GPU Programming
III – Optimizations

Stefan BORNHOFEN

Optimization

A good understanding of low-level GPU mechanisms will optimize performance.

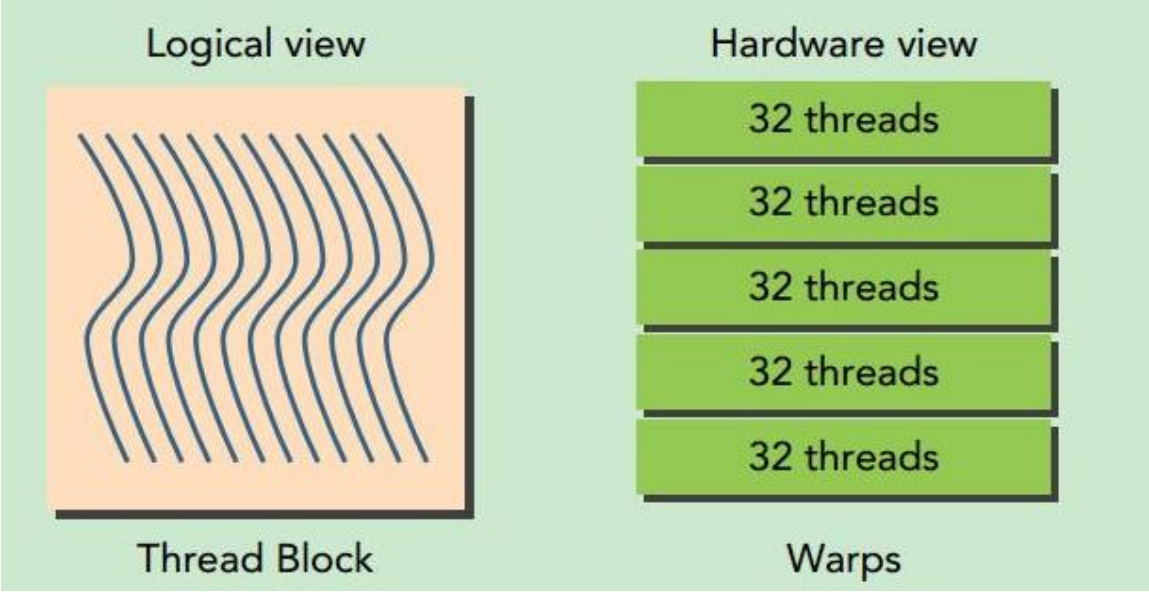
- Warps
- Latency hiding
- Divergent branching
- Memory coalescing
- Bank conflicts

- Use peak performance benchmarks to guide optimization.

Warps

Thread blocks are partitioned into warps comprised of 32 threads each. This allows for

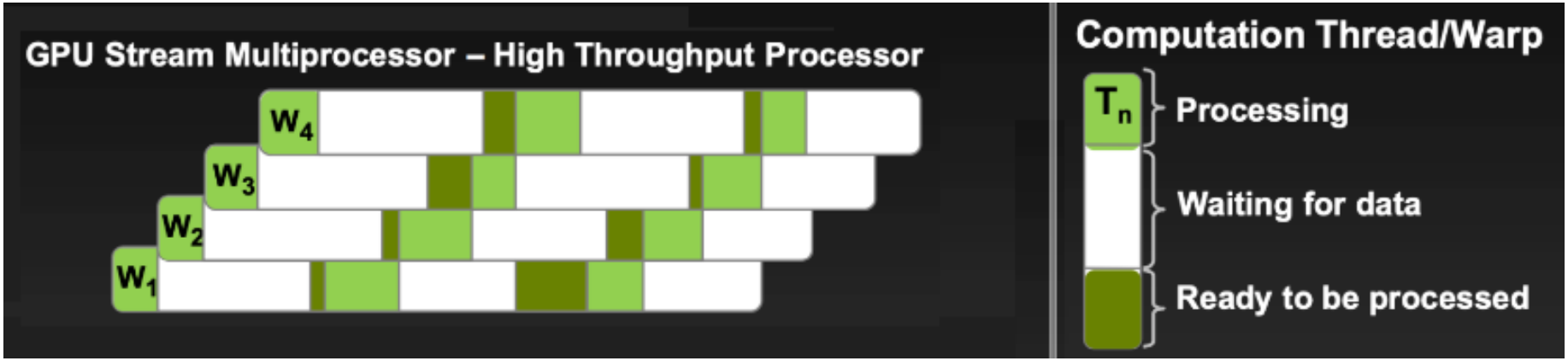
- the simplification of thread management
- the sharing of data and resources among threads
- the masking of memory latency with effective scheduling.



Latency hiding

Latency is the number of clock cycles for a warp to finish executing an instruction and become available to process the next one.

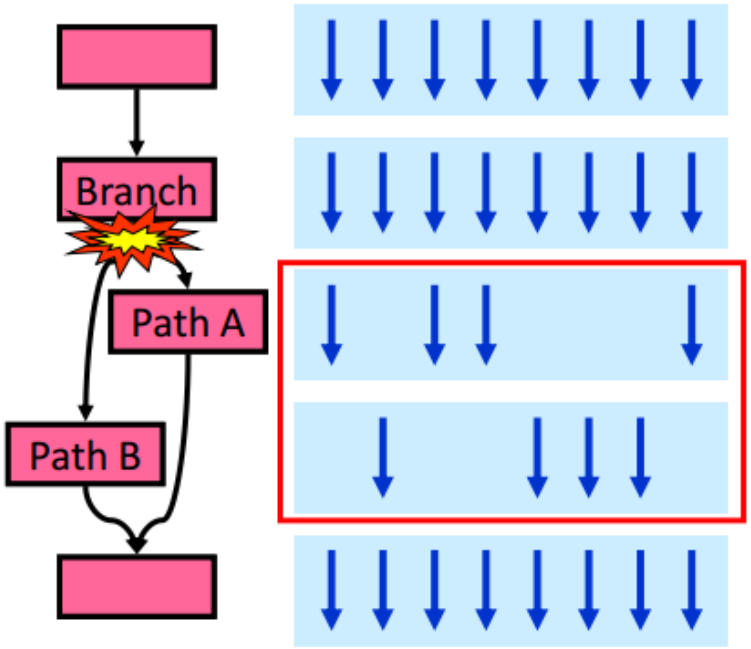
The streaming multiprocessors switch between warps with no apparent overhead. Effective warp scheduling hides latency and maximizes throughput.



Branching within warps

If a branch occurs only one part of the warp is executed.

Always look out for having 32 consecutive threads take the same path, or you will degrade the overall performance of your GPU code.



Exercise

Time two kernels with a branching concerning

- every other warp
- every other thread

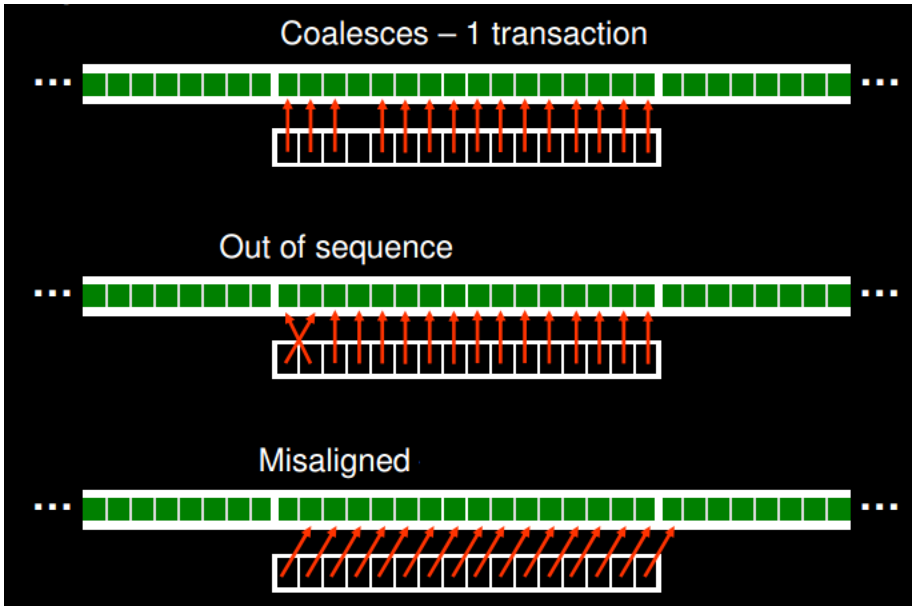
```
__global__ void warp_ok(...) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if ((threadIdx.x/32)%2==0) { ... }
      else { ... }
}

__global__ void warp_ko(...) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (threadIdx.x%2==0) { ... }
      else { ... }
}
```

Memory Coalescing

The simultaneous global memory accesses by each thread of a warp during the execution of a single read or write instruction will be coalesced into a single access if:

- The address of the first element is aligned to 16 times the element's size
- The size of the memory elements are 4, 8, or 16 bytes
- The elements form a contiguous block of memory
- The Nth element is accessed by the Nth thread in the half-warp

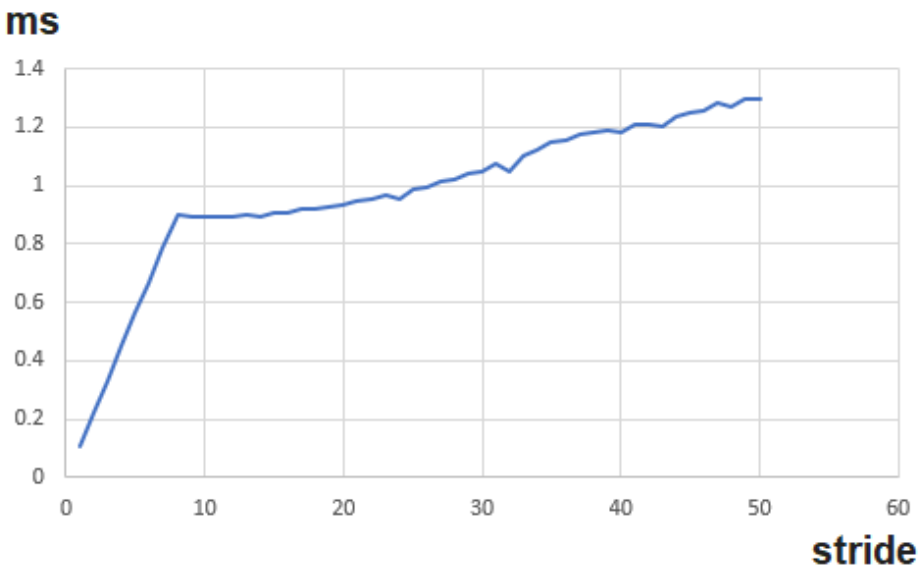
Compute capability 1.2 and higher hardware can coalesce accesses that fall into aligned 128-byte segments.



Exercise

Time a kernel accessing global memory with different strides.

```
__global__ void coalesce
(float* d_idata, float* d_odata, int datasize, int stride) {
	int i = blockIdx.x*blockDim.x + threadIdx.x;
	int j = (stride*i) % datasize;
	d_odata[j] = d_idata[j];
}
```



Bank conflicts

Shared memory is divided into banks for simultaneous accesses.

The number of banks is

- 16 on older GPUs (compute capability 1.x)
- 32 on modern GPUs (compute capability 2.x).

Successive 32-bit words assigned to successive banks.

Each bank can service one address per cycle.

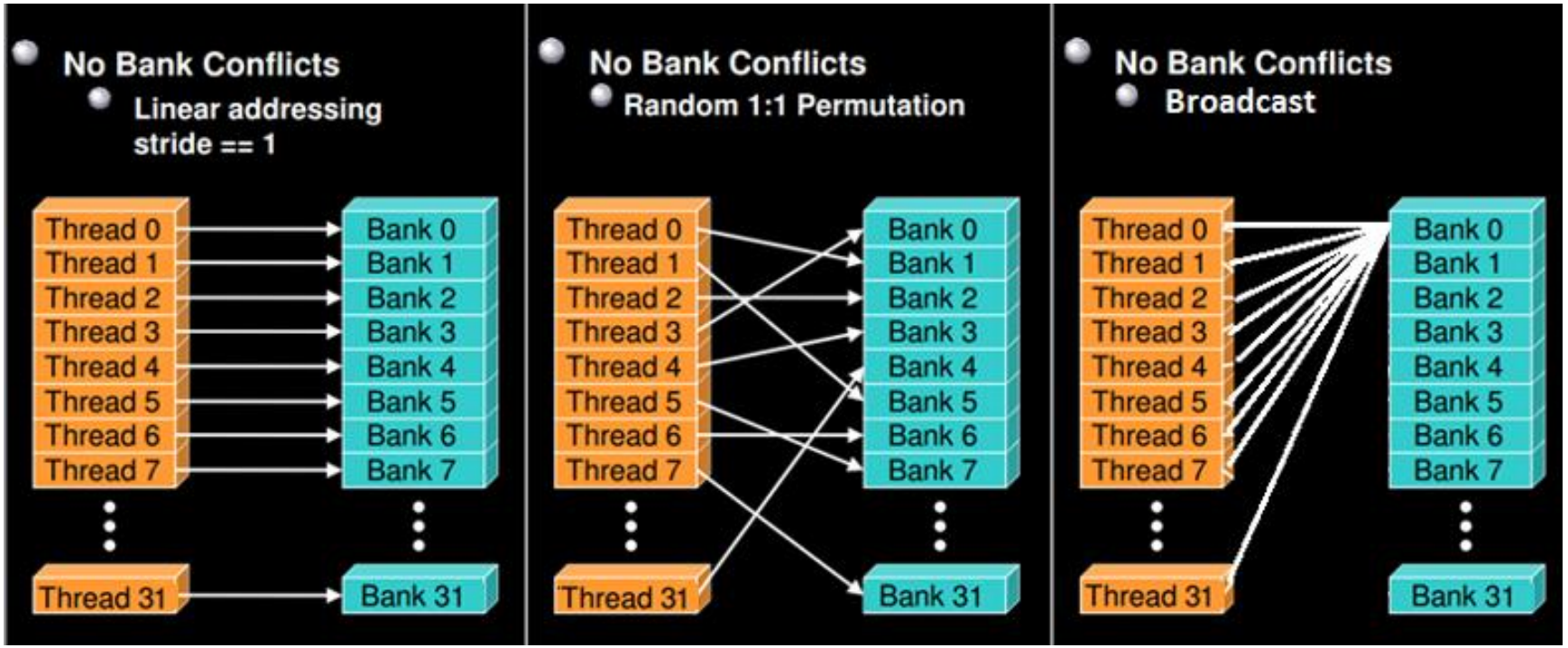
Multiple simultaneous accesses of a warp to a bank result in a bank conflict

Bank	32-bit word			
0	0	32	64	96
1	1	33	65	97
2	2	34	66	98
3	3	35	67	99
4	4	36	68	100
5	5	37	69	101
6	6	38	70	102
7	7	39	71	103
8	8	40	72	104
9	9	41	73	105
10	10	42	74	106
11	11	43	75	107
12	12	44	76	108
13	13	45	77	109
14	14	46	78	110
15	15	47	79	111
16	16	48	80	112
17	17	49	81	113
18	18	50	82	114
19	19	51	83	115
20	20	52	84	116
21	21	53	85	117
22	22	54	86	118
23	23	55	87	119
24	24	56	88	120
25	25	57	89	121
26	26	58	90	122
27	27	59	91	123
28	28	60	92	124
29	29	61	93	125
30	30	62	94	126
31	31	63	95	127

Bank conflicts

The fast case (no bank conflict)

- All threads of a warp access different banks
- All threads of a warp read the identical address (broadcast)

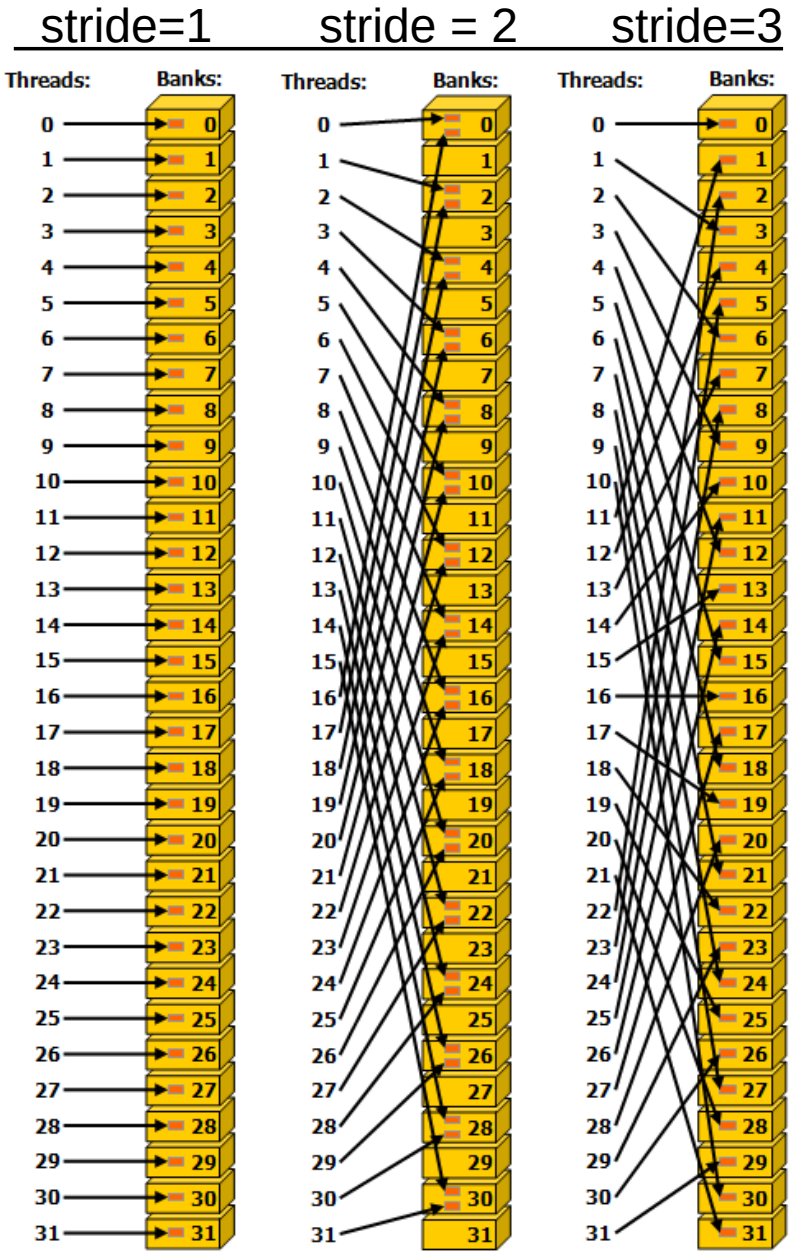


Unlike global memory, there is no penalty for randomly accessing shared memory.

Bank conflicts

The slow case (bank conflict)

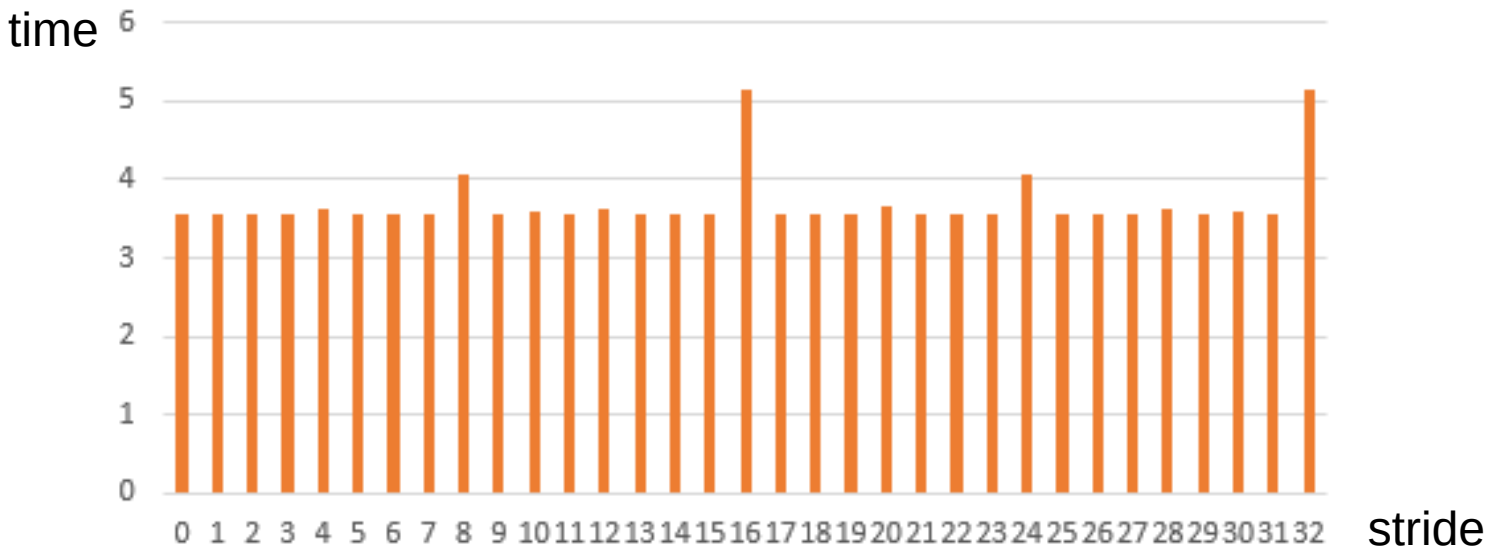
- Multiple threads in the same warp access the same bank
- The accesses are serialized
- The hardware splits a conflicting memory request into as many separate conflict-free requests as necessary,



Exercise

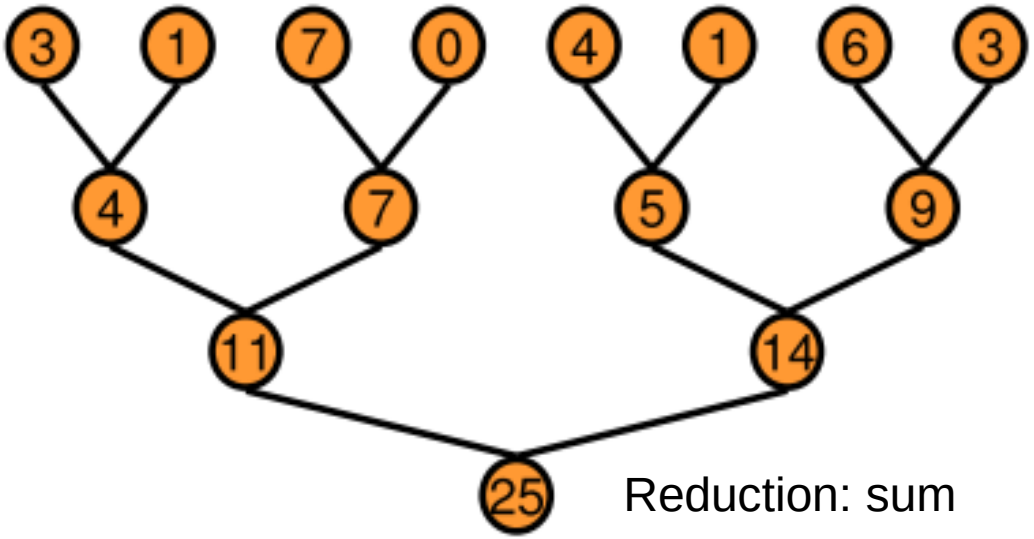
Time a kernel with potential bank conflicts.

```
__global__ void bank(float* d_idata, float* d_odata, int stride)
{
  __shared__ double sm[512]; // double amplifies the conflicts
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  sm[threadIdx.x] = d_idata[i];
  __syncthreads();
  int n = (stride * threadIdx.x) % 512;
  d_odata[i] =  = sm[n];
}
```



Reduction

- Class of parallel algorithms that read N input data and produce a single result (sum, min, max, and, or, dot product…)
- Reduction is a bandwidth-bound algorithm
- Classic example to demonstrate a number of optimization strategies



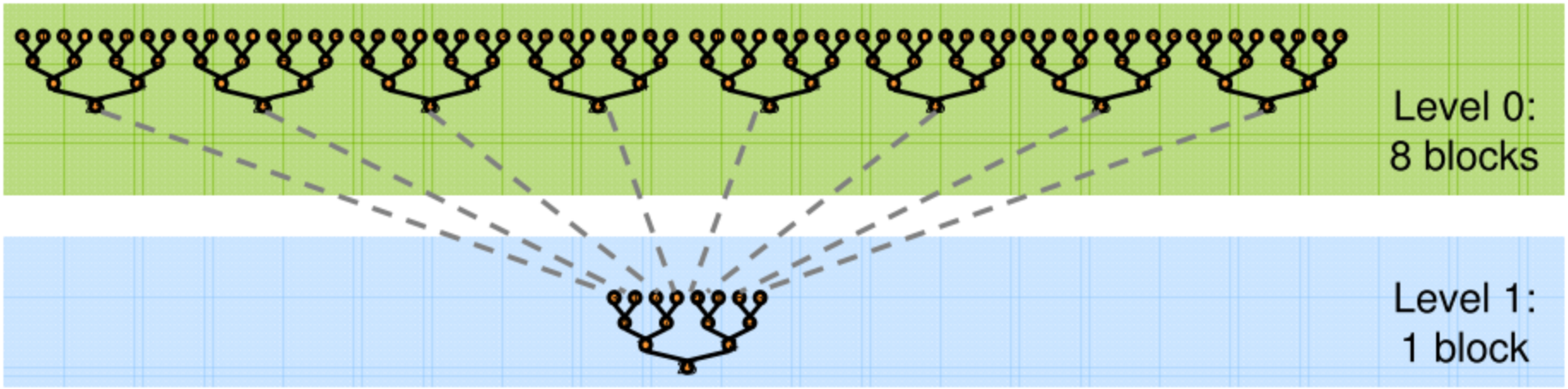
Reduction: sum

Memory traffic

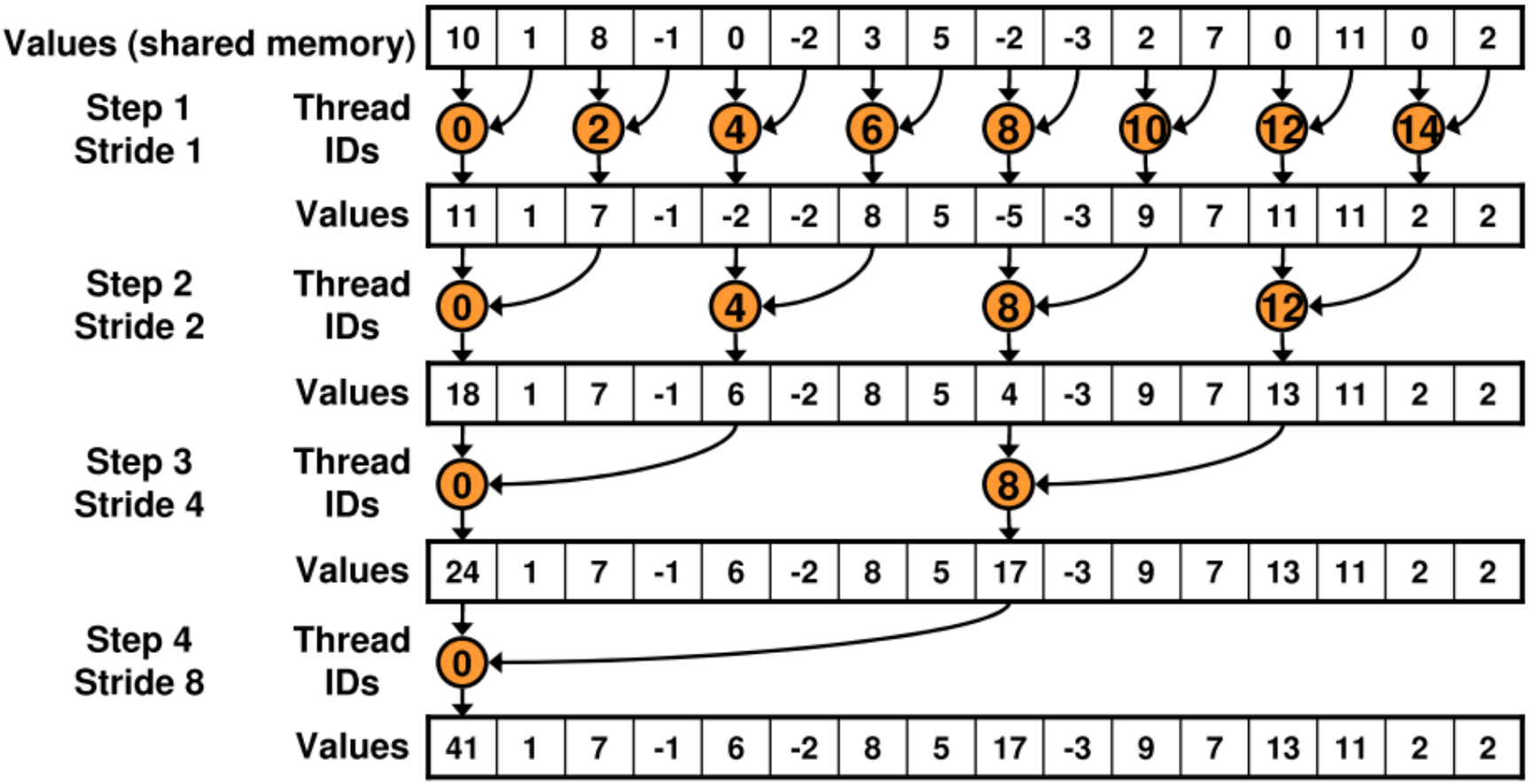
Ideal: n reads, 1 write.

Block size 512 threads:

- Level 0: Read n items, write back n/512 items.
- Level 1: Read n/512 items, write back 1 item.
- Total: n + n/256 + 1



Solution 1: Interleaved addressing



Solution 1: Interleaved addressing

```
__global__ void reduce(int* g_idata, int* g_odata) {
  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] = g_idata[i];
  __syncthreads();
  // 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();
  }
  // write result for this block to global memory
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Exercice

Implement the parallel reduction algorithm (with level 1 CPU reduction) and compute its bandwitdh

$1000*(datasize + datasize/nbThreads) / (1000^3*t\_in\_ms)$ Gb/s

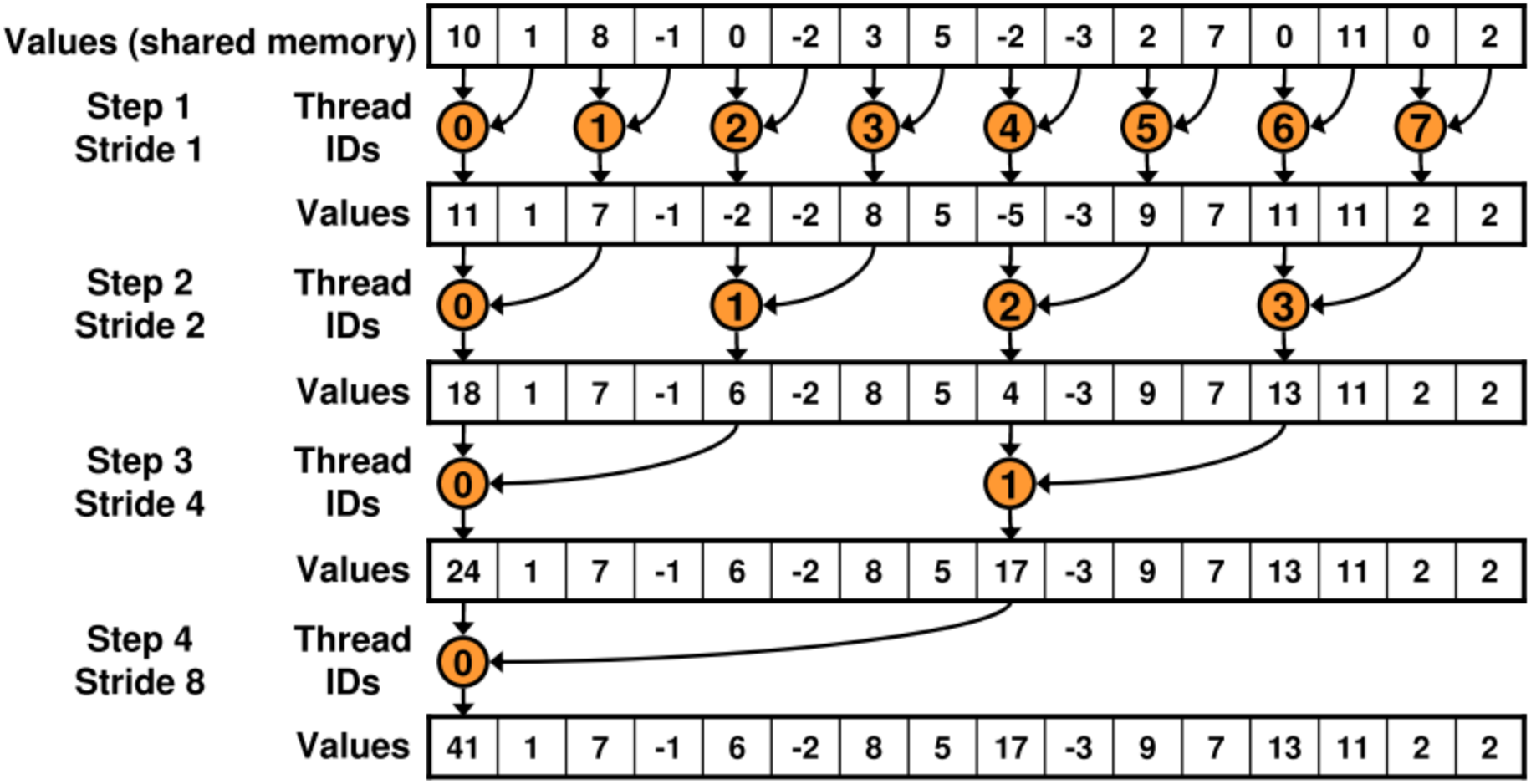
and don’t forget to check your GPU code with the results of a reference CPU code.

Can we do better?

```
__global__ void reduce1(int* g_idata, int* g_odata) {
  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] = g_idata[i];
  __syncthreads();
  // 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();
  }
  // write result for this block to global memory
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

Solution 2: Interleaved addressing II



Solution 2: Interleaved addressing II

Replace divergent branch in inner loop

```
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:

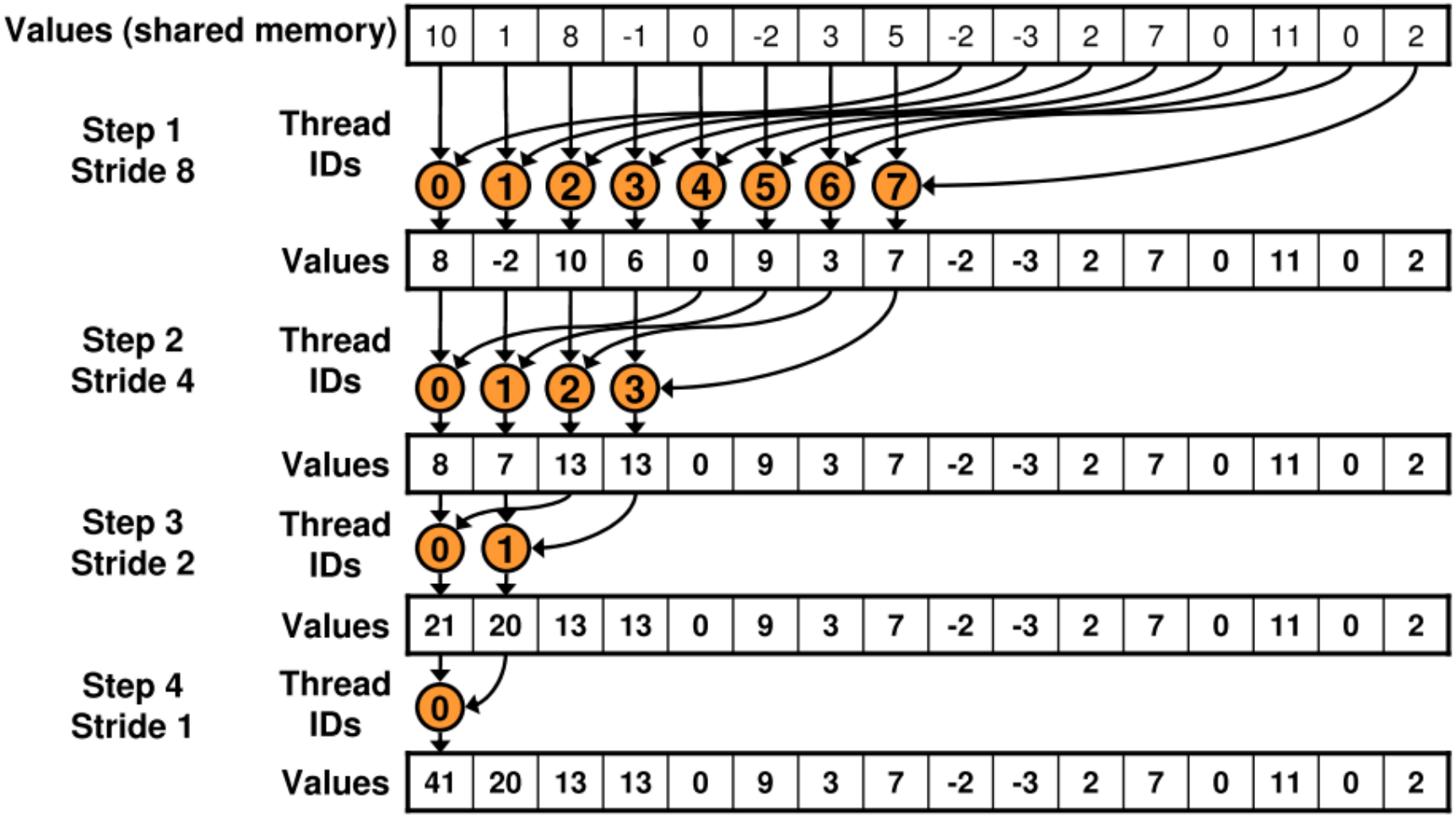
```
for (unsigned int s = 1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

Can we do better?

Shared Memory Bank Conflicts

- In the same warp, we acess the index
  - 0 and 32
  - 1 and 33, …
- Index n et n+32 are stored in the same memory bank
- Sequential addressing is conflict free

Solution 3: Sequential addressing



Solution 3: Sequential addressing

Replace strided indexing in 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 threadID-based indexing:

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

Can we do better?

Idle threads

- Half of the threads are idle on first loop iteration!
- This is wasteful

Solution 4: First add during load

Halve the number of blocks, and replace single load

```
// 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();
```

with two loads and first add of the reduction:

```
// perform first add of reduction upon reading from
// global memory and writing to shared memory
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();
```

Can we do better?

- As reduction proceeds, the number of “active” threads decreases
- When s <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32
  - We don’t need to __syncthreads()
  - We don’t need “if (tid < s)” because it doesn’t save any work

Solution 5: harmonize last warp

Replace the universal loop

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

with two loops, where the second loop addresses the last warp:

```
for (unsigned int s = blockDim.x / 2; s > 32;s >>= 1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32)
{
  for (int s = 32; s > 0; s /= 2)
    sdata[tid] += sdata[tid + s];
}
```

Get wrong results?

CUDA 2.x has a load/store architecture that can bring shared data into registers until the next __syncthreads call. Now that we are using warp-synchronous programming for the last steps (no __syncthreads), we need to declare our shared memory **volatile**. The compiler will not optimize it into registers that might not be written back to the shared memory before another thread reads it.

```
volatile extern __shared__ int sdata[];
```

Can we do better?

Loop raking

Combine sequential and parallel reduction.

Use less blocks and have a thread do more work than just fetch something to shared memory: each thread loads and sums multiple elements into shared memory.

Solution 6: Multiple adds

Replace load and add of two elements

```
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();
```

with a while-loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
unsigned int gridsize = (blockDim.x * 2) * gridDim.x;
sdata[tid] = 0;
while (i < datasize) {
  sdata[tid] += g_idata[i] + g_idata[i + blockDim.x];
  i += gridsize;
}
__syncthreads();
```

Final optimized kernel

```
__global__ void reduce(int* g_idata, int* g_odata, unsigned int datasize) {
    volatile extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    unsigned int gridsize = (blockDim.x * 2) * gridDim.x;
    sdata[tid] = 0;
    while (i < datasize) {
        sdata[tid] += g_idata[i] + g_idata[i + blockDim.x];
        i += gridsize;
    }
    __syncthreads();
    for (unsigned int s = blockDim.x / 2; s > 32;s >>= 1)
    {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    if (tid < 32)
    {
        for (int s = 32; s > 0; s /= 2)
            sdata[tid] += sdata[tid + s];
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```