VUB VRIJE UNIVERSITEIT BRUSSEL ULB UNIVERSITÉ LIBRE DE BRUXELLES

GPU Computing

» Lesson 3: Architecture & Strategy - part I

Gauthier Lafruit & Jan Lemeire

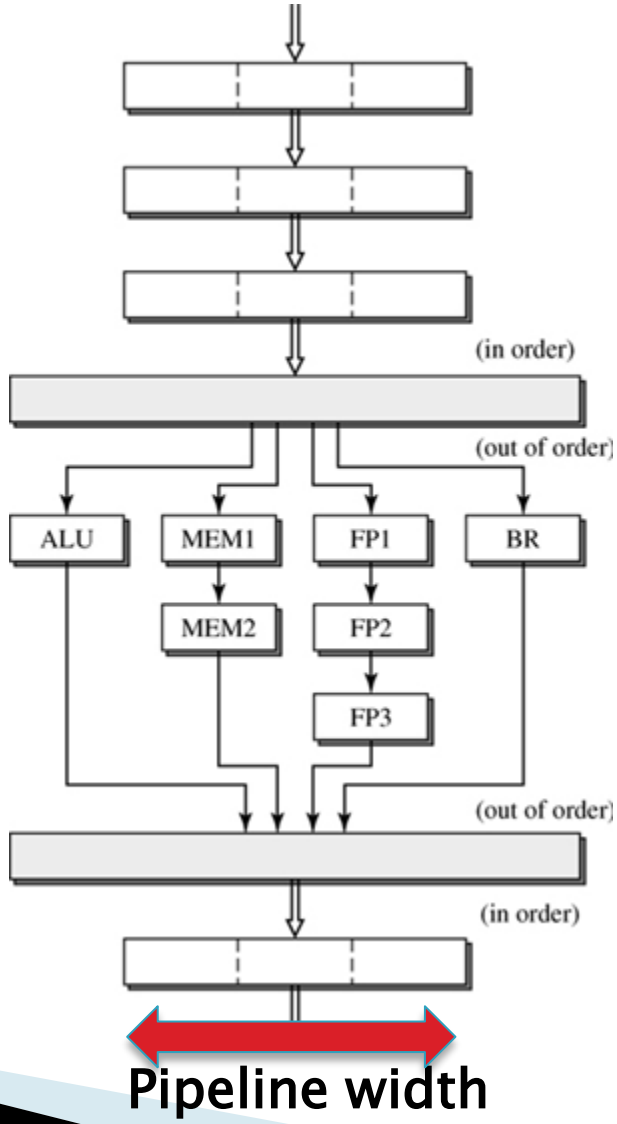
2022-2023

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

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!

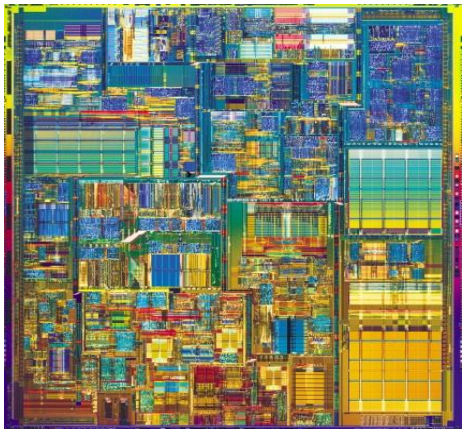
manual

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

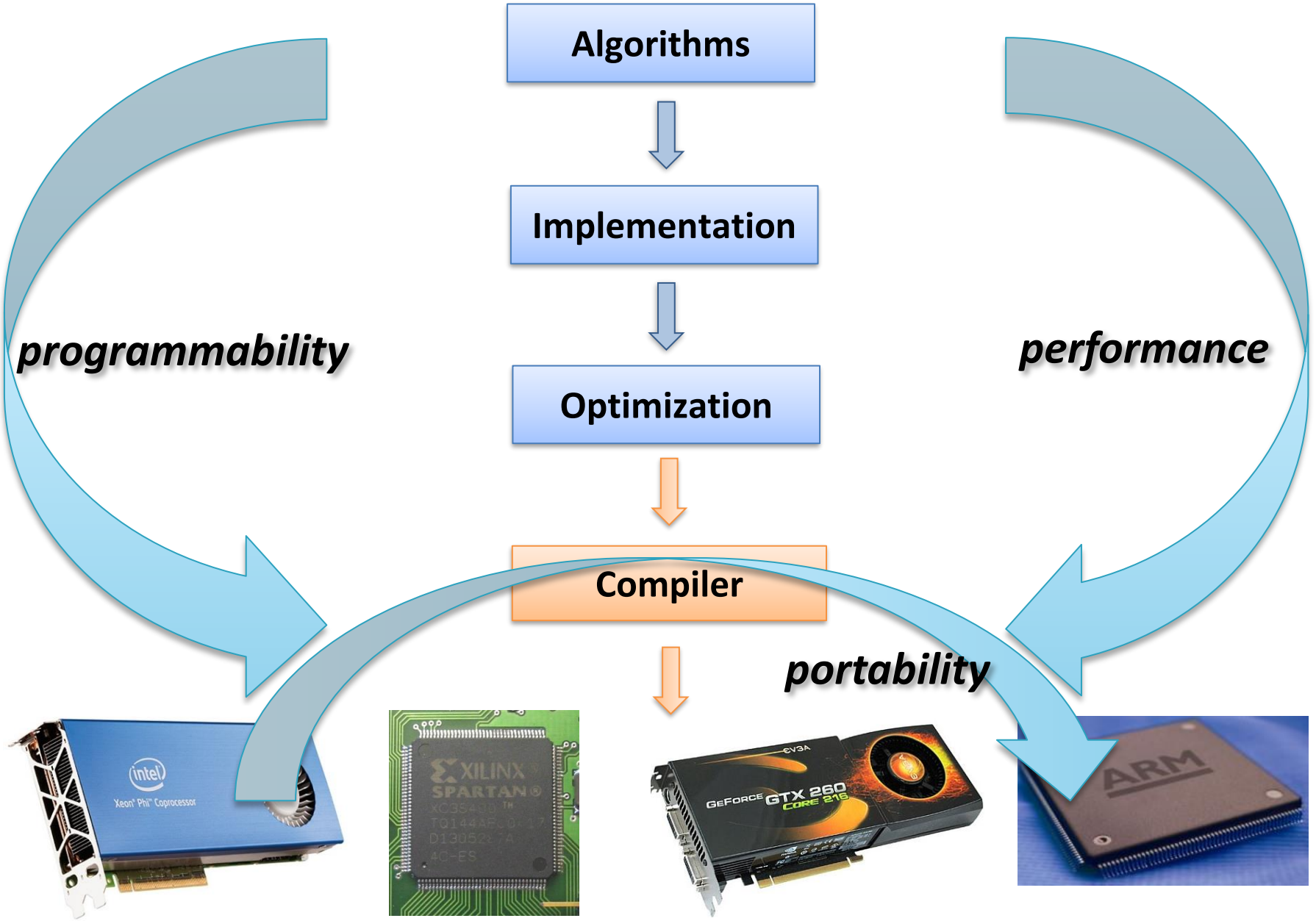
  ⇔ Modern (sequential) CPUs based on simple Von Neumann architecture

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

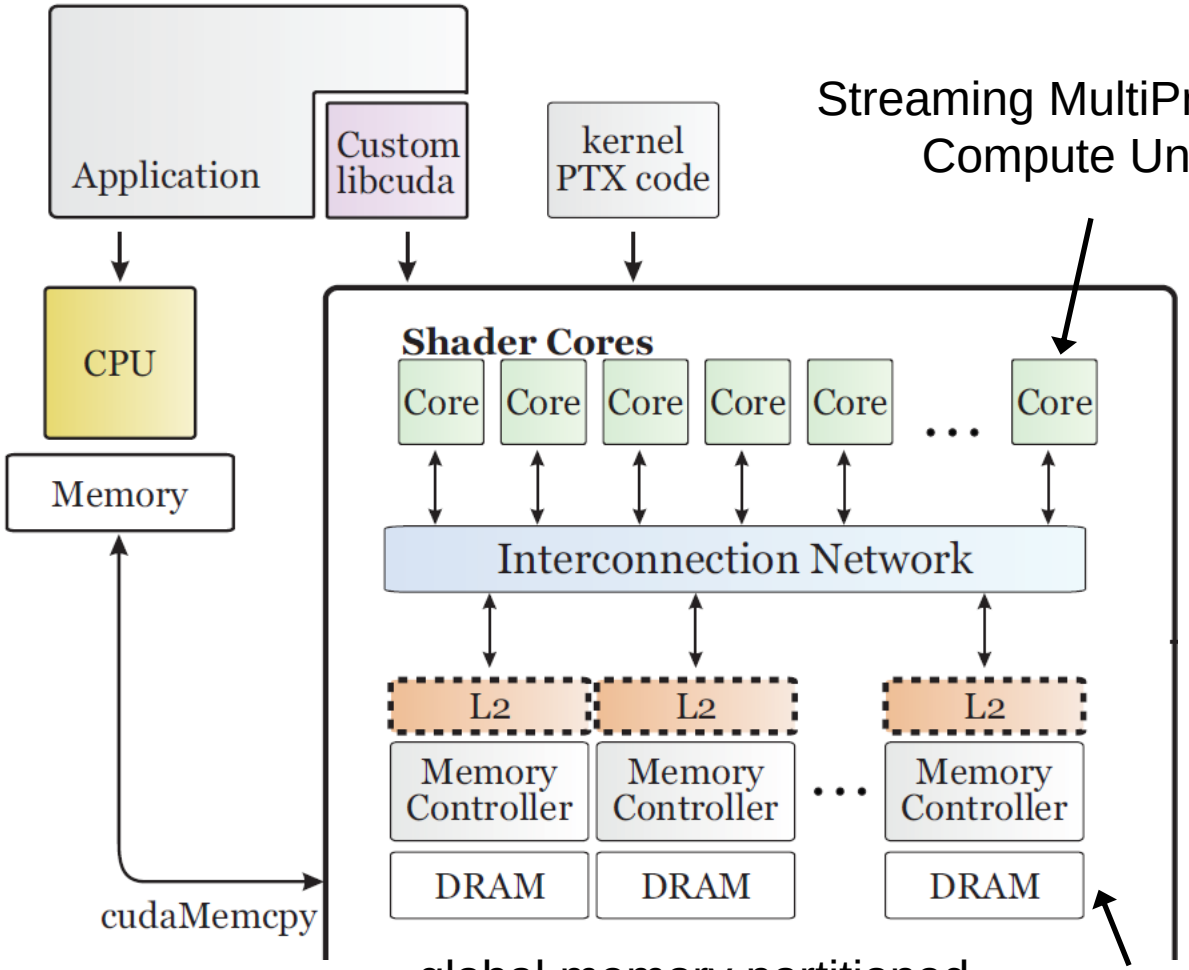
Fill the pipelines

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

1. Running thread blocks on the different multiprocessors
2. Simultaneous multithreading: several hardware threads active at the same time
   - *Discussed next*
3. Single Instruction Multiple Threads (SIMT)
   - *Discussed later*

Architecture

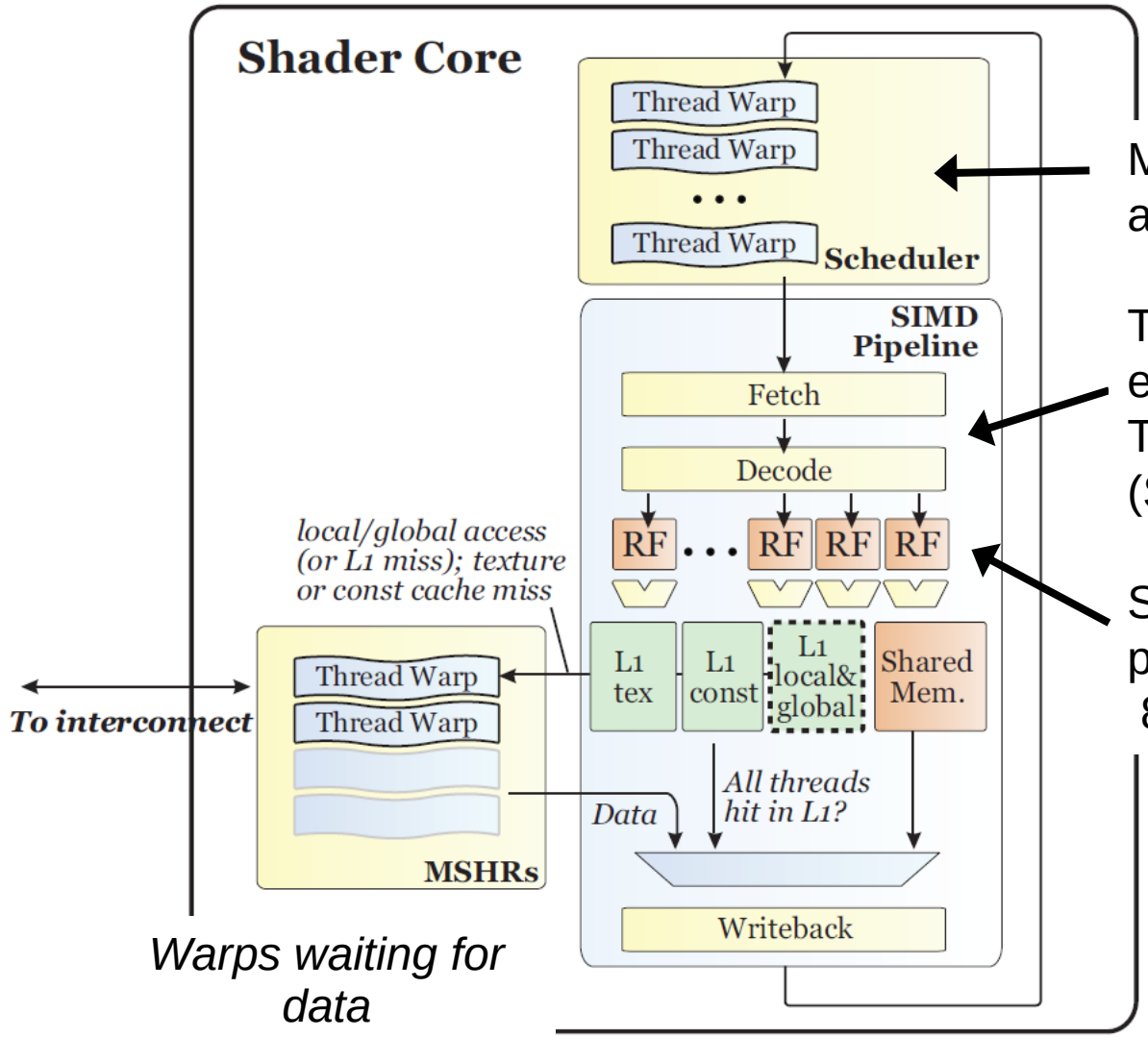
GPU Architecture



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)

Scalar Processors (width of pipeline):
8 – 32 – 192 - 128

Properties of different architectures

N(Π) = #multiprocessors
|ω| = warp size
Group & Warp slots: maximum # thread blocks 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

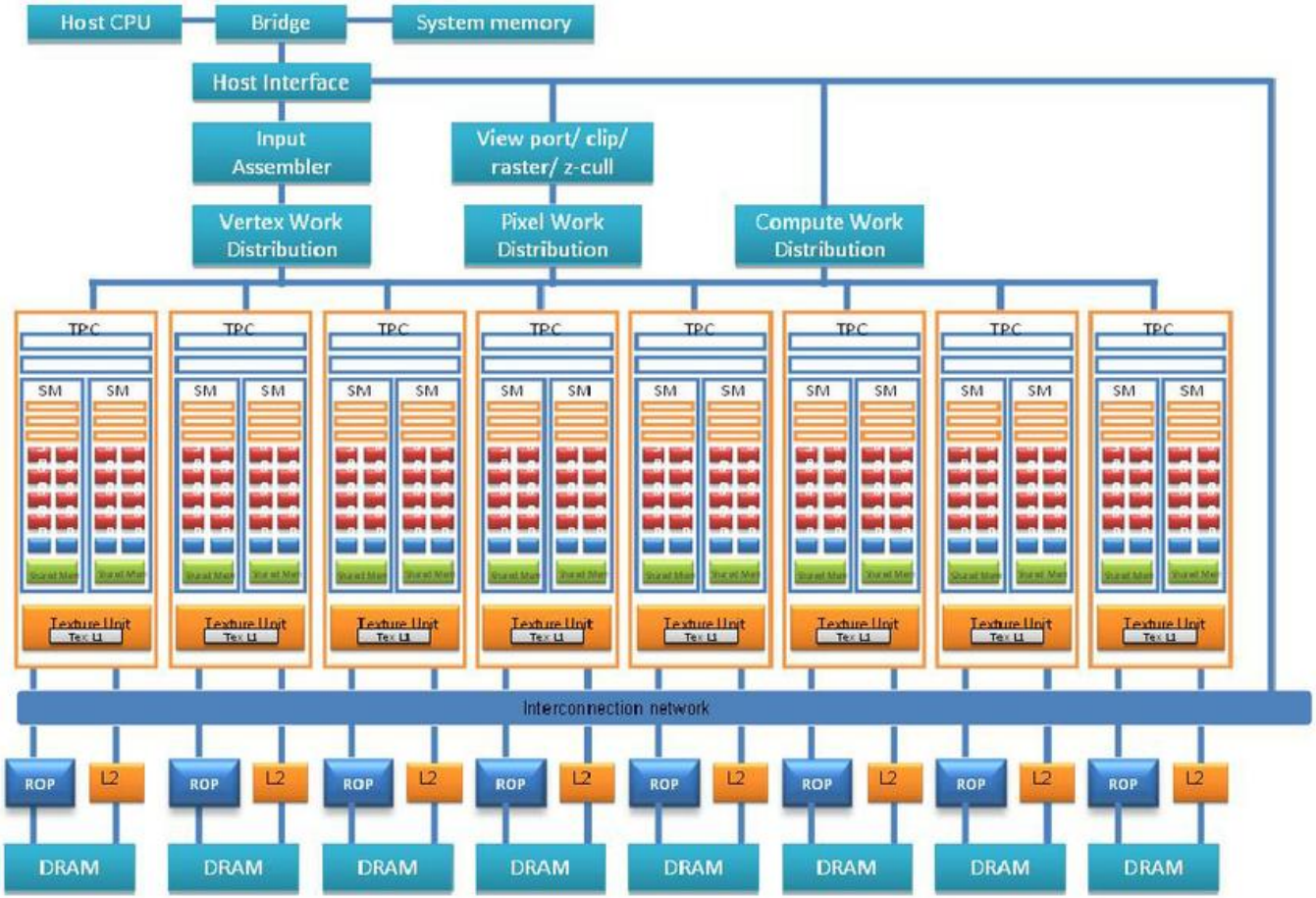
Full name	Abbreviated name
NIVDIA Tesla C2050	Fermi
NIVDIA GeForce GTX 650 Ti	Kepler
NIVDIA Quadro K620	Maxwell
NIVDIA GeForce GTX 1060 6GB	Pascal
AMD Radeon R9 380	Tonga

VRIJE UNIVERSITEIT BRUSSEL

https://en.wikipedia.org/wiki/CUDA

The different Nvidia architectures

2 SMs (Compute Units) are grouped into one TPC



CUDA Compute Capability can be queried, also in GPU Caps Viewer

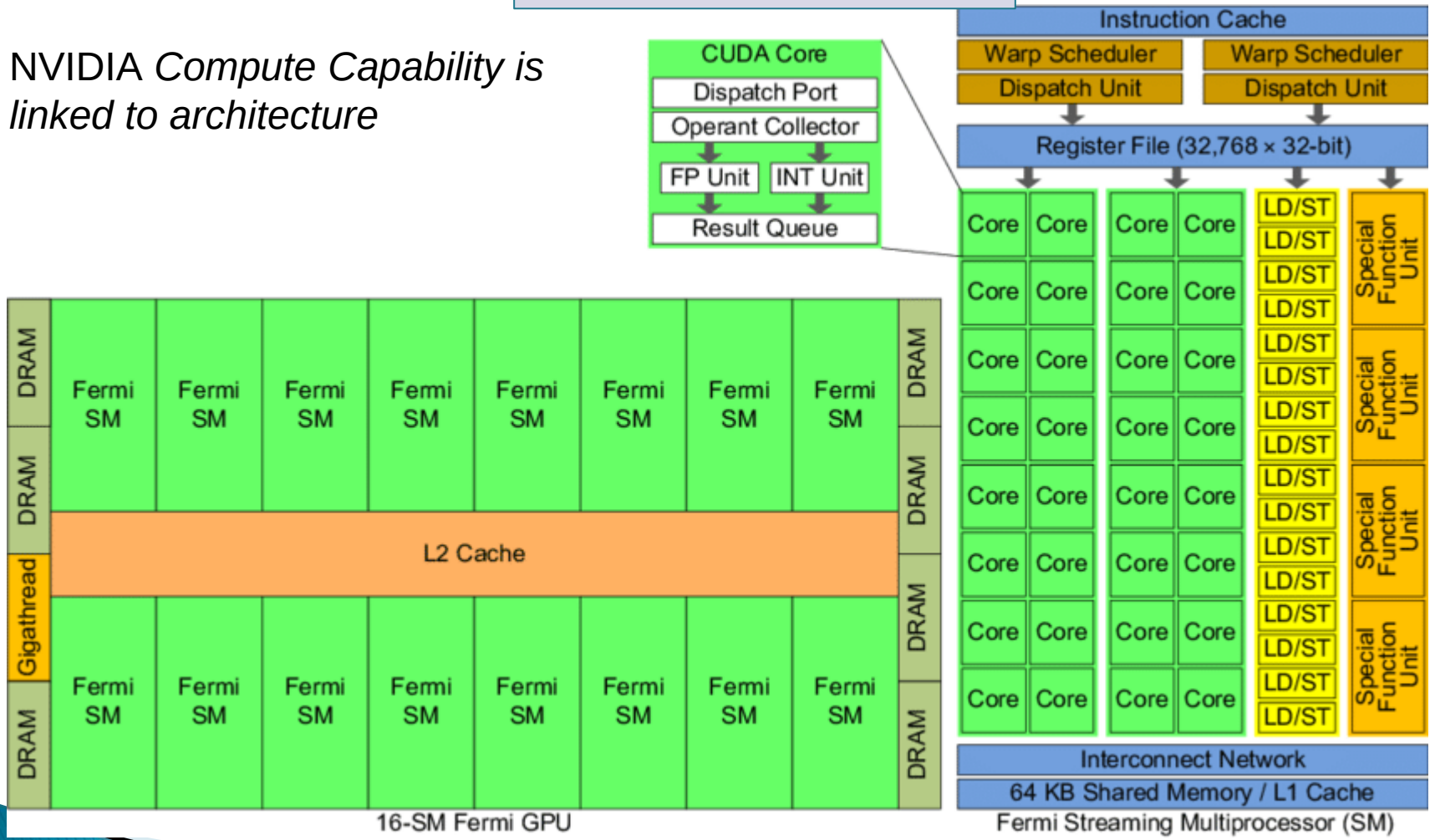
1st generation: Tesla

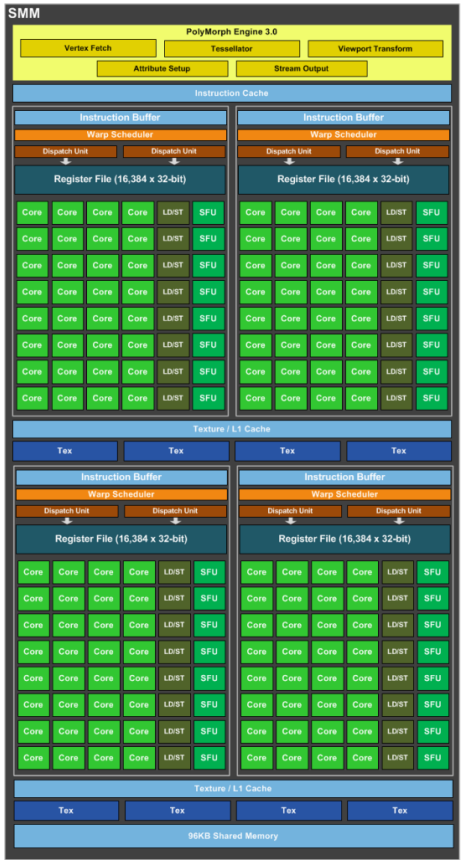
Compute Capability = 1.x

2nd generation: Fermi

Compute Capability = 2.x

NVIDIA *Compute Capability is linked to architecture*





This is only half of an SM

3rd generation: Kepler

Compute Capability = 3.x

4rd generation: Maxwell

Compute Capability = 4.x or 5.x

5th generation: Pascal

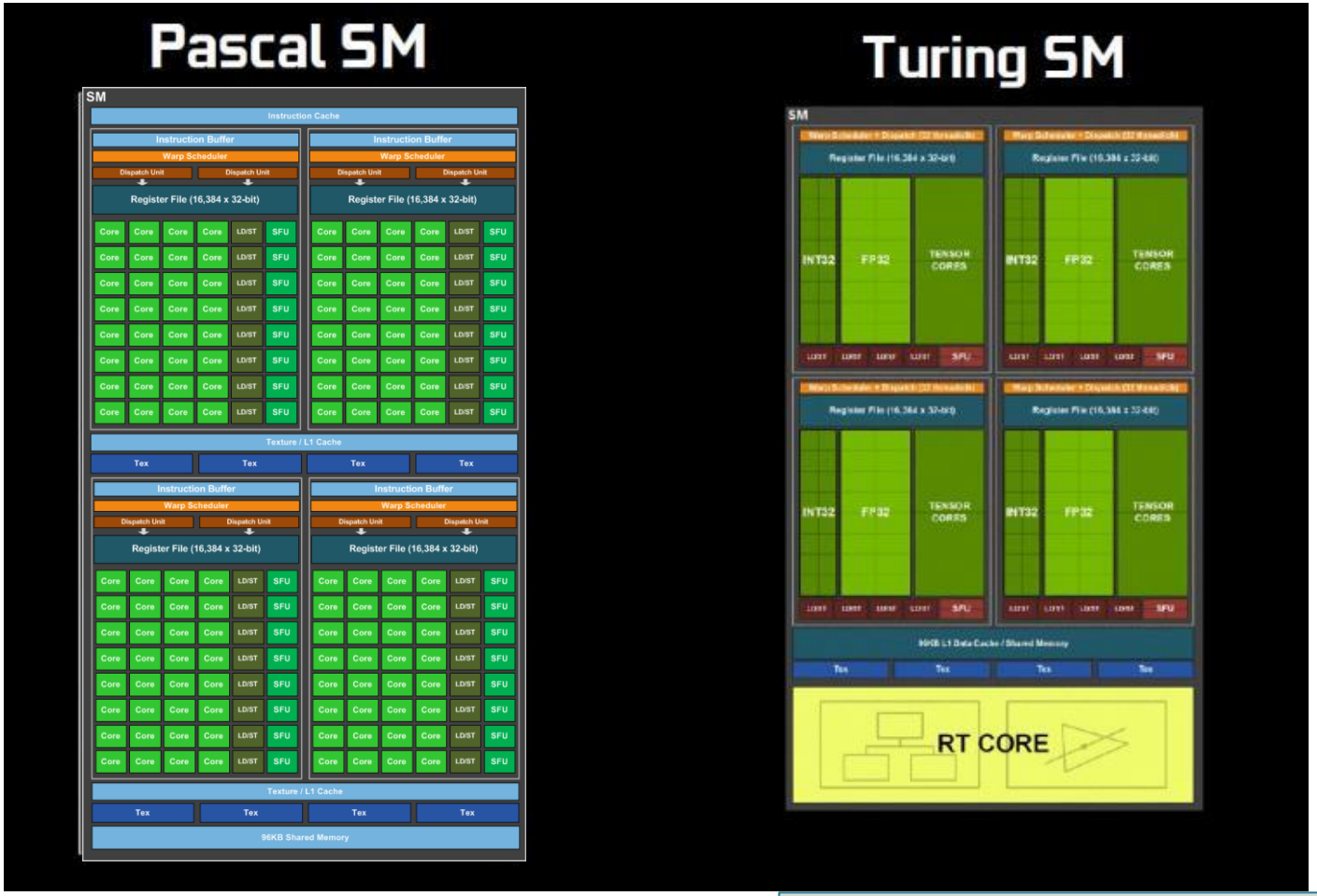
Compute Capability = 6.x

VRIJE UNIVERSITEIT BRUSSEL

Compute Capability = 8.x
if < 8.5: Volta

5th generation: Pascal

6th generation: Volta & Turing



Without double precision (DP) units

7th generation: Ampere

Compute Capability = 9.x

Simultaneous multithreading

VRIJE UNIVERSITEIT BRUSSEL

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

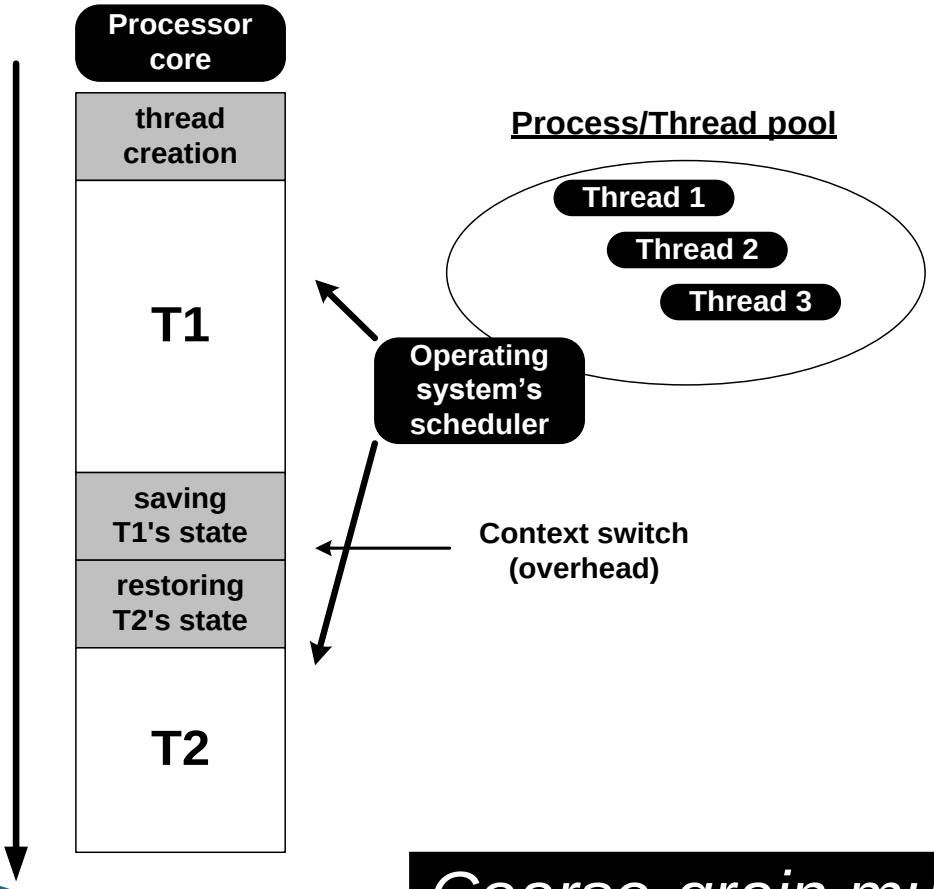
Multithreading on CPU

- 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

Overhead

- Processor 'sees' only 1 thread
- Called **Software threads**

Running threads on same CPU core



- Executed one by one
- Context switch
  - Thread's state in core: instruction fetch buffer, return address stack, register file, control logic/state, …
  - Supported by hardware
- Takes time!

Coarse-grain multithreading

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*

VRIJE UNIVERSITEIT BRUSSEL

Simultaneous Multithreading

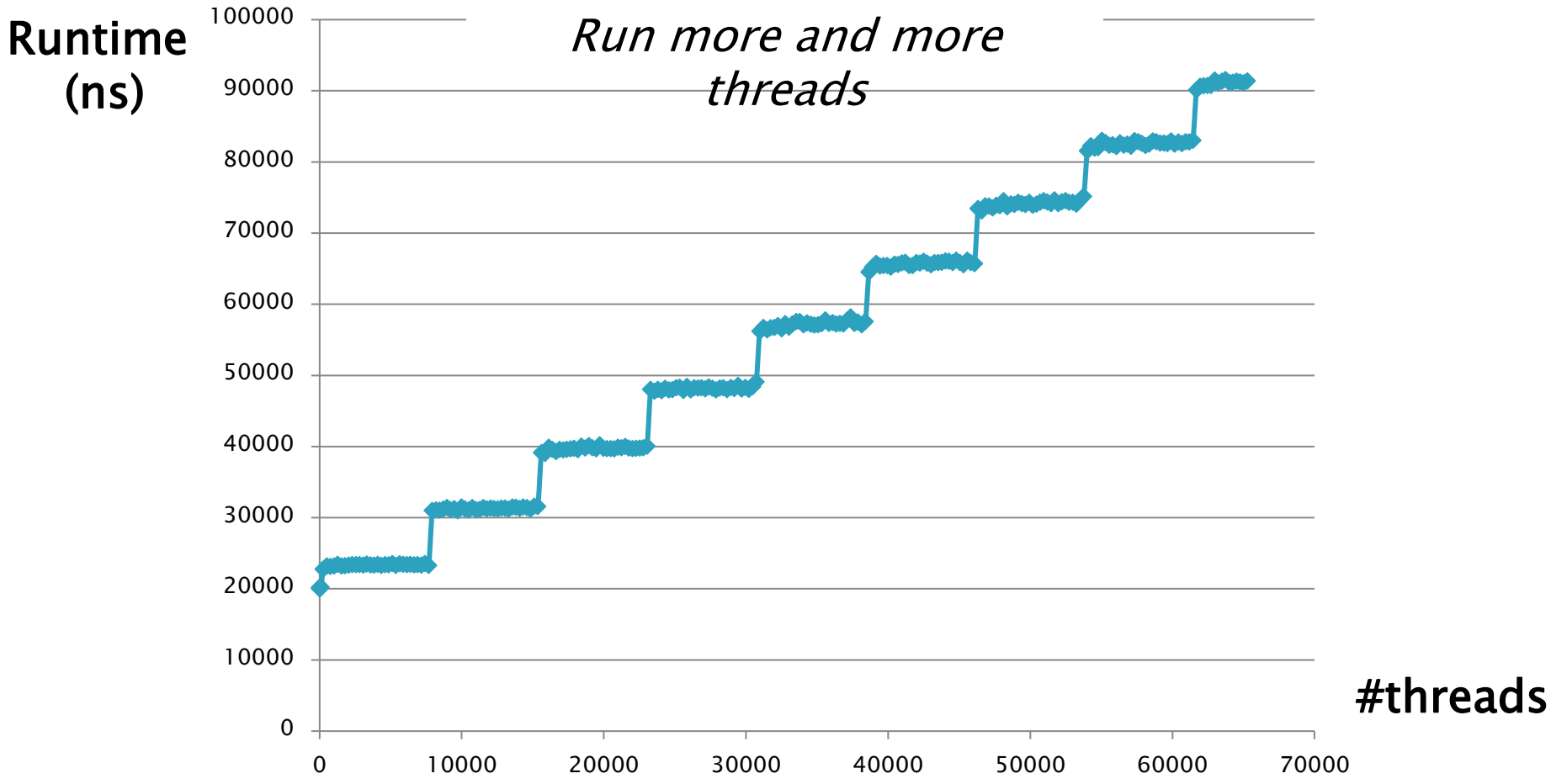
- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HyperThreading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

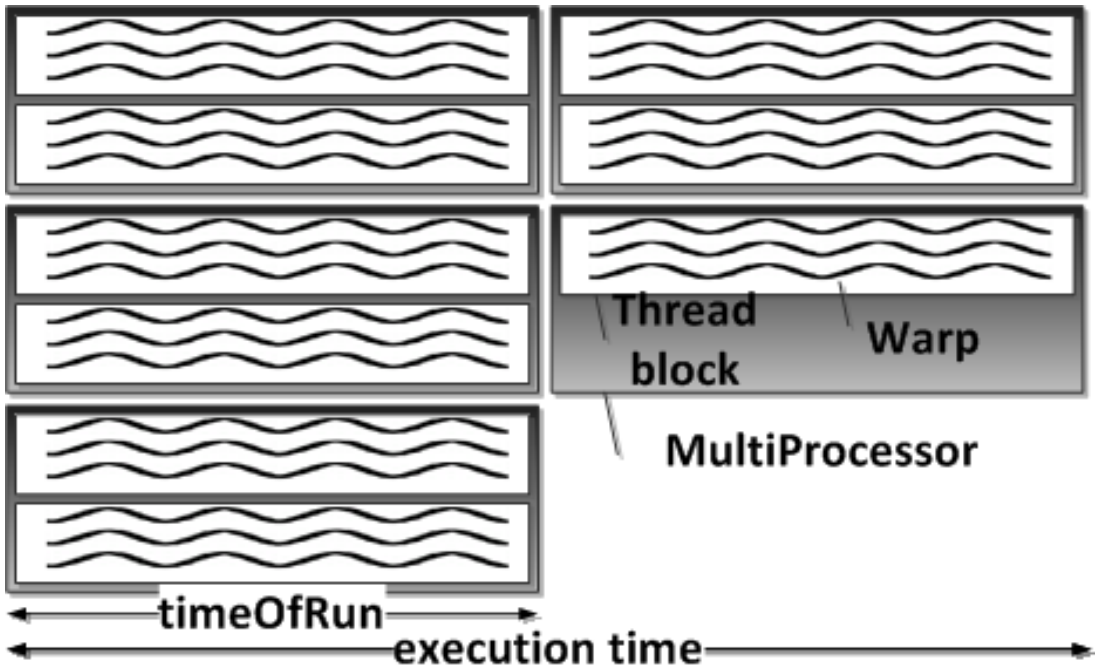
Running a simple addition kernel



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

VRIJE UNIVERSITEIT BRUSSEL

The execution on a GPU



- Thread blocks are scheduled on MultiProcessors .
- Warps of active threads are scheduled on the multiprocessor

VRIJE UNIVERSITEIT BRUSSEL

Concurrency

- Keep all processing units busy!
  - Enough threads
- All Multiprocessors (MPs)
- All Scalar Processors (SPs)
- Full pipeline of scalar processor
  - Pipeline of up to 24 stages

What determines the occupancy

VRIJE UNIVERSITEIT BRUSSEL

Occupancy

- Occupancy $= \#concurrent\ warps\ running\ on\ a$ Multiprocessor
- A higher occupancy means that more work can be scheduled and in general a higher performance
- Limited resources will limit the **number of thread blocks** that can be simultaneously active and run concurrently:
  1. Registers needed per block
     - Each kernel's local variables are stored in register memory
  2. Local memory needed per block
  3. Maximum number of concurrent thread blocks
  4. Maximum number of thread
- The most constrained resource determines the occupancy
  - Each Multiprocessor has resources (depends on architecture, can be queried)
    - For Pascal architecture: 256KB registers, 96KB local memory, max. 32 blocks, max. 2048 threads(=64 warps)

The effect of occupancy will be studied with the Pipeline Model