KOÇ UNIVERSITY

Matrix Transpose in CUDA

Didem Unat

dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

Matrix Transpose

- Out-of-place transpose: i.e. the input and output are separate arrays in memory
- For simplicity of presentation, we'll consider only square matrices
- We will compare the performance of the matrix transpose kernel against a matrix copy kernel
  - which is an upper bound of effective bandwidth
  - Because copy performance indicates the performance that we would like the matrix transpose to achieve

Effective Bandwidth

- Effective (or Sustained Bandwidth)
  - Amount of data moved during computation/ Execution Time

- For both matrix copy and transpose, the relevant performance metric is **effective bandwidth,**
  - Why?
- Why not use **flops rate**?

Effective Bandwidth

- Effective (or Sustained Bandwidth)
  - Amount of data moved during computation/ Execution Time

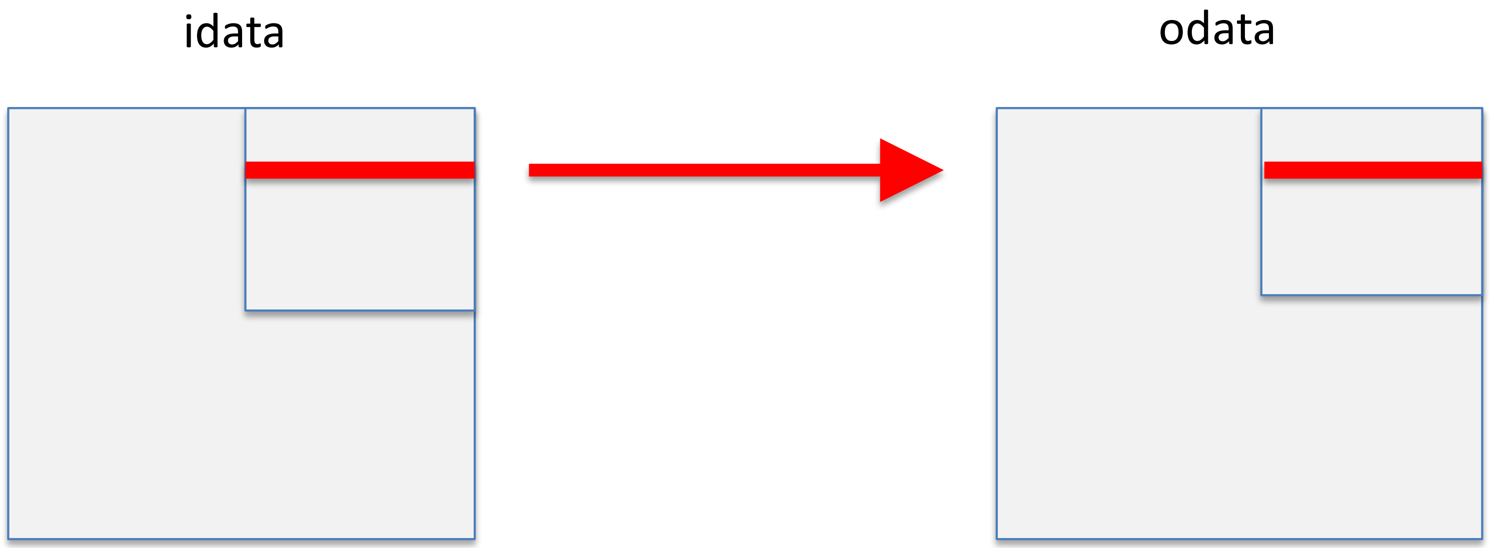
- For both matrix copy and transpose, the relevant performance metric is effective bandwidth, why?

- For Copy/Transpose

$$\underbrace{\mathtt{N * N}}_{\boxed{\text{Matrix size}}} \mathtt{*} \underbrace{\mathtt{4}}_{\boxed{\text{Bytes per element}}} \mathtt{*} \underbrace{\mathtt{2}}_{\boxed{\text{Read and Write}}} \mathtt{/} \underbrace{\mathtt{1024}^3}_{\boxed{\text{Converting to GB (10^9)}}} \mathtt{/\ Exec\_Time\_in\_secs = GB/s}$$

Lab-8

- Start with implementing the simple copy operation
- We will gradually optimize the transpose kernel
- Kernel 1: performs copy from idata to odata



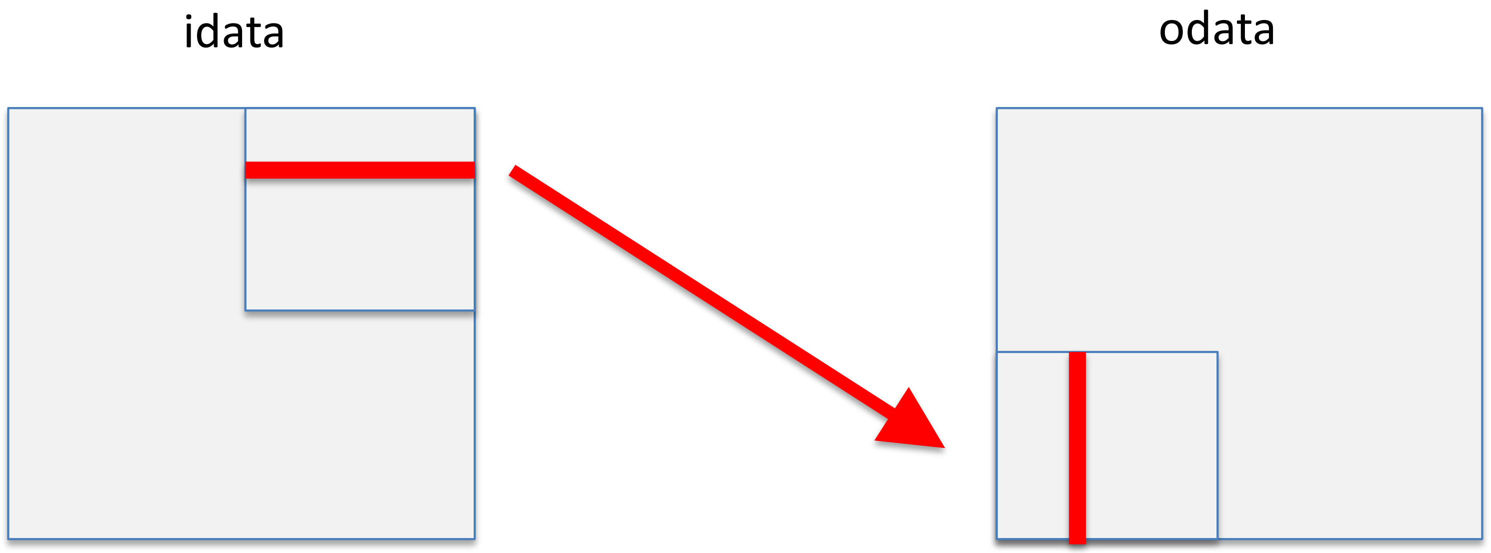
Kernel 1: Simple Copy

```
__global__ void copy(float *odata, float *idata, int width,
                     int height)
{
    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 = index_in;

    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS){
        odata[index_out] = idata[index_in];
        //update index_out and in with j
    }
}
```

- Each thread copies TILE_DIM/BLOCK_ROWS elements of the matrix in a loop
- For example:
  - Create 32 x 8 threads in a block, TILE_DIM= 32, BLOCK_ROWS=8
  - A thread block copies 32 x 32 elements
- Both reads and writes are coalesced

Naïve Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



Kernel 2: Naïve Transpose

- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(float *odata, const
                        float *idata, int width, int height)
{
   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 + width * xIndex;

   for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
       odata[index_out] = idata[index_in];
       //update index_out and in
}
```

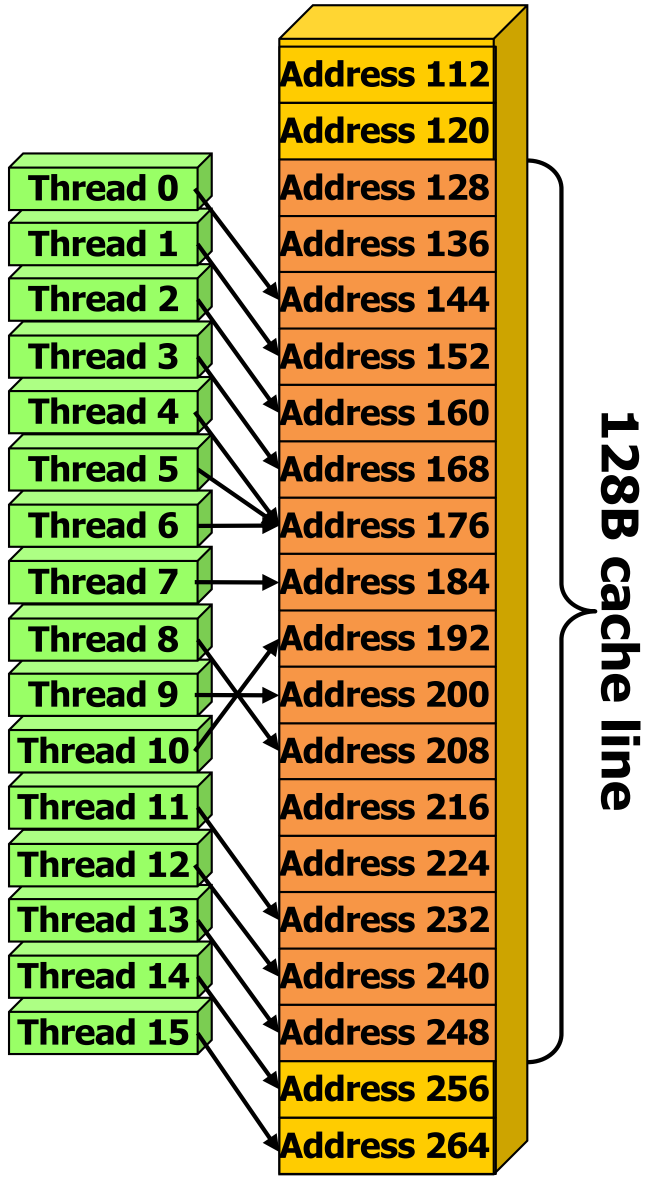
Test Results

- For 4096 x 4096 test matrix
- Naïve transpose achieves a fraction of effective bandwidth on the device
    - There is a room for improvement.

Tesla V100	Effective Bandwidth (GB/s)
Copy (Kernel 1)	718.7
Transpose Naïve (Kernel 2)	163.7

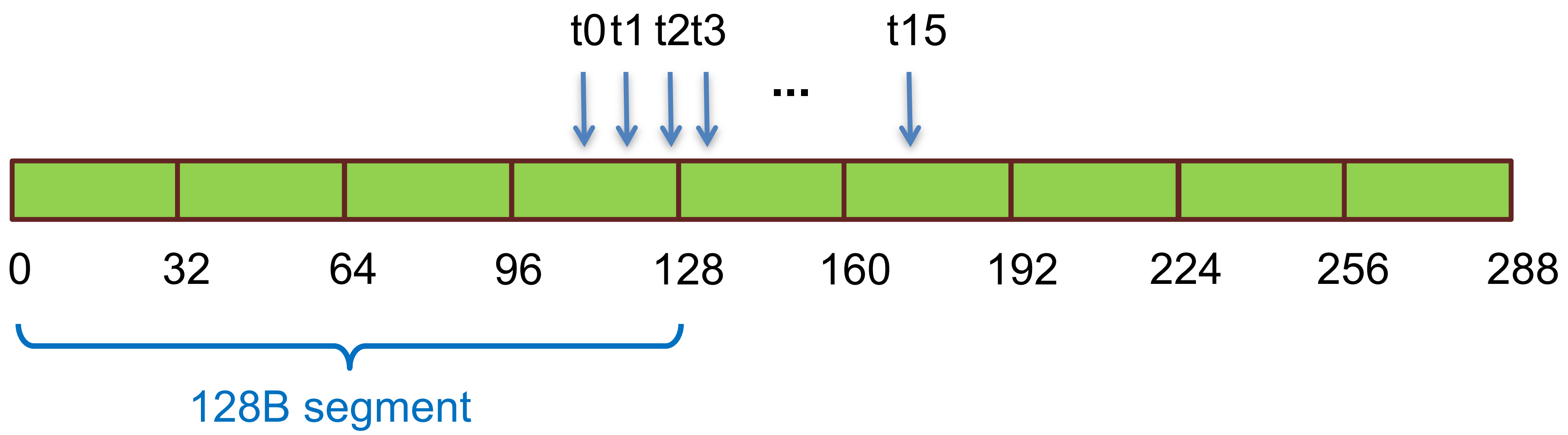
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don’t use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32/64/128 bytes
  - Unaligned accesses will cost more



Threads 0-15 access 4-byte words at addresses 116-176

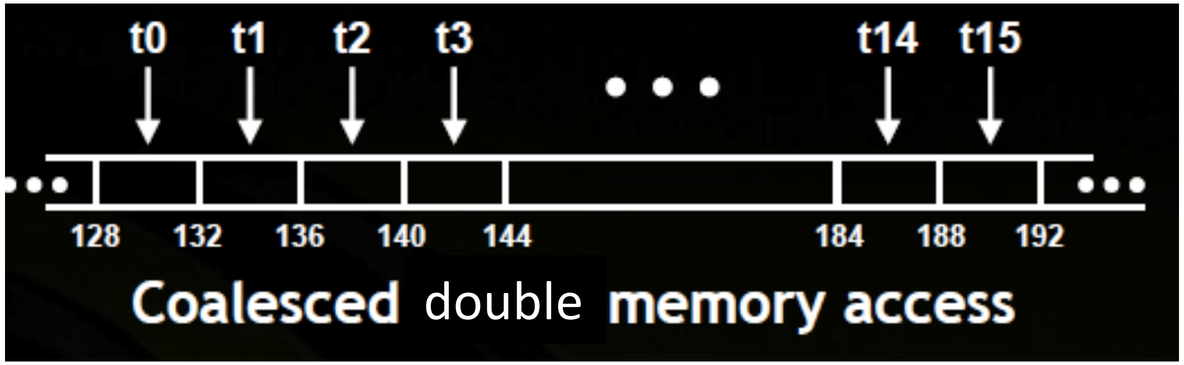
- For example, thread 0 accesses address 116
- 128-byte segment: 0-127
- This causes two cache line transfers



Memory Coalescing

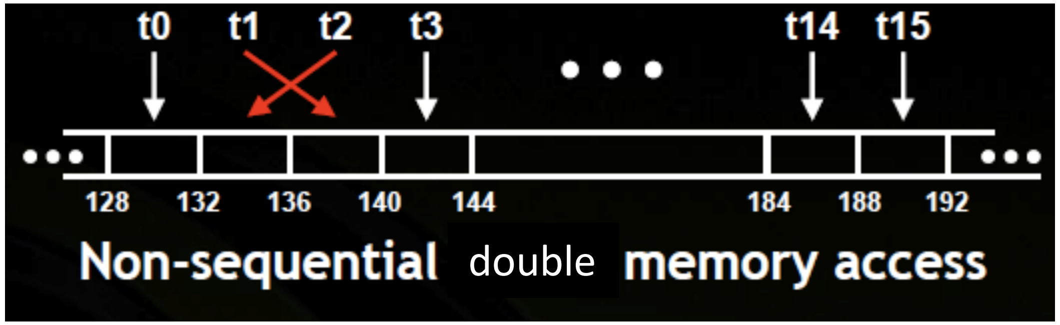
- Loads and stores by threads of a warp are coalesced by the device into one memory operation if certain access requirements are met
  - Warp: 32 adjacent threads

- Simple Access Pattern
  - *k*th thread accesses the *k*th word in a cache line
  - 4-byte words by 32 threads
    - Single transfer of 128 Bytes

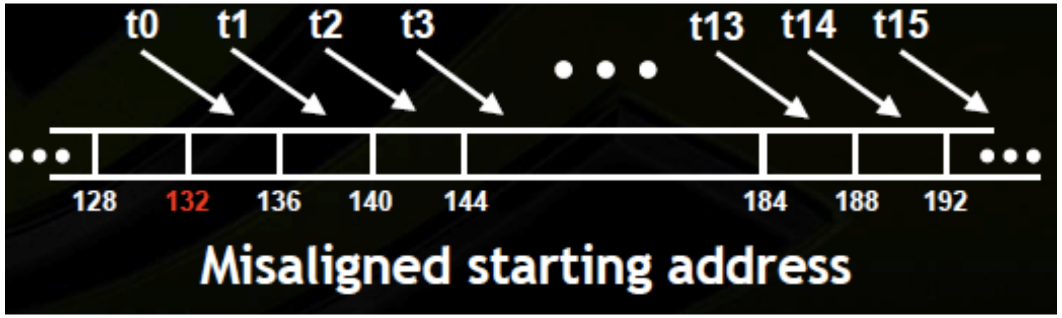


Nvidia Corp.

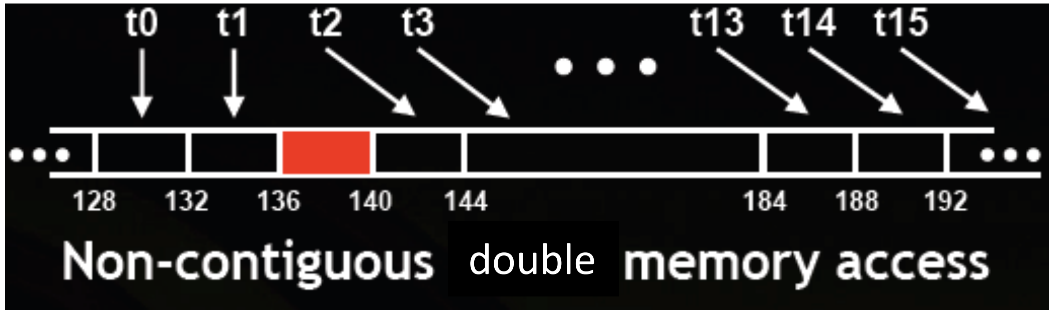
Coalesced Memory Accesses



if accesses by the threads of the warp had been permuted within this segment, still only one 128-byte transaction would have been performed – No problem



if sequential threads in a warp access memory that is sequential but not aligned with the cache lines, two 128-byte L1 cache will be requested.

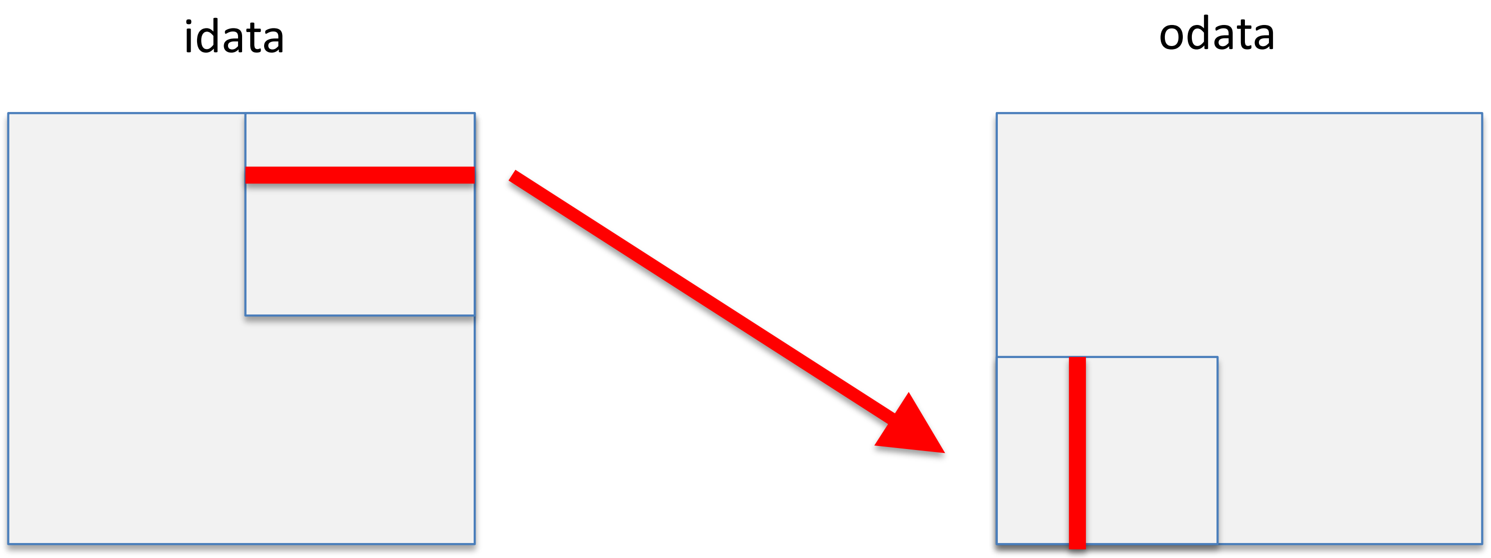


Gaps in the memory addresses causes multiple cache lines to be fetched

Nvidia Corp.

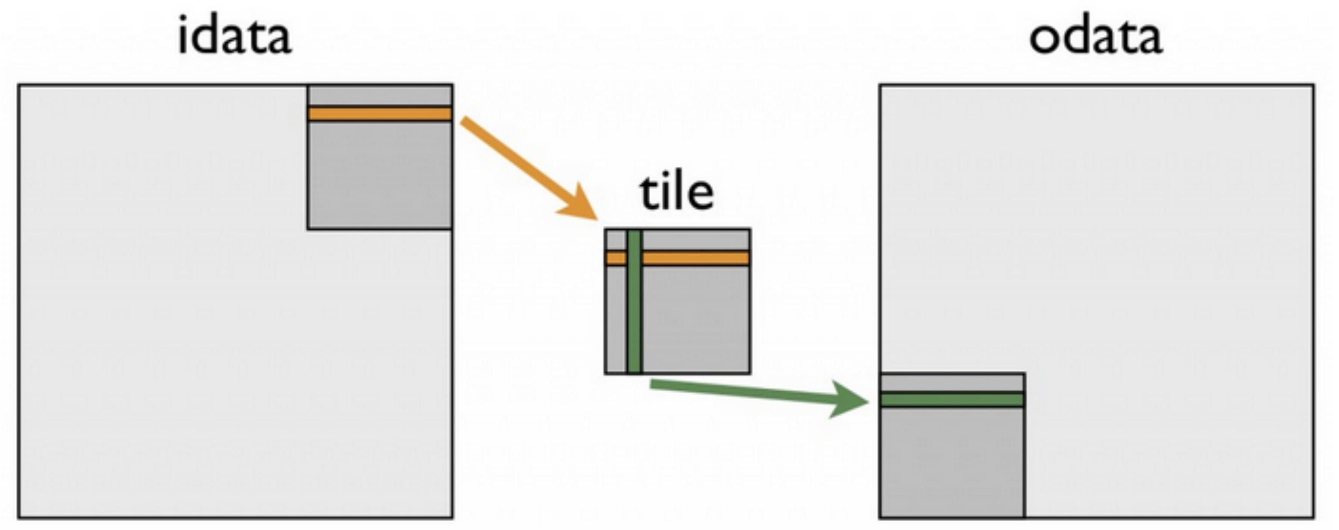
Naïve Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



Coalesced Transpose via Shared Memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires __syncthreads() since threads access data in shared memory stored by other threads



Kernel 3: Transpose in Shared Memory

```
__global__
void transposeCoalesced(float *odata, const float *idata)
{
     __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;

    // Load data from global memory to shared memory ;

    // synchronize
    // take transpose of the indices
    // calculate index_out

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Kernel 3: Transpose in Shared Memory

```
__global__
void transposeCoalesced(float *odata, const float *idata)
{
     __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;

    // Load data from global memory to shared memory in a
loop;

    // synchronize
    // take transpose of the indices
   // calculate index_out
   // Store into odata in a loop
    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Test Results

- For 4096 x 4096 test matrix
- Naïve transpose achieves a fraction of effective bandwidth on the device
  - There is a room for improvement.

Tesla V100	**Effective Bandwidth (GB/s)**
Copy (Kernel 1)	718.7
Transpose Naïve (Kernel 2)	163.7
Transpose Coalesed (Kernel 3)	534.2

Shared Memory Bank Conflicts

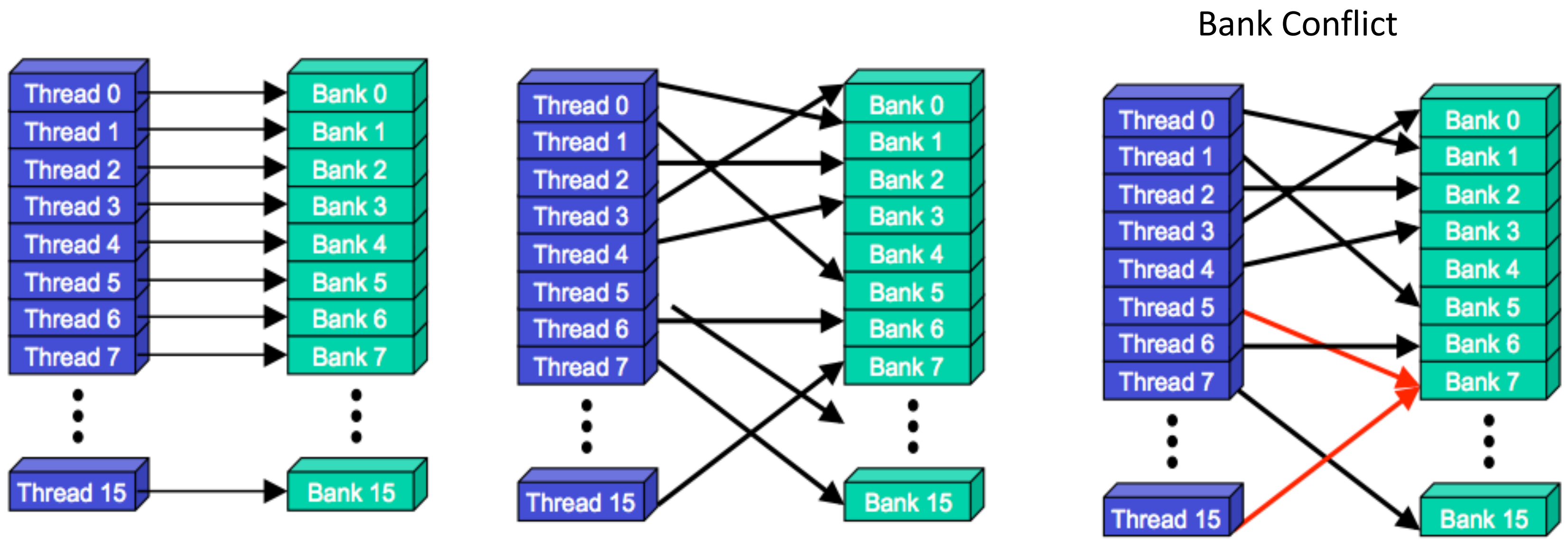
- Shared memory is divided into 16 equally-sized memory modules, called banks, which are organized such that successive 32-bit words are assigned to successive banks.

- 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.

- 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.

Shared Memory Banks

- A load or store of n addresses spanning n distinct memory banks can be serviced simultaneously
- Multiple addresses map to same memory bank
  - Accesses are serialized
  - Exception: if all access the same address, then broadcast, no problem



Bank Conflict

- The coalesced transpose uses a 32 × 32 shared memory array of floats.
- For this sized array, all data in columns k and k+16 are mapped to the same bank.
- 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.

Kernel 4: Bank Conflicts in Transpose

- Shared memory tiles (32x32) of floats
  - Data in columns are mapped to the same bank
  - 32-way bank conflict reading columns in tile
- Solution is simple
  - Pad shared memory array (no other changes in the src)
  - Data in anti-diagonals are same bank

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

Test Results

- For 4096 x 4096 test matrix
- Naïve transpose achieves a fraction of effective bandwidth on the device
  - There is a room for improvement.

Tesla V100	Effective Bandwidth (GB/s)
Copy (Kernel 1)	718.7
Transpose Naïve (Kernel 2)	163.7
Transpose Coalesed (Kernel 3)	534.2
No Bank Conflict (Kernel 4)	690.0

Summary

- Performing coalesced accesses give performance boost
- When multiple addresses map to same memory bank, accesses are serialized. There is a simple solution for that
- Having a reference implementation help reason about performance optimizations
  - Here is the copy kernel for transpose is a good reference.

Solutions

Kernel 1

```
//Kernel 1:
__global__ void copy(float *odata, float *idata, int
width, int height)
{
    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];
    }

}
```

Kernel 2

```
// Kernel 2:
__global__ void transposeNaive(float *odata, float
*idata, int width, int height)
{
    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];
    }
}
```

Kernel 3: Transpose in Shared Memory

```
__global__
void transposeCoalesced(float *odata, const float *idata)
{
     __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;

    tile[threadIdx.y][threadIdx.x] = idata[index_in];

     __syncthreads();
    xIndex = blockIdx.y * TILE_DIM + threadIdx.
    yIndex = blockIdx.x * TILE_DIM + threadIdx.
   int index_out = xIndex + yIndex * height ;

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Each thread loads one element into shared memory

Kernel 3: Transpose in Shared Memory

```
__global__ void transposeCoalesced(float *odata, float *idata)
{
     __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;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS) {
     tile[threadIdx.y+j][threadIdx.x] = idata[index_in+j*width];
    }
    __syncthreads();
    xIndex = blockIdx.y * TILE_DIM + thread
    yIndex = blockIdx.x * TILE_DIM + thread
    int index_out = xIndex + yIndex * height ;
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS){
      odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+j];
    }
}
```

Each thread loads TILE_DIM/BLOCK_ROWS elements into shared memory