

CS 677: Parallel Programming for Many-core Processors

Lecture 3

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

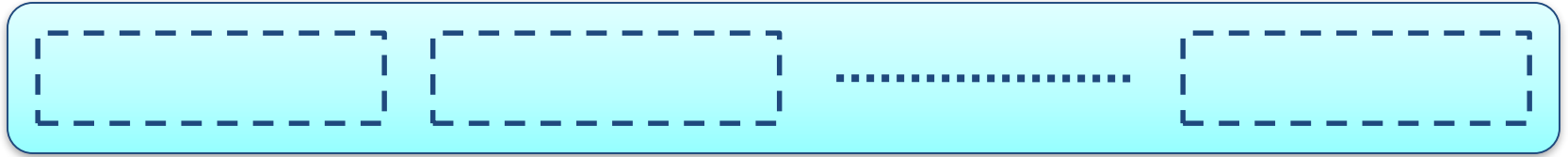
Overview

- A Common Programming Strategy
- Threading Hardware
- Memory Hardware
- Control Flow
 - Simple Reduction

A Common Programming Strategy

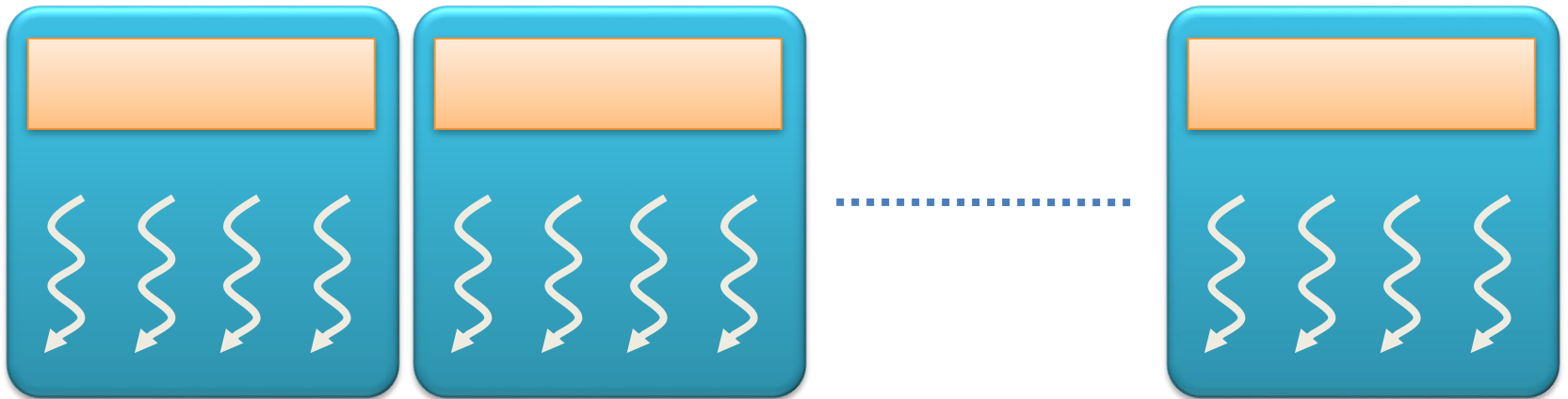
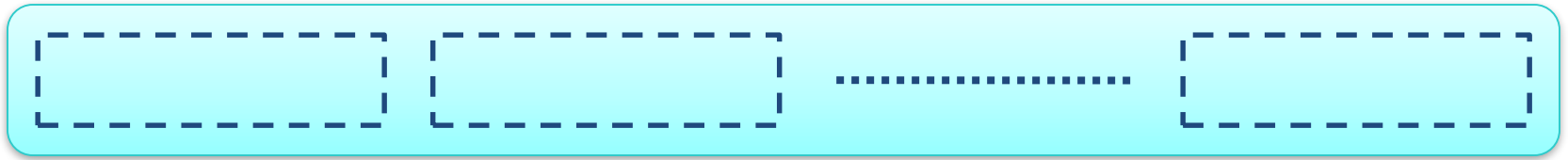
- Global memory resides in device memory (DRAM)
 - Much slower access than shared memory
- **Tile data** to take advantage of fast shared memory:
 - Generalize from `adjacent_difference` example
 - Lecture 2, slides 35-40
 - Divide and conquer

A Common Programming Strategy



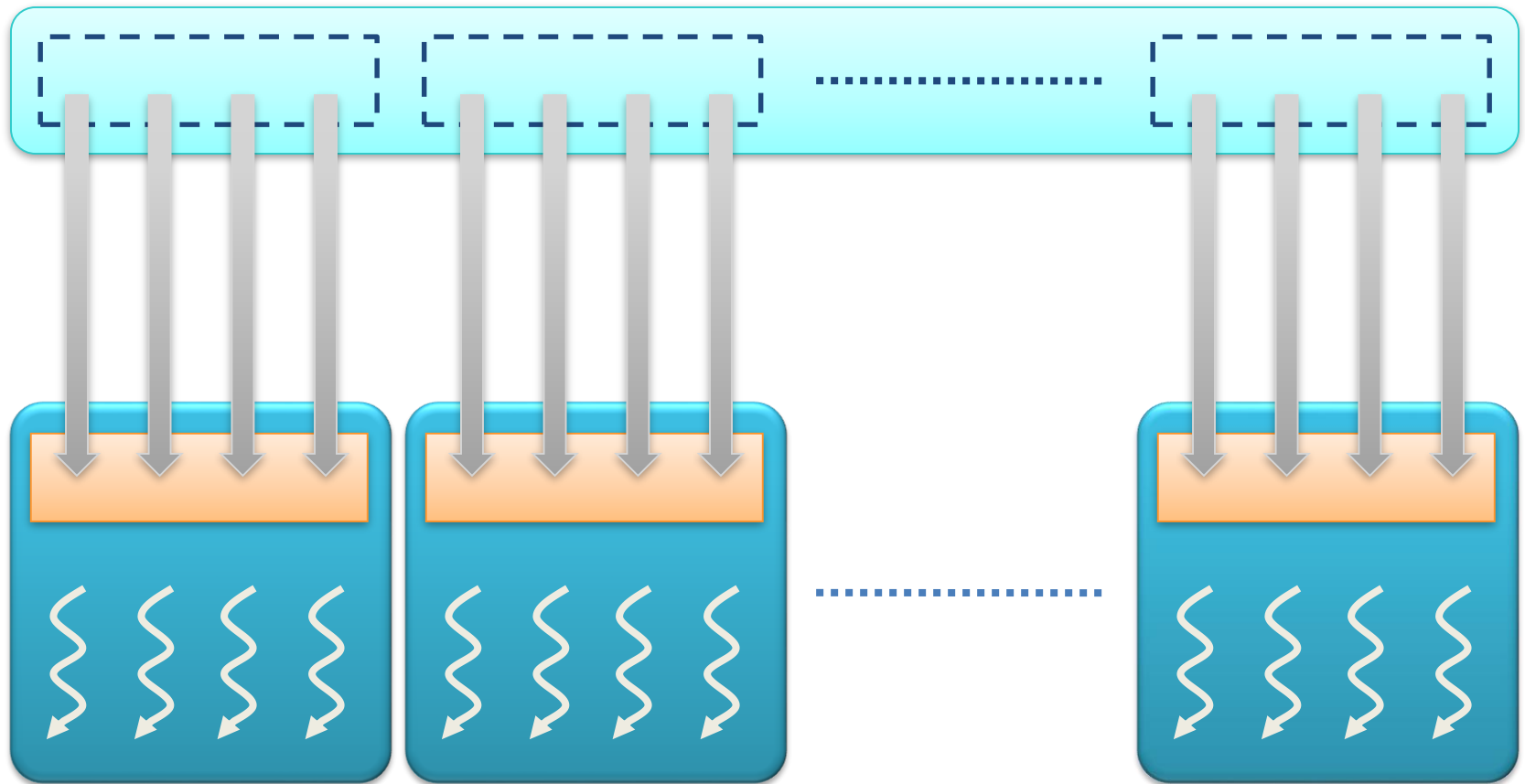
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



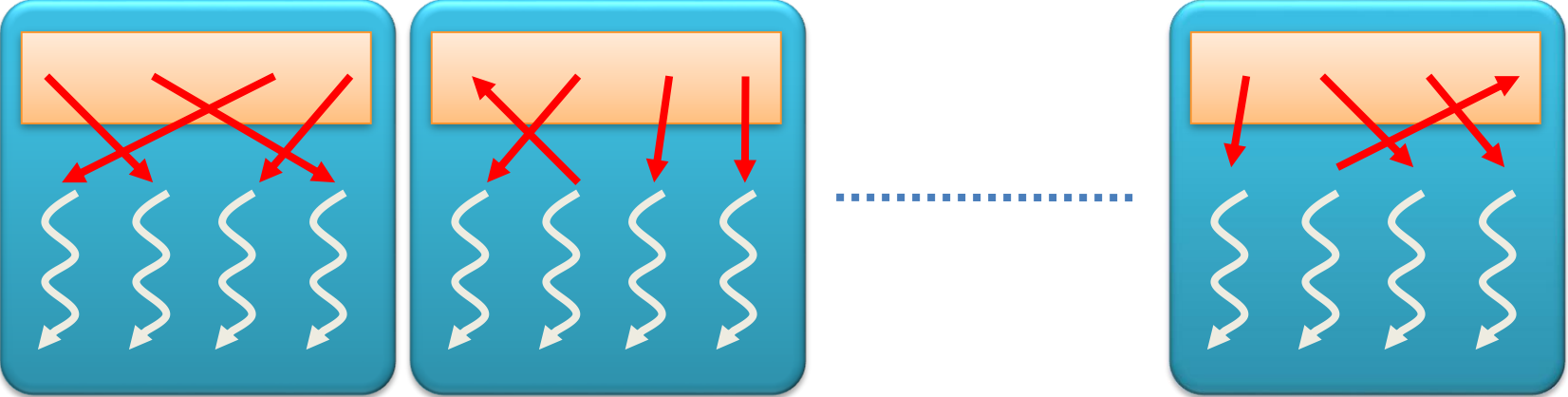
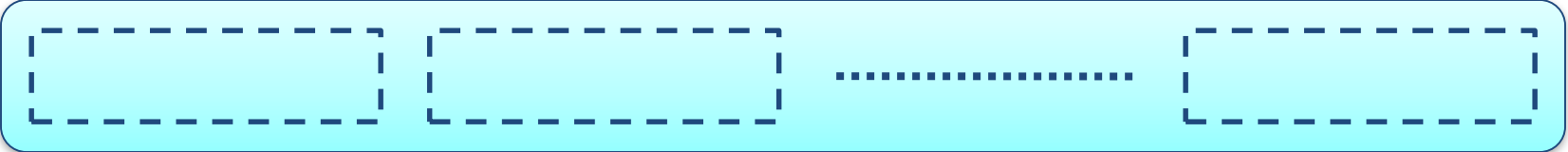
- Handle each data subset with one **thread block**

A Common Programming Strategy



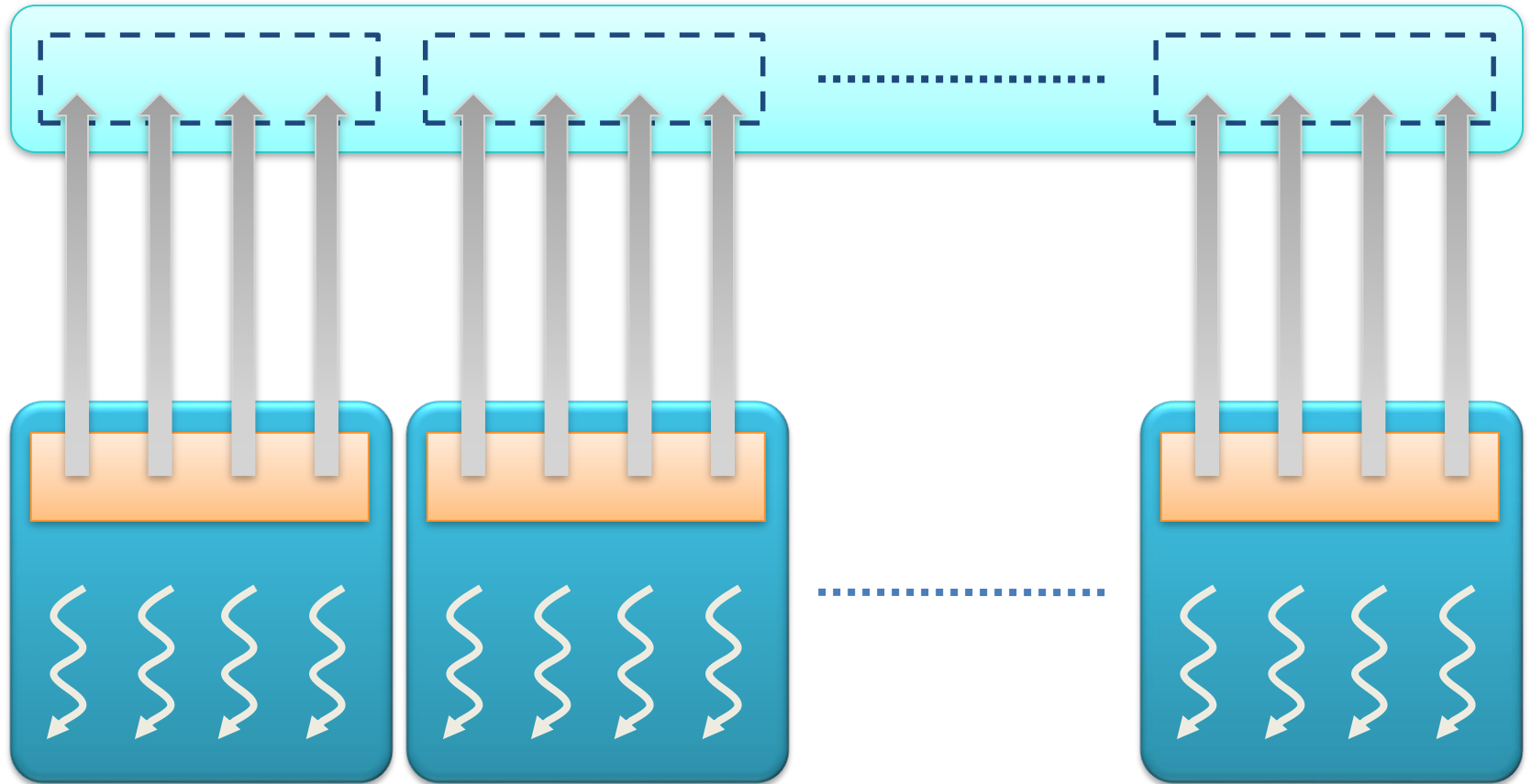
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

A Common Programming Strategy



- Perform the computation on the subset from shared memory

A Common Programming Strategy



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

A Common Programming Strategy

- Carefully partition data according to access patterns
- Read-only → `__constant__` memory (fast)
- R/W & shared within block → `__shared__` memory (fast)
- R/W within each thread → registers (fast)
- Indexed R/W within each thread → local memory (slow)
- R/W inputs/results → `cudaMalloc`'ed global memory (slow)

Communication Through Memory

- Question:

```
__global__ void race(void)
{
    __shared__ int my_shared_variable;
    my_shared_variable = threadIdx.x;

    // what is the value of
    // my_shared_variable?
}
```

Communication Through Memory

- This is a **race condition**
- The result is **undefined**
- The order in which threads access the variable is undefined without explicit coordination
- Use barriers (e.g., `__syncthreads`) or atomic operations (e.g., `atomicAdd`) to enforce **well-defined** semantics

Threading Hardware

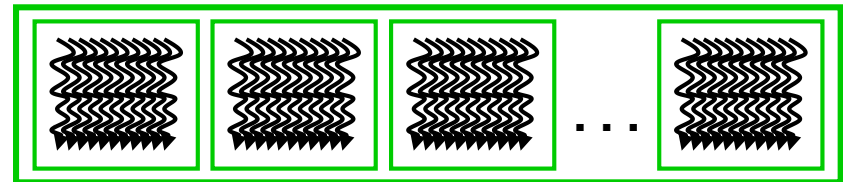
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
 - Serial C code executes on CPU
 - Parallel Kernel C code executes on GPU thread blocks

CPU Serial Code

GPU Parallel Kernel

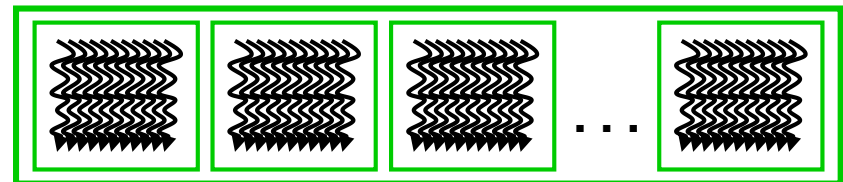
KernelA<<< nBlk, nTid >>>(args);



CPU Serial Code

GPU Parallel Kernel

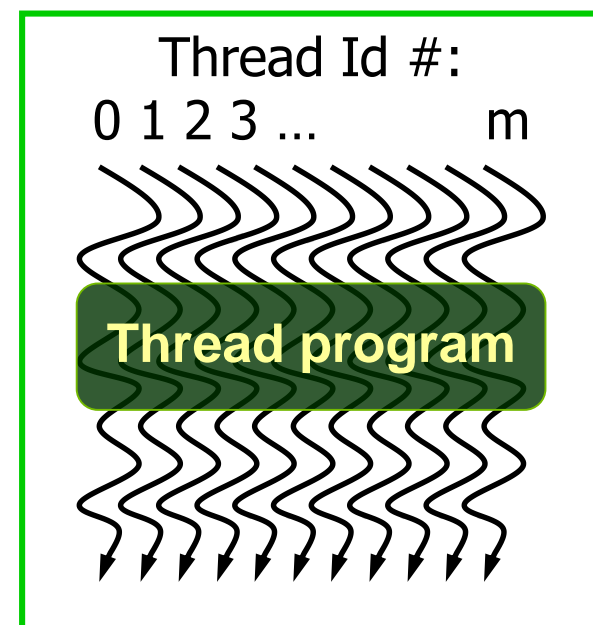
KernelB<<< nBlk, nTid >>>(args);



CUDA Thread Block: Review

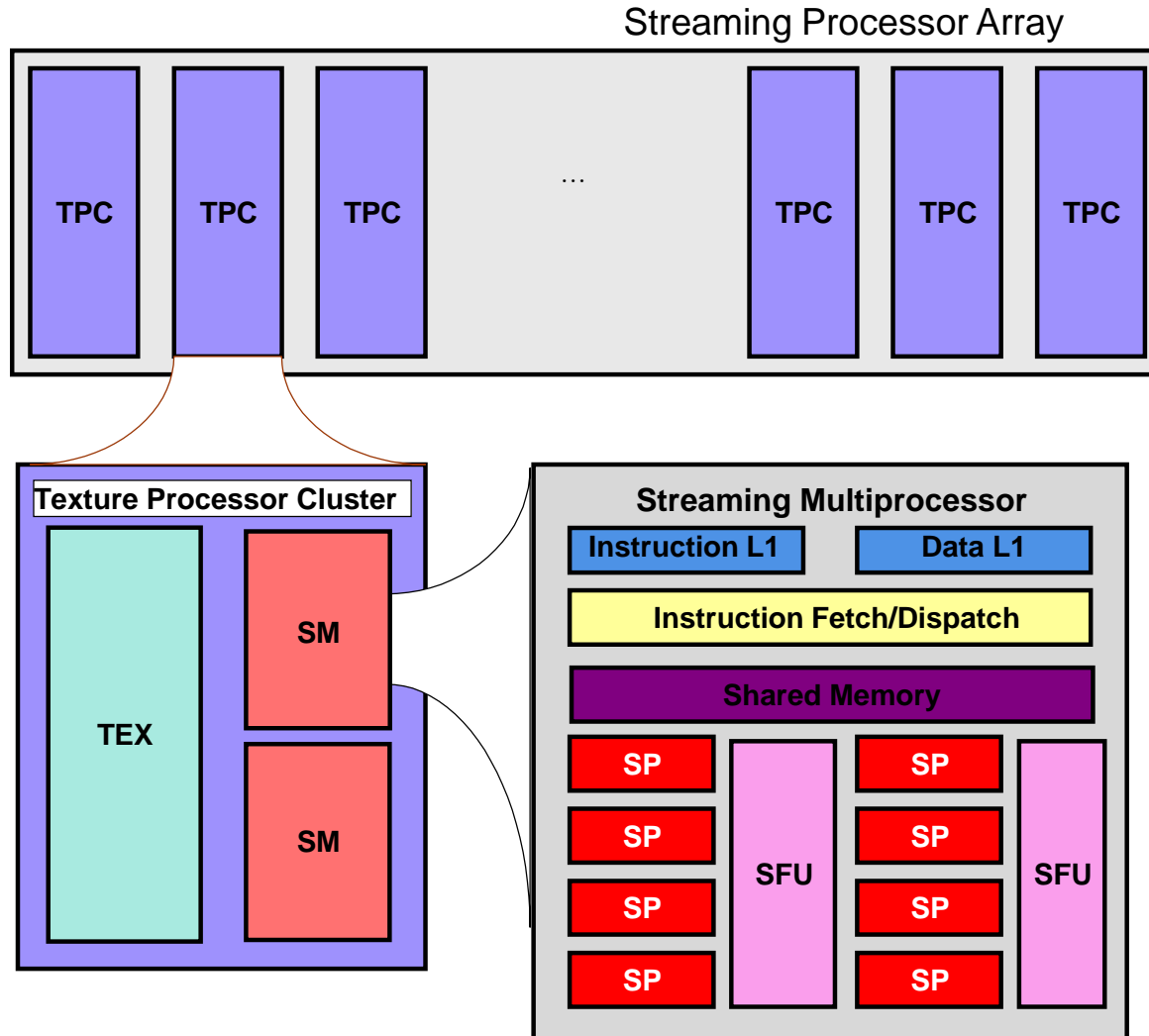
- Programmer declares (Thread) Block:
 - Block size 1 to 512 concurrent threads
 - Block shape 1D, 2D, or 3D
 - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have **thread id** numbers within Block
- Thread program uses **thread id** to select work and address shared data

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

GeForce-8 Series HW Overview

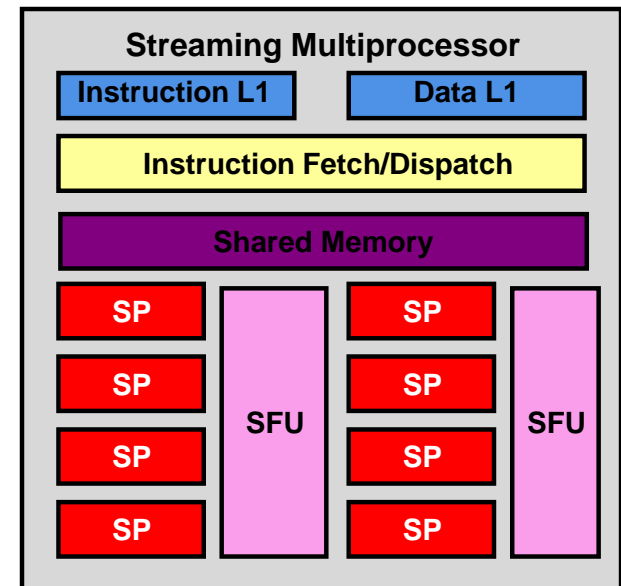


CUDA Processor Terminology

- SPA
 - Streaming Processor Array
- TPC
 - Texture Processor Cluster (2 or more SM + TEX)
- SM
 - Streaming Multiprocessor (8 or more SP)
 - Multi-threaded processor core
 - Fundamental processing unit for CUDA thread block
- SP
 - Streaming Processor
 - Scalar ALU for a single CUDA thread

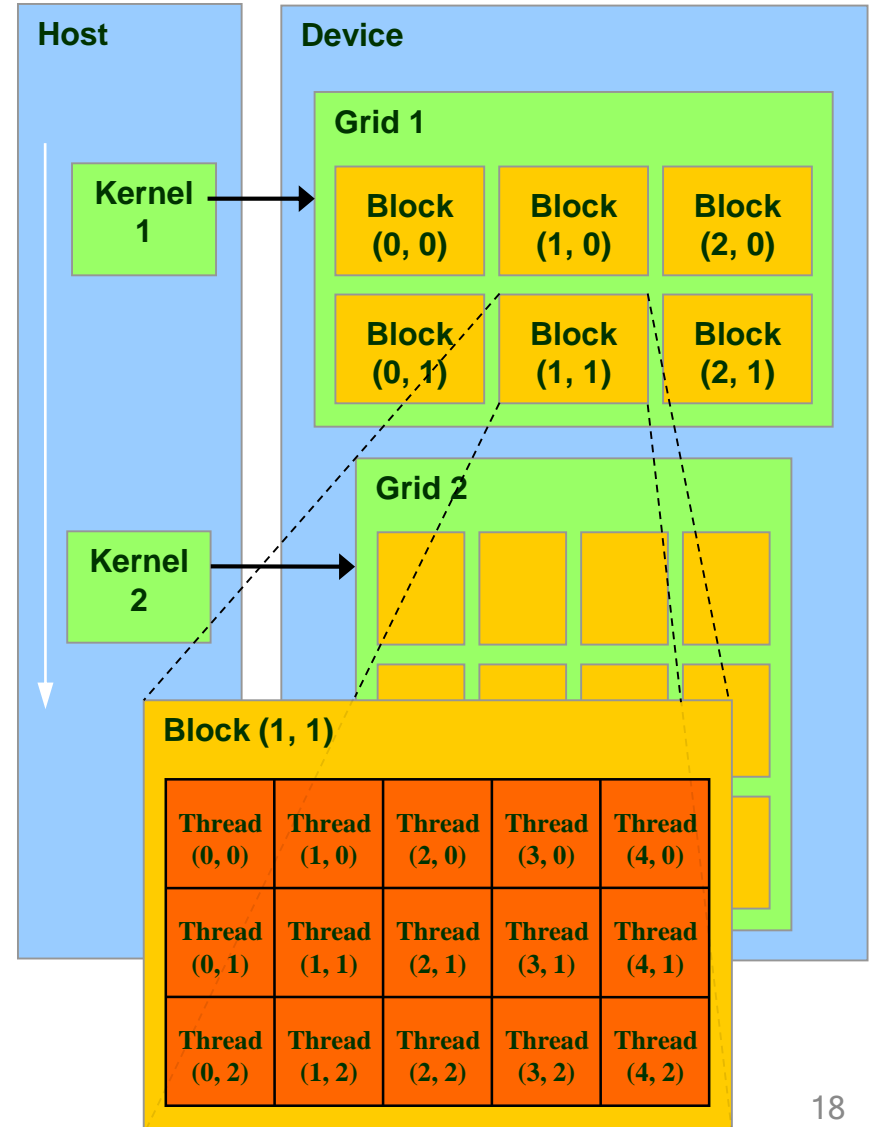
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
 - 8 Streaming Processors (SP)
 - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
 - 1 to 512 threads active
 - Shared instruction fetch per 32 threads
 - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- texture and global memory access

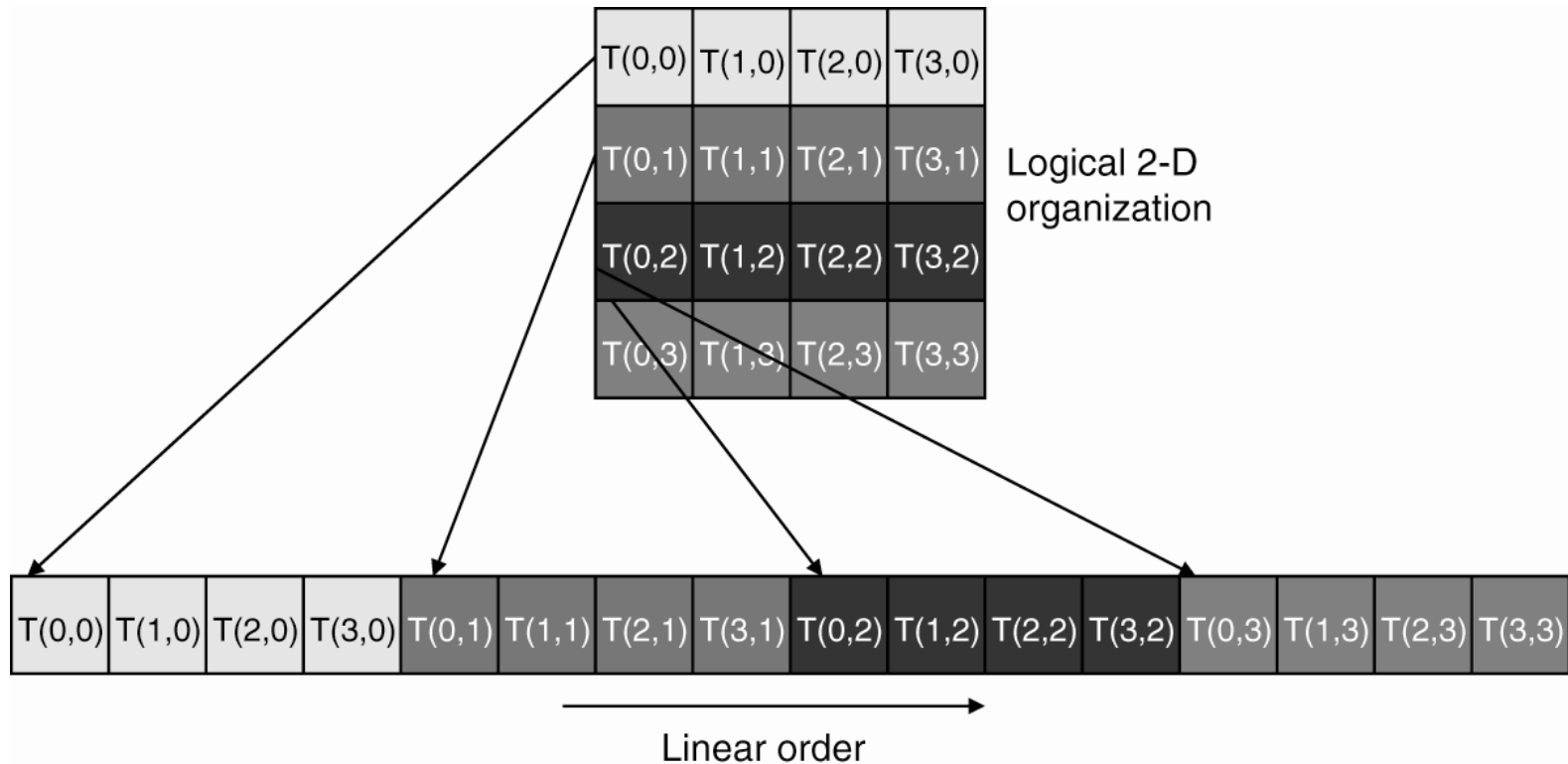


Thread Lifecycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM's
 - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
 - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
 - SPA can distribute more Thread Blocks

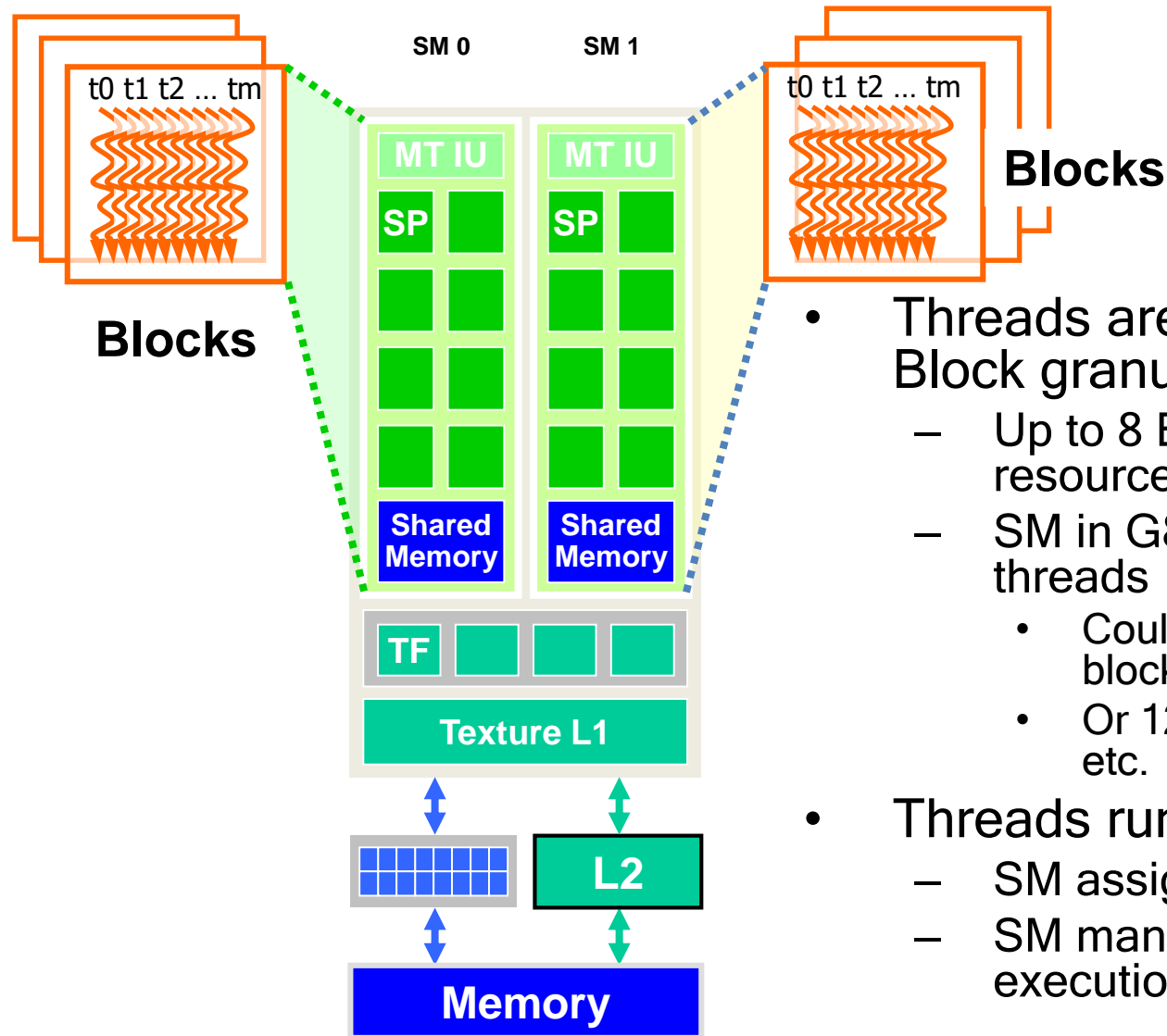


Threads in Linear Order



- If the block was 3D, we would start with threads whose `threadIdx.z=0`, then `threadIdx.z=1`, etc.

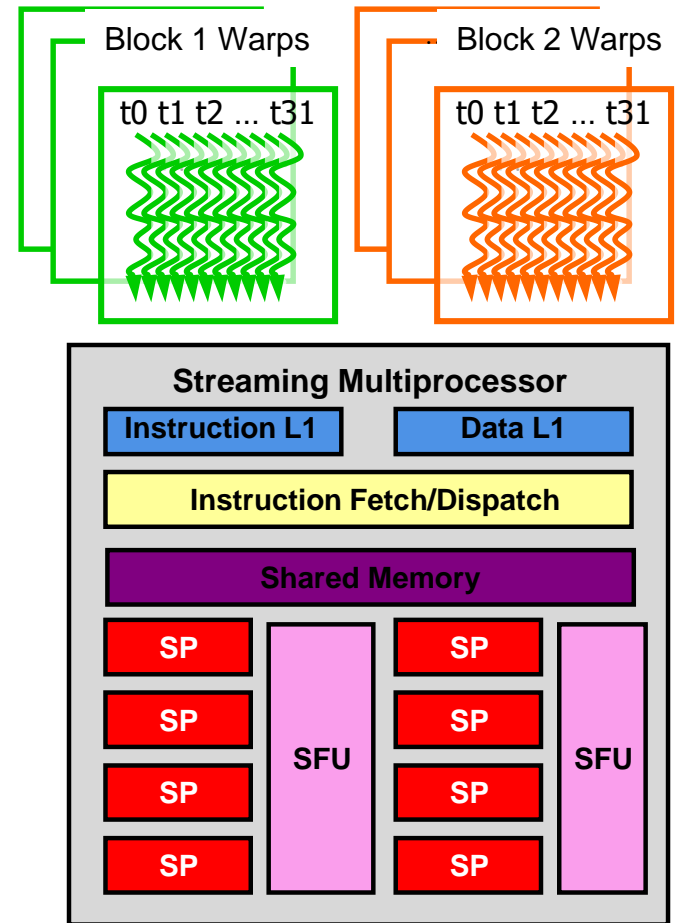
SM Executes Blocks



- Threads are assigned to SMs in Block granularity
 - Up to 8 Blocks to each SM as resource allows
 - SM in G80 can take up to 768 threads
 - Could be $256 \text{ (threads/block)} * 3 \text{ blocks}$
 - Or $128 \text{ (threads/block)} * 6 \text{ blocks, etc.}$
- Threads run concurrently
 - SM assigns/maintains thread id #s
 - SM manages/schedules thread execution

Thread Scheduling/Execution

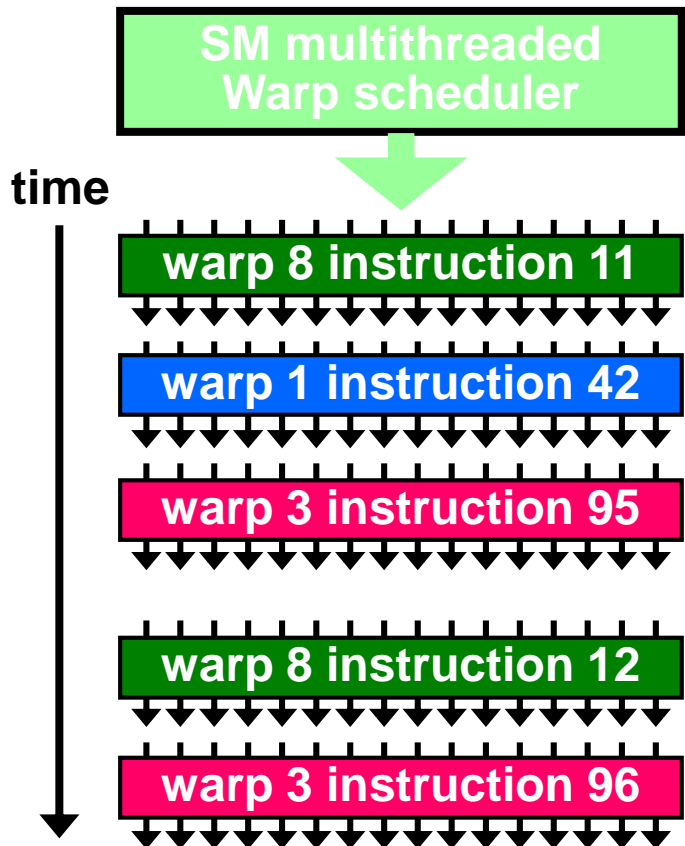
- Each Thread Blocks is divided in 32-thread Warps
 - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ Warps
 - There are $8 * 3 = 24$ Warps
 - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



SM Warp Scheduling



- SM hardware implements 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 when selected



4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80

- If one global memory access is needed for every 4 instructions
- A minimum of 13 Warps are needed to fully tolerate 200-cycle memory latency

SM Instruction Buffer - Warp Scheduling

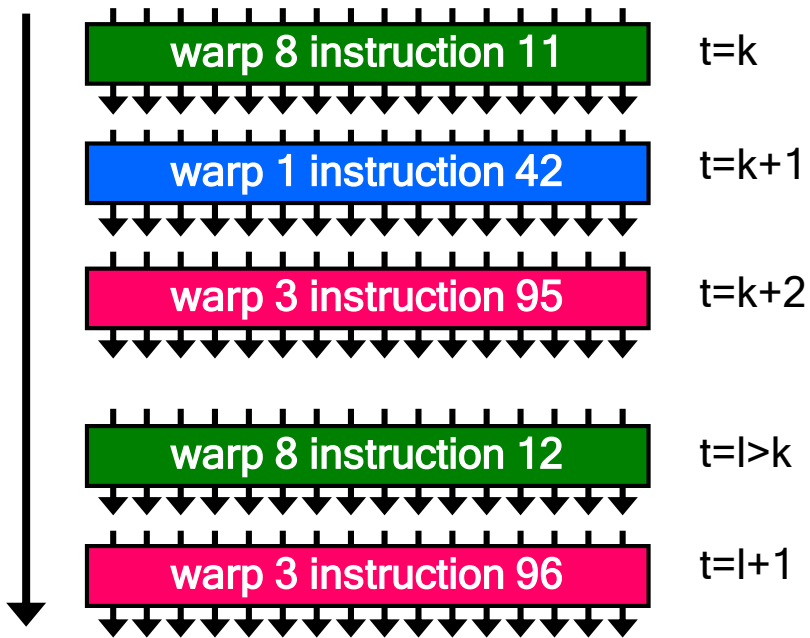
- Fetch one warp instruction/cycle
 - from instruction L1 cache
 - into any instruction buffer slot
- Issue one “ready-to-go” warp instruction/cycle
 - from any warp - instruction buffer slot
 - operand **scoreboarding** used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp

Scoreboarding

- How to determine if an instruction is ready to execute?
- A *scoreboard* is a table in hardware that tracks
 - instructions being fetched, issued, executed
 - resources (functional units and operands) they need
 - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

Scoreboarding Example

- Consider three separate instruction streams: **warp1**, **warp3** and **warp8**

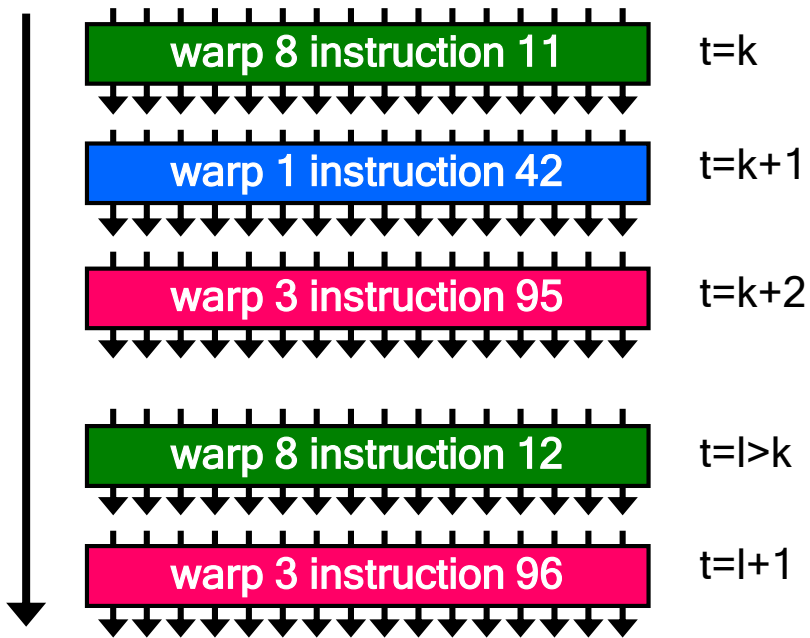


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
...		

Schedule at time k
←

Scoreboarding Example

- Consider three separate instruction streams: **warp1**, **warp3** and **warp8**

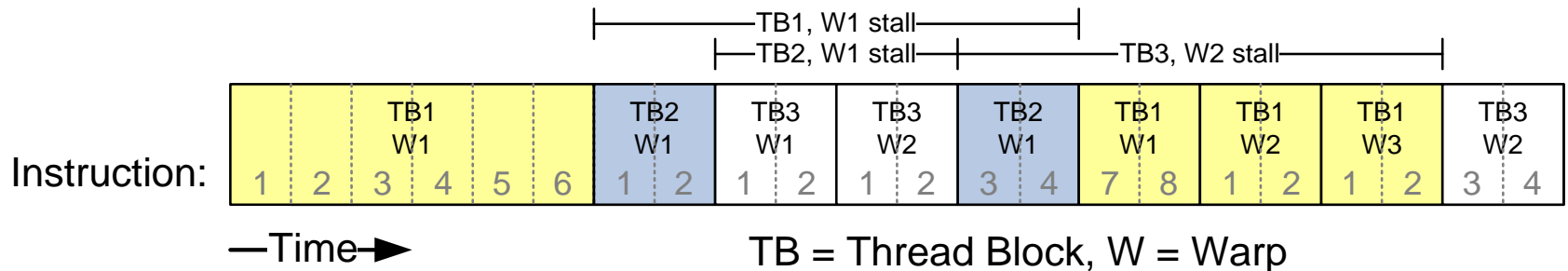


Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
...		

Schedule at time $k+1$ ←

Scoreboarding

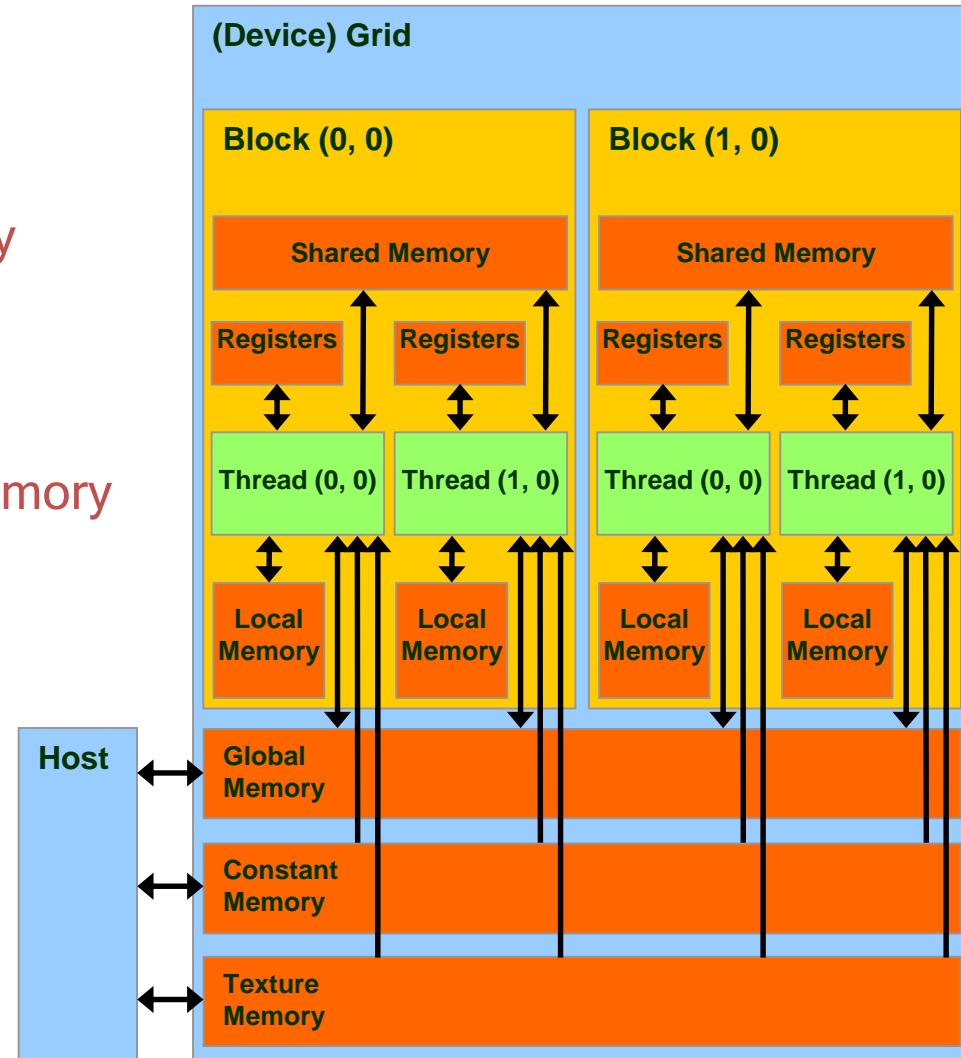
- All register operands of all instructions in the Instruction Buffer are scoreboardd
 - Status becomes ready after the needed values are deposited
 - prevents hazards
 - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
 - any thread can continue to issue instructions until scoreboarding prevents issue
 - allows Memory/Processor ops to proceed in shadow of other waiting Memory/Processor ops



Memory Hardware

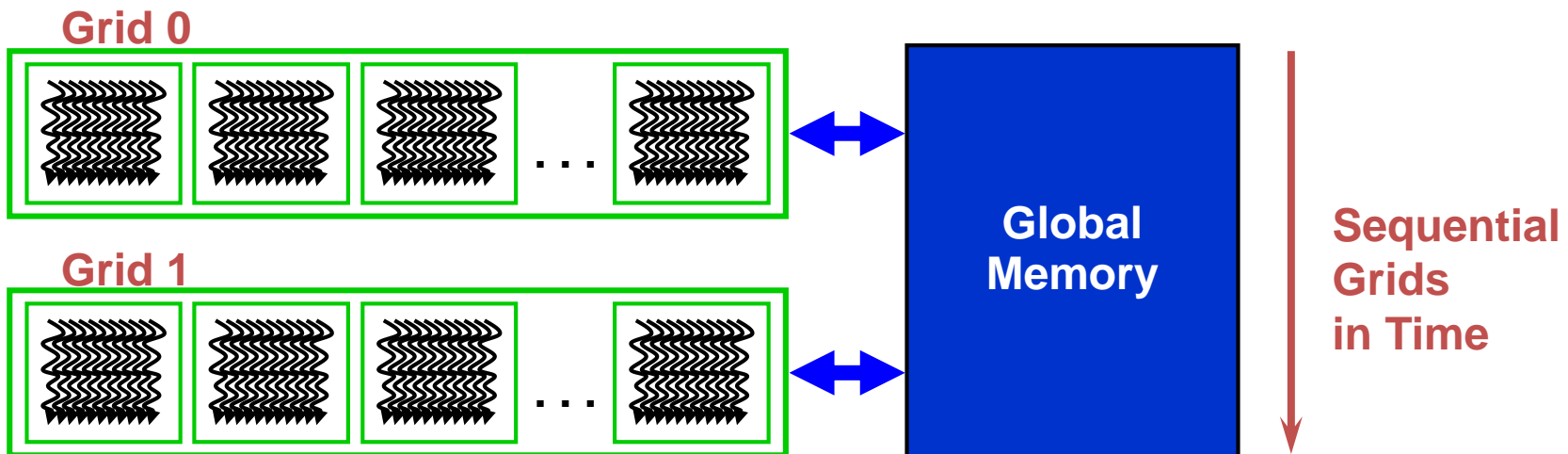
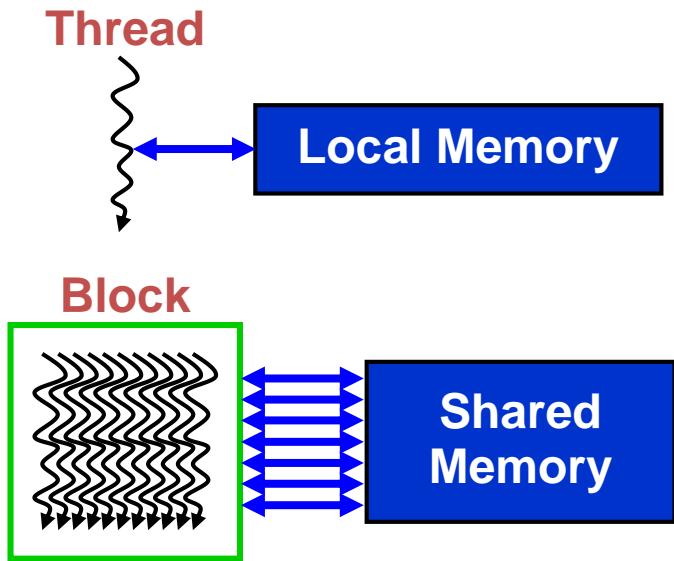
CUDA Device Memory Space: Review

- Each thread can:
 - R/W per-thread **registers**
 - R/W per-thread **local memory**
 - R/W per-block **shared memory**
 - R/W per-grid **global memory**
 - Read only per-grid **constant memory**
 - Read only per-grid **texture memory**
- The host can R/W **global, constant, and texture memories**

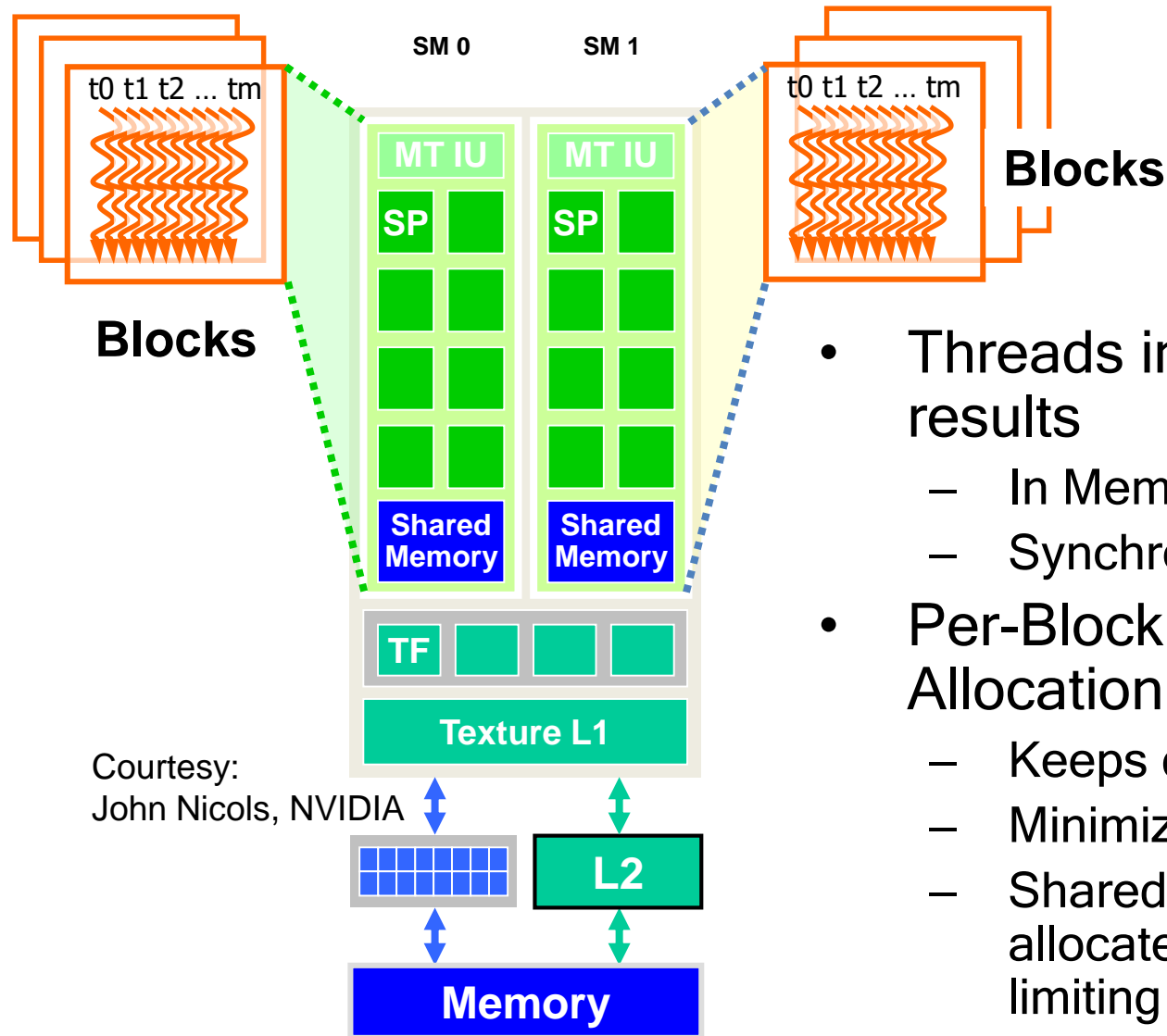


Parallel Memory Sharing

- Local Memory: per-thread
 - Private per thread
 - Auto variables, register spill
- Shared Memory: per-Block
 - Shared by threads of the same block
 - Inter-thread communication
- Global Memory: per-application
 - Shared by all threads
 - Inter-Grid communication



SM Memory Architecture



- Threads in a block share data & results
 - In Memory and Shared Memory
 - Synchronize at barrier instruction
- Per-Block Shared Memory Allocation
 - Keeps data close to processor
 - Minimize trips to global Memory
 - Shared Memory is dynamically allocated to blocks, one of the limiting resources

Courtesy:
John Nicols, NVIDIA

Texture Memory

- Read only
- More closely related to graphics pipeline
- Small, but can be faster than global memory due to cache
 - More relaxed coalescing requirements
 - Optimized for 2D spatial locality
 - Can pack 4 8-bit ints into 1 float
 - Converts data to [0.0 .. 1.0] or [-1.0 .. 1.0] range
 - Automatic boundary handling

⇒ out of scope for now

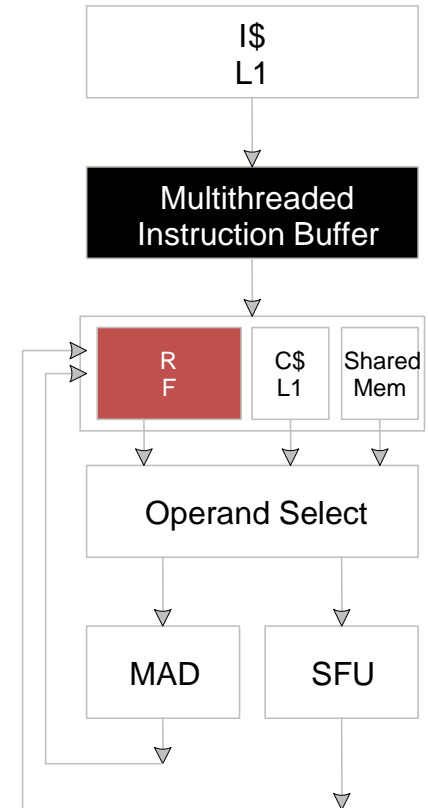
See <http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html> if interested

SM Register File

- Register File (RF)
 - 32 KB (8K entries) for each SM in G80
- TEX pipe can also read/write RF
 - 2 SMs share 1 TEX in G 80, 3 SMs per TEX in GTX 200
 - Related to graphics mode (out of scope)
- Load/Store pipe can also read/write RF

MAD: Multiply and Add unit

SFU: Super Function Unit - where more complex instructions are executed

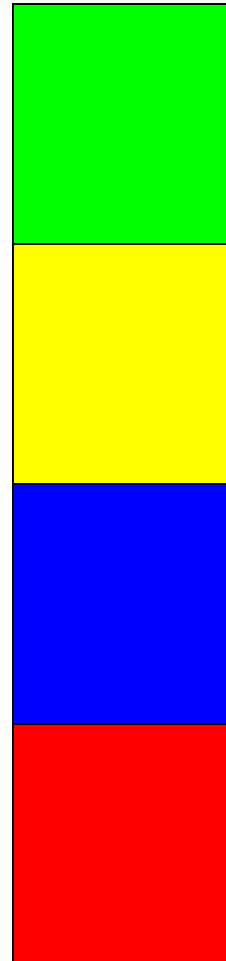


Programmer View of Register File

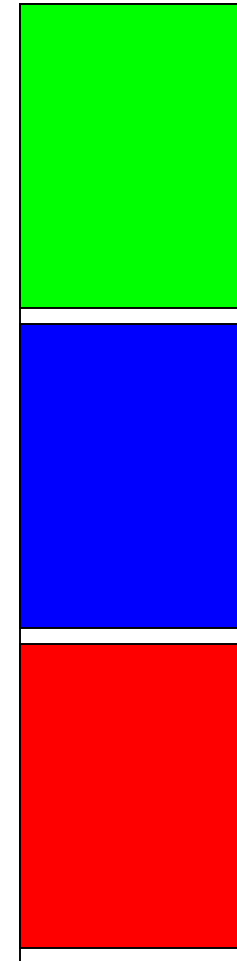
- There are 8192 registers in each SM in G80
 - This is an implementation decision, not part of CUDA
 - Registers are dynamically partitioned across all blocks assigned to the SM
 - Once assigned to a block, the register is NOT accessible by threads in other blocks
 - Each thread in the same block only access registers assigned to itself

(This has changed but the example is still useful)

4 blocks



3 blocks



Matrix Multiplication Example

- If each Block has 16X16 threads and each thread uses 10 registers, how many threads can run on each SM?
 - Each block requires $10 \times 256 = 2560$ registers
 - $8192 = 3 * 2560 + \text{change}$
 - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
 - Each Block now requires $11 \times 256 = 2816$ registers
 - $8192 < 2816 * 3$
 - Only two Blocks can run on an SM, **1/3 reduction of parallelism!!!**

More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
 - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
 - This allows for finer grain threading than traditional CPU threading models
 - The compiler can tradeoff between **instruction-level parallelism** and **thread level parallelism**

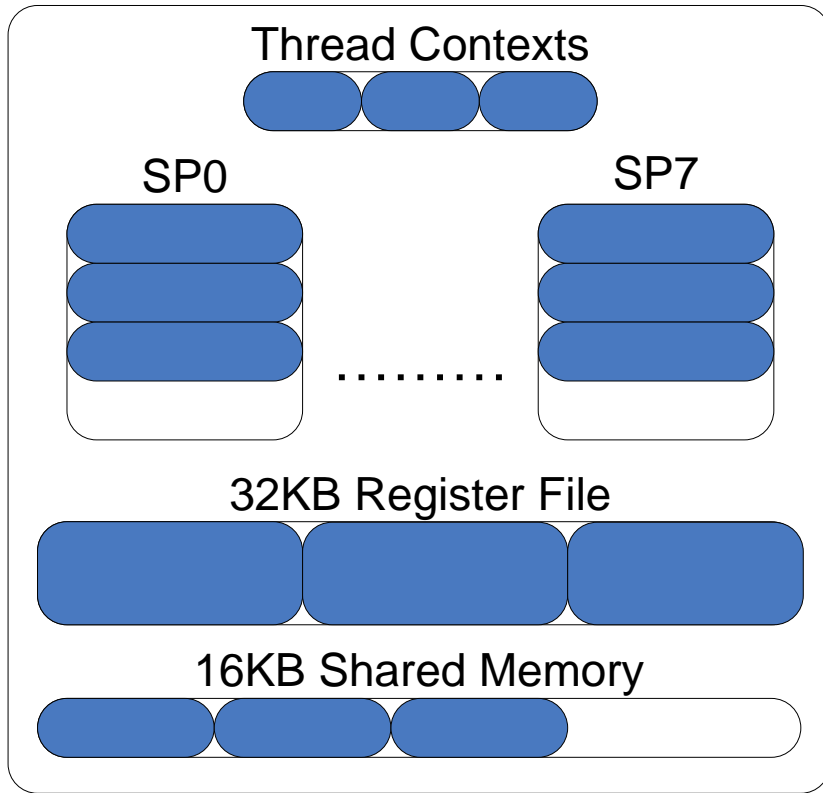
ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads take 200 cycles
 - 3 Blocks can run on each SM
- If a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
 - Only two can run on each SM
 - However, one only needs $200/(8*4) = 7$ Warps to tolerate the memory latency
 - Two blocks have 16 Warps. The performance can be actually higher!

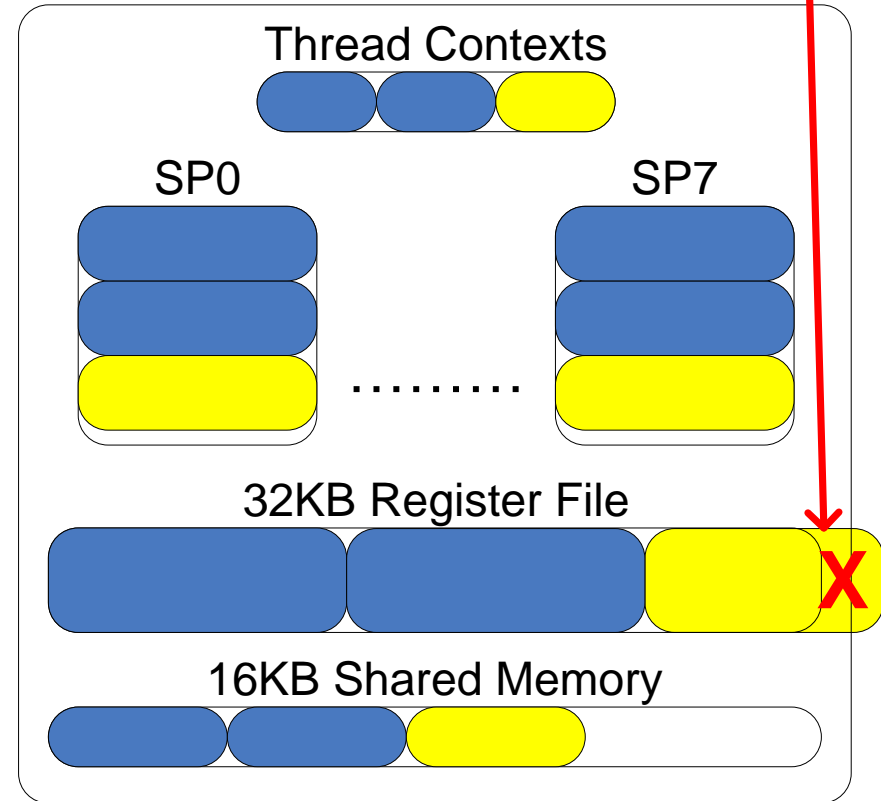
Resource Allocation Example

TB0 TB1 TB2

Insufficient registers to allocate 3 blocks



(a) Pre-“optimization”



(b) Post-“optimization”

Increase in per-thread performance, but fewer threads:
Lower overall performance in this case

CUDA Occupancy Calculator

The screenshot shows an Excel spreadsheet titled "CUDA_Occupancy_calculator.xlsm". The spreadsheet is divided into several sections:

- 3.) GPU Occupancy Data is displayed here and in the graphs:**

Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	50%
- Physical Limits for GPU Compute Capability:** 1.3

Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2
- Allocation Per Thread Block**

Warps	4
Registers	3584
Shared Memory	1024
- Maximum Thread Blocks Per Multiprocessor** (Blocks)

Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16
- Thread Block Limit Per Multiprocessor highlighted** RED

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Memory Layout of a Matrix in C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

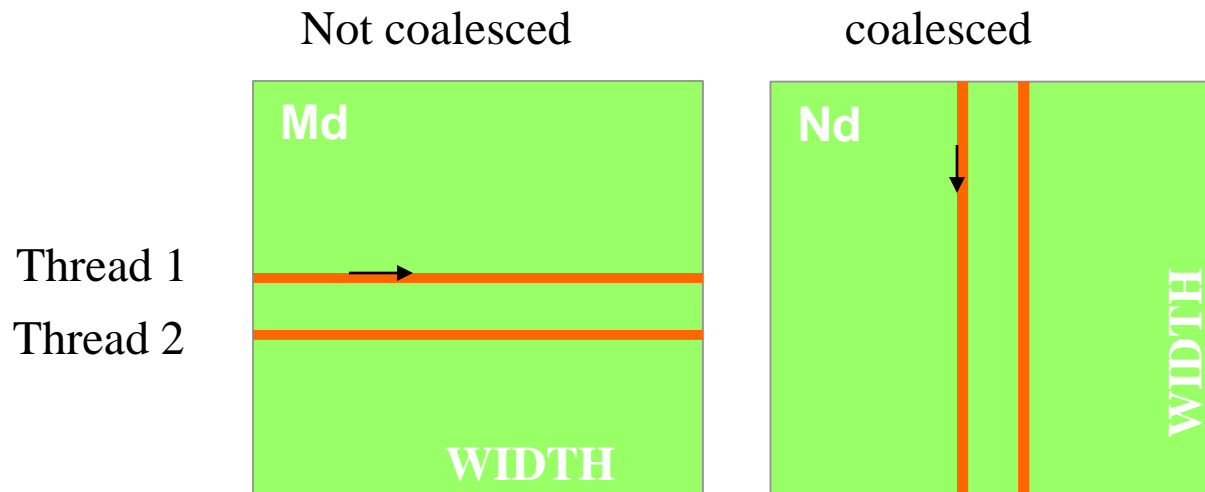
M



$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

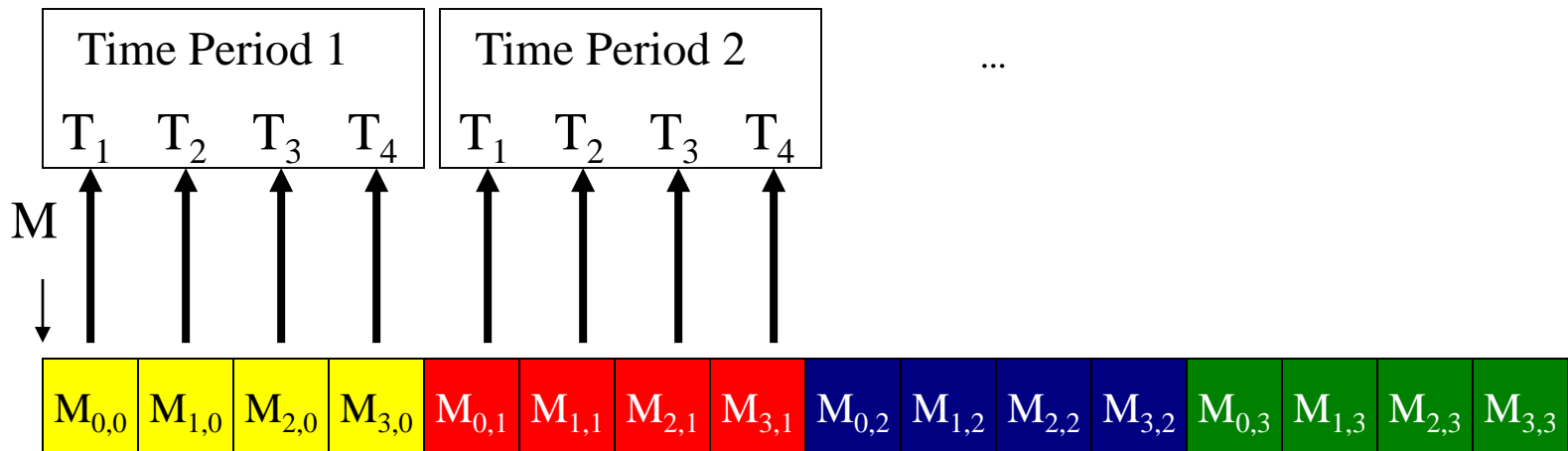
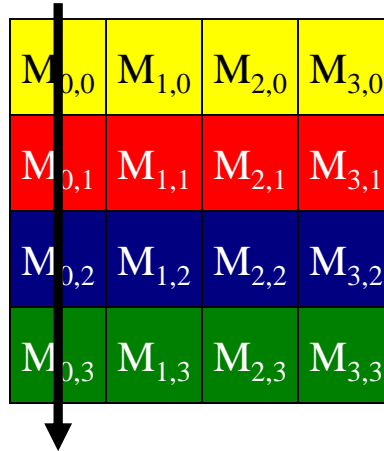
Memory Coalescing*

- When accessing global memory, peak performance utilization occurs when all threads in a half warp access continuous memory locations

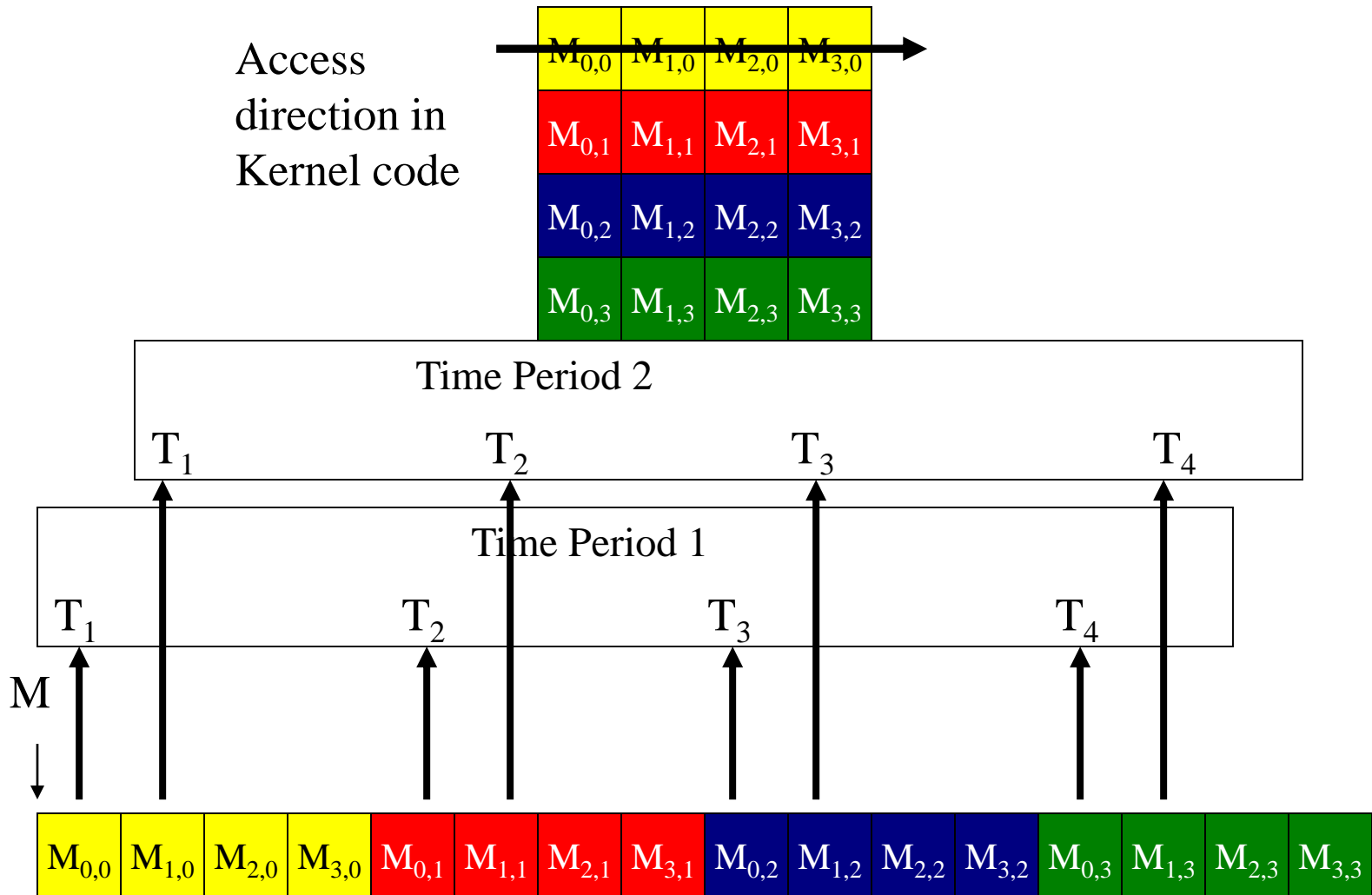


Memory Layout of a Matrix in C

Access
direction in
Kernel code



Memory Layout of a Matrix in C



Matrix Multiplication

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.     Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
14.     __syncthreads();
    }
15. Pd[Row*Width + Col] = Pvalue;
}
```

Why this works:

- threads in warp have same ty
- adjacent threads read adjacent elements from memory

* Coalescing since 2013

- GPUs now have cache
- => Coalescing is less important as it is done by the hardware
- Make sure you have enough cache available for each warp
- There may still be some loss of performance (20-50%) due to uncoalesced access

Cache (Compute Capability 3.x)

- L1 cache for each multiprocessor
- L2 cache shared by all multiprocessors
- Both are used to cache accesses to local or global memory, including temporary register spills
- Cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured

Configuring the Cache

- The same on-chip memory is used for both L1 and shared memory. It can be configured as:
 - 48 KB of shared memory and 16 KB of L1 cache
 - 16 KB of shared memory and 48 KB of L1 cache
 - 32 KB of shared memory and 32 KB of L1 cache
- **using** `cudaFuncSetCacheConfig()`

Cache Preferences

```
// Host code

// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferEqual: shared memory is 32 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel,
    cudaFuncCachePreferShared);
```


Cache Preferences

- The default cache configuration is "prefer none"
- If a kernel has no preference, then it will default to the preference of the current CPU thread/context
- If the current thread/context also has no preference, then most recent cache configuration will be used
 - unless a different cache configuration is required to launch the kernel (e.g., due to shared memory requirements)
- The initial configuration is 48 KB of shared memory and 16 KB of L1 cache

Constants

- Immediate address constants (`#define`)
- Indexed address constants
- Constants stored in DRAM, and cached on chip
 - L1 per SM
- A constant value can be broadcast to all threads in a warp
 - Extremely efficient way of accessing a value that is common for all threads in a block!

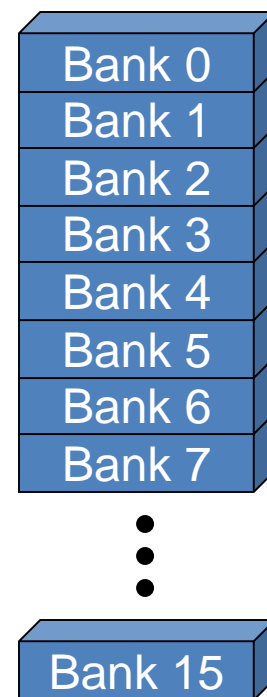
```
// specify as global variable
__device__ __constant__ float gpuGamma[2];
// copy gamma value to constant device memory
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));
...
// access as global variable in kernel
res = gpuGamma[0] * threadIdx.x;
```

Shared Memory

- Each SM has 16 or more KB of Shared Memory
 - 16 banks of 32-bit words
 - 64-bit access is also supported now
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
 - read and write access

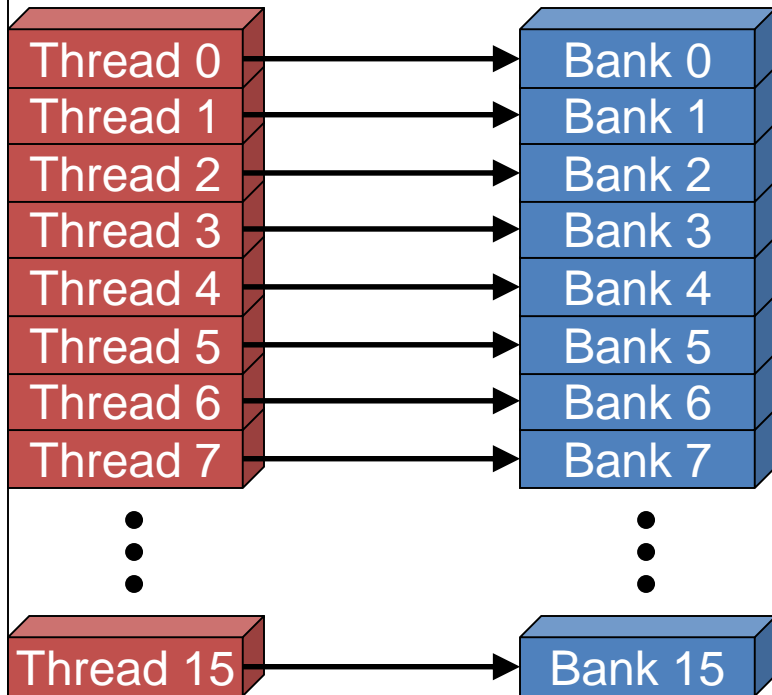
Parallel Memory Architecture

- In a parallel machine, many threads access memory
 - Therefore, memory is divided into **banks**
 - Essential to achieve high bandwidth
- Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a **bank conflict**
 - Conflicting accesses are serialized

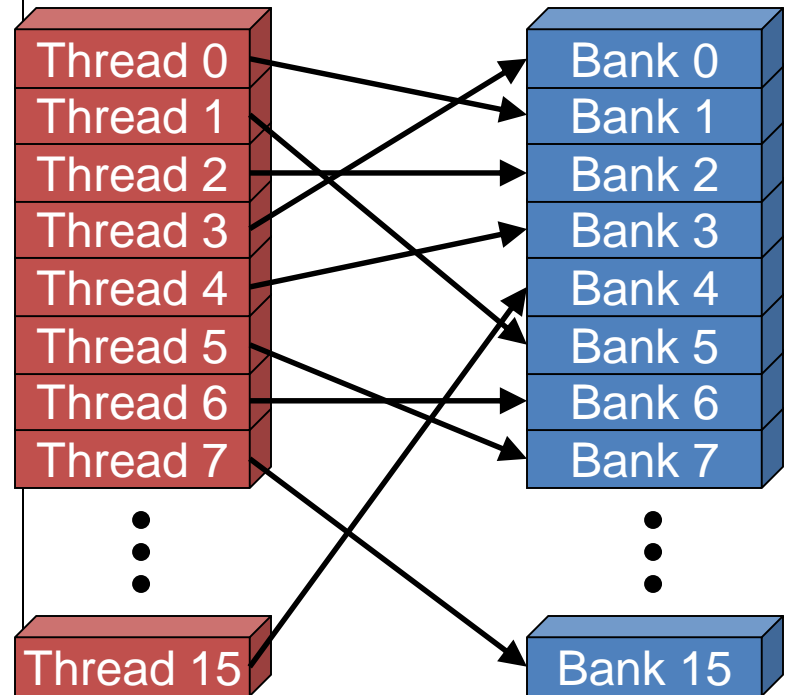


Bank Addressing Examples

- No Bank Conflicts
 - Linear addressing stride == 1

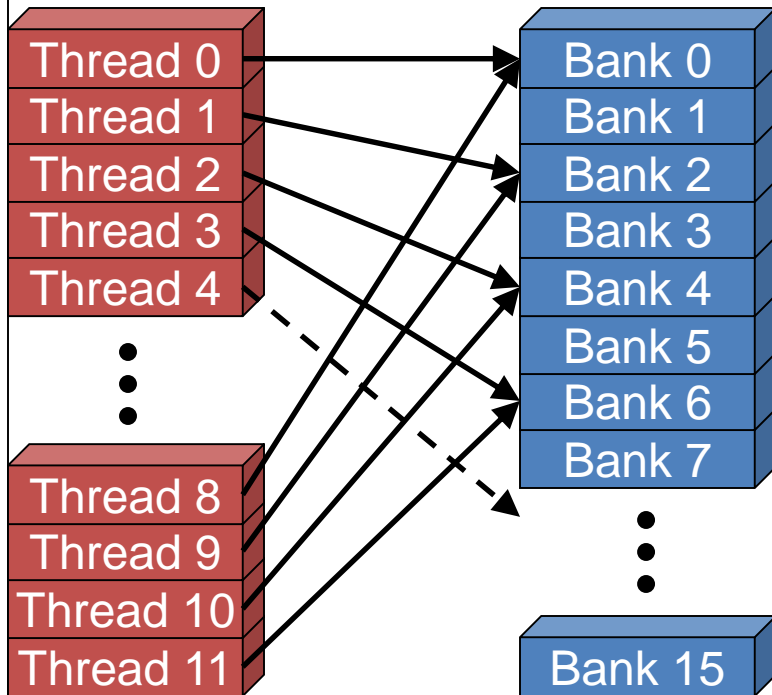


- No Bank Conflicts
 - Random 1:1 Permutation

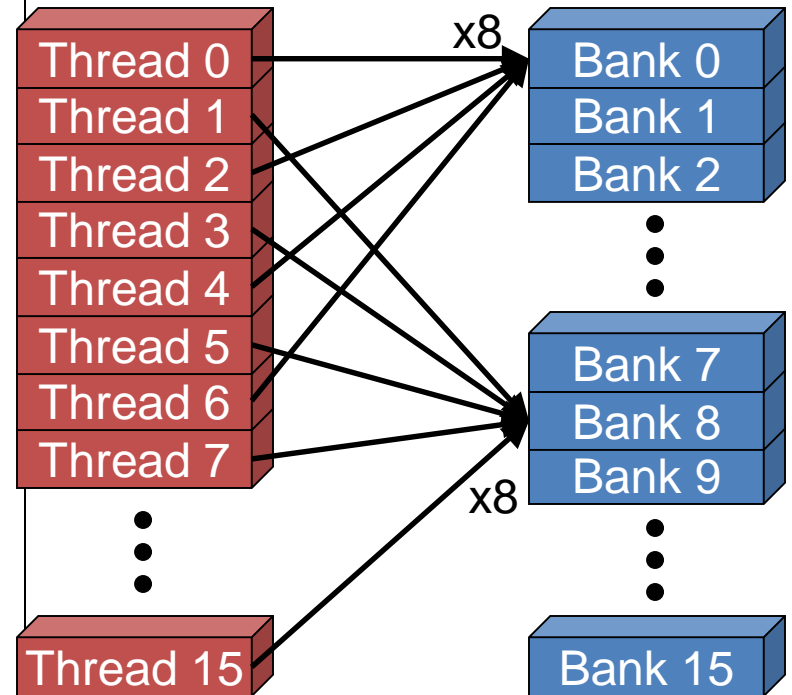


Bank Addressing Examples

- 2-way Bank Conflicts



- 8-way Bank Conflicts



How Addresses Map to Banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
 - So $\text{bank} = \text{address} \% 16$
 - Same as the size of a half-warp
 - No bank conflicts between different half-warps, only within a single half-warp

Shared Memory Bank Conflicts

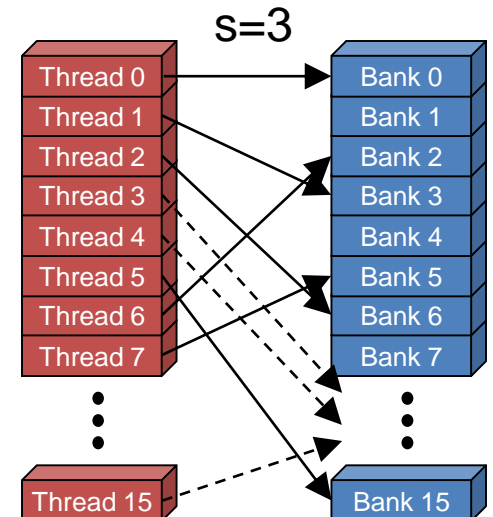
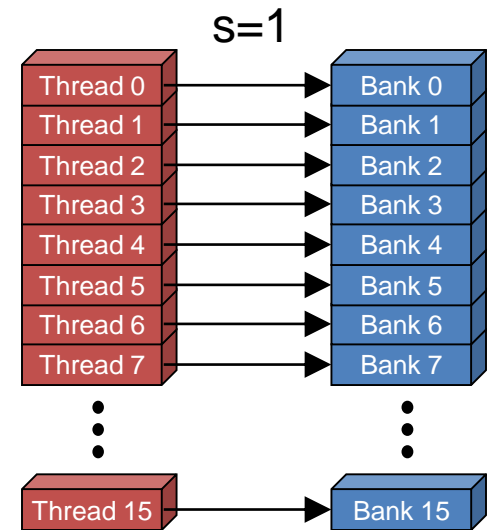
- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
 - If all threads of a half-warp access different banks, there is no bank conflict
 - If all threads of a half-warp access an identical address, there is no bank conflict (broadcast)
- The slow case:
 - Bank Conflict: multiple threads in the same half-warp access the same bank
 - Must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank

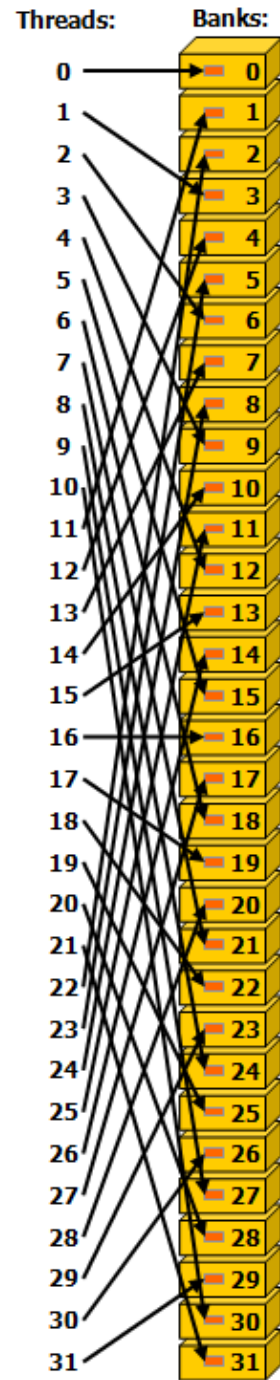
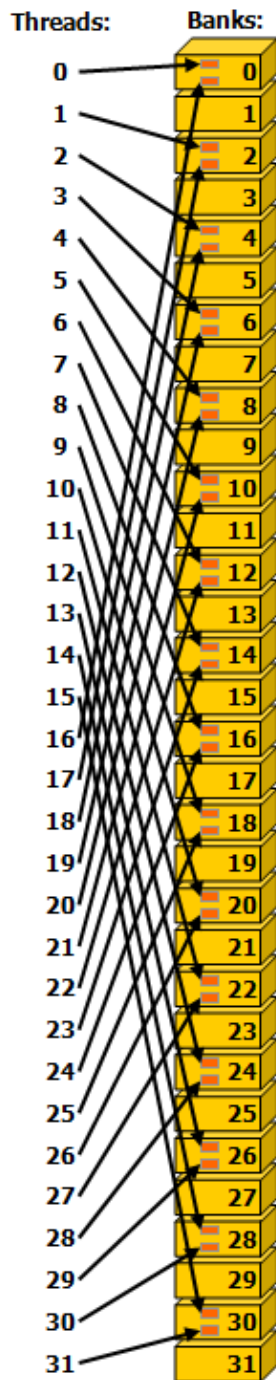
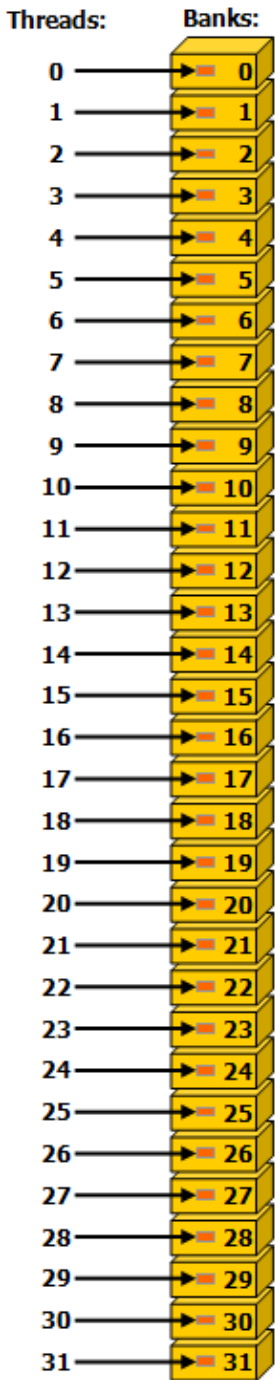
Linear Addressing

- Given:

```
__shared__ float shared[256];  
float foo =  
    shared[baseIndex + s * threadIdx.x];
```

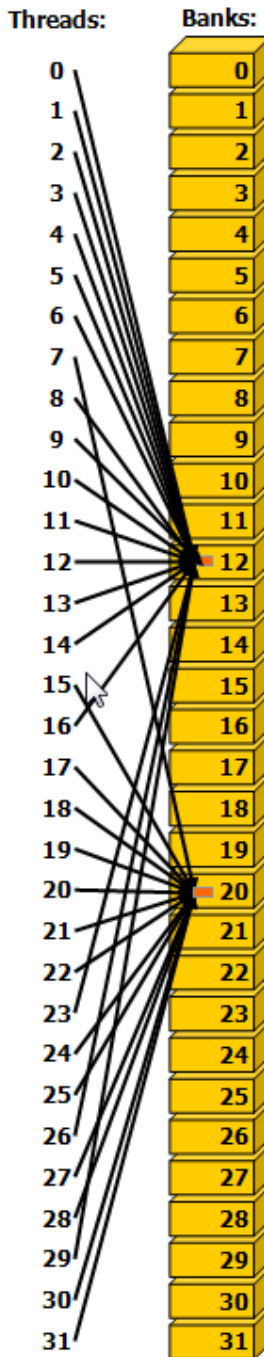
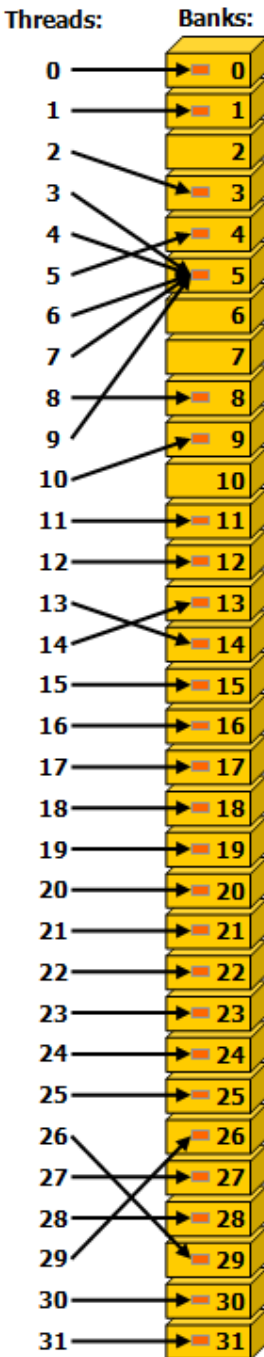
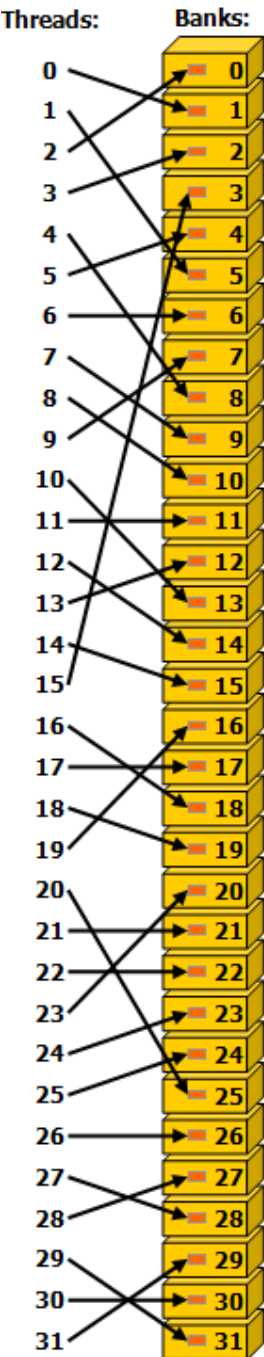
- This is only bank-conflict-free if s shares no common factors with the number of banks
 - 16 on G80, so s must be **odd**





Compute Capability 3.x

- 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 (no bank conflict)
- Right: Linear addressing with a stride of three 32-bit words (no bank conflict)
- More flexible definition of alignment within banks enables last two examples



Compute Capability 3.x

- Left: Conflict-free access via random permutation
- Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5
- Right: Conflict-free broadcast access (threads access the same word within a bank)

Control Flow

Control Flow Instructions

- Main performance concern with branching is divergence
 - Threads within a single warp take different paths
 - Different execution paths are serialized on GPU
 - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
 - Example with divergence:
 - `If (threadIdx.x > 2) { }`
 - This creates two different control paths for threads in a block
 - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
 - Example without divergence:
 - `If (threadIdx.x / WARP_SIZE > 2) { }`
 - Also creates two different control paths for threads in a block
 - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel
- Examples
 - Sum reduction: sum of all values in the array
 - Max reduction: maximum of all values in the array
- Typically parallel implementation:
 - Recursively halve # threads, add two values per thread
 - Takes $\log(n)$ steps for n elements, requires $n/2$ threads

A Vector Reduction Example

- Assume an in-place reduction using shared memory
 - The original vector is in device global memory
 - The shared memory is used to hold a partial sum vector
 - Each iteration brings the partial sum vector closer to the final sum
 - The final solution will be in element 0

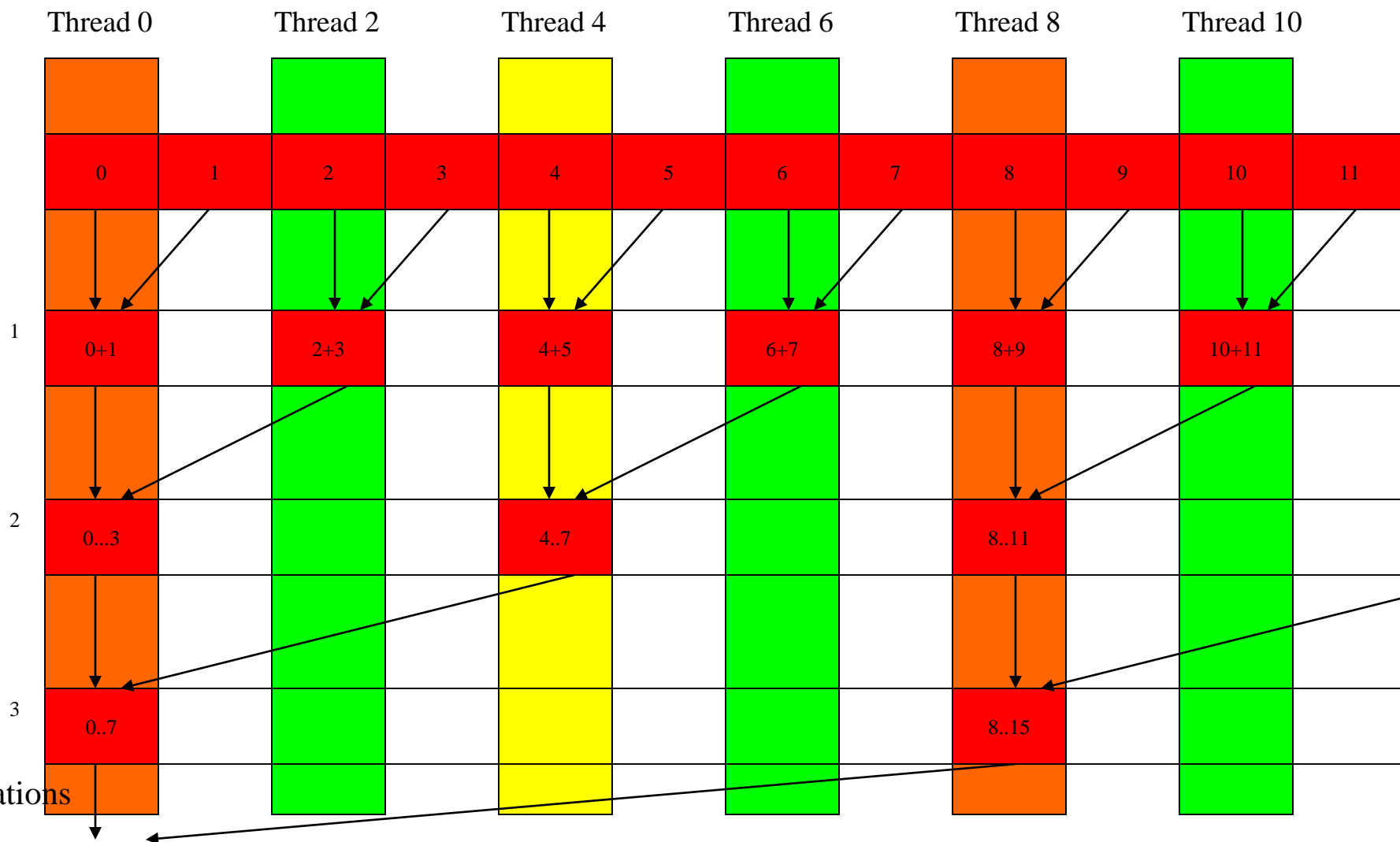
A simple implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[]

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


Vector Reduction with Branch Divergence



Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
 - Threads that perform addition and threads that do not
 - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence
- No more than half of threads will be executing at any time
 - All odd index threads are disabled right from the beginning!
 - On average, less than $\frac{1}{4}$ of the threads will be activated for all warps over time.
 - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence
 - This can go on for a while, up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire

Shortcomings of the implementation

- Assume we have already loaded array into

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

BAD: Divergence due to interleaved branch decisions

A better implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[]
```

```
unsigned int t = threadIdx.x;
```

```
for (unsigned int stride = blockDim.x/2;  
     stride > 0;  stride >>= 1)
```

```
{
```

```
    __syncthreads();
```

```
    if (t < stride)
```

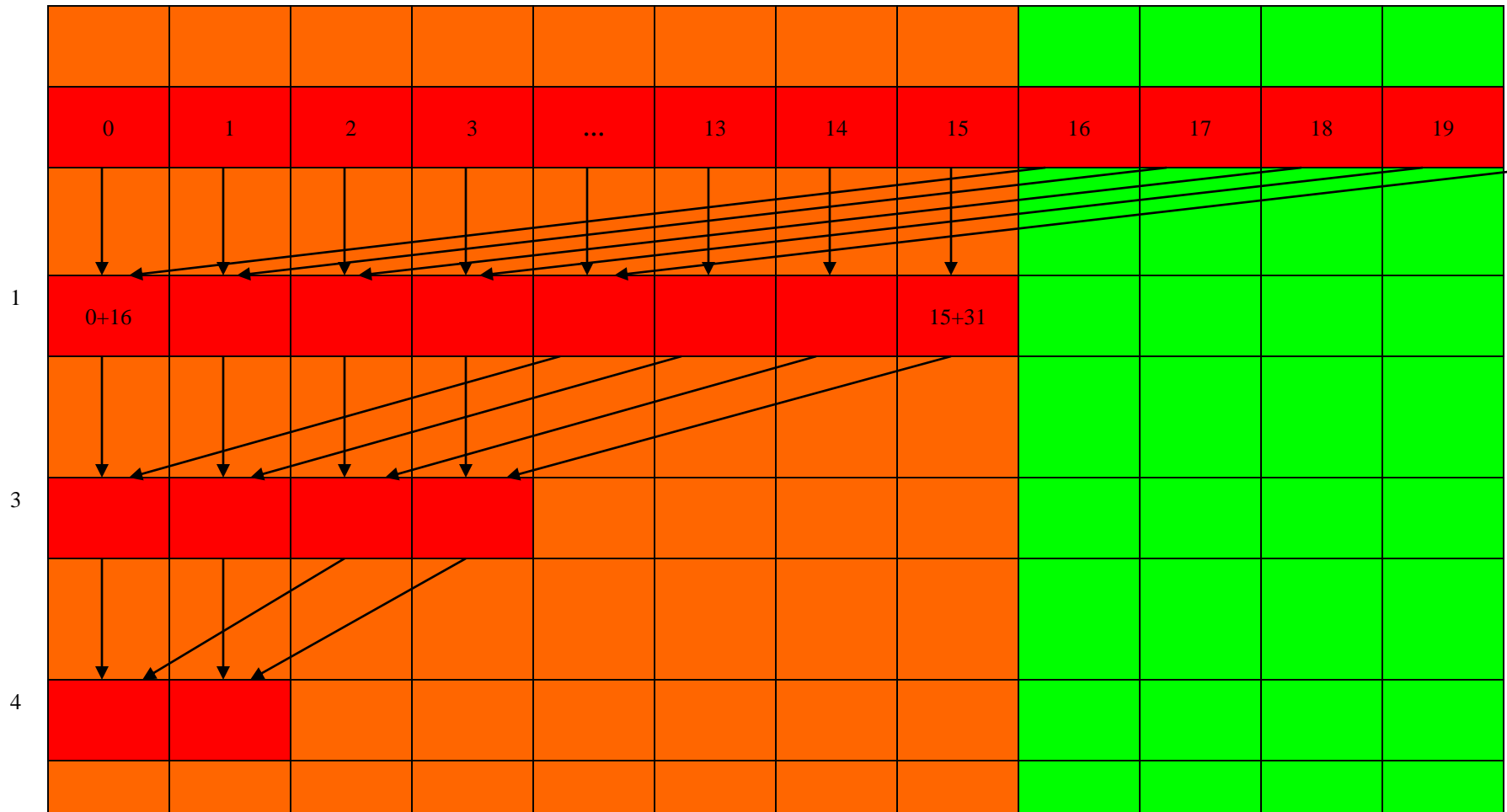
```
        partialSum[t] += partialSum[t+stride];
```

```
}
```

No Divergence until ≤ 16 sub-sums

Thread 0 Thread 1 Thread 2

Thread 14 Thread 15



Prefetching and Instruction Mix

Prefetching

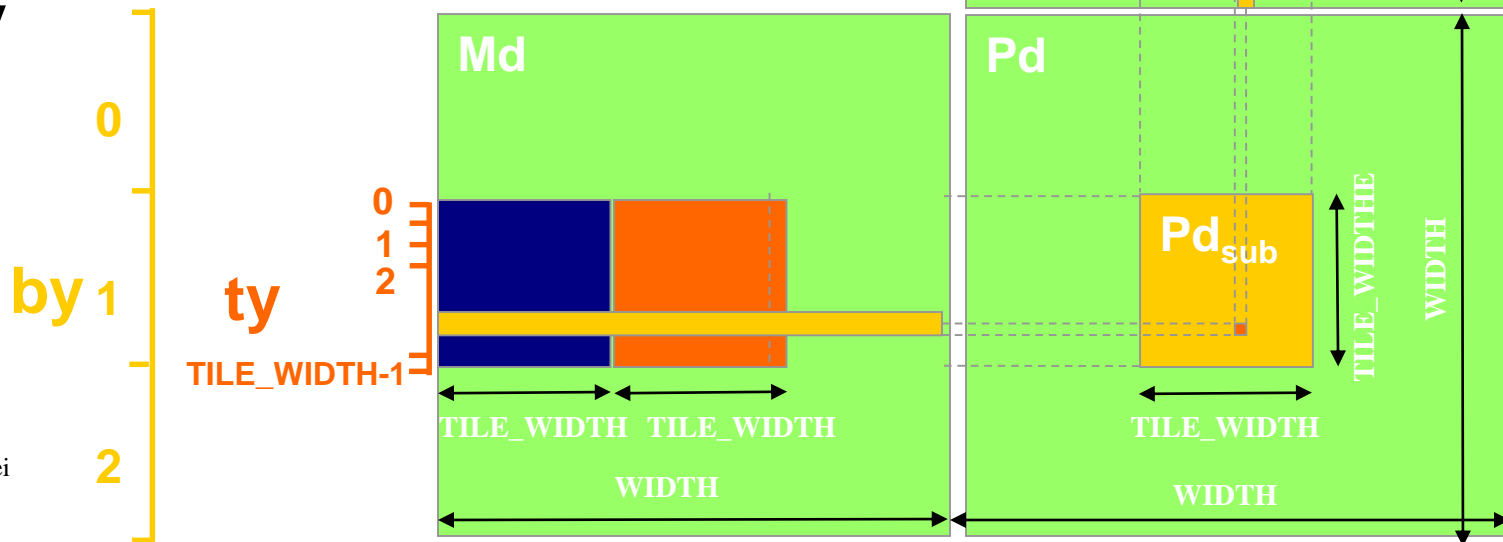
- One could double buffer the computation, getting better instruction mix within each thread
 - This is classic software pipelining in ILP compilers

```
Loop {  
  
Load current tile to shared memory  
  
syncthreads()  
  
Compute current tile  
  
syncthreads()  
}
```

```
Load next tile from global memory  
  
Loop {  
Deposit current tile to shared memory  
syncthreads()  
  
Load next tile from global memory  
  
Compute current tile  
  
syncthreads()  
}
```

Prefetch

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
-



Instruction Mix Considerations

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

There are very few mul/add between branches and address calculation

Loop unrolling can help. (Be aware that any local arrays used after unrolling will be dumped into Local Memory)

```
Pvalue += Ms[ty][k] * Ns[k][tx] + ...
          Ms[ty][k+15] * Ns[k+15][tx];
```

Unrolling

```
Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads () ;

    // compute results for tile
    for (i = 0; i < 16; i++)
    {
        Ctemp += As[ty][i]
            * Bs[i][tx];
    }

    __syncthreads () ;
}
C[indexC] = Ctemp;
```

(b) Tiled Version

```
Ctemp = 0;
for (...) {
    __shared__ float As[16][16];
    __shared__ float Bs[16][16];

    // load input tile elements
    As[ty][tx] = A[indexA];
    Bs[ty][tx] = B[indexB];
    indexA += 16;
    indexB += 16 * widthB;
    __syncthreads () ;

    // compute results for tile
    Ctemp +=
        As[ty][0] * Bs[0][tx];
    ...
    Ctemp +=
        As[ty][15] * Bs[15][tx];

    __syncthreads () ;
}
C[indexC] = Ctemp;
```

(c) Unrolled Version