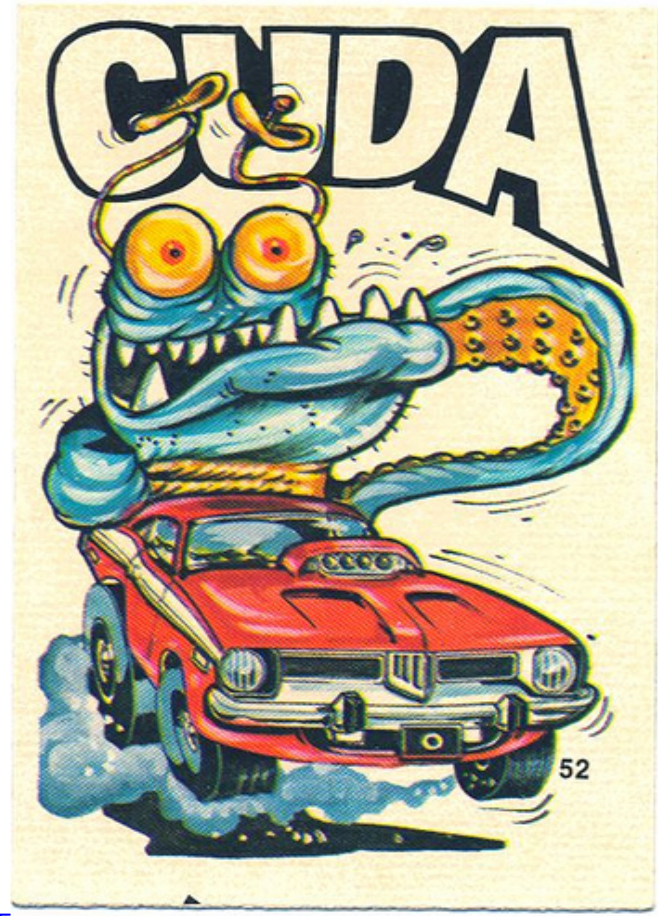
CS-206 Concurrency

Lecture 12

CUDA



Spring 2015

Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

Adapted from slides originally developed by Babak Falsafi, David Kirk and Andreas Moshovos

Where are We?

M	T	Lecture & Lab W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	[illegible]	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

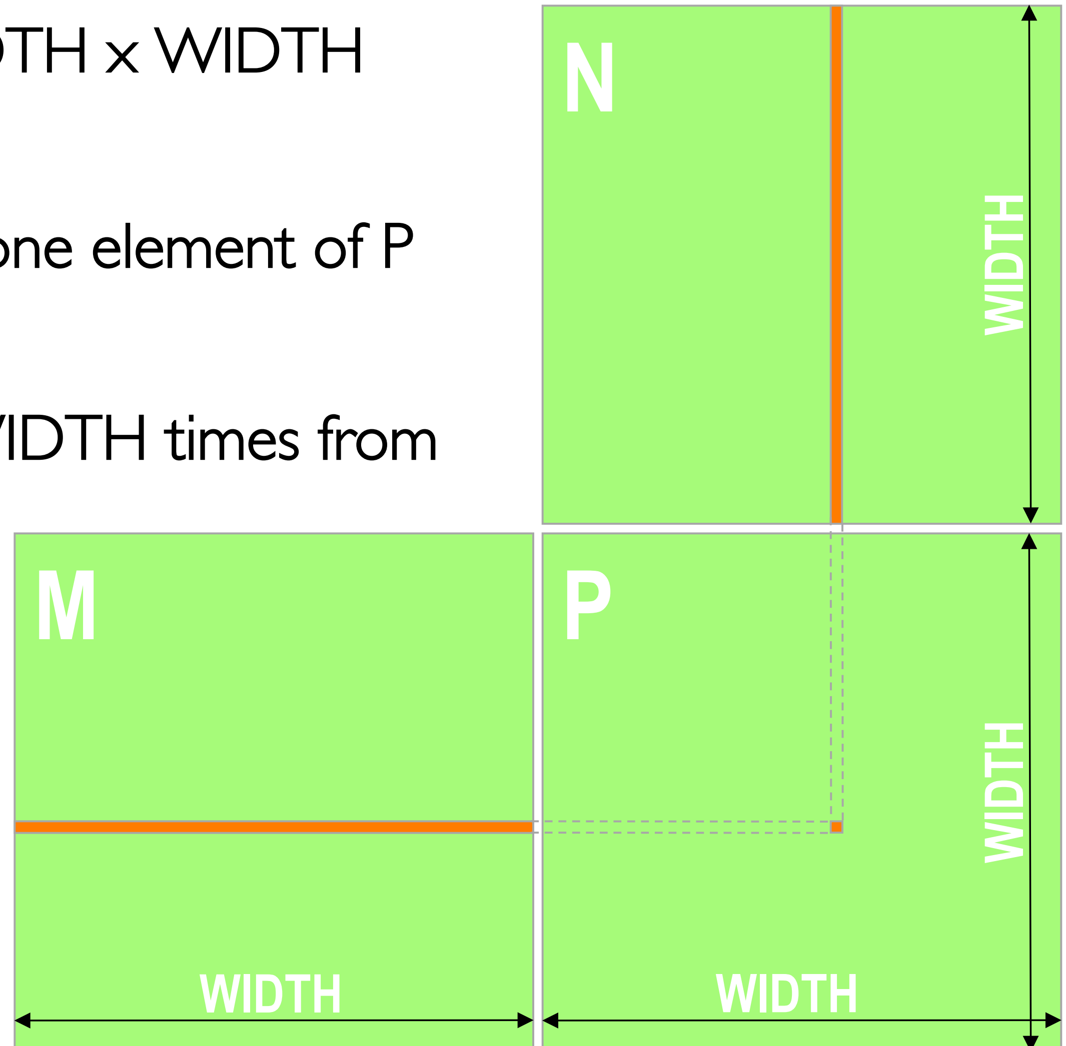
- Matrix Multiply
  - Basic
- Performance
  - Shared memory/Tiling
  - WARPs
  - Memory bank conflicts
  - Loop overhead

Can you do this one now?

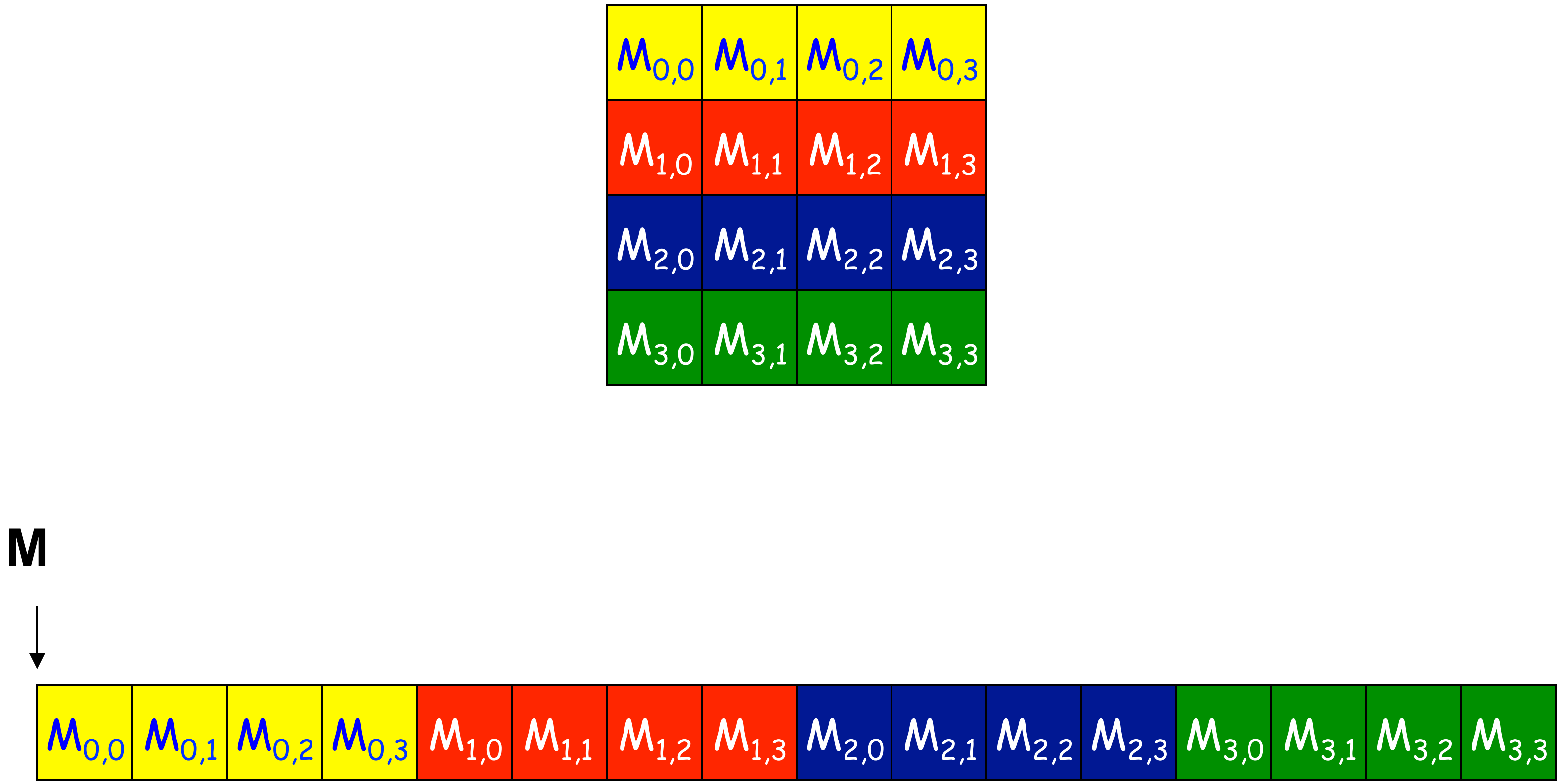
$$\left(\boldsymbol{C}\right) = \left(\boldsymbol{A}\right) \bullet \left(\boldsymbol{B}\right)$$

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- One thread calculates one element of P
- M and N are loaded WIDTH times from global memory

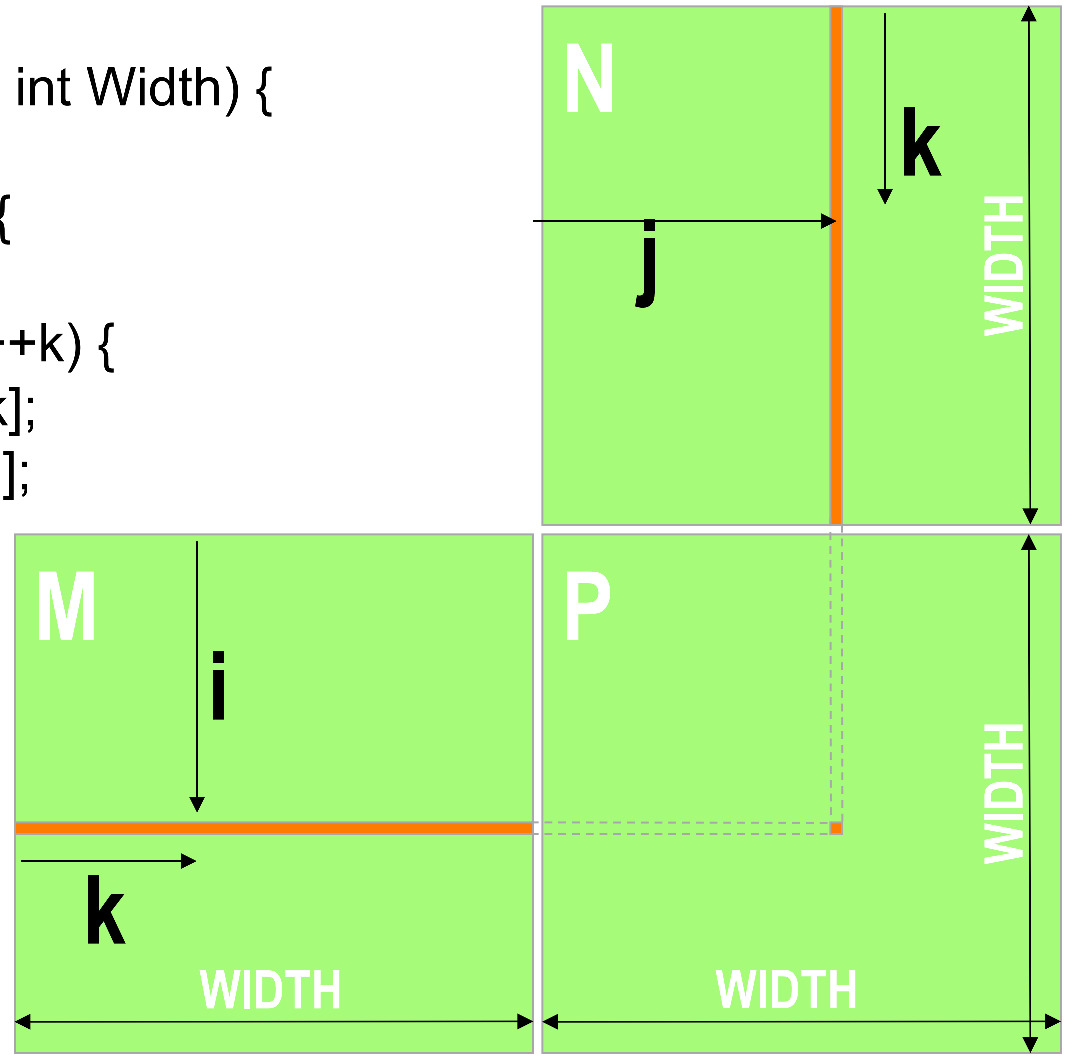


Memory Layout of a Matrix in C



Step 1: Matrix Multiplication
A Simple Host Version in C

```
// Matrix multiplication on (CPU) host
void MatrixMulOnHost (float* M,
                      float* N, float* P, int Width) {
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      float sum = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[i * width + k];
        float b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice (float* M, float* N, float* P, int Width) {
   int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
   …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

     // Allocate P on the device
    cudaMalloc(&Pd, size);
```

Step 3: Output Matrix Data Transfer (Host-side Code)

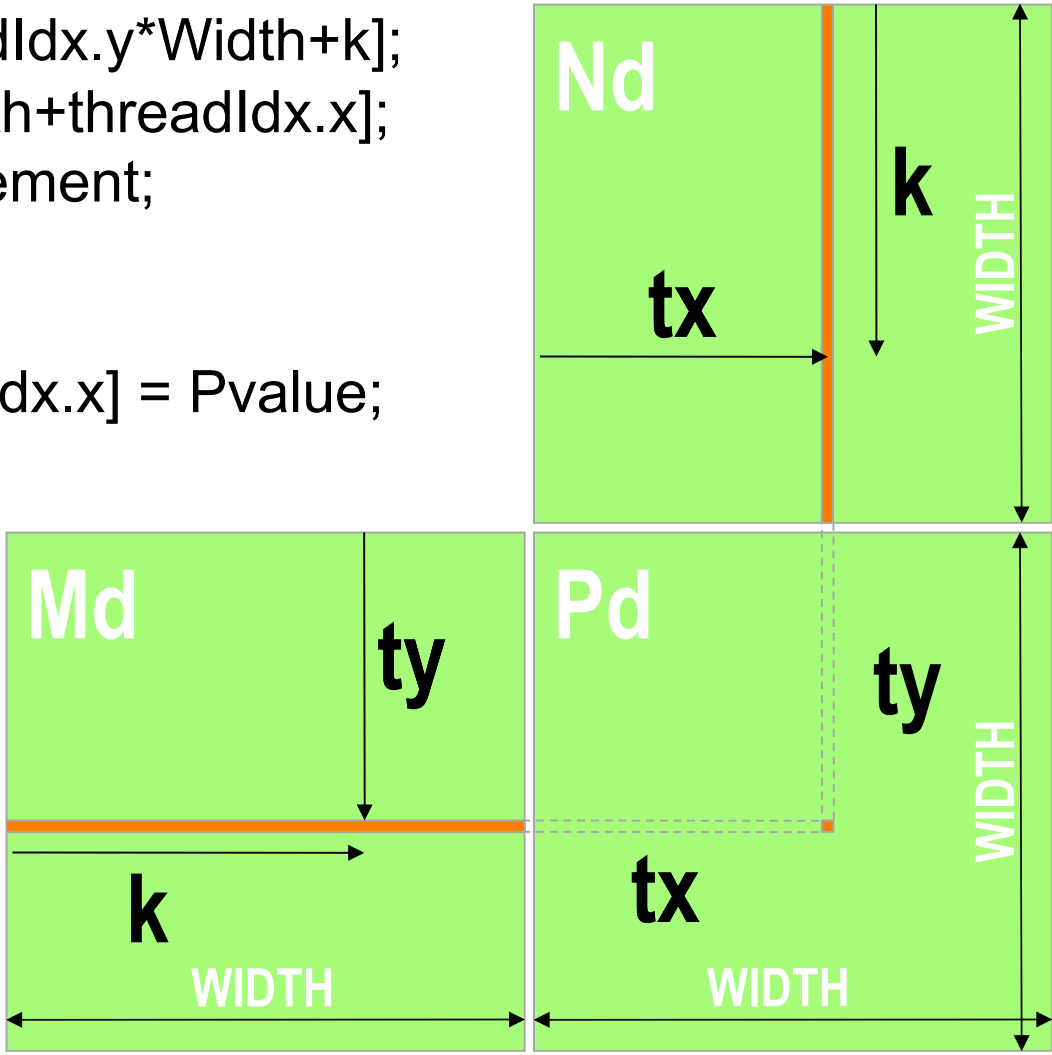
```
2.  // Kernel invocation code – to be shown later
    …

3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

Step 4: Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__
void MatrixMulKernel (float* Md, float* Nd, float* Pd, int Width) {

   // Pvalue is used to store the element of the matrix
   // that is computed by the thread
   float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

```
for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
 }

 Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

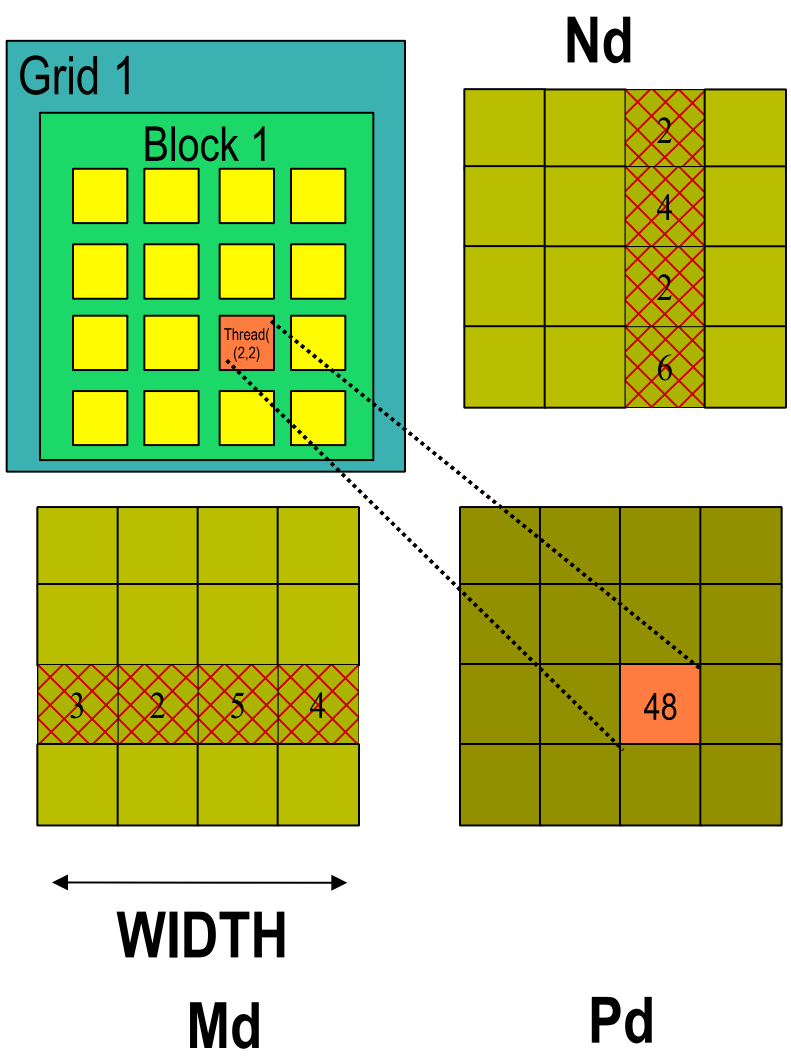


Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
   dim3 dimGrid(1, 1);
   dim3 dimBlock(Width, Width);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Only One Thread Block Used

- Each thread computes one Pd element
  - Loads row of matrix Md
  - Loads column of matrix Nd
  - Performs one multiply and addition
- Compute to global memory access ratio close to 1:1
  - not very high!
- Size of matrix limited by the number of threads allowed in a thread block

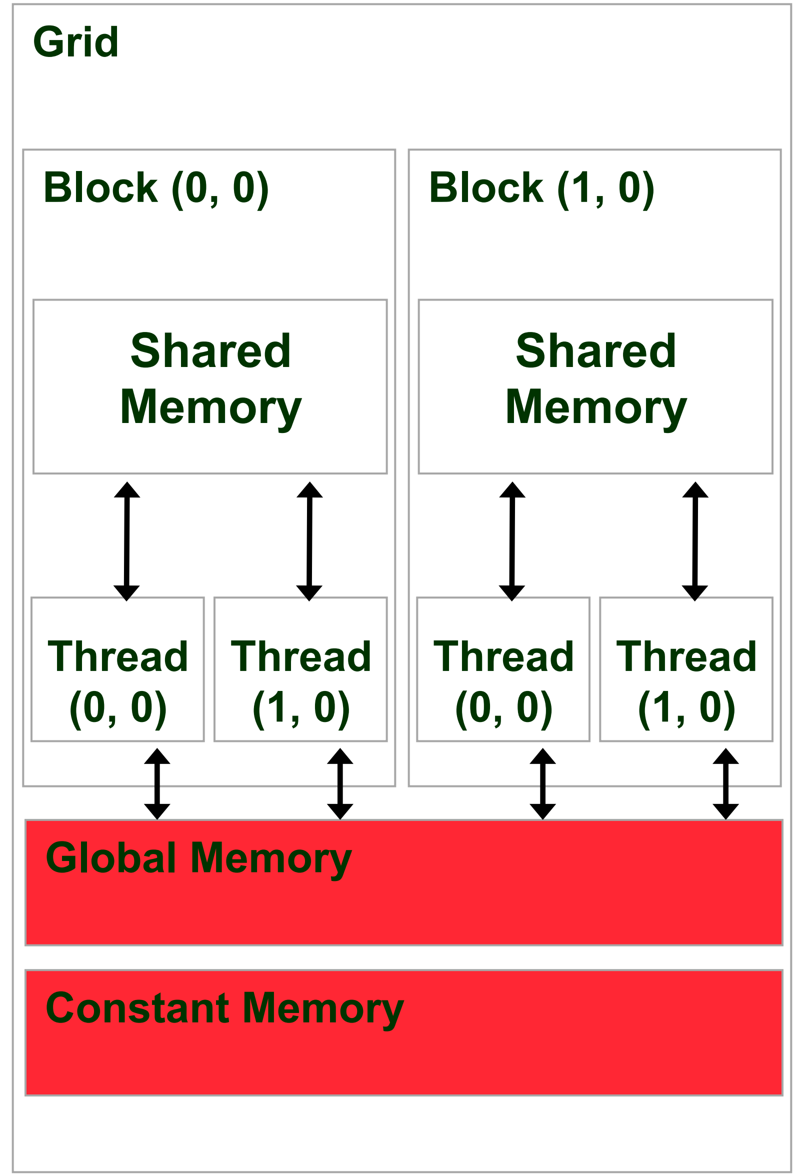


What is the required memory bandwidth?

All accesses to global memory

In inner loop (k from 0 to WIDTH)

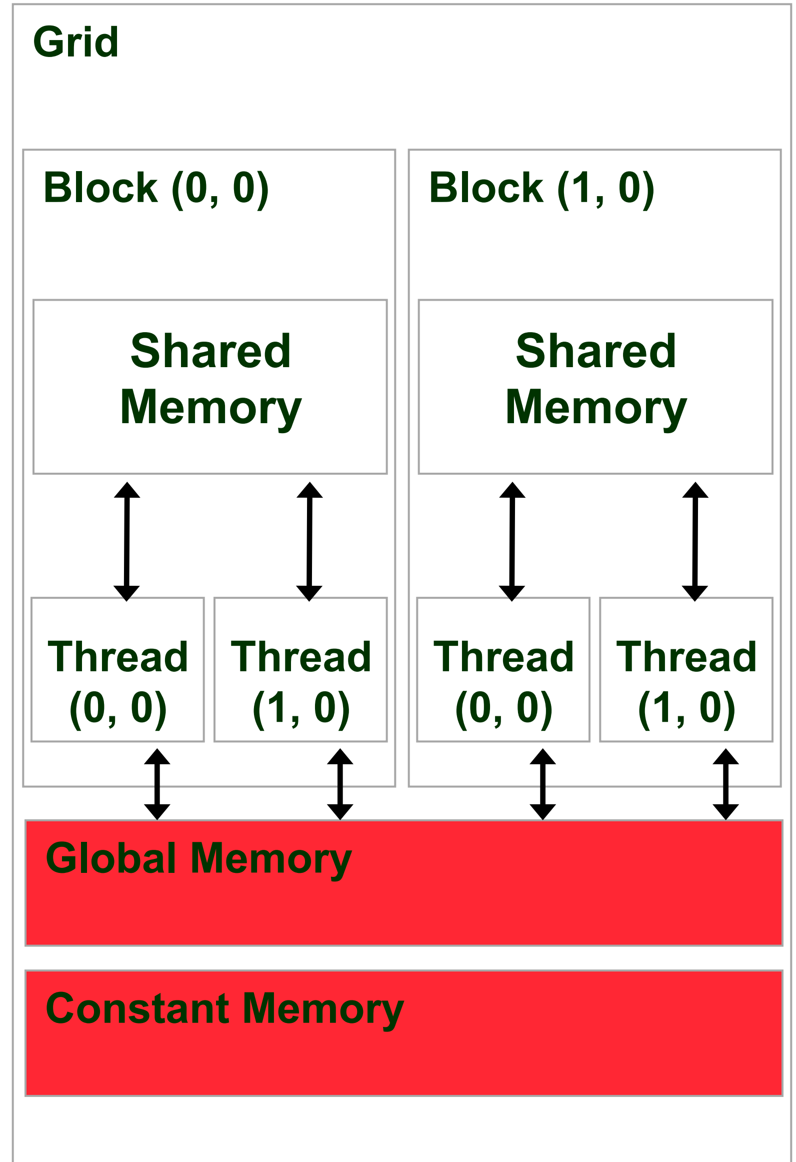
- 2 memory accesses (8 bytes) floating-point per multiply-add (2 FLOP)
- Assume peak arithmetic performance is 5 TFLOPs
- How many GB/s bandwidth to Global Memory?



But, actual bandwidth is much much lower!!!

Global memory bandwidth~300 GB/s

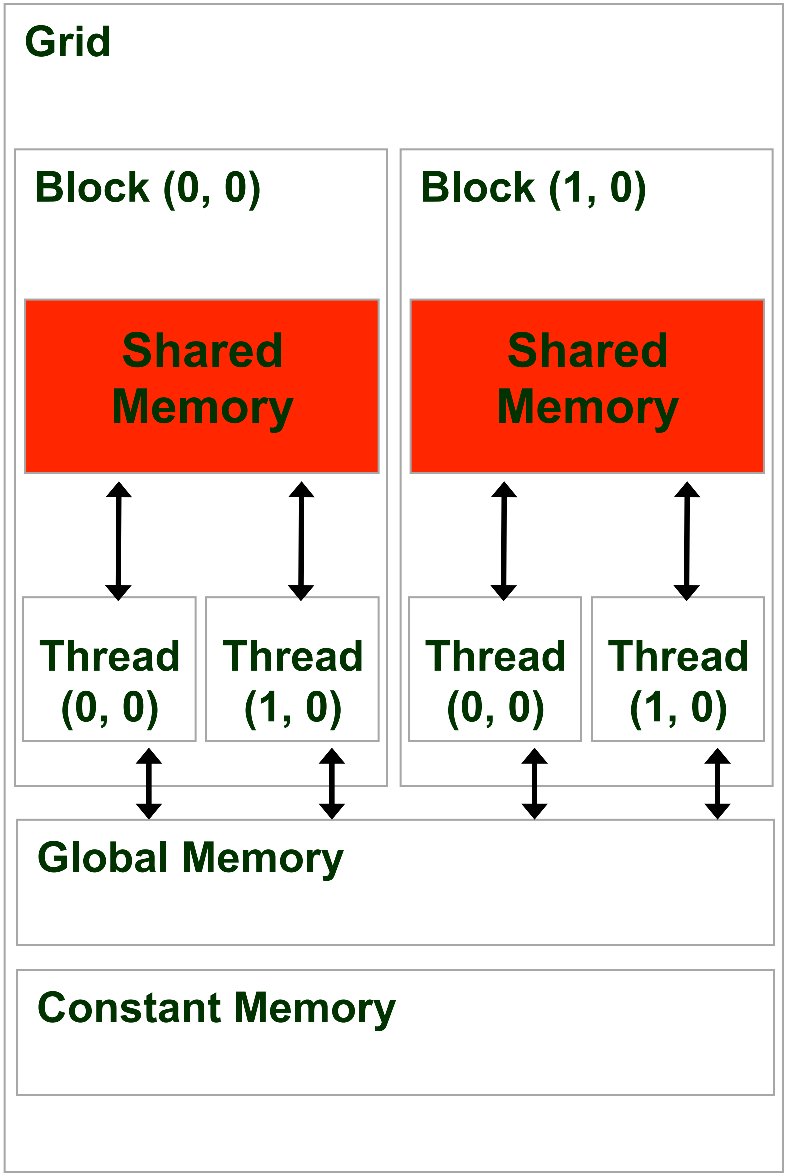
- How many FLOPS would our matrix multiply run at?
- How much slower is that than the peak bandwidth?
- What do we do????



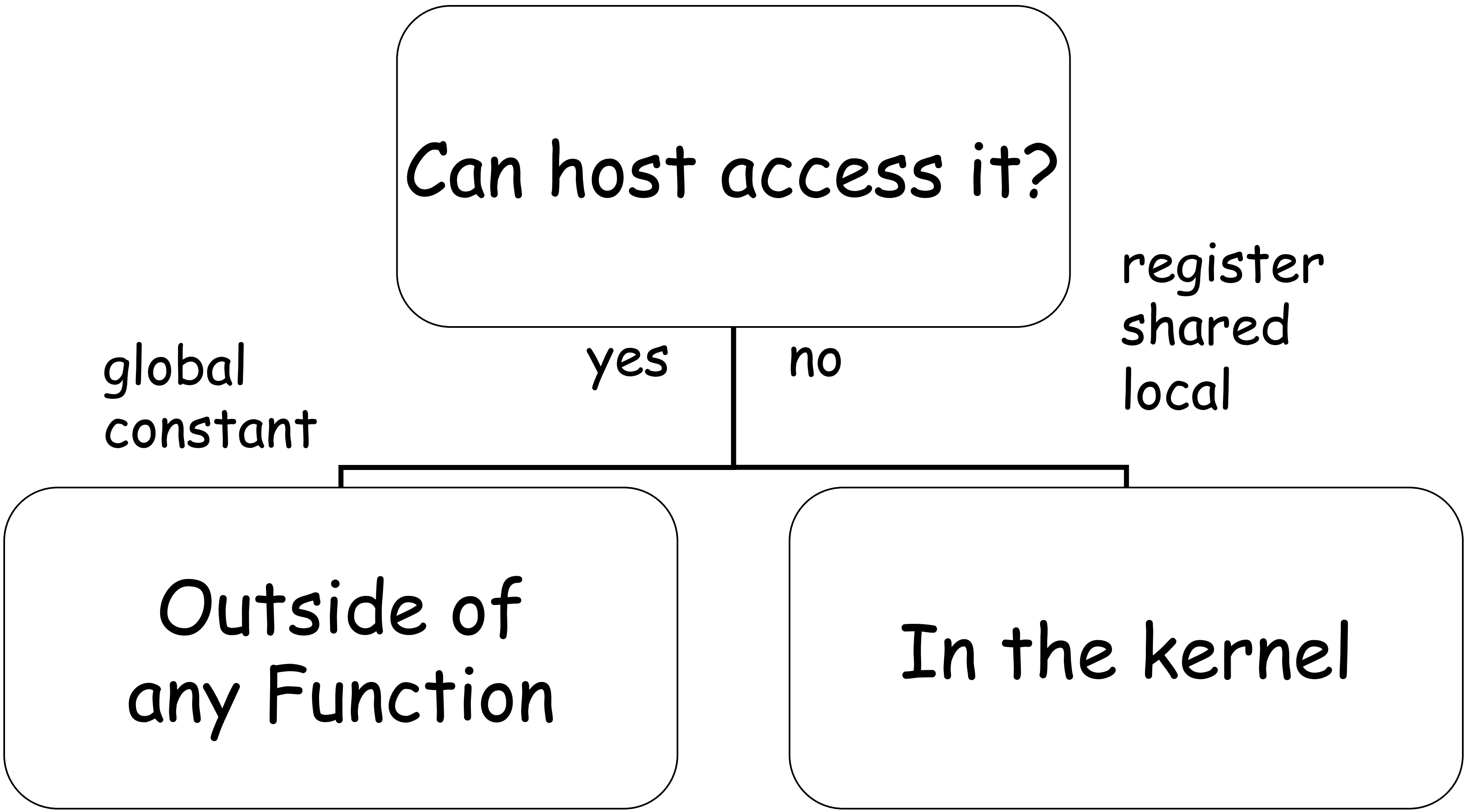
Use Shared Memory

- Global memory is DRAM (slow)
- Shared memory is on-chip (fast)

- Partition data into tiles that fit in shared memory
- Use the tiles in parallel
  - Load tile using multiple threads
  - Compute in parallel
  - Copy results back to global memory in parallel
- Compute in shared memory

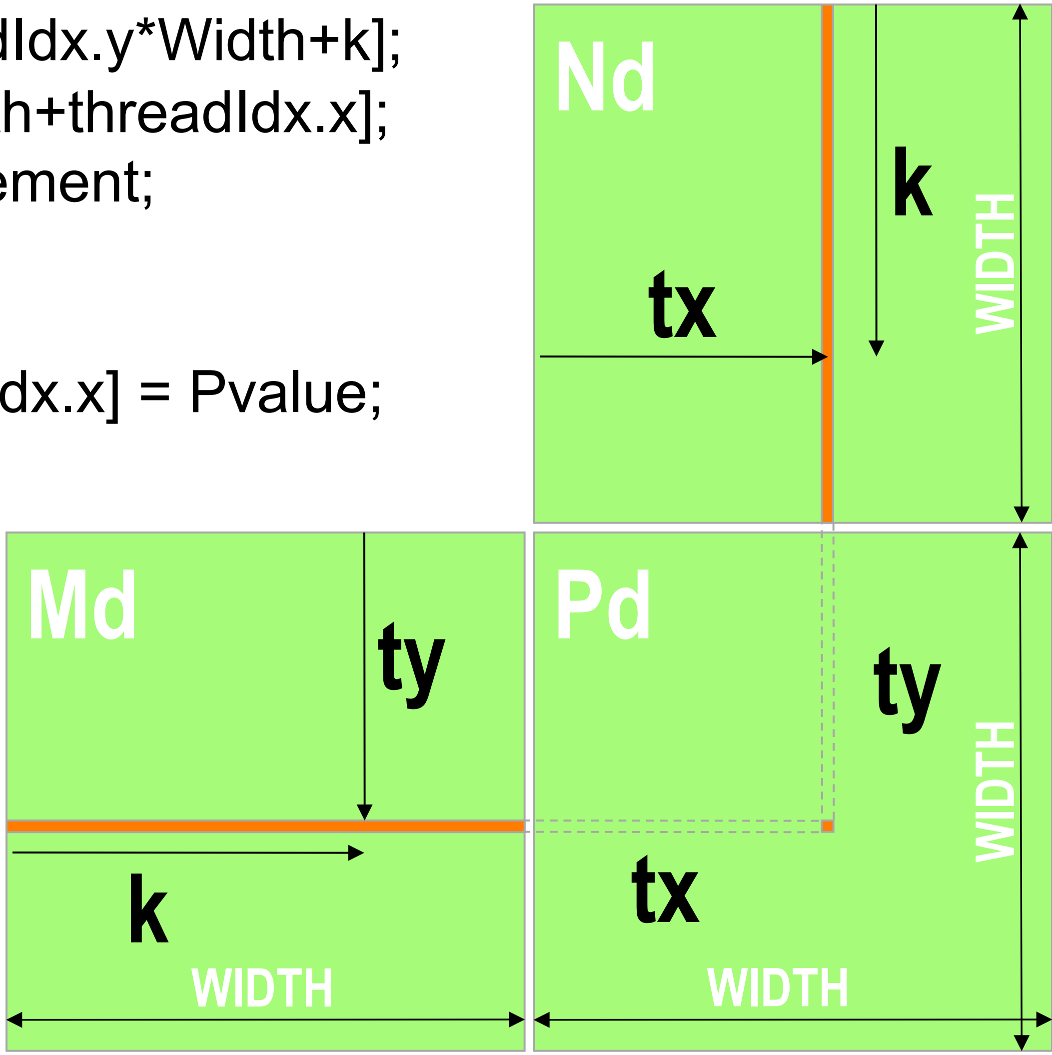


Where to Declare Variables?



Back to Matrix Multiply: Divide it into tiles

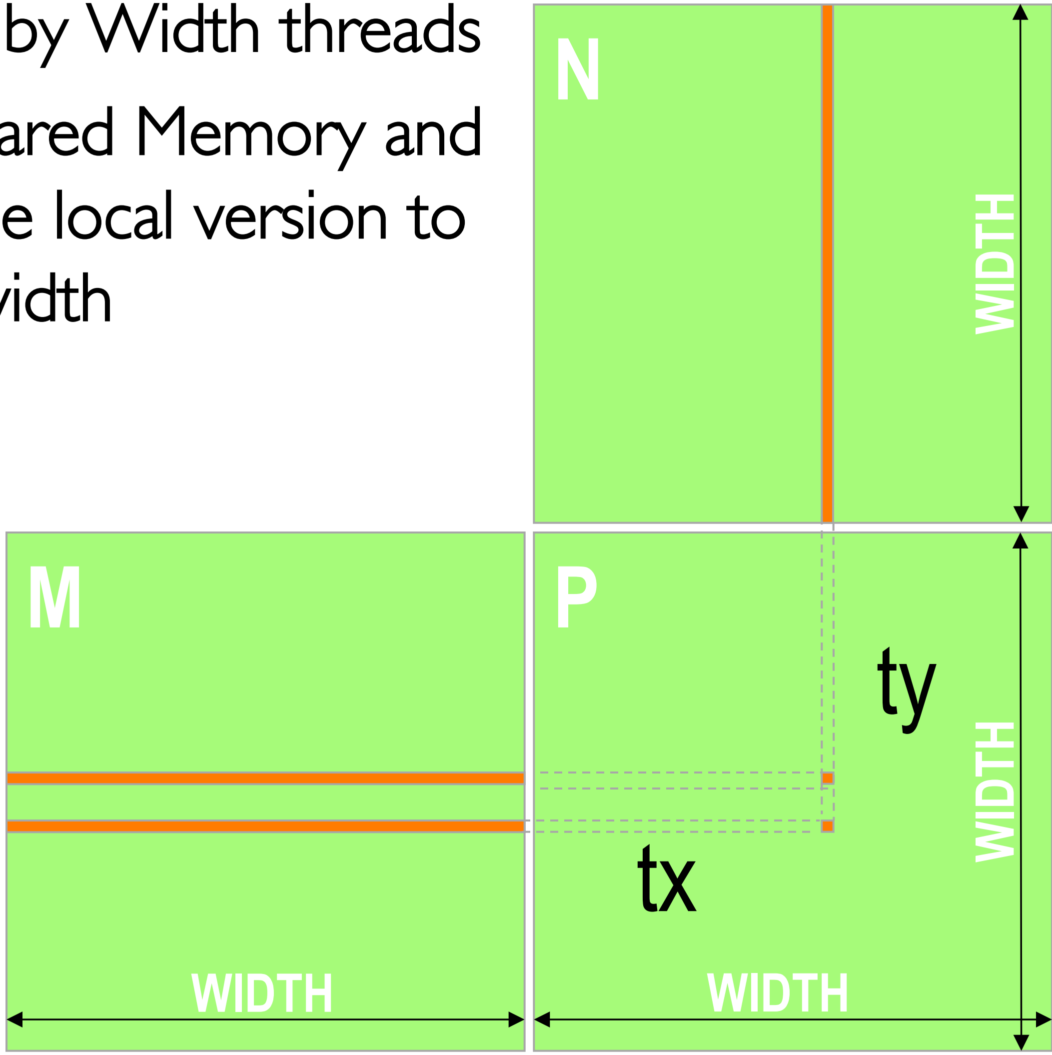
```
for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
 }

 Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



Idea: Use shared memory to reuse data

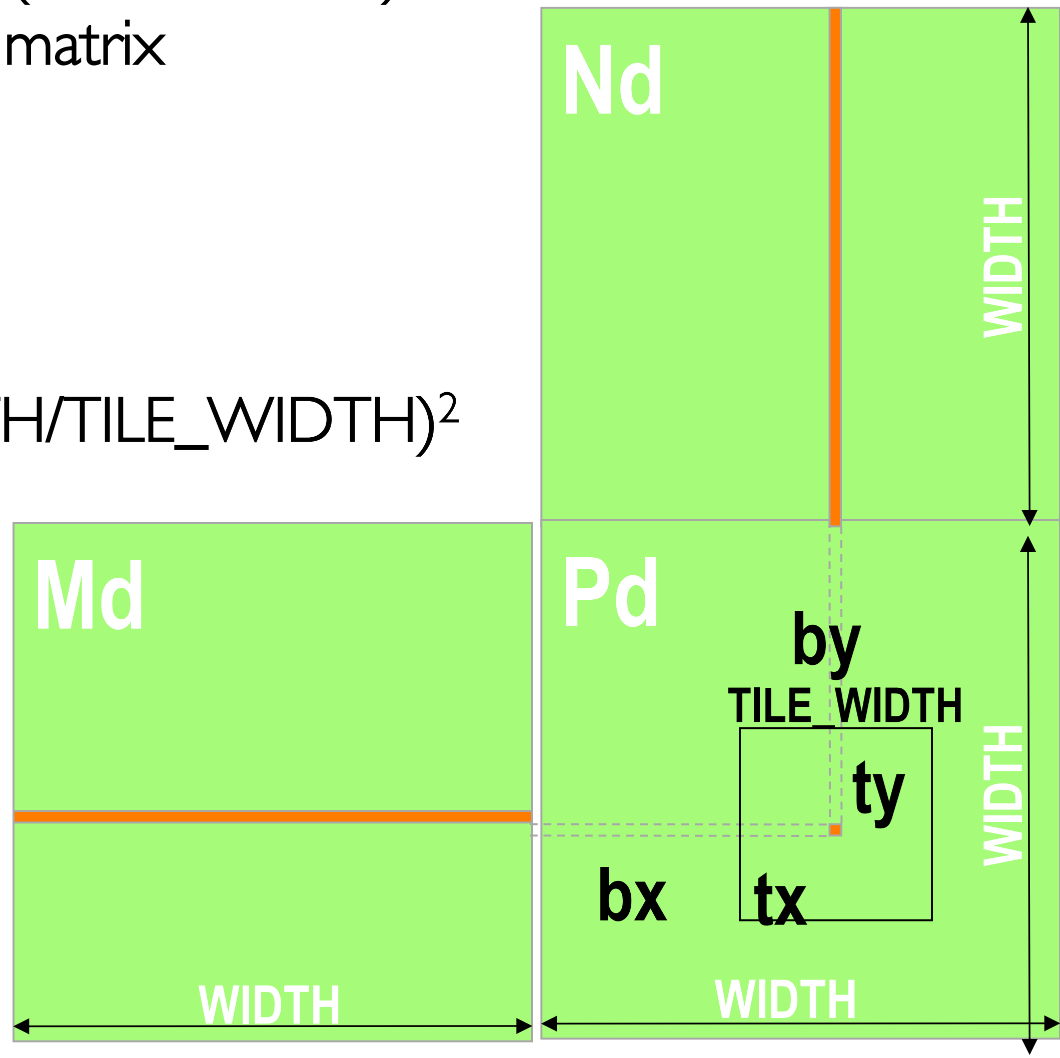
- Each input element is read by Width threads
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth

➔ Tiled algorithms



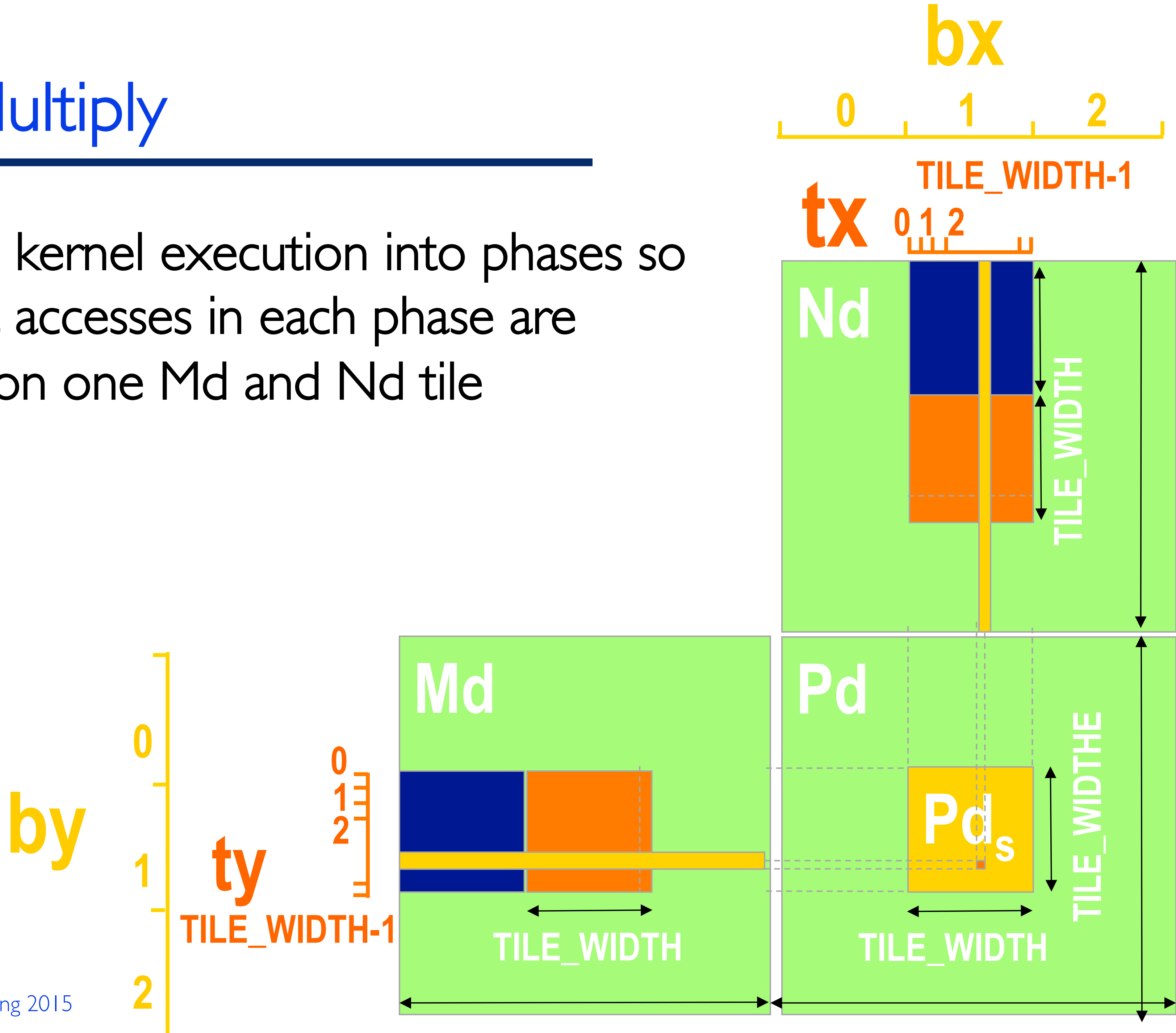
Handling Arbitrary Sized Square Matrices

- Each 2D block to compute a $(\text{TILE\_WIDTH})^2$ sub-matrix (tile) of the result matrix
- $(\text{TILE\_WIDTH})^2$ threads
- Generate 2D Grid of $(\text{WIDTH}/\text{TILE\_WIDTH})^2$ blocks

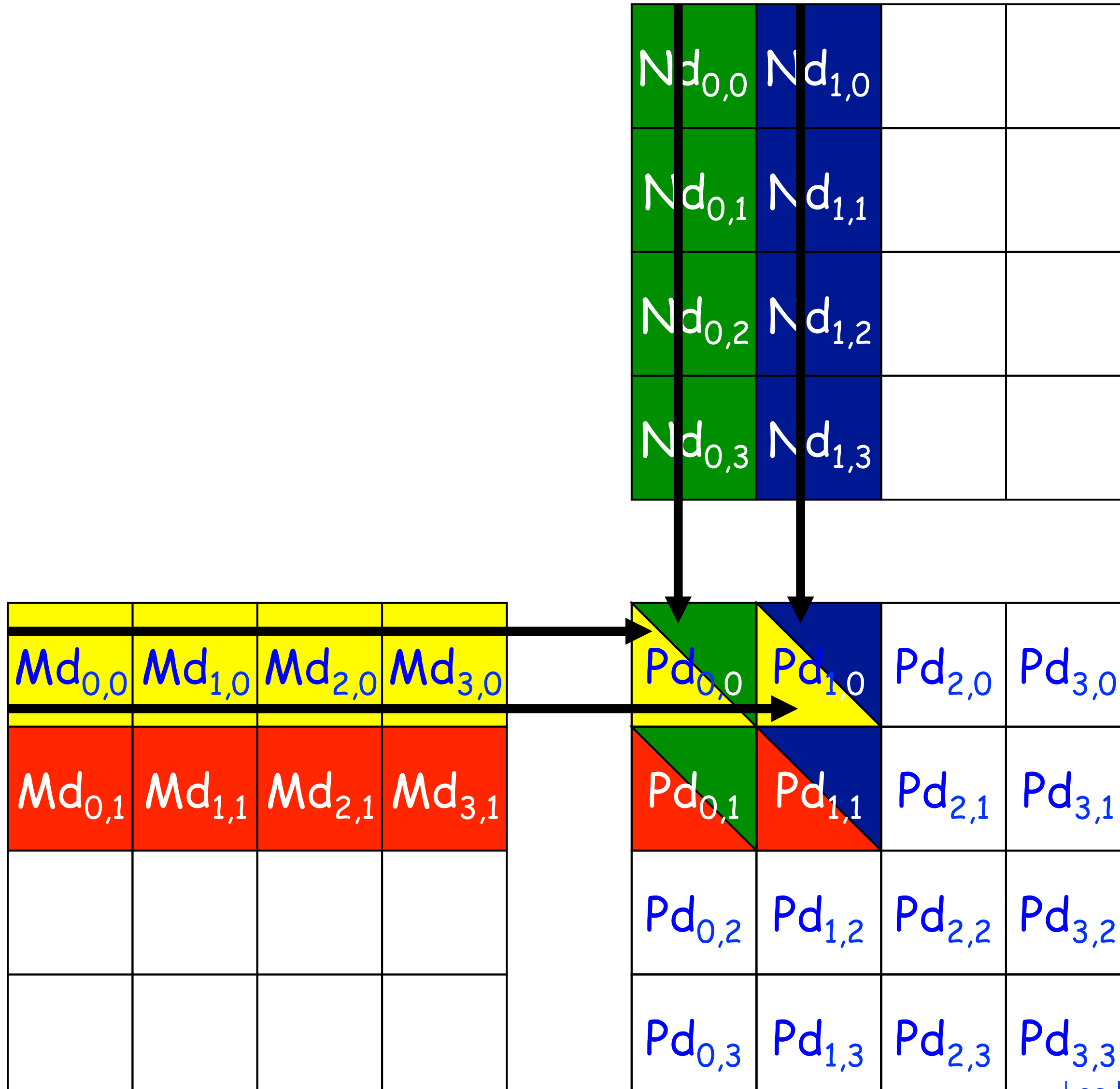


Tiled Multiply

Break up kernel execution into phases so that data accesses in each phase are focused on one Md and Nd tile



A Small Example



Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

$P_{0,0}$ thread$_{0,0}$	$P_{1,0}$ thread$_{1,0}$	$P_{0,1}$ thread$_{0,1}$	$P_{1,1}$ thread$_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

First-order Size Considerations

- Each thread block should have many threads
  - TILE_WIDTH of 64 gives 64*64 = 4096 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 16*16= 64 Thread Blocks
- Each thread block performs 2*4096 = 8192 float loads from global memory for 4096 * (2*64) = 524K mul/add operations
  - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

```
// Set up the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH, Width /  TILE_WIDTH);
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {

   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

   int bx = blockIdx.x;  int by = blockIdx.y; int tx = threadIdx.x; int ty = threadIdx.y;
   // Identify the row and column of the Pd element to work on
   int Row = by * TILE_WIDTH + ty; int Col = bx * TILE_WIDTH + tx;  float Pvalue = 0;
   // Loop over the Md and Nd tiles required to compute the Pd element
   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
   // Collaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
       __syncthreads();
   }
  Pd[Row*Width+Col] = Pvalue;
}
```


Tiled Kernel

Must sync threads when loading/computing

- All threads load tile together
- All thread compute together
- But, loading & computing can not be overlapped!
  - Why not?

- How do we keep them apart?
- Barrier synchronization
  - __syncthreads()
  - Also, called "barrier" synchronization
  - All threads reach barrier, wait for others, then continue

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {

  __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y; int tx = threadIdx.x; int ty = threadIdx.y;
  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty; int Col = bx * TILE_WIDTH + tx;  float Pvalue = 0;
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
  // Coolaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
        __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += Mds[ty][k] * Nds[k][tx];
       __syncthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```


Tiled Kernel

Shared Memory Bandwidth with 64x64 tiles

- Each core has 96KB shared memory
  - Size is implementation dependent!
  - Assume TILE_WIDTH = 64
  - Each GPU block holds a tile (64x64)
  - We share elements along TILE_WIDTH (for M and N)
  - Assuming 20 TB/s
  - How much do we cut the required bandwidth?
  - How many tiles can we fit?

Shared Memory Bandwidth with 128x128 tiles

- Each core has 96KB shared memory
  - Size is implementation dependent!
  - Assume TILE_WIDTH = 128
  - How much does memory bandwidth improve?

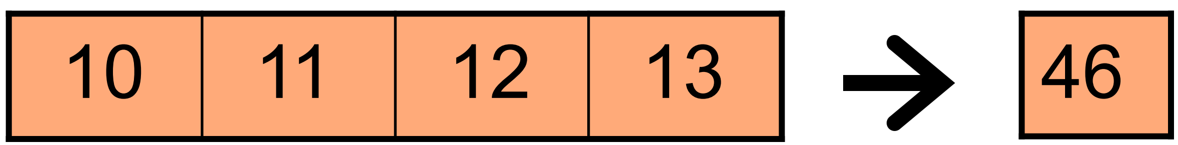
Shared Memory Bandwidth

- Each core has 96KB shared memory
  - Size is implementation dependent!
  - Assume TILE_WIDTH = 64, each block uses 2*4096*4B = 32KB
  - Can have up to 3 Thread Blocks actively executing
  - 3*8192= 24K pending loads. (2 per thread, 4096 threads per block)

- 64x64 tiling reduces accesses to the global memory by 64x
  - 300 GB/s bandwidth can now support (300/4)*64 = 4.8 TFLOPS!

Reduction Operations

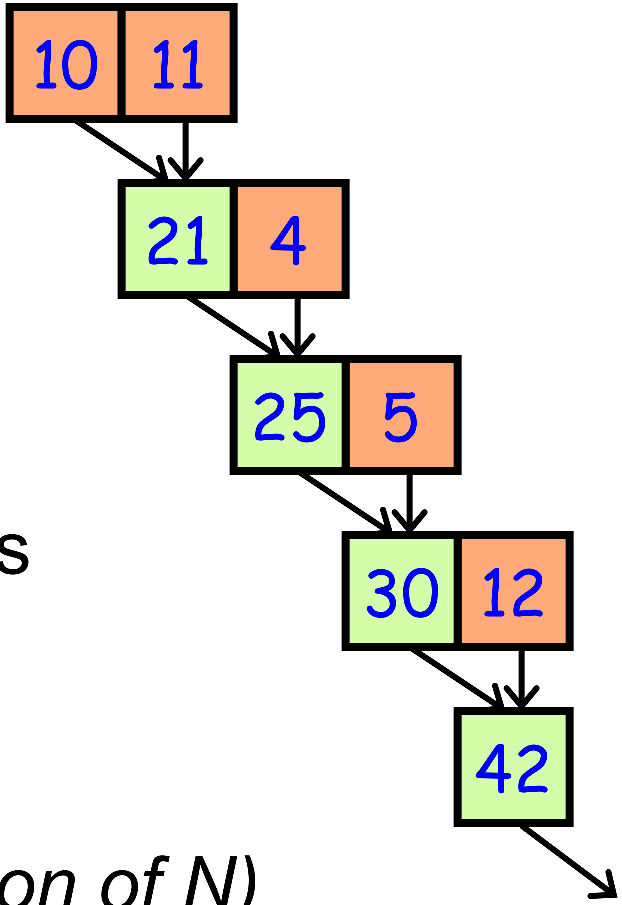
- Multiple values are reduced into a single value
  - ADD, MUL, AND, OR, ...



- Useful primitive

- Easy enough to allow us to focus on optimization techniques

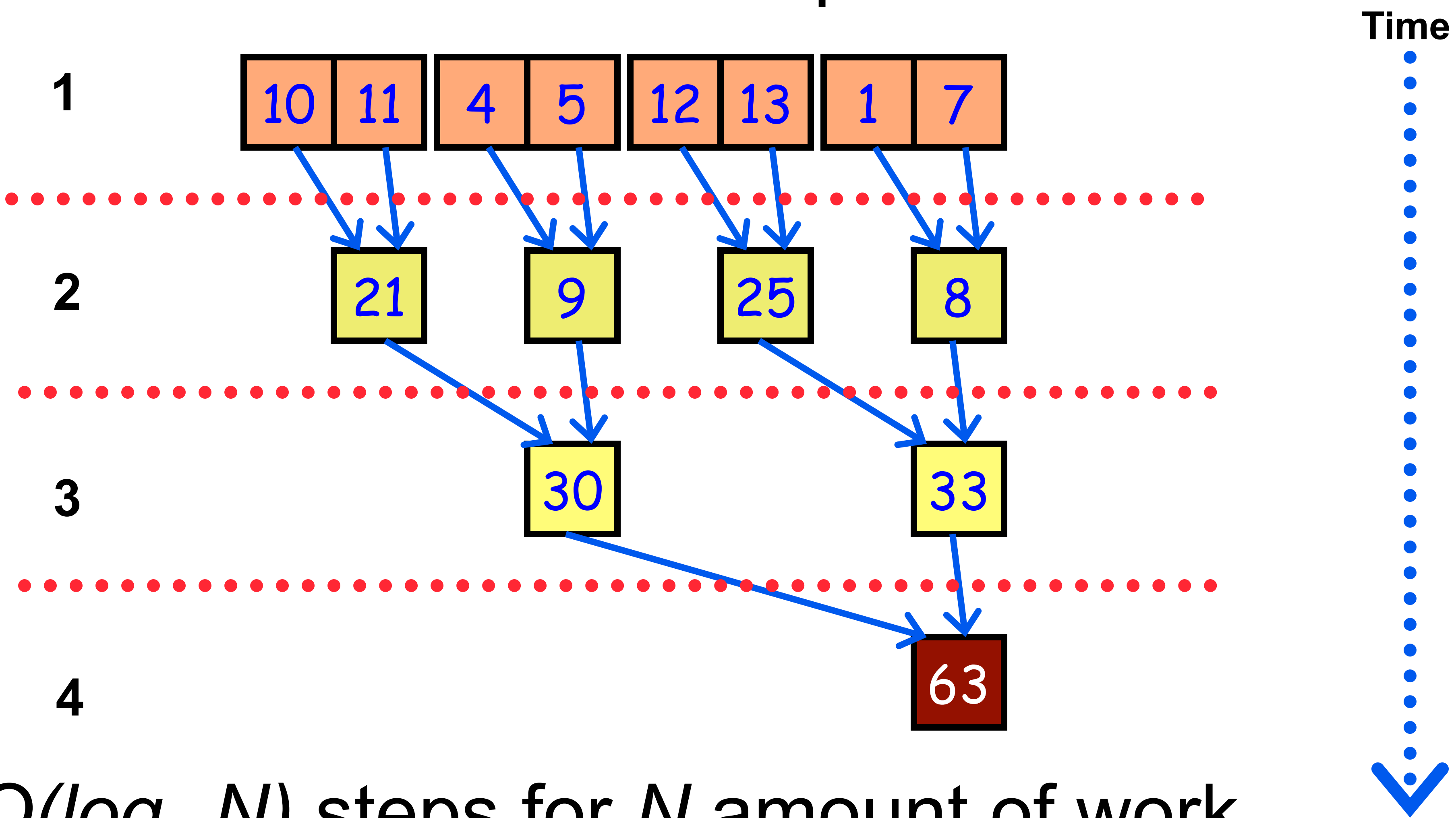
Sequential Reduction



- Start with the first two elements
  ➔ partial result
- Process the next element
- O(N) *(i.e., runtime linear function of N)*

Parallel Reduction

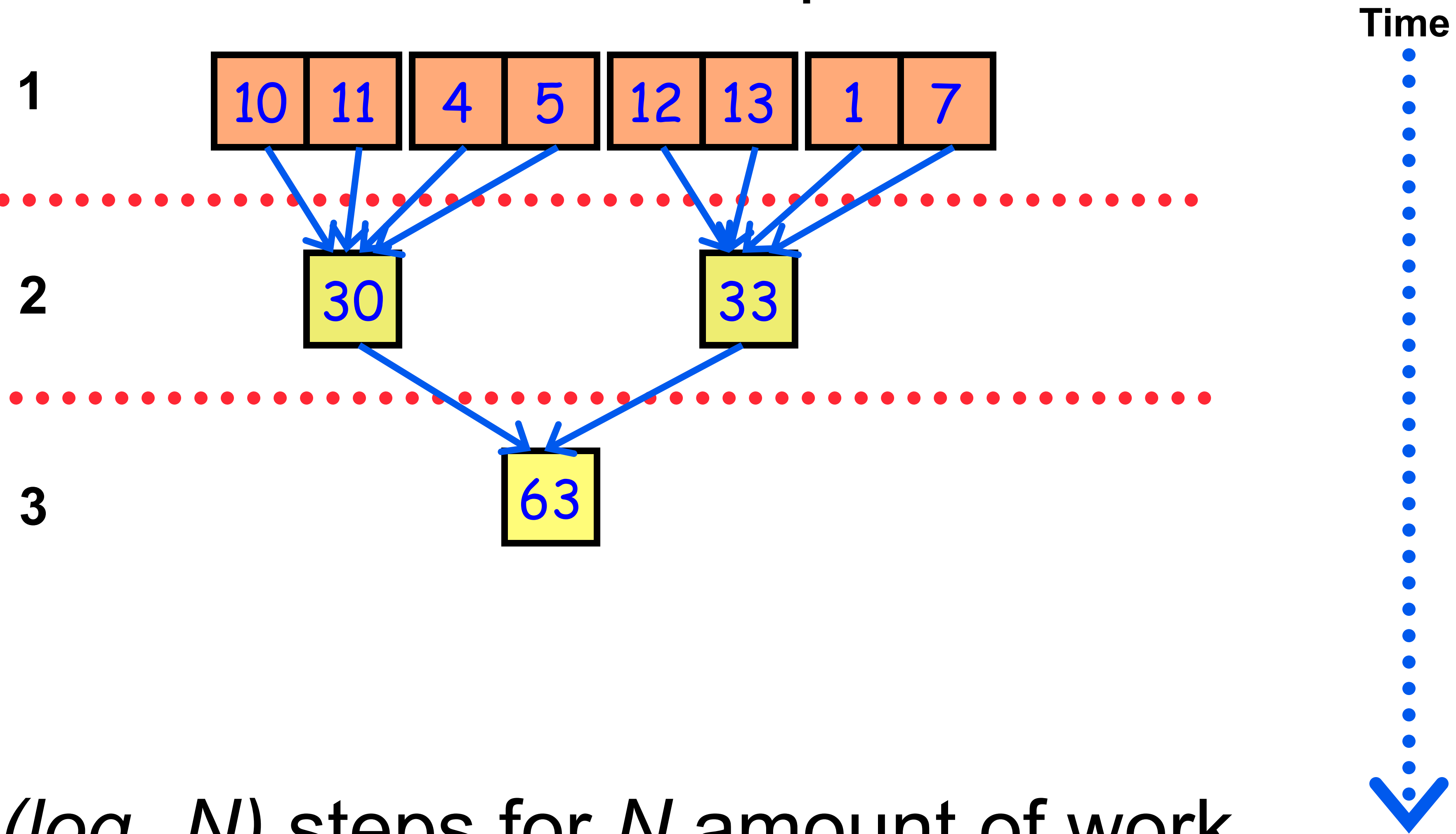
Pair-wise reduction in steps – Tree-like



$O(\log_2 N)$ steps for N amount of work

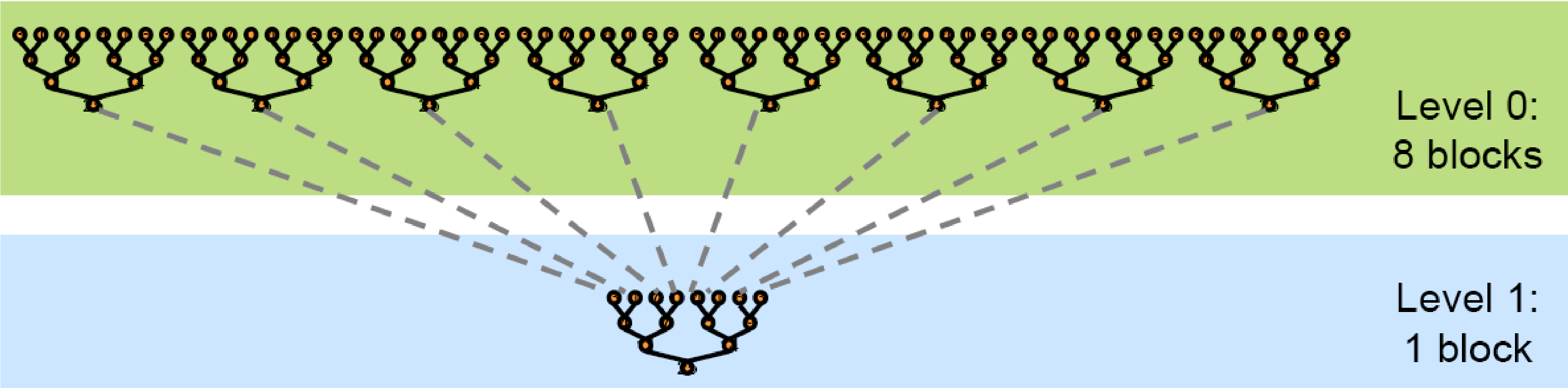
Different-degree trees possible

Pair-wise reduction in steps – Tree-like



$O(log_2\ N)$ steps for N amount of work

Reduction: Big Picture

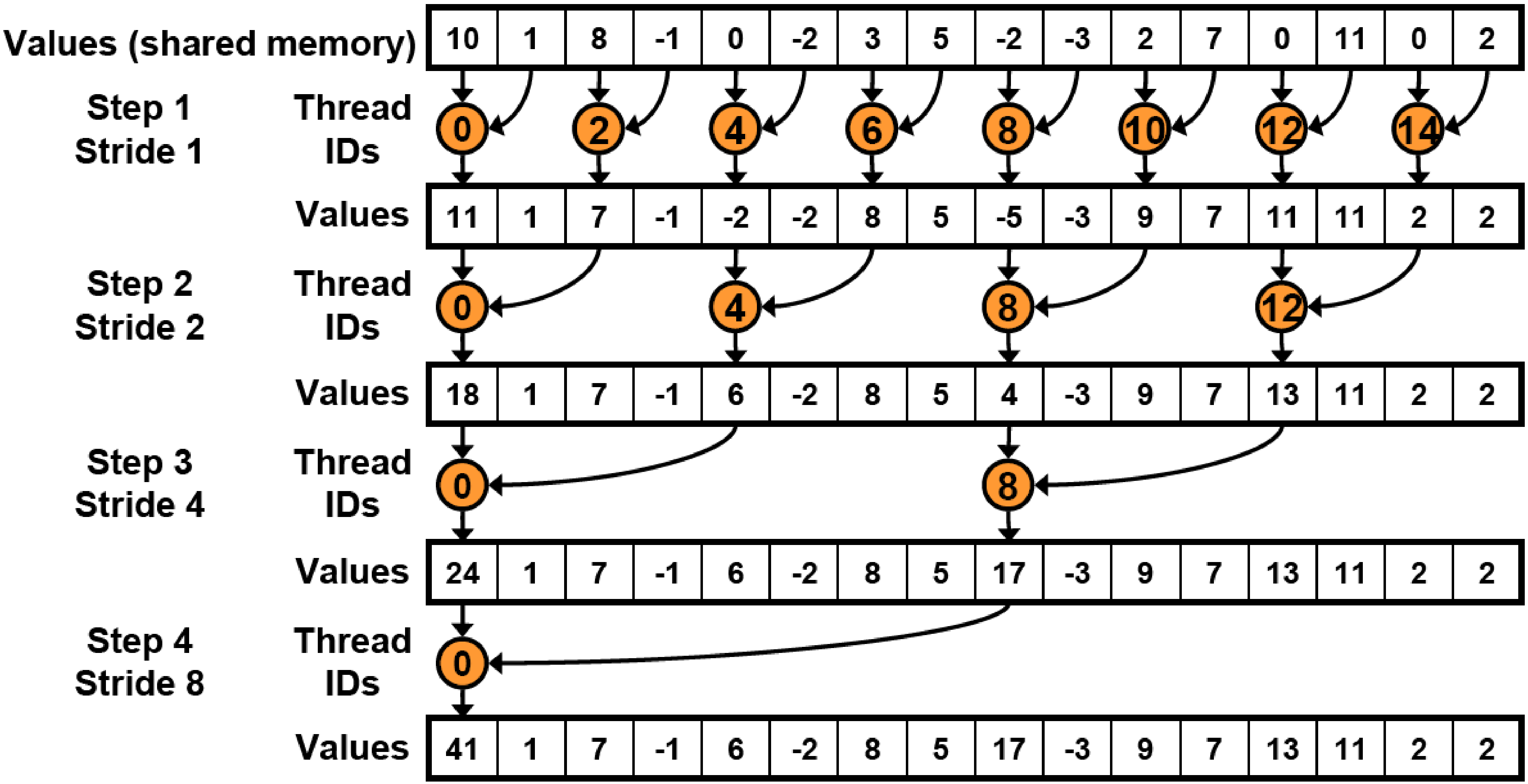


- The code for all levels is the same
- The same kernel code can be called multiple times
- Caveat: still a highly sequential operation
  - Do not expect 100x speedup with a few elements/thread

Reduction Kernel #1: Strategy

- Each thread loads one element into shared memory
- Reduce: Proceed in log*N* steps
  - In each step, half of the threads are active, reducing two elements
- Terminate: when one thread left
- Last thread writes back to global memory

Reduction Steps



Reduction Kernel #1: Interleaved Accesses

```
__global__ void reduce0(int *g_idata, int *g_odata, int n) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Allocating Shared Memory

```
__global__ void reduce0(int *g_idata, int *g_odata, int i) {
    extern __shared__ int sdata[];
```

- How many elements in sdata?
- Specify when calling the kernel:
  - `reduce0<<<blocks, threads, smemSize>>>(in, …`

Performance for Kernel #1

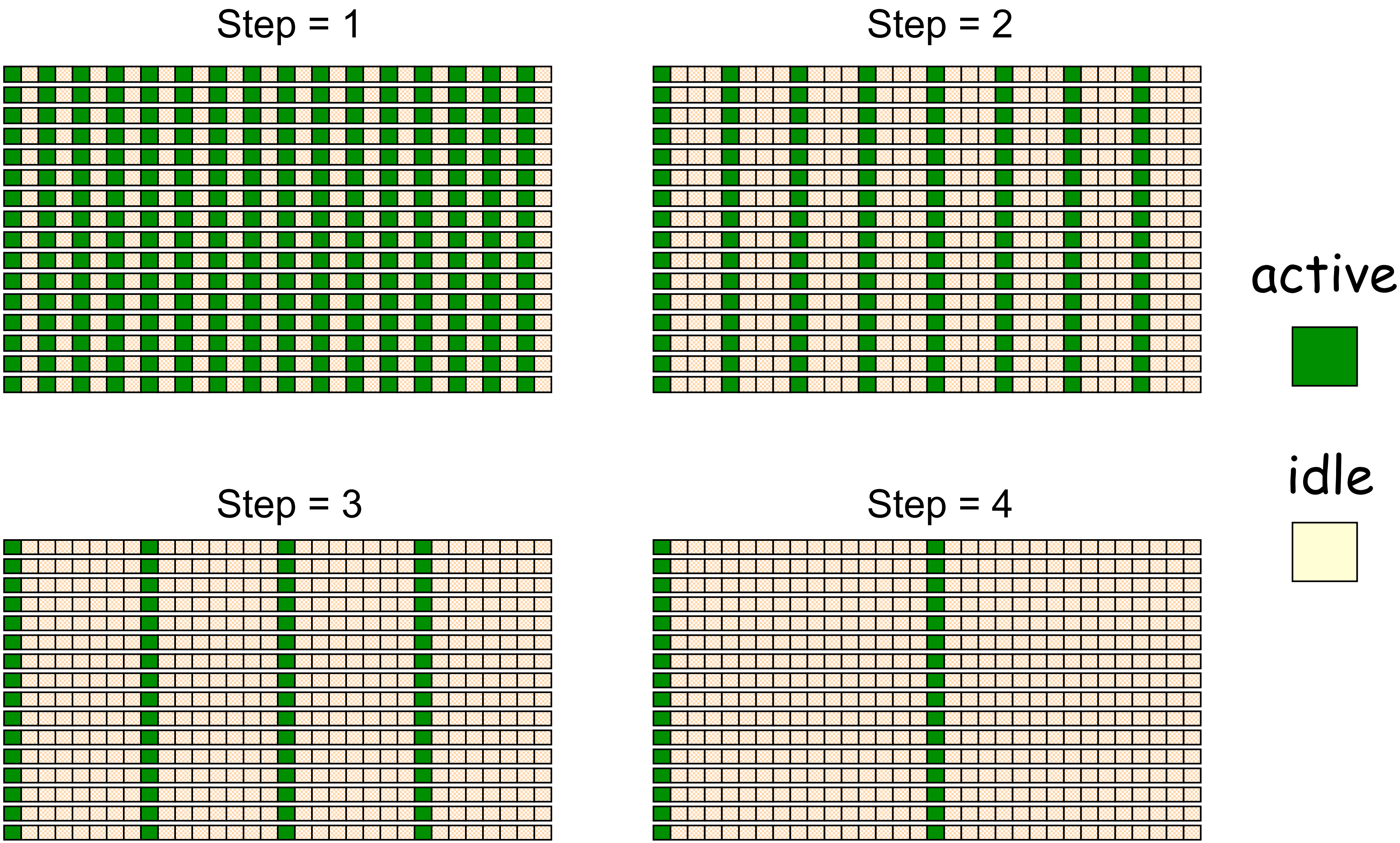
	Time (2^{22} ints)
Kernel 1: interleaved addressing with divergent branching	**4.25ms**

Reduction Kernel #1: Interleaved Accesses

```
__global__ void reduce0(int *g_idata, int *g_odata, int n) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Highly divergent code leads to poor performance

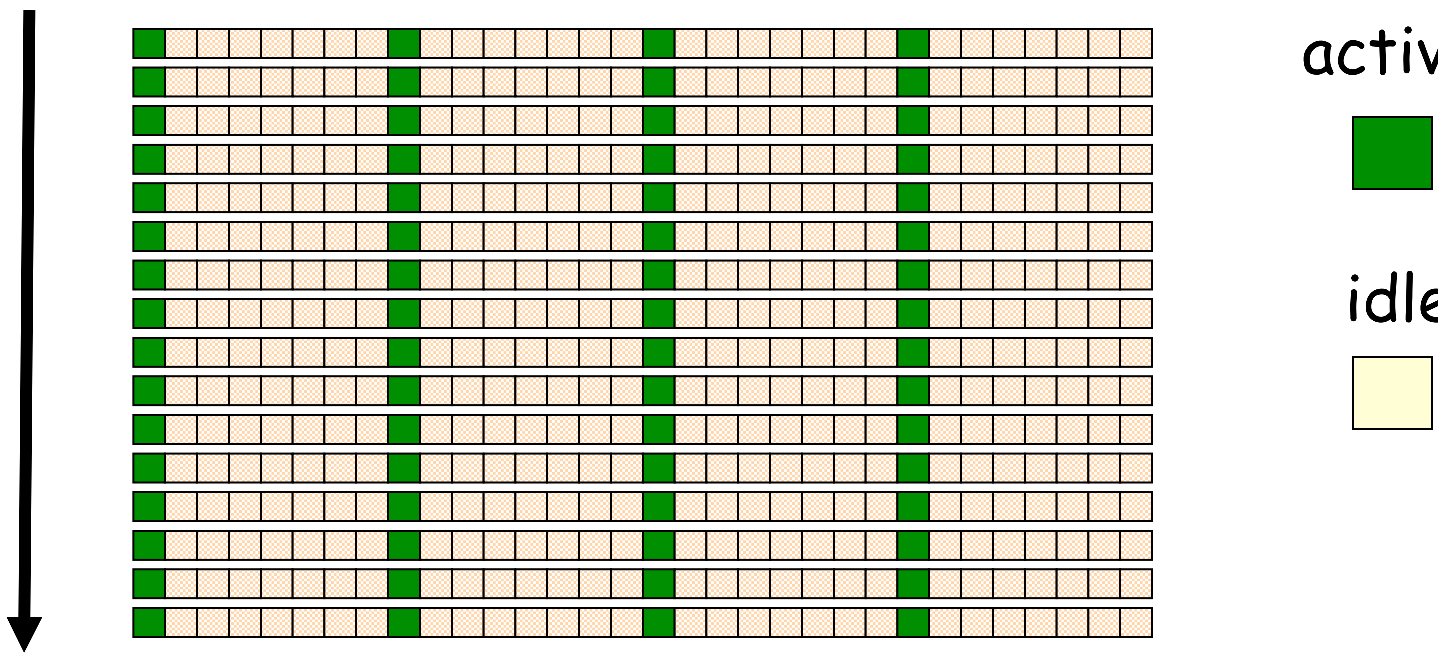
Lots of idle threads!



How will these run?

together
this way

Good!



Step = 3

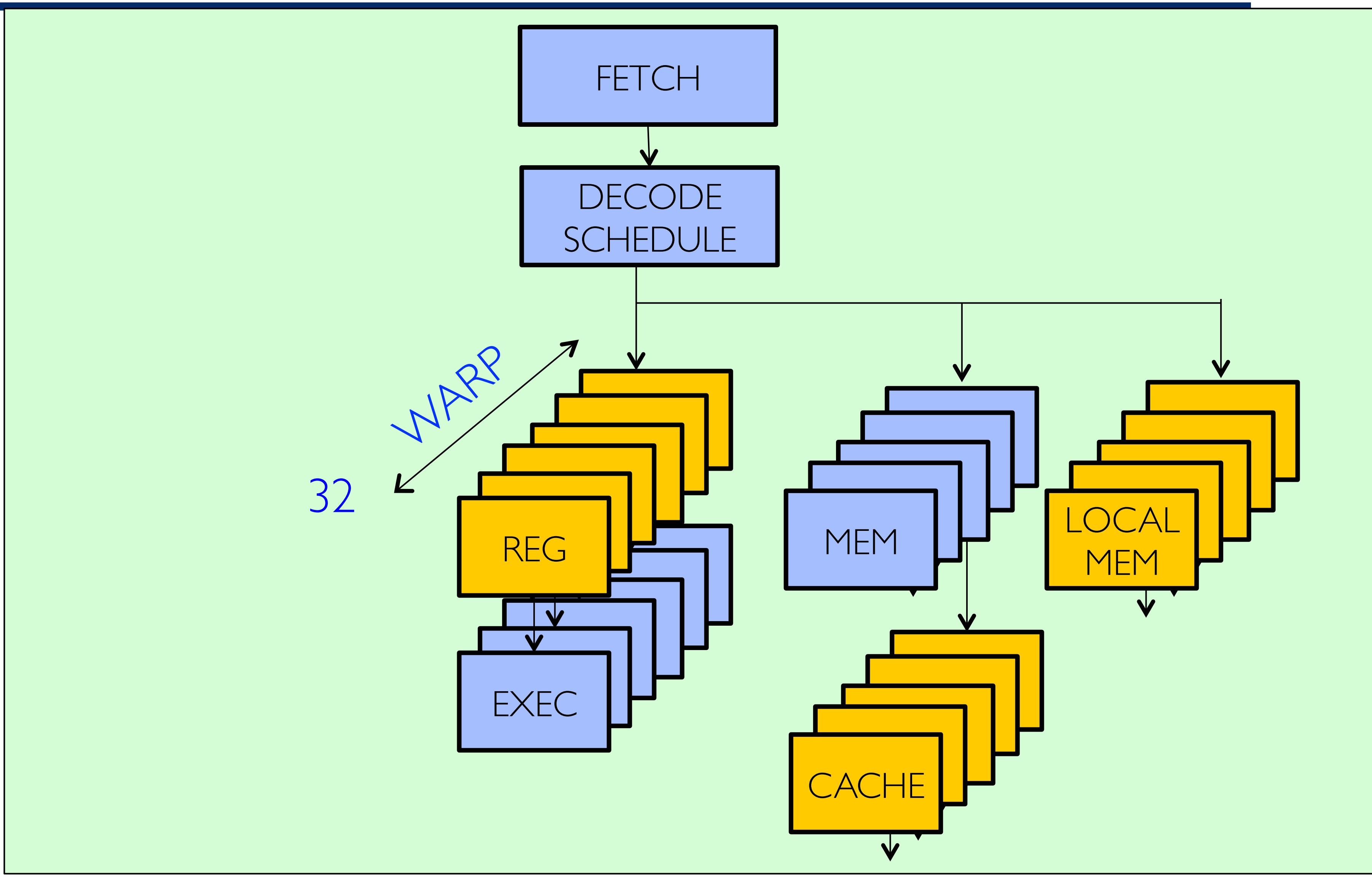
How will these run?

Step = 3

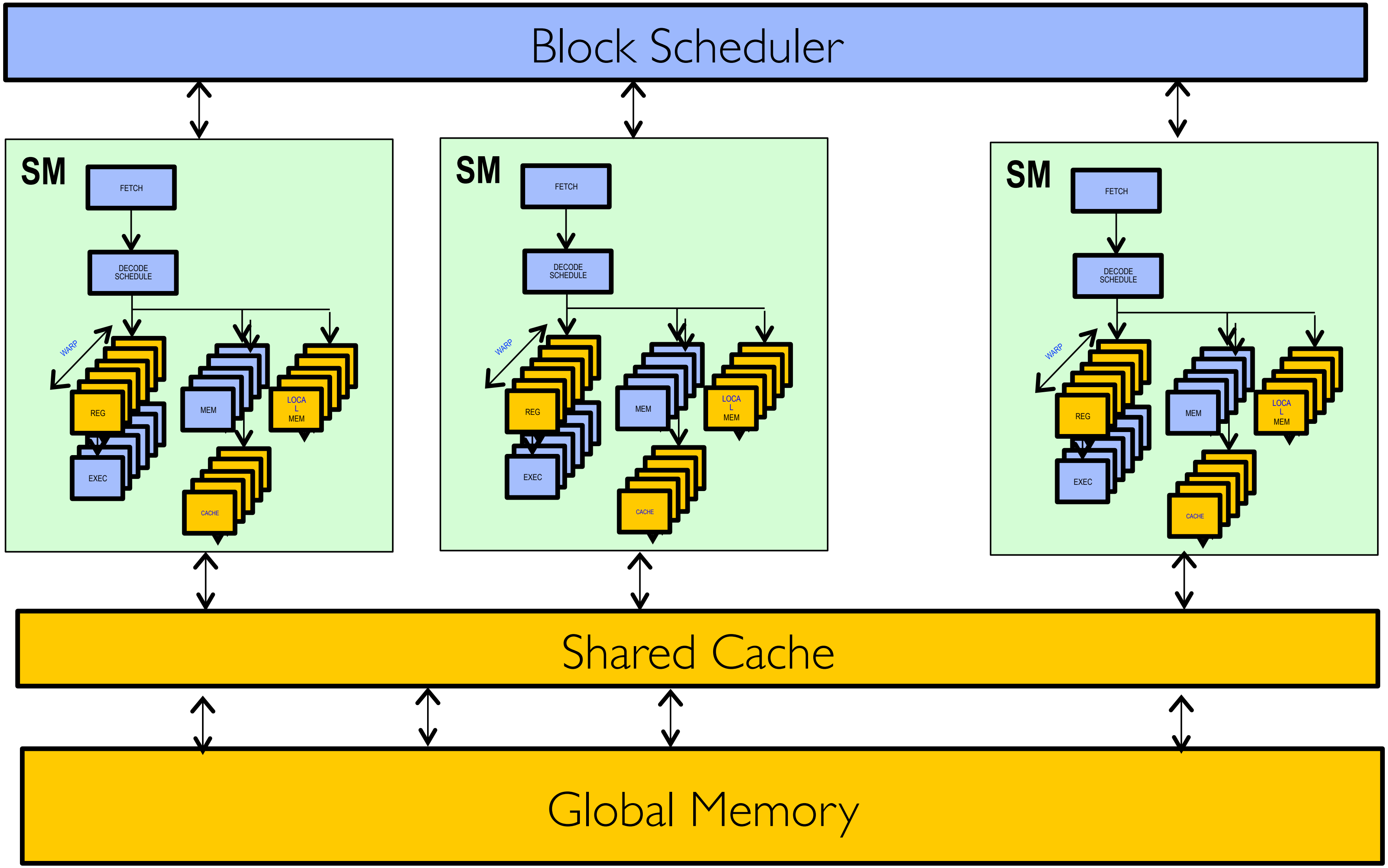
Need some control over scheduling

- No order among threads in a block
- But, threads are grouped to run together
- The grouping is called a "warp"
- Warp grouping follows sequential thread id

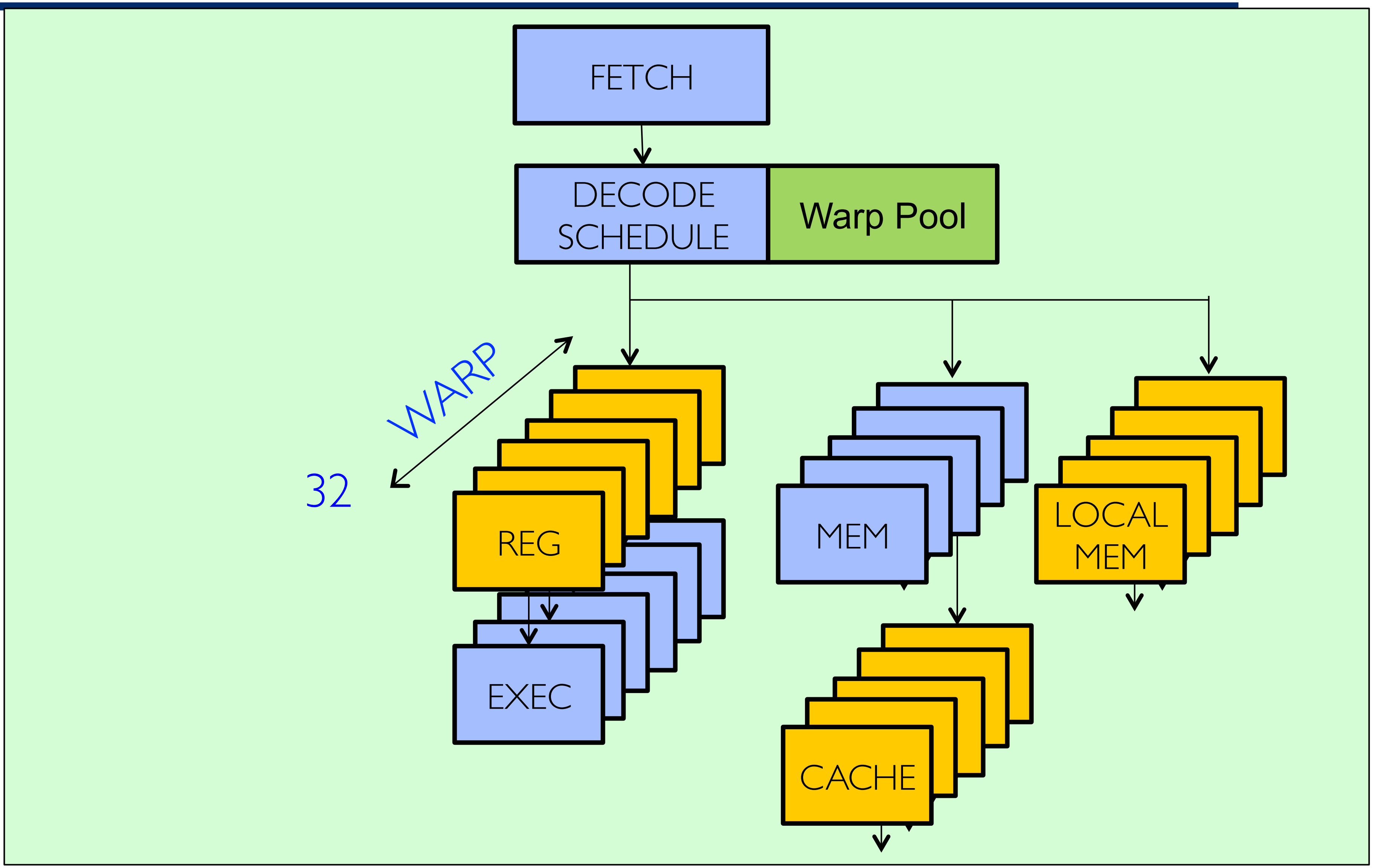
GPU Core: Streaming Multiprocessor (SM)



GPU Multicore: SM's connected via memory



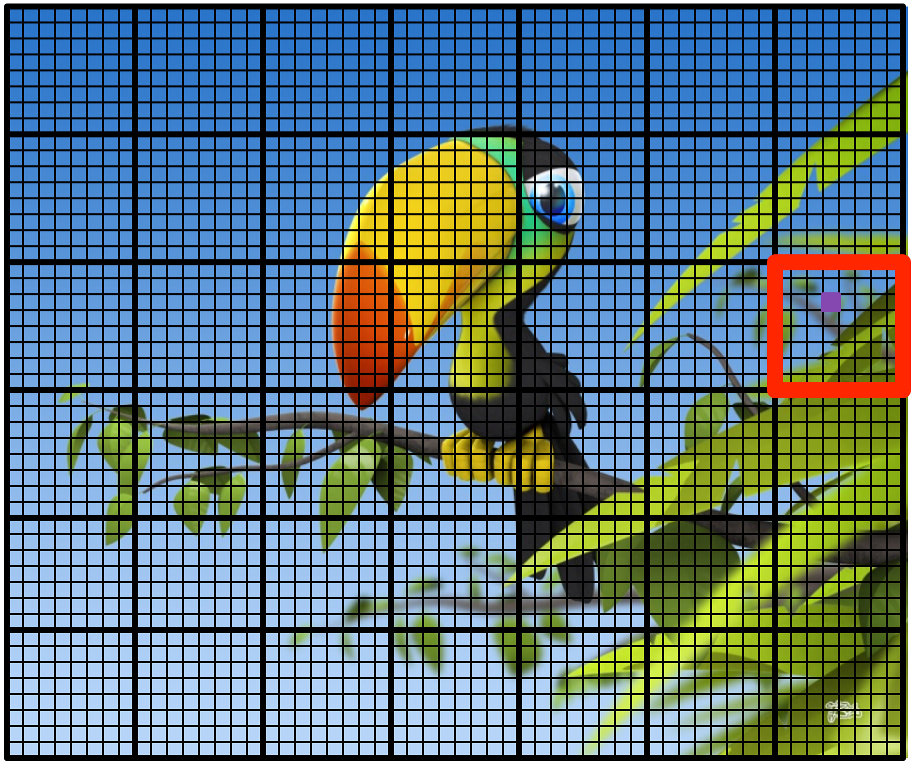
GPU Core – Streaming Multiprocessor (SM)



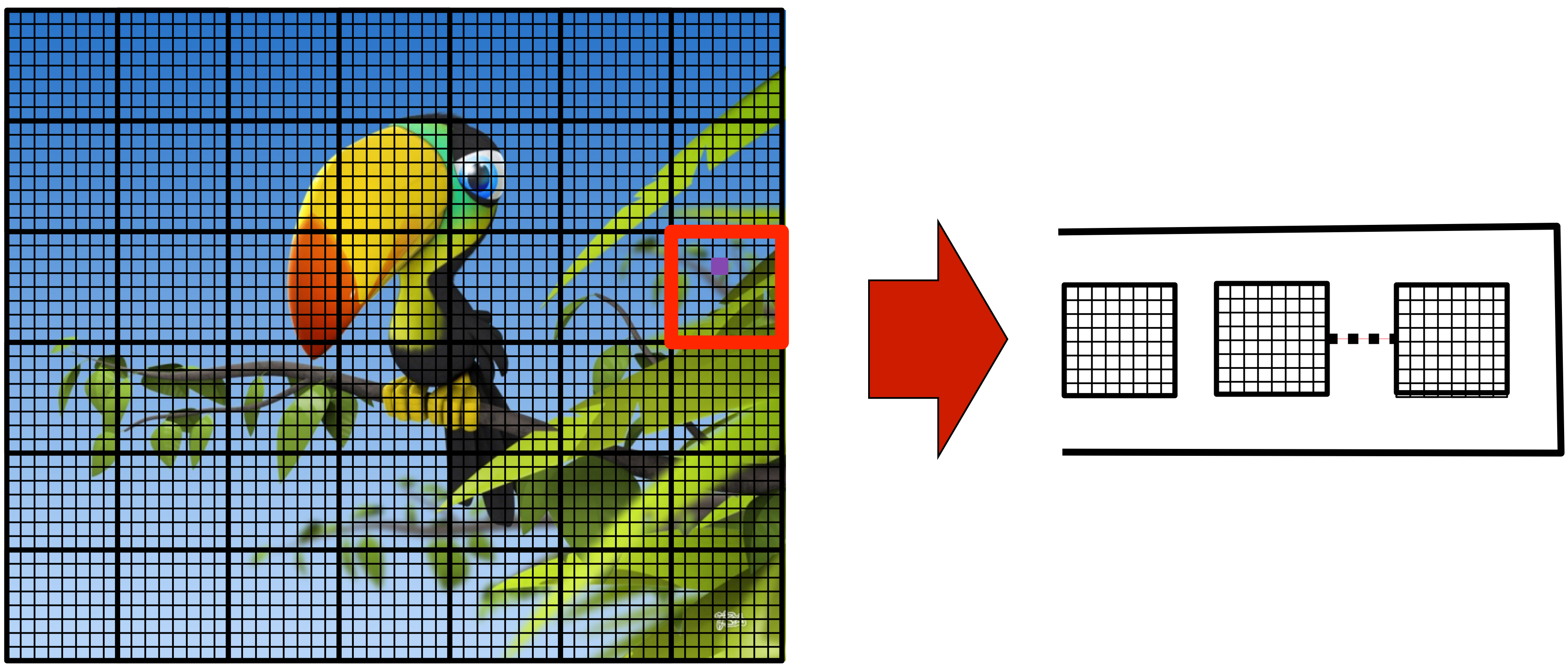
Fade example

- Each thread will process one pixel

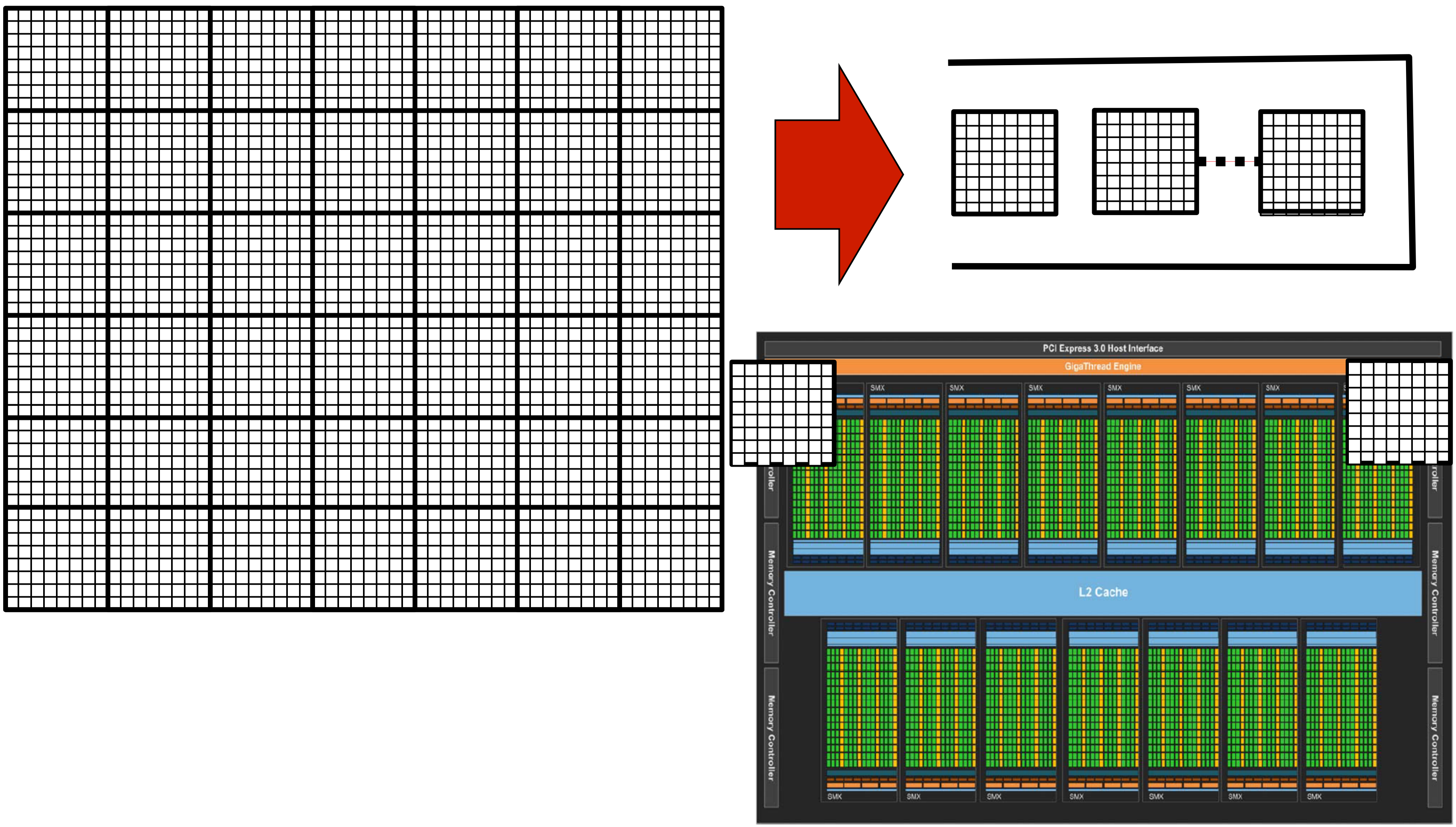
```
for all elements do in parallel
    a[i] = a[i] * f;
```



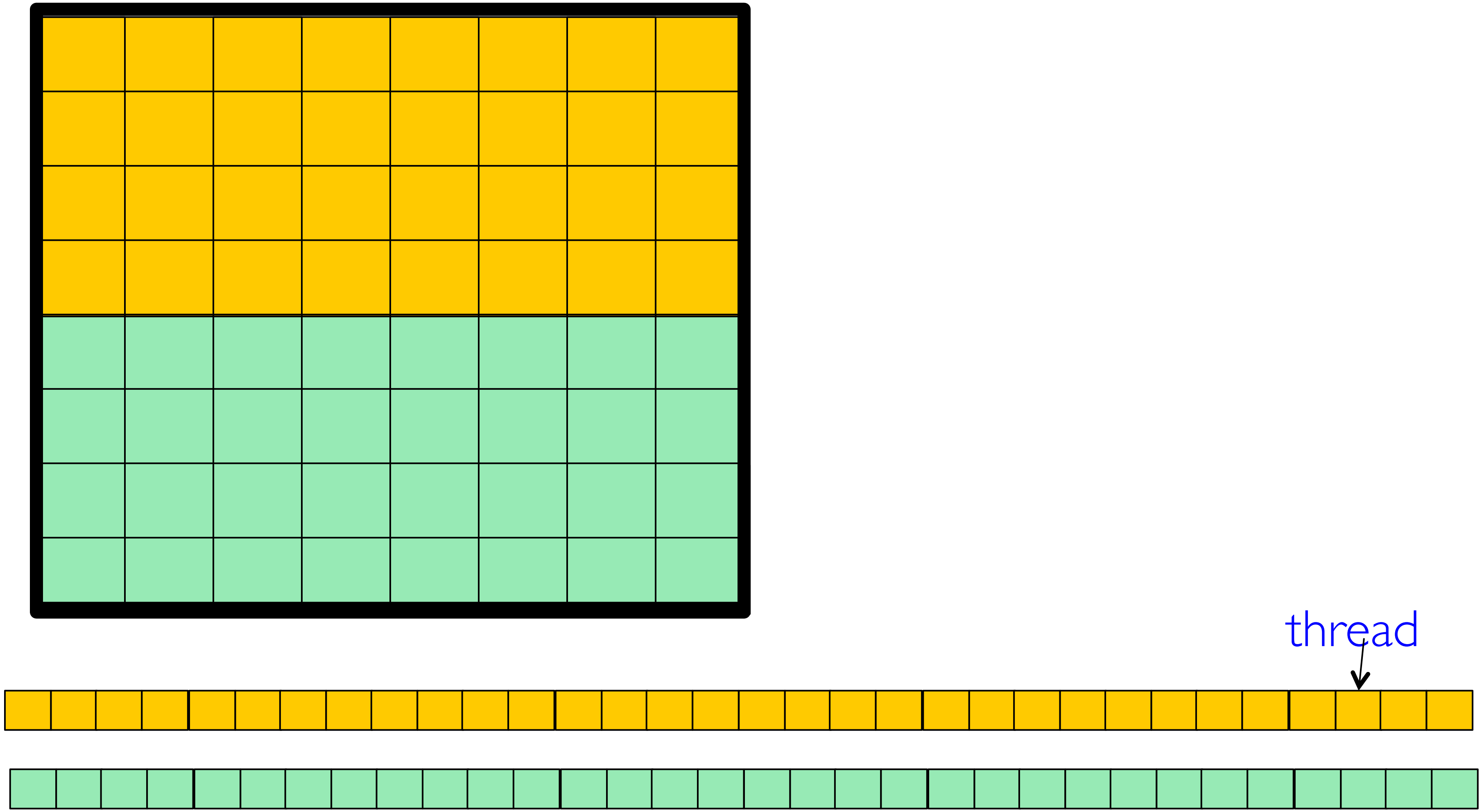
Decompose into blocks



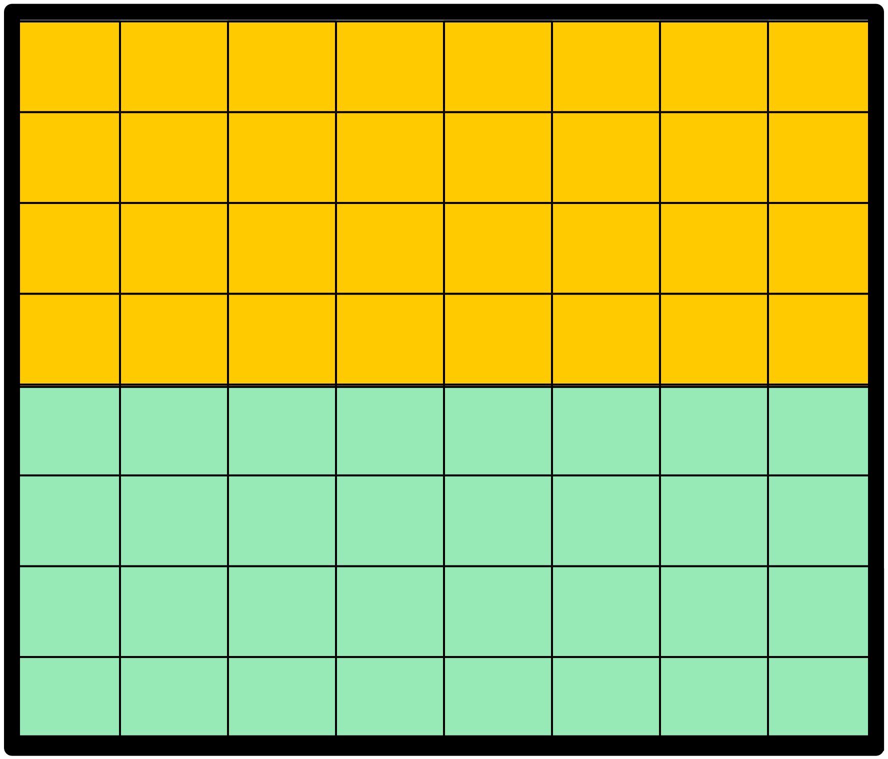
Assign each block to a core (SM)

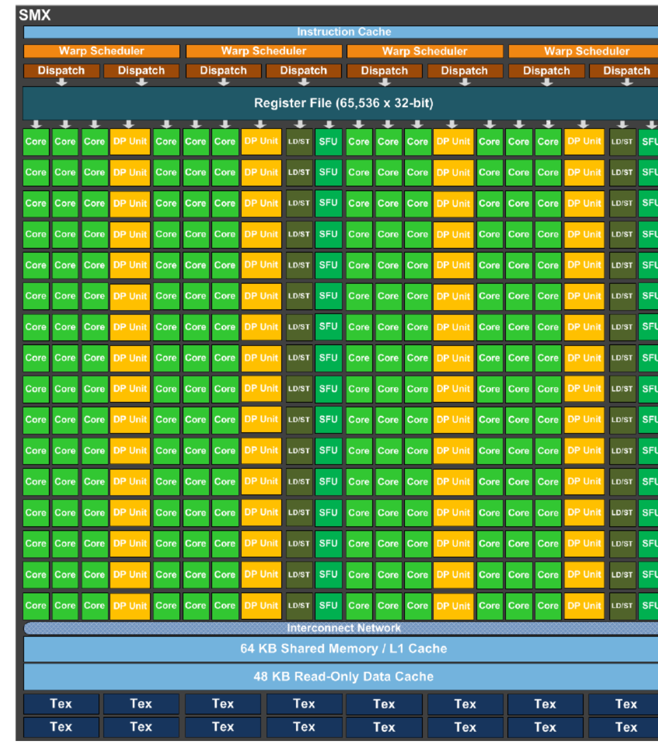


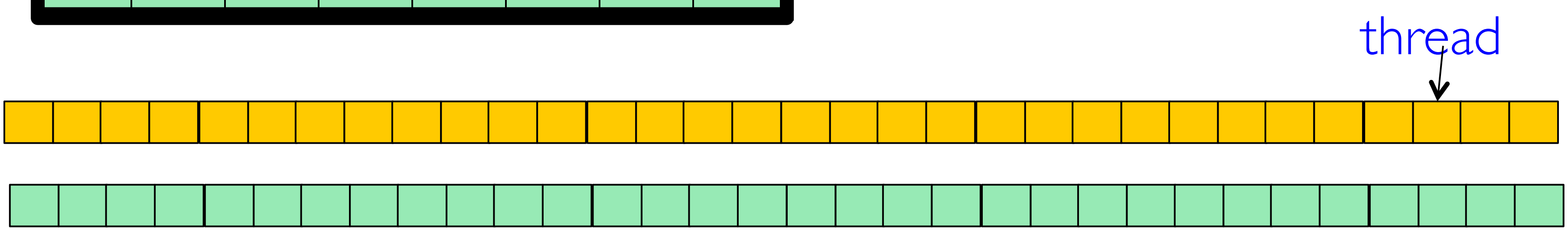
Decompose a Block into Warps



Execute Warps onto cores (SMs)

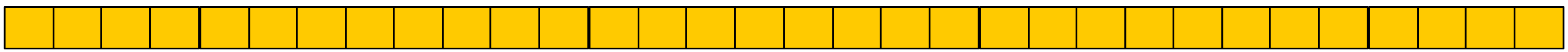






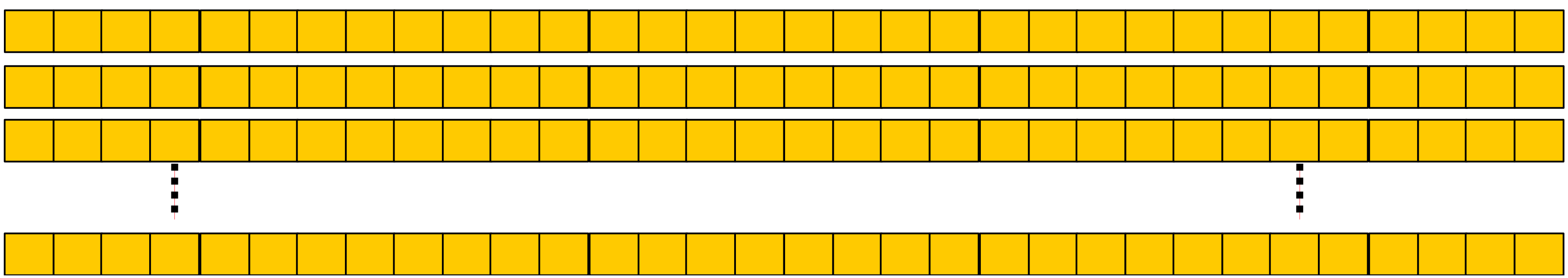
Warp vs. Thread vs. Instructions

Warp is 32 Threads



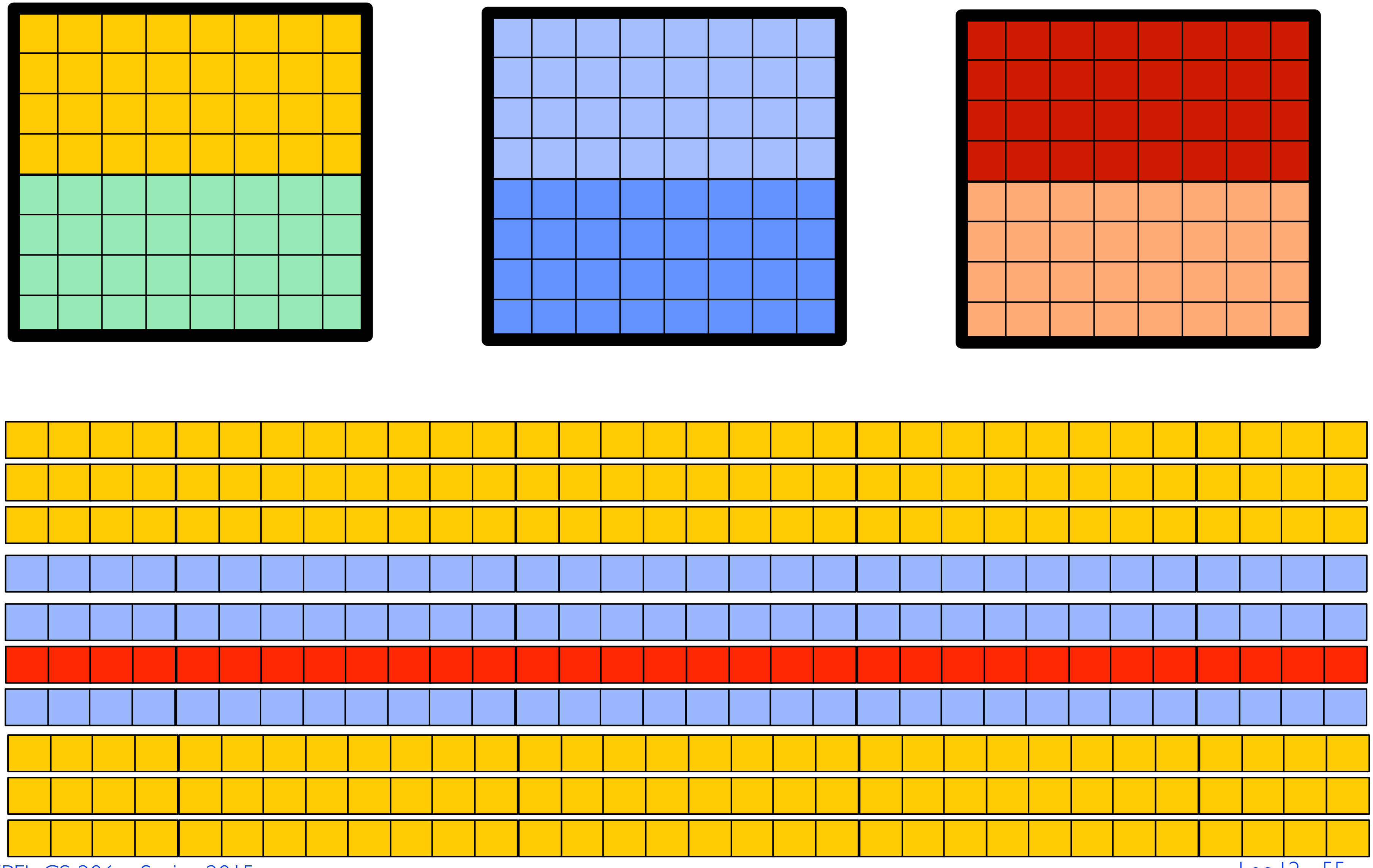
Each Thread has the same program

Each thread will execute many instructions

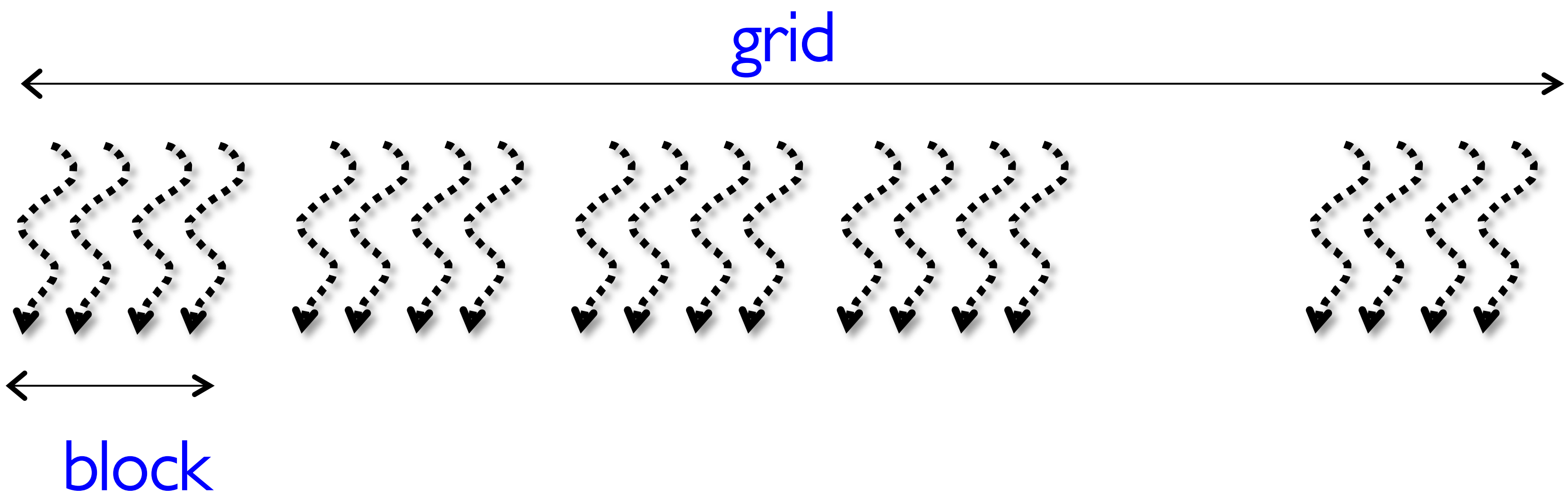


```
__global__ void fadepic (float *a, float f, int N)
 {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) a[i] = a[i] * f;
 }
```

Warp scheduling – Hiding Stalls



Exposing Locality to Programmer



Threads within a group can co-operate and coordinate

Communication & Synchronization

thread 10

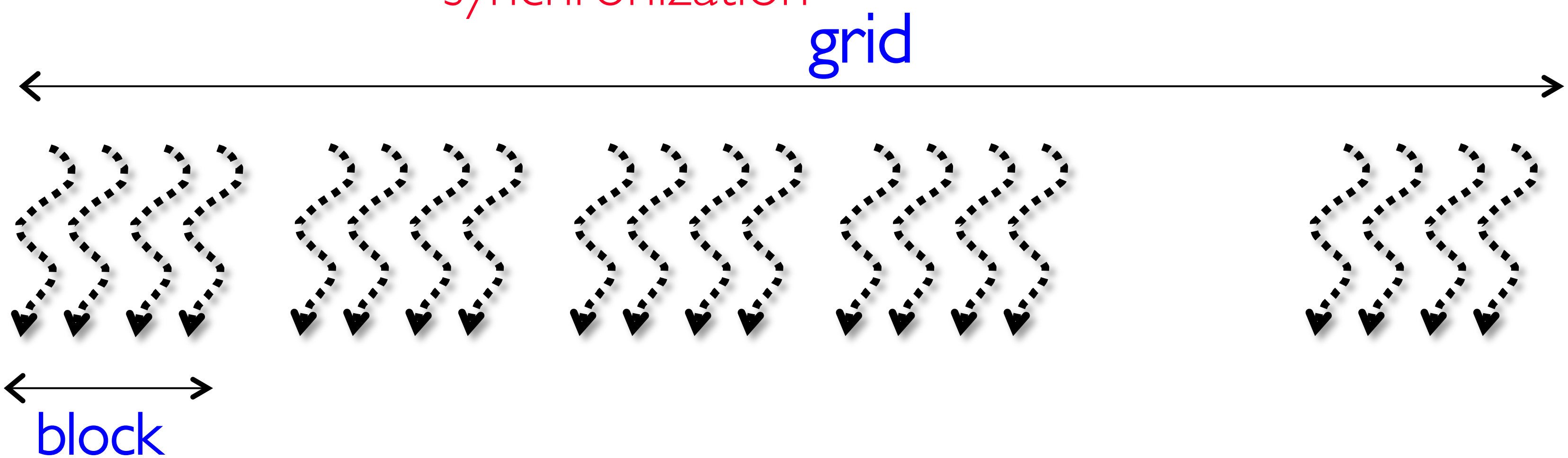
```
a[10] = in[10]
sync
a[10] += a[11]
```

thread 11

```
a[11] = in[11]
sync
```

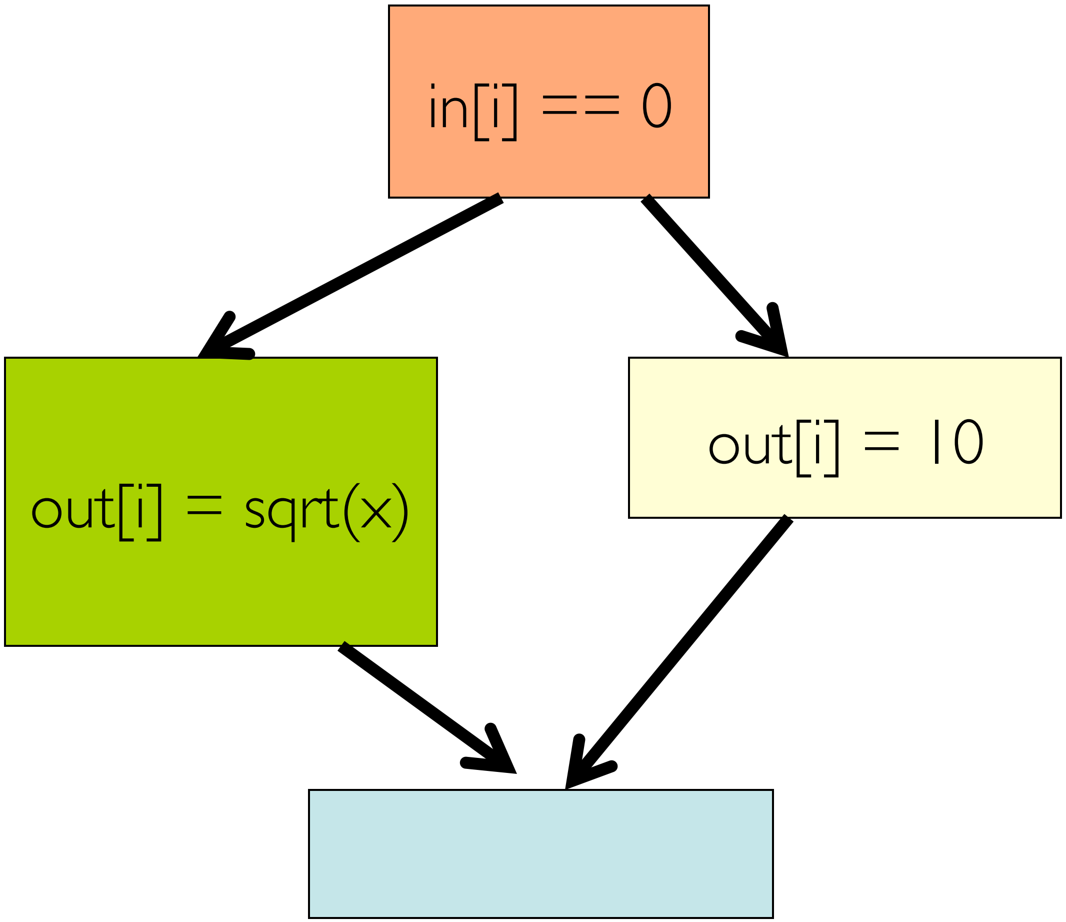
communication

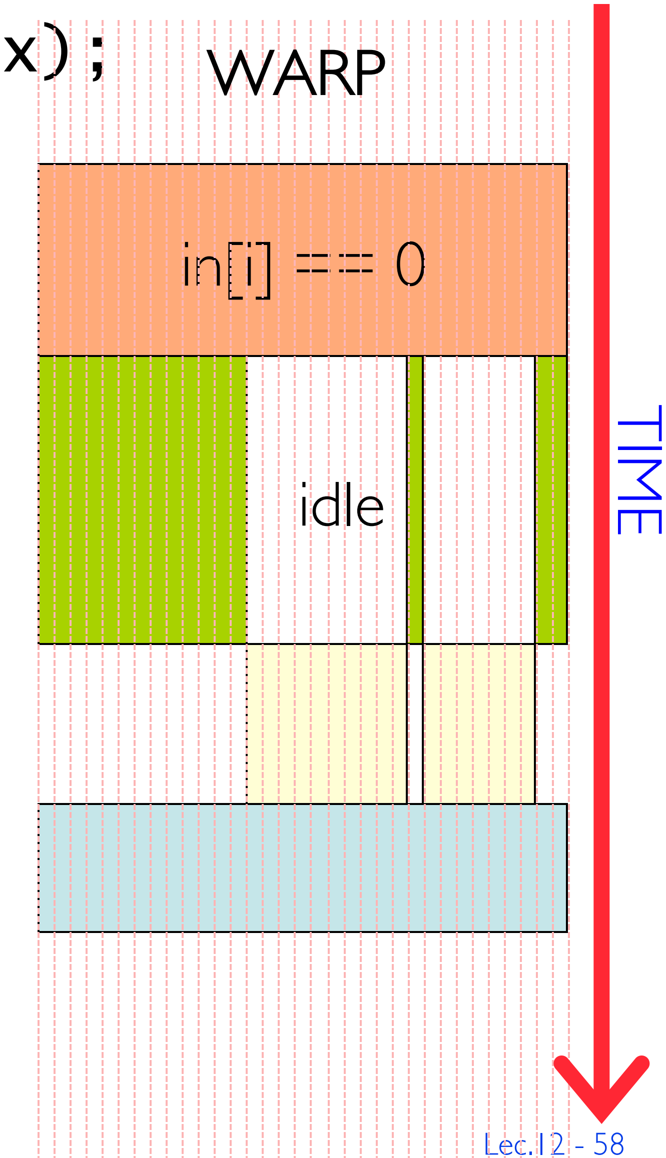
synchronization



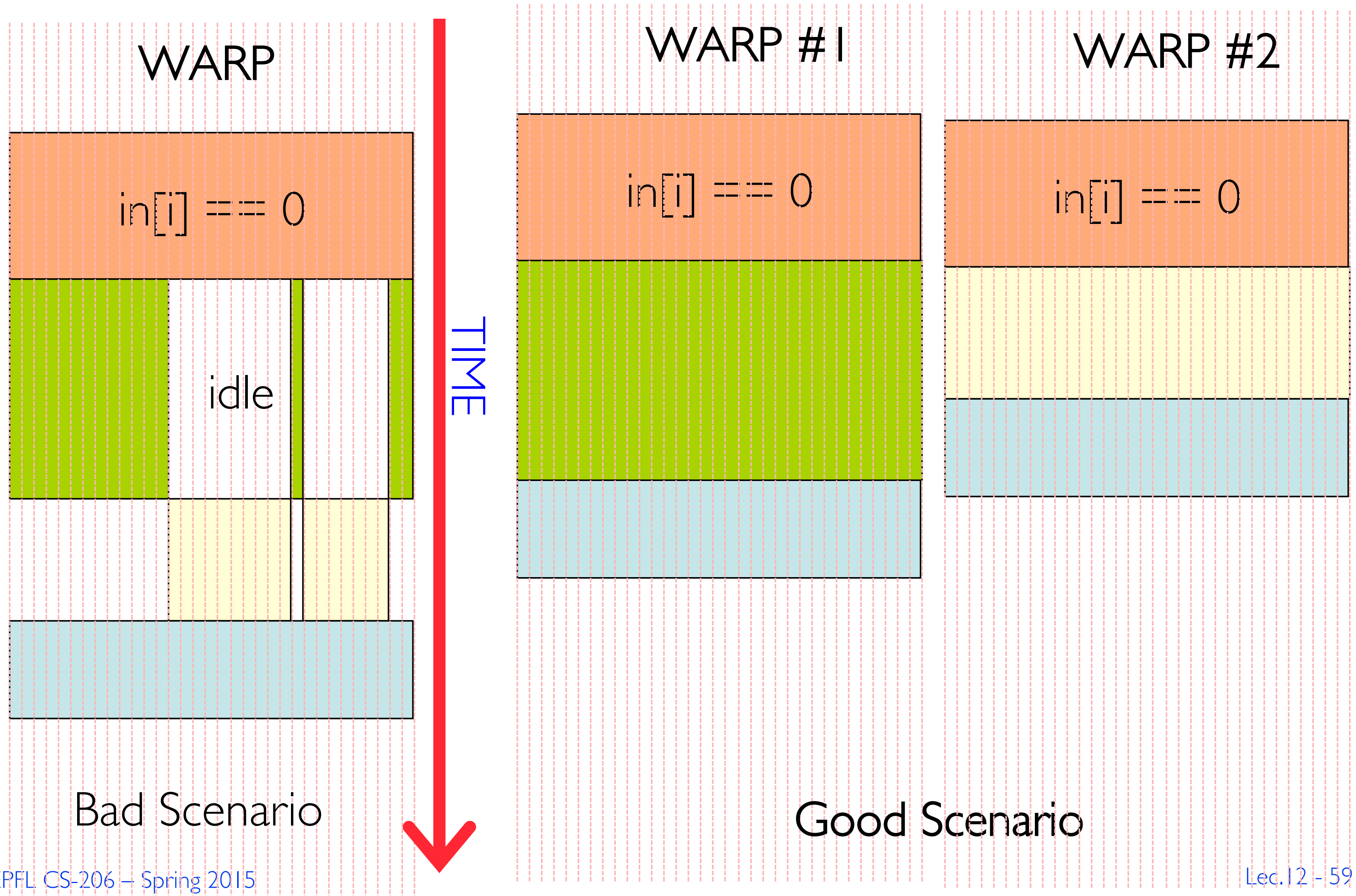
WARP Execution and Control Flow Divergence

```
if (in[i] == 0) out[i] = sqrt(x);
else out[i] = 10;
```





Control Flow Divergence Contd.

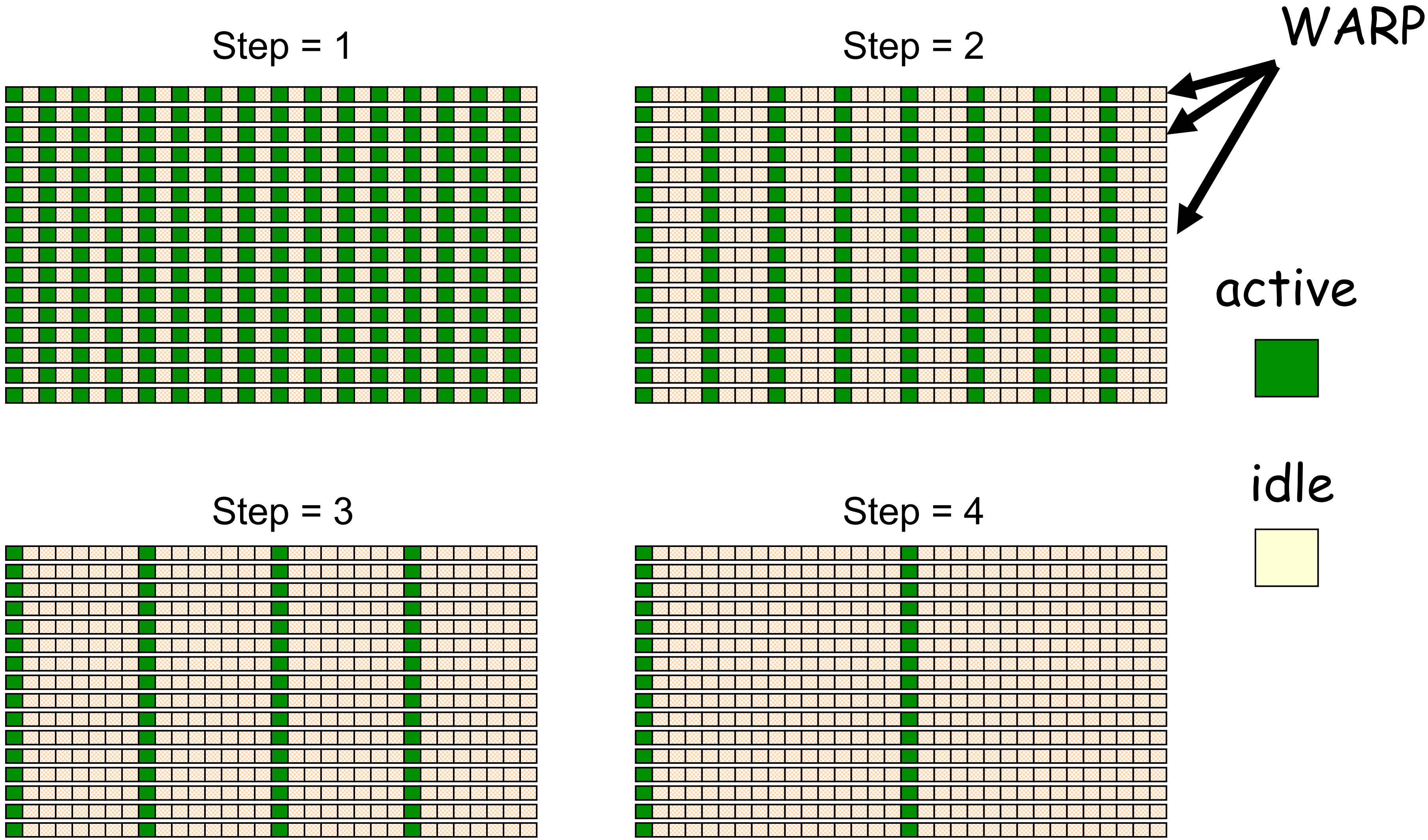


Back to Reduction Kernel # 1

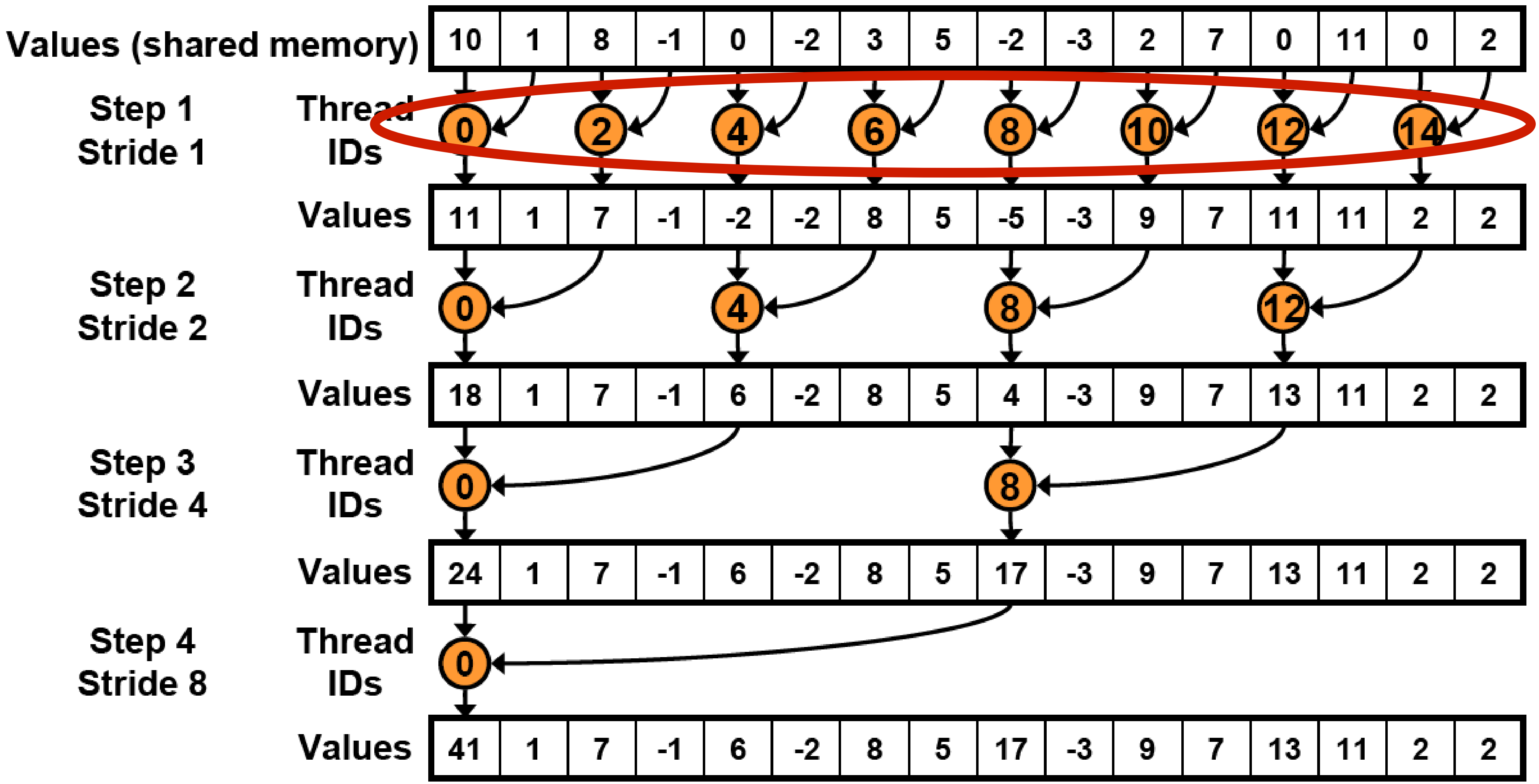
```
__global__ void reduce0(int *g_idata, int *g_odata, int n) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Highly divergent code leads to poor performance

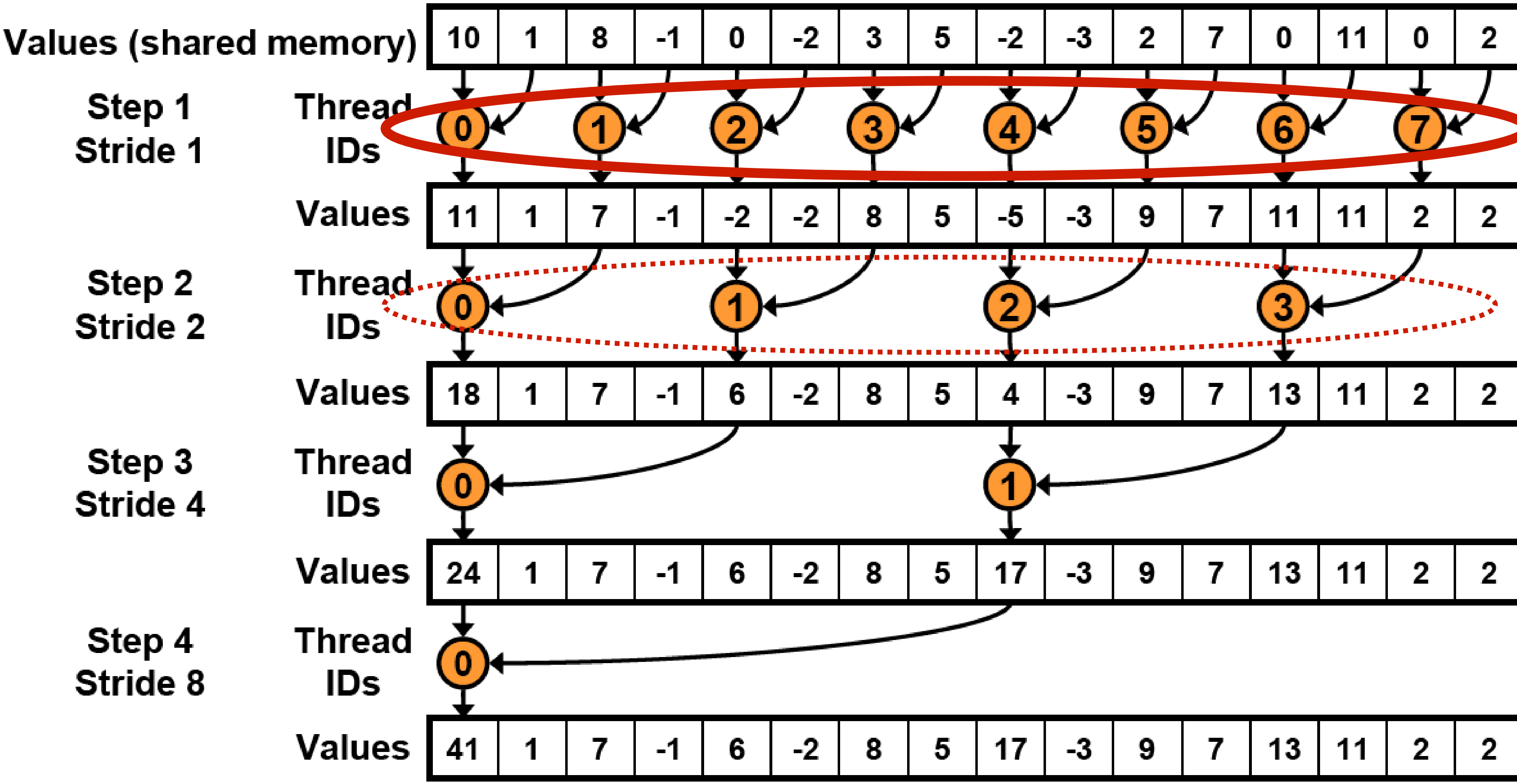
Divergent threads in warps!



Lots of idle threads/warp



Group all active threads together!



Reduction Kernel #2: Non-divergent threads

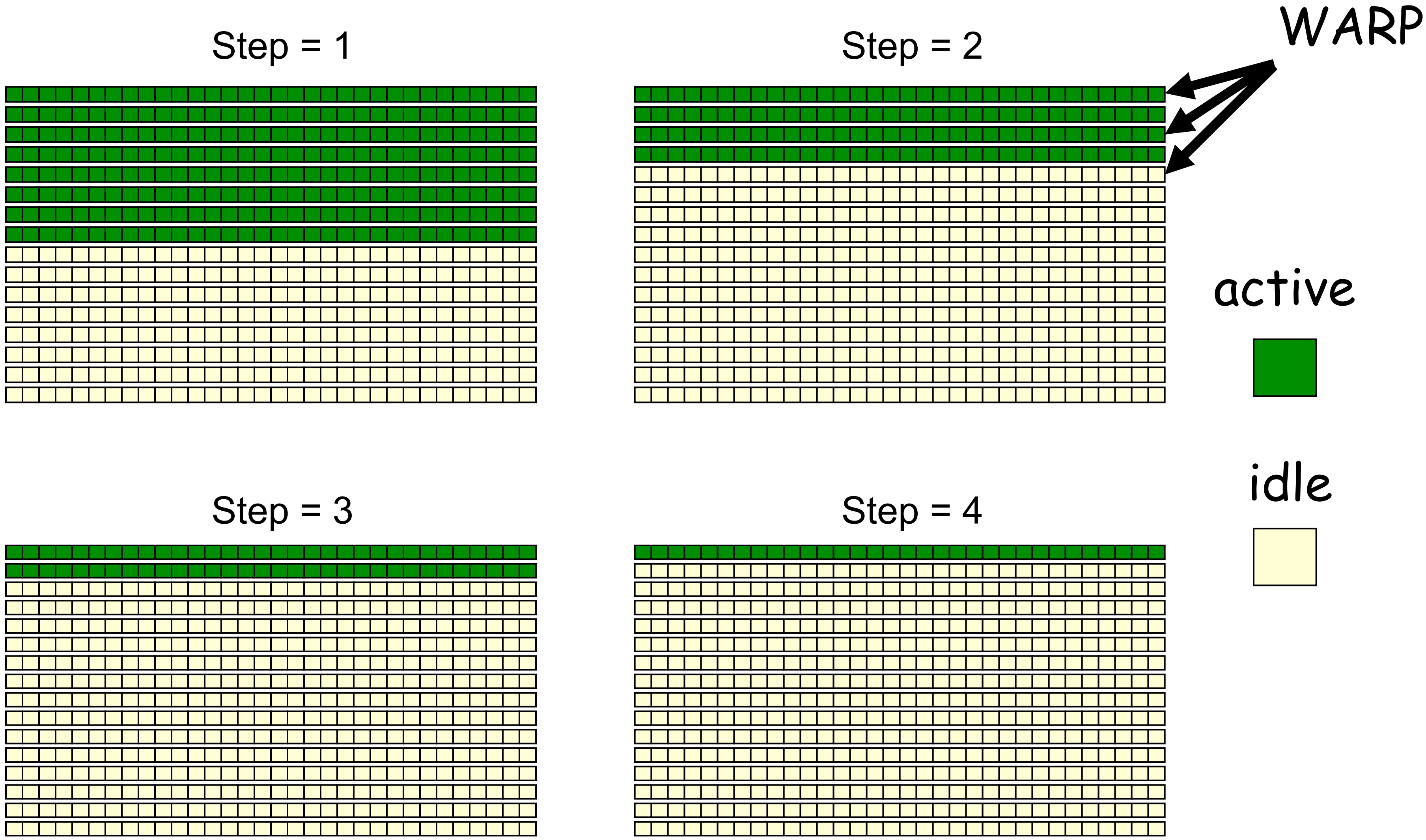
Replace the divergent branching code

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

With strided index and non-divergent branch

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x / s) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

Non-divergent threads



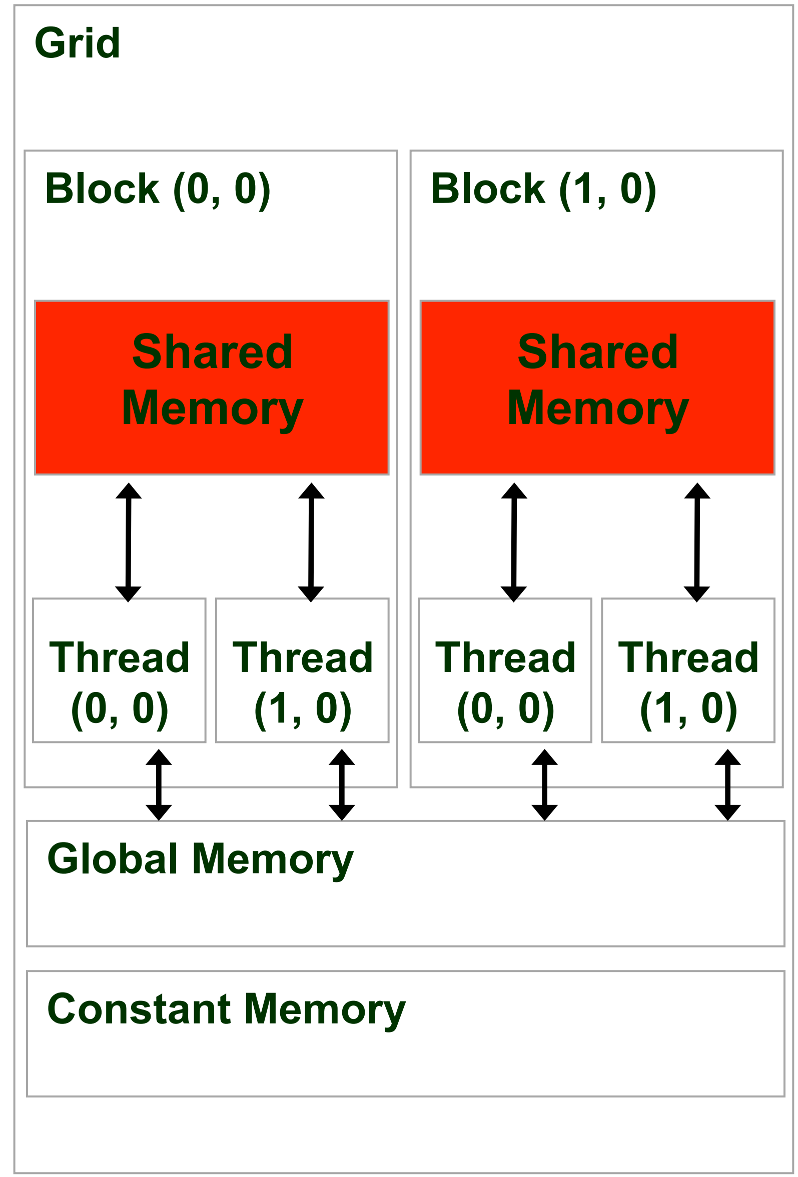
Performance for 4M element reduction

	Time (2^{22} ints)	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	**4.25ms**		
Kernel 2: interleaved addressing non-divergent branching	**3.32 ms**	**1.28x**	**1.28x**

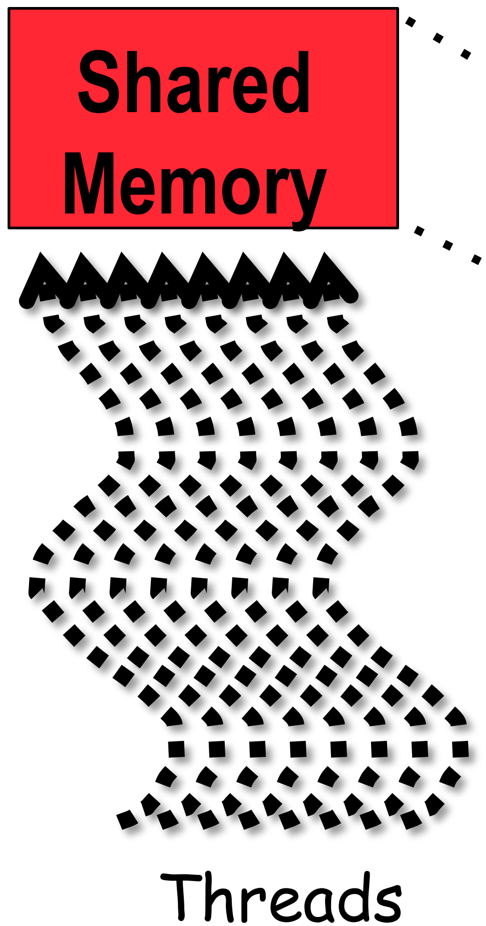
- Hmm…..not enough parallelism
- What gives?

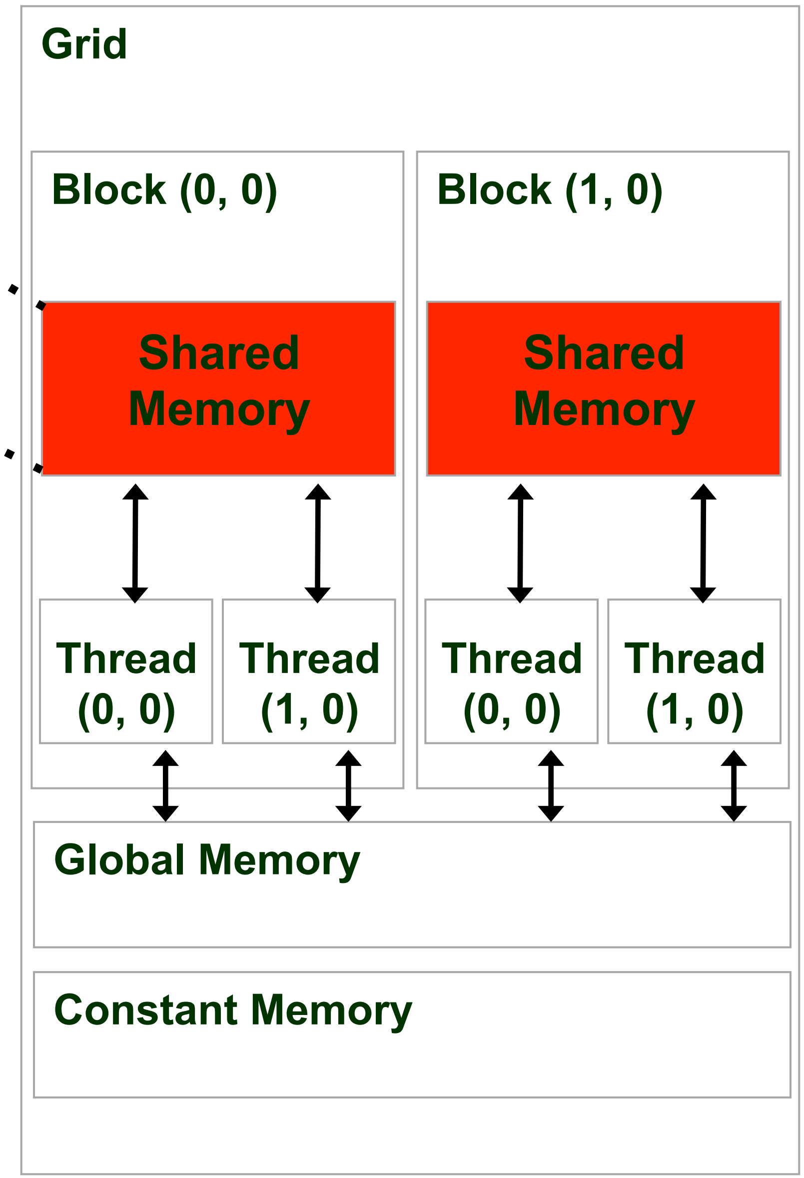
Recall: Using Shared Memory

- Load temporally into shared memory
- For inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns



Shared Memory is a bottleneck!

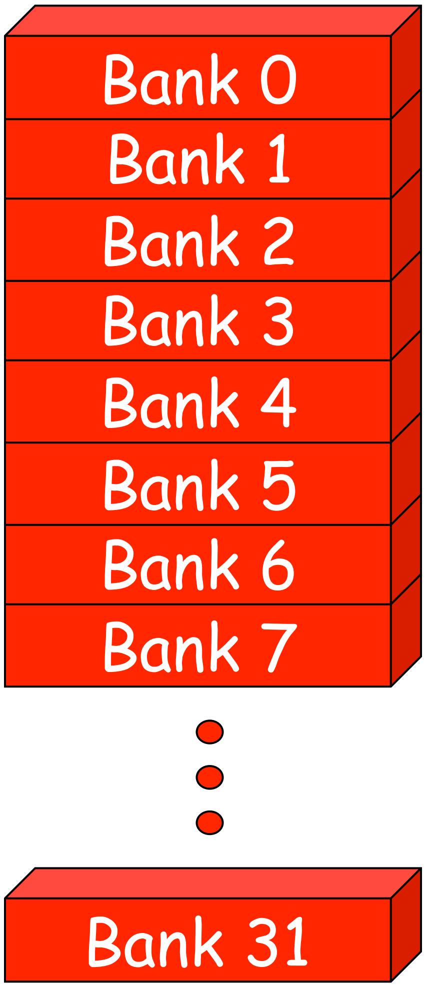




- How do we let tens of threads access memory?

Shared memory is banked!

- Parallel access to shared memory
  - Causes contention
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- A memory can service as many simultaneous accesses as it has banks
  - Typically, one access per two cycles
- Multiple simultaneous accesses to a bank result in a conflict
  - Conflicting accesses are serialized



Shared Memory Bank Conflicts

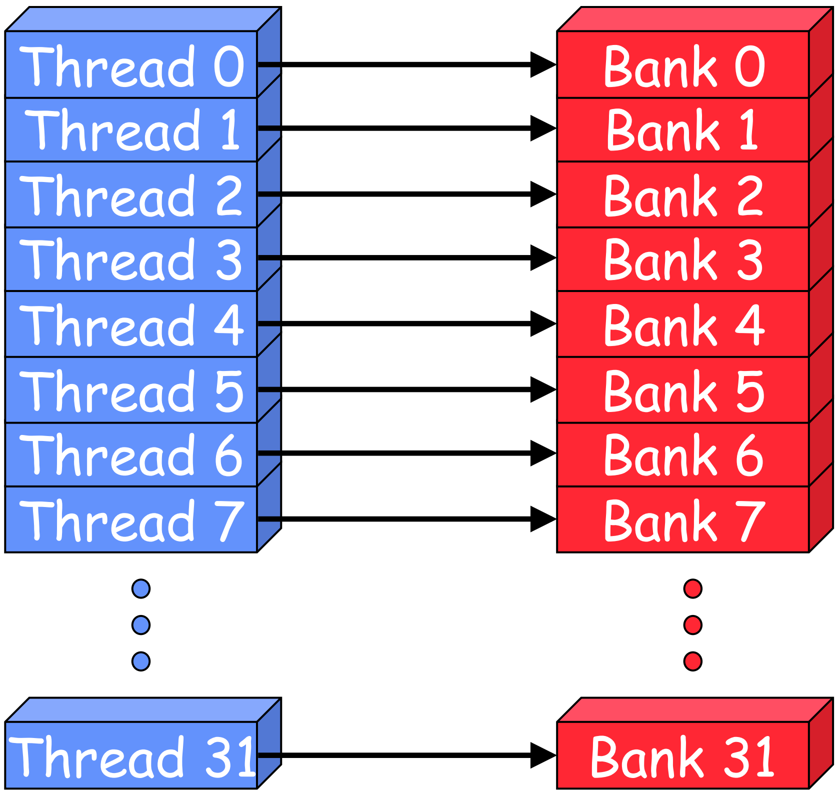
- Organization (machine dependent):
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
  - 4- or 8-byte interleaving → 2x for double floats

- Performance:
  - E.g., 4 bytes per bank per 2 clocks per core
  - Memory accesses are issued per 32 threads (warp)
  - Serialization: threads accessing different words in the same bank
    - Accesses are serialized
  - Multicasting: threads accessing the same word in the same bank
    - Accesses are parallel

Bank Addressing Examples

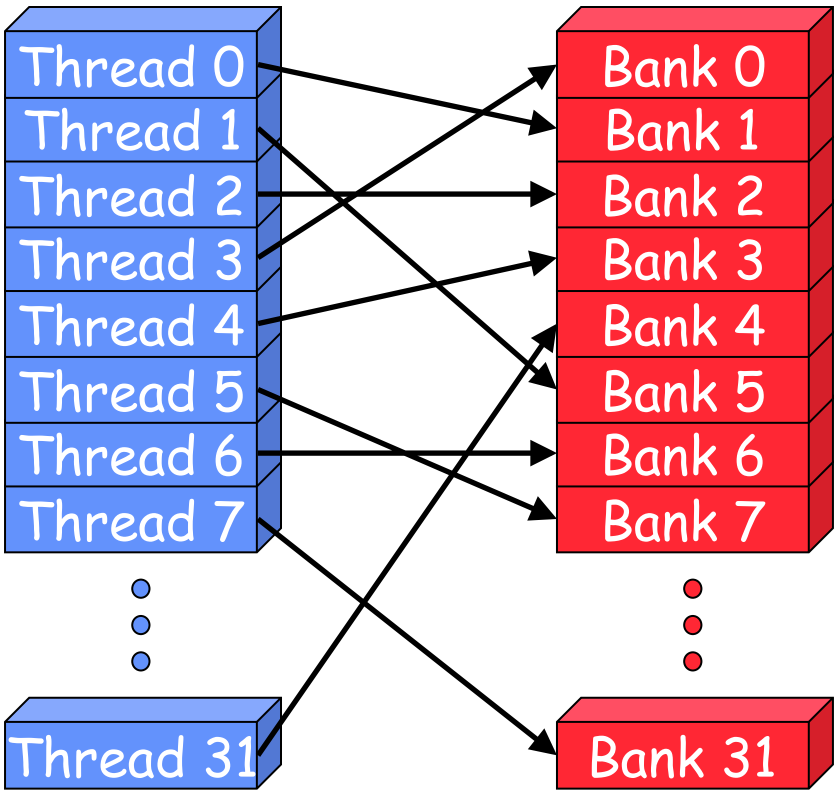
No Bank Conflicts

Linear addressing stride = 1

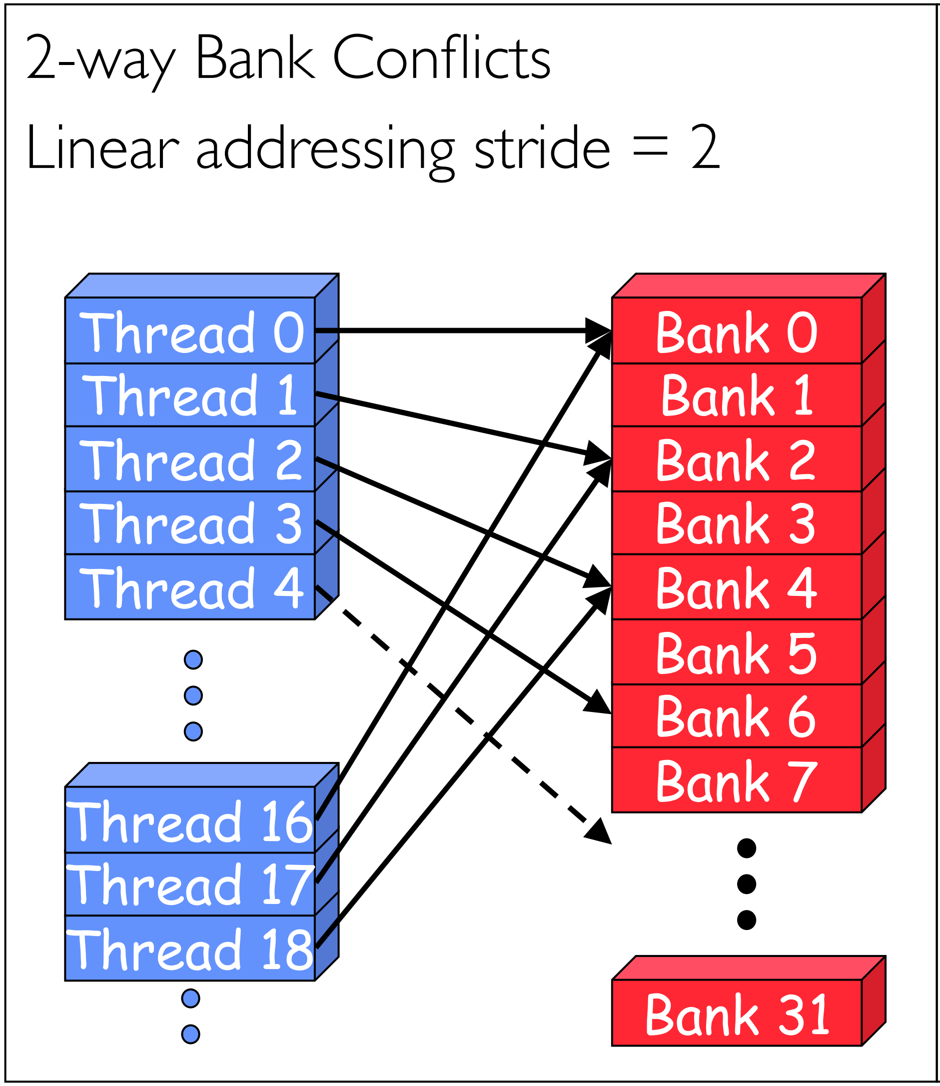


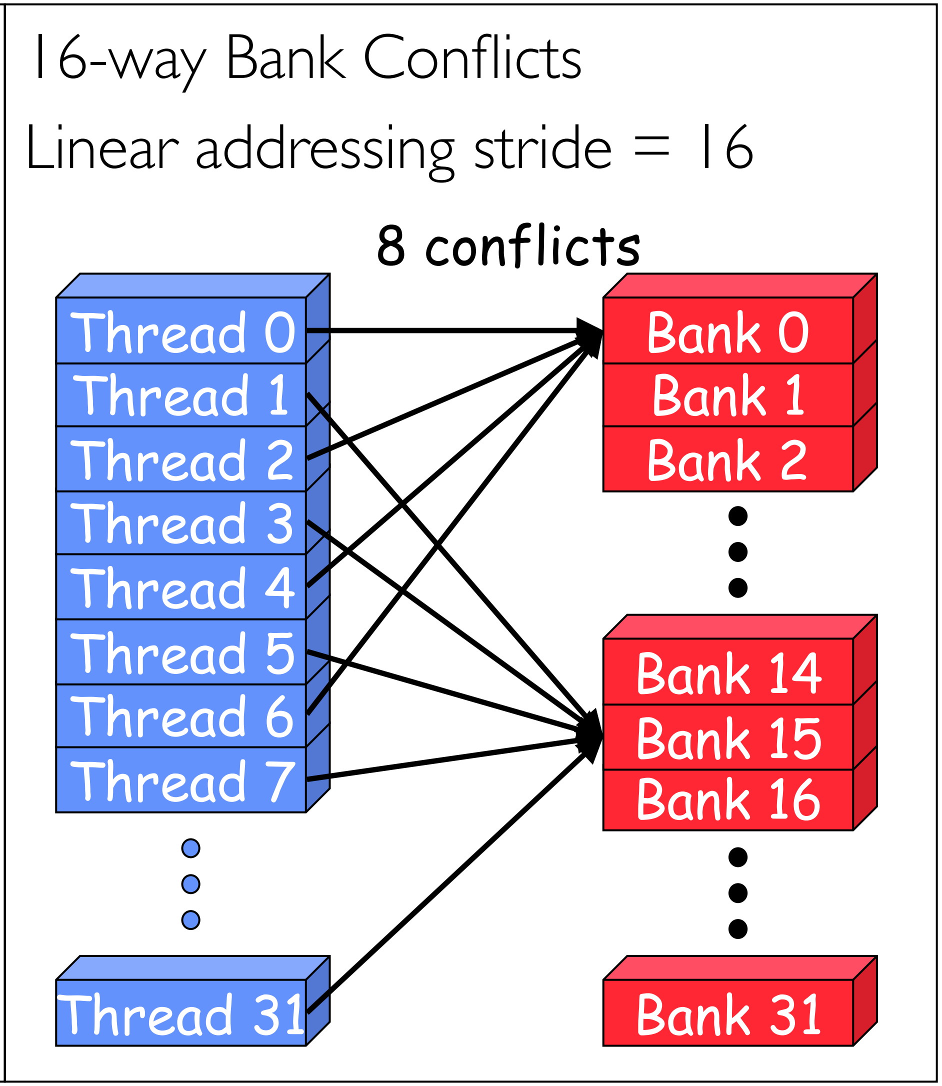
No Bank Conflicts

Random 1:1 Permutation



Bank Addressing Examples





Shared Memory Performance Summary

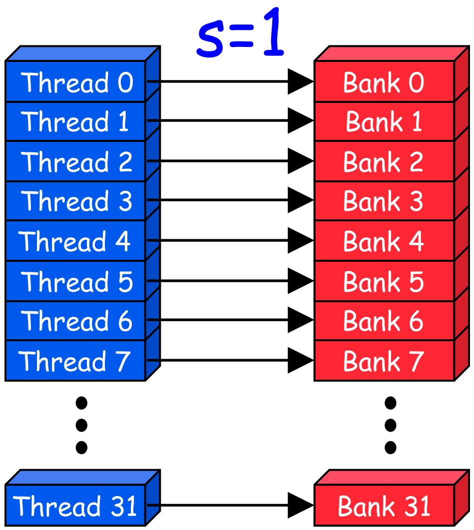
- The fast case:
  - All threads access different banks, no bank conflict
  - No two different words are accessed in the same bank

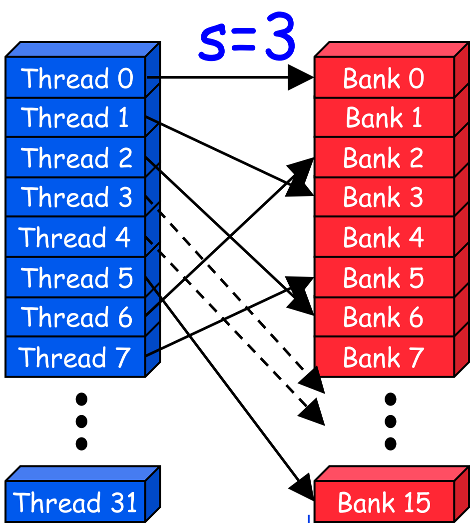
- The slow case:
  - Bank conflict: multiple threads access different words in the same bank
  - Must serialize accesses
  - Cost = max # of simultaneous accesses to a single bank

Linear Addressing

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

- This is only conflict-free if s shares no common factors with the number of banks
- With 32 banks, s must be odd





Example with 32 banks

```
shared[baseIndex + s * threadIdx.x];
```

Calculate the degree of conflict for s=1, s=2, s=3, s=4

Example with 32 banks

```
shared[baseIndex + s * threadIdx.x];
```

Calculate the degree of conflict for s=1, s=2, s=3, s=4

s=1

Accesses to bank 0: 0

s=2

Accesses to bank 0: 0, 16

s=3

Accesses to bank 0: 0

s=4

Accesses to bank 0: 0, 8, 16, 24

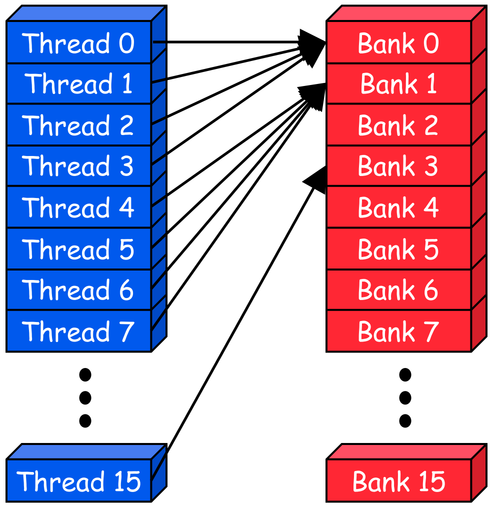
Data types & bank conflicts

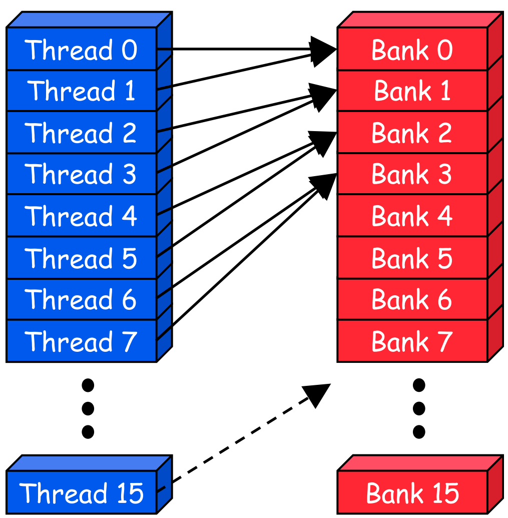
- This has no conflicts if type of shared is 32-bits

```
foo = shared[baseIndex + threadIdx.x]
```

- Multicast for all 32-bit & smaller data types

```
__shared__ char shared[];

__shared__ short shared[];

__shared__ int shared[];

__shared__ float shared[];
```

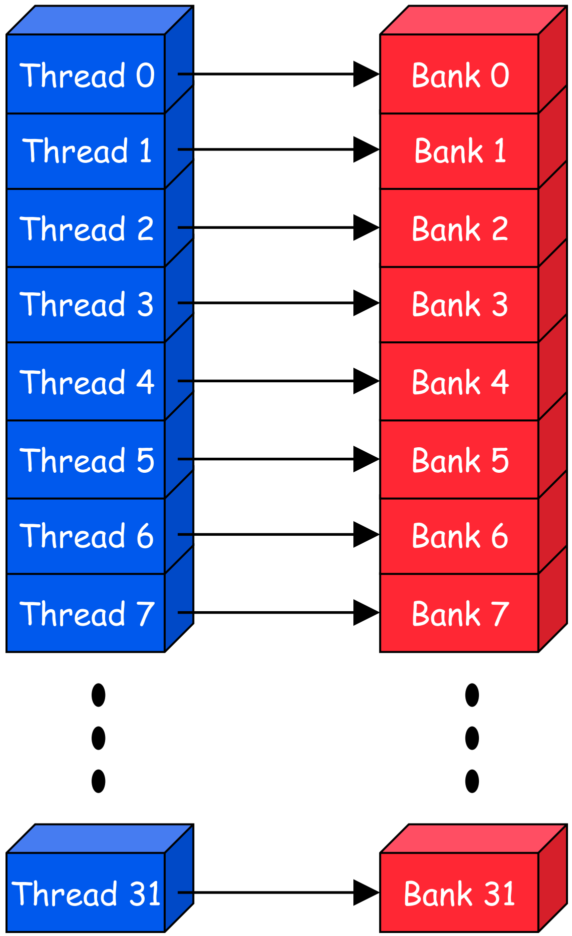




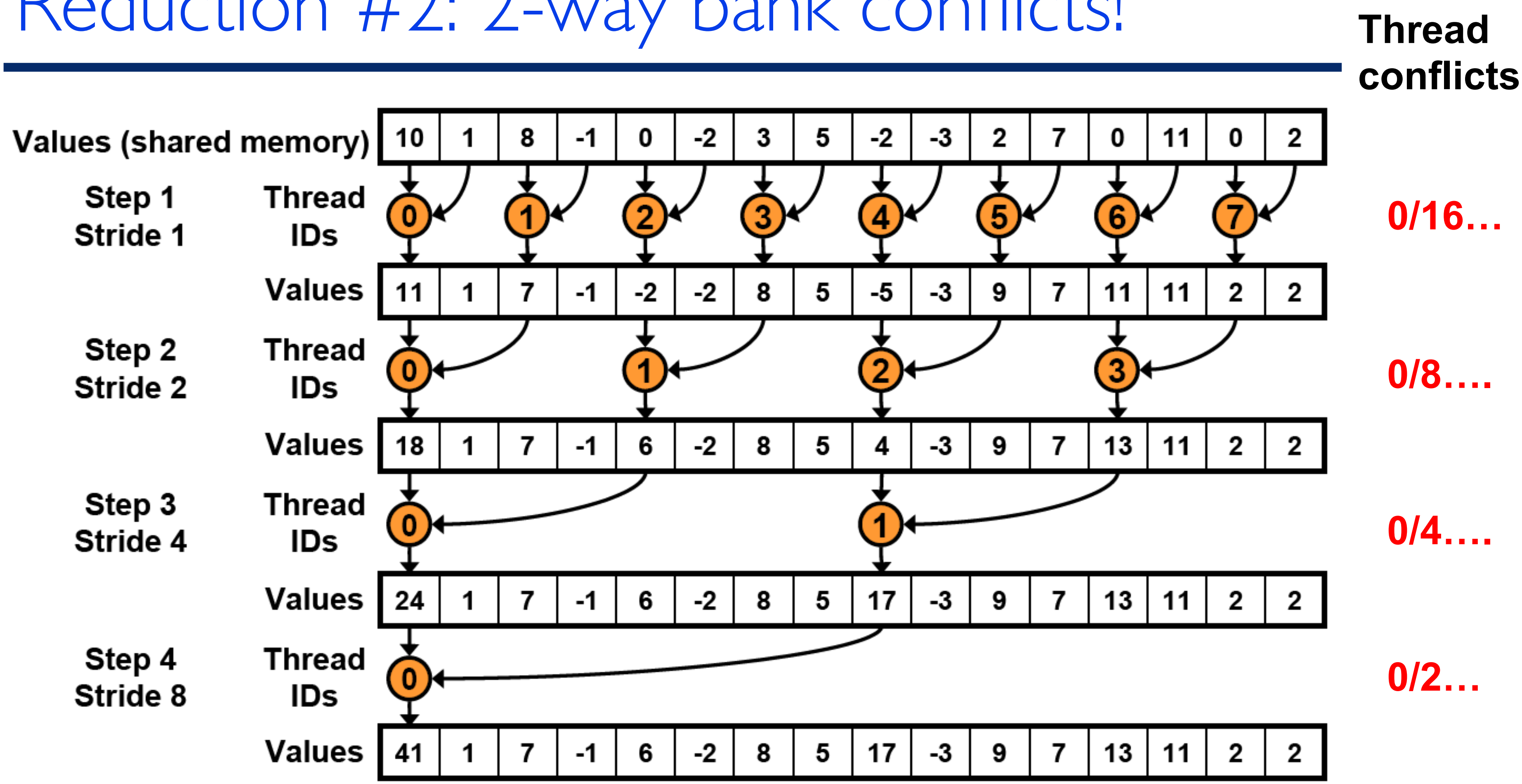
Example: Good Array Access Pattern

- Each thread loads one element in every consecutive group of blockDim elements

```
shared[tid] =
       global[tid];
shared[tid + blockDim.x] =
       global[tid + blockDim.x];
```



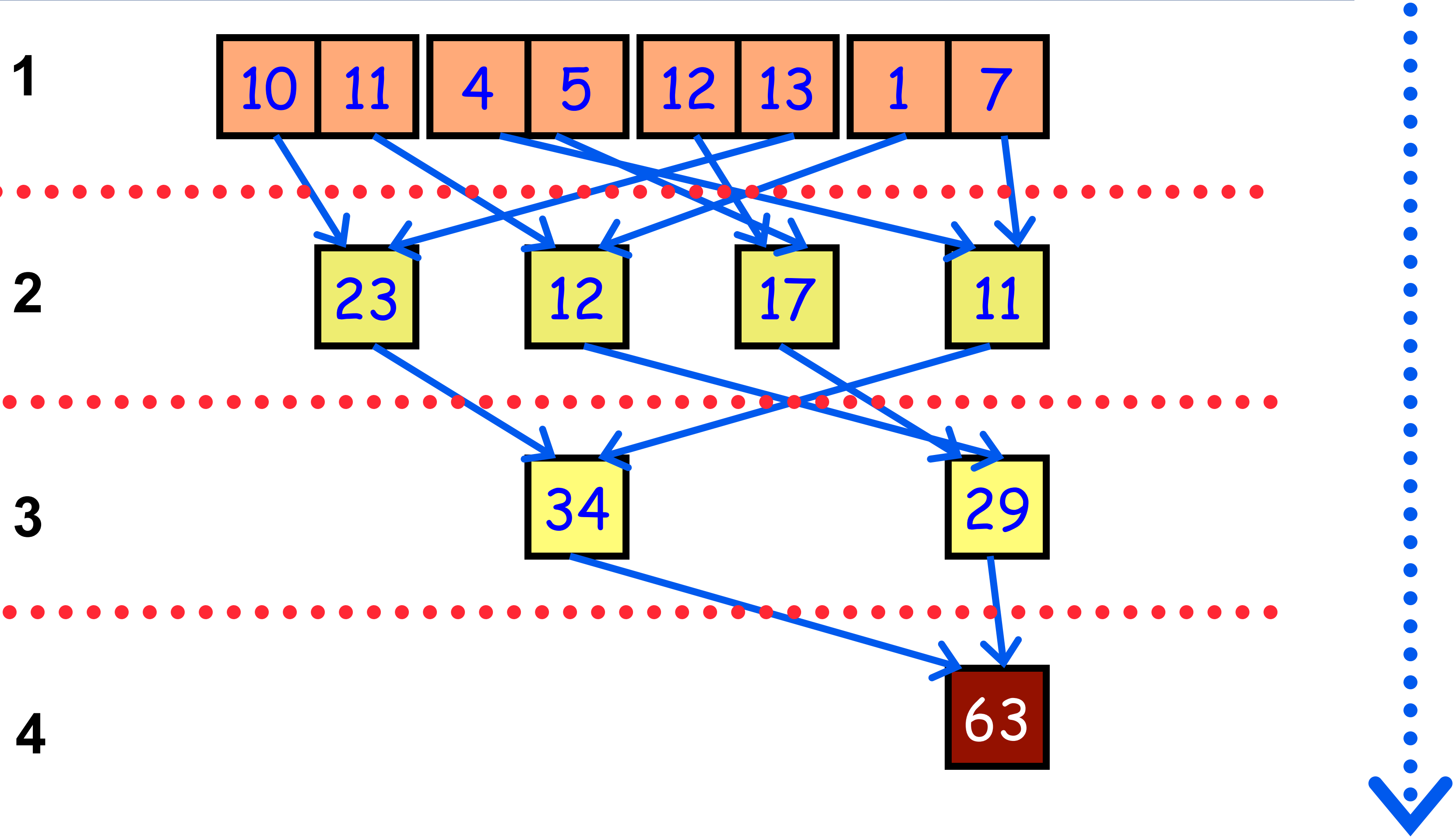
Reduction #2: 2-way bank conflicts!



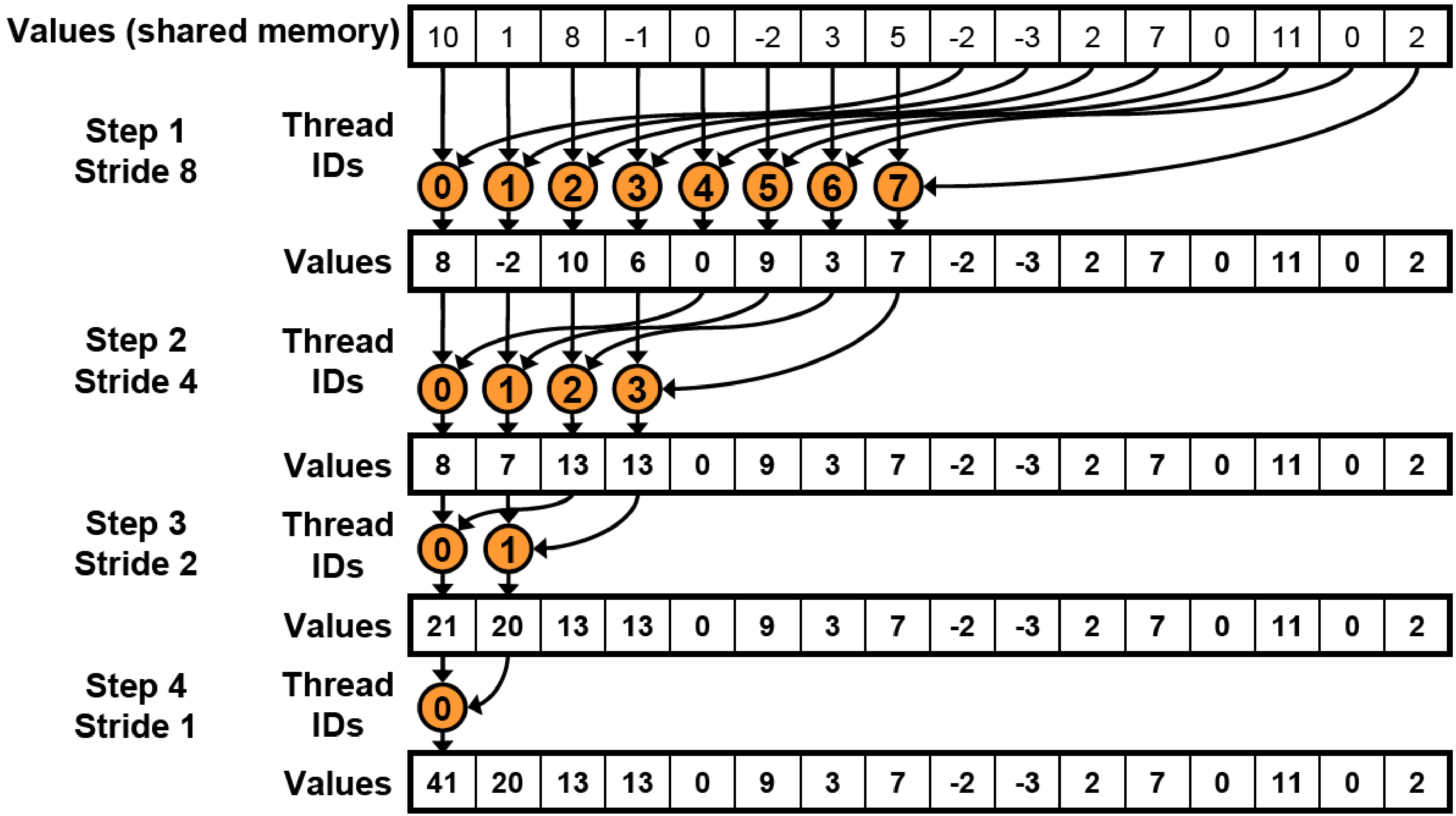
Assuming 32 banks and 32 threads:

- 2-way bank conflicts at every step

Observe: Arbitrary Unique Pairs OK



Reduction #3: Thread-sequential Accesses



Reduction #3: Code Changes

- Replace stride indexing in the inner loop

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index  = 2 * s * tid;

        if (index < blockDim.x == 0) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

- With reversed loop and threadID-based indexing

```
// do reduction in shared mem
for (unsigned int s = blockDim.x/2; s > 0; s /= 2) {

        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

Performance for 4M element reduction

	Time (2^{22} ints)	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	**4.25ms**		
Kernel 2: interleaved addressing non-divergent branching	**3.32 ms**	**1.28x**	**1.28x**
Kernel 3: sequential addressing	**2.06 ms**	**1.61x**	**2.06x**

Reduction #3: Bad resource utilization

- All threads read one element
- First step: half of the threads are idle
- Next step: another half becomes idle

Reduction #4:
Read two elements and do the first step

- Original: Each thread reads one element

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

- Read and reduce the first two elements

```
// each thread loads two elements from global to shared mem
// end performs the first step of the reduction
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x* blockDim.x * 2 + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];
__syncthreads();
```

Performance for 4M element reduction

	Time (2^{22} ints)	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	**4.25ms**		
Kernel 2: interleaved addressing non-divergent branching	**3.32 ms**	**1.28x**	**1.28x**
Kernel 3: sequential addressing	**2.06 ms**	**1.61x**	**2.06x**
Kernel 4: first step during global load	**1.05 ms**	**1.96x**	**4.04x**

Reduction #4: Still way off

- Memory bandwidth is still underutilized
  - We know that reductions have low arithmetic density

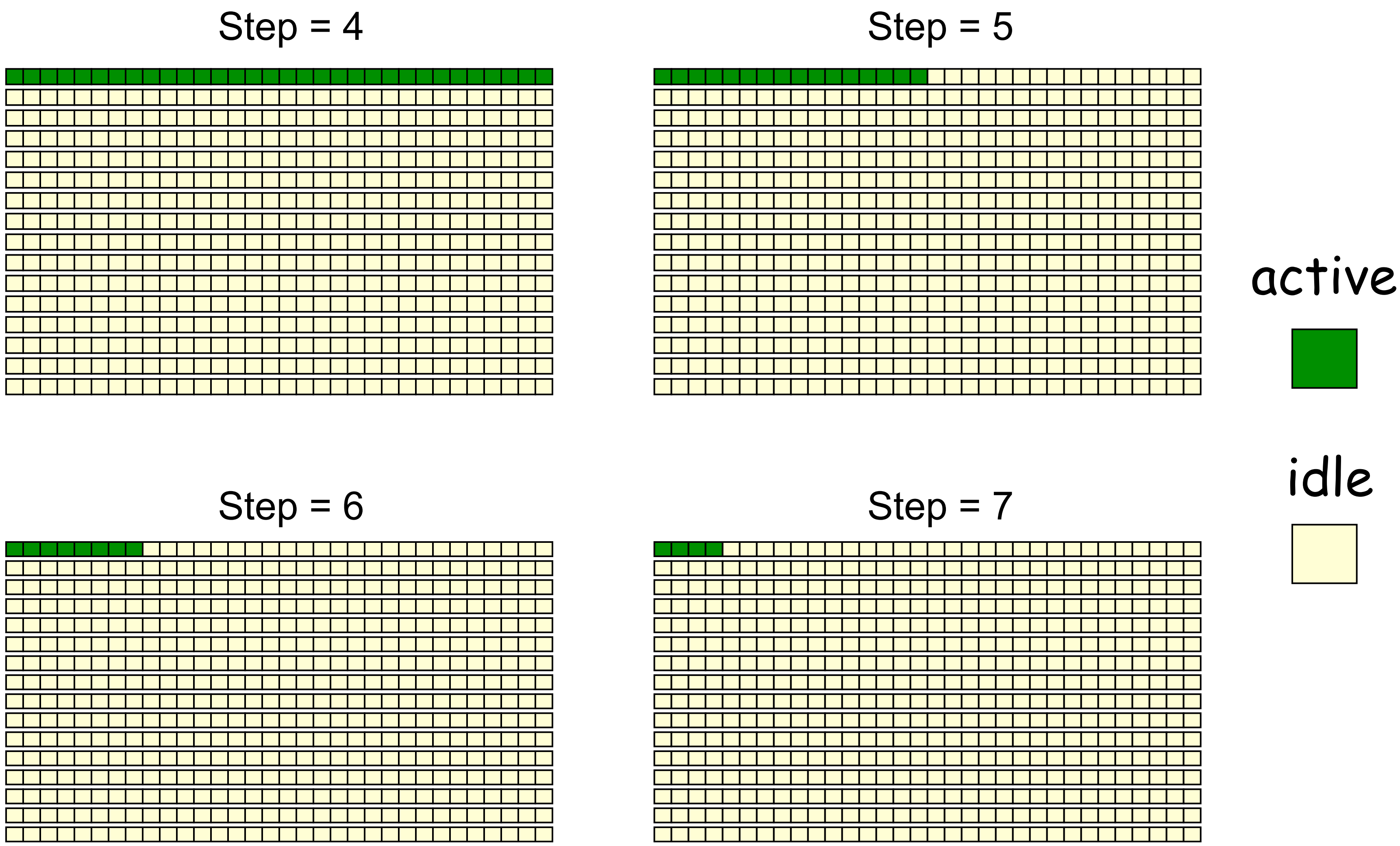
- What is the potential bottleneck?
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - Address arithmetic and loop overhead
  - Synchronization overhead

- Unroll loops to eliminate these “extra” instructions

Unrolling the last warp

- At every step the number of active threads halves
  - When s <=32 there is only one warp left
- Instructions are SIMD-synchronous within a warp
  - They all happen in lock step
  - No need to use __syncthreads()
  - We don't need "if (tid < s)" since it does not save any work
    - All threads in a warp will "see" all instructions whether they execute them or not
- Unroll the last 6 iterations of the inner loop
  - s <= 32

Last warps



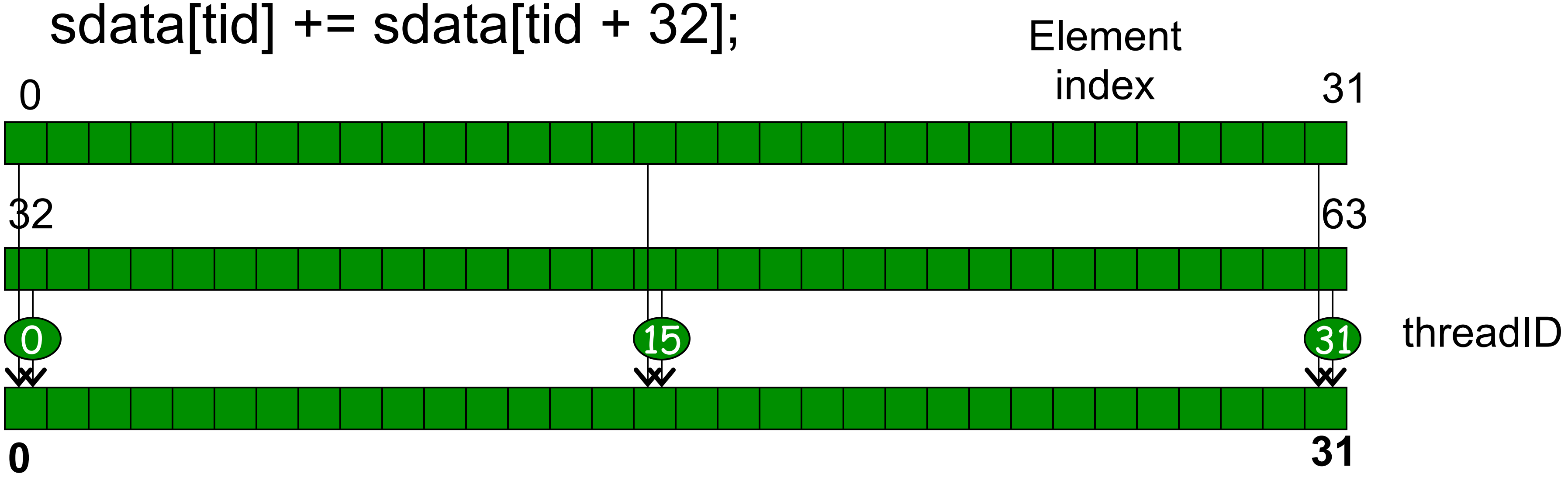
Reduction #5: Unrolling the last 6 iterations

```
// do reduction in shared mem
 for (unsigned int s = blockDim.x/2; s > 32; s /= 2) {

        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
 }
```

```
 if (tid <32)
{
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid + 8];
        sdata[tid] += sdata[tid + 4];
        sdata[tid] += sdata[tid + 2];
        sdata[tid] += sdata[tid + 1];
 }
```

Unrolling the last warp: A Closer Look

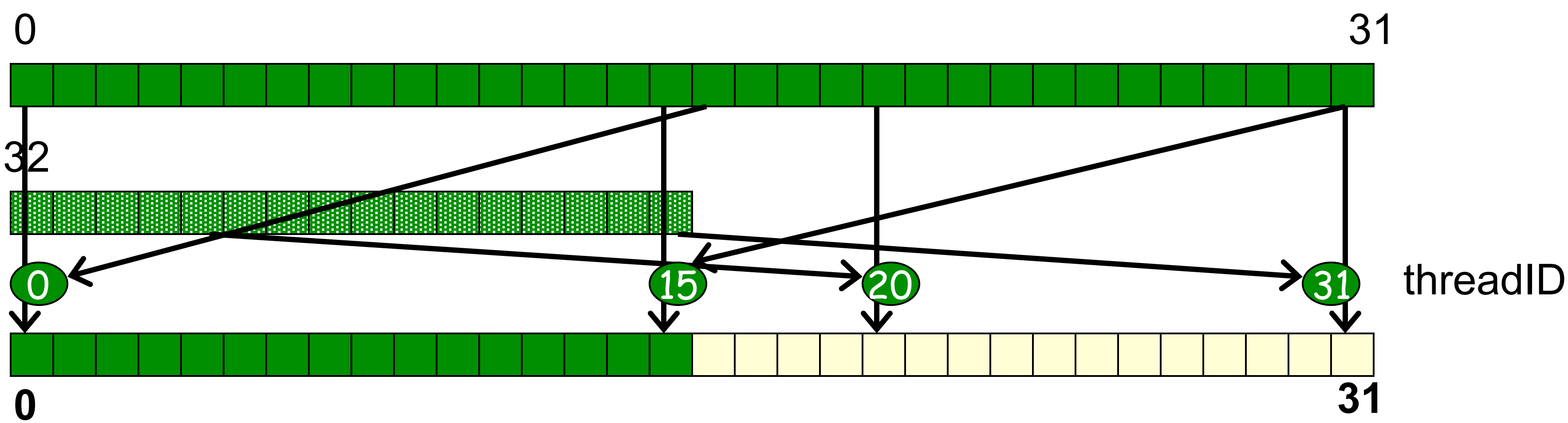
sdata[tid] += sdata[tid + 32];



All threads doing useful work

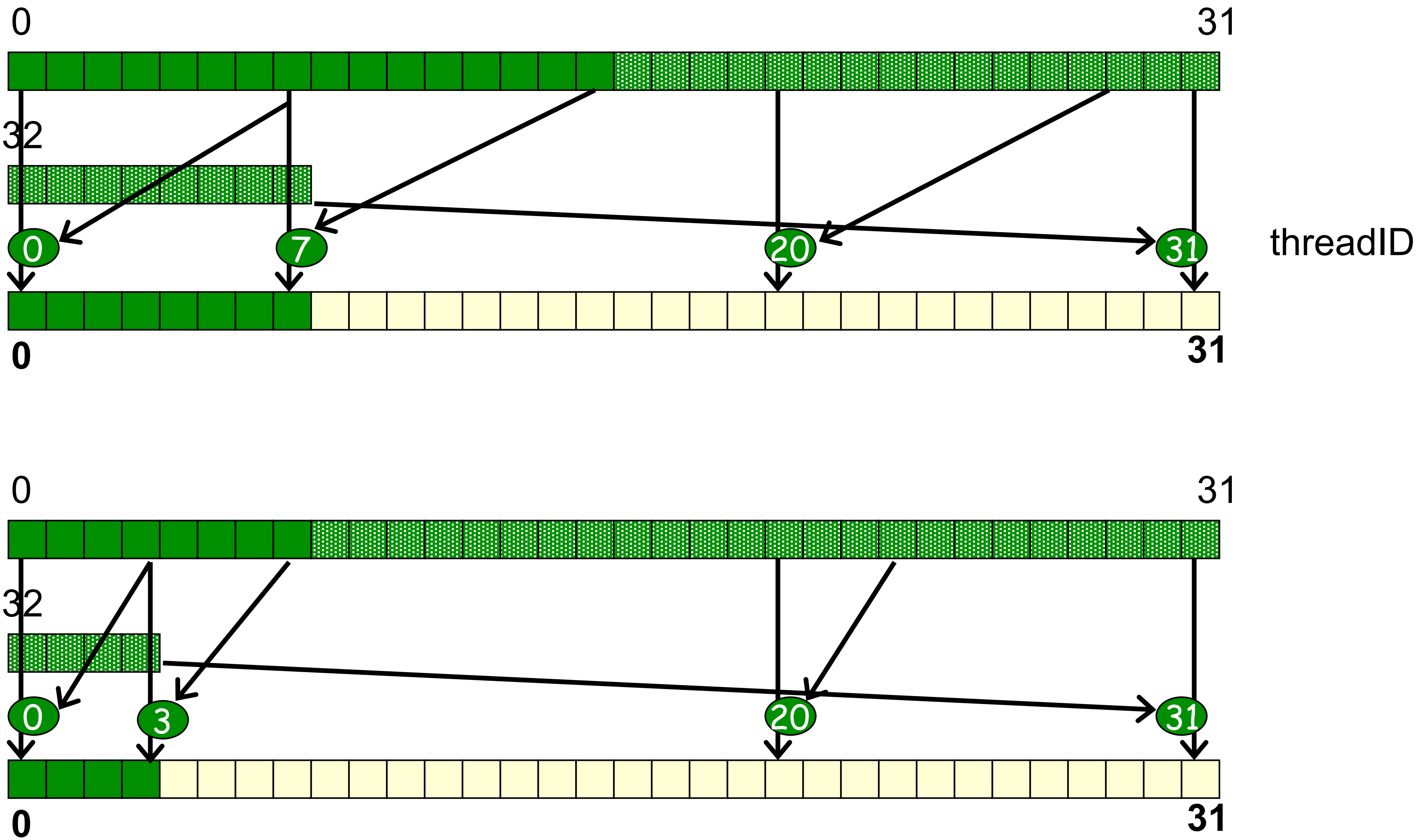
Unrolling the Last WARP: A Closer Look

sdata[tid] += sdata[tid + 16];

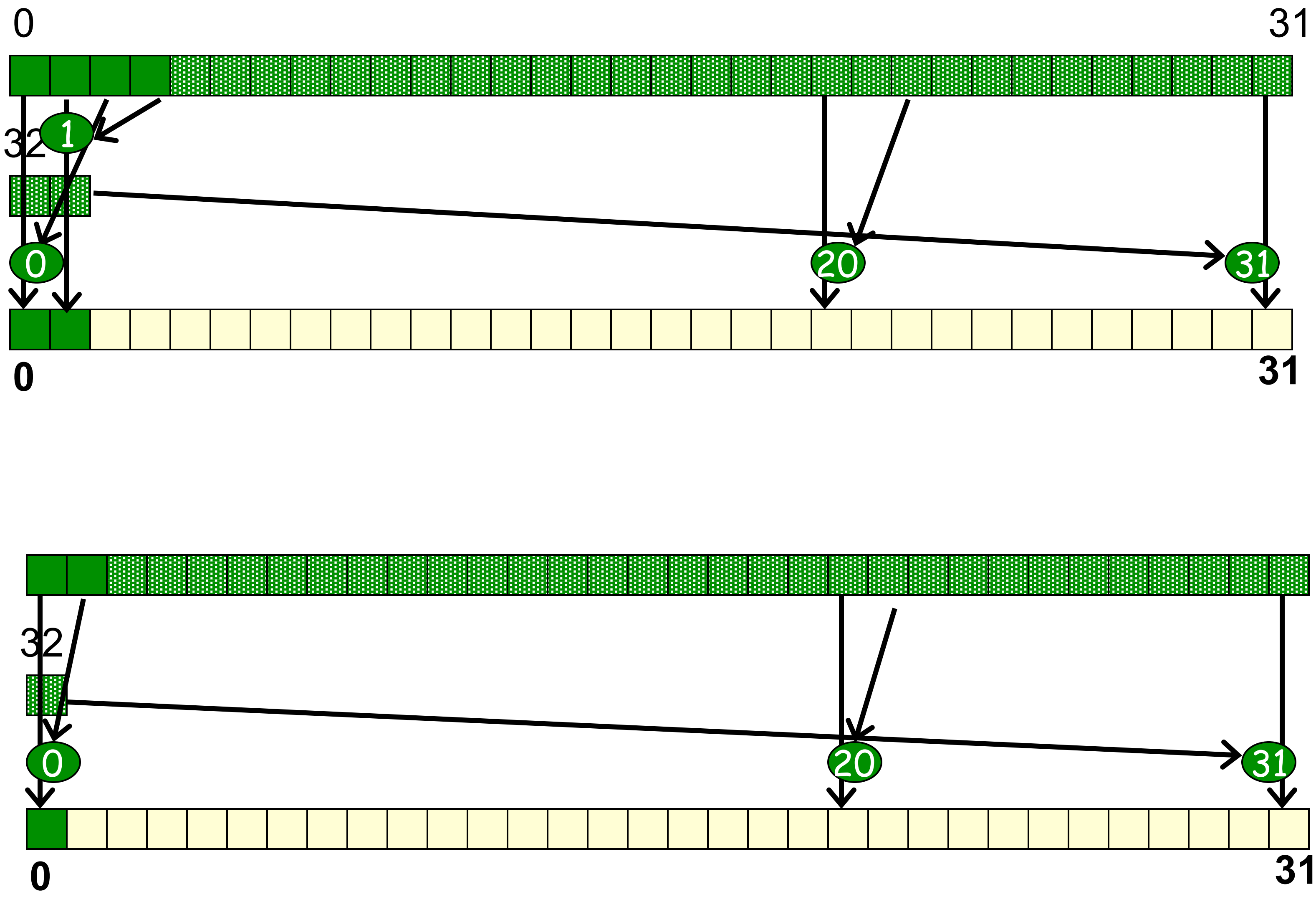


- Half of the threads do useless work (thrown away)
- Elements 16-31 are inputs to threads 0-14
- But threads 0-15 read them before they get written by threads 16-31
  - All reads proceed in “parallel” first
  - All writes proceed in “parallel” last
- But, threads 16-31 are doing useless work
  - The units and bandwidth are there → no harm (only power)

Unrolling the last warp: A Closer Look



Unrolling the last warp: A Closer Look



Performance for 4M element reduction

	Time (2^{22} ints)	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	**4.25ms**		
Kernel 2: interleaved addressing non-divergent branching	**3.32 ms**	**1.28x**	**1.28x**
Kernel 3: sequential addressing	**2.06 ms**	**1.61x**	**2.06x**
Kernel 4: first step during global load	**1.05 ms**	**1.96x**	**4.04x**
Kernel 5: Unroll last warp	**0.73ms**	**1.43x**	**5.82x**

Summary

- Performance (efficiency) is everything
- Need to assign work, schedule memory carefully

- Techniques:
  - Tiling and shared memory
  - WARPs
  - Avoiding bank conflicts
  - Loop unrolling