



# CUDA Performance Optimization

## GPU Programming with CUDA

April 20-11, 2015 | 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

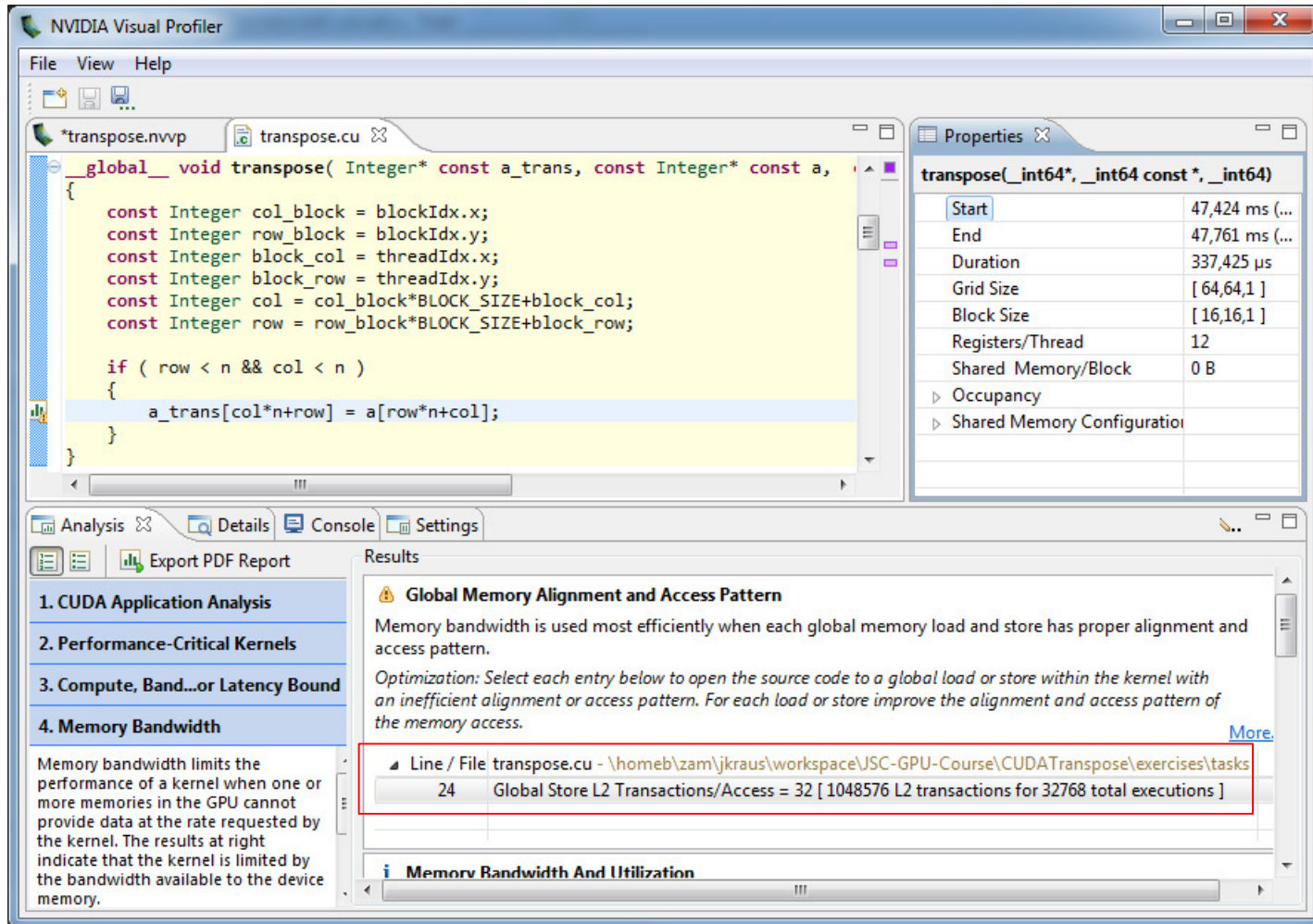
## CPU VERSION:

```
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 source code for a CUDA kernel named `transpose`. The kernel takes a pointer to a 2D array `a_trans` and a constant integer `n` as input. It calculates the row and column indices for each thread based on the block dimensions and thread index. The kernel then performs a transpose operation by swapping the row and column indices in the array access.

```

_global__ void transpose( Integer* const a_trans, const Integer* const a,
{
    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;

    if ( row < n && col < n )
    {
        a_trans[col*n+row] = a[row*n+col];
    }
}
  
```

The Properties window on the right shows the execution details for the `transpose(_int64*, _int64 const *, _int64)` kernel. The execution time is 47,424 ms (Start) to 47,761 ms (End), with a duration of 337,425 μs. The grid size is [64,64,1] and the block size is [16,16,1]. The kernel uses 12 registers per thread and 0 B of shared memory per block.

The Analysis window shows the results of the performance analysis. The first section, "Global Memory Alignment and Access Pattern", explains that memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. It provides an optimization tip: "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." The second section, "Memory Bandwidth And Utilization", shows the results for the kernel. The line number 24 is highlighted, indicating a global store. The results show that the kernel has 32 L2 transactions for 1048576 L2 transactions for 32768 total executions.

## 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

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

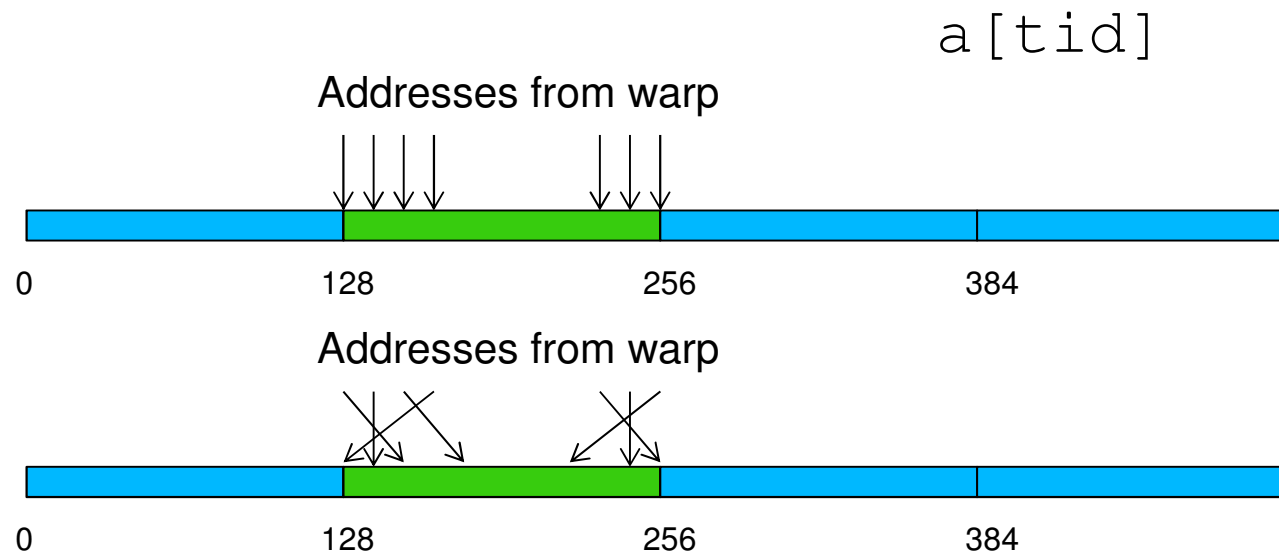
- more memory read than used => performance penalty

## Coalescing Details

- Details in CUDA Programming Guide
- L1-cached access
  - 128-byte transactions
  - default for CC 2.x (Fermi) global memory
  - local memory for CC 2.x and 3.x
- L2-cached access
  - 32-byte transactions
  - default for CC 3.x (Kepler) global memory

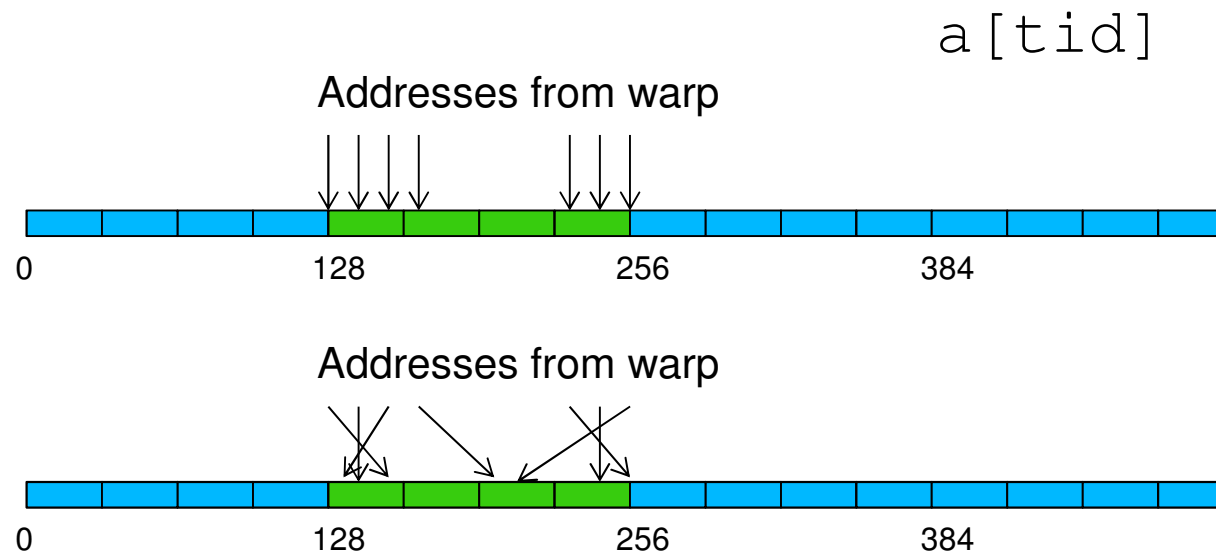
## L1-Cached Thread Index Access

- 32 adjacent threads requesting 32 aligned or permuted 4 byte words
- All addresses fall in one (128byte) cache line
- Bus utilization: 100%
- Transactions: 1



## 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





## L1-Cached Shifted Access

- 32 adjacent threads requesting 32 misaligned consecutive 4 byte words
- All addresses fall in 2 cache lines
- Bus utilization: 50%
- Transactions: 2

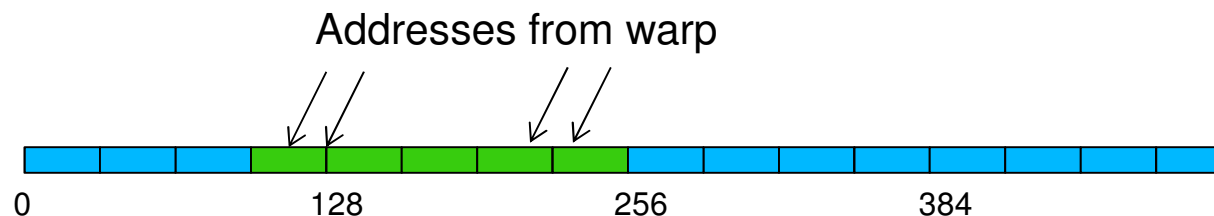


- Stencil computations ( $a[tid - 1] + a[tid + 1]$ )
- $a[M][N]$ ,  $a[42][tid]$ ,  $N$  not multiple of transaction size
- Use `cudaMallocPitch()` to allocate 2D arrays or pad manually

## L2-Cached Shifted Access

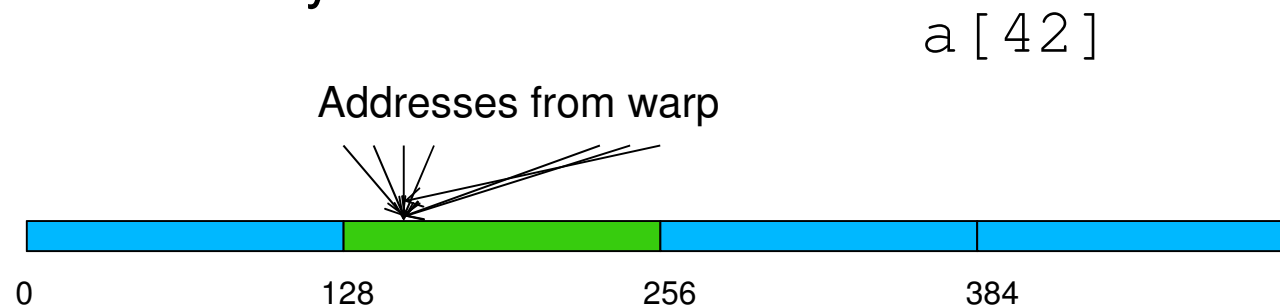
- 32 adjacent threads requesting 32 aligned 4 byte words
- All addresses fall within 5 segments
- 160 bytes move across bus while 128 bytes are required
- Bus utilization: 80%
- Transactions: 5

`a[tid - 1]`



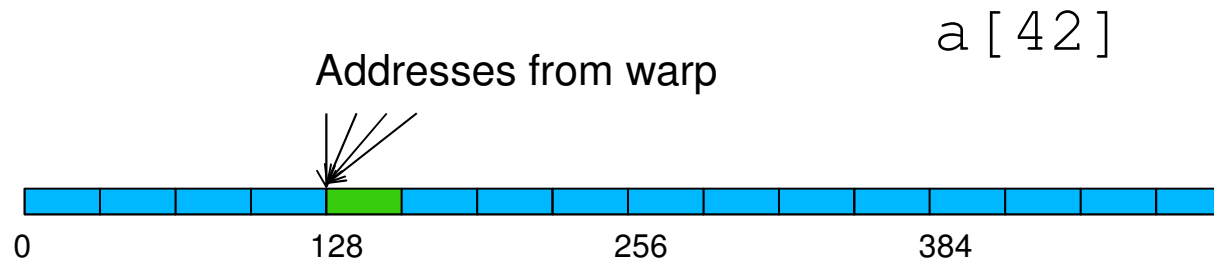
## L1-Cached Single Access

- 32 adjacent threads requesting the same 4 byte word
- Addresses fall in single cache line
- Warp requires 4 bytes but 128 bytes transferred
- Bus utilization is only 3.125%



## L2-Cached Single Access

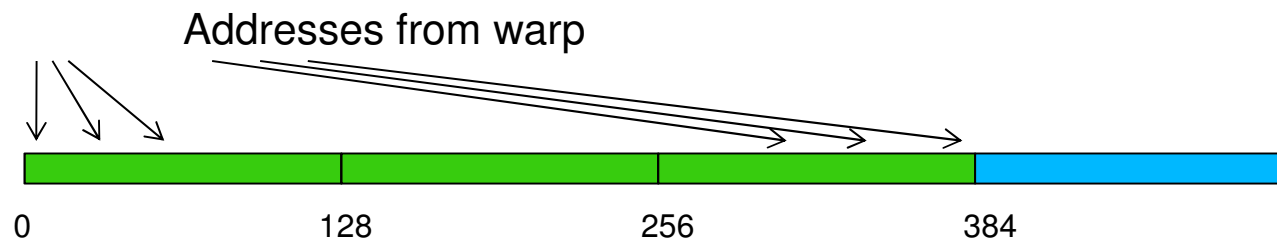
- 32 adjacent threads requesting the same 4 byte word
- Addresses fall in 1 segment
- Warp requires 4 bytes but 32 bytes transferred
- Bus utilization is only 12.5%



## L1-Cached Strided Access

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

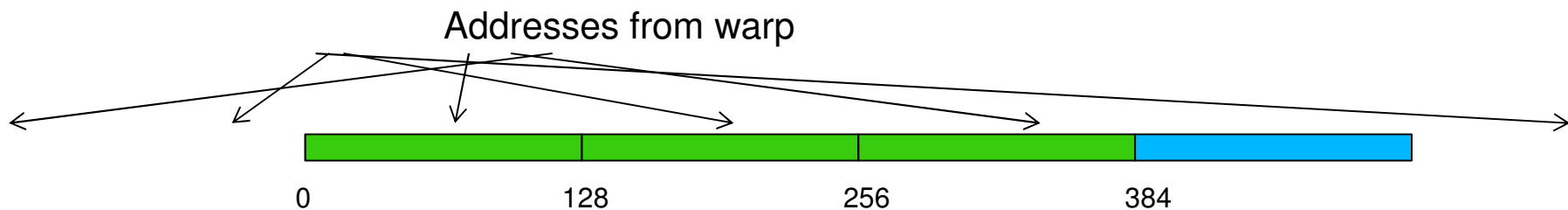
$a[3 * tid]$



- `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

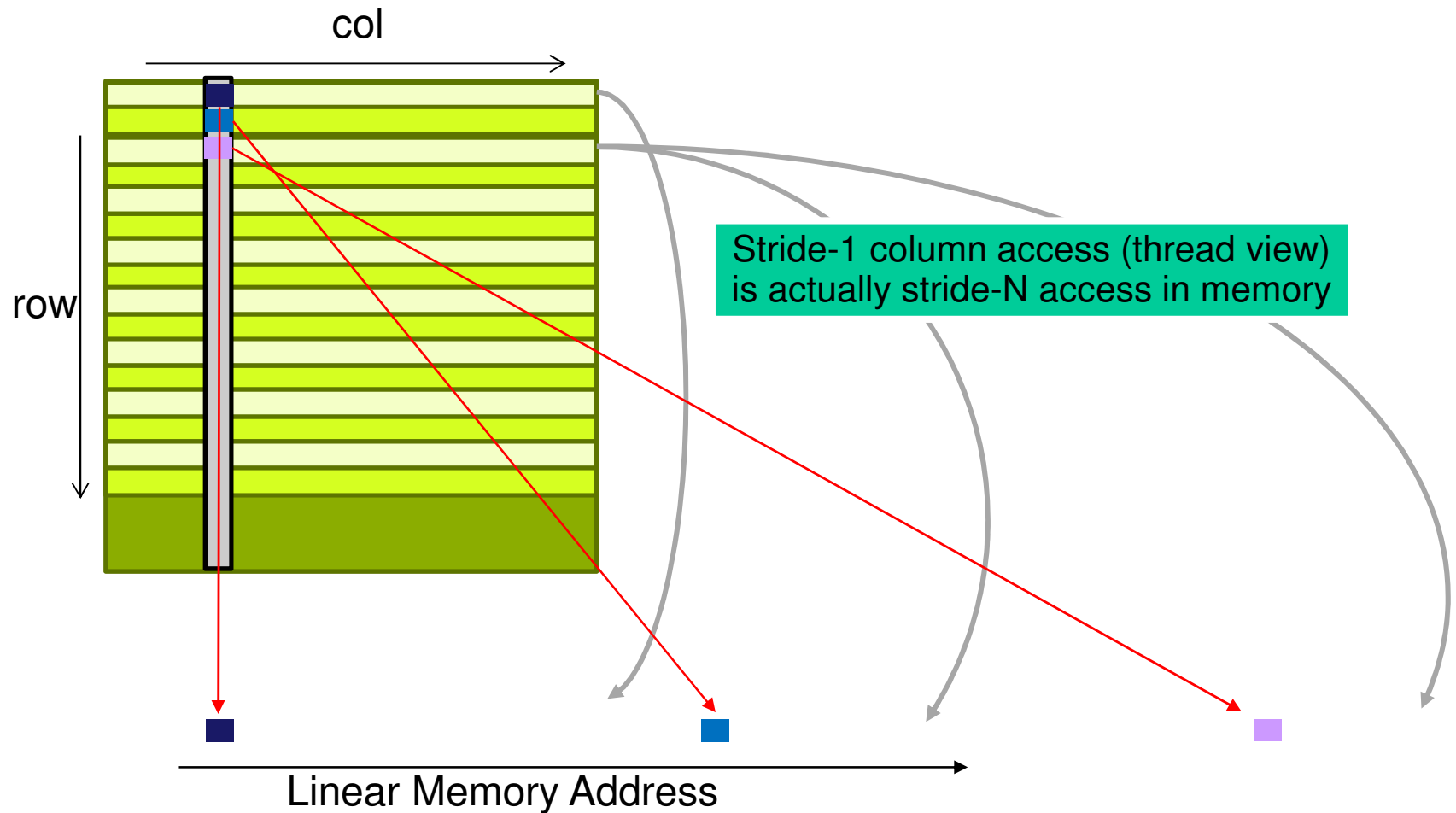
## L1-Cached Fully Random Access

- 32 adjacent threads requesting 32 4-byte random words
- All addresses fall in 32 cache lines
- Bus utilization: only 3.125%
- Transactions: 32

$$a[b[tid]]$$


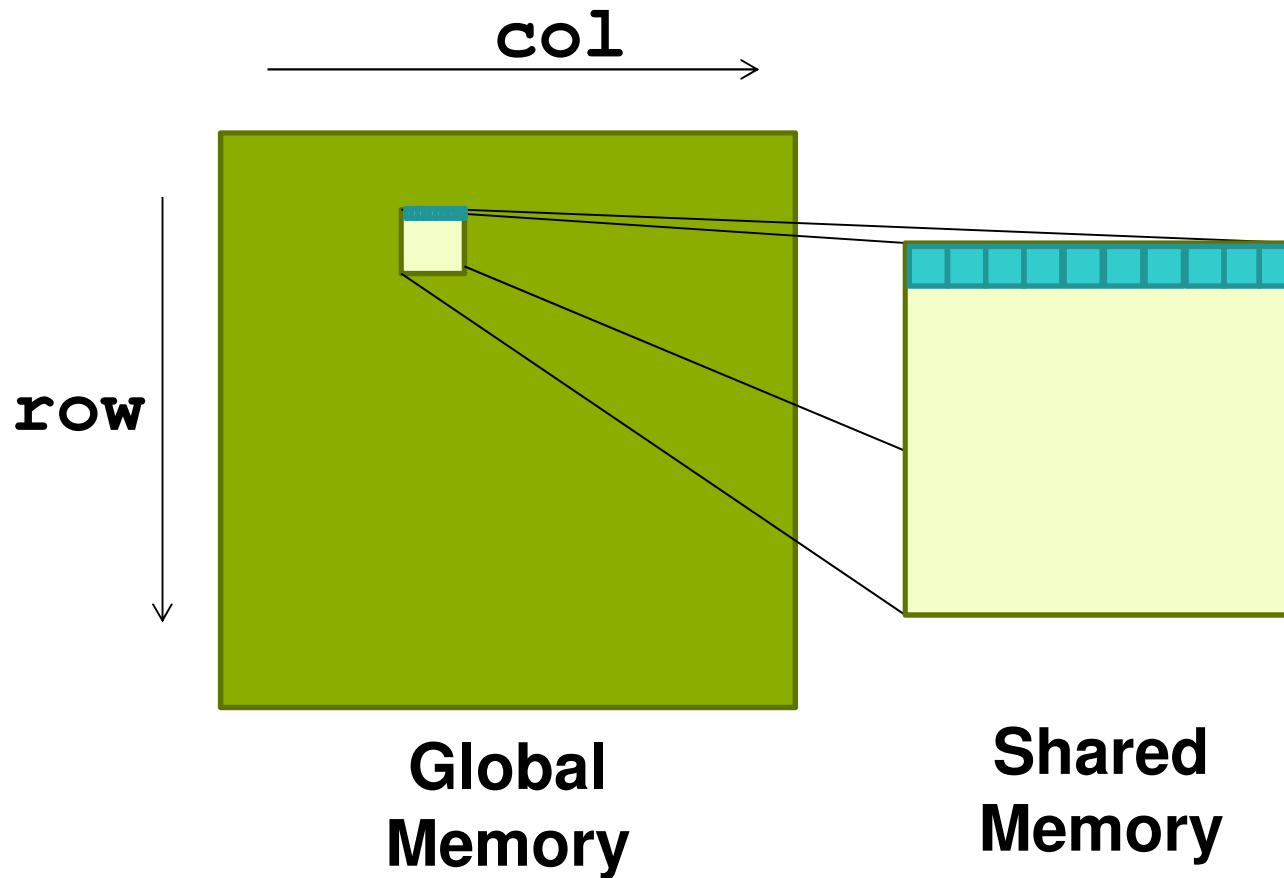
- pointer chasing: lists, trees etc.

# Matrix Transpose Access Pattern



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

# Matrix Transpose using shared memory

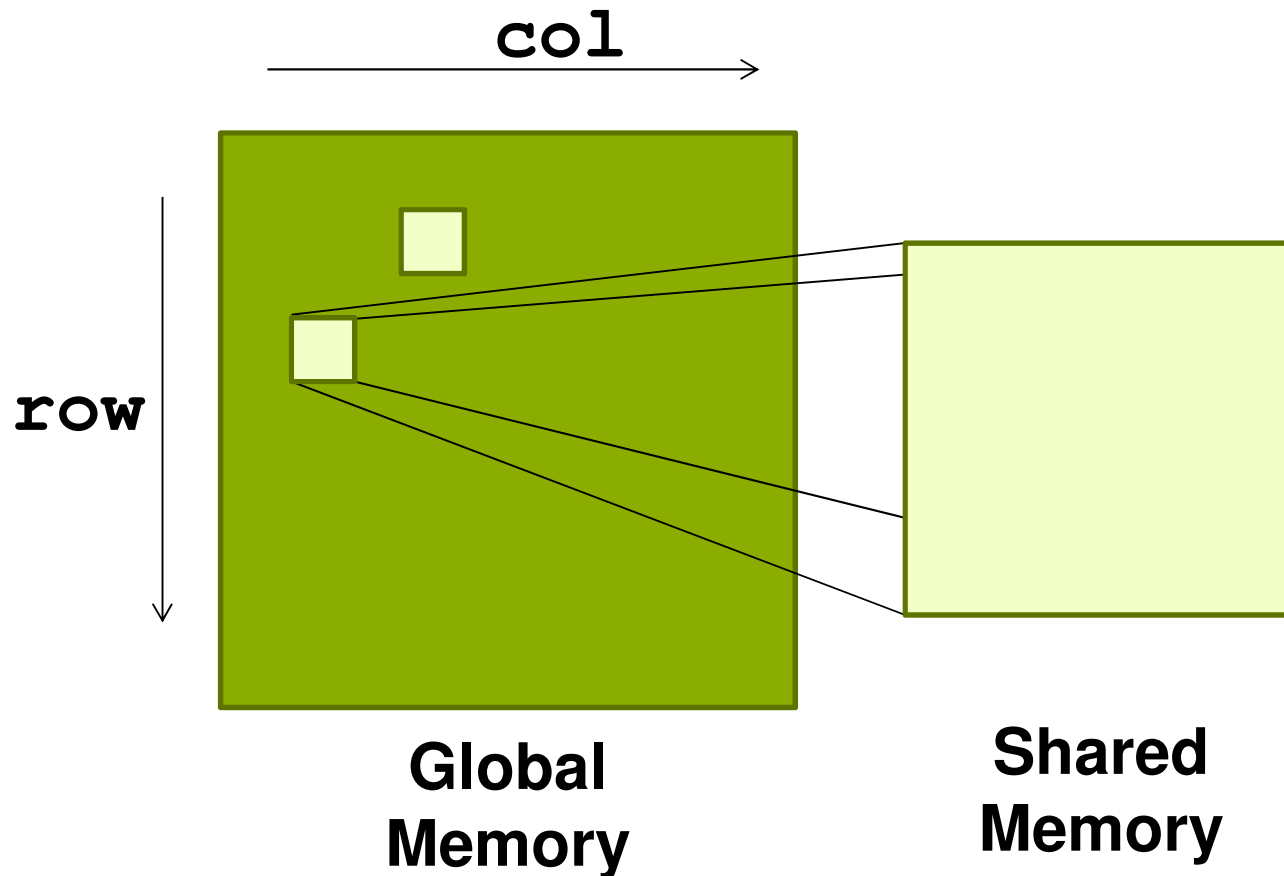


- Row is loaded fully coalesced

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



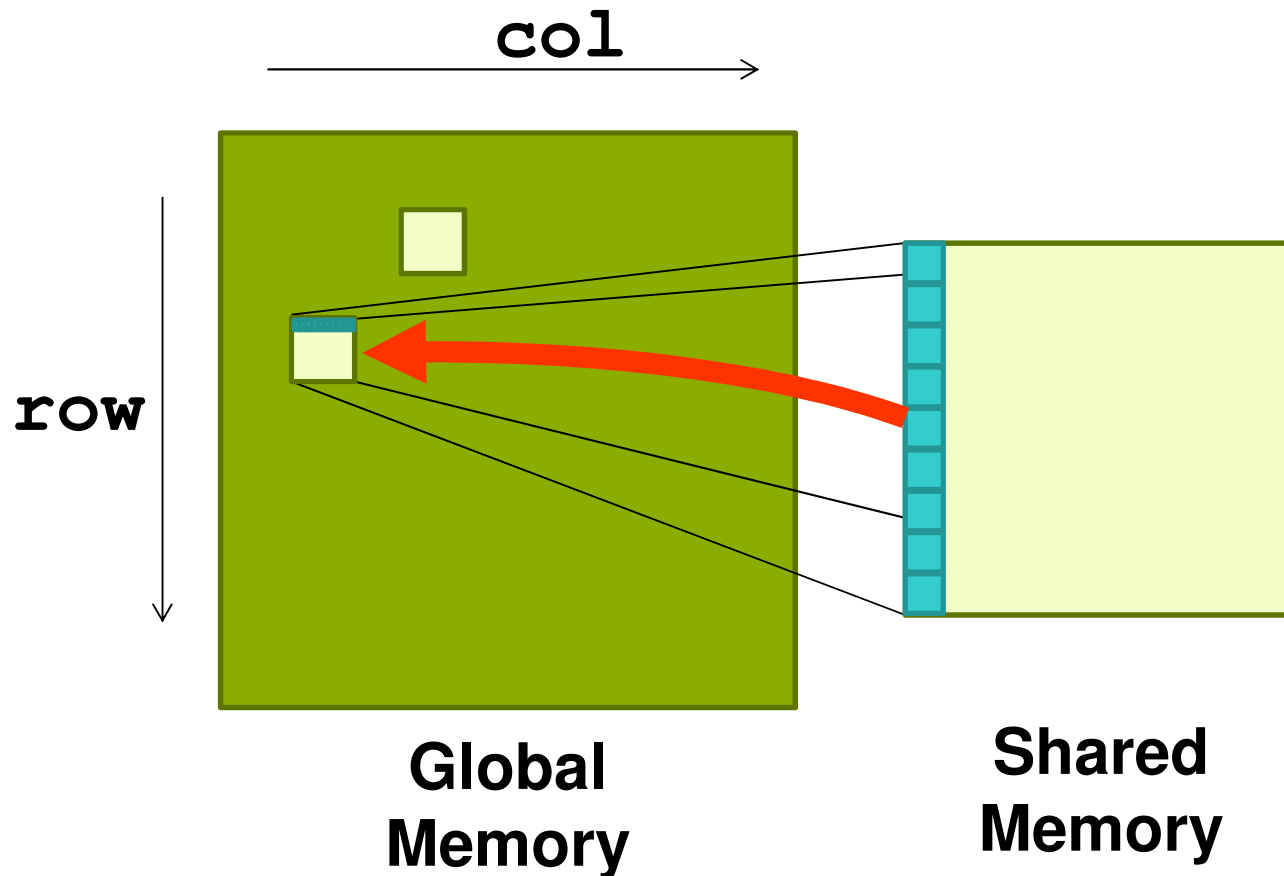
# Matrix Transpose using shared memory



- 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

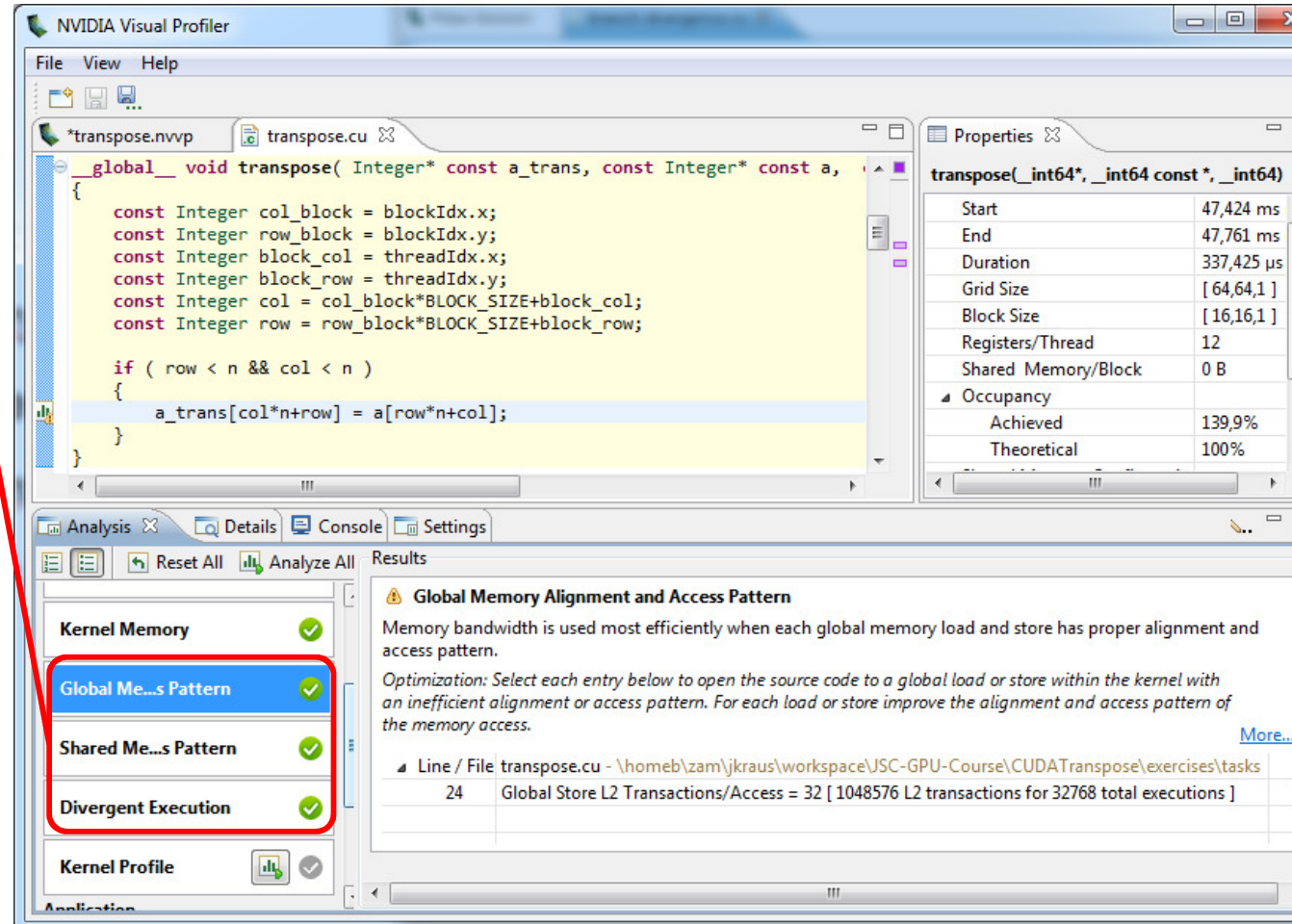


- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal
- Column of shared is written to a row of the matrix
  - Transposes the Block
  - Coalesced Write

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 ...`



The screenshot shows the NVIDIA Visual Profiler interface. The top pane displays the source code for a CUDA kernel named `transpose`. The bottom pane shows the 'Analysis' results, with a red box highlighting the 'Global Memory Pattern', 'Shared Memory Pattern', and 'Divergent Execution' sections, all of which show a green checkmark indicating they are optimized. The 'Global Memory Alignment and Access Pattern' section provides details on memory bandwidth usage and offers optimization advice.

**Source Code:**

```

_global__ void transpose( Integer* const a_trans, const Integer* const a,
{
    const Integer col_block = blockIdx.x;
    const Integer row_block = blockIdx.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;

    if ( row < n && col < n )
    {
        a_trans[col*n+row] = a[row*n+col];
    }
}
  
```

**Analysis Results:**

Category	Status
Kernel Memory	✓
Global Memory Pattern	✓
Shared Memory Pattern	✓
Divergent Execution	✓
Kernel Profile	✓

**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	Description
24 / transpose.cu - \home\zam\jkraus\workspace\JSC-GPU-Course\CUDATranspose\exercises\tasks	Global Store L2 Transactions/Access = 32 [ 1048576 L2 transactions for 32768 total executions ]

## 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

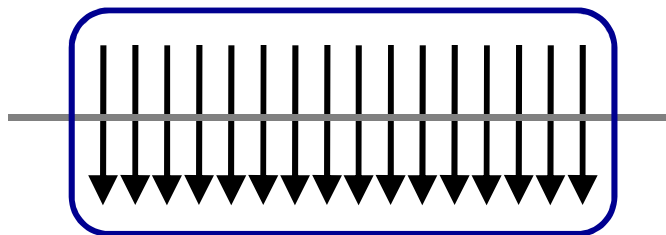
- Solution is in

`CUDATranspose\exercises\solutions\transpose.cu`

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

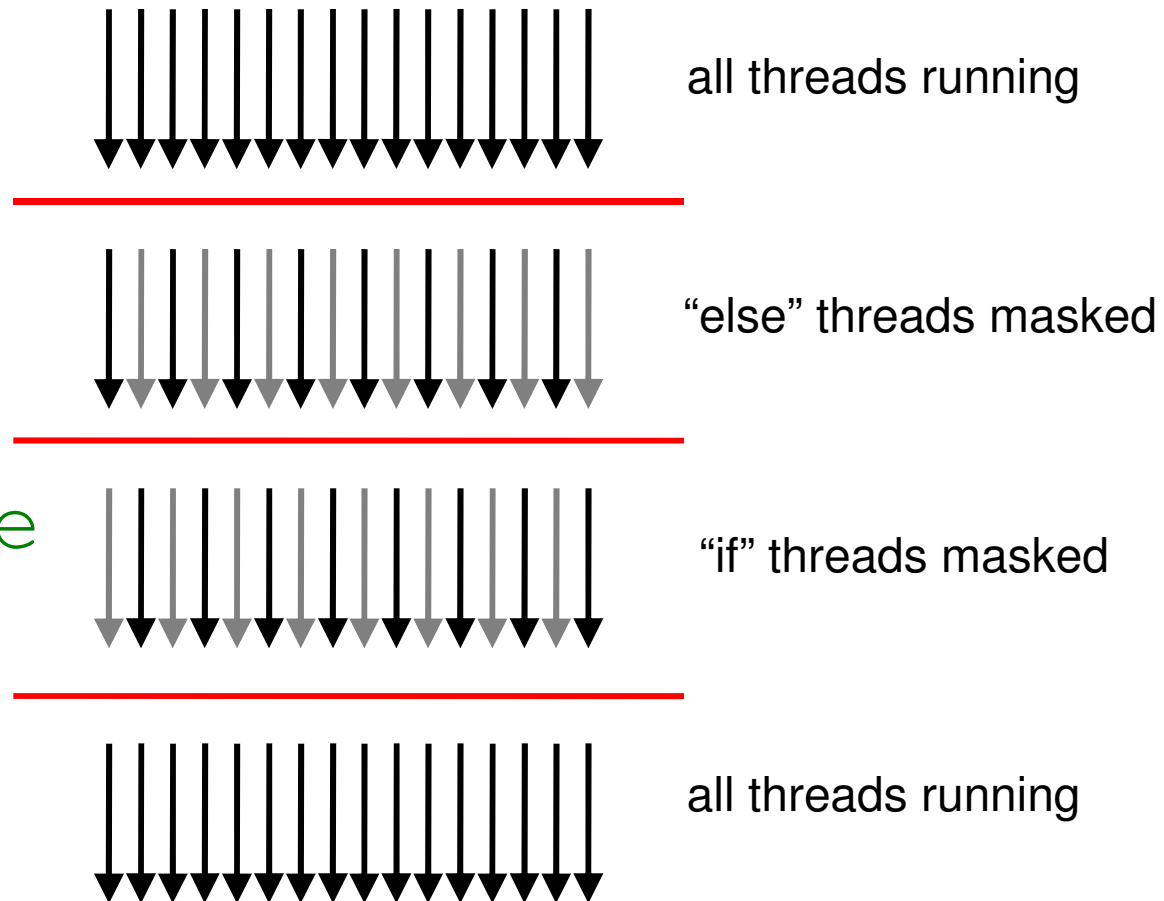
## Warps

- 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

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

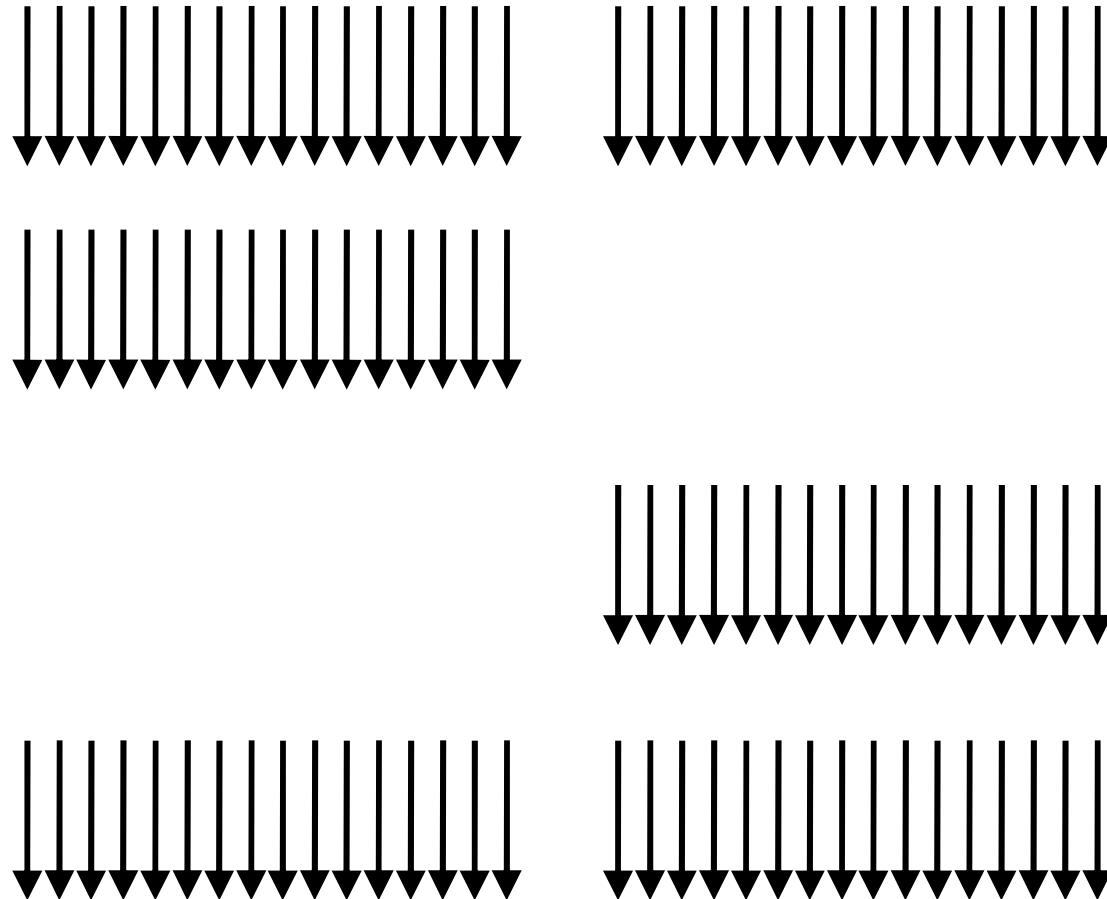


divergence *within* warp => performance penalty

```
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