



Lecture 6

GPU Architecture

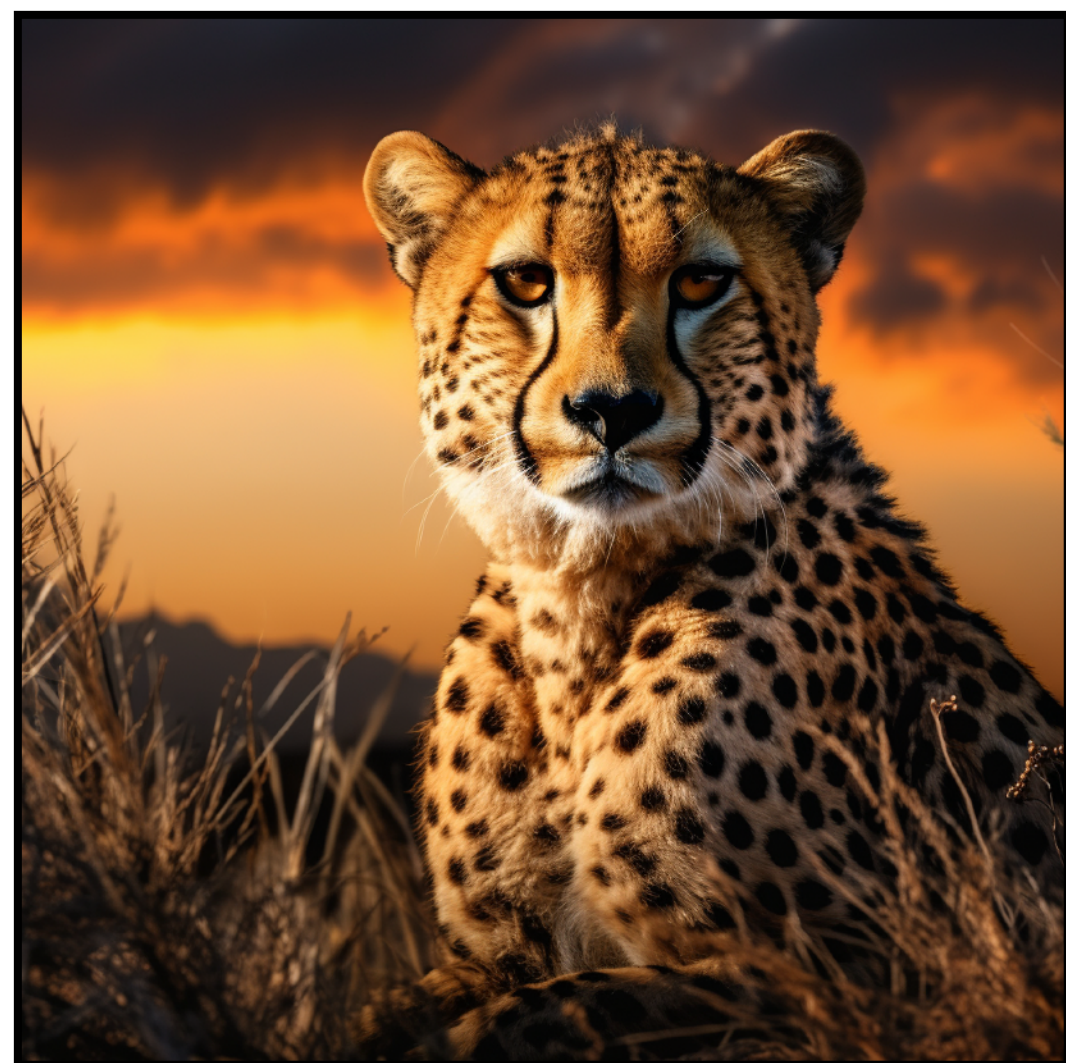
A peek into the world of accelerated computing

**CS328 - Numerical Methods for
Visual Computing and Machine Learning**

Prof. Wenzel Jakob

Today's lecture

CPUs



MidJourney: a Cheetah

- Evolved to run single threaded code run as fast as possible
- Nowadays: several cores (~4-16) for add'l parallelization
- Optimize **latency**

GPUs



ChatGPT: an army of snails

- Run massively parallel code
- Must have 10K - millions of "work items" to use effectively
- Optimize **throughput**

Learning goals:

- Understand how GPU cores differ from CPU cores
- Understand which workloads run better on GPUs vs CPUs.

3D Graphics, around 1996

Nintendo 64 — 320x400 pixels, 30 frames per second, runs at 62.5 Mhz



[Source: https://www.youtube.com/watch?v=gnf_YdBQPyo]

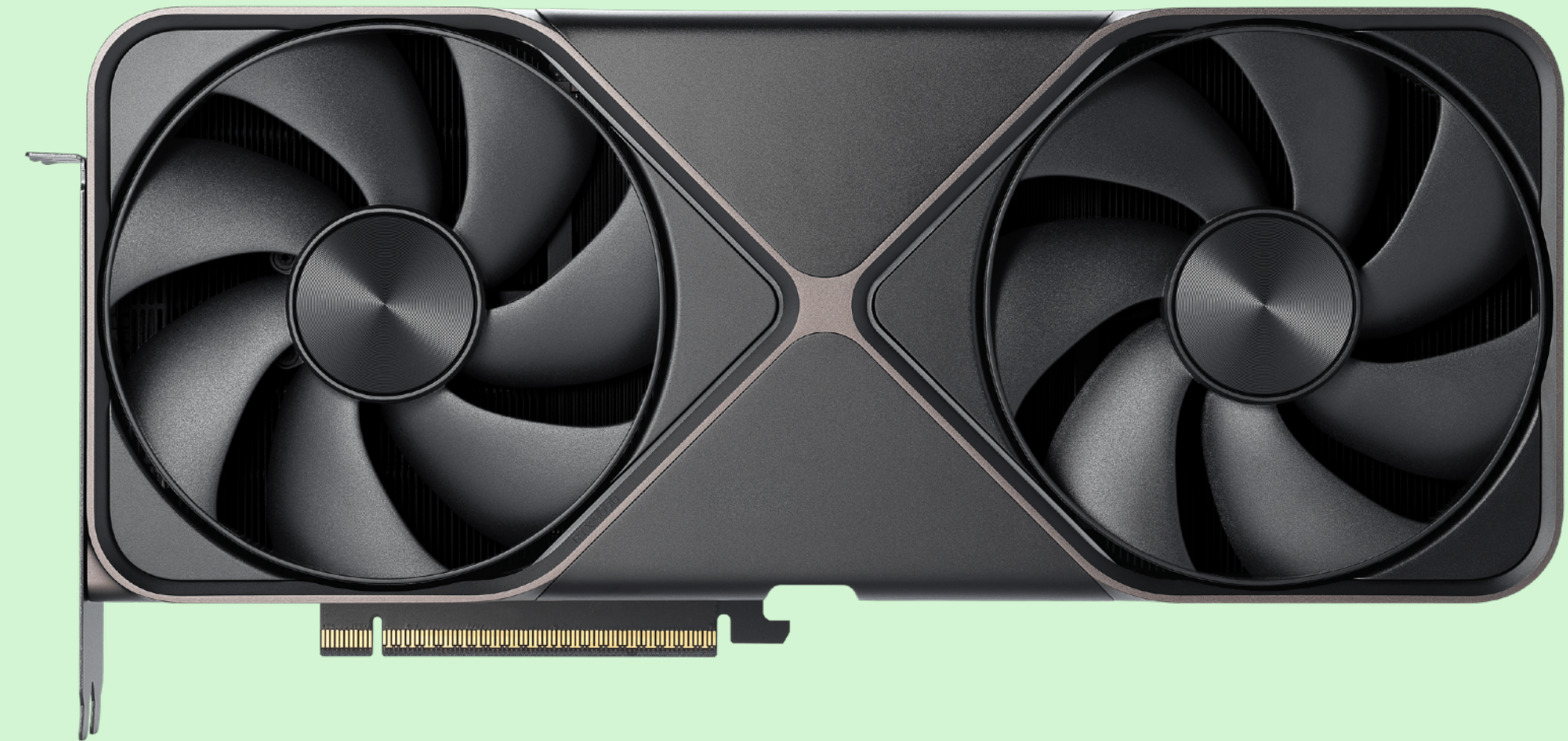


2025 consumer & data center GPUs



AMD RX 7900 XTX (~1K CHF)

(Will present details of this card)



NVIDIA RTX 5090 (~2K CHF)

3D Graphics, 2025 — Unreal Engine



[Source: <https://www.youtube.com/watch?v=Uv5CiqSNf4k>]

3D Graphics, 2025 — Unreal Engine



Rendering a triangle

```
out vec3f color;
```

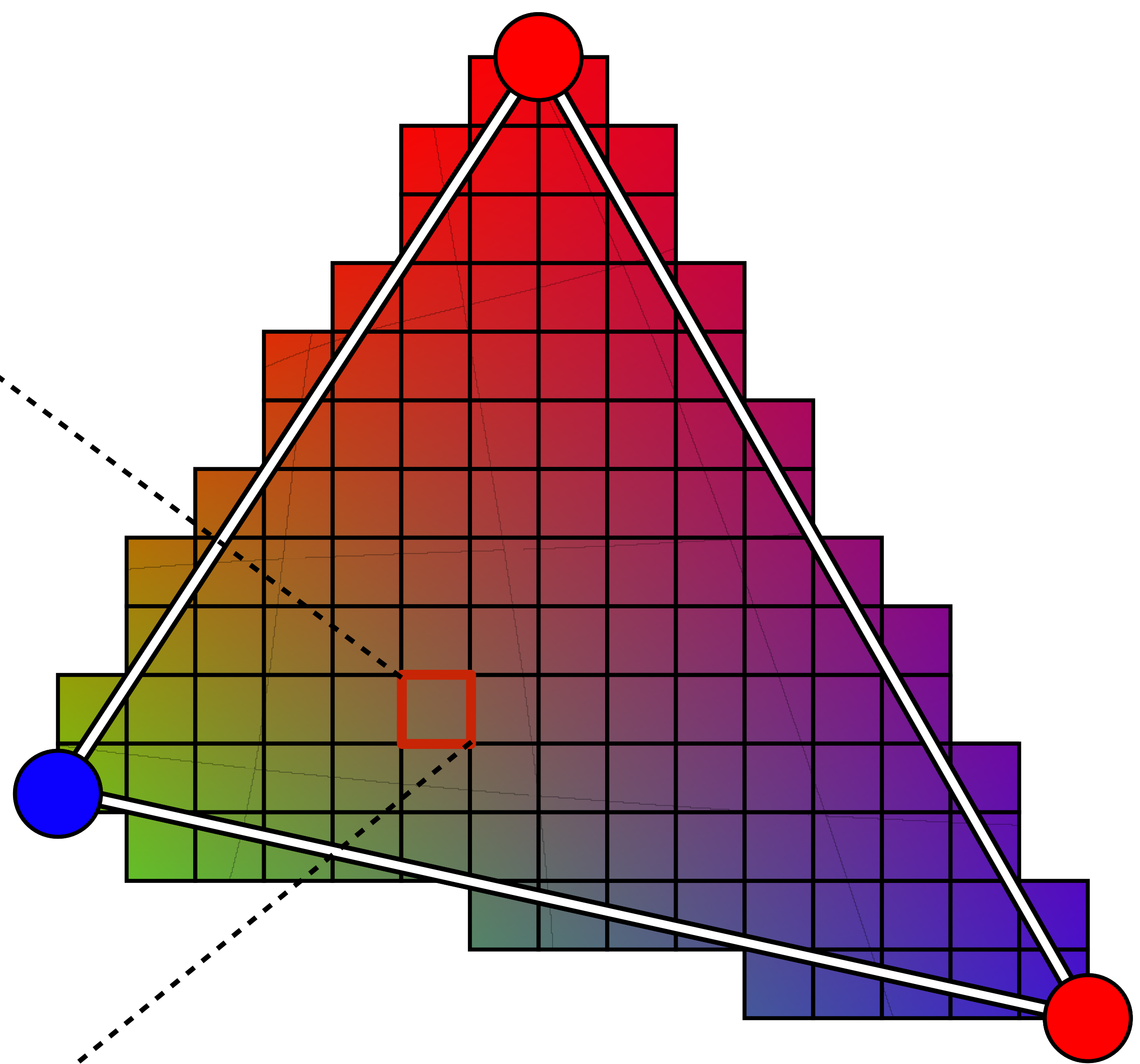
```
void main() {
```

```
    vec3f col = load(u, v);
```

```
    col = col * max(0,  
        dot(light_dir, normal));
```

```
    return color;
```

```
}
```



Graphics is brutal!

- Modern requirements:
 - 4K resolution (3840 × 2160 pixels)
 - 60 frames per second
- That's **~500 million** pixels per second...
 - × # of instructions to produce color
 - Complex *shaders* have many more instructions
 - × overdraw, multiple passes
- Millions of triangles in the scene
- Bottom line: ½ - 1½ **trillion** operations per second!!! 🤯
- Normal processors can't keep up with that.

```
out vec3f color;

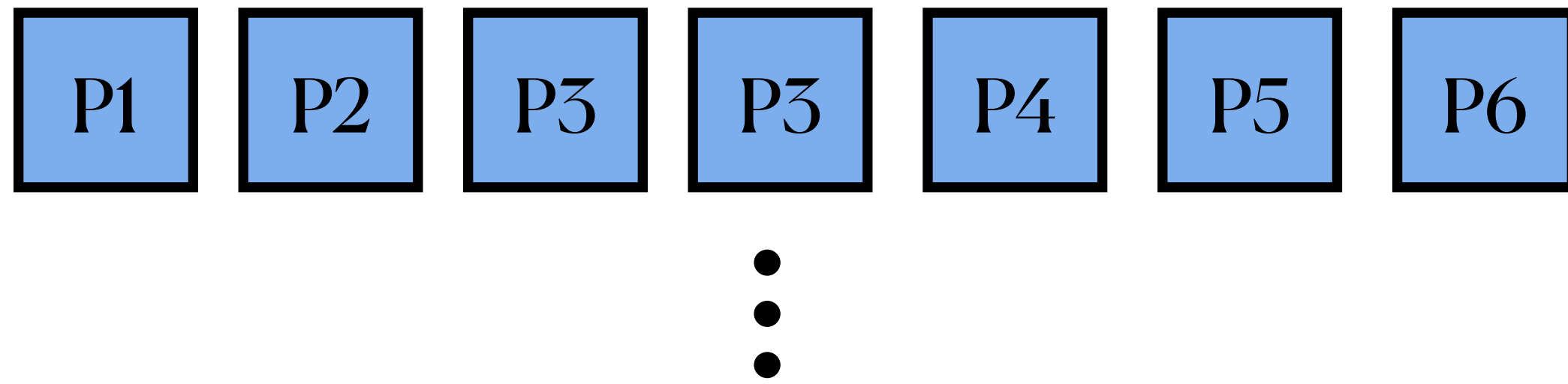
void main() {
    vec3f col = load(u, v);
    col = col * max(0,
        dot(light_dir, normal));
    return color;
}
```



```
LOAD_TEXTURE color, u, v
MUL temp, light_dir.x, normal.x
FMA temp, light_dir.y, normal.y, temp
FMA temp, light_dir.z, normal.z, temp
MAX temp, temp, 0
MUL color.x, color.x, temp
MUL color.y, color.y, temp
MUL color.z, color.z, temp
```

(hypothetical machine instructions)

The origins of GPUs



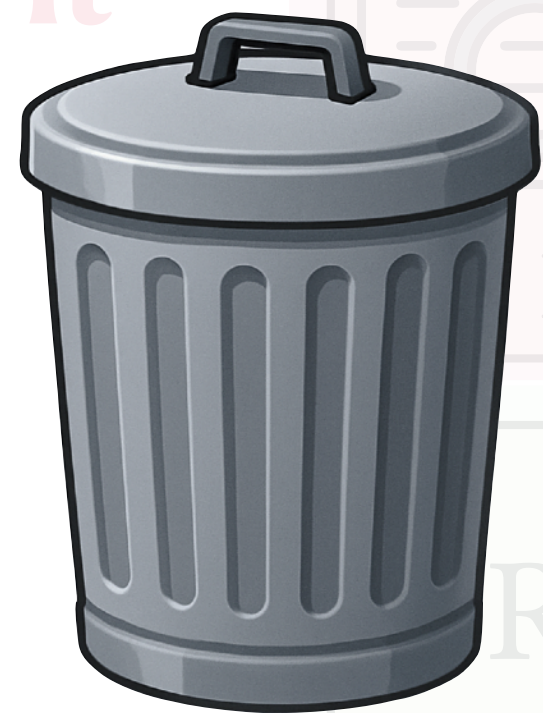
- This is an "embarrassingly parallel" problem.
- Let's compute all the pixels in parallel

- *Problem:* CPU-style processors are **too big**, can only have a few.

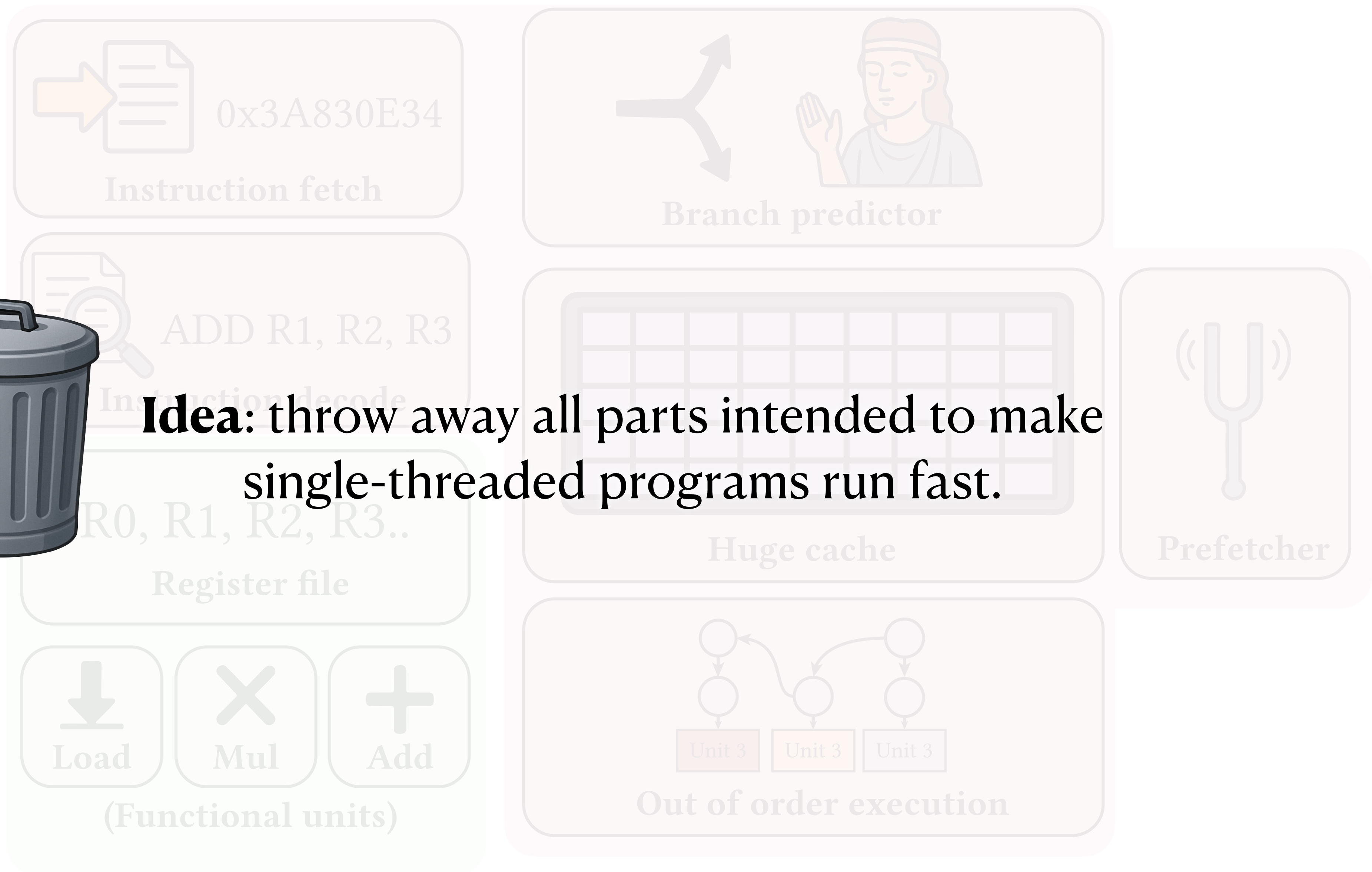
Will have to make some tough compromises..

Review: CPU cores

(What we *had to do* to make it fast)

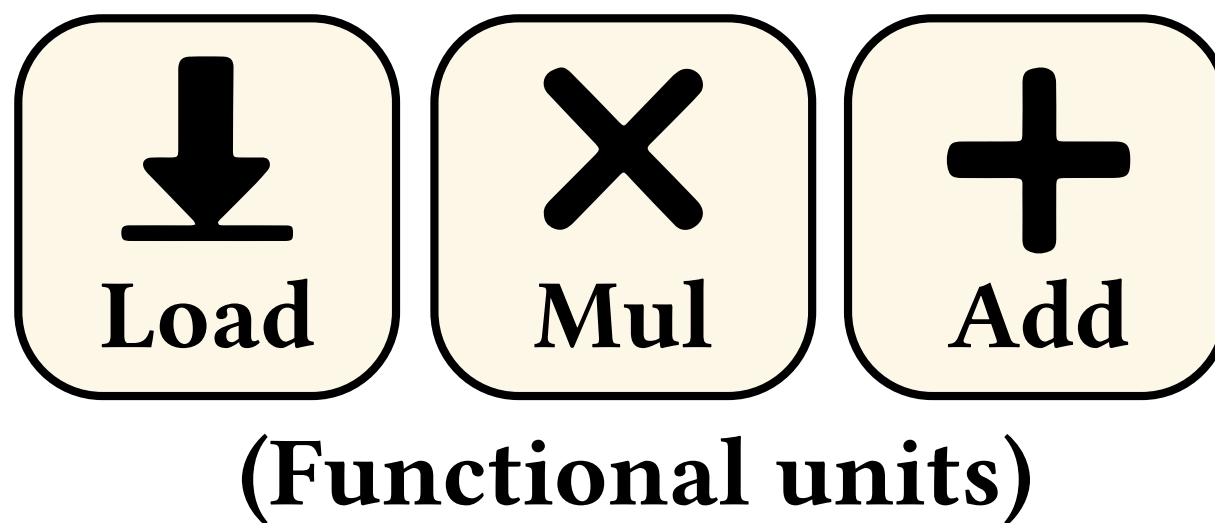
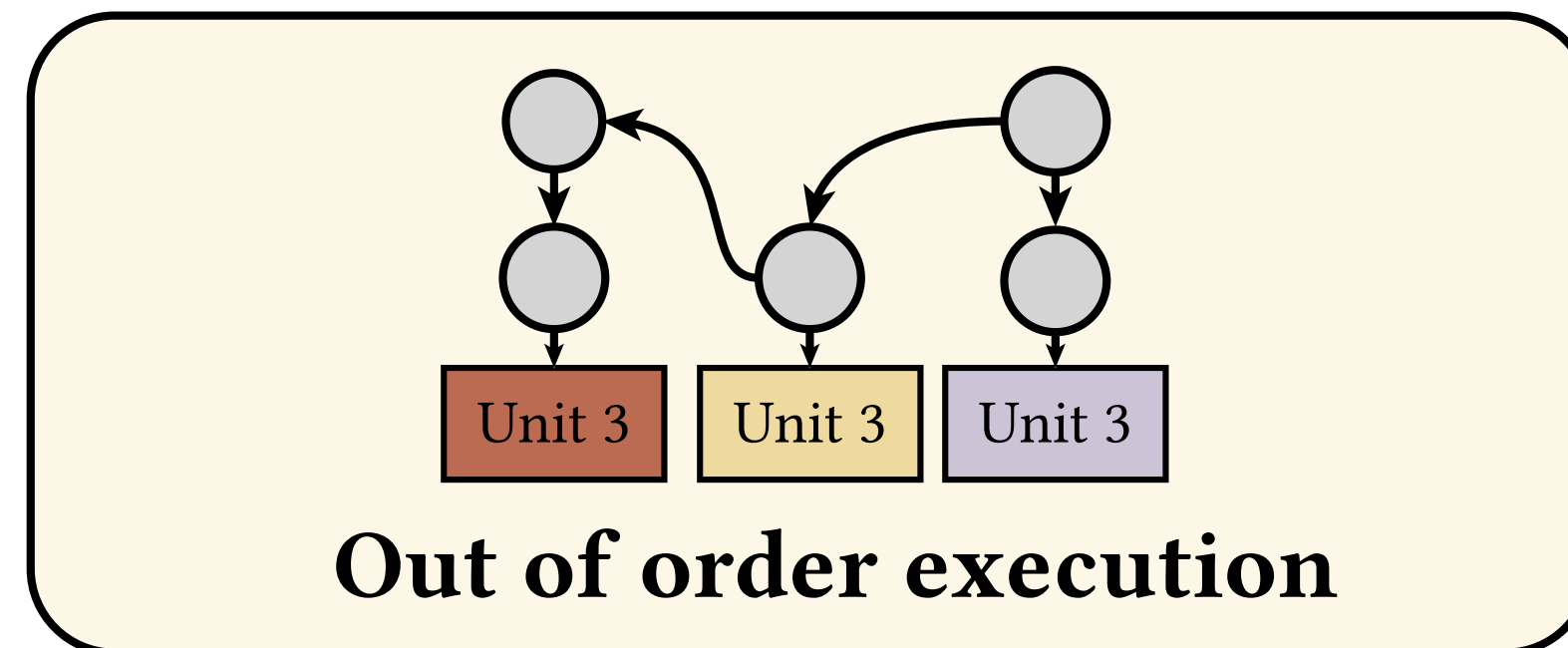
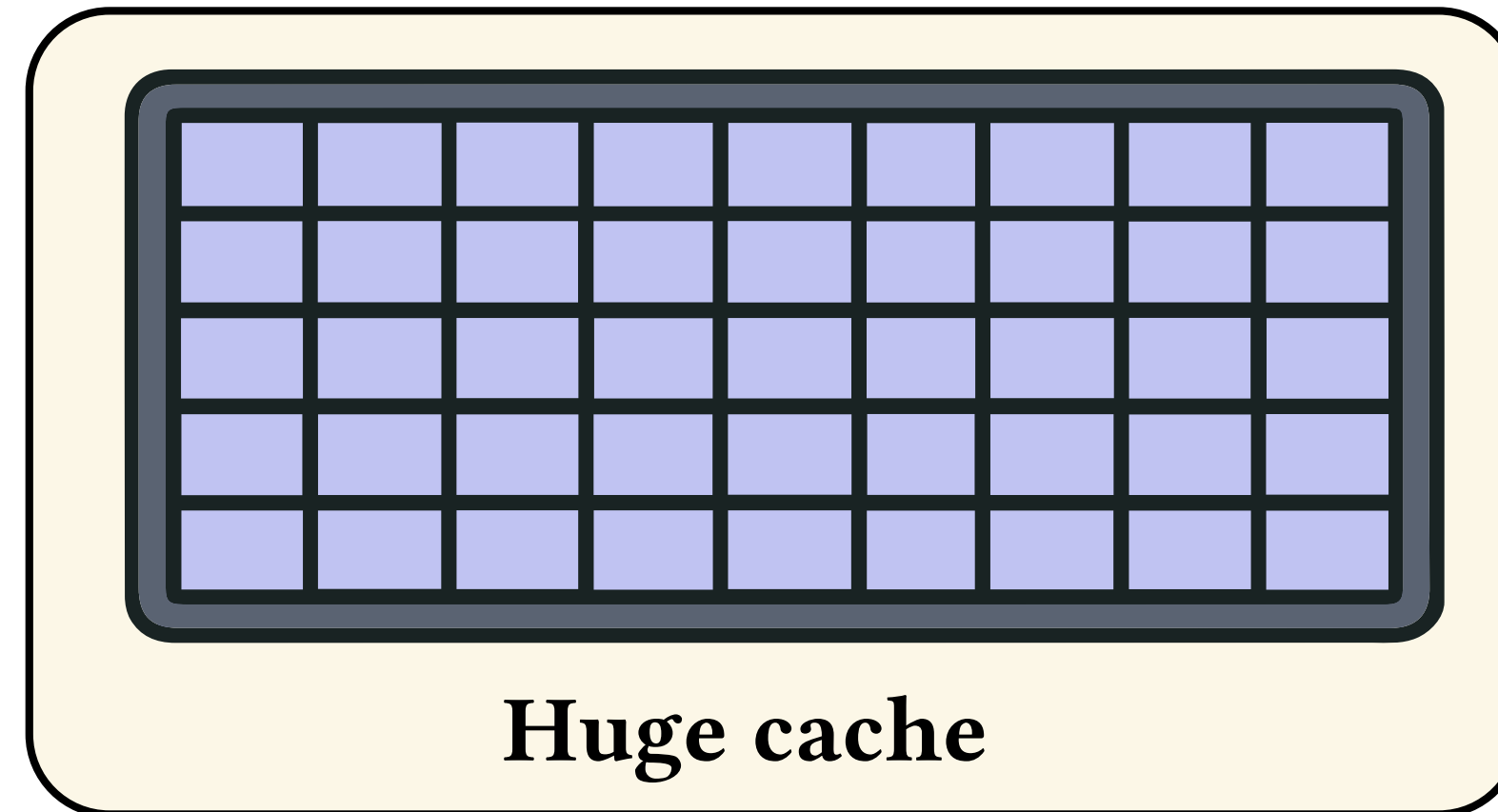
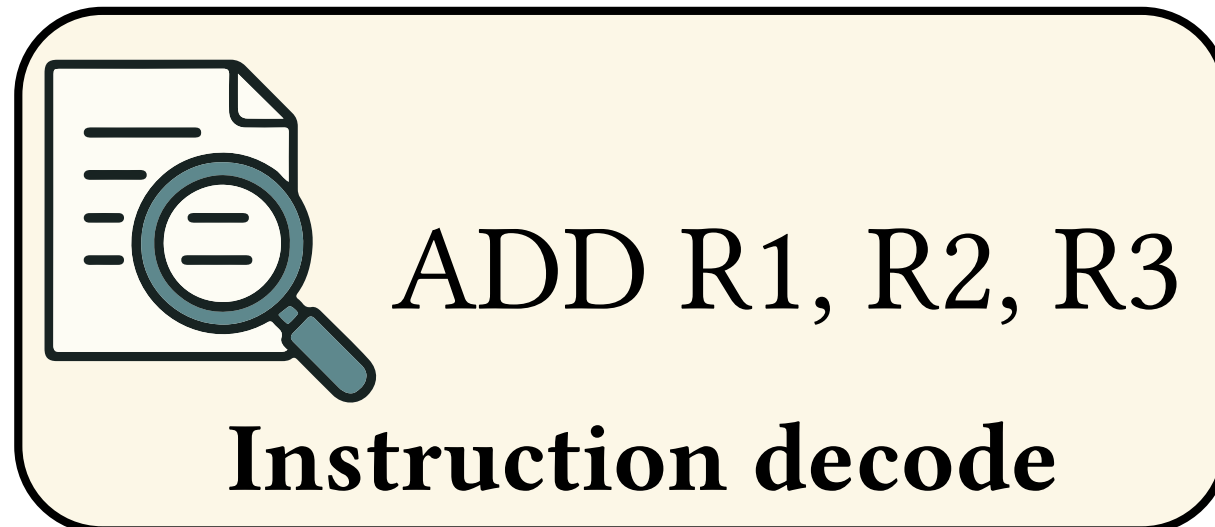
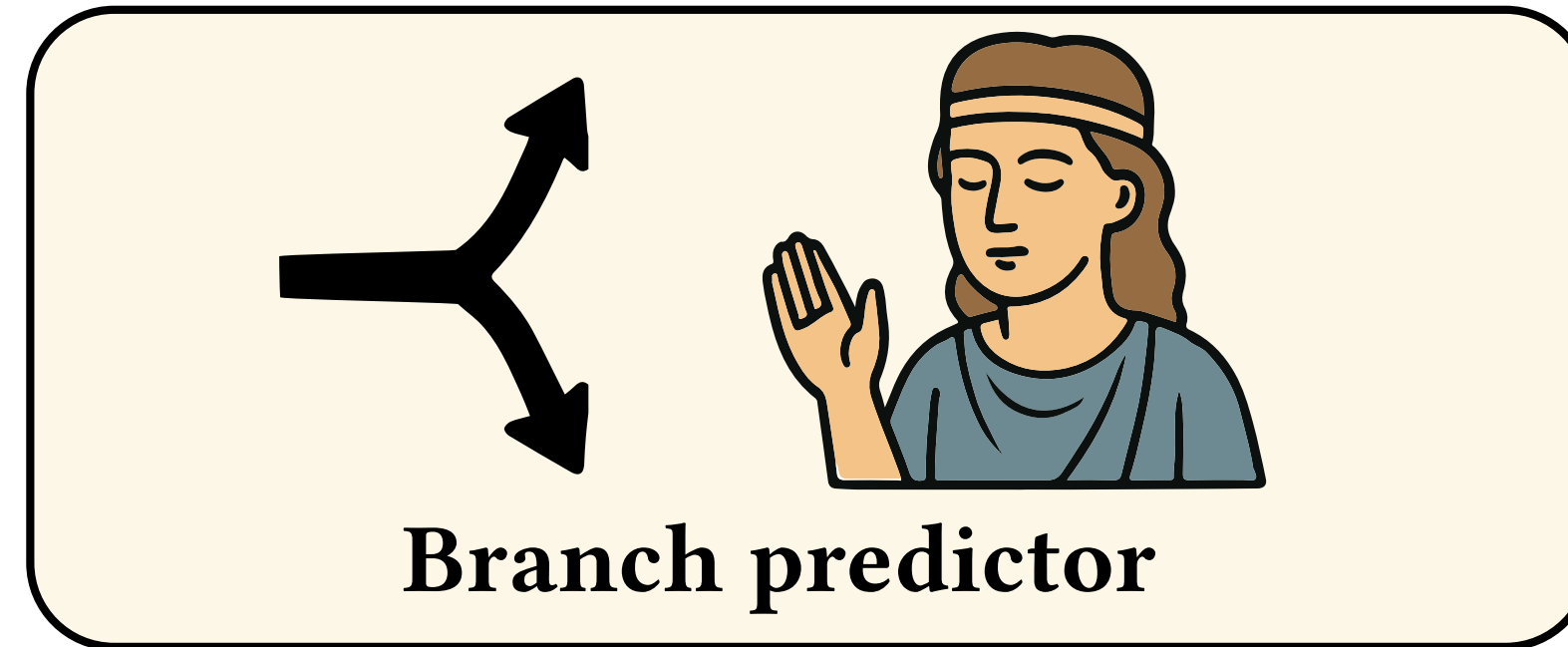
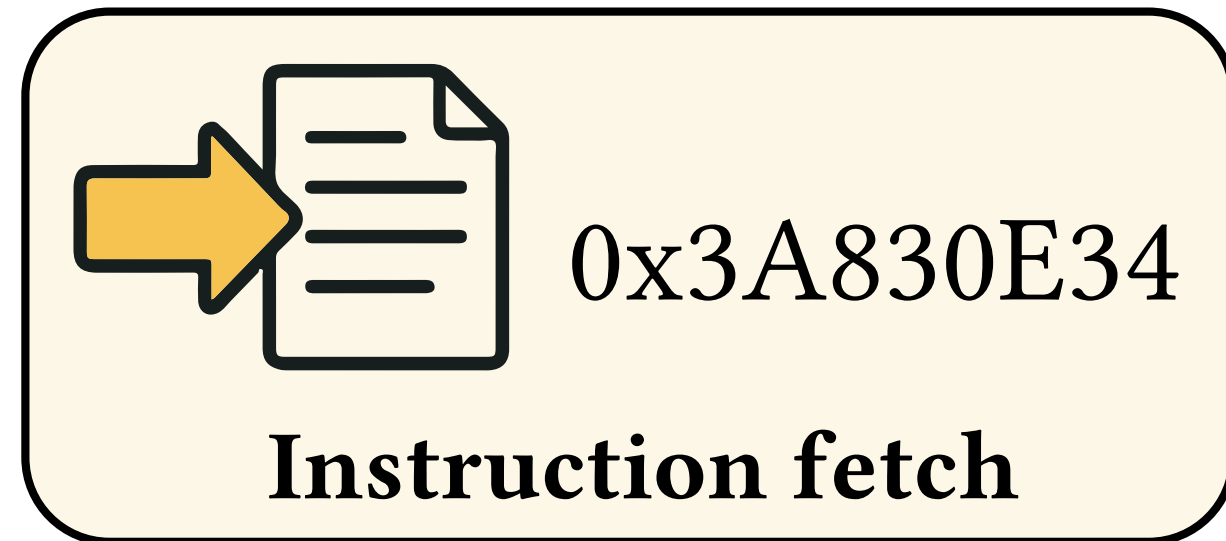


(What we *want to do*)



Idea: throw away all parts intended to make single-threaded programs run fast.

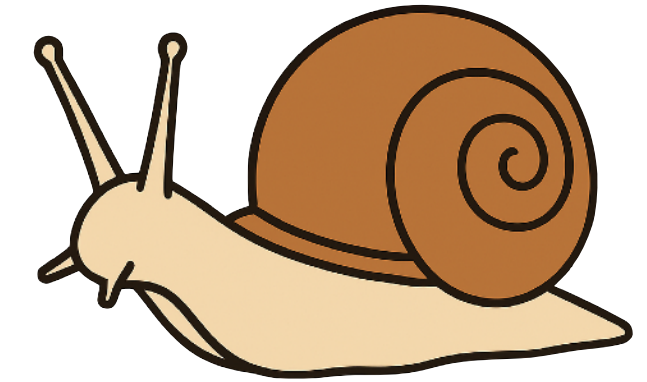
From CPU → GPU



Implications:

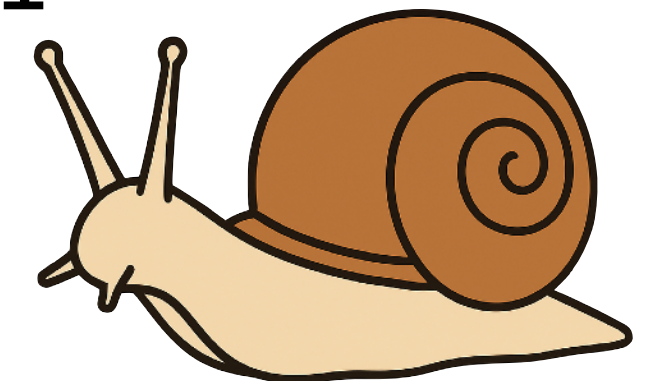
Without branch predictor:

```
IF R0 == 0  
▶ JUMP [ADDR]  
....
```

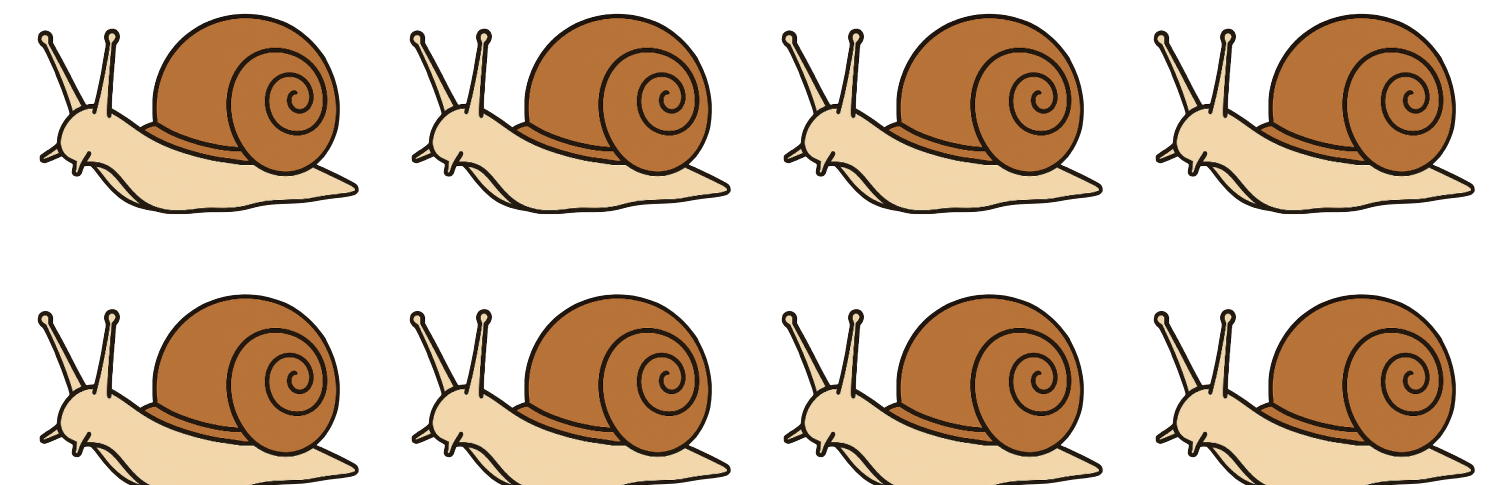


Without out of order execution:

```
LOAD R0, [ADDR]  
▶ ADD R0, R0, 1  
...
```

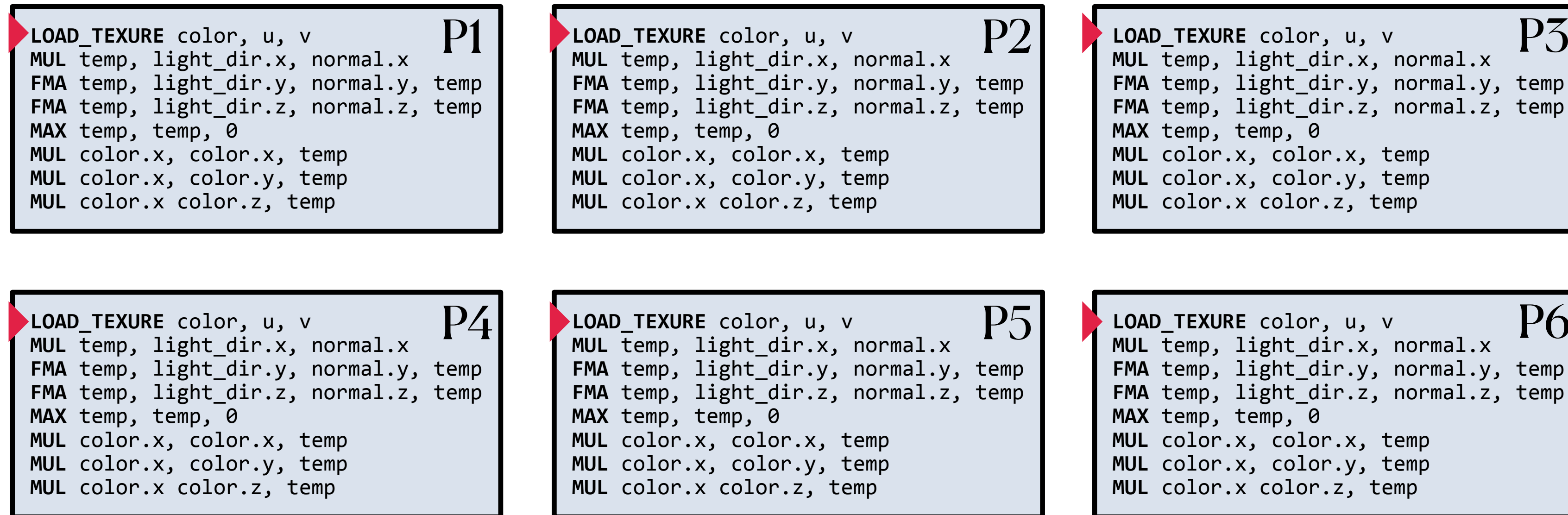


Without big, fast caches:




From CPU → GPU


- Let's compute all the pixels in parallel



- All processors are evolving in lockstep. Let's exploit this in the architecture.

From CPU → GPU

 0x3A830E34
Instruction fetch

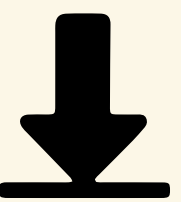
 ADD R1, R2, R3
Instruction decode

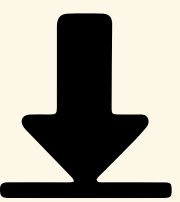

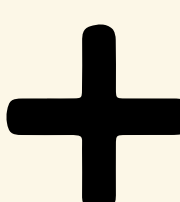
R0, R1, R2, R3..
Register file

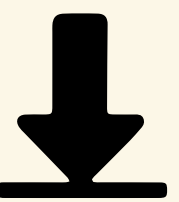

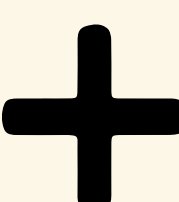
R0, R1, R2, R3..
Register file

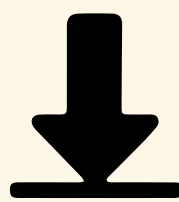
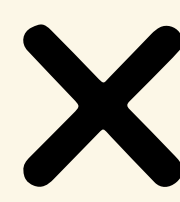

R0, R1, R2, R3..
Register file

R0, R1, R2, R3..
Register file

  
Load Mul Add

  
Load Mul Add

  
Load Mul Add

  
Load Mul Add

(Functional units)

(Functional units)

(Functional units)

(Functional units)

R0, R1, R2, R3..

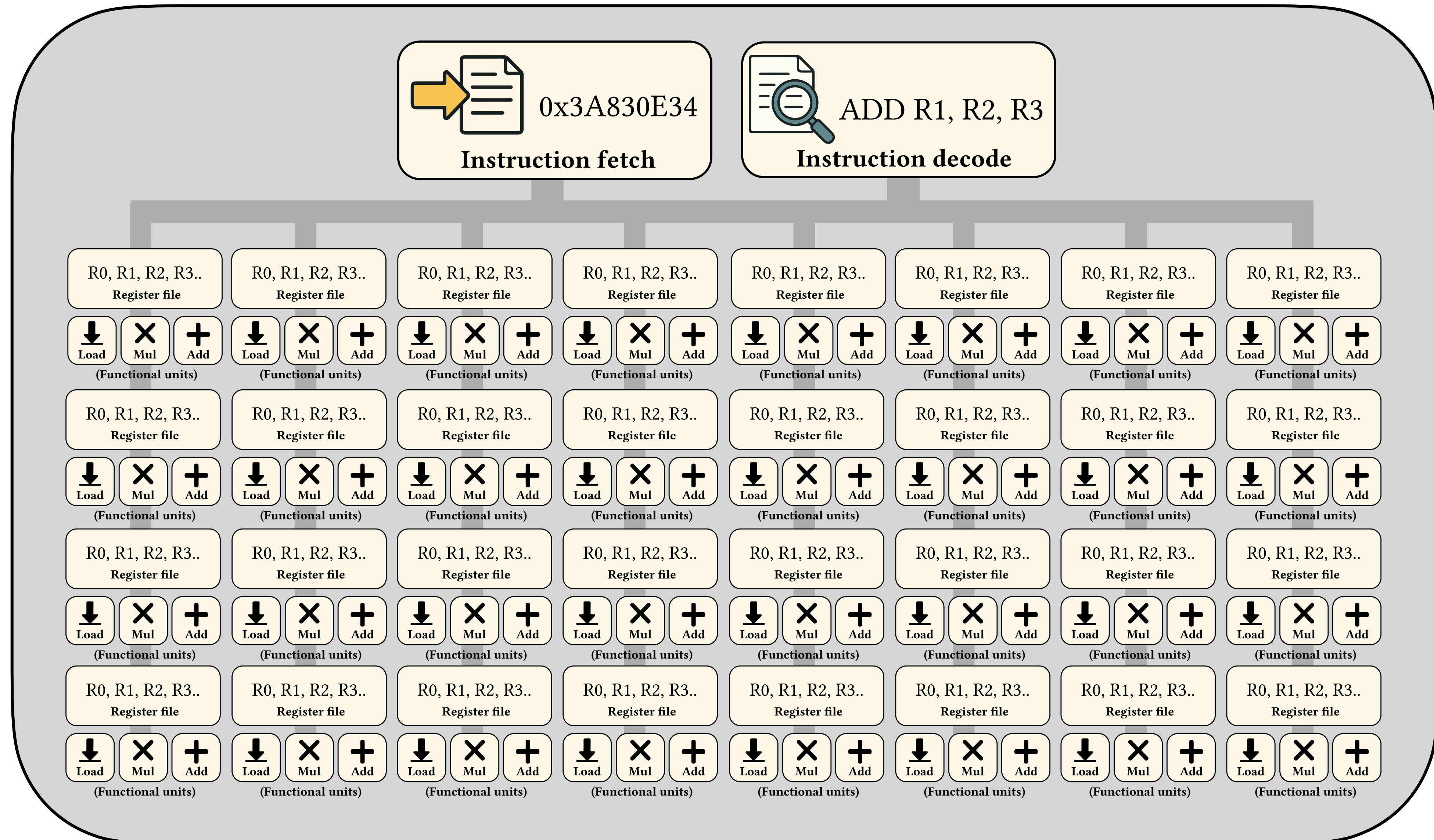
R0, R1, R2, R3..

R0, R1, R2, R3..

R0, R1, R2, R3..

From CPU → GPU

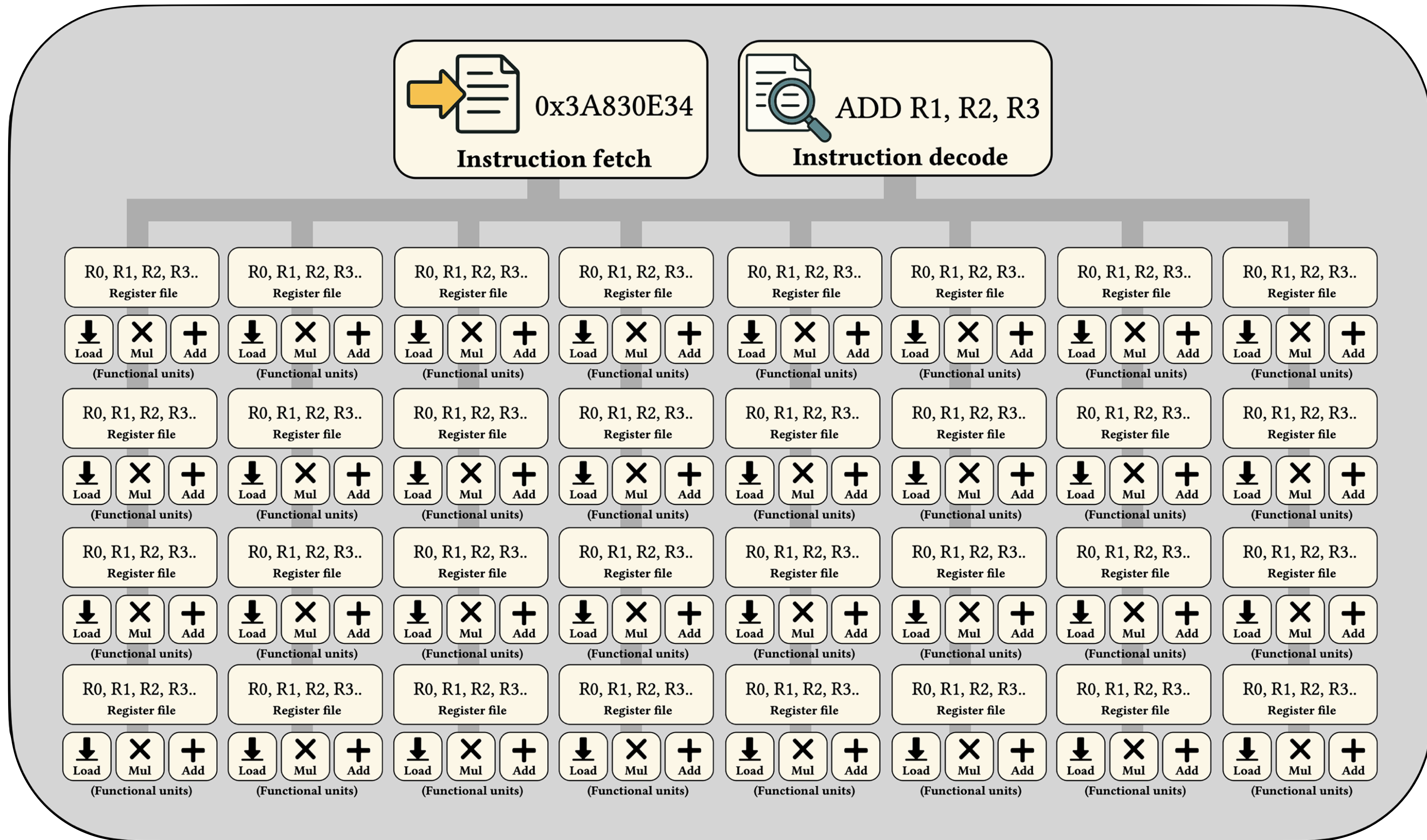
Idea: 32 tiny "processors" sharing fetch + decode.
Similar to vector operations on CPUs.



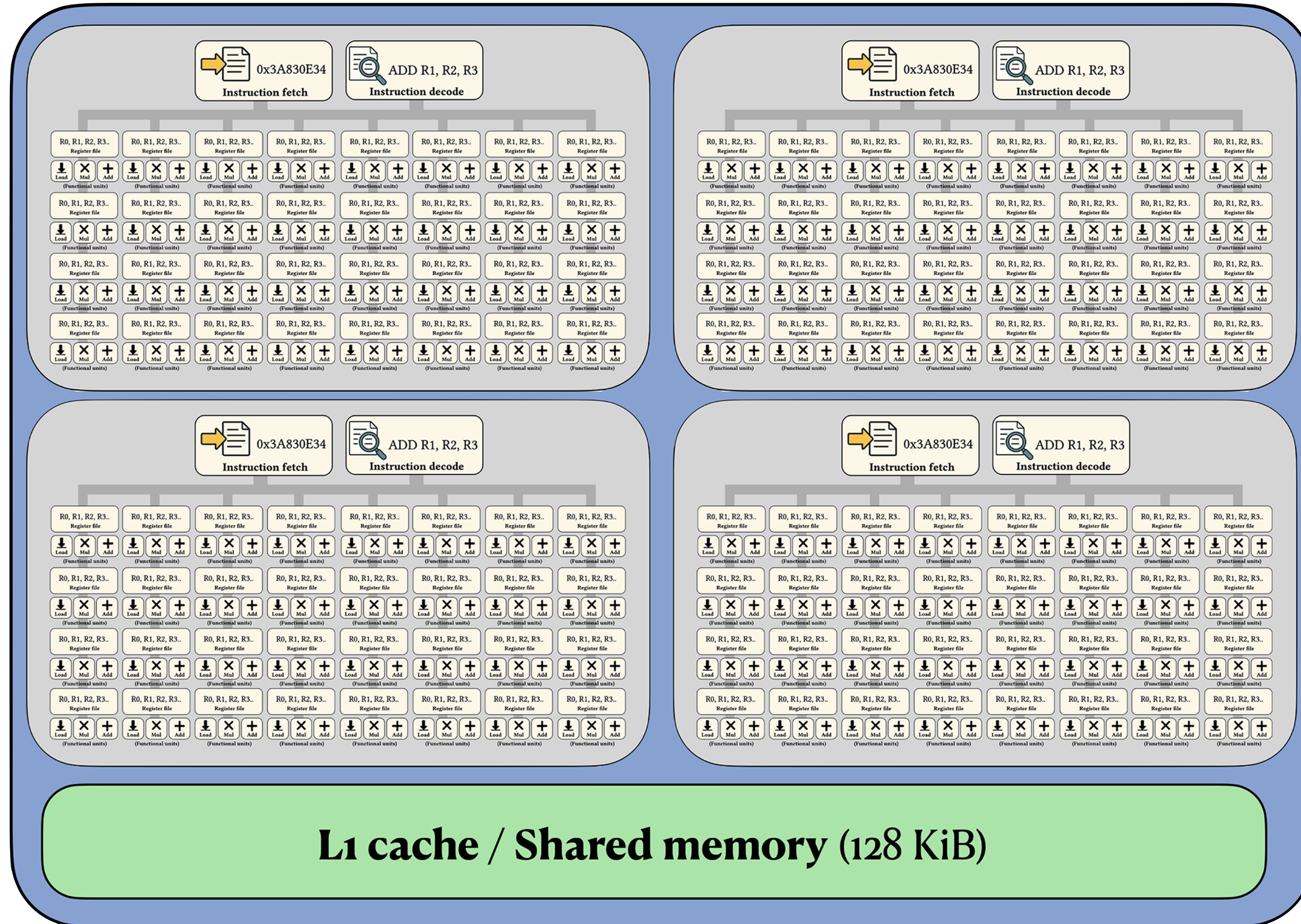
From CPU → GPU

"SM Partition"

(SM = "Streaming Multiprocessor")

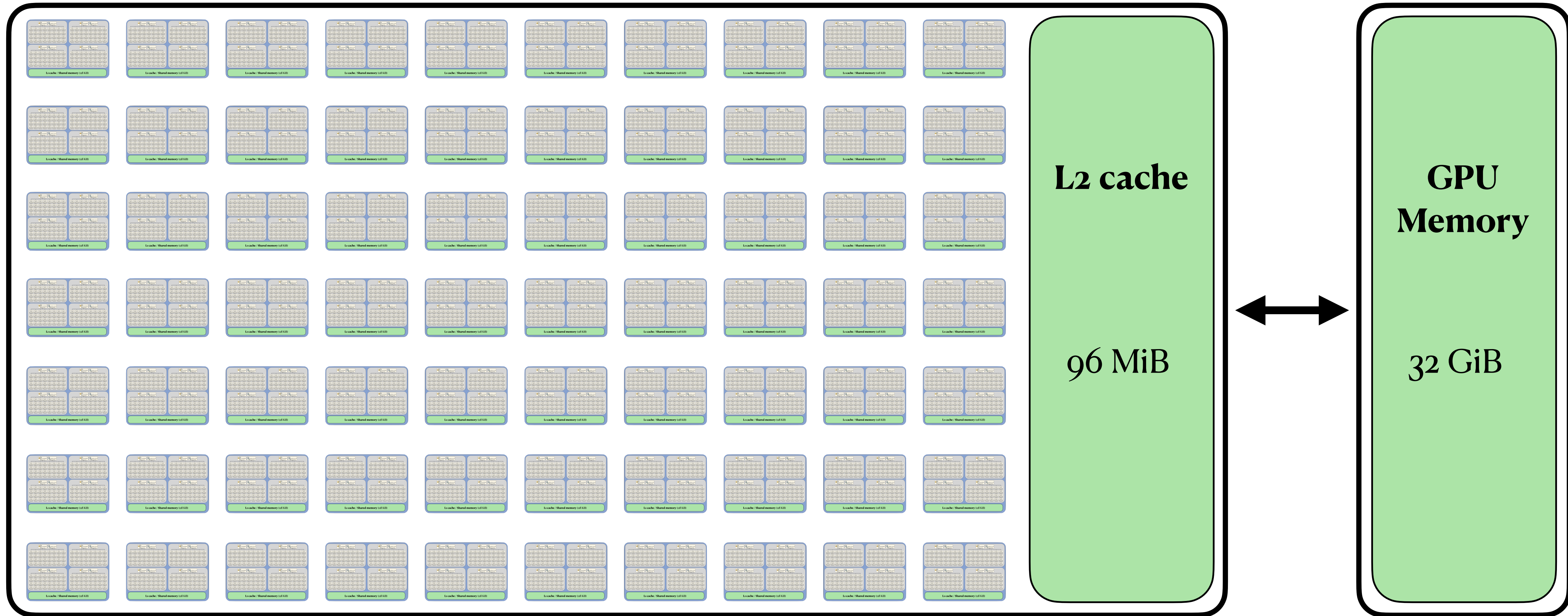


From CPU → GPU Streaming Multiprocessor (SM)



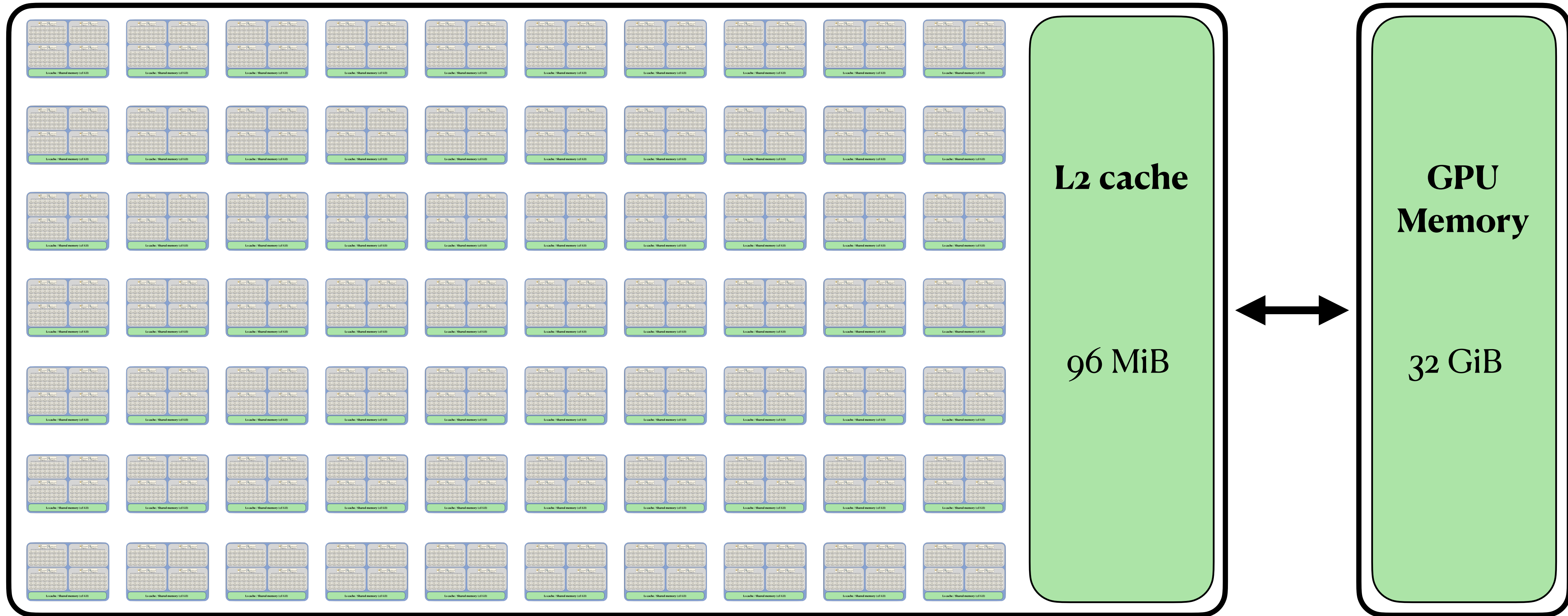
From CPU → GPU

170 SMs × 4 partitions × 32 processing elements = 21'760 "cores"



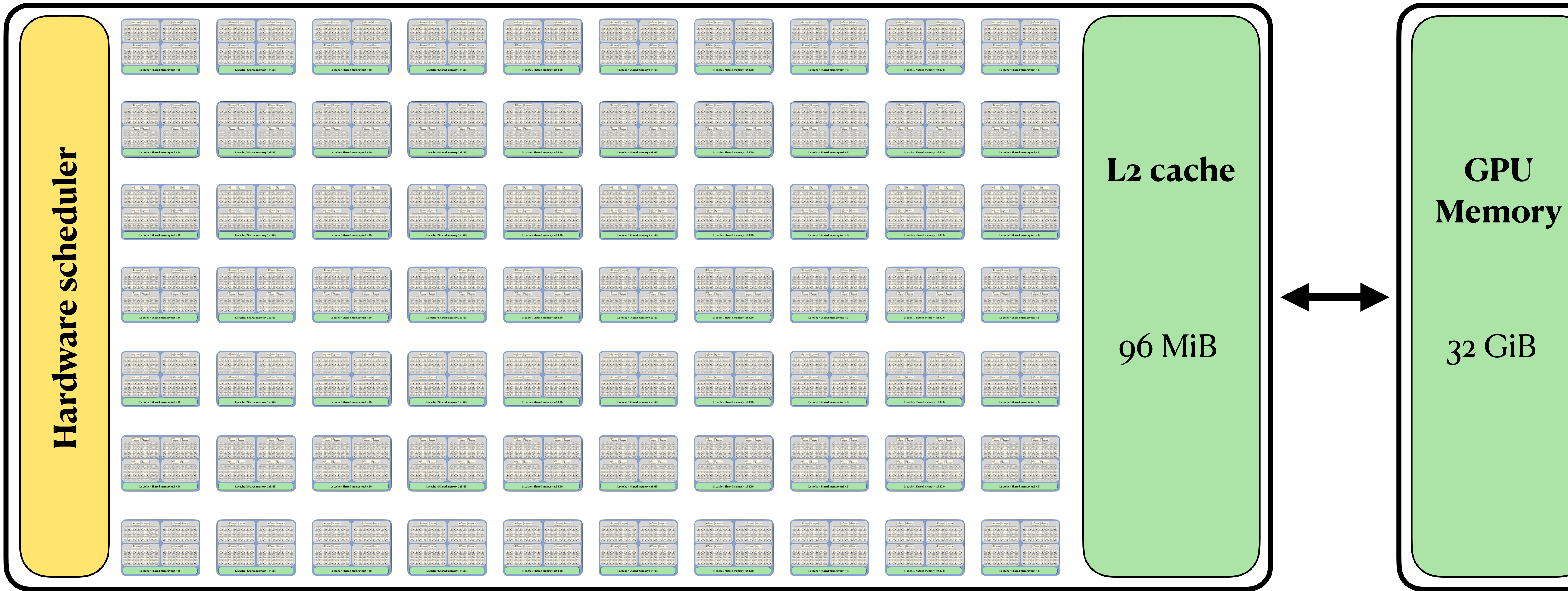
From CPU → GPU

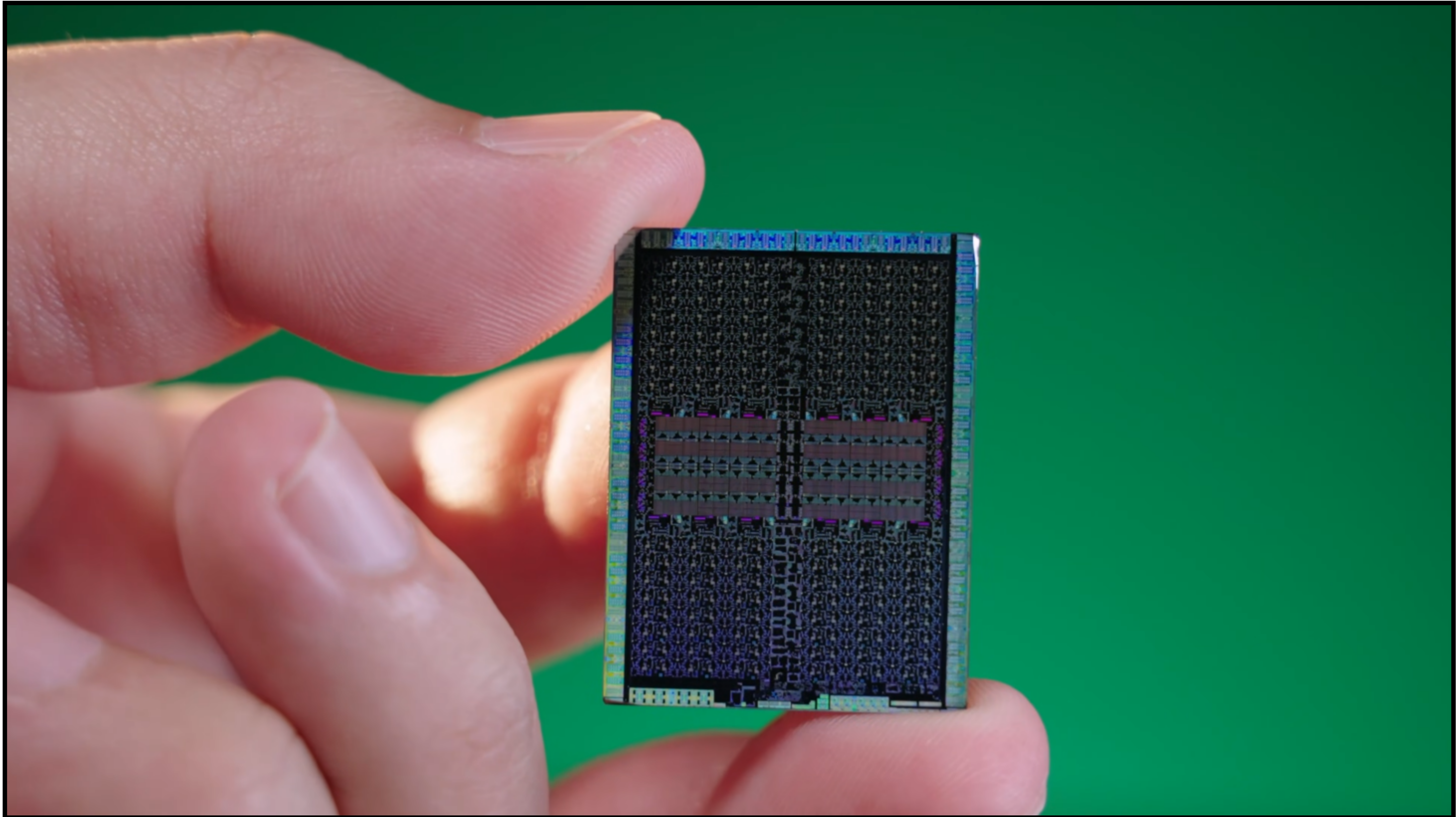
170 SMs × 4 partitions × 32 processing elements = 21'760 "cores"



From CPU → GPU

170 SMs × 4 partitions × 32 processing elements = 21'760 "cores"

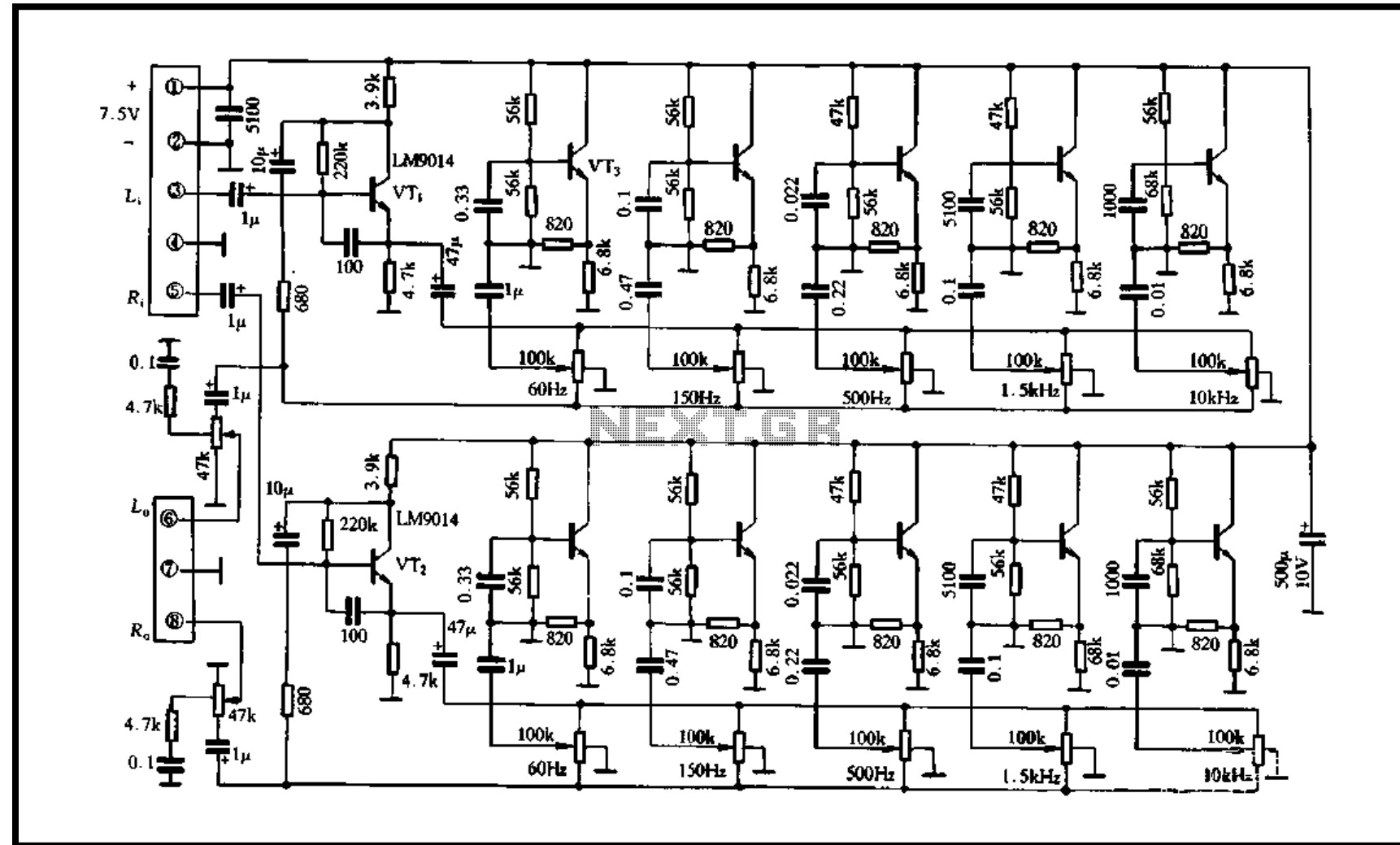




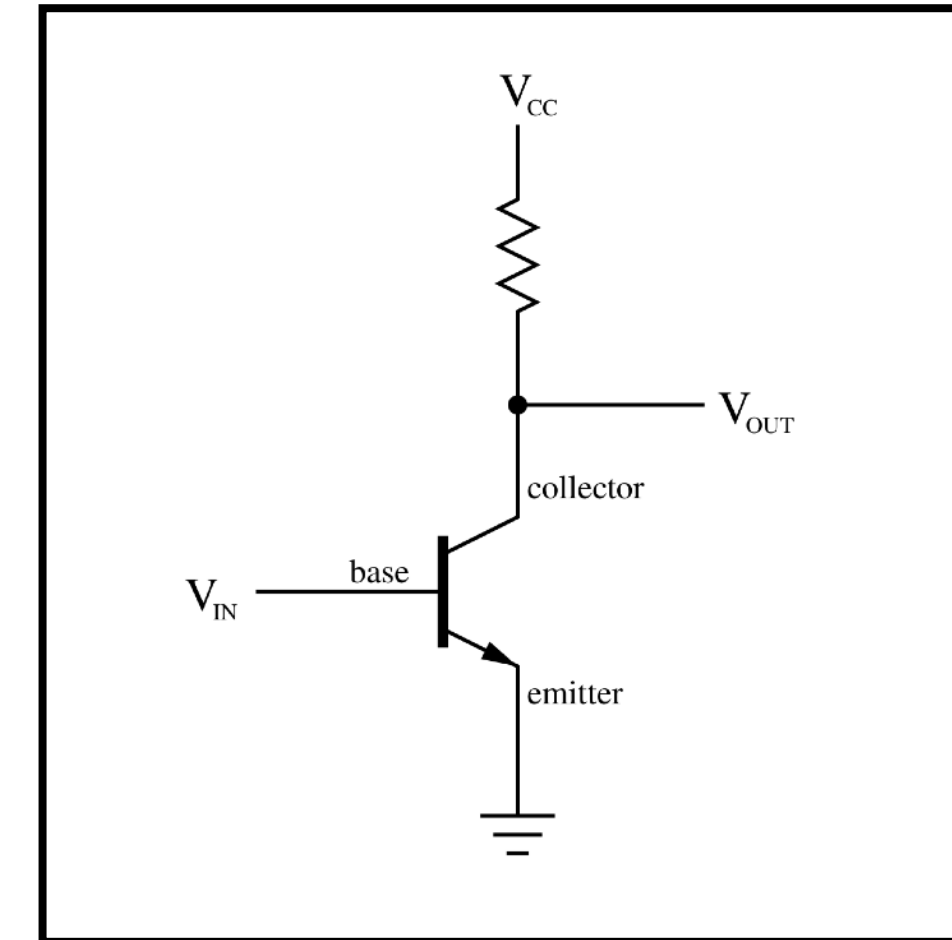
GeForce RTX 5090's Blackwell GB202 chip, [[Source](#)]

Design choices

Fast, complex circuit

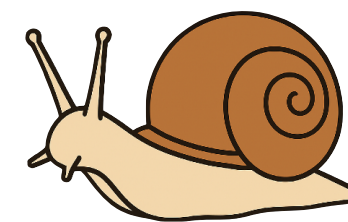


Slow, simple circuit



VS.

Latencies of different operations (approximate, worst case)



RTX 5090

Intel Core Ultra 9 285K

Clock frequency

2.4 Ghz

5.7 Ghz

Load (L1 hit)

30 cycles

4 cycles

Load (L2 hit)

200 cycles

17 cycles

Load (Miss)

800 cycles

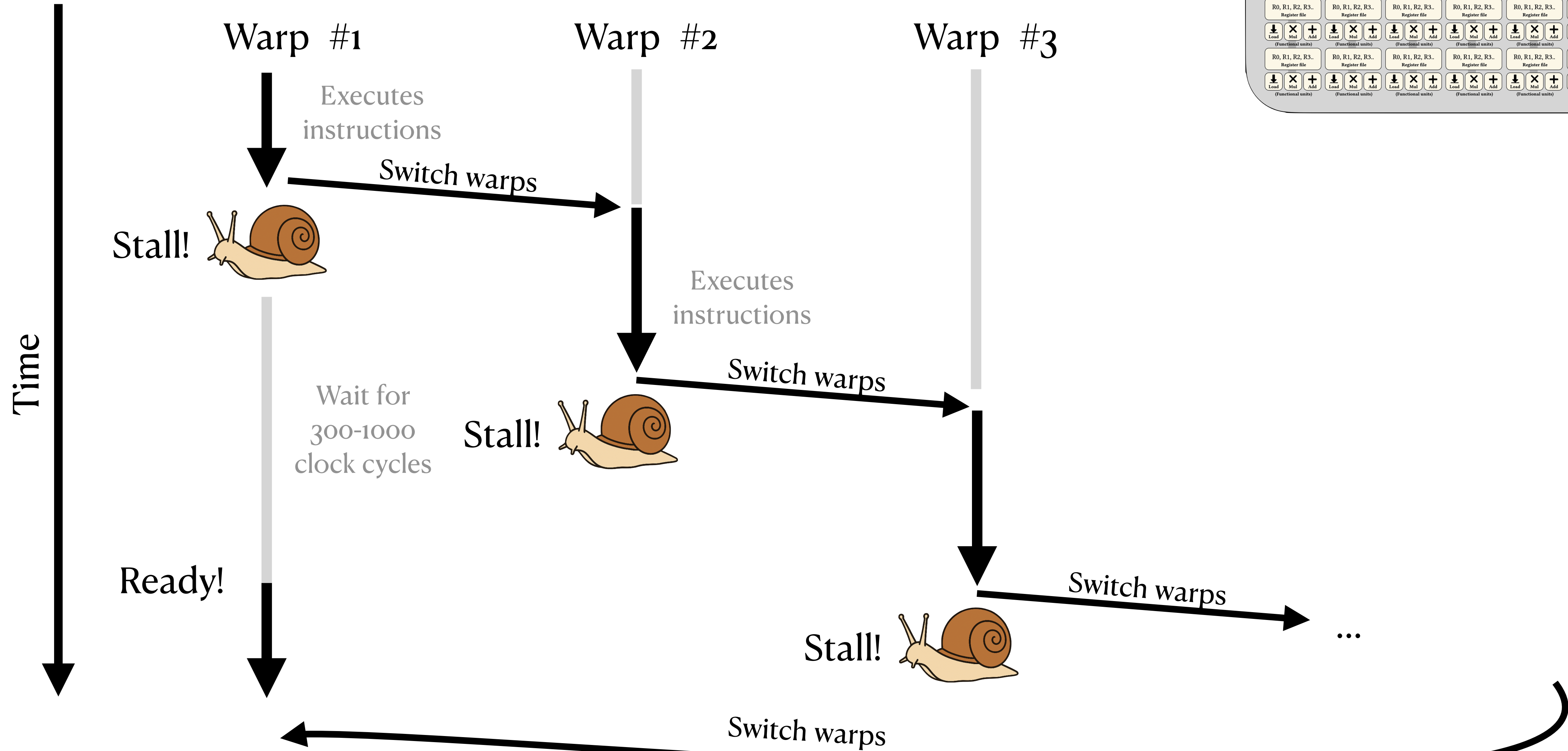
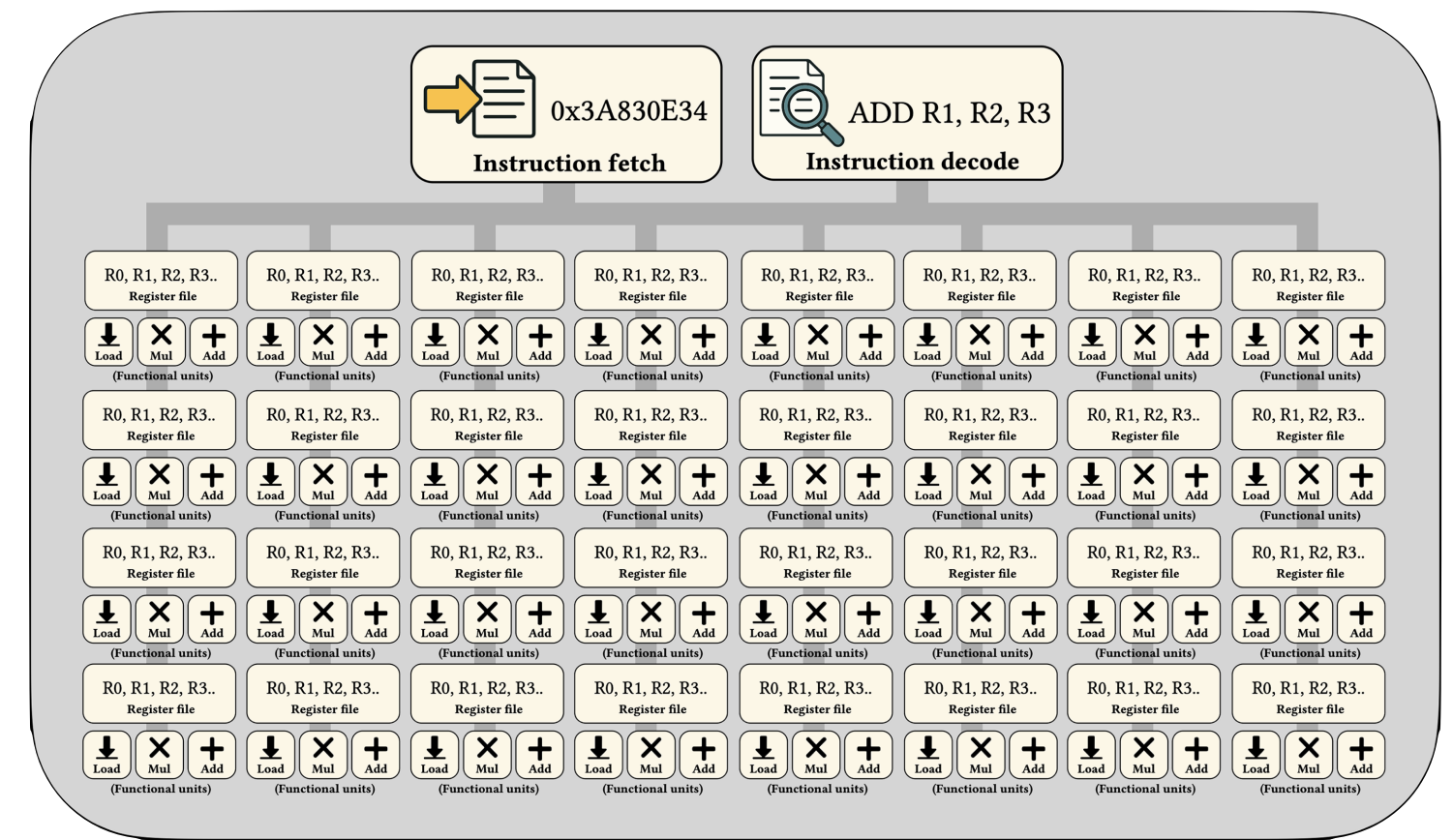
565 cycles

Is that it?

Just a lot of small, slow processors?

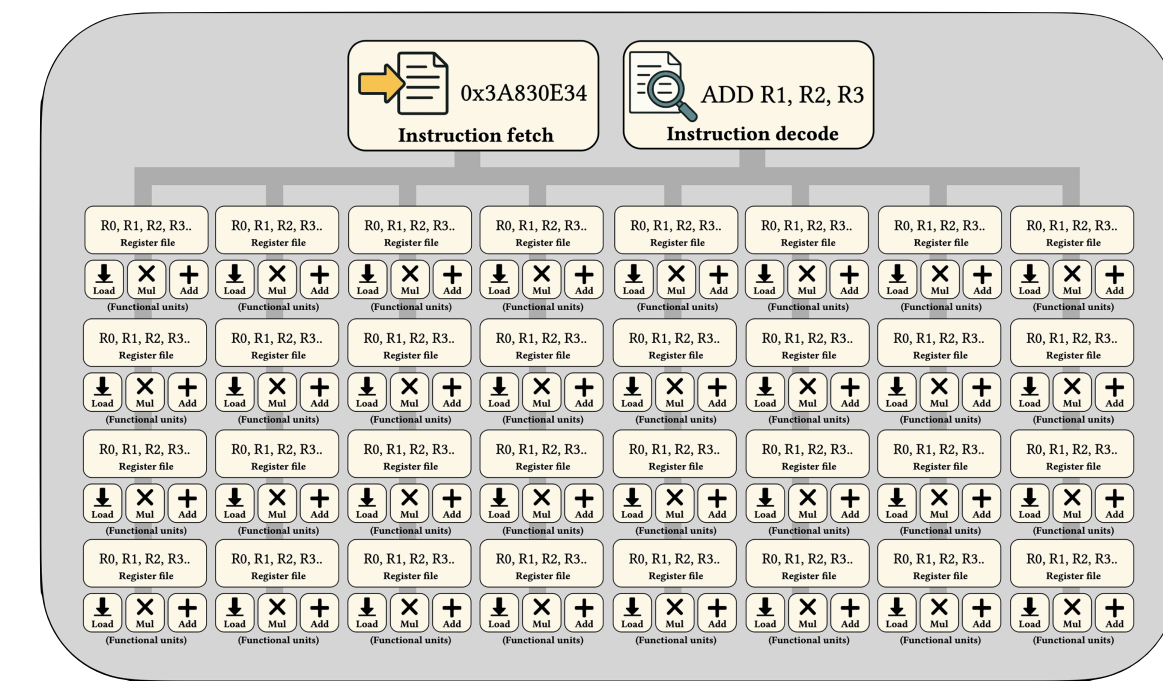
Stalls

SM partition executes "warps" (group of 32 threads)



Part 1: Latency hiding

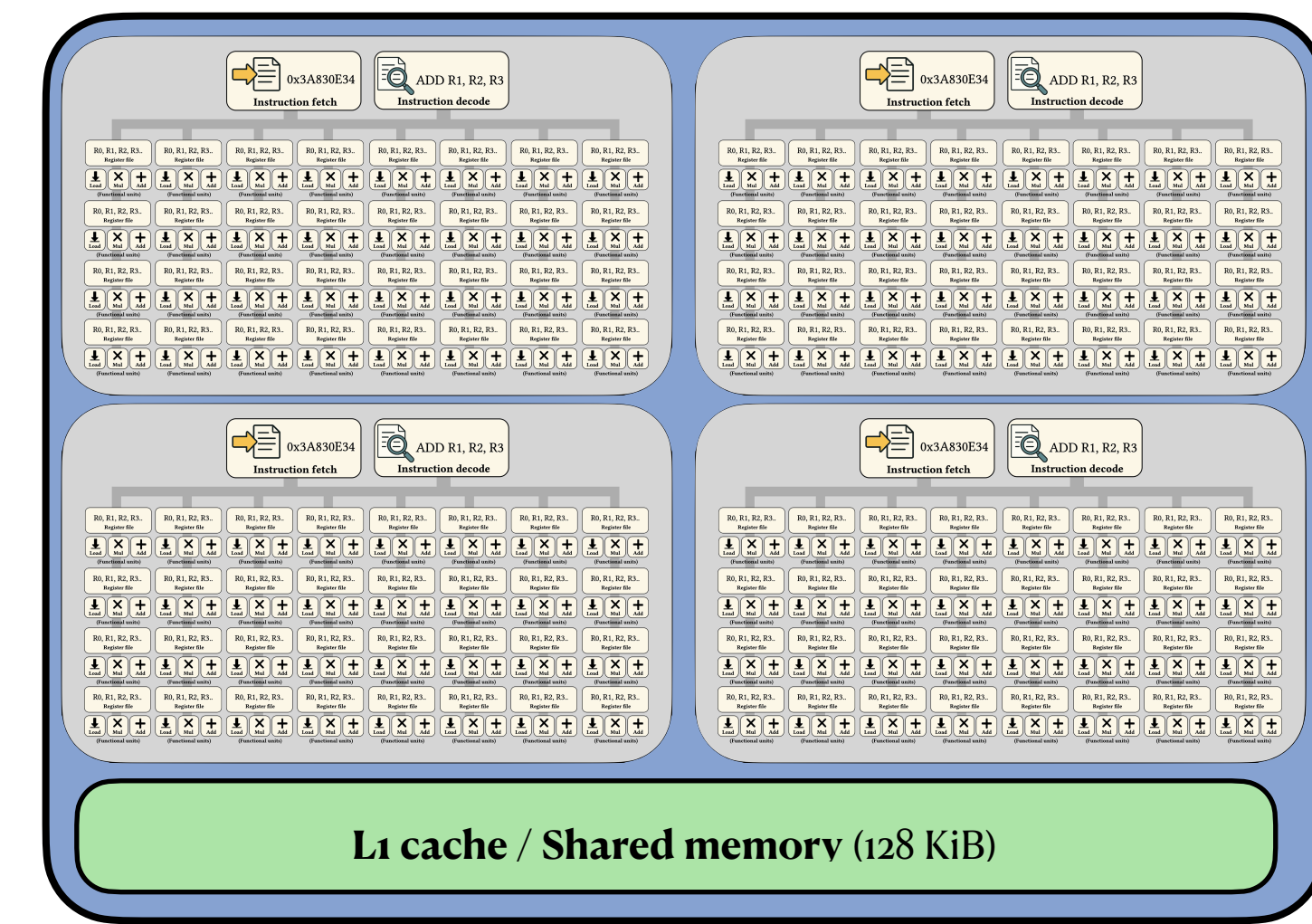
- Because the cores on GPUs are so basic, they will **stall all the time**.
 - Latencies from stalls are *terrible* for performance.
- Stalls are unavoidable. But GPUs can usually "hide" this latency.
- Key idea: run **many** warps per SM partition.
 - Switch to another warp whenever a stall happens.
 - Switching is "for free" (consumes no clock cycles).
 - If all goes well: do useful work at every clock cycle.
- Follows high level pattern: optimize **throughput**, not latency.



Huge register file

Part 2: shared memory

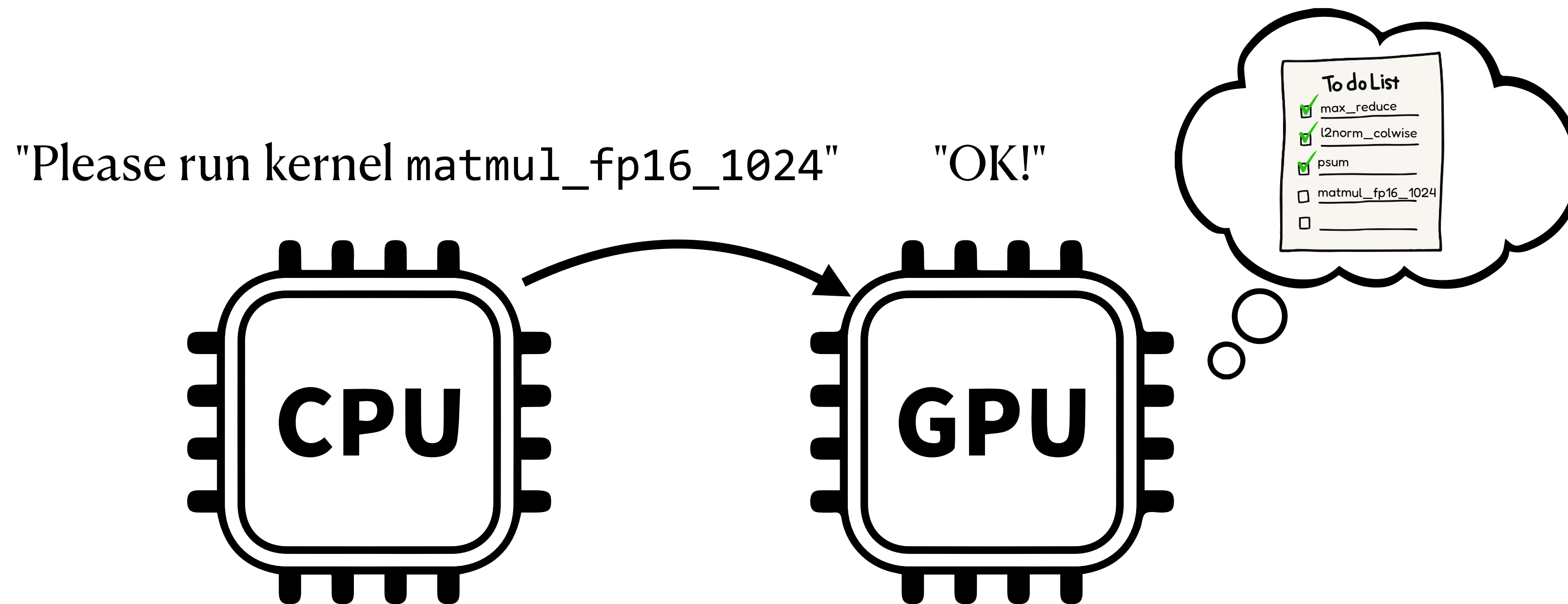
- Most programs read and write a lot of memory
- On GPUs, this means that they will stall constantly.
- Solution: use **shared memory**.
 - Copy a small part of the problem into local ("shared") memory
 - Do a **lot of work** with this copy, which is fast to access.
 - Then write back the result.



From CPU → GPU

GPUs are "compute accelerators"

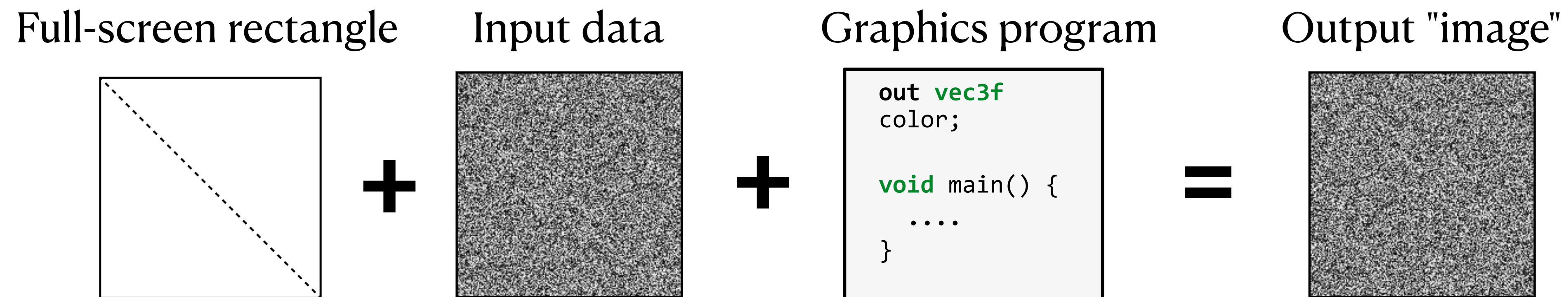
- You cannot run a browser or an operating system on a GPU.
- The GPU merely helps to accelerate tasks. Its basic computational unit is called a **kernel**



- A **kernel launch** consists of **code** + a **launch grid**
(The grid specifies how many instances of the program to launch, e.g, 128 x 128 x 32 times).

GPUs for General Purpose Computing

- Originally, GPUs could only render triangles. Main use: **computer games!**
- People soon realized that their architecture is *interesting* for non-graphics usage.
- Hacky workarounds:



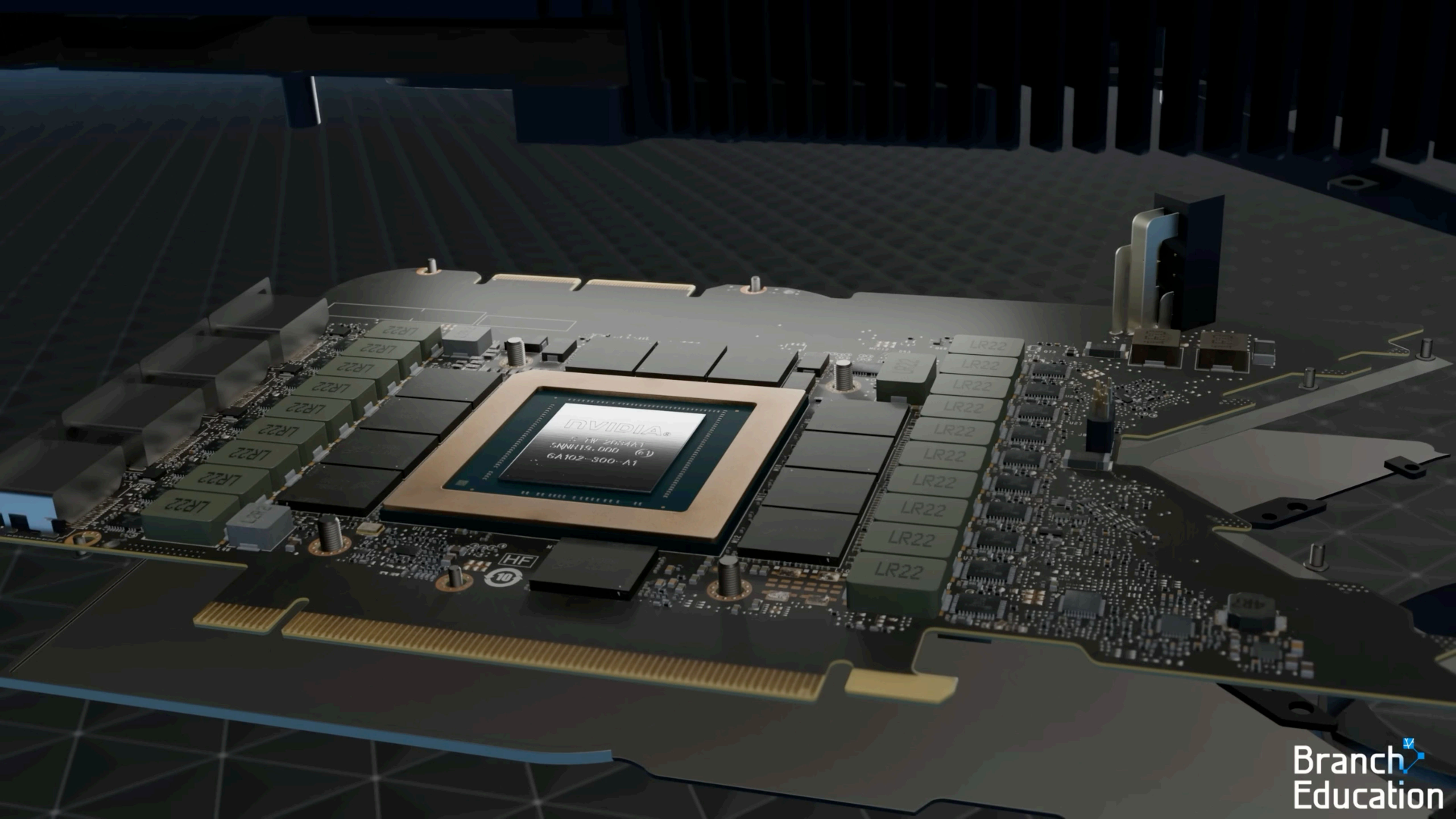
- Details: *Brook for GPUs: Stream Computing on Graphics Hardware [Buck et al. 2004]*
- Today's frameworks make GPU programming easier:
CUDA (NVIDIA), **ROCm** (AMD), **SYCL** (Intel)

Demo time

Recommended: How do GPUs work



Link: <https://www.youtube.com/watch?v=h9Z4oGN89MU>



Developments in the last 5 years

HOME > NVDA · NASDAQ

NVIDIA Corp

\$179.83 ↑ 1,202.17% +166.02 5Y

After Hours: **\$180.69** (↑ 0.48%) +0.86

Closed: Oct 15, 6:03:51 PM UTC-4 · USD · NASDAQ · Disclaimer

1D 5D 1M 6M YTD 1Y 5Y MAX

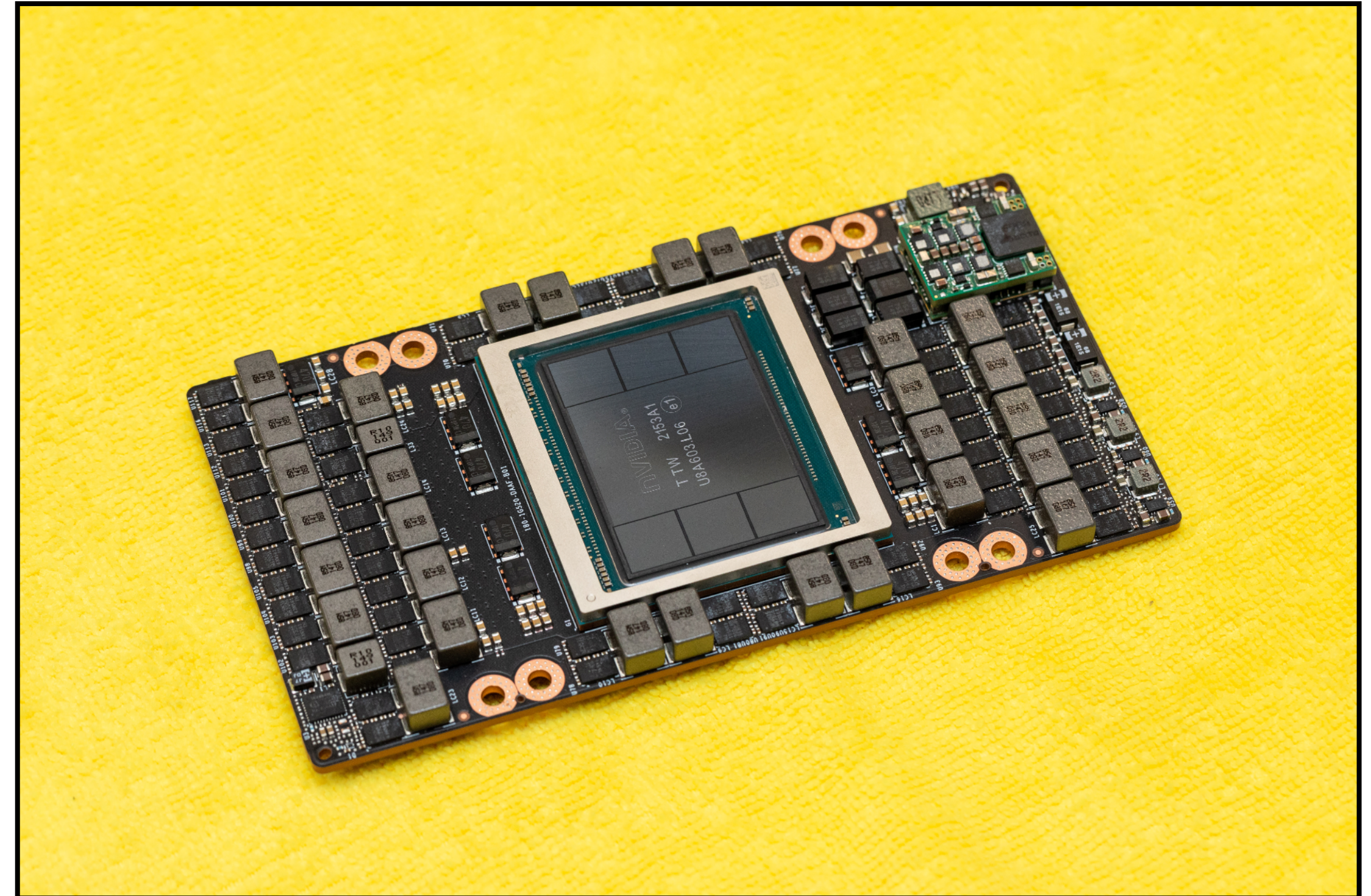


NVIDIA's stock price went up 1'202.17%

Currently the most valuable company on earth (**market capitalization**)

Spoiler: it's not because of computer games.

Datacenter GPUs



Price of a datacenter GPU: 50 000 CHF



The emergence of artificial intelligence

GPUs are really, really good at matrix multiplication.

ChatGPT 5 ▾

Hey, Wenzel. Ready to dive in?

Can you tell me what the fundamental mathematical operation underlying artificial intelligence is?



Excellent question — and one with both a simple and a deep answer.

At the **most fundamental mathematical level**, the core operation underlying nearly all of artificial intelligence — especially modern machine learning and deep learning — is **linear algebra**, specifically:

💡 **Matrix multiplication** (and by extension, vector dot products).

Emergence of Tensor Cores

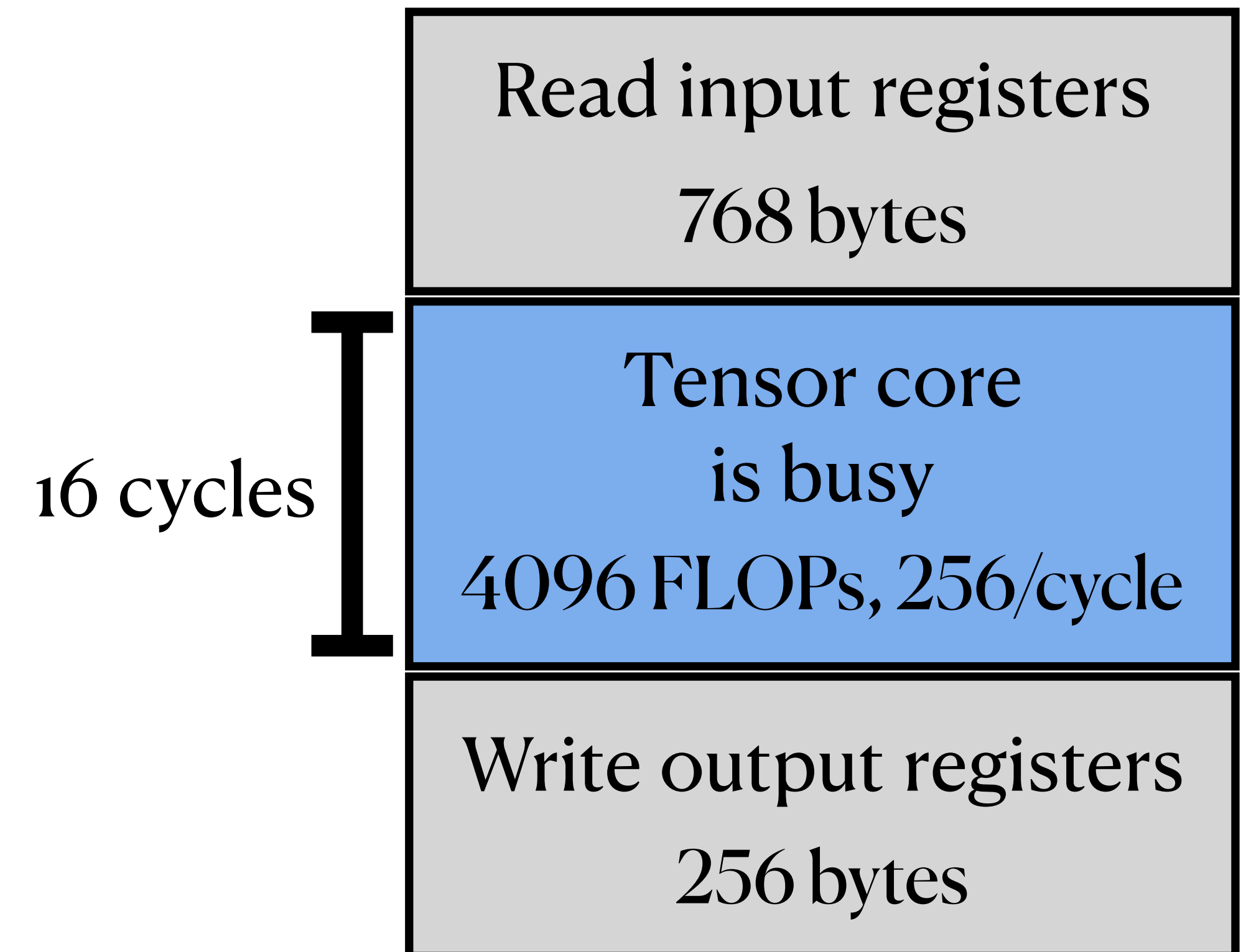
- Fancy name for a matrix multiplication unit
 - Like a "FMA" (fused-multiply-add) machine instruction
 - FMA: $D = A * B + C$
 - Tensor core: $D = A @ B + C$
 - Multiplies a fixed matrix size quickly (e.g., $256 \times 32 @ 32 \times 256$)
 - Nowadays most of the computational "power" of a GPU is in the tensor core.
 - Recent trend: very low precision floating point (16 bit \rightarrow 8 bit \rightarrow 4 bit "floats")

		16 bit floats	8 bit floats	4 bit "floats"
Percentage of FLOP/s in tensor core:	B200 (data center)	98.4%	99.2%	99.6%
	RTX 5090	75.0%	87.49%	93.8%

Tensor Core on consumer GPUs: **HMMMA.16816**

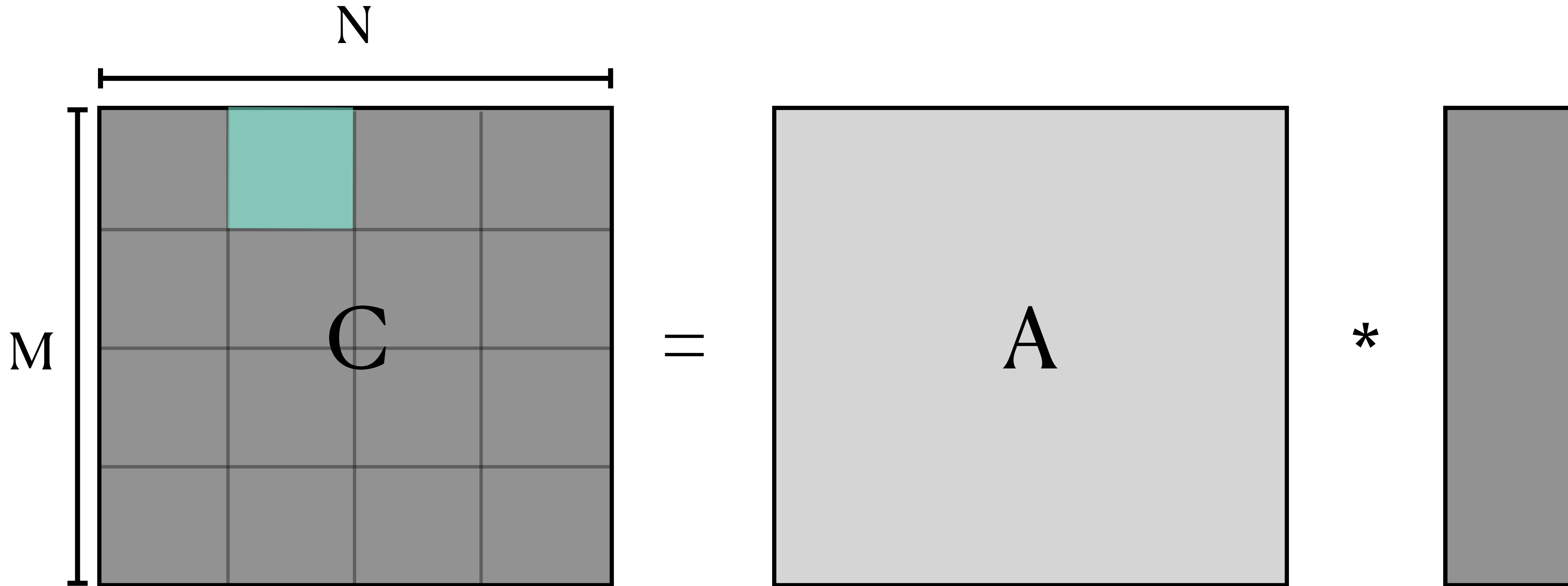
16 x 8 x 16 half precision matrix multiplication

- 256 flops/cycle per SM partition
- Some data for Ada A6000 (my old GPU)
 - **64 TB/s bandwidth** needed just for inputs!
 - **Global memory:** 960 GB/s (ideal)
 - **L2 cache:** 5 TB/s (measured)
 - **Shared memory:** 45.5 TB/s (ideal)

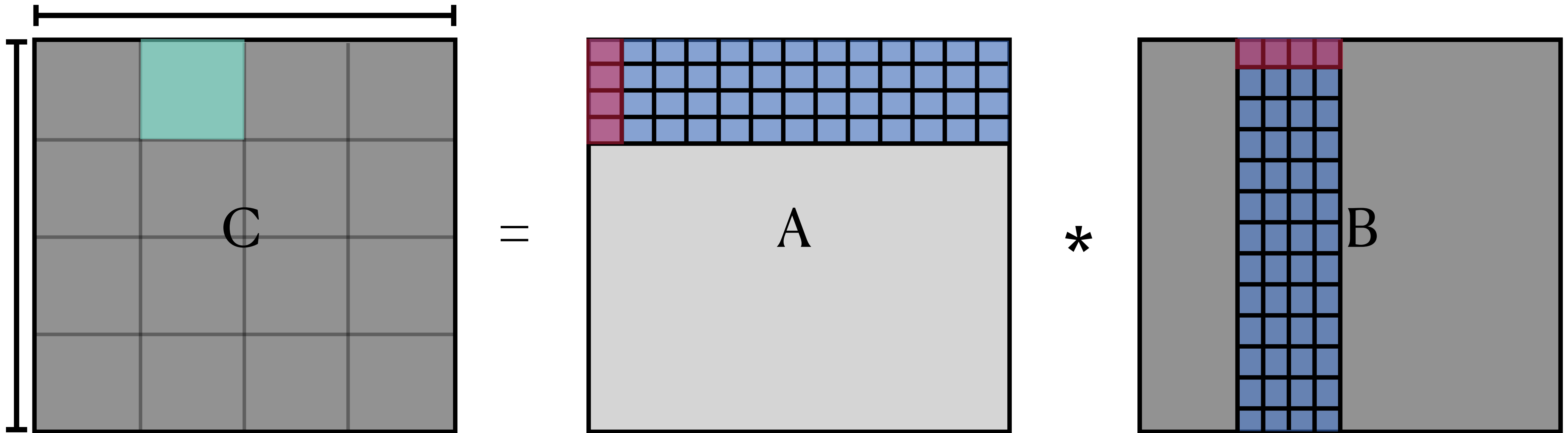
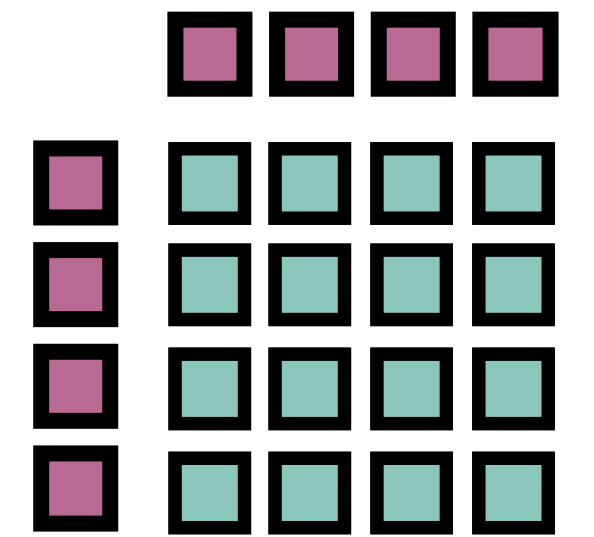


Not even shared memory is fast enough. How can one even use this operation?

Block matrix multiplication



Block matrix multiplication



 = 16x16 block for HMMA.16816

In summary

- Key property of matrix multiplication: $O(n^3)$ compute for $O(n^2)$ data.
 - Exploit this by fetching parts of a matrix into local ("shared") memory
 - Then do a lot of computation with these parts
 - Allows matrix computation to be **compute-bound** instead of **memory-bound**.
- GPUs are much better at multiplying big matrices than CPUs
- Matrix multiplication is the key operation underlying ML / AI (in particular, large language models use **huge** matrices)