CS4402-9535: High-Performance Computing with CUDA

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

UWO-CS4402-CS9535

Plan

Plan

Matrix Transpose Characteristics (1/2)

- We optimize a transposition code for a matrix of floats. This operates out-of-place:
  - input and output matrices address separate memory locations.
- For simplicity, we consideran $n \times n$ matrix where 32 divides n.
- We focus on the device code:
  - the host code performs typical tasks: data allocation and transfer between host and device, the launching and timing of several kernels, result validation, and the deallocation of host and device memory.
- Benchmarks illustrate this section:
  - we compare our **matrix transpose** kernels against a **matrix copy** kernel,
  - for each kernel, we compute the **effective bandwidth**, calculated in GB/s as twice the size of the matrix (once for reading the matrix and once for writing) divided by the time of execution,
  - Each operation is run `NUM_REFS` times (for **normalizing the measurements**),
  - This looping is performed **once over the kernel** and once **within the kernel**,

Matrix Transpose Characteristics (2/2)

- We present hereafter different kernels called from the host code, each addressing different performance issues.
- All kernels in this study launch thread blocks of dimension 32x8, where each block transposes (or copies) a tile of dimension 32x32.
- As such, the parameters `TILE_DIM` and `BLOCK_ROWS` are set to 32 and 8, respectively.
- Using a thread block with fewer threads than elements in a tile is advantageous for the matrix transpose:
  - each thread transposes several matrix elements, four in our case, and much of the cost of calculating the indices is amortized over these elements.
- This study is based on a technical report by Greg Ruetsch (NVIDIA) and Paulius Micikevicius (NVIDIA).

A simple copy kernel (1/2)

```
__global__ void copy(float *odata, float* idata, int width,
                                               int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

  for (int r=0; r < nreps; r++) { // normalization outer loop
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    }
  }
}
```

A simple copy kernel (2/2)

- `odata` and `idata` are pointers to the input and output matrices,
- `width` and `height` are the matrix x and y dimensions,
- `nreps` determines how many times the loop over data movement between matrices is performed.
- In this kernel, `xIndex` and `yIndex` are global 2D matrix indices,
- used to calculate `index`, the 1D index used to access matrix elements.

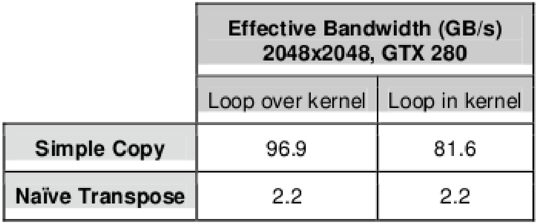
```
// removing normalization
__global__ void copy(float *odata, float* idata, int width,
                                              int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      odata[index+i*width] = idata[index+i*width];
}
```

A naive transpose kernel

```
  _global__ void transposeNaive(float *odata, float* idata,
                                int width, int height, int nreps)
{
   int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
   int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
   int index_in = xIndex + width * yIndex;
   int index_out = yIndex + height * xIndex;

     for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
       odata[index_out+i] = idata[index_in+i*width];
     }

}
```

Naive transpose kernel vs copy kernel

The performance of these two kernels on a 2048x2048 matrix using a GTX280 is given in the following table:



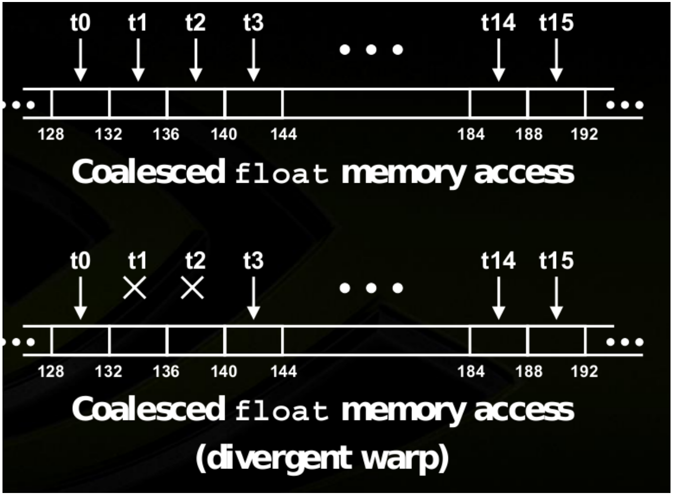
	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

The minor differences in code between the copy and naive transpose kernels have a profound effect on performance.

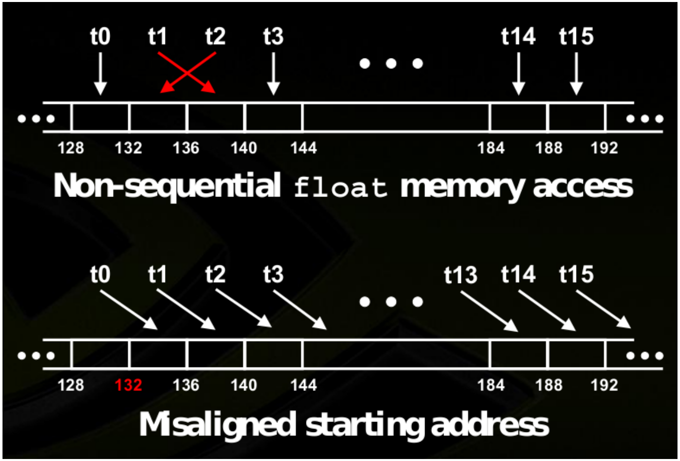
Coalesced Transpose (1/11)

- Because device memory has a much higher latency and lower bandwidth than on-chip memory, special attention must be paid to: **how global memory accesses are performed?**
- The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be **coalesced** into a single access if:
  1. The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
  2. The address of the first element is aligned to 16 times the element's size.
  3. The elements form a contiguous block of memory.
  4. The *i*-th element is accessed by the *i*-th thread in the half-warp.
- Coalescing happens even if some threads do not access memory (**divergent warp**)

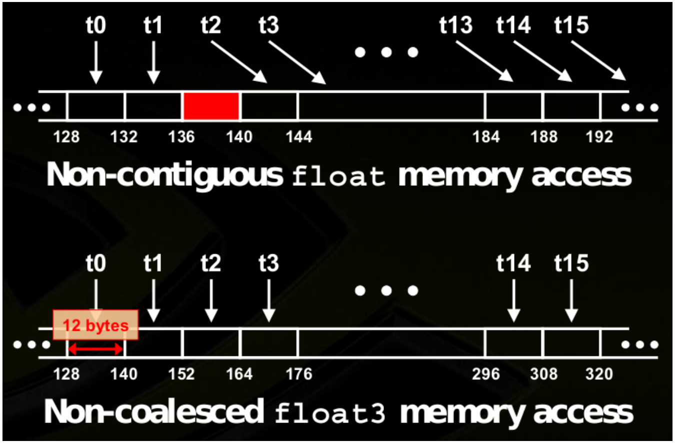
Coalesced Transpose (2/11)



Coalesced Transpose (3/11)



Coalesced Transpose (4/11)



Coalesced Transpose (5/11)

- **Allocating device memory through cudaMalloc()** and choosing `TILE_DIM` **to be a multiple of 16 ensures alignment** with a segment of memory, therefore all loads from `idata` are coalesced.
- Coalescing behavior differs between the simple copy and naive transpose kernels when writing to `odata`.
- In the case of the naive transpose, for each iteration of the `i`-loop a half warp writes one half of a column of floats to different segments of memory:
  - resulting in 16 separate memory transactions,
  - regardless of the compute capability.

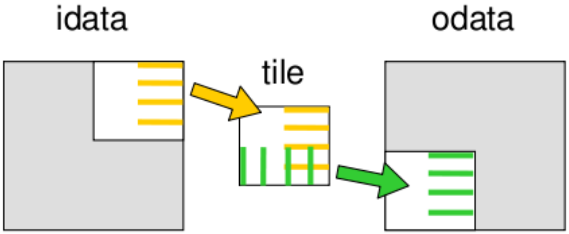
Coalesced Transpose (6/11)

- The way to avoid uncoalesced global memory access is
  1. to read the data into shared memory and,
  2. have each half warp access noncontiguous locations in shared memory in order to write contiguous data to `odata`.
- There is no performance penalty for noncontiguous access patterns in shared memory as there is in global memory.
- a `__synchthreads()` call is required to ensure that all reads from `idata` to shared memory have completed before writes from shared memory to `odata` commence.

Coalesced Transpose (7/11)

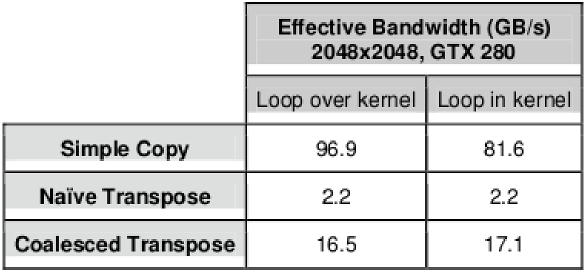
```
__global__ void transposeCoalesced(float *odata,
            float *idata, int width, int height) // no nreps param
{
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
  }   __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
} }
```

Coalesced Transpose (8/11)



1. The half warp writes four half rows of the idata matrix tile to the shared memory 32x32 array `tile` indicated by the yellow line segments.
2. After a `__syncthreads()` call to ensure all writes to tile are completed,
3. the half warp writes four half columns of tile to four half rows of an odata matrix tile, indicated by the green line segments.

Coalesced Transpose (9/11)



	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

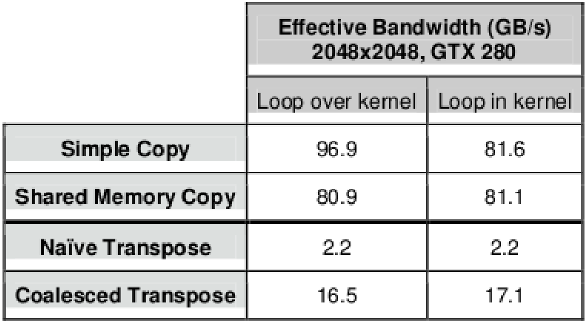
While there is a dramatic increase in effective bandwidth of the coalesced transpose over the naive transpose, there still remains a large performance gap between the coalesced transpose and the copy:

- One possible cause of this performance gap could be the synchronization barrier required in the coalesced transpose.
- This can be easily assessed using the following copy kernel which utilizes shared memory and contains a `__syncthreads()` call.

Coalesced Transpose (10/11)

```
  _global__ void copySharedMem(float *odata, float *idata,
                              int width, int height) // no nreps param
{
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index = xIndex + width*yIndex;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index+i*width];
  }
  __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] =
        tile[threadIdx.y+i][threadIdx.x];
} }
```

Coalesced Transpose (11/11)



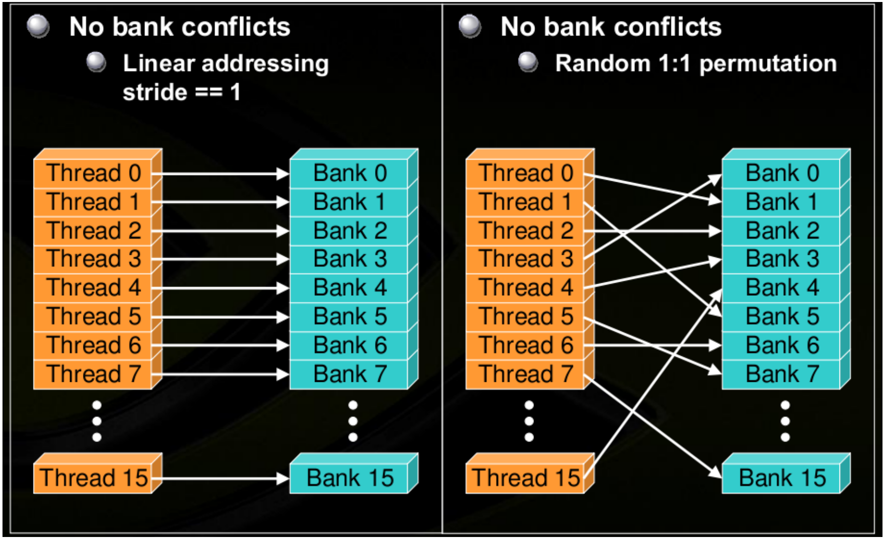
	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

The shared memory copy results seem to suggest that the use of shared memory with a synchronization barrier has little effect on the performance, certainly as far as the *Loop in kernel* column indicates when comparing the simple copy and shared memory copy.

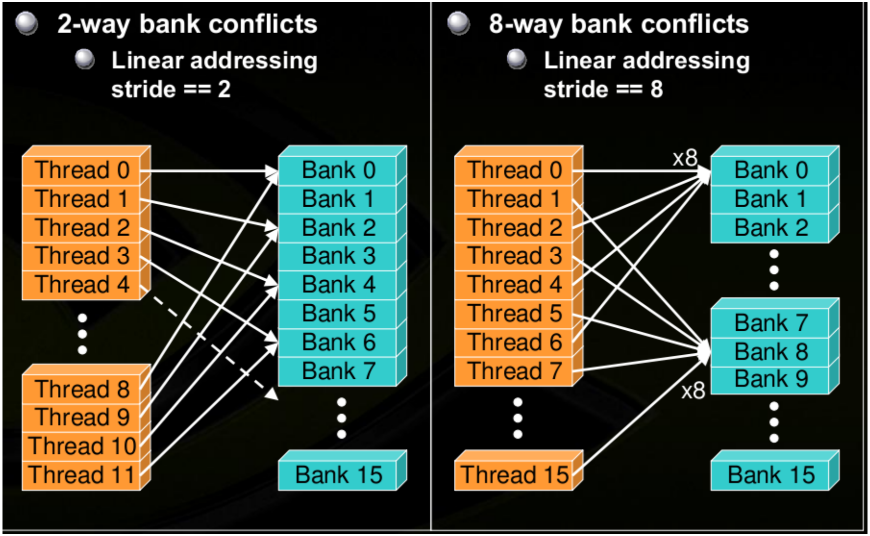
Shared memory bank conflicts (1/6)

1. Shared memory is divided into 16 equally-sized memory modules, called **banks**, which are organized such that successive 32-bit words are assigned to successive banks.
2. These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory the **threads in a half warp should access shared memory associated with different banks.**
3. The **exception to this rule is** when all threads in a half warp read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads of the half warp in one transaction.
4. One can use the `warp_serialize` flag when profiling CUDA applications to determine whether shared memory bank conflicts occur in any kernel.

Shared memory bank conflicts (2/6)



Shared memory bank conflicts (3/6)



Shared memory bank conflicts (4/6)

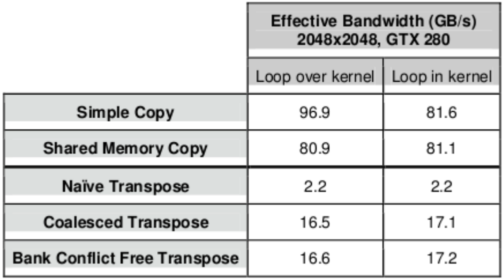
1. The coalesced transpose uses a 32×32 shared memory array of floats.
2. For this sized array, all data in columns `k` and `k+16` are mapped to the same bank.
3. As a result, when writing partial columns from `tile` in shared memory to rows in `odata` the half warp experiences a 16-way bank conflict and serializes the request.
4. A simple way to avoid this conflict is to pad the shared memory array by one column:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

Shared memory bank conflicts (5/6)

- The padding does not affect shared memory bank access pattern when writing a half warp to shared memory, which remains conflict free,
- but by adding a single column now the access of a half warp of data in a column is also conflict free.
- The performance of the kernel, now coalesced and memory bank conflict free, is added to our table on the next slide.

Shared memory bank conflicts (6/6)



	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2

- While padding the shared memory array did eliminate shared memory bank conflicts, as was confirmed by checking the `warp_serialize` flag with the CUDA profiler, it has little effect (when implemented at this stage) on performance.
- As a result, there is still a large performance gap between the coalesced and shared memory bank conflict free transpose and the shared memory copy.

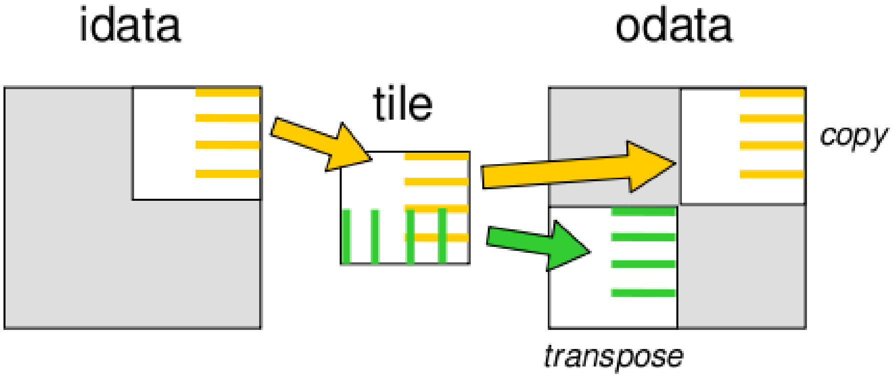
Decomposing Transpose (1/6)

- To investigate further, we revisit the data flow for the transpose and compare it to that of the copy.
- There are essentially two differences between the copy code and the transpose:
  - transposing the data within a tile, and
  - writing data to transposed tile.
- We can isolate the performance between each of these two components by implementing two kernels that individually perform just one of these components:

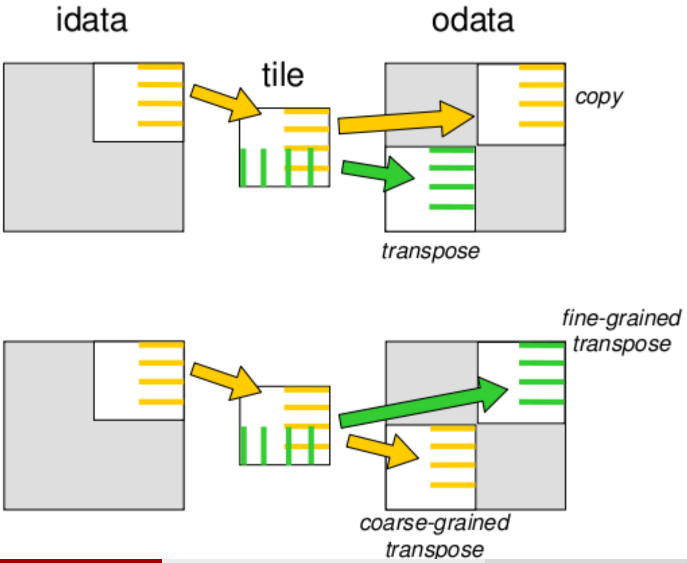
fine-grained transpose: this kernel transposes the data within a tile, but writes the tile to the location.

coarse-grained transpose: this kernel writes the tile to the transposed location in the `odata` matrix, but does not transpose the data within the tile.

Decomposing Transpose (2/6)



Decomposing Transpose (3/6)



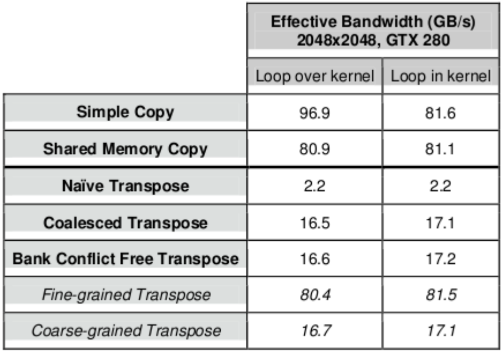
Decomposing Transpose (4/6)

```
  _global__ void transposeFineGrained(float *odata,
            float *idata, int width, int height)
{
   __shared__ float block[TILE_DIM][TILE_DIM+1];
   int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
   int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
   int index = xIndex + (yIndex)*width;
     for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
       block[threadIdx.y+i][threadIdx.x] =
         idata[index+i*width];
     }
     __syncthreads();
     for (int i=0; i < TILE_DIM; i += BLOCK_ROWS) {
       odata[index+i*height] =
         block[threadIdx.x][threadIdx.y+i];
     }
}
```

Decomposing Transpose (5/6)

```
__global__ void transposeCoarseGrained(float *odata,
      float *idata, int width, int height)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      block[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }  syncthreads();
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      odata[index_out+i*height] =
        block[threadIdx.y+i][threadIdx.x];
}  }
```

Decomposing Transpose (6/6)



	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*

The fine-grained transpose has performance similar to the shared memory copy, whereas the coarse-grained transpose has roughly the performance of the coalesced transpose. Thus the performance bottleneck lies in writing data to the transposed location in global memory.

Partition Camping (1/4)

- Just as shared memory performance can be degraded via bank conflicts, an analogous performance degradation can occur with global memory access through **partition camping.**
- Global memory is divided into either 6 **partitions** (on 8- and 9-series GPUs) or 8 partitions (on 200-and 10-series GPUs) of 256-byte width.
- To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions.
- **partition camping** occurs when:
  - global memory accesses are directed through a subset of partitions,
  - causing requests to queue up at some partitions while other partitions go unused.

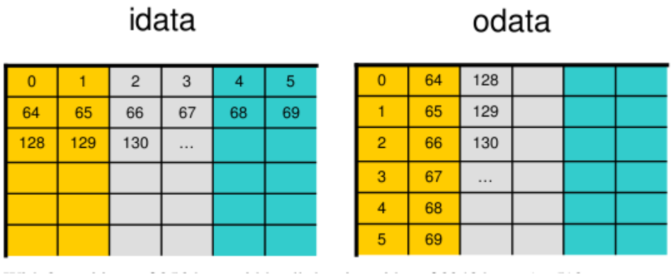
Partition Camping (2/4)

- Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important.
- When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as:

  ```
  bid = blockIdx.x + gridDim.x*blockIdx.y;
  ```

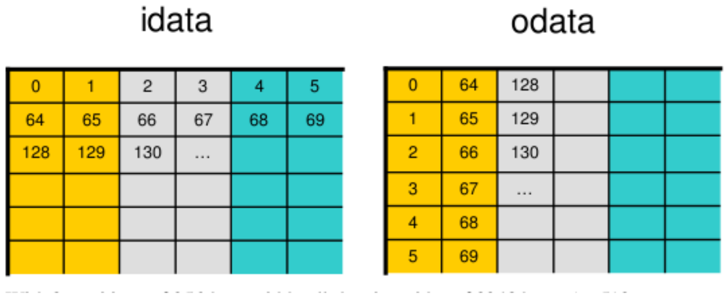
  which is a row-major ordering of the blocks in the grid.
- Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed.
- How quickly and the order in which blocks complete cannot be determined.
- So active blocks are initially contiguous but become less contiguous as execution of the kernel progresses.

Partition Camping (3/4)



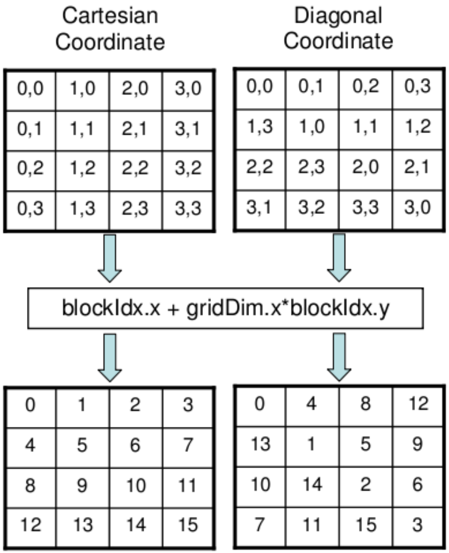
- With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition.
- Any float matrix with $512 \times k$ columns, such as our 2048x2048 matrix, will contain columns whose elements map to a single partition.
- With tiles of 32×32 floats whose one-dimensional block IDs are shown in the figures, the mapping of `idata` and `odata` onto the partitions is depectide below.

Partition Camping (4/4)



- Cconcurrent blocks will be accessing tiles row-wise in `idata` which will be roughly equally distributed amongst partitions
- However these blocks will access tiles column-wise in `odata` which will typically access global memory through just a few partitions.
- Just as with shared memory, padding would be an option (potentially expensive) but there is a better one . . .

Diagonal block reordering (1/7)



Cartesian Coordinate:

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Diagonal Coordinate:

0,0	0,1	0,2	0,3
1,3	1,0	1,1	1,2
2,2	2,3	2,0	2,1
3,1	3,2	3,3	3,0

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

Diagonal block reordering (2/7)

- The key idea is to view the grid under a **diagonal coordinate system**.
- If `blockIdx.x` and `blockIdx.y` represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by the following mapping:

  ```
  blockIdx_y = blockIdx.x;
  blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
  ```
- One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of `blockIdx` fields, except using `blockIdx_x` and `blockIdx_y` in place of `blockIdx.x` and `blockIdx.y`, respectively, throughout the kernel.
- This is precisely what is done in the `transposeDiagonal` kernel hereafter.

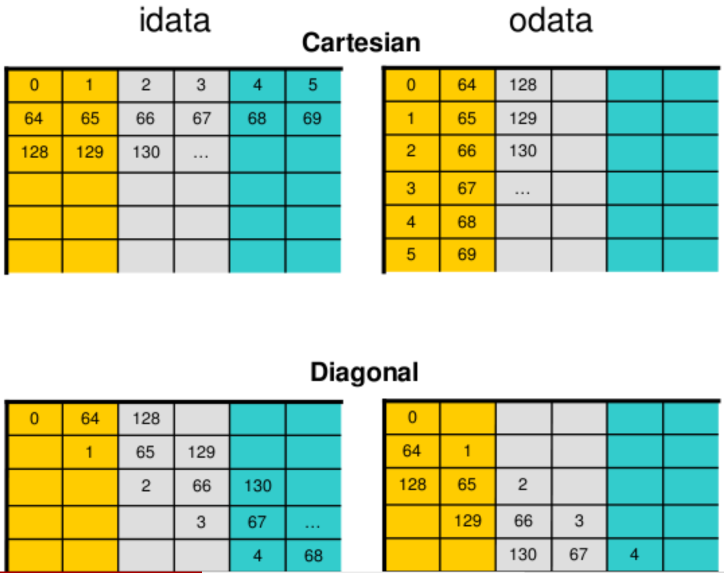
Decomposing Transpose (3/7)

```
__global__ void transposeDiagonal(float *odata,
            float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];
  int blockIdx_x, blockIdx_y;
  // diagonal reordering
  if (width == height) {
    blockIdx_y = blockIdx.x;
    blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
  } else {
    int bid = blockIdx.x + gridDim.x*blockIdx.y;
    blockIdx_y = bid%gridDim.y;
    blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
  }
```

Decomposing Transpose (4/7)

```
  int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
  yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }
    __syncthreads();
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
    }
}
```

Diagonal block reordering (5/7)



Cartesian — idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	...		

Cartesian — odata

0	64	128			
1	65	129			
2	66	130			
3	67	...			
4	68				
5	69				

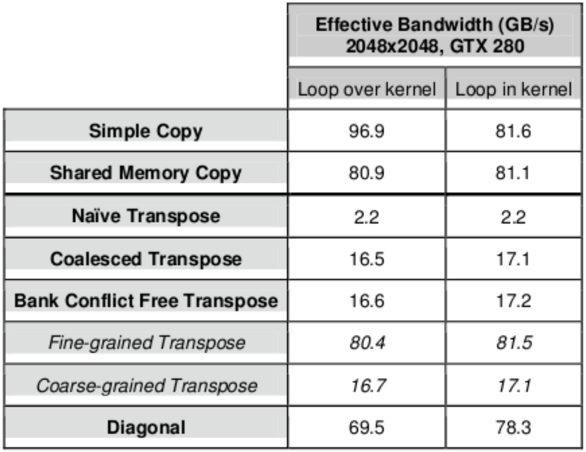
Diagonal — idata

0	64	128			
	1	65	129		
		2	66	130	
			3	67	...
				4	68

Diagonal — odata

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	

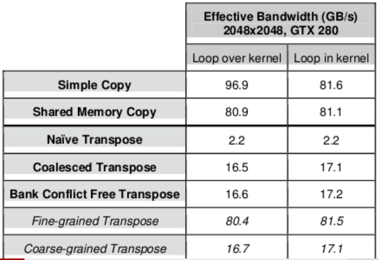
Diagonal block reordering (6/7)



	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*
Diagonal	69.5	78.3

Diagonal block reordering (7/7)

- The bandwidth measured when looping within the kernel over the read and writes to global memory is within a few percent of the shared memory copy.
- When looping over the kernel, the performance degrades slightly, likely due to additional computation involved in calculating `blockIdx_x` and `blockIdx_y`. However, even with this performance degradation the diagonal transpose has over four times the bandwidth of the other complete transposes.



	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*

Plan

Four principles

- Expose as much parallelism as possible
- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency
- Optimize instruction usage for maximum throughput

Expose Parallelism

- Structure algorithm to maximize independent parallelism
- If threads of same block need to communicate, use shared memory and `__syncthreads()`
- If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
- Recall that there is no synchronization mechanism between blocks
- High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Take advantage of asynchronous kernel launches by overlapping CPU computations with kernel execution.

Optimize Memory Usage: Basic Strategies

- Processing data is cheaper than moving it around:
  - Especially for GPUs as they devote many more transistors to ALUs than memory
- Basic strategies:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Write kernels with high arithmetic intensity (ratio of arithmetic operations to memory transactions)
  - Sometimes recompute data rather than cache it

Minimize CPU $< - >$ GPU Data Transfers

- CPU $< - >$ GPU memory bandwidth much lower than GPU memory bandwidth
- Minimize CPU $< - >$ GPU data transfers by moving more code from CPU to GPU
    - Even if sometimes that means running kernels with low parallelism computations
    - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory
- Group data transfers: One large transfer much better than many small ones.

Optimize Memory Access Patterns

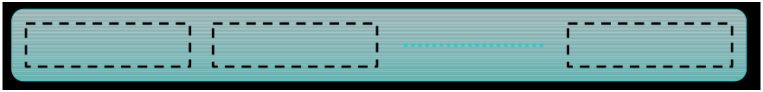
- Effective bandwidth can vary by an order of magnitude depending on access pattern:
  - Global memory is not cached on `G8x`.
  - Global memory has High latency instructions: 400-600 clock cycles
  - Shared memory has low latency: a few clock cycles
- Optimize access patterns to get:
  - Coalesced global memory accesses
  - Shared memory accesses with no or few bank conflicts and
  - to avoid partition camping.

A Common Programming Strategy

1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
4. Perform the computation on the subset from shared memory.
5. Copy the result from shared memory back to global memory.

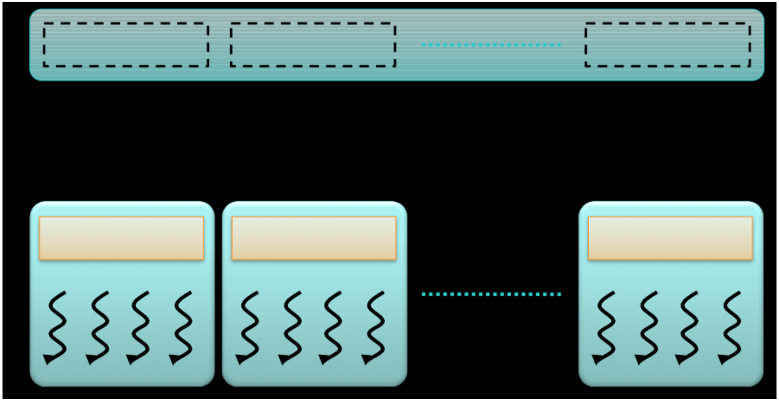
A Common Programming Strategy

Partition data into subsets that fit into shared memory



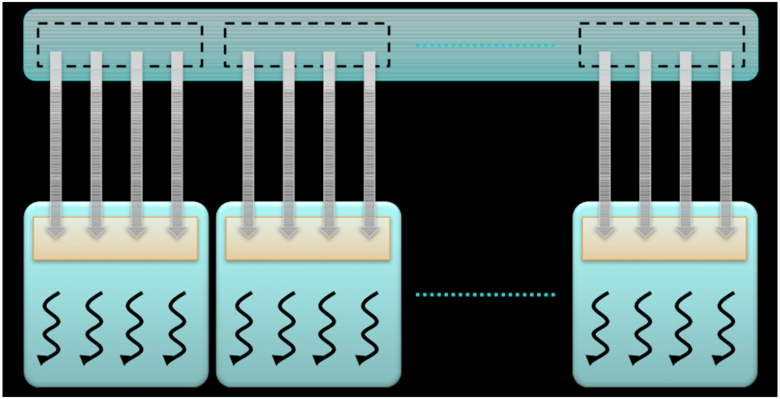
A Common Programming Strategy

Handle each data subset with one thread block



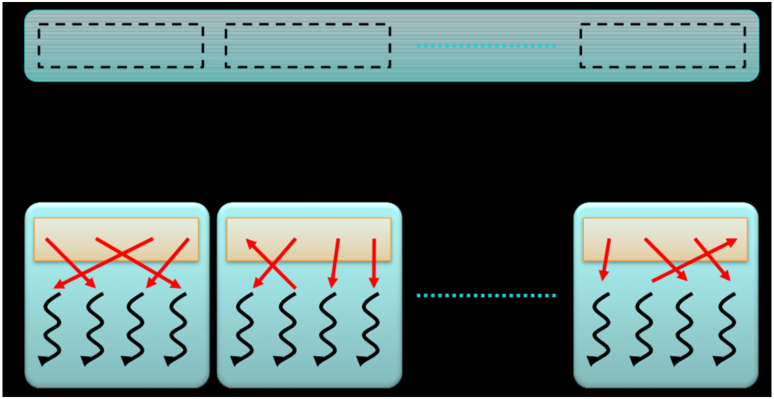
A Common Programming Strategy

Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.



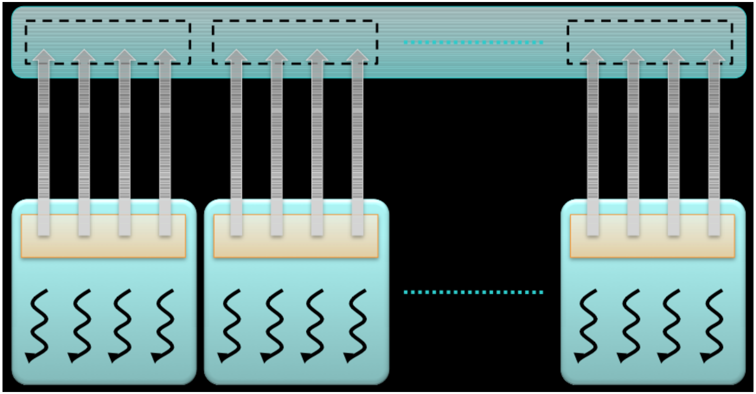
A Common Programming Strategy

Perform the computation on the subset from shared memory.



A Common Programming Strategy

Copy the result from shared memory back to global memory.

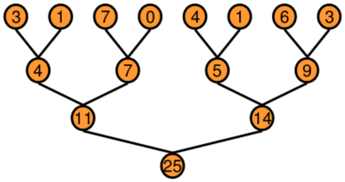


A Common Programming Strategy

- Carefully partition data according to access patterns
- If read only, use `__constant__` memory (fast)
- for read/write access within a tile, use `__shared__` memory (fast)
- for read/write scalar access within a thread, use registers (fast)
- R/W inputs/results cudaMalloc'ed, use global memory (slow)

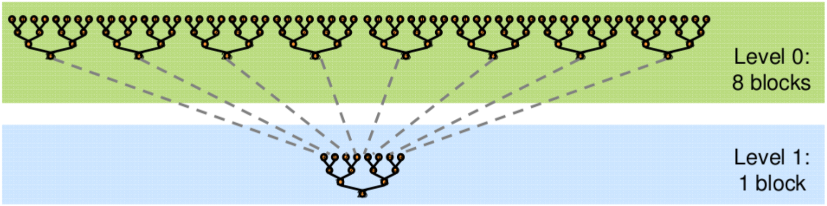
Plan

Parallel reduction: presentation



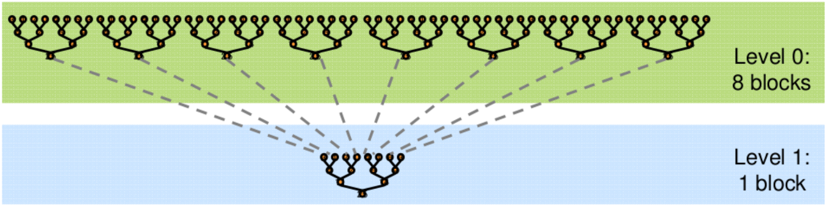
- Common and important data parallel primitive.
- Easy to implement in CUDA, but hard to get right.
- Serves as a great optimization example.
- This section is based on slides and technical reports by Mark Harris (NVIDIA).

Parallel reduction: challenges



- One needs to be able to use multiple thread blocks:
  - to process very large arrays,
  - to keep all multiprocessors on the GPU busy,
  - to have each thread block reducing a portion of the array.
- But how do we communicate partial results between thread blocks?

Parallel reduction: CUDA implementation strategy



- We decompose computation into multiple kernel invocations
- For this problem of parallel reduction, all kernels are in fact the same code.

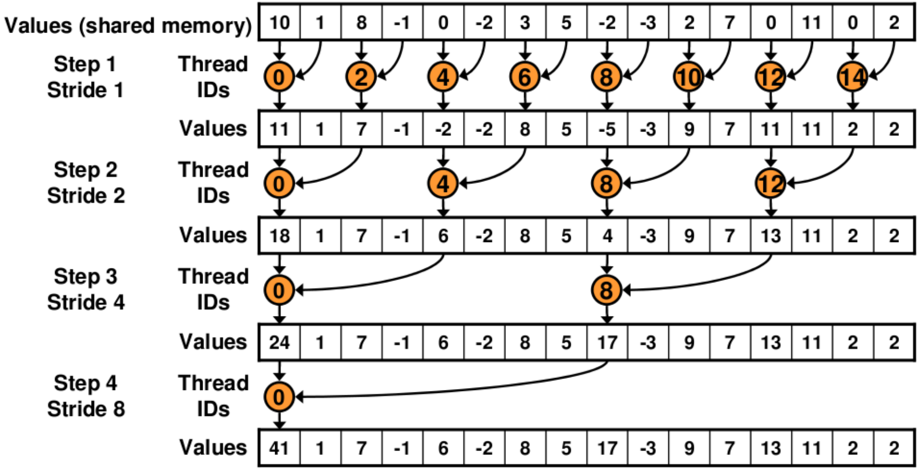
Parallel reduction: what is our goal?

- We should use the right metric between:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity:
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- We will use G80 GPU (following Mark Harris tech report) for this example:
  - 384-bit memory interface, 1800 MHz
  - $384 \times 1800/8 = 86.4 GB/s$

Parallel reduction: interleaved addressing (1/2)

```
__global__ void reduce0(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];
}
```

Parallel reduction: interleaved addressing (2/2)



Parallel reduction: branch divergence in interleaved addressing (1/2)

- Main performance concern with branching is **divergence.**
  - Branch divergence occurs when threads in the same warp take different paths upon a conditional branch.
  - **Penalty:** different execution paths are likely to serialized (at compile time).
- One should be careful branching when branch condition is a function of thread ID.
  - Below, branch granularity is less than warp size:

    ```
    If (threadIdx.x > 2) { }
    ```

  - Below, branch granularity is a whole multiple of warp size:

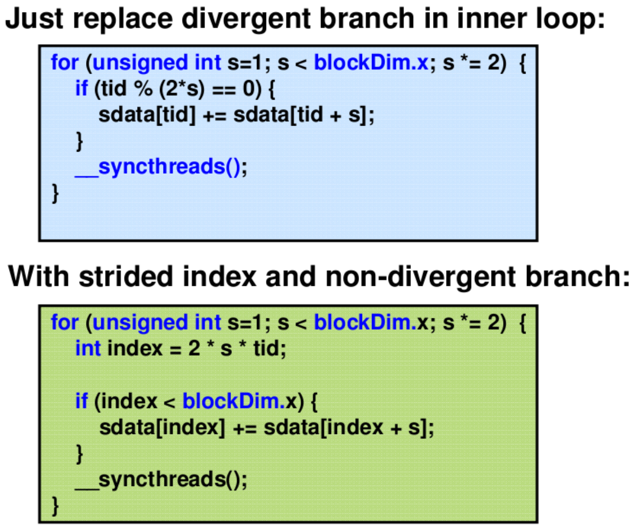
    ```
    If (threadIdx.x / WARP_SIZE > 2) { }
    ```

Parallel reduction: branch divergence in interleaved addressing (2/2)

```
__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();
  }
```

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

Parallel reduction: non-divergent interleaved addressing



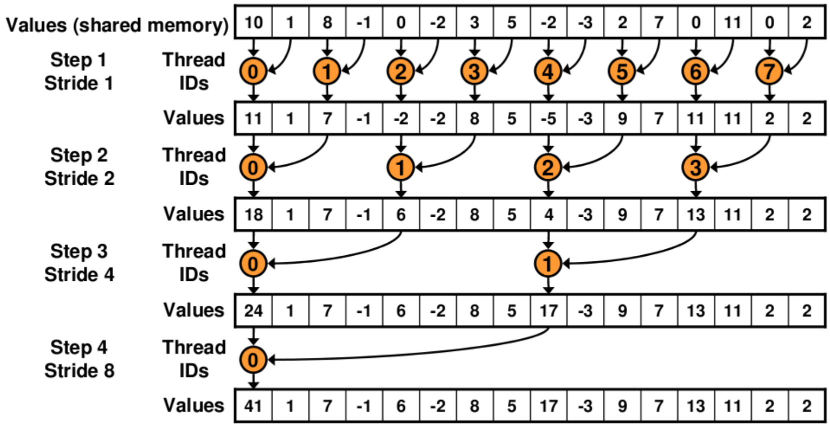
Just replace divergent branch in inner loop:

```
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();
}
```

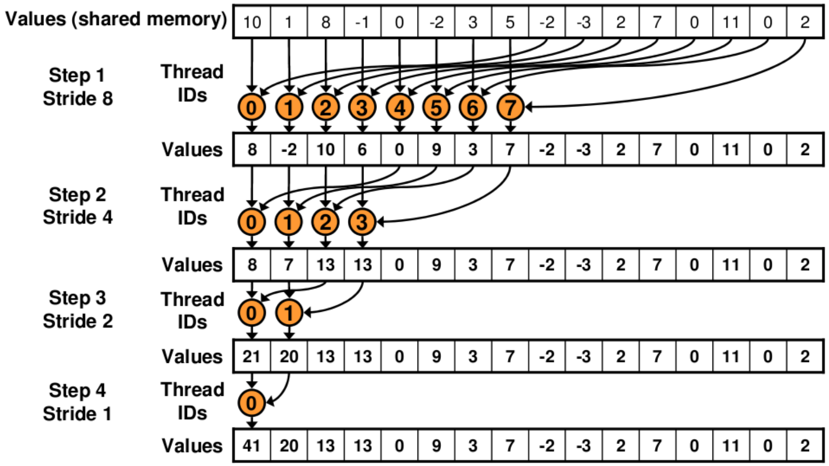
Parallel reduction: shared memory bank conflicts



New Problem: Shared Memory Bank Conflicts

12

Parallel reduction: sequential addressing (1/2)



Sequential addressing is conflict free

14

Parallel reduction: sequential addressing (2/2)

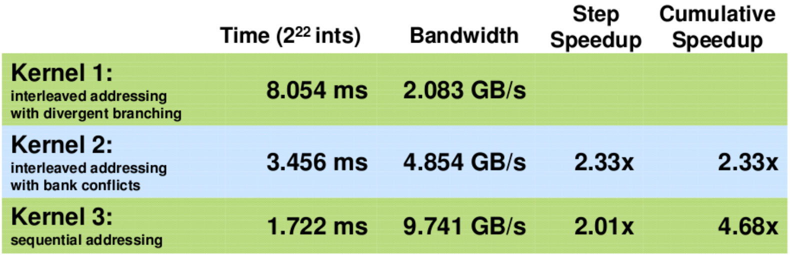
Just replace strided indexing in inner loop:

```
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();
}
```

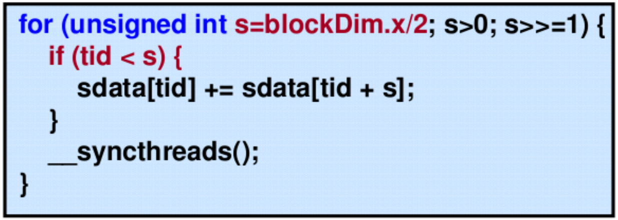
Parallel reduction: performance for 4Mb element reduction



	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: idle threads (1/2)

Problem:



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

Half of the threads are idle on first loop iteration!

This is wasteful...

Parallel reduction: idle threads (2/2)

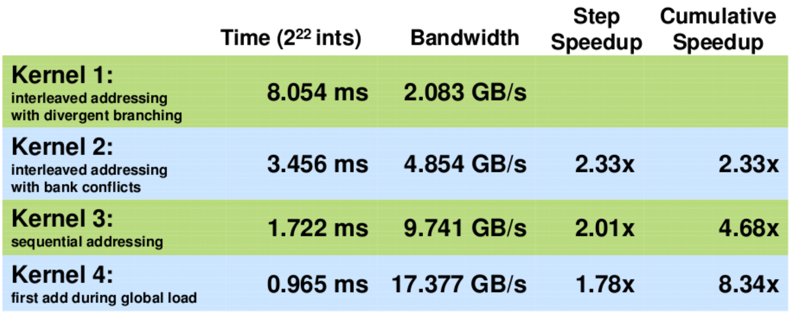
Halve the number of blocks, and replace single load:

```
// 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();
```

Parallel reduction: instruction bottlenecks (1/2)



	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

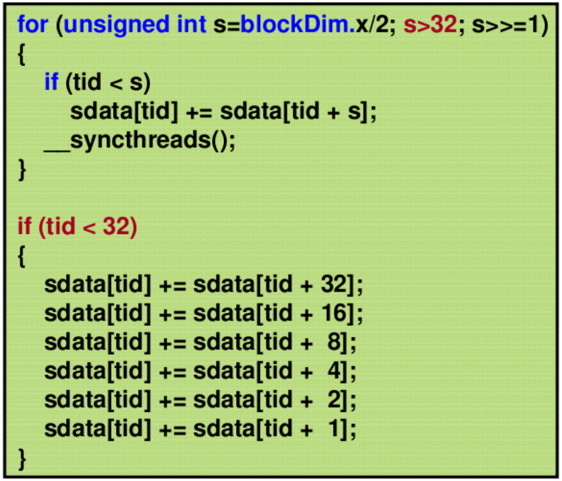
Parallel reduction: instruction bottlenecks (2/2)

- 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:
    - auxiliary instructions that are not loads, stores, or arithmetic for the core computation,
    - in other words: address arithmetic and loop overhead.
- **Strategy: unroll loops**.

Parallel reduction: unrolling the last warp (1/3)

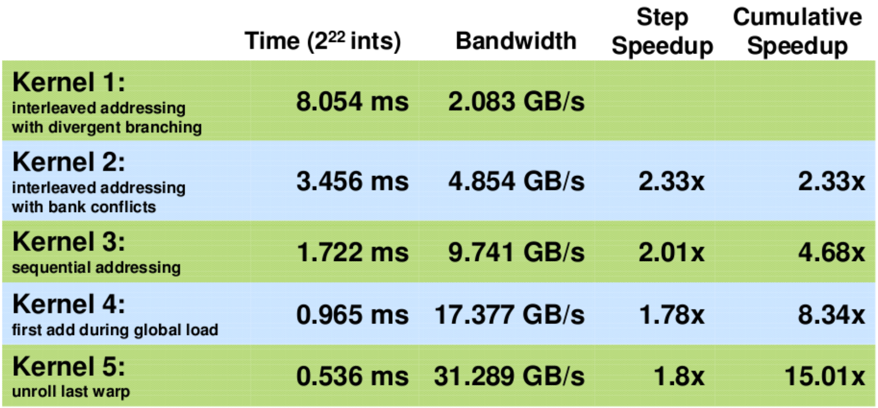
- As reduction proceeds, the number of active threads decreases;
  - When $s \leq 32$, we have only one warp left.
- Instructions are SIMD synchronous within a warp
- That implies when $s \leq 32$:
  - We do not need to use `__syncthreads()`
  - We do not need to perform the test `if (tid < s)` because it doesn't save any work.
- **Let's unroll the last 6 iterations of the inner loop!**

Parallel reduction: unrolling the last warp (2/3)



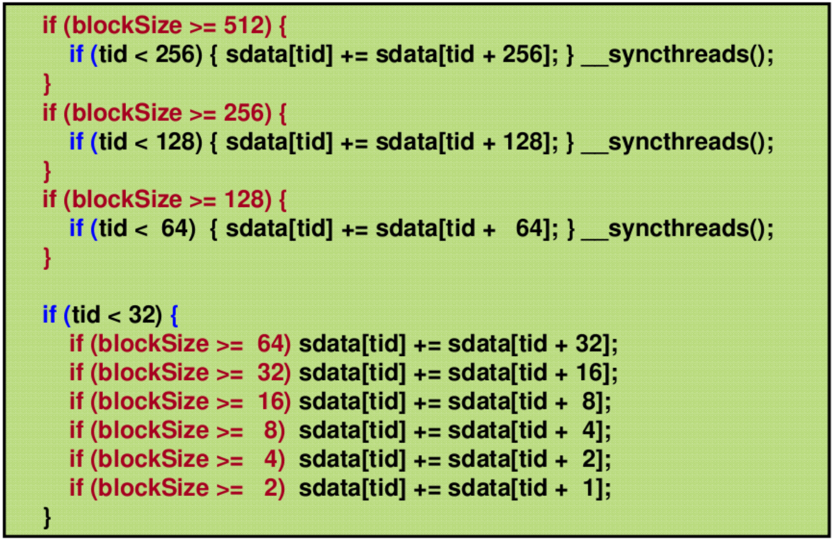
```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

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

Parallel reduction: unrolling the last warp (3/3)



	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

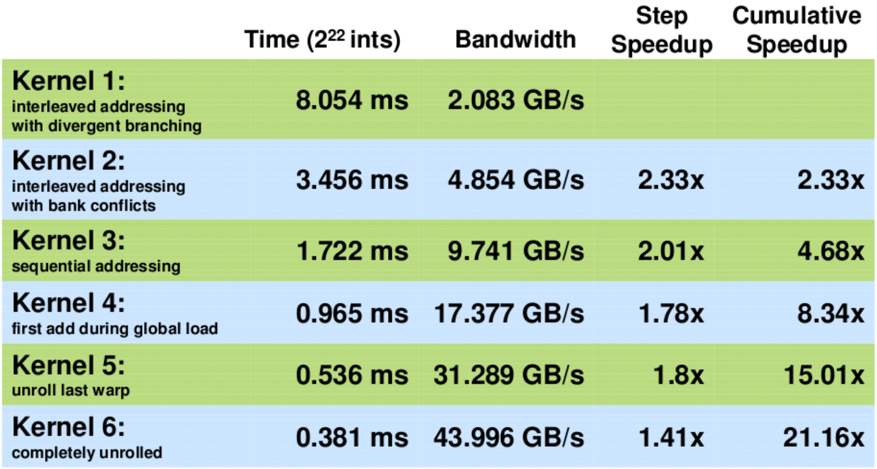
Parallel reduction: complete unrolling (1/2)



```
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) {
    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];
}
```

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

Parallel reduction: complete unrolling (2/2)



	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

Parallel reduction: coarsening the base case (1/6)

- The work and span of the whole reduction process are $\Theta(n)$ and $\Theta(\log(n))$, respectively.
- If we allocate $\Theta(n)$ threads (for each kernel call) we necessarily do $\Theta(n\log(n))$ work in total, that is, a significant overhead factor.
- Therefore, we need to allocate $\Theta(n/\log(n))$ threads, with each thread doing $\Theta(\log(n))$ work.
- On G80, best perf with 64-256 blocks of 128 threads with 1024-4096 elements per thread.

Parallel reduction: coarsening the base case (2/6)

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*(blockSize*2) + threadIdx.x;
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();
```

Parallel reduction: coarsening the base case (3/6)

Replace load and add of two elements:

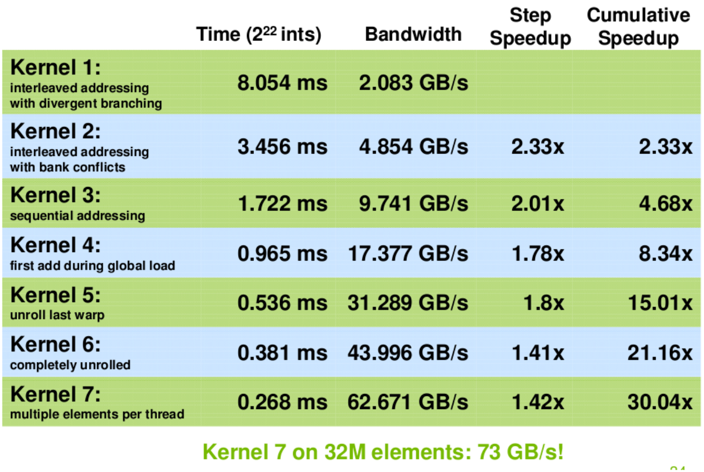
```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = th
unsigned int i = blo
unsigned int gridSi
sdata[tid] = 0;

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

Note: gridSize loop stride to maintain coalescing!

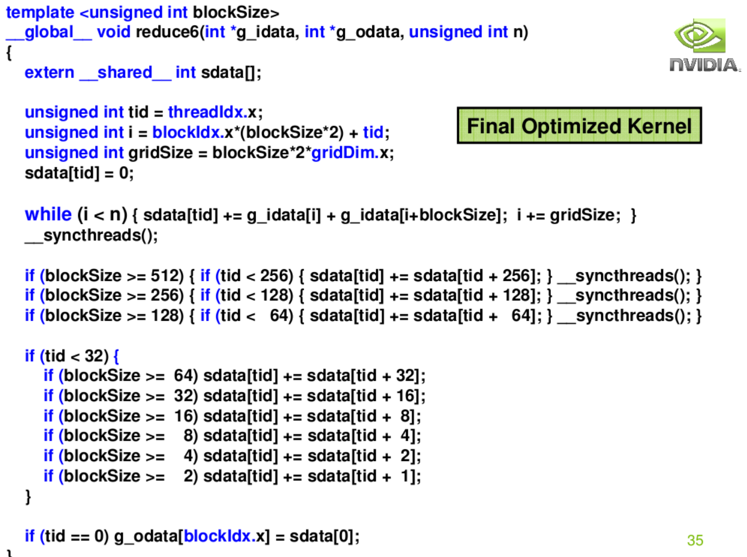
Parallel reduction: coarsening the base case (4/6)



	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

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

Parallel reduction: coarsening the base case (5/6)

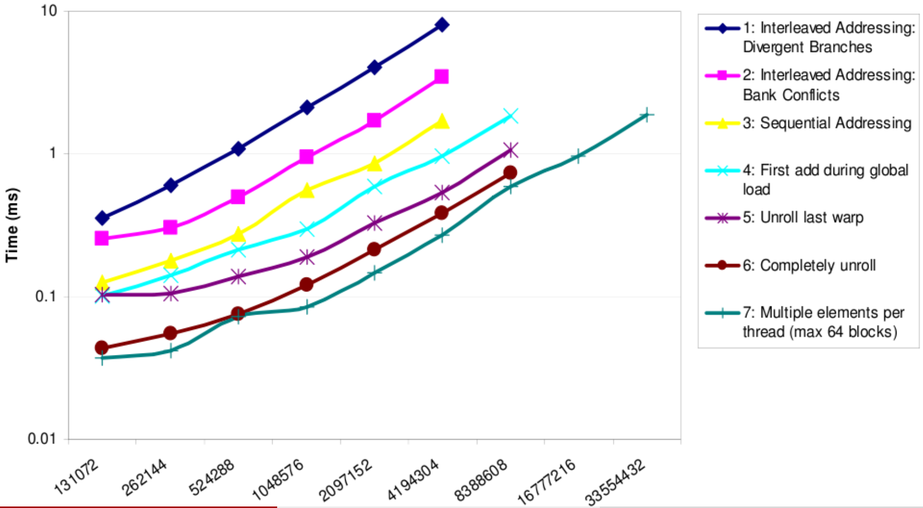


Final Optimized Kernel

```
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) {
      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 (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

35

Parallel reduction: coarsening the base case (6/6)



Plan

Parallel scan: presentation

- Another common and important data parallel primitive.
- This problem seems inherently sequential, but there is an efficient parallel algorithm.
- Applications: sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, building histograms and data structures (graphs, trees, etc.) in parallel.

Parallel scan: definitions

- Let S be a set, let $+ : S \times S \to S$ be an associative operation on S with 0 as identity. Let $A[0 \cdots n-1]$ be an array of n elements of S.
- Tthe *all-prefixes-sum* or *inclusive scan* of A computes the array B of n elements of S defined by

$$B[i] = \begin{cases} A[0] & \text{if} \quad i = 0 \\ B[i-1] + A[i] & \text{if} \quad 0 < i < n \end{cases}$$

- The *exclusive scan* of A computes the array B of n elements of S:

$$C[i] = \begin{cases} 0 & \text{if} \quad i = 0 \\ C[i-1] + A[i-1] & \text{if} \quad 0 < i < n \end{cases}$$

- An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity.
- Similarly, an inclusive scan can be generated from an exclusive scan.
- We shall focus on exclusive scan.

Parallel scan: sequential algorithm

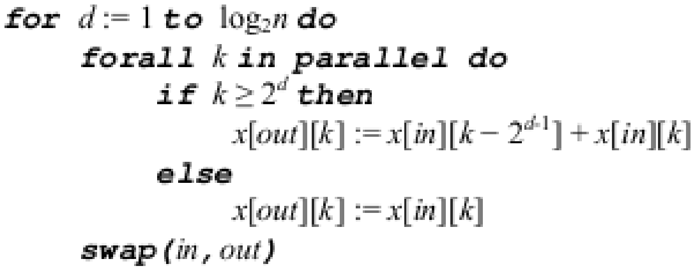
```
void scan( float* output, float* input, int length)
{
    output[0] = 0; // since this is a prescan, not a scan
    for(int j = 1; j < length; ++j)
    {
        output[j] = input[j-1] + output[j-1];
    }
}
```

Parallel scan: naive parallel algorithm (1/4)

```
for $d := 1$ to $\log_2 n$ do
    forall $k$ in parallel do
        if $k \geq 2^d$ then $x[k] := x[k - 2^{d-1}] + x[k]$
```

- This algorithm is not work-efficient since its work is $O(n\log_2(n))$. We will fix this issue later.
- In addition is not suitable for a CUDA implementation either. Indeed, it works in place which is not feasible for a sufficiently large array requiring several thread blocks

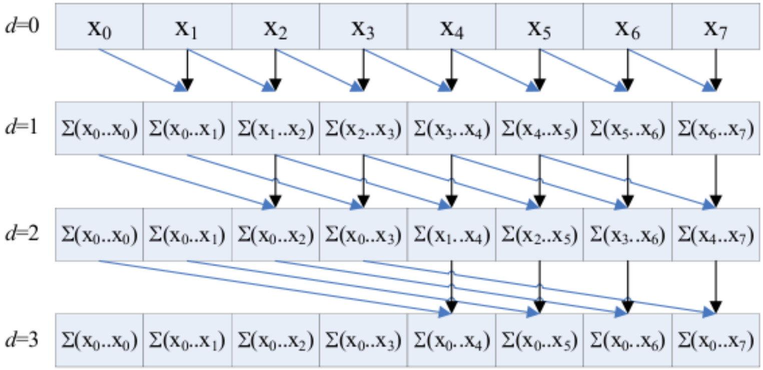
Parallel scan: naive parallel algorithm (2/4)



```
for $d := 1$ to $\log_2 n$ do
    forall $k$ in parallel do
        if $k \geq 2^d$ then
            $x[out][k] := x[in][k - 2^{d-1}] + x[in][k]$
        else
            $x[out][k] := x[in][k]$
    swap($in$, $out$)
```

In order to realize CUDA implementation potentially using many thread blocks, one needs to use a double-buffer.

Parallel scan: naive parallel algorithm (3/4)

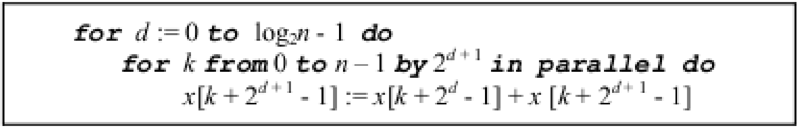

d=0
x0 x1 x2 x3 x4 x5 x6 x7
d=1
Σ(x0..x0) Σ(x0..x1) Σ(x1..x2) Σ(x2..x3) Σ(x3..x4) Σ(x4..x5) Σ(x5..x6) Σ(x6..x7)
d=2
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x1..x4) Σ(x2..x5) Σ(x3..x6) Σ(x4..x7)
d=3
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x0..x4) Σ(x0..x5) Σ(x0..x6) Σ(x0..x7)

Computing a scan of an array of 8 elements using the nave scan algorithm. The CUDA version (next slide) can handle arrays only as large as can be processed by a single thread block running on 1 GPU multiprocessor.

Parallel scan: naive parallel algorithm (4/4)

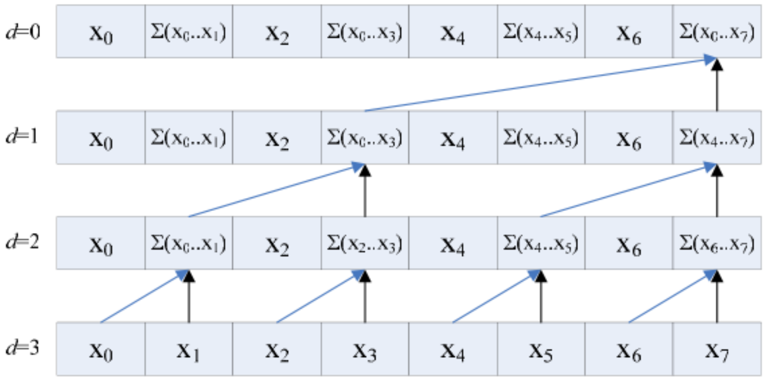
```
__global__ void scan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // This is exclusive scan, so shift right by one and set first elt to 0
    temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();

    for (int offset = 1; offset < n; offset *= 2)
    {
        pout = 1 - pout; // swap double buffer indices
        pin  = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads();
    }

    g_odata[thid] = temp[pout*n+thid1]; // write output
}
```

Parallel scan: work-efficient parallel algorithm (1/6)



```
for $d := 0$ to $\log_2 n - 1$ do
    for $k$ from $0$ to $n-1$ by $2^{d+1}$ in parallel do
        $x[k + 2^{d+1} - 1] := x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
```

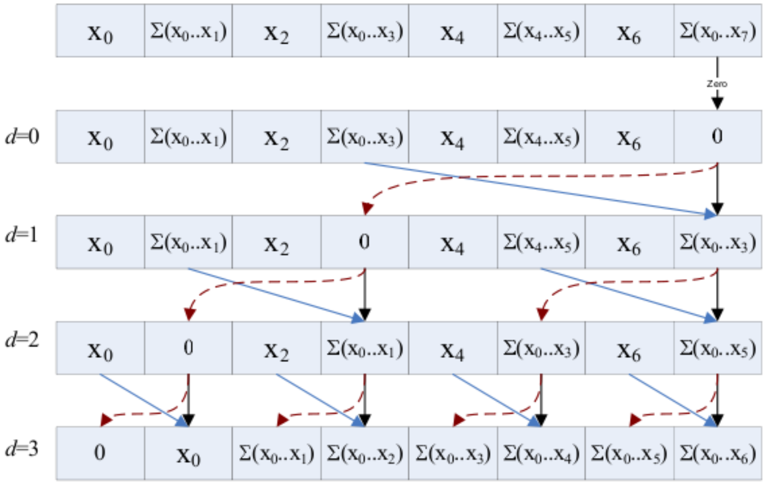
Parallel scan: work-efficient parallel algorithm (2/6)



Parallel scan: work-efficient parallel algorithm (3/6)

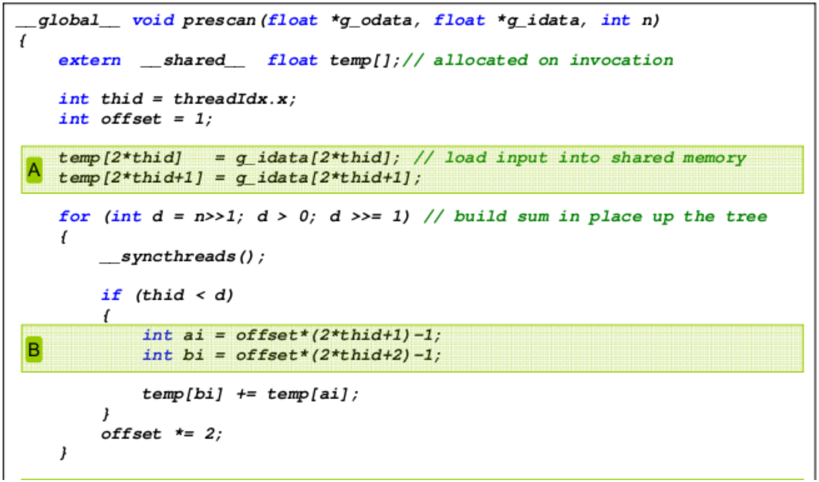
```
x[n-1] := 0;
for i := log(n) downto 1 do
    for k from 0 to n-1 by 2^(2*d) in parallel do {
        t := x[k + 2^d -1];
        x[k + 2^d -1] := x[k + 2^(d-1) -1];
        x[k + 2^(d-1) -1] := t + x[k + 2^(d-1) -1];
    }
```

Parallel scan: work-efficient parallel algorithm (4/6)



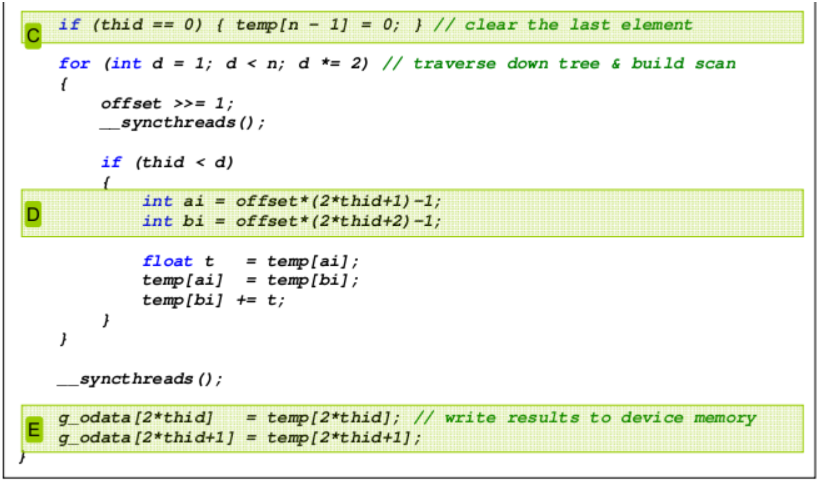
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$
Zero
d=0
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | 0
d=1
x_0 | $\Sigma(x_0..x_1)$ | x_2 | 0 | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_3)$
d=2
x_0 | 0 | x_2 | $\Sigma(x_0..x_1)$ | x_4 | $\Sigma(x_0..x_3)$ | x_6 | $\Sigma(x_0..x_5)$
d=3
0 | x_0 | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$

Parallel scan: work-efficient parallel algorithm (5/6)



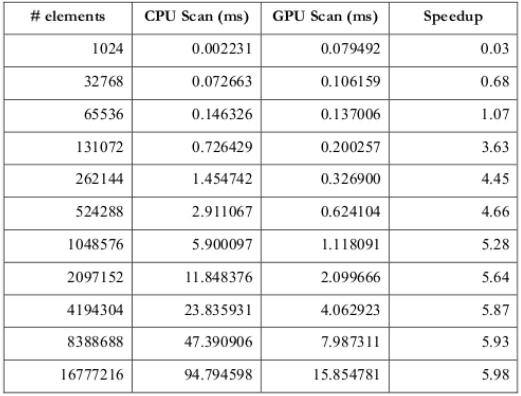
```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
```

Parallel scan: work-efficient parallel algorithm (6/6)



```
    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t    = temp[ai];
            temp[ai]   = temp[bi];
            temp[bi]  += t;
        }
    }

    __syncthreads();

    g_odata[2*thid]    = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

Parallel scan: performance



# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Performance of the work-efficient, bank conflict free Scan implemented in CUDA compared to a sequential scan implemented in C++. The CUDA scan was executed on an NVIDIA GeForce 8800 GTX GPU, the sequential scan on a single core of an Intel Core Duo Extreme 2.93 GHz.

Plan

Plan

Exercise 1 (1/4)

(1) Write a C function incrementing a `float` array `A` of size `N`

(2) Write a CUDA kernel incrementing a `float` array `A` of size `N` for a 1D grid, using 1D thread blocks, and assuming that each thread increments one element.

(3) Assuming that each thread block counts 64 threads, write the host code launching the kernel (including memory allocation on the device and host-device data transfers)

Exercise 1 (2/4)

(1) Write a C function incrementing a `float` array `A` of size `N`

```
void increment_Array_On_Host(float* A, int N)
{
     int i;
     for (i=0; i< N; i++)
         A[i] = A[i] + 1.f;
}
```

Exercise 1 (3/4)

(2) Write a CUDA kernel incrementing a `float` array `A` of size `N` for a 1D grid, using 1D thread blocks, and assuming that each thread increments one element.

```
__global__ void increment_On_Device(float *A, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N)
      A[idx] = A[idx]+1.0f;
}
```

Exercise 1 (4/4)

(3) Assuming that each thread block counts 64 threads, write the host code launching the kernel (including memory allocation on the device and host-device data transfers)

```
float *A_h;
float *A_d;
cudaMalloc((void **) &A_d, size);
// Allocate memory on the host for A and initialize A
...................................................
cudaMemcpy(A_d, A_h, sizeof(float)*N,
           cudaMemcpyHostToDevice);
int bSize = 64;
intnBlocks = N/bSize + (N%bSize == 0?0:1);
Increment_On_Device <<< nBlocks, bSize >>> (A_d, N);
cudaMemcpy(A_h, A_d, sizeof(float)*N, cudaMemcpyDeviceToHost
free(A_h);
cudaFree(A_d);
```

Exercise 2 (1/4)

We recall below the *Sieve of Eratosthenes*

```
def eratosthenes_sieve(n):
    # Create a candidate list within which non-primes will be
    # marked as None; only candidates below sqrt(n) need be checked
    candidates = range(n+1)
    fin = int(n**0.5)
    # Loop over the candidates, marking out each multiple.
    for i in xrange(2, fin+1):
        if not candidates[i]:
            continue
        candidates[2*i::i] = [None] * (n//i - 1)
    # Filter out non-primes and return the list.
    return [i for i in candidates[2:] if i]
```

Write a CUDA kernel implementing the Sieve of Eratosthenes on an input `n`:

(1) Start with a naive single thread-block kernel not using shared memory;

(2) Then, use shared memory and multiple thread blocks.

Exercise 2 (2/4)

(1) A naive kernel not using shared memory.

```
__global__ static void Sieve(int * sieve,int sieve_size)
{
 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 if (idx > 1) {
  for(int i=idx+idx;i <  sieve_size;i+=idx)
   sieve[i] = 1;
 }
}
```

The launching code could be:

```
cudaMalloc((void**) &device_sieve, sizeof(int) * sieve_size);

Sieve<<<1, sqrt(sieve_size), 0>>>(device_sieve, sieve_size);
```

But this would be quite inefficient. WHy?

Exercise 2 (3/4)

(1) A kernel using shared memory.

```
__global__ static void Sieve(int * sieve,int sieve_size)
{
 int b_x = blockIdx.x;
 int b_w = blockDim.x;
 int t_x = threadIdx.x;
 int offset = b_x * b_w;
 int ix = offset + tid;
 int t_y = threadIdx.y;

// copy the segment (tile) to shared memory
_shared__ int A[b_w];  A[tid] = sieve[ix];
__syncthreads();

knocker = tid;
// tid knocks down numbers that are multiple
// of knocker in the range [offset, offset + b_w)
```

Exercise 2 (4/4)

(1) A kernel using shared memory.

```
knocker = t_y;
// tid knocks down numbers that are multiple
// of knocker in the range [offset, offset + b_w[
int start = (offset % knocker == 0)
? offset : (offset / knocker +1) * knocker;

for (int jx = start; jx < offset + b_w; jx += knoecker)
      A[jx - offset] =1;
__syncthreads();

sieve[ix] = A[tid];
}
```

This code is almost correct . . . Let's fix it!

Exercise 3 (1/4)

Write a CUDA kernel (and the launching code) implementing the reversal of an input integer `n`. This reversing process will be out-of-place. As in the previous exercise:

(1) start with a naive kernel not using shared memory

(2) then develop a kernel using shared memory.

Exercise 3 (2/4)

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}

    int numThreadsPerBlock = 256;
    int numBlocks = dimA / numThreadsPerBlock;
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid,
         dimBlock >>>( d_b, d_a );
```

Exercise 3 (3/4)

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    extern __shared__ int s_data[];
    int inOffset   = blockDim.x * blockIdx.x;
    int in  = inOffset + threadIdx.x;
    // Load one element per thread from device memory and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];
    // Block until all threads in the block have
    // written their data to shared mem
    __syncthreads();
    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
}
```

Exercise 3 (4/4)

```
int numThreadsPerBlock = 256;
int numBlocks = dimA / numThreadsPerBlock;
int sharedMemSize = numThreadsPerBlock * sizeof(int);
// launch kernel
dim3 dimGrid(numBlocks);
dim3 dimBlock(numThreadsPerBlock);
reverseArrayBlock<<< dimGrid, dimBlock,
          sharedMemSize >>>( d_b, d_a );
```