

Lecture 10: CUDA Optimization

Concurrent and Parallel Programming (INF-3201)
Autumn 2014

John Markus Bjørndalen (with foils from Lars Tiede)

Outline

- Overview
- CUDA Hardware
- Memory optimizations
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

Optimization overview

- Optimize algorithms for GPU
 - GPUs can support 30 000 concurrently active threads
⇒ maximize independent parallelism
- Optimize memory transfer between CPU and GPU
 - The transfer (through PCIe bus) is costly
⇒ maximize computation per data transferred
 - Ex.: Matrix multiplication: N^3 operations, $3N^2$ elements transferred
⇒ operations/elements = $O(N)$
 - ⇒ Keep data on GPU as long as possible
 - Do more computation (even low parallelism computations) on GPU
- Optimize memory access on GPU
 - Use shared memory (on-chip, fast)
 - Use coalesced memory access to global memory (off-chip DRAM)
- Keep GPU multiprocessors busy
 - Minimize resource usage/thread ⇒ maximize #active blocks

Outline

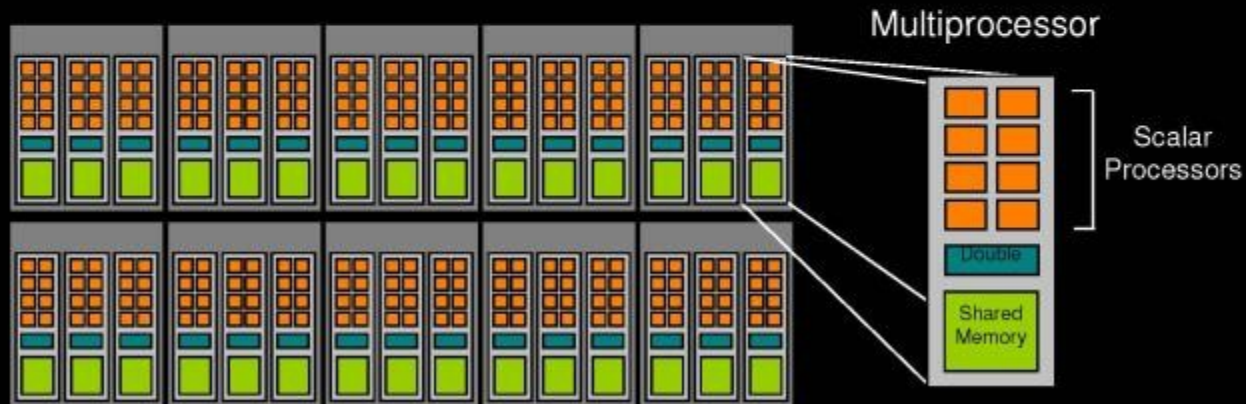
- Overview
- **CUDA Hardware**
- Memory optimizations
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

CUDA Architecture

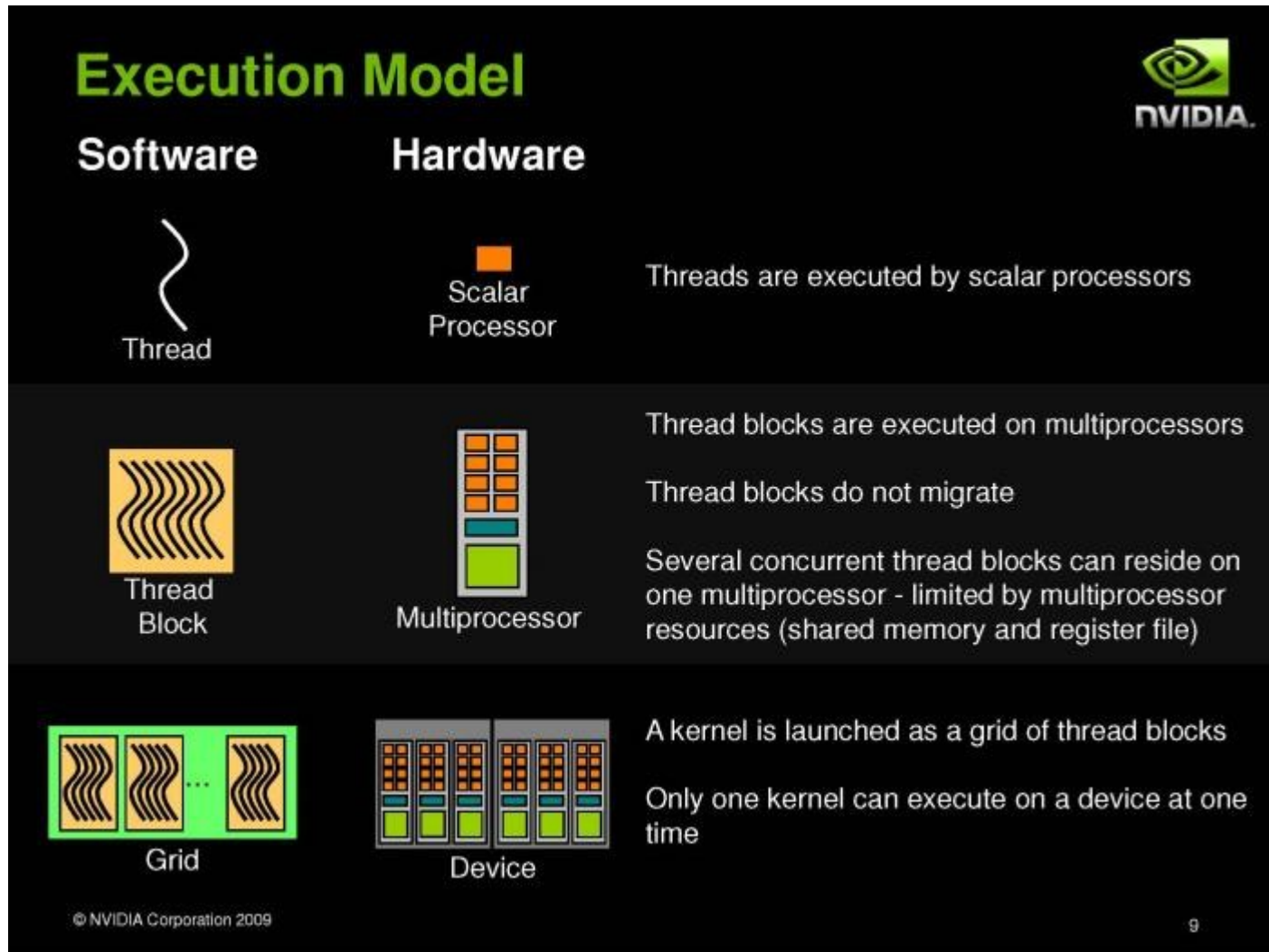
10-Series Architecture



- 240 **Scalar Processor (SP) cores** execute kernel threads
- 30 Streaming Multiprocessors (SMs) each contain
 - 8 scalar processors
 - 2 Special Function Units (SFUs)
 - 1 double precision unit
 - **Shared memory** enables thread cooperation



Execution model



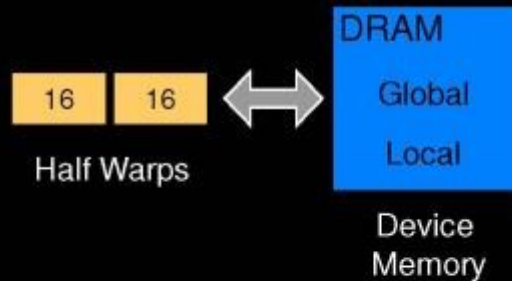
Warps

Warps and Half Warps



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

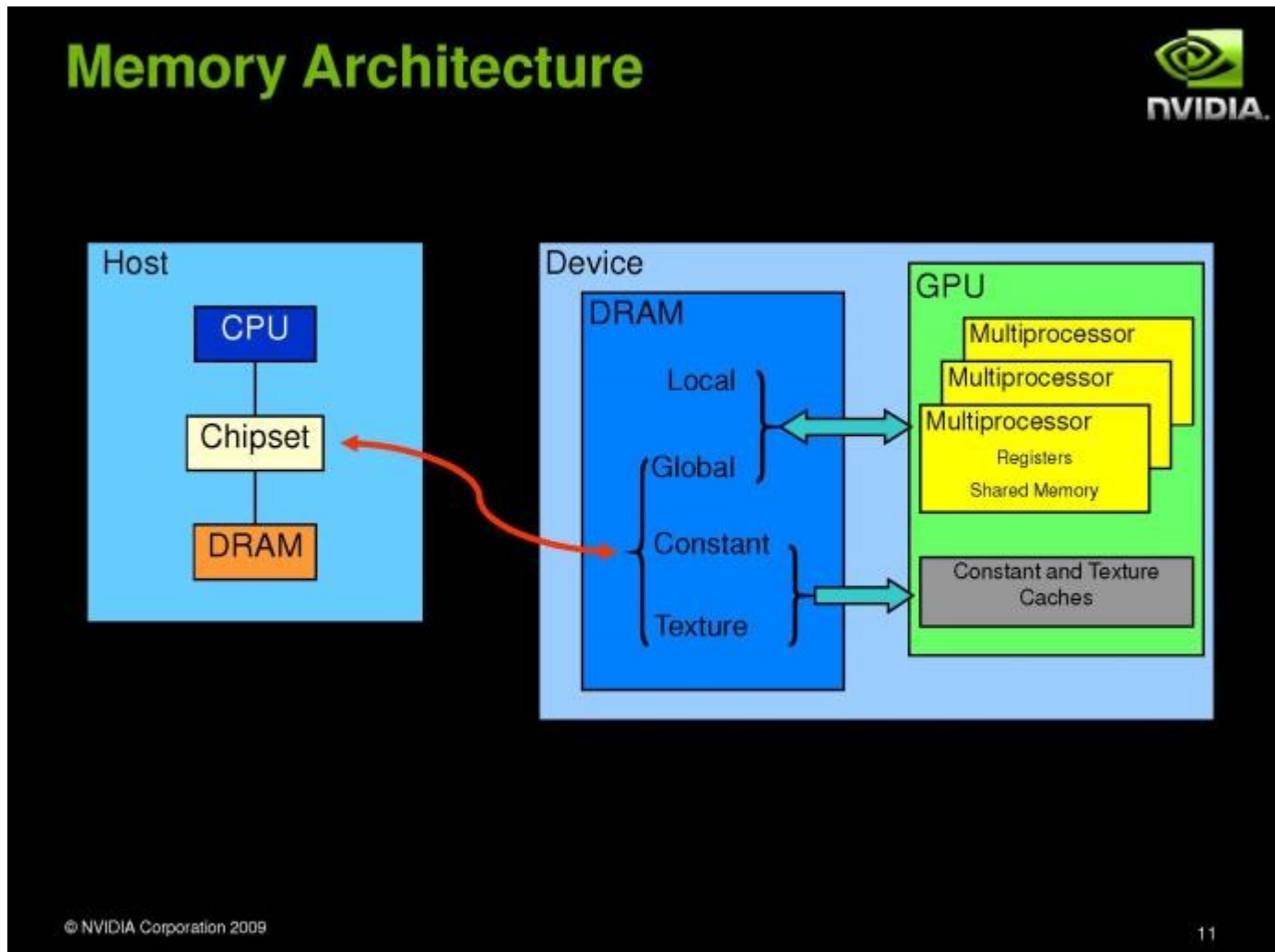
SIMT execution model

- “Single instruction, multiple threads”, and each thread *can* branch differently.
 - However, all threads in a “warp” are executed in lockstep regardless of divergent branching.
- Example:
 - if threadId is even:
 do A
 - else:
 do B
 - Execution:
 - Threads with even threadId execute A, threads with odd threadId idle.
 - Then, threads with odd threadId execute B, and threads with even threadId idle.

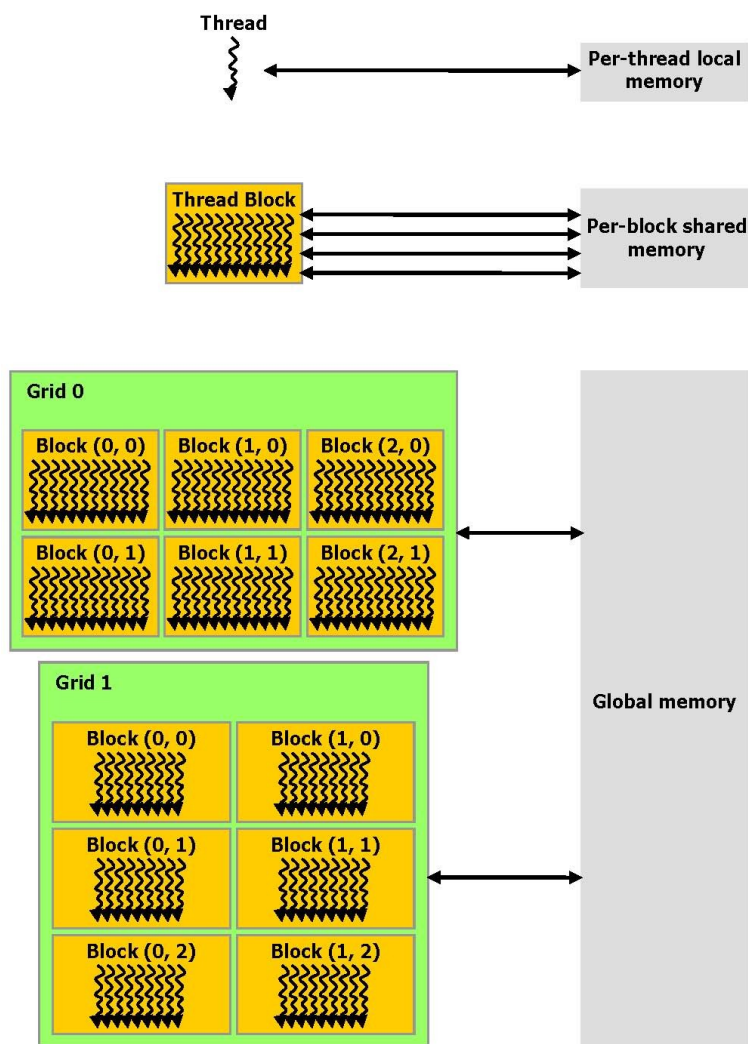
Outline

- Overview
- CUDA Hardware
- **Memory optimizations**
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

Memory architecture



Memory hierarchy



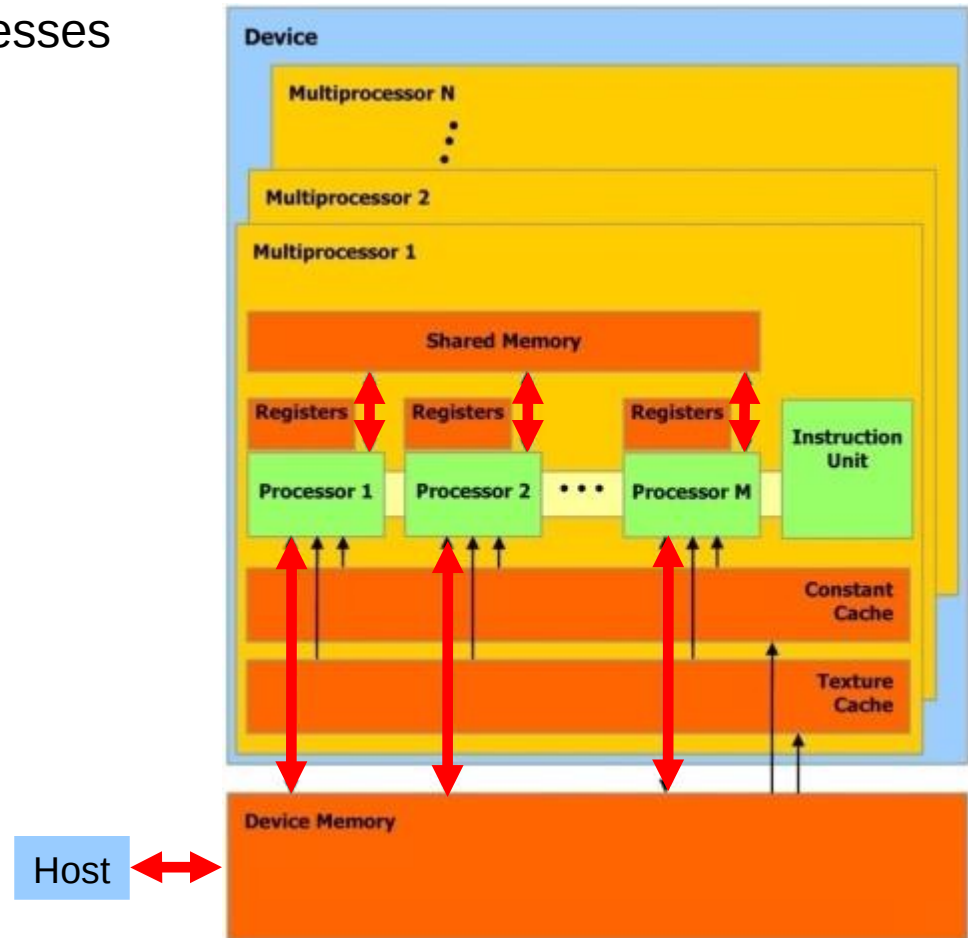
- Local memory: Off-chip
- Register: On-chip (Fermi: 128KB)
- Lifetime of thread

- On-chip, small (Fermi: 16KB/48KB)
- Fast as registers
- Variables: `__shared__`, lifetime of block

- Off-chip DRAM, large (Fermi: up to 1TB)
- Persistent across kernel launches
- Kernel I/O (host ↔ device)
- Variables: `__device__`, lifetime of application

Memory optimization

- Optimizing host-device data transfer
- Coalescing global memory accesses
- Using shared memory



Host-Device Data Transfers

- **Device to host** memory bandwidth much lower than **device to device** bandwidth
 - ❑ 8GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX280)
 - ❑ 31.51GB/s (PCI-e 4.0 x16) vs. 280GB/s (K40), 2x240 (K80)
- Minimize transfers
 - ❑ Intermediate data structures should be created, operated on, and destroyed without ever copying them to host memory
 - ❑ even if running some kernels on the device do not show performance gains
- Group transfers
 - ❑ One large transfer much better than many small ones

Pinned host memory

- Enables highest bandwidth between host and device
 - 5.2 GB/s on PCI-e x16 Gen2
- Allocated using `cudaMallocHost()`
- See the “bandwidthTest” CUDA SDK sample
- Use with caution!!
 - Allocating too much pinned host memory can reduce overall system performance
 - Test your systems and apps to learn their limits

Asynchronous memory copy

- Asynchronous host-device memory copy frees up CPU on CUDA devices
 - `cudaMemcpyAsync()` (vs. blocking `cudaMemcpy()`)
 - requires **pinned host memory** and a **stream ID**
- Stream = Sequence of operations that execute in order
 - Operations in different streams can be interleaved and overlapped
 - Stream ID used as argument to asynchronous calls and kernel launches
- Stream API:
 - 0 = default stream
 - `cudaMemcpyAsync(dst, src, size, direction, 0);`

Overlapping data transfers and computation

Asynchronous data transfers enable overlap of data transfers with computation

Overlap **host computation** with data transfer on all CUDA devices

```
cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, 0);  
kernel<<<grid, block>>>(d_a);  
cpuFunction(); //overlapped
```

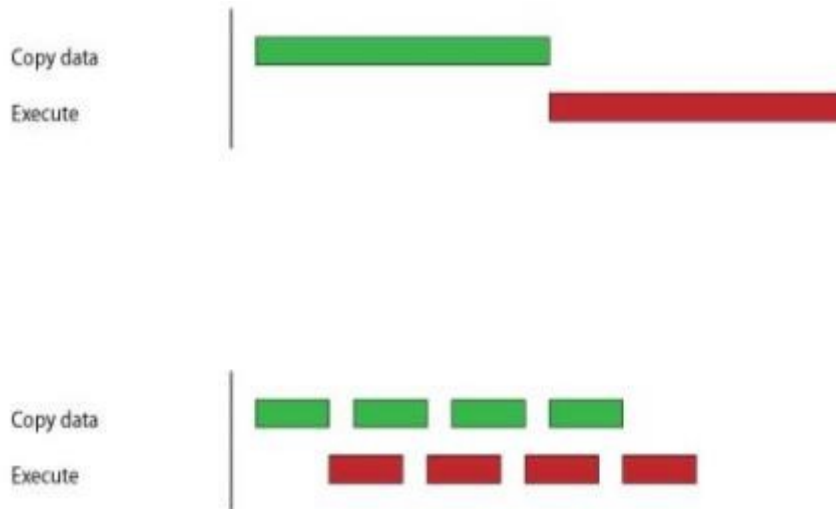
Overlap **kernel computation** with data transfer on devices with **compute capability ≥ 1.1** (ifilab: Yes!)

```
cudaStreamCreate(&stream1); //non-zero stream ID  
cudaStreamCreate(&stream2);  
cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, stream1);  
kernel<<<grid, block, 0, stream2>>>(d_otherData); //overlapped
```


Staged concurrent copy and execution

Sequential

```
cudaMemcpy(a_d, a_h, N*sizeof(float), dir);  
kernel<<<N/nThreads, nThreads>>>(a_d);
```



Concurrent

```
size=N*sizeof(float)/nStreams;  
for (i=0; i<nStreams; i++) {  
    offset = i*N/nStreams;  
    cudaMemcpyAsync(a_d+offset, a_h+offset,  
                    size, dir, stream[i]);  
}  
  
for (i=0; i<nStreams; i++) {  
    offset = i*N/nStreams;  
    kernel<<<N/(nThreads*nStreams),  
            nThreads, 0, stream[i]>>>(a_d+offset);  
}
```



Timelines for sequential and concurrent copy and kernel execution

Figures from CUDA Best Practices Guide 3.1, Nvidia.

GPU/CPU Synchronization

- Context (i.e., “GPU process”) based
 - `cudaDeviceSynchronize()`
 - Blocks until all previously issued CUDA calls from a CPU program complete
- Stream based
 - `cudaStreamSynchronize(stream)`
 - Blocks until all CUDA calls issued to given stream complete
 - Exception: stream id == 0 --> sync all streams
 - `cudaStreamQuery(stream)`
 - Indicates whether stream is idle
 - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
 - Does not block CPU thread

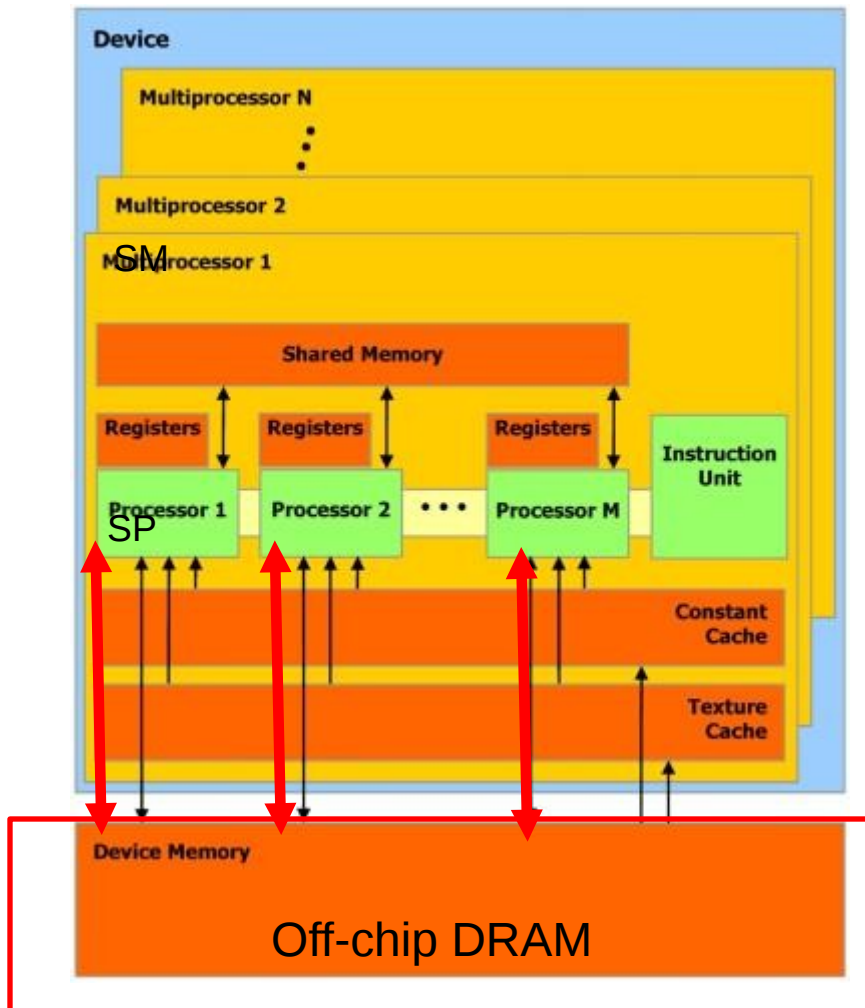
GPU/CPU Synchronization

- Stream based using events
 - Events can be inserted into streams:
 - `cudaEventRecord(event, stream)`
 - Event is recorded when GPU reaches it in a stream
 - Recorded = assigned a timestamp (GPU clocktick)
 - Useful for timing
 - `cudaEventSynchronize(event)`
 - Blocks until given event is recorded
 - `cudaEventQuery(event)`
 - Indicates whether event has recorded
 - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
 - Does not block CPU thread

Outline

- Overview
- CUDA Hardware
- Memory optimizations
 - Data transfers between host and device
 - Coalesced memory accesses
 - Shared memory
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

Memory model



CUDA Programming ver. 2.1

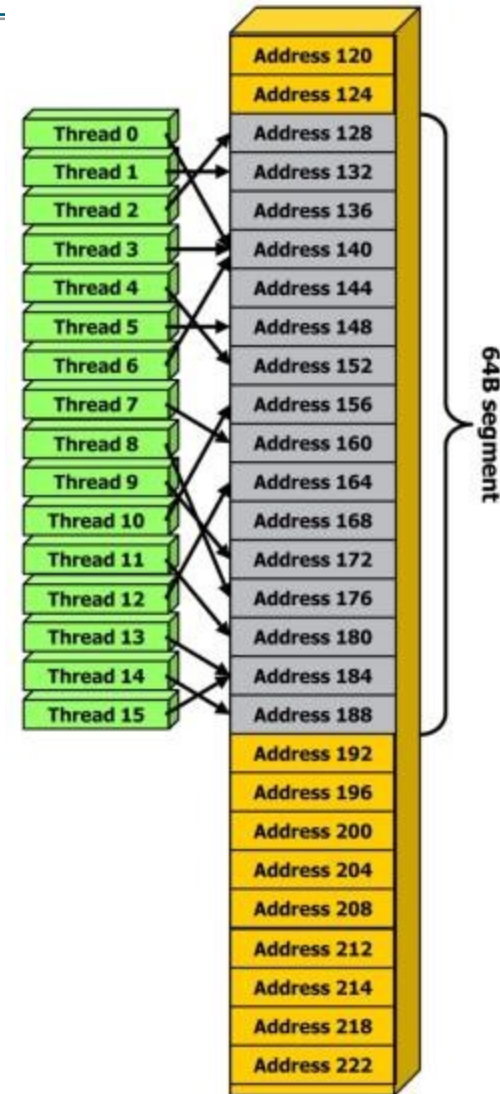
- Local storage
 - Each thread has its own local storage (e.g. registers)
- Shared memory
 - Each thread block has its own shared memory
 - Accessible by all threads in the block
 - Low latency: a few cycles
 - High throughput: 38-44 GB/s per SM \Rightarrow over 1.1 TB/s per 30 SMs
- Global memory (off-chip DRAM)
 - Accessible by all threads
 - High latency: 400-800 cycles
 - Throughput: 102-140 GB/s

Global memory accesses

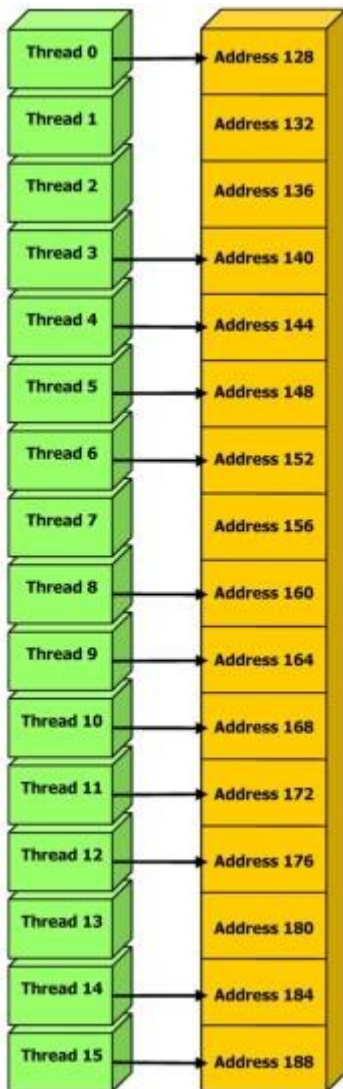
- GPU with compute capability < 2.0 (i.e. before Fermi)
 - Ifilab 2011: Quadro FX 580, compute capability 1.1
 - Global memory is not cached
 - Highest latency instructions: 400-600 clock cycles
 - Likely to be a performance bottleneck
 - Optimizations can greatly increase performance
- Ifilab 2012: Quadro 600, compute capability 2.1
 - Has some high-bandwidth caching for global memory
 - L1 cache per SM, not coherent (16-48 kb)
 - L2 cache for whole GPU, coherent (512-768kb)

Global memory

- Always accessed via 32B, 64B or 128B memory transactions
- Transaction segment must be aligned
 - First address = multiple of segment size
- **Coalesced** memory accesses
 - The global memory accesses of threads within a half-warp can be coalesced into one transaction
 - Coalescing is achieved even if the warp is divergent
 - i.e. there are some inactive threads that do not actually access memory.



Coalescing: C.C. 1.1- vs. C.C. 1.2+

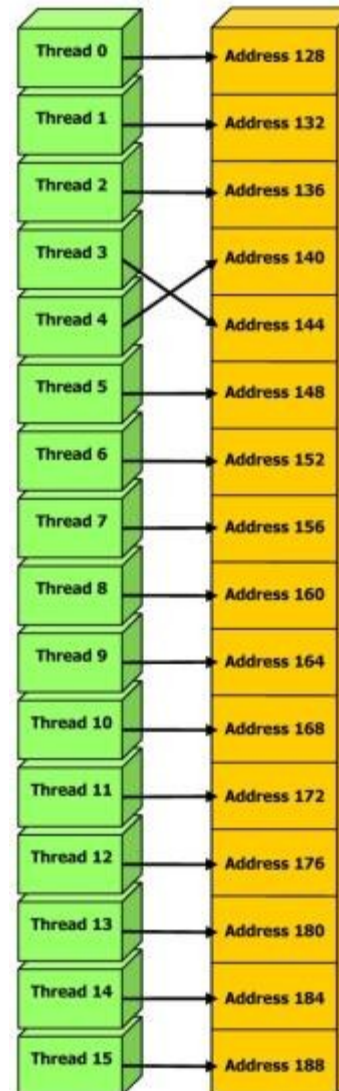


C.C. 1.1- & C.C. 1.2+

- k-th thread accesses k-th word in segment

- even divergent warp

⇒ 1 64B transaction



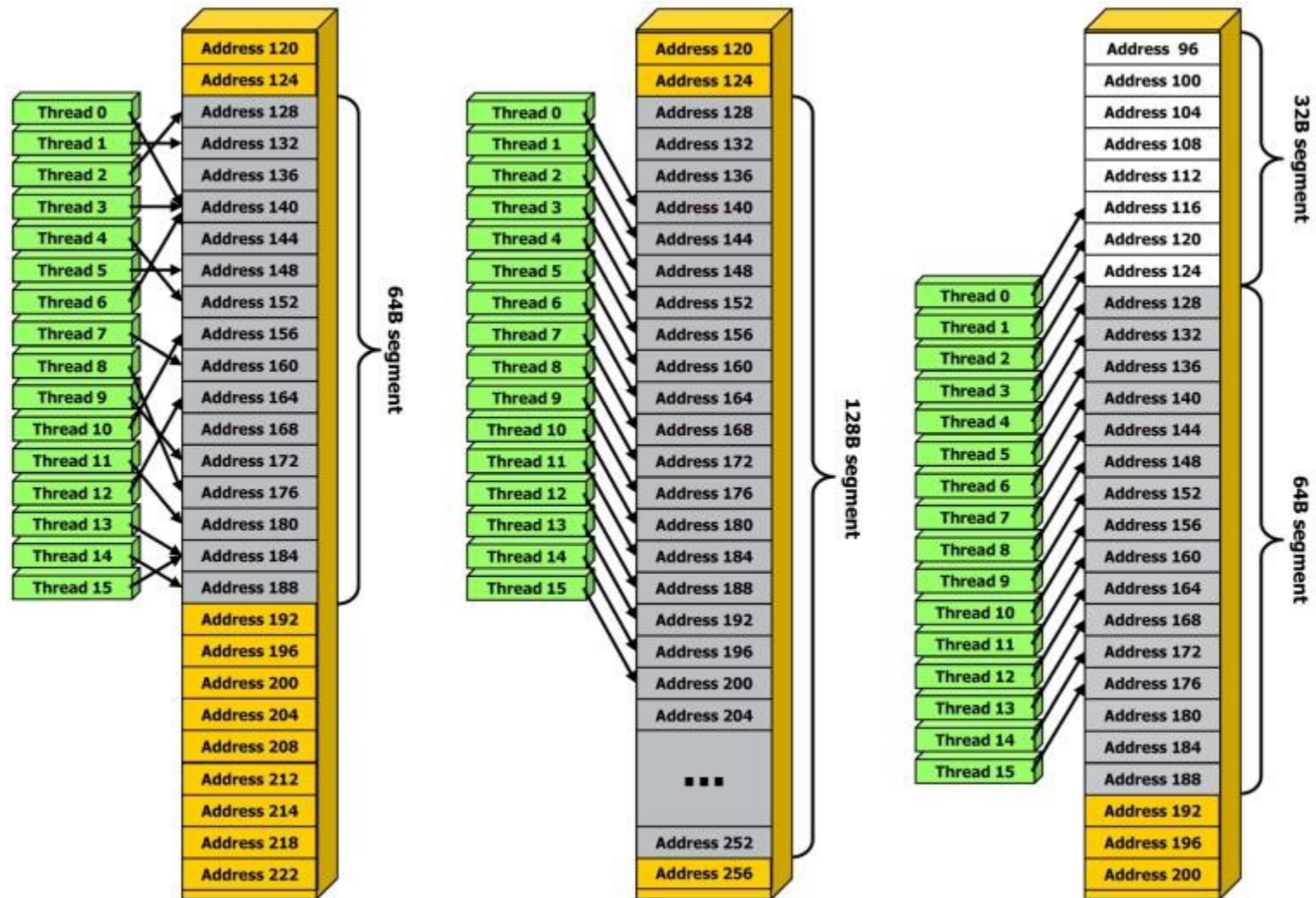
C.C. 1.1-:

16 32B transactions
(one per thread)

C.C. 1.2+:

1 64B transaction

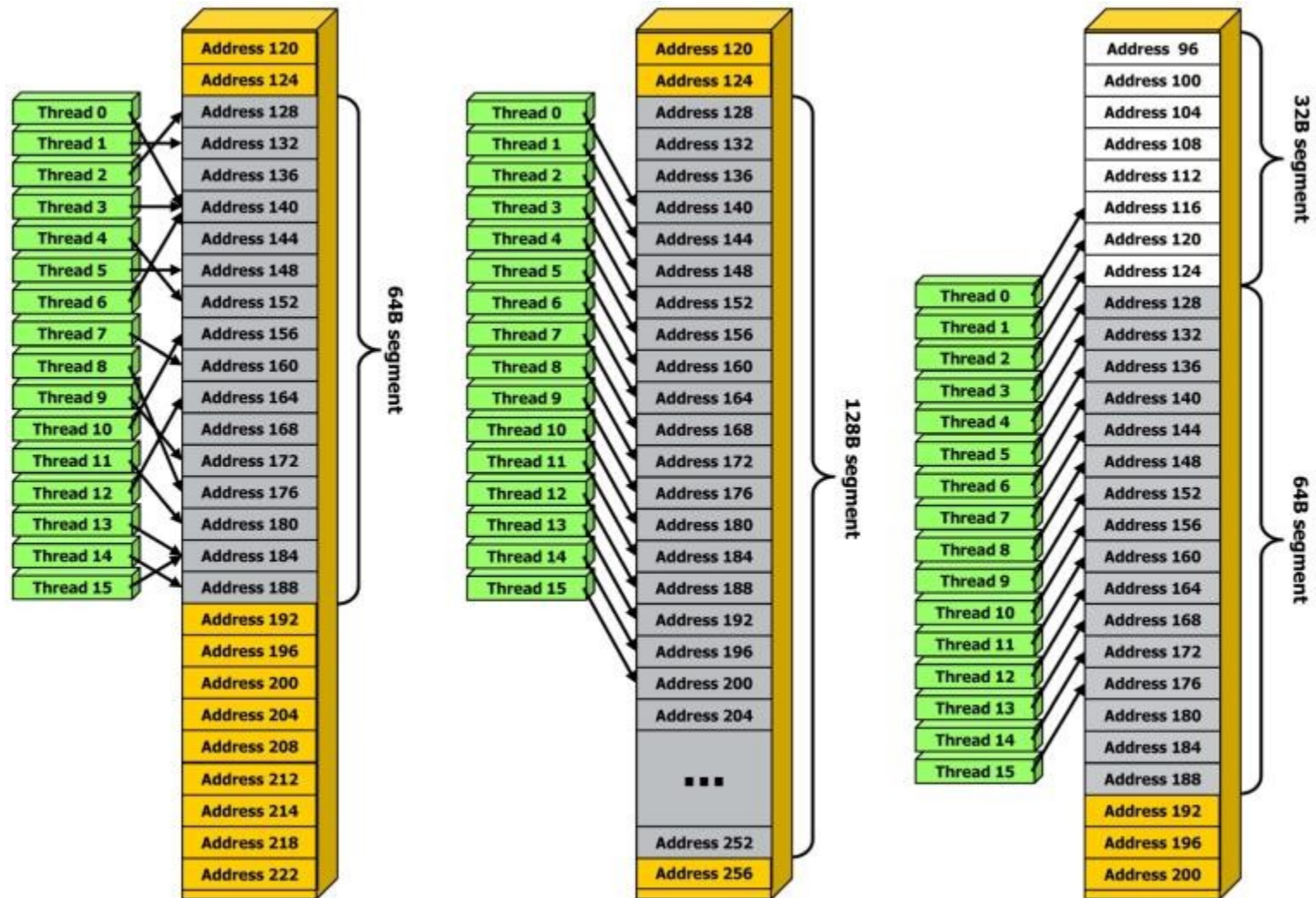
Coalescing: C. C. 1.2+



Coalescing protocol, C.C. 1.2+

- Find the memory segment that contains the address requested by the **lowest-numbered active thread**
 - 32B segment for 1B words, 64B segment for 2B words, 128B segment for 4, 8 and 16B words
- Find all other active threads whose requested address lies in the same segment
- Reduce the transaction size, if possible
 - If size = 128B and only the lower/upper half is used, reduce transaction to 64B
 - If size = 64B ~~~~~, reduce transaction to 32B
- Carry out the transaction and mark the serviced threads as inactive
- Repeat until all threads in the half-warp are serviced.

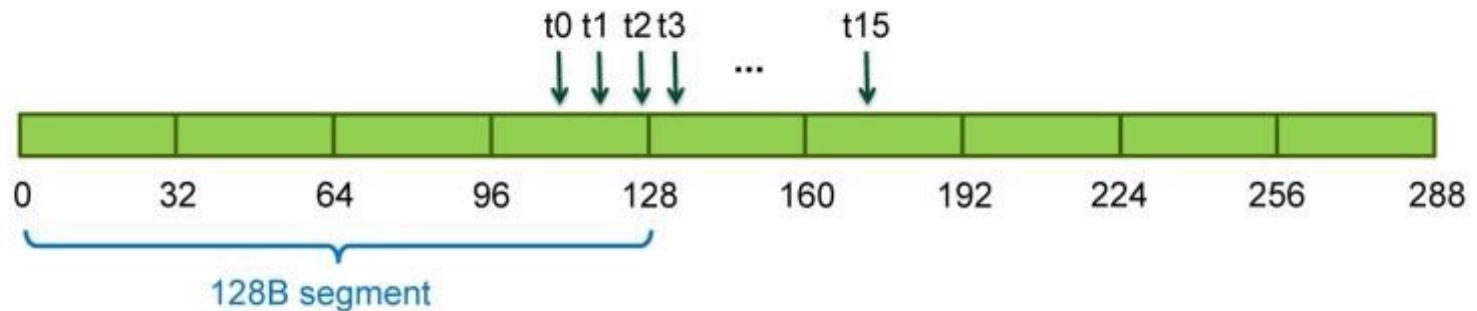
Coalescing: C. C. 1.2+



Step-by-step

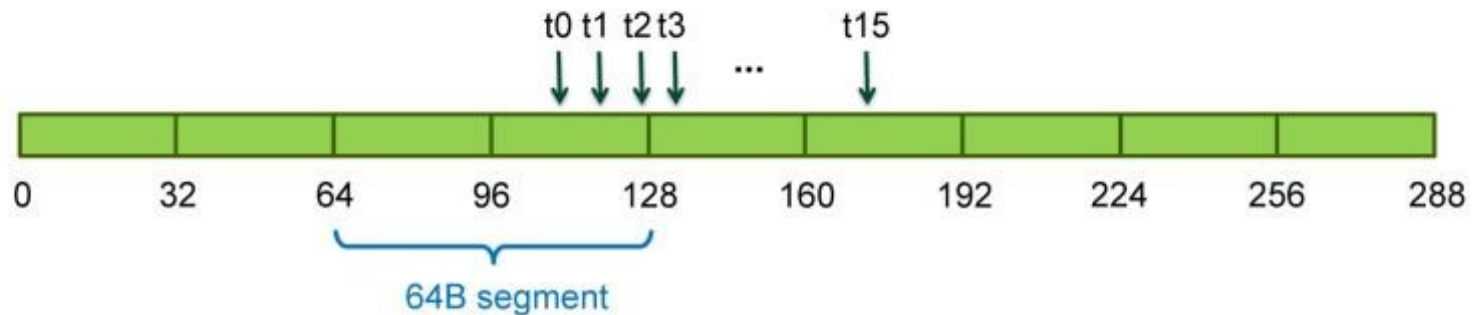
Threads 0-15 access 4-byte words at addresses 116-176

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



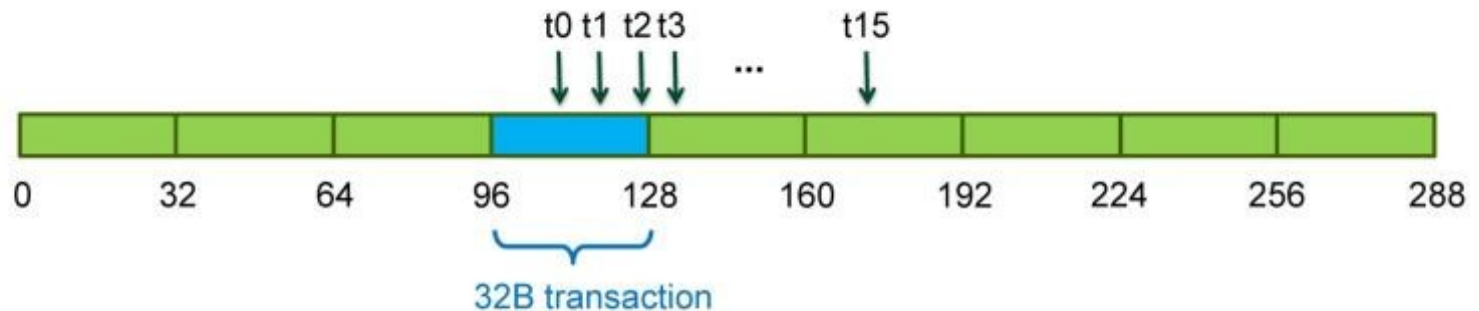
Threads 0-15 access 4-byte words at addresses 116-176

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



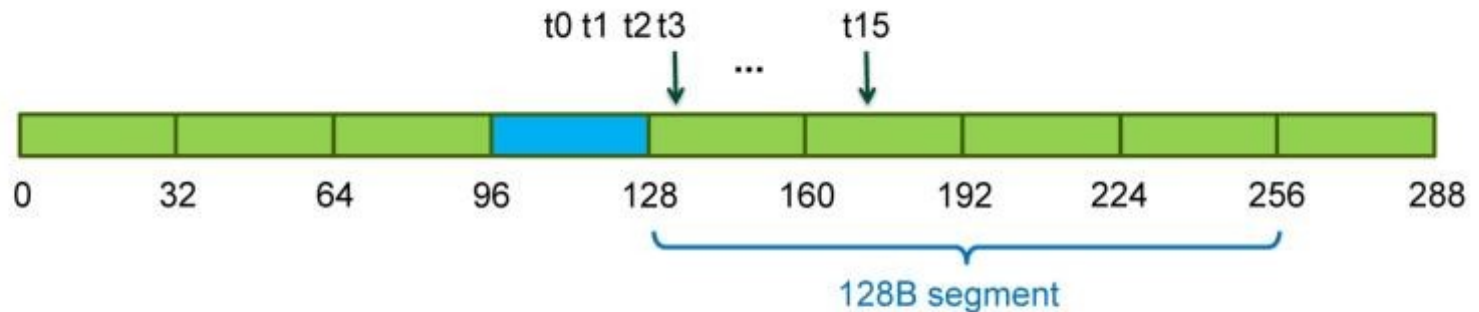
Threads 0-15 access 4-byte words at addresses 116-176

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



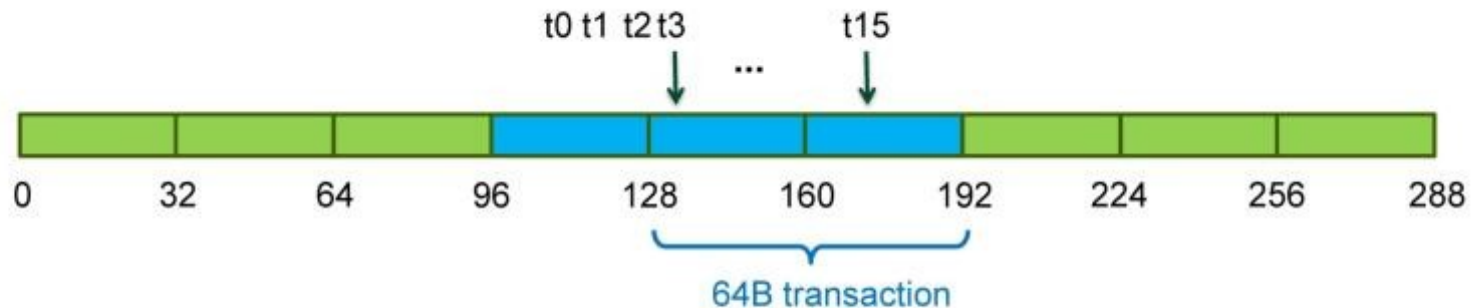
Threads 0-15 access 4-byte words at addresses 116-176

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



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

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

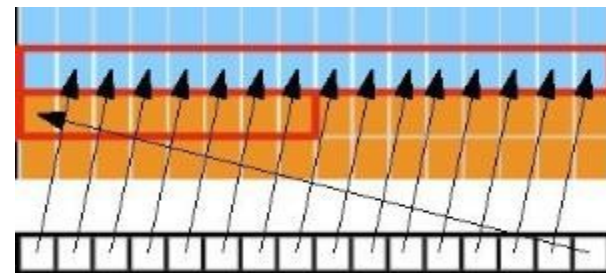
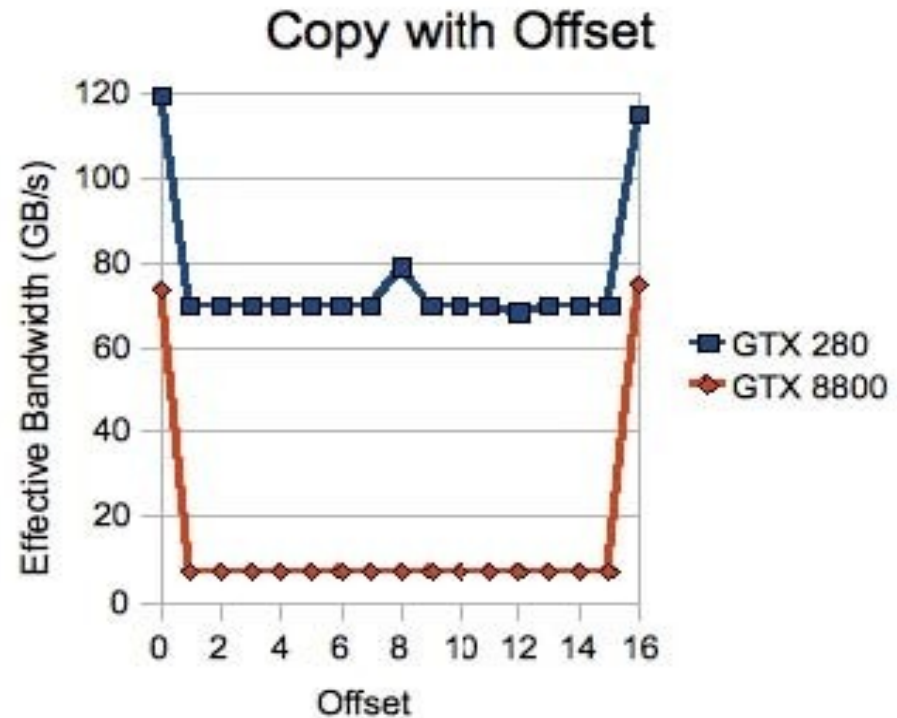


Effects of Misaligned Accesses

- Kernel

```
__global__ void offsetCopy(float *odata,
                           float* idata,
                           int offset)
{
    int xid = blockIdx.x * blockDim.x +
            threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

- GTX 8800, compute cap. 1.0
 - 1 64B- vs. 16 32B-transactions
 - 74 GBps vs. 7 GBps
- GTX 280, compute cap. 1.3
 - 1 64B- vs. 2 transactions
 - 120 GBps vs. 70 GBps



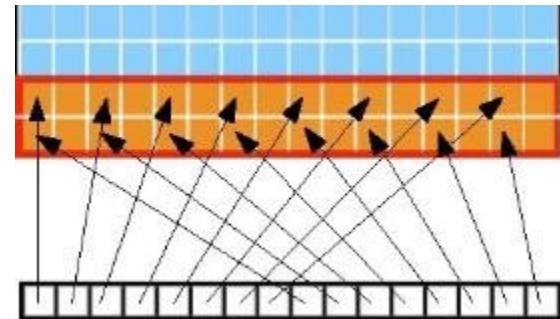
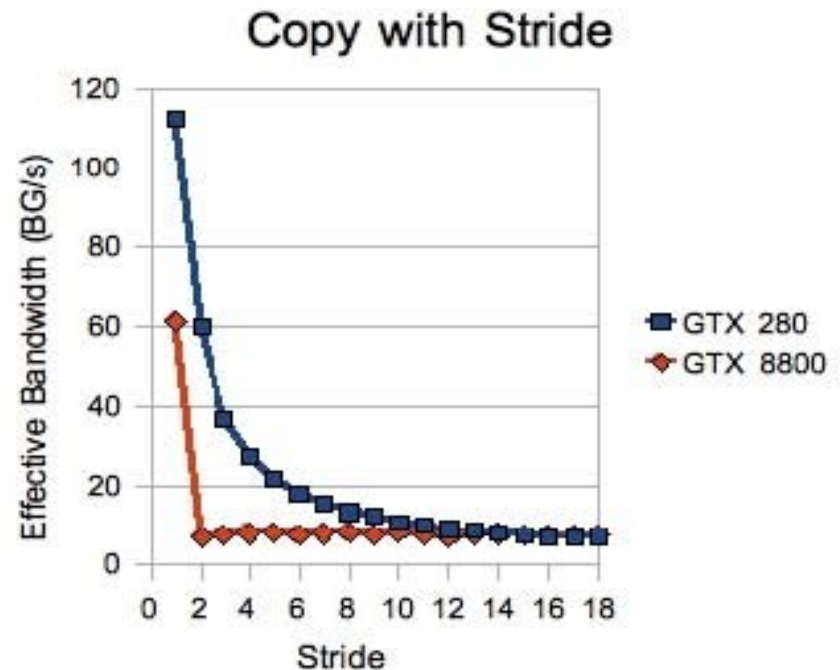
Strided accesses

- Kernel

```
__global__ void strideCopy(float *odata,
                          float* idata,
                          int stride)
{
    int xid = (blockIdx.x*blockDim.x +
              threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- C.C. 1.1- : 16 transactions
- C.C. 1.2+ : effective bandwidth decreases with increasing stride
 - Stride=2: half the elements in 128B-transaction not used
 - Stride=32: 16 transactions issued
- Use shared memory to avoid strided global mem. accesses

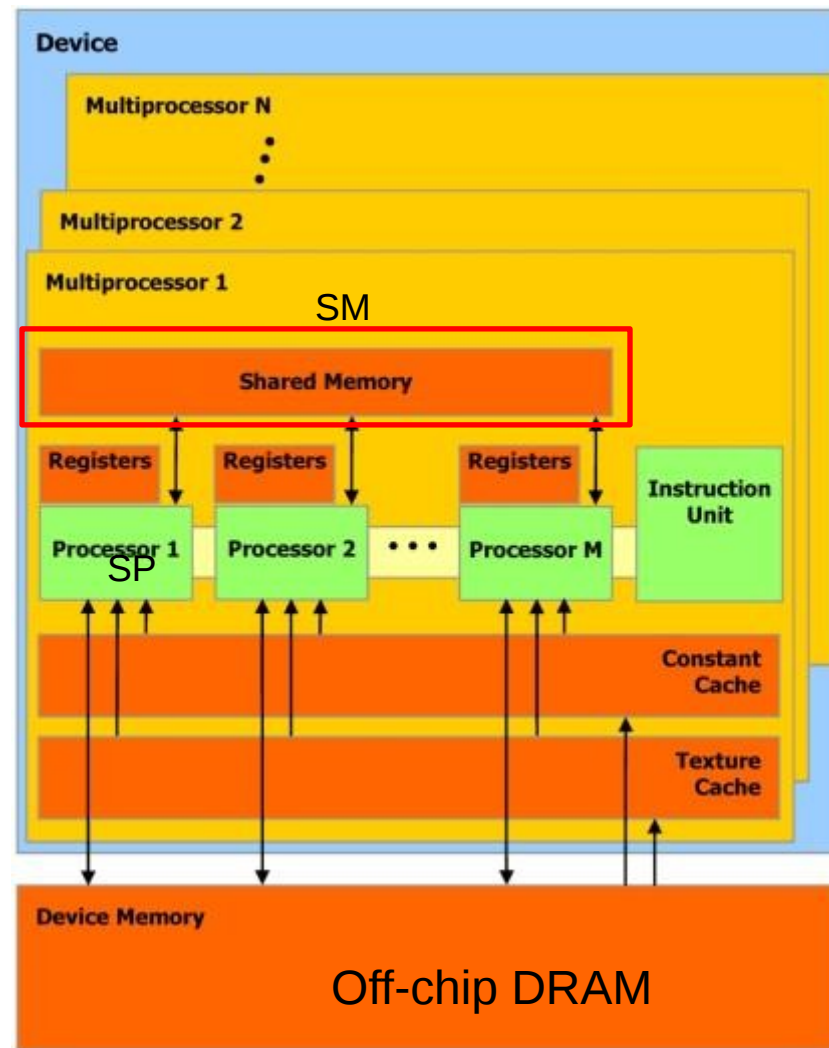
Figures from CUDA Best Practices Guide 3.1, Nvidia



Outline

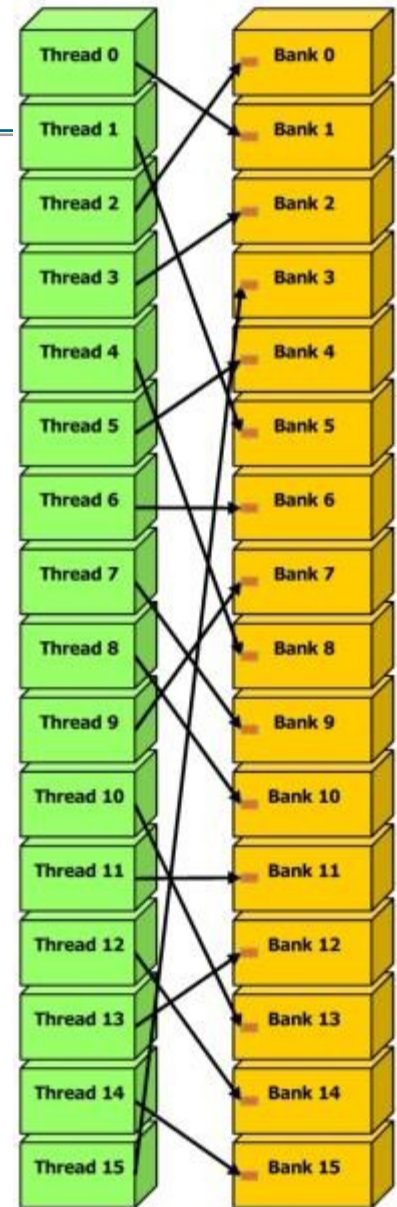
- Overview
- CUDA Hardware
- Memory optimizations
 - Data transfers between host and device
 - Coalesced memory accesses
 - Shared memory
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

Memory model



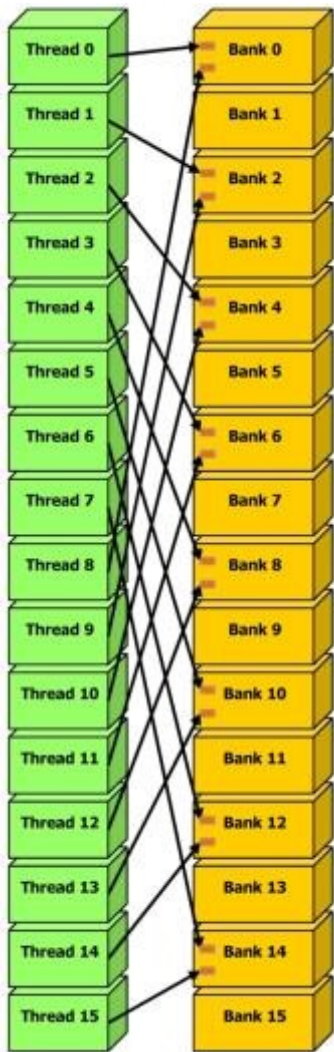
Shared memory

- Uses
 - Inter-thread cooperation within a block
 - Cache data to reduce redundant global memory accesses
- Organization:
 - 16 (C.C. 1.x), 4B wide banks
 - Can be accessed simultaneously by 16 threads (half-warp)
 - Successive 4B words belong to different banks
- Performance
 - Bandwidth: 4B per bank per clock cycle
 - Bank conflicts: if n (of 16) threads access the **same bank**, n accesses are executed serially.
 - If threads (of a warp) write to the **same location**
 - Only one of the threads performs the write
 - Broadcast: a 4B word can be **read** by n threads simultaneously



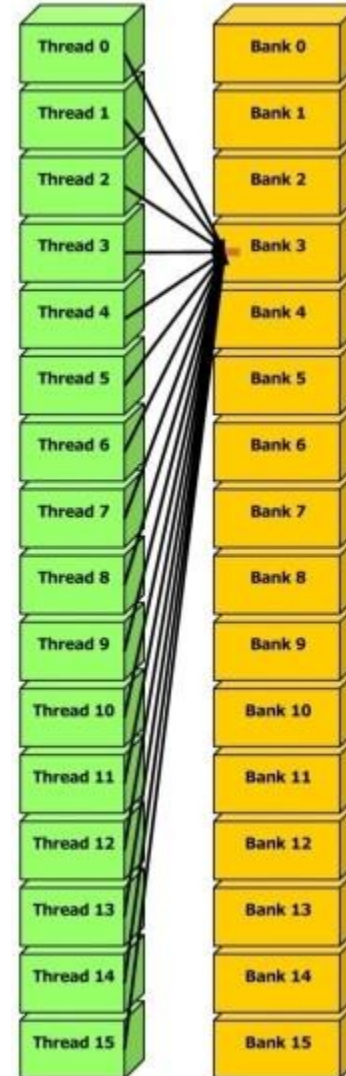
CUDA Prog. 2.1

Bank conflicts



2-way bank conflict:
Addressing with a stride
of 2 4B-words

Service 1 conflict-free
subset of accesses per step.



No bank conflict
due to broadcast

Shared memory: C.C. 2.0 vs. C.C. 1.3-

- 32 banks
- Bank conflicts:
 - only occur if two or more threads access any bytes within **different** 4B words belonging to the same bank.
 - If two or more threads access any bytes within the **same** 4B word, there is no bank conflict between these threads

Ex:

```
__shared__ char shared[32]  
char data = shared[BaseIndex + tid]
```

- C.C. 1.3-: 4-way bank conflict since shared[0], ... shared[3] belong to the same bank.
- C.C. 2.0: no bank conflict since shared[0], ... shared[3] within the same 4B word.

Outline

- Overview
- CUDA Hardware
- Memory optimizations
 - Data transfers between host and device
 - Coalesced memory accesses
 - Shared memory
- Execution Configuration Optimizations
- Task scheduling
- Atomic operations for non-blocking synchronization

Occupancy

- Thread instructions are executed sequentially
⇒ executing other warps is the only way to hide latencies and keep the hardware busy
- Occupancy = $\frac{\text{\#active warps}}{\text{max \#active warps}}$
 - per multiprocessor
- Limited by resource usage:
 - Registers
 - Shared memory

Determining resource usage

- Compile the kernel code with `—ptxas-options=-v` flag to `nvcc`
 - Open the `.cubin` file with a text editor and look for the “code” section

```
code {
    name = BlackScholesGPU
    lmem = 0    //local memory per thread
    smem = 68  //shared memory per threadblock
    reg = 20   //registers per thread
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09 0x00200780 ...
```
- [CUDA Occupancy calculator](#)
 - CUDA SDK tools

Register dependency

- Read-after-write register dependency
 - Instruction's result can be read ~24 cycles later
 - Scenario:

```
int x = y + 5;  
z = x + 3;
```
- To completely hide the latency:
 - Run at least 192 threads (6 warps) per multiprocessor
 - Then, you have 24 threads per scalar processor
 - x for first thread is ready to be read in line 2 right after 24th thread finished executing line 1
 - Scalar procesors busy all the time -> "latency is hidden"!
 - Note: threads needn't be in the same thread block

Thread and block heuristics

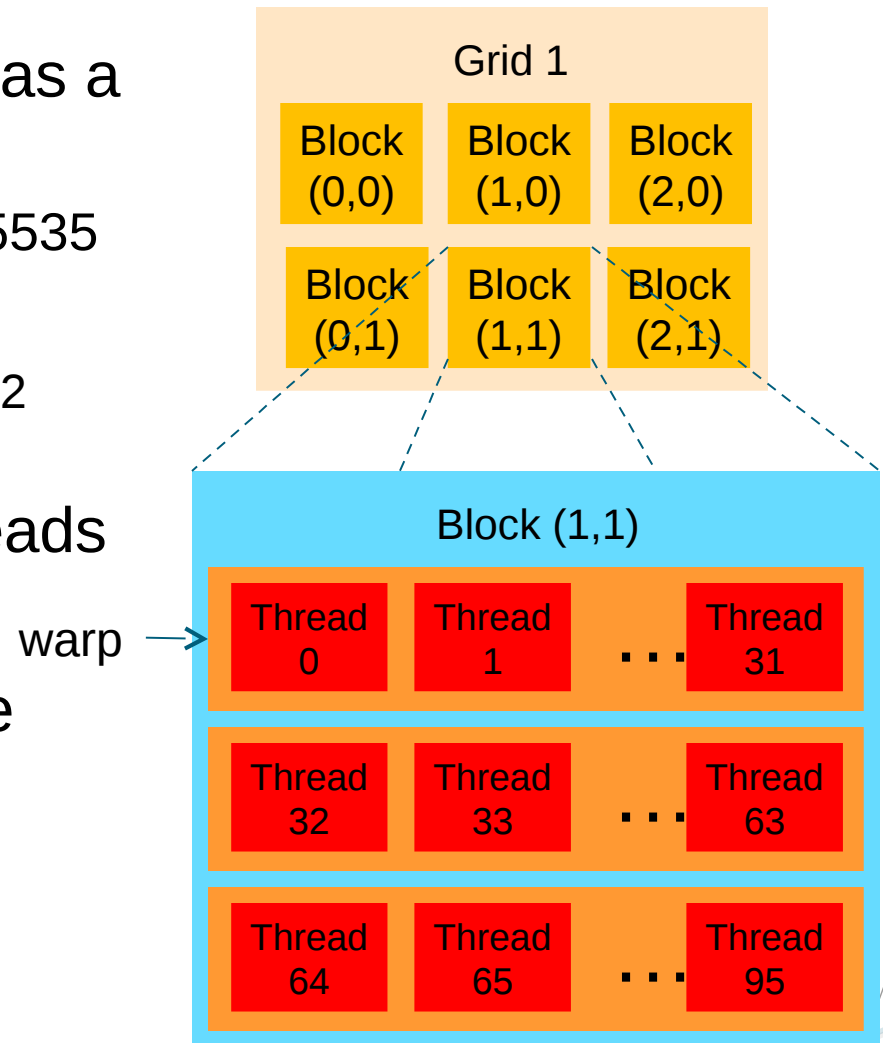
- Goal: balance latency hiding (occupancy) and resource utilization
 - Grid size:
 - $\text{\#blocks} > \text{\#multiprocessors}$
 - $\text{\#blocks} / \text{\#multiprocessors} > 2$
 - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
 - Subject to resource availability – registers, shared memory
 - $\text{\#blocks} > 1000$ to scale to future devices
 - Block size:
 - $\text{\#threads/block} = \text{a multiple of warp size (i.e. 32)}$
 - More threads per block -> better memory latency hiding
 - But, more threads per block -> fewer registers per thread
 - Experiment!

Outline

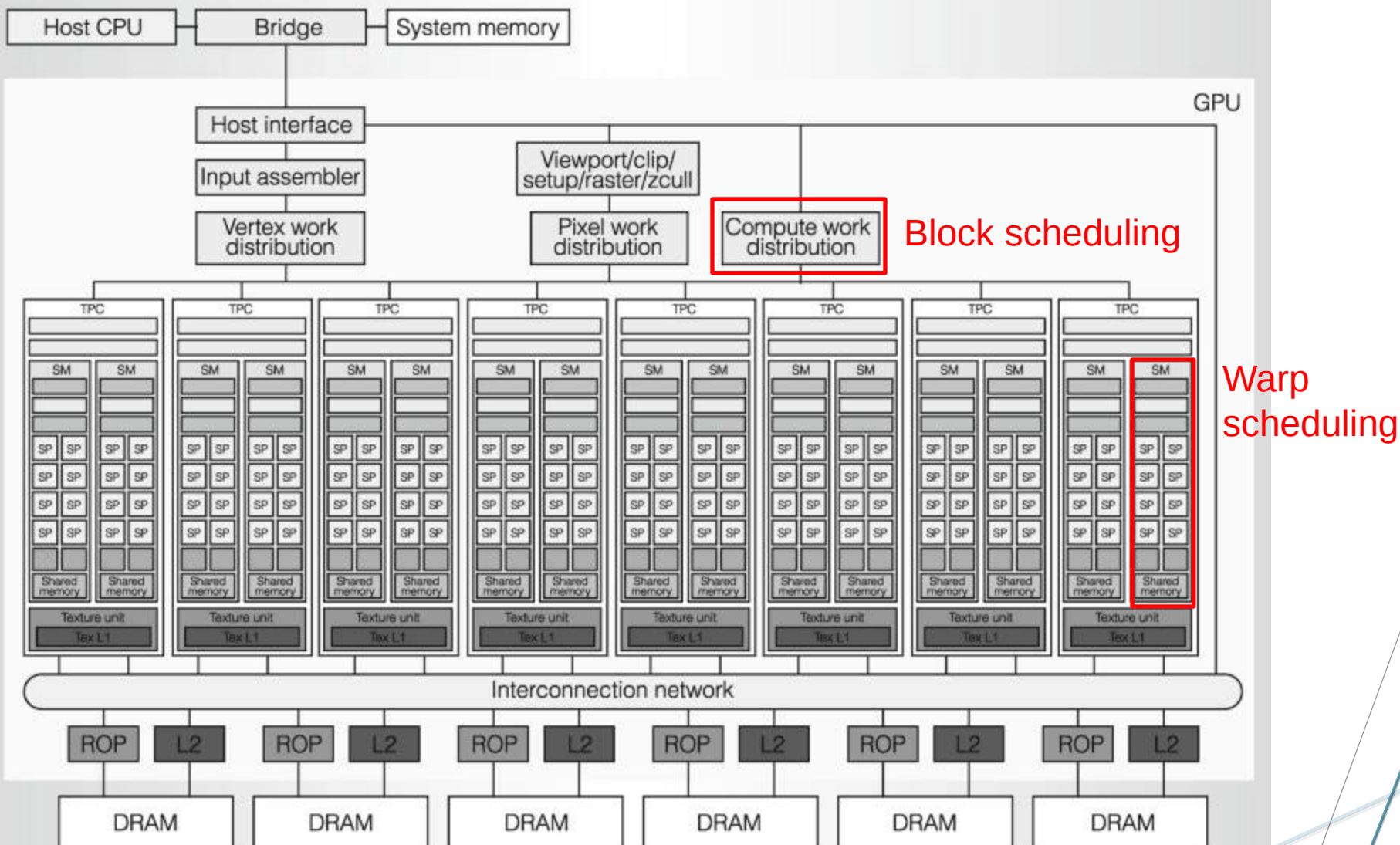
- Overview
- CUDA Hardware
- Memory optimizations
 - Data transfers between host and device
 - Coalesced memory accesses
 - Shared memory
- Execution Configuration Optimizations
- **Task scheduling**
- Atomic operations for non-blocking synchronization

What are “tasks” in CUDA?

- Parallel code is launched as a **grid** of thread blocks
 - Max x- or y-dimension is 65535
- A **block** consists of warps
 - Compute capability 2.0: max 32 warps/block
- A **warp** consists of 32 threads
- Tasks to be scheduled are blocks and warps

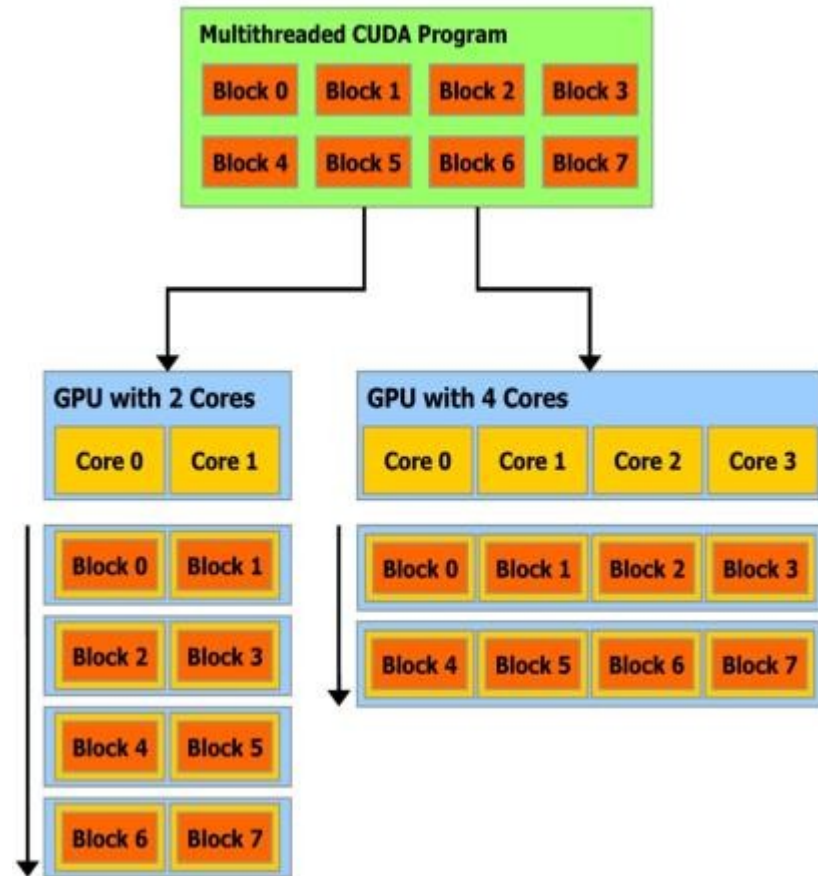


CUDA Architecture



Block scheduling

- CUDA Prog. 1.1 (Nov. '07)
 - The issue order of the blocks within a grid of thread blocks is undefined
- IEEE Micro '08
 - The compute work distributor delivers blocks to SMs with **sufficient resources** in a **round-robin** scheme **without preemption**
- Blocks do not migrate
- Several concurrent thread blocks can reside on one SM
 - Max 8 blocks / SM



CUDA Programming 3.0

Potential deadlock

- Scenario 1
 - Threadblock scheduling
 - **Non-preemptive**: once a threadblock is active, it holds the resources (register/smem) until all of its threads complete execution.
 - **Blocking synchronization**

```
if (blockID  $\neq$  i)
    while (S = 0); /*busy-wait*/
    ...
else
    S $\leftarrow$ 1;
    ...
```

- Threadblock **i** has not run yet and will get activated only when some currently active threadblock finishes

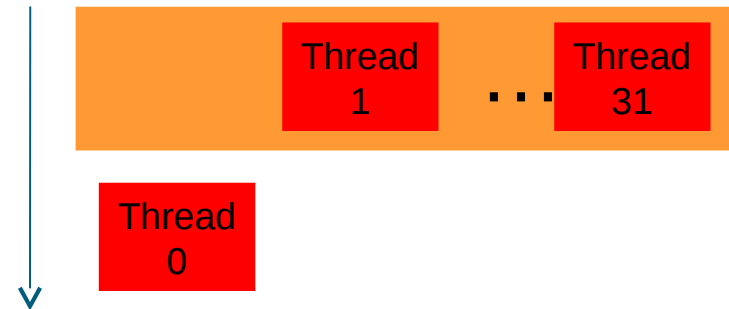
Warp scheduling

- The SM warp scheduler is a **priority-based** scheduler
- A scoreboard qualifies each warp for issue each cycle
- The scheduler prioritizes all ready warps and selects the one with highest priority for issue
 - Compute capability 2.0: 48 warps / SM
- “Zero”-overhead warp scheduling
- Compute capability 2.0
 - 1st scheduler issues instructions for a warp with an odd ID and 2nd one for a warp with an even ID

Potential deadlock

- Scenario 2:
 - Divergent warps
 - Different execution paths within a warp are serialized
 - Blocking synchronization
 - using a busy-waiting loop: the path with a busy-waiting loop is scheduled first.
 - using barrier primitive `__syncthreads()` in different paths

```
if (threadID ≠ 0)
    while (S=0); /*busy-wait*/
...
else
    S←1;
...
```



⇒ non-blocking synchronization ⇒ atomic operations

Outline

- Overview
- CUDA Hardware
- Memory optimizations
 - Data transfers between host and device
 - Coalesced memory accesses
 - Shared memory
- Task scheduling
- Atomic operations for non-blocking synchronization

Examples

```
atomicAdd( address, val)
atomically {
    old ← *address;
    *address ← (old + val);
    return old;
}
```

```
atomicSub()
atomicExch()
...
```

```
atomicCAS( address, compare, val)
atomically {
    old ← *address;
    if (old == compare)
        *address ← val;
    return old;
}
```

Atomic operations

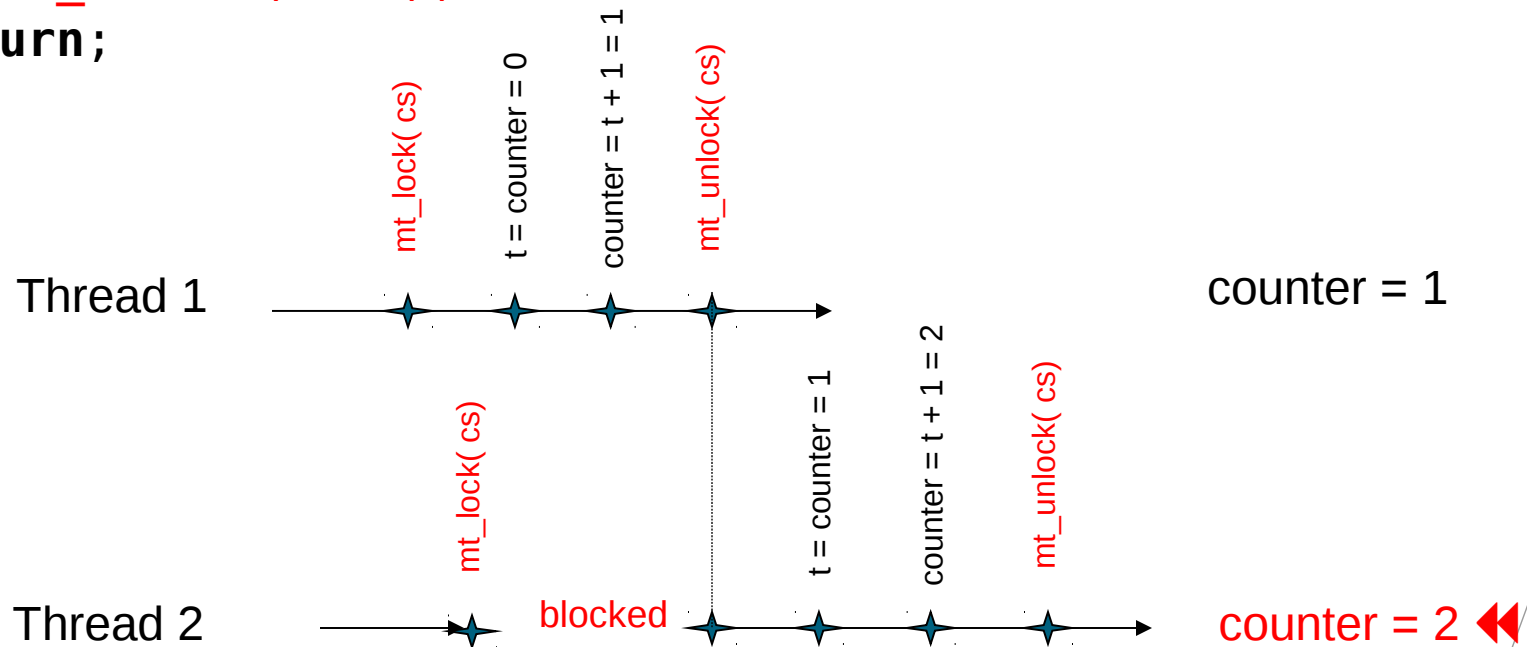
- History
 - Compute Capability

C.C. 1.0	C.C. 1.1	C.C. 1.2 and above
<ul style="list-style-type: none">• No atomic ops• No synchronization mechanism between blocks (CUDA Prog. 1.1)	<ul style="list-style-type: none">• Atomic ops only for global memory	<ul style="list-style-type: none">• Atomic ops for both shared memory and global memory

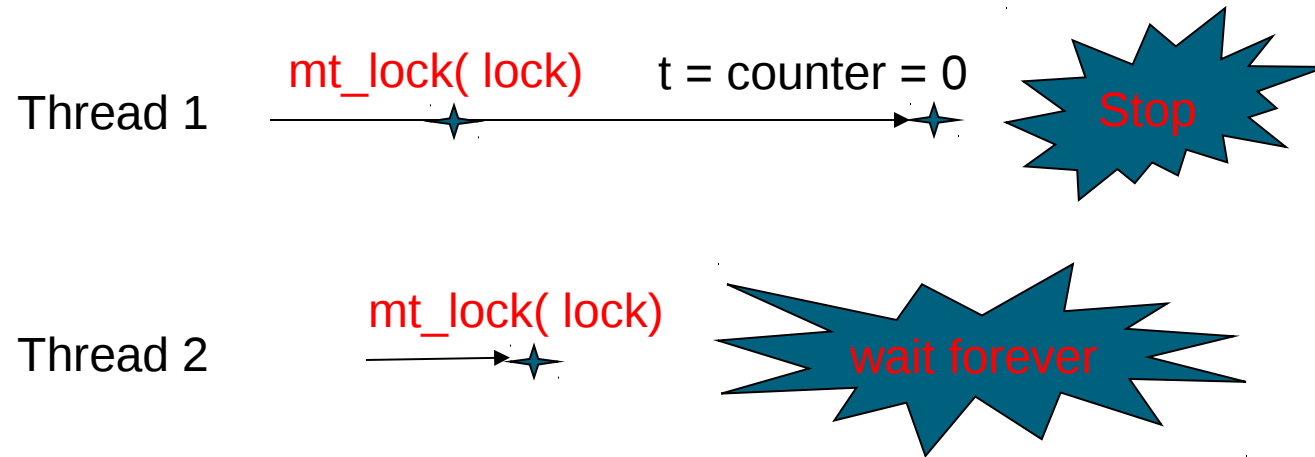
- Why should processors, even graphics processors, support atomic operations?
 - Facilitate inter-thread cooperation
 - Concurrent data structures (e.g. stack, queue, list)
 - Non-blocking synchronization
 - Deadlock-freedom

Blocking synchronization: Mutual exclusion

```
shared counter = 0; cs = free;  
Increment() {  
    mutex_lock( &cs); //synch. point  
    t = counter;  
    counter = t + 1;  
    mutex_unlock( &cs);  
    return;  
}
```



Deadlock!




Non-blocking synchronization

```
shared c = 0;
Increment() {
    do {
        t = c;
        new = t + 1;
    } while( atomicCAS( &c, t, new) != t)
    return new;
}
```

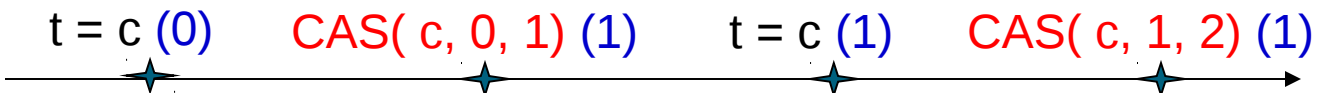
```
atomicCAS( address, compare, val)
atomically {
    old ← *address;
    if (old == compare)
        *address ← val;
    return old;
}
```

Thread 1 $t = c$ (0) $\text{CAS}(c, 0, 1)$ (0)



Result = 1

Thread 2 $t = c$ (0) $\text{CAS}(c, 0, 1)$ (1) $t = c$ (1) $\text{CAS}(c, 1, 2)$ (1)



Result = 2 ◀◀

References

- CUDA Best Practices Guide 3.1, Nvidia, 2009
- Introduction to CUDA Programming, Nvidia, 2008
- CUDA Programming Guide 3.1, Nvidia, 2009
- Nvidia CUDA Webinars
- David B. Kirk & Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, ISBN-13: 978-0123814722.