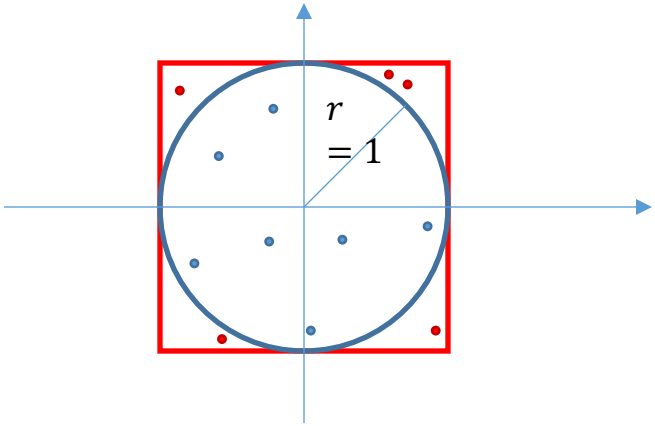
Bevezetés a GPU-k programozásába

Berényi Dániel
2026 február 26.

Motivation

- Let's compute π in python and C++, and measure how much time it takes.

- We will use Monte-Carlo integration for it by generating random (x, y) pairs on a square of edge length 2, and counting the ratio of those that fall into the circle of radius 1 to all points:



Motivation

- We try to be comparable and write the same simple code, use the same random generators, etc:

MC.py

```python
import numpy as np
import time

from numpy.random import Generator, MT19937, SeedSequence
sg = SeedSequence(1234)

def calc_pi():
    N = 1_000_000
    accum = 0.0
    seedseq = np.random.SeedSequence(1234)

    # https://numpy.org/devdocs/reference/random/bit_generators/mt19937.html
    generator = np.random.Generator(np.random.MT19937(seedseq))
    for i in range(N):
        x = generator.uniform(-1.0, 1.0)
        y = generator.uniform(-1.0, 1.0)
        if x*x + y*y < 1.0:
            accum += 1
    measure = 2.0 * 2.0
    result = measure * accum/float(N)
    return result

if __name__ == "__main__":
    t0 = time.time_ns()
    res = calc_pi()
    t1 = time.time_ns()
    print("Result = ", res)
    print("Time   = ", (t1-t0)/1000000.0, " ms")
```

mc.cpp

```cpp
#include <chrono>       // high_resolution_clock
#include <iostream>     // std::cout
#include <random>       // std::mt19937, std::uniform_real_distribution

using floating_millisecond = std::chrono::duration<double, std::milli>;

auto calc_pi = []() {
    const int N = 1'000'000;
    double accum = 0.0;
    std::seed_seq seedseq{1234};

    auto distribution = std::uniform_real_distribution<double>{-1.0, 1.0};
    auto generator = std::mt19937{seedseq};
    for(int i=0; i<N; ++i) {
        double x = distribution(generator);
        double y = distribution(generator);
        if(x*x + y*y < 1.0)
            accum += 1.0; }
    const double measure = 2.0 * 2.0;
    const double result = measure * accum / static_cast<double>(N);
    return result; };

int main() {
    auto t0 = std::chrono::high_resolution_clock::now();
    auto res = calc_pi();
    auto t1 = std::chrono::high_resolution_clock::now();
    std::cout << "Result = " << res << std::endl;
    std::cout << "Time   = " << floating_millisecond{t1-t0}.count() << " ms" << std::endl; }
```

PROBLÉMÁK KIMENET HIBAKERESÉSI KONZOL TERMINÁL

```
PS C:\Users\mate\Source\MC> python3.10.exe .\MC.py
Result =  3.142404
Time   =  2485.0782  ms
PS C:\Users\mate\Source\MC>
```

```
PS C:\Users\mate\Source\MC> .\.vscode\build\Release\MC-PI.exe
Result = 3.13954
Time   = 16.8292 ms
PS C:\Users\mate\Source\MC>
```

Motivation

Results: Million Samples per second (larger is better):

Technology	Performance [million samples / sec]
Naïve python	0.4
Smarter python	370
Naïve C++	770

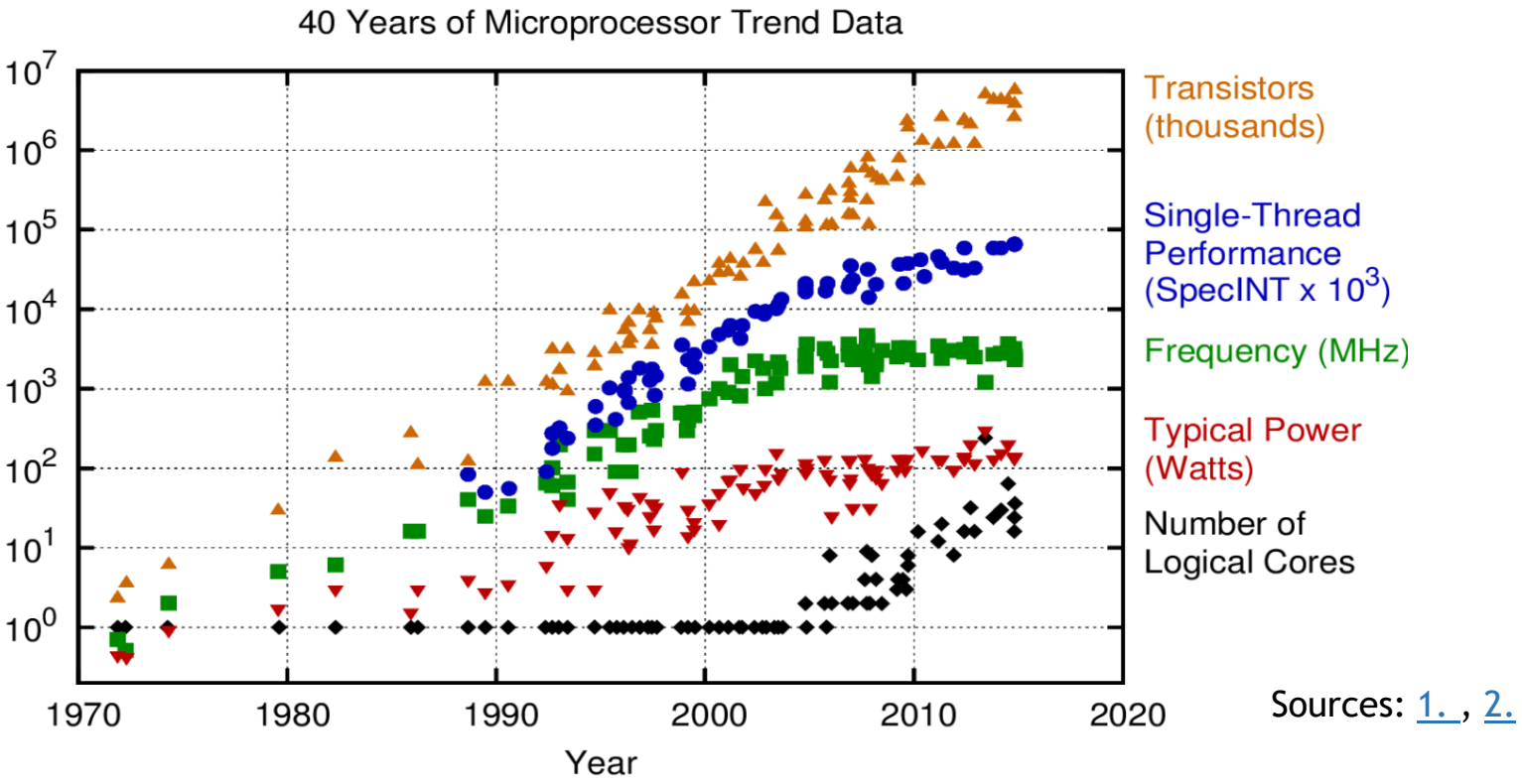
Motivation

Results: Million Samples per second (larger is better):

Technology	Performance [million samples / sec]
Naïve python	0.4
Smarter python	370
Naïve C++	770
OpenCL CPU	6100
OpenCL IGP	13420
OpenCL dGPU	38450

Hardware: CPU: Ryzen 7 6800HS, IGP: Radeon 680M, dGPU: Radeon 6800S

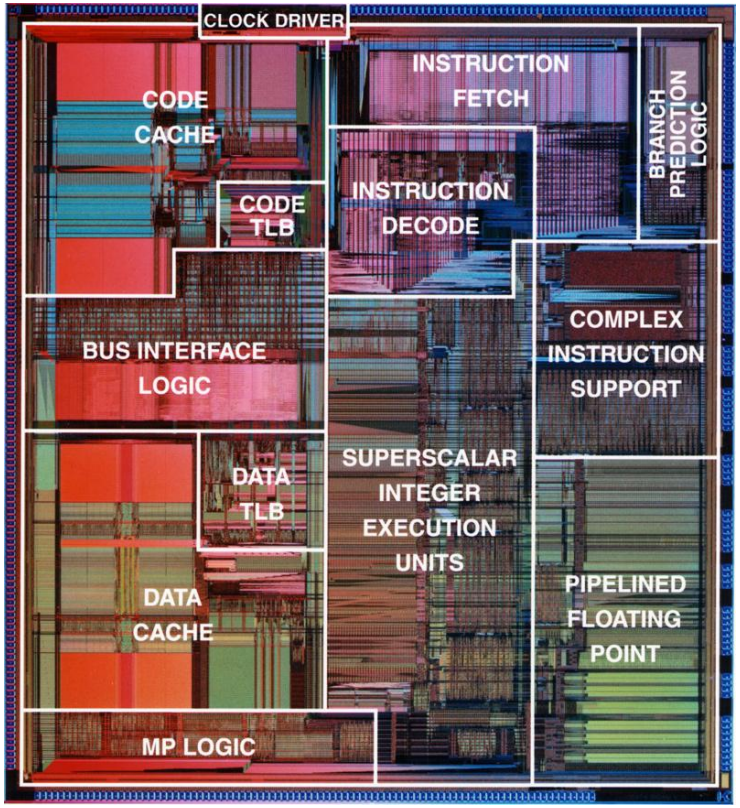
Introduction: parallelism



Sources: 1. , 2.

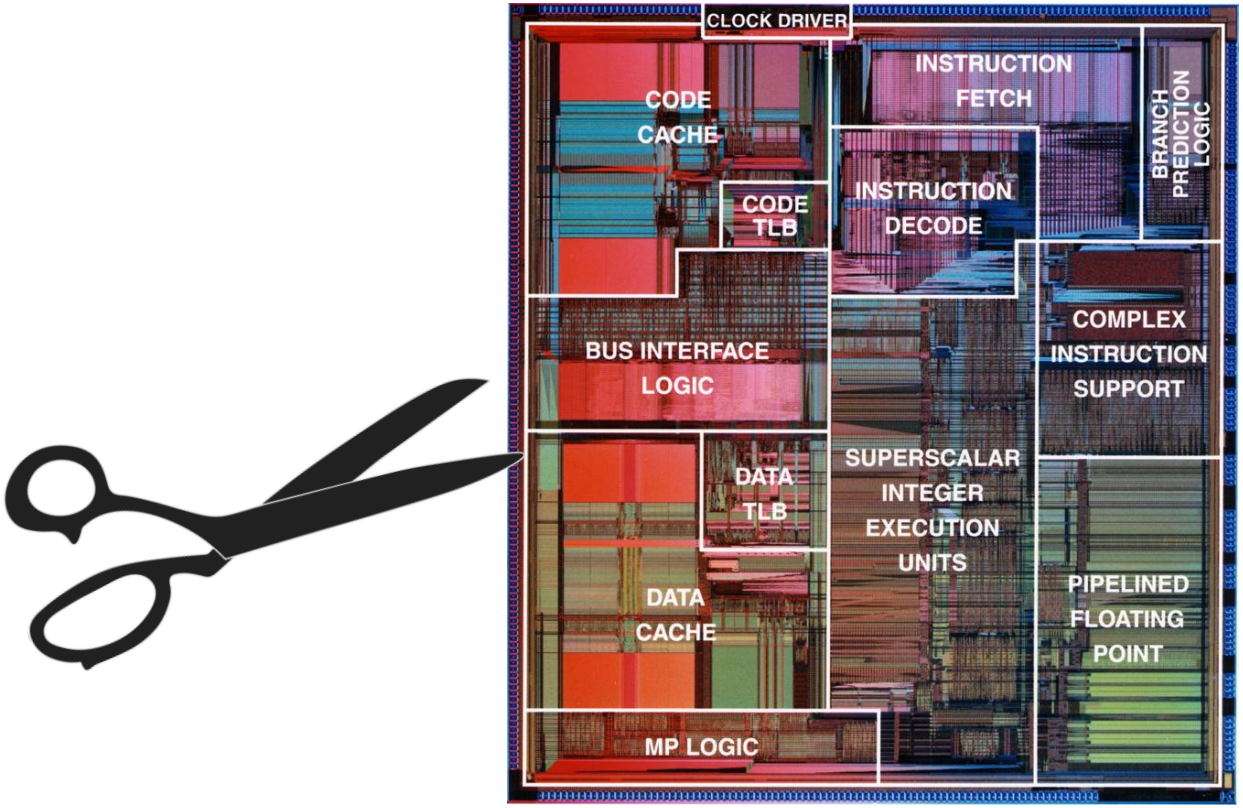
In a dream…

Let’s look at this Intel Pentium annotated die shot:
(1993)

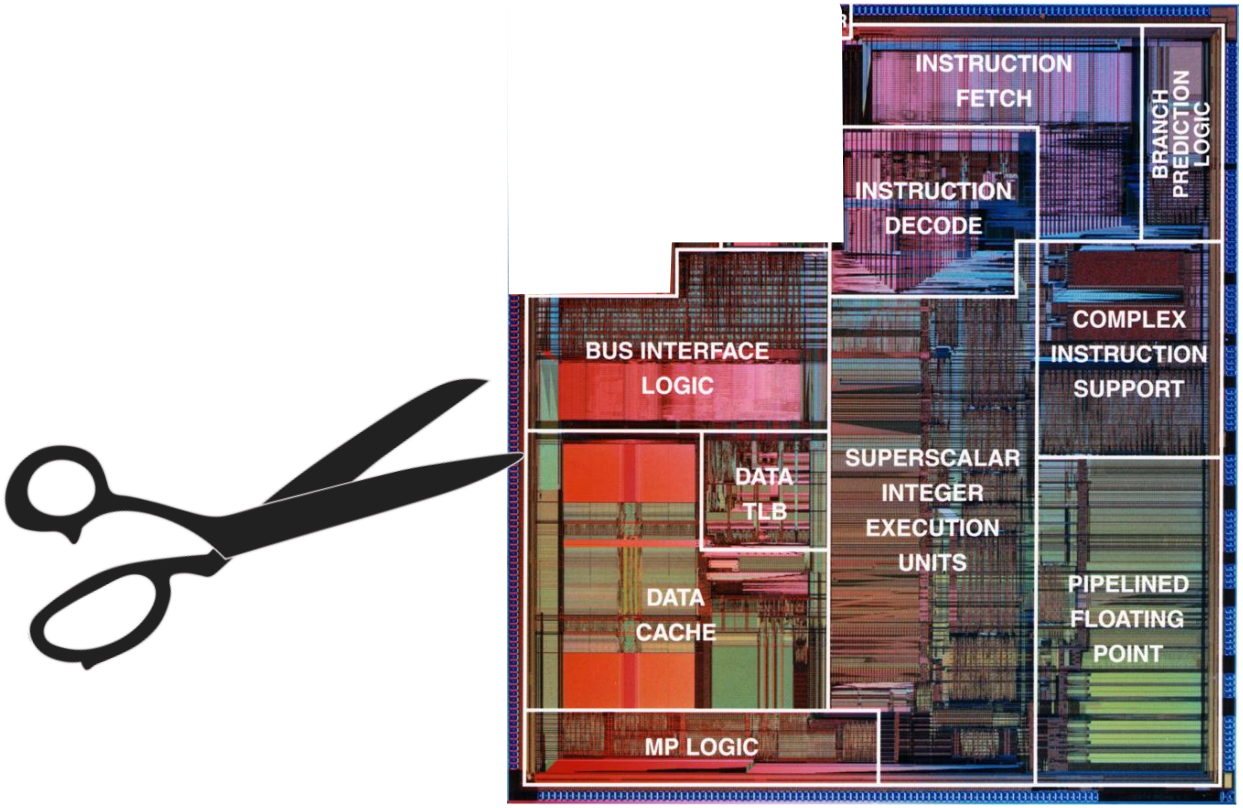


Actual ALU units are < ~40% of the area

In a dream...


CLOCK DRIVER
CODE CACHE
INSTRUCTION FETCH
BRANCH PREDICTION LOGIC
CODE TLB
INSTRUCTION DECODE
BUS INTERFACE LOGIC
COMPLEX INSTRUCTION SUPPORT
DATA TLB
SUPERSCALAR INTEGER EXECUTION UNITS
DATA CACHE
PIPELINED FLOATING POINT
MP LOGIC

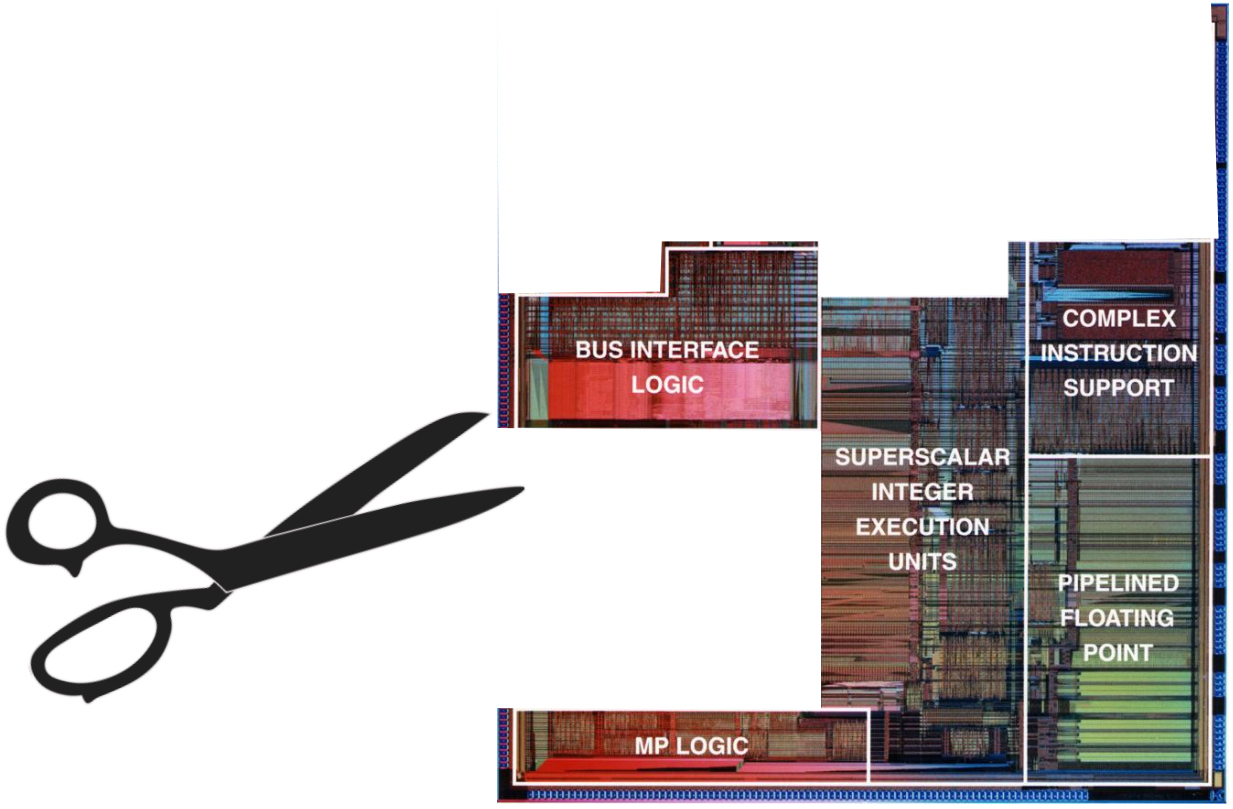
In a dream...


INSTRUCTION FETCH
BRANCH PREDICTION LOGIC
INSTRUCTION DECODE
COMPLEX INSTRUCTION SUPPORT
BUS INTERFACE LOGIC
DATA TLB
SUPERSCALAR INTEGER EXECUTION UNITS
DATA CACHE
PIPELINED FLOATING POINT
MP LOGIC

In a dream…


INSTRUCTION FETCH
BRANCH PREDICTION LOGIC
INSTRUCTION DECODE
BUS INTERFACE LOGIC
COMPLEX INSTRUCTION SUPPORT
SUPERSCALAR INTEGER EXECUTION UNITS
PIPELINED FLOATING POINT
MP LOGIC

In a dream…


BUS INTERFACE
LOGIC
COMPLEX
INSTRUCTION
SUPPORT
SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT
MP LOGIC

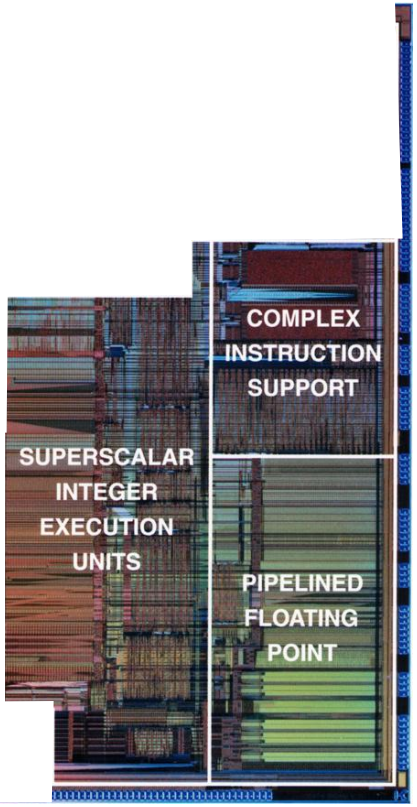
In a dream…


COMPLEX
INSTRUCTION
SUPPORT
SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT
MP LOGIC

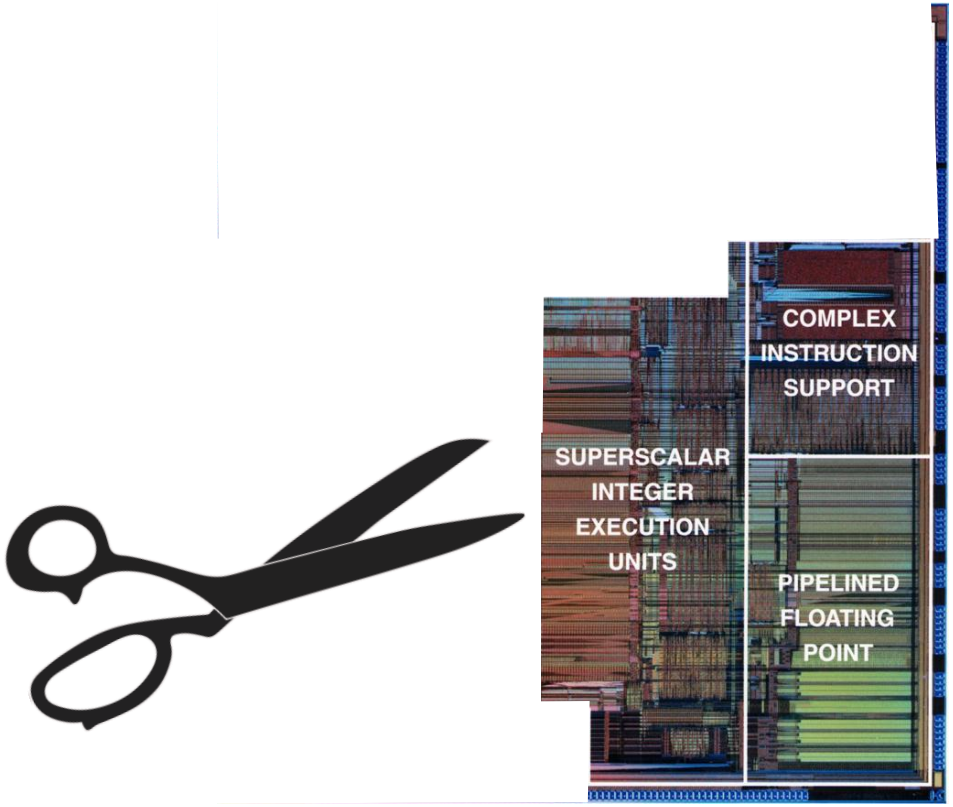
In a dream…


COMPLEX
INSTRUCTION
SUPPORT
SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT

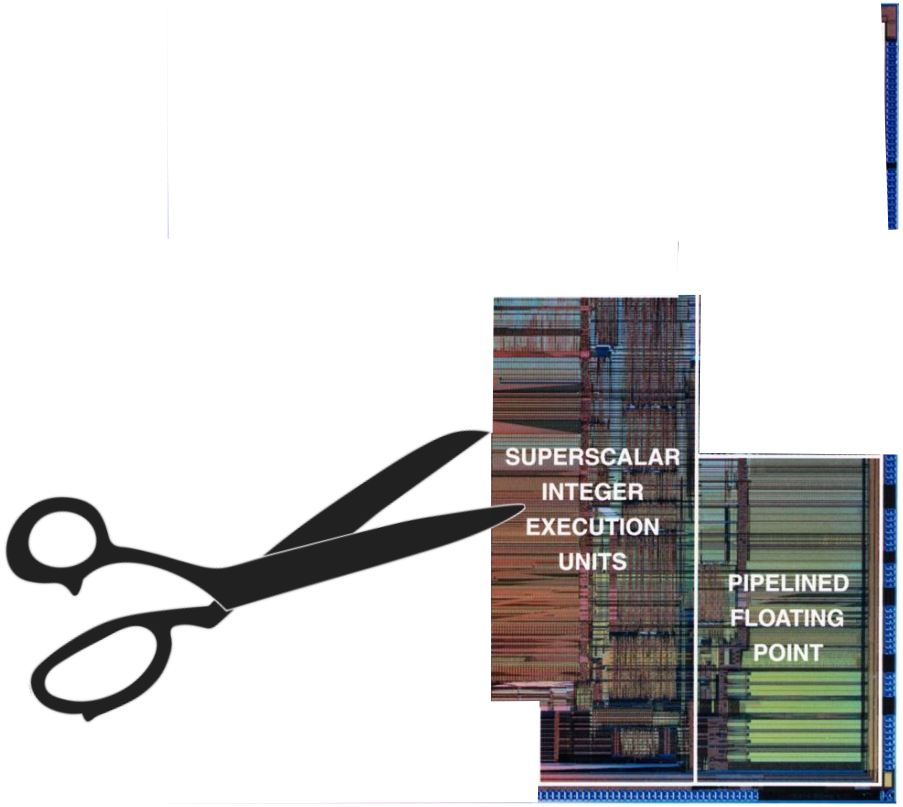
In a dream…


COMPLEX
INSTRUCTION
SUPPORT
SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT

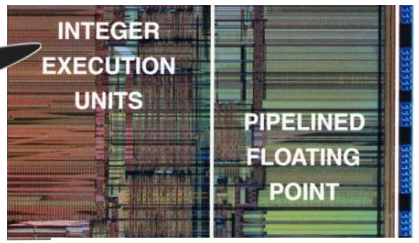
In a dream…


COMPLEX
INSTRUCTION
SUPPORT
SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT

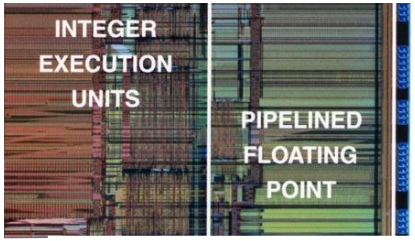
In a dream…


SUPERSCALAR
INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT

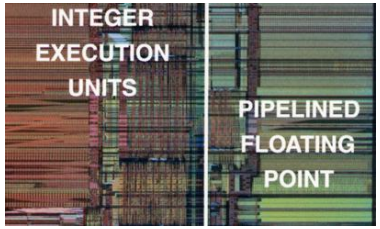
In a dream…



In a dream…



In a dream…



In a dream…

In a dream…

In a dream…

In a dream…

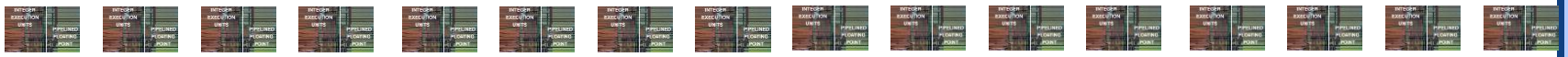

INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT


INTEGER
EXECUTION
UNITS
PIPELINED
FLOATING
POINT

In a dream...

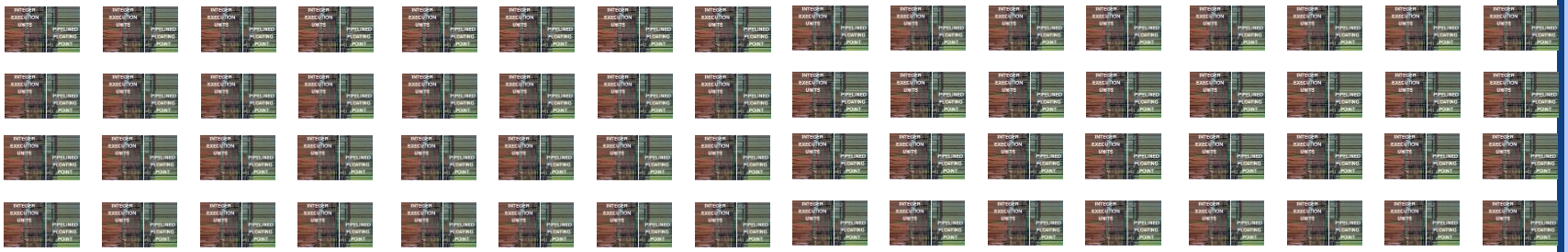
In a dream...

In a dream...

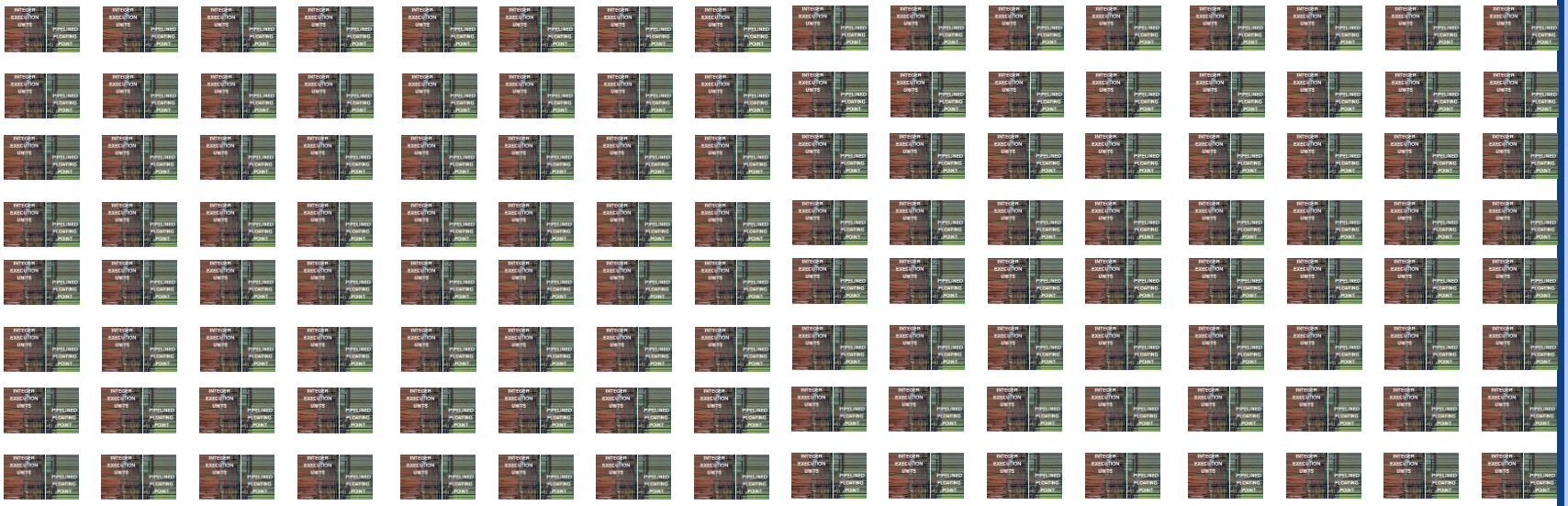


In a dream...

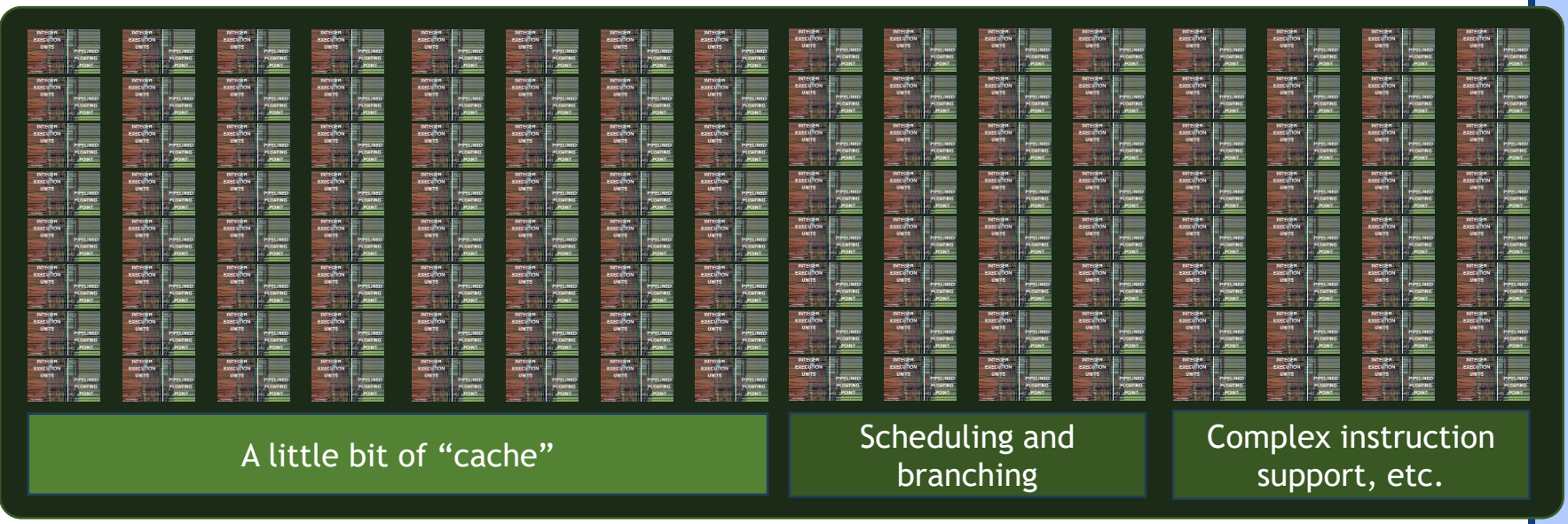
In a dream...



In a dream...



In a dream…


A little bit of “cache”
Scheduling and branching
Complex instruction support, etc.

Then you wake up…

Then you wake up…

Was it a dream,
or was it a glimpse of the future?

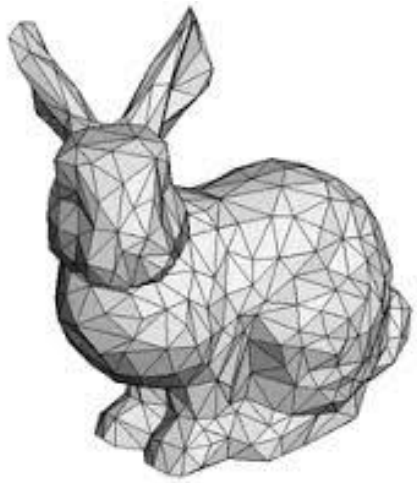
What were GPUs built for

What is actually we are trying to solve here?

What were GPUs built for

Incremental image synthesis

- Vertex transformation
- Rasterization



What were GPUs built for

Incremental image synthesis:

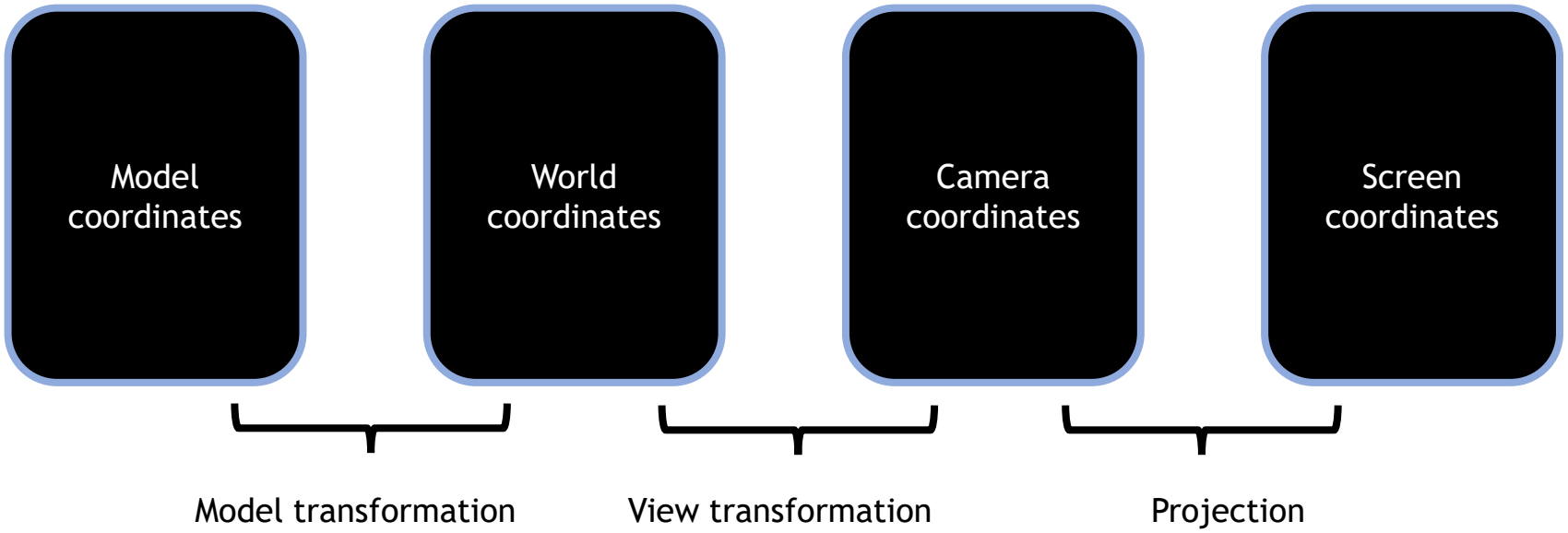
Vertex transformation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ c_1 & c_2 & c_3 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

- Linear transformations acting on 3-dimensional Euclidean space can be described by 3×3 matrices
- Parallelism-preserving transformations are called affine transformations
- "If *X* is the point set of an affine space, then every affine transformation on *X* can be represented as the composition of a linear transformation on *X* and a translation of *X*."
- So we can describe projections and displacements if we use 4×4 matrices

What were GPUs built for

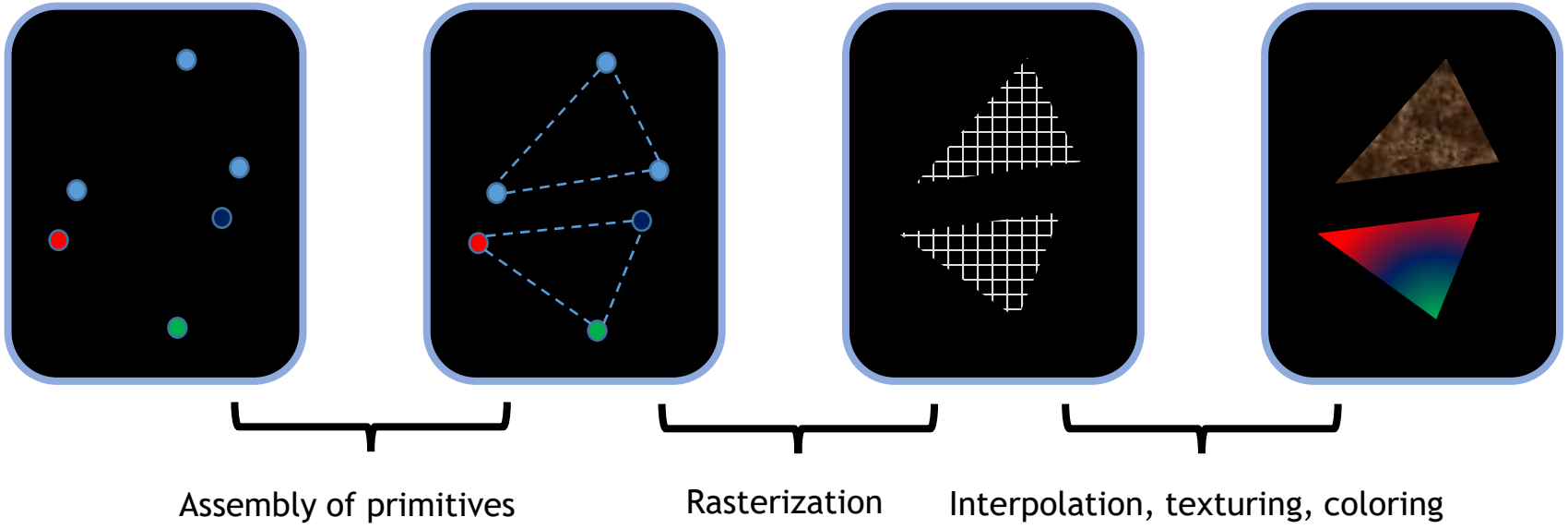
Incremental image synthesis: Vertex transformation



Each transformation is described by a 4x4 matrix.

What were GPUs built for

Incremental image synthesis: Rasterization



These steps can be implemented with fixed function and programmable hardware elements (shaders).

What were GPUs built for

The rule of four:

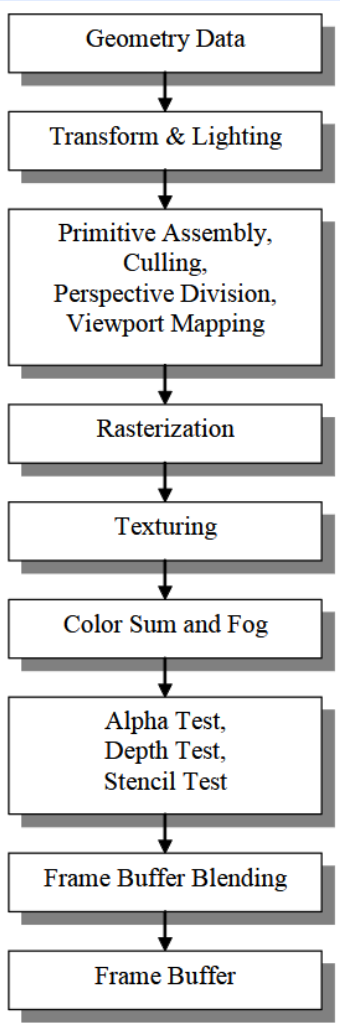
The GPUs are designed to work with 4-wide vectors (floats, ints) since:

- Vertex transformations operate on 4-vectors
- Colors are represented by 3+1 components:
  red green blue and alpha (transparency)

What were GPUs built for

In the early days, rendering was done by a fixed function pipeline similar to this:

(discrete choice from a few pre-implemented algorithms)

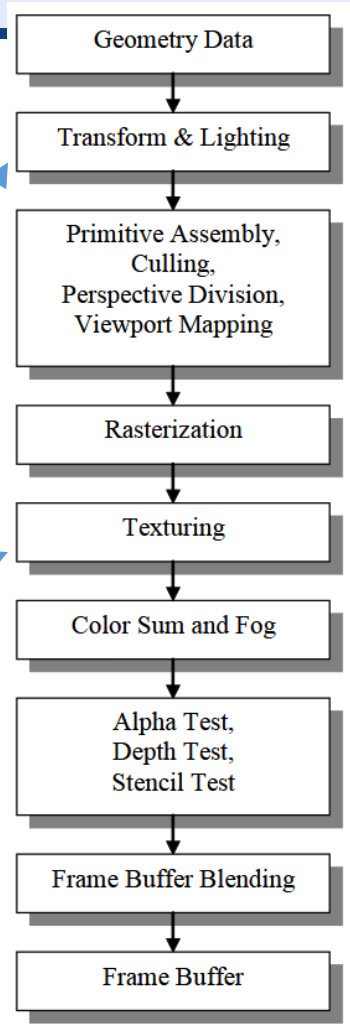


source

What were GPUs built for

Fixed function pipeline (abstract):

Programmable shading started by making these two stages programmable

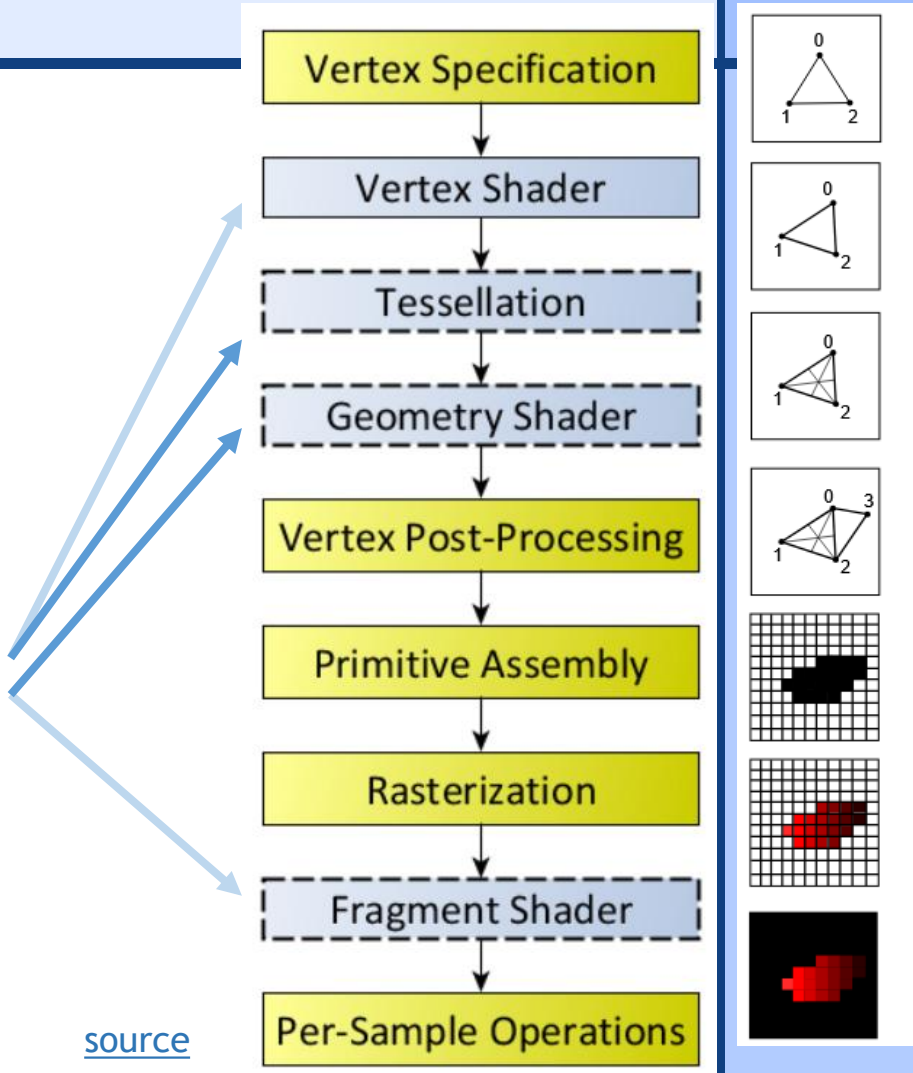


source

What were GPUs built for

Programmable pipeline:

Then other stages started to be programmable too

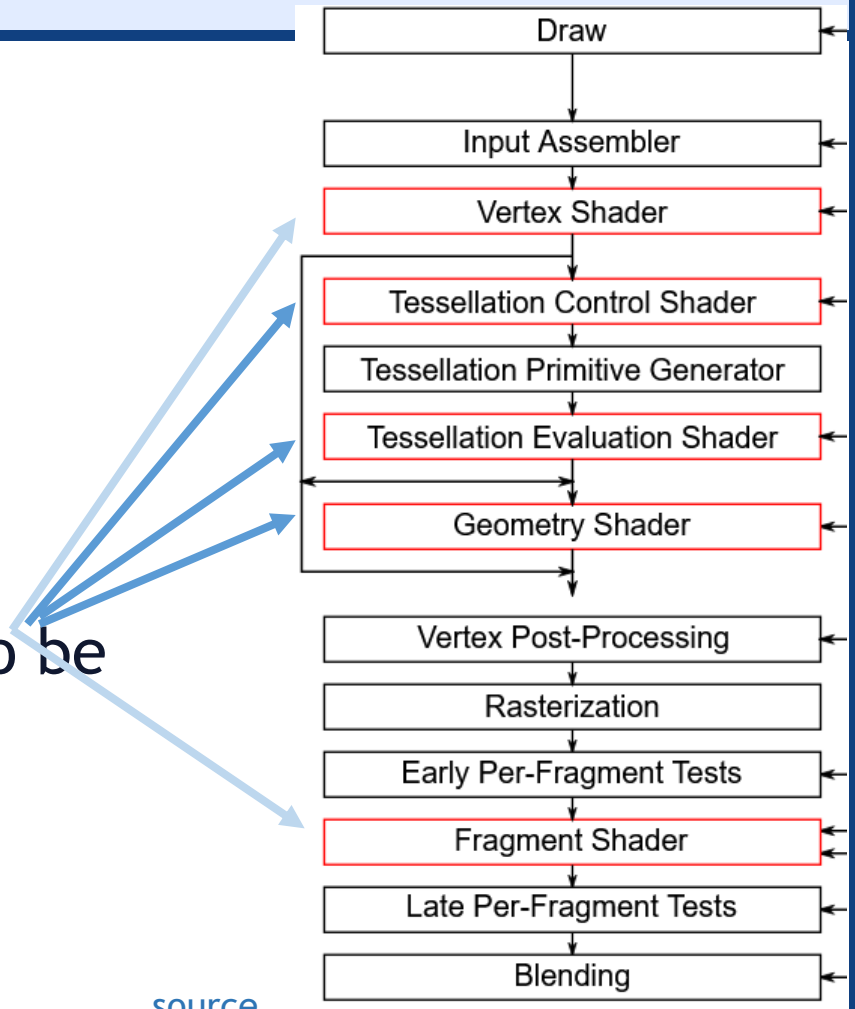


source

What were GPUs built for

Programmable pipeline:

Then other stages started to be programmable too



source

- What were GPUs not built for? (originally)

What were GPUs not built for? (originally)

When the programmable shader stages appeared, people already started to abuse the graphics API to do science:

- Large linear algebra computations
- Fluid simulations
- Lattice QCD
- …

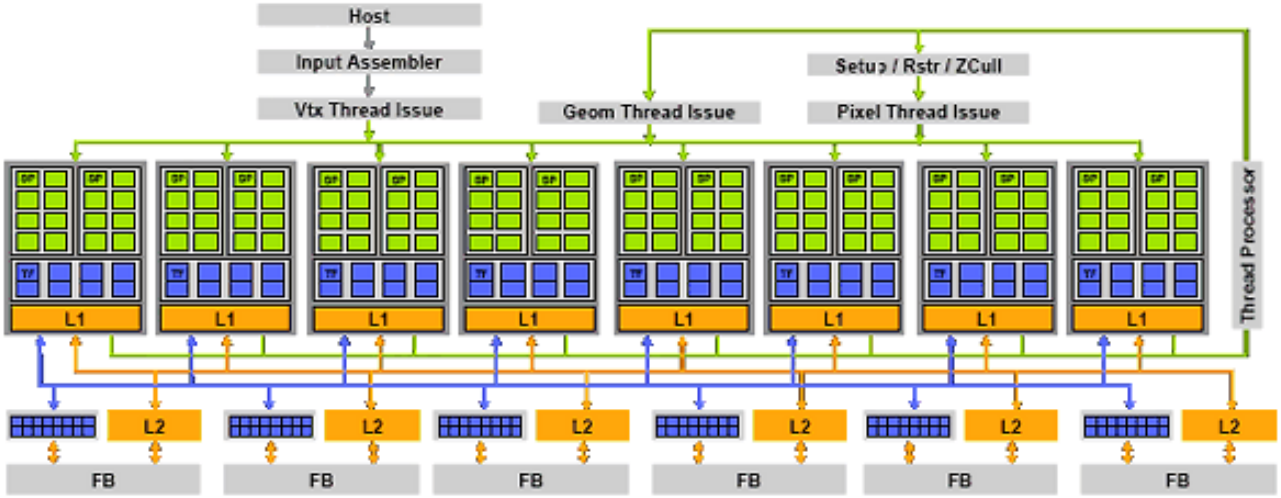
What were GPUs not built for? (originally)

Originally, when programmable shading first appeared, there was literally separate hardware for vertex and fragment shaders.

Then it all changed with the GeForce 8800 card in 2006

What were GPUs not built for? (originally)

Then it all changed with the GeForce 8800 card in 2006



Same hardware units do both vs and fs computation!

What were GPUs not built for? (originally)

- The same year NVIDIA introduced CUDA for general purpose programming (rise of the GPGPU concept)
- Compute shaders first appeared in 2009 as an alternative pipeline in graphics
- Compute APIs started to proliferate in the 2010s

What were GPUs not built for? (originally)

Users wanted to:

- browse photos
- watch movies
- make video calls
- stream games online

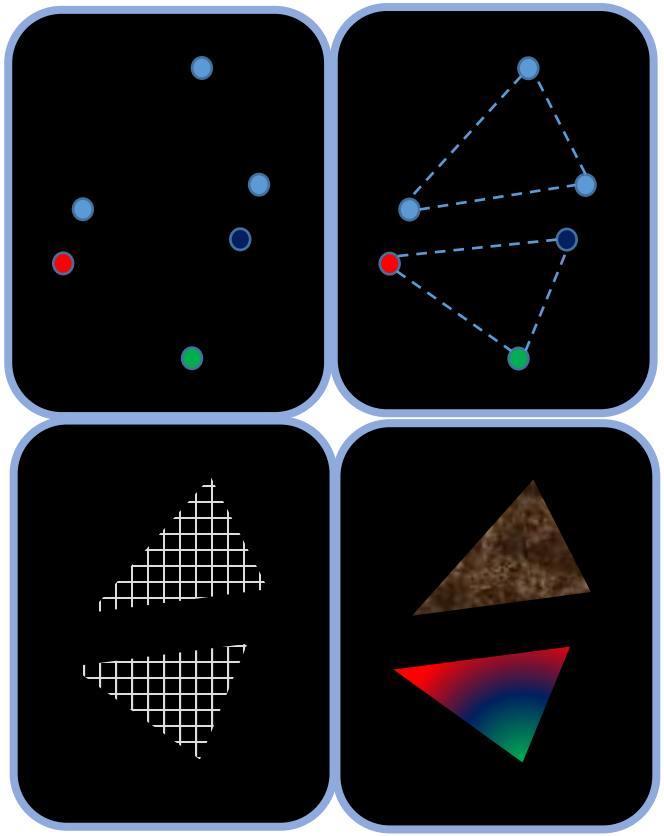
To make this efficient
(low power consumption,
no interruption to rendering tasks),

image/video encoding/decoding got integrated
via dedicated units into the GPUs

What were GPUs not built for? (originally)

- Raytracing

What were GPUs not built for? (originally)



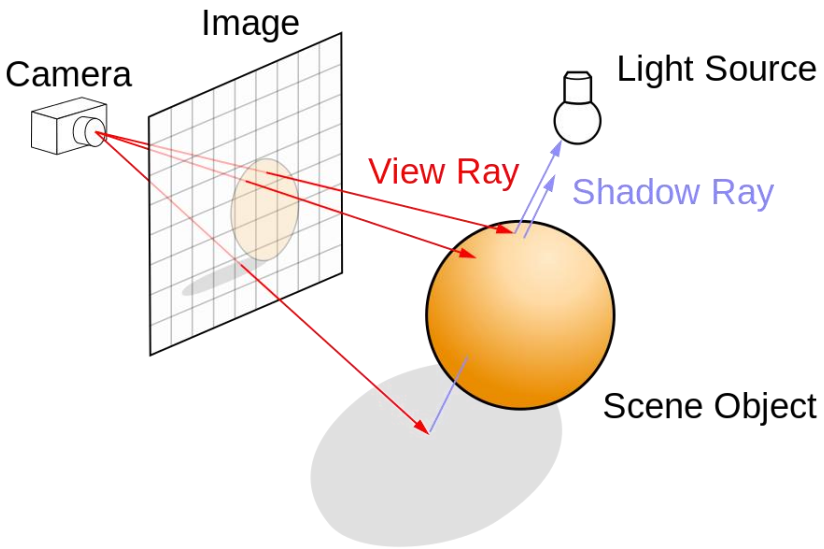
Incremental image synthesis / rasterization is actually just a crude approximation of how light actually composes into a 2D image in our eyes or a camera's sensor

It is crude, but fast / cheap

Best choice for real-time graphics (games)

Some graphics problems are difficult to solve
(e.g. multiple colored transparent objects, shadows, etc).

What were GPUs not built for? (originally)



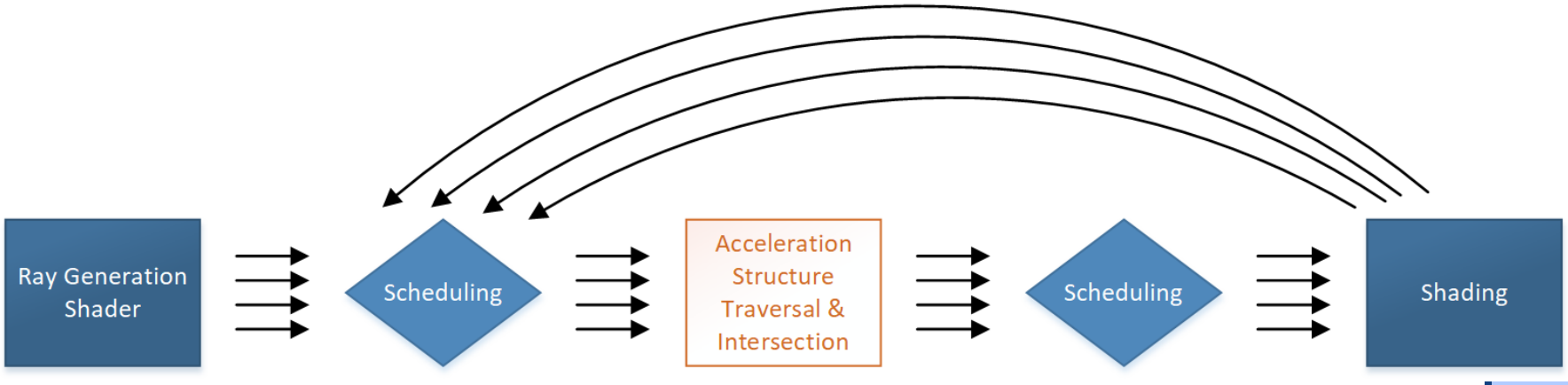
Ray tracing:
physically (more) correct imaging

- It follows the path of light rays between light sources and pixels
- The rays sum up the colors from different surfaces
- Mathematically, the Monte-Carlo integration of the render equation takes place:

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

What were GPUs not built for? (originally)

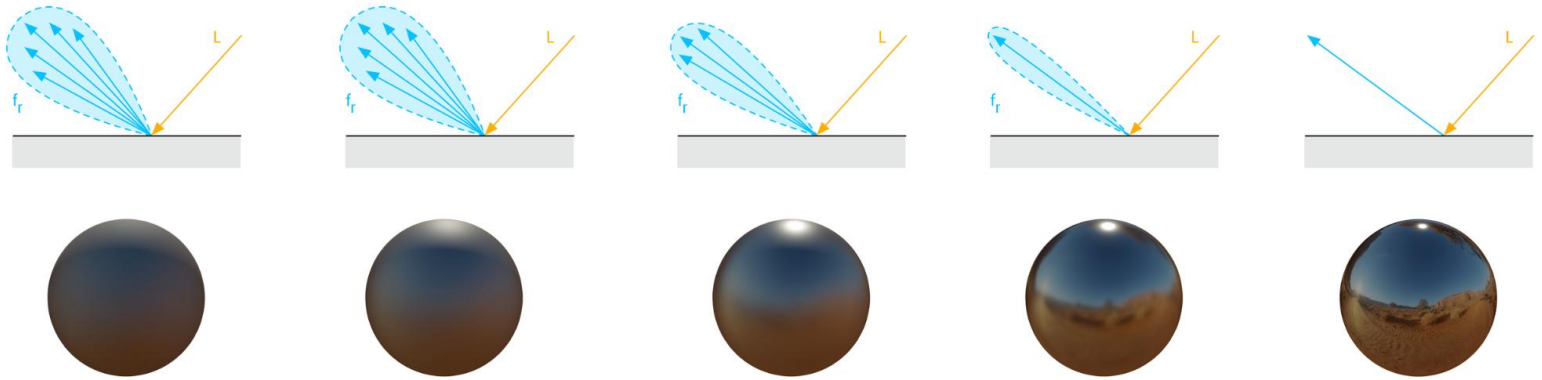
Hardware accelerated raytracing support landed in GPUs in 2018. This is mostly accelerating ray-surface intersection.

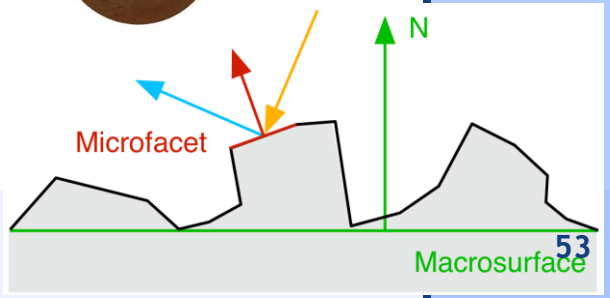


What were GPUs not built for? (originally)

- While not specifically a hardware feature, 2010s saw a rapid rise of Physically Based Rendering in graphics

This approach is centered on modelling the surface properties more in line with how actually light behaves





What were GPUs not built for? (originally)

- Physically Based Rendering

Some aspects:

- Proper handling of reflection/refraction (Fresnel),
- diffusion, translucency, transparency
- Energy conversation respected
- Easier parameterization
- Improved consistency across rendering methods

Physically Based Rendering book (free)

What were GPUs not built for? (originally)

- Physically Based Rendering - Fresnel

See also

Physically Based Rendering book (free)

What were GPUs not built for? (originally)

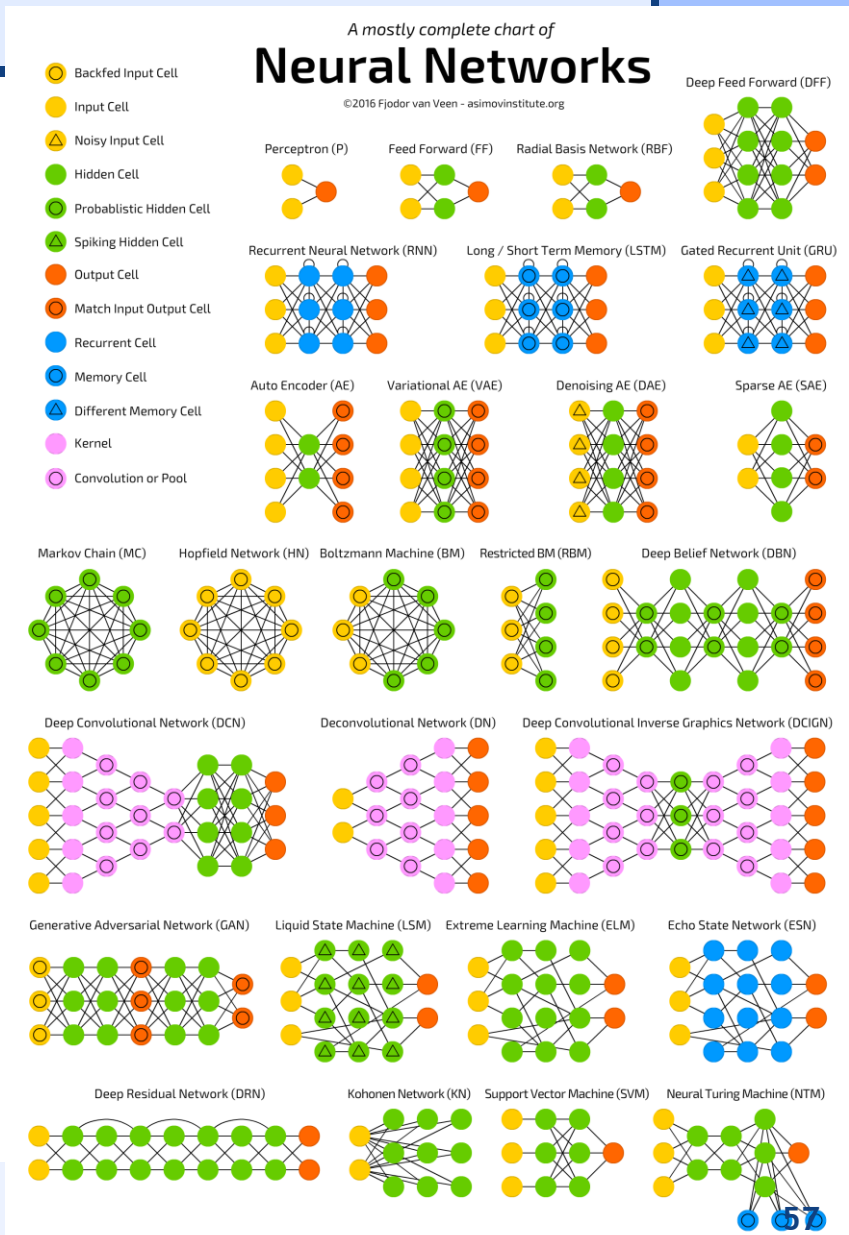
- Computing hashes

Turns out GPUs are good at this too...

The crypto currency hype caused significant shortages in GPU hardware stocks in 2018 and 2020-2021

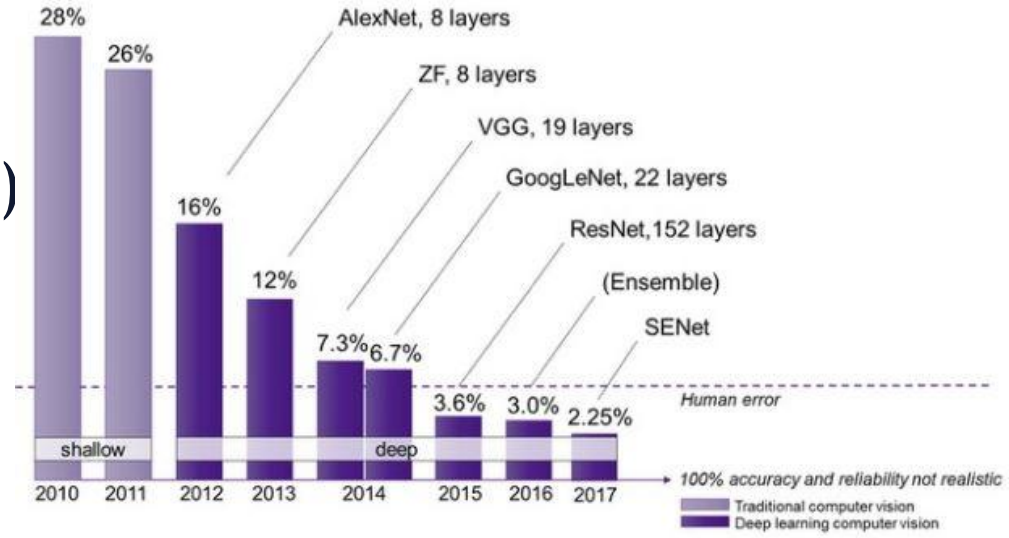
What were GPUs not built for? (originally)

- Neural networks (mid 2010s)



What were GPUs not built for? (originally)

- Neural networks (mid 2010s)



Turns out GPUs are good at
massive volumes of linear algebra operations
and convolutions

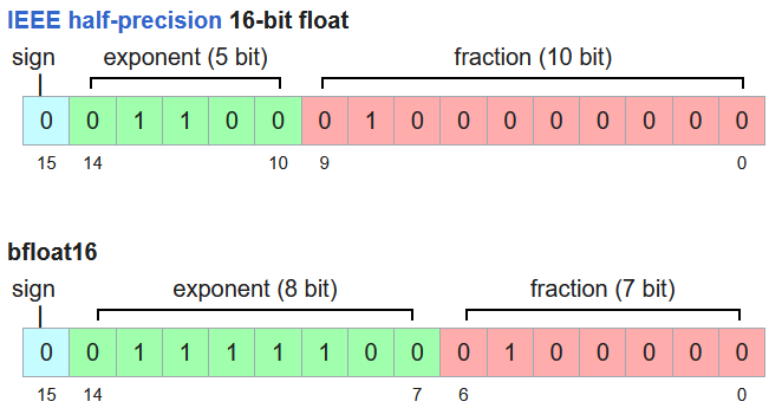
The revolution of image processing was mostly driven by GPUs

What were GPUs not built for? (originally)

- Neural networks (mid 2010s)

It turned out, that you can do inference with reduced precision (most workloads are bw limited)

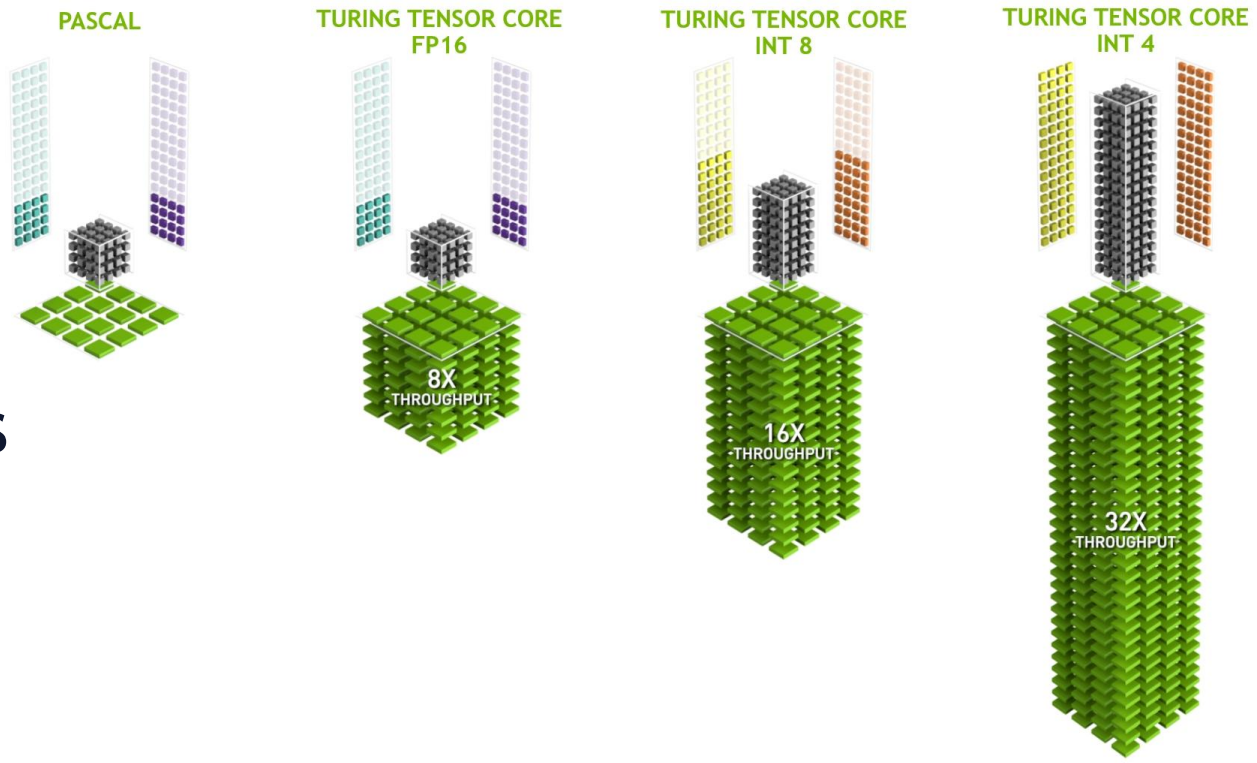
Rise of ‘alternative’ floating point formats, like bfloat16



What were GPUs not built for? (originally)

- Neural networks (mid 2010s)

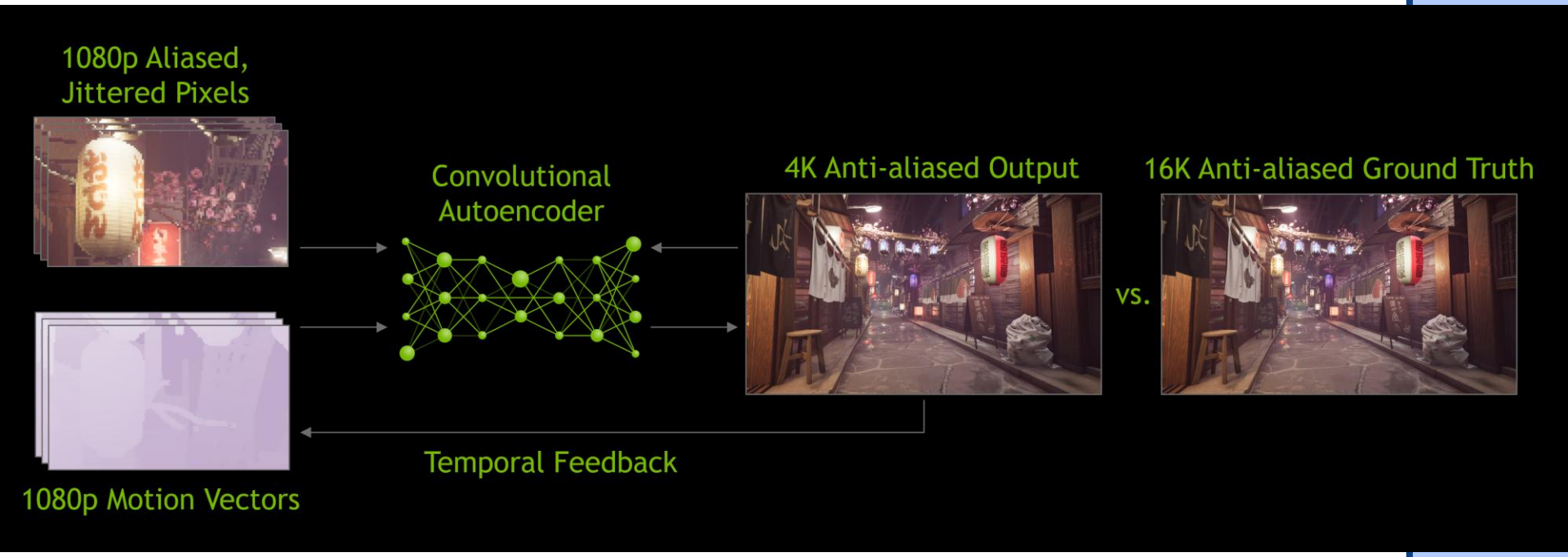
Turns out you can do linalg ops much faster in reduced precision



NVIDIA introduces Tensor Cores

What were GPUs not built for? (originally)

- Deep Learning Super Sampling in real-time graphics (2019)

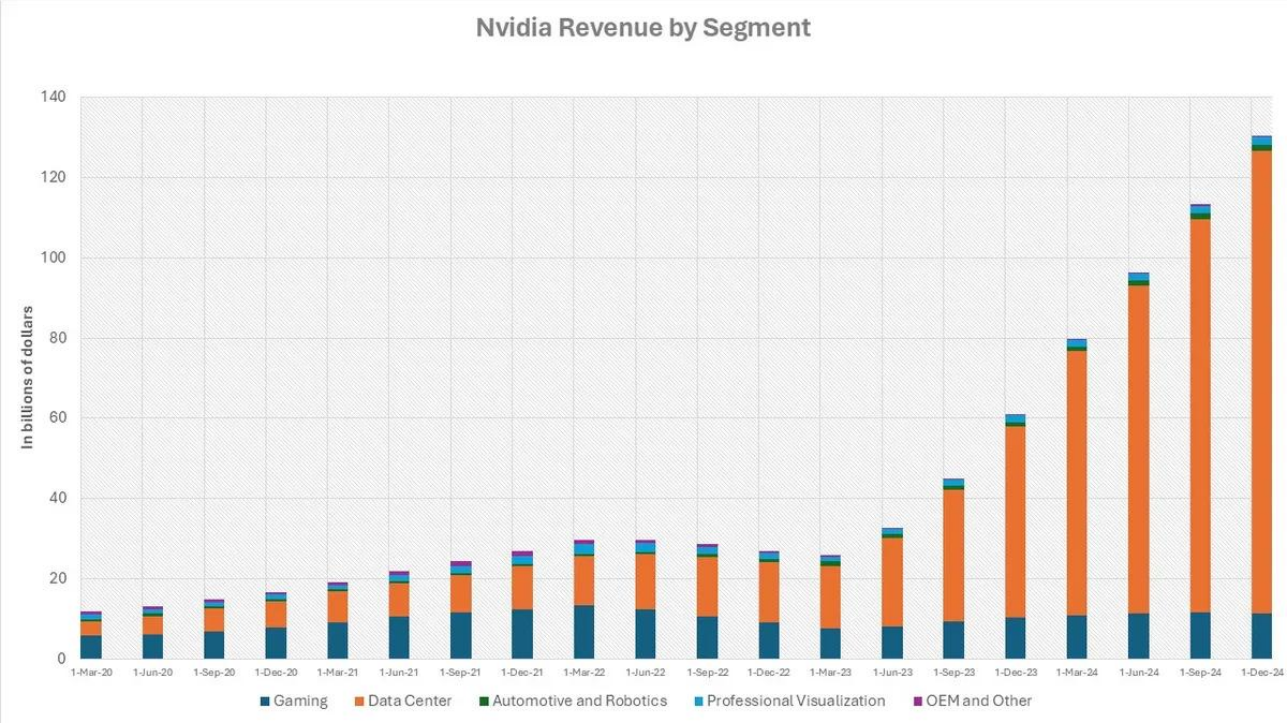


What were GPUs not built for? (originally)

- The AI revolution (2023-present)

What were GPUs not built for? (originally)

- The AI revolution (2023-present)
- Large Language Models drive incentives

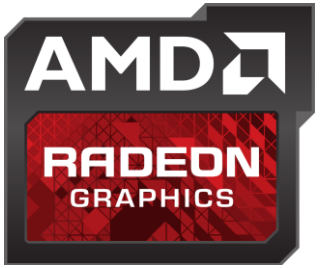


Who makes GPUs?

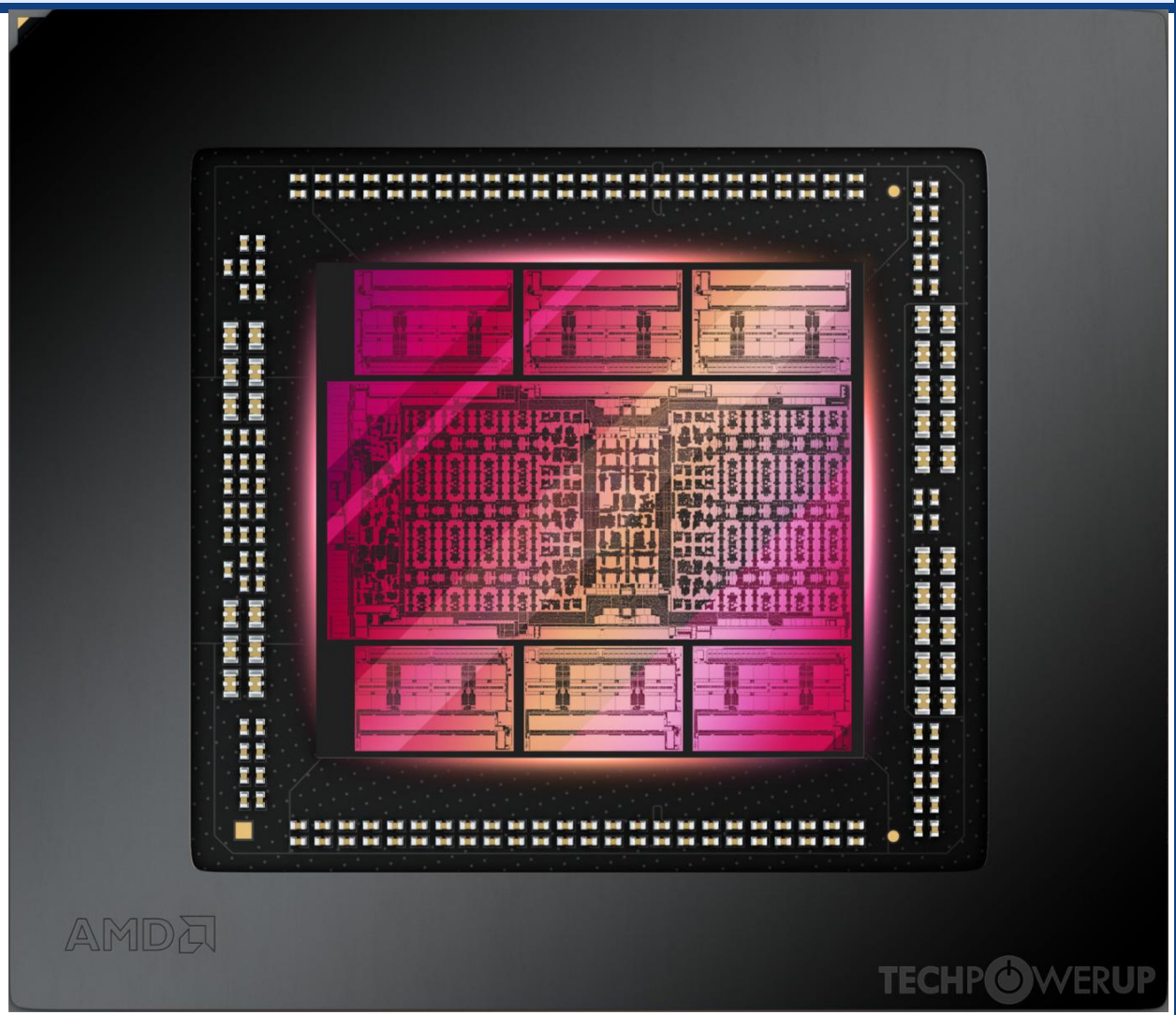
There are basically three major GPU chip development companies on the market:

GPUs for desktop, server and embedded systems

GPUs and CPUs for desktop and server environments and more recently, embedded chips

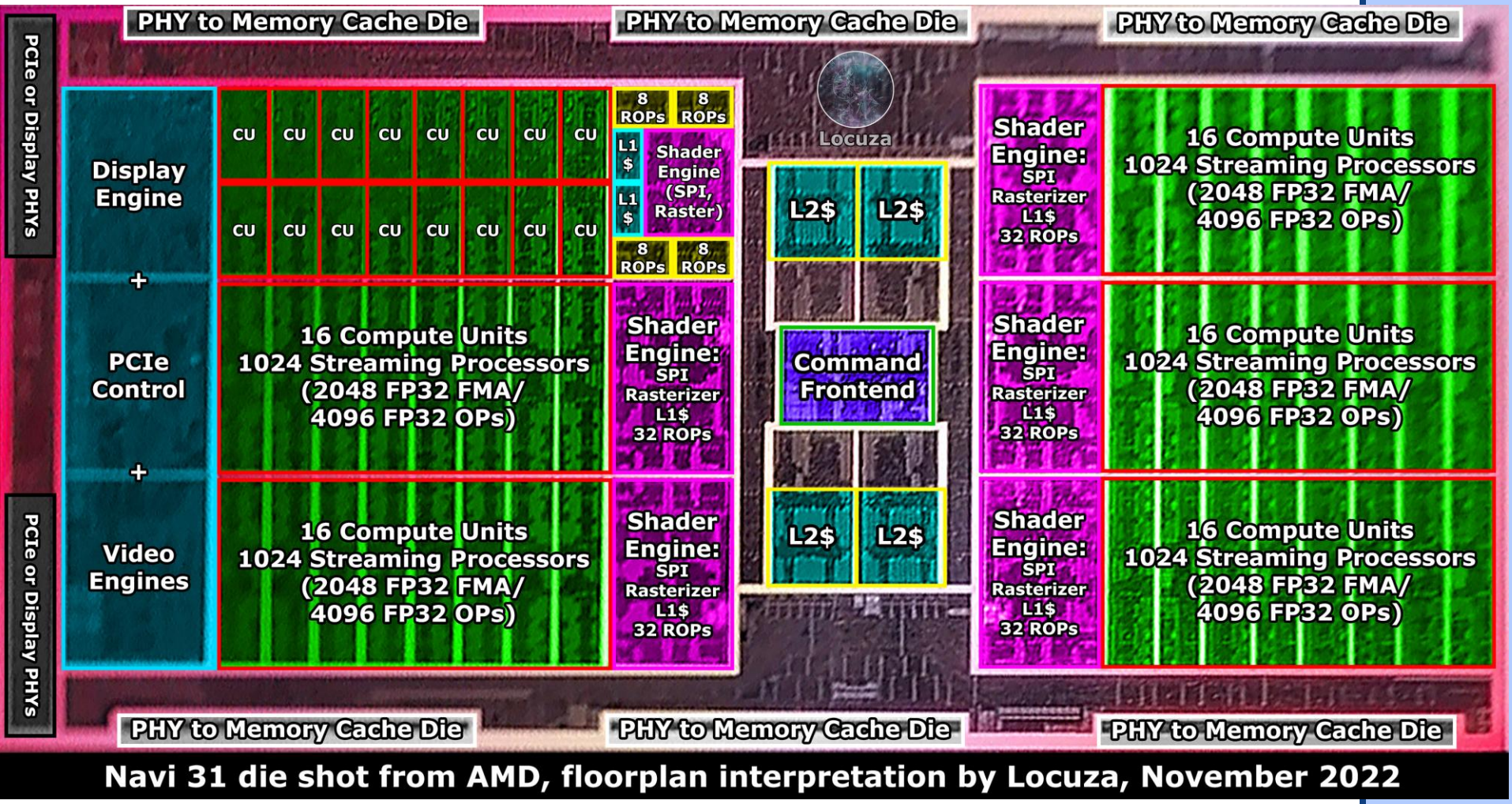


AMD Navi 31



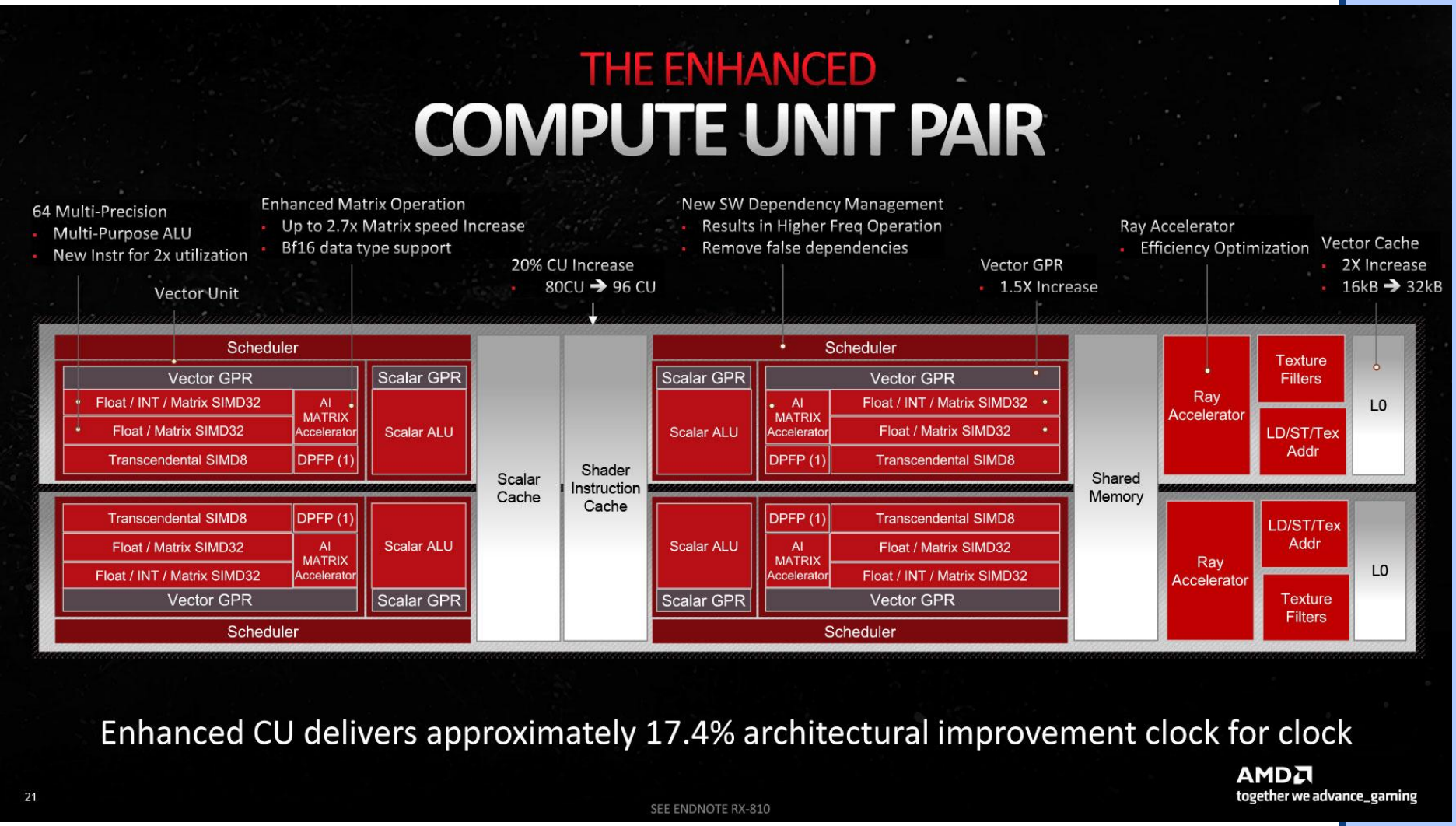
ISA Docs

AMD Navi 31



Navi 31 die shot from AMD, floorplan interpretation by Locuza, November 2022

AMD Navi 31

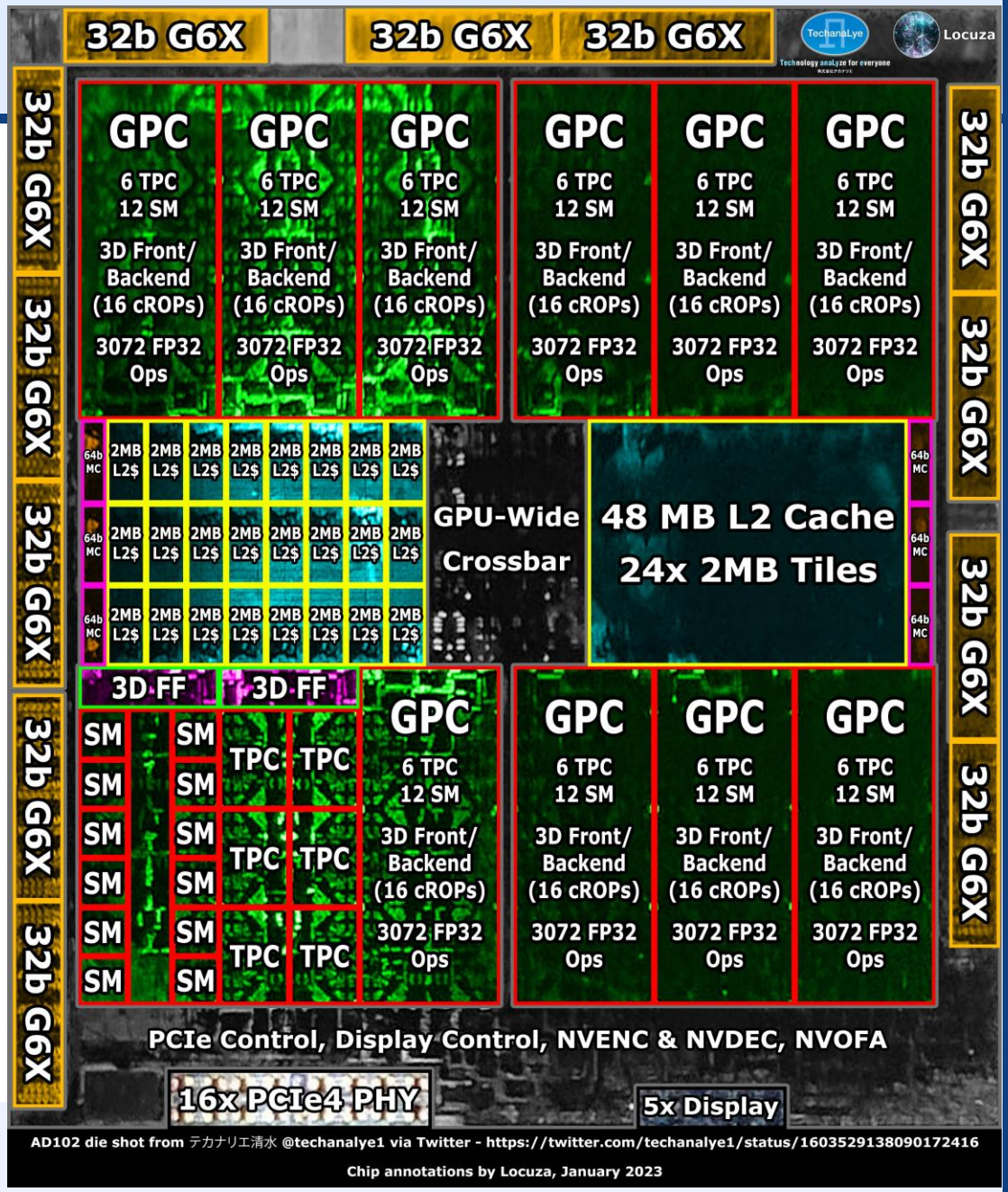


Enhanced CU delivers approximately 17.4% architectural improvement clock for clock

SEE ENDNOTE RX-810

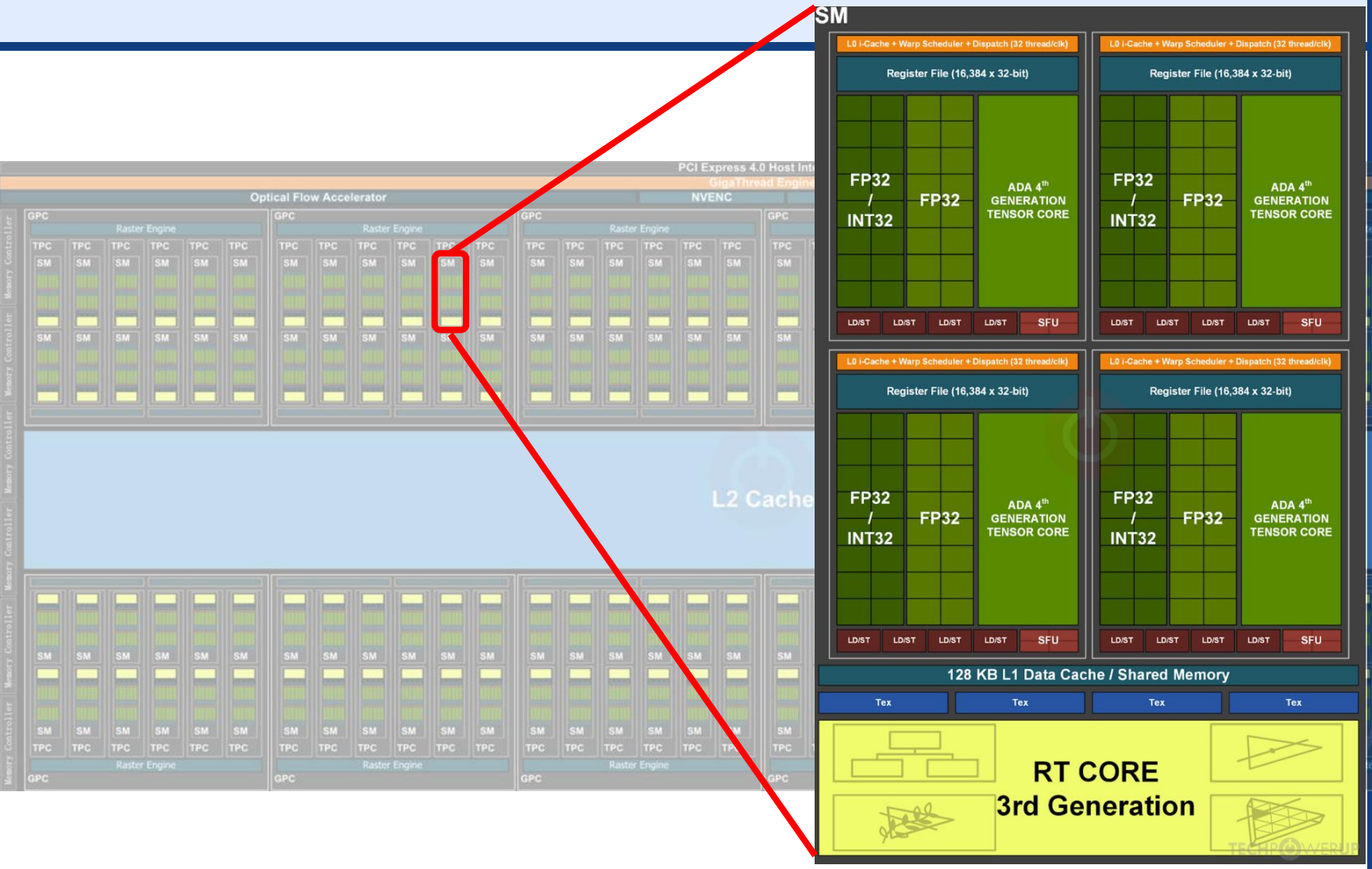
AMD
together we advance_gaming

NVIDIA AD102

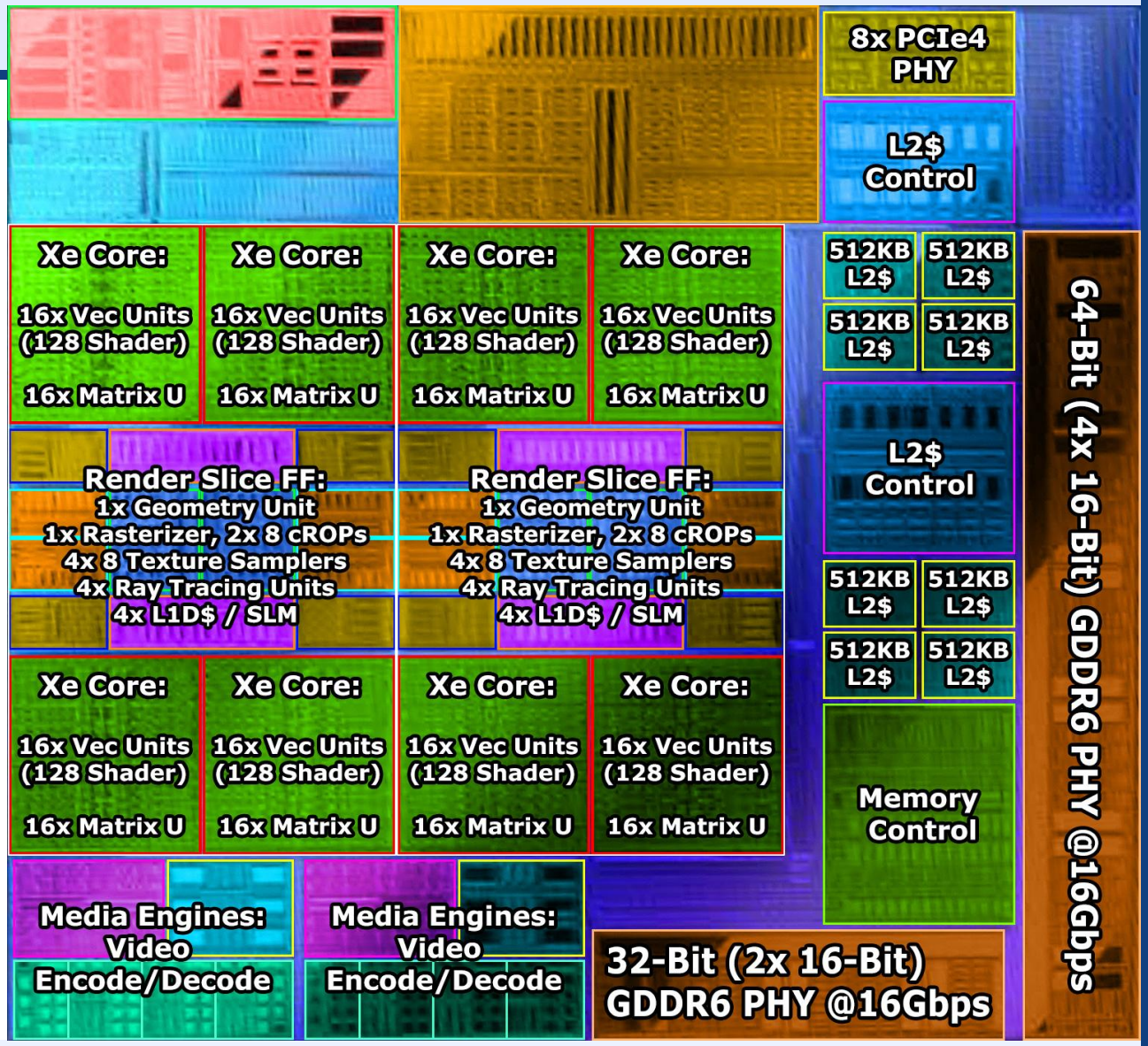


NVIDIA AD102

NVIDIA AD102



Intel DG2-128



Who makes GPUs?

However, there are others too:

- Apple: always doing their own thing
- ARM, Qualcomm, Imagination Technologies, Samsung

  Mostly mobile and embedded and other low power and/or otherwise special applications

GPUs vs CPUs

- How GPUs are different from CPUs?

GPUs vs CPUs

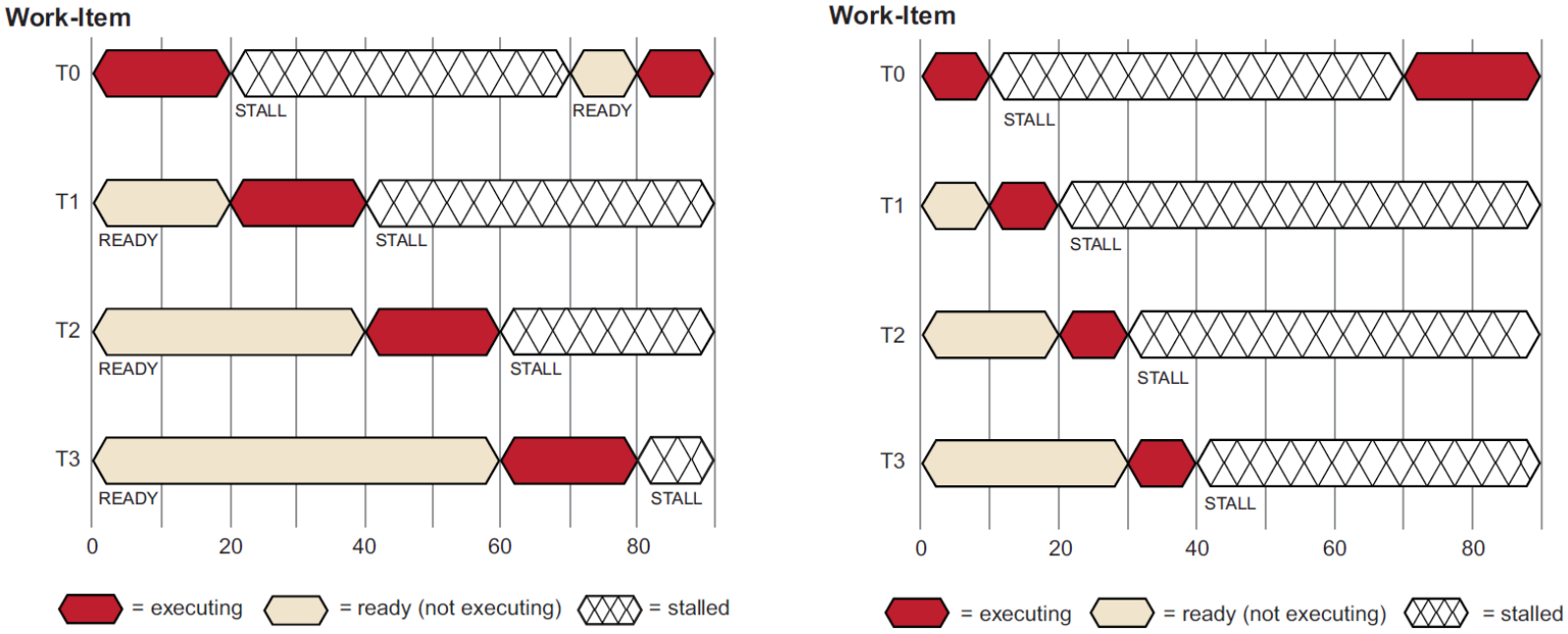
- By looking at the ratios, you could see that GPUs have much less cache per number of processing elements compared to CPUs

- So, they need to wait a lot for data to arrive from memory…

  How is this going to work at all?

GPUs vs CPUs

- Latency hiding



If we have enough work,
we can hide latency by switching over to another thread and make progress

GPUs vs CPUs

- Cost of operations on the GPU (NVIDIA Ampere arch)

THE MEMORY ACCESSES LATENCIES

Memory type	*CPI (cycles)*
Global memory	290
L2 cache	200
L1 cache	33
Shared Memory (ld/st)	(23/19)

- Compare to e.g. AMD Zen4:
  - L1: 4-8 cycles
  - L2: 14-17 cycles
  - L3: ~50 cycles
  - RAM: ~300 cycles

GPUs vs CPUs

- Cost of operations on the GPU (NVIDIA Ampere arch)

Operation	Cycles
Add / Mul / FMA / logical	2-4
Div	66-426
Rcp (1/x)	198-244
Sqrt	190-340

- Compare to AMD Zen4:
  - Add/Mul/FMA: 1-4
  - Div / rcp: 11-13
  - Sqrt: 15-21

GPUs vs CPUs

- How does this work?
  All this is pretty much in contrast to CPUs!

GPUs vs CPUs

CPU

- The scheduler is a software run by the OS
- Threads run random/unknown programs
- Cores run threads completely independently

- Relies on caches -> cache issues -> performance issues

GPU

- The scheduler is in hardware

- All threads run the same program
- Execution is tightly linked across groups of execution units

Branching hurts more!

- Much less reliance on cache -> almost free switches between threads

GPUs vs CPUs

CPU

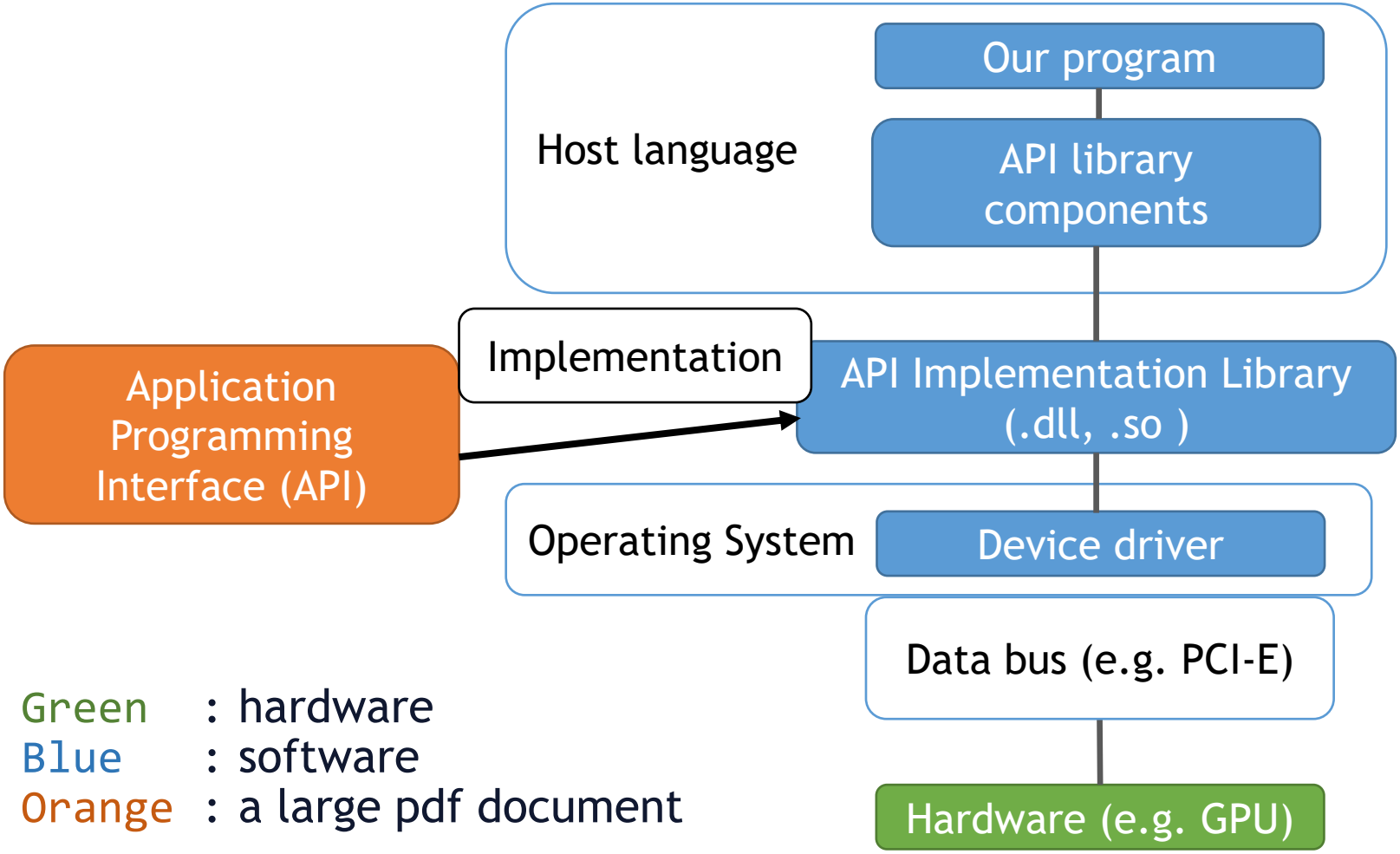
- Latency optimized hardware (low throughput)
- This is closer to a *general* solution to execution
- 10s of threads
- arbitrary workloads

GPU

- Throughput optimized hardware (very high latency)
- This is closer to a *special* solution to execution
- 1000s of threads
  But an individual thread is “weaker”
- optimized to execute the same massively parallel program

- How can we program GPUs?

What's what?

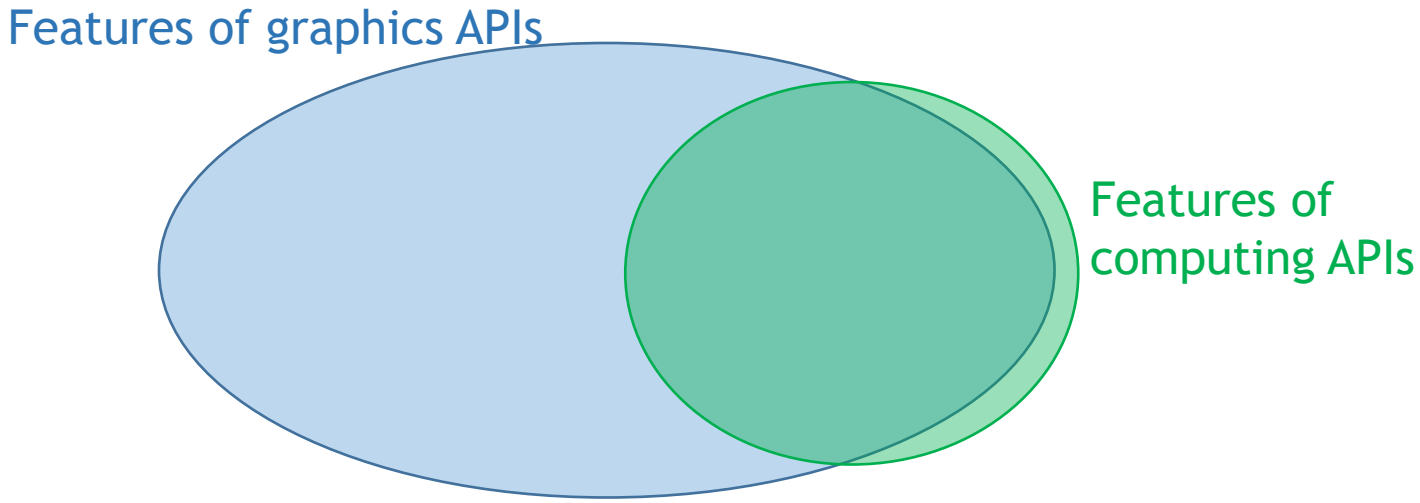


Green : hardware
Blue : software
Orange : a large pdf document

GPU APIs

The GPUs are accessed via an Application Programming Interface (API)

There are basically two types of APIs: graphics and compute



This is largely because graphics uses lots of *specialized* features, while compute is much more *general*

GPU APIs

	Graphics APIs	Compute APIs
“High level”	OpenGL, Direct3D (9-10-11)	CUDA, OpenCL, HIP, SYCL
“Low level”	Vulkan, Direct3D 12, Metal, webGPU	

The Khronos group

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

The Khronos group is a non-profit organization for developing royalty-free standards

It manages many of the key GPU standards and many others

It has wide support base: hardware, software companies, universities, research centers and individuals are contributing

Promoter Members:

Open Graphics Library

- The oldest existing graphics API from the 90s
- Khronos standard for quite some time, so it is open and anyone can implement it
- Fossil C API, state machine operation (doesn't even match OOP abstraction)
- Newest hardware features are not supported
- The device side language is GLSL (OpenGL Shading language)

For the web, there is webGL

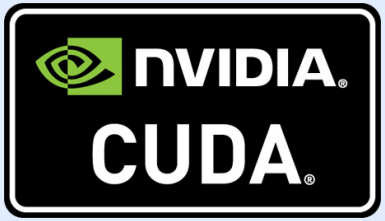
DirectX (Direct3D)

Microsoft® DirectX®

- Microsoft's leading 3Ds graphics API
- Native technology on Windows
  is one of the key leaders of game development
- C++ COM (Common Object model)
- The device language is
  HLSL (High-level Shading language)

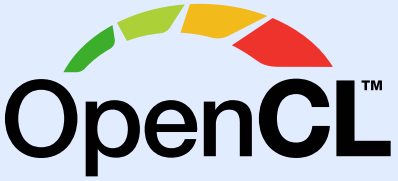
- The intermediate representation is
  DXIL (DirectX Intermediate language)

CUDA



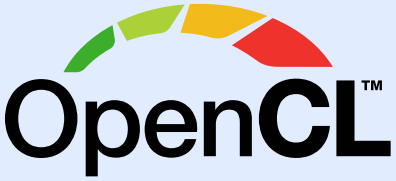
- Nvidia's general purpose compute API
- Since 2007
- Two levels of access
  - Driver API (C driver API)
  - Runtime API: C++98 language extension, in which special operators and decorators appear to separate host and device side code
- Closed technology, its capabilities are adapted to the current NVIDIA hardware capabilities

Open Computing Language



- Apple's initiative in response to the success of CUDA, development was transferred to the Khronos group
- Since 2009
- Separate host and device side language
  - C API for host code
  - C-like language for device-side code (static C++14 from OpenCL 2.1)
- Open technology, anyone can participate in its development and anyone can implement it

Open Computing Language



- Released in 2020, version 3.0 is a reboot of the technology
  - They want to improve the technology's biggest problem, accessibility, by making it easier to implement
  - The 2.x versions demanded too much from an implementation
    - In the end, every manufacturer found something to complain about
      - Meaning: all of them had something they couldn't/did not want to implement
- OpenCL 3.0 reverts to the 1.2 requirements and makes all new features optional, treated as extensions (Source compatible with 1.2)
- It is (also) available through translation layers
  - OpenCLOn12 (DirectX blog , Collabora blog)
  - clvk / clspv

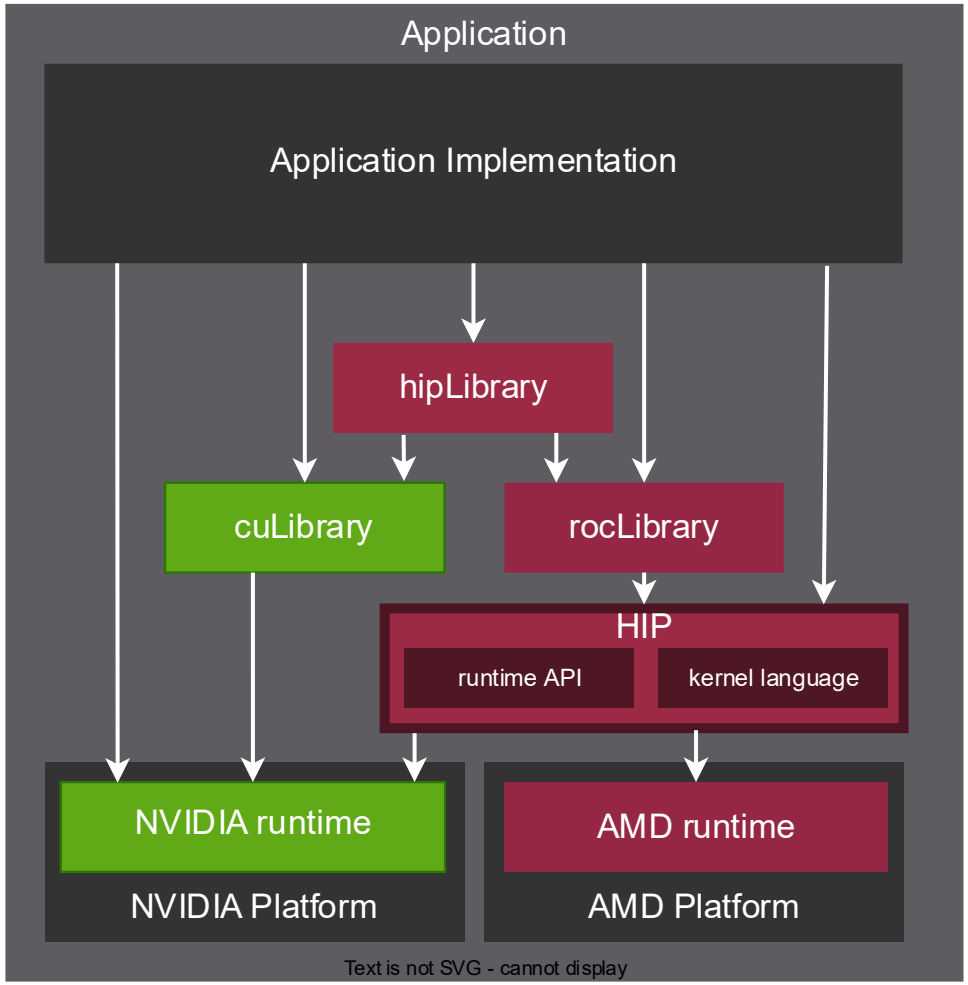
Heterogeneous-Computing Interface for Portability (HIP)

- AMD's answer to the massive success of CUDA and the missed breakthrough of OpenCL
- Since 2016
- Multi-level access (full stack open)
  - Driver API (C driver API)
  - Runtime API: C++98 language extension in which special decorators appear to separate host and device-side code (always device-side compiler bleeding-edge Clang trunk)
- Open technology, its capabilities are adapted to current CUDA API and AMD hardware capabilities
  - Crude copy of CUDA

Heterogeneous-Computing Interface for Portability (HIP)

AMD ROCm

What is HIP?



SYCL

- Higher-level Compute API compared to OpenCL (Since 2014)
- Aims at C++ compatibility
  - The host and device-side code is separated cleanly through the template library
- Open Khronos standard, anyone can implement it
- SYCL 1.2.1 strictly builds on OpenCL as a runtime environment
- SYCL 2020 and later also allows other runtime environments and various backends (abstracts OpenCL, CUDA, HIP)
  - Intel's DPC++ Implementation
  - AdaptiveCpp (University of Heidelberg)
  - triSYCL
  - Intel LLVM

Vulkan

- A low-level graphics API that is available on most platforms
- Open, Khronos standard, available since 2014
- It has one of the broadest industrial alliances behind it
- Its new flexible intermediate language (SPIR-V) has already leaked into OpenCL (mandatory from 2.2)
- For long it didn't have a separate device-side language, it just loads a SPIR-V binary, and it can come from anywhere
  Recently Khronos adopted the Slang shading language
- Prospective convergence with OpenCL

Metal

- Apple's own low-level graphics API
- This must be used on Apple platforms,
  all other APIs are essentially no longer supported there
- Its device-side language is
  Metal Shading Language
  (similar to HLSL)

webGPU

- WebGPU is a W3C standard for low-level graphics
- Not just for web!

- Javascript / C++ / rust /C API

- Device side language is its own webGPU shading language (WGSL)

- Still developing, not yet finalized standard, mostly pushed by Google motivated by AI compute workloads in browsers
- Has the prospect to really run and be supported “everywhere”