

GPU COMPUTING

LECTURE 07 - SCHEDULING OPTIMIZATIONS

Kazem Shekofteh

Kazem.shekofteh@ziti.uni-heidelberg.de

Institute of Computer Engineering

Ruprecht-Karls University of Heidelberg

Inspired from lectures by Holger Fröning

Based on “Optimizing Parallel Reduction in CUDA” by Mark Harris

REMINDER: SCHEDULING

Thread hierarchy: threads are organized in blocks, each thread block (Cooperative Thread Array, CTA¹) can be mapped to one SM

Parallel slackness leveraged to hide latencies

In essence, memory access latencies

Start many more threads/CTAs than resources available

Thread block

No dependencies/guarantees for different CTAs

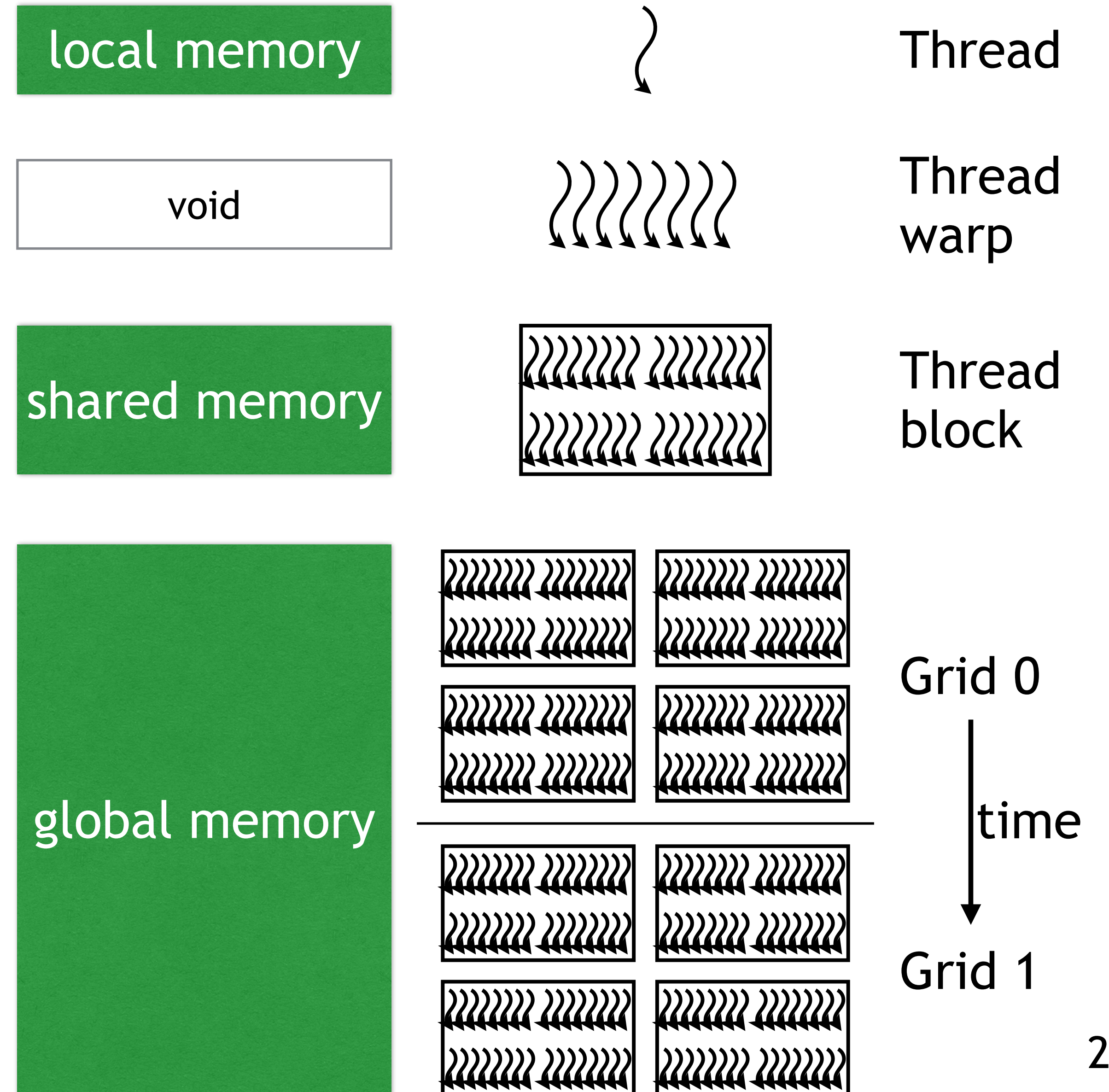
Up to 4 CTAs can be scheduled to one SM (for Kepler, implementation dependent)

Threads within a block

Warp of 32 threads as scheduling unit (for Kepler, implementation dependent)

Implementation-dependent optimizations => compatibility and code maintenance issues

¹ PTX term



REMINDER: SCHEDULING

How are CTAs executed?

Abstraction that is defined by user

Independent of the actual architecture

=> CTAs are opaque to the user

How are thread warps executed?

Abstraction that is defined by JIT compiler

Architecture-dependent

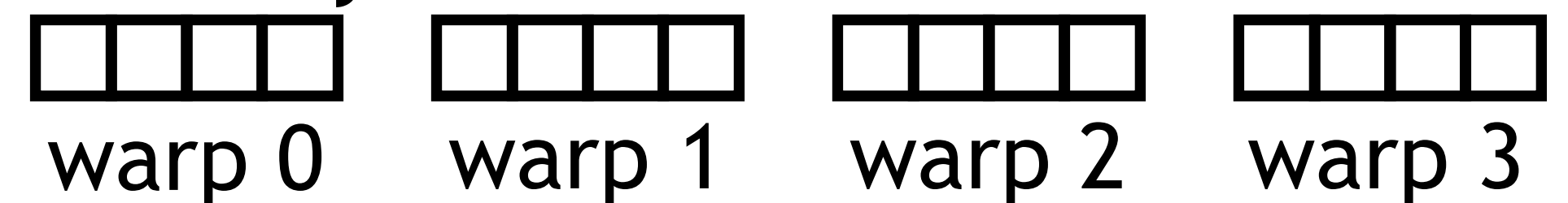
=> Warps are transparent to the user

Both are scheduling entities

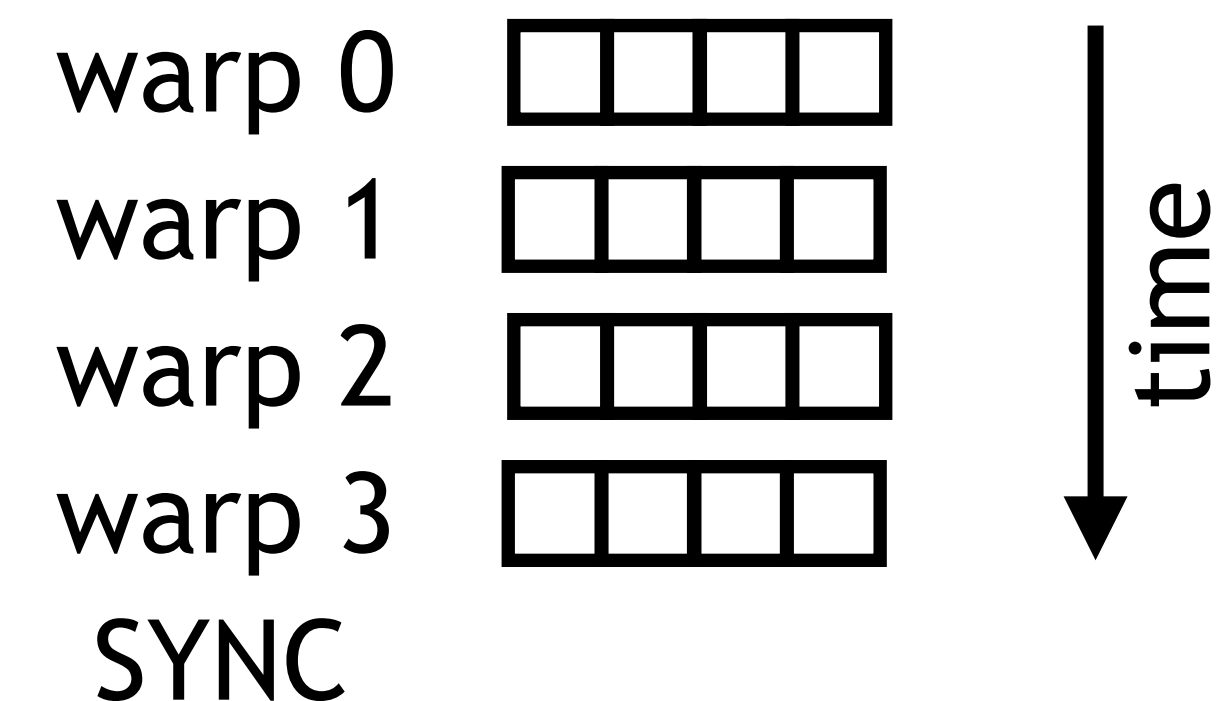
Threads in a CTA:



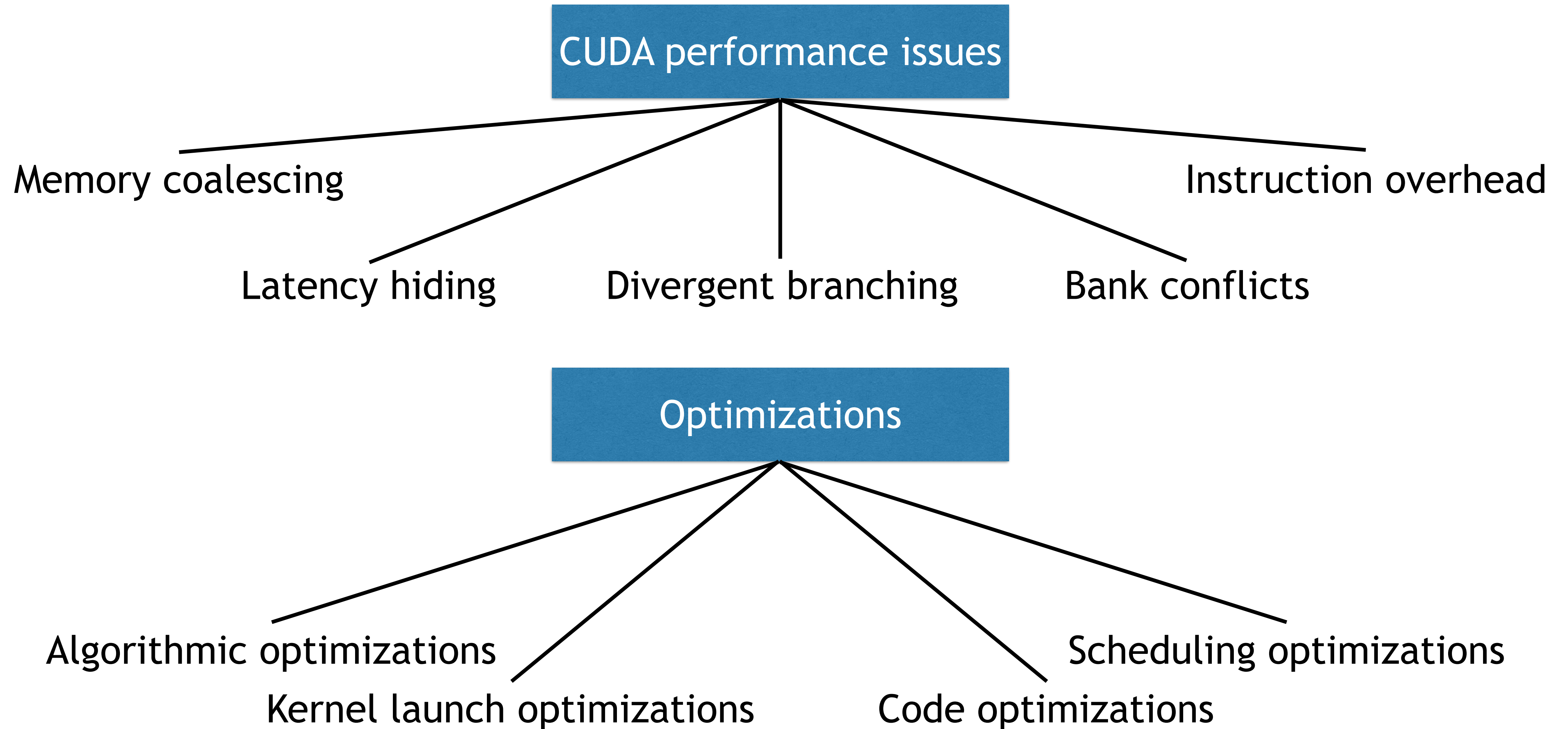
Actually look like this:



And are executed like this:



OPTIMIZATIONS



REDUCTION EXAMPLE

PARALLEL REDUCTION

Common and important data parallel primitive

Global sum, histogram, etc.

Often associative operations -> reordering opportunity :)

Pretty easy to implement in CUDA

Way harder to get it right (fast)

Optimization example for scheduling issues

6 different versions here (could be more)

$$s = \sum_{i=0}^N f(x_i)$$

$$p = \prod_{i=0}^N f(x_i)$$

$$h_k = \sum_{i=0}^N (x_i == k) ? 1 : 0$$

PARALLEL REDUCTION ON A GPU

Tree-based reduction within each CTA

=> Multiple CTAs required

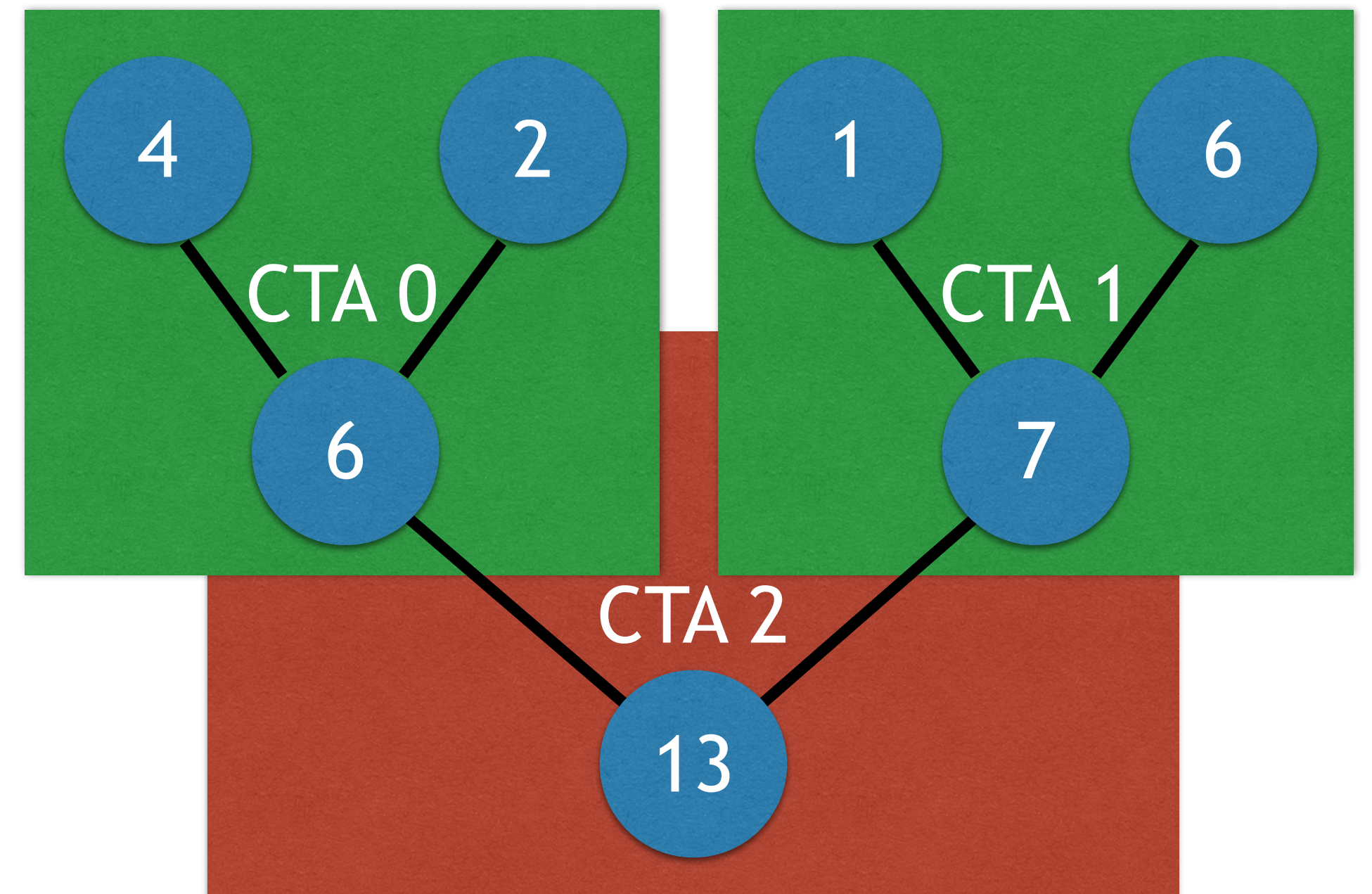
To process very large arrays

For high utilization of the GPU (one CTA per SM)

How to communicate/synchronize partial results between CTAs?

Kernel completion boundaries to the rescue!

I.e., kernel re-launch



EXCURSION: GLOBAL SYNCHRONIZATION

Global synchronization would solve this problem and many others easily

But there is no global synchronization! Why?

Global operations expensive under scalability constraints

High SM count

Impact on scheduling guarantees (progress)

Scheduling is non-preemptive

Can't synchronize more CTAs than can execute concurrently

Would limit block count to: $\#CTAs \leq \#SMs * resident_blocks_per_SM$

“Persistent threads”

Conflictive with required parallel slackness to hide memory latency

KERNEL DECOMPOSITION

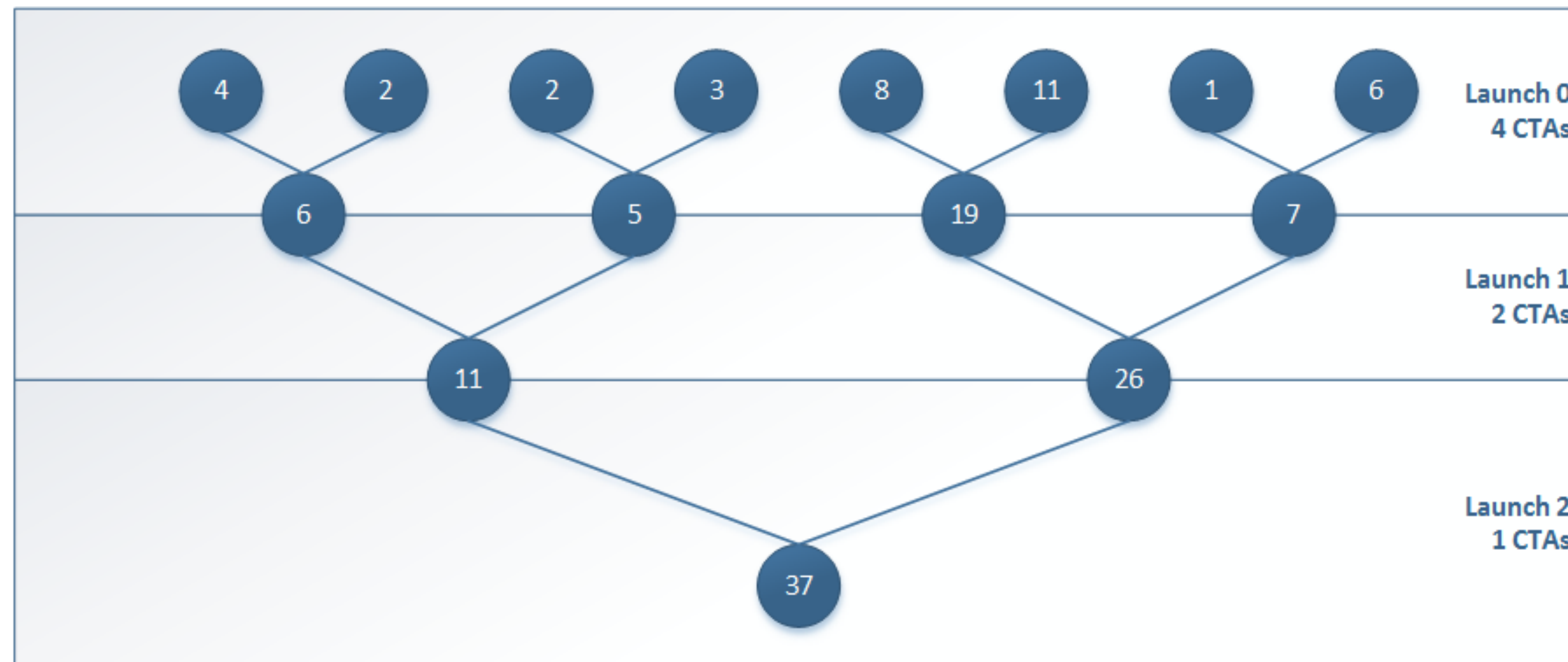
Solution: decompose into multiple kernels

Kernel completion boundary serves as global synchronization point

Negligible HW overhead, low SW overhead

For reductions, code for all levels is the same

Associativity: $a + (b + c) = (a + b) + c$



SIX REDUCTION OPTIMIZATIONS

METHODOLOGY

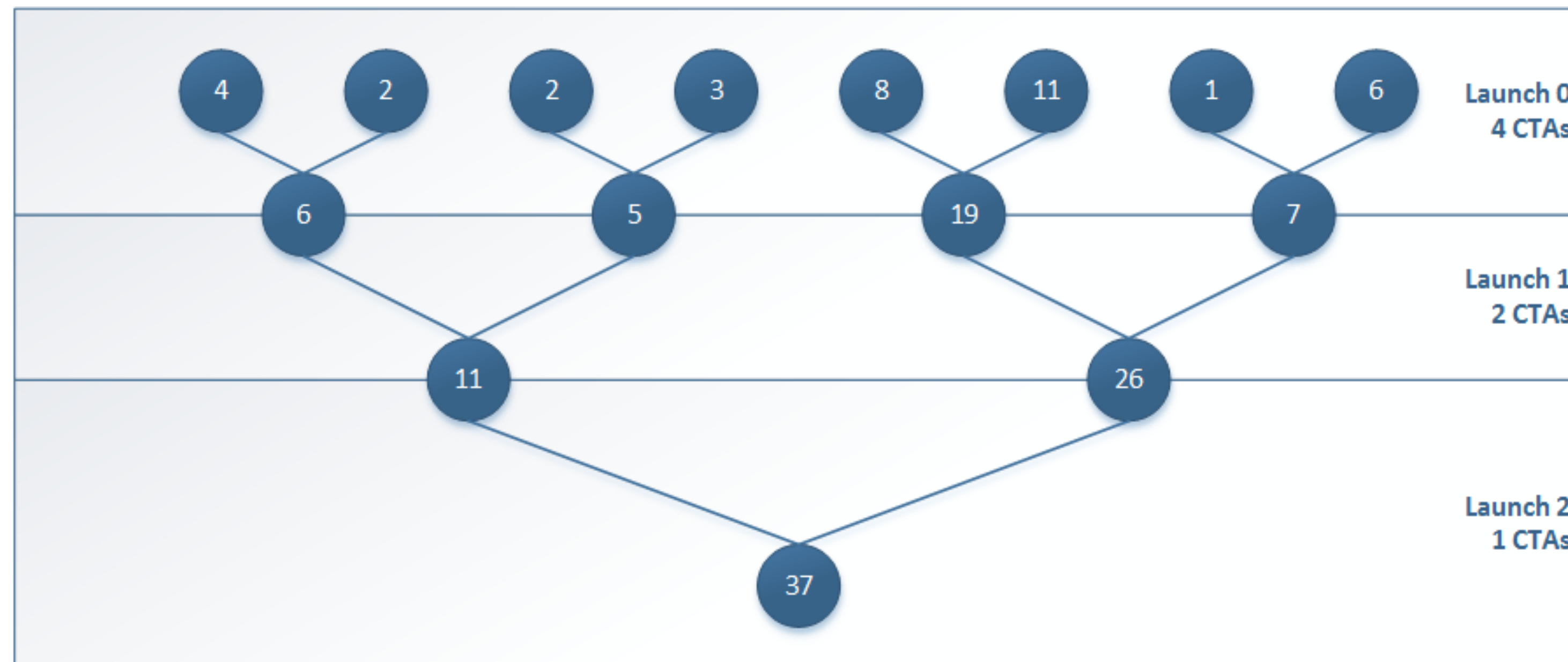
4M elements

Vary thread count (per CTA) for performance analysis

Note that we keep thread count constant for all iterations

Other solutions differ regarding this

Subject to next optimization (exercise)



REDUCTION #1: INTERLEAVED ADDRESSING

`blockDim.x` must be a power-of-two

1. Collective load

Coalescing issues?

2. Utilization

Every n -th thread computes, stride increases with loop iteration

3. Synchronization


Why `syncthreads`?

```
__global__ void Reduction0a_kernel( int *out, int *in, size_t N )
{
    extern __shared__ int sPartials[];
    const int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    // each thread loads one element from global to shared mem
    sPartials[tid] = in[i];
    __syncthreads();

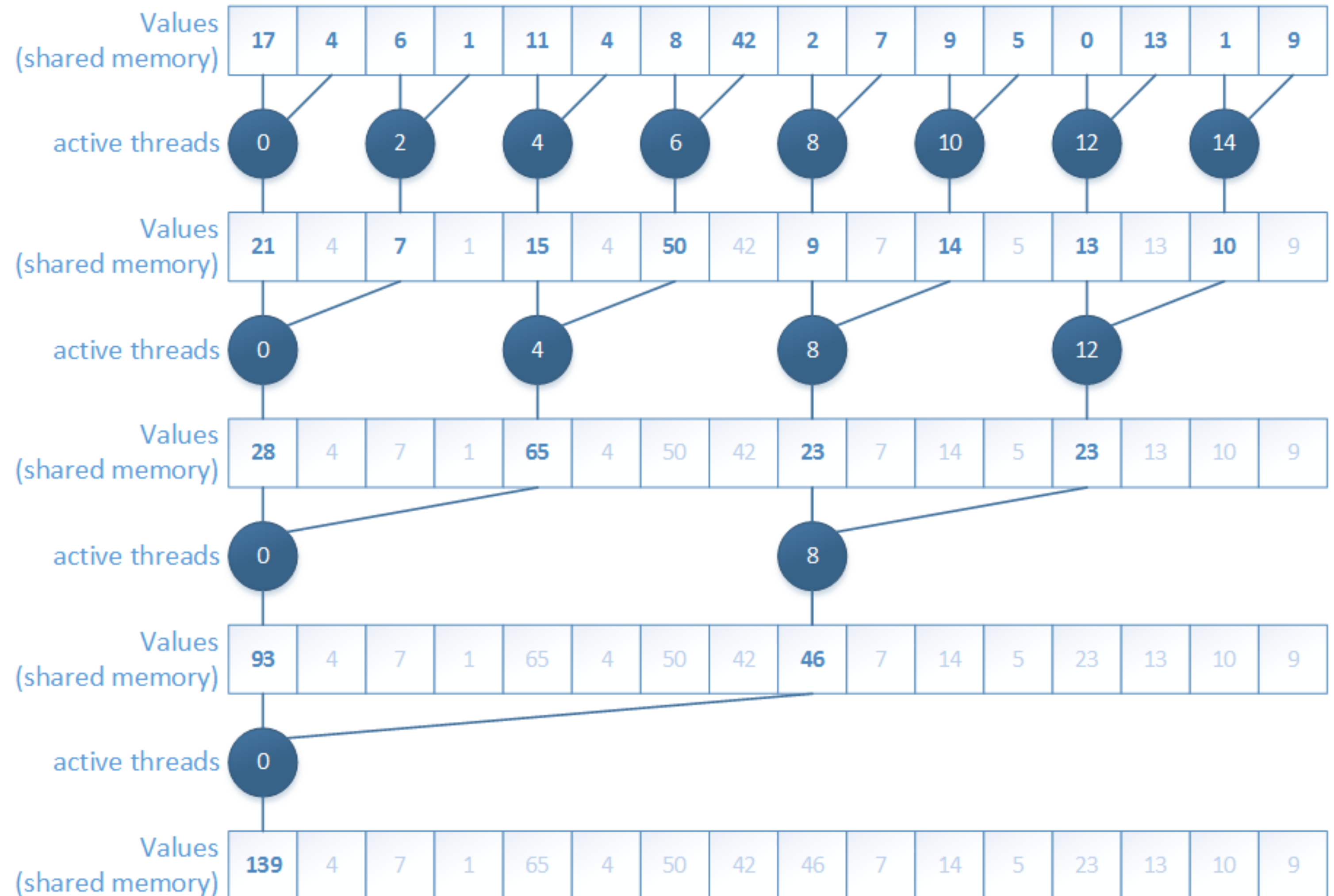
    // do reduction in shared mem
    for ( unsigned int s = 1; s < blockDim.x; s *= 2 ) {
        if ( tid % ( 2 * s ) == 0 ) {
            sPartials[tid] += sPartials[tid + s];
        }
        __syncthreads();
    }

    if ( tid == 0 ) {
        out[blockIdx.x] = sPartials[0];
    }
}
```



REDUCTION #1: INTERLEAVED ADDRESSING

Two loads per thread,
but sequentially



REDUCTION #1: INTERLEAVED ADDRESSING

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77

Problem: branch divergence

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

REDUCTION #2: INTERLEAVED ADDRESSING NON-DIVERGENT

Every second thread
Every fourth thread
...

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

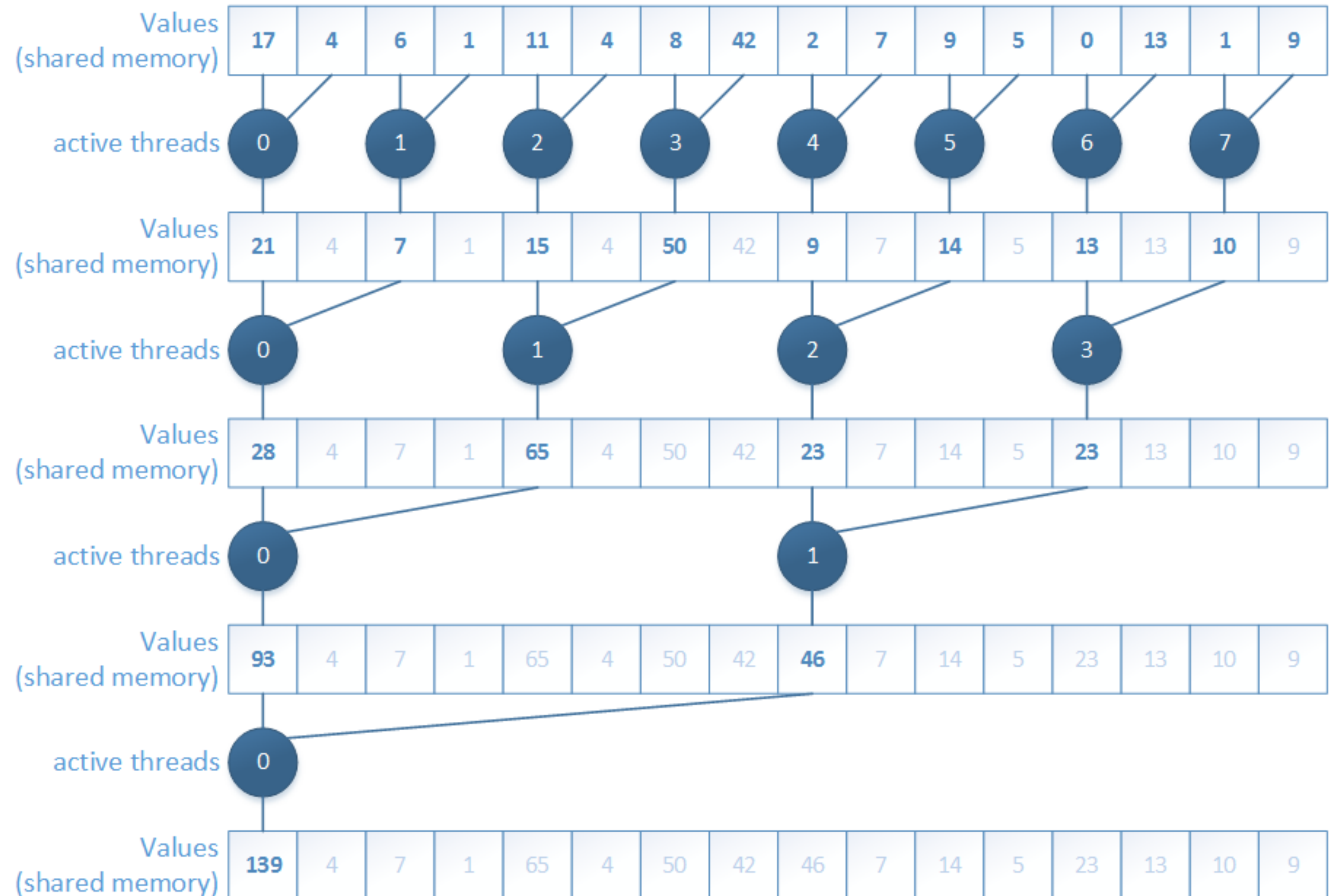
Solution: re-sort add ops
Modified if-clause

Every thread, if index
within bounds
(consecutive tid's)

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

REDUCTION #2: INTERLEAVED ADDRESSING NON-DIVERGENT

Simple renaming of threads
 Which elements belong to one bank?
 Remember these are SP-floats, i.e. 4B



REDUCTION #2: INTERLEAVED ADDRESSING NON-DIVERGENT

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77
intrlvd non-div	10,46	18,33	23,88	18,96	14,5	10,02	128	23,88

New problem: shared memory bank conflicts

Shared memory is best accessed using `tid`

```
int index = 2 * s * tid;  
if ( index < blockDim.x ) {  
    sPartials[index] += sPartials[index + s];  
}
```

REDUCTION #3: SEQUENTIAL ADDRESSING NON-DIVERGENT

Replace strided indexing with thread-ID based indexing

=> Block access instead of stride index

Start with half the threads being active
(`blockDim.x/2`) = offset

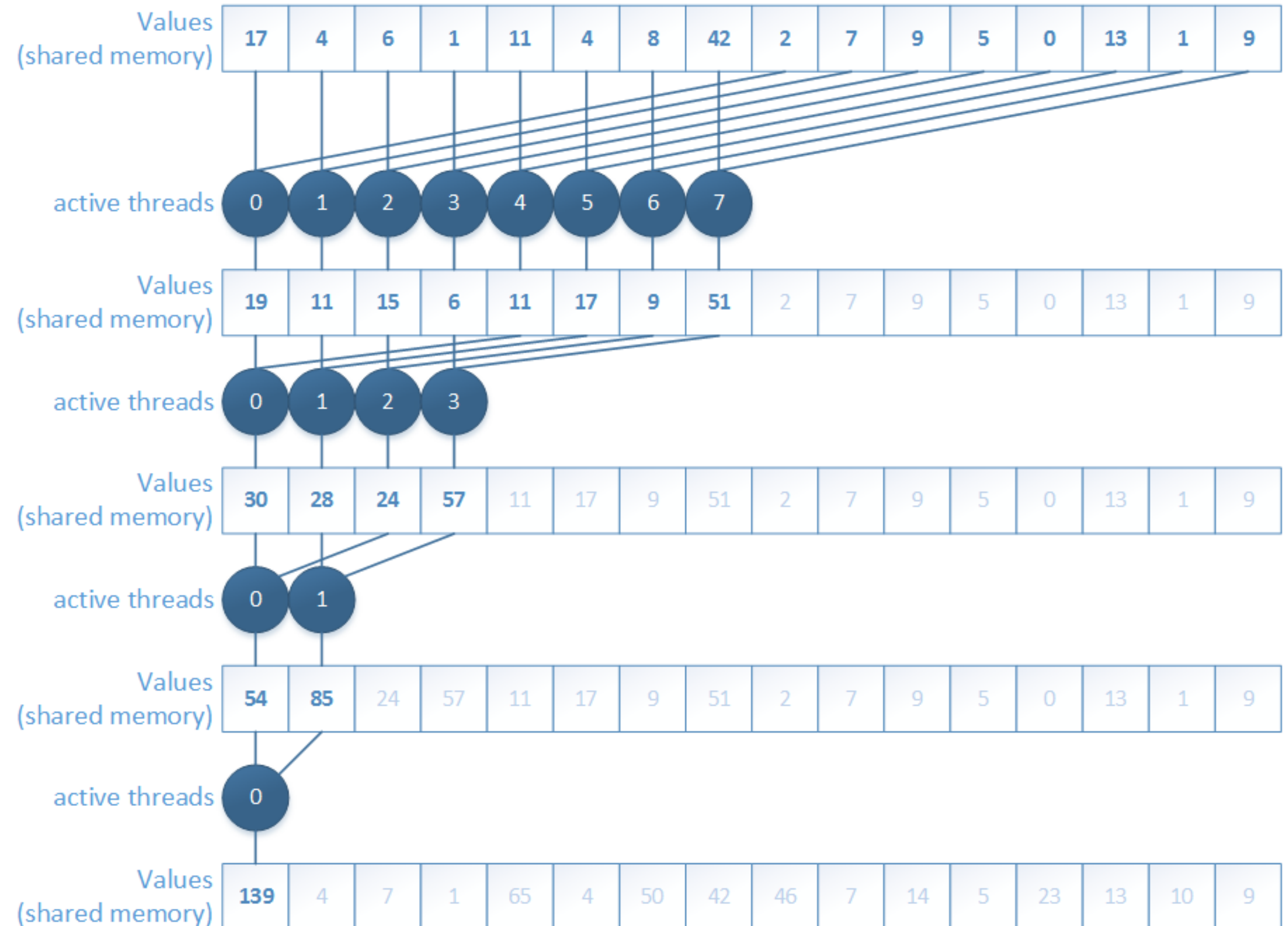
Addition of elements
`tid` and `tid+s`

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

```
<snip>
// do reduction in shared mem
for ( unsigned int o = blockDim.x / 2; o > 0; o >>= 1 ) {
    if ( tid < o ) {
        sPartials[tid] += sPartials[tid + o];
    }
    __syncthreads();
}
<snip>
```

REDUCTION #3: SEQUENTIAL ADDRESSING NON-DIVERGENT

Block-wise access to shared memory



REDUCTION #3: SEQUENTIAL ADDRESSING NON-DIVERGENT

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77
intrlvd non-div	10,46	18,33	23,88	18,96	14,5	10,02	128	23,88
seq. non-div	11,05	19,54	30,83	27,51	23,67	17,99	128	30,83

New problem: idle threads

During the first operation, half of the threads are idling!

o starts with `blockDim.x/2`

```
if ( tid < o ) {  
    sPartials[tid] += sPartials[tid + o];  
}  
__syncthreads();
```

REDUCTION #4: FIRST ADD DURING LOAD

Modified kernel launch
with only half the
number of blocks

Replace single load
with 2 loads and first
add

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
// each thread loads one element from global to shared mem
sPartials[tid] = in[i];
__syncthreads();
...
for ( unsigned int o = blockDim.x / 2; o > 0; o >>= 1 ) {
    if ( tid < o ) {
        sPartials[tid] += sPartials[tid + o];
    }
}
...
```

```
...
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
// perform first level of reduction
// read from global memory, write to local memory
sPartials[tid] = in[i] + in[i+blockDim.x];
__syncthreads();

for ( unsigned int o = blockDim.x / 2; o > 0; o >>= 1 ) {
    if ( tid < o ) {
        sPartials[tid] += sPartials[tid + o];
    }
    __syncthreads();
}
...
```

REDUCTION #4: FIRST ADD DURING LOAD

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77
intrlvd non-div	10,46	18,33	23,88	18,96	14,5	10,02	128	23,88
seq. non-div	11,05	19,54	30,83	27,51	23,67	17,99	128	30,83
first add	21,68	37,15	58,03	51,31	43,75	33,66	128	58,03

Still far from peak

Instruction overhead

Instructions for control flow that are no loads, stores or core computations

Address arithmetic, loop control

REDUCTION #5: UNROLLING THE LAST WARP

Number of active threads decreases over time

Remember that a warp consists of 32 threads

Implementation-dependent

Instructions are synchronous within a warp

Scheduler broadcasts instructions, threads can nullify the output

i.e., for $s \leq 32$ only one warp left

=> No need for `__syncthreads()`

=> No need for `if (tid < s)`

Loop unrolling the last 6 iterations of the inner loop

REDUCTION #5: UNROLLING THE LAST WARP

Without
unrolling, all
warps execute
every instruction

```
__global__ void Reduction0e_kernel( int *out, int *in, bool echo )
{
    extern __shared__ int sPartials[];
    const int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    // perform first level of reduction
    // read from global memory, write to local memory
    sPartials[tid] = in[i] + in[i+blockDim.x];
    __syncthreads();

    for ( unsigned int s = blockDim.x / 2; s > 32; s >>= 1 ) {
        if ( tid < s ) {
            sPartials[tid] += sPartials[tid + s];
        }
        __syncthreads();
    }

    if ( tid < 32 && blockDim.x >= 64) sPartials[tid] += sPartials[tid + 32];
    if ( tid < 16 && blockDim.x >= 32) sPartials[tid] += sPartials[tid + 16];
    if ( tid < 8 && blockDim.x >= 16) sPartials[tid] += sPartials[tid + 8];
    if ( tid < 4 && blockDim.x >= 8) sPartials[tid] += sPartials[tid + 4];
    if ( tid < 2 && blockDim.x >= 4) sPartials[tid] += sPartials[tid + 2];
    if ( tid < 1 && blockDim.x >= 2) sPartials[tid] += sPartials[tid + 1];

    if ( tid == 0 ) {
        out[blockIdx.x] = sPartials[0];
    }
}
```


REDUCTION #5: UNROLLING THE LAST WARP

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77
intrlvd non-div	10,46	18,33	23,88	18,96	14,5	10,02	128	23,88
seq. non-div	11,05	19,54	30,83	27,51	23,67	17,99	128	30,83
first add	21,68	37,15	58,03	51,31	43,75	33,66	128	58,03
unrolling	22,59	36,91	68,38	62,35	53,06	43,78	128	68,38

Complete unrolling?

We need to know the number of iterations at compile time

- Limit of 1024 threads per block
- Power-of-two block sizes

Easy unroll for a fixed block size

How to stay generic though?

-> C++ Templates!

- Template parameters will be evaluated at compile time

- Larger code

REDUCTION #6: COMPLETE UNROLLING

Template
parameters are
evaluated at
compile time
=> Inner loop
highly optimized

```
template <unsigned int blockSize> __global__ void Reduction0f_kernel( int *out,
                                                                    int *in, bool echo )
{
    extern __shared__ int sPartials[];
    const int tid = threadIdx.x;

    unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
    // perform first level of reduction
    // read from global memory, write to local memory
    sPartials[tid] = in[i] + in[i+blockSize];
    __syncthreads();

    if (blockSize >= 1024) {
        if (tid < 512) { sPartials[tid] += sPartials[tid + 512]; } __syncthreads();
    }
    if (blockSize >= 512) {
        if (tid < 256) { sPartials[tid] += sPartials[tid + 256]; } __syncthreads();
    }
    if (blockSize >= 256) {
        if (tid < 128) { sPartials[tid] += sPartials[tid + 128]; } __syncthreads();
    }
    if (blockSize >= 128)
        if (tid < 64) { sPartials[tid] += sPartials[tid + 64]; } __syncthreads();
    }

    if ( tid < 32 && blockSize >= 64) sPartials[tid] += sPartials[tid + 32];
    if ( tid < 16 && blockSize >= 32) sPartials[tid] += sPartials[tid + 16];
    if ( tid < 8 && blockSize >= 16) sPartials[tid] += sPartials[tid + 8];
    if ( tid < 4 && blockSize >= 8) sPartials[tid] += sPartials[tid + 4];
    if ( tid < 2 && blockSize >= 4) sPartials[tid] += sPartials[tid + 2];
    if ( tid < 1 && blockSize >= 2) sPartials[tid] += sPartials[tid + 1];
    if ( tid == 0 ) {
        out[blockIdx.x] = sPartials[0];
    }
}
```

REDUCTION #6: COMPLETE UNROLLING

Avoiding block size at compile time completely by using a switch statement

Here: block size has to be a power of two

=> only 10 possible block sizes

```
void Reduction0f_wrapper ( int dimGrid, int dimBlock, int smemSize, int *out, int *in, bool echo )
{
    switch ( dimBlock ) {
        case 1024:
            Reduction0f_kernel<1024><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
        case 512:
            Reduction0f_kernel< 512><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
        case 256:
            Reduction0f_kernel< 256><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
        ... <snip> ...
        case 4:
            Reduction0f_kernel< 4><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
        case 2:
            Reduction0f_kernel< 2><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
        case 1:
            Reduction0f_kernel< 1><<< dimGrid, dimBlock, smemSize >>>(out, in, echo); break;
    }
}
```

REDUCTION #5: UNROLLING THE LAST WARP

Throughput [GB/s]	32	64	128	256	512	1024	maxThr	maxBW
intrlvd div	7,39	12,57	16,77	14,67	12,33	9,05	128	16,77
intrlvd non-div	10,46	18,33	23,88	18,96	14,5	10,02	128	23,88
seq. non-div	11,05	19,54	30,83	27,51	23,67	17,99	128	30,83
first add	21,68	37,15	58,03	51,31	43,75	33,66	128	58,03
unrolling	22,59	36,91	68,38	62,35	53,06	43,78	128	68,38
templated	26,47	41,19	42,98	40,01	34,1	29,78	128	42,98

Less performance :/

Code size increase?

Next optimization could look at optimizing the amount of ILP

I.e. multiple adds per thread

Not shown here anymore

TYPES OF OPTIMIZATION

Algorithmic optimizations

Changes to addressing

See examples

Algorithm cascading

Not shown here

In essence, combine sequential and parallel reduction by having a thread sum multiple elements

=> Increasing ILP

Code optimizations

Loop unrolling within thread warps

See examples

Templating

Note that templating had little success, likely because of the non-optimal number of iterations (resp. the second iteration should use a different block size)

VOLTA'S INDEPENDENT THREAD SCHEDULING

PASCAL'S (AND BEFORE) SIMT MODEL

Single program counter per warp,
combined with an “active mask”,
and single call stack

Resource efficient

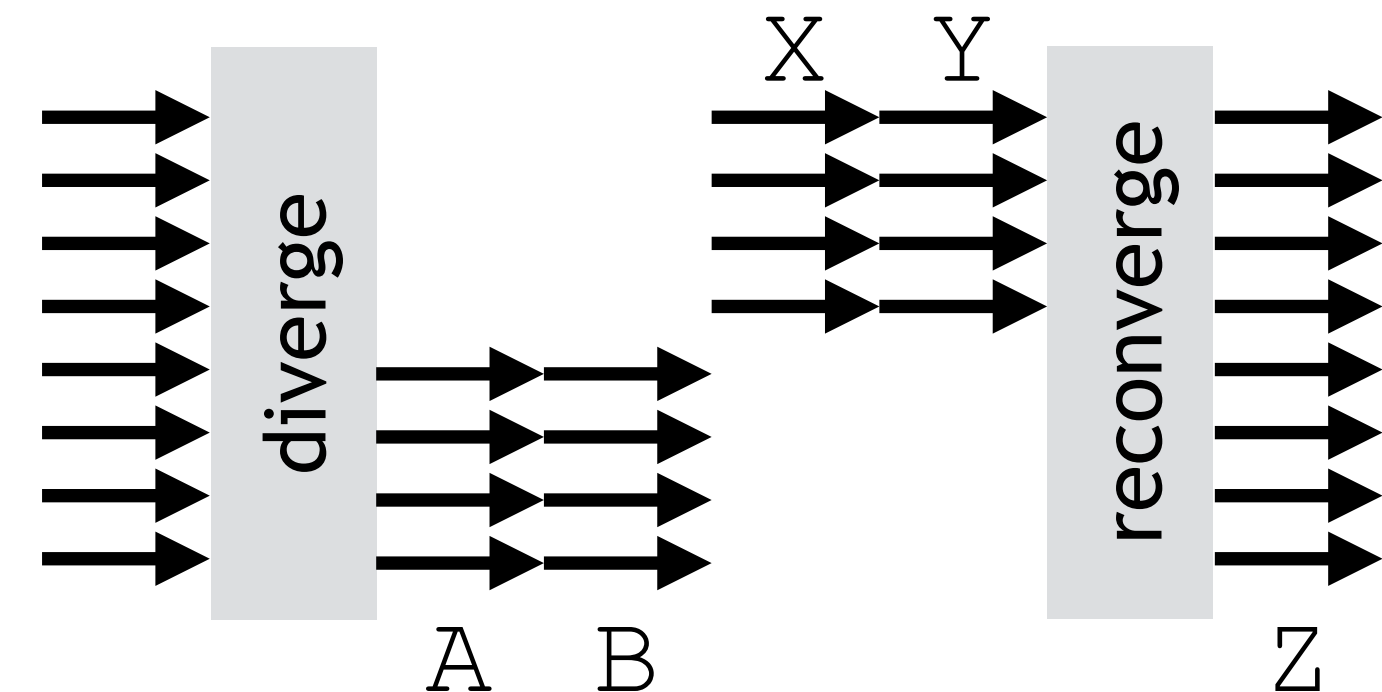
Performance penalty for divergent
control flow -> branch serialization

Deadlock possibility

When sharing data among non-coherent
threads of a single warp

-> Avoid fine-grain synchronization or
use lock-free algorithms

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}
```



VOLTA'S (AND AFTER) SIMT MODEL

Independent Thread Scheduling (ITS)

Maintains execution state per thread

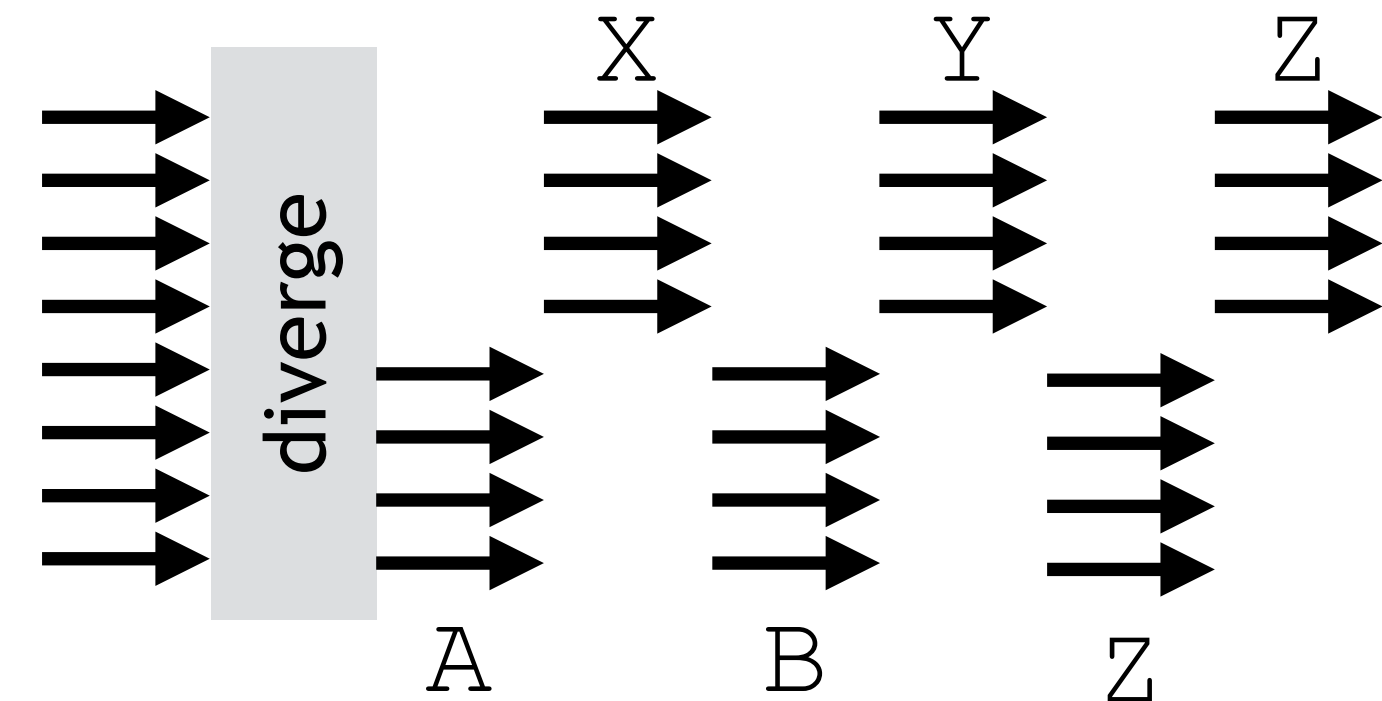
Yielding any thread is now possible

Schedule optimizer: *“determines how to group active threads from the same warp together into SIMT units”*

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}
```

Execution is still SIMT

“... threads can now diverge and reconverge at sub-warp granularity, and Volta will still group together threads which are executing the same code and run them in parallel.”



Z in the example is not reconverged

Conservative: if statements A,B,X,Y all contain no synchronization operation, it is safe to reconverge on Z

Warp synchronization `__syncwarp()` to force reconvergence

STARVATION-FREE ALGORITHMS

ITS supports starvation-free algorithms

Aka finite bypass: any process (or concurrent part) of an algorithm is bypassed at most a finite number times before being allowed access to the shared resource

Guaranteed to execute correctly so long as the system ensures that all threads have eventually (fair) access to a contended resource

Consider a lock (mutual exclusion)

Thread #0 holds the lock, but thread #1 is scheduled for execution and impedes the progress of thread #0

Volta's ITS: thread #0 will eventually (question of when, not if) be scheduled for execution

WARP-LEVEL INSTRUCTIONS

A shuffle instruction (SHFL) enables a thread to directly read a register from another thread of the same warp

Since Kepler

Four shuffle intrinsics: `__shfl()`, `__shfl_down()`, `__shfl_up()`, `__shfl_xor()`

```
int __shfl_down(int var, unsigned int delta, int width=warpSize);
```

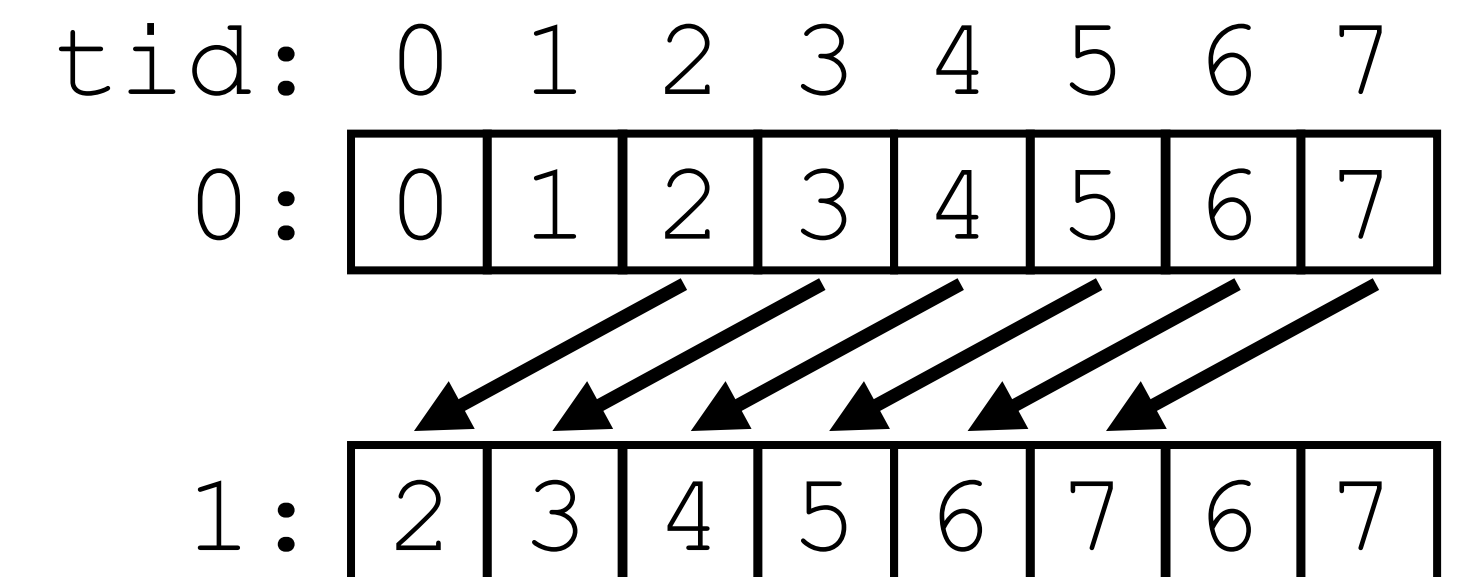
A shuffle instruction replaces a multi-instruction shared memory sequence

Increase effective bandwidth (+ reduce latency)

Reduce shared memory usage

Pre-Volta: no need for synchronization as execution is warp-synchronous

```
0: int i = threadIdx.x % 32;  
1: int j = __shfl_down(i, 2, 8);
```



ESCAPE THE NEW FEATURE

Use warp-level primitives in their sync-variant

E.g., `void __syncwarp(unsigned mask=FULL_MASK)`

Or implement a warp-level reduction tree using `__shfl_down_sync()`

Use the new concept of cooperative thread groups

<https://developer.nvidia.com/blog/cooperative-groups/>

Compile for Pascal architecture

`nvcc` with options `-arch=compute_60 -code=sm_70`

Exercise

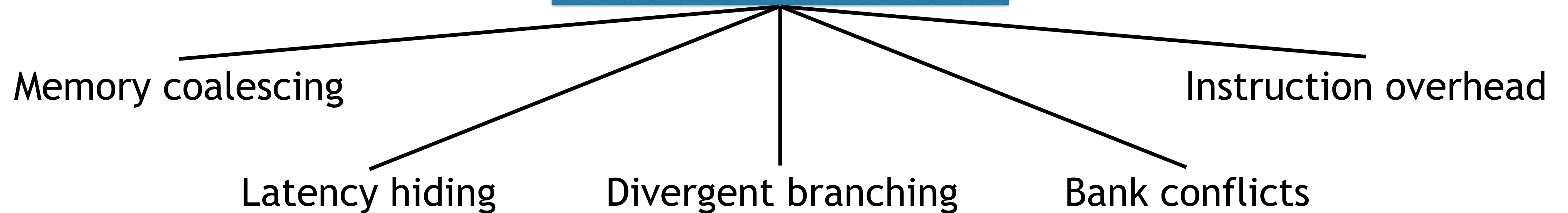
First part: focus on compilation-based escape

Second part: use either cooperative thread groups or warp-level primitives, compare to performance of first part

WRAPPING UP

SUMMARY

CUDA performance issues



Optimizing code

1. Choose right performance goal (GFLOP/s or GB/s)
2. Identify type of bottleneck: memory, computations, instruction overhead
3. Optimize the algorithm
4. Unroll loops
5. Templating for optimal code

Good optimizations know when to stop
=> Maintain readability, maintainability, portability