Lecture 6: Optimizing CUDA Programs Part 1: Reduction

CS599: Programming Massively Parallel Multiprocessors and Heterogeneous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

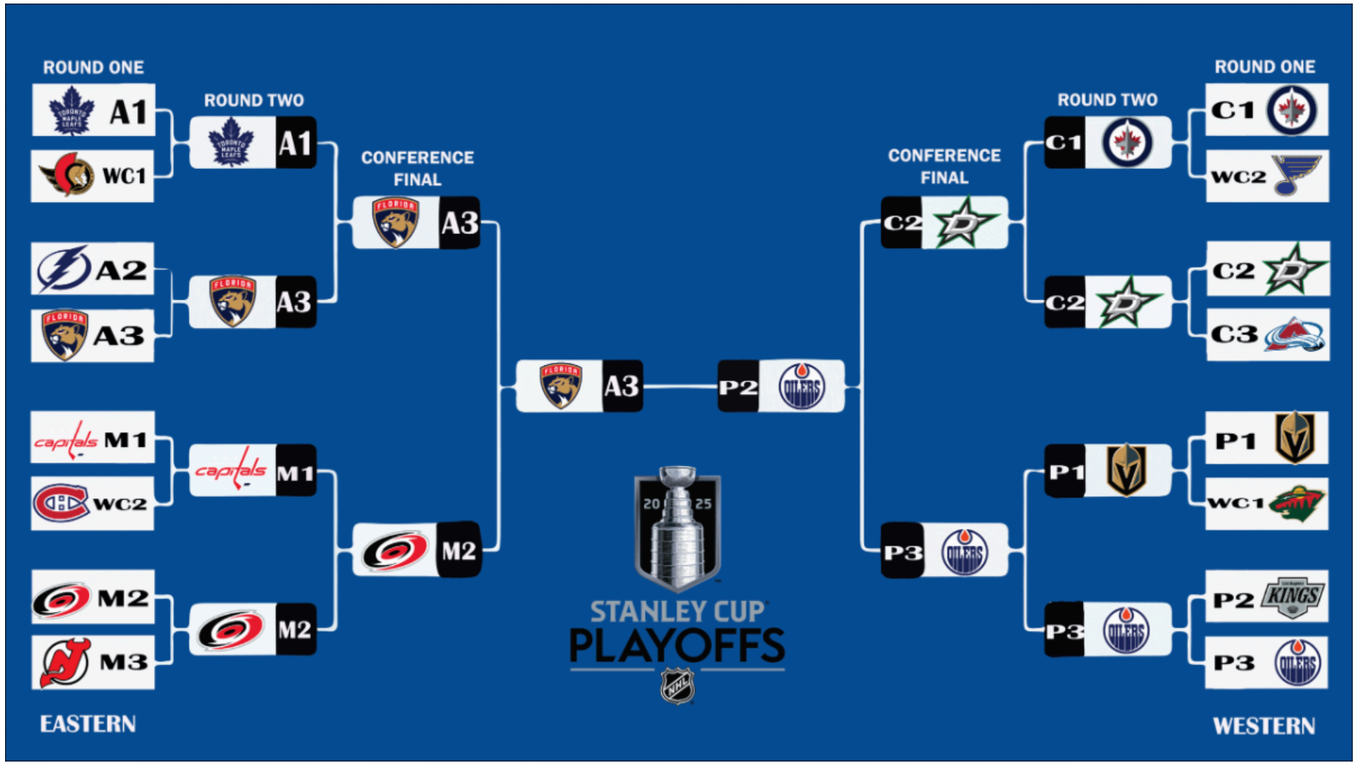
Optimizing Reduction

Slide Title

- What is reduction?
- Has become a canonically CUDA optimization (see Mark Harris's webinar, PMPP, net) example to explore
  - We are going to take our own journey on the V100
    1. Base case (assume very large vector and basic parallel algorithm as starting point)
    2. Reduce thread divergence
    3. Reduce SMEM bank conflicts
    4. Explore improving efficiency
    5. Note many more things to try

What is reduction?

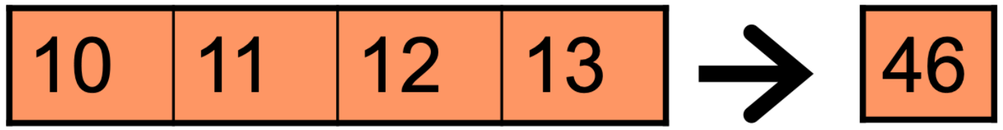
Let there be one



Reductions

- Objective: reduce multiple values into a single value with an operation such as
  - ADD, MUL, AND, OR, MIN, MAX ….
  - Assume operation is
    - associative $(a \boxdot b) \boxdot c \equiv a \boxdot (b \boxdot c)$
    - commutative $a \boxdot b \equiv b \boxdot a$

- E.g.:



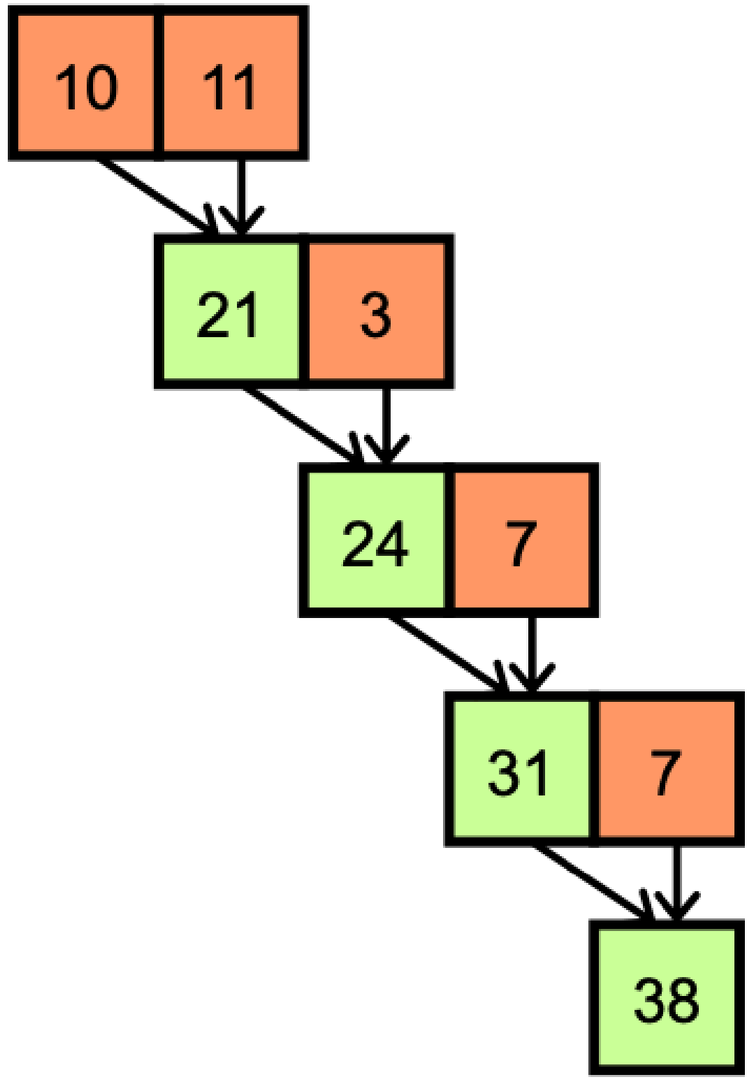
- Useful primitive, often part of a larger computation e.g., map-reduce, sorting, count++

- Simple and easy enough to allow us to focus on optimization techniques

Sequential Reduction

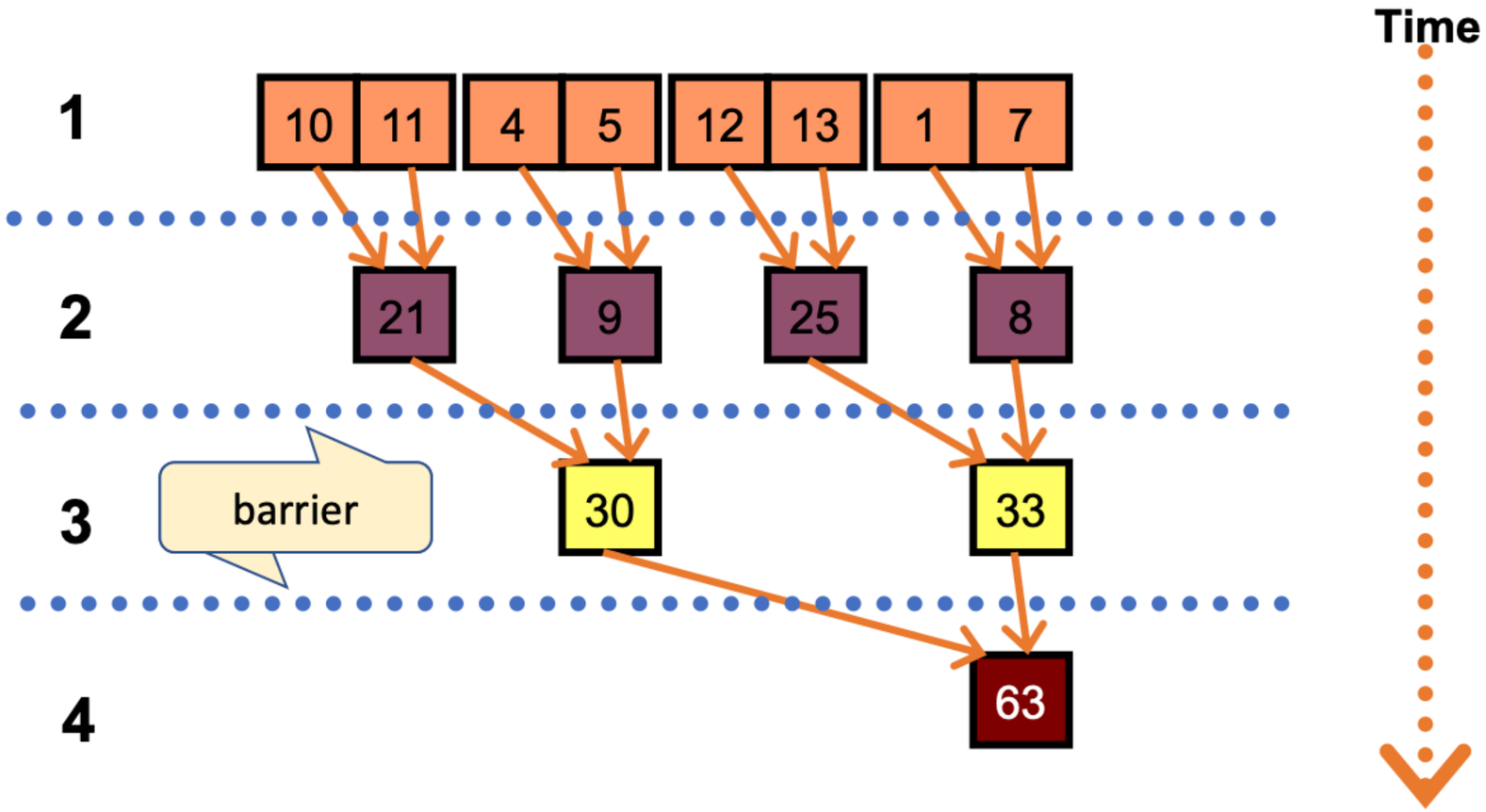
- Start with the first two elements
  --> partial result
- In a loop
  - Process the next element
  - Until all elements processed

➔ O(*N*)



Reduction: How to parallelize

Idea: pairwise reduction in steps:
tree-like computation structure:



➔ $\log_2$ N steps

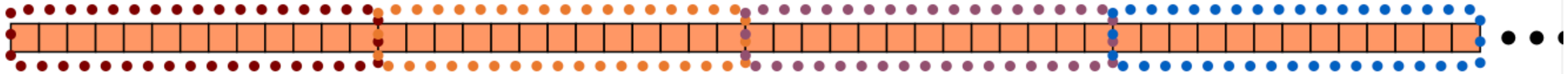
➔ Key issue: how to barrier

Possible CUDA Strategies

- Single block

  + synchronization easy: __syncthreads()
  - max 1024 threads; parallelism limited by # cores / SMX
    ➔ low resource utilization
    ➔ limited speedup

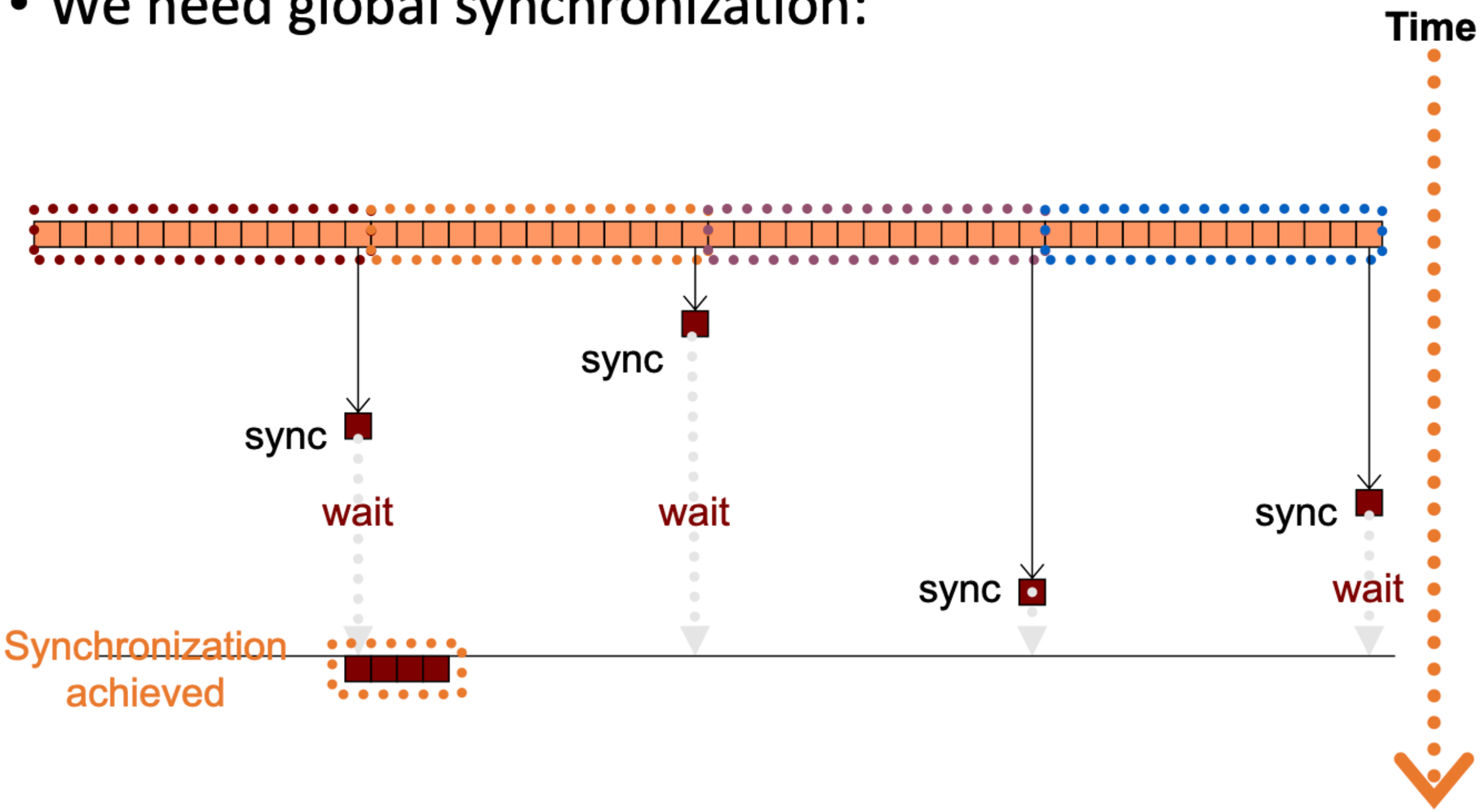
- Multiple blocks: have each block process a portion of data



  - key issue: synchronization:
    how do we know each block has finished

Multiple block strategy problem

- We need global synchronization:

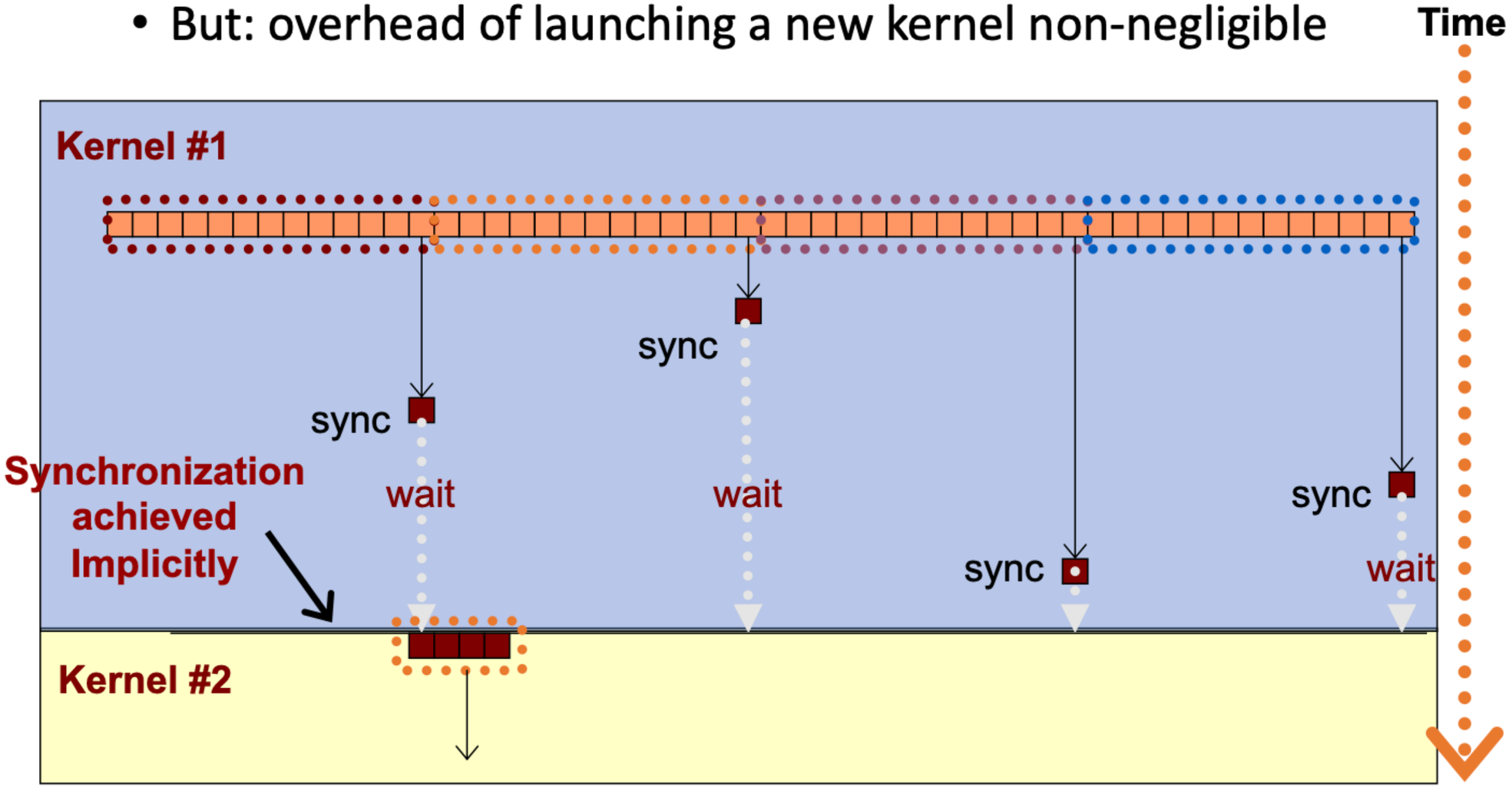


- **But CUDA does not offer global synchronization!**

Well not traditionally and regardless can you see why it could be problematic?

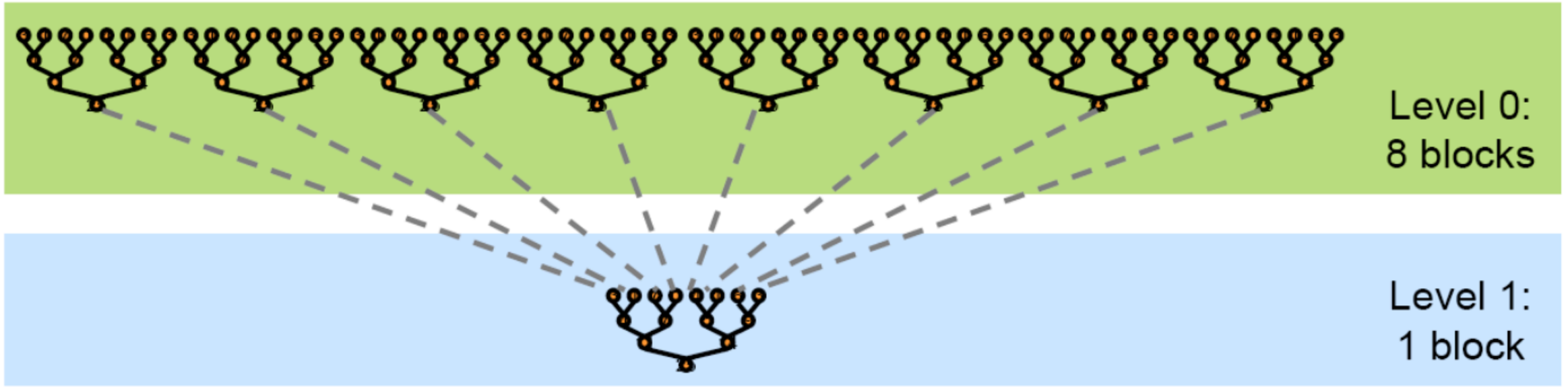
Sync by decomposing into multiple kernels

- Implicit Synchronization between kernel invocations
  - But: overhead of launching a new kernel non-negligible



- Note: in this case Kernel #1 ≡ Kernel #2

Decomposition strategy: big picture

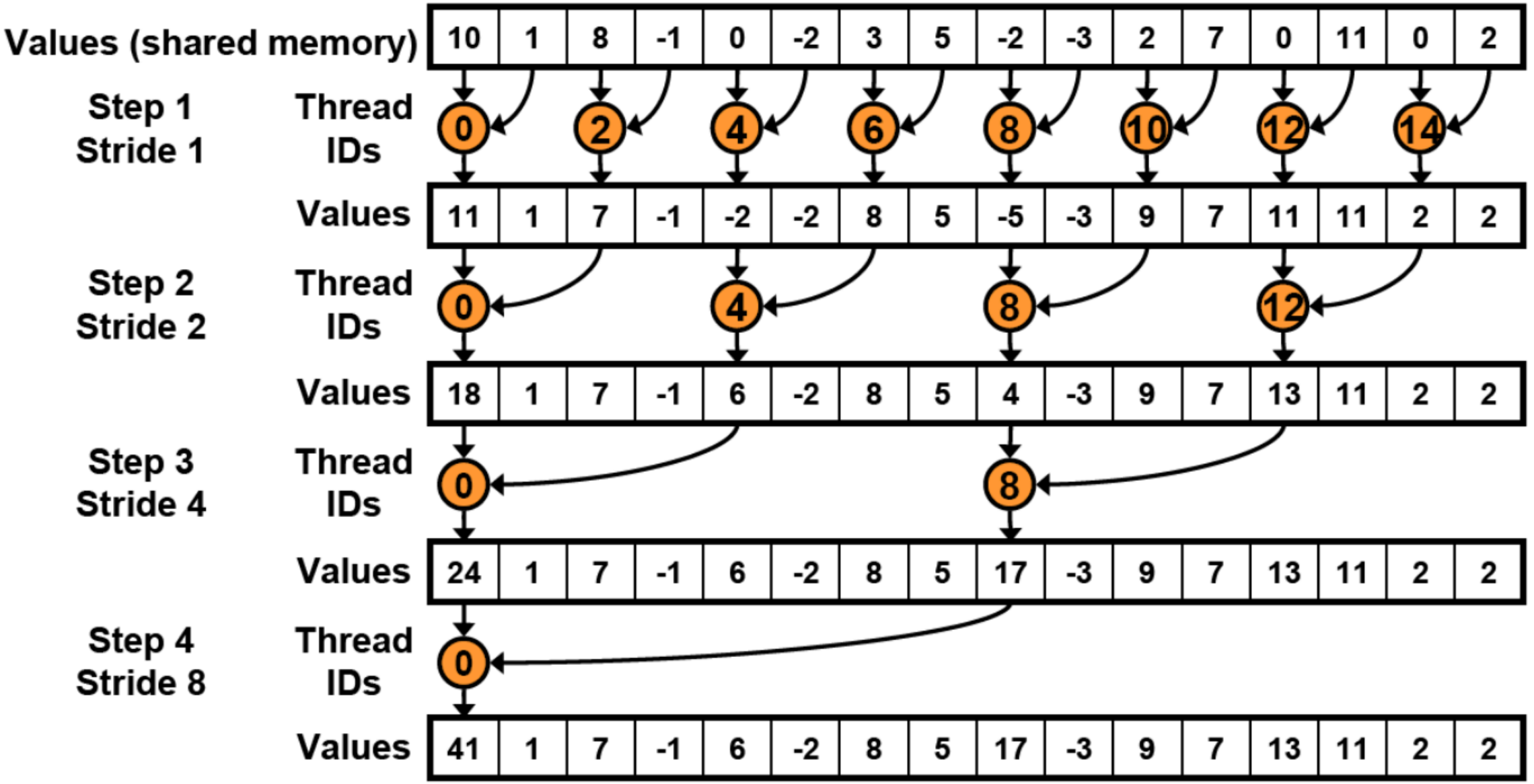


- **The code for all levels is the same**
- **The same kernel code can be called multiple times**

Reduction #1: strategy for each block

- Make GMEM accesses coalesced
  - Load data into shared memory:
    - Each thread loads 1 element from GMEM to SMEM
- Reduce: Proceed in log *N* steps
  - Each thread reduces two elements
    - The first two elements by the first thread
    - The next two by another thread
    - And so, on
  - At the end of each step:
    - Deactivate half of the threads
  - Terminate: when one thread left
- Write result back to global memory

Reduction #1: Steps within block



Reduction #1: kernel code

```
__global__ void reduce1(float *d_ivec, float *d_ovec, unsigned int n) {
  extern __shared__ float sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i<n) ? d_ivec[i] : 0.0;
  __syncthreads();

  for (unsigned int s=1; s<blockDim.x; s *= 2) {
    if ( (tid % (2*s)) == 0 ) {
      sdata[tid] += sdata[tid+s];
    }
    __syncthreads();
  }

  if (tid==0) d_ovec[blockIdx.x] = sdata[0];
}
```

Reduction #1: kernel code

```
__global__ void reduce1(float *d_ivec, float *d_ovec, unsigned int n) {
  extern __shared__ float sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i<n) ? d_ivec[i] : 0.0;
  __syncthreads();




}
```

each threads copies an element
from GMEM to SMEM

Reduction #1: kernel code

```
__global__ void reduce1(float *d_ivec, float *d_ovec, unsigned int n) {
  extern __shared__ float sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i<n) ? d_ivec[i] : 0.0;
  __syncthreads();

  for (unsigned int s=1; s<blockDim.x; s *= 2) {
    if ( (tid % (2*s)) == 0 ) {
      sdata[tid] += sdata[tid+s];
    }
    __syncthreads();
  }


}
```

s is stride: 2, 4, 8, 16,...

only tids divisible by s*2 do partipate

reduction in SMEM

each iteration is a parallel level of reduction
__syncthread is **in** Loop!

Reduction #1: kernel code

```
__global__ void reduce1(float *d_ivec, float *d_ovec, unsigned int n) {
  extern __shared__ float sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i<n) ? d_ivec[i] : 0.0;
  __syncthreads();

  for (unsigned int s=1; s<blockDim.x; s *= 2) {
    if ( (tid % (2*s)) == 0 ) {
      sdata[tid] += sdata[tid+s];
    }
    __syncthreads();
  }

  if (tid==0) d_ovec[blockIdx.x] = sdata[0];
}
```

0th thread of block writes result of block reduction to global memory

Reduction #1: host code

```
rc = cudaMalloc((void **)&d_mem, 2*bytes);
assert(rc == cudaSuccess);
rc = cudaMemcpy(d_mem, h_vec, bytes, cudaMemcpyHostToDevice);
assert(rc == cudaSuccess);

float *d_ivec = (float *)d_mem;
float *d_ovec = &(d_ivec[n]);

for (int len=n; len>1; len=len/blksize) {
  gridsize = (len + blksize - 1) / blksize; // ceil(len/blksize)

  reduceKernel<<<gridsize,blksize,blksize*sizeof(float)>>>(d_ivec,d_ovec,n);

  d_ivec   = d_ovec;                  // output of reduction is input for next
  d_ovec   = &(d_ivec[gridsize]); // place next ouput to the right of input
}

rc = cudaDeviceSynchronize();
assert(rc == cudaSuccess);
rc = cudaMemcpy(&dresult, d_ivec, sizeof(float), cudaMemcpyDeviceToHost);
assert(rc==cudaSuccess);
```

Reduction #1: host code

```
rc = cudaMalloc((void **)&d_mem, 2*bytes);
assert(rc == cudaSuccess);
rc = cudaMemcpy(d_mem, h_vec, bytes, cudaMemcpyHostToDevice);
assert(rc == cudaSuccess);

float *d_ivec = (float *)d_mem;
float *d_ovec = &(d_ivec[n]);

for (int len=n; len>1; len=len/blksize) {
  gridsize = (len + blksize - 1) / blksize; // ceil(len/blksize)

  reduceKernel<<<gridsize,blksize,blksize*sizeof(float)>>>(d_ivec,d_ovec,n);

  d_ivec   = d_ovec;                  // output of reduction is input for next
  d_ovec   = &(d_ivec[gridsize]); // place next ouput to the right of input
}

rc = cudaDeviceSynchronize();
assert(rc == cudaSuccess);
rc = cudaMemcpy(&dresult, d_ivec, sizeof(float), cudaMemcpyDeviceToHost);
assert(rc==cudaSuccess);
```

Reduction #1: host code

- 32, 64, 128, 256, 1024 ?
- According to Occupancy Calculator: all good -- eg. 64,128, 256, 512, 1024
- # reg 16 (according to ncu) 14 reported by nvcc

Compute Capability:	7.0	Threads Per Block:	512
Shared Memory Size Config (bytes):	98304	Registers Per Thread:	16
Global Load Cache Mode:	L1+L2 (ca)	User Shared Memory Per Block (bytes):	2048
		Block Barriers:	1

Apply Automatically | Apply | Reset

Tables | Utilization | Graphs | GPU Data

Occupancy Data:

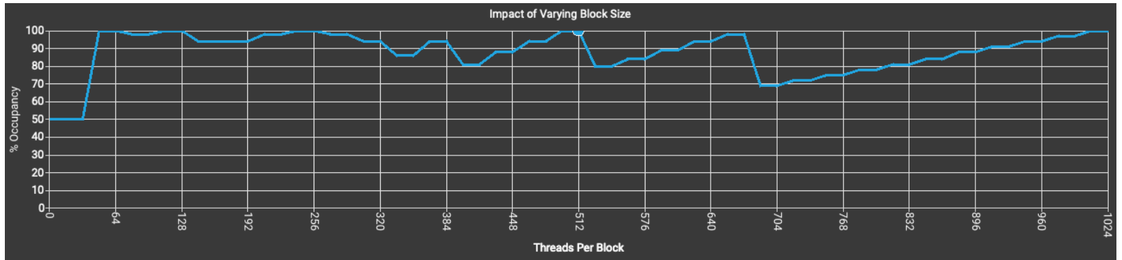
Property	Value
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	100 %

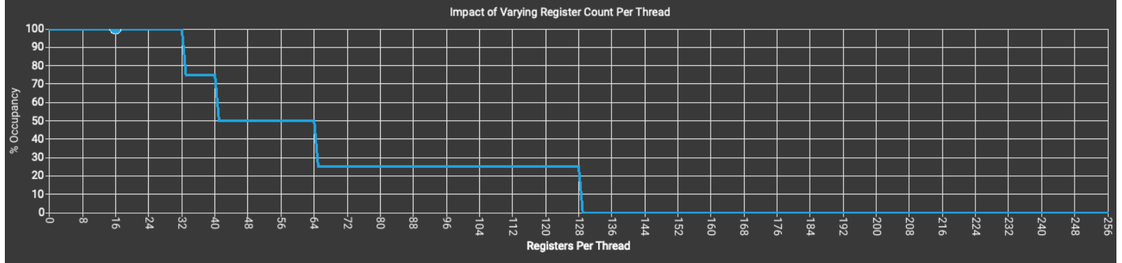
Allocated Resources:

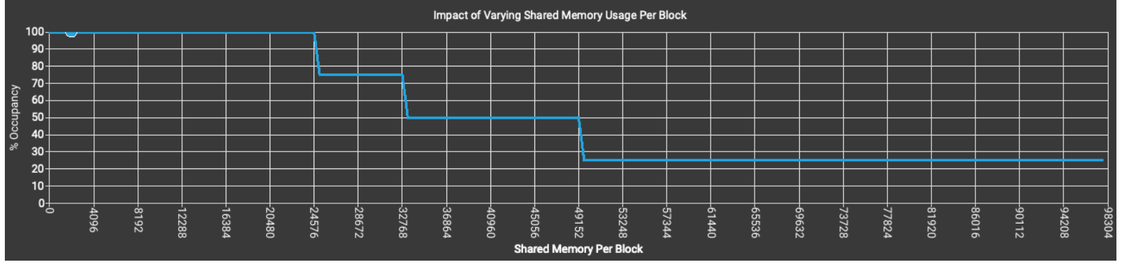
Resources	Per Block	Limit Per SM	Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	16	64	4
Registers (Warp limit per SM due to per-warp reg count)	16	128	8
Shared Memory (Bytes)	2048	98304	48

Occupancy Limiters:

Limited By	Blocks per SM	Warps Per Block	Warps Per SM
Max Warps or Max Blocks per Multiprocessor	4		
Registers per Multiprocessor	8		
Shared Memory per Multiprocessor	48		







Reduction #1: host code

- 32, 64, 128, 256, 1024 ?
- According to Occupancy Calculator: all good -- eg. 64,128, 256, 512, 1024
- # reg 16 (according to ncu) 14 reported by nvcc

Compute Capability:	7.0	Threads Per Bl	512
Shared Memory Size Config (bytes):	98304	Registers Per T	16
Global Load Cache Mode:	L1+L2 (ca)	User Shared Memory Per Block (bytes):	2048
			1

Apply Reset

BlkSize: 512

sizeof(float)xBlkSize = 4 * 512 = 2048

16 register/thread

Tables Utilization Graphs

Occupancy Data:

Property	Value
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	100 %

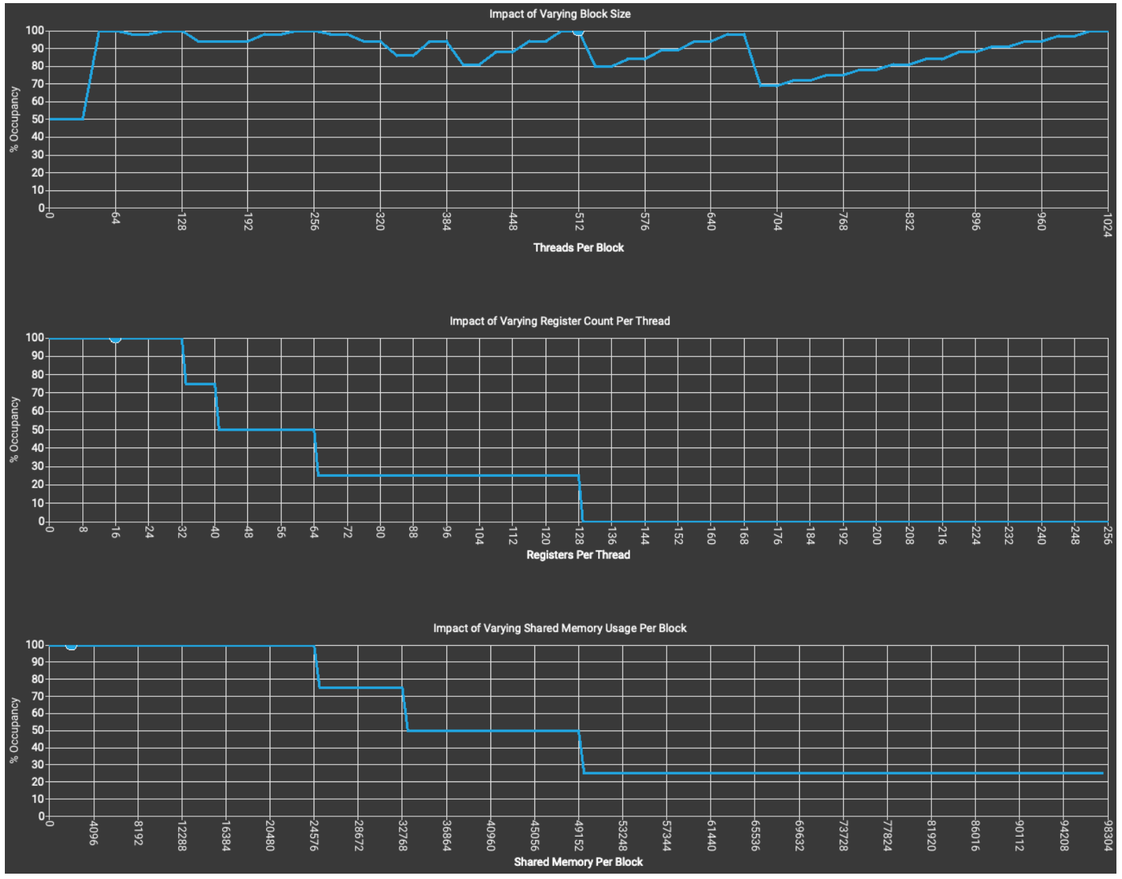
100% SM occupancy

Allocated Resources:

Resources	Per Block	Limit Per SM	Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	16	64	4
Registers (Warp limit per SM due to per-warp reg count)	16	128	8
Shared Memory (Bytes)	2048	98304	48

Occupancy Limiters:

Limited By	Blocks per SM	Warps Per Block	Warps Per SM
Max Warps or Max Blocks per Multiprocessor	4		
Registers per Multiprocessor	8		
Shared Memory per Multiprocessor	48		



Reduction #1: Performance I

On vector of ~ 1 Billion elements (4 GiB == 1 Gig 4 byte floats) on V100

BlkSize	Time(ms)		
32	54.05		
64	26.60		
128	20.58		
256	23.70		
512	27.65		
1024	34.50		

Reduction #1: Performance: Good enough

How do we know when performance is good enough?

Optimization goal: maximize GPU performance

Two key performance components:

1. Compute bandwidth – GFLOPs
2. Memory bandwidth – GB/s

Want to be as close to theoretical max on one or the other

Reduction #1: Performance: Good enough

How do we know when performance is good enough?

Optimization goal: maximize GPU performance

Two key performance components:

1. Compute bandwidth – GFLOPs
2. Memory bandwidth – GB/s

Want to be as close to theoretical max on one or the other

Reduction has low compute intensity

➔ memory bandwidth will be limiter

Reduction #1: Performance I

BlkSize	Time(ms)	TFLOPS	GB/s (GMEM)
32	54.05	0.02	84.59
64	26.60	0.04	166.56
128	20.58	0.05	212.01
256	23.70	0.05	182.67
512	27.65	0.04	155.94
1024	34.50	0.03	124.75

Bandwidth calc:

Start: N = 2^{30}

For each kernel/step:

- N reads and N/BlkSize writes
- input size reduced by BlkSize: ie. N=N/BlkSize

Convert to bytes and divide by time

For BlkSize = 512: [1,073,741,824 + 2,097,152 + 2,097,152 + 4,096 + 4,096 + 8 + 8 + 1] * 4

Reduction #1: Performance I

BlkSize	Time(ms)	TFLOPS	GB/s (GMEM)
32	54.05	0.02	84.59
64	26.60	0.04	166.56
128	20.58	0.05	212.01
256	23.70	0.05	182.67
512	27.65	0.04	155.94
1024	34.50	0.03	124.75

Far from theoretical max: 898 GB/s

Bandwidth calc:

Start: $N = 2^{30}$

For each kernel/step:

- N reads and N/BlkSize writes
- input size reduced by BlkSize: ie. N=N/BlkSize

Convert to bytes and divide by time

For BlkSize = 512: [1,073,741,824 + 2,097,152 + 2,097,152 + 4,096 + 4,096 + 8 + 8 + 1] * 4

Memory Bandwidth

https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#theoretical-bandwidth-calculation

https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#effective-bandwidth-calculation

Theoretical vs Effective

- Calculating theoretical GPU memory bandwidth (from specs of device)
  - V100 : HBM2 (double data rate) Memory : clk 877 MHz and 4096-bit-width memory interface
    - Bytes Transfered in a second / 10^9 = GB/s
    - (0.877 x 10^9 x (4096/8) x 2) / 10^9 = **898 GB/s**
- Effective Bandwidth Calculation
  - Giga Bytes transfered / time in seconds
    - ((Bytes Read + Bytes Written) / 10^9) / time in seconds
      - Eg. If memcopy of a 2048 x 2048 matrix of floats takes 43 microseconds (0.000043 seconds) then the effective Bandwidth is approximately: ((20482 x 4 x 2)/109)/0.000043 = 780.33 which is 780.33/898 = .87 of Theoretical

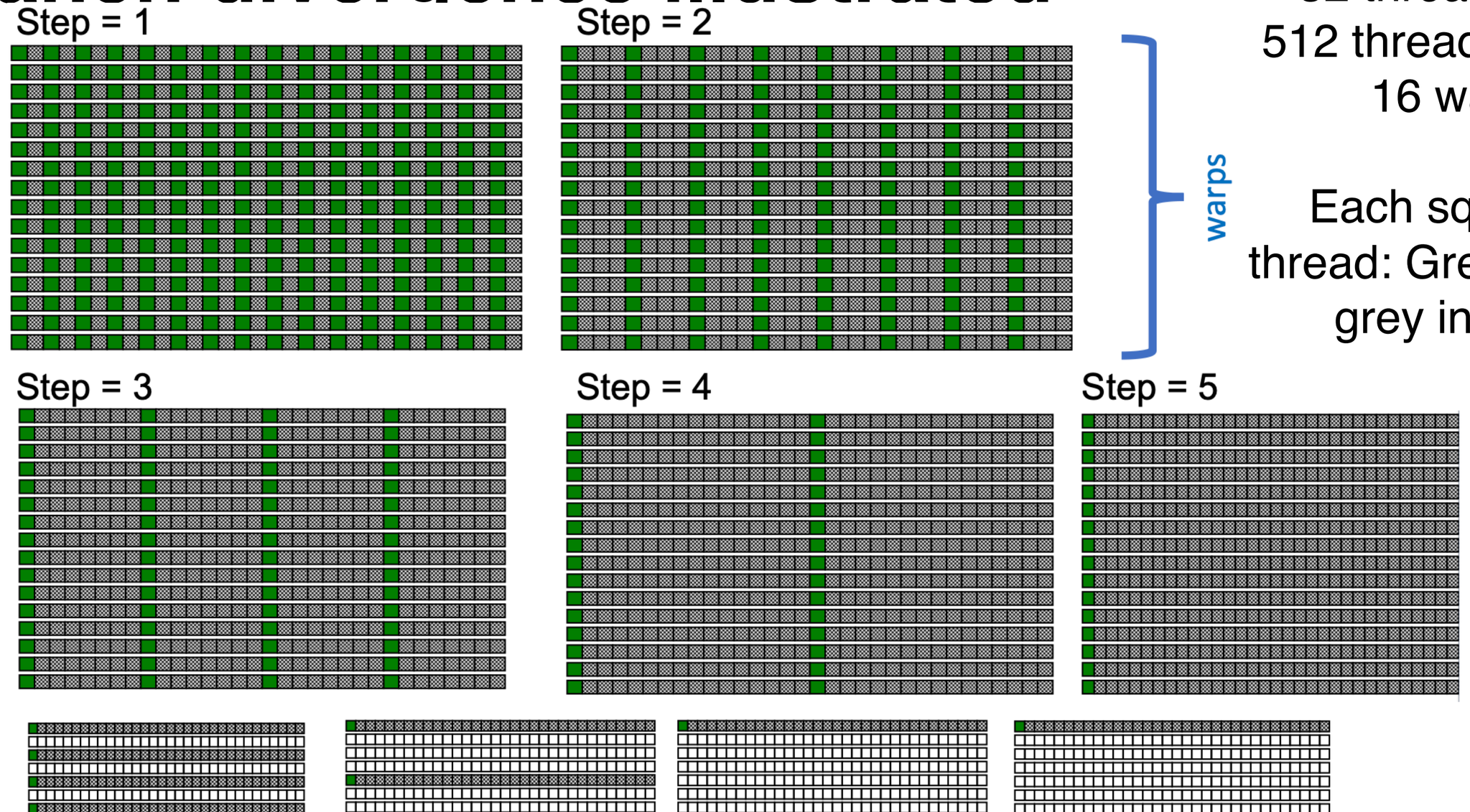
Harry Wood, Public domain, via Wikimedia Commons

Reduction #1 Problem: branch divergence

```
__global__ void reduce1(float *d_ivec, float *d_ovec, unsigned int n) {
  extern __shared__ float sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = (i<n) ? d_ivec[i] : 0.0;
  __syncthreads();

  for (unsigned int s=1; s<blockDim.x; s *= 2) {
    if ( (tid % (2*s)) == 0 ) {
      sdata[tid] += sdata[tid+s];
    }
    __syncthreads();
  }

  if (tid==0) d_ovec[blockIdx.x] = sdata[0];
}
```

Highly divergent code
Negatively affects performance

Branch divergence illustrated



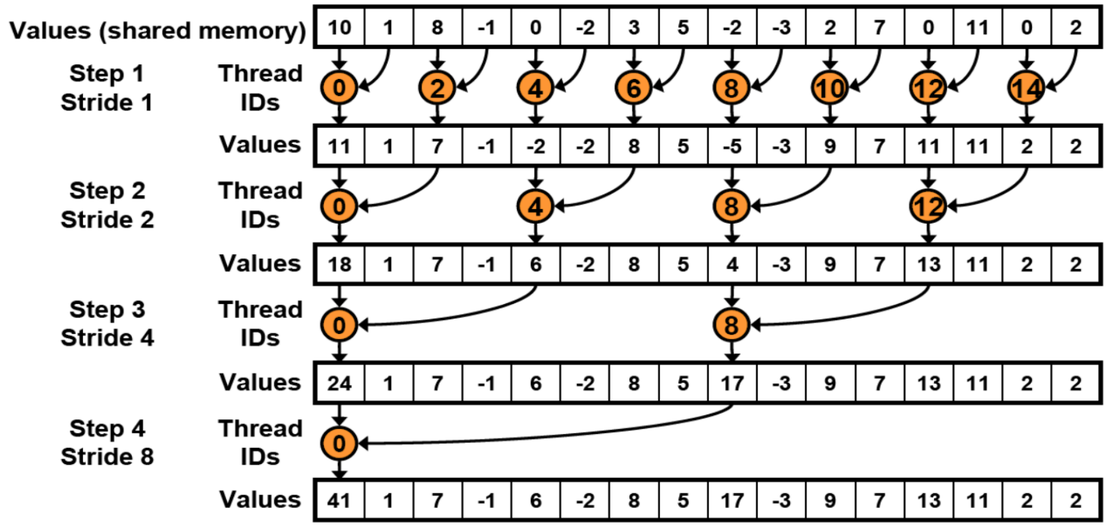
32 threads/warp
512 threads/block = 16 warps

Each square: a thread: Green active, grey inactive

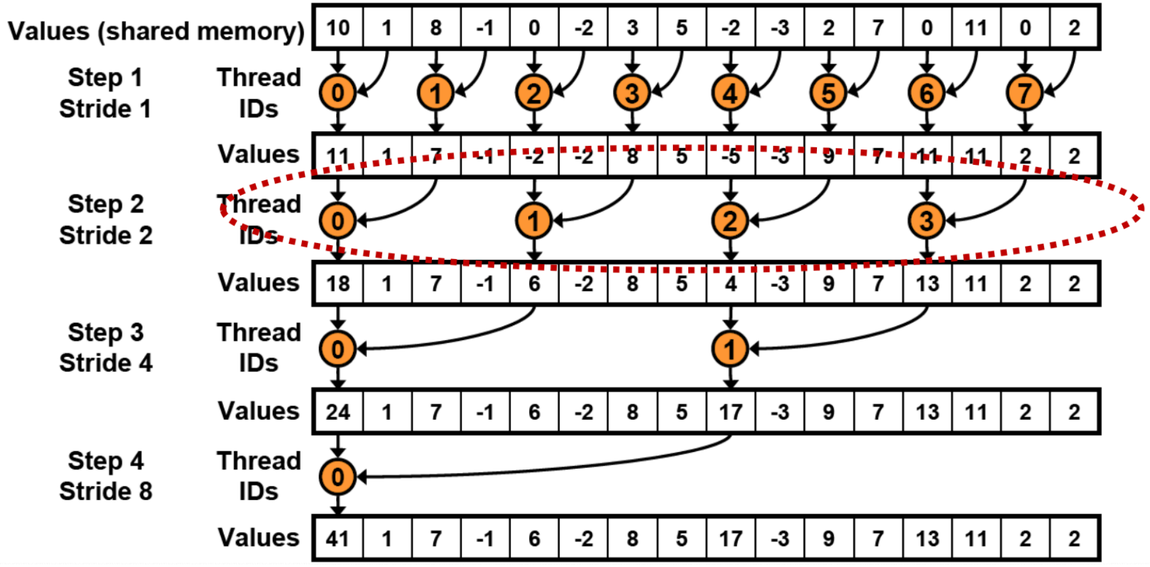
Branch divergence fix

Idea: have the the threads with the lowest tids in each block do the work:

Before:



After:



Reduction #2

Repace the divergent branching code:

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

With code using strided index and non-divergent branch:

```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  int index = 2 * s * tid;

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

Reduction #2

Repace the divergent branching code:

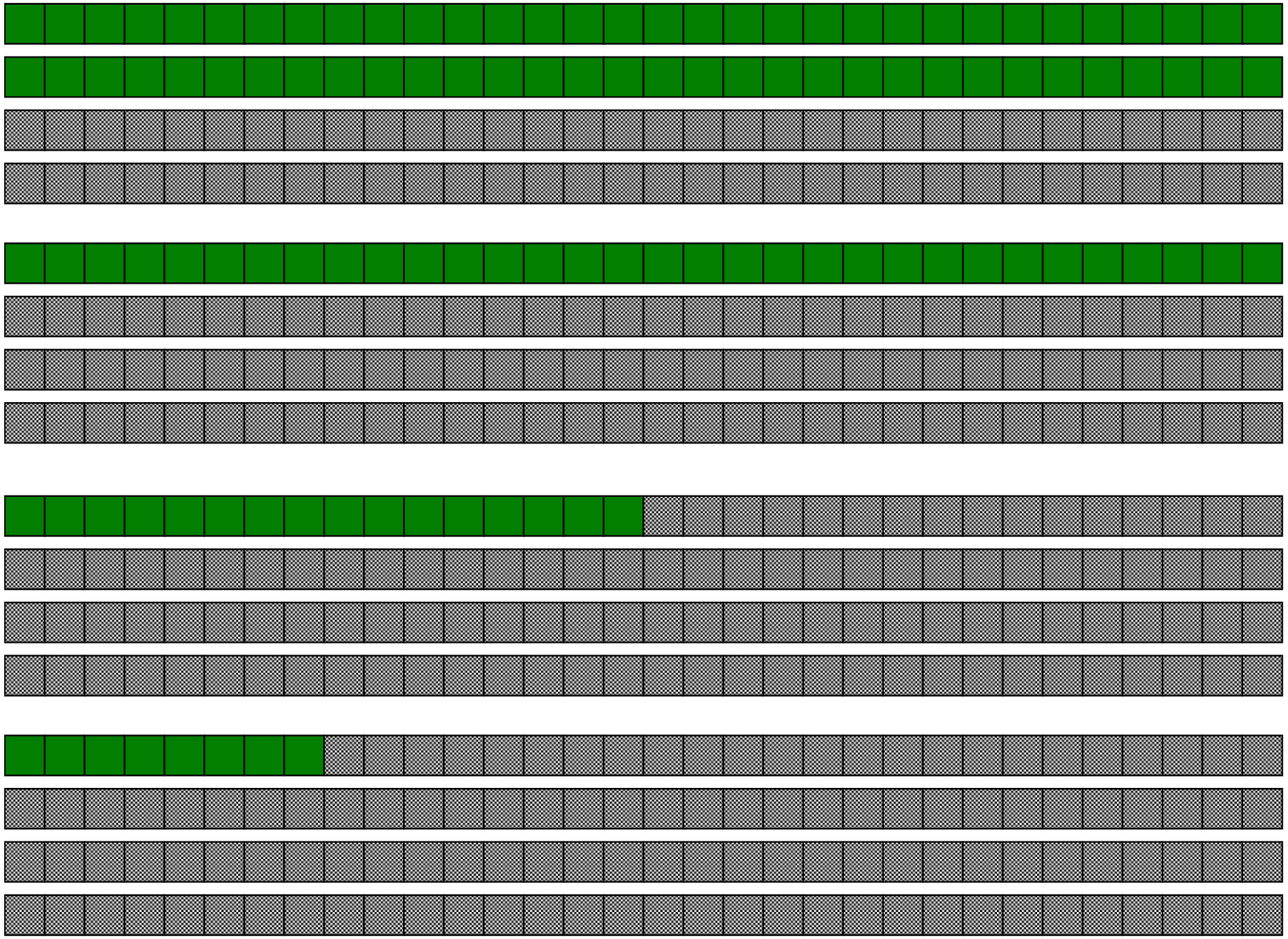
```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  if ( (tid % (2*s)) == 0 ) {
    sdata[tid] += sdata[tid+s];
  }
  __syncthreads();
}
```

With code using strided index and non-divergent branch:

```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  int index = 2 * s * tid;

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

Reduction #2: warp behavior

This is a block with 128 threads – shown as an example

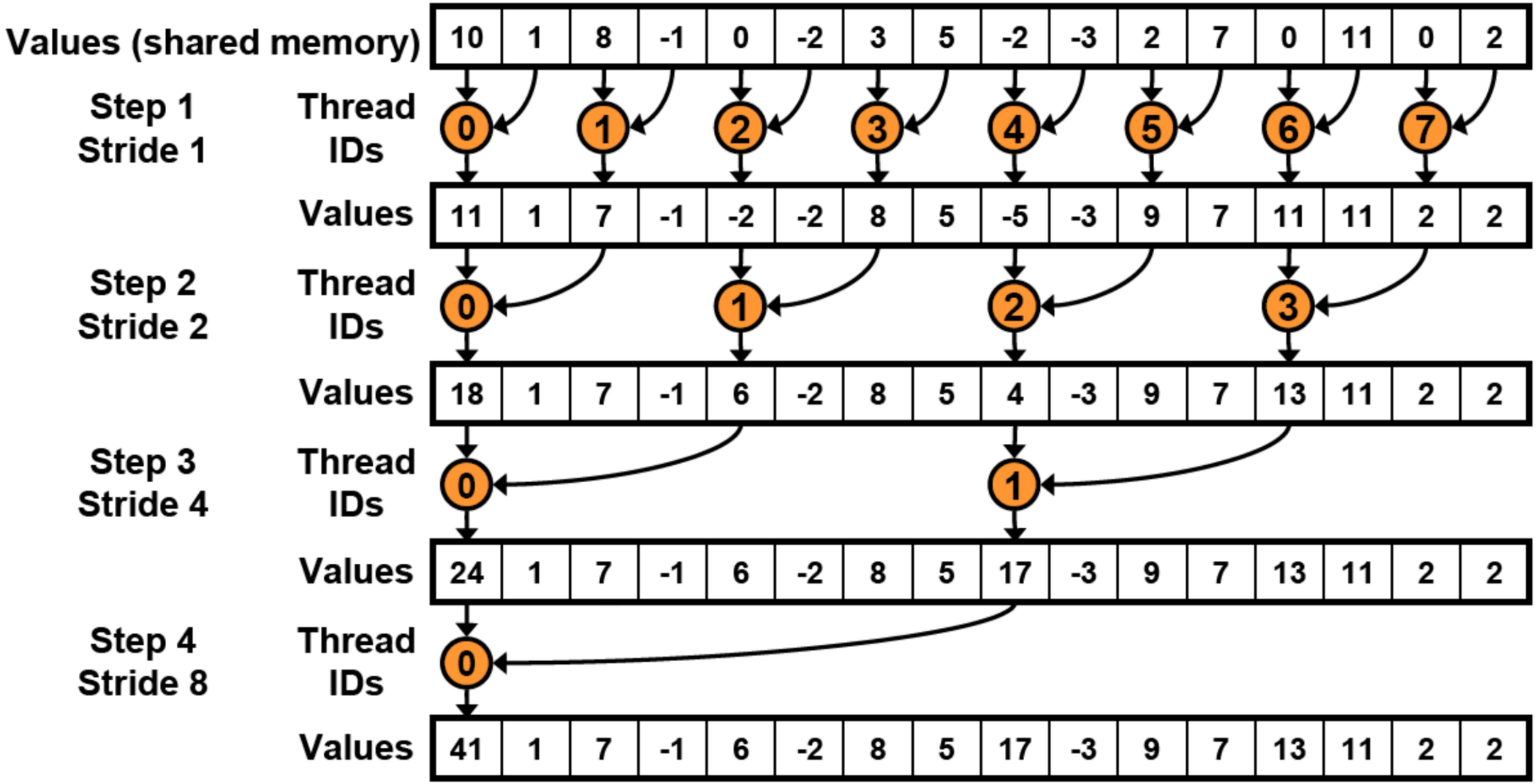


4 warps per block

Reduction #2: Performance

	Red #1		Red #2		Sp'up
BSz	ms	GB/s	ms	GB/s	
32	54.05	84.59	54.06	84.57	0.99
64	26.60	166.56	26.61	166.51	0.99
128	20.58	212.01	**14.50**	300.92	**1.42**
256	23.70	182.67	15.04	287.91	**1.58**
512	27.65	155.94	17.05	252.96	**1.62**
1024	34.50	124.75	20.73	207.64	**1.66**

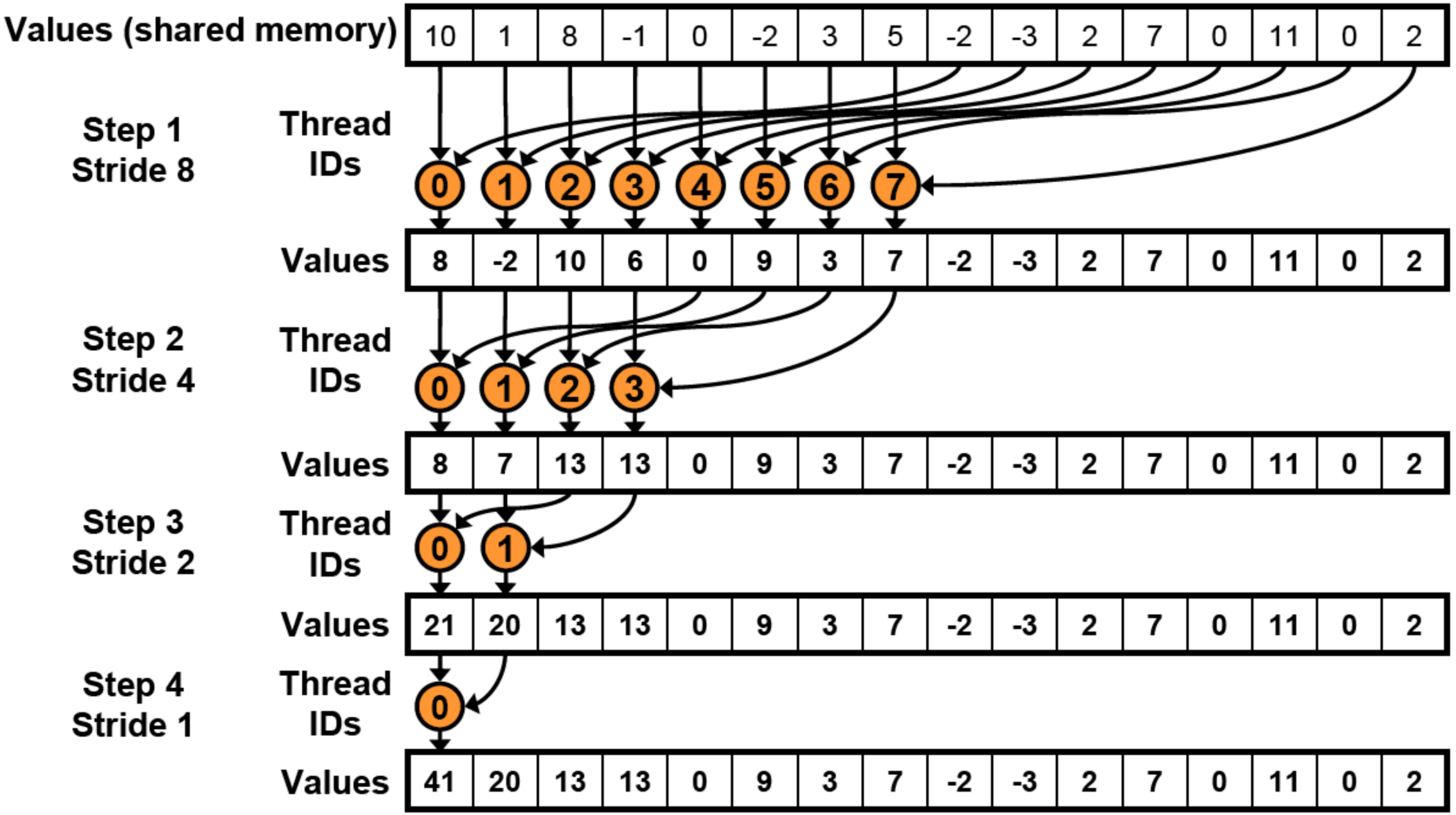
Reduction #2 problem: SMEM bank conflicts



2-way bank conflicts at every step

To see the conflicts consider the SMEM accesses by a single warp

Fix: coalesced accesses to SMEM



Reduction #3

Repace the stride indexing in the inner loop:

```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  int index = 2 * s * tid;

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

With reversed loop and tid-based indexing:

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

Reduction #3: Performance

	Red #1		Red #2		Red #3		Sp'up	Cum Sp'up
BSz	ms	GB/s	ms	GB/s	ms	GB/s		
32	54.05	84.59	54.06	84.57	54.06	84.58	1.00	0.99
64	26.60	166.56	26.61	166.51	26.61	166.53	1.00	0.99
128	20.58	212.01	14.50	300.92	13.21	330.20	**1.10**	**1.57**
256	23.70	182.67	15.04	287.91	**11.32**	382.45	**1.32**	**2.09**
512	27.65	155.94	17.05	252.96	12.96	332.61	**1.31**	**2.13**
1024	34.50	124.75	20.73	207.64	16.46	261.41	**1.26**	**2.10**

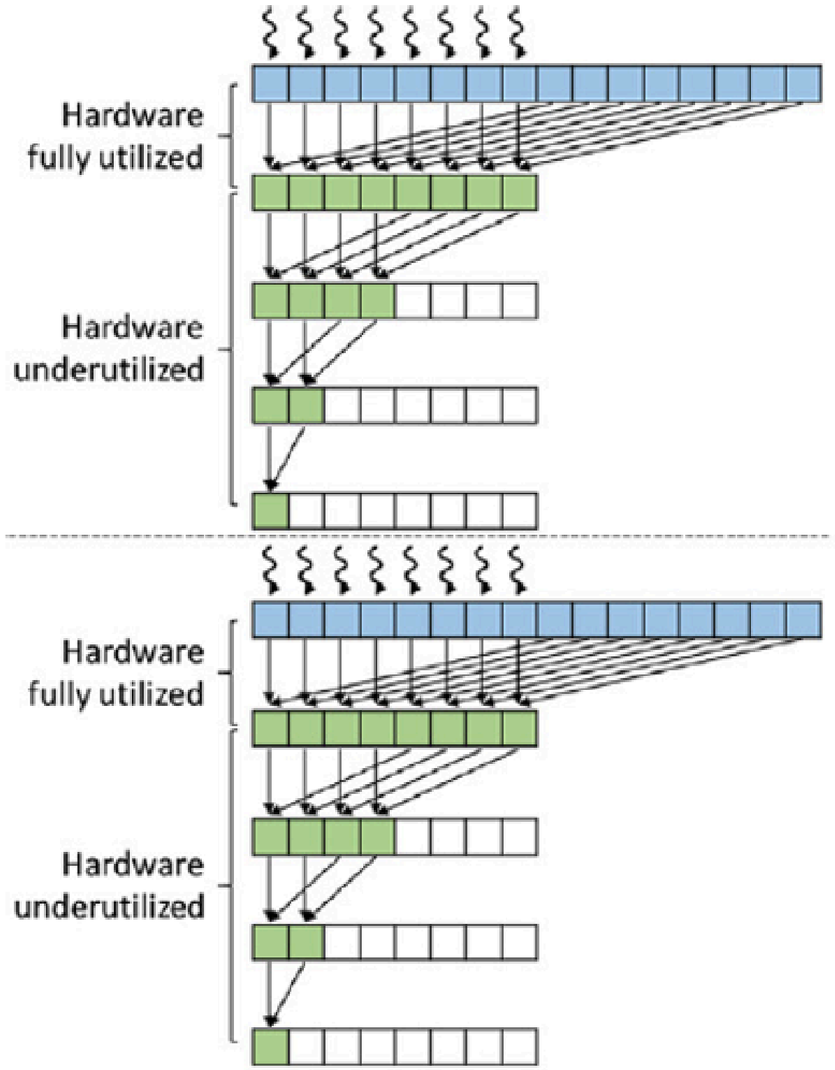
Reduction #2: Resource Efficiency

- All threads read 1 element from GMEM & write to SMEM
- Then:
  - First step: read SMEM back while half of the threads are idle
  - Next step: another half becomes idle
  - . . .

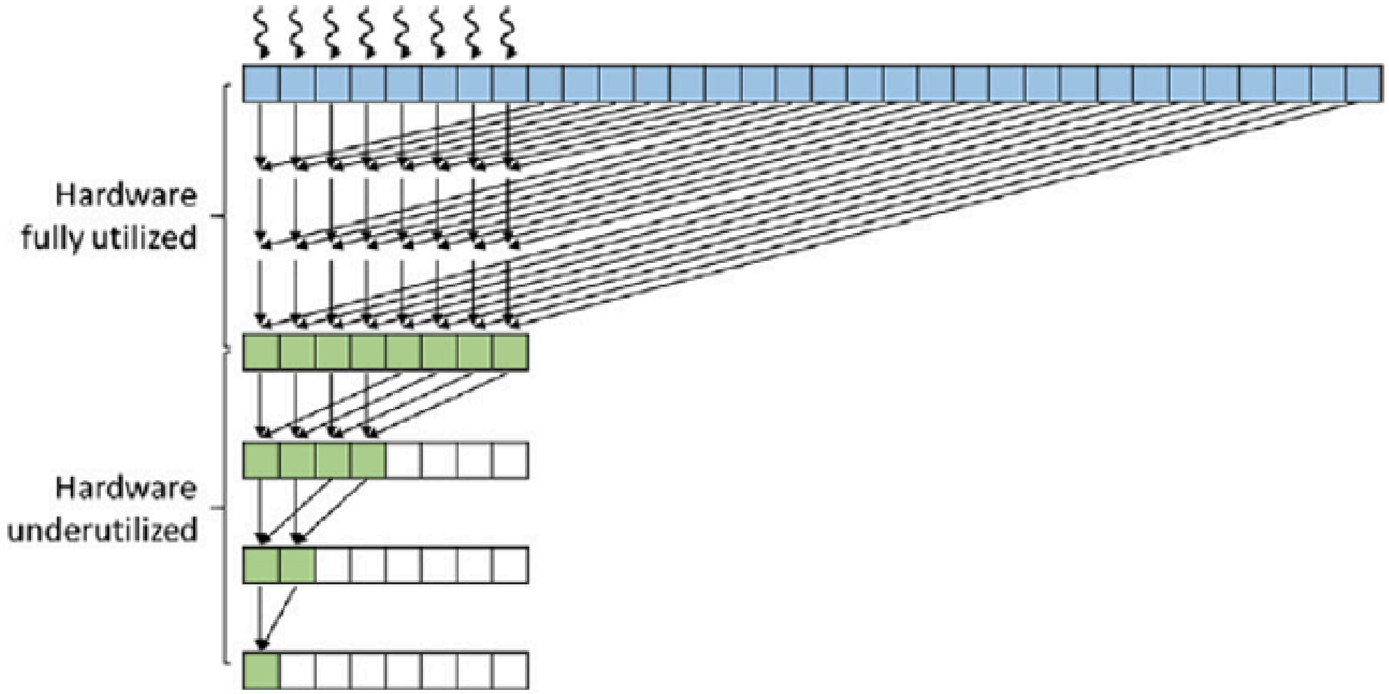
Can we be more efficient?

- Idea: do first reduction step when reading el. from GMEM

Reduction #2: Resource Efficiency



(A) Execution of two original thread blocks serialized by the hardware



(B) Execution of one coarsened thread block doing the work of two original thread blocks

Reduction #4

Original each thread reads one element

```
extern __shared__ float sdata[];
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;

sdata[tid] = (i<n) ? d_ivec[i] : 0;
__syncthreads();
```

Now read two and do the first reduction step

```
extern __shared__ float sdata[];
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x *
                   (blockDim.x*2)   // note * 2
                   + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0);
__syncthreads();
```

Reduction #4: Performance

	Red #3		Red #4		Sp'up	Cum Sp'up
BSz	ms	GB/s	ms	GB/s		
32	54.06	84.58	26.61	166.56	**2.03**	**2.03**
64	26.61	166.53	13.21	330.19	**2.01**	**2.01**
128	13.21	330.20	6.59	656.86	**2.00**	**3.12**
256	11.32	382.45	**6.30**	684.03	**1.80**	**3.75**
512	12.96	332.61	7.11	605.13	**1.82**	**3.88**
1024	16.46	261.41	8.77	490.22	**1.88**	**3.93**

Reduction #5-9: Performance

That worked so well -- lets increase the initial reductional reduction a few times and see what happens -- remember no need to synchronize: 4, 8, 16, 32, 64

reduce5: Now read four and do the first reduction step

```
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x * (blockDim.x*4) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0);
__syncthreads();
```

Reduction #5-9: Performance

reduce5: Now read four and do the first reduction step

```
unsigned int tid = threadIdx.x;
unsigned int i   = blockIdx.x * (blockDim.x*4) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0);
__syncthreads();
```

reduce6: Now read eight and do the first reduction step

```
unsigned int i   = blockIdx.x * (blockDim.x*8) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0) +

             ((i+(4*blockDim.x) < n) ? d_ivec[i+(4*blockDim.x)] : 0.0) +
             ((i+(5*blockDim.x) < n) ? d_ivec[i+(5*blockDim.x)] : 0.0) +
             ((i+(6*blockDim.x) < n) ? d_ivec[i+(6*blockDim.x)] : 0.0) +
             ((i+(7*blockDim.x) < n) ? d_ivec[i+(7*blockDim.x)] : 0.0);
__syncthreads();
```

Reduction #5-9: Performance

```
unsigned int i   = blockIdx.x * (blockDim.x*16) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0) +
             ((i+(4*blockDim.x) < n) ? d_ivec[i+(4*blockDim.x)] : 0.0) +
             ((i+(5*blockDim.x) < n) ? d_ivec[i+(5*blockDim.x)] : 0.0) +
             ((i+(6*blockDim.x) < n) ? d_ivec[i+(6*blockDim.x)] : 0.0) +
             ((i+(7*blockDim.x) < n) ? d_ivec[i+(7*blockDim.x)] : 0.0) +
             ((i+(8*blockDim.x) < n) ? d_ivec[i+(8*blockDim.x)] : 0.0) +
             ((i+(9*blockDim.x) < n) ? d_ivec[i+(9*blockDim.x)] : 0.0) +
             ((i+(10*blockDim.x) < n) ? d_ivec[i+(10*blockDim.x)] : 0.0) +
             ((i+(11*blockDim.x) < n) ? d_ivec[i+(11*blockDim.x)] : 0.0) +
             ((i+(12*blockDim.x) < n) ? d_ivec[i+(12*blockDim.x)] : 0.0) +
             ((i+(13*blockDim.x) < n) ? d_ivec[i+(13*blockDim.x)] : 0.0) +
             ((i+(14*blockDim.x) < n) ? d_ivec[i+(14*blockDim.x)] : 0.0) +
             ((i+(15*blockDim.x) < n) ? d_ivec[i+(15*blockDim.x)] : 0.0);
__syncthreads();
```

reduce7: Now read 16 and do the first reduction step

reduce8: Now read 32 and do the first reduction step

```
unsigned int i   = blockIdx.x * (blockDim.x*32) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0) +
             ((i+(4*blockDim.x) < n) ? d_ivec[i+(4*blockDim.x)] : 0.0) +
             ((i+(5*blockDim.x) < n) ? d_ivec[i+(5*blockDim.x)] : 0.0) +
             ((i+(6*blockDim.x) < n) ? d_ivec[i+(6*blockDim.x)] : 0.0) +
             ((i+(7*blockDim.x) < n) ? d_ivec[i+(7*blockDim.x)] : 0.0) +
             ((i+(8*blockDim.x) < n) ? d_ivec[i+(8*blockDim.x)] : 0.0) +
             ((i+(9*blockDim.x) < n) ? d_ivec[i+(9*blockDim.x)] : 0.0) +
             ((i+(10*blockDim.x) < n) ? d_ivec[i+(10*blockDim.x)] : 0.0) +
             ((i+(11*blockDim.x) < n) ? d_ivec[i+(11*blockDim.x)] : 0.0) +
             ((i+(12*blockDim.x) < n) ? d_ivec[i+(12*blockDim.x)] : 0.0) +
             ((i+(13*blockDim.x) < n) ? d_ivec[i+(13*blockDim.x)] : 0.0) +
             ((i+(14*blockDim.x) < n) ? d_ivec[i+(14*blockDim.x)] : 0.0) +
             ((i+(15*blockDim.x) < n) ? d_ivec[i+(15*blockDim.x)] : 0.0) +
             ((i+(16*blockDim.x) < n) ? d_ivec[i+(16*blockDim.x)] : 0.0) +
             ((i+(17*blockDim.x) < n) ? d_ivec[i+(17*blockDim.x)] : 0.0) +
             ((i+(18*blockDim.x) < n) ? d_ivec[i+(18*blockDim.x)] : 0.0) +
             ((i+(19*blockDim.x) < n) ? d_ivec[i+(19*blockDim.x)] : 0.0) +
             ((i+(20*blockDim.x) < n) ? d_ivec[i+(20*blockDim.x)] : 0.0) +
             ((i+(21*blockDim.x) < n) ? d_ivec[i+(21*blockDim.x)] : 0.0) +
             ((i+(22*blockDim.x) < n) ? d_ivec[i+(22*blockDim.x)] : 0.0) +
             ((i+(23*blockDim.x) < n) ? d_ivec[i+(23*blockDim.x)] : 0.0) +
             ((i+(24*blockDim.x) < n) ? d_ivec[i+(24*blockDim.x)] : 0.0) +
             ((i+(25*blockDim.x) < n) ? d_ivec[i+(25*blockDim.x)] : 0.0) +
             ((i+(26*blockDim.x) < n) ? d_ivec[i+(26*blockDim.x)] : 0.0) +
             ((i+(27*blockDim.x) < n) ? d_ivec[i+(27*blockDim.x)] : 0.0) +
             ((i+(28*blockDim.x) < n) ? d_ivec[i+(28*blockDim.x)] : 0.0) +
             ((i+(29*blockDim.x) < n) ? d_ivec[i+(29*blockDim.x)] : 0.0) +
             ((i+(30*blockDim.x) < n) ? d_ivec[i+(30*blockDim.x)] : 0.0) +
             ((i+(31*blockDim.x) < n) ? d_ivec[i+(31*blockDim.x)] : 0.0);
__syncthreads();
```

Reduction #5-9: Performance

reduce9: Now read 64 and do the first reduction step
Of course this code could optimized to be a loop with compiler unrolling and a single check hoisted out of the loop.

if divisible by reduction factor then skip individual checks

```
unsigned int i   = blockIdx.x * (blockDim.x*64) + threadIdx.x;

sdata[tid] = ((i<n) ? d_ivec[i] : 0.0) +
             ((i+blockDim.x < n) ? d_ivec[i+blockDim.x] : 0.0) +
             ((i+(2*blockDim.x) < n) ? d_ivec[i+(2*blockDim.x)] : 0.0) +
             ((i+(3*blockDim.x) < n) ? d_ivec[i+(3*blockDim.x)] : 0.0) +
             ((i+(4*blockDim.x) < n) ? d_ivec[i+(4*blockDim.x)] : 0.0) +
             ((i+(5*blockDim.x) < n) ? d_ivec[i+(5*blockDim.x)] : 0.0) +
             ((i+(6*blockDim.x) < n) ? d_ivec[i+(6*blockDim.x)] : 0.0) +
             ((i+(7*blockDim.x) < n) ? d_ivec[i+(7*blockDim.x)] : 0.0) +
             ((i+(8*blockDim.x) < n) ? d_ivec[i+(8*blockDim.x)] : 0.0) +
             ((i+(9*blockDim.x) < n) ? d_ivec[i+(9*blockDim.x)] : 0.0) +
             ((i+(10*blockDim.x) < n) ? d_ivec[i+(10*blockDim.x)] : 0.0) +
             ((i+(11*blockDim.x) < n) ? d_ivec[i+(11*blockDim.x)] : 0.0) +
             ((i+(12*blockDim.x) < n) ? d_ivec[i+(12*blockDim.x)] : 0.0) +
             ((i+(13*blockDim.x) < n) ? d_ivec[i+(13*blockDim.x)] : 0.0) +
             ((i+(14*blockDim.x) < n) ? d_ivec[i+(14*blockDim.x)] : 0.0) +
             ((i+(15*blockDim.x) < n) ? d_ivec[i+(15*blockDim.x)] : 0.0) +
             ((i+(16*blockDim.x) < n) ? d_ivec[i+(16*blockDim.x)] : 0.0) +
             ((i+(17*blockDim.x) < n) ? d_ivec[i+(17*blockDim.x)] : 0.0) +
             ((i+(18*blockDim.x) < n) ? d_ivec[i+(18*blockDim.x)] : 0.0) +
             ((i+(19*blockDim.x) < n) ? d_ivec[i+(19*blockDim.x)] : 0.0) +
             ((i+(20*blockDim.x) < n) ? d_ivec[i+(20*blockDim.x)] : 0.0) +
             ((i+(21*blockDim.x) < n) ? d_ivec[i+(21*blockDim.x)] : 0.0) +
             ((i+(22*blockDim.x) < n) ? d_ivec[i+(22*blockDim.x)] : 0.0) +
             ((i+(23*blockDim.x) < n) ? d_ivec[i+(23*blockDim.x)] : 0.0) +
             ((i+(24*blockDim.x) < n) ? d_ivec[i+(24*blockDim.x)] : 0.0) +
             ((i+(25*blockDim.x) < n) ? d_ivec[i+(25*blockDim.x)] : 0.0) +
             ((i+(26*blockDim.x) < n) ? d_ivec[i+(26*blockDim.x)] : 0.0) +
             ((i+(27*blockDim.x) < n) ? d_ivec[i+(27*blockDim.x)] : 0.0) +
             ((i+(28*blockDim.x) < n) ? d_ivec[i+(28*blockDim.x)] : 0.0) +
             ((i+(29*blockDim.x) < n) ? d_ivec[i+(29*blockDim.x)] : 0.0) +
             ((i+(30*blockDim.x) < n) ? d_ivec[i+(30*blockDim.x)] : 0.0) +
             ((i+(31*blockDim.x) < n) ? d_ivec[i+(31*blockDim.x)] : 0.0) +
             ((i+(32*blockDim.x) < n) ? d_ivec[i+(32*blockDim.x)] : 0.0) +
             ((i+(33*blockDim.x) < n) ? d_ivec[i+(33*blockDim.x)] : 0.0) +
             ((i+(34*blockDim.x) < n) ? d_ivec[i+(34*blockDim.x)] : 0.0) +
             ((i+(35*blockDim.x) < n) ? d_ivec[i+(35*blockDim.x)] : 0.0) +
             ((i+(36*blockDim.x) < n) ? d_ivec[i+(36*blockDim.x)] : 0.0) +
             ((i+(37*blockDim.x) < n) ? d_ivec[i+(37*blockDim.x)] : 0.0) +
             ((i+(38*blockDim.x) < n) ? d_ivec[i+(38*blockDim.x)] : 0.0) +
             ((i+(39*blockDim.x) < n) ? d_ivec[i+(39*blockDim.x)] : 0.0) +
             ((i+(40*blockDim.x) < n) ? d_ivec[i+(40*blockDim.x)] : 0.0) +
             ((i+(41*blockDim.x) < n) ? d_ivec[i+(41*blockDim.x)] : 0.0) +
             ((i+(42*blockDim.x) < n) ? d_ivec[i+(42*blockDim.x)] : 0.0) +
             ((i+(43*blockDim.x) < n) ? d_ivec[i+(43*blockDim.x)] : 0.0) +
             ((i+(44*blockDim.x) < n) ? d_ivec[i+(44*blockDim.x)] : 0.0) +
             ((i+(45*blockDim.x) < n) ? d_ivec[i+(45*blockDim.x)] : 0.0) +
             ((i+(46*blockDim.x) < n) ? d_ivec[i+(46*blockDim.x)] : 0.0) +
             ((i+(47*blockDim.x) < n) ? d_ivec[i+(47*blockDim.x)] : 0.0) +
             ((i+(48*blockDim.x) < n) ? d_ivec[i+(48*blockDim.x)] : 0.0) +
             ((i+(49*blockDim.x) < n) ? d_ivec[i+(49*blockDim.x)] : 0.0) +
             ((i+(50*blockDim.x) < n) ? d_ivec[i+(50*blockDim.x)] : 0.0) +
             ((i+(51*blockDim.x) < n) ? d_ivec[i+(51*blockDim.x)] : 0.0) +
             ((i+(52*blockDim.x) < n) ? d_ivec[i+(52*blockDim.x)] : 0.0) +
             ((i+(53*blockDim.x) < n) ? d_ivec[i+(53*blockDim.x)] : 0.0) +
             ((i+(54*blockDim.x) < n) ? d_ivec[i+(54*blockDim.x)] : 0.0) +
             ((i+(55*blockDim.x) < n) ? d_ivec[i+(55*blockDim.x)] : 0.0) +
             ((i+(56*blockDim.x) < n) ? d_ivec[i+(56*blockDim.x)] : 0.0) +
             ((i+(57*blockDim.x) < n) ? d_ivec[i+(57*blockDim.x)] : 0.0) +
             ((i+(58*blockDim.x) < n) ? d_ivec[i+(58*blockDim.x)] : 0.0) +
             ((i+(59*blockDim.x) < n) ? d_ivec[i+(59*blockDim.x)] : 0.0) +
             ((i+(60*blockDim.x) < n) ? d_ivec[i+(60*blockDim.x)] : 0.0) +
             ((i+(61*blockDim.x) < n) ? d_ivec[i+(61*blockDim.x)] : 0.0) +
             ((i+(62*blockDim.x) < n) ? d_ivec[i+(62*blockDim.x)] : 0.0) +
             ((i+(63*blockDim.x) < n) ? d_ivec[i+(63*blockDim.x)] : 0.0);
__syncthreads();
```

Reduction #4: Performance

	Red #5 (4)		Red #6 (8)		Red #7 (16)		Red #8 (32)		Red #9 (64)	
BSz	ms	GB/s	ms	GB/s	ms	GB/s	ms	GB/s		
32	13.21	330.33	-	-	5.03	857.25	4.96	867.67	-	-
64	6.60	656.33	5.03	856.16	5.00	864.60	-	-	4.93	871.31
128	4.97	867.23	4.96	867.15	-	-	4.91	874.29	4.92	872.82
256	4.94	870.91	4.92	873.42	4.91	875.47	4.91	875.18	4.93	872.02
512	5.00	859.36	4.91	875.77	**4.90**	876.47	4.93	872.14	4.99	860.51
1024	5.46	786.22	**4.90**	876.63	4.91	873.41	4.92	872.98	-	

best reduce 4: BSz: 256 ms: 6.30 GB/s 684.03

4.90ms is **1.29** x better

best reduce 1: BSz: 128 ms: 20.58 GB/s 212.01

4.90ms is **4.20** x better than where we started

Where to go from here

Still lots to explore but...

- If our BW approximations are right we have done quite well without -- too much effort ... but the code is brittle and needs lots of work to make robust (validate measures, confirm correctness)
- But there is room and lots to explore
  - Loop unrolling to reduce overheads and improve ILP
    - This could help quite a bit but can get ugly quickly. C++ templates and macros can help
  - try optimizing warp reduce with __reduce_add_sync
  - try thread group api to construct cross block barriers and avoid multiple kernel invocations
    - may allow intermediate results to stay in shared memory and registers
    - beware impact on block scheduling
  - try using vector instructions and tensor accelerator
  - try async global to local memory transfers
- Use profiles to confirm and why optimizations are working and guide next steps

What we did

Optimizations applied

- coalesced memory accesses to get data into SMEM
  - preserved this when we added later optimization
- reduced thread divergence
- reduced SMEM bank conflicts
- improved utilization by having threads move more data from GMEM (thread coarsening)
  - tried several versions