Performance (Memory) Optimization

National Tsing Hua University

2019, Summer Semester

Communication vs Computation

- ## Peak performance for Kepler
  - The peak processing performance is 3935 Gflops.
  - The bandwidth is 250GB/s, which equals to 63G floating point data per second.
  - The ratio is about 60 times
- ## Instruction execution
  - Each computation **instruction takes 1~4 cycles**
  - Each load/store instruction for **global memory** access takes **400~800 cycles**
  - Memory access to **shared memory** can **be 1~20 cycles**
  - The ratio is about 100 times

Data Pre-fetch and Reuse

- **GPU has faster memory spaces (but smaller)**
  - Shared memory / L1 cache
  - Register file
- **Solution:**
  - Hardware: prefetch data to shared memory or registers for later computation (hardware)
  - Software/Programmer: minimize memory usage & reuse the data in **shared memory** or **registers** as many times as possible

Outline

- **Host memory**
  - Pined memory
  - Asynchronous computation & data transfer
  - Streams
- **Global/Local memory**
  - Memory coalescing
  - Tiled algorithm
- **Shared memory**
  - Bank conflicts avoidance
  - Memory padding
- **Address linearization**

Outline

- **Host memory**
  - Pined memory
  - Asynchronous computation & data transfer
  - Streams
- Global/Local memory   like continuous data
  - Memory coalescing
  - Tiled algorithm
- Shared memory   like striping data
  - Bank conflicts avoidance
  - Memory padding
- Address linearization

1. Page-Locked Data Transfers

- **cudaMallocHost() allows allocation of page-locked ("pinned") host memory**

```
cudaMalloc ( &dev1, size ) ;
cudaMallocHost( &host1, size ) ;   //allocate host memory for GPU to access
                                    pin the memory
…
cudaMemcpy ( dev1, host1, size, H2D ) ;
```

- **Enables highest cudaMemcpy performance**
- **Use with caution!!**
  - Allocating too much page-locked memory can reduce overall system (host) performance
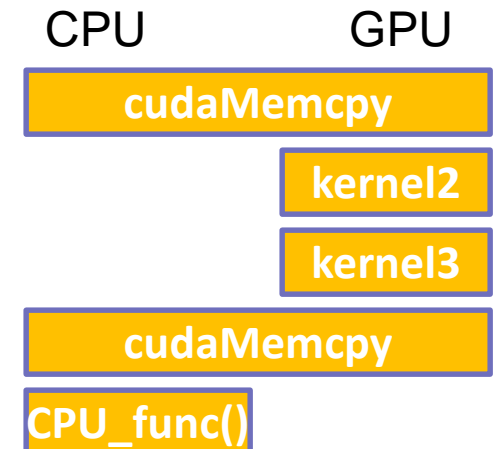
2. Overlap CPU & GPU Computations

- To facilitate concurrent execution between host and device, some function calls are asynchronous:
  - **Control is returned to the host thread before the device has completed the requested task**.
- Asynchronous functions:
  - **Kernel launches**
  - Asynchronous memory copy and set options: **cudaMemcpyAsync, cudaMemsetAsync**
  - **cudaMemcpy** within the **same device**
  - H2D cudaMemcpy of **64kB or less**

Synchronous Computation

```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…
// cudaMemcpy blocks until copy is completed
cudaMemcpy ( dev1, host1, size, H2D ) ;
// two kernels are serialized and executed on device
kernel2 <<< grid, block>>> ( …, dev2, … );
kernel3 <<< grid, block>>> ( …, dev3, … );
// cudaMemcpy starts after kernels finish
// and blocks until copy is completed
cudaMemcpy ( host4, dev4, size, D2H ) ;
CPU_func();
…
```

Kernels from a single thread are serialized

- CPU and GPU are synchronized due to cudaMemcpy
- Kernel functions from the same process (default stream) are always serialized, and cannot be overlapped on GPU

CPU   GPU

| cudaMemcpy |
| kernel2 |
| kernel3 |
| cudaMemcpy |
| CPU_func() |

Asynchronous Computation

```
cudaMalloc(&dev1, size) ;
double* host1=(double*) malloc (&host1, size);
...
cudaMemcpy (dev1, host1, size, H2D) ;
kernel2 <<< grid, block >>> ( …, dev1, … );
kernel3 <<< grid, block >>> ( …, dev1, … );
CPU_method ();
cudaMemcpy ( host1, dev1, size, D2H ) ;
...
```
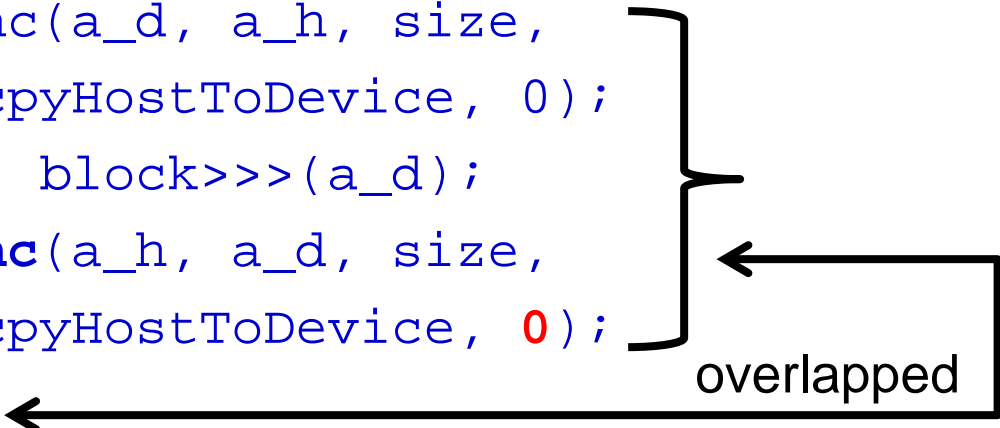
CPU & GPU overlapped

	CPU	GPU
	cudaMemcpy	
CPU_func()		**kernel2**
		kernel3
	cudaMemcpy	
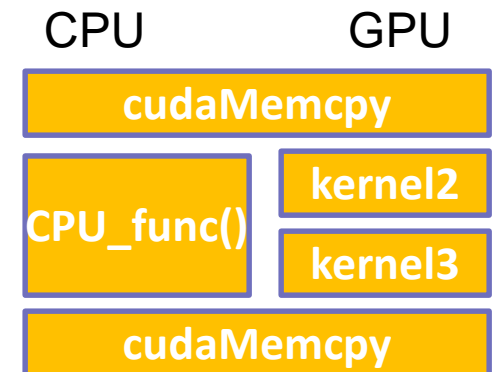
Asynchronous Data Transfers

- **Asynchronous host-device memory copy returns control immediately to CPU**
  - cudaMemcpyAsync(dst, src, size, dir, stream);
  - **requires pinned host memory** (allocated by "cudaMallocHost")
- **Overlap CPU computation with data transfer**
  - **0 = default stream**

```
cudaMemcpyAsync(a_d, a_h, size,
        cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cudaMemcpyAsync(a_h, a_d, size,
        cudaMemcpyHostToDevice, 0);
CPU_method();
```

overlapped

3. CUDA Streams

- CUDA Stream is a technique to **overlap the execution of a kernel**, and **hide data transfer delay from computations**
  - Operations in different streams can be interleaved and, when possible, they can even run concurrently
  - **Operations in the same stream are still serialized and executed in order**
- Consider a kernel process a huge dataset
  - Without stream, the kernel computation can only start after the dataset is transferred

H2D	kernel	D2H

  - With stream, we can partition the dataset, assign each partition to a stream, and execute them in a pipeline

H1	H2	H3		
	K1	K2	K3	
		D1	D2	D3
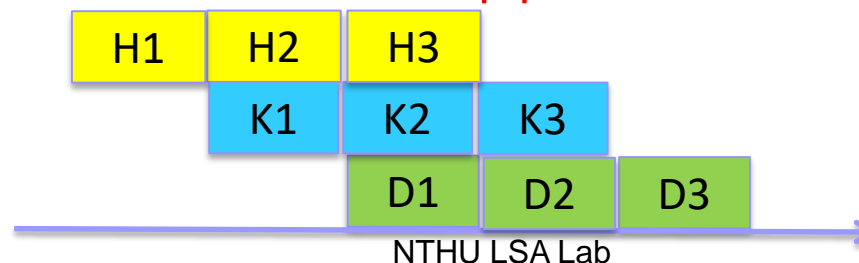
CUDA Streams

- kernel launch
  - `kernel<<<grid,block,0,stream-id>>>(/*…*/);`
- Stream-id must be allocated and destroyed
  - `cudaStream_t *stream;`
  - `cudaStreamCreate(&stream);`
  - `cudaStreamDestroy(stream);`
- Memory copy can be either synchronous or asynchronous. **But synchronous memcpy prevents streams from running in parallel**
- If asynchronous copy is used, host memory must be pinned

CUDA Streams

```
cudaStream_t stream[2];
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(/*…*/,cudaMemcpyHostToDevice,stream[i]);
    kernel<<<100,512,0,stream[i]>>>(/*…*/);
    cudaMemcpyAsync(/*…*/,cudaMemcpyDeviceToHost,stream[i]);
}
cudaStreamDestroy(stream[0]);
cudaStreamDestroy(stream[1]);
```

pined(page locked mem)

Stream based Synchronization

- `cudaStreamSynchronize(stream-id)`
  - Blocks host until all CUDA calls in stream stream-id complete
- `cudaEventRecord (event, stream-id )`
  - Insert 'events' in streams
  - Event is recorded when GPU reaches it in a stream
- `cudaEventSynchronize (event)`
  - Blocks CPU thread until event is recorded
- `cudaStreamWaitEvent (steam-id, event,0)`
  - Block a **GPU stream** until event reports completion

Example: Explicit Sync between Streams

```
cudaEvent_t event;
cudaEventCreate (&event); // create event
// 1) H2D copy of new input
cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );
cudaEventRecord (event, stream1); // record event
// 2) D2H copy of previous result
cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );
// wait for event in stream1
cudaStreamWaitEvent ( stream2, event );
// 3) must wait for 1 and 2
kernel <<< , , , stream2 >>> ( d_in, d_out );
asynchronousCPUmethod ( … ) // Async GPU method
```
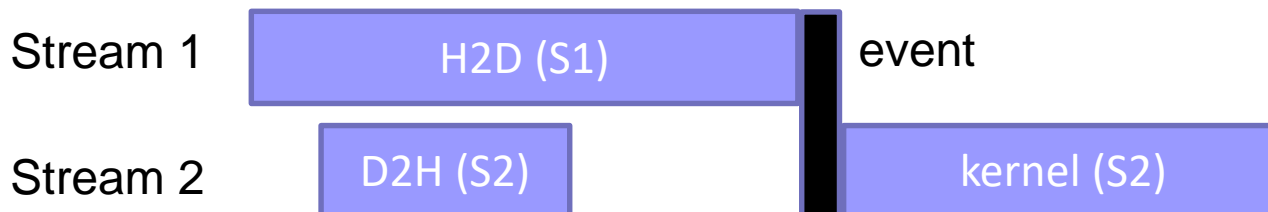
Stream 1 — H2D (S1) — event

Stream 2 — D2H (S2) — kernel (S2)

Outline

- Host memory
  - Pined memory
  - Asynchronous computation & data transfer
  - Streams
- Global/Local memory
  - Memory coalescing
  - Tiled algorithm
- Shared memory
  - Bank conflicts avoidance
  - Memory padding
- Address linearization

Local Memory Cache

- **L1 & L2 are used to cache local memory contents**
  - ➢ L1: On chip memory. Same as share memory
    - ◆ Programmers can decide the ratio of shared memory and L1 cache
 - ➢ L2: Off chip memory Cache. Same as global memory
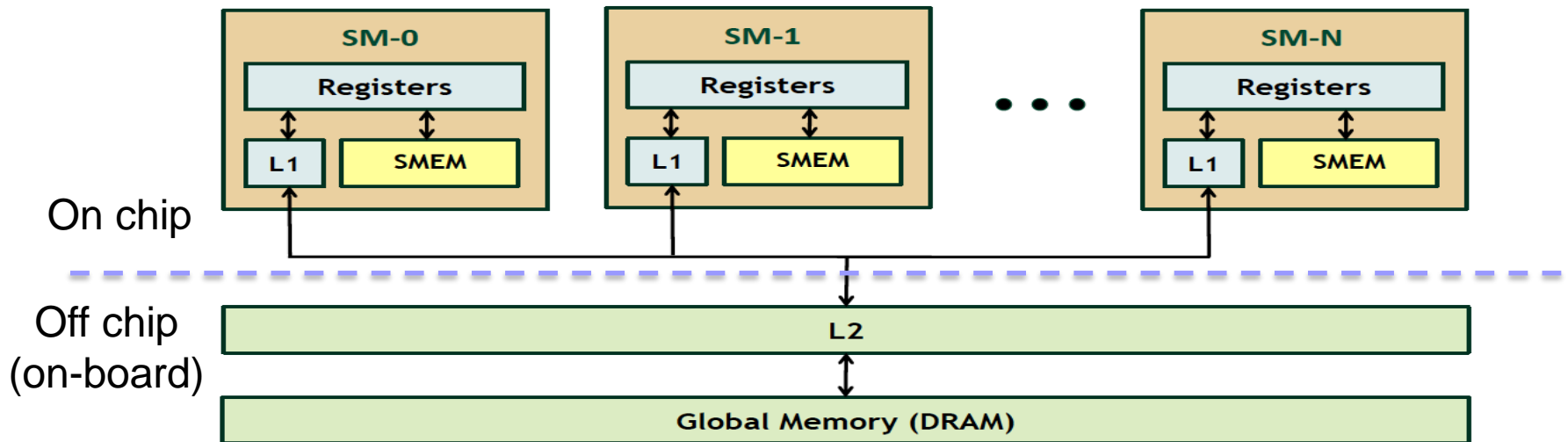
Coalesced Memory Access

- Accessing data in the global memory is critical to the performance of a CUDA application
  - DRAM is slow comparing to other on-chip memory
- Recall that all threads in a warp execute the same instruction
  - When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations
  - In this favorable case, the hardware coalesces all memory accesses into a consolidated access (single transaction) to consecutive **DRAM** locations (off-chip memory)

Coalesced Memory Access

- Coalesced access addresses from a warp
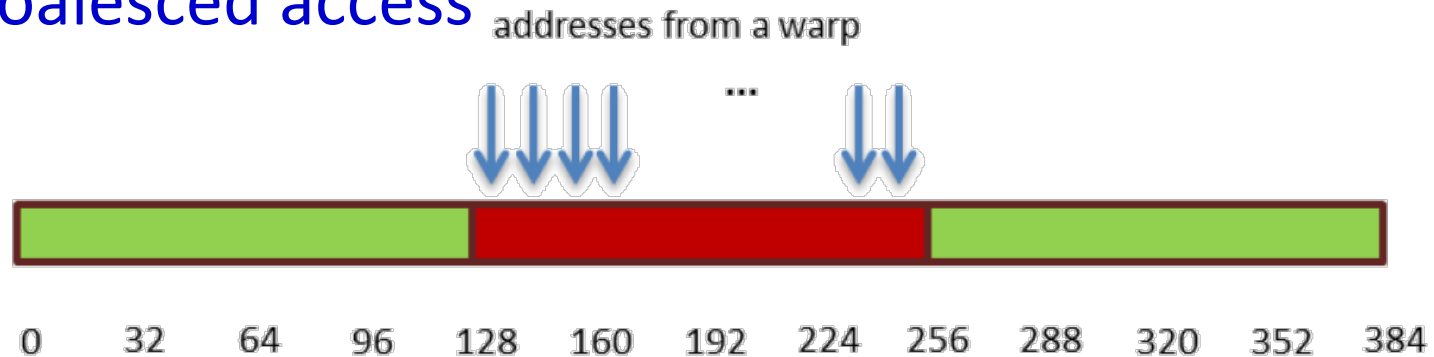
- **Unaligned** sequential addresses that fit into **two** 128-byte **L1-cache lines**
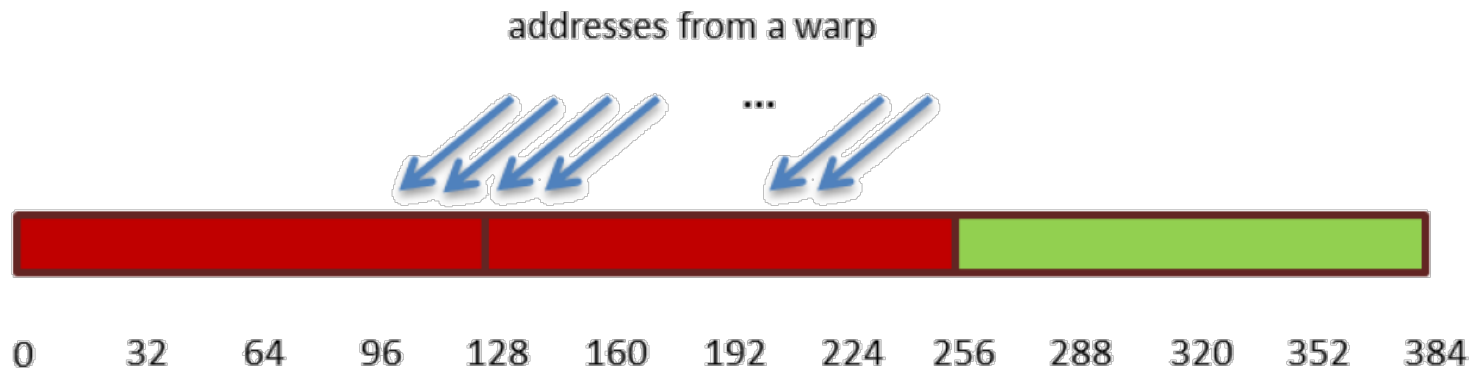
Misaligned Access Without Caching

- Misaligned sequential addresses that fall within five 32-byte **L2 cache segments**
  - No extra data reading

addresses from a warp
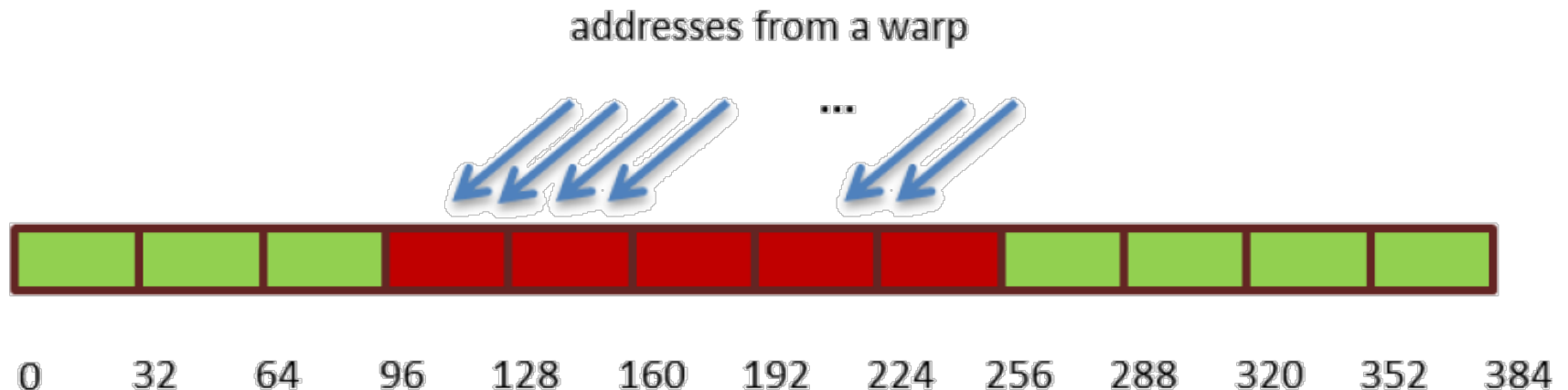
- Sometimes, it will be faster than (L1) cached memory access
  - **If data are not reused**

Example: Matrix Transpose

- **SDK Sample ("transpose")**
- **Illustrates coalescing using shared memory**
  - **Speedups for even small matrices**

Uncoalesced Transpose
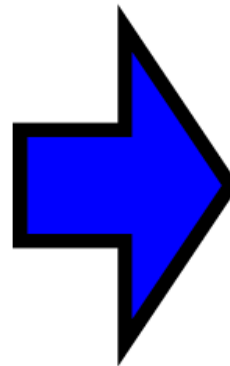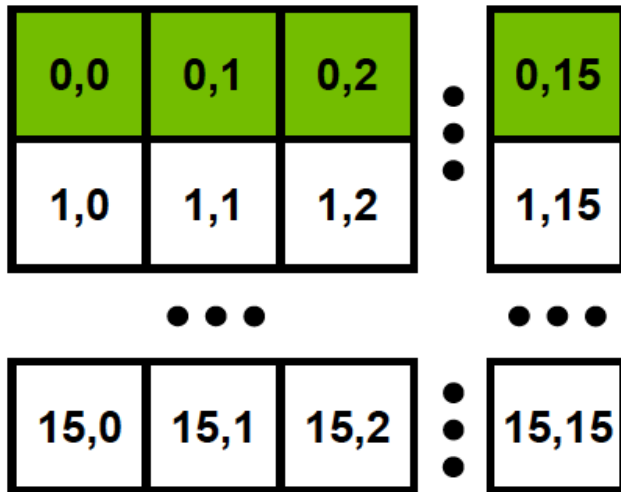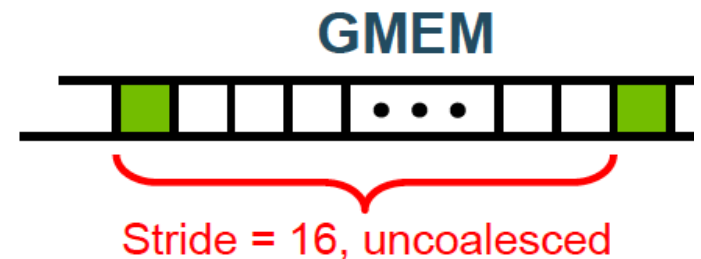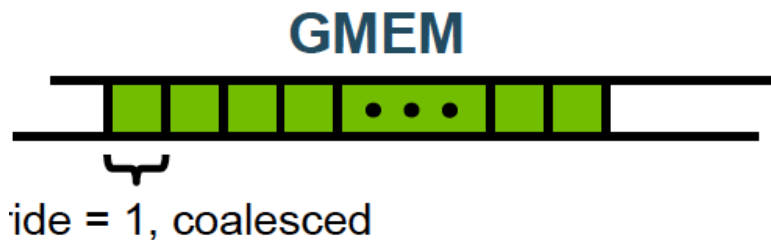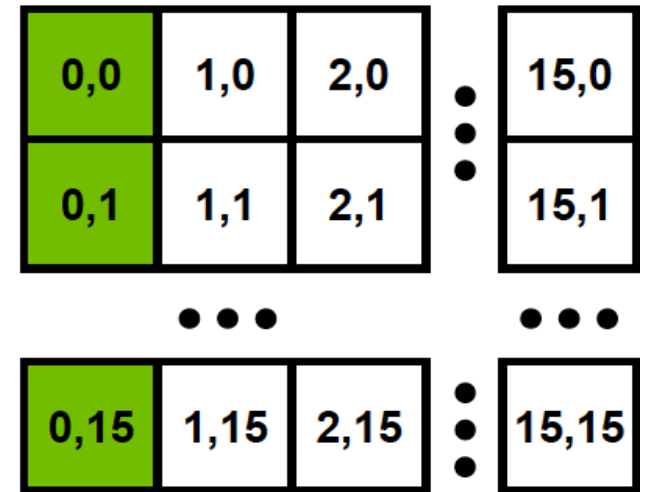
Coalesced Transpose

- Coalescing through shared memory
- Make both read & write become continuous for global memory
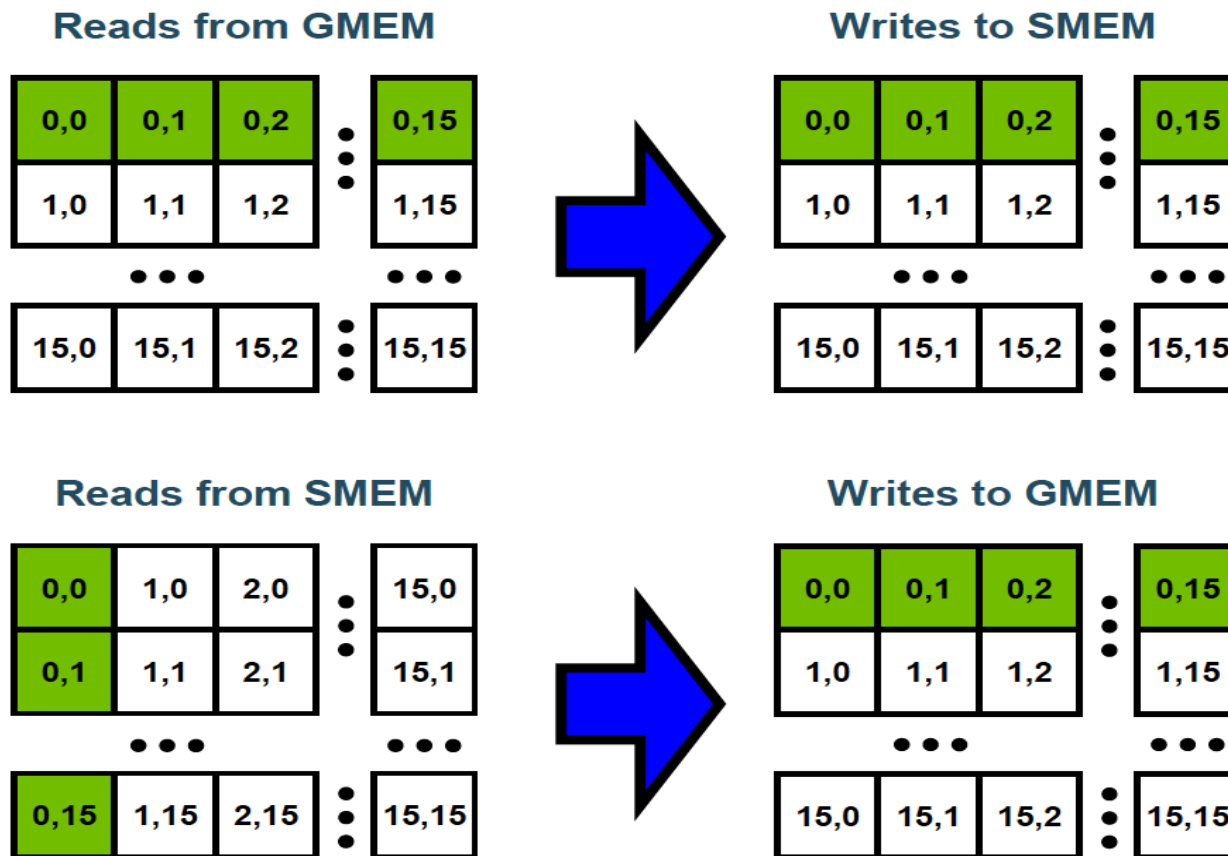
Reads from GMEM

0,0	0,1	0,2	⋮	0,15
1,0	1,1	1,2	⋮	1,15
• • •				• • •
15,0	15,1	15,2	⋮	15,15

➡

Writes to SMEM

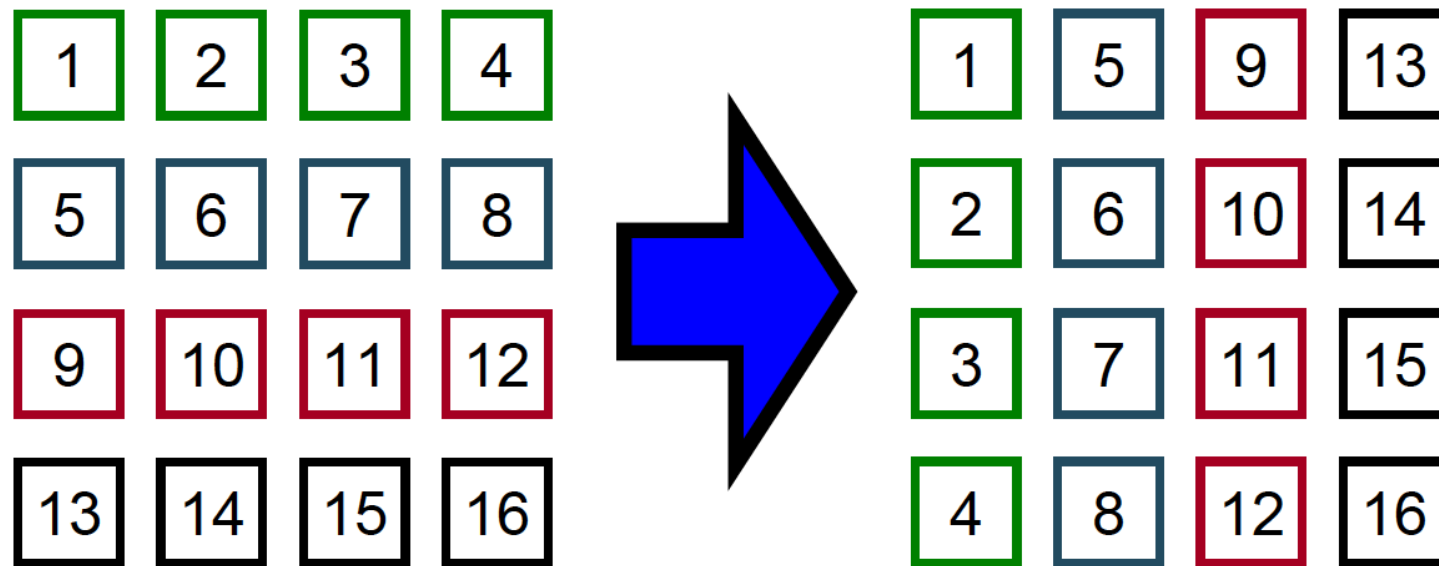| 0,0 | 0,1 | 0,2 | ⋮ | 0,15 |
1,0	1,1	1,2	⋮	1,15
• • •				• • •
15,0	15,1	15,2	⋮	15,15

Reads from SMEM

0,0	1,0	2,0	⋮	15,0
0,1	1,1	2,1	⋮	15,1
• • •				• • •
0,15	1,15	2,15	⋮	15,15

➡

Writes to GMEM

0,0	0,1	0,2	⋮	0,15
1,0	1,1	1,2	⋮	1,15
• • •				• • •
15,0	15,1	15,2	⋮	15,15

Outline

- Host memory
  - Pined memory
  - Asynchronous computation & data transfer
  - Streams
- Global/Local memory
  - Memory coalescing
  - Tiled algorithm
- Shared memory
  - Bank conflicts avoidance
  - Memory padding
- Address linearization

Example: Matrix Multiply

■ Compute C = A x B, where A, B, C are N by N matrices

```
For i = 1:N
    For j = 1:N
        For k = 1:N
            C[i][j]+=A[i][k]*B[k][j]
```

Let each thread compute one element C[i][j]

■ Compute to Global Memory Access (CGMA) ratio

➢ Compute = 1 multiplication + 1 addition; Memory access = 2

➔ CGMA = 1

■ K20x (Kepler)

➢ Compute = 3950 GFLOPs; Global memory BW = 250GB/s

➢ Compute / Comm. = 3950x4/250 ≈ 64

➔ CGMA must increase to 64! Floating point takes 4 bytes

Load Everything to Shared Memory

- Share memory is 100 times faster than global memory
- If N^2 threads are used:
  - Each thread only needs to loads 2 element, and does 2N computations
  - CGMA = N (When N > 64, memory access will not be the bottleneck anymore)

```
For i = 1:N
    For j = 1:N
        For k = 1:N
            C[i][j]+=A[i][k]*B[k][j]
```

- But shared memory is small
  - The data needs to be stored is $3N^2$ integers or floats
  - If N=1024, size = 12MB (i.e., 3*1,024*1,024*4)

Load Everything to Shared Memory

- Matrix_Mul<<<1, N, **2*N*N**>>>(A, B, C, N);
  - ➢ The third parameter is the size of shared memory.

```
extern __shared__ int S[];
inline int Addr(int matrixIdx, int i, int j, int N)  {
    return (N*N*matrixIdx + i*N+ j);
}
__global__ void Matrix_Mul(int* A, int* B,int* C, int* N) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    //move data to shared memory
    S[Addr(0, i, j, N)]=A[Addr(0, i, j, N)];
    S[Addr(1, i, j, N)]=B[Addr(0, i, j, N)];
    __syncthreads();
    // do computation
    for(int k=0; k<*N; k++)
        C[Addr(1, i, j, N)]=S[Addr(0, j, k, N)]*S[Addr(0, k, j, N)];
}
```

Block(Tiled) Algorithm

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of data

- Not all problems can be partitioned into independent subsets

TILE_WIDTH = 2

Block(Tiled) Algorithm
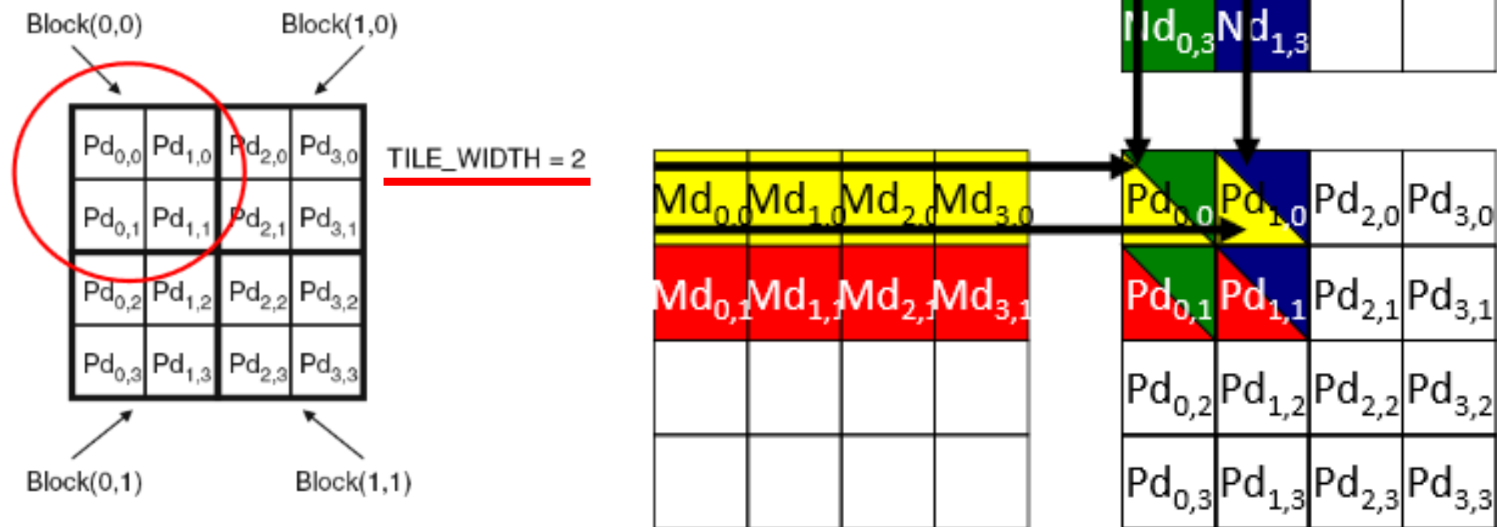
- **Rewrite for-loop by TILE_WIDTH**

Total required data accesses
$$= 2 \times (\text{TILE\_WIDTH})^2$$

Total computing $= 2 \times (\text{TILE\_WIDTH})^3$

```
For i' = 1:N step TILE_WIDTH
    For j' = 1:N step TILE_WIDTH
        For k' = 1:N step TILE_WIDTH
            For i = i': i'+ TILE_WIDTH - 1
                For j = j': j'+ TILE_WIDTH - 1
                    For k = k': k'+ TILE_WIDTH - 1
                        C[i][j]+=A[i][k]*B[k][j]
```

- **We can find a small enough TILE_WIDTH, such that all the values needed by C[i][j] are in shared memory**

    ➔ Every data is re-used TILE_WIDTH times

- **Given 48KB shared memory:** Include output array C[][]

    ➢ Max tiled size = (48KB/4B/3)^(1/2) = 64

    ➢ CGMA = number of data re-use = TILE_WIDTH = 64!

NTHU LSA Lab

29

```
extern __shared__ int S[];
inline int Addr(int matrixIdx, int i, int j, int N)  {
    return (N*N*matrixIdx + i*N+ j);
}
__global__ void Matrix_Mul(int* A, int* B,int* C, int* N) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    //move data to shared memory
    S[Addr(0, i, j, N)]=A[Addr(0, i, j, N)];
    S[Addr(1, i, j, N)]=B[Addr(0, i, j, N)];
    __syncthreads();
    // do computation
    for(int k=0; k<*N; k++)
        C[Addr(1, i, j, N)]=A[Addr(0, j, k, N)]*B[Addr(0, k, j, N)];
}
void main()  {
   for(int i=0; i<N; i+=TILE_WIDTH)
    for(int j=0; j<N; j+=TILE_WIDTH){
      cudaMemcpy(d_A, &A[i,j], sizeof(int)*TILE_WIDTH*TILE_WIDTH, H2D);
      cudaMemcpy(d_B, &B[i,j], sizeof(int)*TILE_WIDTH*TILE_WIDTH, H2D);
       Matrix_Mul<<<1, N, 2*N*N>>>(d_A, d_B, d_C, TILE_WIDTH);
      cudaMemcpy(&C[i,j], d_C, sizeof(int)*TILE_WIDTH*TILE_WIDTH), D2H;
    }
}
```

Tiled Algorithm

- Block algorithms or tiled algorithms:
  - Split the inputs into blocks to fit into shared (cache) memory
  - Increase data reuse, minimize global memory access

- Larger CGMA ratio does not always guarantee better performance.
  - CGMA ratio should be large enough to hide the communication cost, not the larger the better
  - Block algorithms cause overhead due to increasing computations or number of thread blocks

Outline

- Host memory
  - Pined memory
  - Asynchronous computation & data transfer
  - Streams
- Global/Local memory
  - Memory coalescing
  - Tiled algorithm
- Shared memory
  - Bank conflicts avoidance
  - Memory padding
- Address linearization

Shared Memory Architecture

- **Many threads accessing memory**
  - Therefore, memory is divided into banks
  - Successive 32-bit (4Bytes) words assigned to successive banks
- **Each bank can service one address per cycle**
  - A memory can service as many simultaneous accesses as it has banks
- **Multiple simultaneous accesses to a bank result in a bank conflict**
  - Conflicting accesses are serialized
- **Shared memory is as fast as register if no bank conflict**

| Bank0 |
| Bank1 |
| Bank2 |
| Bank3 |
| Bank4 |
| Bank5 |
| Bank6 |
| Bank7 |

...

| Bank15 |
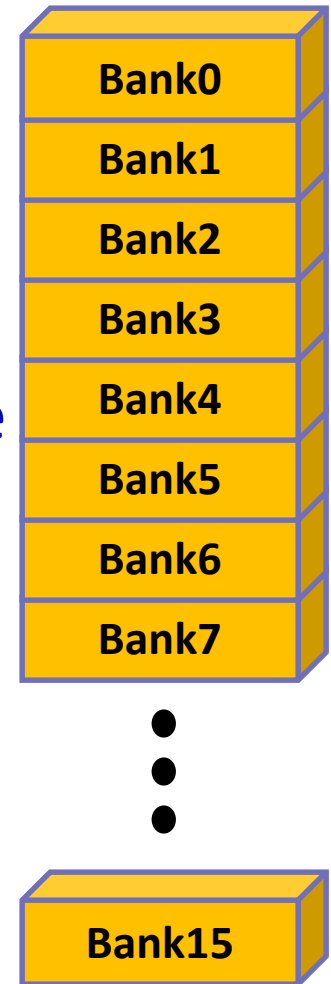
Example: No bank Conflict
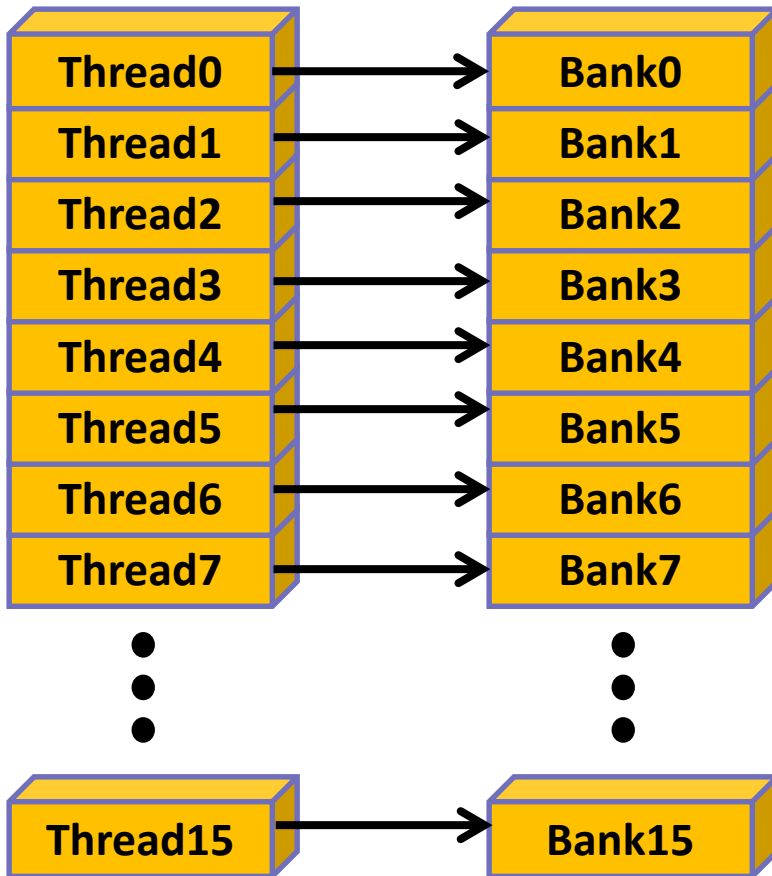
- Linear addressing
- Random 1:1 Permutation
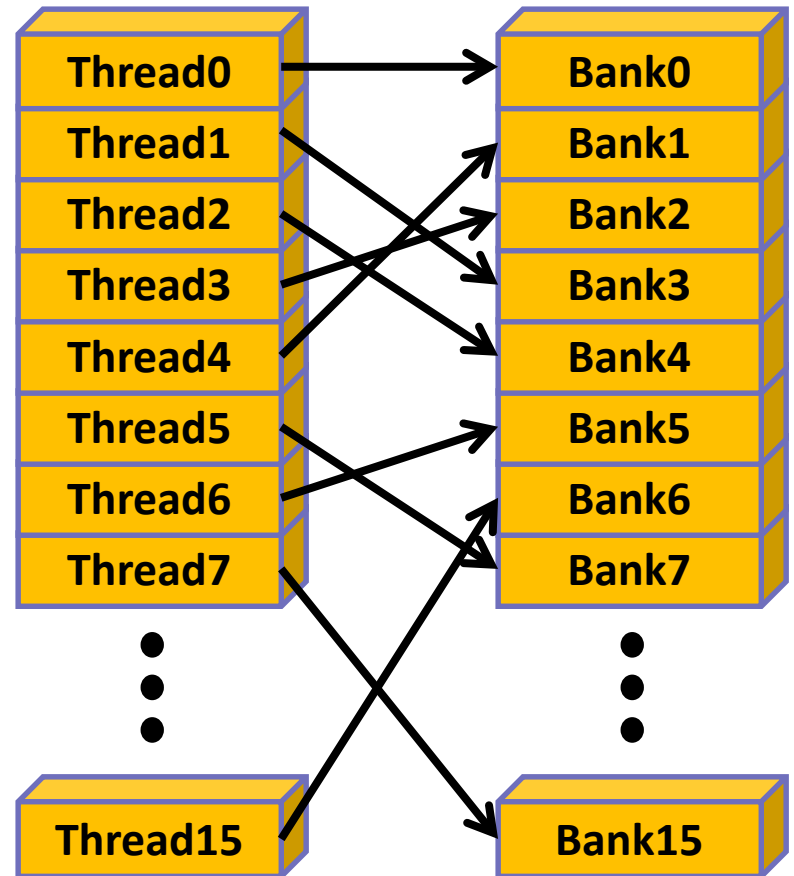
Example: No bank Conflict

- If all threads of a half-warp read the identical address, there is no bank conflict (using broadcast)
  - Assume warp size is 8
  - Thread0~3 access the same data & in the same half-warp
  - The rest of threads also have 1:1 permutation and no conflict
  - But not for write access
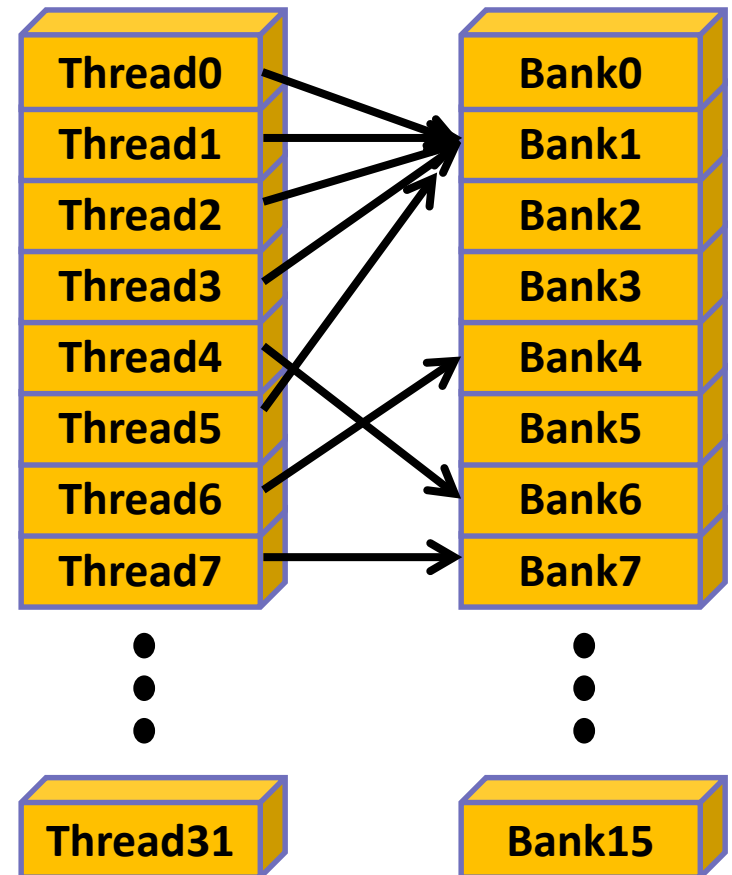
Example: Bank Conflict

- **n-way bank conflict**
  - ➢ Each bank has n different memory access

- **Ex: 2-way bank conflict**

```
__shared__ int array[2][32];
int offset = threadIdx.x*2;
int temp = array[offset/32][offset%32];
```
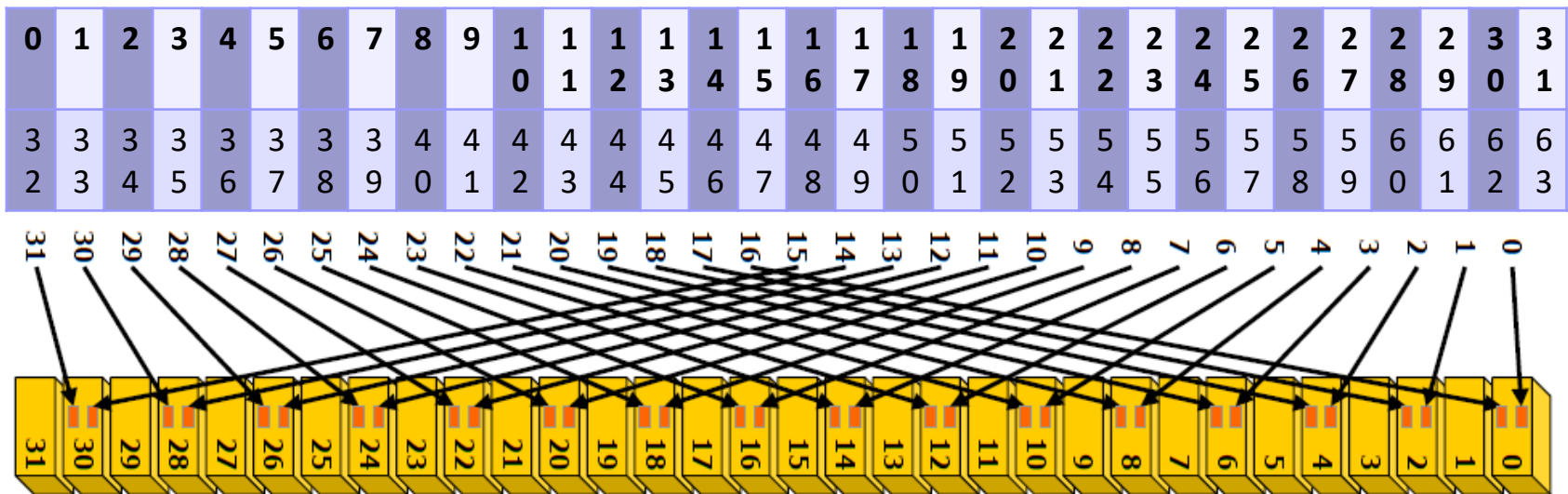
Bank Conflict Avoidance

- **Change shared memory access pattern**
  - Linear addressing access
  - 1:1 permutation
  - Broadcast: half-warp read the identical address

- **Memory padding**
  - Add addition memory space to avoid bank conflict

Example: 2D array

- ## 32x32 SMEM array
  - ➢ Warp accesses a column:
  - ➢ 32-way bank conflicts (threads in a warp access the same bank)
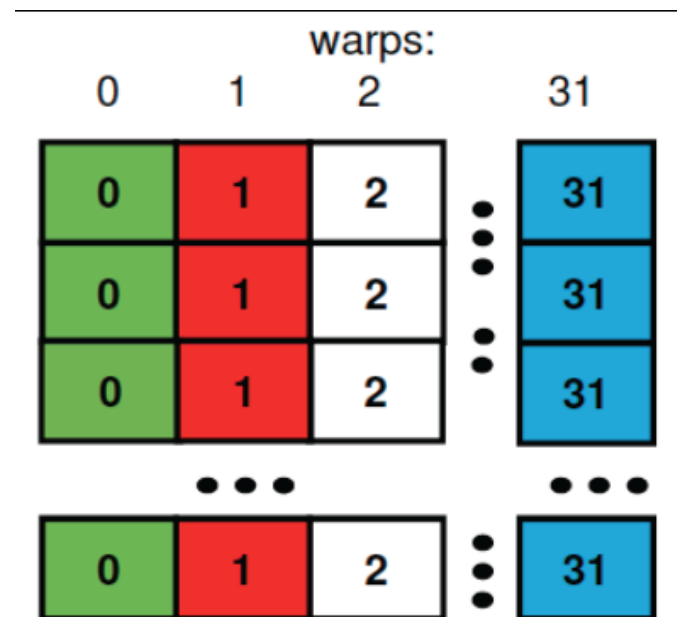
Memory Padding

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts
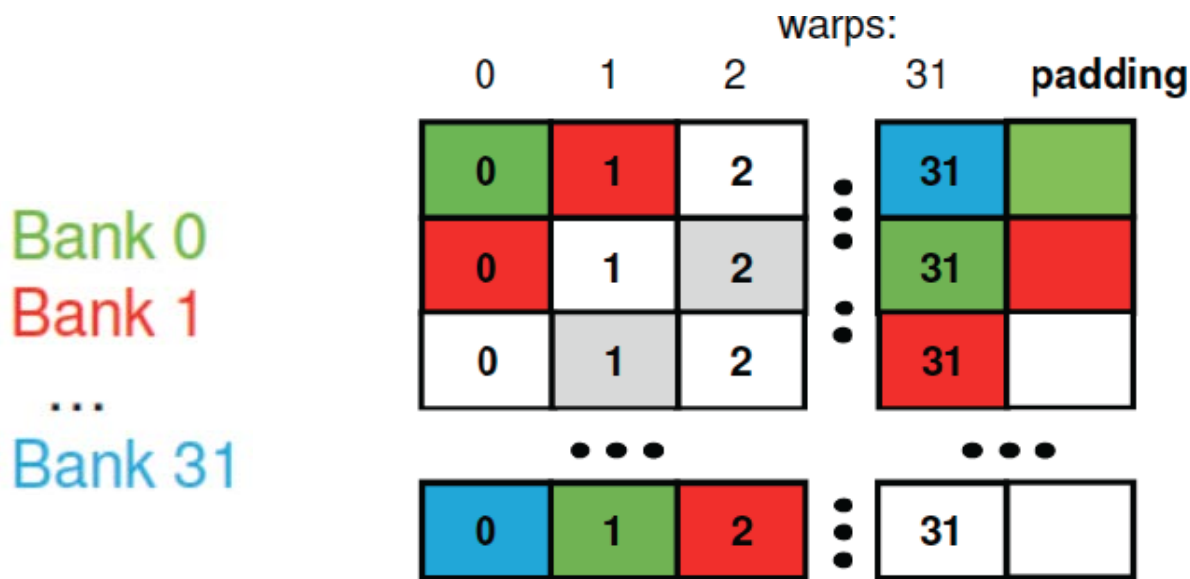
Address linearization (SoA)

- **Address linearization** can avoid **bank conflict** on **shared memory**, and provide **memory coalescing** on **local memory** or **constant memory**

- An array of structures behaves like row major accesses
  - `struct Point { double x; double y; double z;} A[N];`
  - `A[threadIdx].x = …`

A[1].x	A[1].y	A[1].z	A[2].x	A[2].y	A[2].z	A[3].x	A[3].y	A[3].z
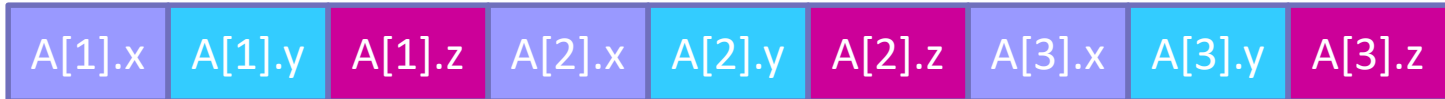
- A structure of arrays behaves like column major
  - `struct PointList{double *x; double *y; double *z;} A;`
  - `A.x[threadIdx] = …`

A[1].x	A[2].x	A[3].x	A[1].y	A[2].y	A[3].y	A[1].z	A[2].z	A[3].z
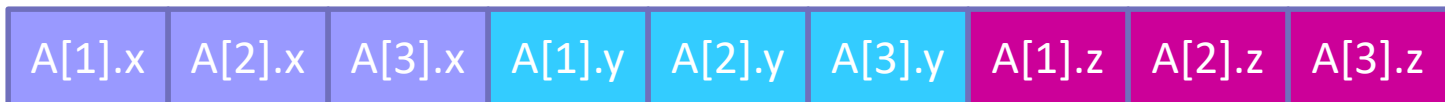
Slides from Mark Harris, NVIDIA Developer Technology
Performance Optimization

AN EXAMPLE OF CUDA

Parallel Reduction

- Common and important data parallel primitive

- Easy to implement in CUDA
  - Harder to get it right

  Performance!

- Serves as a great optimization example

  30x Speedup!
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

Parallel Reduction

- **Tree-based approach used within each thread block**

Run on block1

Run on block2

Block1 needs the result of 14 from block1
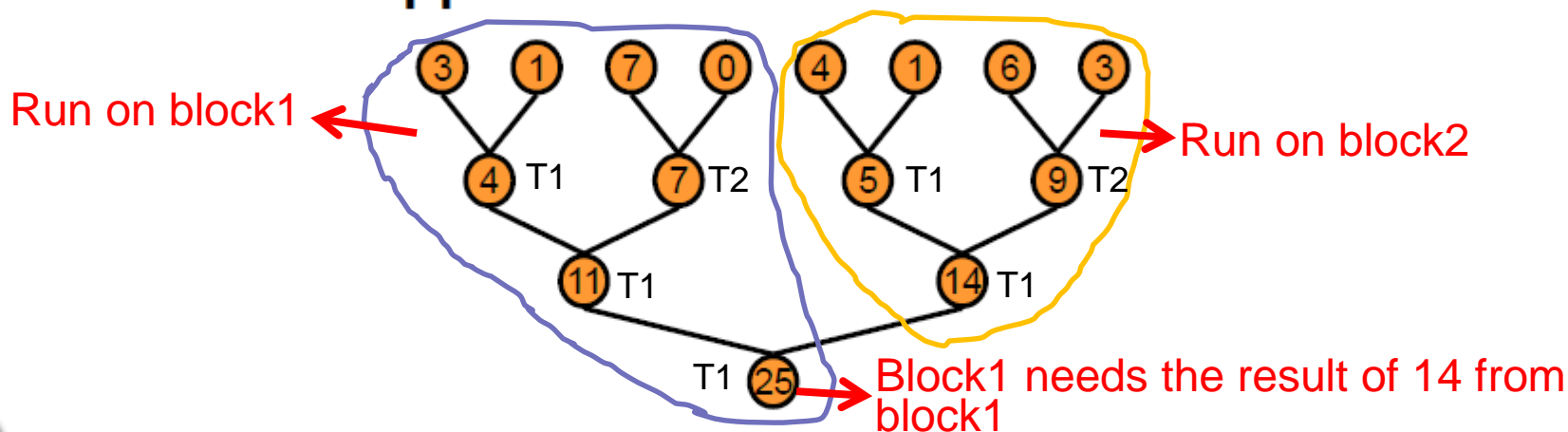
- **Need to be able to use multiple thread blocks**
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- **But how do we communicate partial results between thread blocks?**

Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
    - Global sync after each block produces its result
    - Once all blocks reach sync, continue recursively
- But **CUDA has no global synchronization.** Why?
    - **Expensive to build in hardware** for GPUs with high processor count
    - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency

- Solution: **decompose into multiple kernels**
    - Kernel launch serves as a global synchronization point
    - Kernel launch has negligible HW overhead, low SW overhead

Solution: Kernel Decomposition

○ Avoid global sync by decomposing computation into multiple kernel invocations

If the maximum threads per block is 8

Level 0:
8 blocks
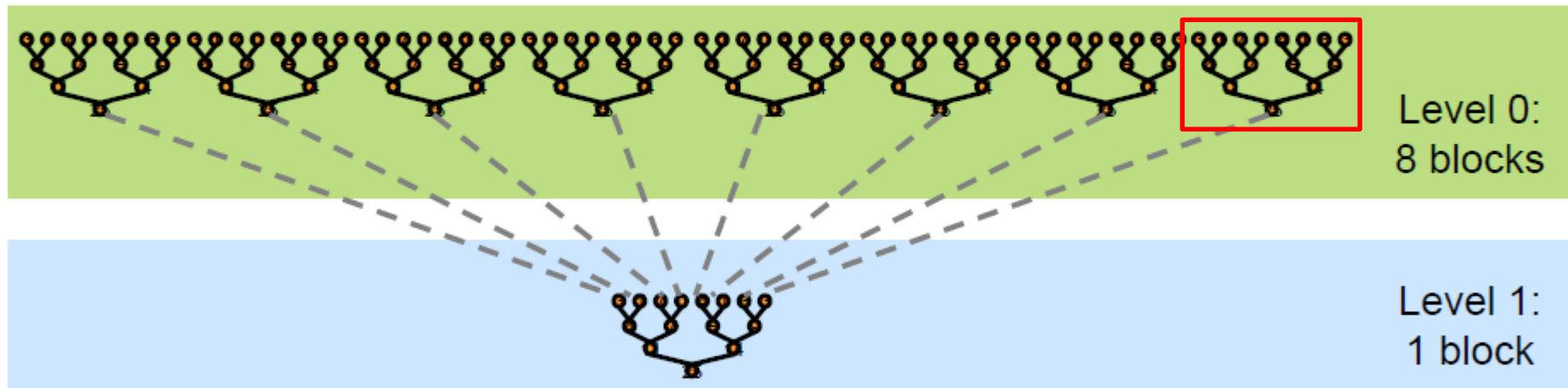
Level 1:
1 block

○ In the case of reductions, code for all levels is the same

  ○ Recursive kernel invocation

Reduction #1: Interleaved Addressing

```
// input/output data is initiated on global memory
__global__ void reduce0(int *g_idata, int *g_odata) {
extern __shared__ int sdata[];  // Use shared memory for computations

  // 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();  // Wait for other threads to finish moving

  // 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();  // Sync between threads in the same block
  }

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

Parallel Reduction: Interleaved Addressing
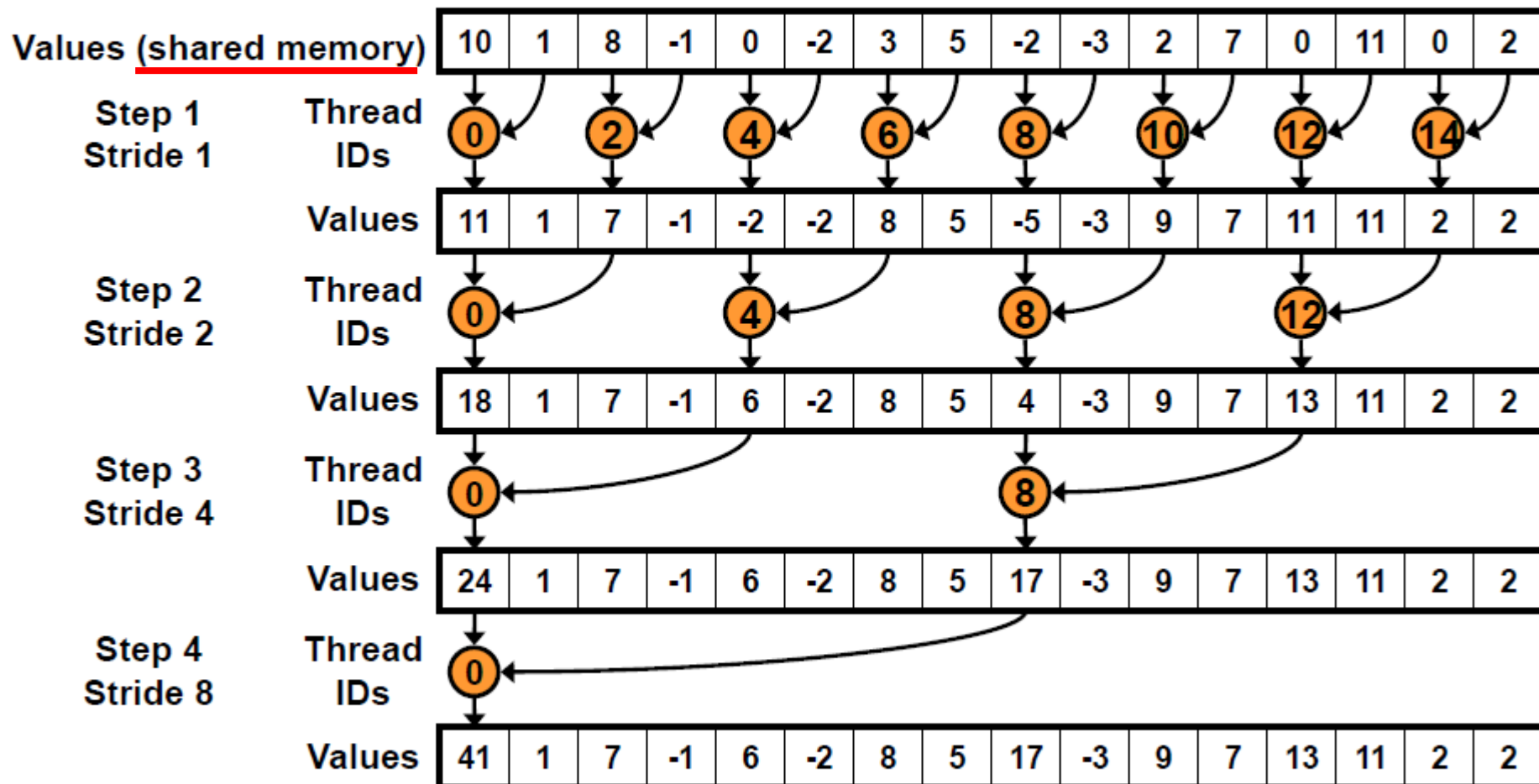
Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Parallel Reduction: Interleaved Addressing

If WARP=4:

| Values (shared memory) | 10 | 1 | 8 | -1 | 0 | -2 | 3 | 5 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 1 Stride 1 — Thread IDs: 0, 2, 4, 6, 8, 10, 12, 14

Values: 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

Step 2 Stride 2 — Thread IDs: 0, 4, 8, 12 — 4WARP

Values: 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2

Step 3 Stride 4 — Thread IDs: 0, 8 — 2WARP

Values: 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Step 4 Stride 8 — Thread IDs: 0 — 1WARP

Values: 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
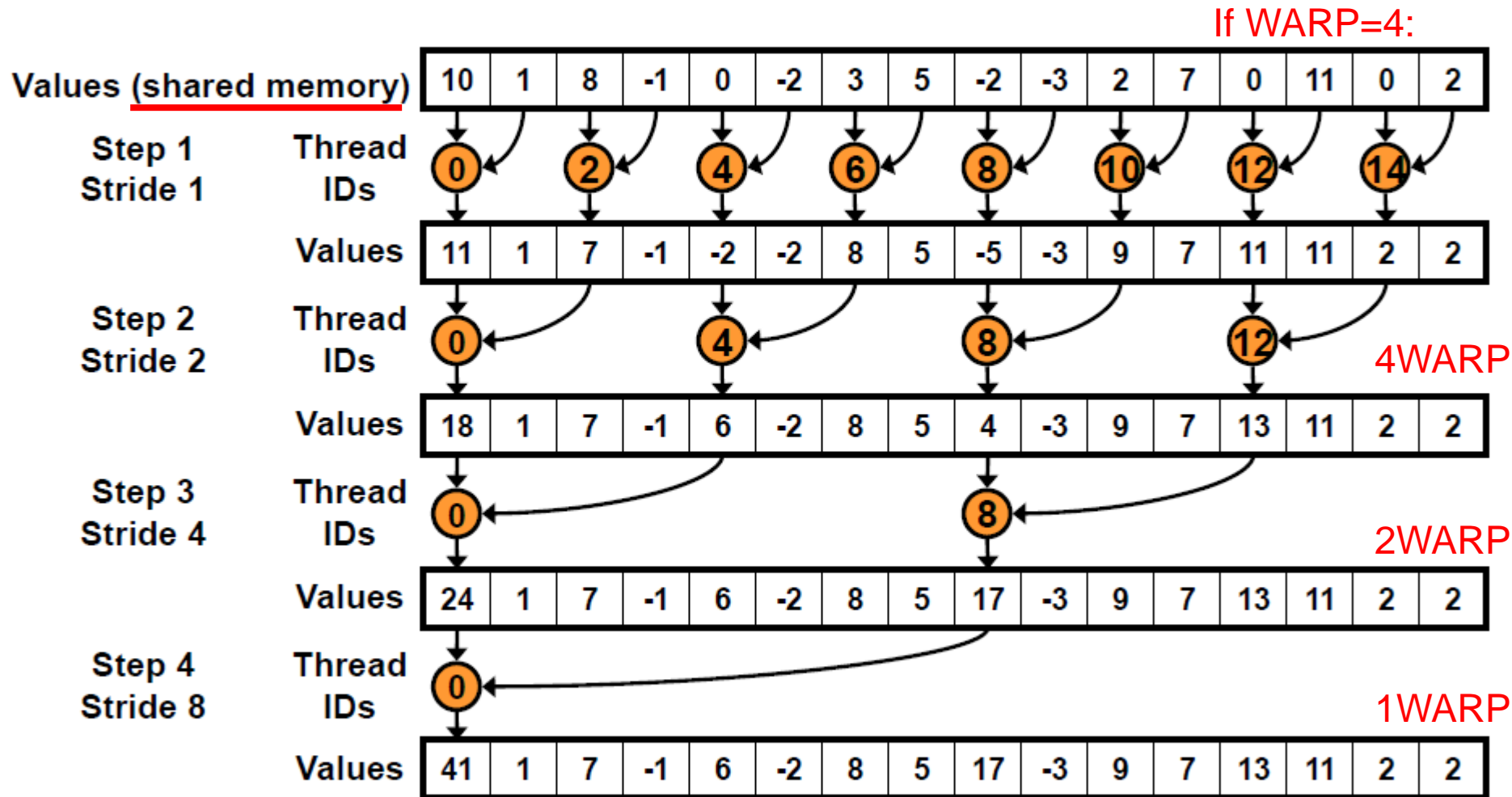
Highly divergent wrap (threadID 0~14)

Reduction #1: Interleaved Addressing

```
__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 mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

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

Parallel Reduction: Interleaved Addressing

If WARP=4:

| Values (shared memory) | 10 | 1 | 8 | -1 | 0 | -2 | 3 | 5 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 1 / Stride 1 — Thread IDs: 0 1 2 3 4 5 6 7

| Values | 11 | 1 | 7 | -1 | -2 | -2 | 8 | 5 | -5 | -3 | 9 | 7 | 11 | 11 | 2 | 2 |

Step 2 / Stride 2 — Thread IDs: 0 1 2 3   1WARP

| Values | 18 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 4 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

Step 3 / Stride 4 — Thread IDs: 0 1   1WARP

| Values | 24 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 17 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

Step 4 / Stride 8 — Thread IDs: 0   1WARP

| Values | 41 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 17 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

NTHU LSA Lab

51

Reduction #2: Interleaved Addressing

Just 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();
}
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Parallel Reduction: Interleaved Addressing

| Values (shared memory) | 10 | 1 | 8 | -1 | 0 | -2 | 3 | 5 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 1 Stride 1 — Thread IDs: 0 1 2 3 4 5 6 7

| Values | 11 | 1 | 7 | -1 | -2 | -2 | 8 | 5 | -5 | -3 | 9 | 7 | 11 | 11 | 2 | 2 |

Step 2 Stride 2 — Thread IDs: 0 1 2 3

| Values | 18 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 4 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

Step 3 Stride 4 — Thread IDs: 0 1

| Values | 24 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 17 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

Step 4 Stride 8 — Thread IDs: 0

| Values | 41 | 1 | 7 | -1 | 6 | -2 | 8 | 5 | 17 | -3 | 9 | 7 | 13 | 11 | 2 | 2 |

New Problem: Shared Memory Bank Conflicts

54

Parallel Reduction: Sequential Addressing

Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

Step 1 Stride 8 — Thread IDs: 0 1 2 3 4 5 6 7

Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 2 Stride 4 — Thread IDs: 0 1 2 3

Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 3 Stride 2 — Thread IDs: 0 1

Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 4 Stride 1 — Thread IDs: 0

Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Sequential addressing is conflict free

55

Reduction #3: Sequential Addressing

Just 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();
}
```
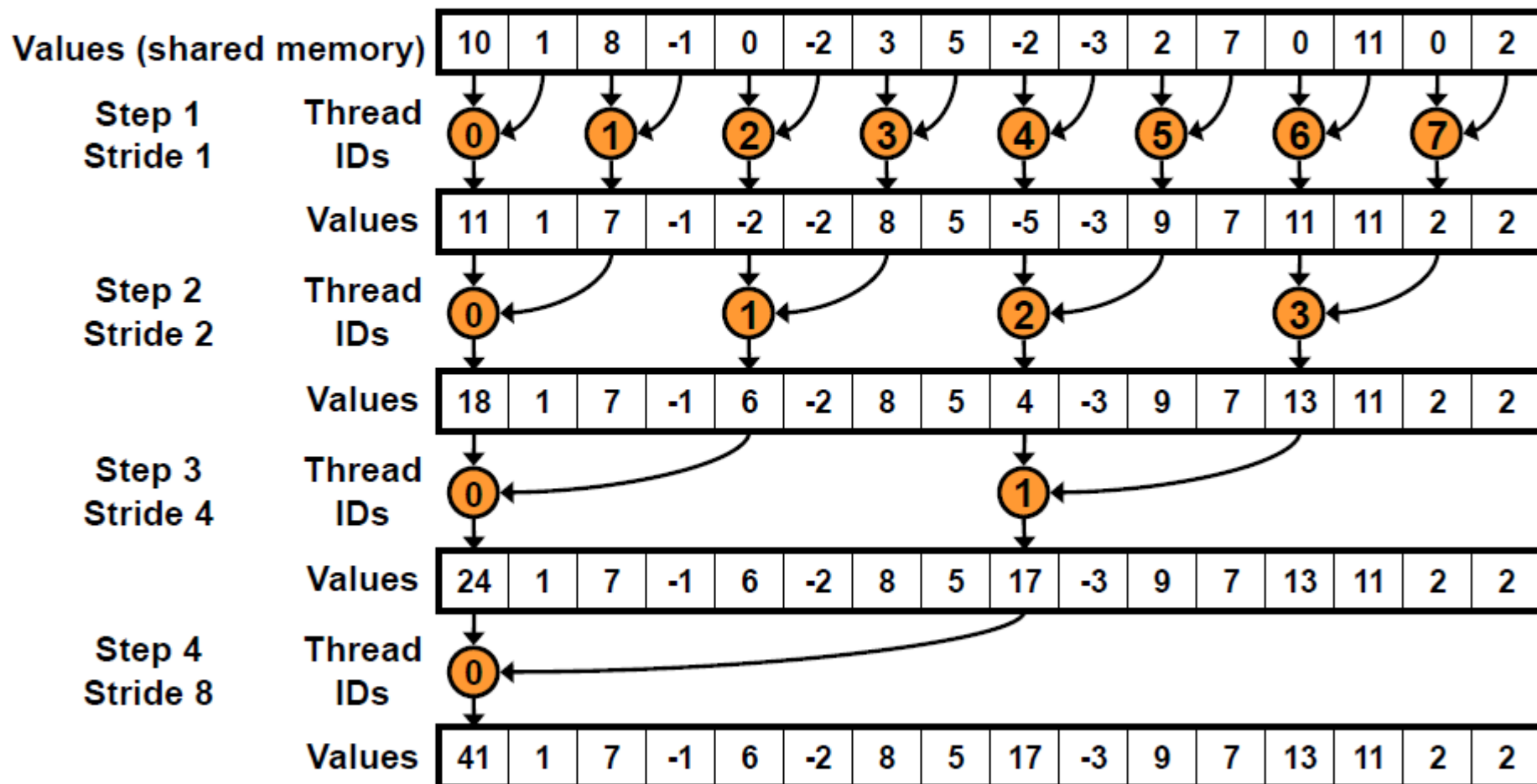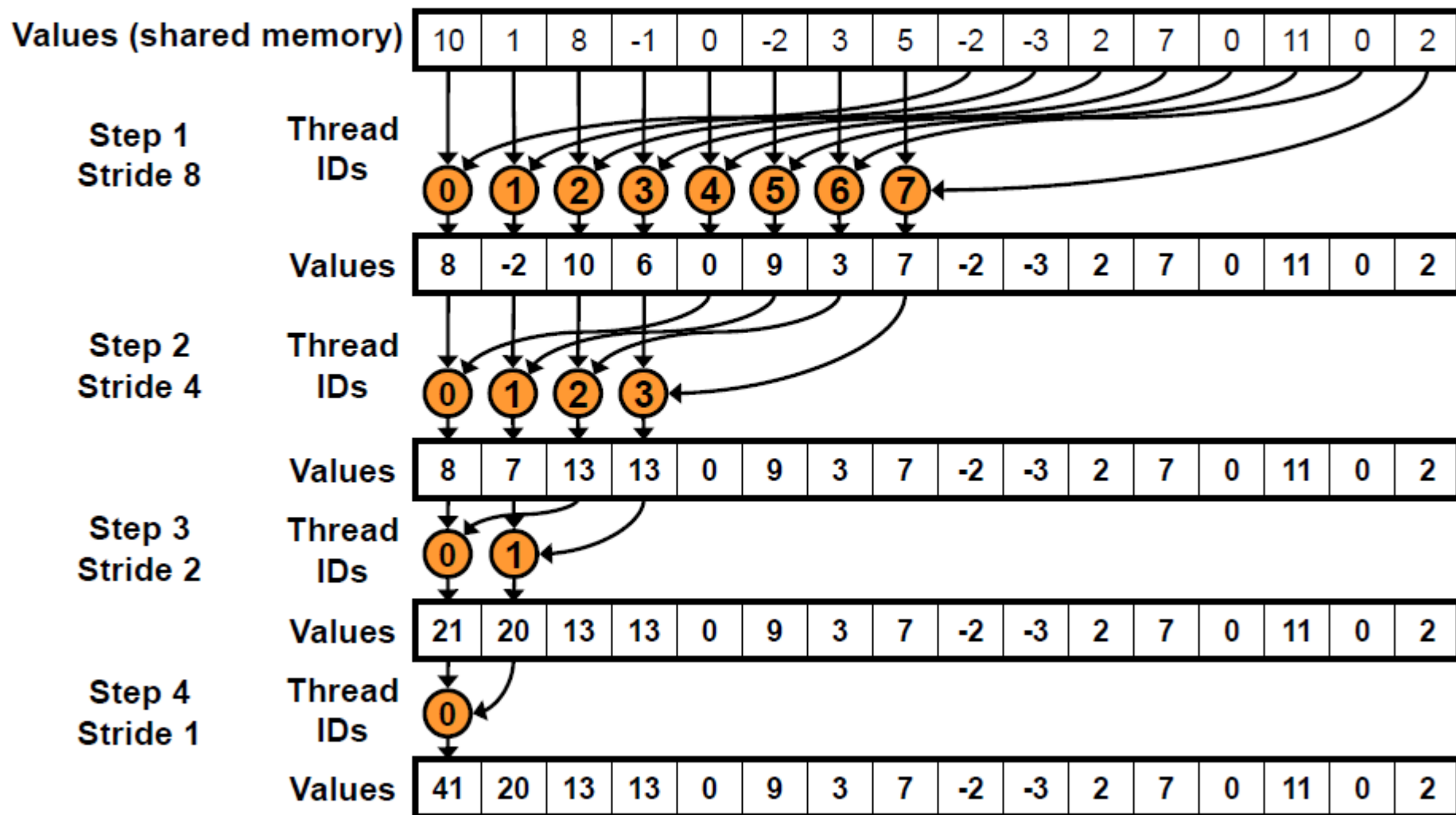
Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Parallel Reduction: Sequential Addressing

Values (shared memory) | 10 | 1 | 8 | -1 | 0 | -2 | 3 | 5 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 1 Stride 8 — **Thread IDs:** 0 1 2 3 4 5 6 7

Values | 8 | -2 | 10 | 6 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 2 Stride 4 — **Thread IDs:** 0 1 2 3

Values | 8 | 7 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 3 Stride 2 — **Thread IDs:** 0 1

Values | 21 | 20 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |

Step 4 Stride 1 — **Thread IDs:** 0

Values | 41 | 20 | 13 | 13 | 0 | 9 | 3 | 7 | -2 | -3 | 2 | 7 | 0 | 11 | 0 | 2 |
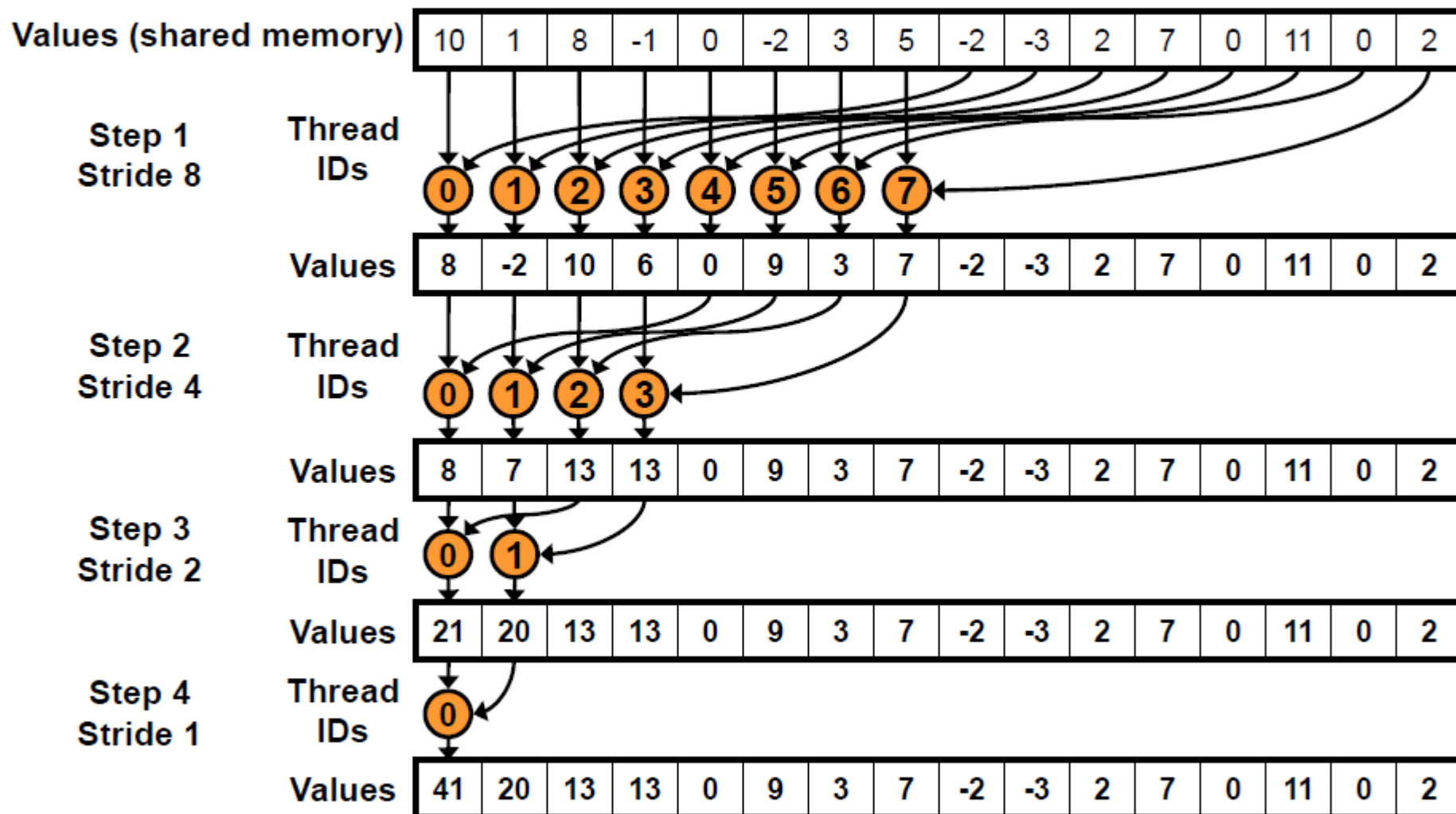
Half of the threads are idle since 1st iteration!   58

Reduction #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 level of reduction,
// reading from global memory, 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();
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Details in backup slides

Kernel 7 on 32M elements: 73 GB/s!

NTHU LSA Lab

```cpp
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

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

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

Backup

Instruction Bottleneck

- **At 17 GB/s, we're far from bandwidth bound**
    - And we know reduction has low arithmetic intensity

- **Therefore a likely bottleneck is instruction overhead**
    - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
    - In other words: address arithmetic and loop overhead

- **Strategy: unroll loops**

Unrolling the Last Warp

- **As reduction proceeds, # "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

- **Let's unroll the last 6 iterations of the inner loop**

Reduction #5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {
    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];
}
```

IMPORTANT:
For this to be correct,
we must use the
"volatile" keyword!

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

 if (tid < 32) warpReduce(sdata, tid);
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

23

Complete Unrolling

- **If we knew the number of iterations at compile time, we could completely unroll the reduction**
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes

- **So we can easily unroll for a fixed block size**
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?

- **Templates to the rescue!**
  - CUDA supports C++ template parameters on device and host functions

Unrolling with Templates

- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

Reduction #6: Completely Unrolled

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}
```

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

    if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

Invoking Template Kernels

- Don't we still need block size at compile time?
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
    {
    case 512:
        reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
        reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
        reduce5<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
        reduce5<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
        reduce5<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
        reduce5<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

27

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

Reference

- **NIVIDA Advanced CUDA Webinar Memory Optimizations**
  - http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf
- **NVIDIA CUDA C/C++ Streams and Concurrency**
  - http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf
- **Mark Harris, NVIDIA Developer Technology**
  - http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf