CUDA/GPU In Depth: Processor

Outline

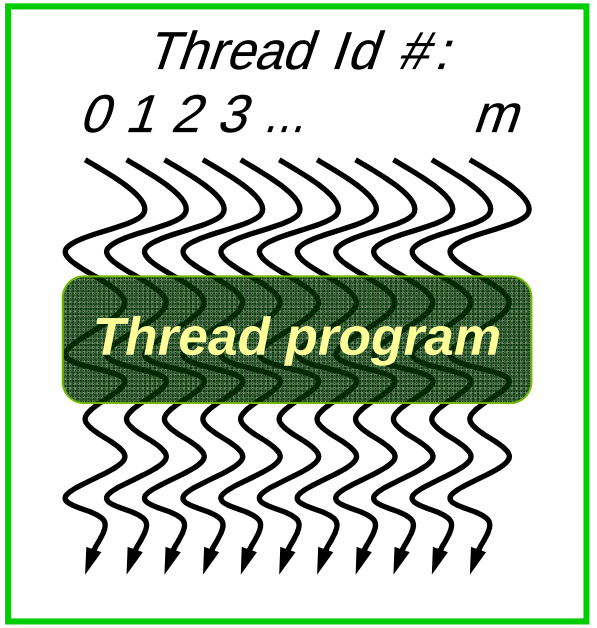
1. Thread Hardware Support and Scheduling

2. Conditionals in SIMT

CUDA Thread Block: Review

- All threads in a block execute the same kernel program (SIMT)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D where the size of the dimension is in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and to address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block

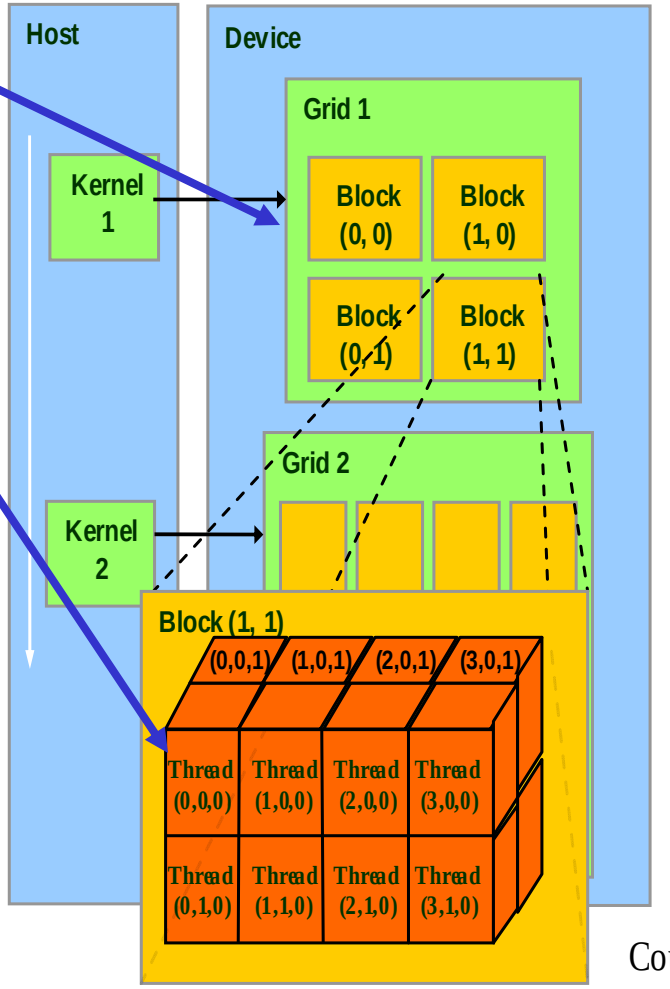


Courtesy: John Nickolls, NVIDIA

Block IDs and Thread IDs: Review

- Group **BLOCKs** into **GRIDs**
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Why? Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

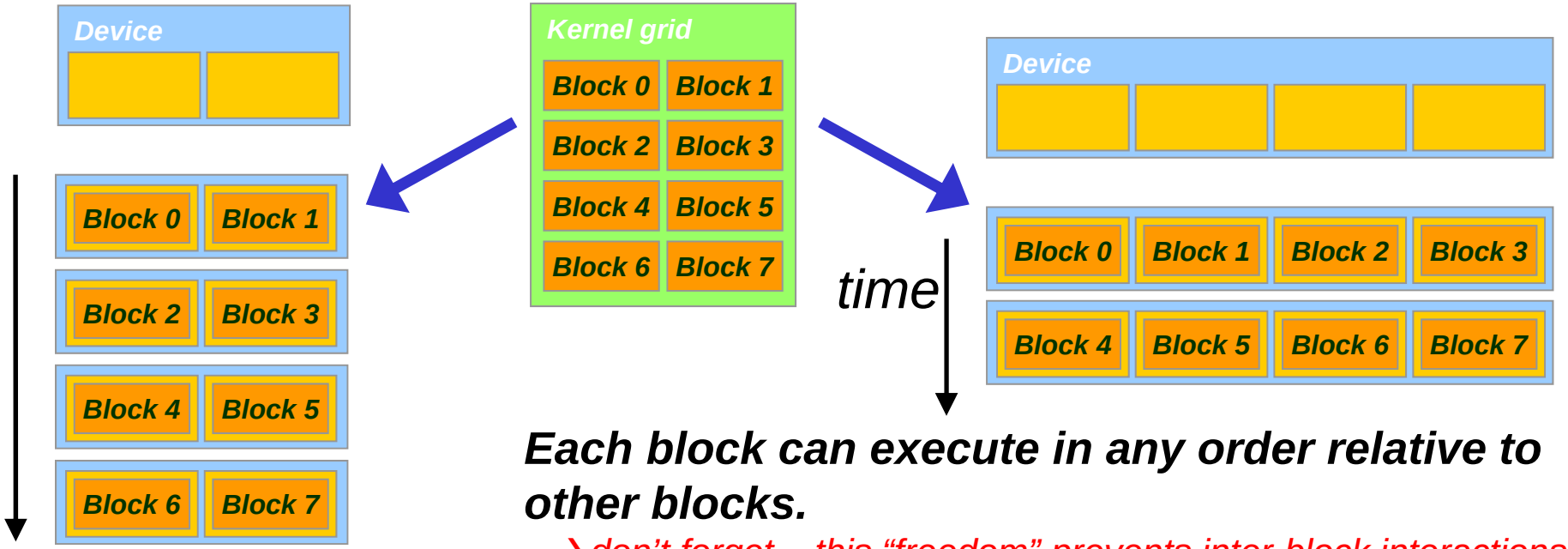


Courtesy: NDVIA

Transparent Scalability: Review

Hardware is free to assign blocks to any processor at any time

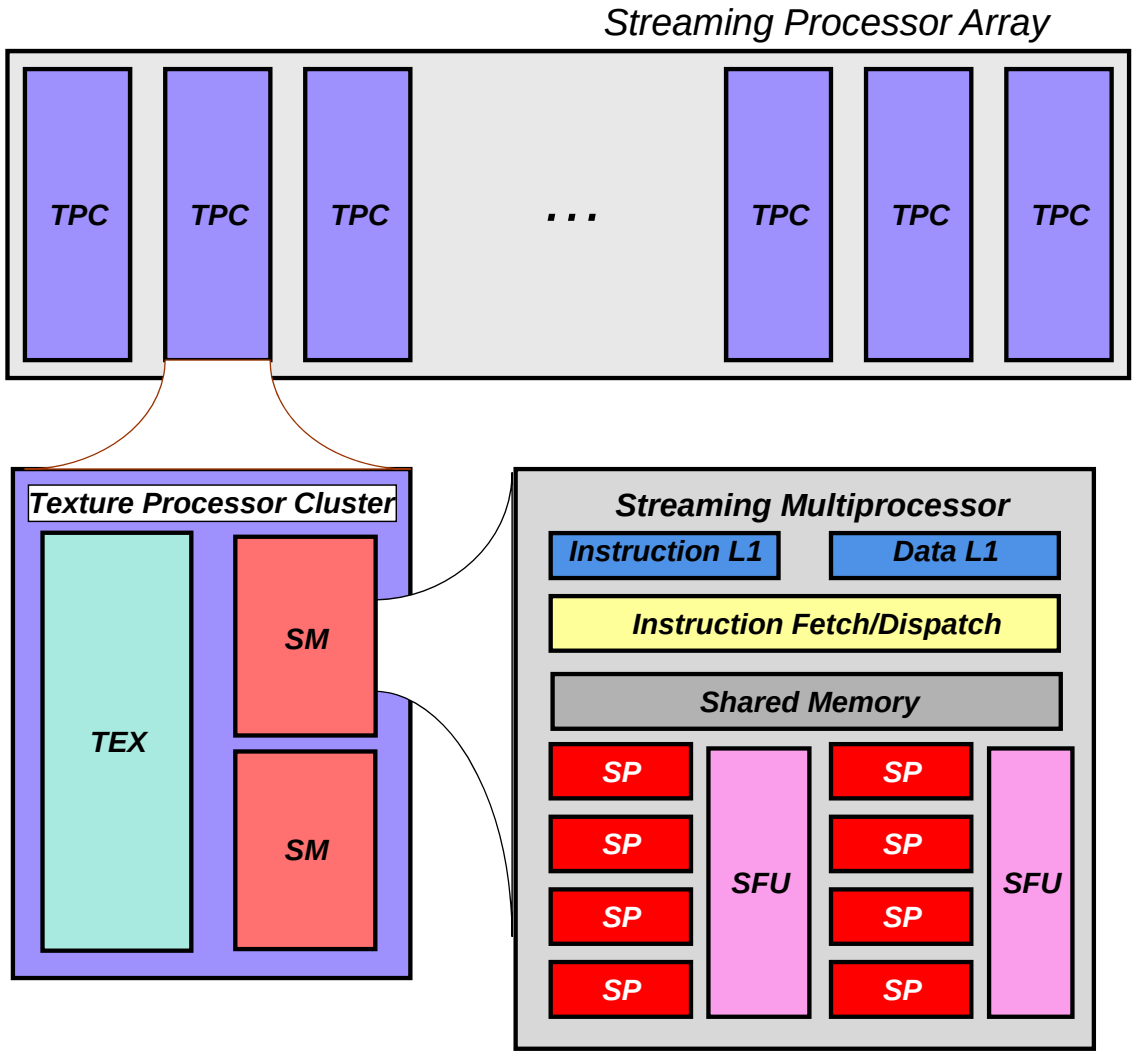
- A kernel scales across any number of SMs
- Results in HW independence



Each block can execute in any order relative to other blocks.

→ *don't forget – this "freedom" prevents inter-block interactions*

HW Overview (through G200)



SPA → Streaming Processor Array (variable across models)

TPC → Texture Processor Cluster (2 SM + TEX)

SM → Streaming Multiprocessor (8 SPs)

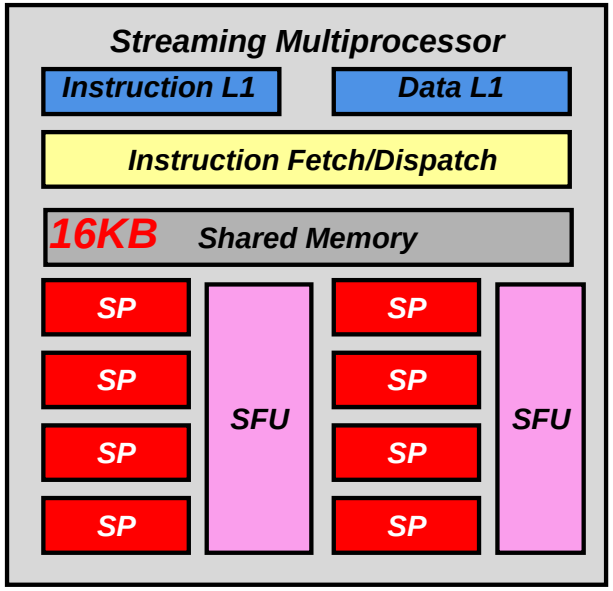
- Multi-threaded processor core
- Fundamental processing unit for CUDA thread block

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 **1024** THREADs active
  - Shared instruction fetch per 32 threads (called a WARP)
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- texture and global memory access



Thread Life Cycle in HW

Grid is launched on the SPA (device)

BLOCKs are serially distributed to all the SMs

- Potentially >1 BLOCK per SM

Each SM launches **WARP**s of THREADs

- WARP = 32 THREADs
- Half WARP = 16 THREADs

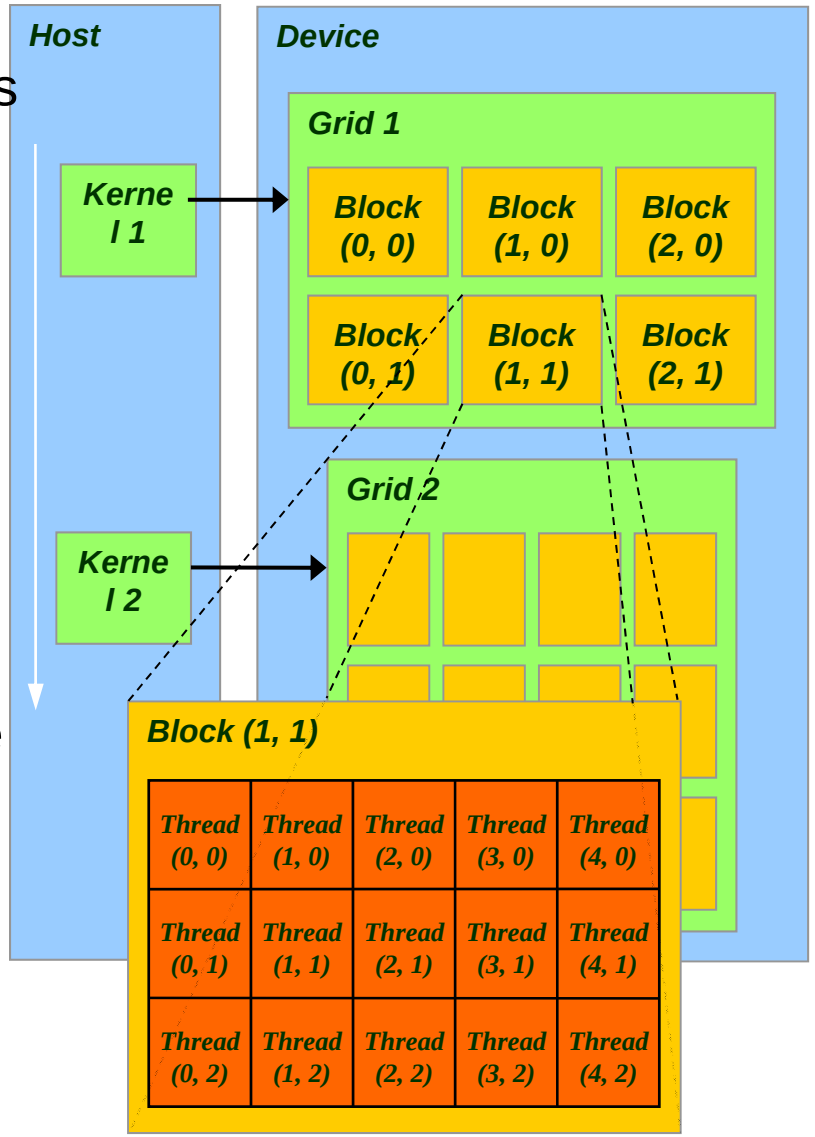
Two levels of parallelism:

- Blocks across SMs
- Threads across SPs

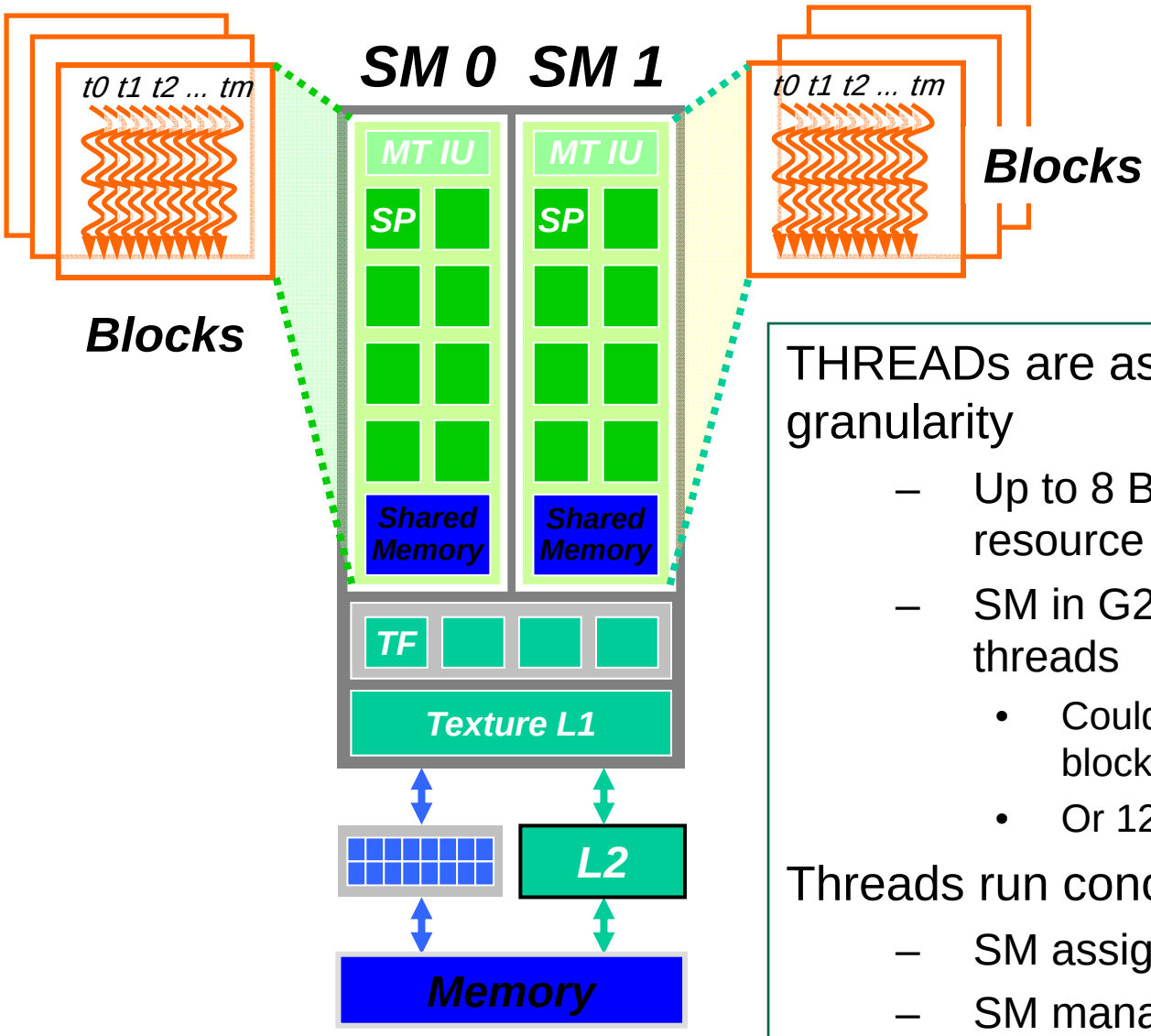
SM schedules and executes WARPs that are ready to run

As WARPs and BLOCKs complete, resources are freed

- SPA can distribute more BLOCKs to SMs



SM Executes Blocks



THREADs are assigned to SMs in BLOCK granularity

- Up to 8 BLOCKs to each SM as resource allows
- SM in G200 can take up to 1024 threads
  - Could be 256 (threads/block) * 4 blocks
  - Or 128 (threads/block) * 8 blocks, etc.

Threads run concurrently

- SM assigns/maintains thread id #s
- SM manages/schedules thread execution

G80 Example: Thread Scheduling

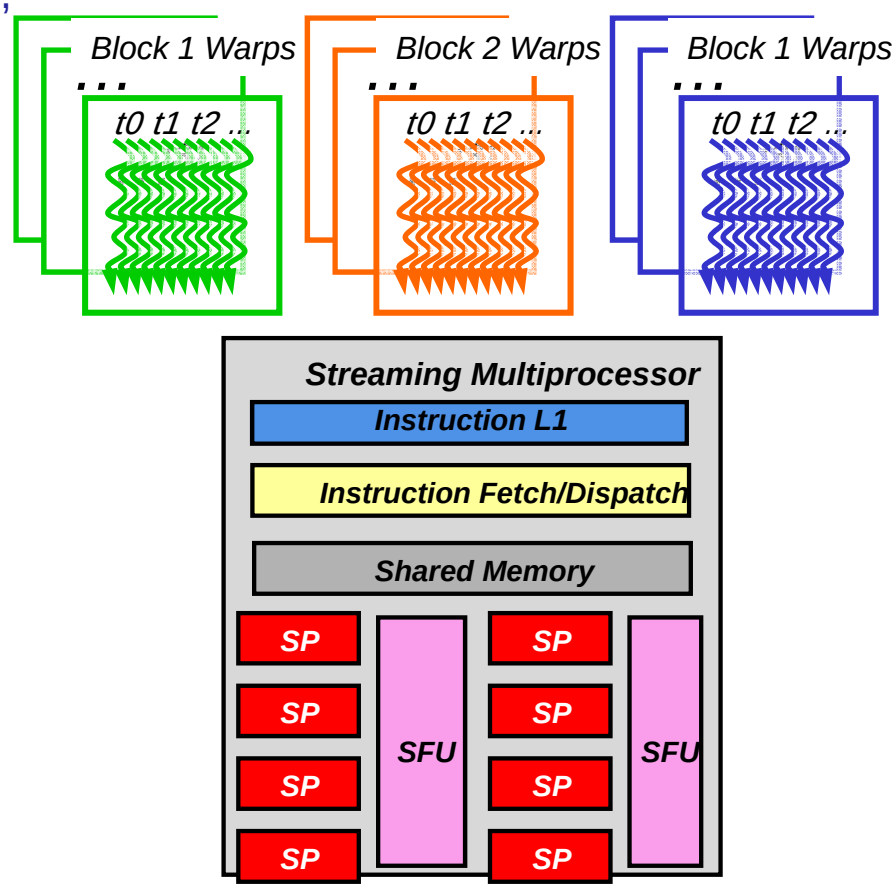
Each BLOCK is executed as 32-thread WARPs

- WARP size is an implementation decision, not part of the CUDA programming model

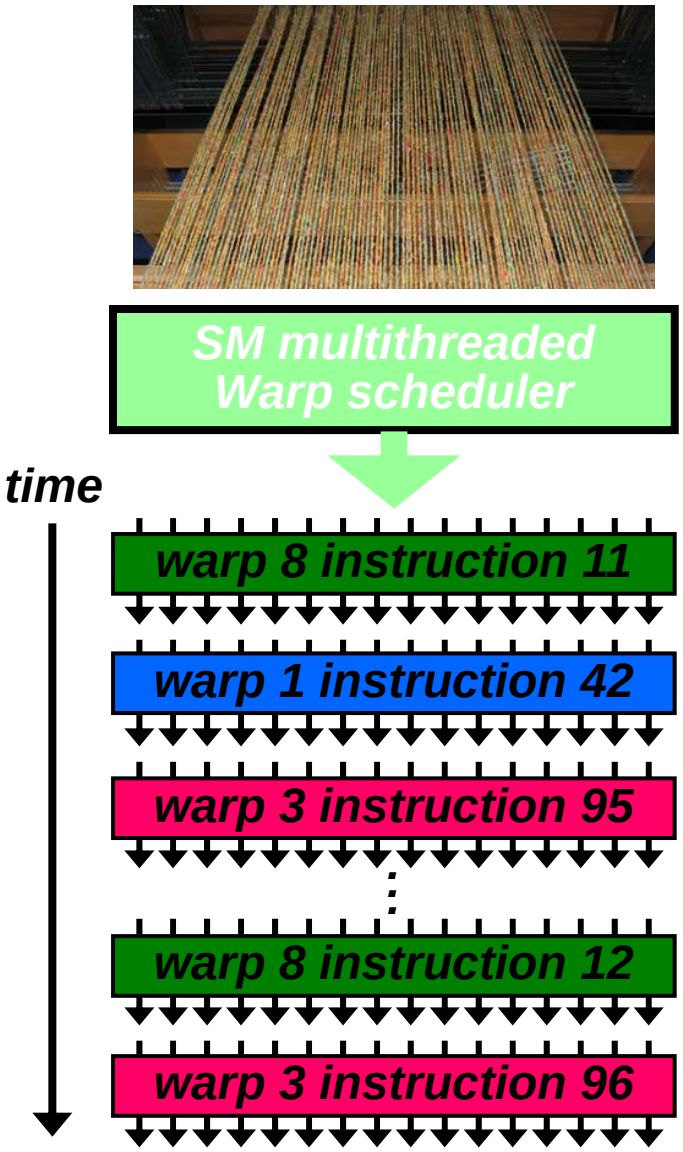
WARPs are scheduling units in SM

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 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

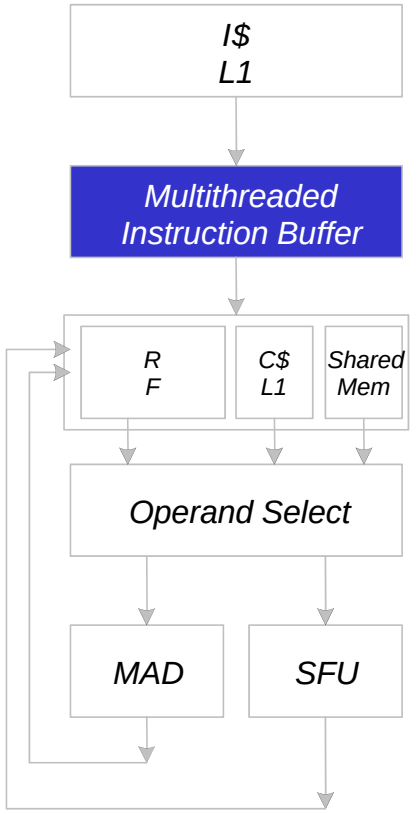
- WARPs whose next instruction has its operands ready for consumption are eligible for execution
- Eligible WARPs are selected for execution on a prioritized scheduling policy
- All THREADs in a WARP execute the same instruction when selected

4 clock cycles needed to dispatch the same instruction for all threads in a WARP in G200

- 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

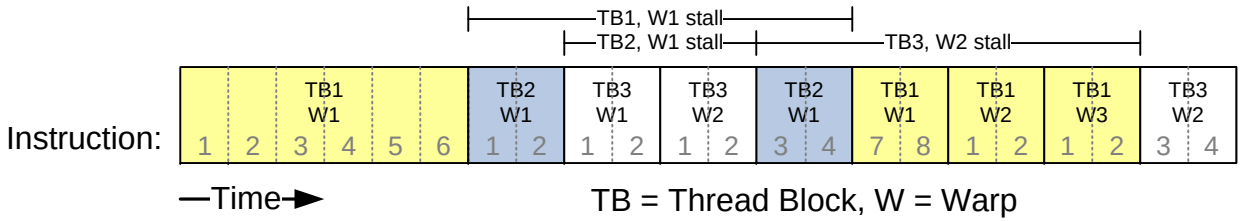
SM Instruction Buffer – WARP Scheduling

- 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



G200 Example: THREAD Scheduling

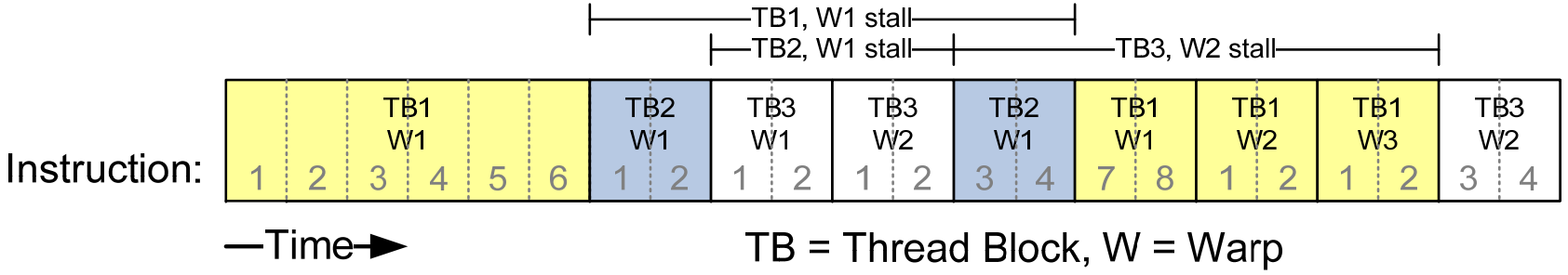
- SM implements zero-overhead WARP scheduling
  - At any time, only one of the WARPs is issued by SM
    - Kepler SMs issue two WARPs at a time
  - WARP whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible WARPs are selected for execution on a prioritized scheduling policy
  - All THREADs in a WARP execute the same instruction when selected



Note: This figure is misleading. Since instructions take 22-28 cycles, WARPs are generally alternated to hide this latency.

Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Instruction 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



Granularity Considerations

- For Matrix Multiplication, should I use 4X4, 8X8, 16X16 or 32X32 tiles?
  - For 4X4, we have 16 threads per block, Since each SM can take up to 1024 threads, the thread capacity allows 64 blocks. However, each SM can only take up to 8 blocks, thus there will be only 128 threads in each SM!
    - Also, there are 8 warps, which is just about the minimum to have a chance at high utilization, but each warp is only half full.
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1024 threads, it could take up to 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
    - There are 16 warps available for scheduling in each SM
    - Warps are full
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks and achieve full capacity unless other resource considerations overrule.
    - There are 32 warps available for scheduling in each SM
  - For 32X32, we have 1024 threads per Block which is not allowed!

Control Flow in the G200

How thread blocks are partitioned

- BLOCKs are partitioned into WARPs
  - Thread IDs within a WARP are consecutive and increasing
  - WARP 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of WARPs may change from generation to generation
  - (Covered next)

- **However, DO NOT rely on any ordering between WARPs**
  - If there are any dependencies between threads, you must __syncthreads() to get correct results

Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there are 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 and 1 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 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

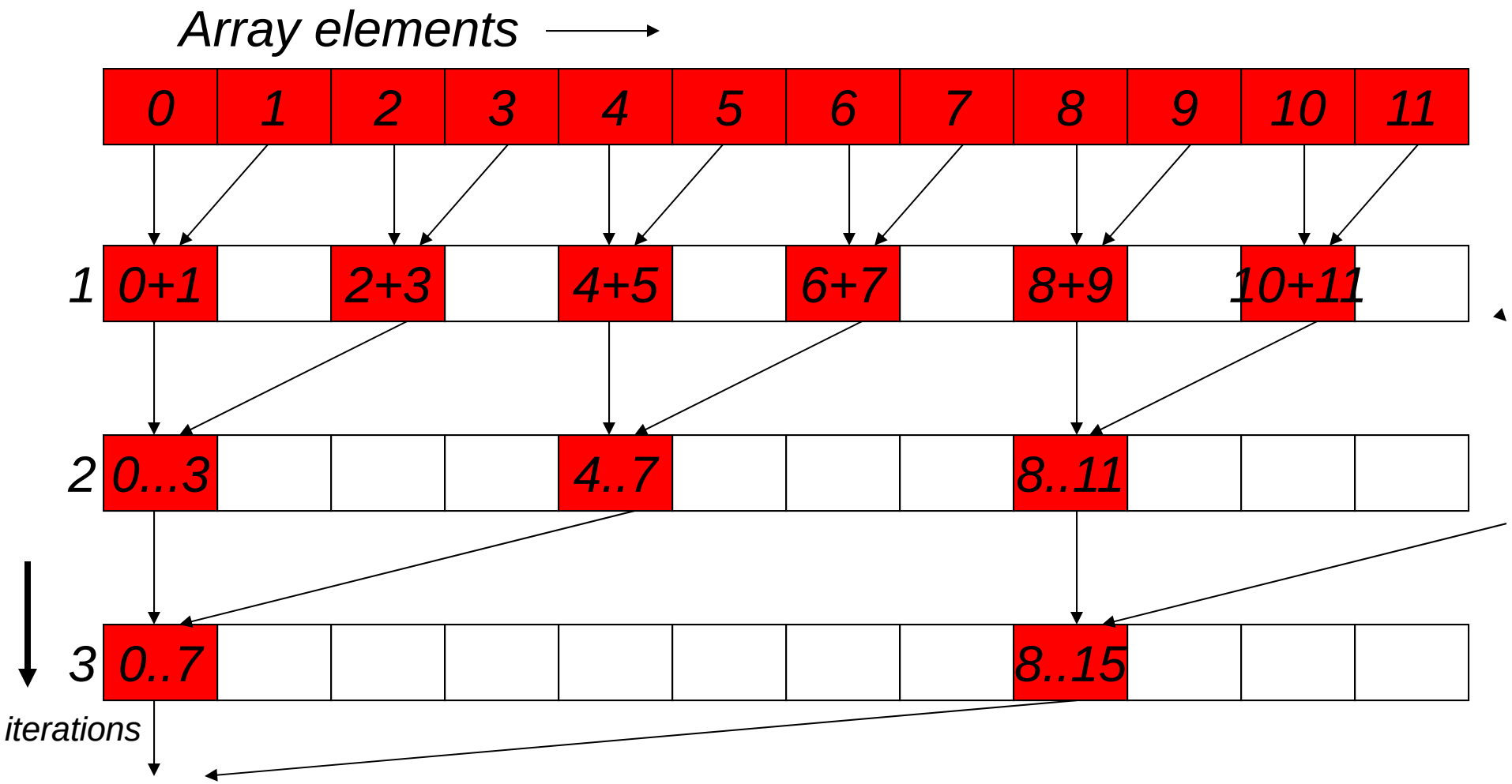
A Simple Implementation – single block

- Assume we have already loaded array into

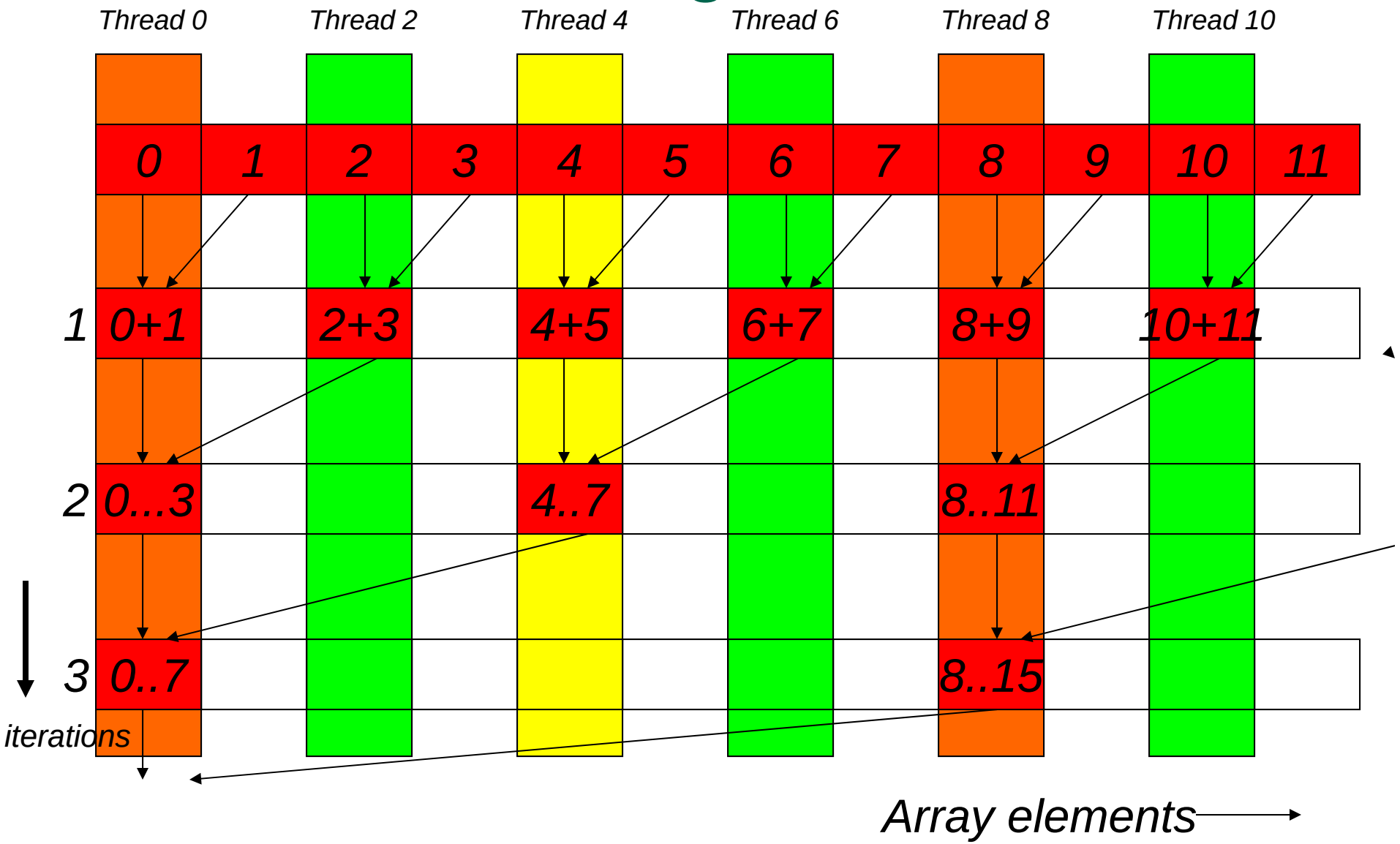
```
__shared__ float partialSum[]


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 Bank Conflicts



Vector Reduction with Branch Divergence



Some Observations

- In each iterations, 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 ¼ of the threads will be activated for all warps over time.
  - After the 5^{th} 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

A slightly better 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];
}
```

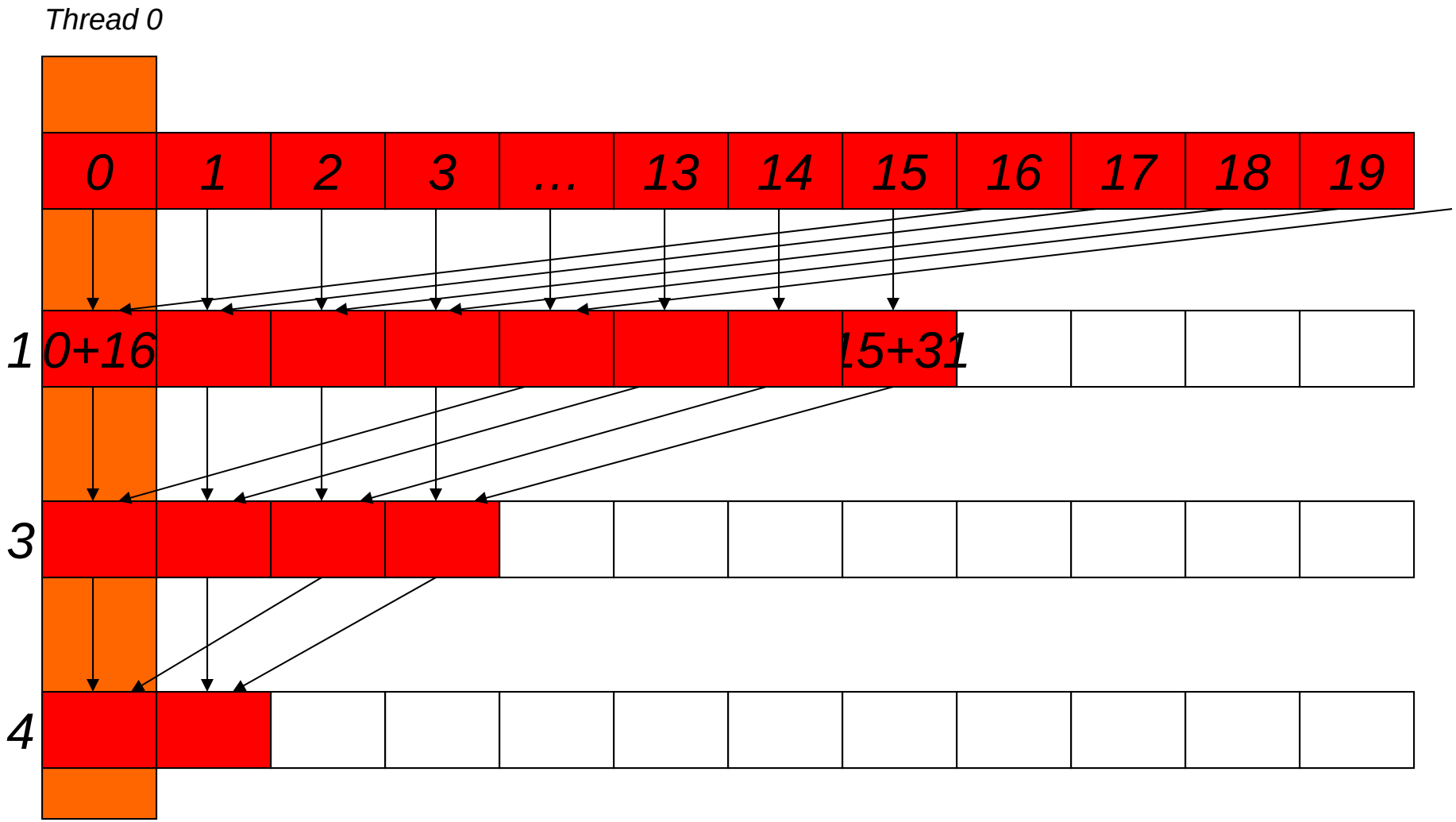
A better implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[]

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

No Divergence until < 16 sub-sums



Observations About the New Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- No bank conflicts either (see next section)

A Potential Further Refinement but bad idea

- For last 6 loops only one warp active (i.e. tid's 0..31)
  - Shared reads & writes SIMD synchronous within a warp
  - So skip `__syncthreads()` and unroll last 5 iterations

```
unsigned int tid = threadIdx.x;
for (unsigned int d = n>>1; d > 32; d >>= 1) {
   __syncthreads();
   if (tid < d)
       shared[tid] += shared[tid + d];
}
__syncthreads();
if (tid <= 32) {  // unroll last 6 predicated steps
   shared[tid] += shared[tid + 32];
   shared[tid] += shared[tid + 16];
   shared[tid] += shared[tid + 8];
   shared[tid] += shared[tid + 4];
   shared[tid] += shared[tid + 2];
   shared[tid] += shared[tid + 1];
}
```

A Potential Further Refinement but bad idea

- For last 6 loops only one warp active (i.e. tid's 0..31)
  - Shared reads & writes SIMD synchronous within a warp
  - So skip __syncthreads() a

```
unsigned int tid = thread
for (unsigned int d = n>>
   __syncthreads();
   if (tid < d)
       shared[tid] += sha
}
__syncthreads();
if (tid <= 32) {  // unrol
   shared[tid] += shared[ti
   shared[tid] += shared[tid
   shared[tid] += shared[tid + 8];
   shared[tid] += shared[tid + 4];
   shared[tid] += shared[tid + 2];
   shared[tid] += shared[tid + 1];
}
```

*This would not work properly if warp size decreases; need __synchthreads() between each statement! However, having ___synchthreads() in **if** statement is problematic.*

Predicated Execution Concept

`<p1> LDR r1,r2,0`

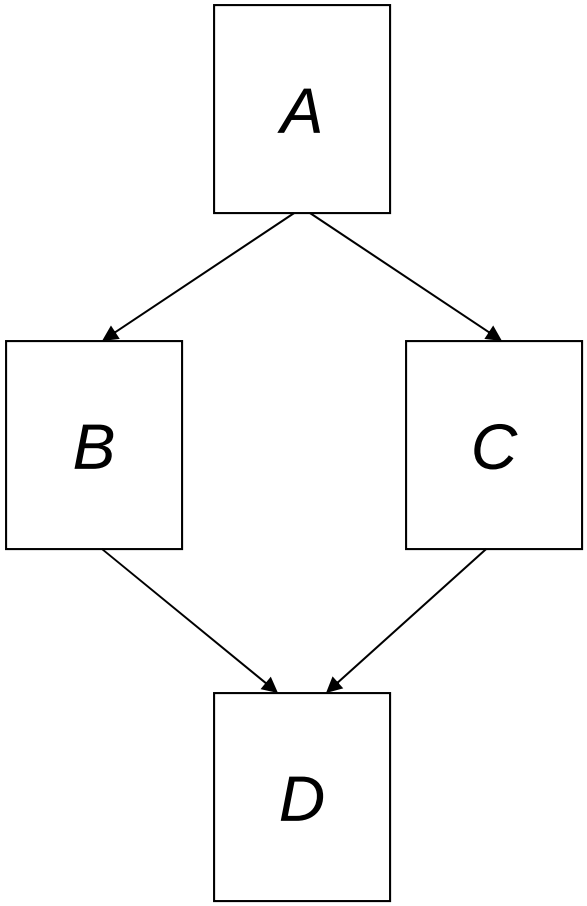
- *If p1 is TRUE, instruction executes normally*
- *If p1 is FALSE, instruction treated as NOP*

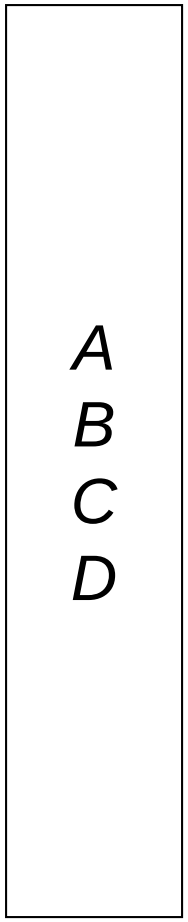
Predication Example

```
:
:
if (x == 10)
   c = c + 1;
:
```

```
      :
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1>  LDR  r1 <- C
<p1>  ADD r1, r1, 1
<p1>  STR  r1 -> C
      :
```

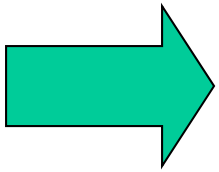
Predication very helpful for if-else





if-else example

```
      :
      :
   p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p1> inst 2 from B
<p1>  :
      :
<p2> inst 1 from C
<p2> inst 2 from C
      :
      :
```



schedule

```
      :
      :
   p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p2> inst 1 from C

<p1> inst 2 from B
<p2> inst 2 from C

<p1>  :
      :
```

The cost is extra instructions will be issued each time the code is executed. However, there is no branch divergence.

Instruction Predication in G200

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)

- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication

- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths