CS3350: Efficient Usage of GPUs Memory Hierrachy: Lesson Learning through Matrix Transposition

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

UWO CS3350

Plan

Plan

Reversing an array (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. We shall proced as follows:

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

(2) then develop a kernel using shared memory.

Reversing an array (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 );
```

Reversing an array (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];
}
```

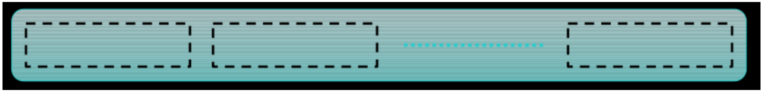
Reversing an array (4/4)

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

Plan

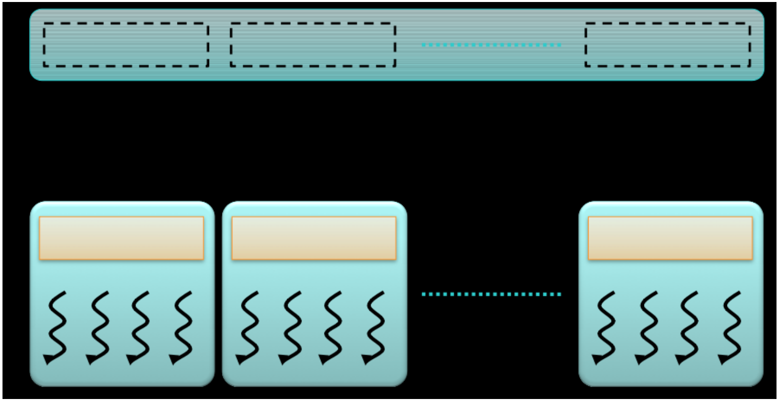
A Common Programming Strategy

Partition data into subsets that fit into shared memory



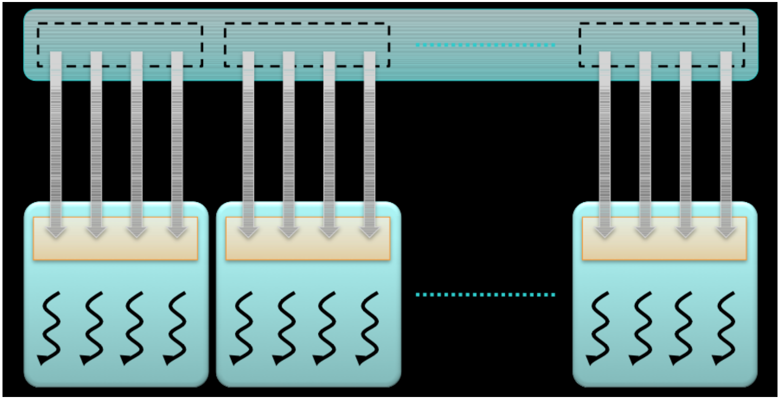
A Common Programming Strategy

Handle each data subset with one thread block



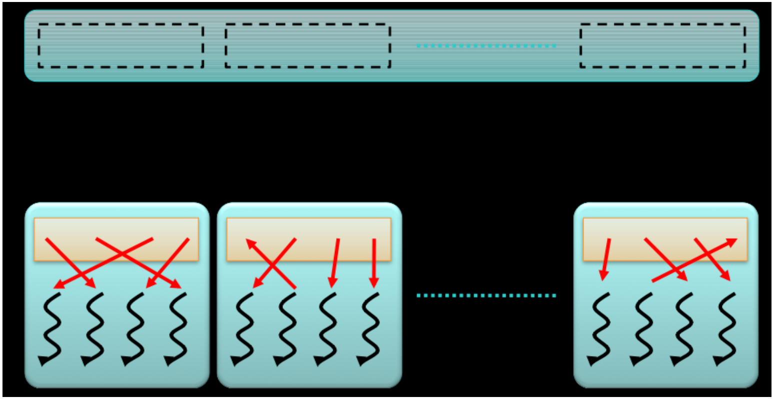
A Common Programming Strategy

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



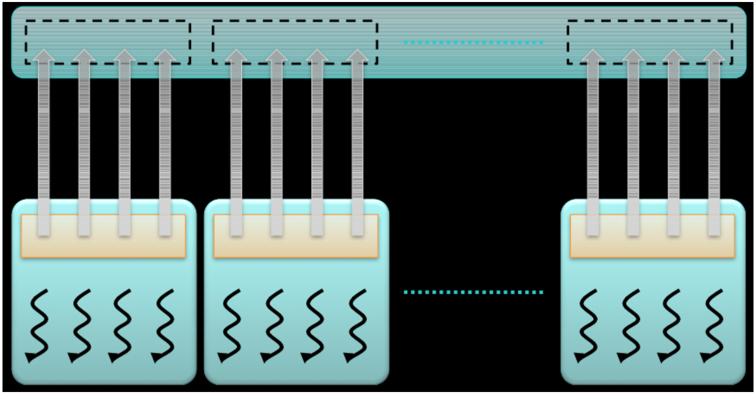
A Common Programming Strategy

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

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 consider an $n \times n$ 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**,
  - The difference between these two timings is kernel launch and synchronization overheads.

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.

```
__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++) {
    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 r=0; r < nreps; r++) {
     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:

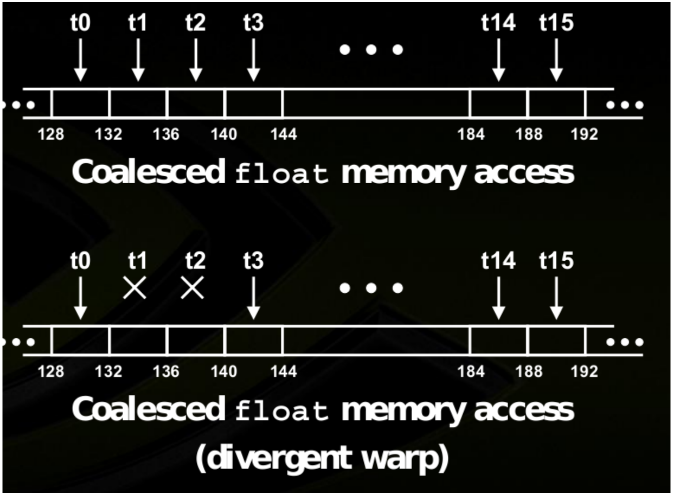
Routine	Bandwidth (GB/s)
copy	105.14
naive transpose	18.82

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

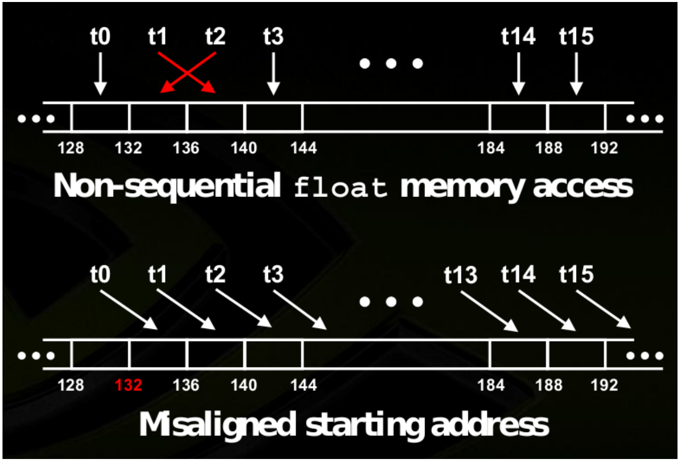
Coalesced Transpose (1/11)

- 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.
- Last two requirements are relaxed with compute capabilities of 1.2.
- Coalescing happens even if some threads do not access memory (**divergent warp**)

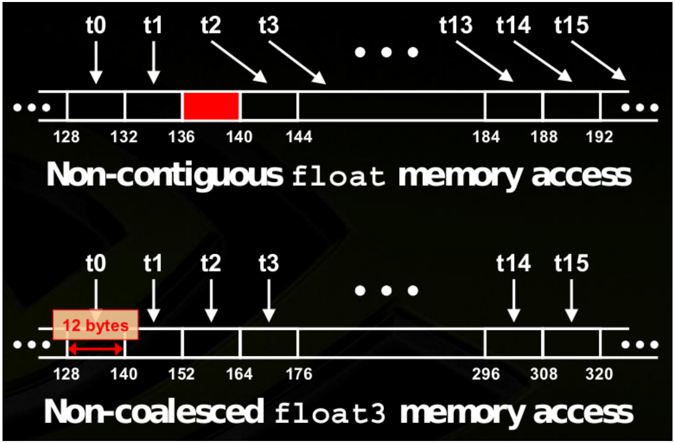
Coalesced Transpose (2/11)



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.

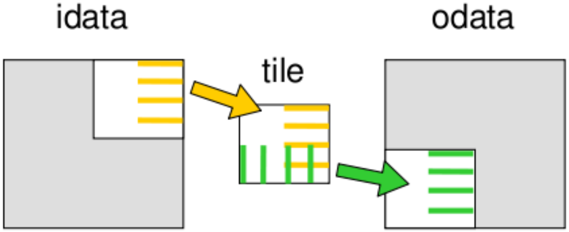
Coalesced Transpose (6/11)

- The way to avoid uncoalesced global memory access is
    1. to read the data into shared memory and,
    2. have each half warp access non-contiguous locations in shared memory in order to write contiguous data to `odata`.
- There is no performance penalty for non-contiguous 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)

Routine	Bandwidth (GB/s)
copy	105.14
shared memory copy	104.49
naive transpose	18.82

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)

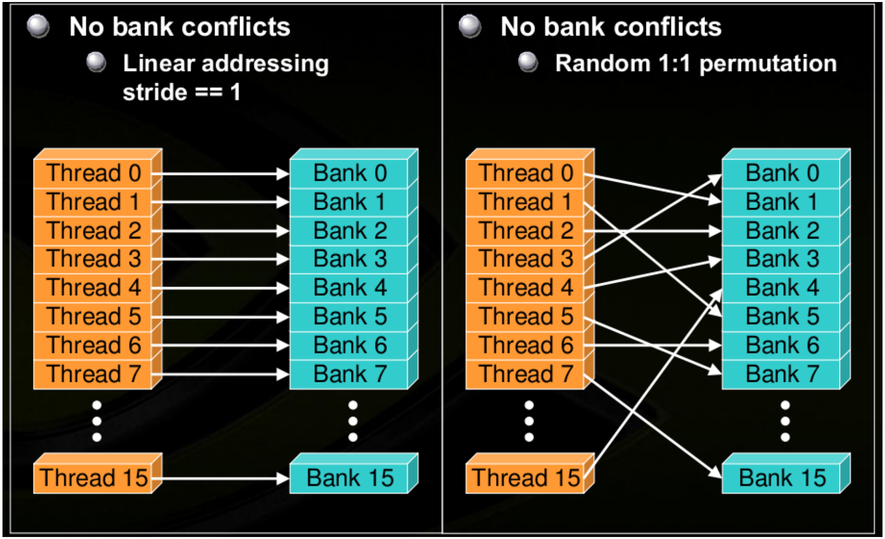
Routine	Bandwidth (GB/s)
copy	105.14
shared memory copy	104.49
naive transpose	18.82
coalesced transpose	51.42

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.

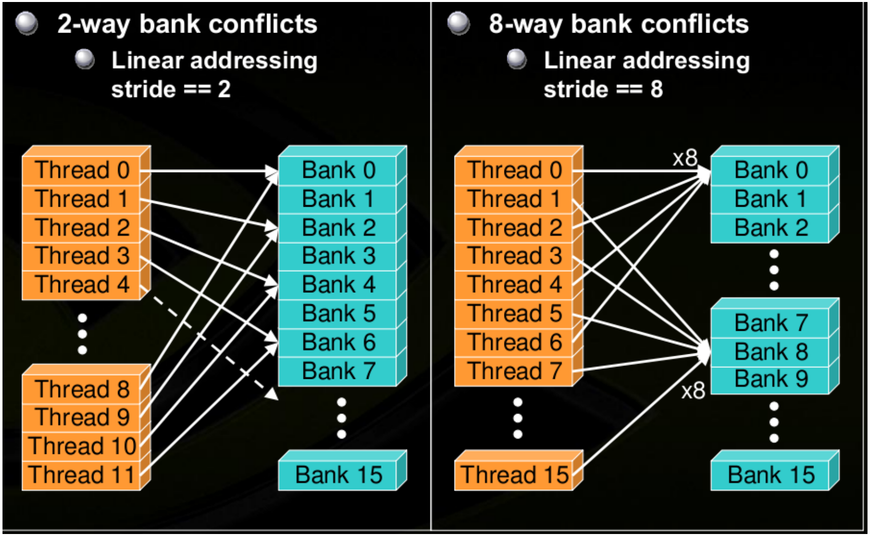
Shared memory bank conflicts (1/6)

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)

```
Device : Tesla M2050
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
                 Routine         Bandwidth (GB/s)
                    copy              105.14
      shared memory copy              104.49
         naive transpose               18.82
     coalesced transpose               51.42
 conflict-free transpose               99.83
```

- 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.