# CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

Stanford University

“You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time.” (Bertrand Meyer)

# GPU Optimization

Optimize data transfer from GPU memory

- Caches are used to optimize memory accesses: L1 and L2 caches.
- Cache behavior is complicated and depends on the compute capability of the GPU.
- We will focus on Turing `sm_75`

## L1 cache

- Used for local memory (memory local to each thread) and register spills (not enough space for all the registers).
- Data that is read-only for the entire lifetime of the kernel (as determined by the compiler) can be cached in L1.
- Local to an SM.

# L2 cache

- Cache accesses to local and global memory
- Shared by all SMs on the GPU
- Memory accesses that are cached in L2 only are serviced with 32-byte memory transactions
- That's 8 `float` or 1 byte per thread in a warp
- If each thread reads a float, that's 4 x 32-bytes.

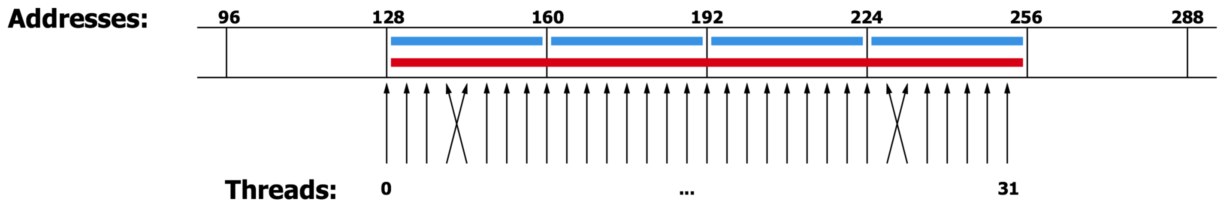
- Each memory request from a warp is broken down into cache line requests that are issued independently.
- A cache line request is serviced at the throughput of the L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Let's make this concrete with a code

```
int xid = blockIdx.x * blockDim.x + threadIdx.x;
if (xid < n)
    odata[xid] = idata[xid];
```

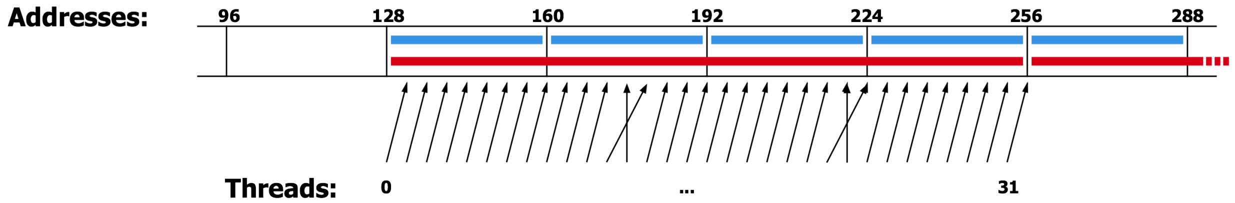
- Warp requests several memory addresses.
- These are translated into cache line requests (with a granularity of 32 bytes).
- Memory requests are serviced.
- Coalesced access: for every 32-byte cache line, all 32 bytes are requested and used by the warp.

# Aligned accesses (sequential/non-sequential)



| Compute capability: | 2.0 and later | |
|---|---|---|
| Memory transactions: | Uncached | Cached |
| | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224 | 1x 128B at 128 |

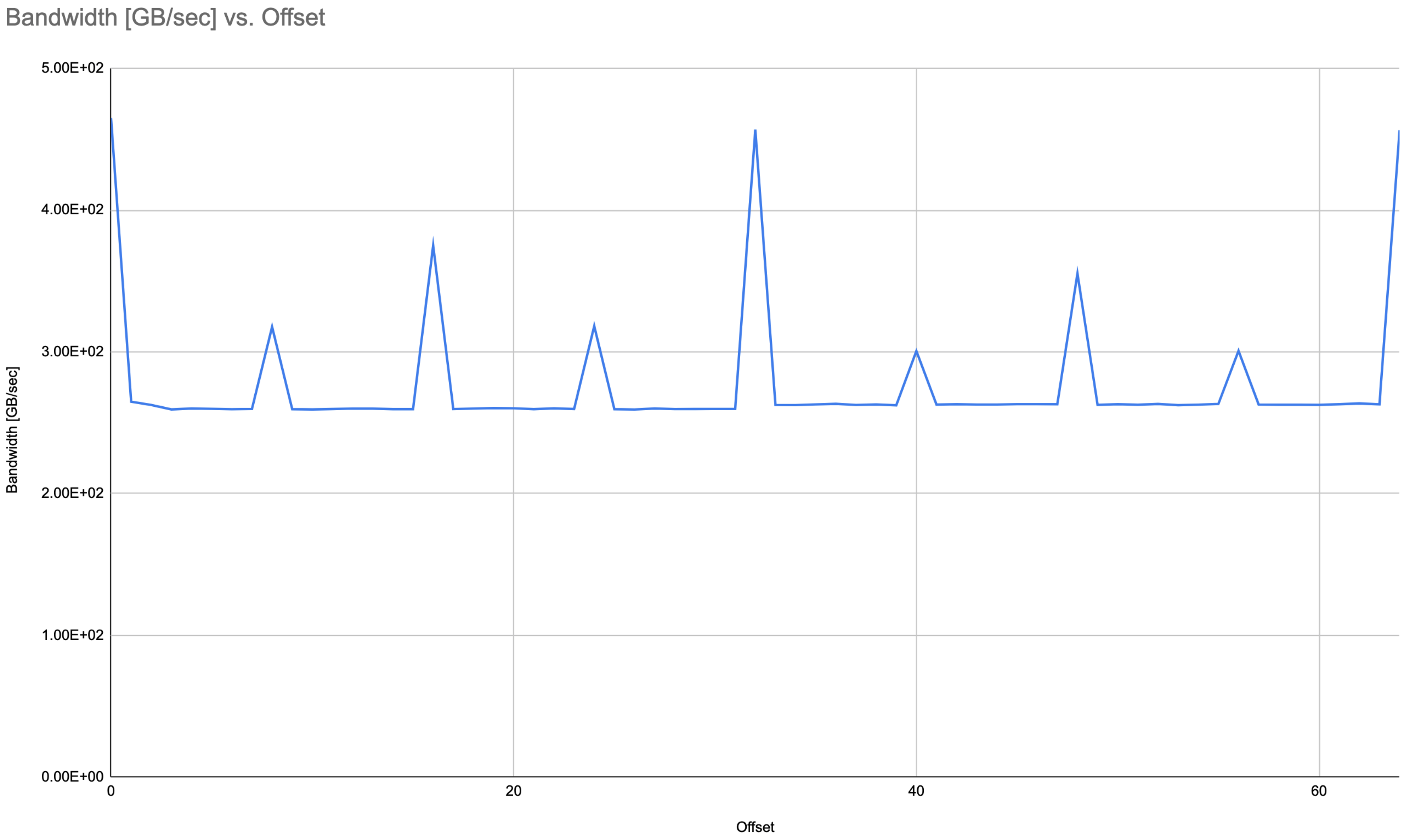
# Mis-aligned accesses (sequential/non-sequential)



| Compute capability: | 2.0 and later | |
|---|---|---|
| Memory transactions: | Uncached | Cached |
| | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224<br>1x 32B at 256 | 1x 128B at 128<br>1x 128B at 256 |

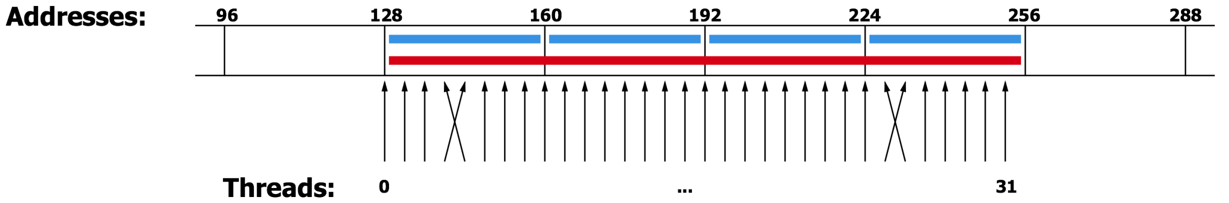
```
int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
if (xid < n)
    odata[xid] = idata[xid];
```

# Turing `icme-gpu` Quadro RTX 6000



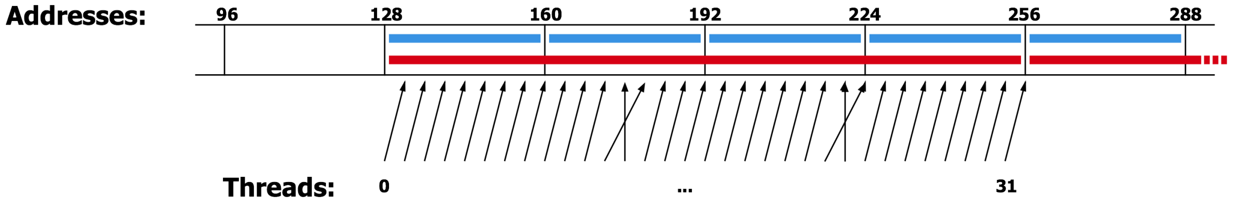
```
int xid = stride * (blockIdx.x * blockDim.x + threadIdx.x);
if (xid < n)
    odata[xid] = idata[xid];
```

## Aligned accesses (sequential/non-sequential)

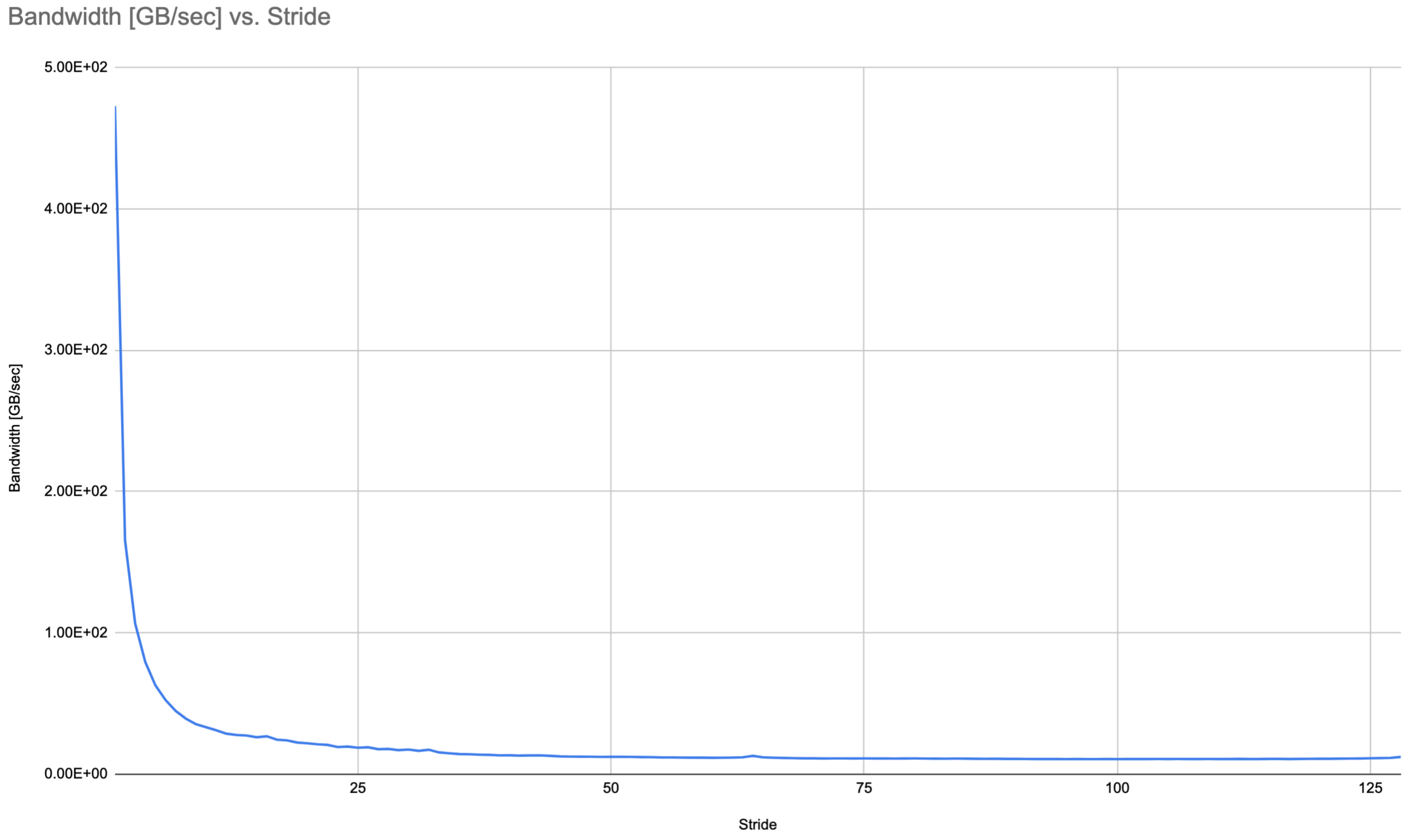


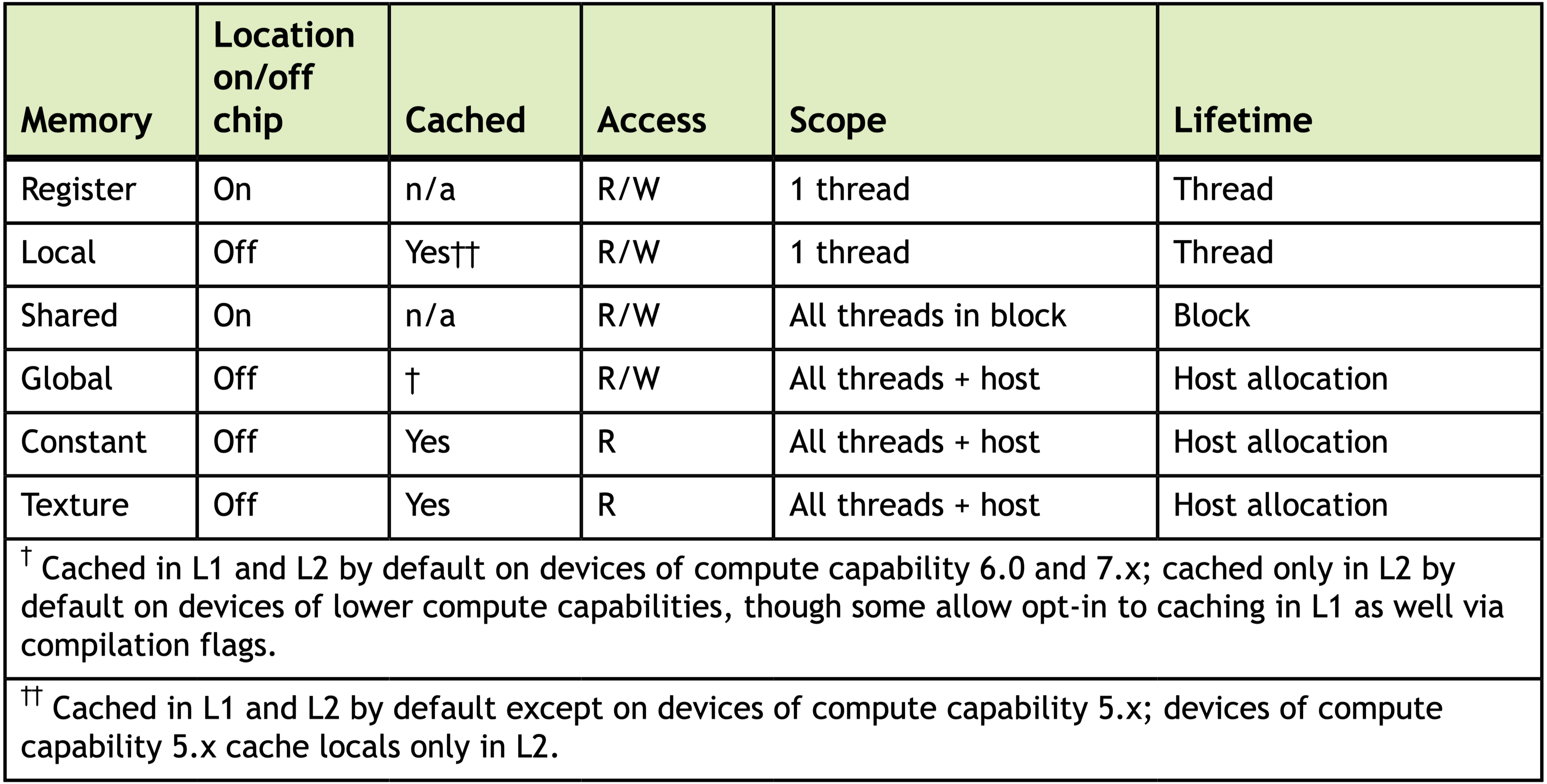
| Compute capability: | 2.0 and later | |
|---|---|---|
| Memory transactions: | Uncached | Cached |
| | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224 | 1x 128B at 128 |

## Mis-aligned accesses (sequential/non-sequential)



| Compute capability: | 2.0 and later | |
|---|---|---|
| Memory transactions: | Uncached | Cached |
| | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224<br>1x 32B at 256 | 1x 128B at 128<br>1x 128B at 256 |


Bandwidth [GB/sec] vs. Stride
5.00E+02
4.00E+02
3.00E+02
2.00E+02
1.00E+02
0.00E+00
Bandwidth [GB/sec]
25
50
75
100
125
Stride



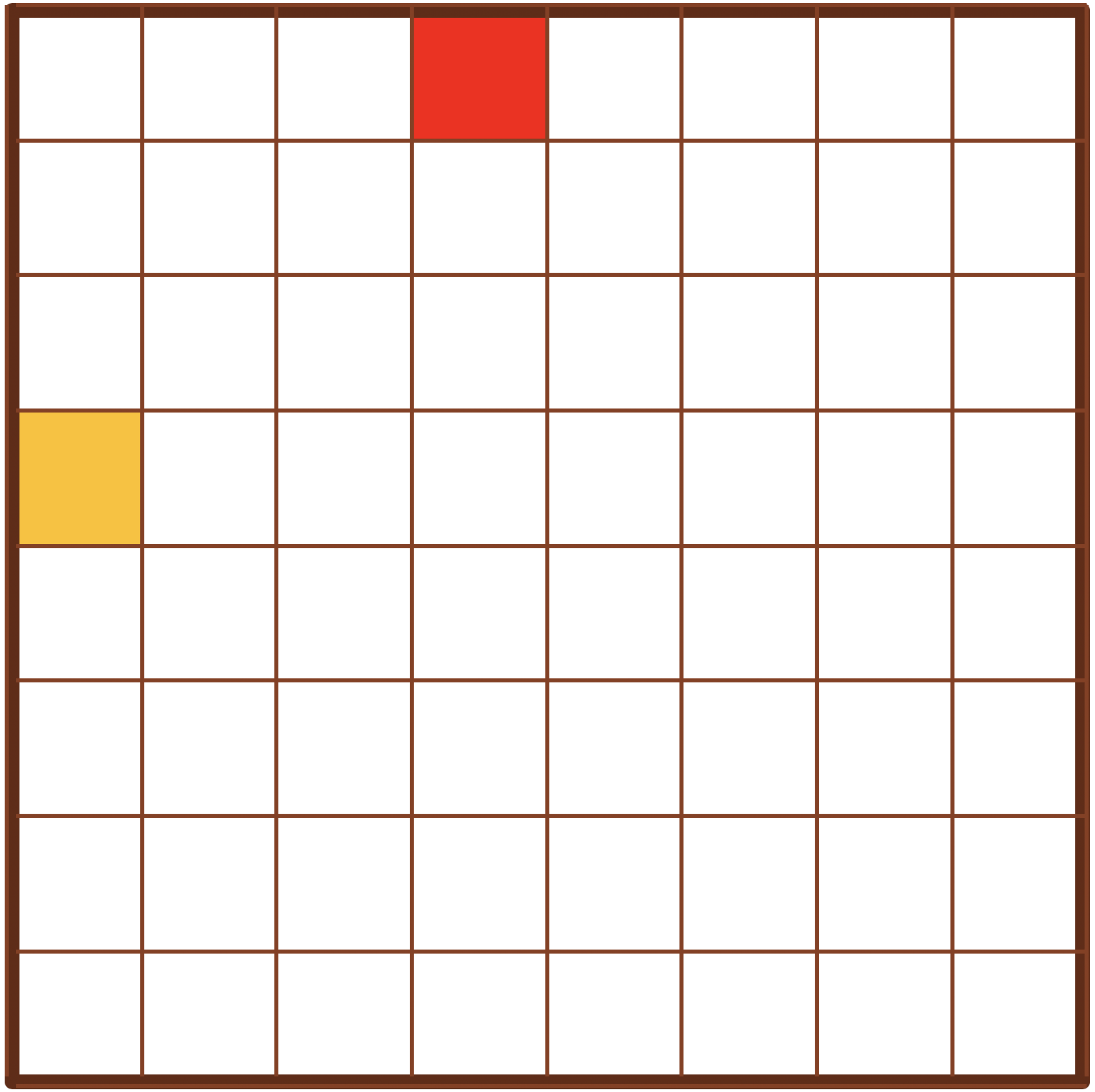
| Memory | Location on/off chip | Cached | Access | Scope | Lifetime |
|---|---|---|---|---|---|
| Register | On | n/a | R/W | 1 thread | Thread |
| Local | Off | Yes†† | R/W | 1 thread | Thread |
| Shared | On | n/a | R/W | All threads in block | Block |
| Global | Off | † | R/W | All threads + host | Host allocation |
| Constant | Off | Yes | R | All threads + host | Host allocation |
| Texture | Off | Yes | R | All threads + host | Host allocation |

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

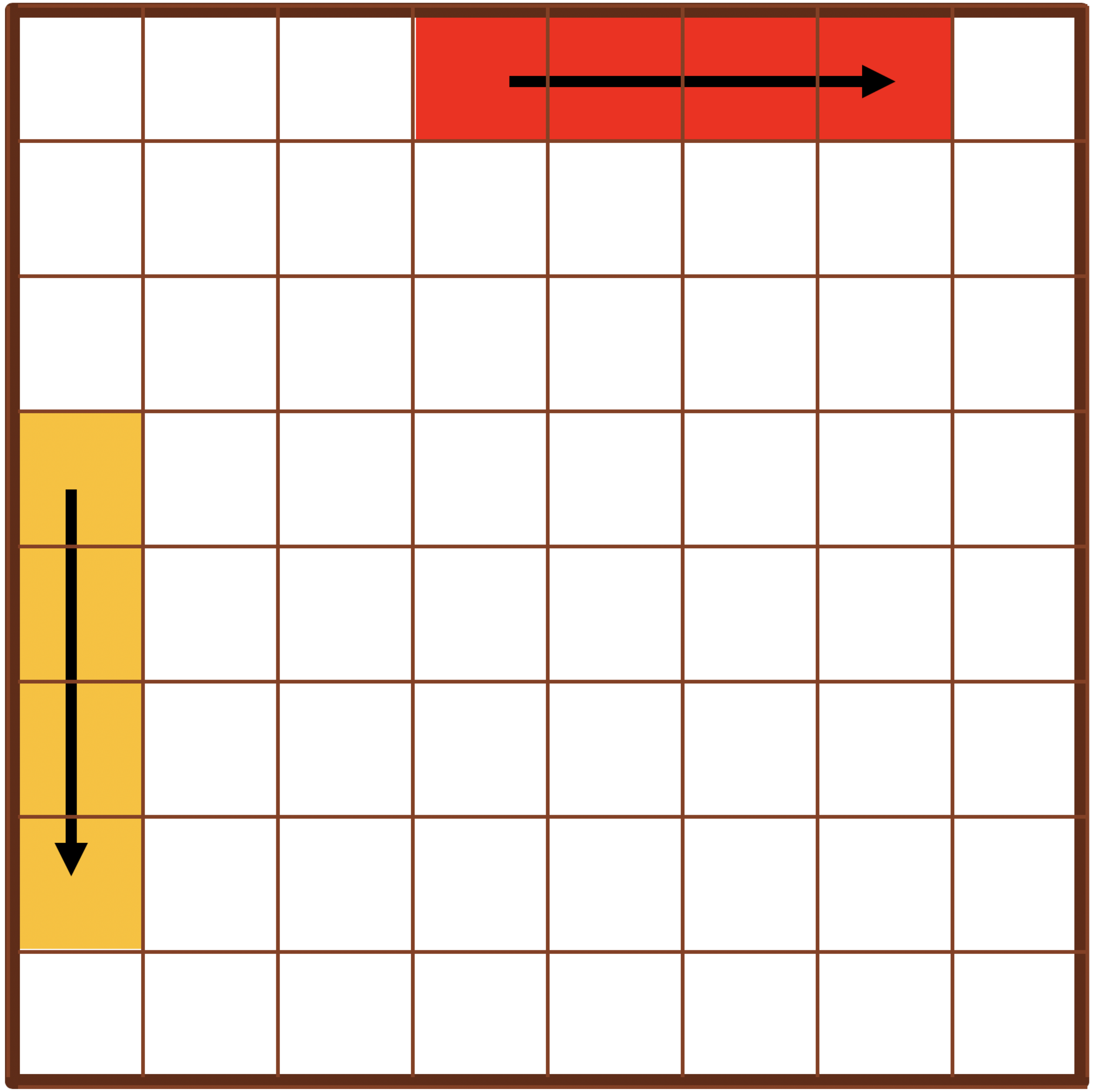
Let’s put all these concepts into play through a specific example: a matrix transpose.

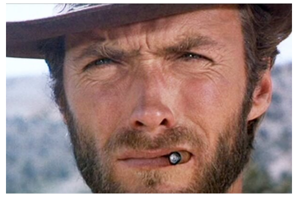
It’s all about bandwidth!

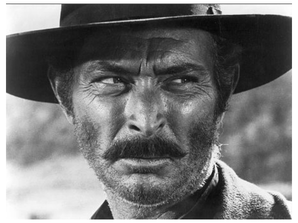


Even for such a simple calculation, there are many optimizations.

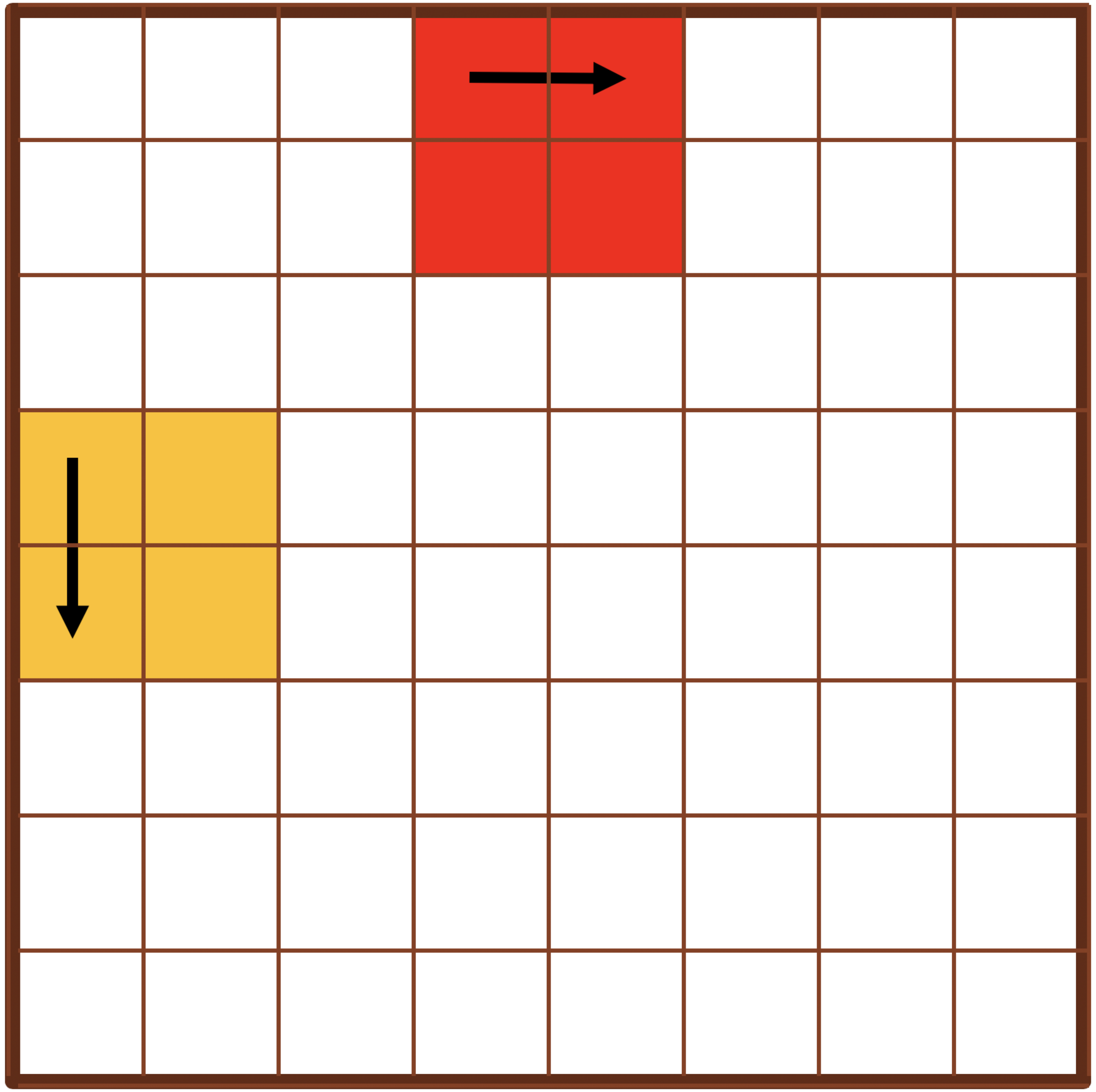
```
const int tid = threadIdx.x + blockDim.x * blockIdx.x;
int col = tid % n_cols;
int row = tid / n_cols;
if(col < n_cols && row < n_rows) {
    array_out[col * n_rows + row] = array_in[row * n_cols + col];
}
```



Read 

Write 

# 2D kernel

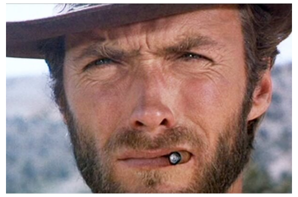


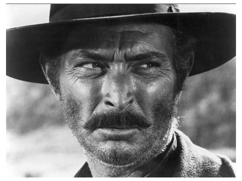
```
const int col = threadIdx.x + blockDim.x * blockIdx.x;
const int row = threadIdx.y + blockDim.y * blockIdx.y;

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

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

For a given warp:

- column: 0 to 7
- row: 0 to 3

Read 

Write  

# Benchmark

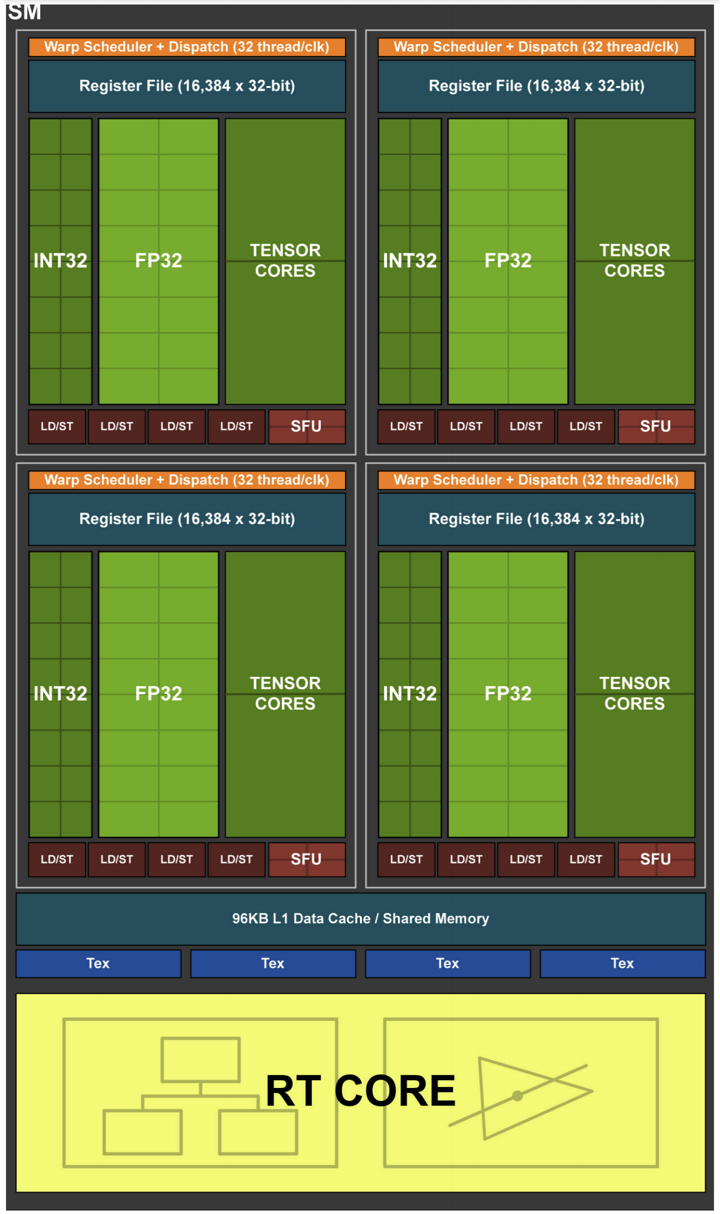
```
darve@icme-gpu:~/$ make; srun --partition=CME --gres=gpu:1 transpose
nvcc -O3 --gpu-architecture=compute_75 --gpu-code=sm_75 -o transpose transpose.cu
Number of MB to transpose: 4096

Bandwidth bench
GPU took 17.7267 ms
Effective bandwidth is 484.577 GB/s

simpleTranspose
GPU took 286.168 ms
Effective bandwidth is 30.0171 GB/s (almost 16x drop)

simpleTranspose2D
GPU took 30.8758 ms
Effective bandwidth is 278.209 GB/s
```

Can we reconcile read and write?

Load in fast **shared memory**

Transpose from **shared memory** is very fast!

# Shared memory



# Facts

On-chip: high bandwidth, low latency

Data in shared memory is only accessible by threads in the same thread block!

```
const int warp_id  = threadIdx.y;
const int lane     = threadIdx.x;

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

`lane`: id of thread inside warp

`block`: variable allocated in shared memory

## Load data

```
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();
```

## Store

```
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];
}
```

# Performance

```
Bandwidth bench
GPU took 17.7267 ms
Effective bandwidth is 484.577 GB/s

simpleTranspose
GPU took 286.168 ms
Effective bandwidth is 30.0171 GB/s

simpleTranspose2D
GPU took 30.8758 ms
Effective bandwidth is 278.209 GB/s

fastTranspose
GPU took 29.64 ms
Effective bandwidth is 289.809 GB/s
```

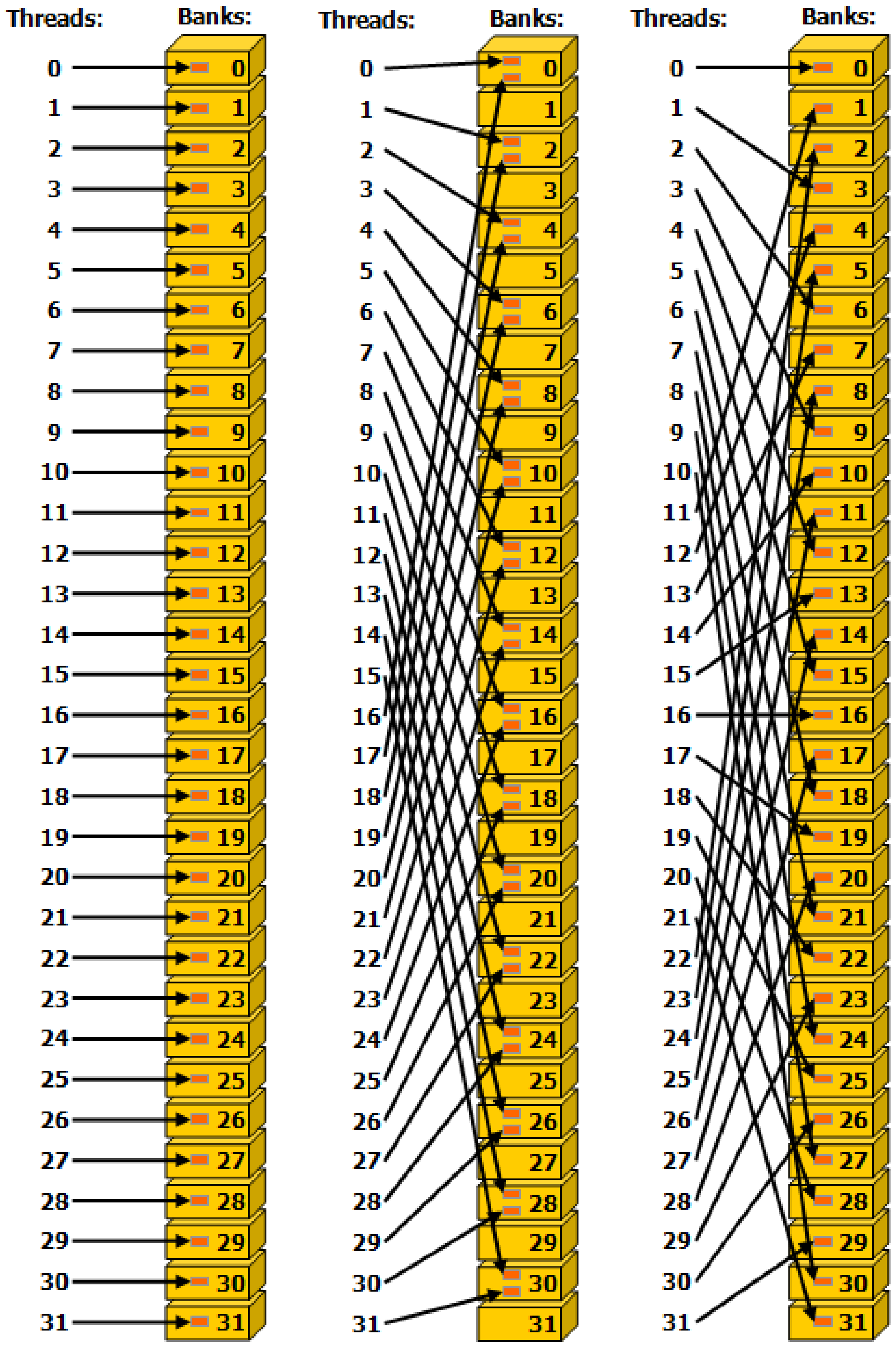
Shared memory suffers from bank conflicts.

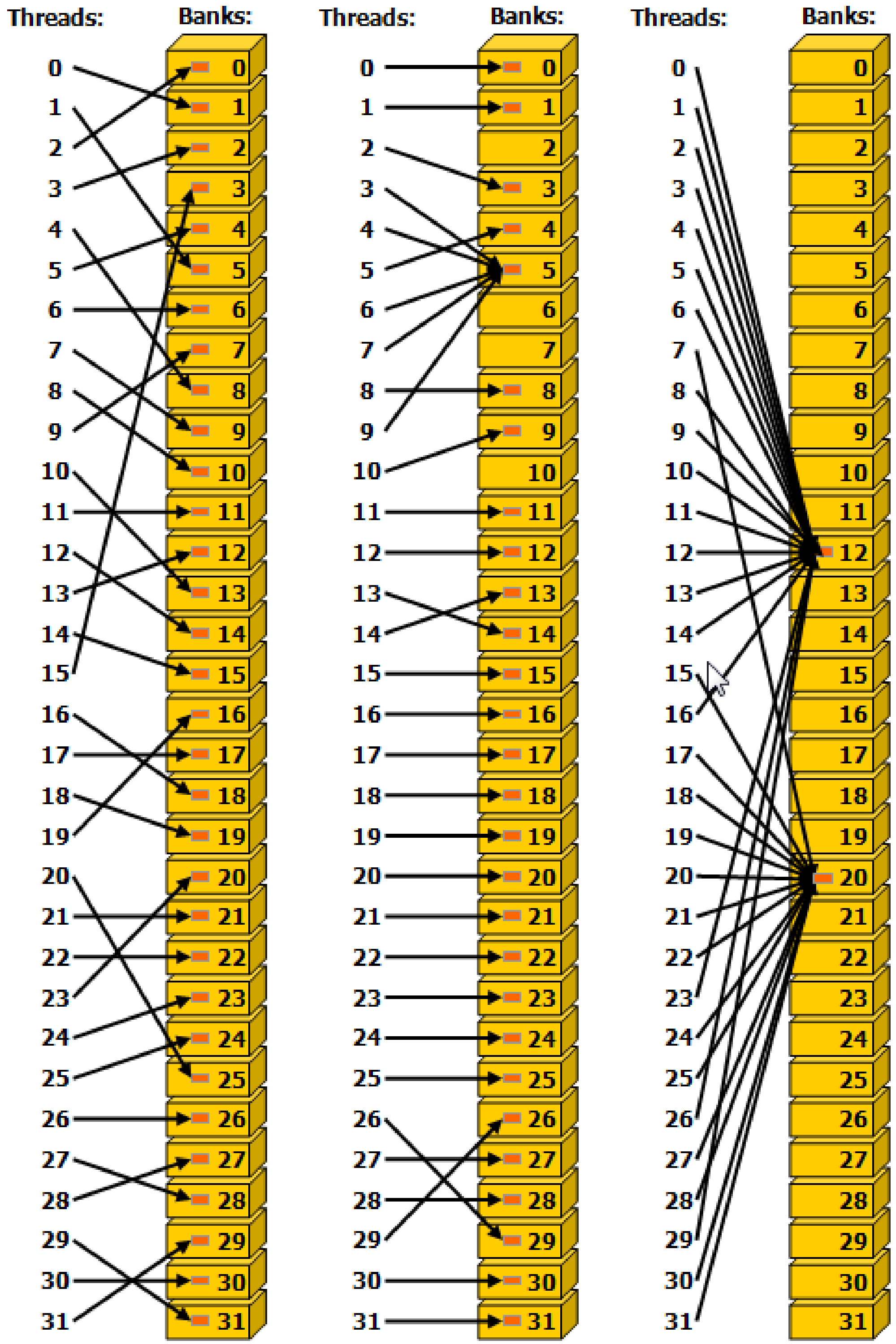
The shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.

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.

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.


Threads:
Banks:
Threads:
Banks:
Threads:
Banks:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31


Threads:
Banks:
Threads:
Banks:
Threads:
Banks:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

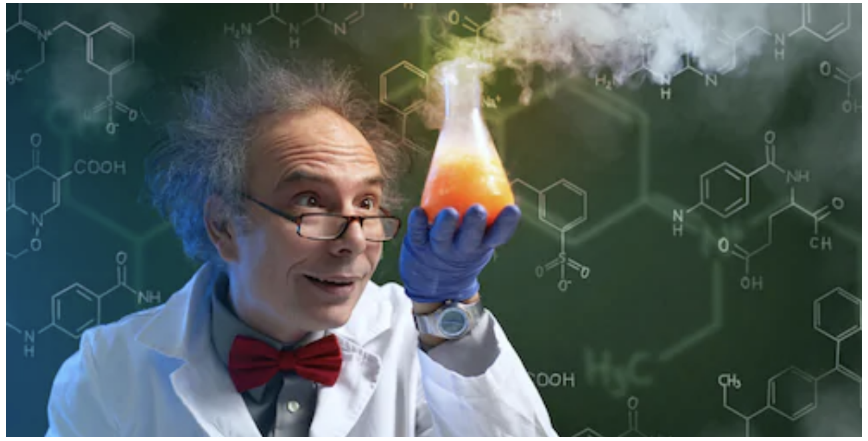
```
block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
```

Stride of 1

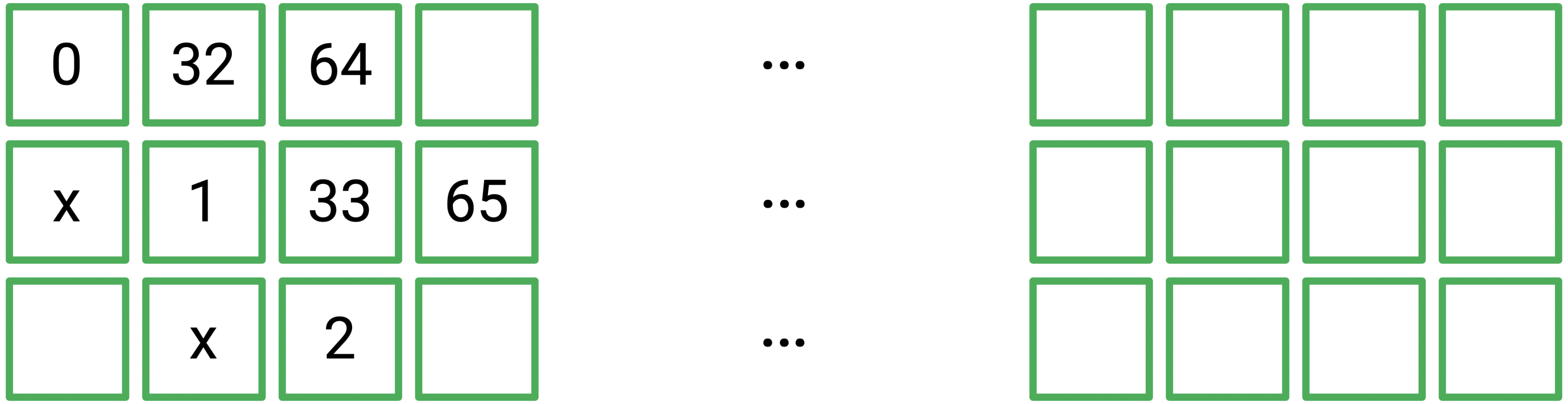
```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```

Stride of 32

The cure!



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



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 32 | 64 | | … | | | | |
| x | 1 | 33 | 65 | … | | | | |
| | x | 2 | | … | | | | |

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

# fastTranspose

```
Bandwidth bench
GPU took 17.7381 ms
Effective bandwidth is 484.263 GB/s

simpleTranspose
GPU took 286.166 ms
Effective bandwidth is 30.0173 GB/s

simpleTranspose2D
GPU took 29.9896 ms
Effective bandwidth is 286.431 GB/s

fastTranspose
GPU took 24.389 ms
Effective bandwidth is 352.205 GB/s
```