

CUDA Performance Optimization

GPU Programming with CUDA

April 25-27, 2016 | Jiri Kraus (NVIDIA) based on work by Andrew V. Adinetz

What you will learn:

- What is memory coalescing
- What is branch divergence

What you will not learn (in this session):

- Exposing enough parallelism
- Expressing data locality

Motivating Example: Matrix Transpose

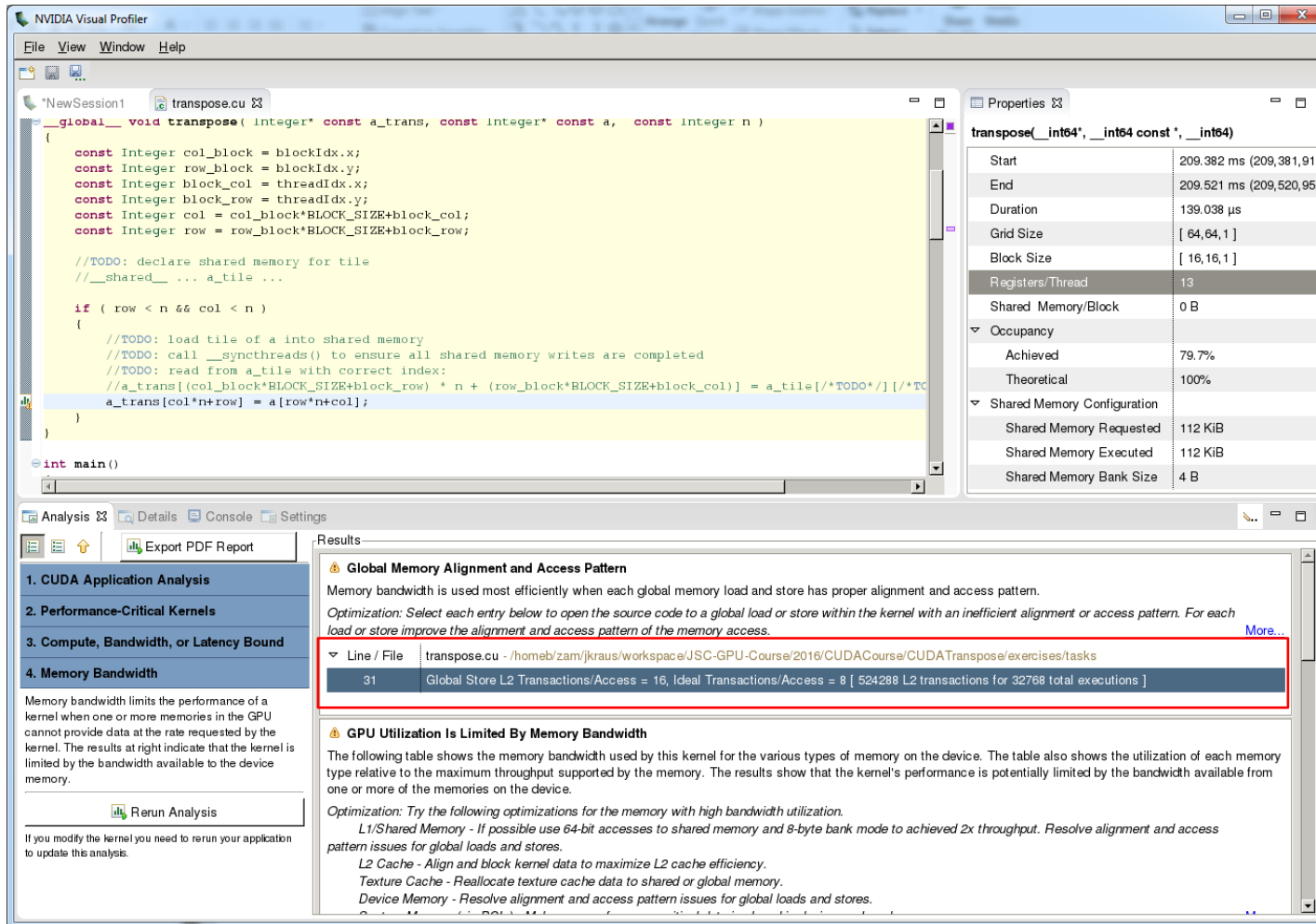
CPU VERSION:

```
void transpose(  
    Integer* a_trans,  
    Integer* a,  
    Integer n ) {  
    for ( Integer row = 0; row < n; ++row ) {  
        for ( Integer col = 0; col < n; ++col )  
        {  
            a_trans[col][row] = a[row][col];  
        }  
    }  
}
```

CUDA VERSION:

```
__global__ void transpose(  
    Integer* a_trans,  
    Integer* a,  
    Integer n ) {  
    Integer row = blockIdx.y*blockDim.y+threadIdx.y;  
    Integer col = blockIdx.x*blockDim.x+threadIdx.x;  
    if ( row < n && col < n )  
        a_trans[col][row] = a[row][col];  
}
```

Motivating Example: Matrix Transpose (demo)



The screenshot displays the NVIDIA Visual Profiler interface. The main window shows the CUDA source code for a matrix transpose kernel. The code includes constants for block sizes and thread indices, and a kernel function that reads from a shared memory tile and writes to global memory. The main function calls the kernel.

```

__global__ void transpose( Integer* const a_trans, const Integer* const a, const Integer n )
{
    const Integer col_block = blockDim.x;
    const Integer row_block = blockDim.y;
    const Integer block_col = threadIdx.x;
    const Integer block_row = threadIdx.y;
    const Integer col = col_block*BLOCK_SIZE+block_col;
    const Integer row = row_block*BLOCK_SIZE+block_row;

    //TODO: declare shared memory for tile
    //__shared__ ... a_tile ...

    if ( row < n && col < n )
    {
        //TODO: load tile of a into shared memory
        //TODO: call __syncthreads() to ensure all shared memory writes are completed
        //TODO: read from a_tile with correct index:
        //a_trans[col_block*BLOCK_SIZE+block_row] * n + (row_block*BLOCK_SIZE+block_col) = a_tile[/*TODO*/][/*TC
        a_trans[col*n+row] = a[row*n+col];
    }
}

int main()
  
```

The Properties window on the right shows the following details for the kernel:

transpose(_int64', _int64 const', _int64)	
Start	209.382 ms (209,381,91)
End	209.521 ms (209,520,95)
Duration	139.038 μs
Grid Size	[64, 64, 1]
Block Size	[16, 16, 1]
Registers/Thread	13
Shared Memory/Block	0 B
Occupancy	
Achieved	79.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

The Results window shows the following analysis:

Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File	transpouse.cu - /home/zam/jkraus/workspace/JSC-GPU-Course/2016/CUDACourse/CUDATranspose/exercises/tasks
31	Global Store L2 Transactions/Access = 16, Ideal Transactions/Access = 8 [524288 L2 transactions for 32768 total executions]

GPU Utilization Is Limited By Memory Bandwidth

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. The results show that the kernel's performance is potentially limited by the bandwidth available from one or more of the memories on the device.

Optimization: Try the following optimizations for the memory with high bandwidth utilization.

- L1/Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput. Resolve alignment and access pattern issues for global loads and stores.
- L2 Cache - Align and block kernel data to maximize L2 cache efficiency.
- Texture Cache - Reallocate texture cache data to shared or global memory.
- Device Memory - Resolve alignment and access pattern issues for global loads and stores.

Memory Transactions and Coalescing

- Access to global memory triggers transactions
 - size of 32 or 128 bytes
 - Transaction are aligned to its size
 - Transaction are always R/W fully
- Coalescing
 - adjacent threads can share transactions

$$\text{degree of coalescing} = \frac{\text{\#bytes requested}}{\text{\#bytes read}}$$

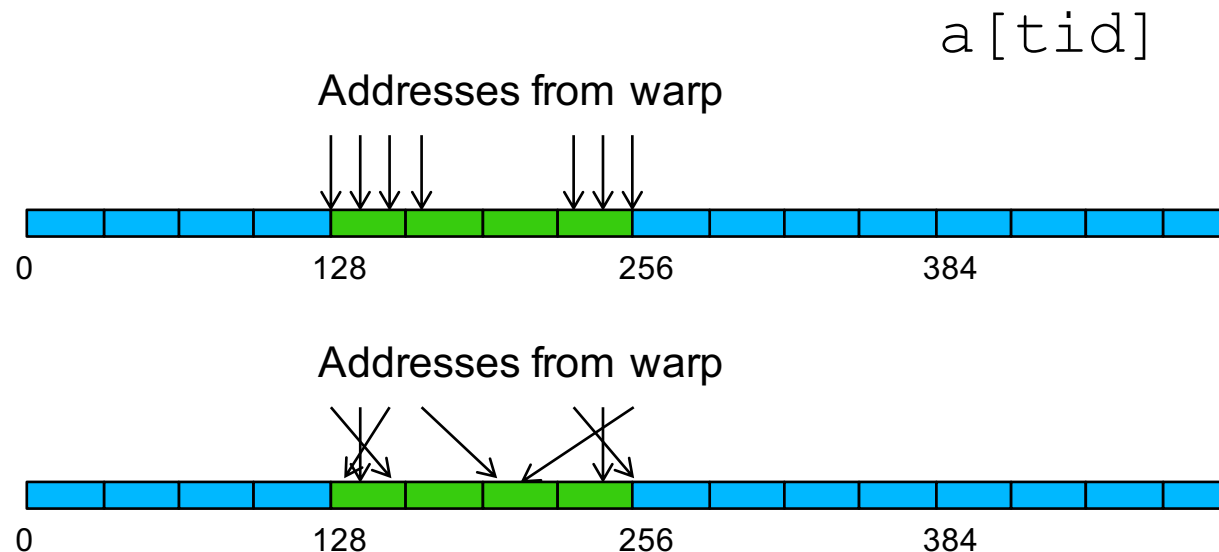
- more memory read than used => performance penalty

Coalescing Details

- Details in CUDA Programming Guide
- L1-cached access
 - 128-byte transactions
 - Used local memory for CC 3.x
- L2-cached access
 - 32-byte transactions
 - default for CC 3.x (Kepler) global memory

L2-Cached Thread Index Access

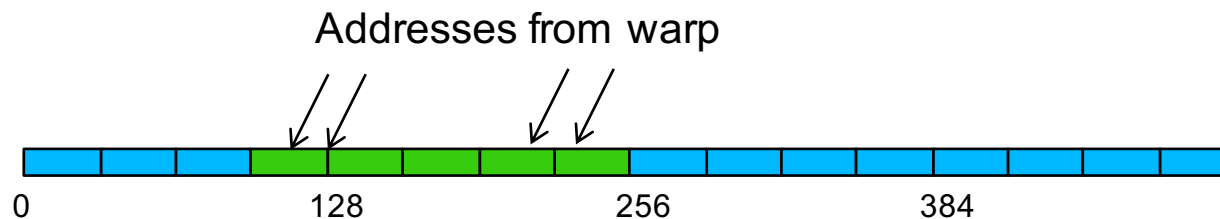
- 32 adjacent threads requesting 32 aligned or permuted 4 byte words
- All addresses fall within 4 segments
- Bus utilization: 100%
- Transactions: 4



L2-Cached Shifted Access

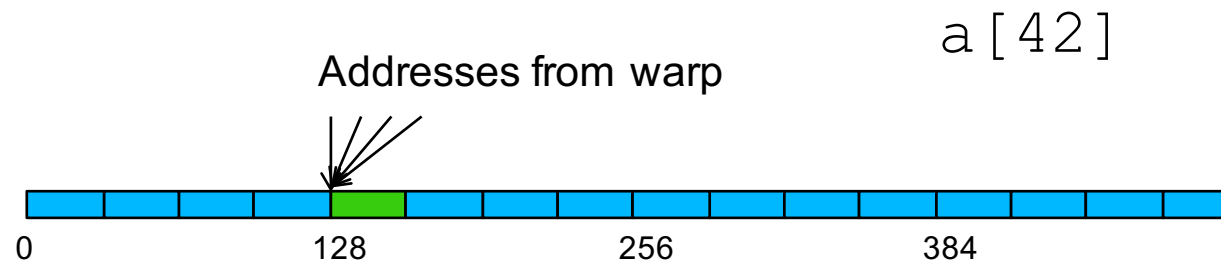
- 32 adjacent threads requesting 32 aligned 4 byte words
- All addresses fall within 5 segments
- 160 bytes move across bus while 128 bytes are required
- Bus utilization: 80%
- Transactions: 5

`a[tid - 1]`



L2-Cached Single Access

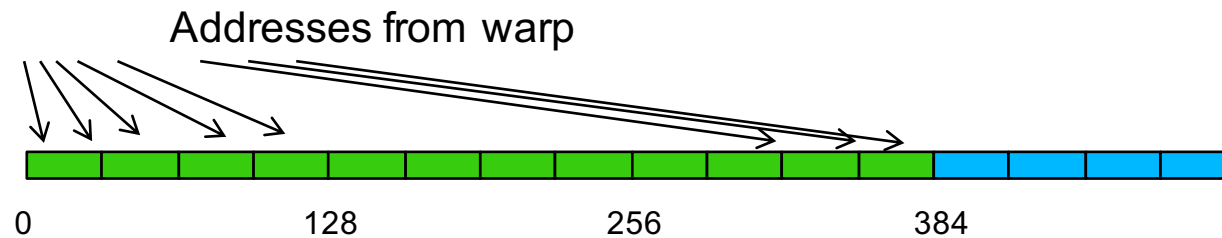
- 32 adjacent threads requesting the same 4 byte word
- Addresses fall in 1 segment
- Warp requires 4 bytes but 32 bytes transferred
- Bus utilization is only 12.5%



L2-Cached Strided Access

- 32 adjacent threads requesting 32 4-byte words with stride 3
- All addresses fall in 12 segments
- Bus utilization: 33%
- Transactions: 12

`a[3 * tid]`

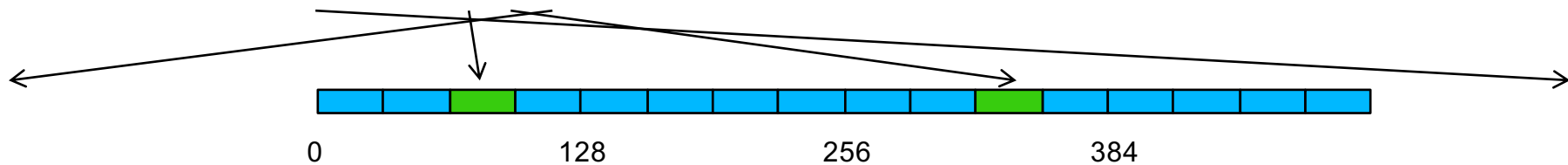


- `struct {float x, y, z;} a; ... a[tid].x`
 - use structure-of-arrays (SoA)
- `float a[M][N]; ... a[tid][42]`
 - multi-dimensional arrays: pay attention to coalescing

L2-Cached Fully Random Access

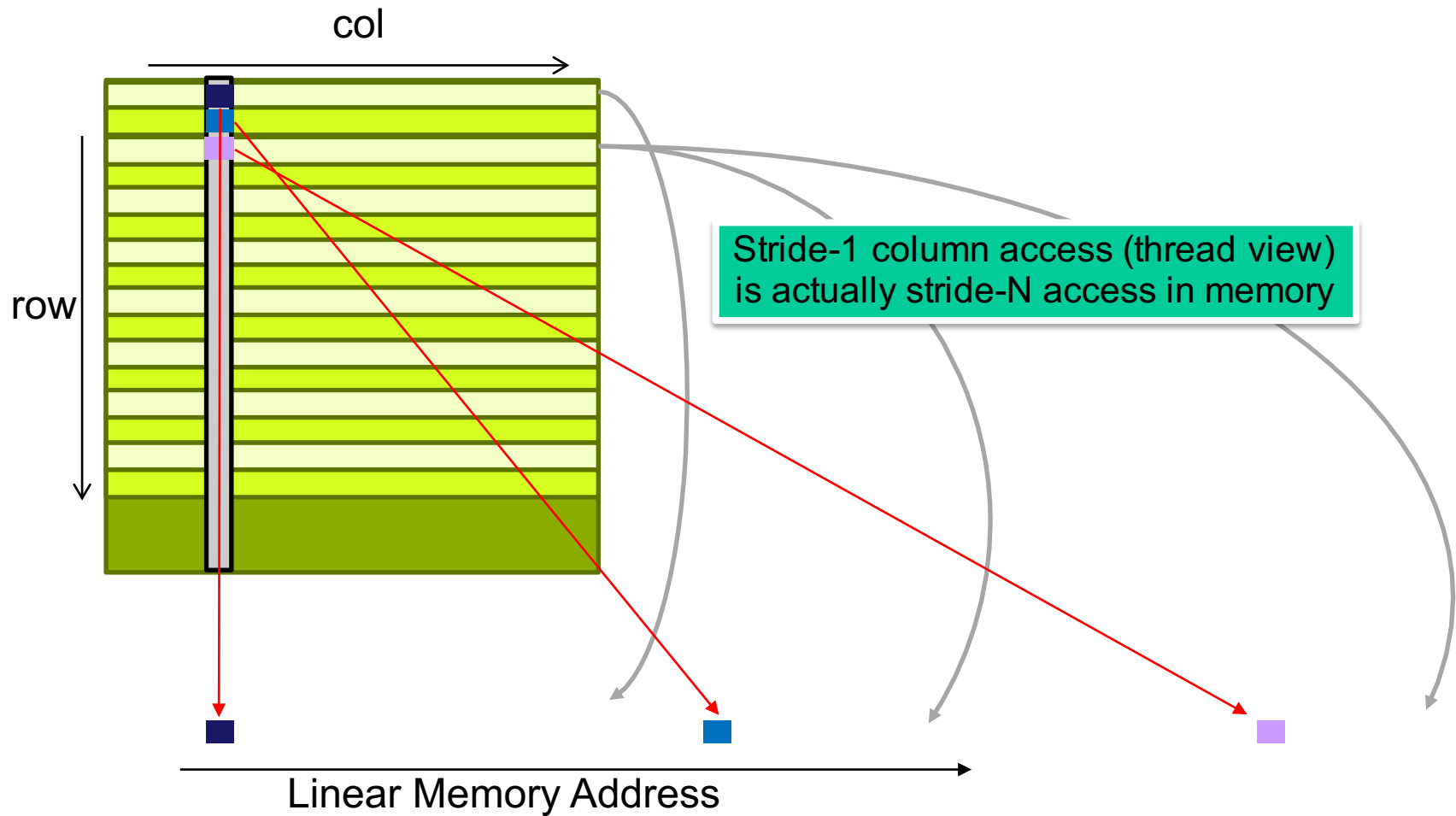
- 32 adjacent threads requesting 32 4-byte random words
- All addresses fall in 32 segments
- Bus utilization: only 12.5%
- Transactions: 32

`a[b[tid]]`



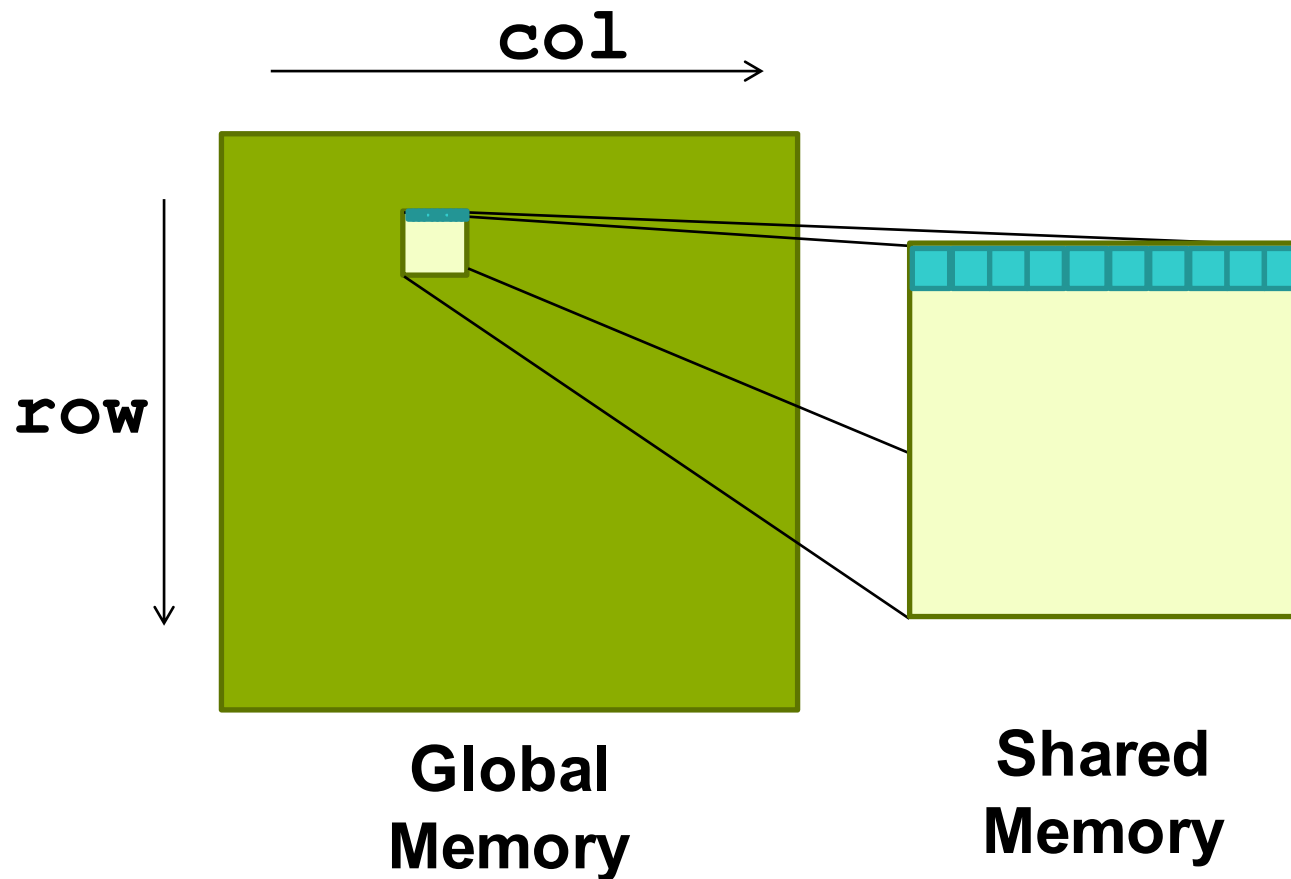
- pointer chasing: lists, trees etc.

Matrix Transpose Access Pattern



Source: [GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero \(NVIDIA\)](#)

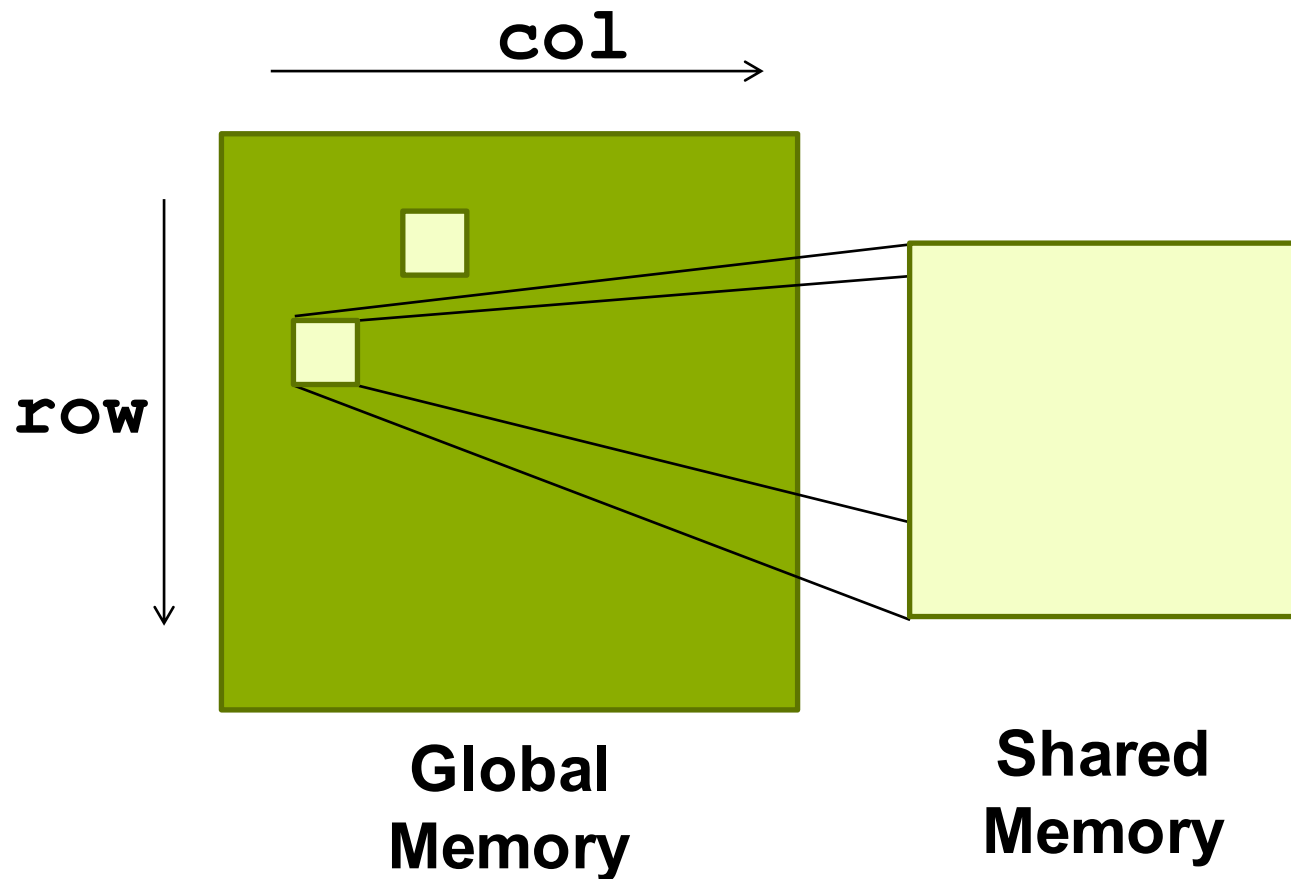
Matrix Transpose using shared memory



- Row is loaded fully coalesced

Source: [GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero \(NVIDIA\)](#)

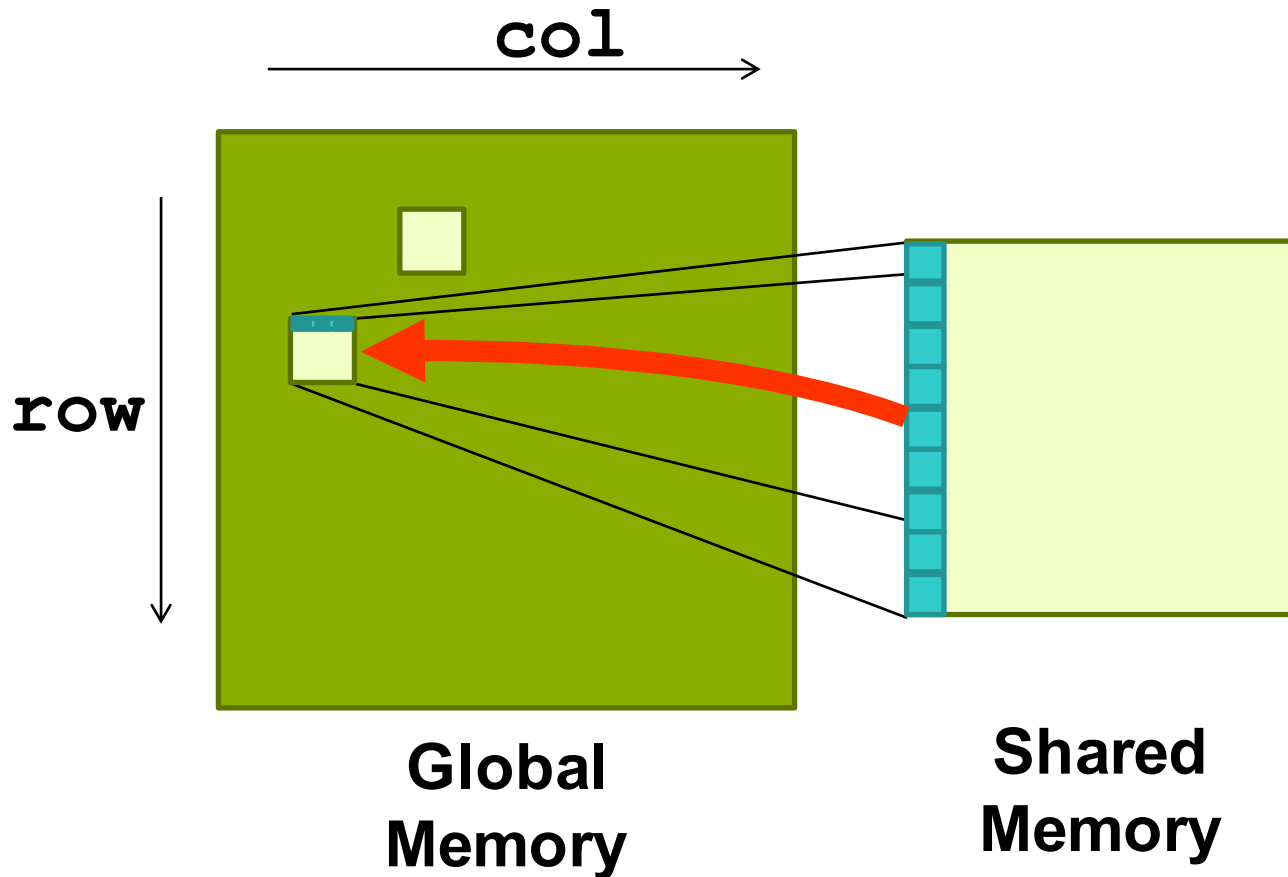
Matrix Transpose using shared memory



- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal

Source: [GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero \(NVIDIA\)](#)

Matrix Transpose using shared memory

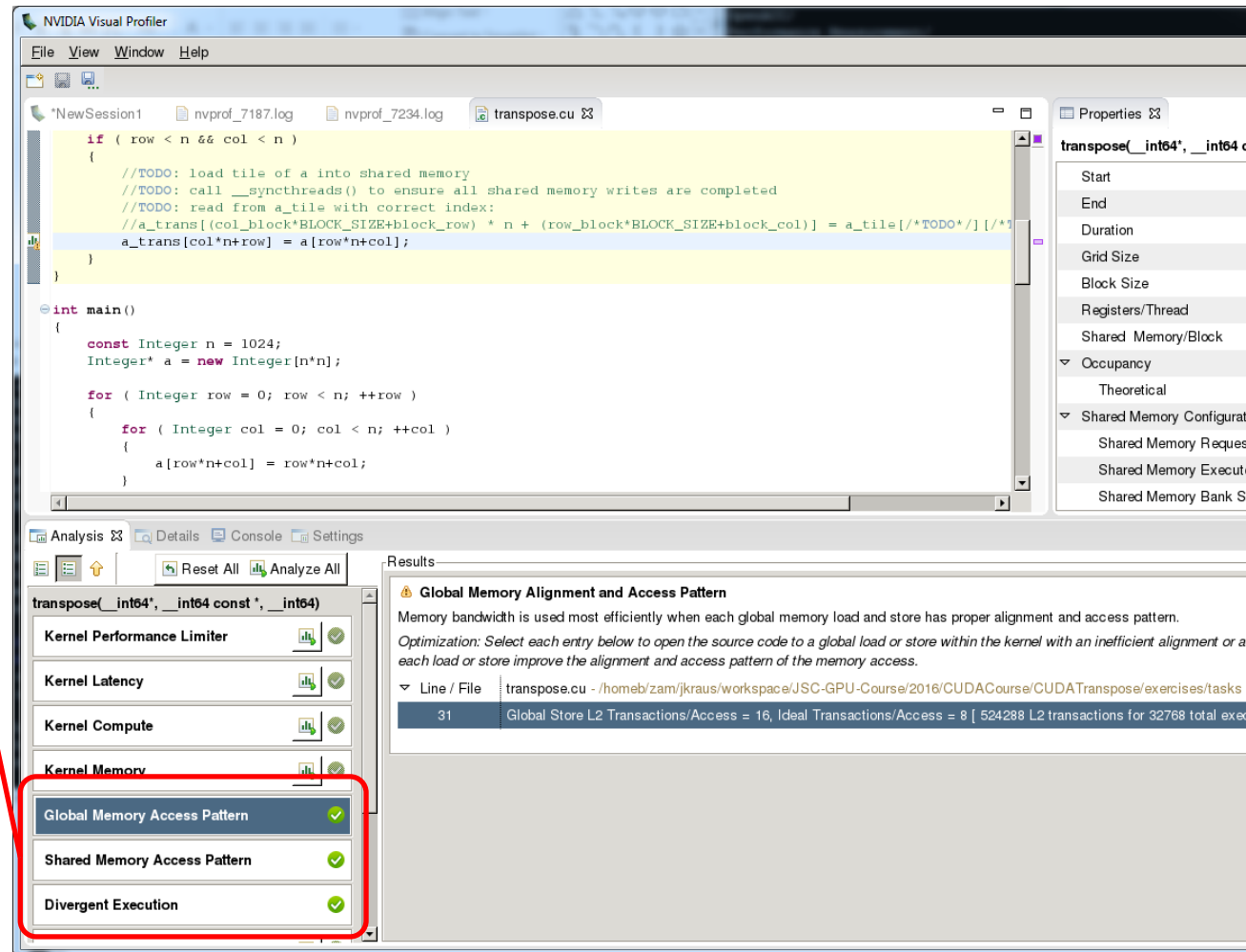


- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal
- Column of shared is written to a row of the matrix
 - Transposes the Block
 - Coalesced Write

Source: [GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero \(NVIDIA\)](#)

Using NVidia Visual Profiler

- Experiments:
 - memory access pattern
 - divergent execution
- Show in source code
 - `nvcc -lineinfo ...`



Task: Coalesced Matrix Transpose

- Use shared memory to coalesced writes to `a_trans`

- Follow TODOs in

`CUDATranspose\exercises\tasks\transpose.cu`

- Check solution with “Memory Access Pattern” experiment

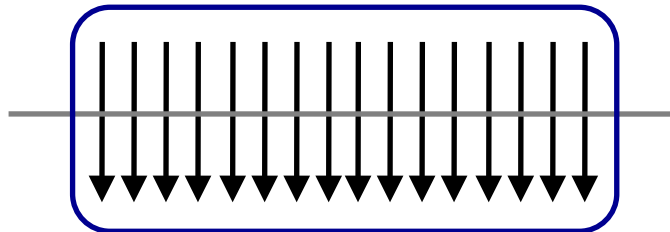
- Solution is in

`CUDATranspose\exercises\solutions\transpose.cu`

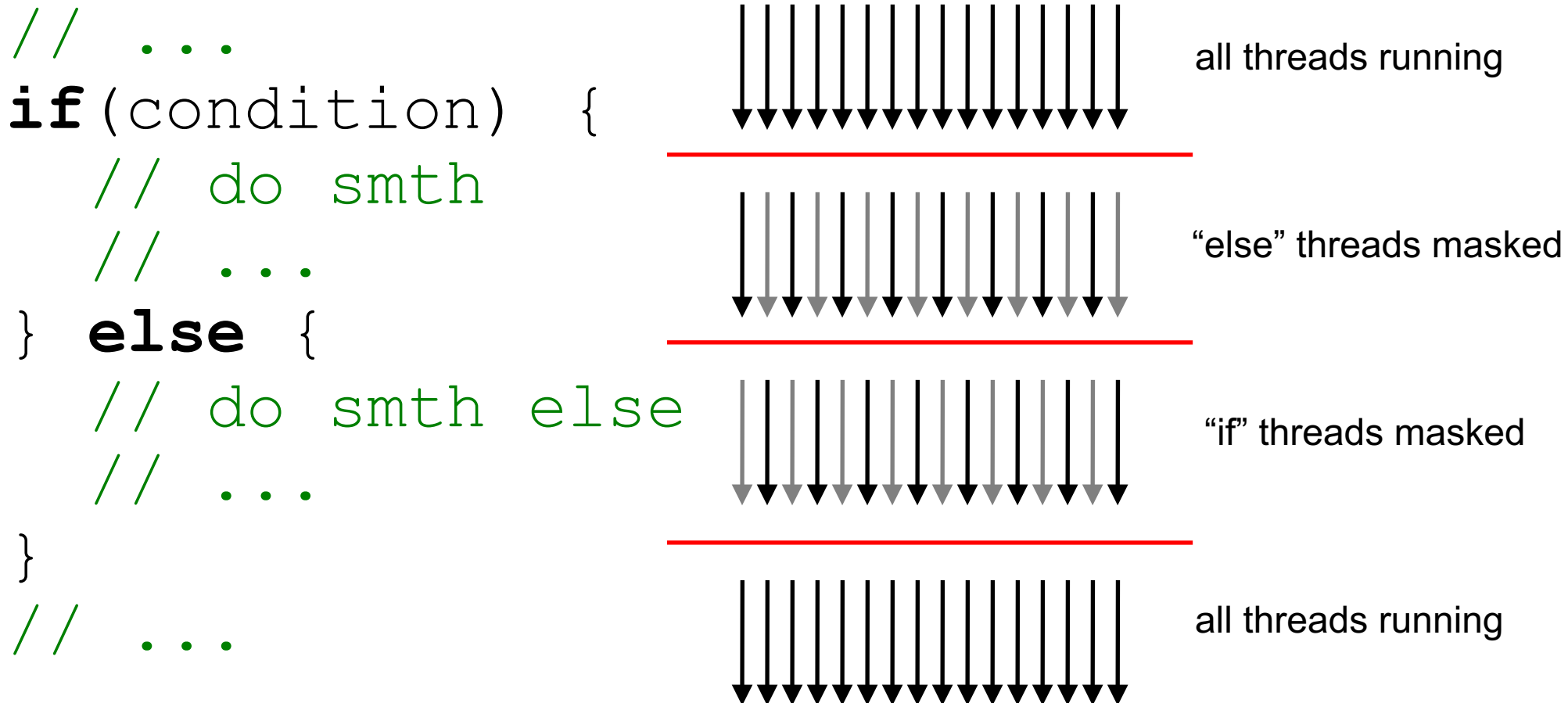
- Slides are in `CUDATranspose\slides\CUDATranspose.pdf`

Warps

- GPUs use the Single Instruction Multiple Threads (SIMT) execution
 - functionally transparent to the programmer
 - but has performance implications
- warp
 - group of synchronously executing threads
 - neighbor threads (mostly x dimension)
 - 32 threads (NVIDIA), 64 threads (AMD, =wavefronts)
 - basic unit of scheduling



Branch Divergence Within Warp



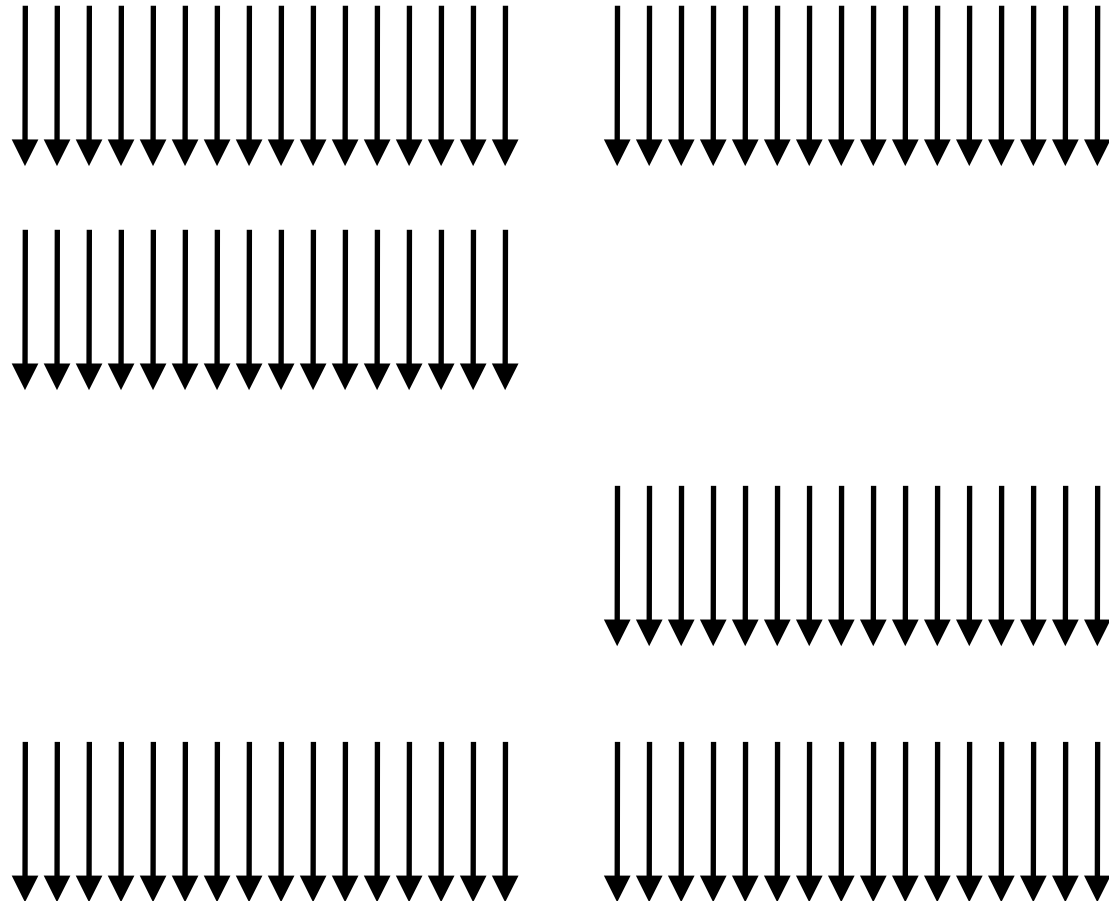
divergence *within* warp => performance penalty

```
if (threadIdx.x % 2 == 0) ...
```

Branch Divergence Between Warps

```

// ...
if (condition) {
    // do smth
    // ...
} else {
    // do smth else
    // ...
}
// ...
  
```



divergence *between* warps => no penalty

```
if (blockIdx.x % 2 == 0) ...
```

Conclusion

- To achieve coalesced global memory access
 - Try to use shared memory
 - Look for different way of storage or better algorithm
- Avoid divergent branches
- Use the tools