# Heterogeneous Computing using openCL lecture 4

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz

## The Big Picture

- Introduction to Heterogeneous Systems
- OpenCL Basics
- Memory Issues
- Scheduling

# Memory Banks

- Memory is made up of *banks*
  - Memory banks are the hardware units that actually store data
- The memory banks targeted by a memory access depend on the address of the data to be read/written
  - Note that on current GPUs, there are more memory banks than can be addressed at once by the global memory bus, so it is possible for different accesses to target different banks
    - Bank response time, not access requests, is the bottleneck
- Successive data are stored in successive banks (strides of 32-bit words on GPUs) so that a group of threads accessing successive elements will produce no bank conflicts

## Bank Conflicts – Local Memory

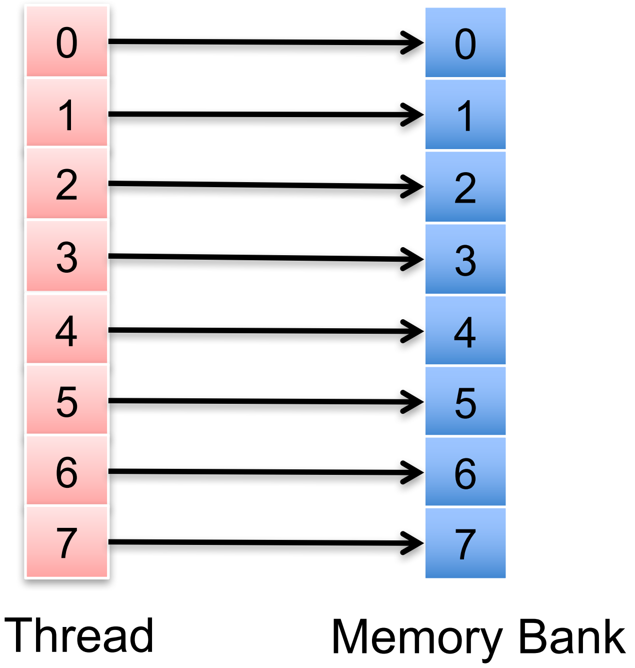
- Bank conflicts have the largest negative effect on local memory operations
  - Local memory does not require that accesses are to sequentially increasing elements
- Accesses from successive threads should target different memory banks
  - Threads accessing sequentially increasing data will fall into this category

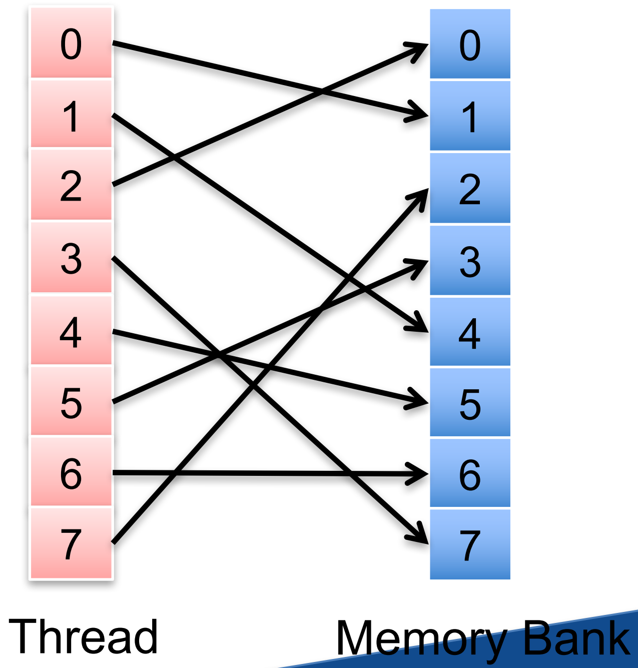
## Bank Conflicts – Local Memory

- On AMD, a wavefront that generates bank conflicts stalls until all local memory operations complete
  - The hardware does not hide the stall by switching to another wavefront
- The following examples show local memory access patterns and whether conflicts are generated
  - For readability, only 8 memory banks are shown

# Bank Conflicts – Local Memory

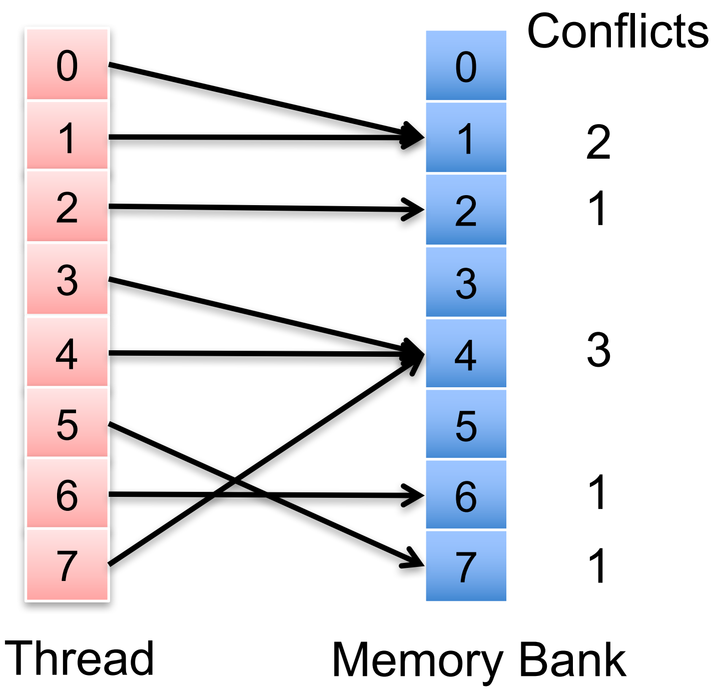
- If there are no bank conflicts, each bank can return an element without any delays
  - Both of the following patterns will complete without stalls on current GPU hardware





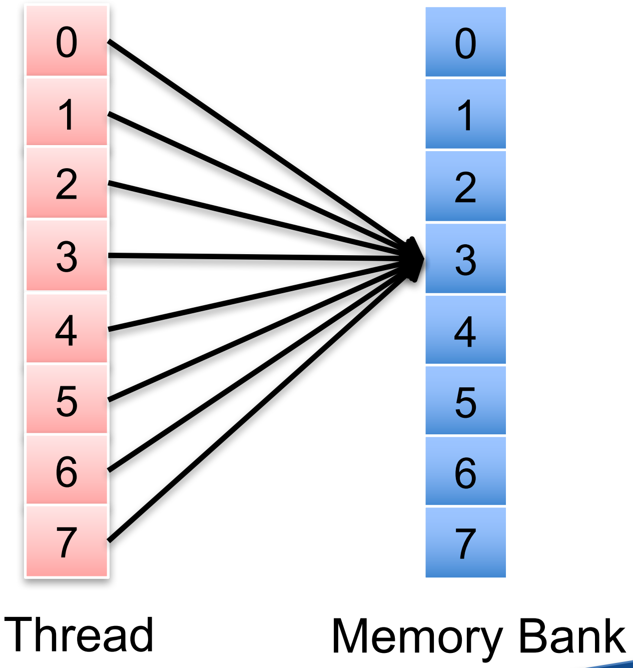
# Bank Conflicts – Local Memory

- If multiple accesses occur to the same bank, then the bank with the most conflicts will determine the latency
  - The following pattern will take 3 times the access latency to complete



# Bank Conflicts – Local Memory

- If all accesses are to the same address, then the bank can perform a broadcast and no delay is incurred
  - The following will only take one access to complete assuming the same data element is accessed



# Bank Conflicts – Global Memory

- Bank conflicts in global memory rely on the same principles, however the global memory bus makes the impact of conflicts more subtle
  – Since accessing data in global memory requires that an entire bus-line be read, bank conflicts within a work-group have a similar effect as non-coalesced accesses
    - If threads reading from global memory had a bank conflict then by definition it manifest as a non-coalesced access
    - Not all non-coalesced accesses are bank conflicts, however
- The ideal case for global memory is when different work-groups read from different banks
  – In reality, this is a very low-level optimization and should not be prioritized when first writing a program

## Summary

- GPU memory is different than CPU memory
  - The goal is high throughput instead of low-latency
- Memory access patterns have a huge impact on bus utilization
  - Low utilization means low performance
- Having coalesced memory accesses and avoiding bank conflicts are required for high performance code
- Specific hardware information (such as bus width, number of memory banks, and number of threads that coalesce memory requests) is GPU-specific and can be found in vendor documentation

# The Big Picture

- Introduction to Heterogeneous Systems
- OpenCL Basics
- Memory Issues
- Optimisations

## Thread Mapping

- Consider a serial matrix multiplication algorithm

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3< P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads
    - Effectively removing the outer two loops
  - Each thread will perform *P* calculations
    - The inner loop will remain as part of the kernel
- Should the index space be MxN or NxM?

# Thread Mapping

- Thread mapping 1: with an MxN index space, the kernel would be:

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

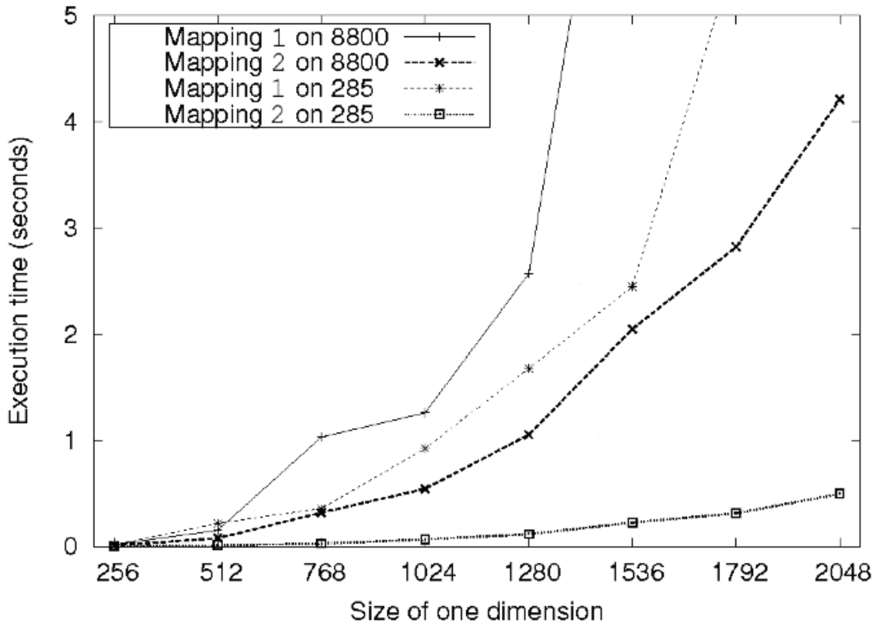
- Thread mapping 2: with an NxM index space, the kernel would be:

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- Both mappings produce functionally equivalent versions of the program

# Thread Mapping

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs



- Notice that mapping 2 is far superior in performance for both GPUs

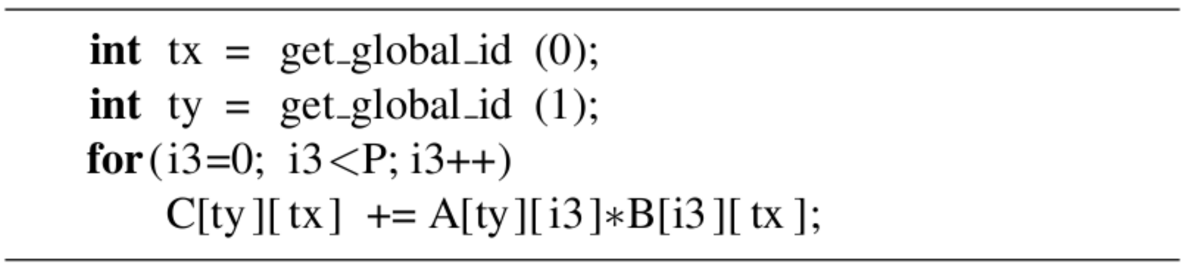
## Thread Mapping

- In mapping 1, consecutive threads (*tx*) are mapped to different rows of Matrix C, and non-consecutive threads (*ty*) are mapped to columns of Matrix B
  - The mapping causes inefficient memory accesses

```
int  tx  =  get_global_id(0);
int  ty  =  get_global_id(1);
for(i3=0;  i3<P; i3++)
      C[tx][ty]  += A[tx][i3]*B[i3][ty];
```

# Thread Mapping

- In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C
  - Accesses to both of these matrices will be coalesced
    - Degree of coalescence depends on the workgroup and data sizes



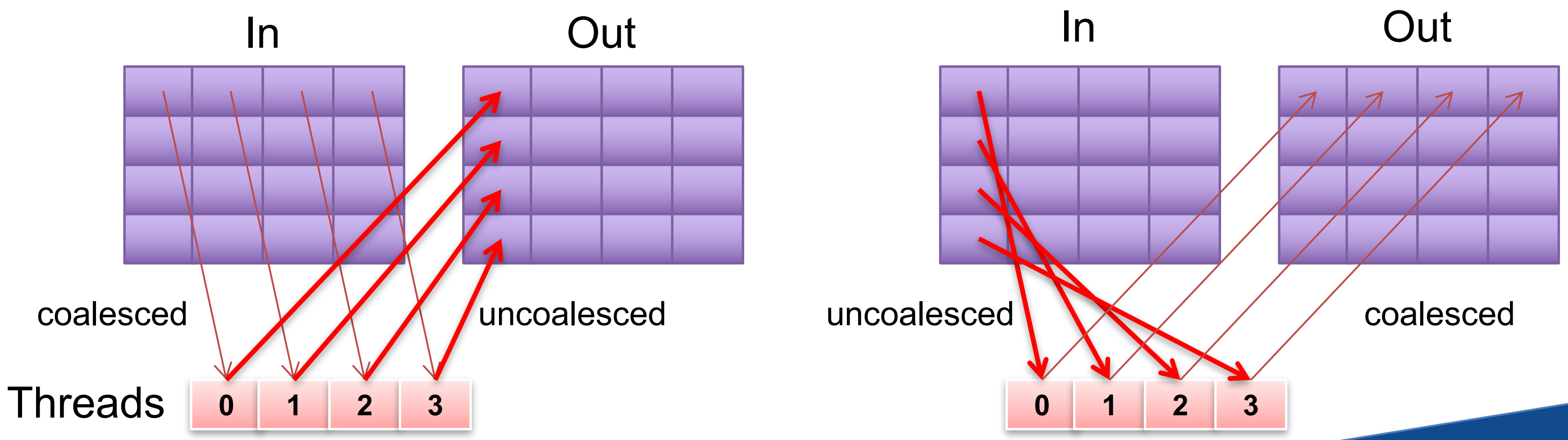
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

# Thread Mapping

- In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C
  – Accesses to both of these matrices will be coalesced
    - Degree of coalescence depends on the workgroup and data sizes

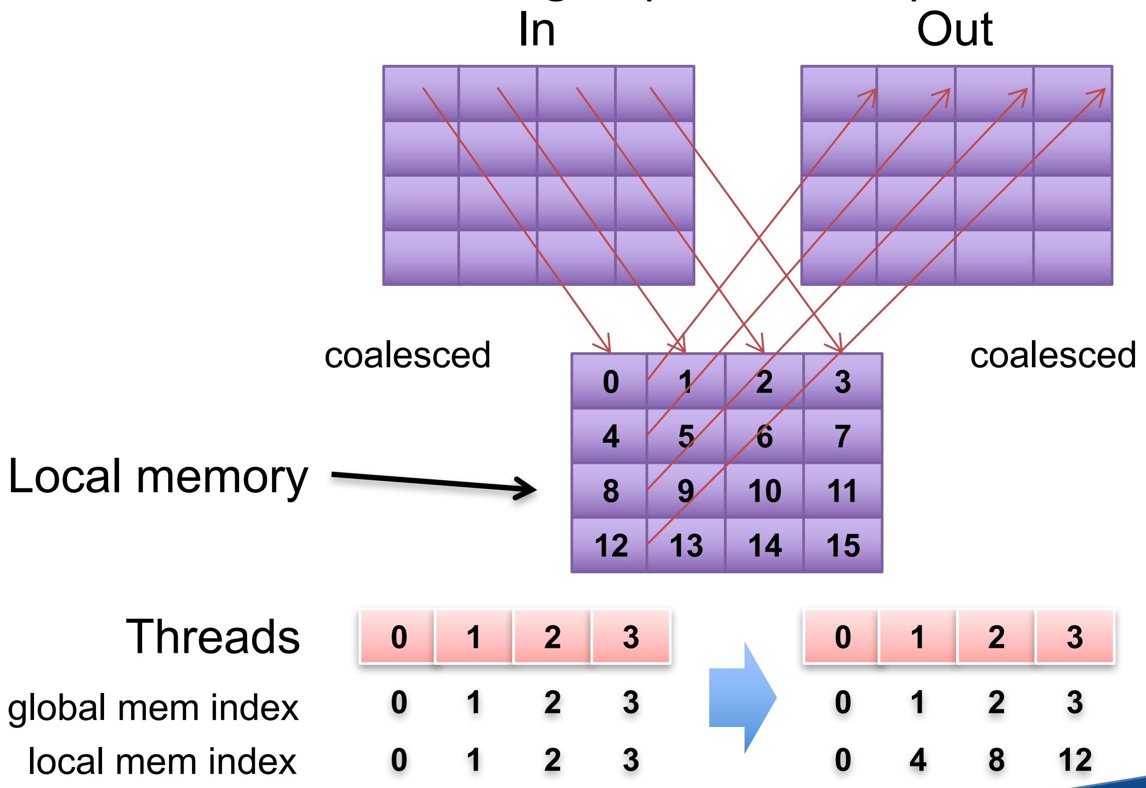
# Matrix Transpose

- A matrix transpose is a straightforward technique
  - Out(x,y) = In(y,x)
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location



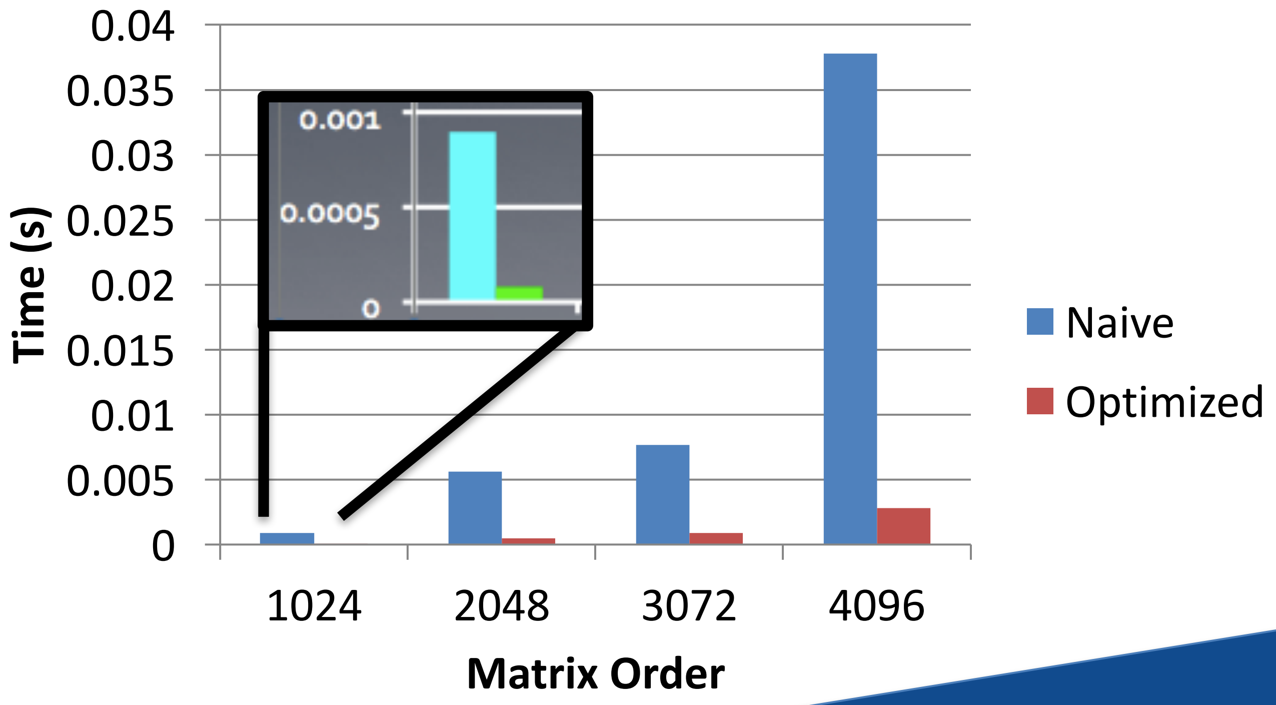
# Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
  - Note that the work group must be square



# Matrix Transpose

- The following figure shows a performance comparison of the two transpose kernels for matrices of size NxM on an AMD 5870 GPU
  - "Optimized" uses local memory and thread remapping



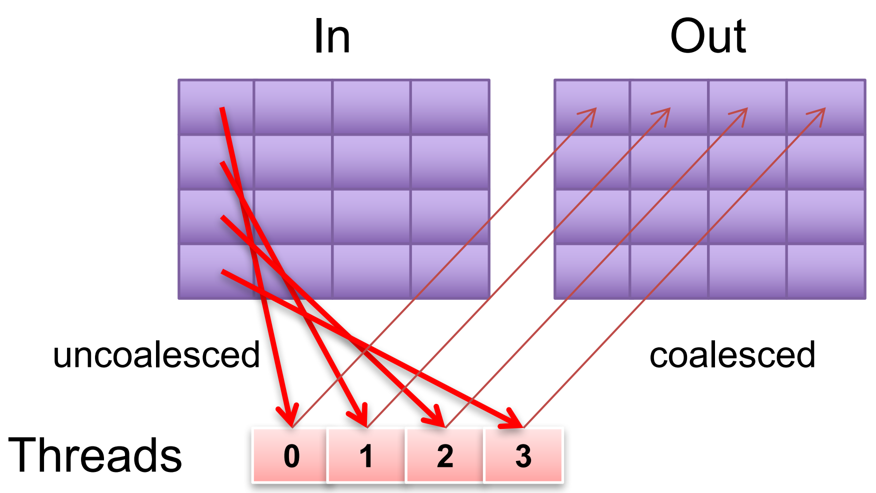
# Runtimes on Fermi

| | CR | CW | SH-no | SH-all |
|---|---|---|---|---|
| 16x16 | 21 | 23 | 34 | 18 |
| 32x32 | 57 | 44 | 110 | 46 |
| 1x256 | 101 | 98 | | |
| 1x512 | 221 | 113 | | |
| 1x1024 | 289 | 117 | | |
| 256x1 | 100 | 160 | | |
| 512x1 | 112 | 208 | | |
| 1024x1 | 117 | 298 | | |

HOST: 302/683 !!!

All times in msec.
Matrix size: 4096x4096

# What happened?

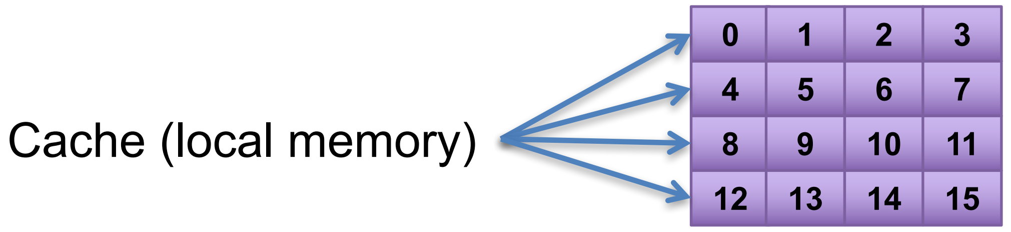


Thread 0 issues read

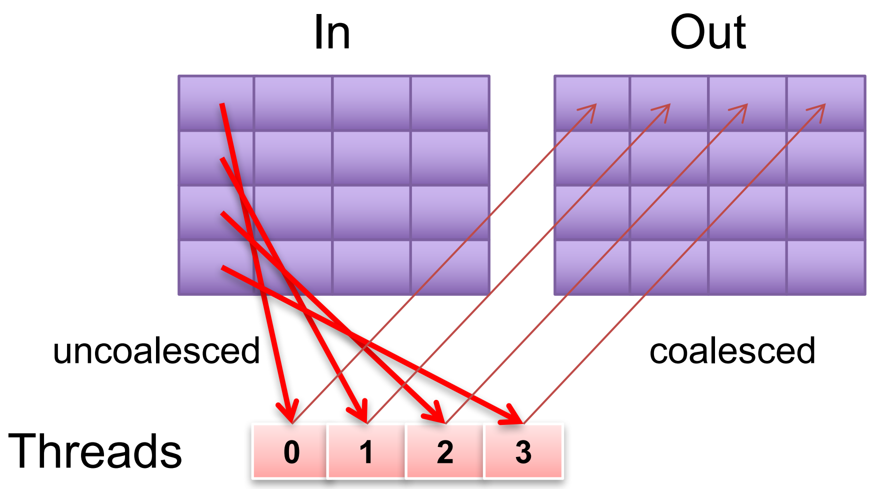
Thread 1 issues read

Thread 2 issues read

Thread 3 issues read



# What happened?



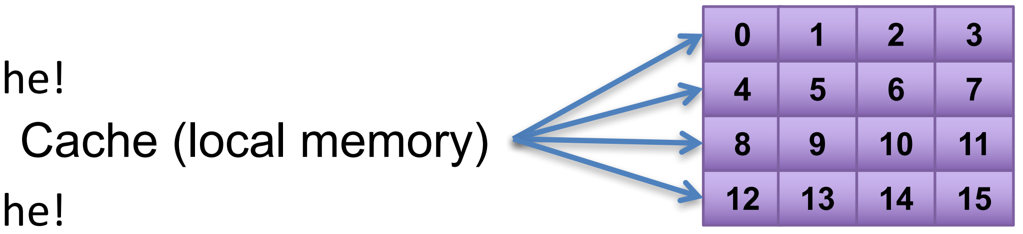
Coalesced write!

Coalesced read from cache!

Coalesced write!

Coalesced read from cache!

…



# Summary

- When it comes to performance, memory throughput and latency hiding are key!
- Main Tools are:
  - Memory choice (global/local/ private)
  - Memory layout (coalescing & indexing)
  - Thread Mapping
  - Workgroup size (synchronisation &latency hiding)