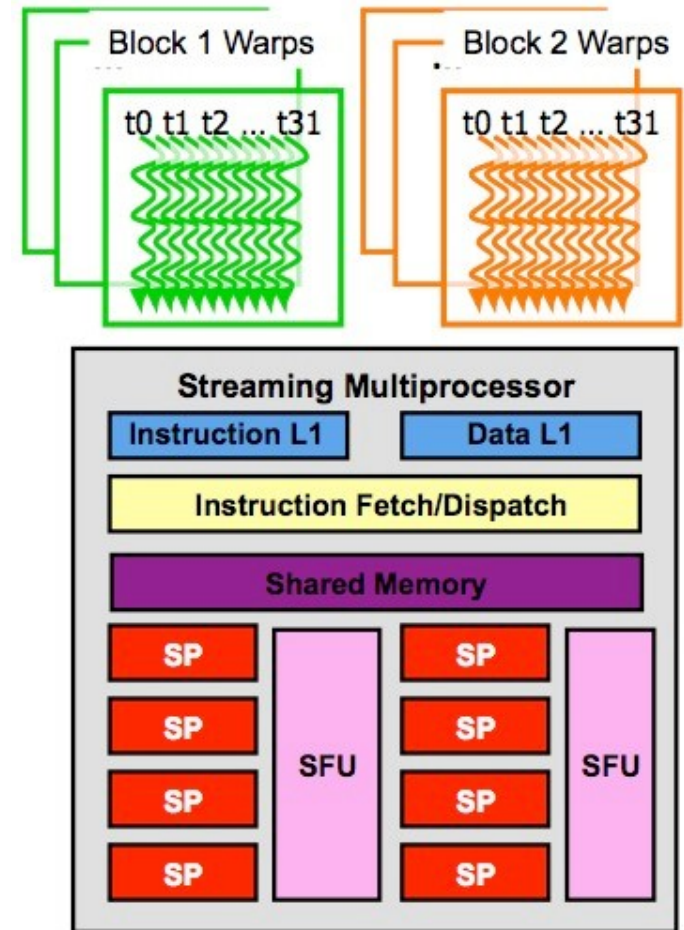
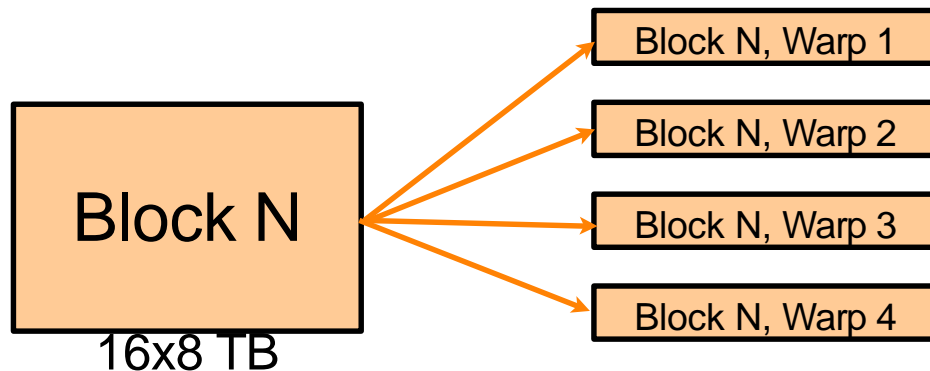


# Warps

- Inside the SM, threads are launched in groups of 32 called warps
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction
- Tesla
  - Warp size: 32
  - Maximum number of threads per SM: 2048
  - Number of SMs: 15 (K40) → 56 (P100) → 80 (V100)
  - Maximum number of threads per block: 1024
  - Max dimension size of a thread block: (1024,1024,64)
  - Max dimension size of a grid size: (2147483647, 65535, 65535)

# Warp designation

- Hardware separates threads of a block into warps
  - All threads in a warp correspond to the same thread block
  - Threads are placed in a warp sequentially
  - Threads are scheduled in warps



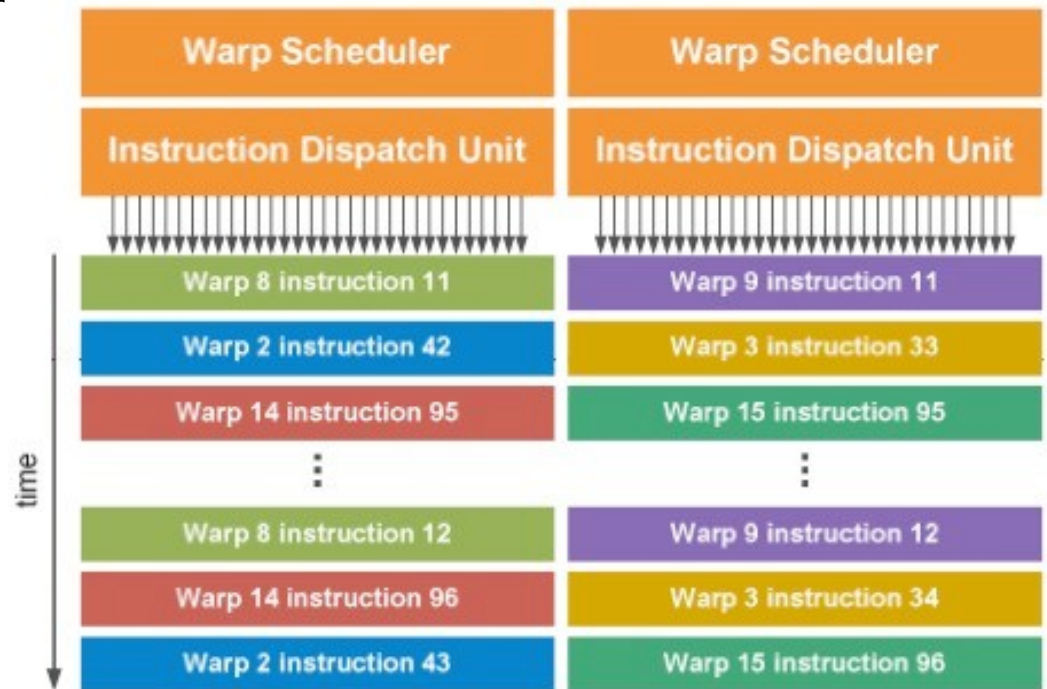


## Warp Scheduling

- The SM implements a zero-overhead warp scheduling
  - Warps whose next instruction has its operands 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

# Warp scheduling

- Fermi
  - Double warp scheduler
  - Each SM has two warp schedulers and two instruction units



# Memory hierarchy

- CUDA works in both the CPU and GPU
  - One has to keep track of which memory is operating on (host - device)
  - Within the GPU there are also different memory spaces

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread **local memory**
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**

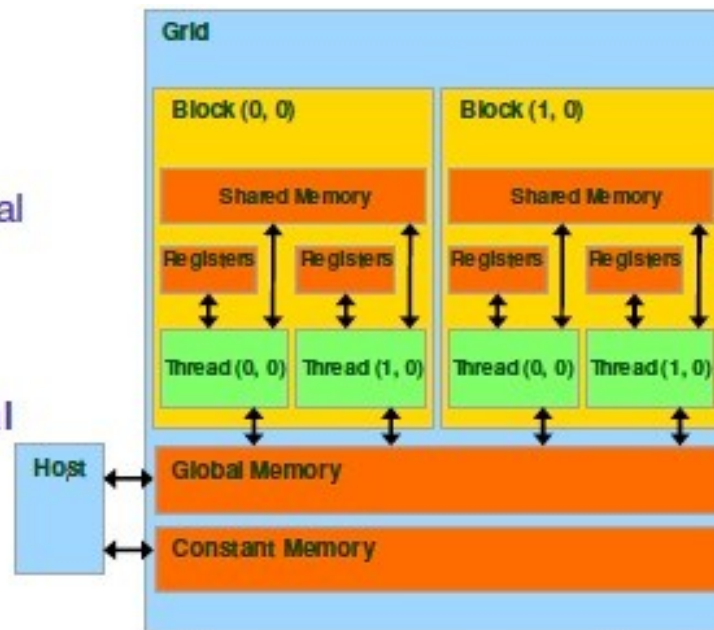


Figure 4.2 GeForce 8800GTX Implementation of CUDA Memories

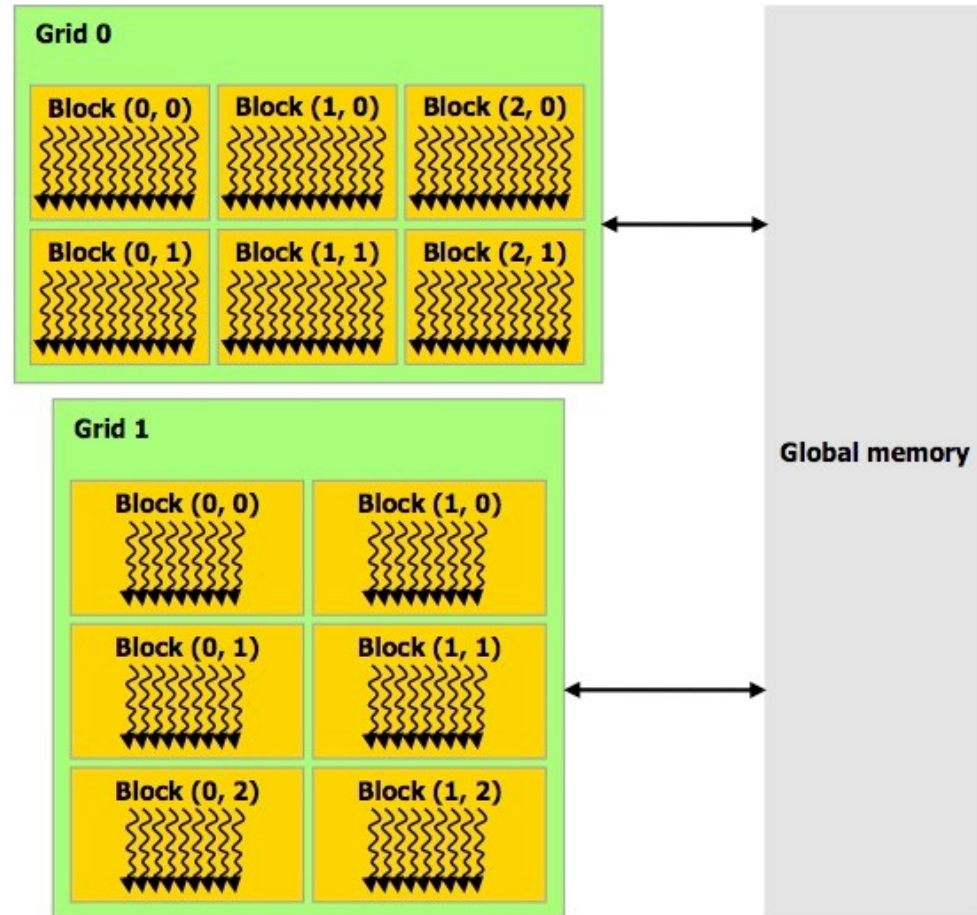


# Memory hierarchy

## ■ Global memory

- Main GPU memory
- Communicates with host
- Can be seen by all threads
- Order of GB
- Off chip, slow (~400 cycles)

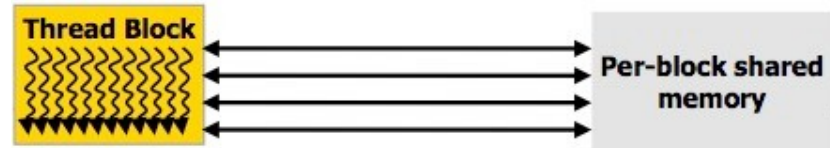
```
__device__ float variable;
```



# Memory hierarchy

## ■ Shared memory

- Per SM
- Seen by threads of the same thread block
- Order of kB
- On chip, fast (~4 cycles)



```
__shared__ float variable;
```

## ■ Registers

- Private to each thread
- On chip, fast

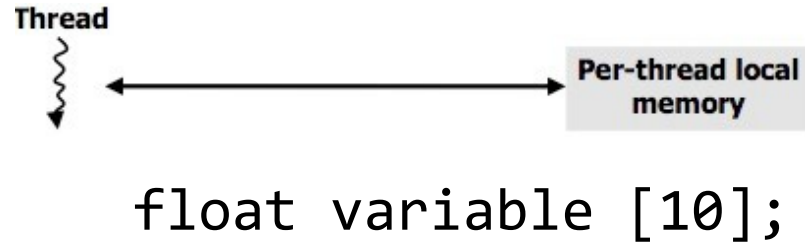


```
float variable;
```

# Memory hierarchy

## ■ Local memory

- Private to each thread
- Off chip, slow
- Register overflows



## ■ Constant memory

- Read only
- Off chip, but fast (cached)
- Seen by all threads
- 64kB with 8kB cache

`__constant__ float variable;`



# Memory hierarchy

## ■ Texture memory

- Seen by all threads
- Read only
- Off chip, but fast (cached) if cache hit
- Cache optimized for 2D locality
- Binds to global

```
texture<type, dim> tex_var;  
cudaChannelFormatDesc();  
cudaBindTexture2D(...);  
tex2D(tex_var, x_index, y_index);
```

Initialize

Options

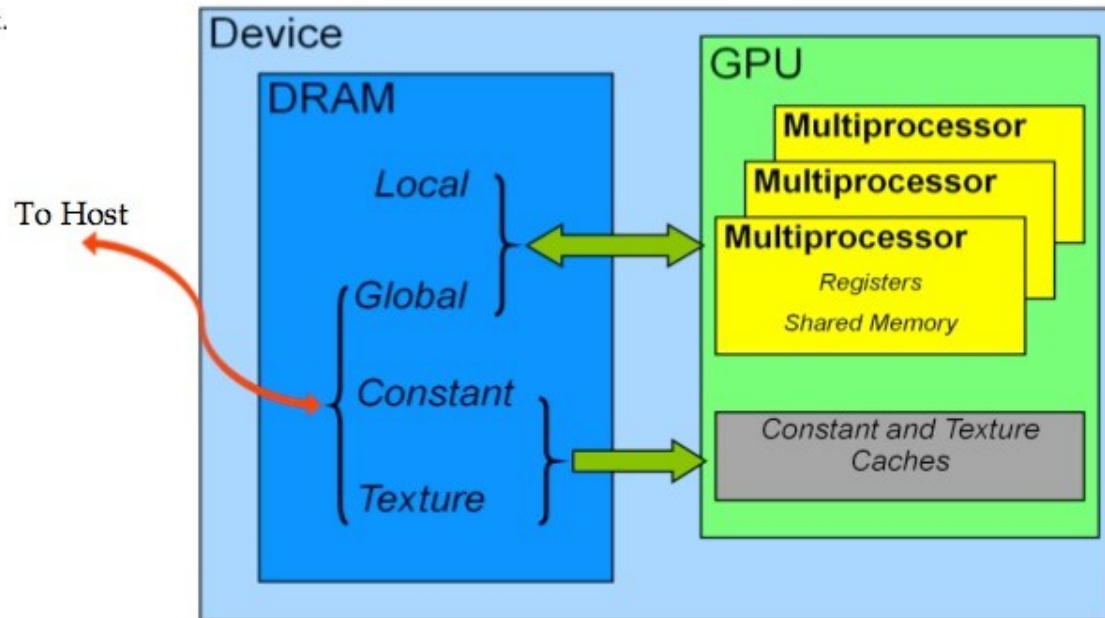
Bind

Fetch

# Memory hierarchy

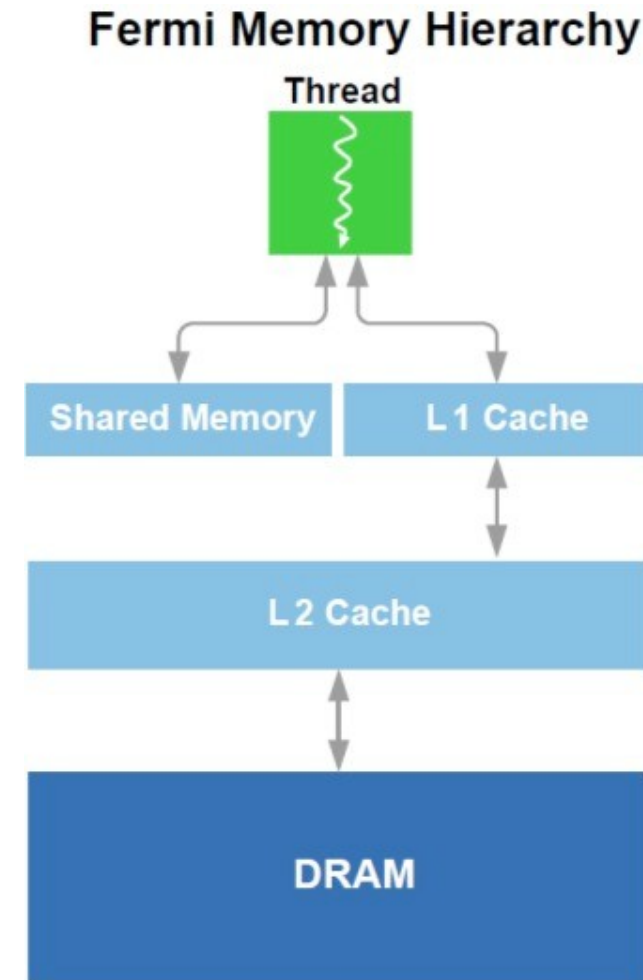
Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation


† Cached only on devices of compute capability 2.x.



# Memory hierarchy

- Case of Fermi
  - Added an L1 cache to each SM
  - Shared + cache = 64kB:
    - Shared = 48kB, cache = 16kB
    - Shared = 16kB, cache = 48kB





## Memory hierarchy

- Use you memory strategically
  - Read only: `__constant__` (fast)
  - Read/write and communicate within a block: `__shared__` (fast and communication)
- Read/write inside thread: registers (fast)
- Data locality: texture



## Resource limits

- Number of thread blocks per SM at the same time is limited by
  - Shared memory usage
  - Registers
  - No more than 8 thread blocks per SM
  - Number of threads

## Resource limits - Examples

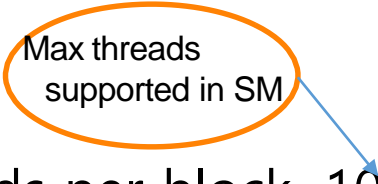
- Number of blocks

- How big should my blocks be? 8x8, 16x16 or 64x64?

- 8x8

- 8\*8 = 64 threads per block,  $1024/64 = 16$  blocks. An SM can have up to 8 blocks: only 512 threads will be active at the same time

Max threads  
supported in SM



- 16x16

- 16\*16 = 256 threads per block,  $1024/256 = 4$  blocks. An SM can take all blocks, then all 1024 threads will be active and achieve full capacity unless other resource overrule

- 64x64

- 64\*64 = 4096 threads per block: doesn't fit in a SM

## Resource limits - Examples

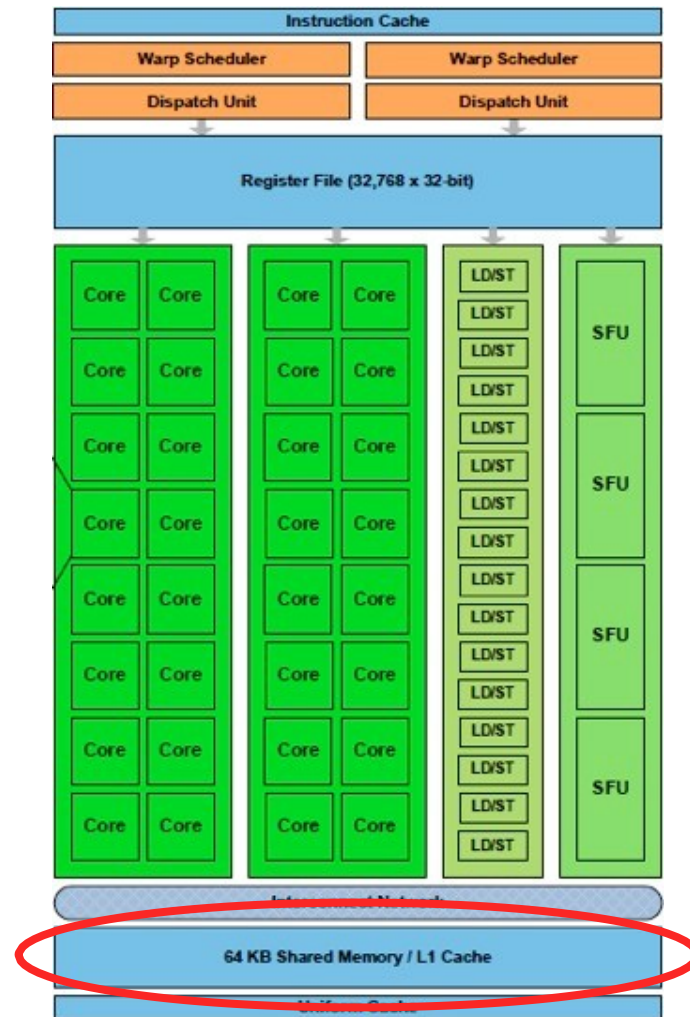
### ■ Registers

- We have a kernel that uses 10 registers. With 16x16 block, how many blocks can run in G80 (max 8192 registers per SM, 768 threads per SM)?
- $10 \times 16 \times 16 = 2560$ . SM can hold  $8192 / 2560 = 3$  blocks, meaning we will use  $3 \times 16 \times 16 = 768$  threads, which is within limits.
- If we add one more register, the number of registers grows to  $11 \times 16 \times 16 = 2816$ . SM can hold  $8192 / 2816 = 2$  blocks, meaning we will use  $2 \times 16 \times 16 = 512$  threads.
- Now as less threads are running per SM is more difficult to have enough warps to have the GPU always busy!



# Shared Memory

- Small (e.g., 48kB per SM)
- Fast (~4 cycles): On-chip
- Private to each block
  - Allows thread communication
- How can we use it?





## Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
    // Each thread will load one element
    int i = threadIdx.x;
    int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
    __shared__ float u_shared[BLOCKSIZE];

    if (I >= N) {return;}
    u_shared[i] = u[I];
    __syncthreads();
    if (I > 0)
    { u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]); }
}
```

Allocate shared array

Load to shared mem

Fetch shared mem



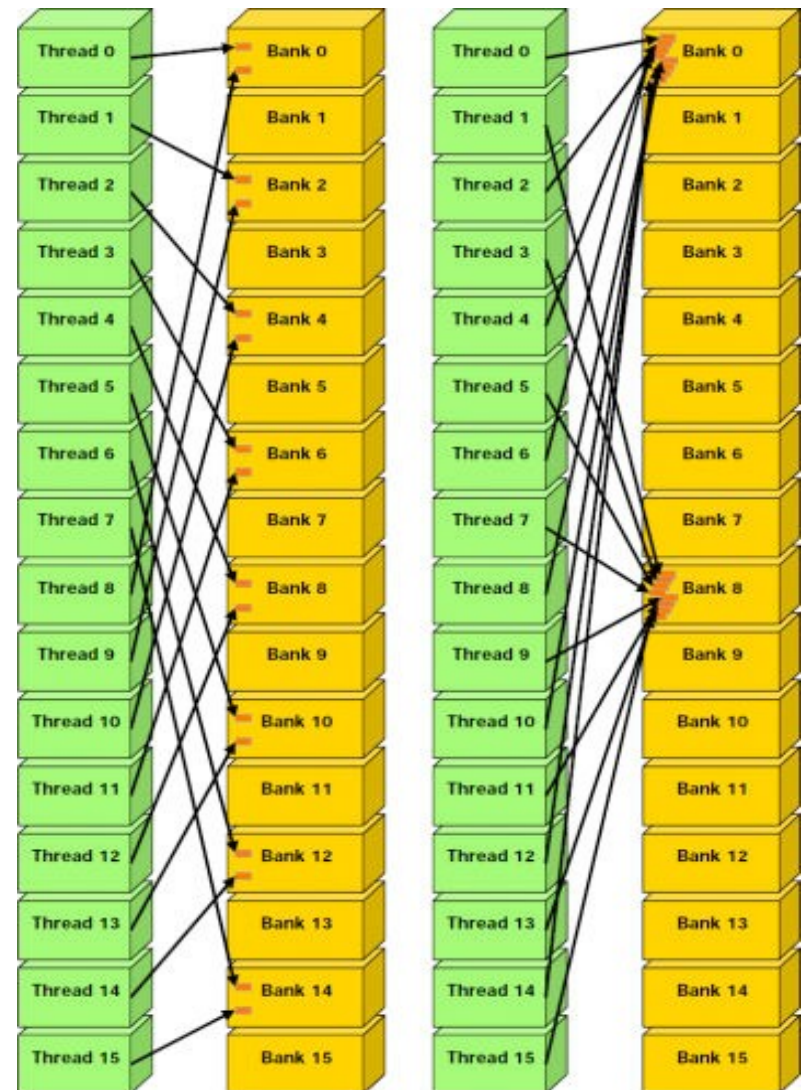
## Shared Memory - Bank conflicts

- Shared memory arrays are subdivided into smaller subarrays called banks
- Different banks can be accessed simultaneously
- If two or more addresses of a memory request are in the same bank, the access is serialized
  - Bank conflicts exist only within a warp

# Bank Conflicts in GPU Shared Memory

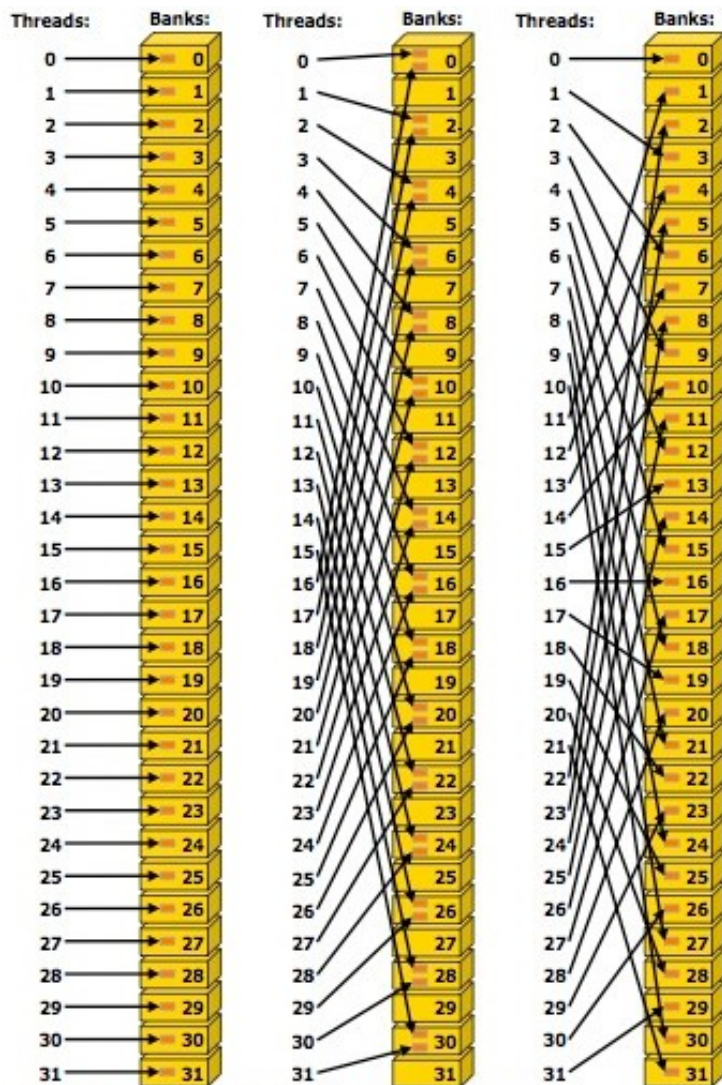
## ■ Bank conflicts

- Arrays in shared memory are divided into banks
- Access to different data in the same bank by more than one thread in the same warp creates a bank conflict
- Bank conflicts serializes the access to the minimum number of non-conflicting instructions

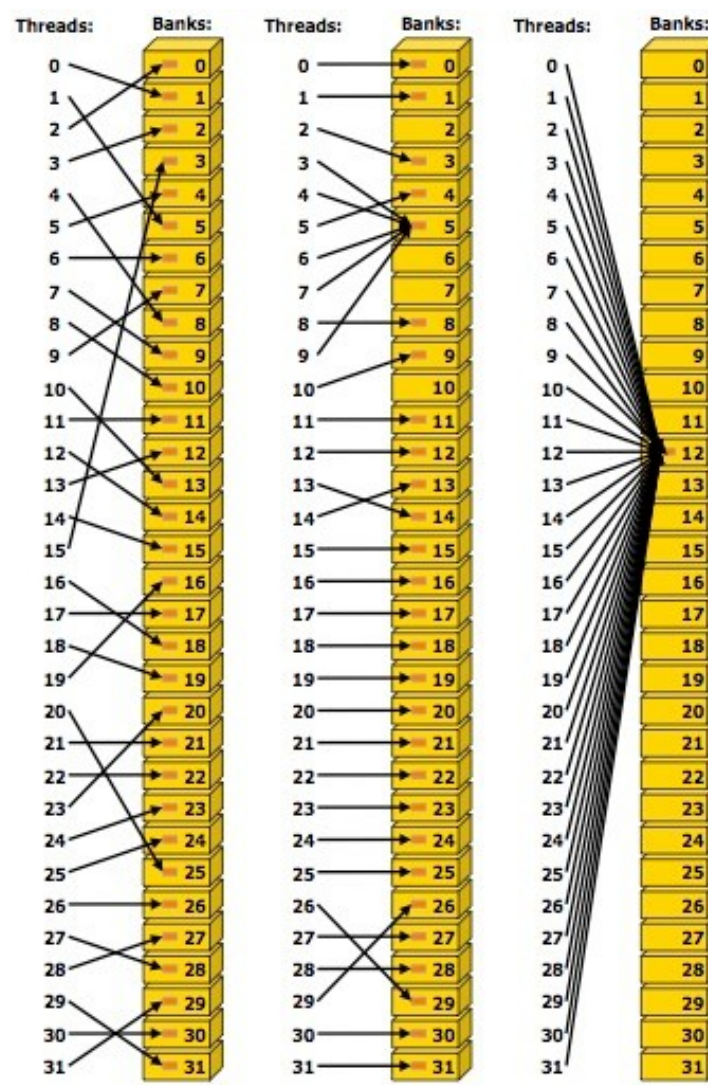




# Shared Memory - Bank conflicts



Left: Linear addressing with a stride of one 32-bit word (no bank conflict).  
 Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).  
 Right: Linear addressing with a stride of three 32-bit words (no bank conflict).



Left: Conflict-free access via random permutation.  
 Middle: Conflict-free access since threads 3, 4, 7, and 9 access the same word within bank 5.  
 Right: Conflict-free broadcast access (all threads access the same word).

## Shared Memory

### ▪ `__syncthreads()`

- Barrier that waits for all threads of the block before continuing
- Need to make sure all data is loaded to shared before access
- Avoids race conditions
- Serializes the code: don't over use!

```
u_shared[i] = u[I];
```

```
__syncthreads();
```

```
if (i>0 && i<BLOCKSIZE-1)
```

```
    u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);
```



## Race condition

- When two or more threads want to access and operate on a memory location without synchronization
- Example: we have the value 3 stored in global memory and two threads want to add one to that value.
  - Possibility 1:
    - Thread 1 reads the value 3 adds 1 and writes 4 back to memory
    - Thread 2 reads the value 4 and writes 5 back to memory
  - Possibility 2:
    - Thread 1 reads the value 3
    - Thread 2 reads the value 3
    - Both threads operate on 3 and write back the value 4 to memory
- Solutions:
  - `__syncthreads()` or atomic operations



## Atomic operations

- Atomic operations deal with race conditions
  - It guarantees that while the operation is being executed, that location in memory is not accessed
  - Still we can't rely on any ordering of thread execution!
  - Types
    - `atomicAdd`
    - `atomicSub`
    - `atomicExch`
    - `atomicMin`
    - `atomicMax` etc...

# Atomic operations

```
__global__ update (int *values, int *who)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    int I = who[i];
    atomicAdd(&values[I], 1);
}
```





## Recap: Warp

- GPUs have many Streaming Multiprocessors (SMs)
- Each block is assigned to an SM
- Each SM has multiple processors but only one instruction unit
- Inside the SM, the block is divided into Warps of threads
  - Warps consist of 32 threads
  - All 32 threads MUST run the exact same set of instructions at the same time due to the fact that there is only one instruction unit
  - Warps are run concurrently in an SM

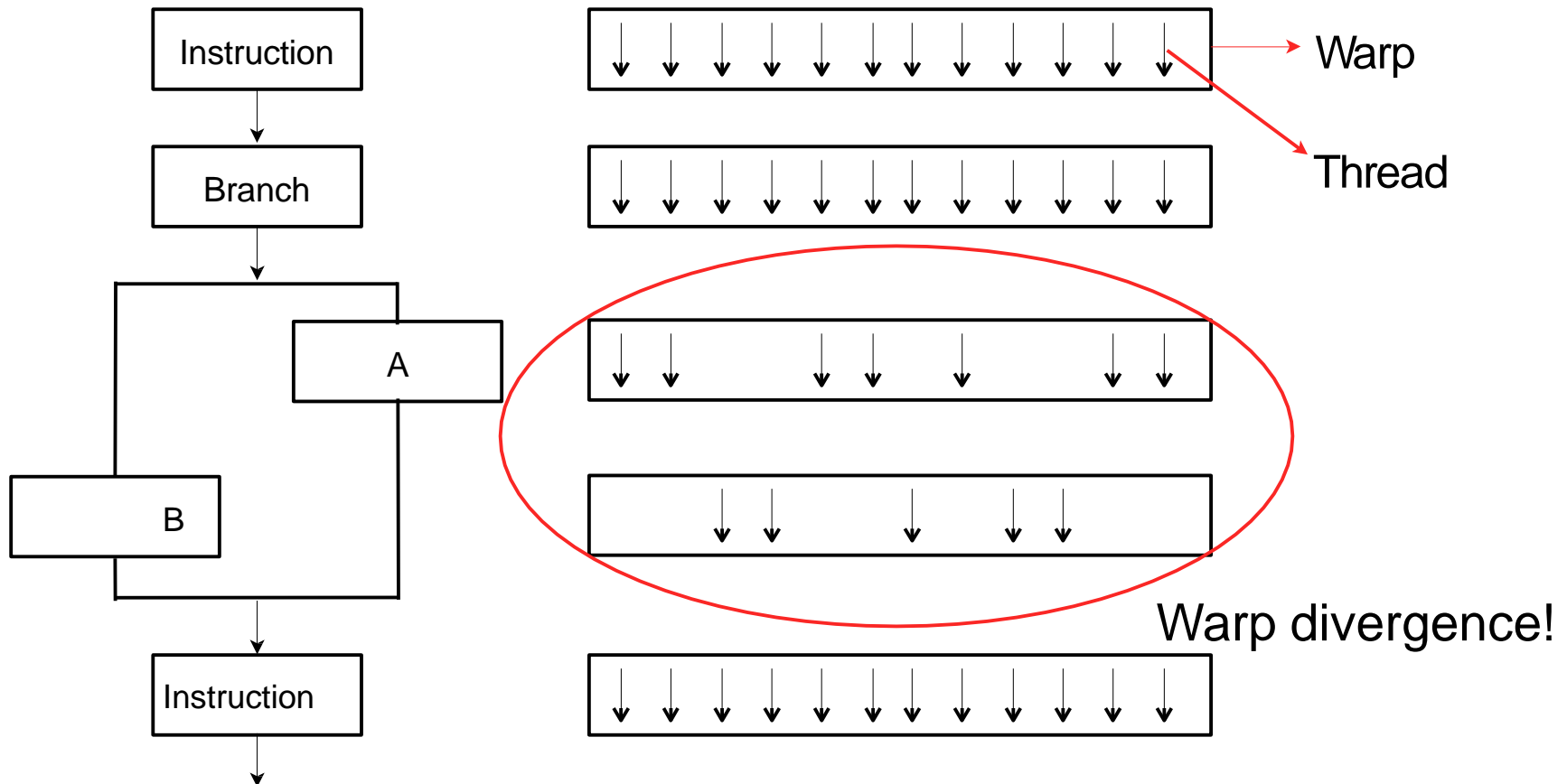
# Warp Divergence

- If statement
  - Threads are executed in warps
  - Within a warp, the hardware is not capable of executing if and else statements at the same time!

```
__global__ void function();  
{  
    ...  
    if (condition) {  
        ...  
    }  
    else {  
        ...  
    }  
}
```

# Warp Divergence

- How does the hardware deal with an if statement?





# Warp Divergence

## ■ Recommendations

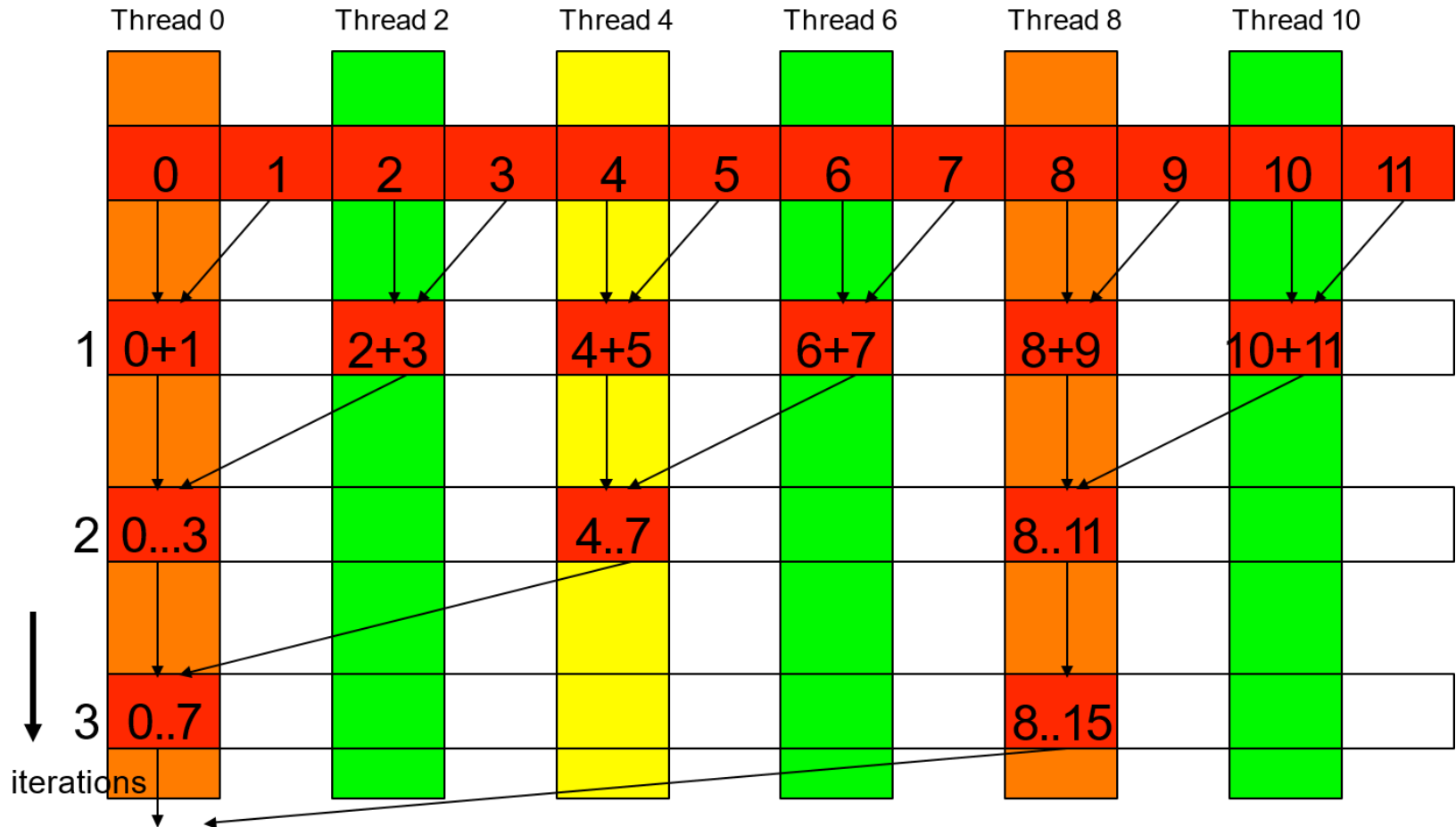
- Try to make every thread in the same warp do the same thing
  - If the if statement cuts at a multiple of the warp size, there is no warp divergence and the instruction can be done in one pass
- Remember threads are placed consecutively in a warp (t0-t31, t32- t63, ...)
  - But we cannot rely on any execution order within warps
- If you can't avoid branching, try to make as many consecutive threads as possible do the same thing

## Warp Divergence: Pair-wise Sum

- Let's implement pair-wise sum
  - To implement in parallel, let's take every other value and add in place it to its neighbor
  - To the resulting array do the same thing until we have only one value

```
__shared__ float partialSum[]  
int t = threadIdx.x  
  
for(int stride = 1; stride < blockDim.x; stride *= 2)  
{  
    __syncthreads();  
    if (t % (2 * stride) == 0)  
        partialSum[t] += partialSum[t + stride];  
}
```

# Warp Divergence: Pair-wise Sum: Illustration



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009  
ECE498AL, University of Illinois, Urbana-Champaign



## Warp Divergence: Pair-wise Sum

- Disadvantages
  - The number of threads decreases per iteration, but we're using much more warps than needed
    - There is warp divergence in every iteration
  - No more than half the threads per thread are being executed per iteration
- Let's change the algorithm a little bit...

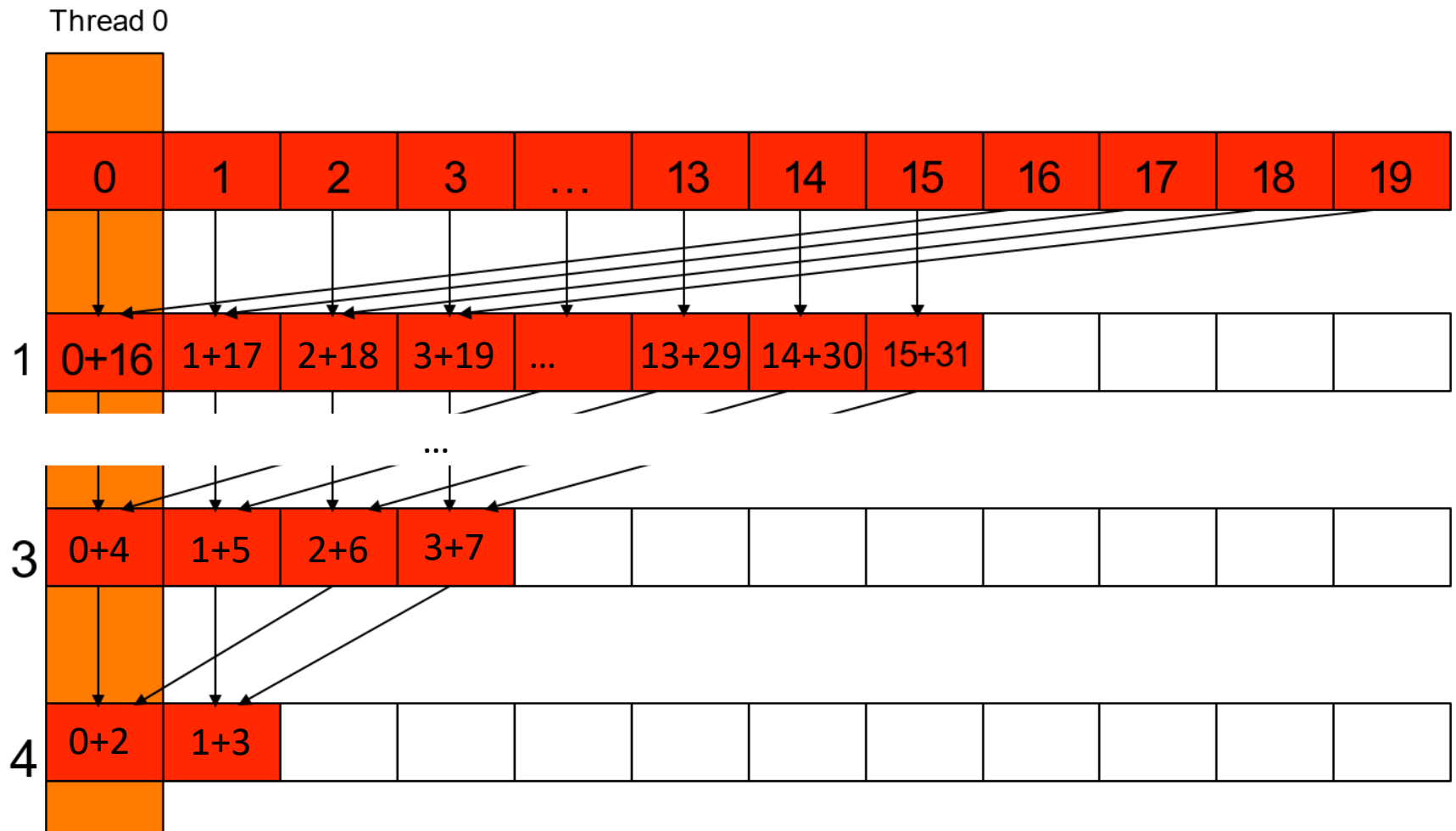
## Improved Pair-wise Sum

- Improved version
  - Instead of adding neighbors, let's add values with stride half a section away
  - Divide the stride by two after each iteration

```
__shared__ float partialSum[];  
int t = threadIdx.x;  
for(int stride = blockDim.x; stride>1; stride>>=1)  
{  
    __syncthreads();  
    if (t<stride)  
        partialSum[t] += partialSum[t+stride];  
}
```



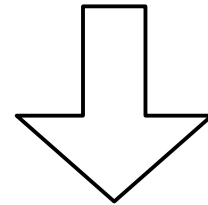
# Improved Pair-wise Sum : Illustration



## Improved Pair-wise Sum

Iteration	Exec. threads	Warps
1	256	16
2	128	8
3	64	4
4	32	2
5	16	1
6	8	1
7	4	1
8	2	1

Threads > Warps



Warp divergence!



## Improved Pair-wise Sum

- We get warp divergence only for the last 5 iterations
- Warps will be shut down as the iteration progresses
  - This will happen much faster than for the previous case
  - Resources are better utilized
  - For the last 5 iterations, only 1 warp is still active



## Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
    - [https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA\\_C\\_Programming\\_Guide.pdf](https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf)
  - CUDA Zone – tools, training, webinars and more  
<https://developer.nvidia.com/cuda-zone>