CME 213

Spring 2017

Eric Darve

Previously in CME213

- Warp = 32 threads
- Block: 1D, 2D, 3D group of threads; max = 1,024 threads
- Grid: 1D, 2D, 3D group of blocks; required for performance and large problems
- Memory access:
  - Analysis based on warp: memory requests made by all threads in warp
  - Map requests from all threads to 128-byte memory segments
  - Peak performance: full segment/cache line used by warp
  - Loss in performance reflected by fraction of cache line not used by warp.

How to calculate the warp ID

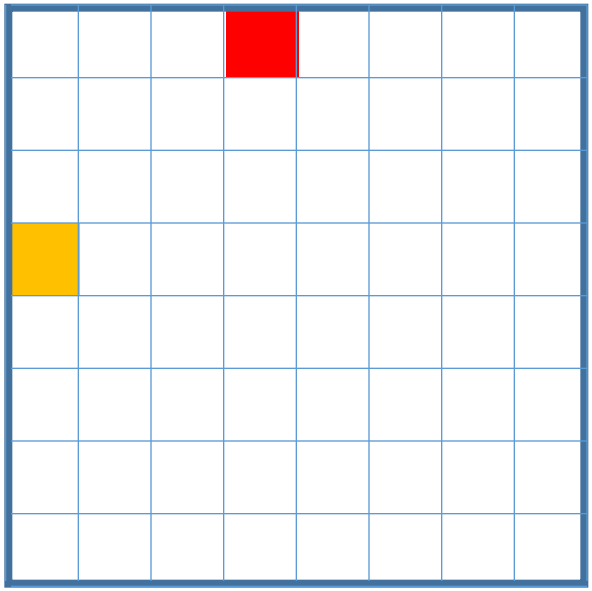
- How do you know if 2 threads are in the same warp?
- Answer: look at the thread ID. Divide by 32: you get the warp ID.
- Thread ID is computed from the 1D, 2d or 3D thread index using:

$$x + yD_x + zD_xD_y$$

```
int tID = threadIdx.x
        + threadIdx.y * blockDim.x
        + threadIdx.z * blockDim.x * blockDim.y;
int warpID = tID/32;
```

Matrix transpose

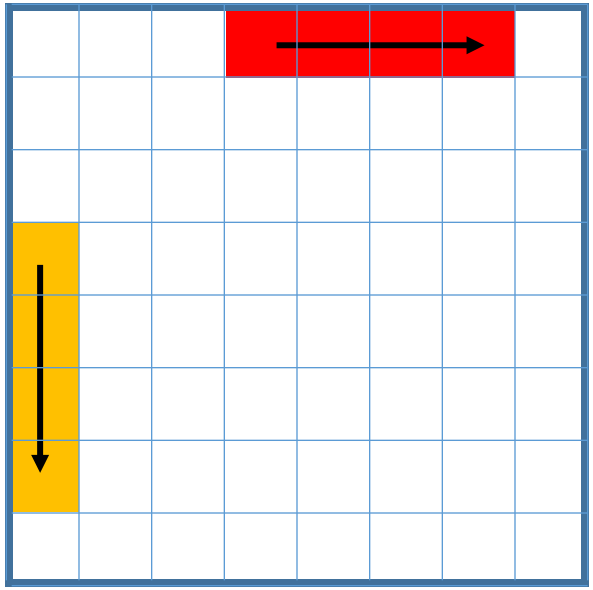
- Let's put all these concepts into play through a specific example: a matrix transpose.
- It's all about bandwidth!
- Even for such a simple calculation, there are many optimizations.



simpleTranspose

```
template<typename T>
__global__
void simpleTranspose(T* array_in, T* array_out, int n_rows, int n_cols) {
    const int tid = threadIdx.x + blockDim.x * blockIdx.x;

    int col = tid % n_cols;
    int row = tid / n_cols;

    if(col < n_cols && row < n_rows) {
        array_out[col * n_rows + row] = array_in[row * n_cols + col];
    }
}
```

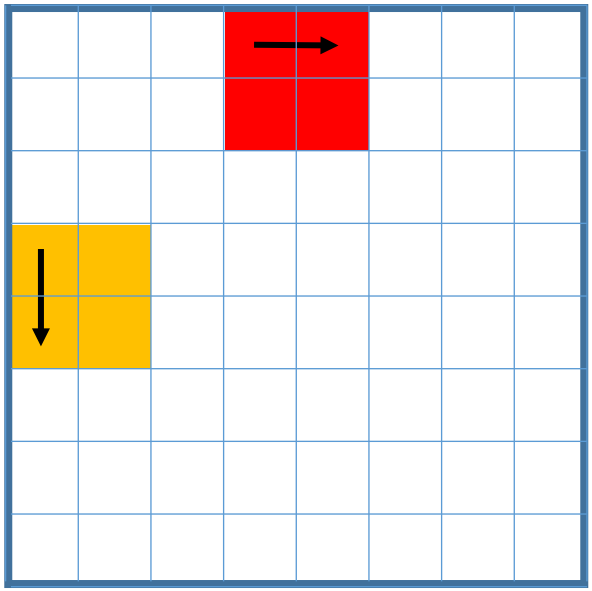
Memory access pattern



Coalesced reads

Strided writes

2D kernel



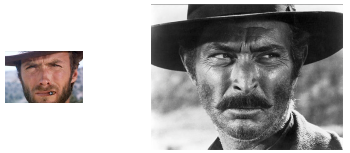
```
template<typename T>
__global__
void simpleTranspose2D(T* array_in, T* array_out, int n_rows, int n_cols) {
    const int col = threadIdx.x + blockDim.x * blockIdx.x;
    const int row = threadIdx.y + blockDim.y * blockIdx.y;

    if(col < n_cols && row < n_rows) {
        array_out[col * n_rows + row] = array_in[row * n_cols + col];
    }
}
```

Access pattern

```
dim3 block_dim(8, 32);
dim3 grid_dim(n / 8, n / 32);
```

- For a given warp:
  - col: 0 to 7
  - row: 0 to 3
- Reads are partially coalesced

- Writes are partially coalesced



Performance

```
Bandwidth bench
GPU took 0.305329 ms
Effective bandwidth is 109.896 GB/s

simpleTranspose
GPU took 22.8106 ms
Effective bandwidth is 16.181 GB/s

simpleTranspose2D
GPU took 6.98102 ms
Effective bandwidth is 52.8717 GB/s
```

```
     Avg        Min        Max  Name
299.22us   298.72us   300.54us  [CUDA memcpy DtoD]
2.0720ms   2.0589ms   2.0858ms  simpleTranspose(int
629.13us   627.36us   631.22us  simpleTranspose2D(i
```

Increase in performance is due to

lore threads are being used (2D vs 1D block)

2D indexing maps naturally to the shape of the matrix

Memory access is tially coalesced

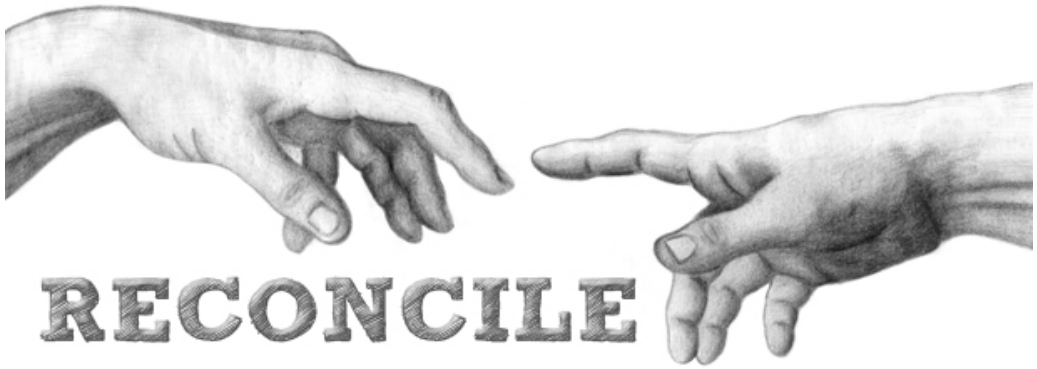
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

0

Total Results: 0

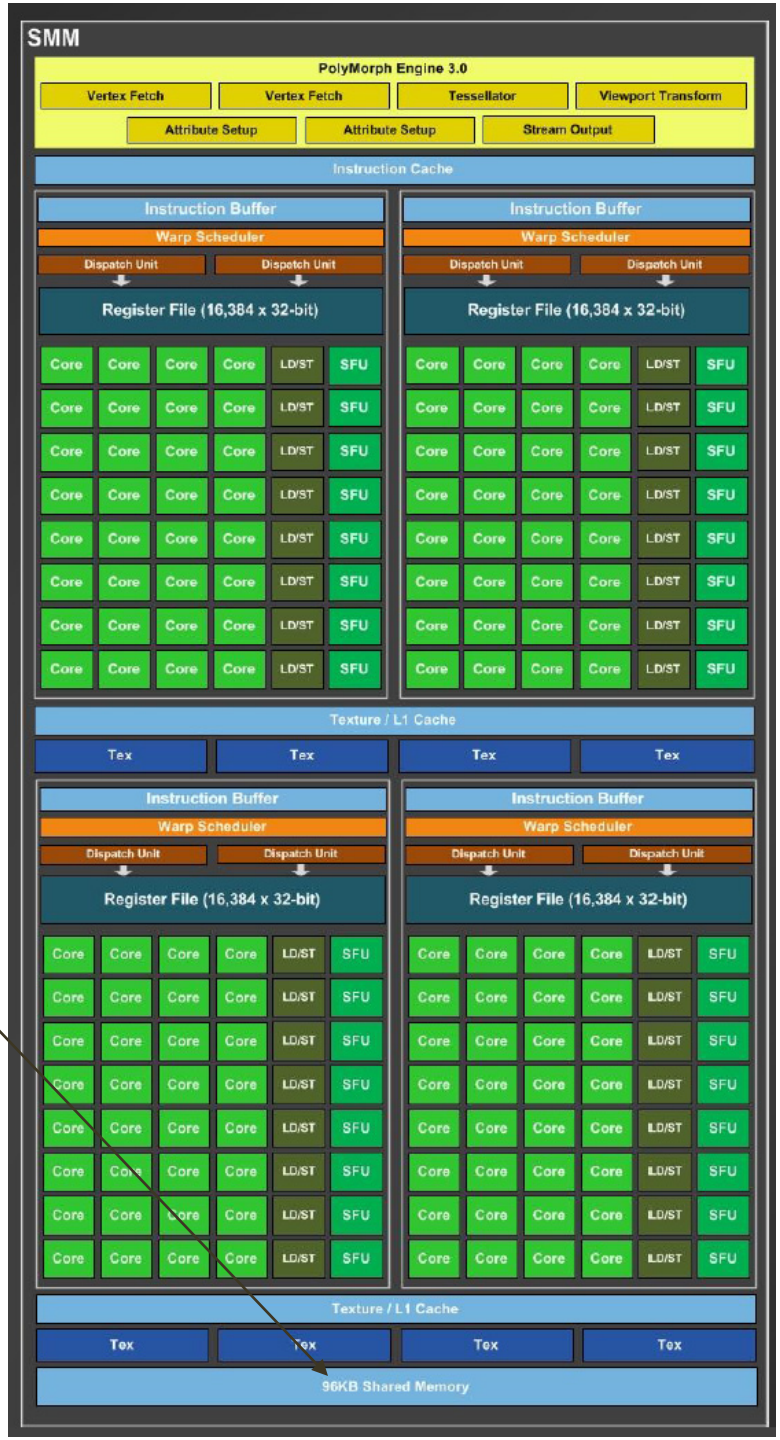
How can you reconcile the reads and writes?



Shared memory

- You need to read data in a coalesced way.
- Transpose locally using fast memory.
- Write data in a coalesced way.
- For this, we need to use shared memory!

It’s here!


SMM
PolyMorph Engine 3.0
Vertex Fetch
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Attribute Setup
Stream Output
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory

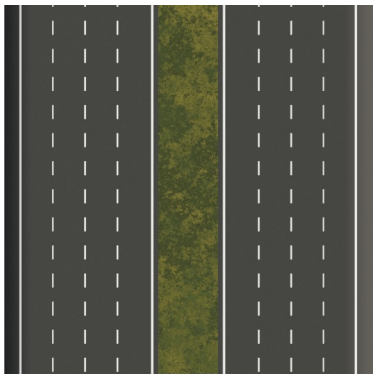
Shared memory

- On-chip: high bandwidth, low latency
- Data in shared memory is only accessible by threads in the same thread block!

```
template<int num_warps>
__global__
void fastTranspose(int* array_in, int* array_out, int n_rows, int n_cols) {
  const int warp_id  = threadIdx.y;
  const int lane     = threadIdx.x;

  __shared__ int block[warp_size][warp_size];
```

Shared memory variable

Imagine a highway with 32 lanes



Load data from global memory

```
const int bc = blockIdx.x;
const int br = blockIdx.y;

// Load 32x32 block into shared memory
int gc = bc * warp_size + lane; // Global column index

for(int i = 0; i < warp_size / num_warps; ++i) {
  int gr = br * warp_size + i * num_warps + warp_id; // Global row index
  block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
}

__syncthreads();
```

Shared memory variable

Global memory

All threads in block participate in this; so we need to wait that all threads are done before continuing.

Write is the same but transpose

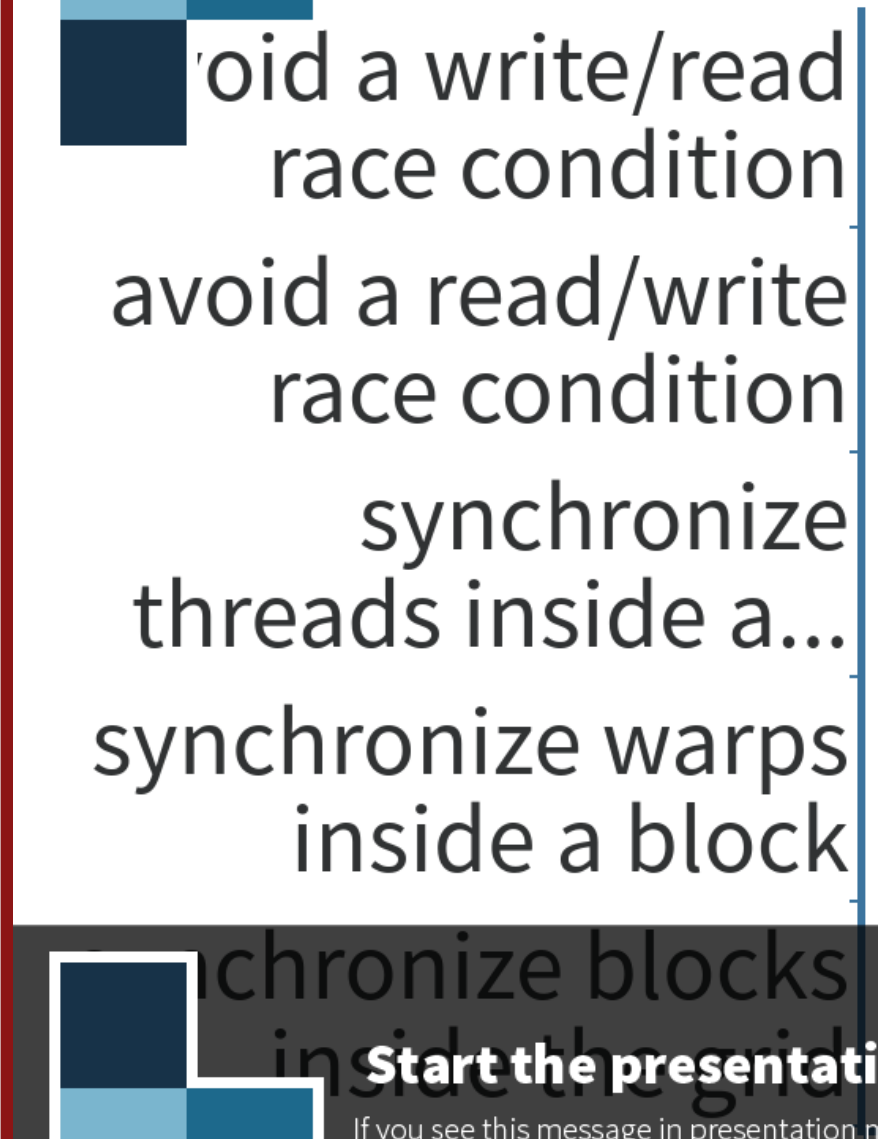
```
int gr = br * warp_size + lane;

for(int i = 0; i < warp_size / num_warps; ++i) {
  int gc = bc * warp_size + i * num_warps + warp_id;
  array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
}
```

You know this works now!

```
lane = threadIdx.x
gr = ... + lane
array_out[gc * n_rows + gr]
```

Stride is 1 with respect to `threadIdx.x`

syncthreads is required to



Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

Performance

```
Bandwidth bench
GPU took 0.306039 ms
Effective bandwidth is 109.641 GB/s

simpleTranspose
GPU took 22.7943 ms
Effective bandwidth is 16.1926 GB/s

simpleTranspose2D
GPU took 6.99616 ms
Effective bandwidth is 52.7573 GB/s

fastTranspose
GPU took 7.3289 ms
Effective bandwidth is 50.3621 GB/s
```

Yeah!

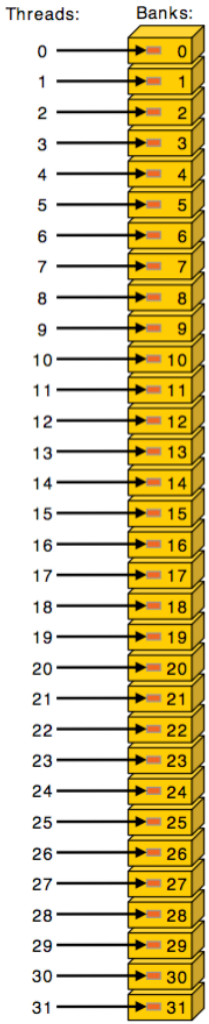
Wait…

- We only went from 53 GB/s down to 50 GB/s
- We are still far from the peak of 110 GB/s
- What happened?

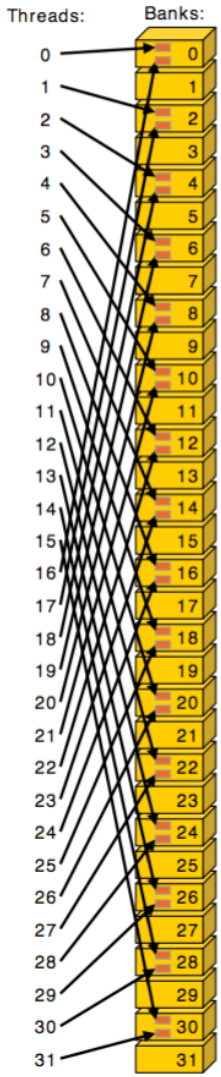
Banks

- Shared memory suffers from bank conflicts.
- The shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.
- Any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.
- However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized.
- Each bank has a bandwidth of 4 bytes per two clock cycles.

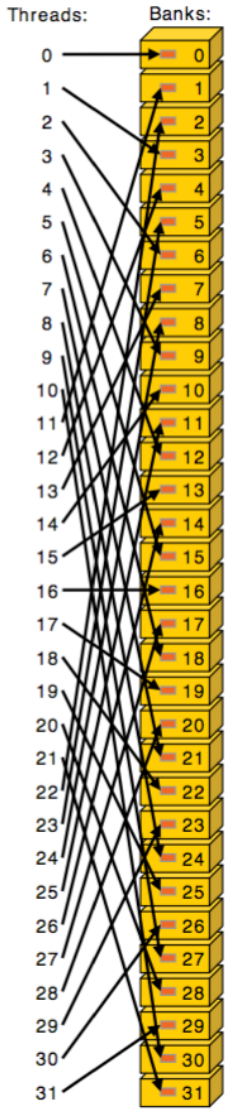
Conflict free



Two way conflict

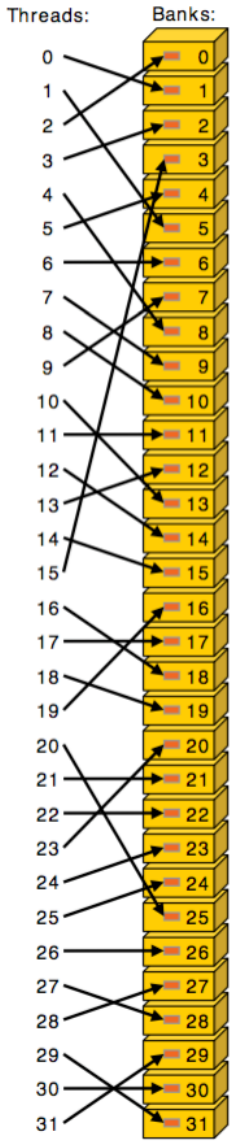


Conflict free



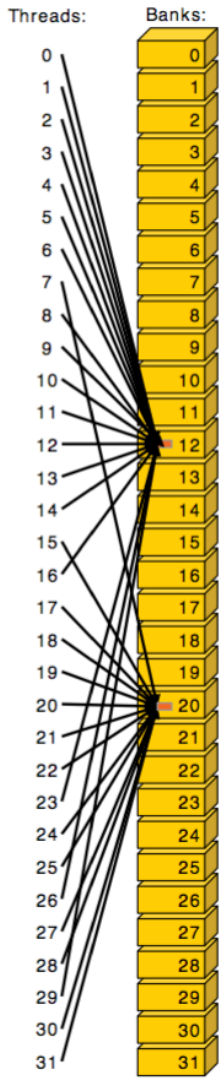
Stride of 3

Conflict free



General permutation

Conflict free



Broadcast capability
Accessing the same word inside the bank

Shared memory writes

```
block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
```

Stride of 1

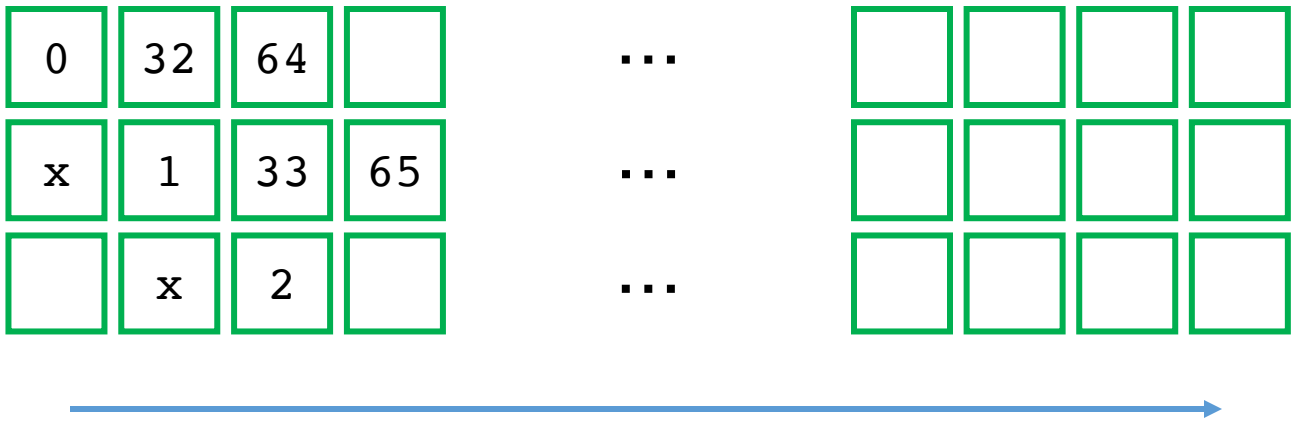
Shared memory reads

```
__shared__ int block[warp_size][warp_size];
```

```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```

Stride of 32!

```
__shared__ int block[warp_size][warp_size+1];
```



32 banks

Contiguous memory space

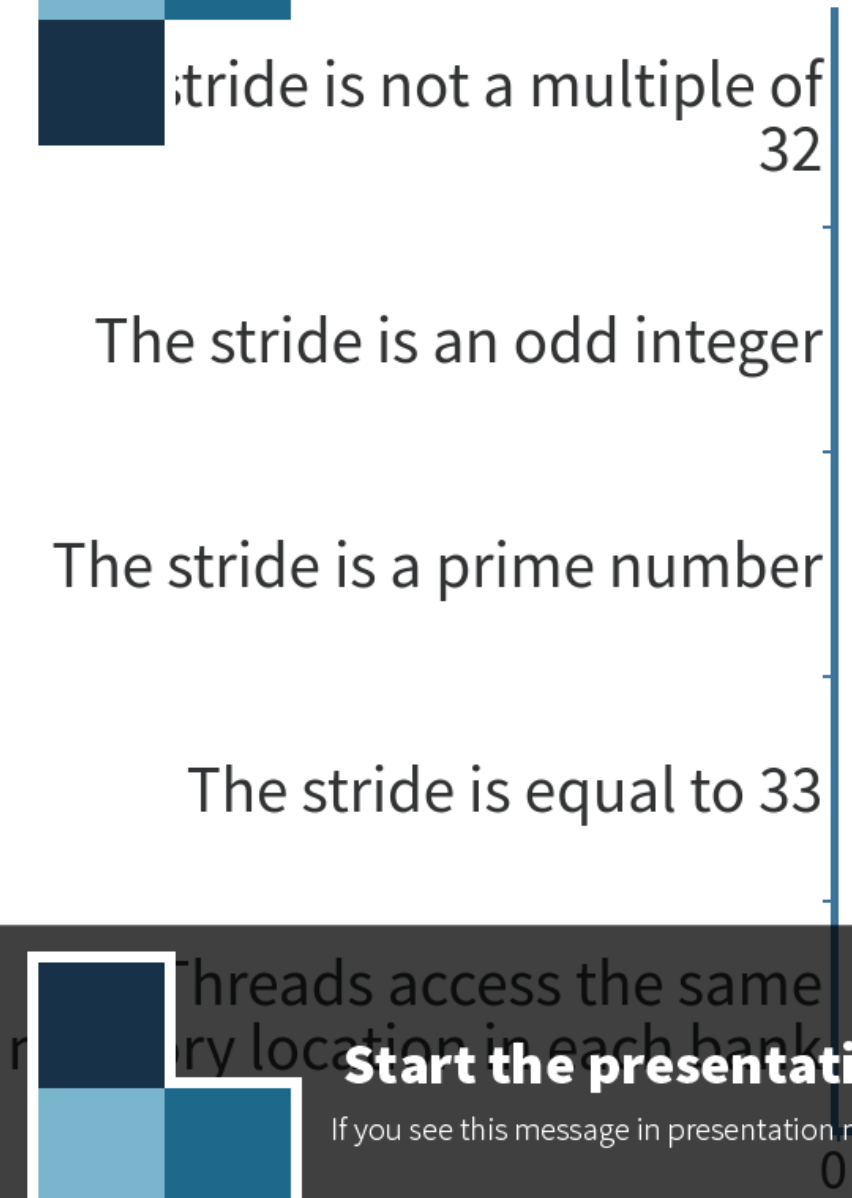
```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```

Loop with stride `num_warps`

```
Bandwidth bench
GPU took 0.305327 ms
Effective bandwidth is 109.897 GB/s

simpleTranspose
GPU took 22.7924 ms
Effective bandwidth is 16.194 GB/s

simpleTranspose2D
GPU took 6.98426 ms
Effective bandwidth is 52.8473 GB/s

fastTranspose
GPU took 3.74134 ms
Effective bandwidth is 98.6541 GB/s
```

```
      Avg        Min        Max   Name
 299.24us 298.68us   300.28us   [CUDA memcpy DtoD]
 2.0703ms   2.0614ms   2.0762ms   simpleTranspose(int
 629.47us 621.31us   633.11us   simpleTranspose2D(i
 335.04us 333.96us   336.33us   void fastTranspose<
```

You avoid bank conflicts when



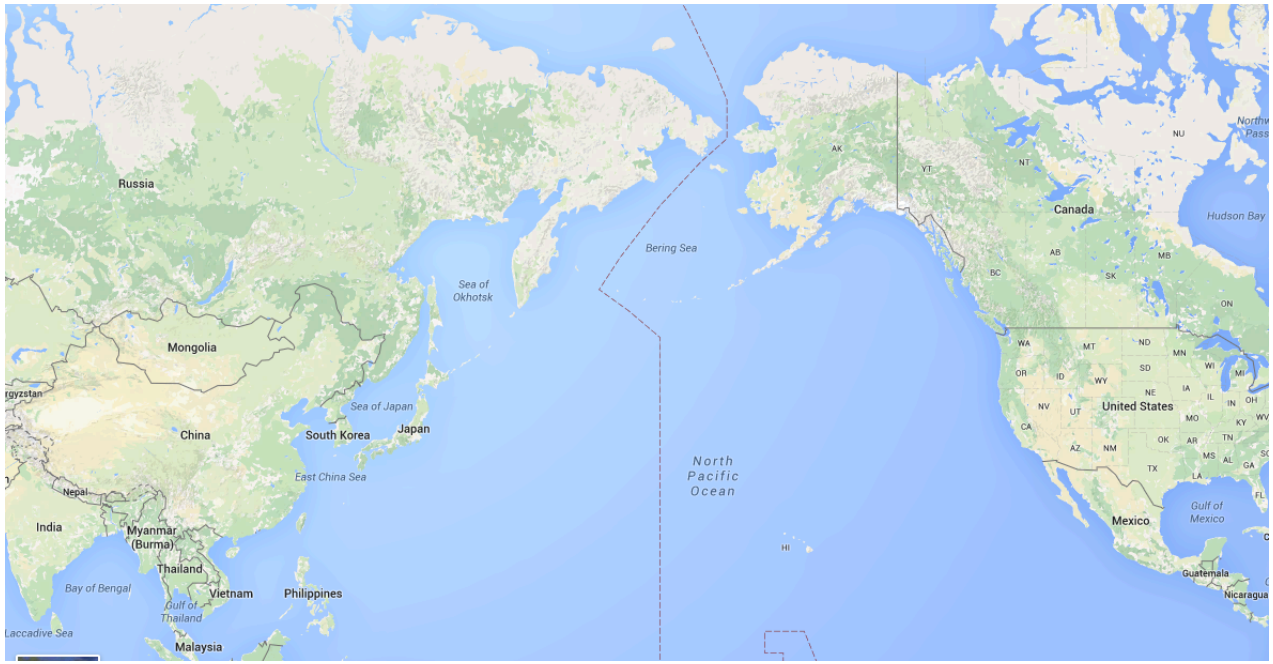
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

Concurrency, latency

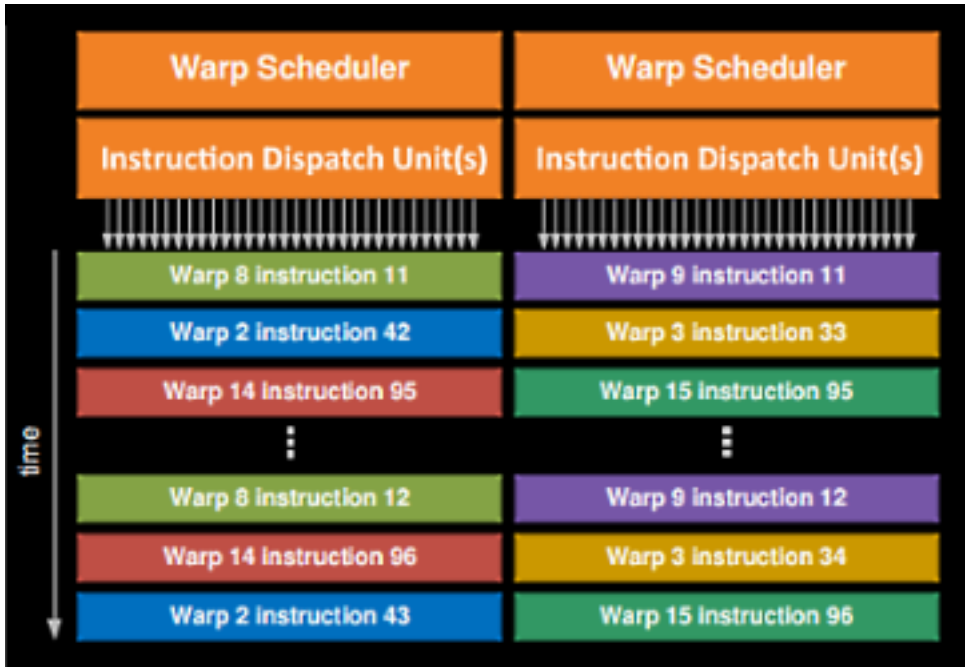
- Imagine you are a pencil manufacturer.
- You outsource your manufacturing plants to China but your market is in the US.
- How do you organize the logistics of the transport?



- You may have great and cheap bandwidth.
- But the latency is very long.
- This works as long as you have enough parallel work to do, i.e., lots of boxes of pencils to ship.
- Then they can be put on slow but big ships.
- Once the shipping pipeline is full, you get cheap and frequent deliveries in large quantities.
- This is the same on a GPU.

Concurrency!

- Concurrency is used to hide memory latency.
- Concurrency is also important for other pipelines in the system, for example in the floating point units.
- The system keeps switching between warps, to avoid being idle.



How is concurrency managed?

- The key is therefore to have as many threads live on an SM as possible. (That's not 100% true all the time but that's a good rule of thumb.)
- This is constrained by 2 requirements:
  - We can only fit a whole number of blocks on an SM.
  - There are hardware limits.
- Main ones:
  - Max dim of grid along x, y, and z: 65,535
  - Max x-y dim of block: 1,024
  - Max no. of threads per block: 1,024
  - Max blocks per SM: 8
  - Max resident warps: 48
  - Max threads per SM: 1,536
  - No. of 4-byte registers per SM: 32 K
  - Max shared mem per SM: 48 KB

How can we make sense of this?

- Use the spreadsheet: CUDA occupancy calculator.
- Use CUDA API:

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

Provides an occupancy prediction based on the block size and shared memory usage of a kernel

`cudaOccupancyMaxPotentialBlockSize`

`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

Returns grid and block size that achieves maximum potential occupancy for a device function.

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	14	
Shared Memory Per Block (bytes)	4224	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	2.0
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	8
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	32768
Max Registers per Thread Block	32768
Max Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	64
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	2

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	72	9
Shared Memory (Bytes)	4224	49152	11

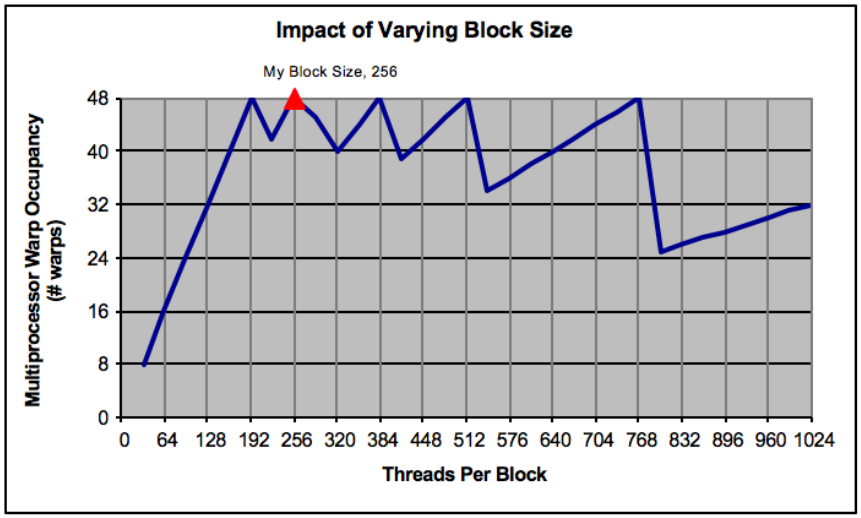
Note: SM is an abbreviation for (Streaming) Multiprocessor

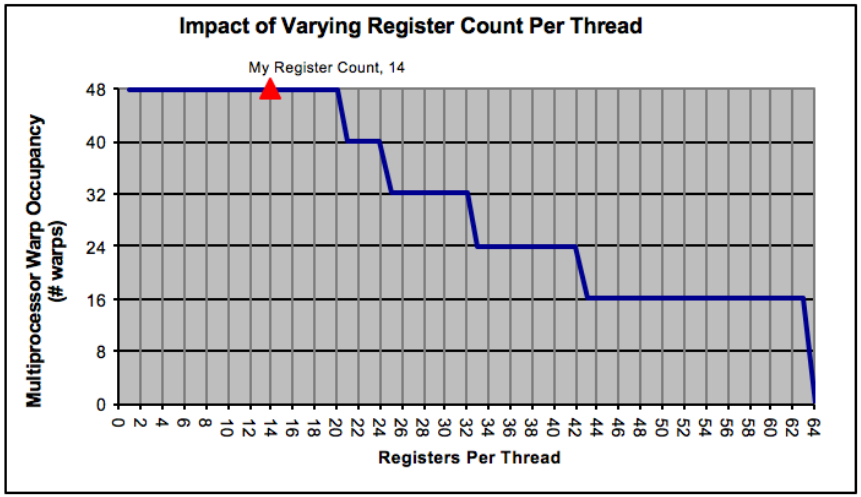
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48
Limited by Registers per Multiprocessor	9		
Limited by Shared Memory per Multiprocessor	11		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48
Occupancy = 48 / 48 = 100%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





How to get kernel information

```
[darve@certainty-a code]$ nvcc --ptxas-options=-v -O3 -arch=sm_20 transpose.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z17simpleTranspose2DPiS_ii' for 'sm_20'
ptxas info    : Function properties for _Z17simpleTranspose2DPiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 56 bytes cmem[0]
ptxas info    : Compiling entry function '_Z15simpleTransposePiS_ii' for 'sm_20'
ptxas info    : Function properties for _Z15simpleTransposePiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 56 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13fastTransposeILi8EEvPiS0_ii' for 'sm_20'
ptxas info    : Function properties for _Z13fastTransposeILi8EEvPiS0_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 14 registers, 4224 bytes smem, 56 bytes cmem[0]
```

```
$ nvcc --ptxas-options=-v
```

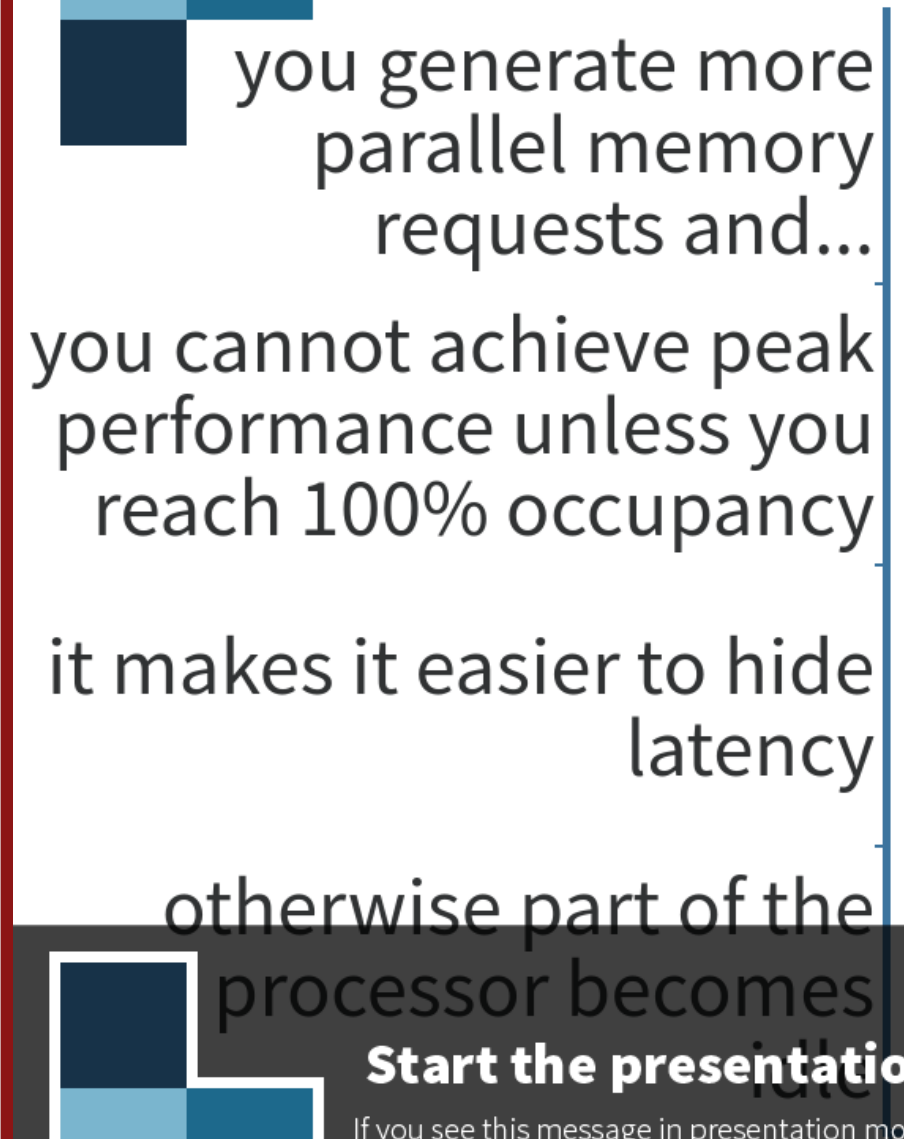
How to choose the number of warps per block?

- With our implementation, the number of warps must divide the L1 cache line size of 32.
- Generally, you pick the smallest number of warps that gives you the highest occupancy.
- Here that number is 8 warps = 256 threads / block.

```
for(int i = 0; i < warp_size / num_warps; ++i) {
  int gc = bc * warp_size + i * num_warps + warp_id;
  array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
}
```

4 iterations required to complete read

It's best to maximize occupancy because



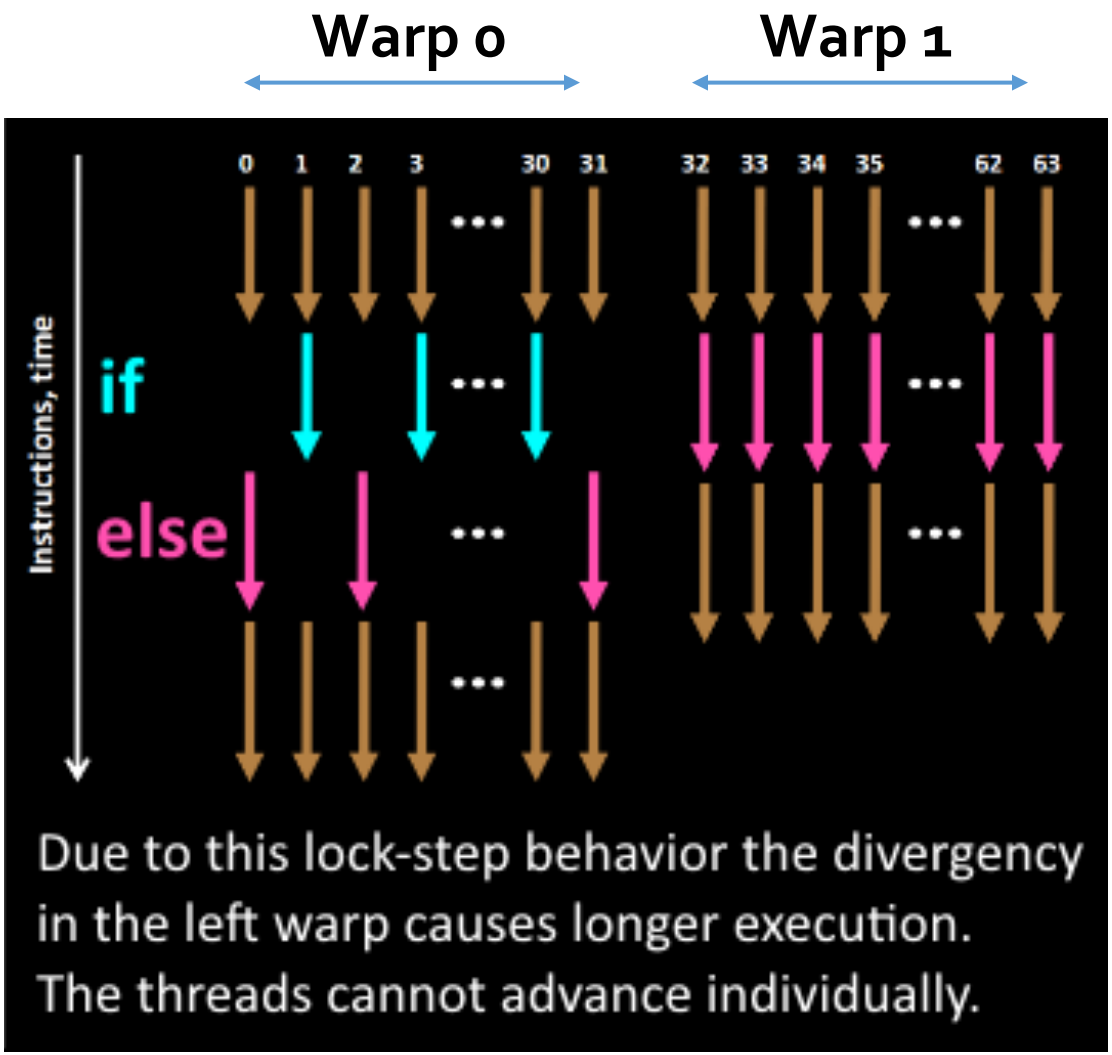
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

Branch divergence

- Should be avoided at all cost.
- This leads to idle threads.


Warp 0
Warp 1
0 1 2 3 ... 30 31
32 33 34 35 ... 62 63
Instructions, time
if
else
Due to this lock-step behavior the divergency
in the left warp causes longer execution.
The threads cannot advance individually.

Branch occurrences

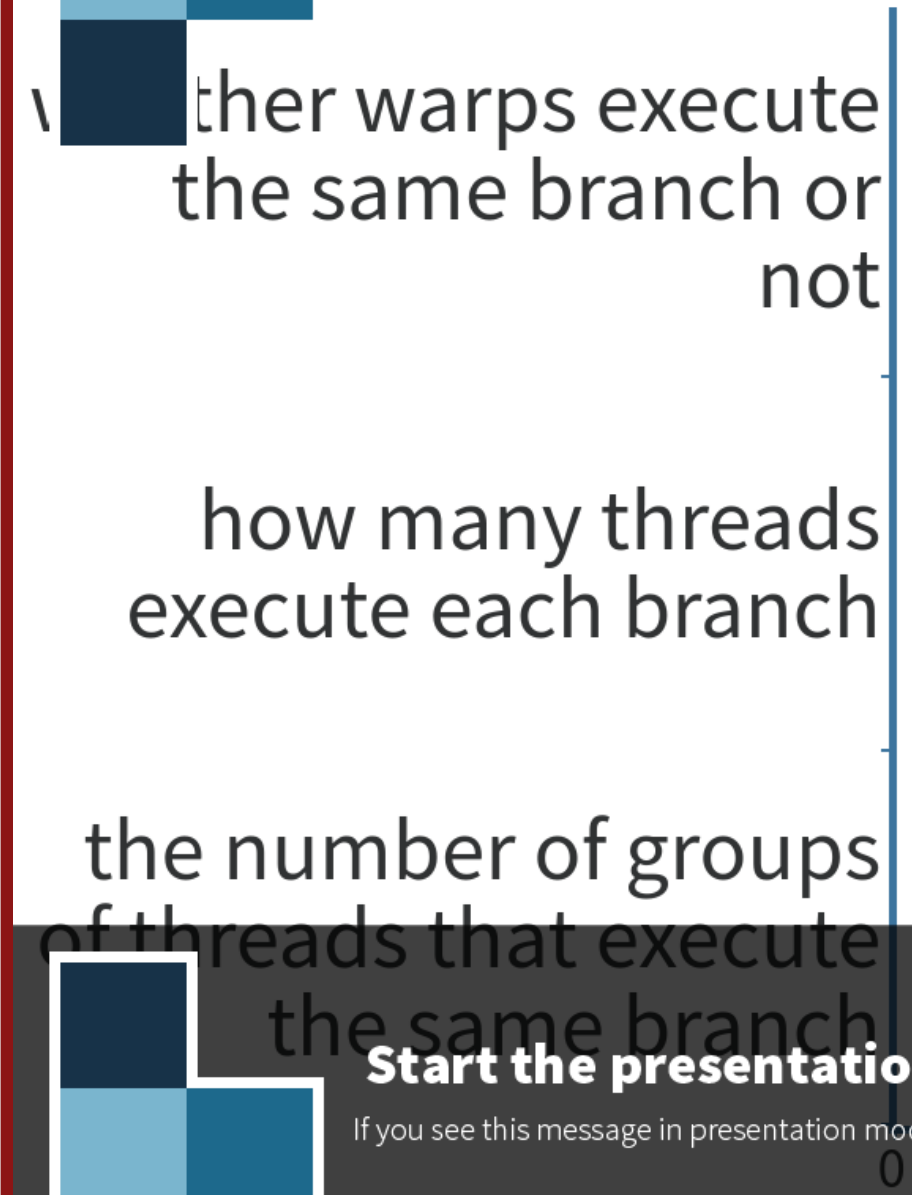
- `If/while` statements
- `For loop` of varying lengths, e.g., processing an unstructured grid
- Branch divergence is an issue only for threads belonging to the same warp.
- If warps do different things, there is no performance impact.

```
__global__ void branch_thread(float* out) {
    int tid = threadIdx.x;

    if(tid%2 == 0) {
        ...;
    } else {
        ...;
    }
}

__global__ void branch_warp(float* out) {
    int wid = threadIdx.x/32;

    if(wid%2 == 0) {
        ...;
    } else {
        ...;
    }
}
```

Divergence!

No divergence!

erformance when executing a branch depen



Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0