Norwegian University of
Science and Technology

The GPU and U

Bart Iver van Blokland

GPUs are commonplace in today's computers

What is a GPU and how does it differ from a CPU?

- **How did the GPU come to be?**
- How does the architecture of a GPU differ from a CPU?

History of the GPU

- Drawing pictures is generally too demanding for a CPU, so we offload that job to a separate graphics processor
    - The CPU can do useful stuff while the video processor is busy
- Very early “graphics chips” were developed for interactive text terminals
- Games have been a driving force in the development of graphics hardware

History of the GPU

The early days:

RAM is too expensive, so you generate the image as it is being sent to the screen

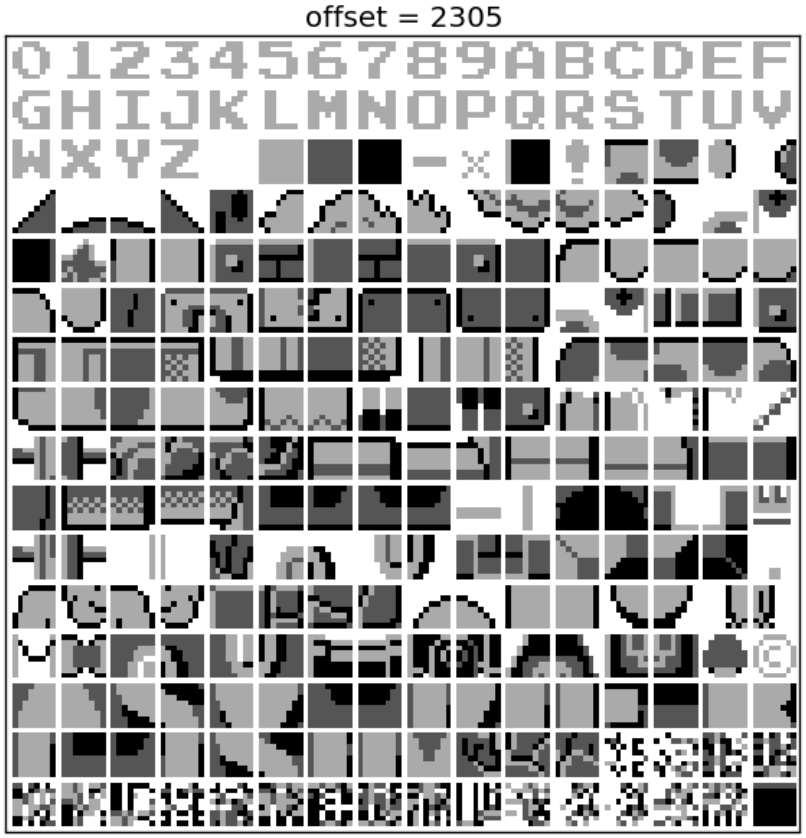
History of the GPU

The 1980’s

Increased capabilities of video processors allow for the creation of 2D scenes with some details by drawing a few small images (called sprites) over and over again.

Their capabilities expanded during the second half of the decennium.

History of the GPU



History of the GPU

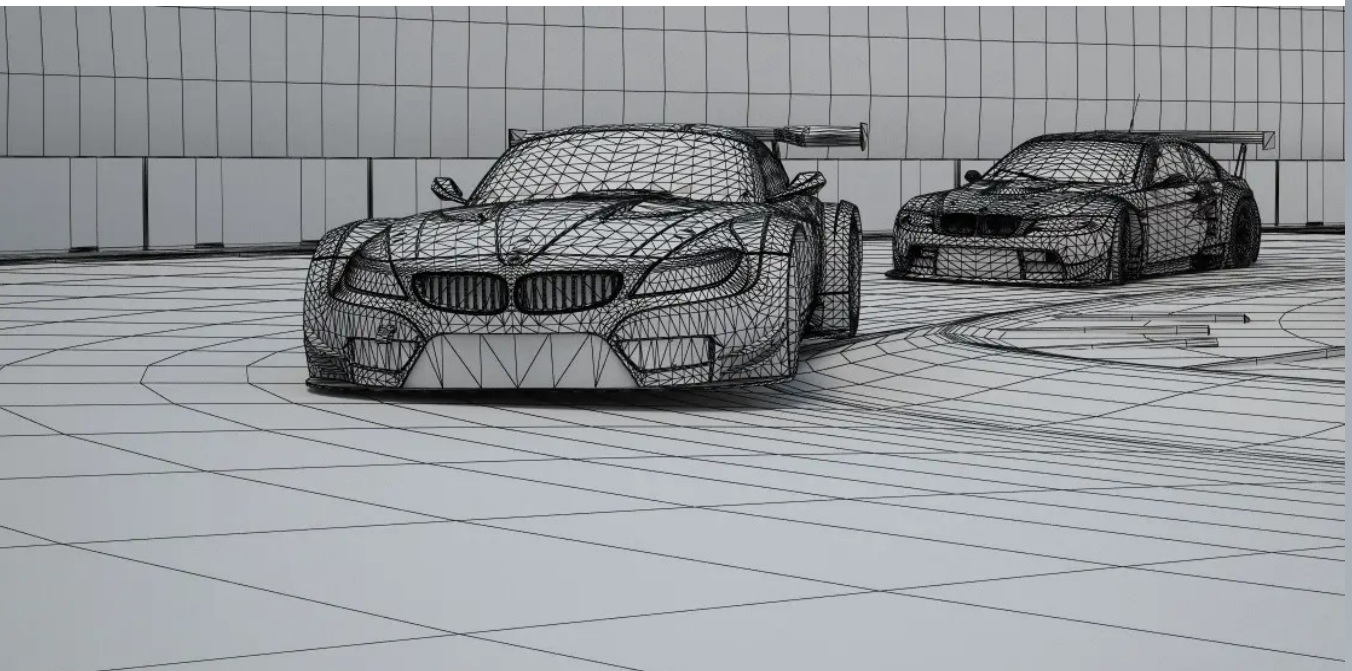
The 1990's

Advent of realtime 3D rendering on consumer hardware. Introduction of APIs such as OpenGL and Direct3D.

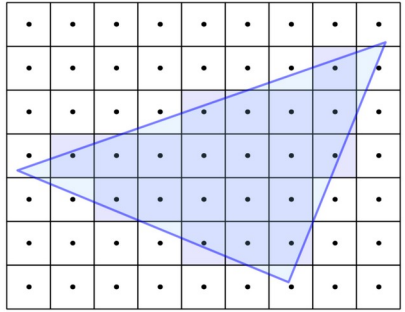
History of the GPU

Before we continue: how 3D is rendered

Step 1: Define and transform triangles



History of the GPU



Before we continue: how 3D is rendered

Step 2: Rasterise triangles and compute their colour

Texturing: pasting images on triangles

Lighting: simulate the effects of light sources

History of the GPU

The early 2000's

Great consolidation of manufacturers around the turn of the century.

State of the industry:

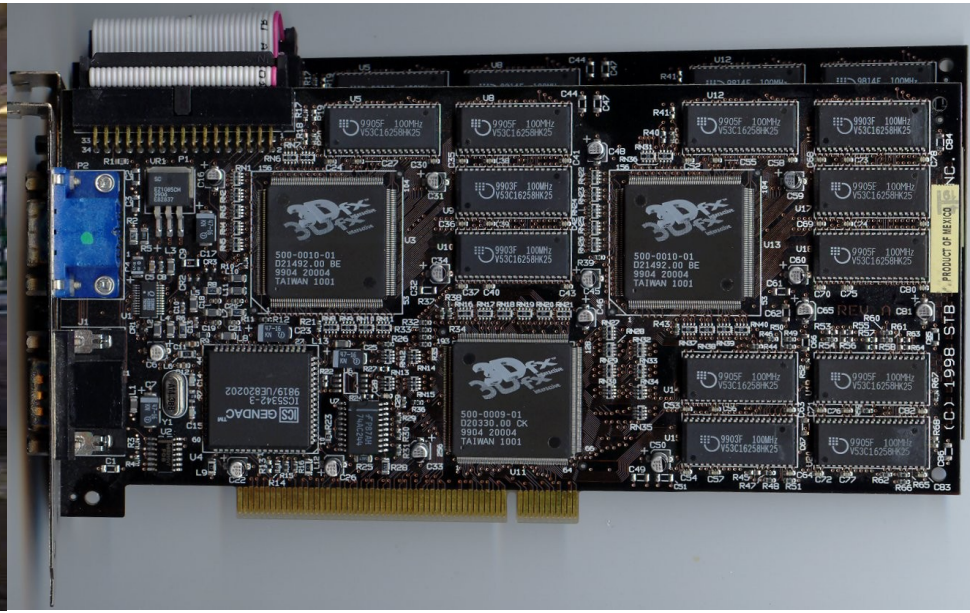
- Everything is done in fixed-function hardware

  Vertex transformations and pixel colours are determined by predefined equations to which the programmer supplies the constant and parameter values.

- Graphics card vendors distinguish their products with features such as higher resolution textures and better lighting.

Developers wanted more freedom to specify how their scenes are drawn..

History of the GPU



History of the GPU

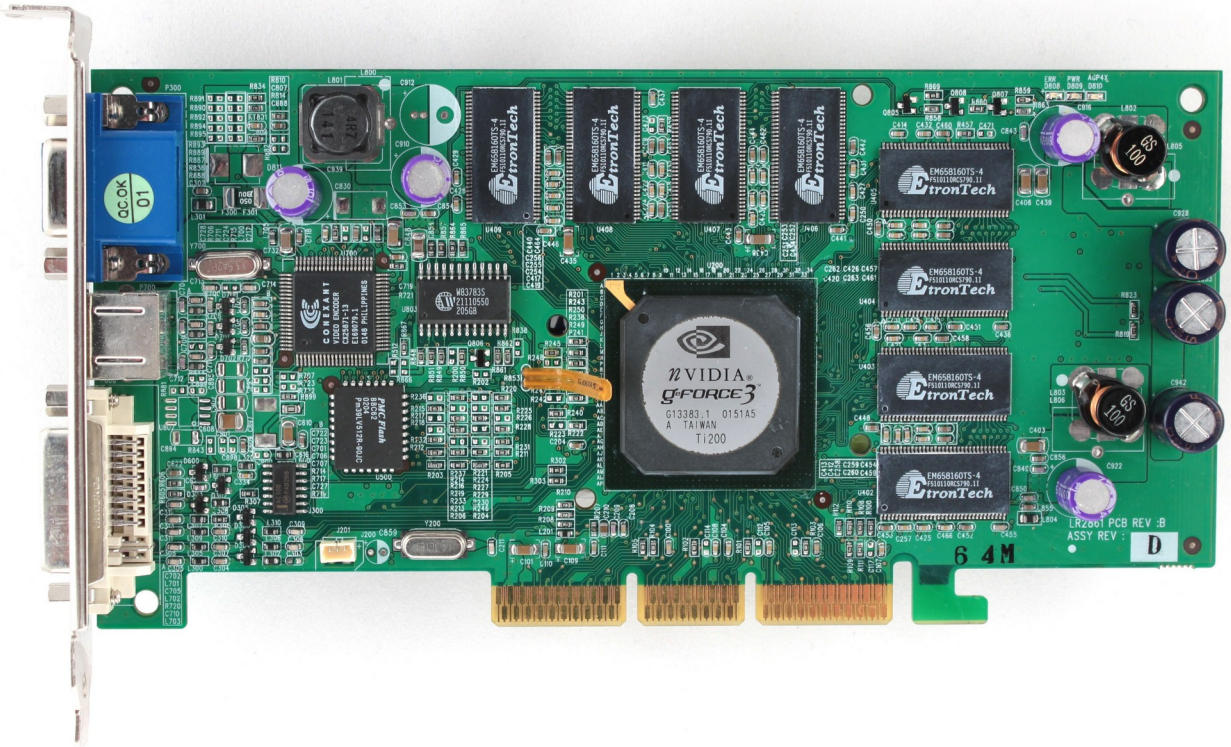
2001: Release of the GeForce 3

First programmable GPU

Vertex positions could be calculated using a short program called a “vertex shader”

Pixel colours were similarly computed using a “pixel shader”

Still mostly fixed function

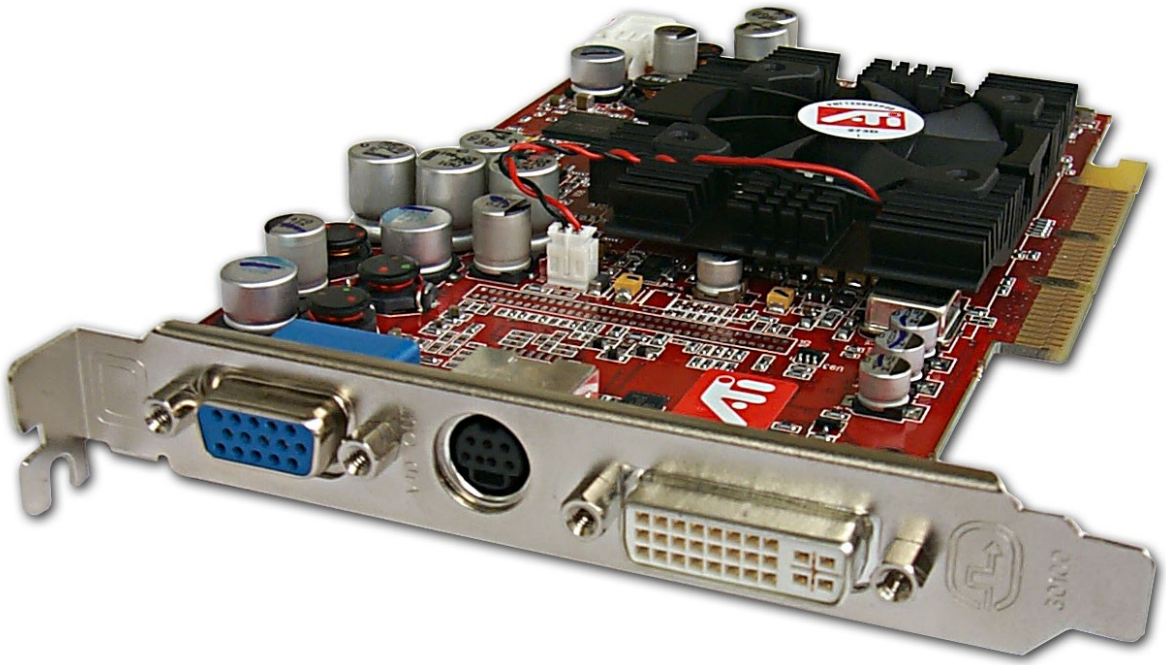


History of the GPU

2002 to 2005:

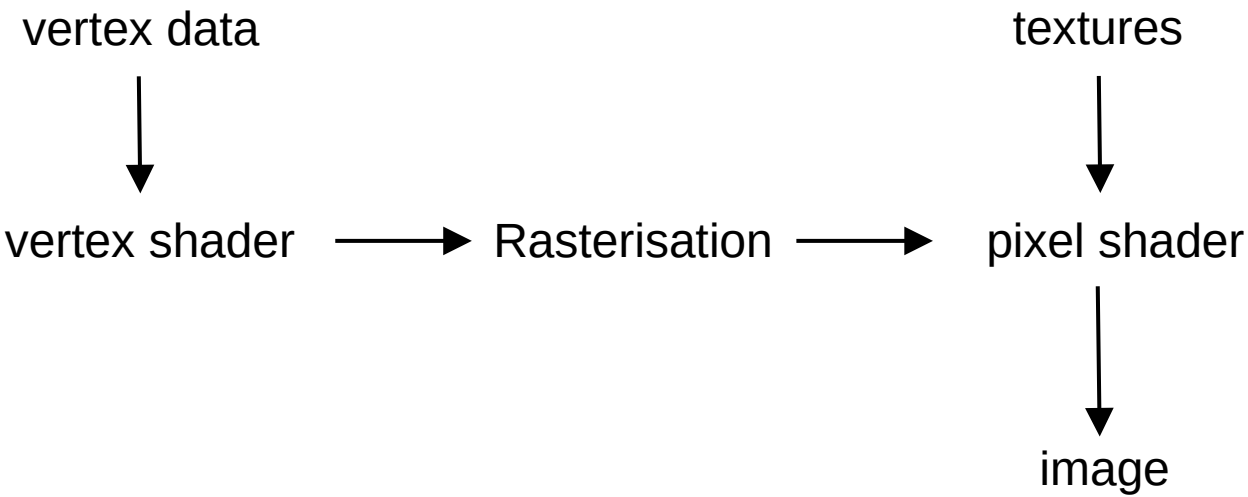
Improvements to the vertex and pixel shader processors increase their flexibility.

Flexibility also leads to the adoption of graphical techniques that were previously impossible to achieve.



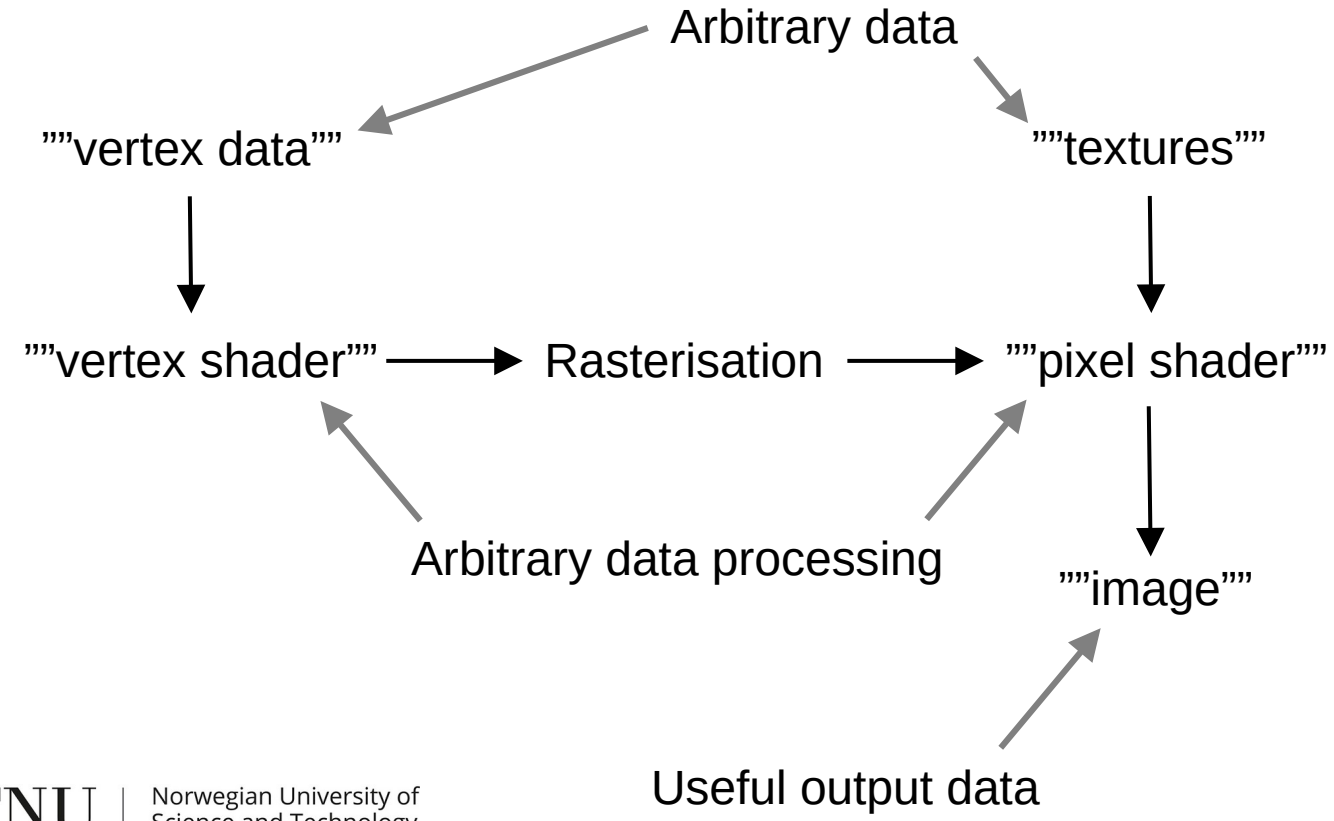
History of the GPU

Realisation: we can exploit this!



History of the GPU

Realisation: we can exploit this!



History of the GPU

2005: The idea of general purpose GPU computing

- Vertex and pixel calculations are independent. GPU's therefore had the ability to process multiple vertex and pixel shaders in parallel.
- For simple programs, exploiting this pipeline yielded much higher throughput than using a conventional CPU.
- Could greatly accelerate scientific computing, such as simulation and image processing.

OpenVIDIA: Parallel GPU Computer Vision

James Fung, Steve Mann, Chris Aimone
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
{fungja,mann}@eecg.toronto.edu, aimone@eyetap.org

ABSTRACT

Graphics and vision are approximate inverses of each other: ordinarily Graphics Processing Units (GPUs) are used to convert "numbers into pictures" (i.e. computer graphics). In this paper, we propose using GPUs in approximately the reverse way: to assist in "converting pictures into numbers" (i.e. computer vision). The OpenVIDIA project uses single or multiple graphics cards to accelerate image analysis and computer vision. It is a library and API aimed at providing a graphics hardware accelerated processing framework for image processing and computer vision. OpenVIDIA explores the creation of a parallel computer architecture consisting of multiple Graphics Processing Units (GPUs) built entirely from commodity hardware. OpenVIDIA uses multiple Graphics Processing Units in parallel to operate as a general-purpose parallel computer architecture. It provides a simple API which implements some common computer vision algorithms. Many components can be used immediately and because the project is Open Source, the code is intended to serve as templates and examples for how similar algorithms are mapped onto graphics hardware. Implemented are image processing techniques (Canny edge detection, filtering), image feature handling (identifying and matching features) and image registration, to name a few.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special–Purpose and Applications–Based Systems—*signal processing systems*; I.4.0 [**Image Processing and Computer Vision**]: General

General Terms

Algorithms, Performance, Design

Keywords

Computer vision, GPU, hardware accelerated computer vision, computer graphics, computer architecture, Radon Transform, OpenVIDIA, chirplet transform, mediated reality

MM'05, November 6–11, 2005, Singapore.

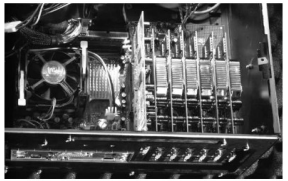


Figure 1: A computer vision machine with 6 PCI graphics cards, and 1 AGP graphics card. Each graphics card runs pattern recognition and computer vision tasks in parallel, creating a cheap, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.

1. INTRODUCTION

OpenVIDIA is a programming framework that uses single or multiple graphics cards to operate as a general-purpose parallel computer architecture for fast computer vision and image processing. Realtime image processing and computer vision algorithms can be computationally intensive, exceeding the capabilities of the CPU. OpenVIDIA utilizes the GPU that is present on modern graphics hardware to expand the computational resources that are available to these algorithms. Furthermore, the design of the GPU architecture provides higher performance for many vision algorithms than the CPU.

One may consider computer graphics and computer vision to be, in some way, "inverses" of one another since the task of graphics is the task of image synthesis whereas image processing can be considered the task of image analysis.

The OpenVIDIA project explores the notion of using computer graphics hardware "in reverse" to accelerate the computer vision task of image analysis even though graphics hardware was initially designed for rendering images, or image synthesis. Furthermore, OpenVIDIA explores a parallel "graphics for vision" architecture created by placing multi-

History of the GPU

2006: the General Purpose GPU (GPGPU)

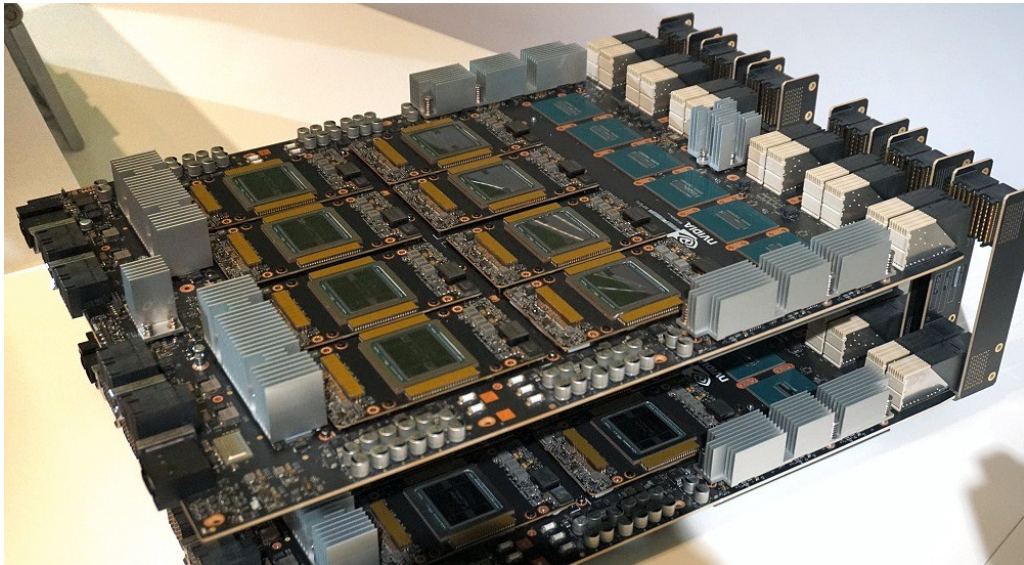
- The modern GPU that we use today
- A general purpose throughput oriented processor that can also render images
- Compute oriented cards are developed with the same GPU architecture but without any display outputs

History of the GPU

2007 – today: scaling up

- A number of GPU computing APIs are developed:
  - CUDA (dominates the market today)
  - OpenCL
  - Vulkan (topic of guest lecture)
  - ROCm
  - SYCL (OpenCL successor)

- Improvements to processing power, more memory capacity, and collaboration between GPUs. Also much higher power consumption.



In conclusion:

- The modern GPU arose because game developers wanted flexibility, which fixed-function hardware could not address
- GPU computing was invented by exploiting graphics hardware for unintended purposes
- Those unintended purposes are now a main driving force behind new GPU (and other accelerator) development

What is a GPU and how does it differ from a CPU?

- How did the GPU come to be?
- **How does the architecure of a GPU differ from a CPU?**

The CPU can be faster than the GPU

Professional sprinter:

+ Covers small distances super quickly
- Takes a while to run 1000 km

City Marathon:

- Individual people are kind of slow
+ Combined distance of runners sums up to 1000km very quickly

The GPU has a number of performance “gotchas”

- Need to understand some intricate parts of its architecture to achieve optimal performance

How does the GPU architecture differ from the CPU?

Let’s start with some specifications and see what each processor type can do

Source: Blender Demo Files, Agent 327 Barbershop

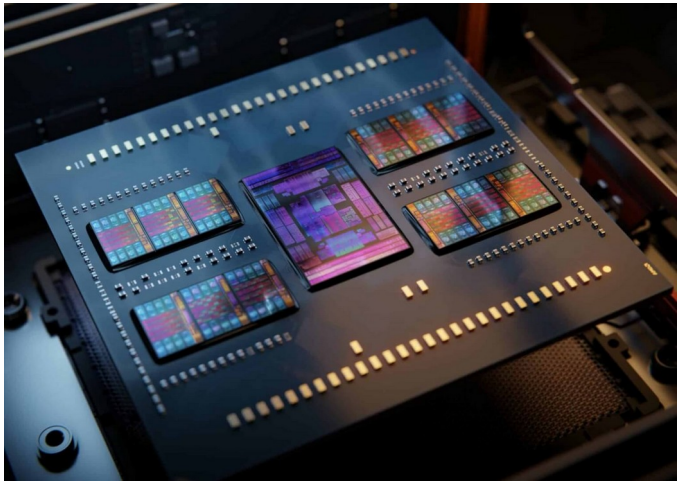
How does the GPU architecture differ from the CPU?

Let’s start with some specifications..

Contender 1: AMD Ryzen 9 7950X

Cores: 16 (32 threads)
Render time: 571 seconds
Power usage: 234W

Contender 2: AMD Epyc 9754



Cores: 128 (256 threads)
Render time: 70 seconds (2 processors)
Power usage: 385W

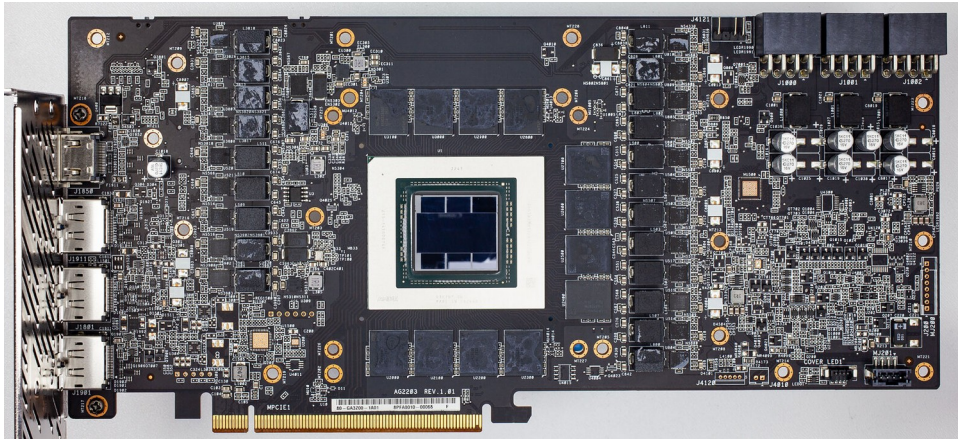
How does the GPU architecture differ from the CPU?

Let’s start with some specifications..

Contender 3: Nvidia GeForce RTX 4090

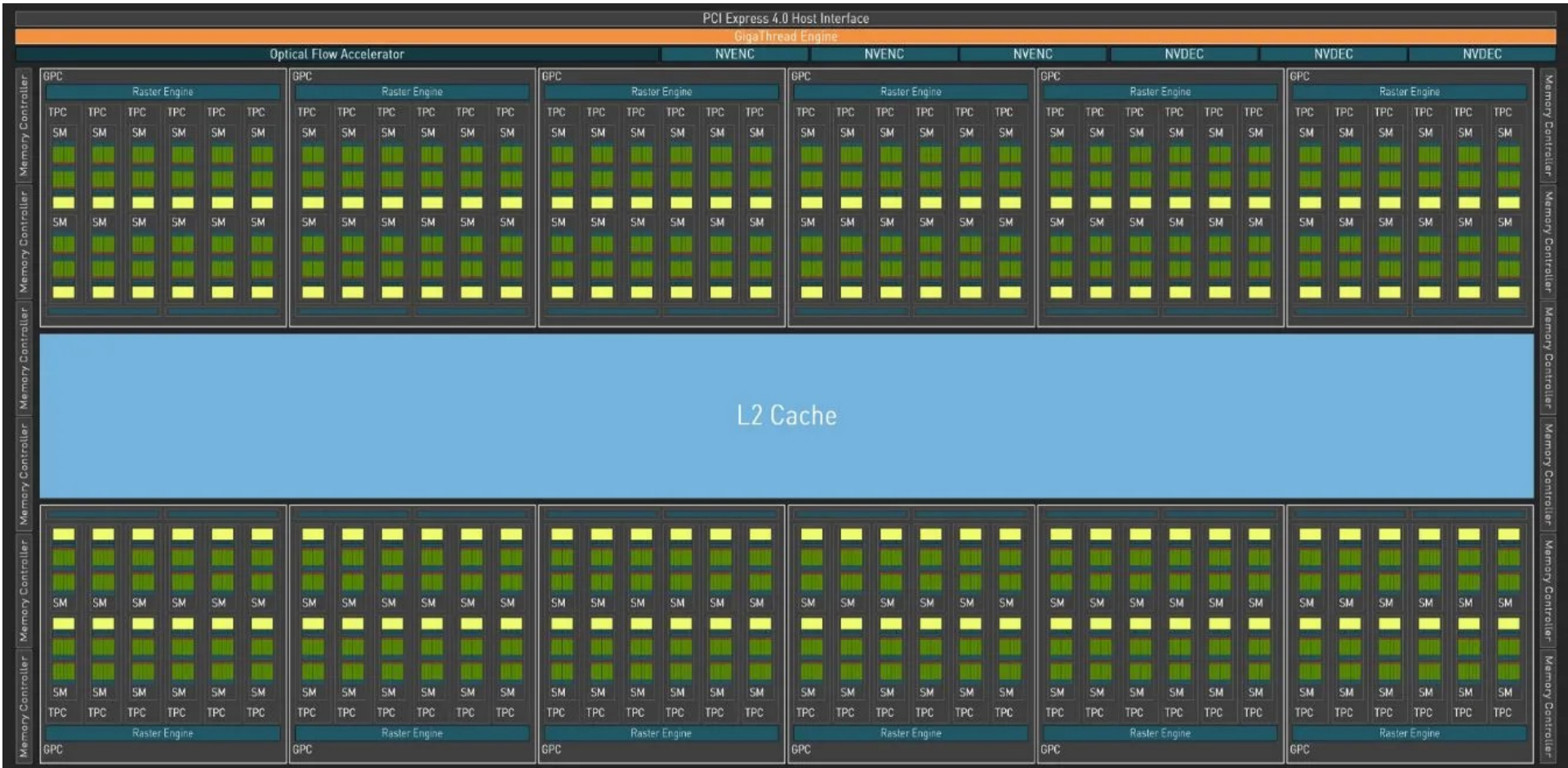
Cores: 16,384
Render time: 9.71 seconds
Power usage: 483W

Contender 4: AMD RX 7900 XTX



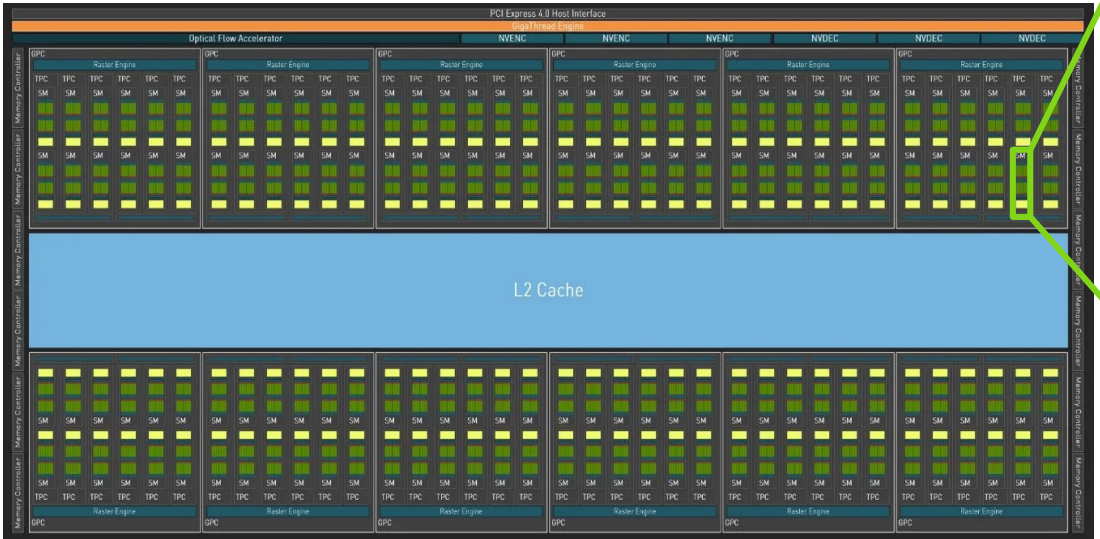
Cores: 6,144
Render time: 21 seconds
Power usage: 320W

1000’s of cores?? How is that possible?



The closest thing a GPU has to a CPU core is the Streaming Multiprocessor (SM)

Each SM has 128 32-bit floating point units



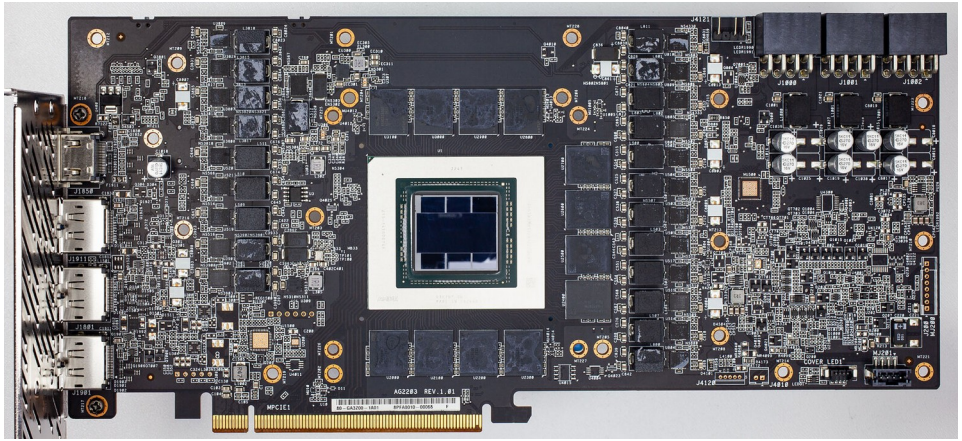
How does the GPU architecture differ from the CPU?

Let’s start with some specifications..

Contender 3: Nvidia GeForce RTX 4090

GPU Cores: 128
Render time: 9.71 seconds
Power usage: 483W

Contender 4: AMD RX 7900 XTX



GPU Cores: 96
Render time: 21 seconds
Power usage: 320W

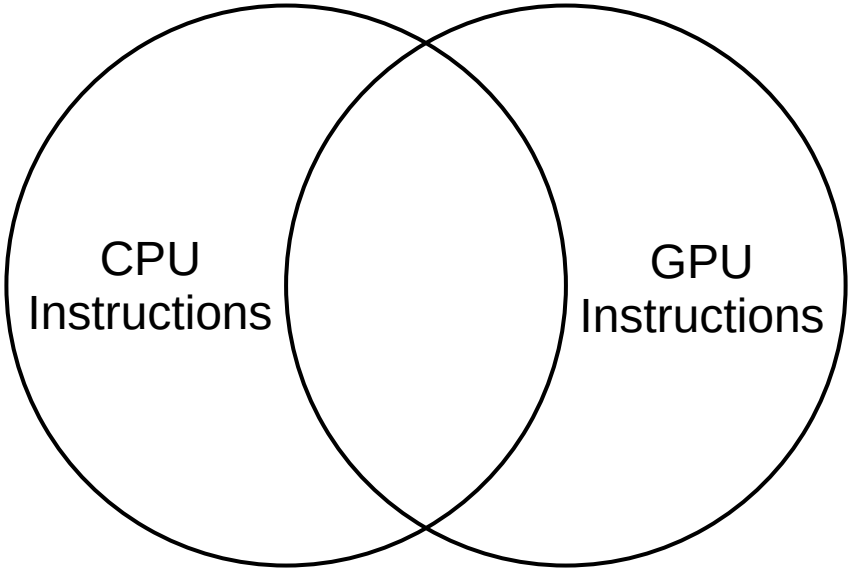
Let’s now take a closer look at how the SM works!

Notes:

- Information and terminology focuses on nvidia, but concepts are effectively vendor agnostic
- Nvidia does not reveal many details about its architecture. Some details are a «best guess» because they are simply not known
- Architecture has changed over time. We’ll be discussing the latest version today (codename Ada Lovelace)

Main similarities between the CPU and GPU:

- Both are general purpose processor cores
- Both use a Superscalar Architecture



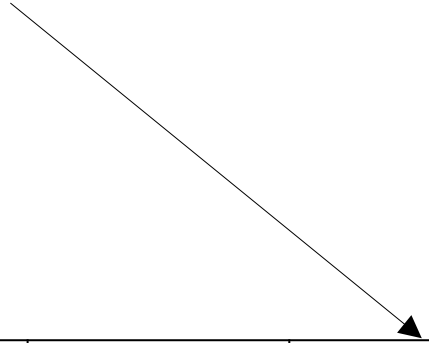
Reminder: Out of Order CPU design

Thread
inst 2
inst 1
inst 0

Port 0	Port 1	Port 2/3	Port 4/9	Port 7/8	Port 5	Port 6
Integer Floating Point 256-bit vector ALU mul/div Branch	Integer Floating Point ALU	Load	Store	Store Address	Integer 512-bit vector	Integer ALU Branch

Reminder: Out of Order CPU design

Thread
inst 2
inst 1
inst 0



Port 0	Port 1	Port 2/3	Port 4/9	Port 7/8	Port 5	Port 6
Integer Floating Point 256-bit vector ALU mul/div Branch	Integer Floating Point ALU	Load	Store	Store Address	Integer 512-bit vector	Integer ALU Branch

Reminder: Out of Order CPU design

Thread
inst 2
inst 1
inst 0

CPU can execute other instructions in the meantime

Blocked while the instruction is executing

Port 0	Port 1	Port 2/3	Port 4/9	Port 7/8	Port 5	Port 6
Integer Floating Point 256-bit vector ALU mul/div Branch	Integer Floating Point ALU	Load	Store	Store Address	Integer 512-bit vector	Integer ALU Branch

Reminder: Out of Order CPU design

Thread A	Thread B
inst 2	inst 2
inst 1	inst 1
inst 0	inst 0

Hyperthreading /
Simultaneous Multithreading:

Multiple independent threads
sharing the same execution ports

Port 0	Port 1	Port 2/3	Port 4/9	Port 7/8	Port 5	Port 6
Integer Floating Point 256-bit vector ALU mul/div Branch	Integer Floating Point ALU	Load	Store	Store Address	Integer 512-bit vector	Integer ALU Branch

GPU Core architecture

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	...	Thread 2047
inst 2	inst 2	inst 2	inst 2	inst 2	inst 2		inst 2
inst 1	inst 1	inst 1	inst 1	inst 1	inst 1		inst 1
inst 0	inst 0	inst 0	inst 0	inst 0	inst 0		inst 0

4x per SM

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Floating Point (32-bit) Integer (32-bit)	Floating Point (32-bit)	Load Store	Low-Precision Matrix (aka tensor core)	Special Functions Unit (SFU)	Ray Tracing Acceleration	Rasterisation Texture Unit

* Port numbering and layout not specified by nvidia

GPU Core architecture

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread
inst 2	inst 2	inst 2	inst 2	inst 2	inst 2
inst 1	inst 1	inst 1	inst 1	inst 1	inst 1
inst 0	inst 0	inst 0	inst 0	inst 0	inst 0

4x per SM

Port 0	Port 1	Port 2	Port 3	Port 4
Floating Point (32-bit) Integer (32-bit)	Floating Point (32-bit)	Load Store	Low-Precision Matrix (aka tensor core)	Special Functions Unit (SFU)

* Port numbering and layout not specified by nvidia

What about their differences?

- Similarities:
  - Both are general purpose processors
  - Both use a superscalar execution architecture

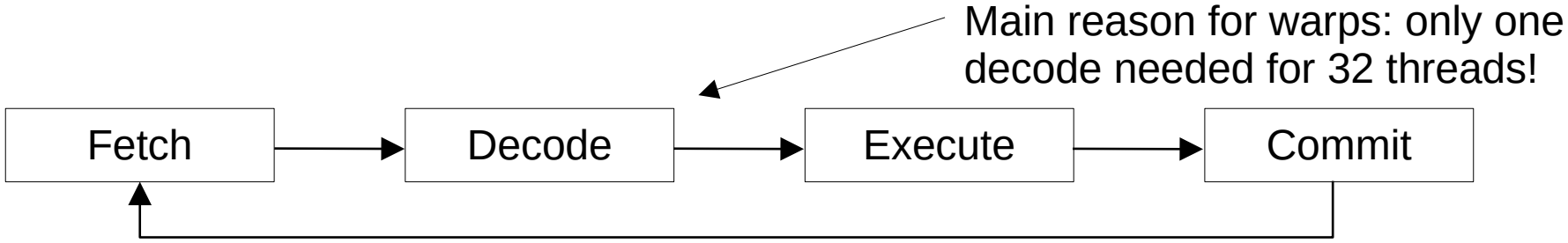
- Differences:
  - Many. They will become very apparent when we discuss the architecture in detail

GPU Architecture

- **Thread execution (warps)**

Thread execution: warps

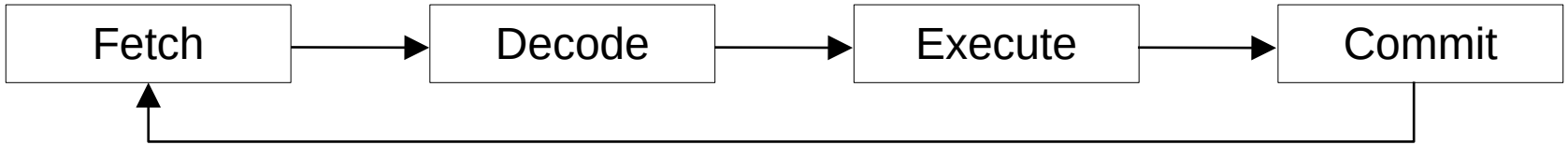
- Each individual thread is executed in order
- A CPU thread is in principle not different from a GPU thread
- Threads are executed in groups of 32 threads* called “warps”
  - Threads in a warp all execute the exact same instructions
  - Grouped execution greatly simplifies core design
  - GPUs are designed to do the same thing over and over again, so this is a reasonable restriction
  - Also has advantages (topic for next week)



*64 on AMD, variable on Intel

Thread execution: warps

Instruction	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	...	T_{31}
`a = input1[i];`	`a = 3`	`a = 2`	`a = 30`	`a = 17`	`a = 7`	`a = 11`	`a = 15`	`a = 21`		`a = 25`
`b = input2[i];`	`b = 8`	`b = 11`	`b = 6`	`b = 20`	`b = 21`	`b = 28`	`b = 14`	`b = 4`		`b = 6`
`c = a + b;`	`c = 11`	`c = 13`	`c = 36`	`c = 37`	`c = 28`	`c = 39`	`c = 29`	`c = 25`		`c = 31`
`c = c * 2;`	`c = 22`	`c = 26`	`c = 72`	`c = 74`	`c = 56`	`c = 78`	`c = 58`	`c = 50`		`c = 62`
`c = c – b;`	`c = 14`	`c = 15`	`c = 66`	`c = 54`	`c = 35`	`c = 50`	`c = 44`	`c = 46`		`c = 56`



GPU Architecture

- Thread execution (warps)
- **Branches in threads**

Thread Divergence

- When a branch occurs (if statement or loop), all threads in the warp must participate, *even if they are not part of that execution path*
    - For if statements: if even one thread chooses the if or else clause, all threads in the warp must participate in executing that path
    - For loops, all threads in the warp must participate until the final thread has finished iterating

Thread divergence

Instruction	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	...	T_{31}
`a = input1[i];`	`a = 3`	`a = 2`	`a = 30`	`a = 17`	`a = 7`	`a = 11`	`a = 15`	`a = 21`		`a = 25`
`b = input2[i];`	`b = 8`	`b = 11`	`b = 6`	`b = 20`	`b = 21`	`b = 28`	`b = 14`	`b = 4`		`b = 6`
`c = a + b;`	`c = 11`	`c = 13`	`c = 36`	`c = 37`	`c = 28`	`c = 39`	`c = 29`	`c = 25`		`c = 31`
`c = c * 2;`	`c = 22`	`c = 26`	`c = 72`	`c = 74`	`c = 56`	`c = 78`	`c = 58`	`c = 50`		`c = 62`
`c = c – b;`	`c = 14`	`c = 15`	`c = 66`	`c = 54`	`c = 35`	`c = 50`	`c = 44`	`c = 46`		`c = 56`
`if(c >= 50) {`	`false`	`false`	`true`	`true`	`false`	`true`	`false`	`false`		`true`
`a = 5;`	`N/A`	`N/A`	`a = 5`	`a = 5`	`N/A`	`a = 5`	`N/A`	`N/A`		`a = 5`
`} else {`										
`a = 10;`	`a = 10`	`a = 10`	`N/A`	`N/A`	`a = 10`	`N/A`	`a = 10`	`a = 10`		`N/A`
`}`										
`c = c – a;`	`c = 4`	`c = 5`	`c = 61`	`c = 49`	`c = 25`	`c = 45`	`c = 34`	`c = 36`		`c = 51`

N/A: thread participates in the warp, but does not execute the instruction

Thread divergence

- Thread divergence is only a problem if threads are following different execution paths
  - Avoid long if/else statements where threads are likely to choose different paths
  - Avoid loops with a highly variable number of iterations

- Tips if these are unavoidable:
  - Branches can often be replaced with maths
  - Rewriting multiple nested loops as a single loop

→ An algorithm which runs well on the CPU does not necessarily do so on the GPU

Flynn’s taxonomy

- The warp-based execution model is a subclass of SIMD in Flynn’s taxonomy, called Single Instruction Multiple Thread (SIMT).

- In many ways similar to SIMD:
  - SIMD: A single instruction executes multiple data operations in a single thread
  - SIMT: A single instruction executes a single data operation in multiple threads

GPU Architecture

- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- **Context switching**

GPU context switching

- Context switching: time sharing a processor by switching out which thread is executing on it
- How does the CPU do a context switch?

GPU context switching

- Context switching: time sharing a processor by switching out which thread is executing on it

- How the CPU does a context switch:

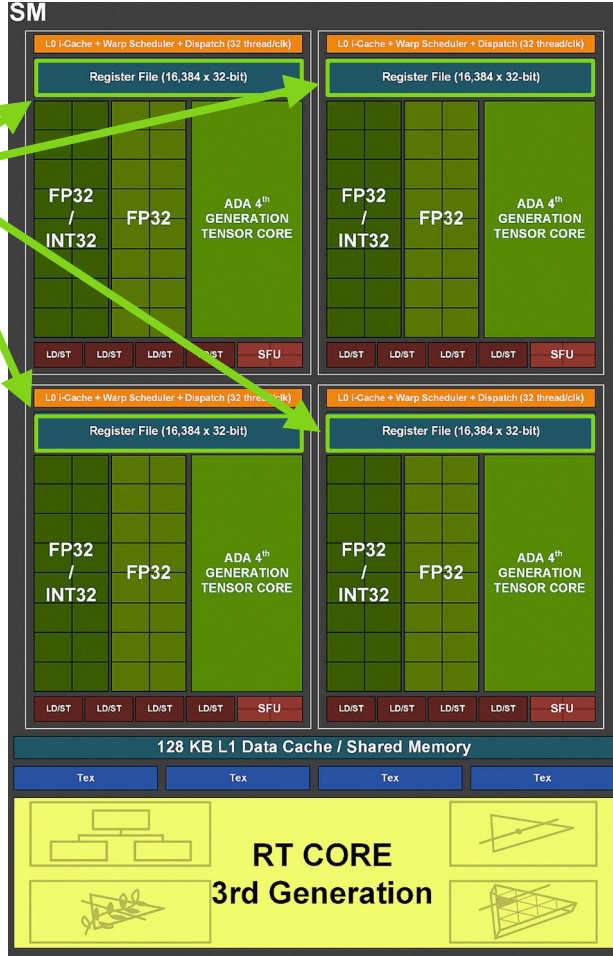
    1. Store the Program Counter (PC)
    2. Store all registers in stack memory
    3. Load registers from new thread from stack
    4. Jump to PC of new thread
    5. Continue executing new thread

- Usually takes many cycles to perform

    - Typically incurs a speed penalty because the cache(s) are usually not set up for the new thread.

GPU Context Switching

- A GPU does a context switch Every Single Clock Cycle
- Register files store all registers of all threads
  - No need to swap registers when they are already there anyway
  - Each SM can store 65,536 registers.
    With a limit of 2048 threads per SM, that is 32 registers / thread on average
- Each cycle, 4 warps are chosen that are able to execute an instruction and assigned to an available port capable of executing that instruction.



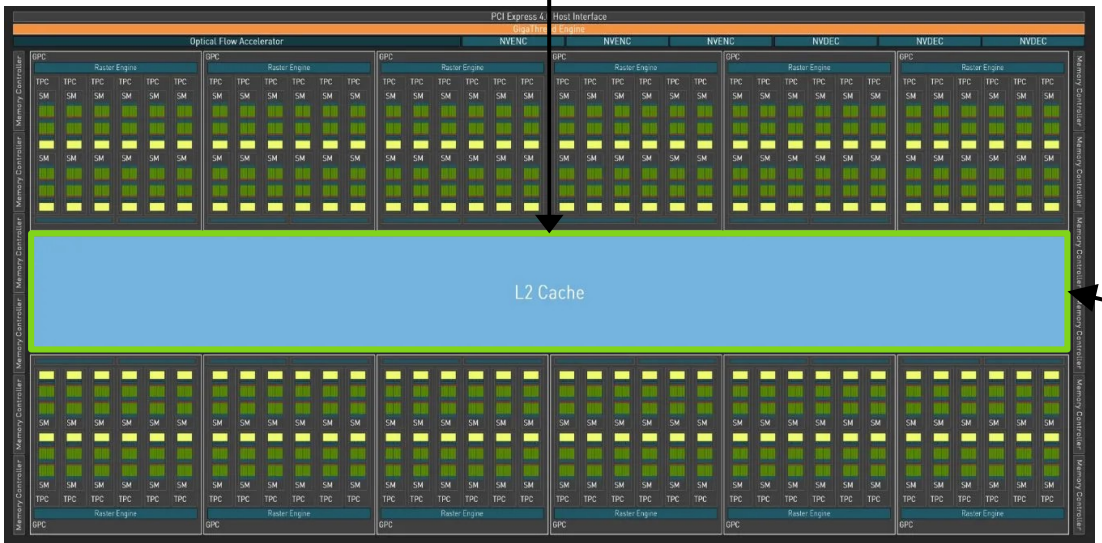
GPU Architecture

- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- Context switching (done each cycle)
- **Why fast thread switching?**

The Memory System

- The GPU has a shared L2 cache
- Each SM has its own L1 cache and a separate instruction cache
- VRAM uses bandwidth-optimised GDDR

VRAM (GDDR6X)



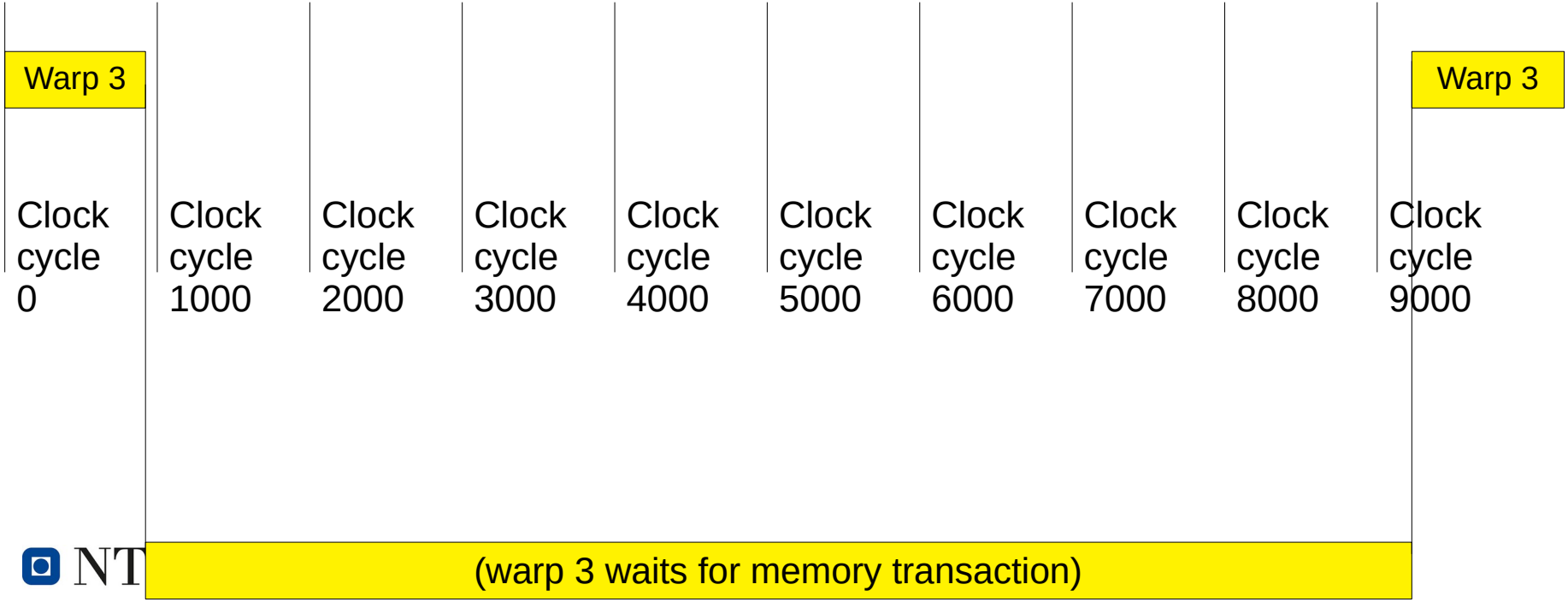
The Memory System

- GDDR tries to maximise bandwidth
  - Regular DDR also aims for low latency
- **The Big Kicker™: latency is terrible**

  And there are 1000’s of threads trying to access it..

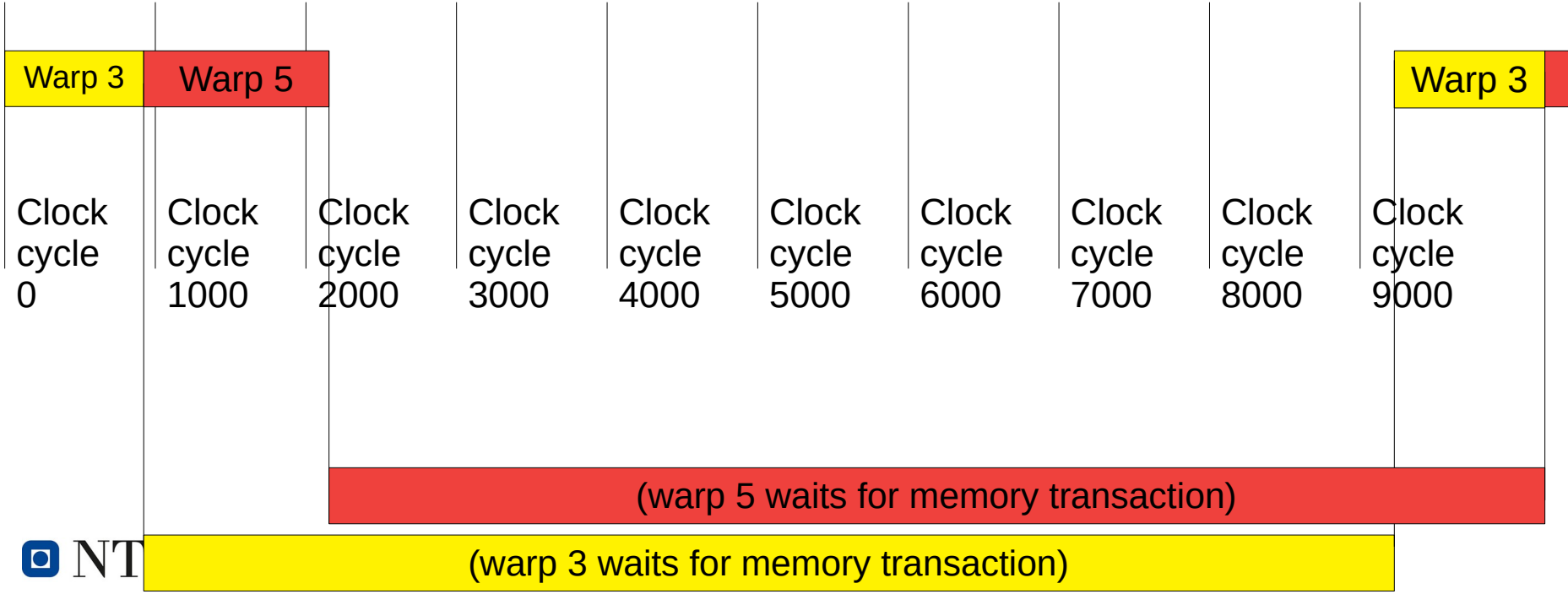
Problem: Warps stall. A lot.

If we were to execute threads as if on a CPU:

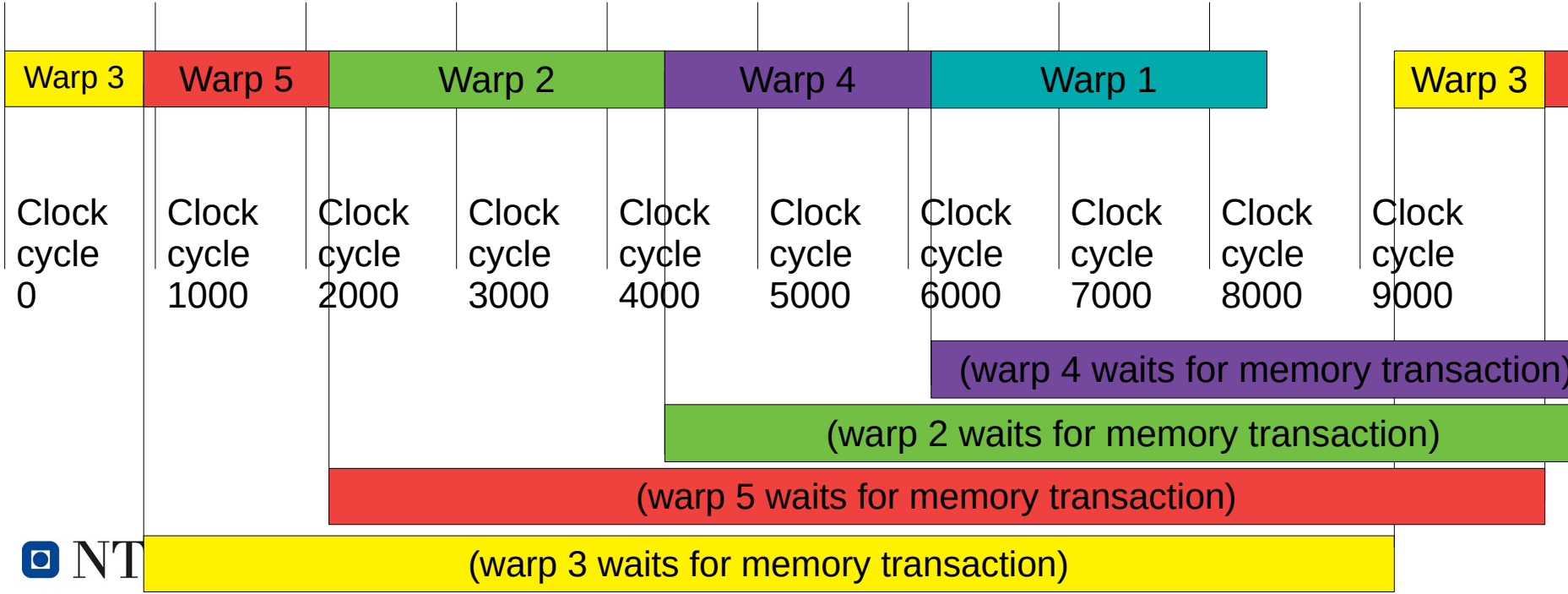


NT

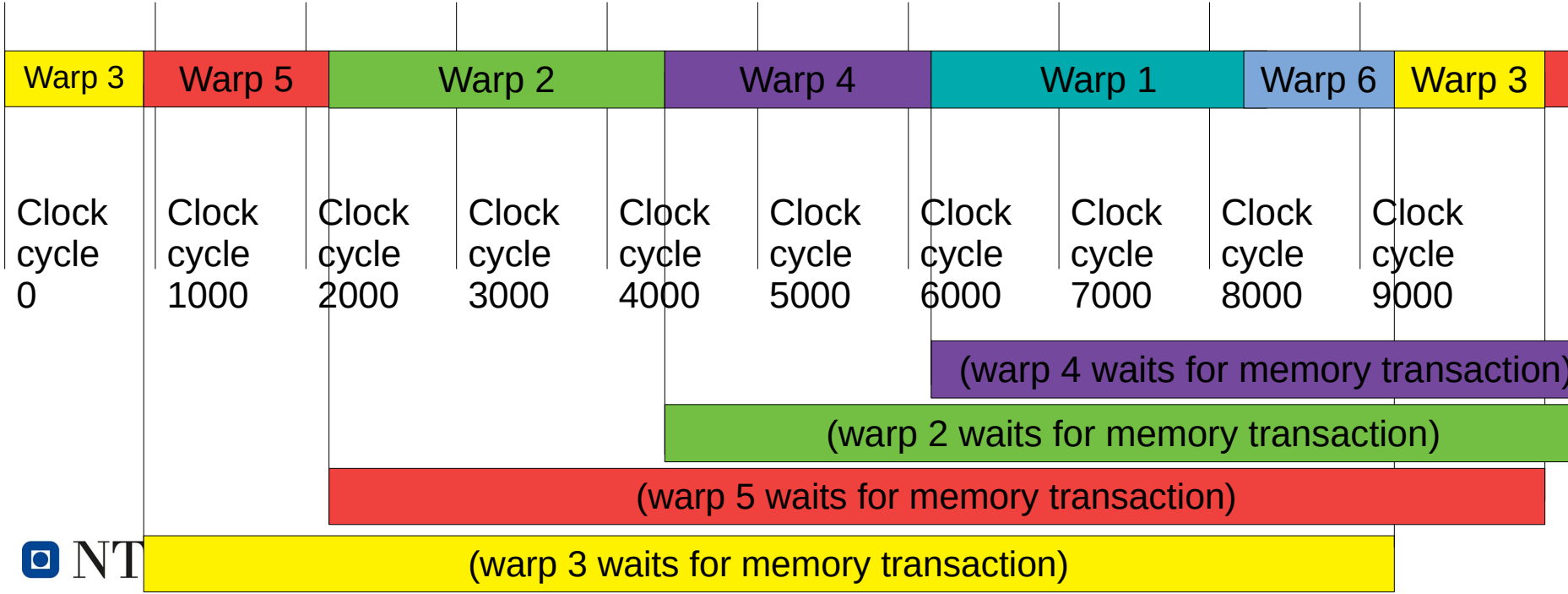
Solution: execute other threads in the meantime



Solution: execute other threads in the meantime



Result: Since an SM is ideally always executing code, ***memory latency is hidden!***



Occupancy: measure for GPU utilisation

$$Occupancy = \frac{Cycles\ SM\ busy}{Total\ Cycles}$$

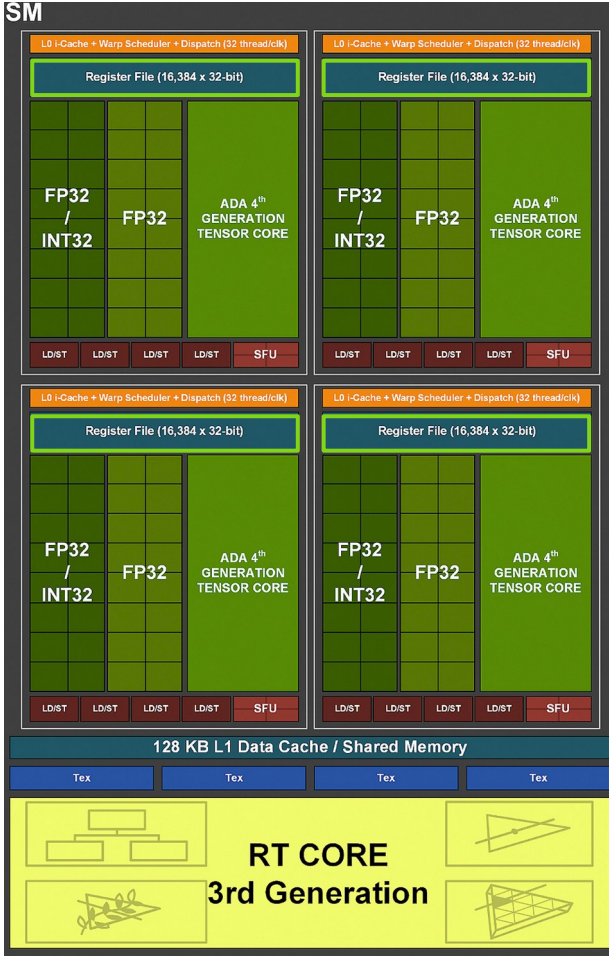
GPU Architecture

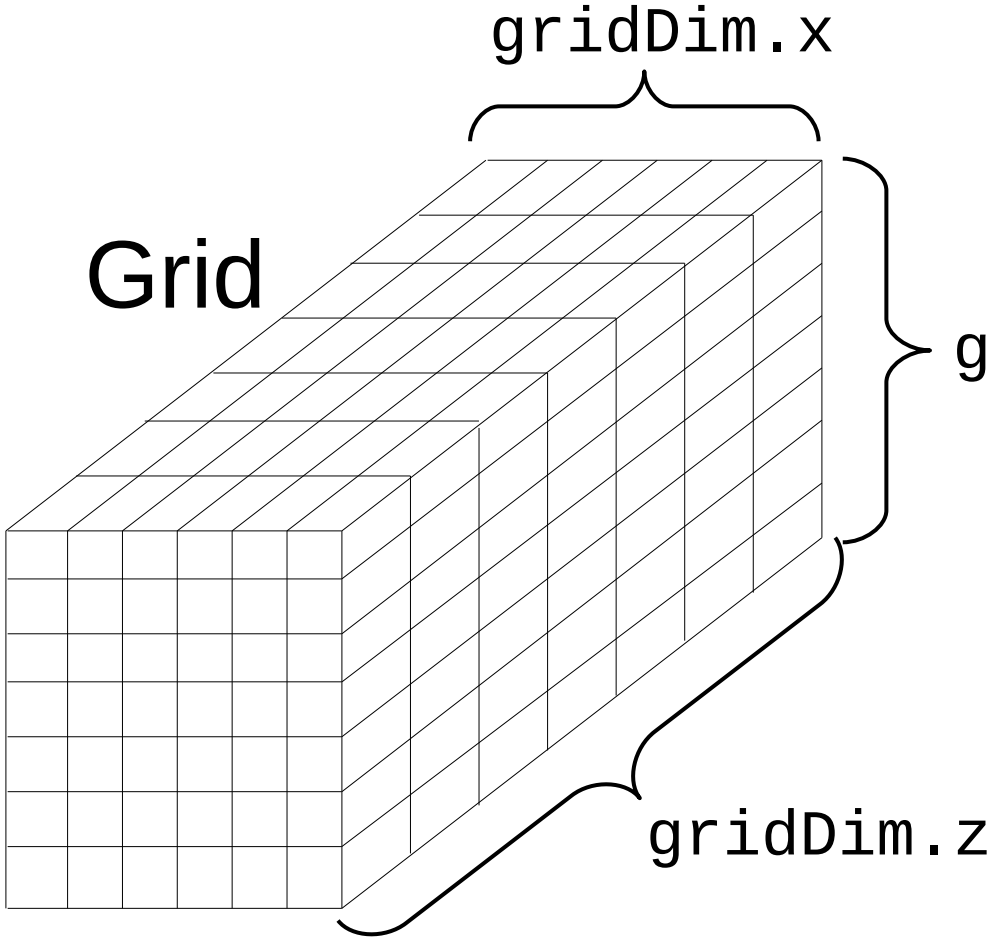
- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- Context switching (done each cycle)
- Latency hiding
- **Thread hierarchy**

Thread Hierarchy

- An SM can only have a certain number of threads active at the same time (usually 2048)
- Our problems are usually much larger, and we tend to use many threads

→ Solution: create a thread hierarchy

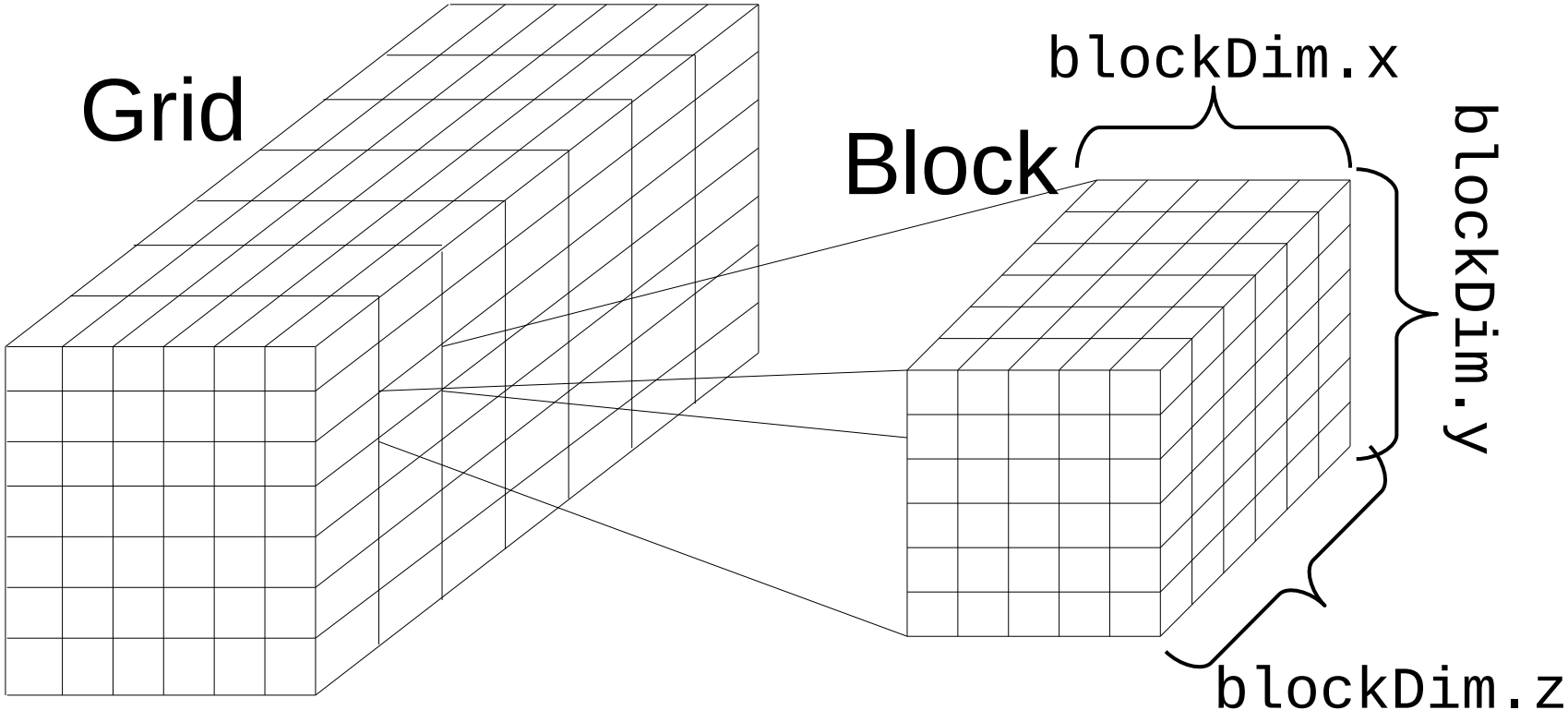




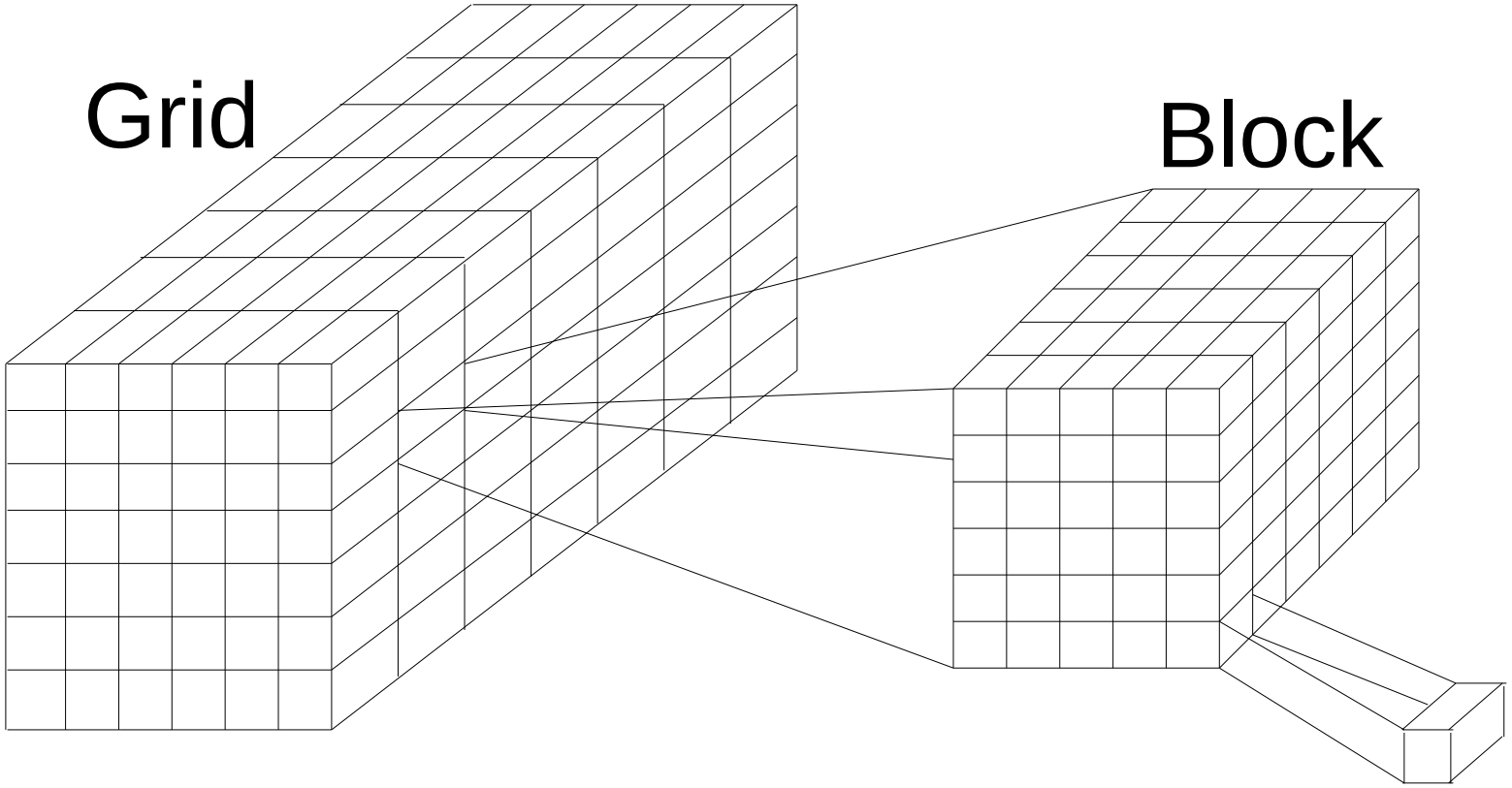
The grid represents all threads that we would like to run

Example: one thread per pixel in an image

A block is a chunk of our grid that is small enough to fit within an SM



A block consists of a number of threads



Which of these is better?

A

```
dim3 blockSize1(5, 5, 4);
dim3 gridSize1(800, 10, 10);
someKernel<<<gridSize1, blockSize1>>>();
```

B

```
dim3 blockSize2(10, 8, 2);
dim3 gridSize2(250, 100, 1);
someKernel<<<gridSize2, blockSize2>>>();
```

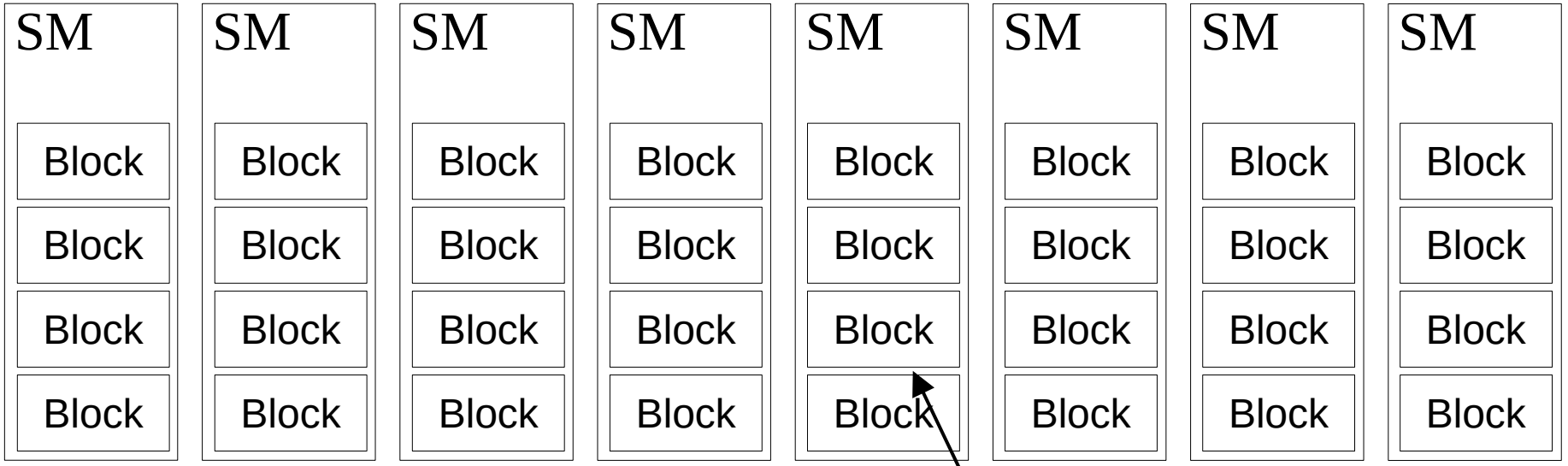
SM	SM	SM	SM	SM	SM	SM	SM
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block

Grid represents a queue of blocks

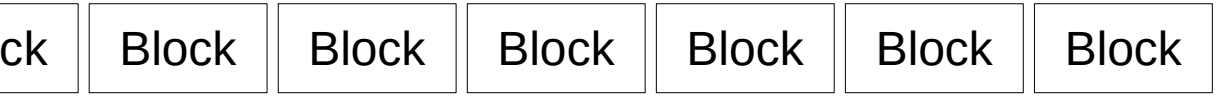
SM	SM	SM	SM	SM	SM	SM	SM
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block		Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block

Grid represents a queue of blocks

ck Block Block Block Block Block Block Block



Grid represents a queue of blocks



SM	SM	SM	SM	SM	SM	SM	SM
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block
Block	Block	Block	Block	Block	Block	Block	Block

Grid represents a queue of blocks

Block	Block	Block	Block	Block	Block	Block

Thread Hierarchy

- Launching a kernel requires specifying the dimensions of the grid and blocks within
- When the kernel is launched, a queue of blocks is created.
- Blocks run until all threads within have finished
- The order in which blocks are run is not defined
- Once a block is finished, the next one in the queue is allocated to that SM
- Blocks should be dimensioned such that they contain a multiple of 32 threads

GPU Architecture

- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- Context switching (done each cycle)
- Latency hiding
- Thread hierarchy
- **Thread interaction**

Thread Interaction

- Interaction between threads within the same warp is cheap, and easy to do within the same block
  - Allows for some of the coolest features of the GPU

- Most common: `__syncthreads();`
  Barrier for all threads in the block

- You can and should not try to synchronise threads across blocks. Launch multiple kernels instead.

Thread Interaction: atomics

- The GPU does not have mutexes.
- The only mechanism for avoiding race conditions are atomic instructions

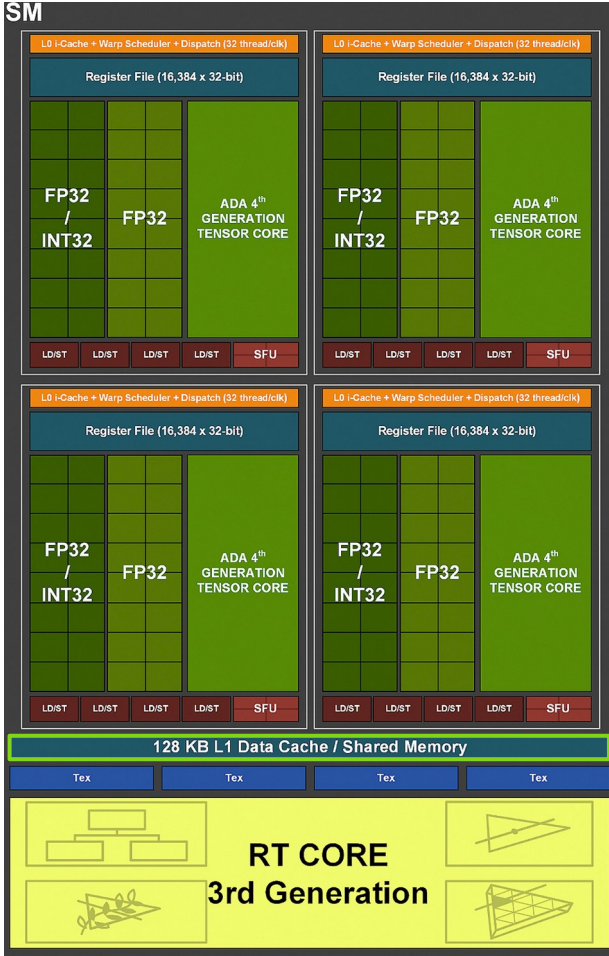
```
int atomicAdd(int* address, int val);
int atomicSub(int* address, int val);
int atomicExch(int* address, int val);
int atomicMin(int* address, int val);
int atomicMax(int* address, int val);
unsigned int atomicInc(..);
unsigned int atomicDec(..);
int atomicCAS(int* address, int compare, int val);
int atomicAnd(int* address, int val);
int atomicOr(int* address, int val);
int atomicXor(int* address, int val);
```

GPU Architecture

- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- Context switching (done each cycle)
- Latency hiding
- Thread hierarchy
- Thread interaction
- **User-manageable L1 cache**

Shared Memory

- Each SM has its own L1 cache (128 KB on Lovelace)
- You can use a part of the L1 cache to use as temporary storage, known as “shared memory”
  - Much faster than main memory (~10x)
- Useful if you know you will use a piece of data many times (e.g. a histogram)



GPU Architecture

- Thread execution (warps of 32 threads)
- Thread divergence (avoid branches)
- Context switching (done each cycle)
- Latency hiding
- Thread hierarchy
- Thread interaction

Tomorrow

- Cooperative groups
- Profiling of CUDA kernels

https://jakevdp.github.io/images/mario_pattern_background.png
https://www.8-bitcentral.com/images/reviews/atari2600/fatalRun2600Screen.jpg
https://external-preview.redd.it/7UaFyYPXHTW2c_f84IT-FI72jCG7Xy6VvB13dRB_YFI.jpg?auto=webp&s=04f1399d8bbafc712d7422bf0c445ff59afa292a
https://upload.wikimedia.org/wikipedia/en/6/69/Wolf3d_pc.png
www.loadthegame.com/wp-content/uploads/2014/09/quake-live-steam-launch.jpg
http://www.vraymaterials.co.uk/tutorials/making-of-gt3-race-scene/
http://img.cadnav.com/allimg/170707/cadnav-1FFG43U3.jpg
By Swaaye at the English-language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=76420413
http://www.emu-france.com/wp-content/uploads/2014/05/glquake.jpg
https://assets.hardwarezone.com/img/2018/06/HGX-2_4.jpg
https://www.allround-pc.com/wp-content/uploads/2020/08/DSC02464-scaled.jpg
https://www.bhphotovideo.com/images/images2500x2500/apple_mj2x2ll_a_watch_sport_smartwatch_38mm_1187194.jpg
https://www.bhphotovideo.com/images/images2500x2500/toshiba_pscpsu_009007_15_6_satellite_c55_series_1222345.jpg
https://cdn.hobbyconsolas.com/sites/navi.axelspringer.es/public/media/image/2014/04/319838-20-mejores-juegos-atari-2600.jpg