高性能计算II(B)

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

School of Computer and Control, University of Chinese Academy of Sciences

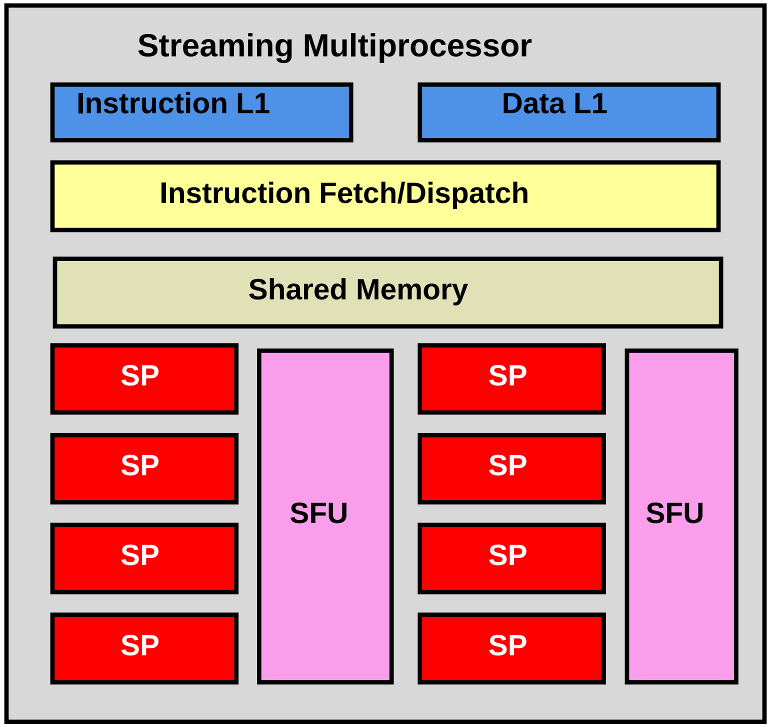
Key Lab of Big Data & Knowledge Management, Chinese Academy of Sciences

Outline

- Threads
- Warp
- Resource allocation
- Control flow

Streaming Multiprocessor (SM)

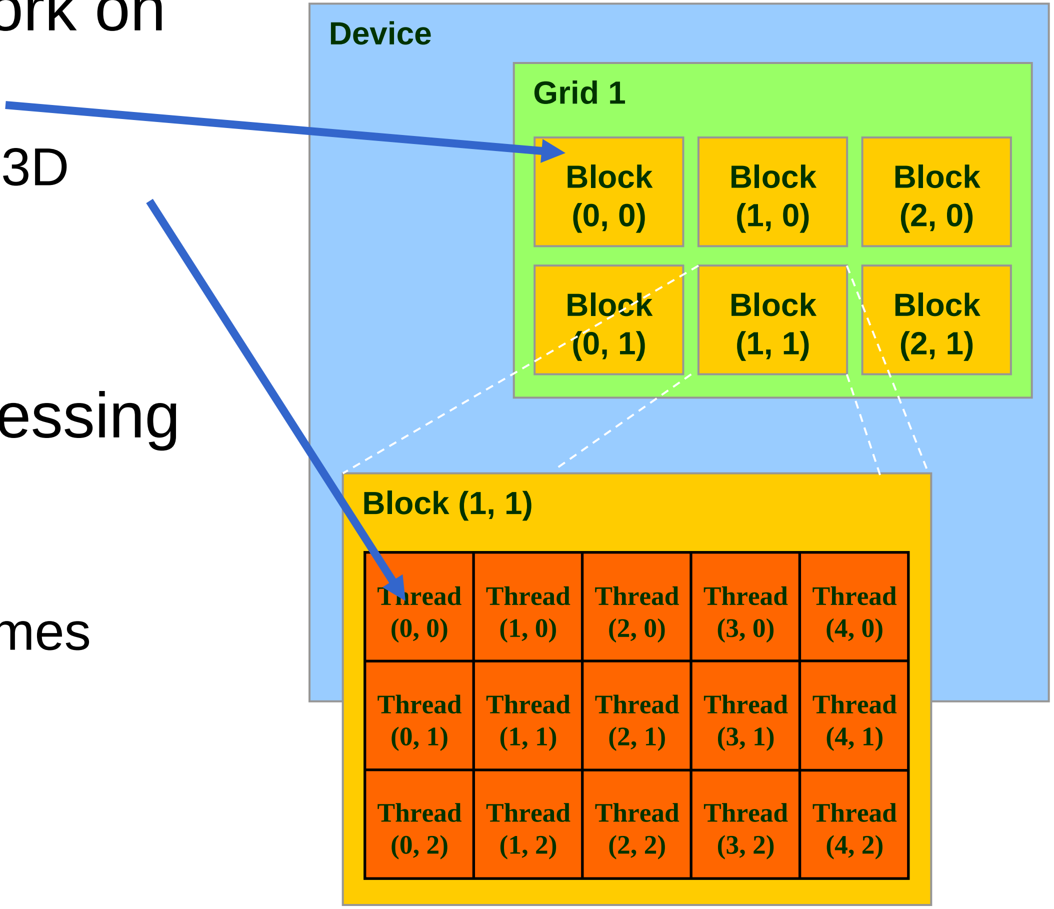
- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Special Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
- 16 KB shared memory per SM
- 86.4 GB/s memory bandwidth



Block IDs and Thread IDs

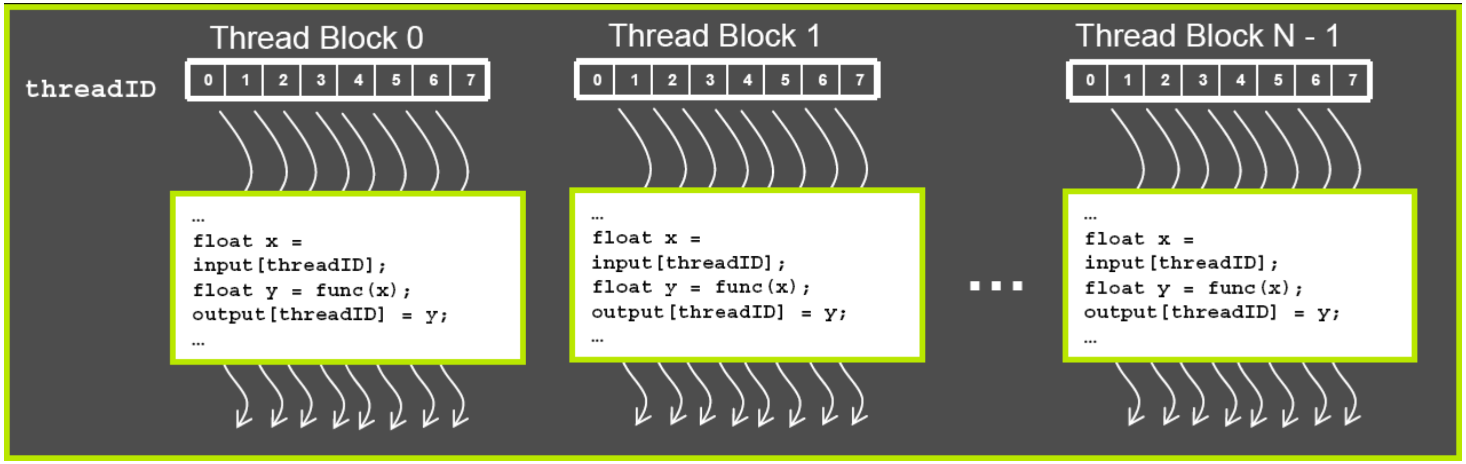
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

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



CUDA Thread Block

- Threads are grouped into thread blocks
- Kernel = a grid of thread blocks



```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```

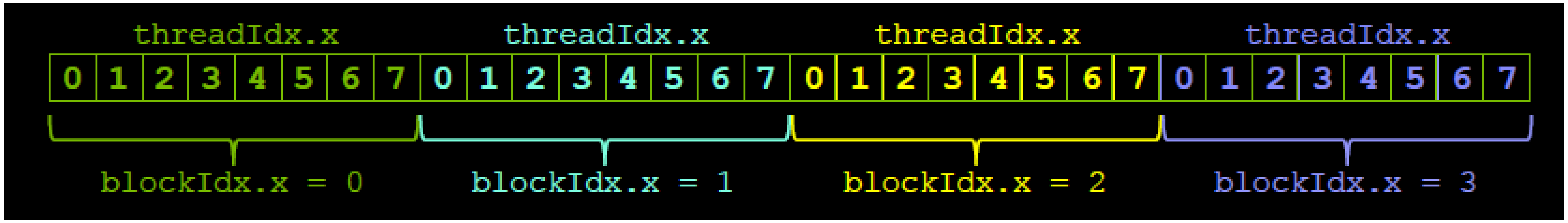
```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```

An Example: Parallel Addition

- Device code

```
__global__ voidadd( int*a, int*b, int*c ) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```



- To index array with 1 thread per entry (using 8 threads/block)
- If we have M threads/block, a unique array index for each entry given by

    int index = threadIdx.x + blockIdx.x * M;

An Example: Parallel Addition (Cont.)

- Host code

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
        int *a, *b, *c;                          //host copies of a, b, c
        int *dev_a, *dev_b, *dev_c;              //device copies of a, b, c
        int size = N * sizeof( int);             //we need space for 512 integers
// allocate device copies of a, b, c
        cudaMalloc( (void**)&dev_a, size );
        cudaMalloc( (void**)&dev_b, size );
        cudaMalloc( (void**)&dev_c, size );

        a = (int*)malloc( size );
        b = (int*)malloc( size );
        c = (int*)malloc( size );
        random_ints( a, N );
        random_ints( b, N );
```

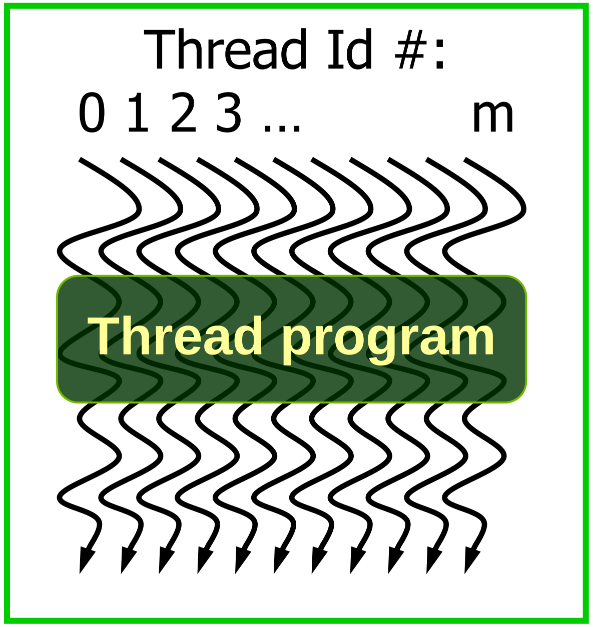
An Example: Parallel Addition (Cont.)

```
    // copy inputs to device
        cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);

// launch add() kernel with N
        add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(
dev_a, dev_b, dev_c);

// copy device result back to host copy of c
        cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);
        free( a ); free( b ); free( c );
        cudaFree( dev_a);
        cudaFree( dev_b);
        cudaFree( dev_c);

        return0;
}
```

CUDA Thread Block (Cont.)

- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data

CUDA Thread Block



Thread Id #:
0 1 2 3 … m

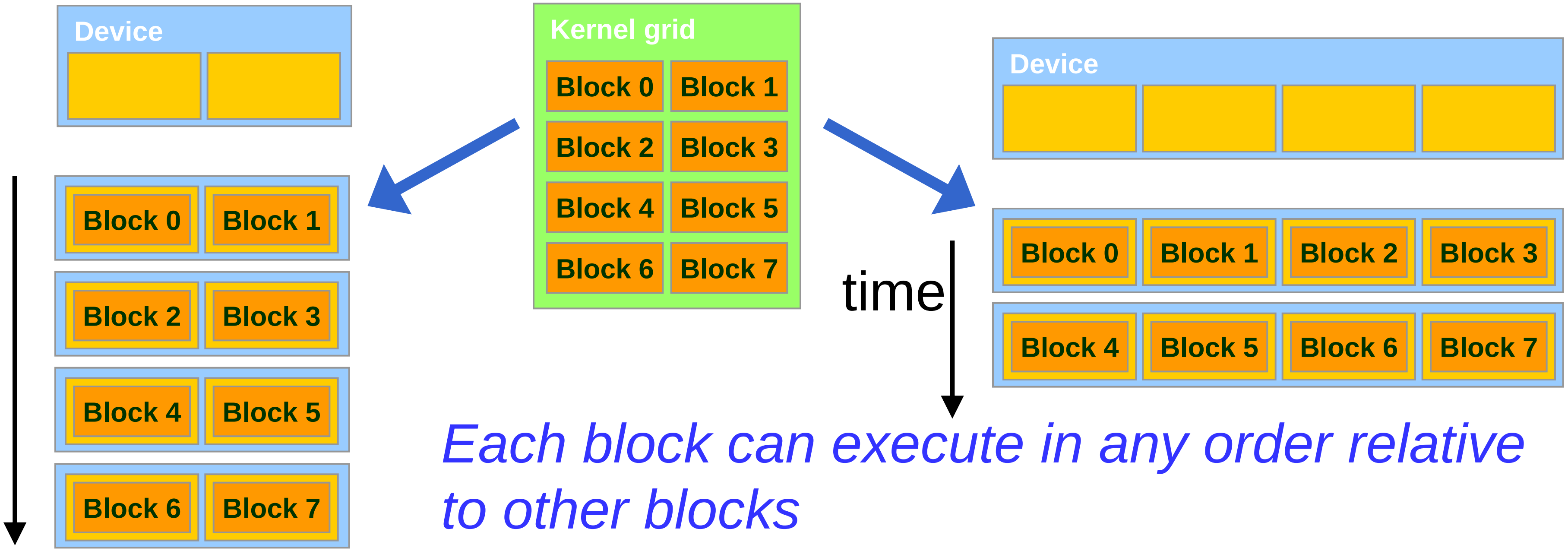
Thread program

CUDA Thread Block (Cont.)

- All threads in a block execute the same kernel program (SPMD)
- Threads in the same block are able to share data and synchronize while doing their share of the work
- Threads in different blocks are not able to cooperate
  - Each block can execute in any order relative to other blocks

Transparent Scalability

- Hardware is free to assign blocks to any stream multi-processor (SM) at any time
  - A kernel scales across any number of parallel processors



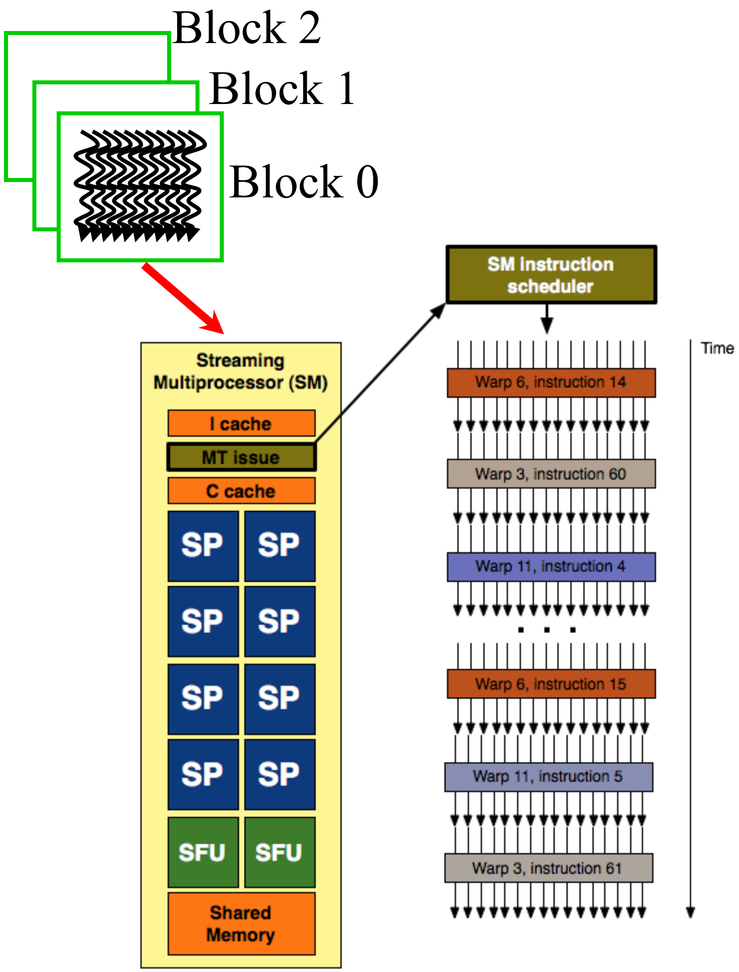
Each block can execute in any order relative to other blocks

Outline

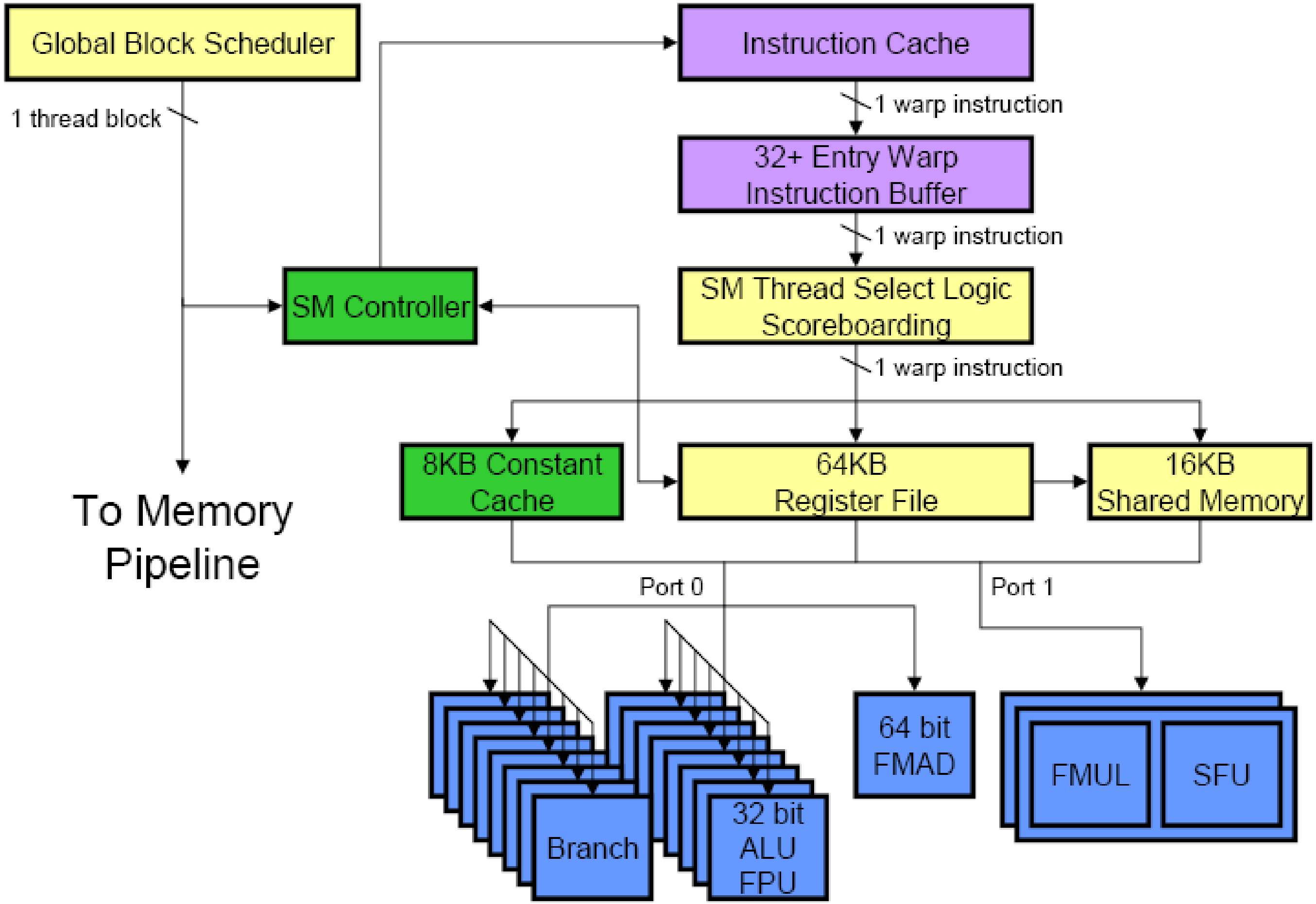
- Threads
- Warp
- Resource allocation
- Control flow

Thread Execution

- A warp of 32 threads physically running on a SM
  - Sharing instructions
  - 4 cycles for 1 warp instruction
    - Why?
  - Dynamically scheduled by SM
    - Executed when operands ready

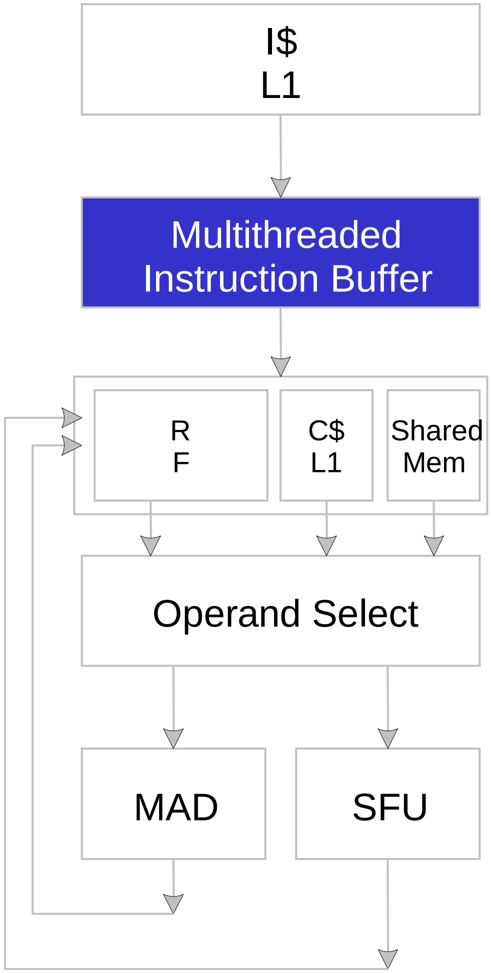


Streaming Multiprocessor (SM)



Warp Scheduling

- One instruction fetch per clock cycle
  - from L1 instruction cache
  - load into the instruction buffer
- One warp is selected to execute whose data is ready
  - from the instruction buffer
  - use scoreboard to prevent hazards
- All threads in a warp execute the same instruction
  - SM broadcasts the instruction to 32 threads in the warp



Scoreboarding

- Scoreboard monitors the operands of the instructions in the instruction buffer
  - Give a ‘ready’ mark when all the operands are ready
  - Prevent hazards
- Scoreboarding guarantees the execution of the warp with no interrupt
- Separate memory/processor pipeline

How Thread Blocks Are Partitioned

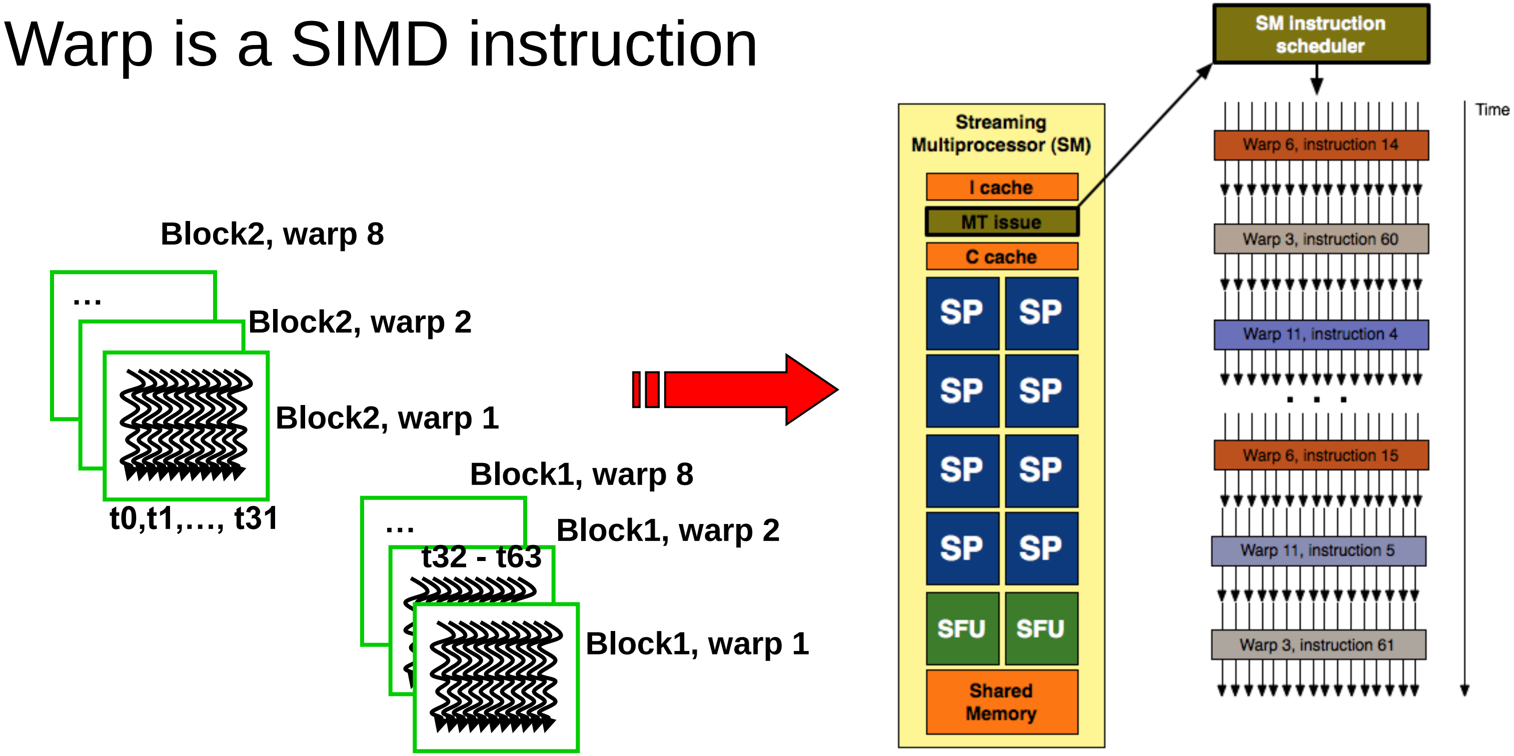
- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- **However, DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must __synchthreads() to get correct results

G80 Example: Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors 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 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
  - 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 !
- Threads run concurrently

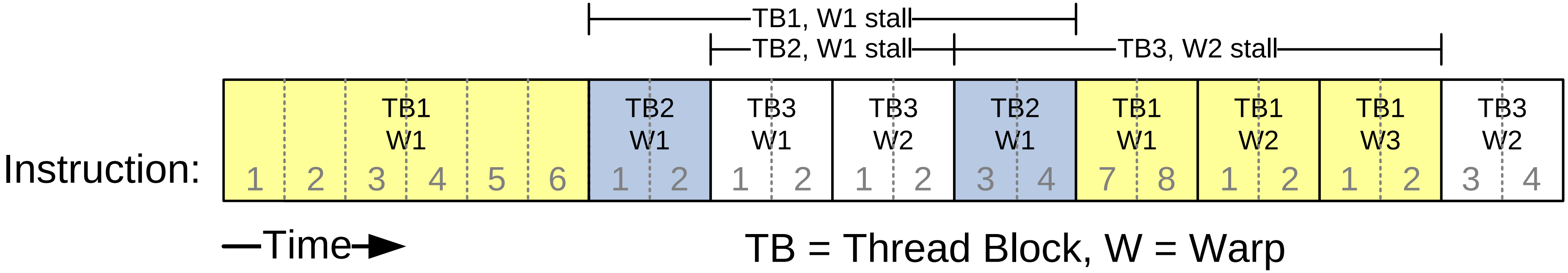
G80 Example: Thread Scheduling

- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
  - Warp is a SIMD instruction



G80 Example: Thread Scheduling (Cont.)

- 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
    - Round-Robin
  - All threads in a warp execute the same instruction when selected



TB = Thread Block, W = Warp

Hiding Memory Latency

- 4 clock cycles needed to dispatch a single instruction for all threads in a warp
  - If one global memory access is needed for every *n* instructions
  - A 400-cycle global memory latency
  - A minimum 400/4*n* warps are needed to tolerate the latency

Outline

- Threads
- Warp
- Resource allocation
- Control flow

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

Matrix Multiplication Example

- If each block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + 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*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two blocks can run on an SM, **1/3 reduction of thread-level parallelism (TLP)**!!!

Dynamic Partitioning

- Dynamic partitioning of SM resources 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 have 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!

G80 Tiling Granularity Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?

  - For 8X8, we have 64 threads per block. Since each SM can take up to 768 threads, it can take up to 12 blocks. However, each SM can only take up to 8 blocks, only 512 threads will go into each SM!

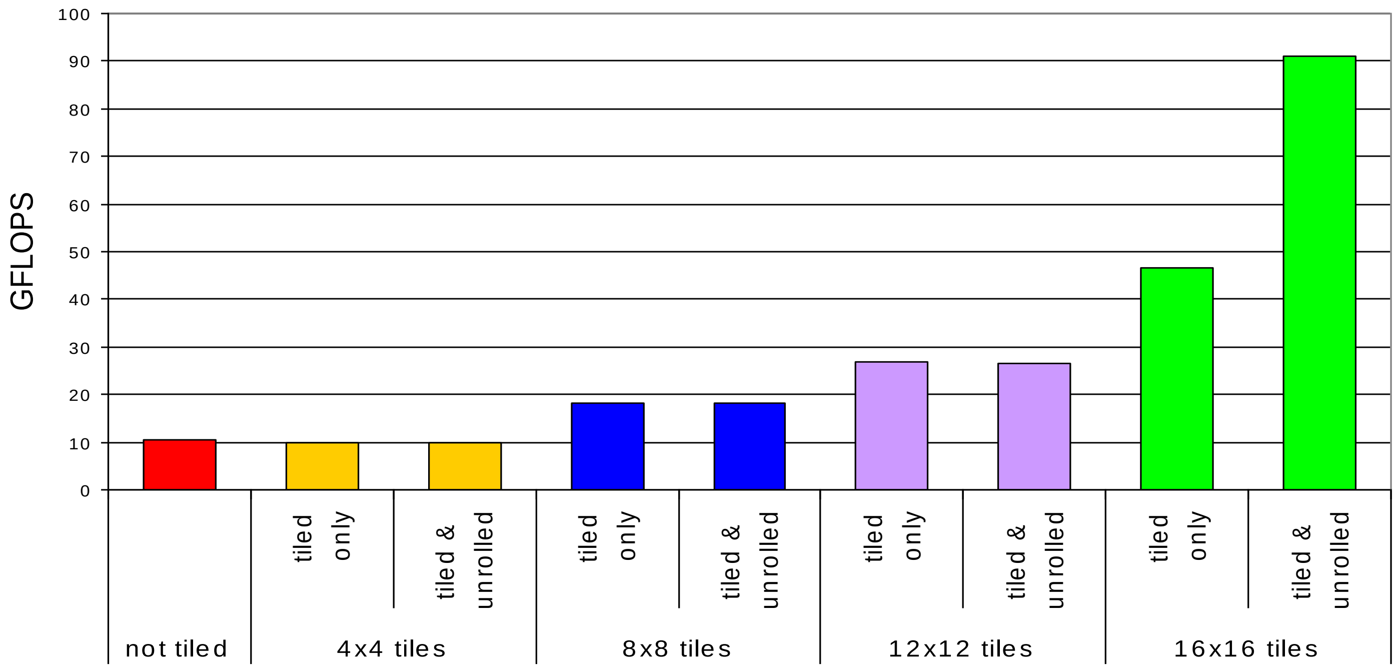
  - For 16X16, we have 256 threads per block. Since each SM can take up to 768 threads, it can take up to 3 blocks and achieve full capacity unless other resource considerations overrule.

  - For 32X32, we have 1024 threads per block. Not even one can fit into an SM!

G80 Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - Dynamic resource: Blocks under the same SM share the 16KB, but different blocks can’t read data of others
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory

Tiling Size Effects



Outline

- Threads
- Warp
- Resource allocation
- 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 in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more

Control Flow Instructions

- 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 a 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/min reduction: maximum/minimum 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

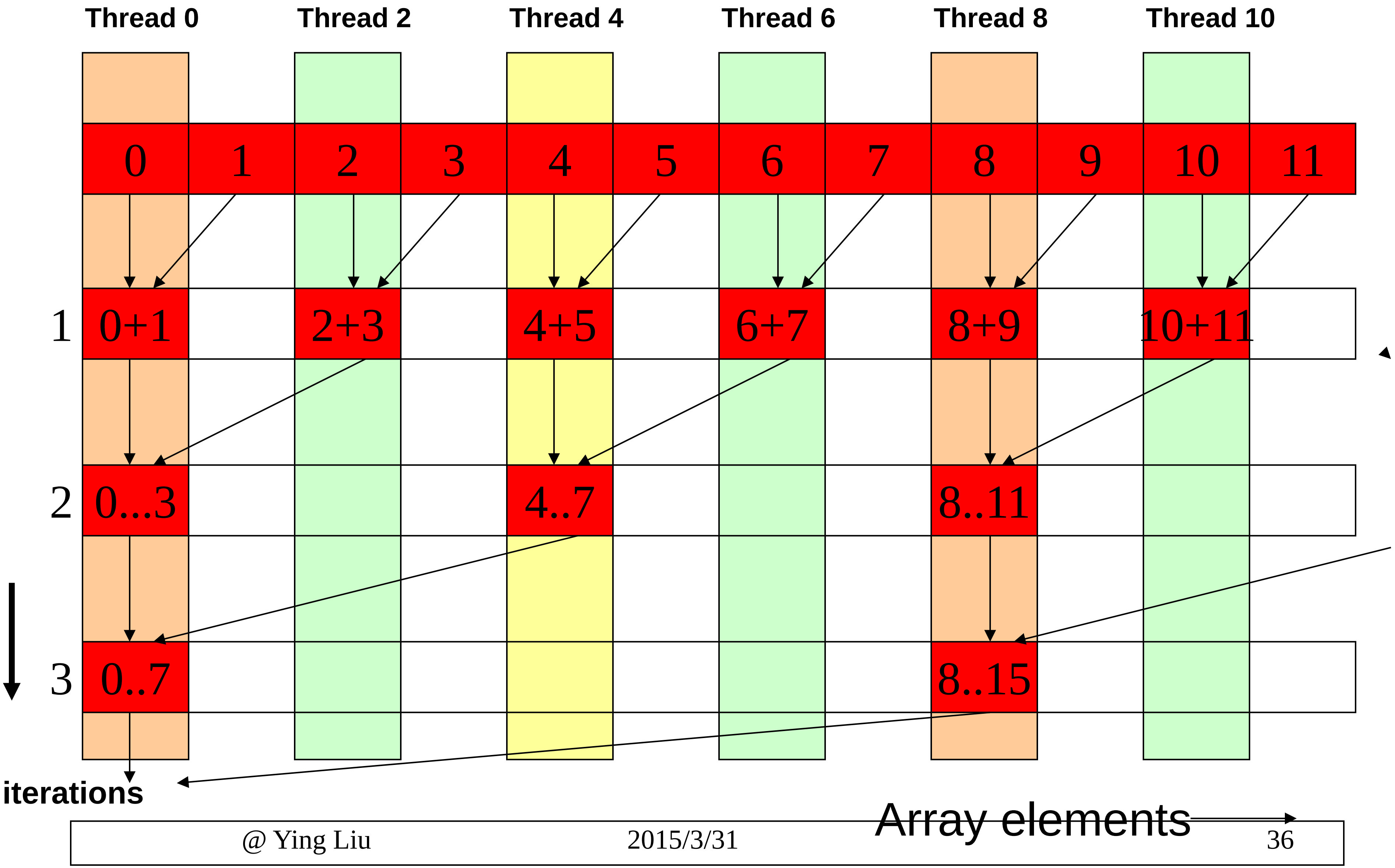
- Use shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

A simple implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[blockDim.x]

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 ¼ 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

Shortcomings of the Implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[blockDim.x]

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];
}
```

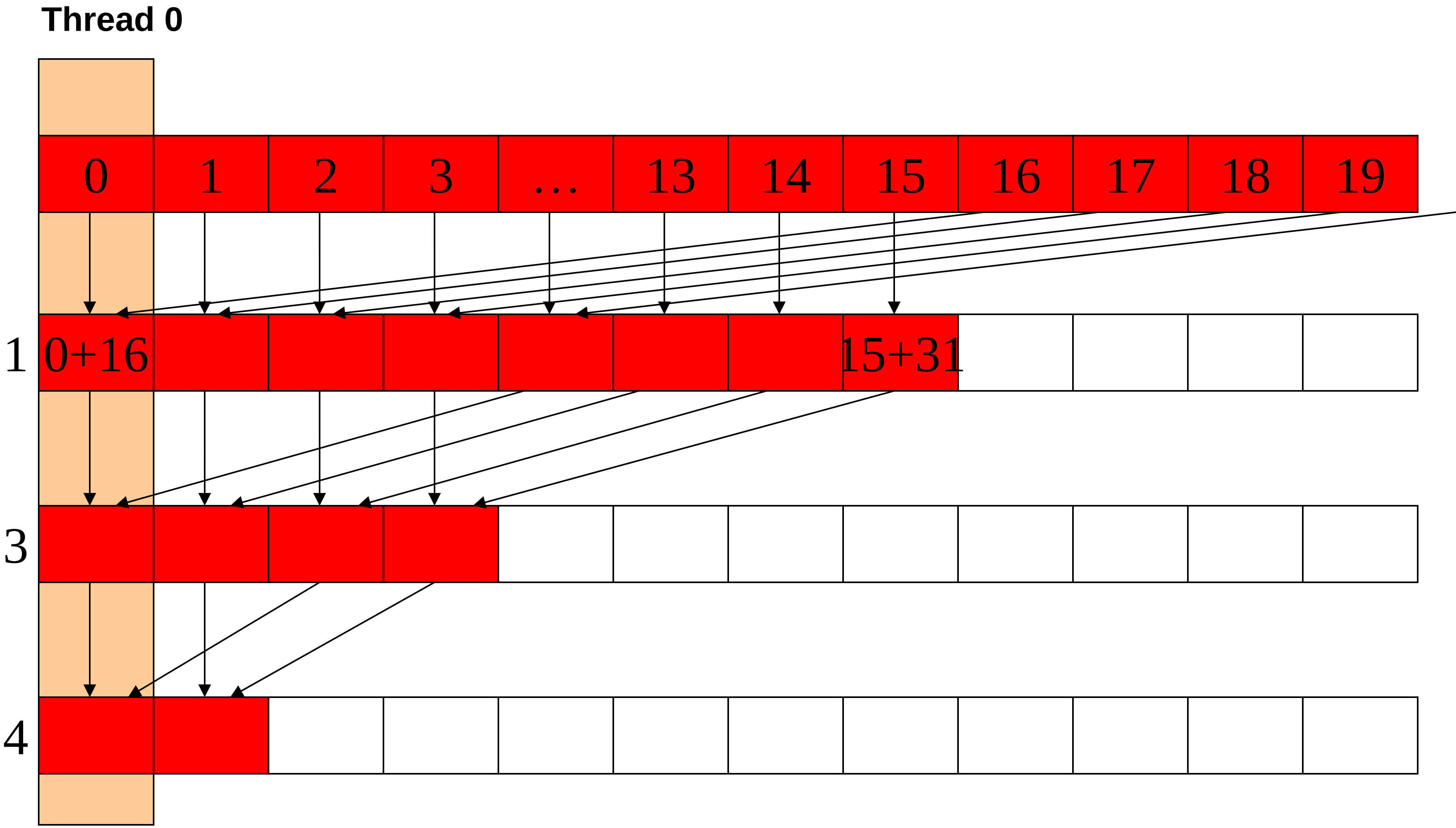
BAD: Divergence due to interleaved branch decisions

A Better Implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[blockDim.x*2]

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



Time Function

clock_t clock( )

Return value: per-multiprocessor counter incremented every clock cycle

  - Sample at the beginning and the end of a kernel
  - Take the difference of the two samples
  - Record the result

- Time by clock() is larger than the time actually spent on executing thread instructions

Time Function

- CUDA Event
  - An accurate timing
  - Implementation: asynchronously record event at any point in a program

Creation of CUDA Event:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Time Function

Timing using CUDA Event:

```
cudaEventRecord(start, 0);
myKernel<<<100, 512>>>(outputDev, inputDev);
…
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Destruction of CUDA Event:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```