VRIJE UNIVERSITEIT BRUSSEL

GPU Computing

Lesson 3: Architecture & Strategy

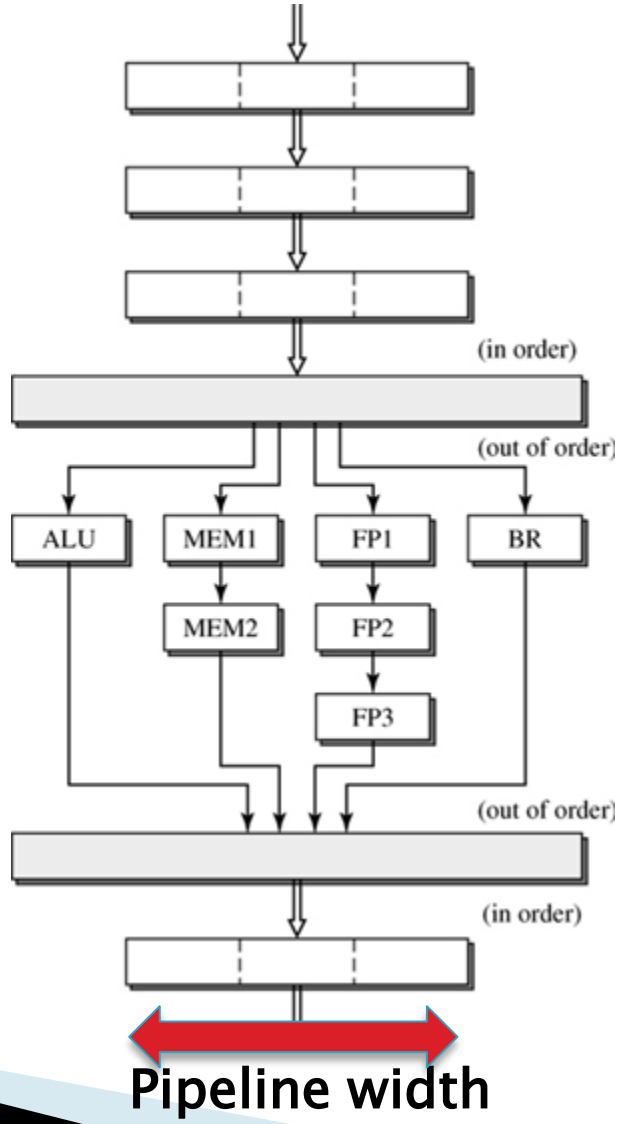
Jan Lemeire
2020-2021

http://parallel.vub.ac.be

The modern CPU

'Sequential' processor: super-scalar out-of-order pipeline

Pipeline depth



Different processing units

Out-of-order execution
Branch prediction
Register renaming
…

CPU computing

Write once
Run everywhere
efficiently!

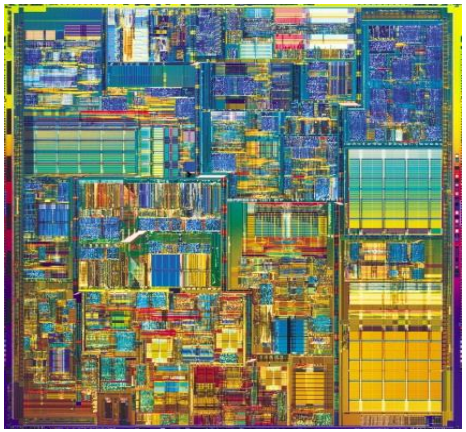
manual

automatic

Algorithm

Implementation

Compiler



Automatic optimization

Low latency of each instruction!

GPU strategy for massive computations

VRIJE UNIVERSITEIT BRUSSEL

GPU architecture strategy

- Light-weight threads, supported by the hardware
  - Thread processors, more than 1000 active threads per core
  - Switching between threads can happen in 1 cycle!
- No caching mechanism, branch prediction, …
  - GPU does not try to be efficient for every program, does not spend transistors on optimization
  - Simple straight-forward sequential programming should be abandoned…
- Less higher-level memory:
  - GPU: 16KB shared memory per SIMD multiprocessor
  - CPU: L2 cache contains several MB's
- Massively floating-point computation power
- RISC instruction set instead of CISC
- Transparent system organization

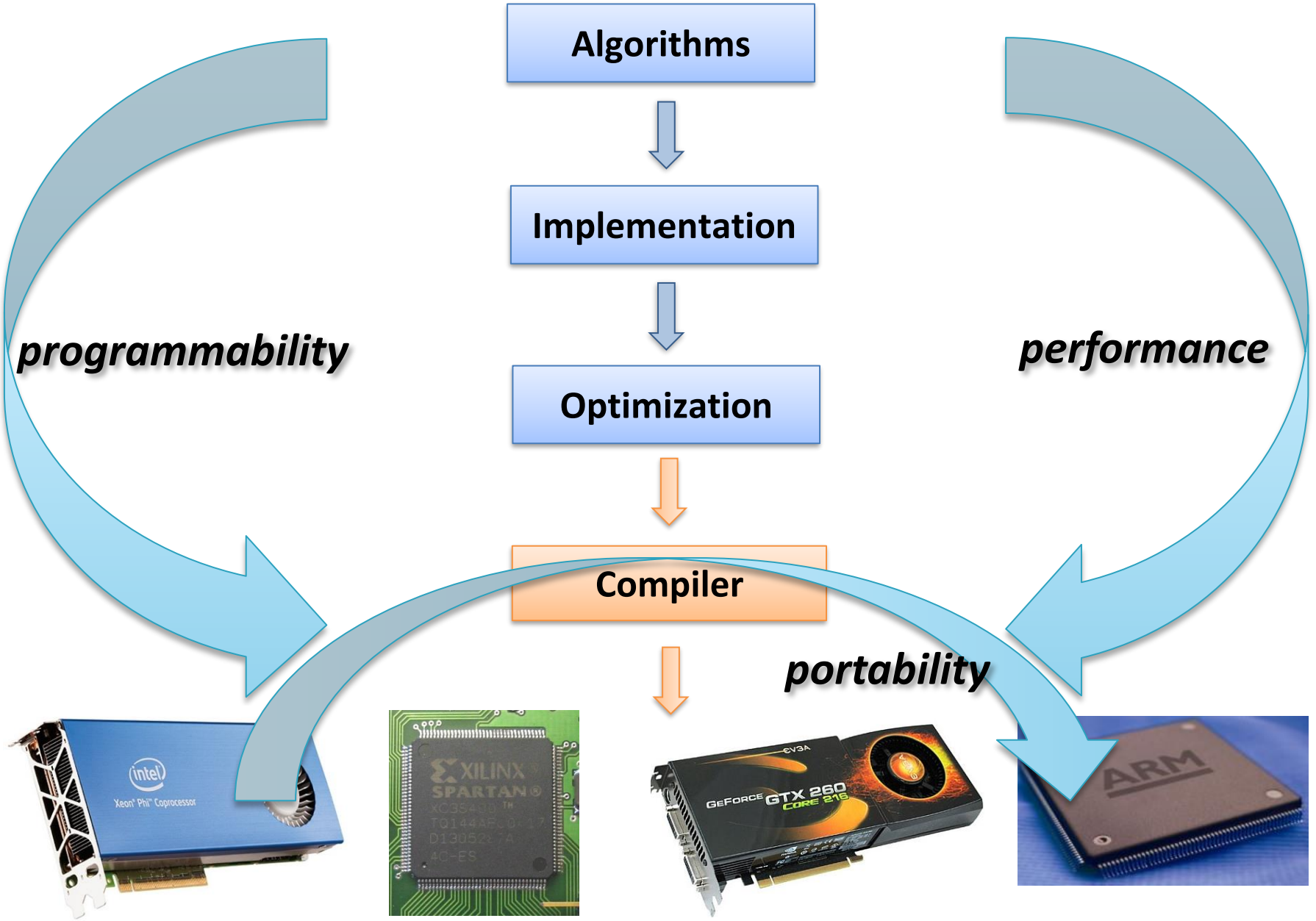
  ⇔ Modern (sequential) CPUs based on simple Von Neumann architecture

VRIJE UNIVERSITEIT BRUSSEL

GPU processor pipeline

- 6-24 stages
- in-order execution!!
- no branch prediction!!
- no forwarding!!
- no register renaming!!
- Memory system:
  - relatively small
  - Until recently no caching
  - On the other hand: much more registers (see later)
- No program call stack and no memory stack!
  - All functions inlined
  - No recursion possible

Challenges of GPU computing



VRIJE UNIVERSITEIT BRUSSEL

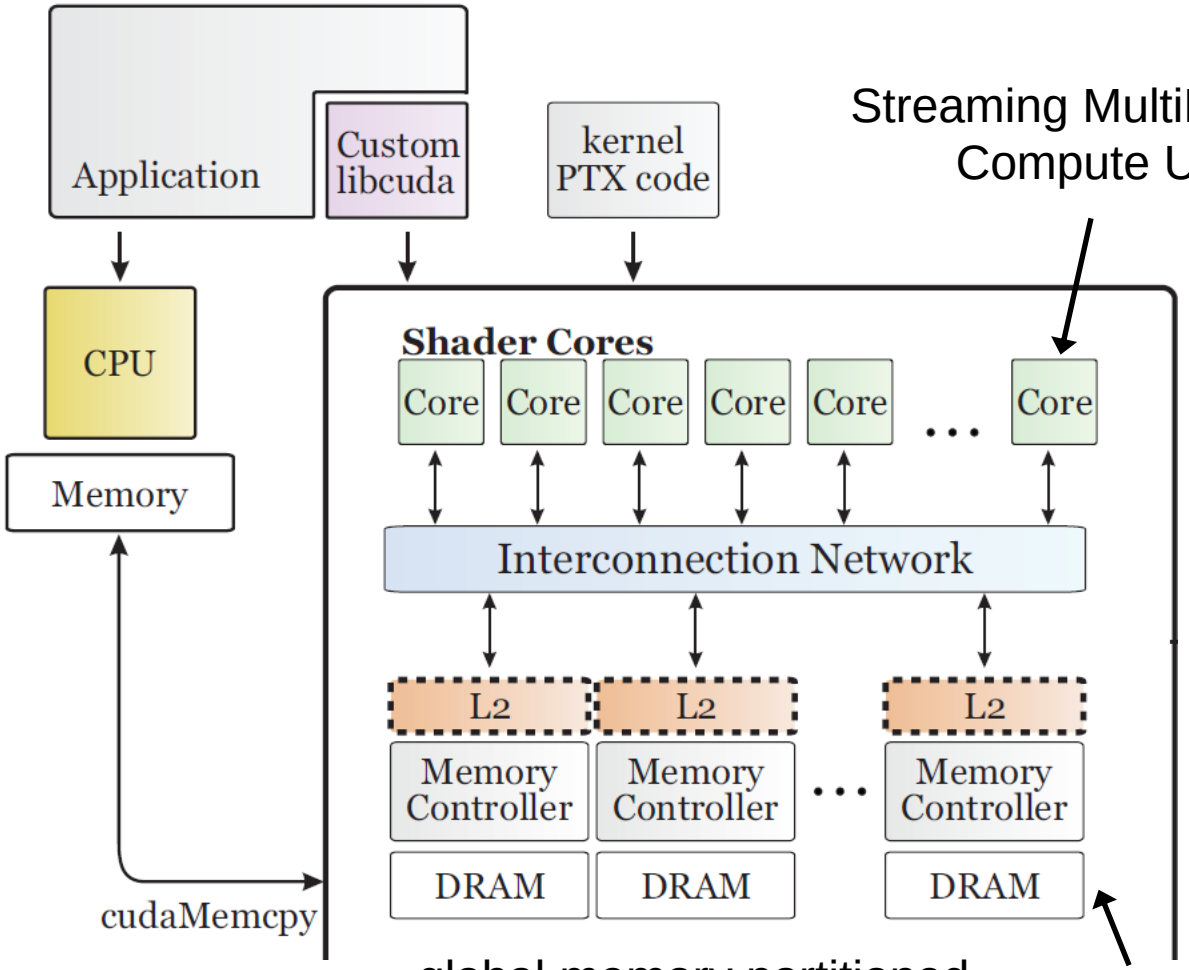
Fill the pipelines

GPUs has several pipelines which will be filled with instructions from different kernel threads through:

1. Running workgroups on the different compute units
2. Simultaneous multithreading: several threads active at the same time
   - *Discussed next*
3. Single Instruction Multiple Threads (SIMT)
   - *Discussed at the end of chapter*

Architecture

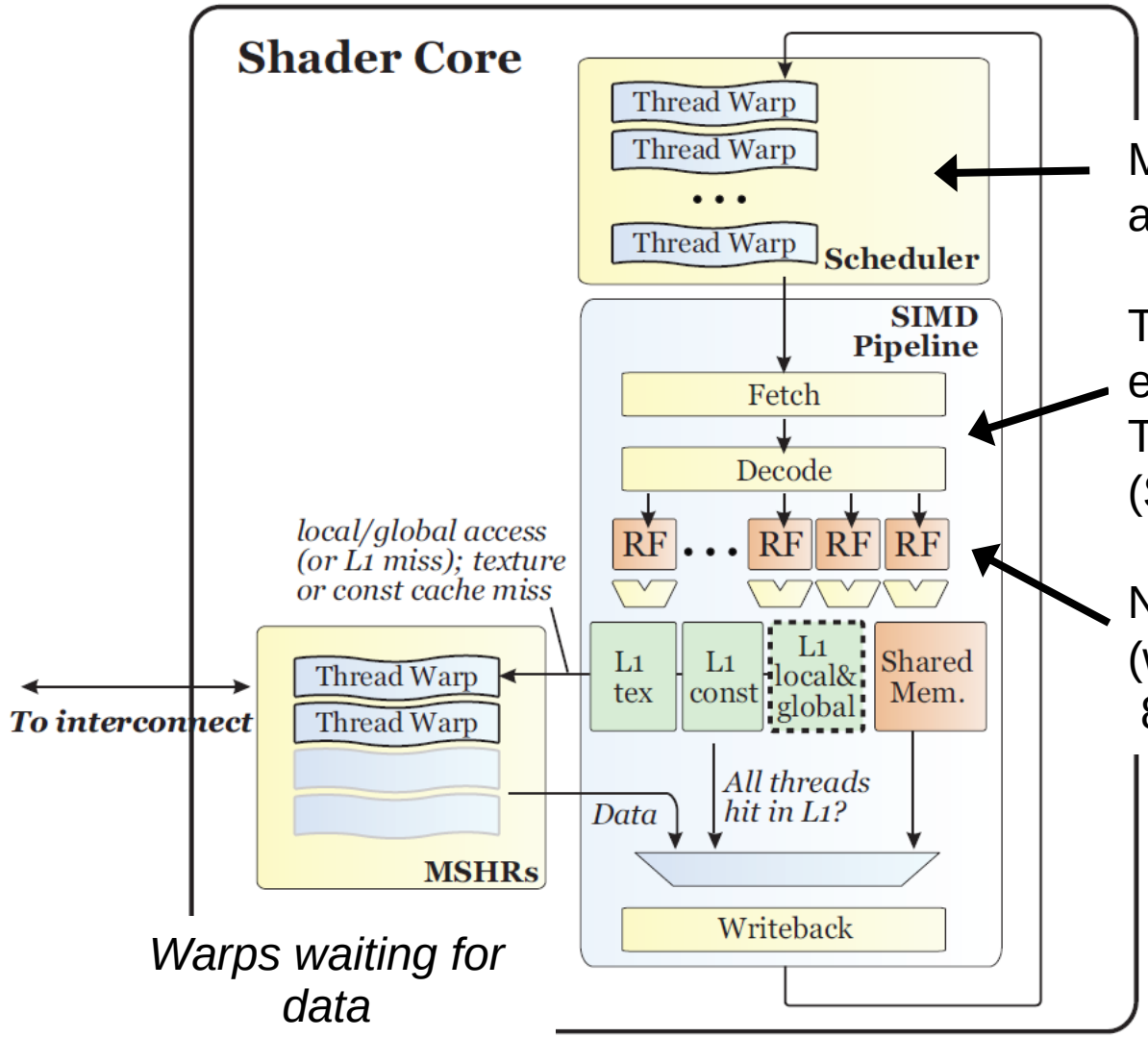
GPU Architecture



Streaming MultiProcessor (Nvidia)
Compute Unit (OpenCL)

global memory partitioned
Every controller can serve 1 request

1 Streaming Multiprocessor = a pipeline

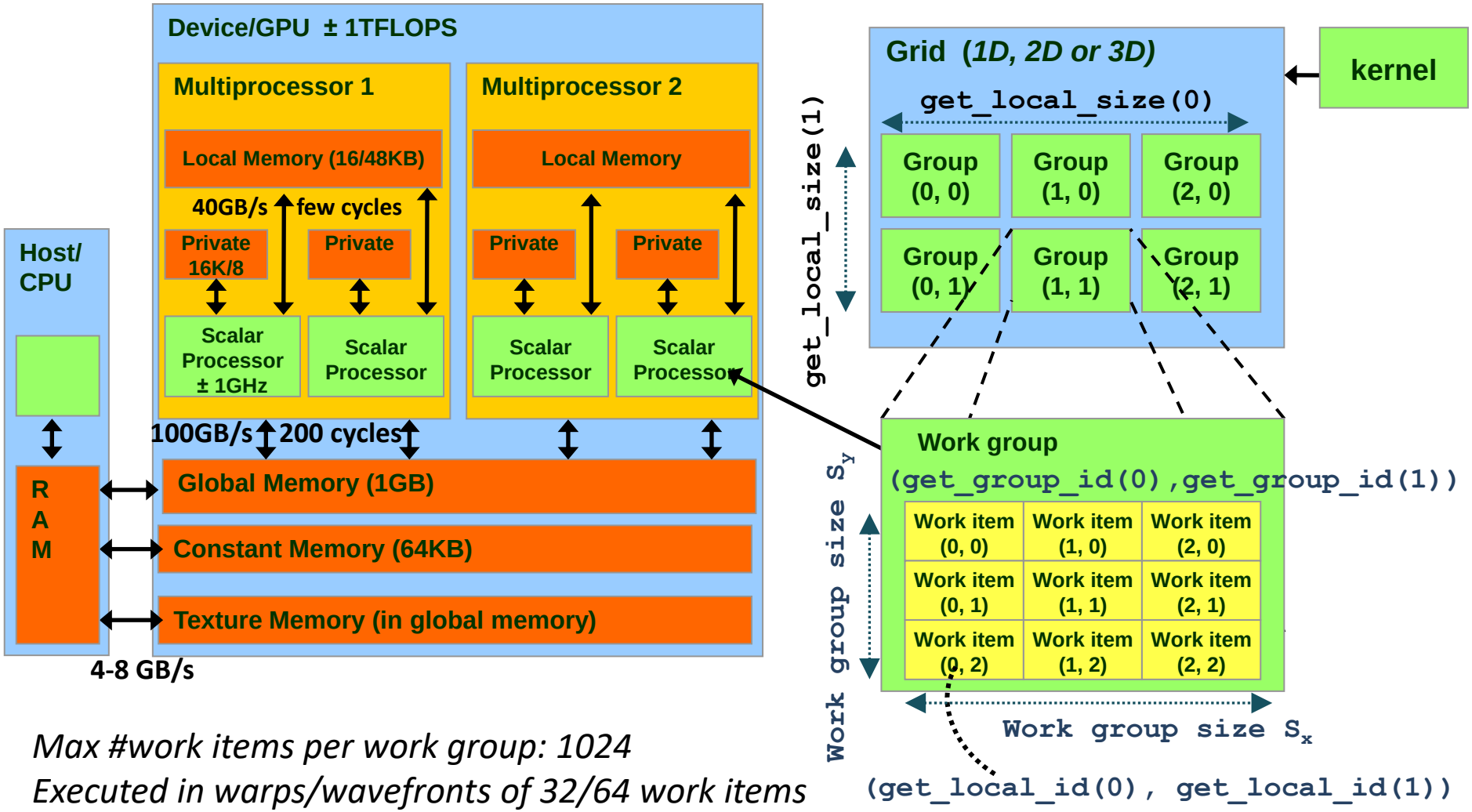


Multiple warps (hardware threads) are simultaneously active

The Same Instruction is executed on Multiple work items/ Threads
(SIMT)

Number of processing elements (width of pipeline):
8 – 32 – 192 - 128

Major GPU Concepts



Max #work items per work group: 1024
Executed in warps/wavefronts of 32/64 work items
Max work groups simultaneously on MP: 8
Max active warps/wavefronts on MP: 24/48

OpenCL terminology

VRIJE UNIVERSITEIT BRUSSEL

Properties of different architectures

N(Π) = #compute units
|ω| = warp size
Group & Warp slots: maximum # work groups or warps

	Fermi	Kepler	Maxwell	Pascal	Tonga
$N(\Pi)$	14	4	3	10	28
f_{clock} (MHz)	1147	1032	1058	1506	1010
issue limit	1	4	4	4	1
$\|\omega\|$	32	32	32	32	64
Resources					
Group slots	8	16	32	32	16
Warp slots	48	64	64	64	40
Local memory (KB)	48	48	64	96	64
Registers (KB)	128	256	256	256	

	Fermi	Kepler	Maxwell	Pascal	Tonga
$max(\|\gamma\|)$	1024	1024	1024	1024	256
max(local memory) (KB)	48	48	48	48	32
ALU count	32	192	128	128	64
SFU count	8	32	32	32	-
RAM Bandwidth (GB/s)	144	86.4	29	192	176
L2 Cache size (KB)	768	256	2048	1536	512
L2 Cache line size (B)	128	128	128	128	64
L1 Cache size (KB)	16	16	64	48	16
max(global memory) (MB)	1024	672	512	1536	2880
RAM Size (MB)	2688	2048	2048	6144	4096

#LD/STO units = 16 32 32 32

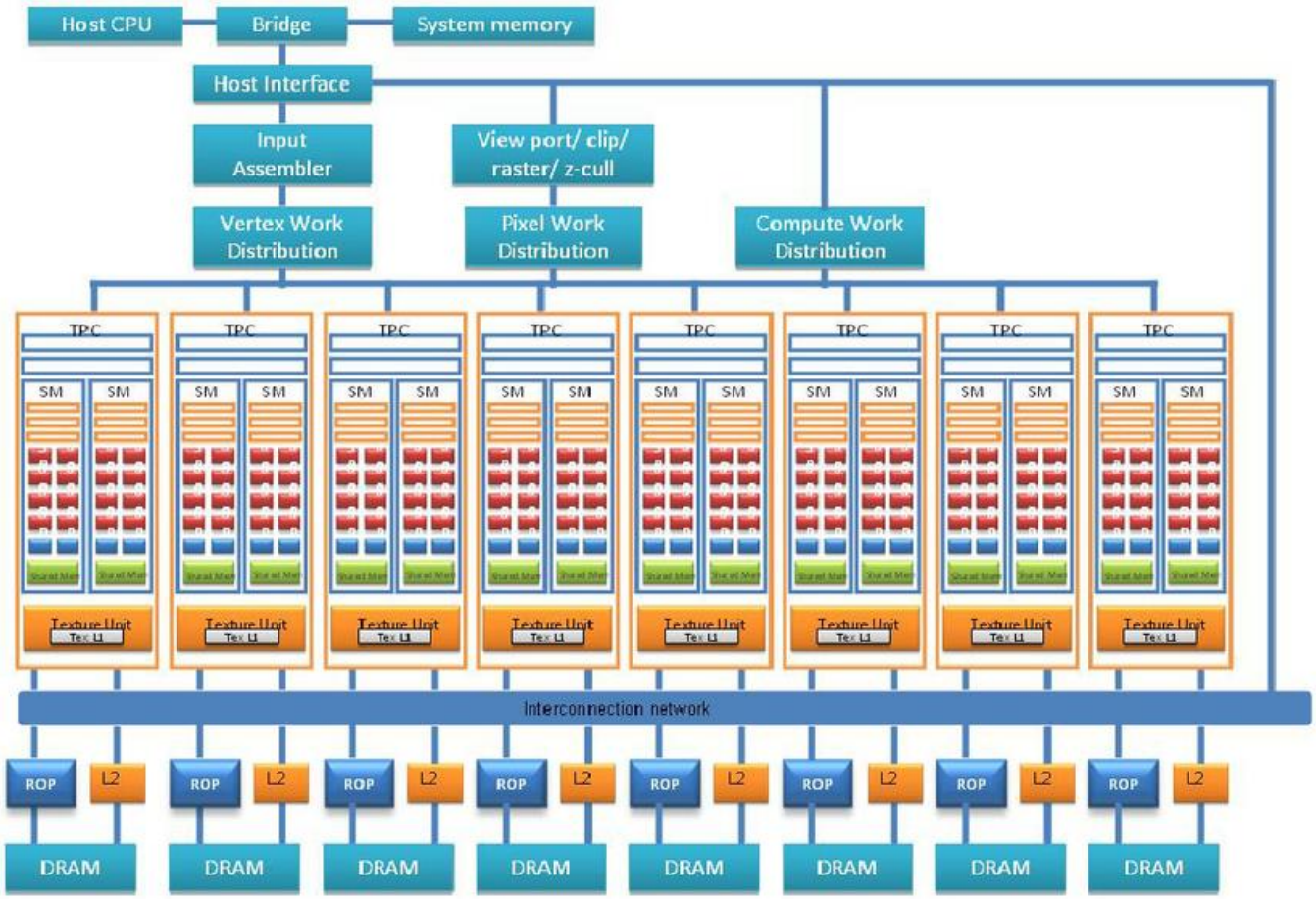
GPUs of our lab and architecture

Full name	Abbreviated name
NIVDIA Tesla C2050	Fermi
NIVDIA GeForce GTX 650 Ti	Kepler
NIVDIA Quadro K620	Maxwell
NIVDIA GeForce GTX 1060 6GB	Pascal
AMD Radeon R9 380	Tonga

VRIJE UNIVERSITEIT BRUSSEL

The different Nvidia architectures

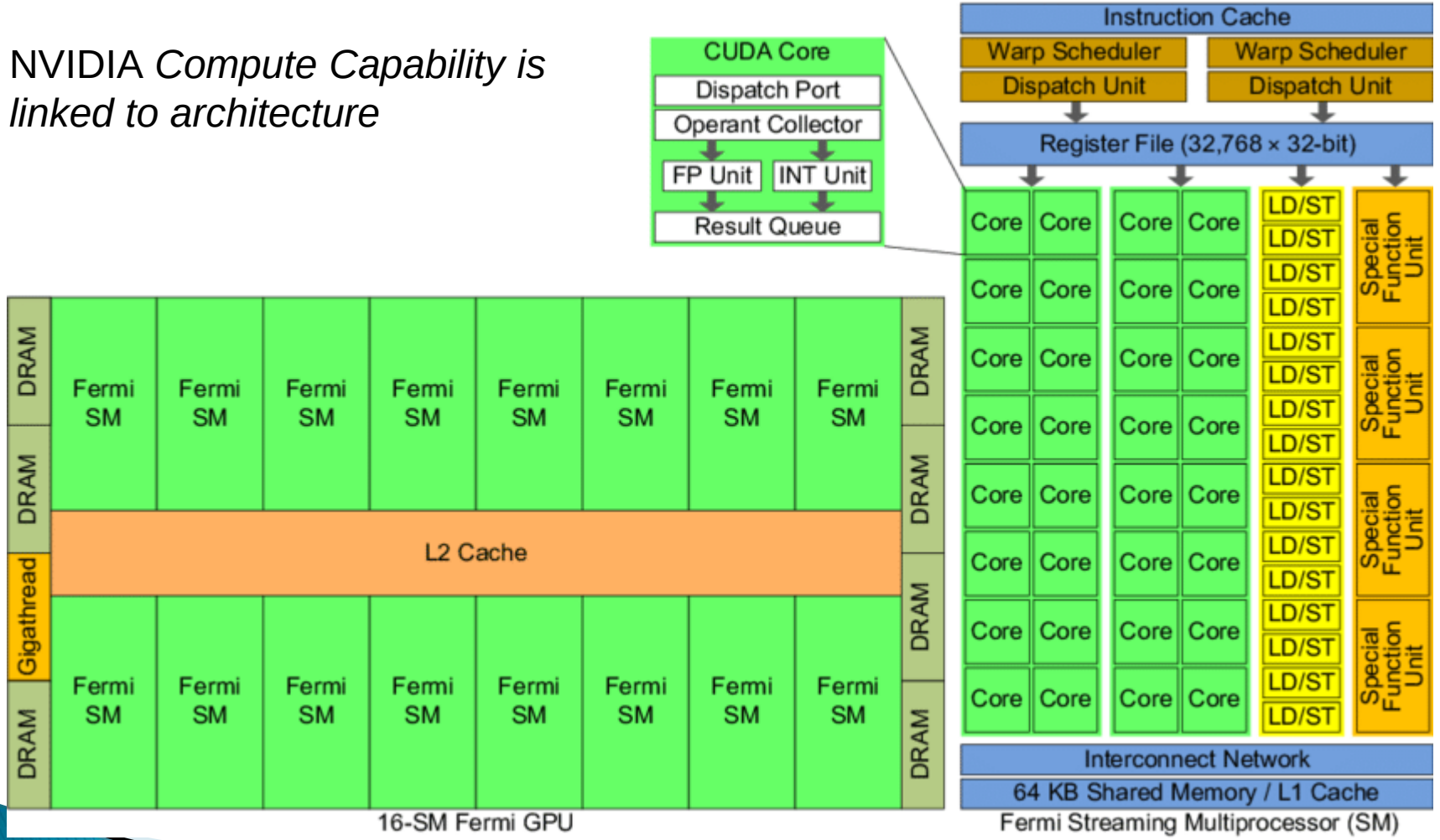
2 SMs (Compute Units) are grouped into one TPC

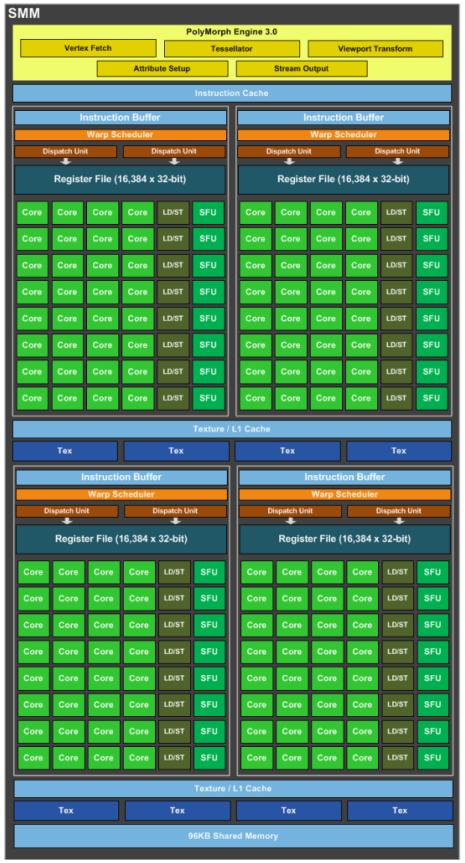


1st generation: Tesla

2nd generation: Fermi

NVIDIA *Compute Capability is linked to architecture*





This is only half of an SM

5th generation: Pascal

4rd generation: Maxwell

3rd generation: Kepler

VRIJE UNIVERSITEIT BRUSSEL

5th generation: Pascal

6th generation: Volta & Turing

Without double precision (DP) units

Simultaneous multithreading

Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

VRIJE UNIVERSITEIT BRUSSEL

Multithreading on CPU

- 1 process/thread active per core
- When activating another thread: *context switch*
  - Stop program execution: flush pipeline (let all instructions finish)
  - Save state of process/thread into **Process Control Block** : registers, program counter and operating system-specific data
  - Restore state of activated thread
  - Restart program execution and refill the pipeline
- Processor 'sees' only 1 threads

Overhead

VRIJE UNIVERSITEIT BRUSSEL

Fine multi-threading: Hardware threads

- In several modern CPUs
  - typically 2 HW threads (Intel: hyperthreading)
- Devote extra hardware for keeping process state
- Thread switching by hardware
  - (almost) no overhead
  - within 1 cycle!
  - *Instructions in flight from different threads*

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HyperThreading
  - Two threads: duplicated registers, shared function units and caches

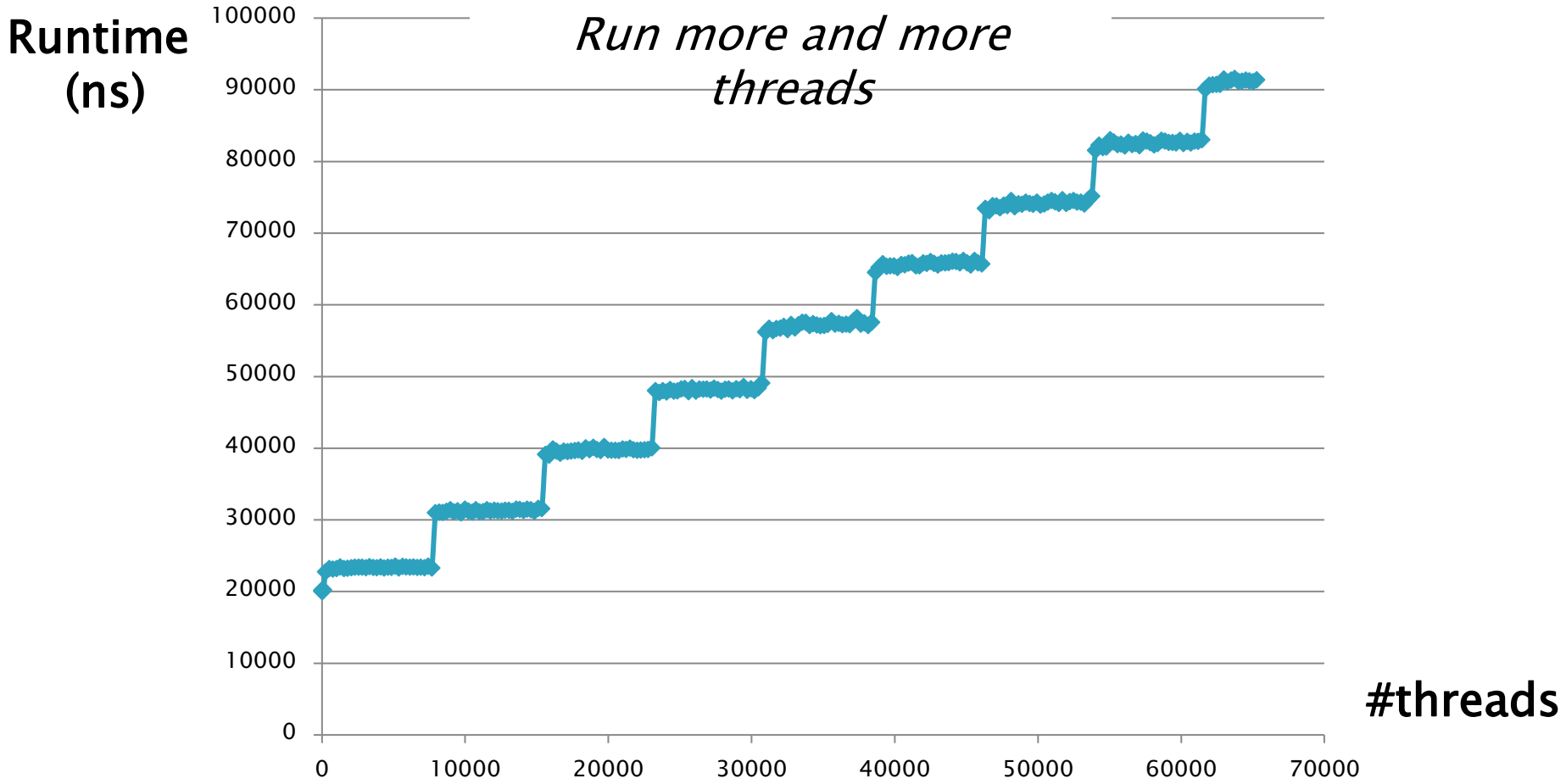
VRIJE UNIVERSITEIT BRUSSEL

Benefits of fine-grained multithreading

- Independent instructions (no bubbles)
- More time between instructions: possibility for *latency hiding*
  - **Hide memory accesses**
- **If pipeline full**
  - **Forwarding not necessary**
  - **Branch prediction not necessary**

VRIJE UNIVERSITEIT BRUSSEL

Running a simple addition kernel



Runtime increases only when all pipelines are full (8000 threads)

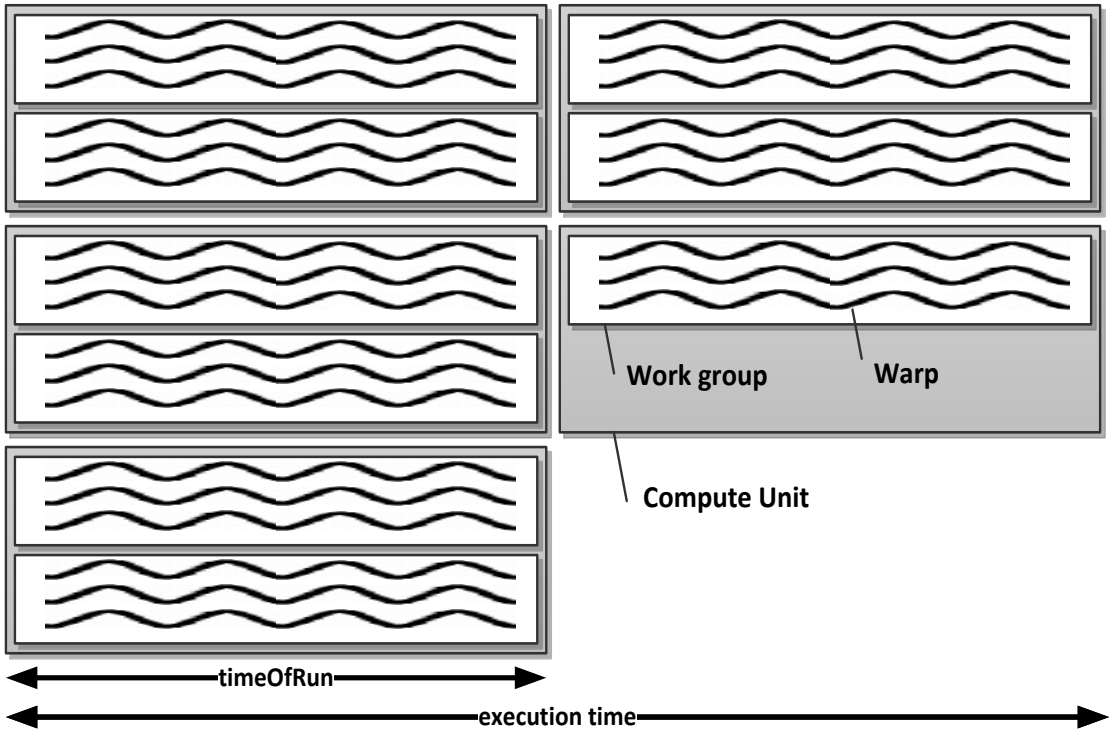
What determines the occupancy

VRIJE UNIVERSITEIT BRUSSEL

Occupancy

- Occupancy $= \#concurrent\ warps\ running\ on\ a\ Compute\ Unit$
- A higher occupancy means that more work can be scheduled and in general a higher performance
- Limited resources will limit the number of work groups that can be simultaneously active and run concurrently:
  1. Registers needed per work group
     - Each kernel's local variables are stored in register memory
  2. Local memory needed per work group
  3. Maximum number of concurrent work groups
  4. Maximum number of work items
- The most constrained resource determines the occupancy
  - Each Compute Unit has resources (depends on architecture, can be queried)
    - For Pascal architecture: 256KB registers, 96KB local memory, max. 32 work groups, max. 2048 work items

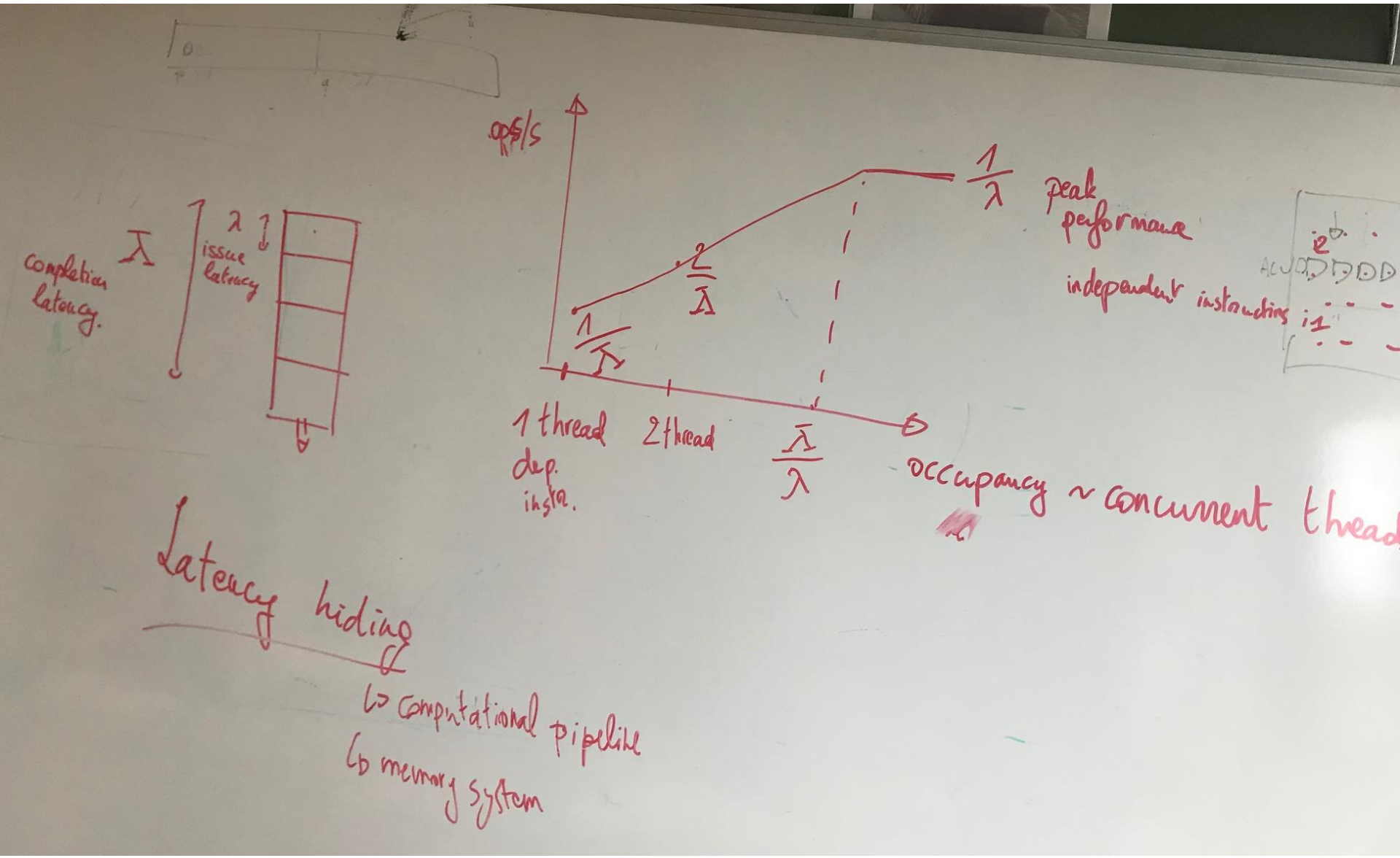
The execution on a GPU



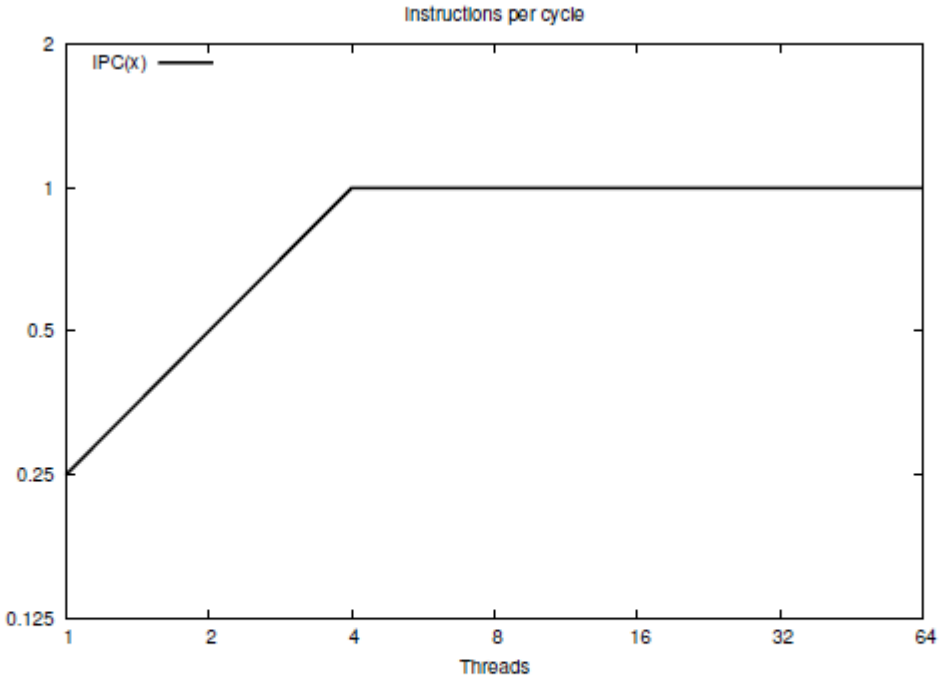
- Work groups are scheduled on compute units (cores).
- Warps of active work groups are scheduled on the core

The pipeline model

Pipeline concepts



Occupancy roofline



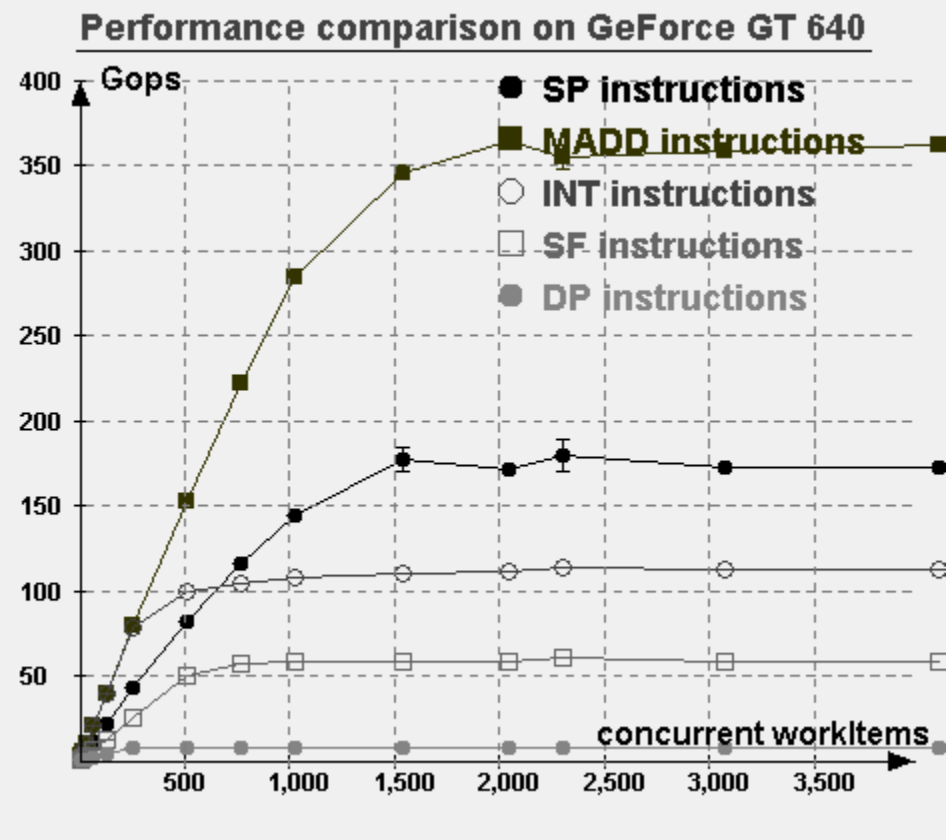
$$IPC_{N \leq \frac{\Lambda}{\lambda}} = \frac{N}{\Lambda}$$

$$IPC_{N \geq \frac{\Lambda}{\lambda}} = \frac{1}{\lambda}$$

Figure 3.2: The occupancy roofline plots the performance of a simple pipeline with $\lambda = 1$ and $\Lambda = 4$ in function of the concurrent number of threads. The ridge of the roofline is reached for $\frac{4}{1}$ threads.

VRIJE UNIVERSITEIT BRUSSEL

Peak performance in function of occupancy

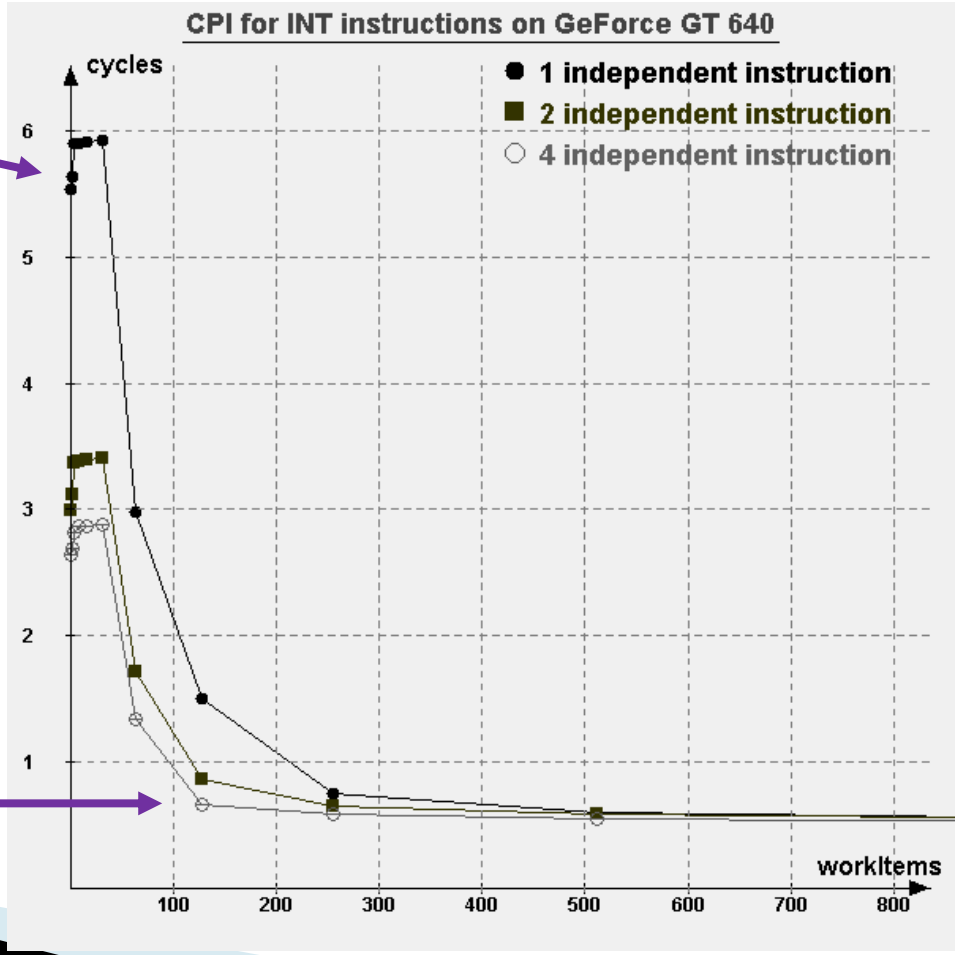


VRIJE UNIVERSITEIT BRUSSEL

Cycles Per Instruction

- The reversed graph

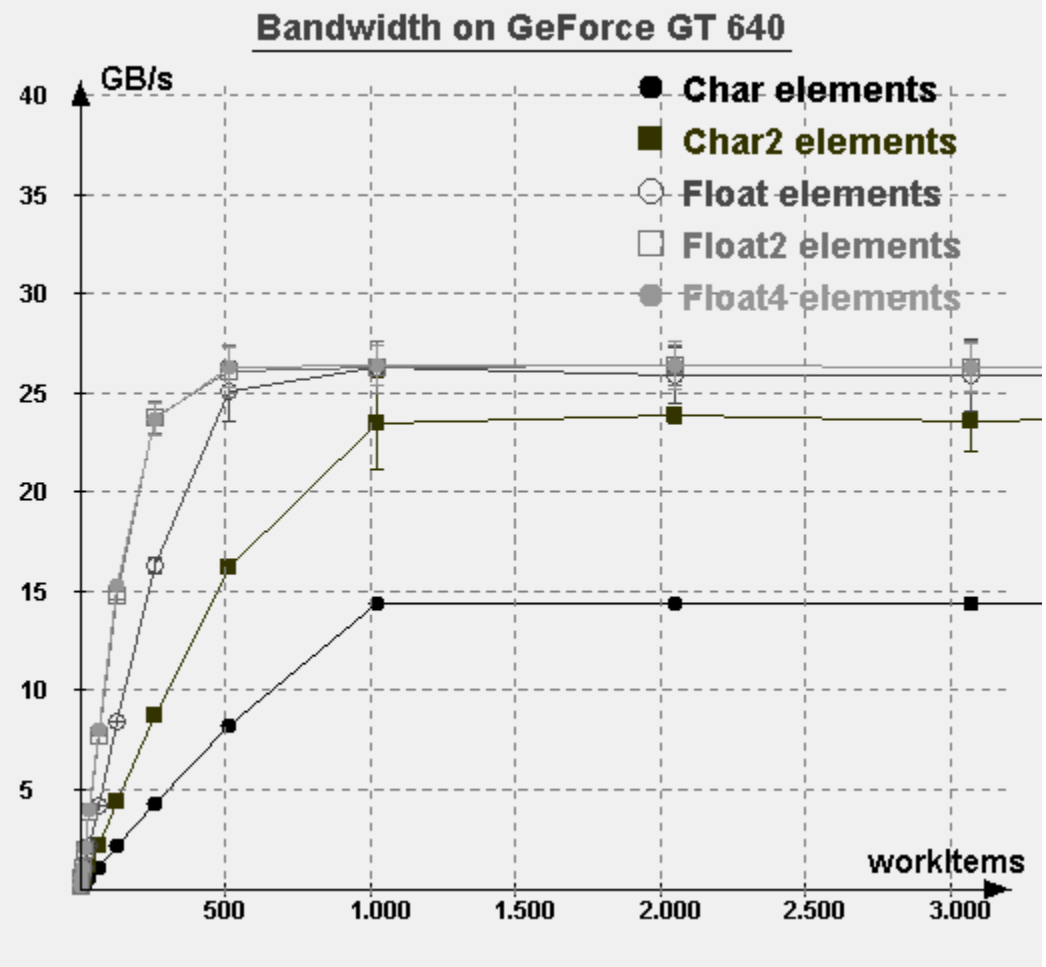
Completion latency



Issue latency

VRIJE UNIVERSITEIT BRUSSEL

Bandwidth



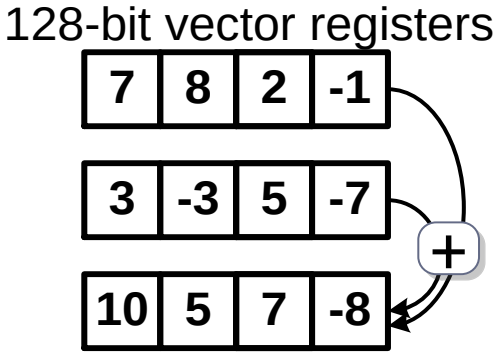
occupancy

Vector processors & SIMD

One way to do several computations at the same time

VRIJE UNIVERSITEIT BRUSSEL

Vector processors



Instructions are performed at once on all elements of the vector registers

- All processing elements execute the same instruction at the same time
  - Multiple data elements in 128-bit or 256-bit wide registers (vector registers)
  - MMX and SSE instructions in x86
- Instead of iterating over the vector (for-loop), one instruction is sufficient

VRIJE UNIVERSITEIT BRUSSEL

Vector processors

- Simplifies synchronization
- Reduced instruction control hardware: an instruction has to be read only once for x number of calculations
- Works best for highly data-parallel applications
- Has long be viewed as the solution for high-performance computing
  - Why always repeating the same instructions (on different data)? => just apply the instruction immediately on all data
- *However*: difficult to program, since less flexible
  - Is OpenCL/SIMT easier?

VRIJE UNIVERSITEIT BRUSSEL

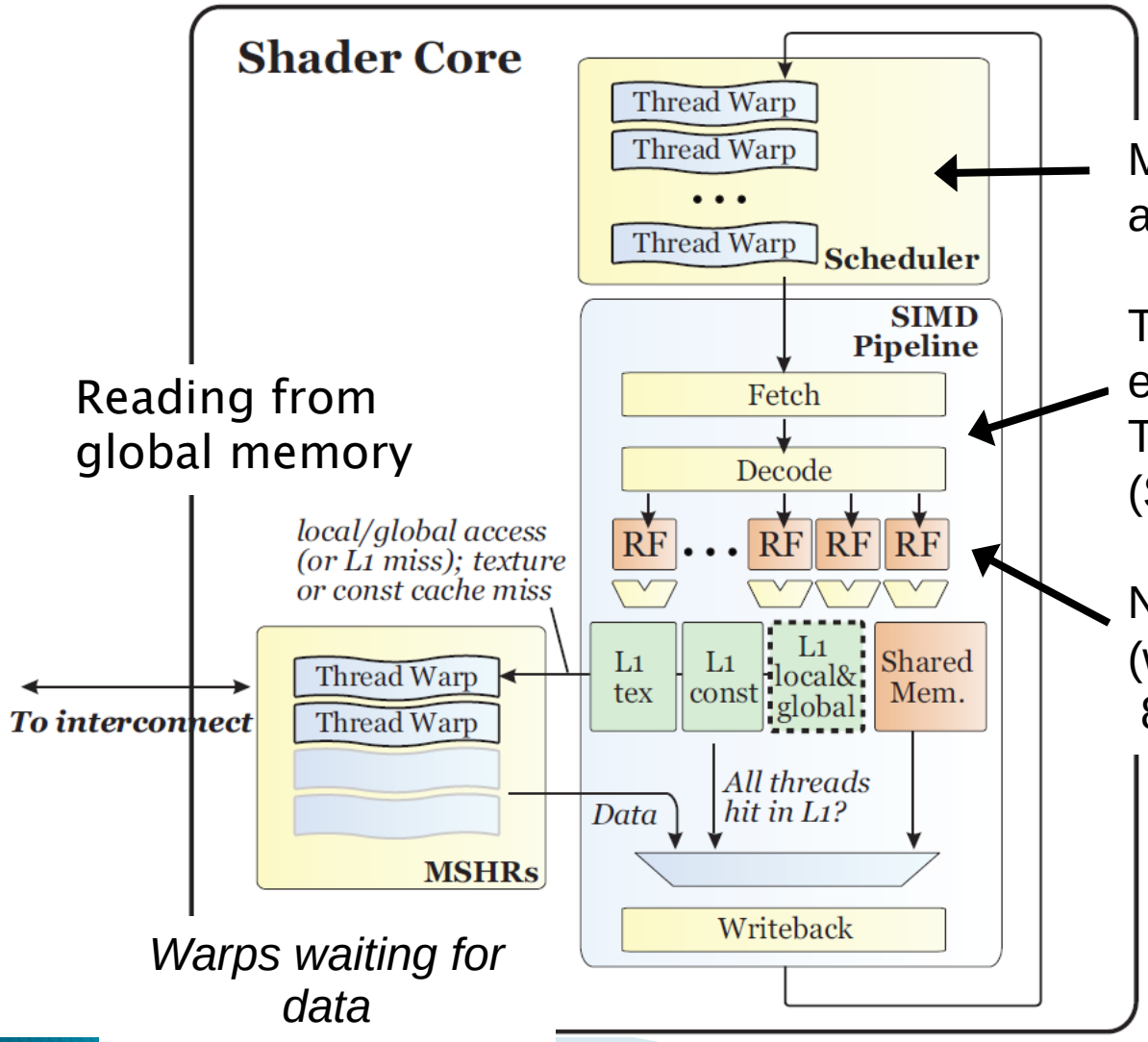
Instruction and Data Streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- Vector processing = Single Instruction on Multiple Data (SIMD)

Level 3
Hardware Threads & SIMT

1 Streaming Multiprocessor = a pipeline



Multiple warps (hardware threads) are simultaneously active

The Same Instruction is executed on Multiple work items/ Threads (SIMT)

Number of processing elements (width of pipeline):
8 – 32 – 192 - 128

VRIJE UNIVERSITEIT BRUSSEL

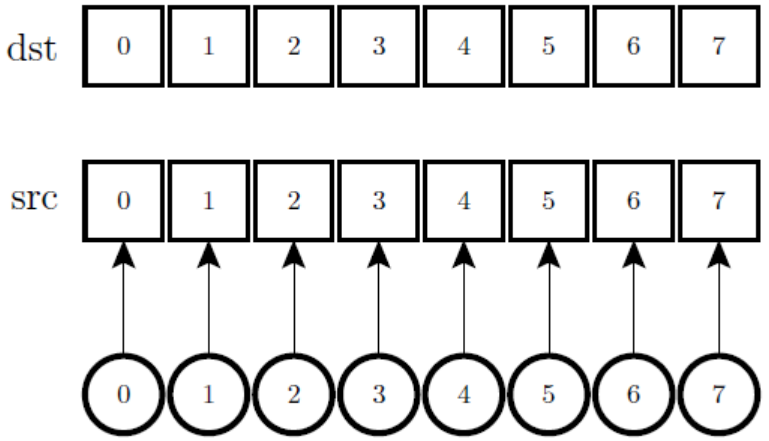
Warp executes work items in lock step

- **Hardware thread (called warp by Nvidia):**
  - Work items are executed together in groups, the instructions of the kernel are executed at the same time they will execute the same instruction
  - Nvidia: 32; AMD: 64; Intel: variable number (8/16/24/32)
- **Consequences:**
  1. Running 1 work item or 32 work items takes the same amount of time
     - Thus: create workgroups which are multiples of 32 or 64 (AMD)
  2. Branching: if work items of the same warp take different branches, all branches will be executed after each other
     - Performance loss
  3. Concurrent memory access: if work items access memory, all work items of the same warp do it simultaneously
     - Not all memory access can be done with the same speed

VRIJE UNIVERSITEIT BRUSSEL

When is SIMT = vector processing?

- Contiguous data access (See lesson 2)



- Warp execution of instructions on the data is similar to vector instructions operating on vector registers.

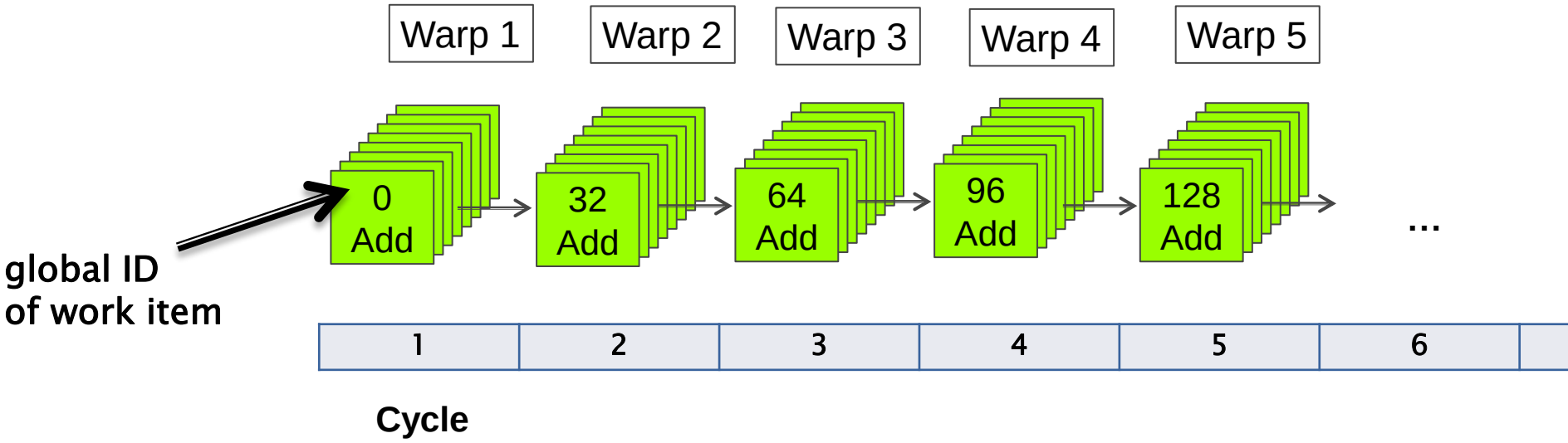
VRIJE UNIVERSITEIT BRUSSEL

Vectors versus SIMT

- Vectors
  - Data should be stored in vector register
  - Instructions are performed onto these registers
  - Harder to program
- SIMT
  - Each thread of a warp can choose on which data it works
  - Easier to program: programmer does not have to worry about *work item-data mapping*

Warp execution

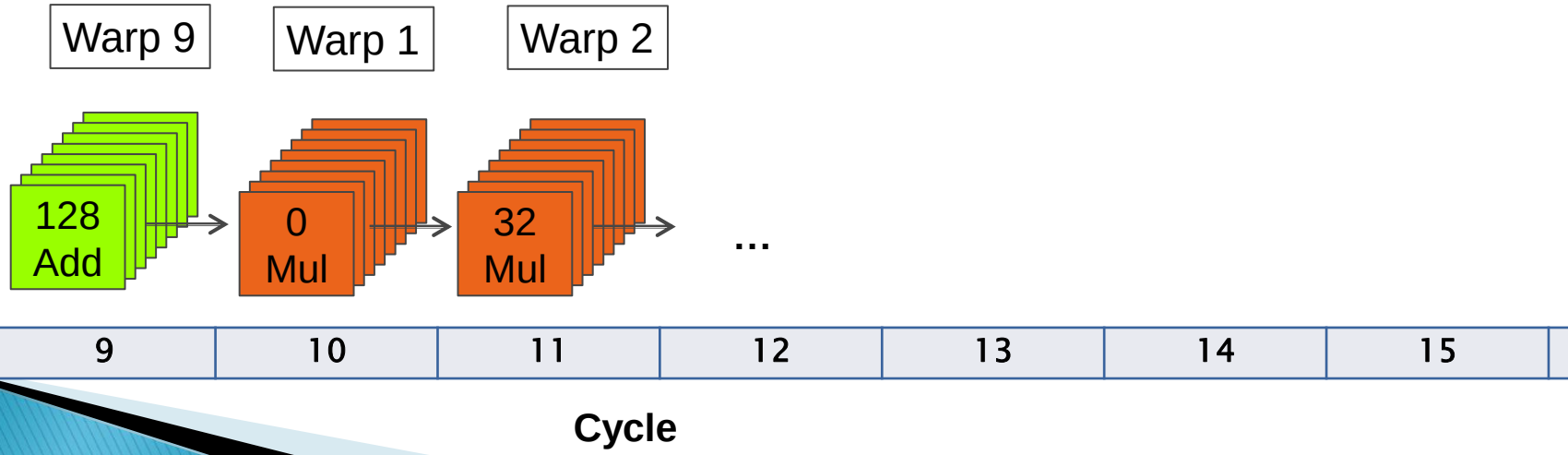
- Work items are sent into the pipeline grouped in a warp
  - ALUs all execute the same instruction in `lockstep': Single Instruction, Multiple Threads (SIMT)
  - Every cycle a new warp can issue an instruction



Warp execution

- On an Nvidia Kepler architecture, a single precision floating point instruction (add or multiplication) takes 9 cycles, which is the length of the pipeline.
  - 8 other warps can be scheduled in the mean time
  - After 9 cycles, the second instruction of the first warp (multiplication) can be issued, next the second warp and so on

  ⇒With 9 warps the pipeline is completely filled, no stalling/idling, the completion latency of 9 cycles is completely hidden.



VRIJE UNIVERSITEIT BRUSSEL

SIMT Conditional Processing

- If work items of a warp follow different branches, the instructions of both branches have to be executed, but are desactivated for some threads.

  => Performance loss!

- **Example**: assume 8 threads, one instruction in if-clause, one in then-clause
  - 3 cyles in which 24 instructions are executed, 8 lost cycles (66% usage)