VRIJE UNIVERSITEIT BRUSSEL

GPU Computing

» Lesson 5: Performance Limiters

Jan Lemeire
2019-2020

http://parallel.vub.ac.be

VRIJE UNIVERSITEIT BRUSSEL

GPU processing power is not for free

Obstacle 1

Hard to implement

Obstacle 2

Hard to get efficiency

- The potential peak performance is given by the roofline model
  - Computational Intensity of kernel determines whether *computation* or *memory* bound.

- However, ***performance limiters*** will introduce **overhead** and result in lower performances
  - Deviations from the peak performance are due to **lost cycles**: cycles during which other instructions could have been executed, the pipeline is not used most efficiently
    - **Idle cycles, or**
    - **Cycles of inefficient execution of instructions**

VRIJE UNIVERSITEIT BRUSSEL

Estimate overhead

- Estimate a performance bound for your kernel
  - *Compute bound*: $\mathbf{t_1}$ = #**operations** / #**operations per second** (peak performance)
  - *Memory bound*: $\mathbf{t_2}$ = # **memory accesses** / #**accesses per second** (bandwidth)
  - Minimal runtime $\mathbf{t_{min}} = \mathbf{max}(\mathbf{t_1}, \mathbf{t_2})$
    expressed by roofline model
- Measure the actual runtime
  - $\mathbf{t_{actual}} = \mathbf{t_{min}} + \mathbf{t_{overhead}}$
- Try to account for and minimize $\mathbf{t_{overhead}}$

Performance Limiters

1. Occupancy

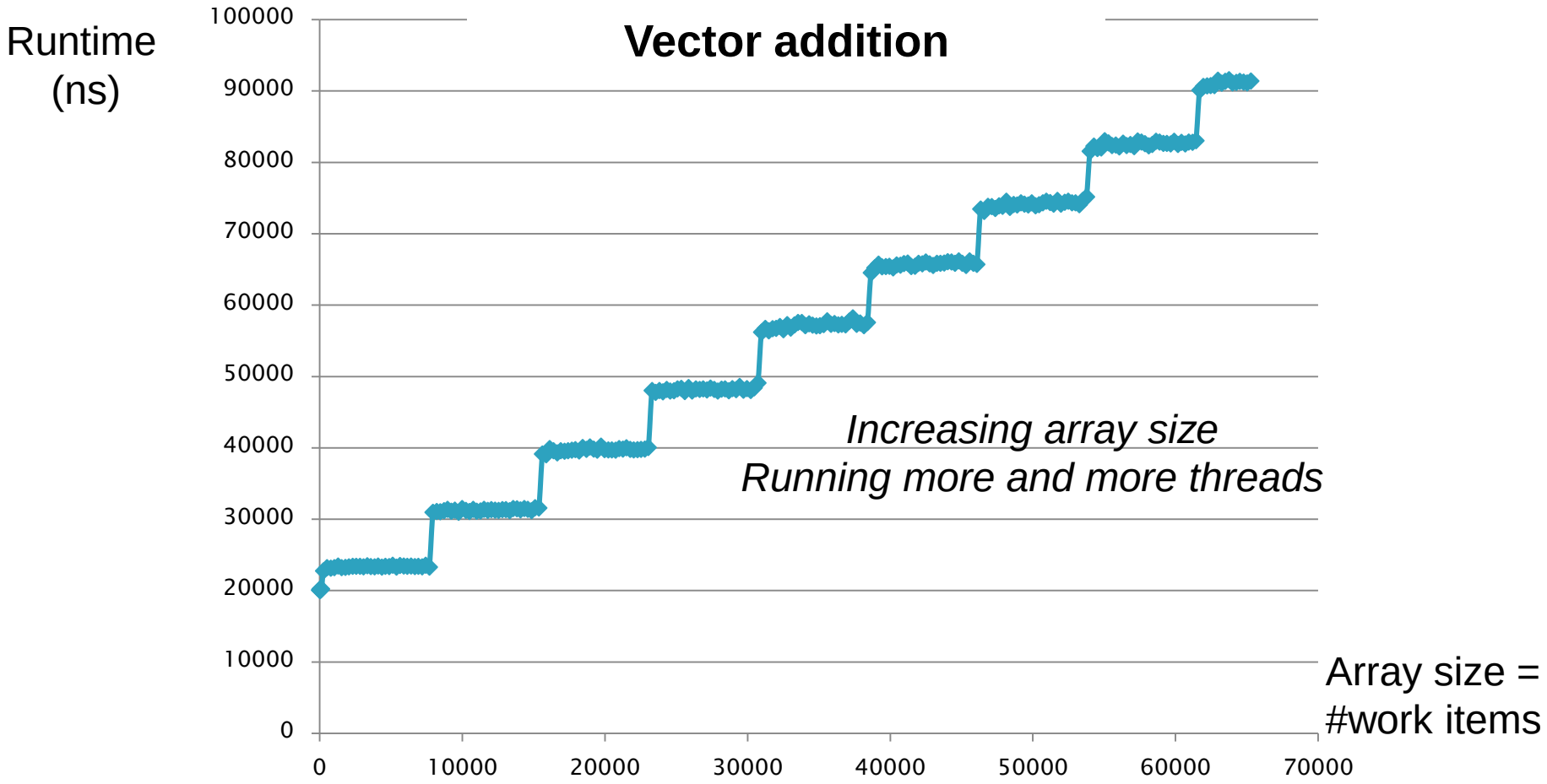
VRIJE UNIVERSITEIT BRUSSEL

Keep all processing units busy

Enough parallelism (work items) is necessary

- For all cores (= MultiProcessors = Compute Units)
- For all Scalar Processors (SPs = Processing Elements)
  - Hardware threads (warps) enable SIMT (lesson 3)
- To fill pipeline of scalar processor
  - With instructions of different warps
  - = Simultaneous multithreading (lesson 3)
  - Results in Latency hiding

The effect of parallelism



Only when all pipelines are full, the runtime increases

VRIJE UNIVERSITEIT BRUSSEL

The effect of parallelism

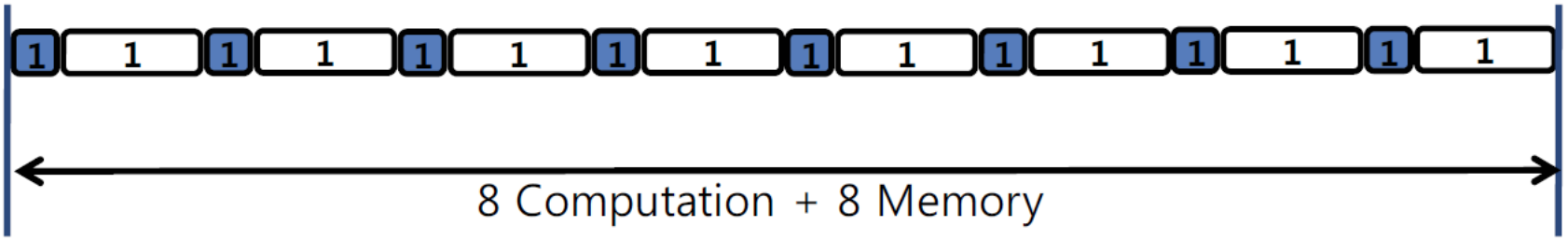
- Processor needs sufficient work groups/work items to keep the system busy, to keep all pipelines full; to get full performance.
- if GPU is not fully used, additional work can be scheduled without cost
  - see previous slide with graph of runtime in function of the number of threads for a vector addition
  - the runtime does not increases as long as GPU is not full.
    - function shaped as a *staircase*
  - only just before the jump to the next step the GPU is fully busy

- Additionally, concurrent threads also needed for latency hiding.

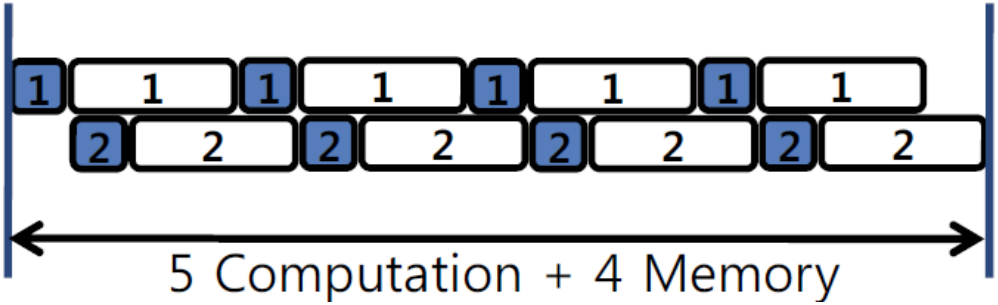
VRIJE UNIVERSITEIT BRUSSEL

Hiding of Memory Latencies

- 1 warp, without latency hiding



- 2 warps running concurrently



- 4 warps running concurrently: full latency hiding

VRIJE UNIVERSITEIT BRUSSEL

Maximize Parallelism & Occupancy

- A great number of work groups:
  - A multiple of the number of cores times the occupancy in work group count
  - If each core can run 4 work groups simultaneously, the number of work groups should be at least 4 * #cores

- *Occupancy* = Number of warps running concurrently on a core
  - Relative occupancy = occupancy divided by maximum number of warps that can run concurrently on a core
  - Is determined by *4 hardware resources*, see lesson 3

Performance Limiters

2. ILP & MLP

VRIJE UNIVERSITEIT BRUSSEL

Dependent Code

- Well-known fact: latency is hidden by launching other threads
- Less-known fact: one can also exploit *Instruction Level Parallelism* (ILP) in one thread.
  - Data level parallelism in one thread.
- **Performance limiter is absence of ILP or MLP**:
  - Dependent instructions can not be parallelized.
  - Dependent memory accesses can not be parallelized.

VRIJE UNIVERSITEIT BRUSSEL

Maximize parallelism on the compute unit

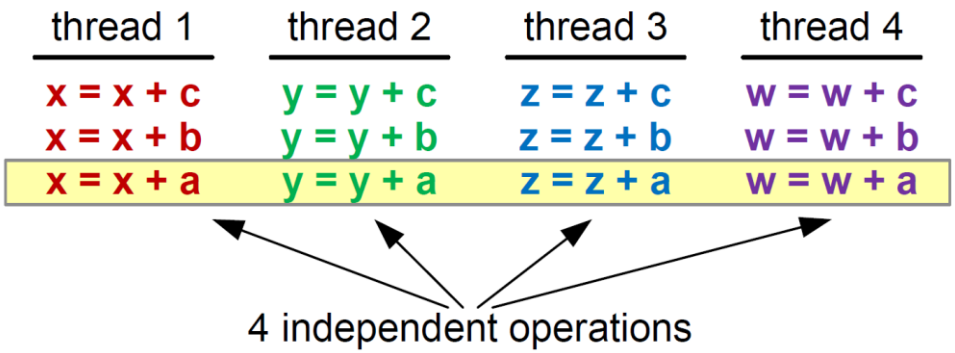
- Occupancy = *Thread-Level Parallelism* (TLP)
  - Scheduler has more choice to fill the pipeline
- *Instruction Level Parallelism* (ILP)
  - Independent instructions within one warp
  - Can be executed concurrently
- *Memory Level Parallelism* (MLP)
  - Independent memory requests for one warp
  - Can be serviced concurrently

Peak performance is reached for lower occupancies (fewer concurrent warps) if the ILP and MLP are increased.

TLP versus ILP and MLP

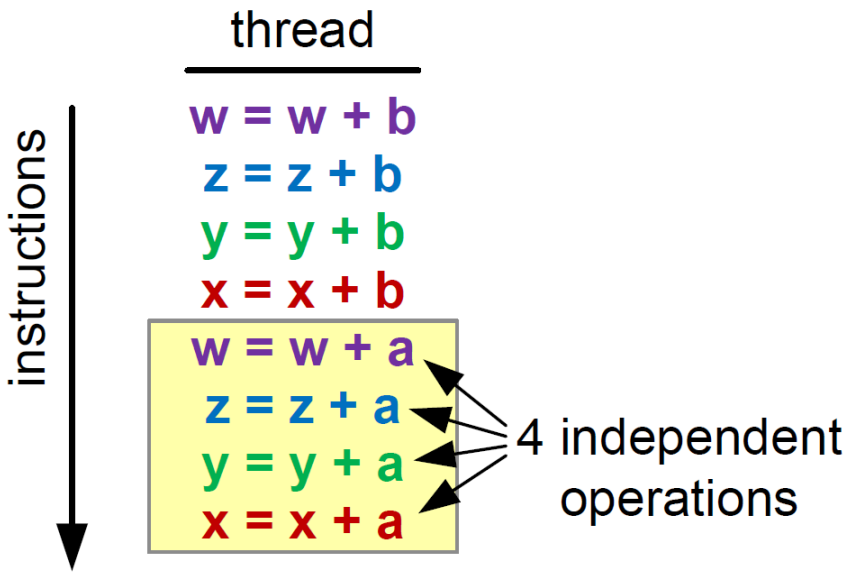
Thread-Level Parallelism

- Independent threads



Instruction-Level Parallelism

- Independent instructions



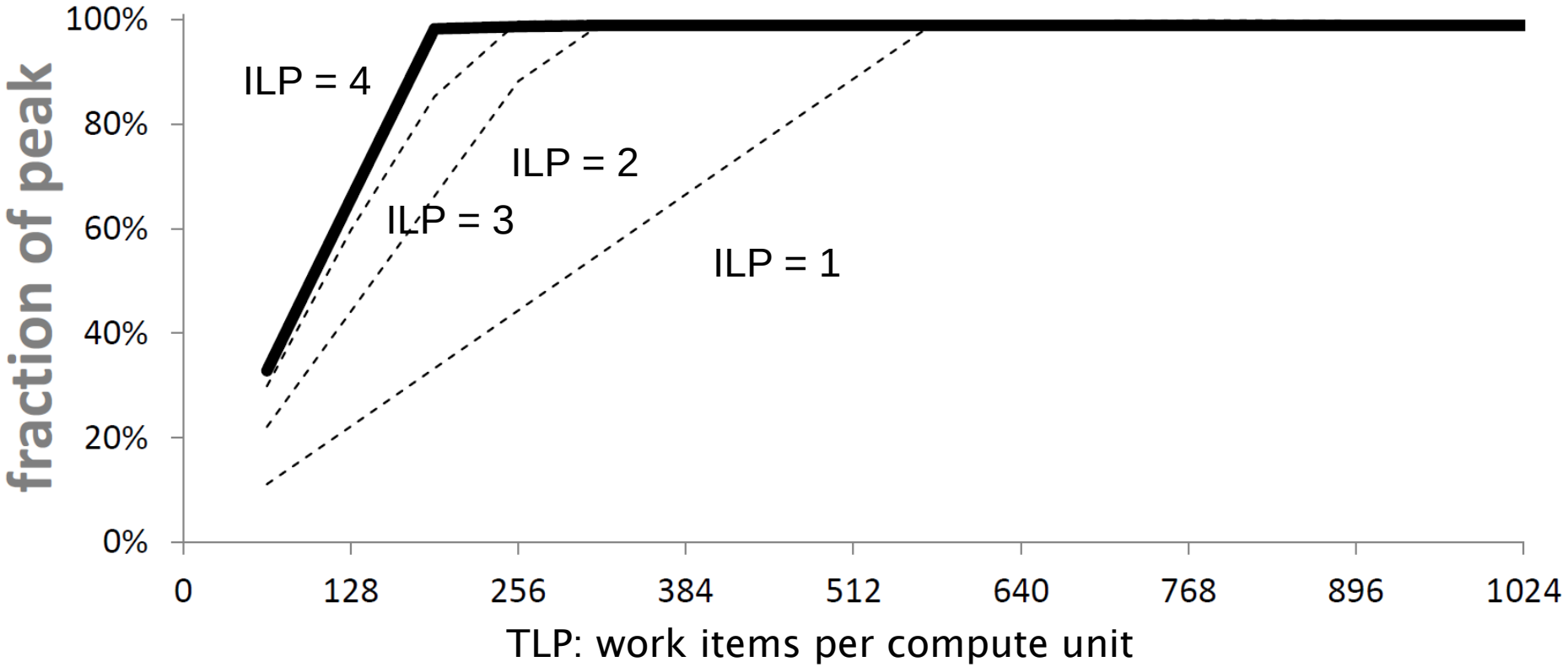
Memory-Level Parallelism

- One thread reading / writing 2, 4, 8, 16, … floating point values

VRIJE UNIVERSITEIT BRUSSEL

Computational Performance

A function of TLP and ILP

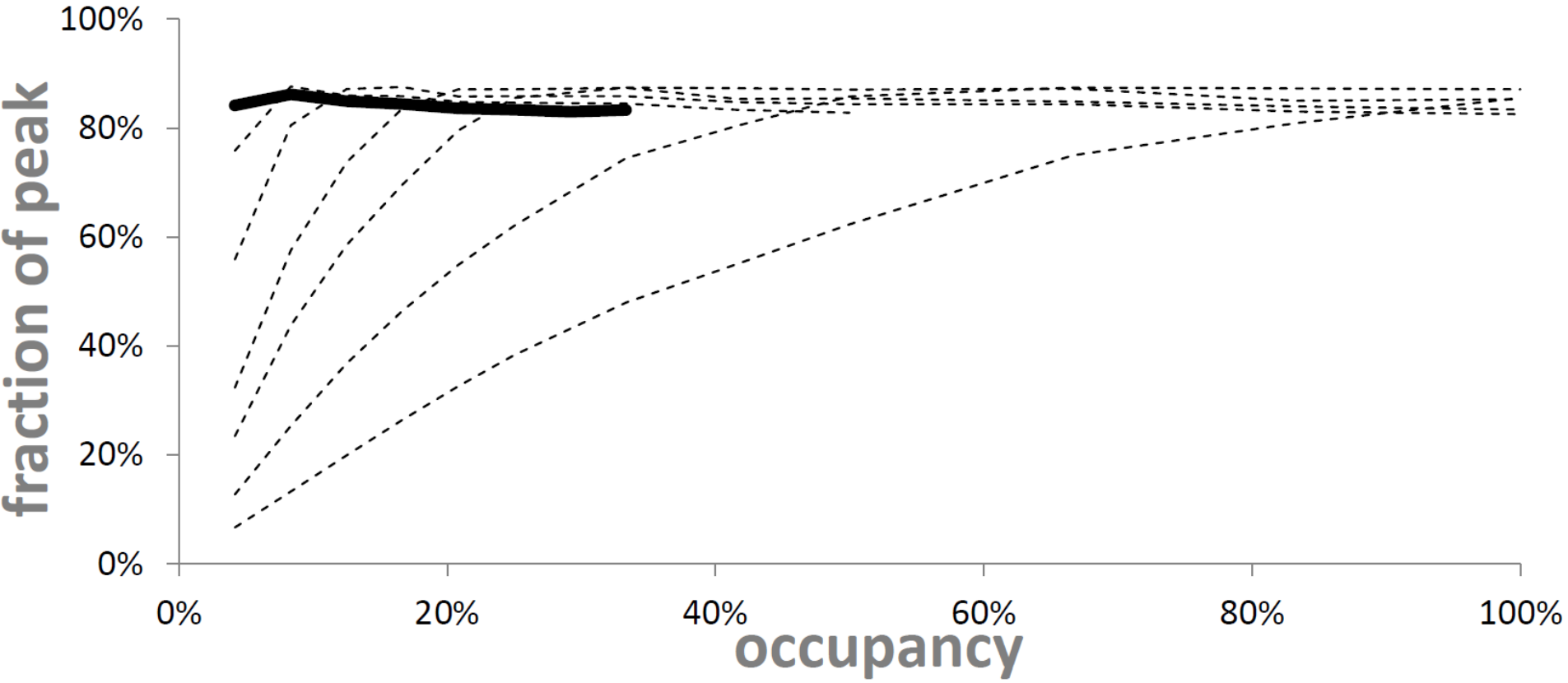


Occupancy roofline

VRIJE UNIVERSITEIT BRUSSEL

Memory throughput
A function of TLP and MLP

- MLP: 1 float, 2 float, 4 float, 8 float, 8 float2, 8 float4 and 14 float4
- TLP: occupancy



Performance Limiters

3. Branch divergence

VRIJE UNIVERSITEIT BRUSSEL

SIMT Conditional Processing

- Unlike threads in a CPU-based program, **SIMT threads cannot follow different execution paths**
  - All threads of a warp/wavefront are executing the same instruction, they are executed **in lockstep**
- Program flow diverged is solved by instruction predication
- Example kernel: `if (x < 5)  y = 5; else y = -5;`
  - The SIMT warp performs all 3 instructions
  - `y = 5;` is only executed by threads for which `x < 5`
  - `y = -5;` is executed by all others
  - a bit is used to enable/disable actual execution
  - See lesson 3
- *Warp branch divergence* decreases performance: cycles are lost

VRIJE UNIVERSITEIT BRUSSEL

Example: tree traversal

- Given: a (search) tree
- Each work item does a lookup in the tree: follows a (different) path in a tree, from root to leave.
  - Implemented with a while-loop
- If not all leaves are at the same depth: the highest depth determines the execution time of a warp/wavefront
- Imbalances in the tree result in many lost cycles

VRIJE UNIVERSITEIT BRUSSEL

Branch Divergence Remedies

- Static thread reordering
  - Group threads which will follow the same execution path
  - Typical in reduction operations, see extended example at the end of lesson
- Dynamic thread reordering
  - Reorder at runtime, e.g. using a lookup table
  - OK if time lost reordering < time won due to reordering

Performance Limiters

4. Synchronization

VRIJE UNIVERSITEIT BRUSSEL

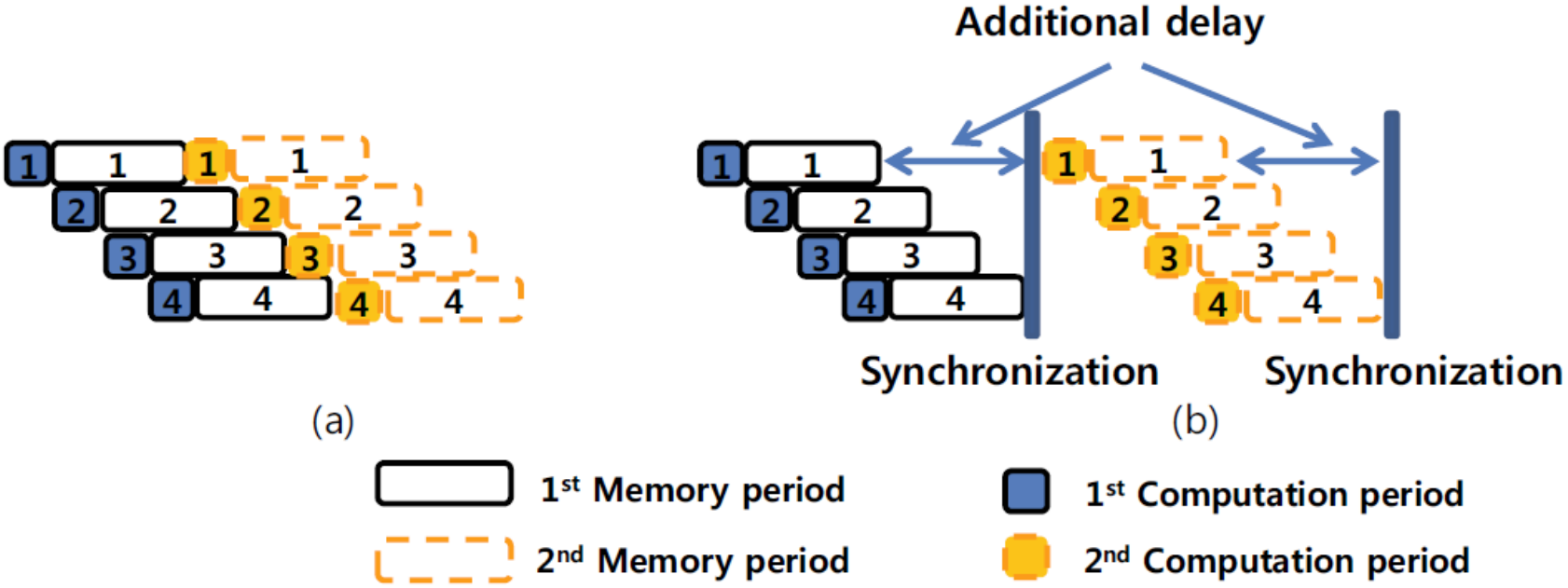
Local and global synchronization (see lesson 2)

- Local synchronization
  - Work items of the same group can synchronize:

    ```
    barrier(CLK_LOCAL_MEM_FENCE);
    ```
  - Work items that reach the barrier must wait
    - Cannot be chosen by the scheduler
    - ➔ Less potential for latency hiding

- Global synchronization should happen across kernel calls
  - A new kernel must be launched to ensure synchronization (work groups have all reached the same spot in the algorithm)
  - Overhead!

Lost cycles due to local synchronization



No synchronization

Barrier after each memory period

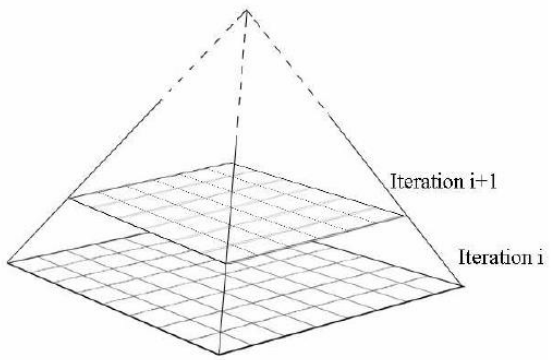
VRIJE UNIVERSITEIT BRUSSEL

Minimize synchronization overhead

- Local synchronization:
  - Keep work groups small → less effect
    - with multiple concurrent work groups latency hiding is still possible
  - No synchronization is needed within a warp because they run in lockstep anyway!

Minimize synchronization overhead

- Global synchronization
  - *Exchange computations for memory access*
  - E.g. Hotspot: simulate heat flow (e.g. on a chip)
    - $Heat_{point} = f(heat_{neighbors})$
    - Points are partitioned over the work groups, each work group simulates NxN points
    - Calculate for NxN points and globally synchronize after each time step?
    - **No**: calculate different iterations independently with overlapping borders for each work group
      - Iteration 0: (N+k)x(N+k) points
      - …
      - Iteration k-1: NxN points



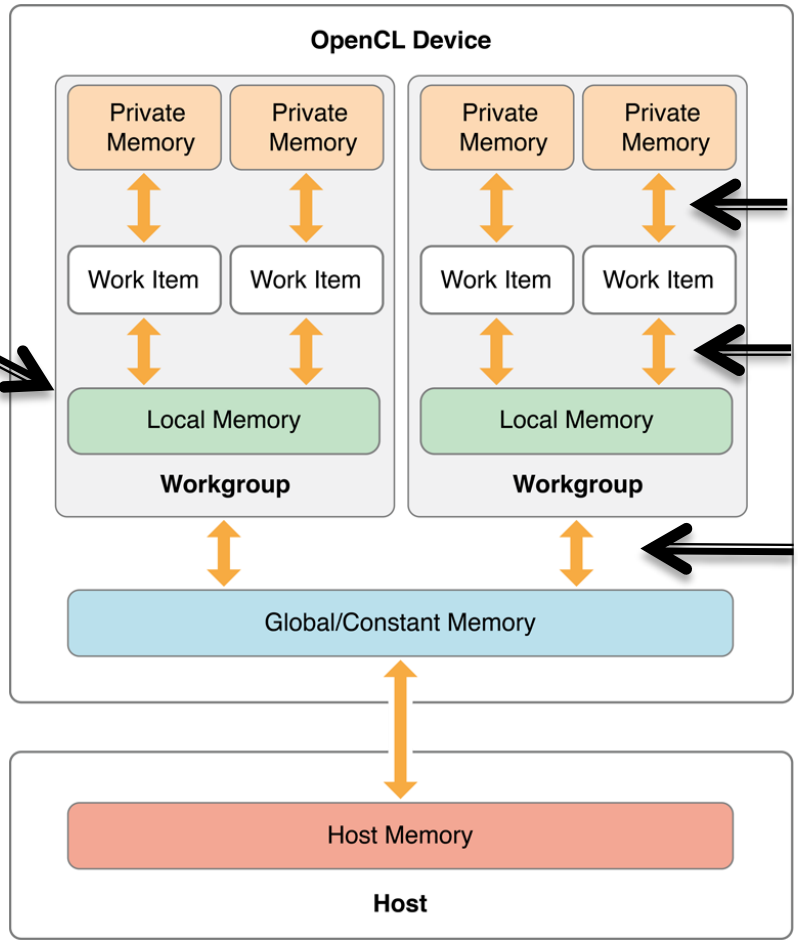
Performance Limiters

5. Memory hierarchy

VRIJE UNIVERSITEIT BRUSSEL

Architecture – Memory Model

Core/Compute unit



VRIJE UNIVERSITEIT BRUSSEL

Exploit memory hierarchy

- Data placement is crucial for performance
- Maximally use local memory and private memory (registers)
  - Copy shared data to local memory
  - See examples of Convolution or Matrix Multiplication

VRIJE UNIVERSITEIT BRUSSEL

Memory Levels

- Global memory
  - Share data between GPU and CPU
  - Large latency and low throughput
    - ➔ Access should be minimized
  - Cached in L2-cache on modern GPUs
- Constant memory
  - Share read-only data between GPU and CPU
  - Is cached in L1 cache
  - Limited size. Typically 64 KB
  - Prefer it to local memory for small read-only data

VRIJE UNIVERSITEIT BRUSSEL

Memory Levels

- Local memory
  - Share data within a work group
  - Use it if the same data is used by multiple work items in the same work group
- Private memory (registers)
  - Lowest latency highest throughput
  - Watch out: private arrays will be stored in global memory, but cached in L1-cache

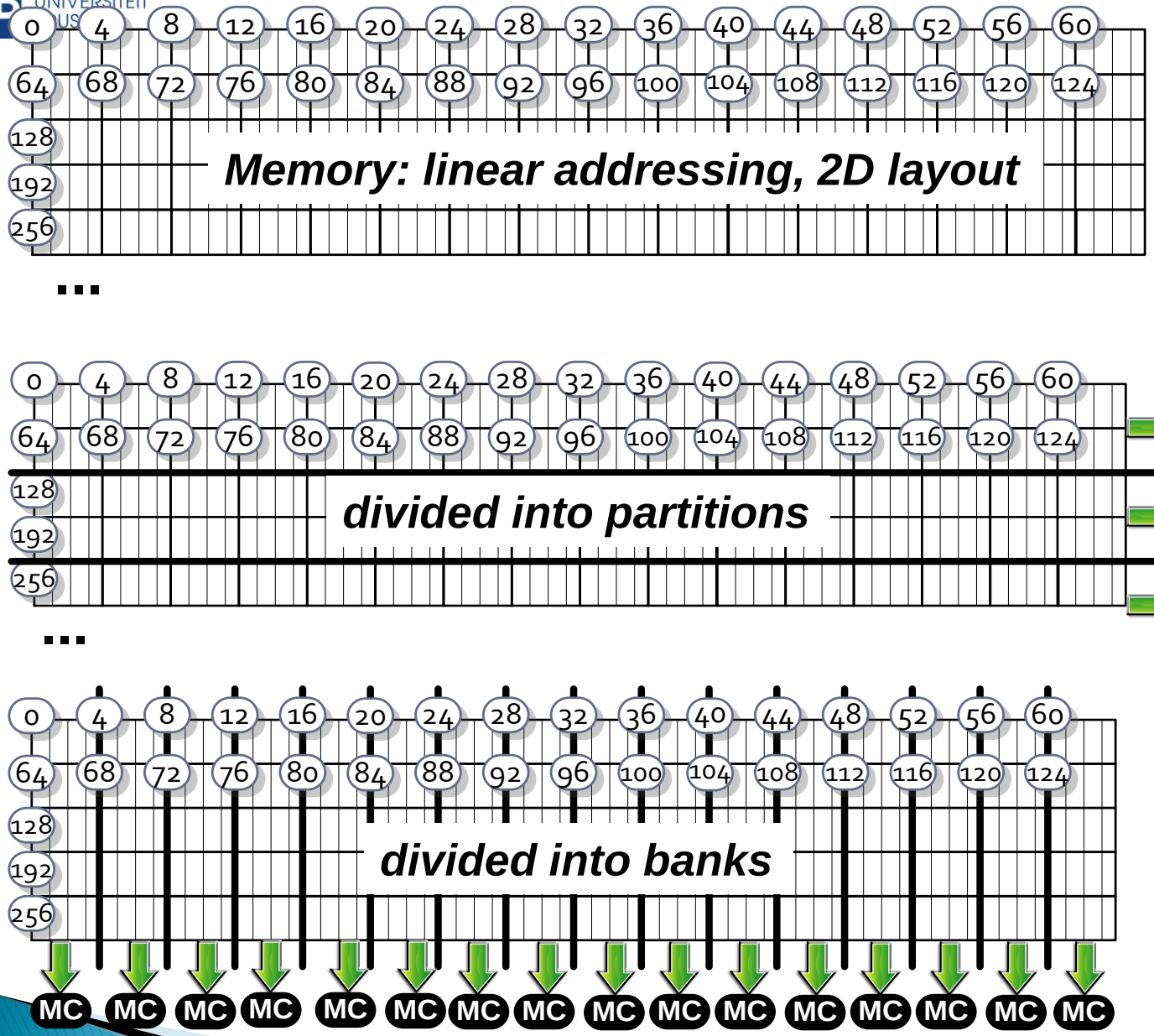
Performance Limiters

6. Concurrent memory access

VRIJE UNIVERSITEIT BRUSSEL

Concurrent Memory Access

- Each Compute Unit has active threads:
  - Simultaneous access of global memory
- Each hardware thread (warp) executes 32/64 kernel threads
  - Simultaneous access of global memory
  - Simultaneous access of local memory
- But: concurrent memory access is limited by the hardware!
  - *Efficient access depends on memory organization*
  - Let's discuss this for global an local memory


0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
64 68 72 76 80 84 88 92 96 100 104 108 112 116 120 124
128
192
256
Memory: linear addressing, 2D layout
...
divided into partitions
MC
MC
MC
divided into banks
MC MC MC MC MC MC MC MC MC MC MC MC MC MC MC MC
Memory Controllers: Can handle 1 request at a time

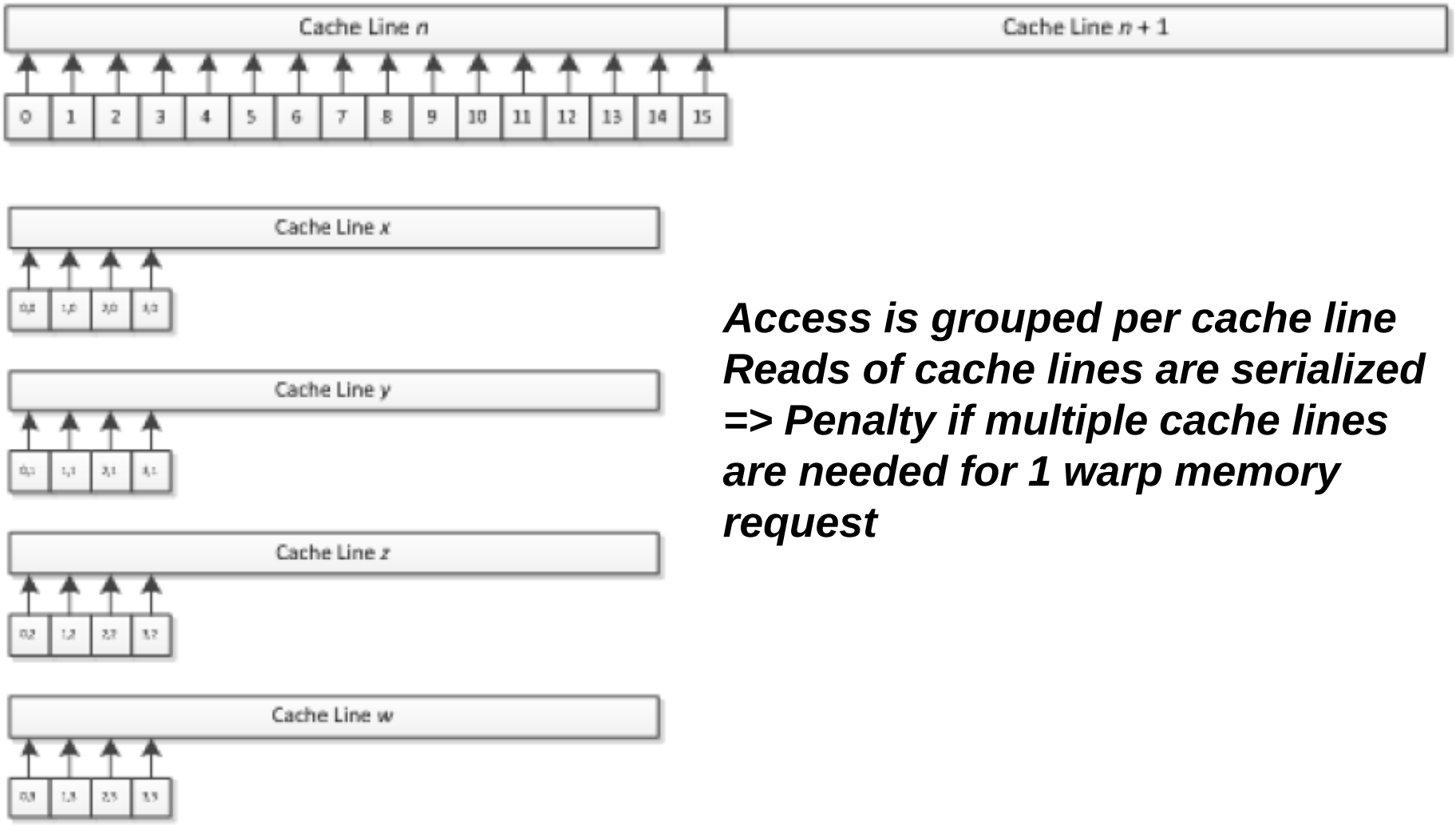
Global memory

- Divided into partitions
  1. NVIDIA GPUs typically have 8 partitions
  2. Memory controller can serve 1 segment at a time (≈ cache line of 4x32 Bytes)
- 1: Active warps of different cores/multiprocessors simultaneously access global memory
  - **Partition camping** when they access the same partition => serialization of memory requests
  - This is difficult to control and overcome…
- 2: Memory **coalescing** for warps
  - Accessed elements of a warp should belong to same aligned segment
  - if not (uncoalesced access), memory requests are serialized => will take more time

Global Memory Access

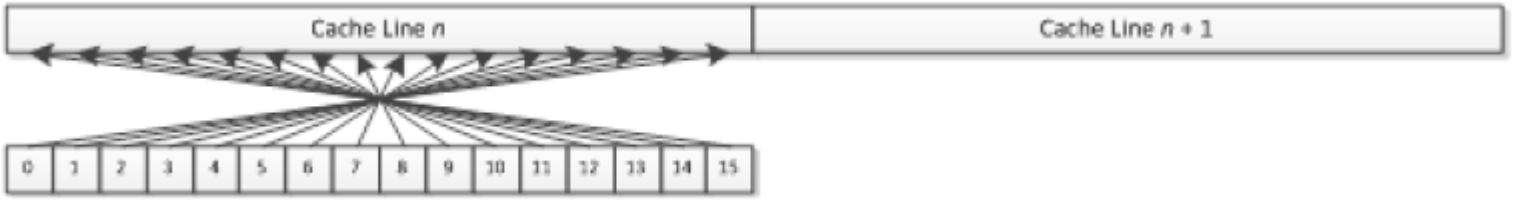
- Global memory is organized in segments (cache line), a memory controller can serve 1 segment at a time.
- Memory requests of warp are handled together
  - Data elements of the same segment are grouped and will be served together
- Ideal situation:
  - All bytes of necessary segments are needed
  - The number of bytes that need to be accessed to satisfy a warp memory request is equal to the number of bytes actually needed by the warp for the given request
- A few examples will clarify this

Concurrent data access

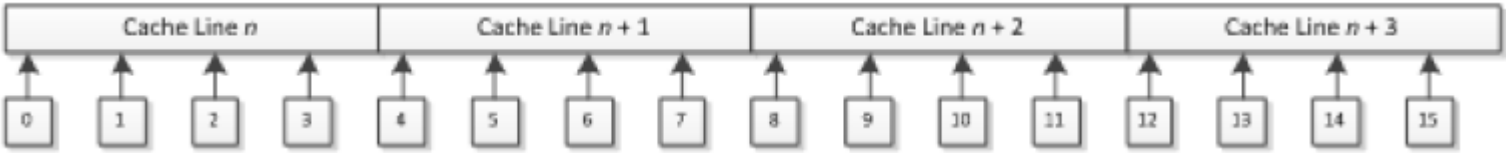


Access is grouped per cache line
Reads of cache lines are serialized
=> Penalty if multiple cache lines are needed for 1 warp memory request

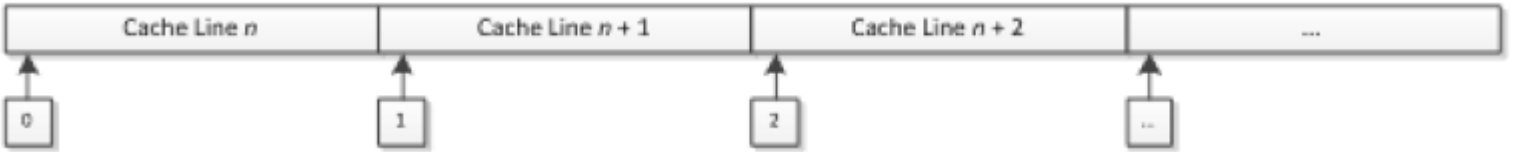
Concurrent data access



Stride of 4 => 1/4th of performance



Stride of 16 => 1/16th of performance

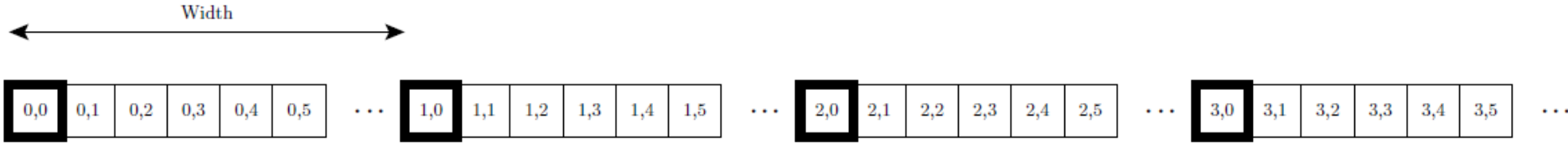


Global Memory Access

Impact of strided access

- 2-D and 3-D data stored in flat memory space
  - Strided access is not a good idea (e.g. access columns)

Quadro K620	AMD HD 7950
Aligned: 26 GB/s	Aligned: 170 GB/s
Strided: 7 GB/s	Strided: 4 GB/s



Global Memory Access

Array of struct vs struct of arrays

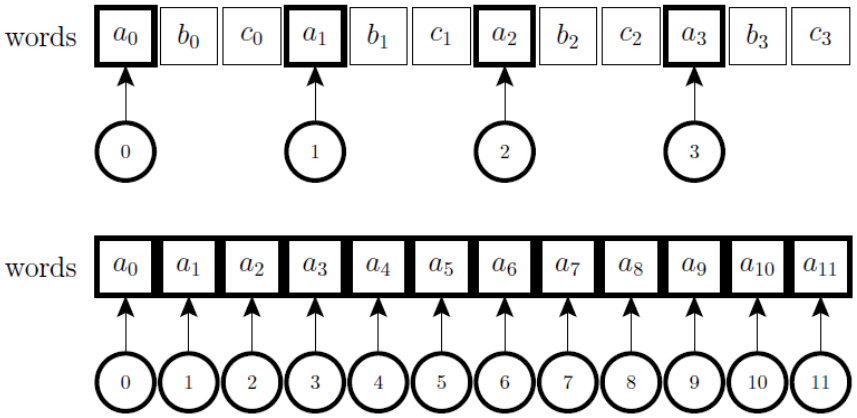
```
typedef struct {
        float a, b, c;
} triplet_t;

__kernel void aos(__global triplet_t
*triplets) {
    float a =
triplets[get_global_id(0)].a;
}

__kernel void soa(__global float *as,
                  __global float *bs,
                  __global float *cs)
{
    float a = as[get_global_id(0)];
}
```

AOS introduces strides

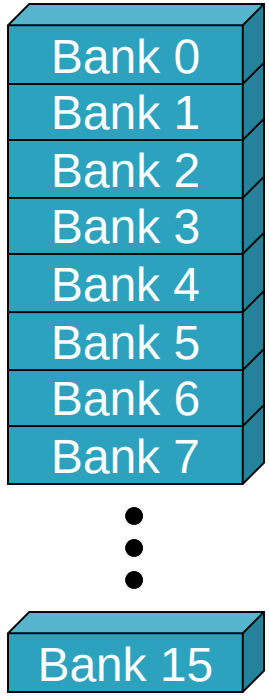
If elements are visited at different moments



SOA removes strides

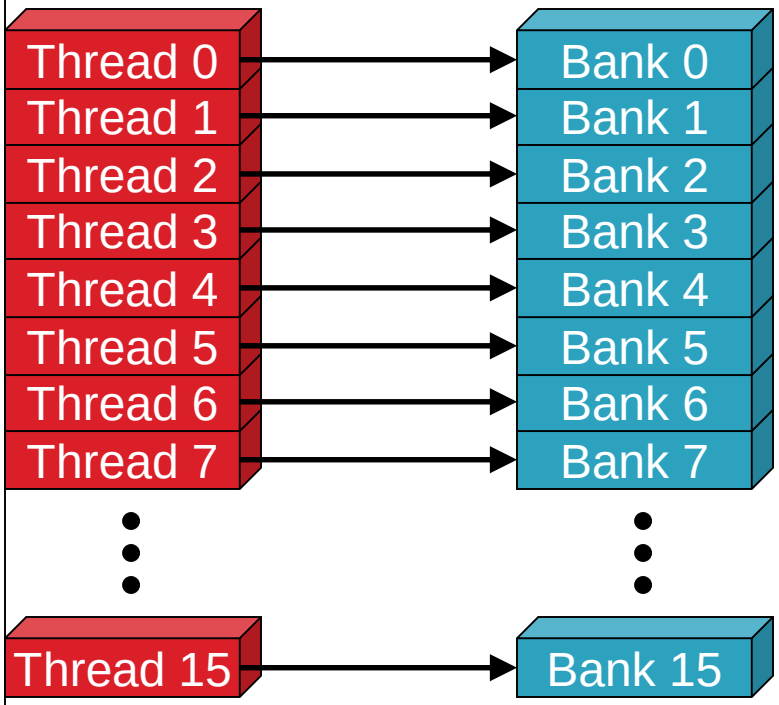
Local Memory access

- Local memory is divided into banks
- Each bank can service one address per cycle
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
  - Cost = max # simultaneous accesses to single bank
- No bank conflicts when
  - All work items of warp access another bank
  - All work items of warp read *the same address*

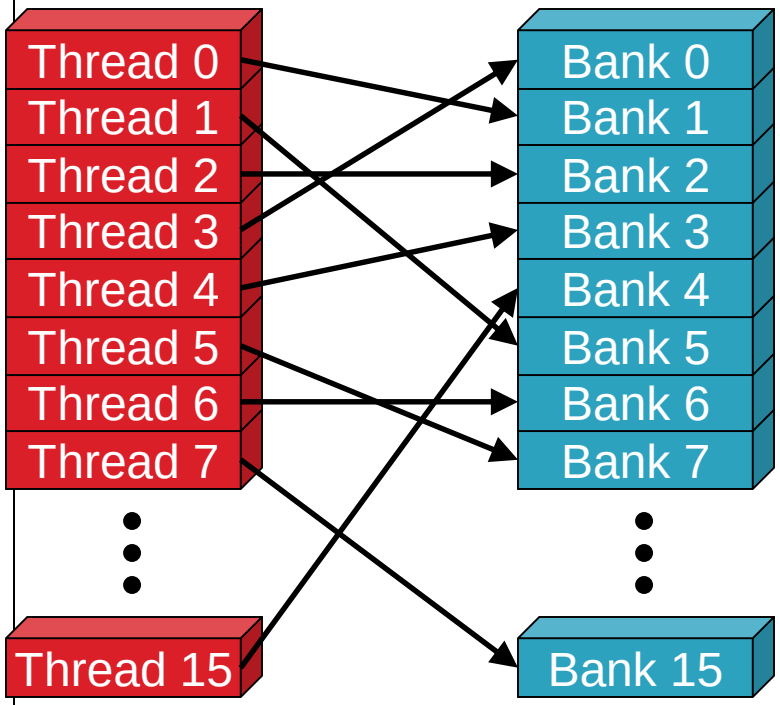


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride of 1

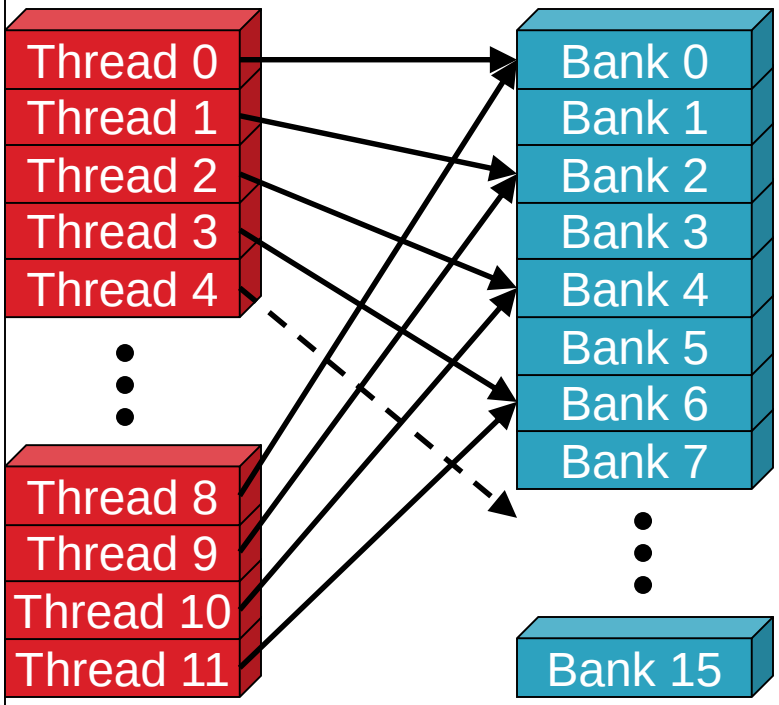


- No Bank Conflicts
  - Random 1:1 Permutation

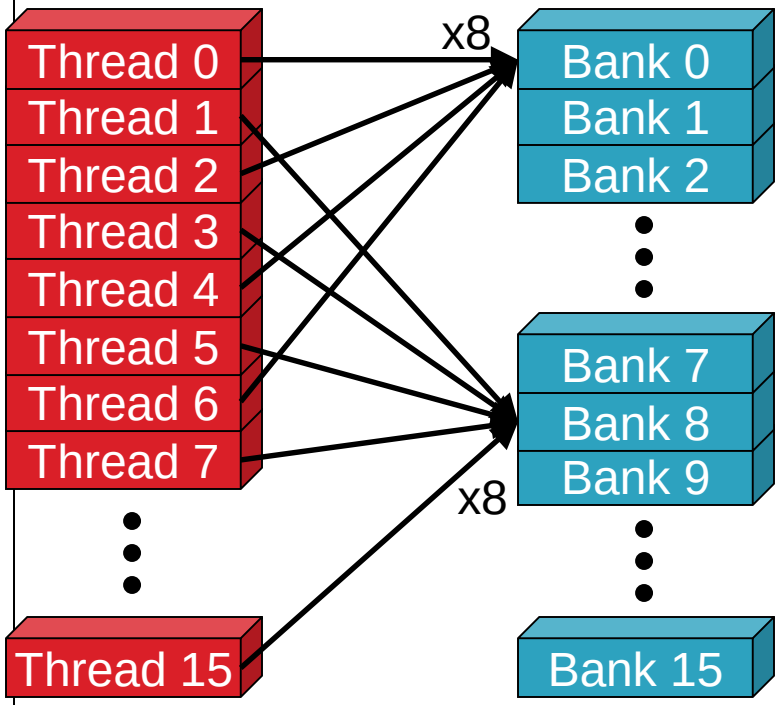


Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride of 2

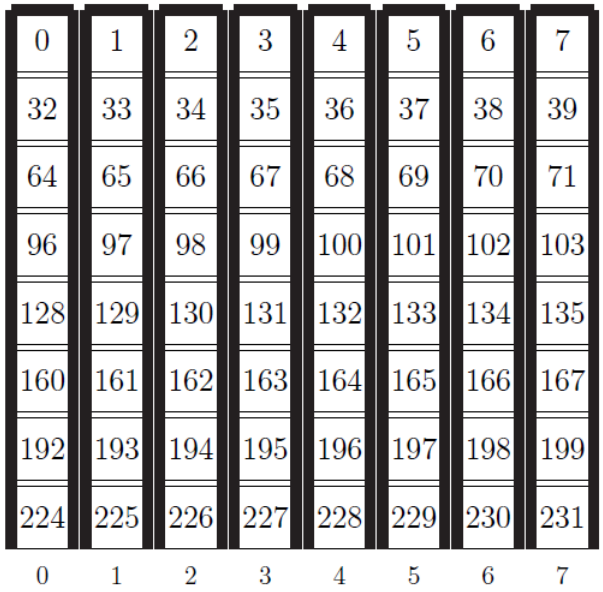


- 8-way Bank Conflicts
  - Linear addressing stride of 8



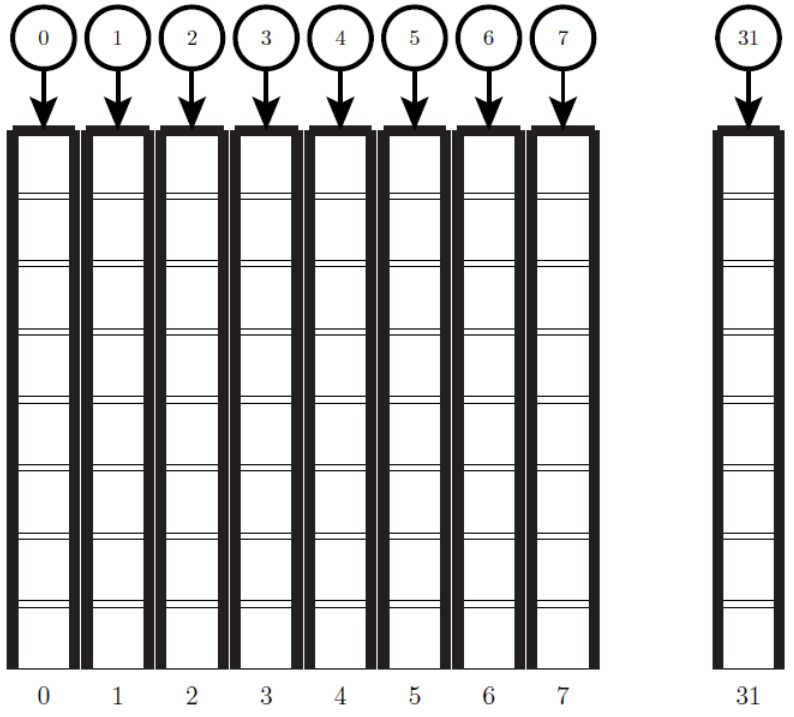
Local Memory access

- Word storage order:
  - Banks are 4 bytes wide



- Row access

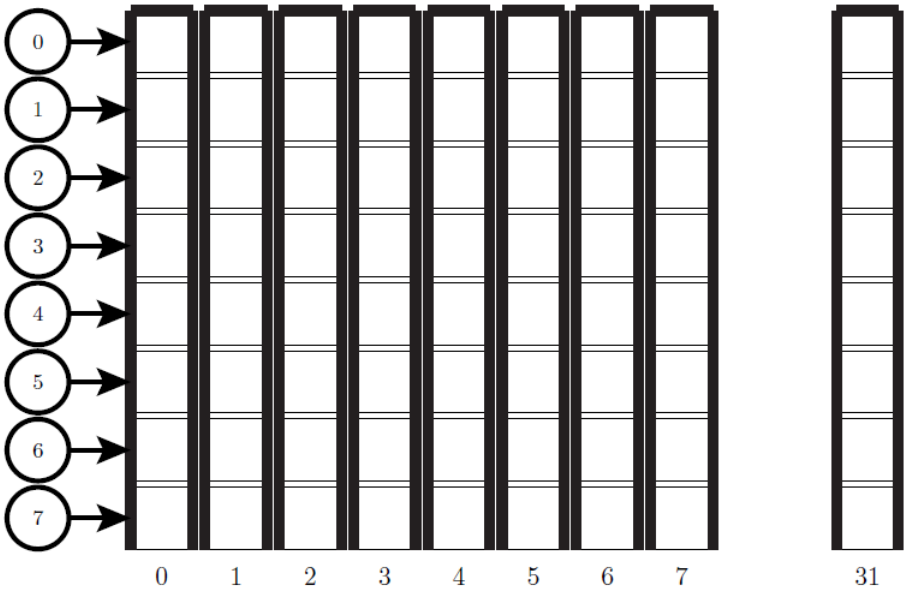
```
__local float sh[32][32];
```



Local Memory access

- Column access

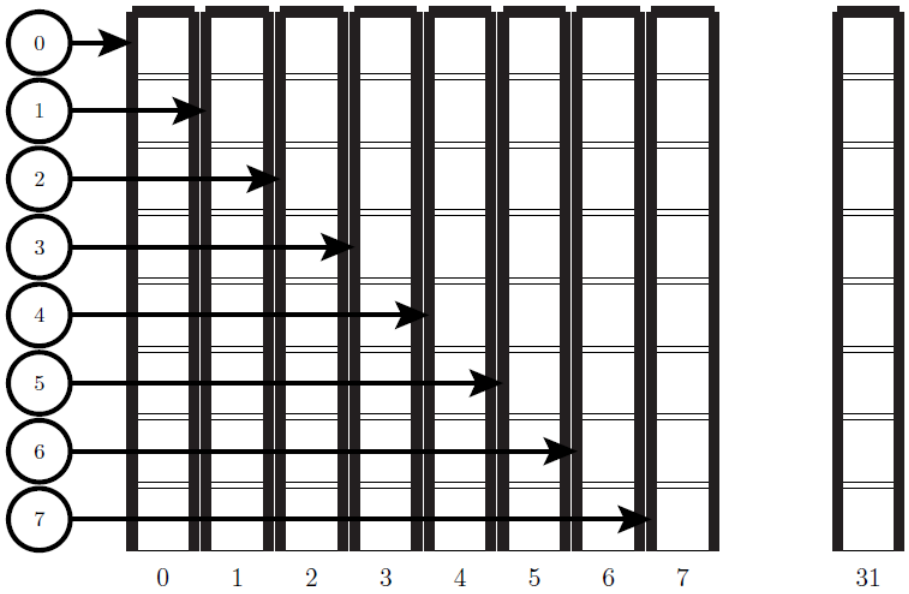
```
__local float sh[32][32];
```



Worst case: Threads of the same warp accessing the same column of a matrix having a width of a multiple of 32

- Column access

```
__local float sh[32][33];
```



Solution: ‘pad’ matrix with an extra column => no more bank conflicts

Performance Limiters

7. Other Performance Considerations

VRIJE UNIVERSITEIT BRUSSEL

Other performance considerations

- Unroll loops with a fixed number of iterations
  - Removes loop overhead
    - Index computations and tests
  - Increases ILP and MLP
  - Use `#pragma unroll`
- Vectorization
  - Use build-in vector types: float2, float4, int2, int4

VRIJE UNIVERSITEIT BRUSSEL

Other performance considerations

- Let one work item process multiple data items
  - Thread index calculation overhead is amortized
  - ILP and MLP will increase
  - Extra potential for loop unrolling
  - Increased data reuse (e.g. through private memory)

Example: Reduction (Parallel Sum)

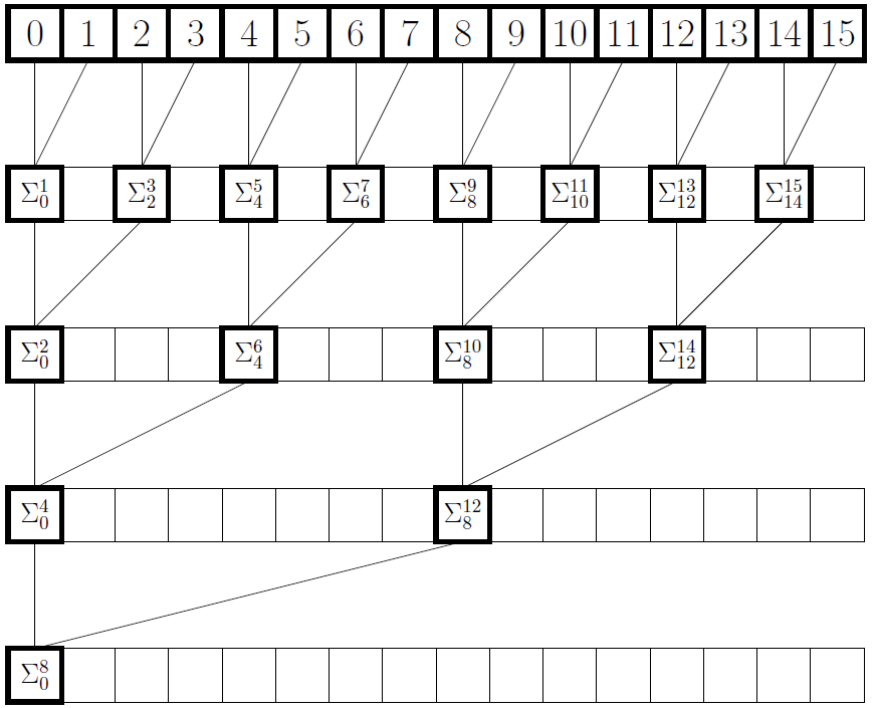
VRIJE UNIVERSITEIT BRUSSEL

Reduction

Parallel Sum:

- Add all elements of an array
- Binary tree algorithm
- Each work group computes 1 part, the total sum over the results of each work group is done on CPU

- 6 different versions



VRIJE UNIVERSITEIT BRUSSEL

Reduction 1

only global memory

```
3  // naive global memory
4  __kernel void reduction1(__global int *src,
5                           __global int *dst,
6                           unsigned int size)
7  {
8      if (get_global_id(0) >= size)
9          return;
10
11     for (int s = 1; s < get_local_size(0); s *= 2) {
12         if (get_local_id(0) % (2*s) == 0) {
13             src[get_global_id(0)] += src[get_global_id(0) + s];
14         }
15         barrier(CLK_GLOBAL_MEM_FENCE);
16     }
17
18     if (get_local_id(0) == 0) {
19         dst[get_group_id(0)] = src[get_global_id(0)];
20     }
21
22     return;
23 }
```

VRIJE UNIVERSITEIT BRUSSEL

Reduction 2

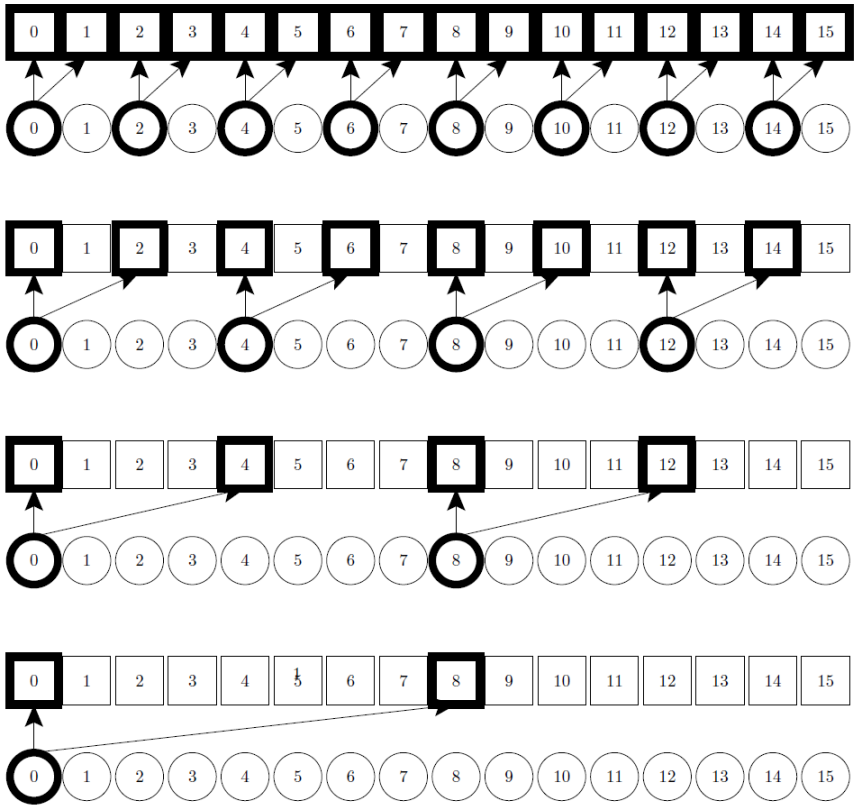
using local memory

```
25 // naive local memory
26 __kernel void reduction2(__global int *src,
27                           __global int *dst,
28                           unsigned int size)
29 {
30     __local int shared[WGSZ];
31     unsigned int gx = get_global_id(0);
32     unsigned int lx = get_local_id(0);
33
34     shared[get_local_id(0)] = gx < size ? src[get_global_id(0)] : 0;
35     barrier(CLK_LOCAL_MEM_FENCE);
36
37     for (int s = 1; s < get_local_size(0); s *= 2) {
38         if (get_local_id(0) % (2*s) == 0) {
39             shared[get_local_id(0)] += shared[get_local_id(0) + s];
40         }
41         barrier(CLK_LOCAL_MEM_FENCE);
42     }
43
44     if (get_local_id(0) == 0) {
45         dst[get_group_id(0)] = shared[0];
46     }
47 }
```

Reduction 3

Reduce idling threads

Each thread starts with 2 elements



But still thread divergence and bank conflicts!

VRIJE UNIVERSITEIT BRUSSEL

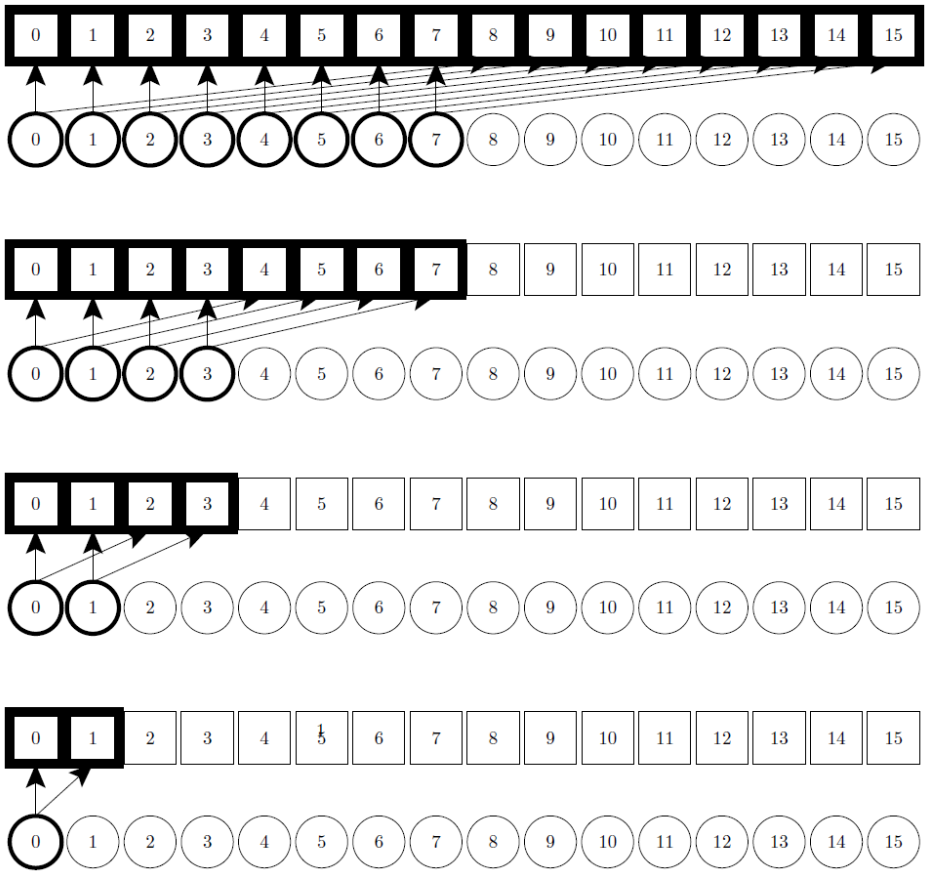
Reduction 3

Reduce idling threads

```
49 // local memory 1:2
50 __kernel void reduction3(__global int *src,
51                            __global int *dst,
52                            unsigned int size)
53 {
54     __local int shared[WGSZ];
55     unsigned int gx1 = get_global_id(0);
56     unsigned int gx2 = gx1 + get_global_size(0);
57     unsigned int lx = get_local_id(0);
58
59
60     shared[get_local_id(0)] = src[gx1];
61     if (gx2 < size) {
62         shared[get_local_id(0)] += src[gx2];
63     }
64
65     barrier(CLK_LOCAL_MEM_FENCE);
66
67     for (int s = 1; s < get_local_size(0); s *= 2) {
68         if (get_local_id(0) % (2*s) == 0) {
69             shared[get_local_id(0)] += shared[get_local_id(0) + s];
70         }
71         barrier(CLK_LOCAL_MEM_FENCE);
72     }
73     if (get_local_id(0) == 0) {
74         dst[get_group_id(0)] = shared[0];
75     }
76 }
```

Reduction 4

Thread reordering



If all threads of a warp are idling
=> the whole warp stops
=> no lost cycles

VRIJE UNIVERSITEIT BRUSSEL

Reduction 4

Thread reordering

```
78 // static thread reordered reduction
79 __kernel void reduction4(__global int *src,
80                           __global int *dst,
81                           unsigned int size)
82 {
83     __local int shared[WGSZ];
84     unsigned int gx1 = get_global_id(0);
85     unsigned int gx2 = gx1 + get_global_size(0);
86     unsigned int lx = get_local_id(0);
87
88     shared[get_local_id(0)] = src[gx1];
89     if (gx2 < size) {
90         shared[get_local_id(0)] += src[gx2];
91     }
92
93     barrier(CLK_LOCAL_MEM_FENCE);
94
95     for (int s = get_local_size(0)/2; s >= 1; s /= 2) {
96         if (get_local_id(0) < s) {
97             shared[get_local_id(0)] += shared[get_local_id(0) + s];
98         }
99         barrier(CLK_LOCAL_MEM_FENCE);
100    }
101
102    if (get_local_id(0) == 0) {
103        dst[get_group_id(0)] = shared[0];
104    }
105 }
```

VRIJE UNIVERSITEIT BRUSSEL

Reduction 5

Multiple elements per work item

```
107 // multiple elements per work item
108 // could have tried int4 values ...
109 __kernel void reduction5(__global int *src,
110                           __global int *dst,
111                           unsigned int size)
112 {
113     __local int shared[WGSZ];
114     unsigned int x0 = get_global_id(0);
115     unsigned int x1 = get_global_id(0) + get_global_size(0);
116     unsigned int x2 = get_global_id(0) + 2*get_global_size(0);
117     unsigned int x3 = get_global_id(0) + 3*get_global_size(0);
118
119     shared[get_local_id(0)] = src[x0] + src[x1] + src[x2];
120     if (x3 < size) {
121         shared[get_local_id(0)] += src[x3];
122     }
123     barrier(CLK_LOCAL_MEM_FENCE);
124
125     for (int s = get_local_size(0)/2; s >= 1; s /= 2) {
126         if (get_local_id(0) < s) {
127             shared[get_local_id(0)] += shared[get_local_id(0) + s];
128         }
129         barrier(CLK_LOCAL_MEM_FENCE);
130     }
131
132     if (get_local_id(0) == 0) {
133         dst[get_group_id(0)] = shared[0];
134     }
135 }
```

VRIJE UNIVERSITEIT BRUSSEL

Reduction 6

removing sync within last warp and loop unrolling

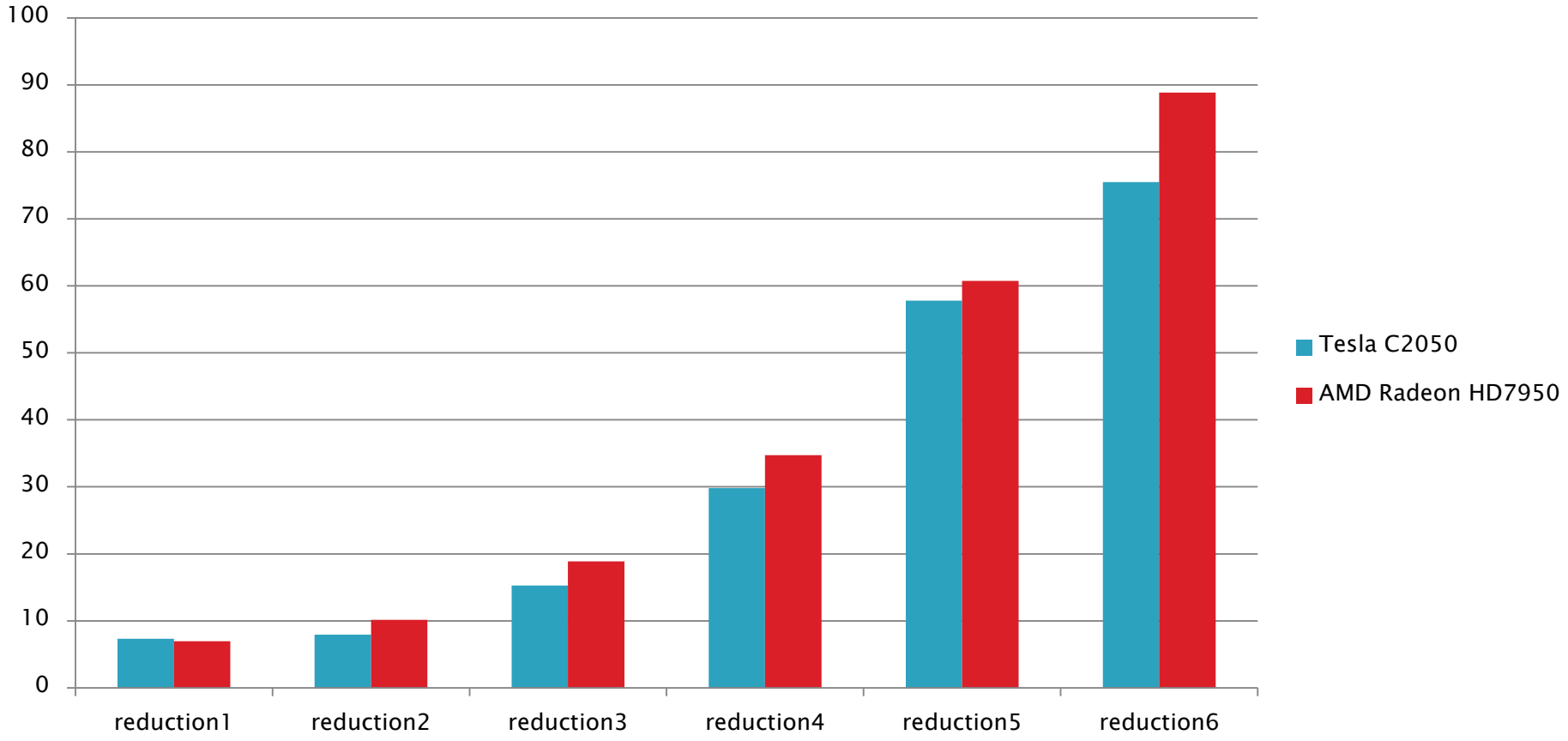
The last 64 elements can be handled by a single warp.

Synchronization is not necessary anymore, since all threads execute in lockstep

```
137 // add unrolling of last warp
138 void unrolled(volatile __local int *shared, int tid)
139 {
140     shared[tid] += shared[tid + 32];
141     shared[tid] += shared[tid + 16];
142     shared[tid] += shared[tid + 8];
143     shared[tid] += shared[tid + 4];
144     shared[tid] += shared[tid + 2];
145     shared[tid] += shared[tid + 1];
146
147 }
148
149 __kernel void reduction6(__global int *src,
150                          __global int *dst,
151                          unsigned int size)
152 {
153     __local int shared[WGSZ];
154     unsigned int x0 = get_global_id(0);
155     unsigned int x1 = get_global_id(0) + get_global_size(0);
156     unsigned int x2 = get_global_id(0) + 2*get_global_size(0);
157     unsigned int x3 = get_global_id(0) + 3*get_global_size(0);
158
159     shared[get_local_id(0)] = src[x0] + src[x1] + src[x2];
160     if (x3 < size) {
161         shared[get_local_id(0)] += src[x3];
162     }
163     barrier(CLK_LOCAL_MEM_FENCE);
164
165     for (int s = get_local_size(0)/2; s > 32; s /= 2) {
166         if (get_local_id(0) < s) {
167             shared[get_local_id(0)] += shared[get_local_id(0) + s
168         }
169         barrier(CLK_LOCAL_MEM_FENCE);
170     }
171
172     if (get_local_id(0) < 32) {
173         unrolled(shared, get_local_id(0));
174     }
175
176     if (get_local_id(0) == 0) {
177         dst[get_group_id(0)] = shared[0];
178     }
179 }
```

VRIJE UNIVERSITEIT BRUSSEL

Resulting Performance [GB/s]



Conclusions

VRIJE UNIVERSITEIT BRUSSEL

Overview

- Effect of the inefficiencies
  1. Occupancy ~ idling
  2. ILP ~ idling
  3. Branching ~ instruction inefficiency
  4. Synchronization ~ idling & synchronization instruction overhead
  5. Memory level ~ latencies
  6. Memory access pattern ~ concurrent memory access ~ latencies

VRIJE UNIVERSITEIT BRUSSEL

Programming for Performance
Minimizing the overall run time

- Minimize idle time
  - Maximize parallelism
  - Minimize dependencies
  - Minimize synchronization
- Minimize software and hardware overheads
  - Memory access
    - Data placement
    - Global memory access patterns
    - Local memory access patterns
  - Computation
    - Minimize excess computations
    - Minimize branching
- **Remembering data access is slow and computation fast**

VRIJE UNIVERSITEIT BRUSSEL

Tips for programming

Program step-by-step, gradually add instructions, verify subresults

1. Print
   - AMD and Intel devices support the use of printf.
   - Add to OpenCL code:

     include #pragma OPENCL EXTENSION cl_amd_printf
   - Print for just a few work items, e.g. if (global_id(0) < 5) …
2. Write subresults to output array
   - Add an additional array in which you store subresults which you can then print on the CPU

VRIJE UNIVERSITEIT BRUSSEL

Tips for optimization

- Make program variants
  - Start with naïve version, gradually add optimized versions
  - *Tip*: use same signature (parameters) for each kernel!
- Make compute-only and memory-only versions to identify main bottleneck
  - Compute-only: put memory access in a conditional as with the microbenchmarks (to trick the compiler)
  - Memory-only: outcomment calculations
  - Ideal memory access pattern: check the influence of the memory access pattern by creating a version with ideal, coalesced bank-conflict-free access