

# GPU Computing

## » Lesson 6: Performance Limiters

Gauthier Lafruit & Jan Lemeire

2022–2023

<http://parallel.vub.ac.be/education/gpu>

***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

# Estimate overhead

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

# Performance Limiters

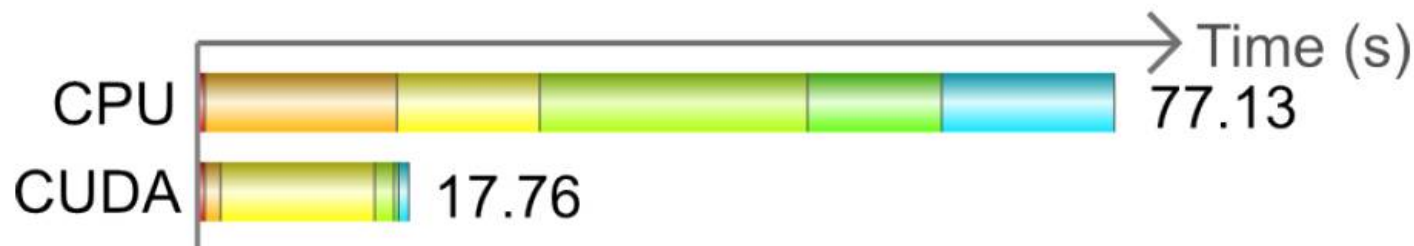
## 0. Limited Parallelism

*Level 1*

# Example: video decoding

## Decoding 1080p video sequence

Stage	CPU (s)	CUDA (s)	
1 MOTION_DECODE	0.64	0.64	
2 MOTION_RENDER	16.16	1.33	← 12 ×
3 RESIDUAL_DECODE	12.00	12.94	
4 WAVELET_TRANSFORM	22.52	1.63	← 14 ×
5 COMBINE	11.27	0.39	← 29 ×
6 UPSAMPLE	14.53	0.85	← 17 ×
Total	77.13	17.76	← 4.3 ×



# Amdahl's Law

- ◆ Limitations of inherent parallelism: a part  $s$  of the algorithm is not parallelizable



$$T_{seq} = (1-s).T_{seq} + s.T_{seq}$$



*parallelizable*



*not parallelizable*

$$T_{par} = \frac{(1-s).T_{seq}}{p} + s.T_{seq}$$



$$Speedup_{max} = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{\frac{(1-s).T_{seq}}{p} + s.T_{seq}} = \frac{p}{1 + (p-1).s}$$

*Assume  
no other  
overhead*

# Amdahl's Law



$$Speedup < \frac{p}{1 + (p-1).s}$$

$$Efficiency < \frac{1}{1 + (p-1).s}$$

If  $p$  is big enough:

$$Speedup < \frac{1}{s}$$

s	Speedup <sub>max</sub>
10%	10
25%	4
50%	2
75%	1.33

If the algorithm limits the amount of possible parallel execution, then the speedup is limited.



# Performance Limiters

## 1. Compute intensity

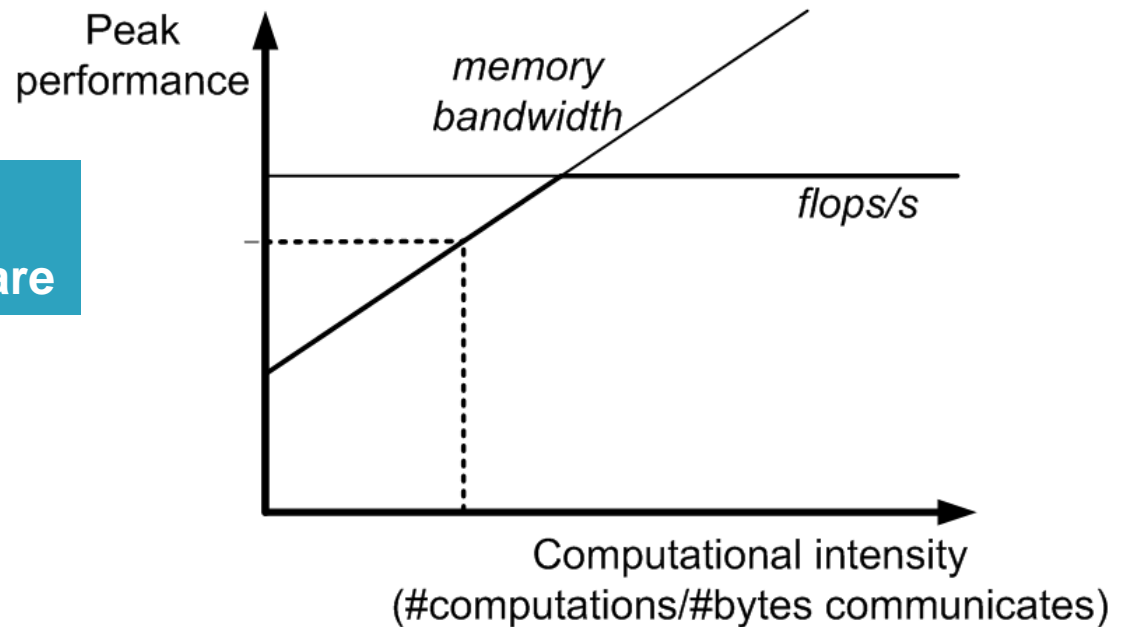
*Level 1*

*What is taking longer: memory transfer or the computations?*

**See lesson 1**

*Depends on Computational Intensity (CI)*

**Roofline model**  
Is determined by the hardware



Equation of Memory line:  $\text{peak performance} = \text{CI} * \text{BW}$

# Performance Limiters

## 2. Occupancy

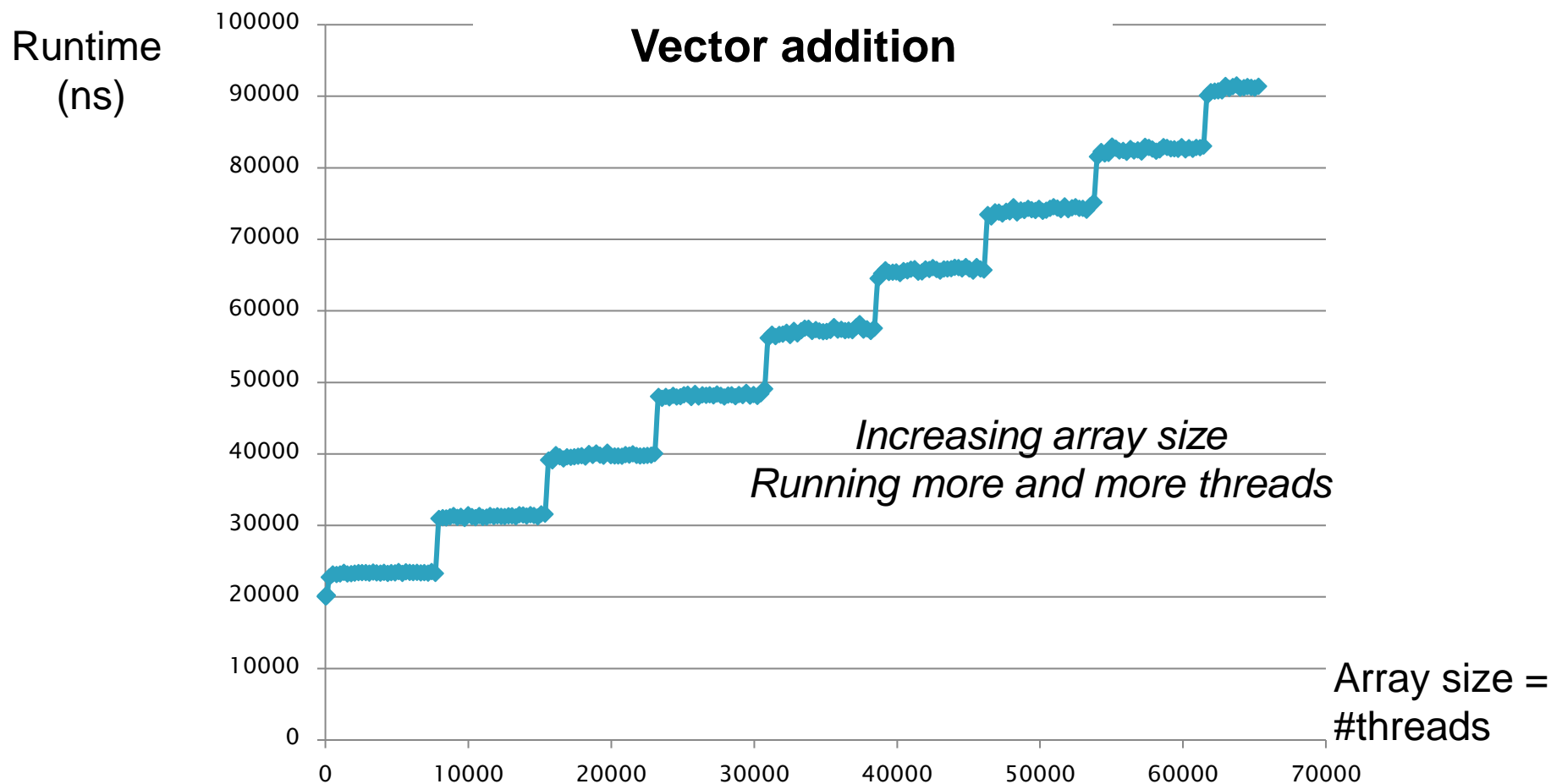
*Level 1*

# Keep all processing units busy

Enough parallelism (threads) is necessary

- ▶ For all cores ( = MultiProcessors)
- ▶ For all Scalar Processors (SPs)
  - 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**

# The effect of parallelism

- ✦ Processor needs sufficient threads 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 increase 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.

# Hiding of Memory Latencies

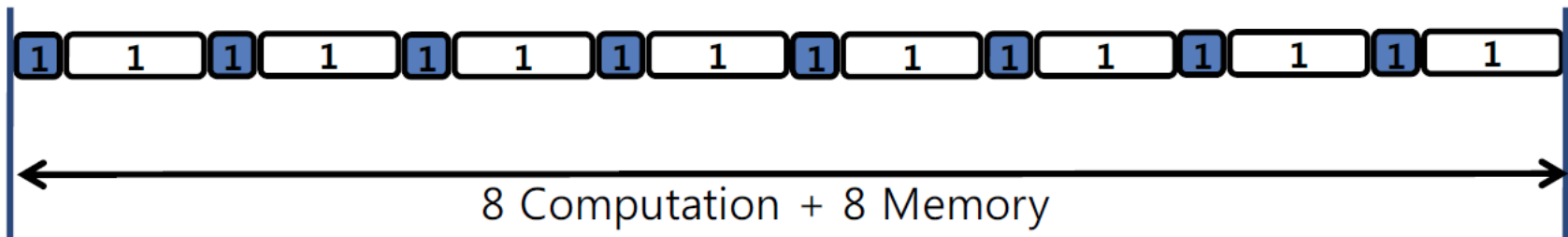


Memory period

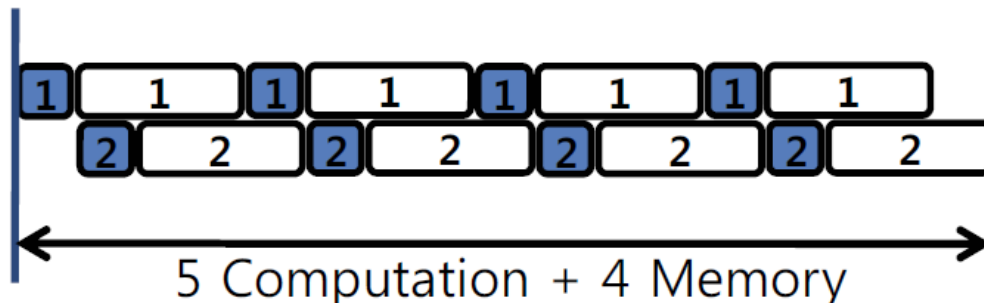


Computation period

- ▶ 1 warp, without latency hiding



- ▶ 2 warps running concurrently



- ▶ 4 warps running concurrently: full latency hiding

# Maximize Parallelism & Occupancy

- ▶ A great number of thread blocks:
  - A multiple of the number of cores times the occupancy in thread block count
  - If each core can run 4 thread blocks simultaneously, the number of thread blocks should be at least  $4 * \text{\#cores}$
  
- ▶ ***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

## 3. ILP & MLP

*Level 1*

# 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.

# Maximize parallelism on the multiprocessor

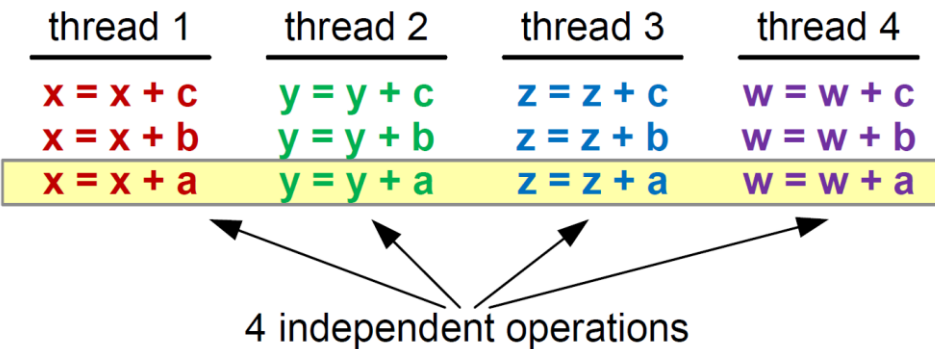
- ▶ Occupancy = *Thread-Level Parallelism* (TLP)
  - Scheduler has more choice to fill the pipeline
- ▶ *Instruction Level Parallelism* (ILP)
  - Independent instructions within one warp
  - Can be executed concurrently
- ▶ *Memory Level Parallelism* (MLP)
  - Independent memory requests for one warp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

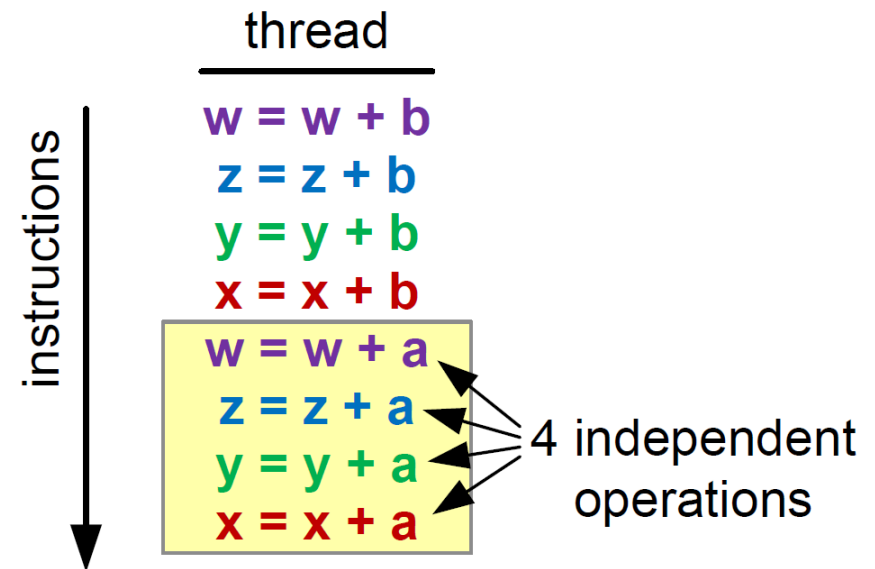
## Thread-Level Parallelism

- Independent threads



## Instruction-Level Parallelism

- Independent instructions

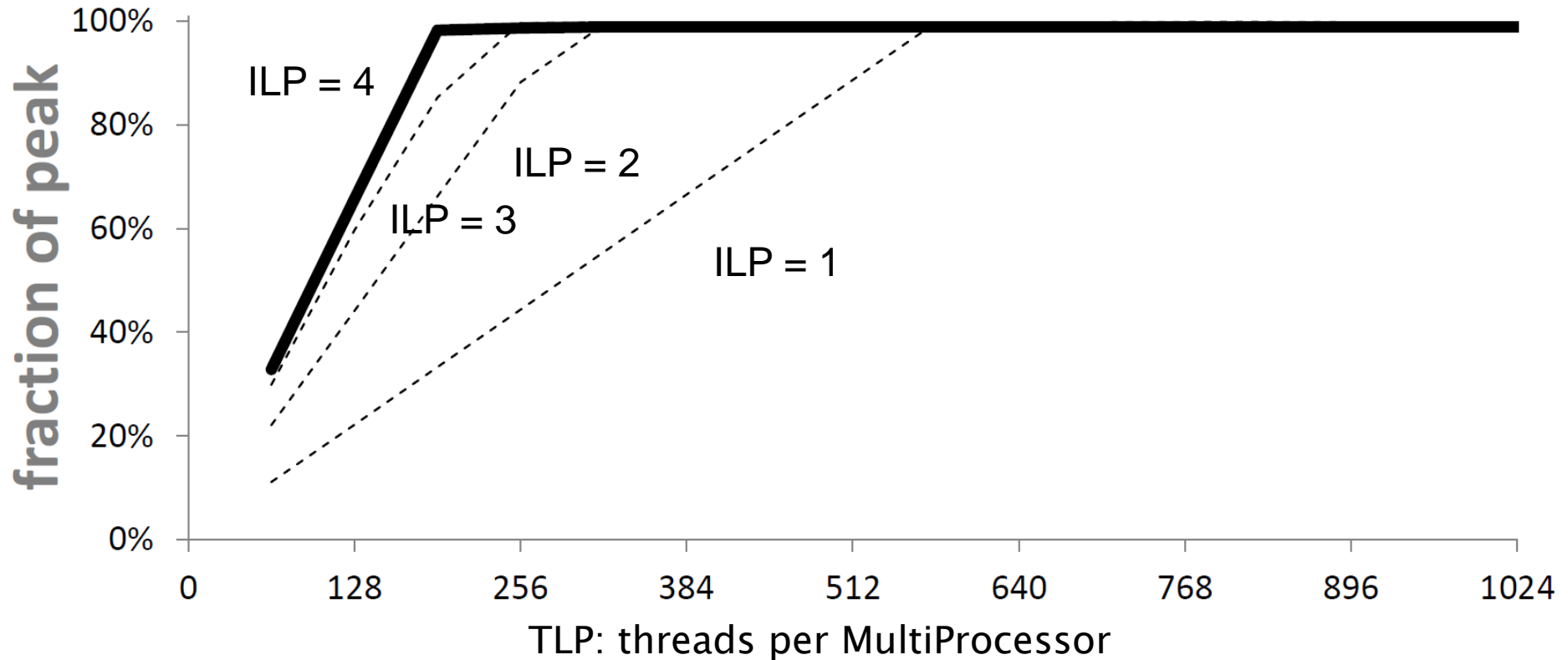


## Memory-Level Parallelism

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

# Computational Performance

## A function of TLP and ILP

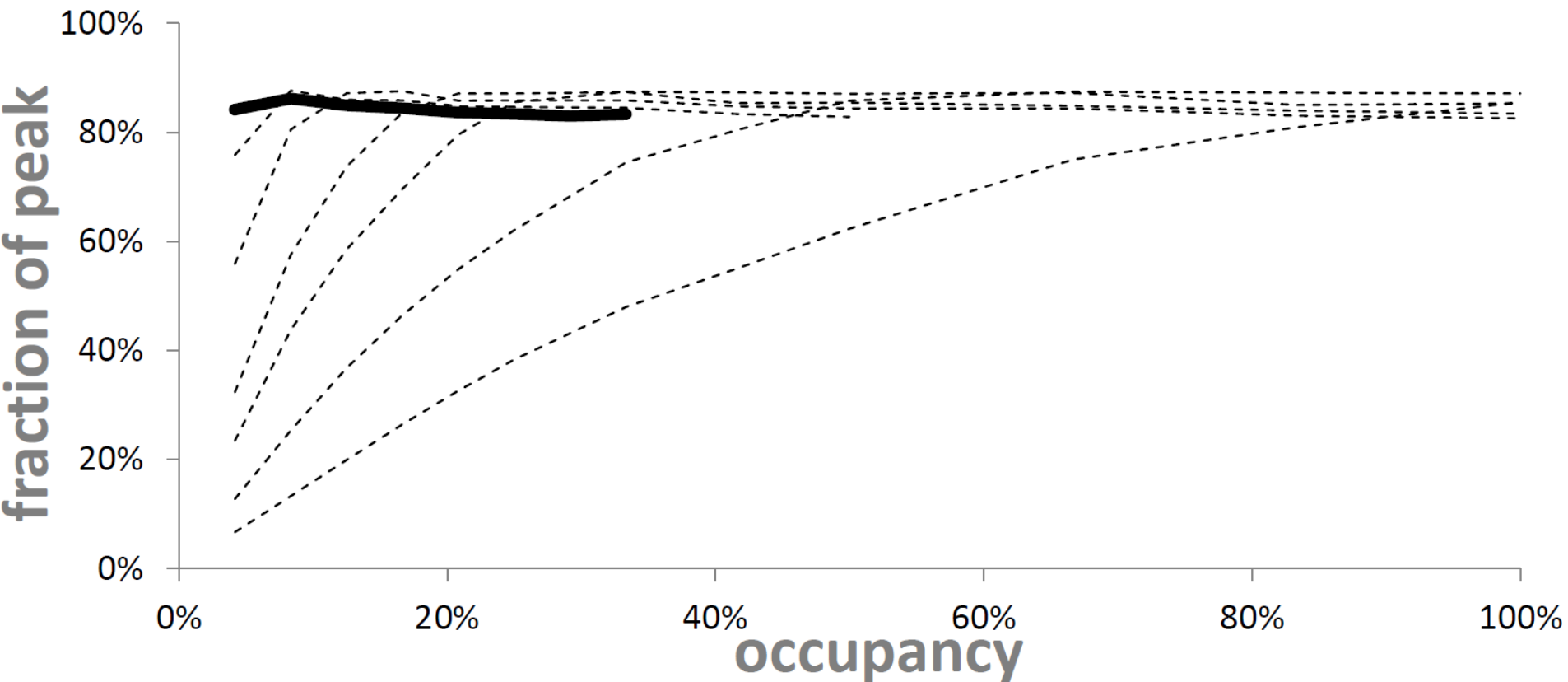


**Occupancy roofline**

# 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

## 4. Synchronization

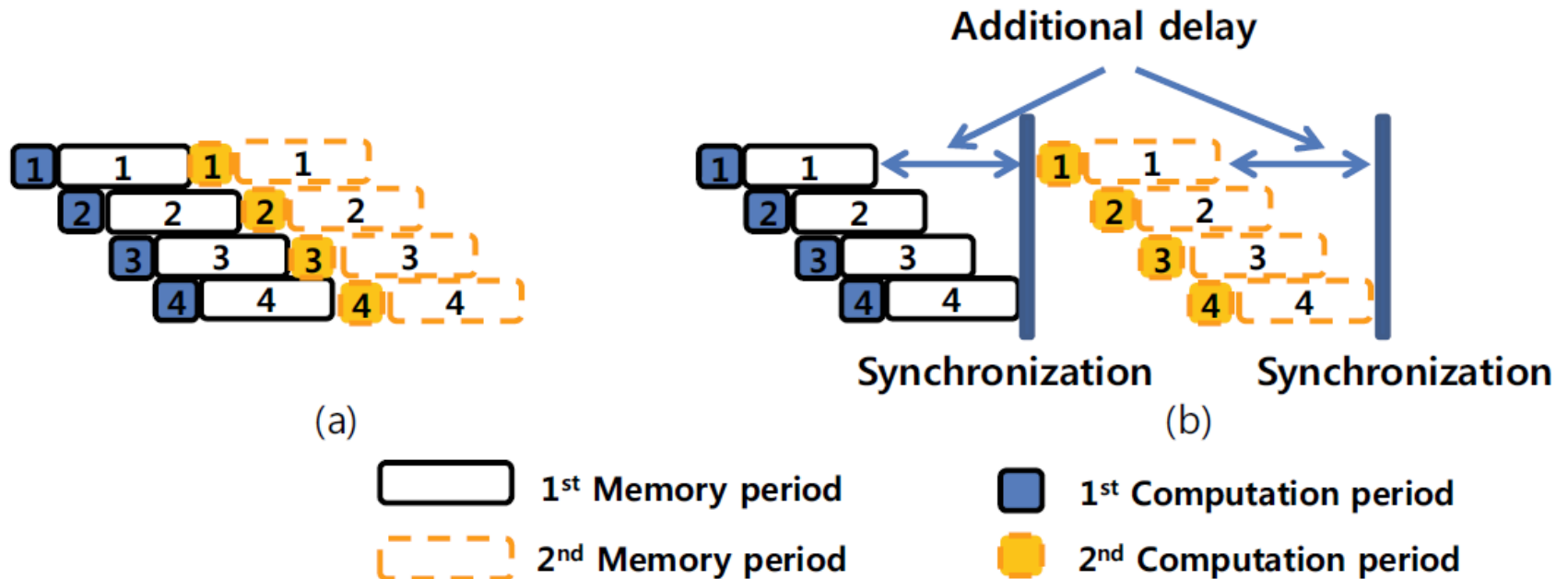
*Level 2*

# Local and global synchronization

- ▶ **Local synchronization**
  - threads of the same group can synchronize:  
    \_\_syncthreads();
  - threads 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 (thread blocks have all reached the same spot in the algorithm)
  - Overhead!



# Lost cycles due to local synchronization



No synchronization

Barrier after each  
memory period

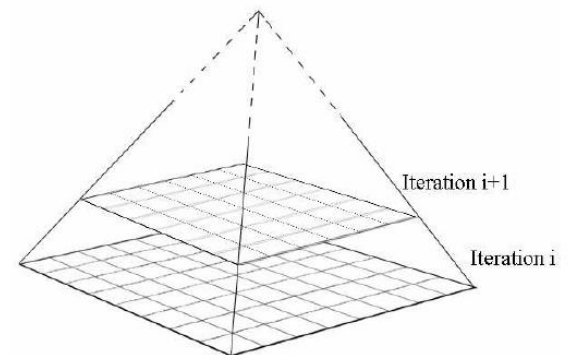
# Minimize synchronization overhead

- ▶ Local synchronization:
  - Keep thread blocks small → less effect
    - with multiple concurrent thread blocks 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)
  - $\text{Heat}_{\text{point}} = f(\text{heat}_{\text{neighbors}})$
  - Points are partitioned over the thread blocks, each thread block simulates  $N \times N$  points
  - Calculate for  $N \times N$  points and globally synchronize after each time step?
  - **No:** calculate different iterations independently with overlapping borders for each thread block
    - Iteration 0:  $(N+k) \times (N+k)$  points
    - ...
    - Iteration  $k-1$ :  $N \times N$  points

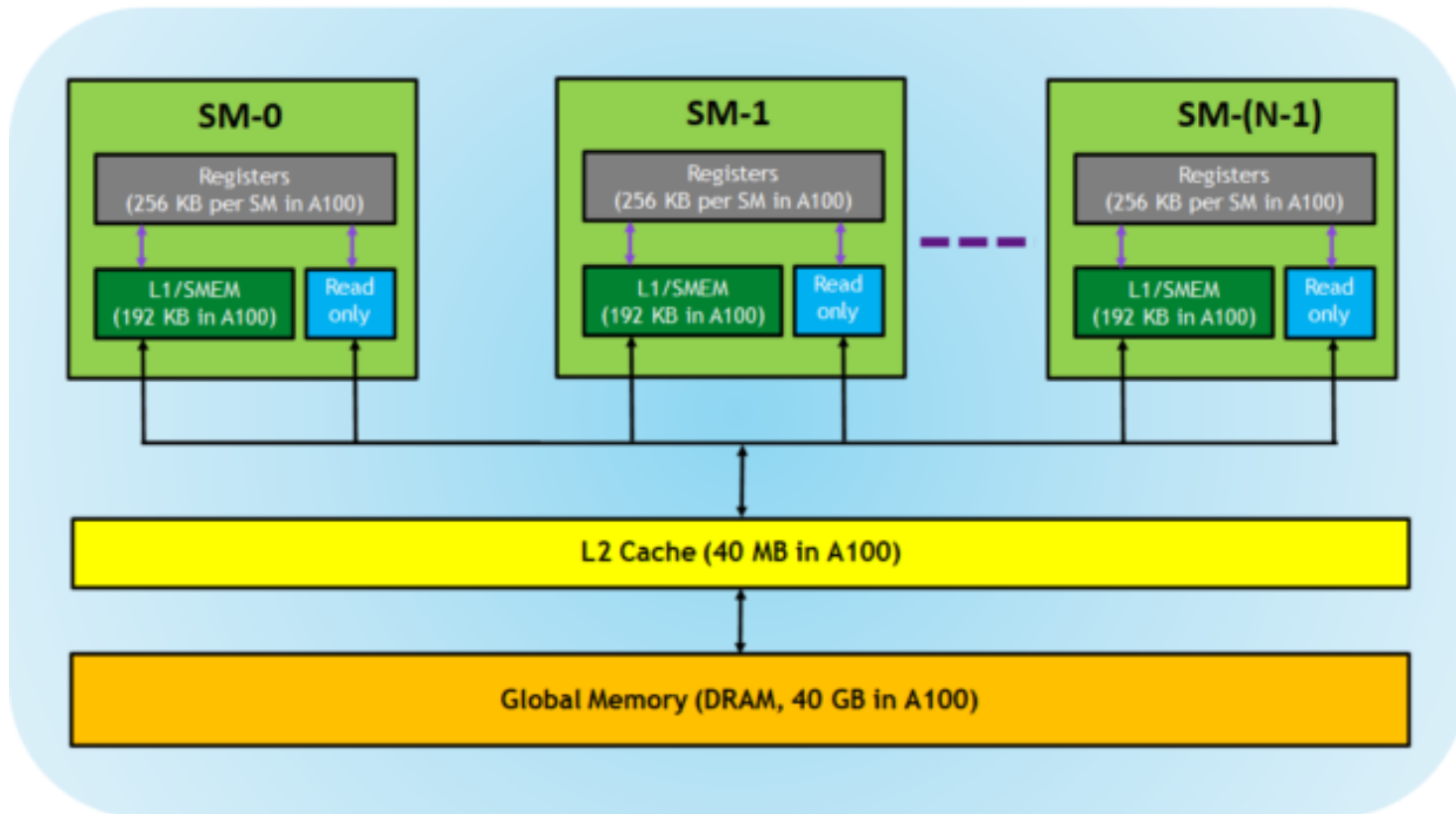


# Performance Limiters

## 5. Memory hierarchy

*Level 2*

# Architecture – Memory Model



# 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

# 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

# Memory Levels

- ▶ **Local memory**
  - Share data within a thread block
  - Use it if the same data is used by multiple threads in the same thread block
- ▶ **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. Branch divergence

*Level 3*

# 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

# Example: tree traversal

- ▶ Given: a (search) tree
- ▶ Each thread 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

# 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

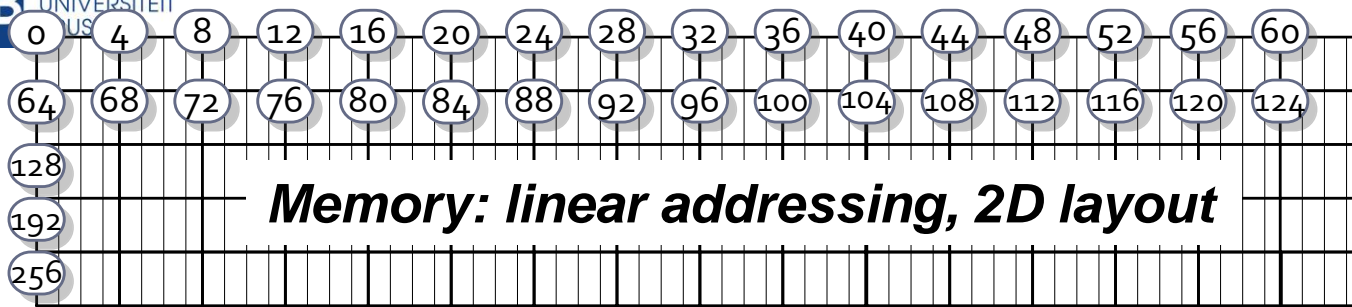
# Performance Limiters

## 7. Concurrent memory access

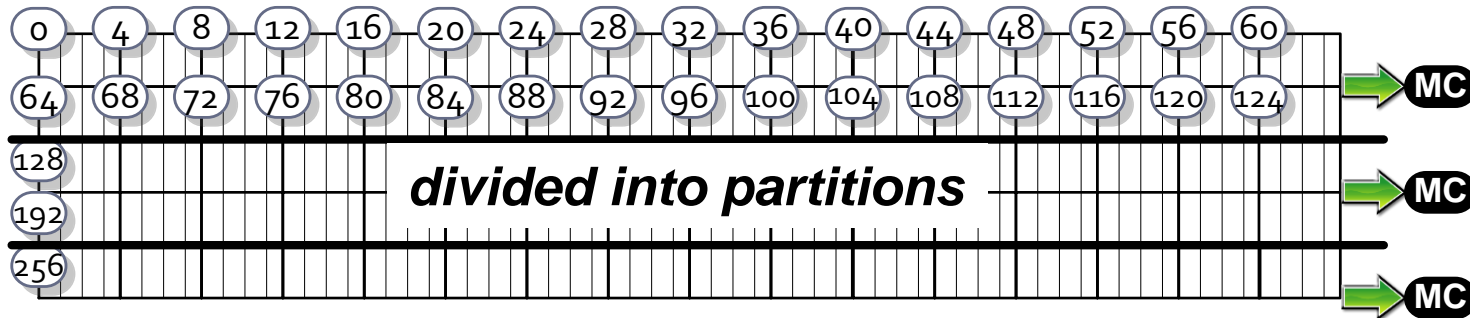
*Level 3*

# Concurrent Memory Access

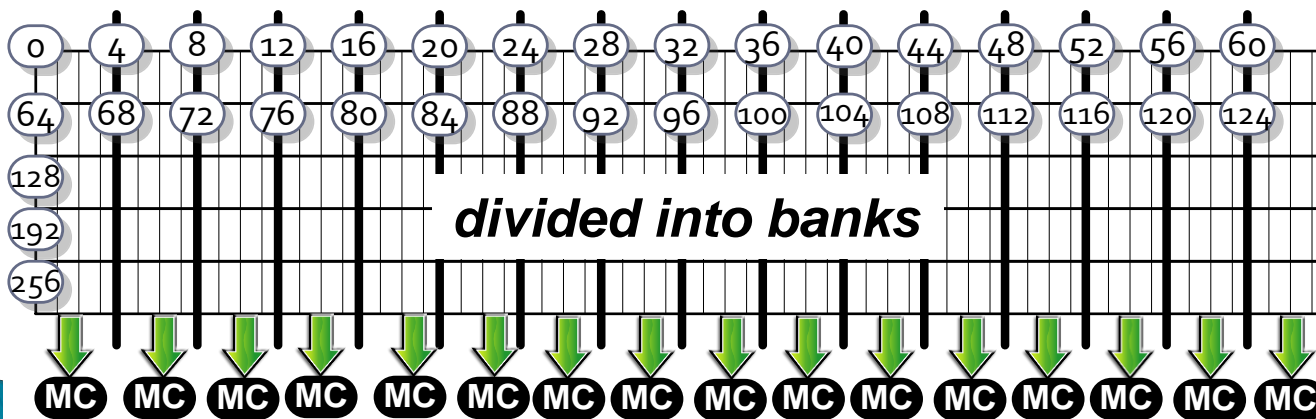
- ▶ Each Multiprocessor has active threads:
  - Simultaneous access of global memory
- ▶ Each hardware thread (warp) executes 32/64 kernel threads
  - Simultaneous access of global memory
  - Simultaneous access of shared memory
- ▶ But: concurrent memory access is limited by the hardware!
  - *Efficient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



...



...



**Memory  
Controllers:  
Can handle 1  
request at a  
time**

# Global memory

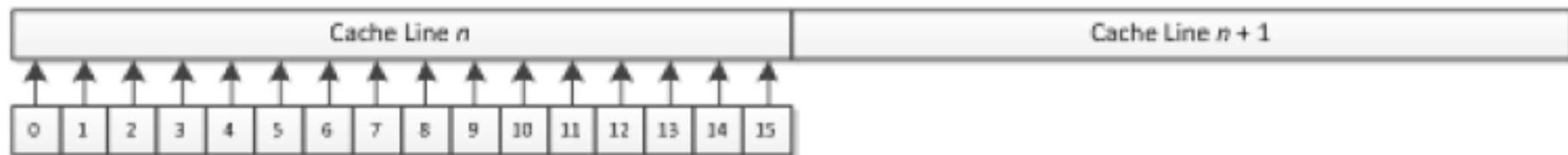
- ▶ Divided into partitions
  1. NVIDIA GPUs typically have 8 partitions
  2. Memory controller can serve 1 segment at a time ( $\approx$  cache line of 4x32 Bytes)
- ▶ 1: Active warps of different cores/multiprocessors simultaneously access global memory
  - **Partition camping** when they access the 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 ( $\approx$  cache line)
  - if not (uncoalesced access), memory requests are serialized => will take more time



# Global Memory Access

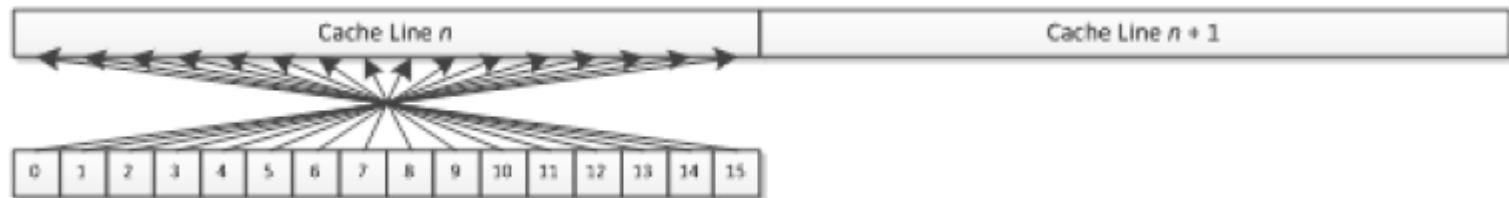
- ▶ Global memory is organized in segments (cache line), a memory controller can serve 1 segment at a time.
- ▶ Memory requests of warp are handled together
  - Data elements of the same segment are grouped and will be served together
- ▶ Ideal situation:
  - All bytes of necessary segments are needed
  - 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

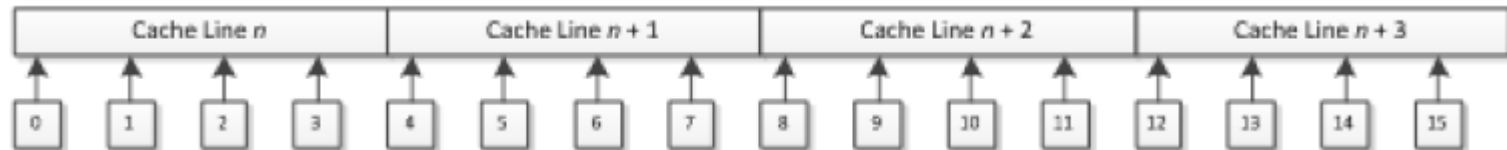


***Access is grouped per cache line  
Reads of cache lines are serialized  
=> Penalty if multiple cache lines  
are needed for 1 warp memory  
request***

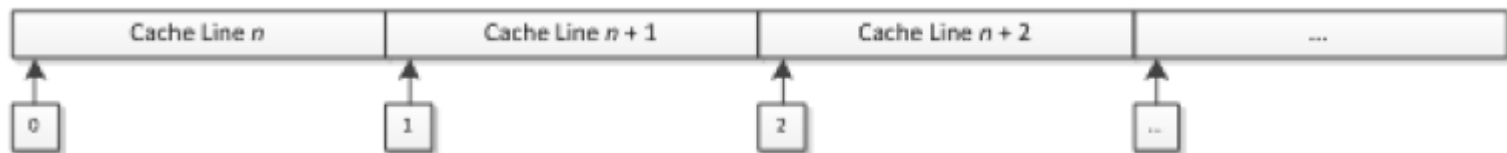
# Concurrent data access



***Stride of 4  $\Rightarrow$  1/4th of performance***



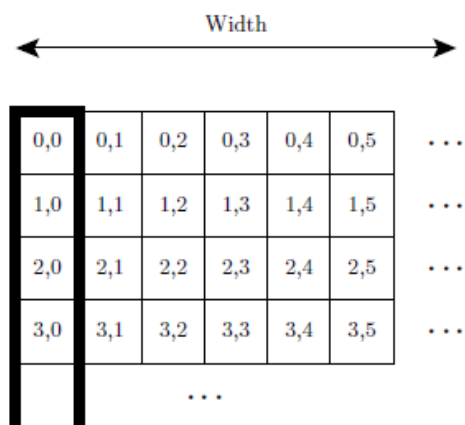
***Stride of 16  $\Rightarrow$  1/16th of performance***



# 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 of a matrix)



Quadro K620

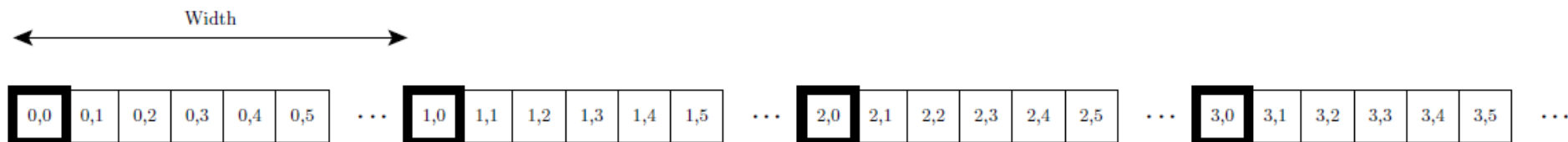
AMD HD 7950

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*

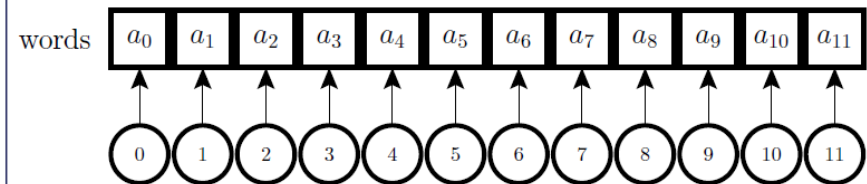
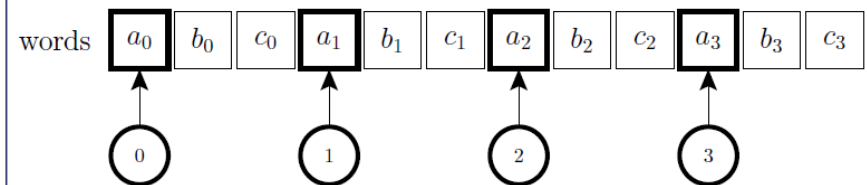
```
typedef struct {
    float a, b, c;
} triplet_t;

__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

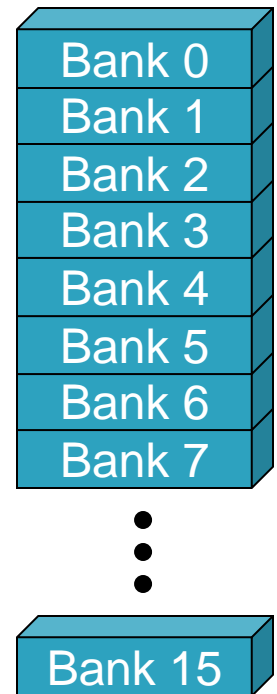
If elements are visited at different moments



SOA removes strides

# shared Memory access

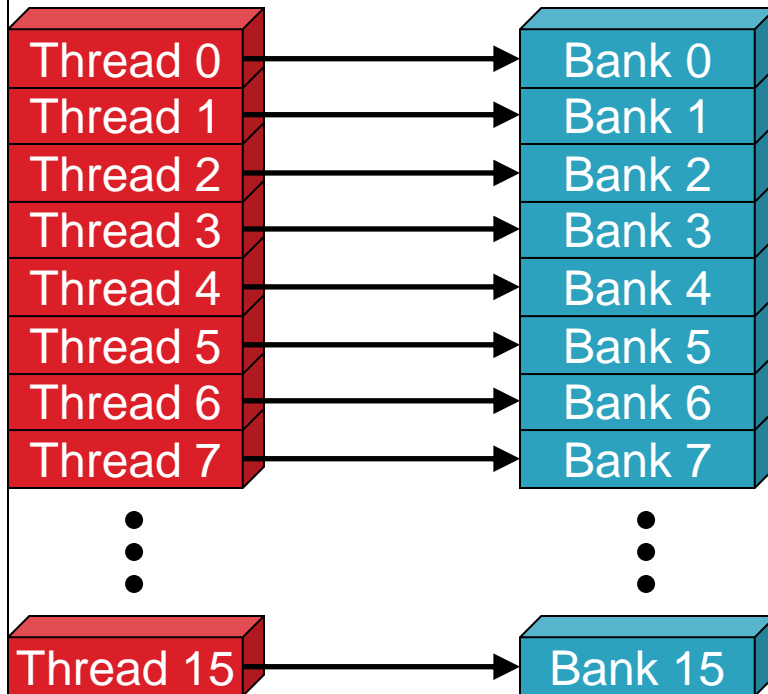
- ▶ shared memory is divided into **banks**
- ▶ Each bank can service one address per cycle
- ▶ Multiple simultaneous accesses to a bank result in a **bank conflict**
  - Conflicting accesses are serialized
  - Cost = max # simultaneous accesses to single bank
- ▶ No bank conflicts when
  - All threads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



# Bank Addressing Examples

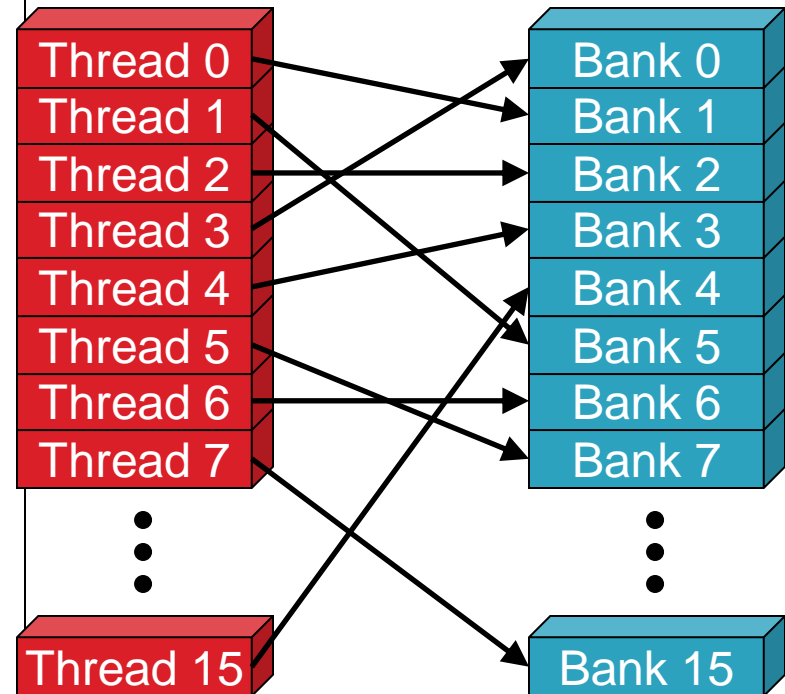
## ▶ No Bank Conflicts

- Linear addressing stride of 1



## ▶ No Bank Conflicts

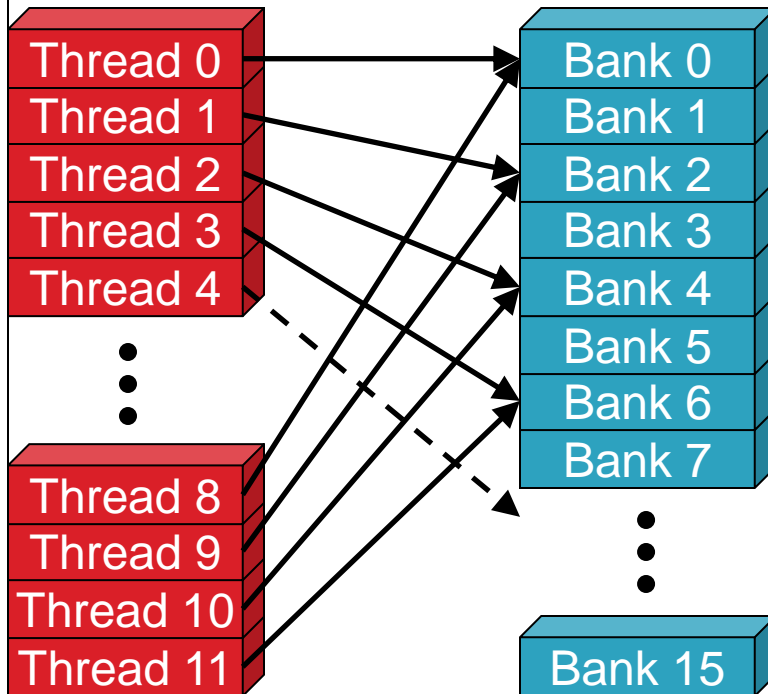
- Random 1:1 Permutation



# Bank Addressing Examples

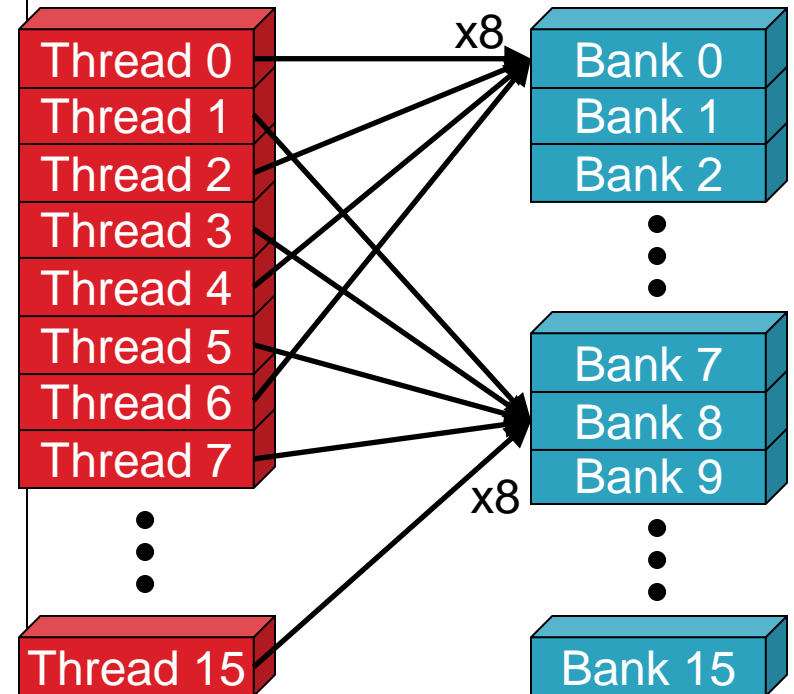
## 2-way Bank Conflicts

- Linear addressing stride of 2



## 8-way Bank Conflicts

- Linear addressing stride of 8

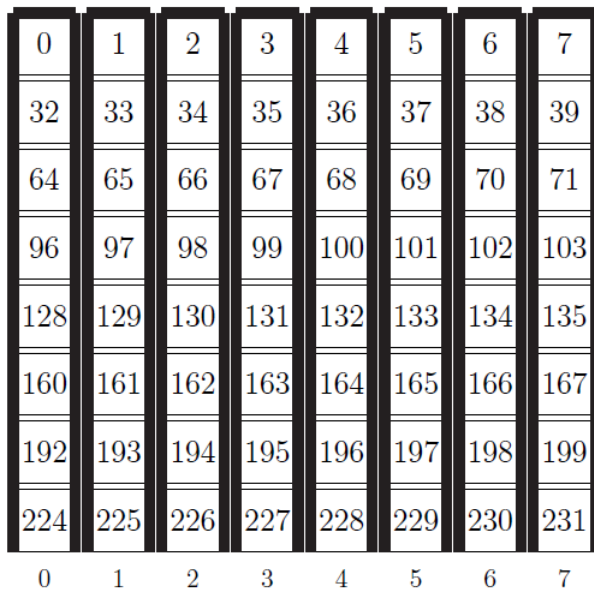




# shared Memory access

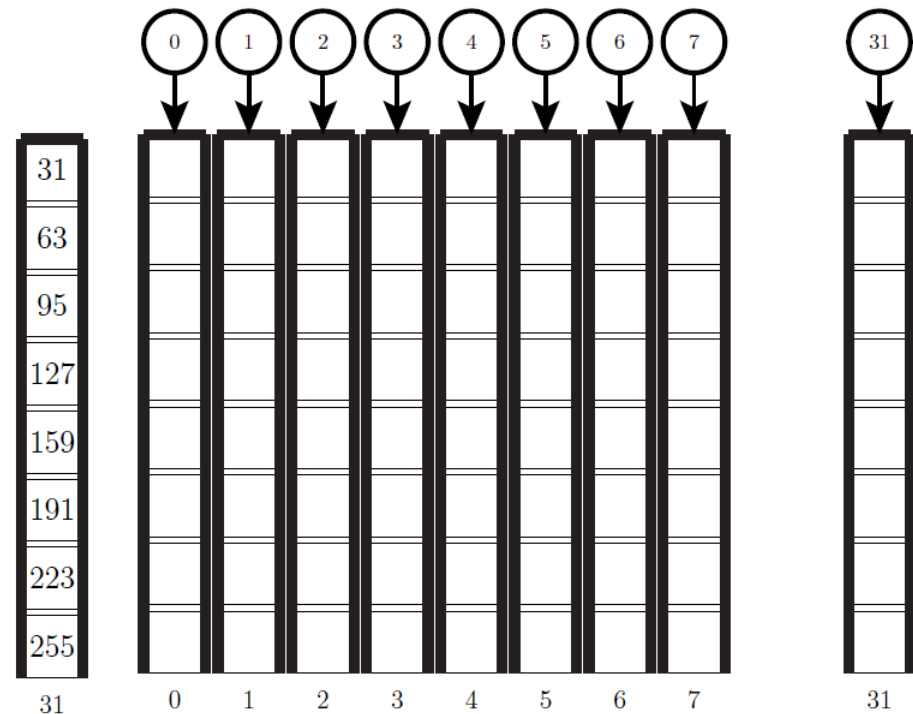
## ▶ Word storage order:

- Banks are 4 bytes wide



## ▶ Row access

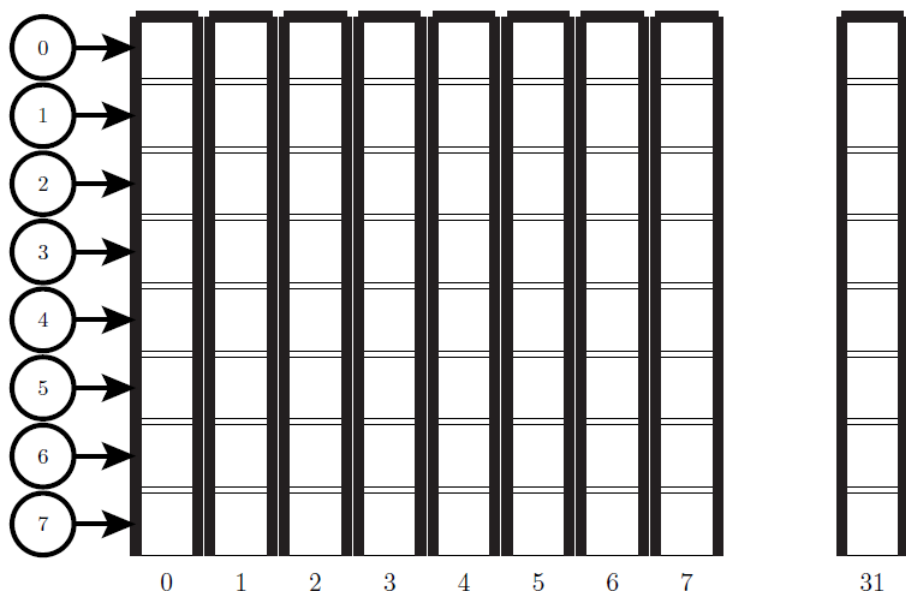
\_\_shared float sh[32][32];



# shared Memory access

## ▶ Column access

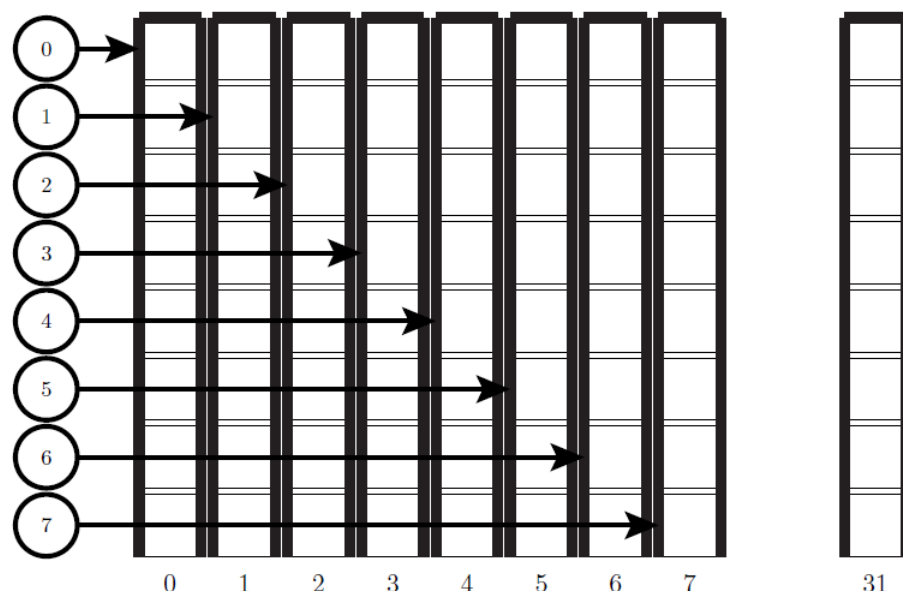
```
__shared float sh[32][32];
```



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

## ▶ Column access

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



Solution: 'pad' matrix with an extra column => *no more bank conflicts*

# Performance Limiters

## 8. Other Performance Considerations

## Other performance considerations

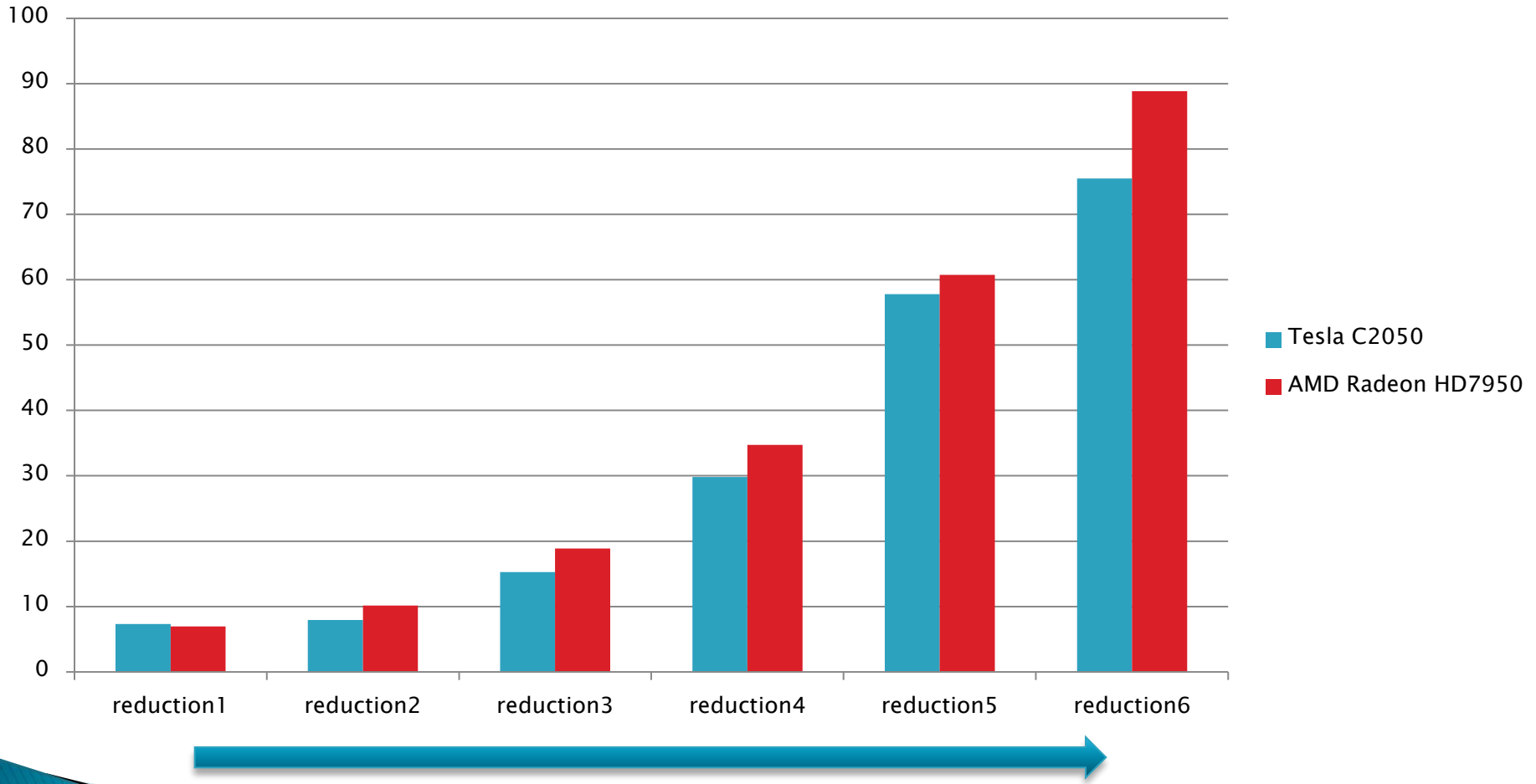
- ▶ Unroll loops with a fixed number of iterations
  - Removes loop overhead
    - Index computations and tests
  - Increases ILP and MLP
  - Use **#pragma unroll**
- ▶ Vectorization
  - Use build-in vector types: float2, float4, int2, int4
  - Especially for AMD GPUs

## Other performance considerations

- ▶ Let one thread 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)

# Resulting Performance [GB/s]



***Optimization***

See white paper  
Mark Harris (NVIDIA):  
*Optimizing Parallel  
Reduction in CUDA*



# Conclusions

# 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

# 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
    - shared memory access patterns
  - Computation
    - Minimize excess computations
    - Minimize branching
- ▶ **Remembering data access is slow and computation fast**

# Tips for programming

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

1. Debug
  - An individual kernel thread can be executed step-by-step
2. Print
  - Supported in CUDA?
3. Write subresults to output array
  - Add an additional array in which you store subresults which you can then print on the CPU

# 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