Matrix Transpose

Get 90% Bandwidth

Matrix Transpose

- Inherently parallel
  - Each element independent of another
- Simple to implement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Matrix Transpose [CPU Transpose]

```
for(int i = 0; i < rows; i++)
    for(int j = 0; j < cols; j++)
        transpose[i][j] = matrix[j][i]
```

- Easy
- $O(n^2)$
- Slow!!!!!!

Matrix Transpose

[Naive GPU Transpose]

- GPU Transpose
  - Launch 1 thread per element
  - Compute index
  - Compute transposed index
  - Copy data to transpose matrix
- O(1) using Parallel compute
- Essentially one memcpy from global-to-global
  - It should be fast, shouldn’t it?

Matrix Transpose

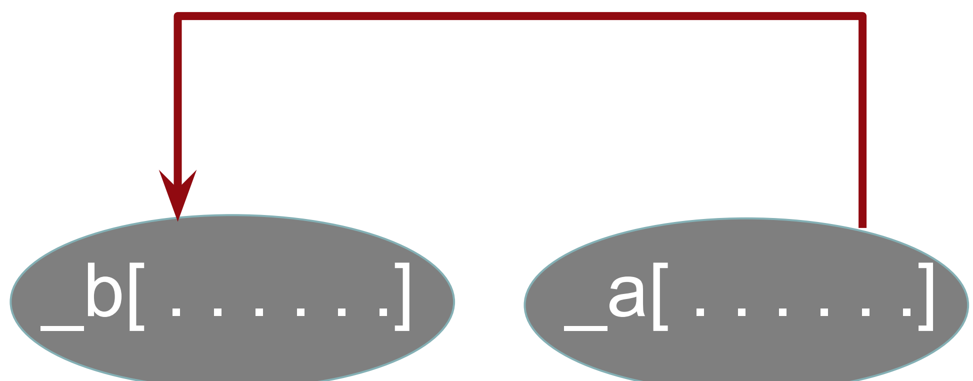
[Naive GPU Transpose]

```
__global__ void matrixTranspose(float *_a, float *_b)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;   // row
    int j = blockIdx.x * blockDim.x + threadIdx.x;   // col

    int index_in = i*cols+j;    // (i,j) from matrix A
    int index_out = j*rows+i;   // transposed index

    b[index_out] = a[index_in];
}
```

2	5	-2	6	6
3	5	3	4	6
4	8	4	-1	3



2	3	4
5	5	8
-2	3	4
6	4	-1
6	6	3

Matrix Transpose

[Naive GPU Transpose]

- Problems?

Matrix Transpose

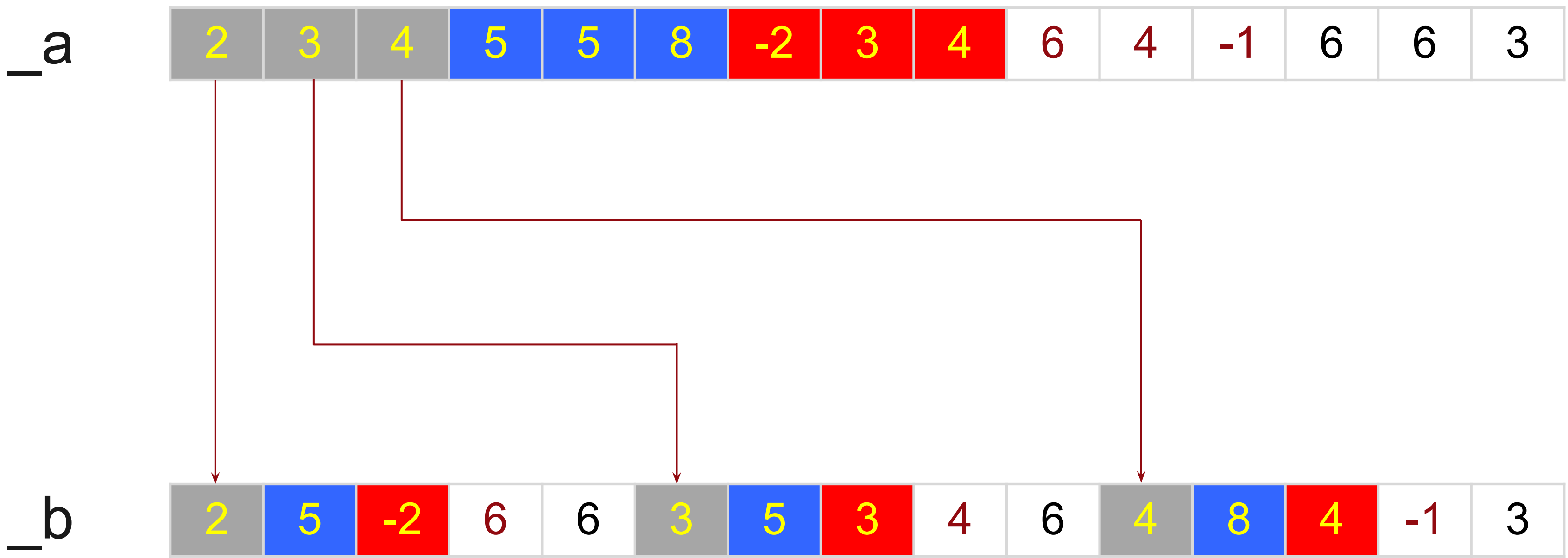
[Naive GPU Transpose]

- Problems?
    - Non-coalesced memory

- Improvements?

GMEM Access Pattern in NT

READ - Coalesced memory access Good!



WRITE - Uncoalesced memory access Bad!

Matrix Transpose

[Naive GPU Transpose]

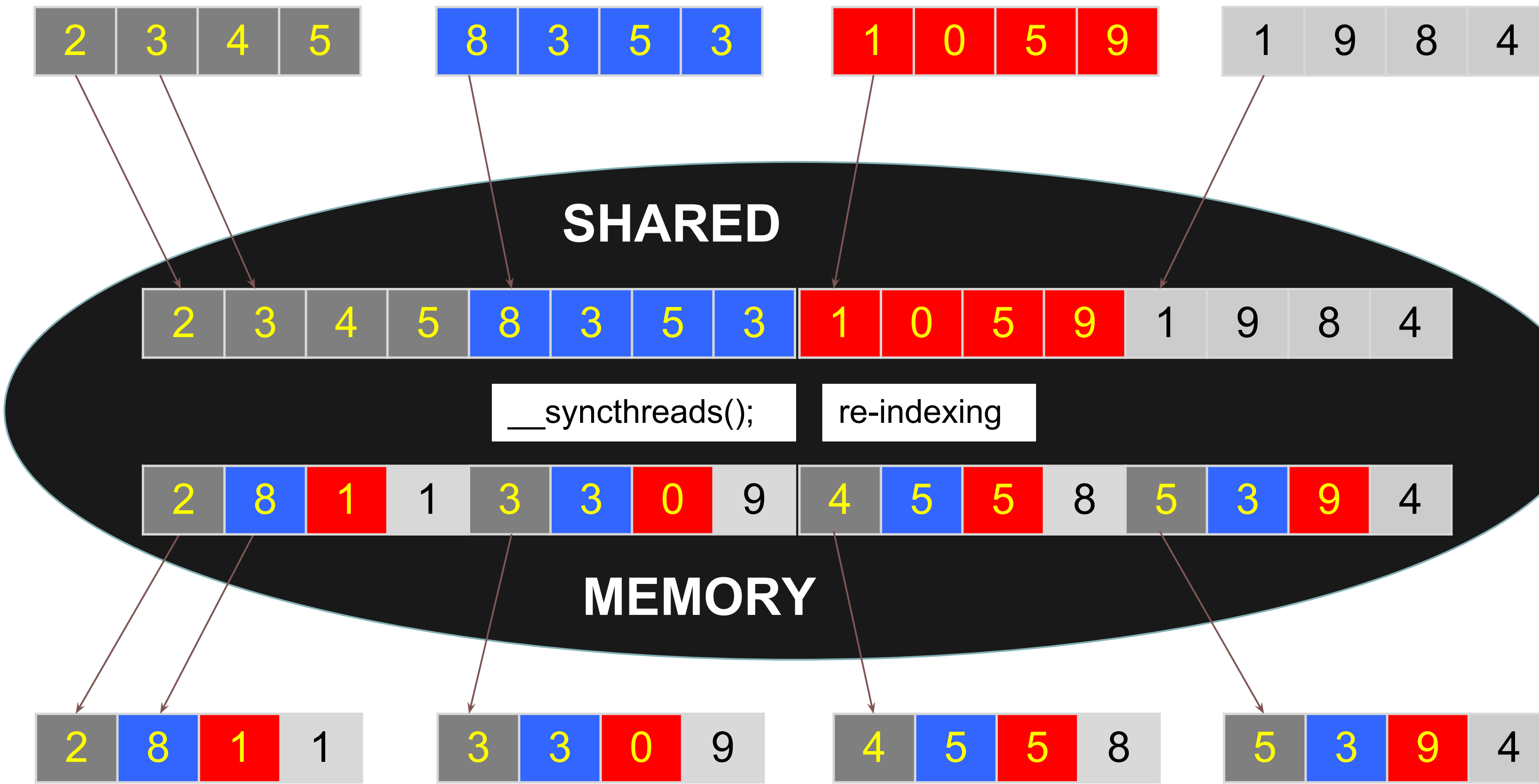
- Problems?
  - Non-coalesced memory

- Improvements?
  - Use shared memory
  - Use coalesced memory access

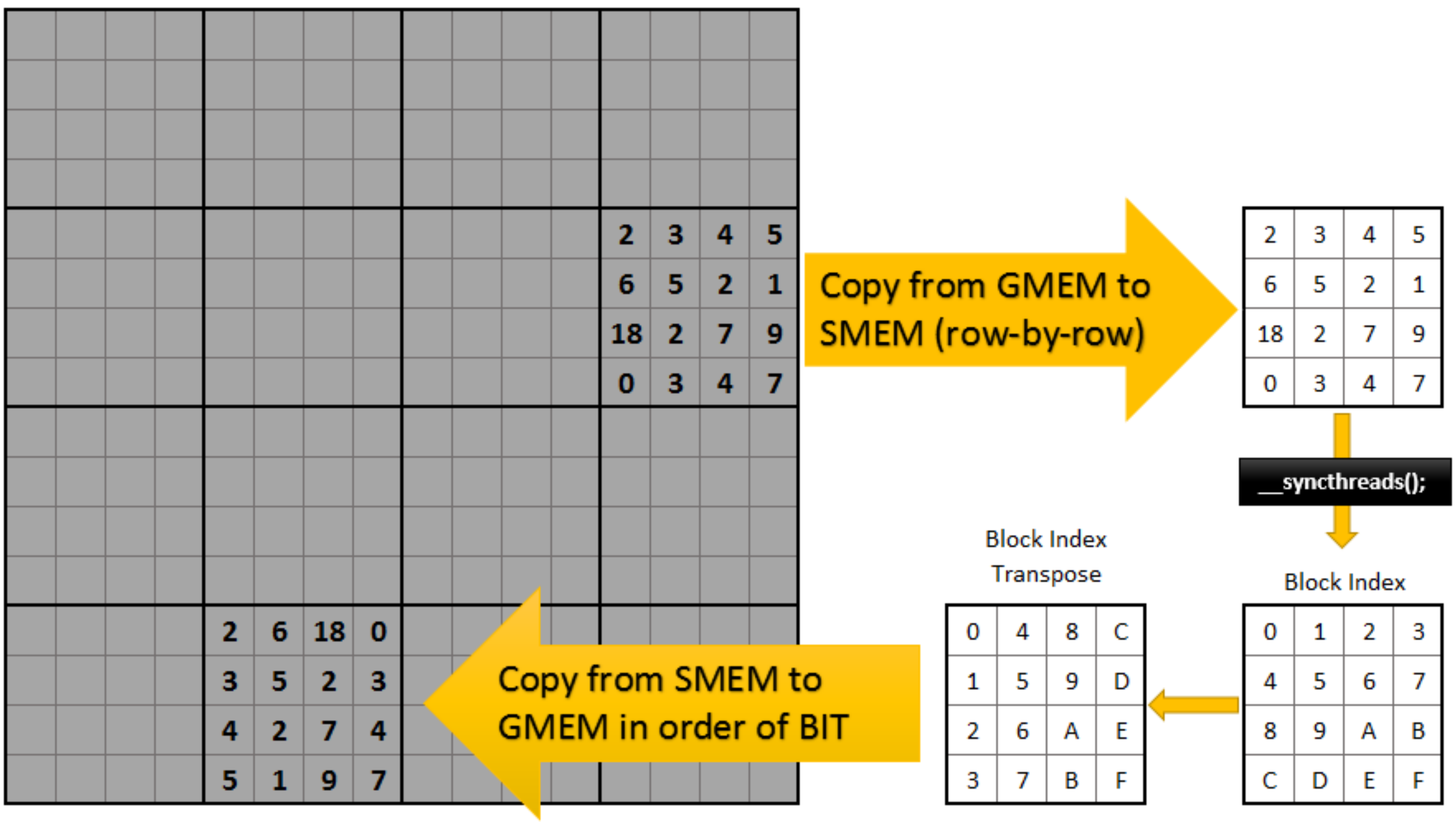
Matrix Transpose [GPU Transpose]

- Use Shared Memory
  - Allows temporary storage of data
  - Use coalesced memory access to global memory
- Walkthrough
  - Compute input index (same as in naive transpose)
  - Copy data to shared memory
  - Compute output index
    - Remember, coalesced memory access
    - Hint, transpose only in shared memory
  - Copy data from shared memory to output

Memory Access Pattern for SMT



Shared Memory Transpose



Transpose: Shared Memory

```
__global__ void matrixTransposeShared(const float *_a,
                                            float *_b)
{
    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y];
    int bx = blockIdx.x * BLOCK_SIZE_X;
    int by = blockIdx.y * BLOCK_SIZE_Y;
    int i  = by + threadIdx.y;   int j  = bx + threadIdx.x; //input
    int ti = bx + threadIdx.y;   int tj = by + threadIdx.x;
//output

    if(i < rows && j < cols)
        mat[threadIdx.x][threadIdx.y] = a[i * cols + j];
    __syncthreads();      //Wait for all data to be copied
    if(tj < cols && ti < rows)
        b[ti * rows + tj] = mat[threadIdx.y][threadIdx.x];
}
```

Matrix Transpose [GPU Transpose]

- Problem?

Matrix Transpose [GPU Transpose]

- Problem?
  - Why are we not even close to max bandwidth?
  - Hint, think “banks”

- Solution?

Matrix Transpose [GPU Transpose]

- Problem?
  - Why are we not even close to max bandwidth?
  - Hint, think “banks”

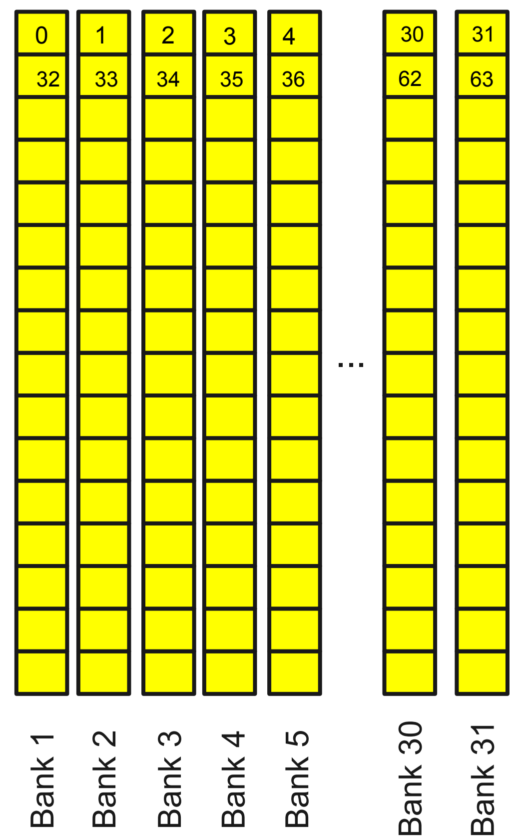
- Solution?
  - Remove bank conflicts

Bank Conflicts

Banks

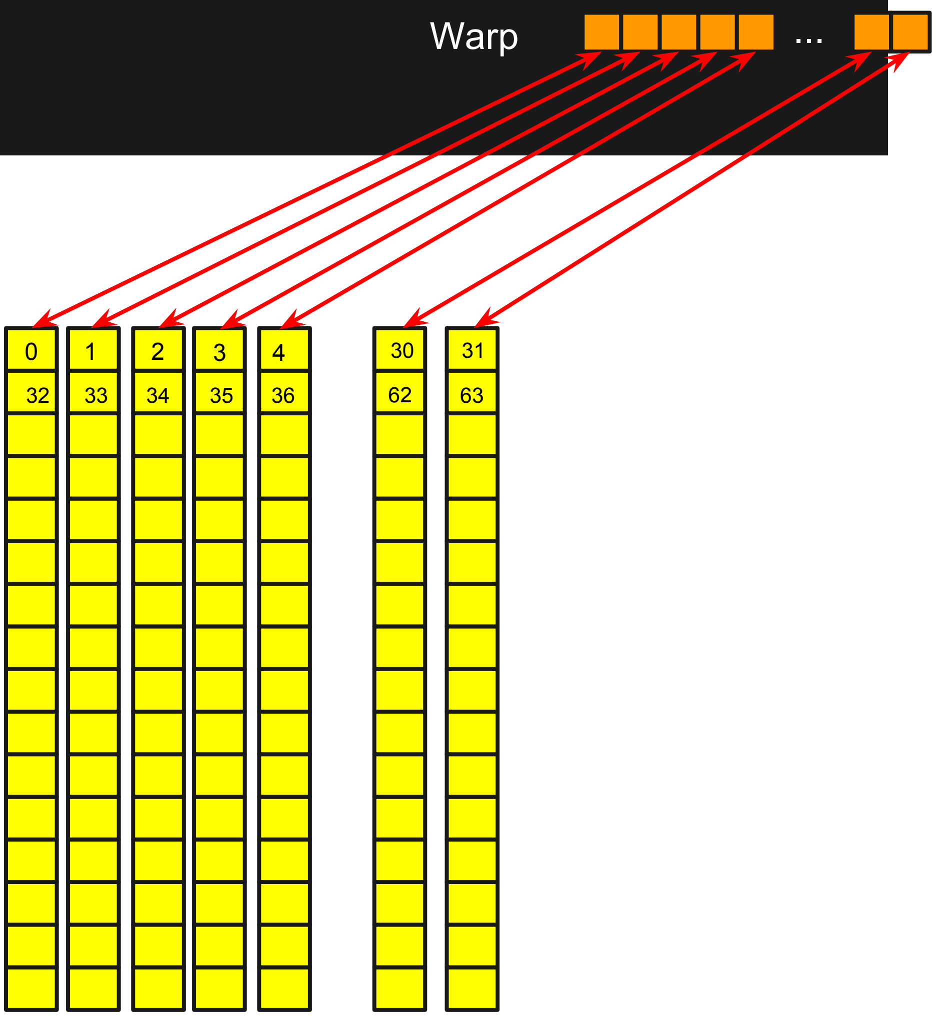
- Shared Memory is organized into 32 banks
- Consecutive shared memory locations fall on different banks

```
__shared__ float tile[64];
```



Banks

- Access to different banks by a **warp** executes in parallel.

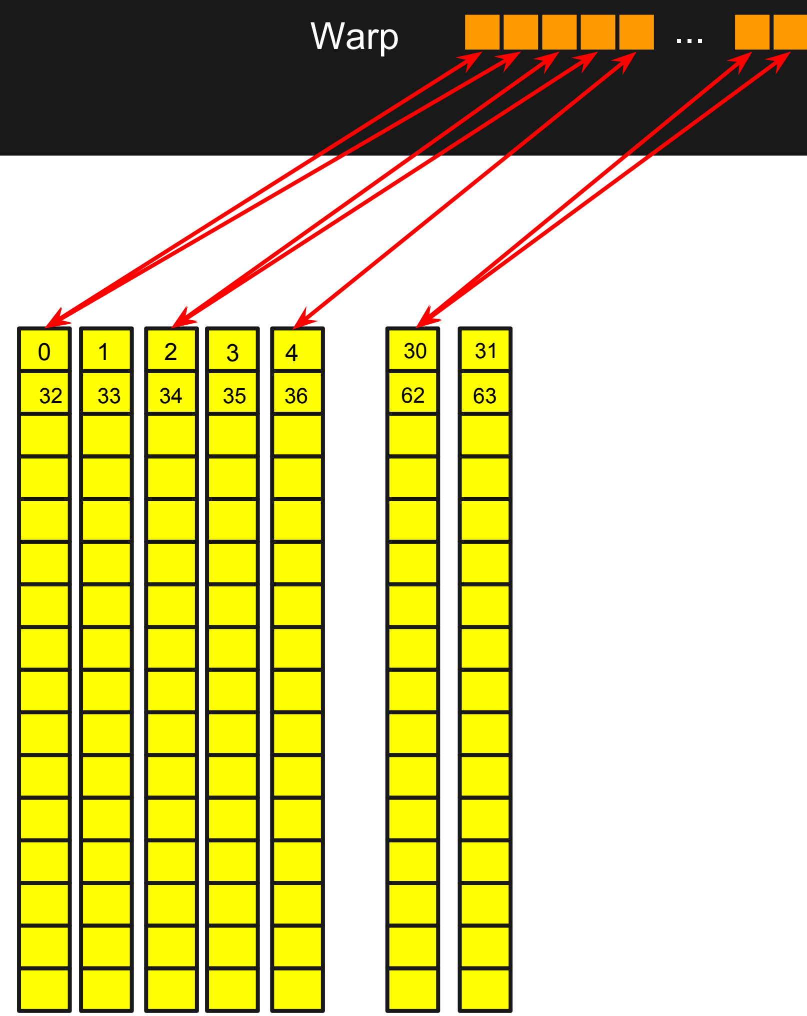



```
__shared__ float tile[64];
int tidx = threadidx.x;
float foo = tile[tidx] - 3;
```

Banks

- Access to the same element in a bank is also executed in parallel.

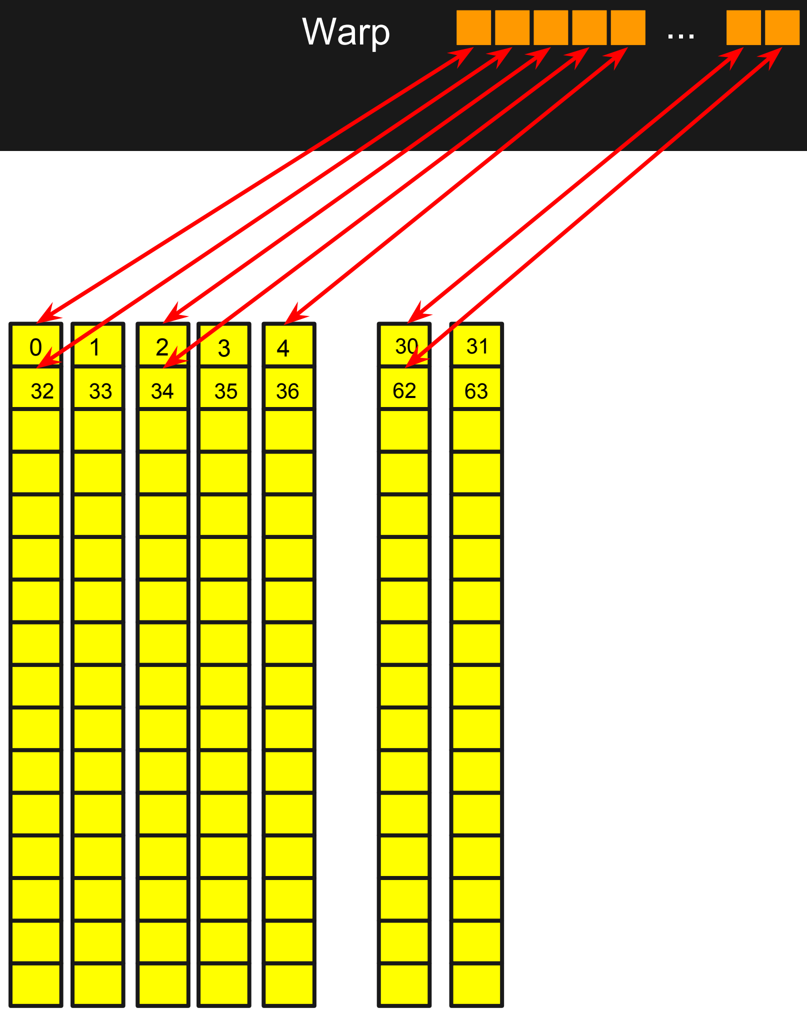
```
__shared__ float tile[64];
int tidx = threadidx.x;
int bar = tile[tidx - tidx % 2];
```



Banks

- Access to the different elements in a bank is executed serially.
- “2 way bank conflict”

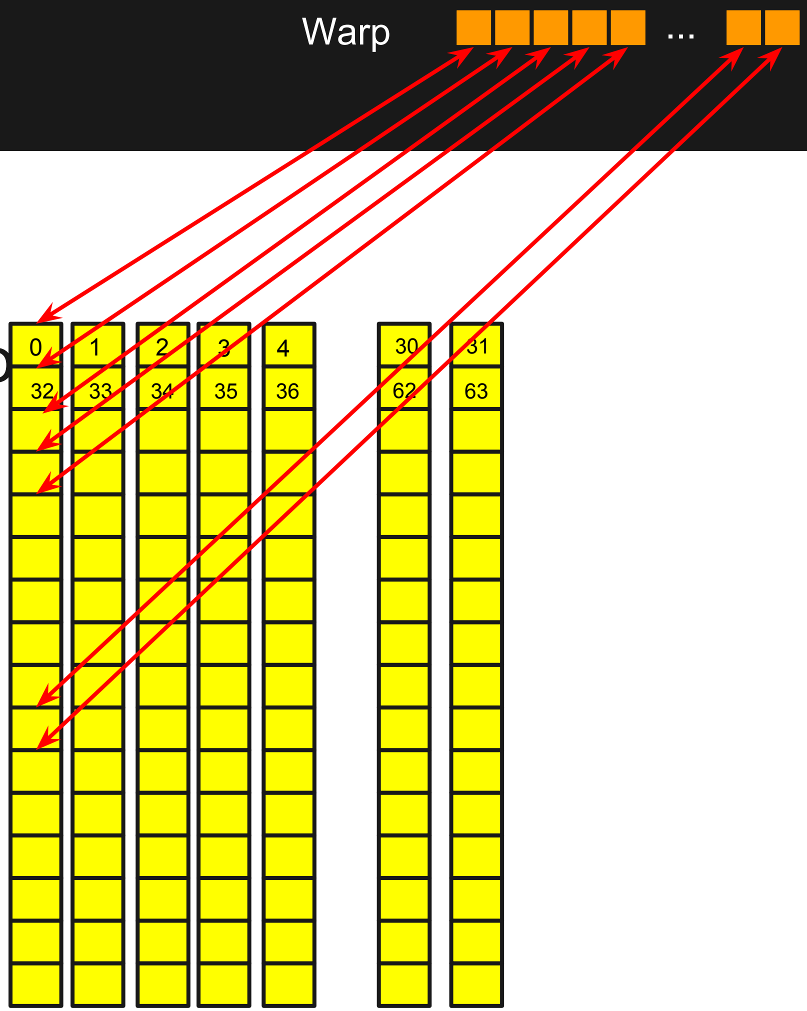
```
__shared__ float tile[64];
int tidx = threadidx.x;
tmp = tile[tidx + tidx % 2*31];
```



Banks

- Access to the different elements in a bank is also executed serially.
- 32 way bank conflict

```
_b[index_out] = tile[tx][ty];
```



Transpose: Shared Memory

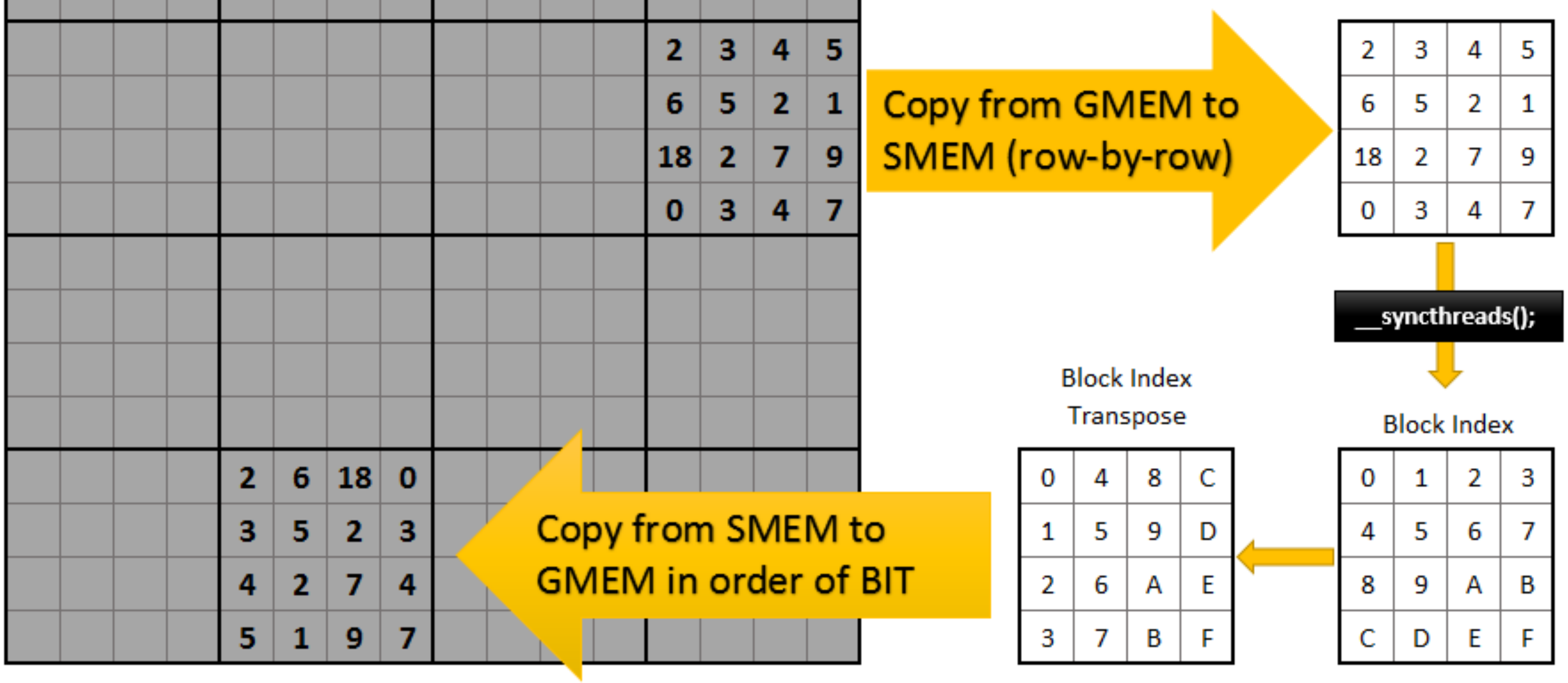
```
__global__ void matrixTransposeShared(const float *_a,
                                           float *_b)
{
    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y];
    int bx = blockIdx.x * BLOCK_SIZE_X;
    int by = blockIdx.y * BLOCK_SIZE_Y;
    int i  = by + threadIdx.y;   int j  = bx + threadIdx.x; //input
    int ti = bx + threadIdx.y;   int tj = by + threadIdx.x;
//output
```

Represents row of the “bank”

Represents bank number or “col”
Same for all threads in the warp

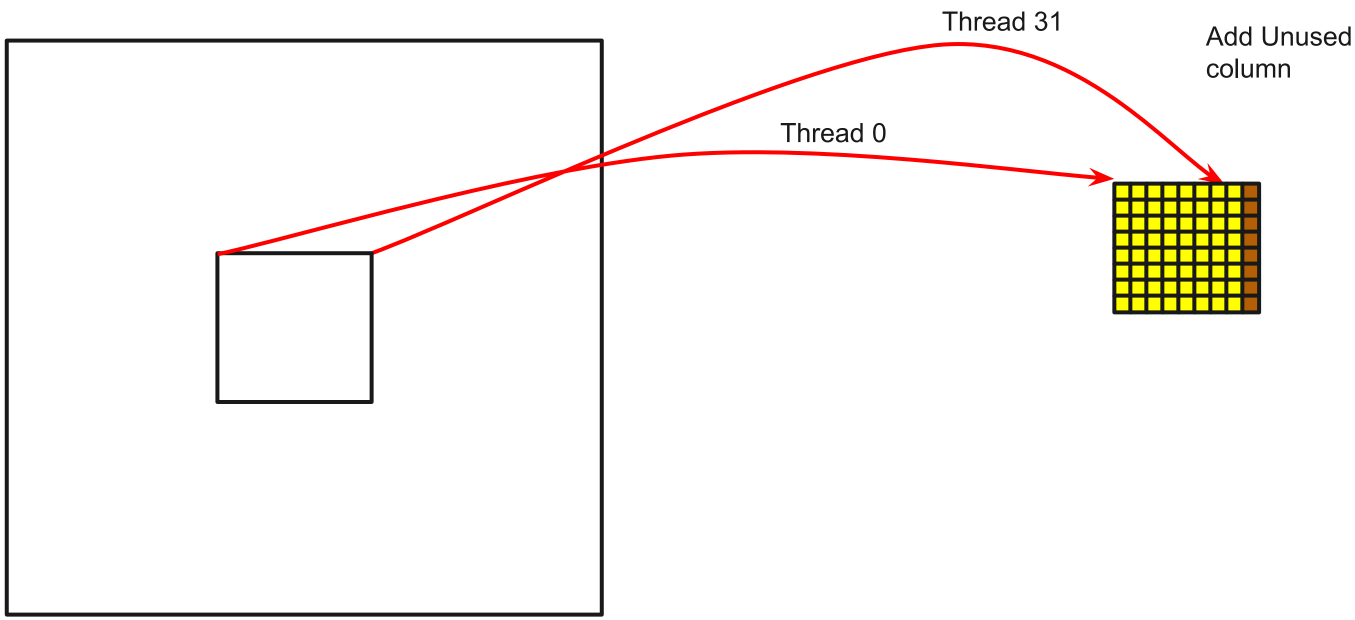
```
    if(i < rows && j < cols)
        mat[threadIdx.x][threadIdx.y] = a[i * cols + j];
    __syncthreads();      //Wait for all data to be copied
    if(tj < cols && ti < rows)
        b[ti * rows + tj] = mat[threadIdx.y][threadIdx.x];
}
```

Shared Memory Transpose



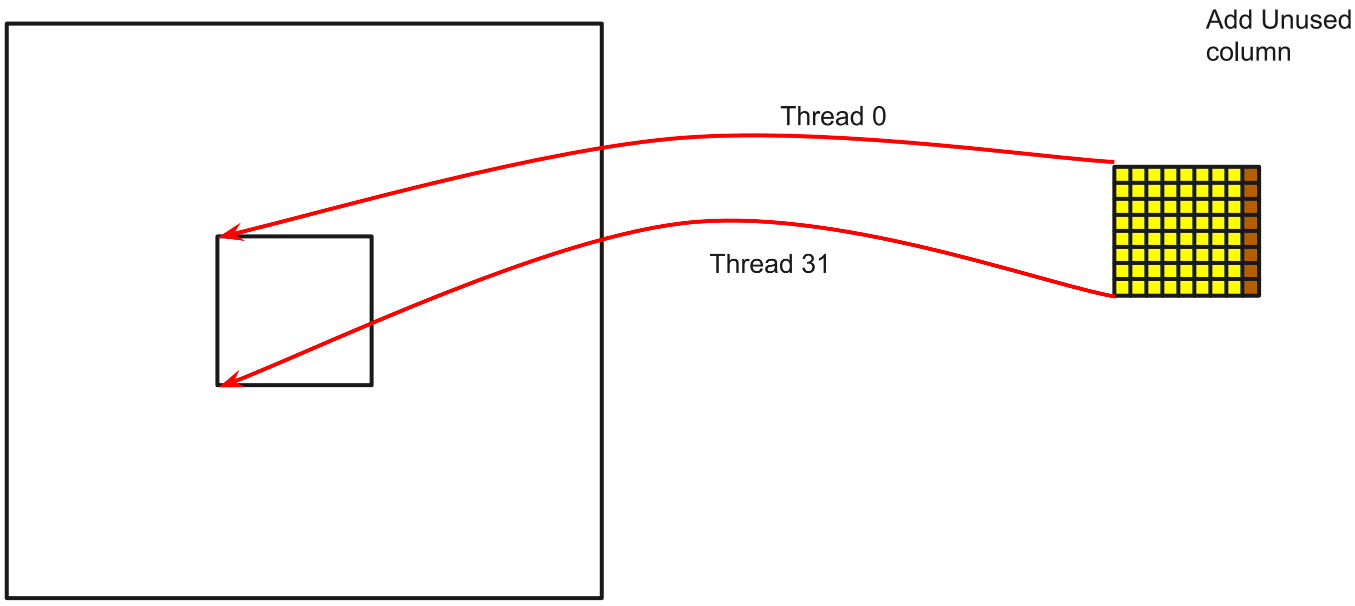
Transpose

- No Bank conflicts



Transpose

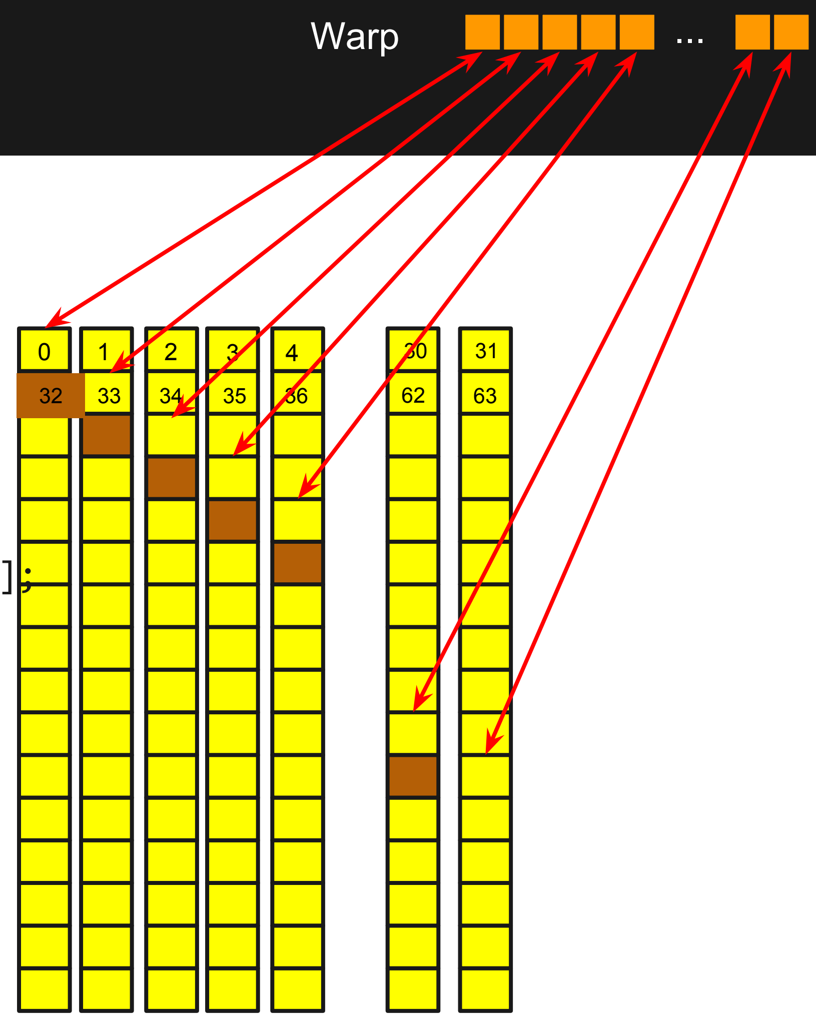
- 32-way Bank conflict!!



Banks

- Resolving bank conflict

```
__shared__ float tile[BLOCKSIZE][BLOCKSIZE+1];
_b[index_out] = tile[tx][ty];
```



Transpose: Shared Memory

No Bank Conflicts

```
__global__ void matrixTransposeSharedwBC(const float *_a,
                                          float *_b)
{
    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y + 1];
    //Rest is same as shared memory version
}
```

Matrix Transpose [GPU Transpose]

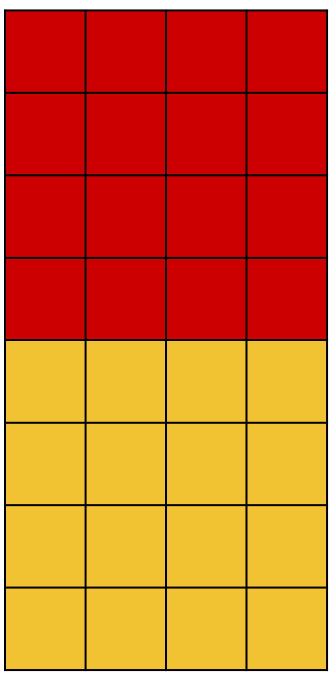
- Very very close to production ready!
- More ways to improve?
  - More work per thread - Do more than one element
  - Loop unrolling

Transpose: Loop Unrolled

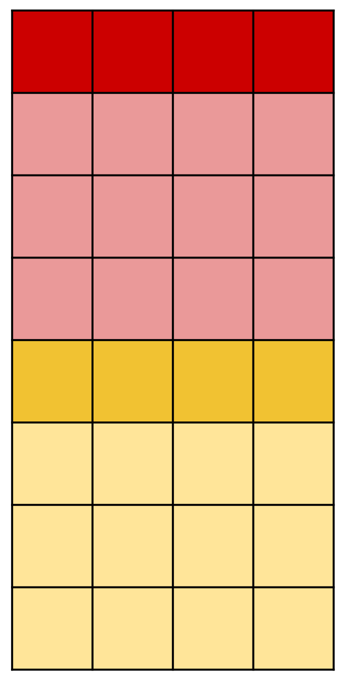
- More work per thread:
  - Threads should be kept light
  - But they should also be saturated
  - Give them more operations
- Loop unrolling
  - Allocate operation in a way that loops can be unrolled by the compiler for faster execution
  - Warp scheduling
    - Kernels can execute 2 instructions simultaneously as long as they are independent

Transpose: Loop Unrolled

- Use same number of blocks, shared memory
- Reduce threads per block by factor (side)



Block Size X = 4
Block Size Y = 4
Threads/Block = 16
Total blocks = 2
Shared mem = 4 x 4



Block Size X = 4 -> TILE
Block Size Y = 1 -> SIDE
Threads/Block = 4
Total blocks = 2
Shared mem = TILE x TILE

Transpose: Loop Unrolled

- Walkthrough
- Host:
  - Same number of blocks
  - Compute new threads per block
- Device:
  - Allocate same shared memory
  - Compute input indices similar to before
  - Copy data to shared memory using loop (k)
    - Unrolled index: add k to y
  - Compute output indices similar to before
  - Copy data from shared memory into global memory
    - Unrolled index: add k to y

Transpose: Loop Unrolled

```
const int TILE = 32; const int SIDE = 8;
__global__ void matrixTransposeUnrolled(const float *_a,
                                              float *_b)
{
    __shared__ float mat[TILE][TILE + 1];
    int x = blockIdx.x * TILE + threadIdx.x;
    int y = blockIdx.y * TILE + threadIdx.y;
#pragma unroll
    for(int k = 0; k < TILE ; k += SIDE) {
        if(x < rows && y + k < cols)
            mat[threadIdx.y + k][threadIdx.x] = a[((y + k) * rows) + x];
    }
    __syncthreads();
    //continuing on next slide
}
```

Transpose: Loop Unrolled

```
const int TILE = 32; const int SIDE = 8;
__global__ void matrixTransposeUnrolled(const float *_a,
                                        float *_b)
{
   //continuing from previous slide
    __syncthreads();
    x = blockIdx.y * TILE + threadIdx.x;
    y = blockIdx.x * TILE + threadIdx.y;
#pragma unroll
    for(int k = 0; k < TILE; k += SIDE)
    {
        if(x < cols && y + k < rows)
            b[(y + k) * cols + x] = mat[threadIdx.x][threadIdx.y + k];
    }
}
```

Performance for 4k x 4k Matrix Transpose (K20)

	Time (ms)	Bandwidth (GB/s)	Step Speedup	Speed Up vs CPU
CPU	166.2	0.807		
Naive Transpose	2.456	54.64	67.67	67.67
Coalesced Memory	1.712	78.37	1.434	97.08
Bank Conflicts	1.273	105.38	1.344	130.56
Loop Unrolling	0.870	154.21	1.463	191.03

Device to Device Memcpy: 167.10 GB/s