INTRODUCTION TO GPU PROGRAMMING

Guest Lecturer: Ben Wibking

CMSE 822: Parallel Programming
Michigan State University

WHAT IS EXASCALE SUPERCOMPUTING?

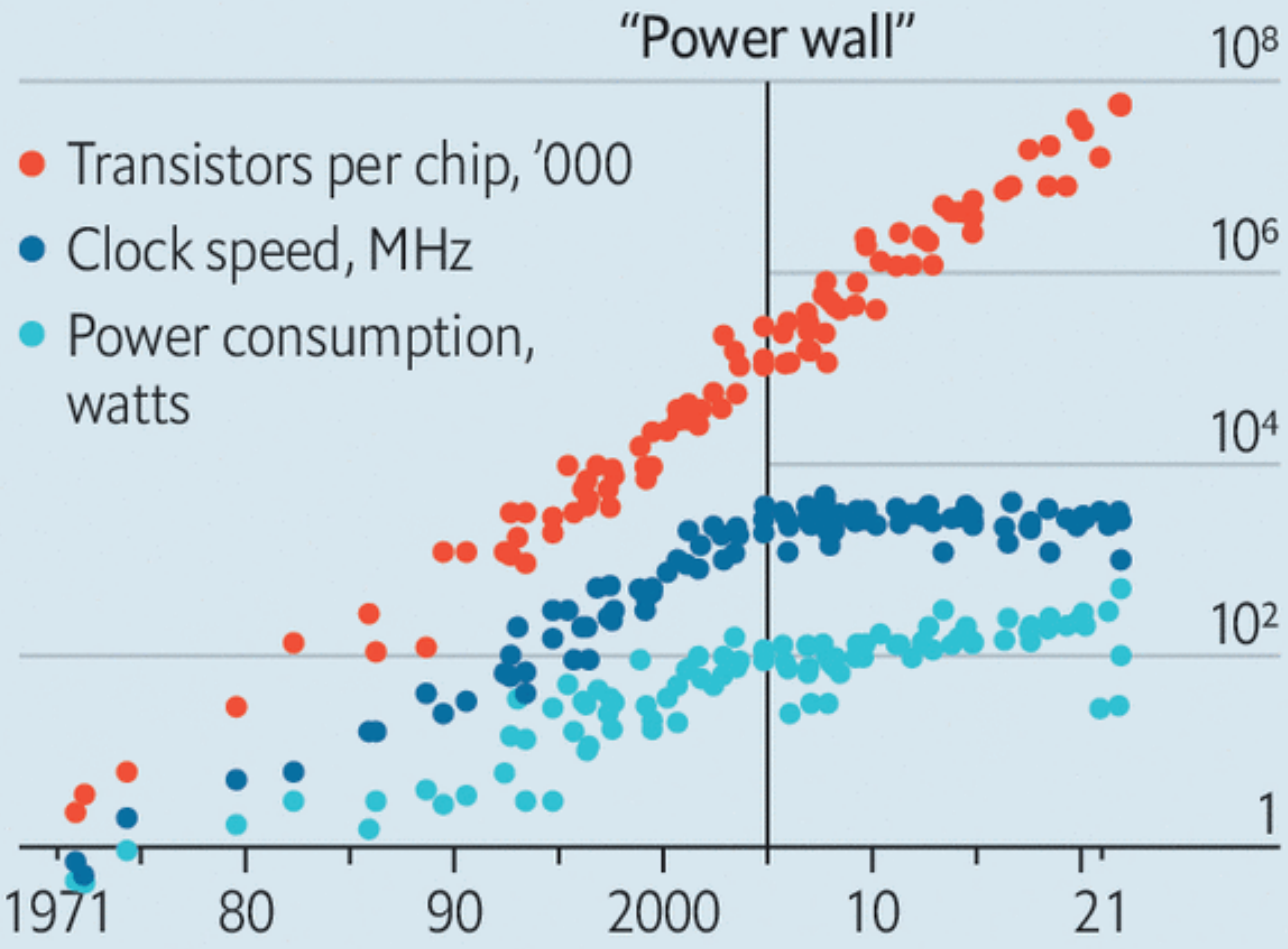
- A tightly-coupled system capable of 1 exaflop = 10^{18} floating-point operations per second
- Nominally measured in terms of performance on a parallel matrix-multiplication benchmark
- More important is application performance: 1,000 times FOM (= faster x larger) simulations compared to petaflop-scale supercomputers

WHY IS EXASCALE SUPERCOMPUTING?

- Required for:
  - Cloud-resolving (~1 km resolution) climate simulations
  - Well-resolved 3D nuclear reactor simulations
  - Well-resolved 3D nuclear weapon simulations
  - Your simulations?

All the small things

Microprocessor engineering, log scale



Sources: Imec; "50 years of microprocessor trend data", by K. Rupp et al., 2022

The Economist, Dec 13 2023.

WHY ARE GPUS USED FOR EXASCALE?

- Nothing else is possible within reasonable electrical power constraints
- A 2008 U.S. DARPA study projected that with existing technology, exascale would require 67 MW of power to run it
- (1 projected exascale system = almost 10% of the capacity of a typical nuclear reactor)
- Had to search for technologies with much better *performance per watt*

GOALS

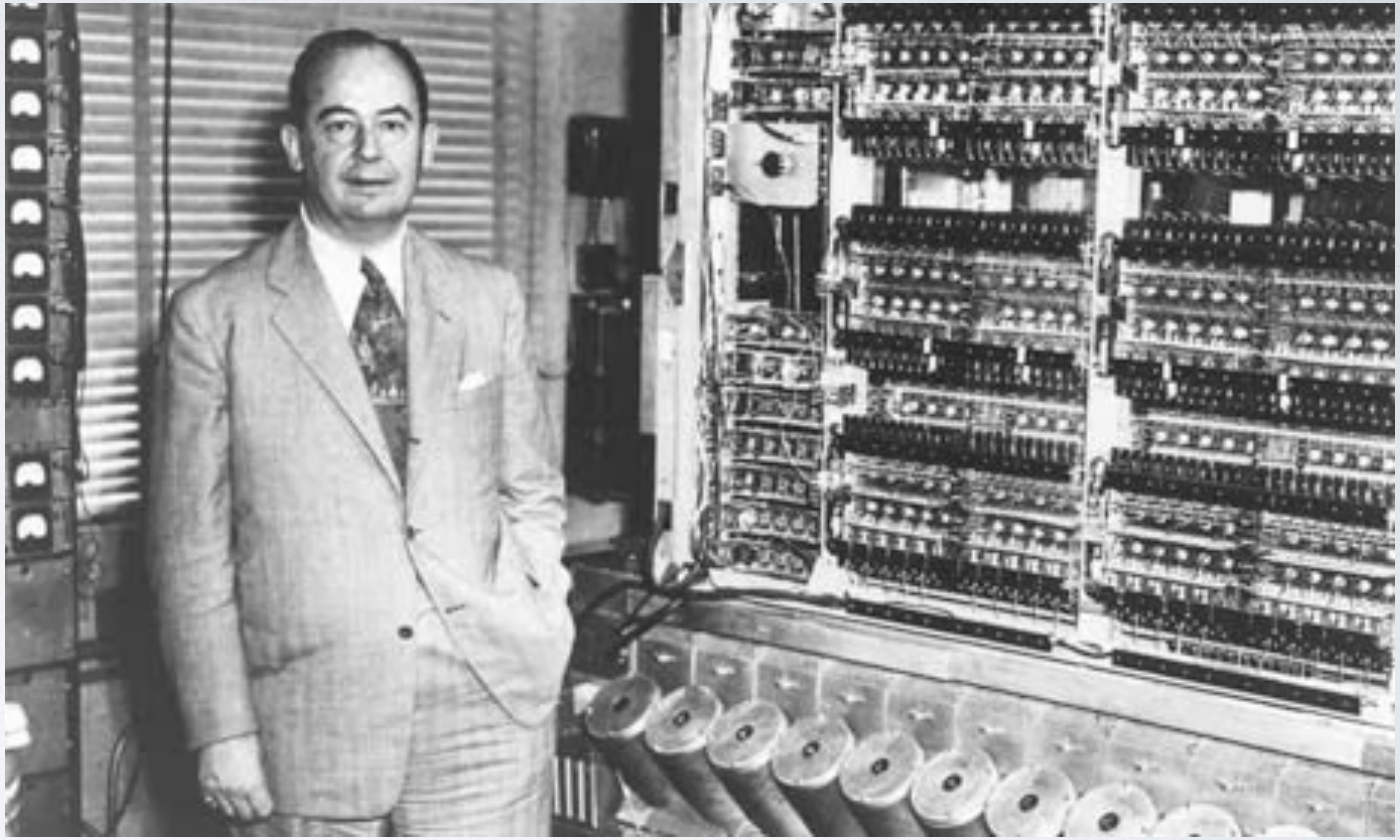
- Understand "what is a CPU?"
- Understand "what is a GPU?"
- Understand how GPUs execute programs
- Understand what kinds of algorithms and applications perform well on GPUs due to their unique architectural features

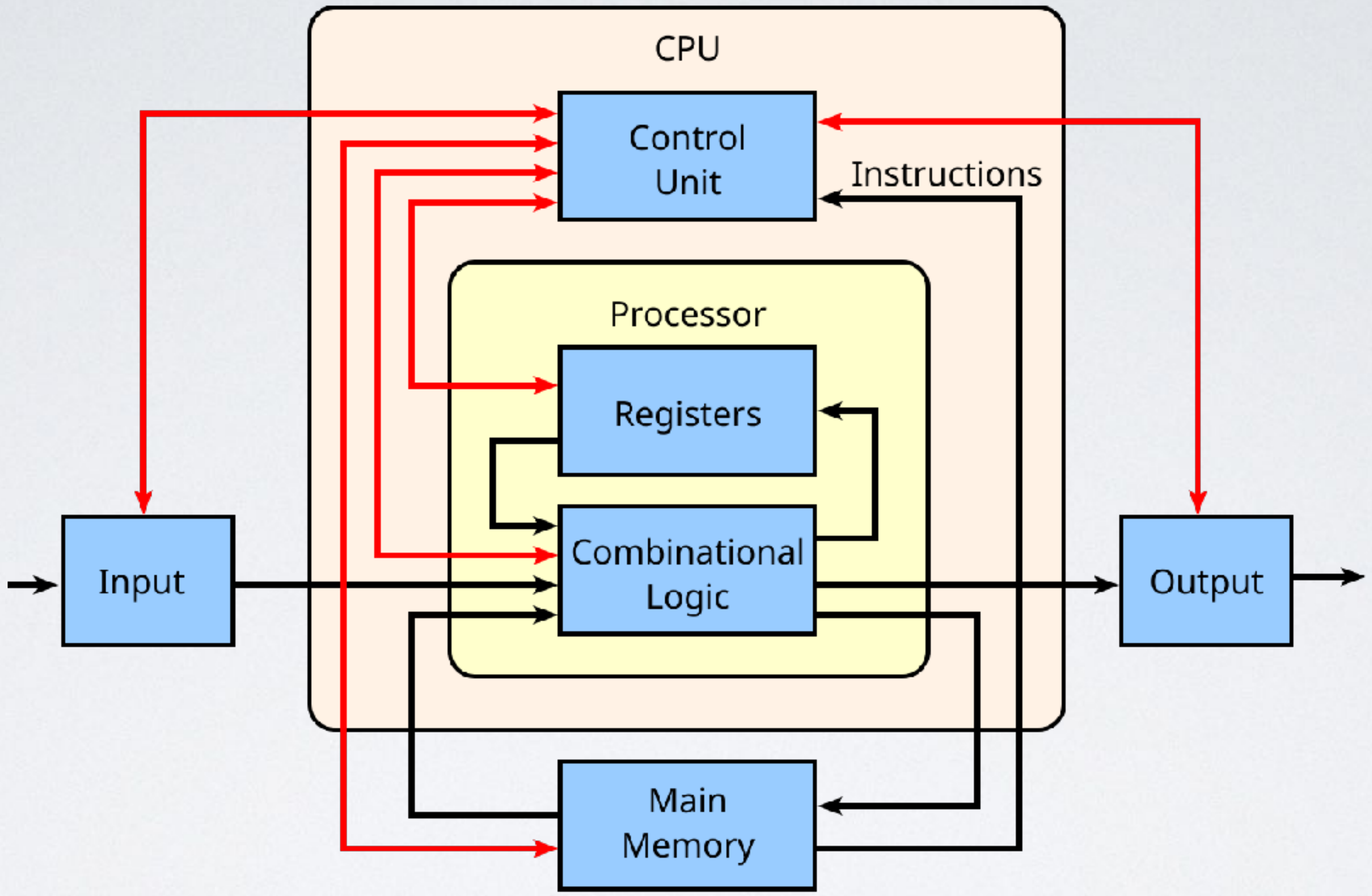
WHAT IS A CPU?

- The "central processing unit" of a computer
- Simple execution model
- Conceptually, executes a linear stream of sequential instructions read from memory

By Pstrahl via Wikipedia - CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=151972144

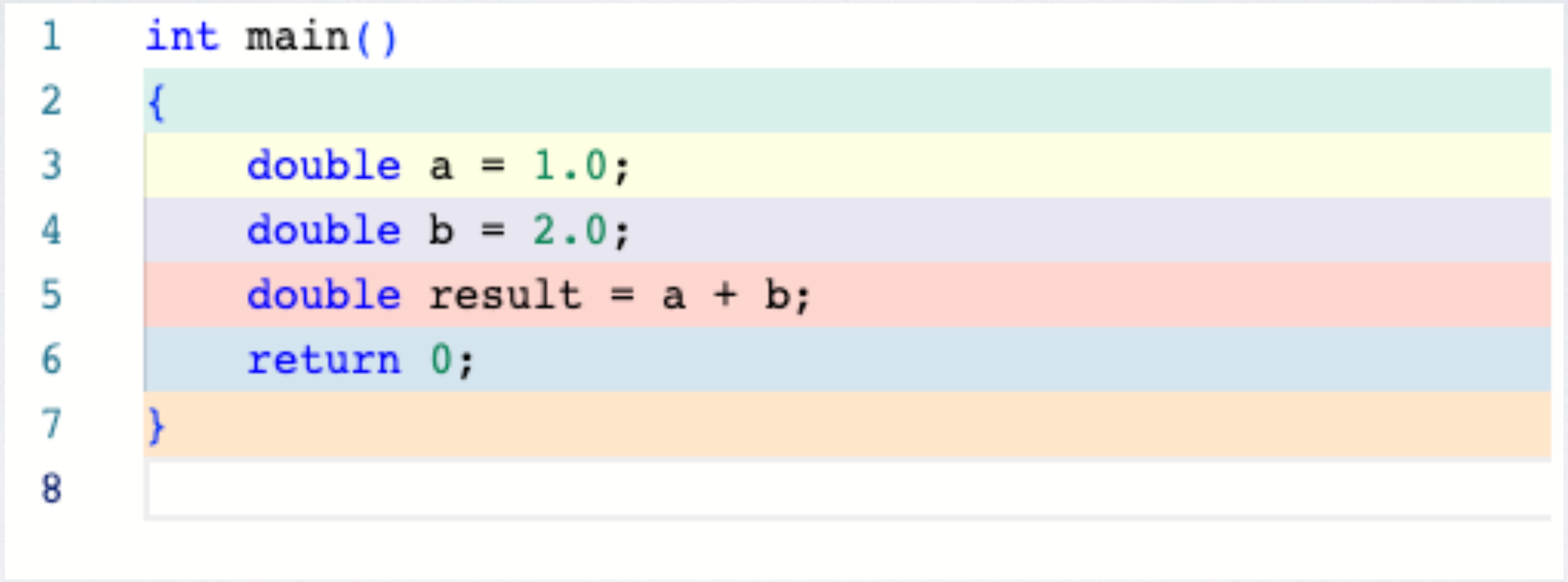
- Conceptually, executes a linear stream of sequential instructions read from memory
- The "von Neumann" computer architecture, after its inventor:


John von Neumann in 1945 with the world's first stored-program computer at the Institute for Advanced Study (Getty Images)



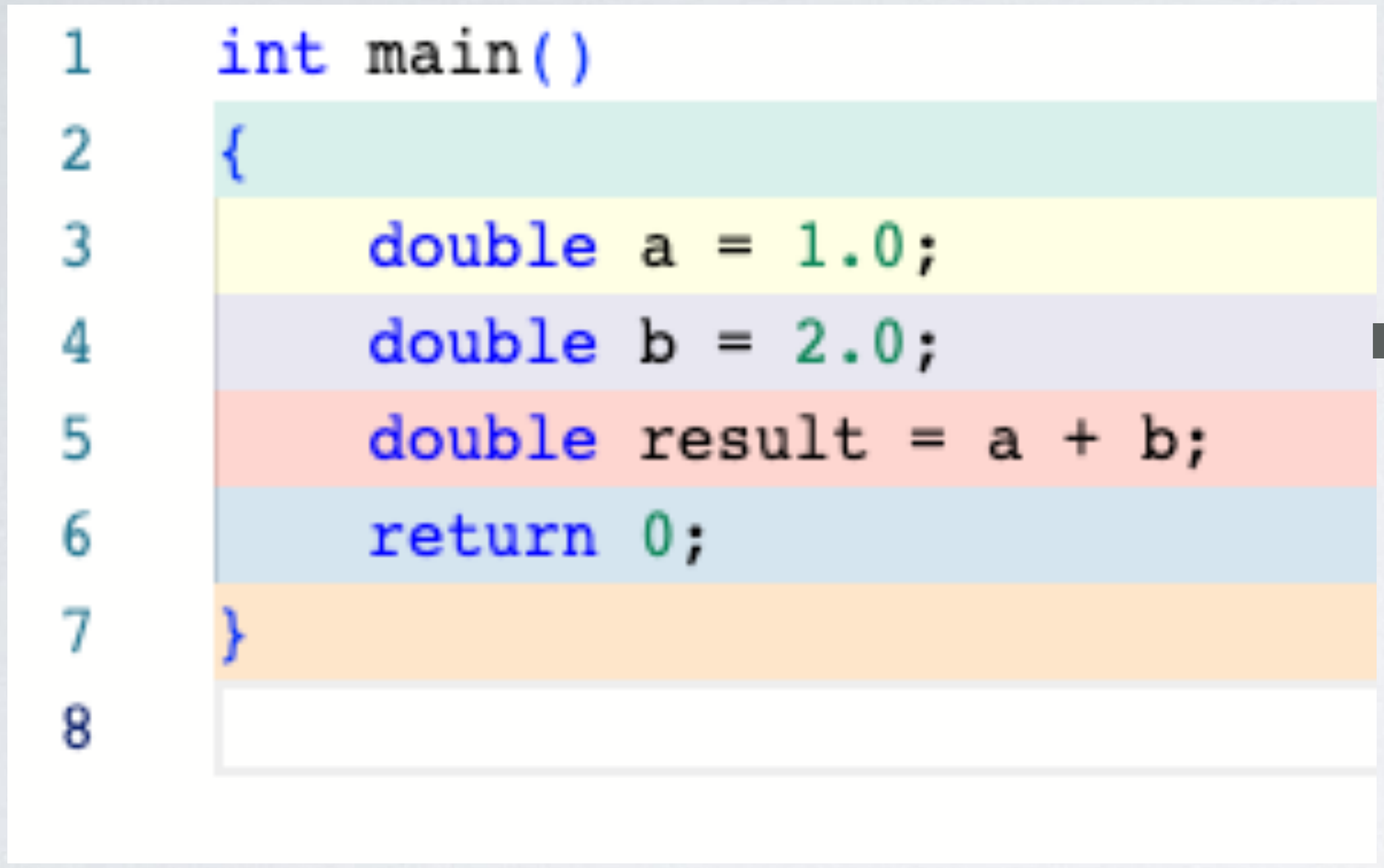
SIMPLIFIED BLOCK DIAGRAM OF A CPU

SIMPLEST POSSIBLE CPU PROGRAM



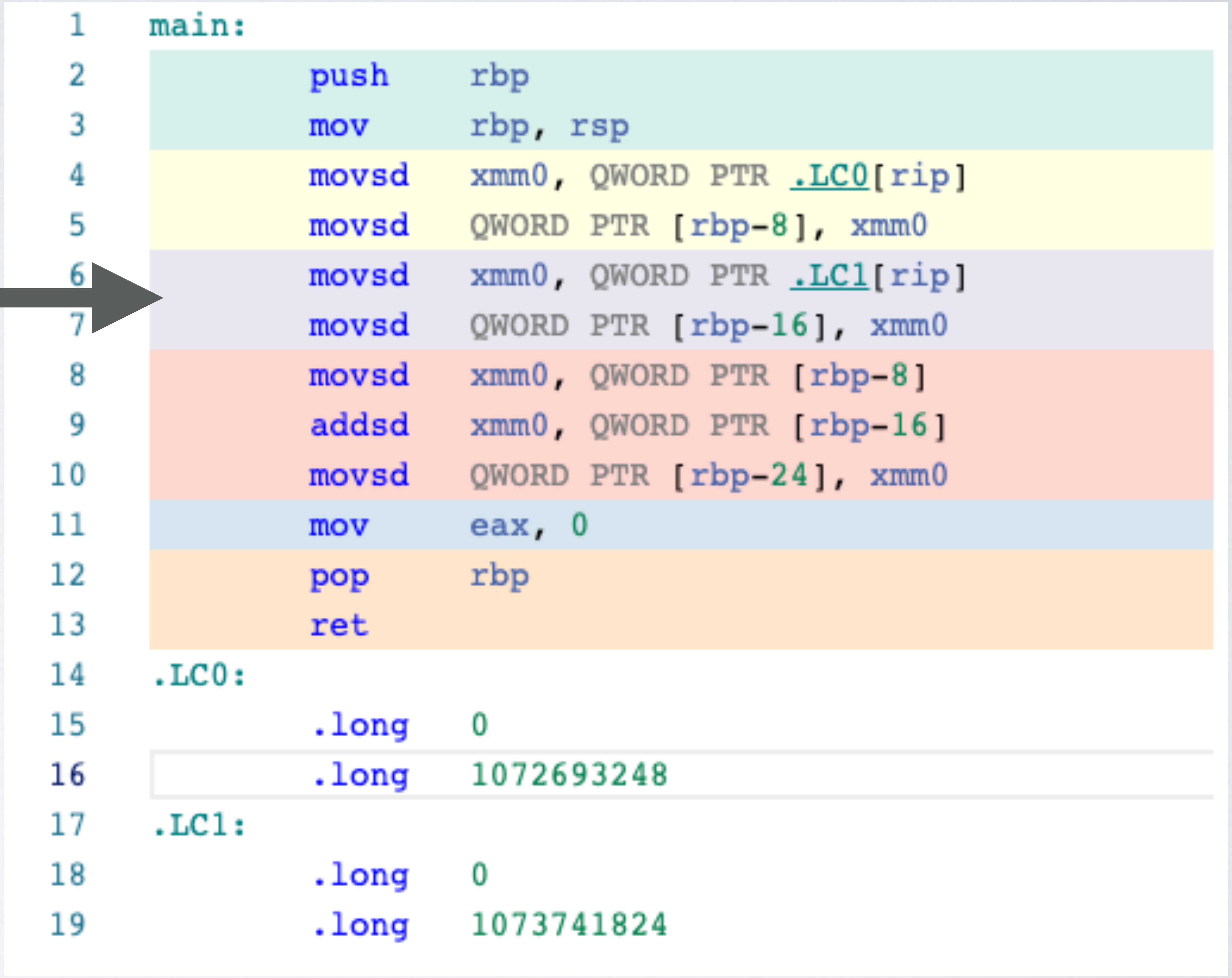
```
int main()
{
    double a = 1.0;
    double b = 2.0;
    double result = a + b;
    return 0;
}

```

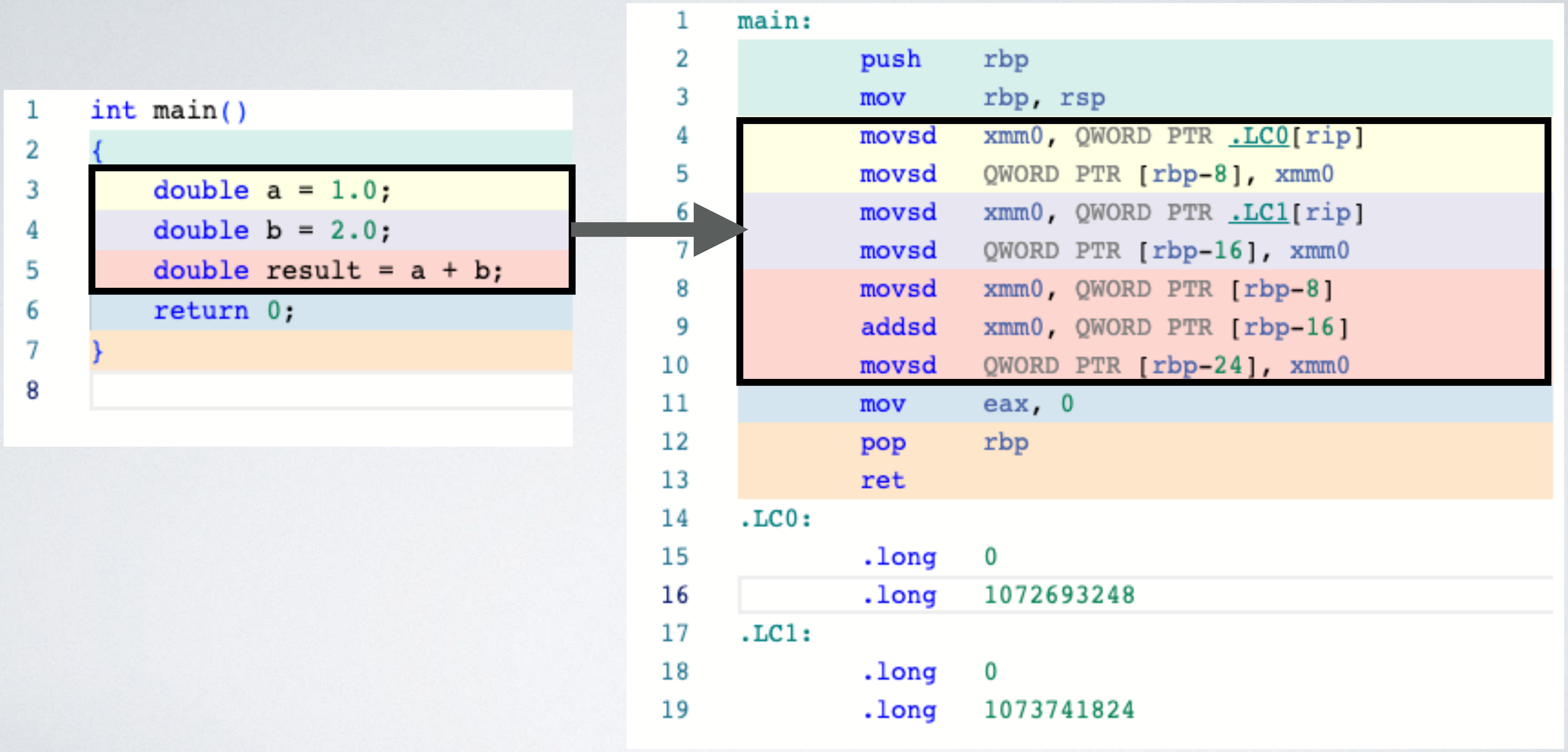


```c
int main()
{
    double a = 1.0;
    double b = 2.0;
    double result = a + b;
    return 0;
}

```

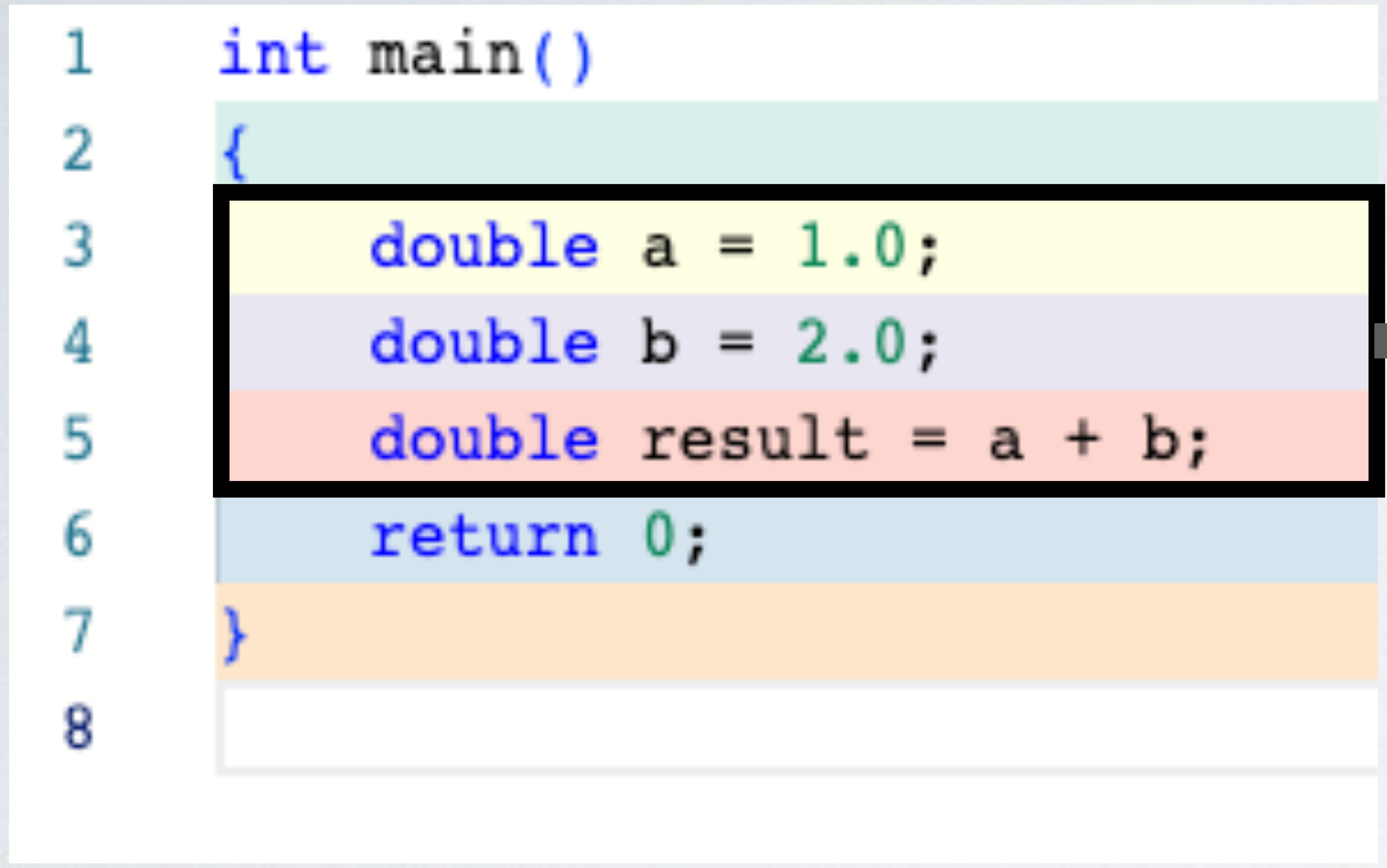


```asm
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```

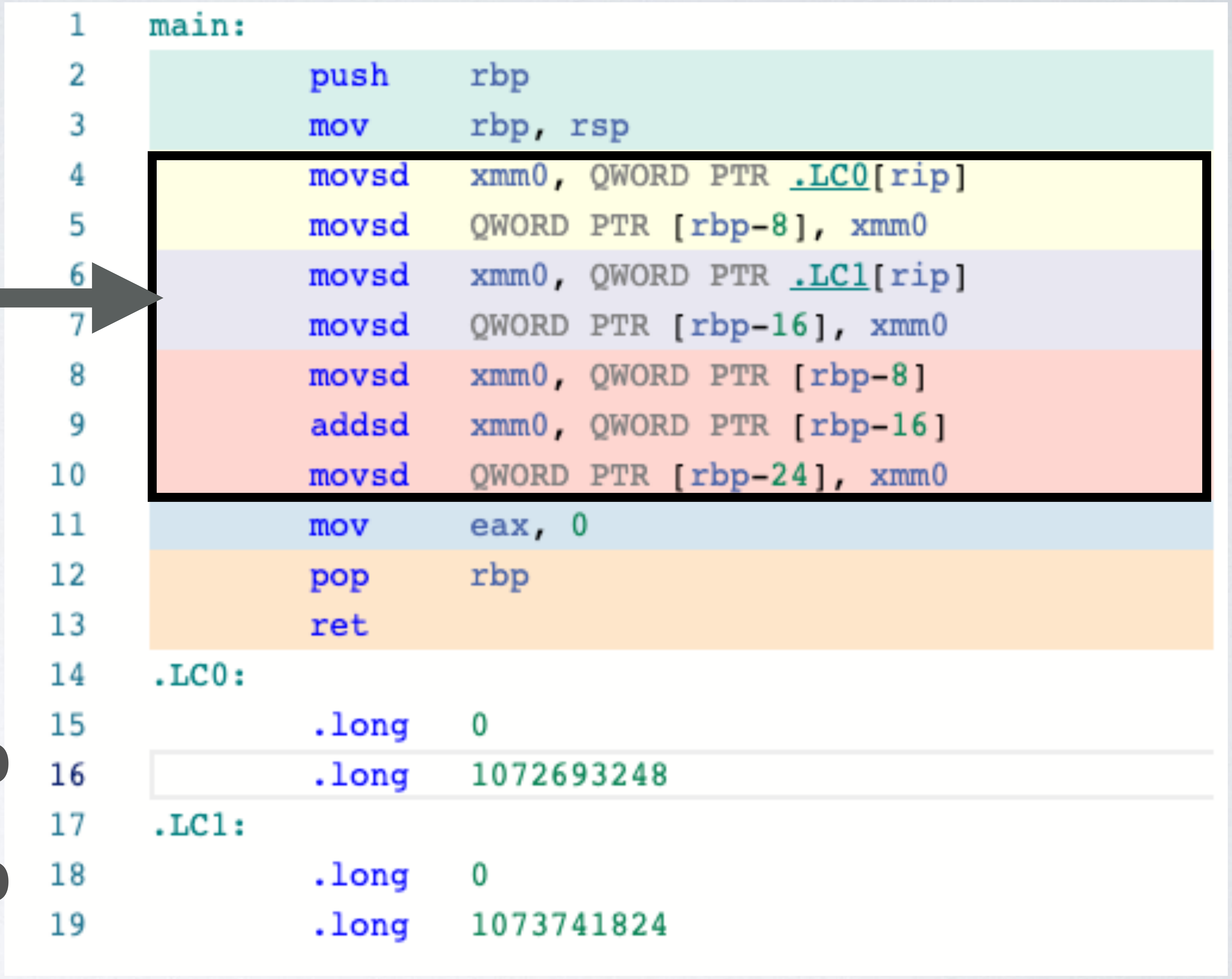


```c
int main()
{
    double a = 1.0;
    double b = 2.0;
    double result = a + b;
    return 0;
}
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```



```c
int main()
{
    double a = 1.0;
    double b = 2.0;
    double result = a + b;
    return 0;
}
```



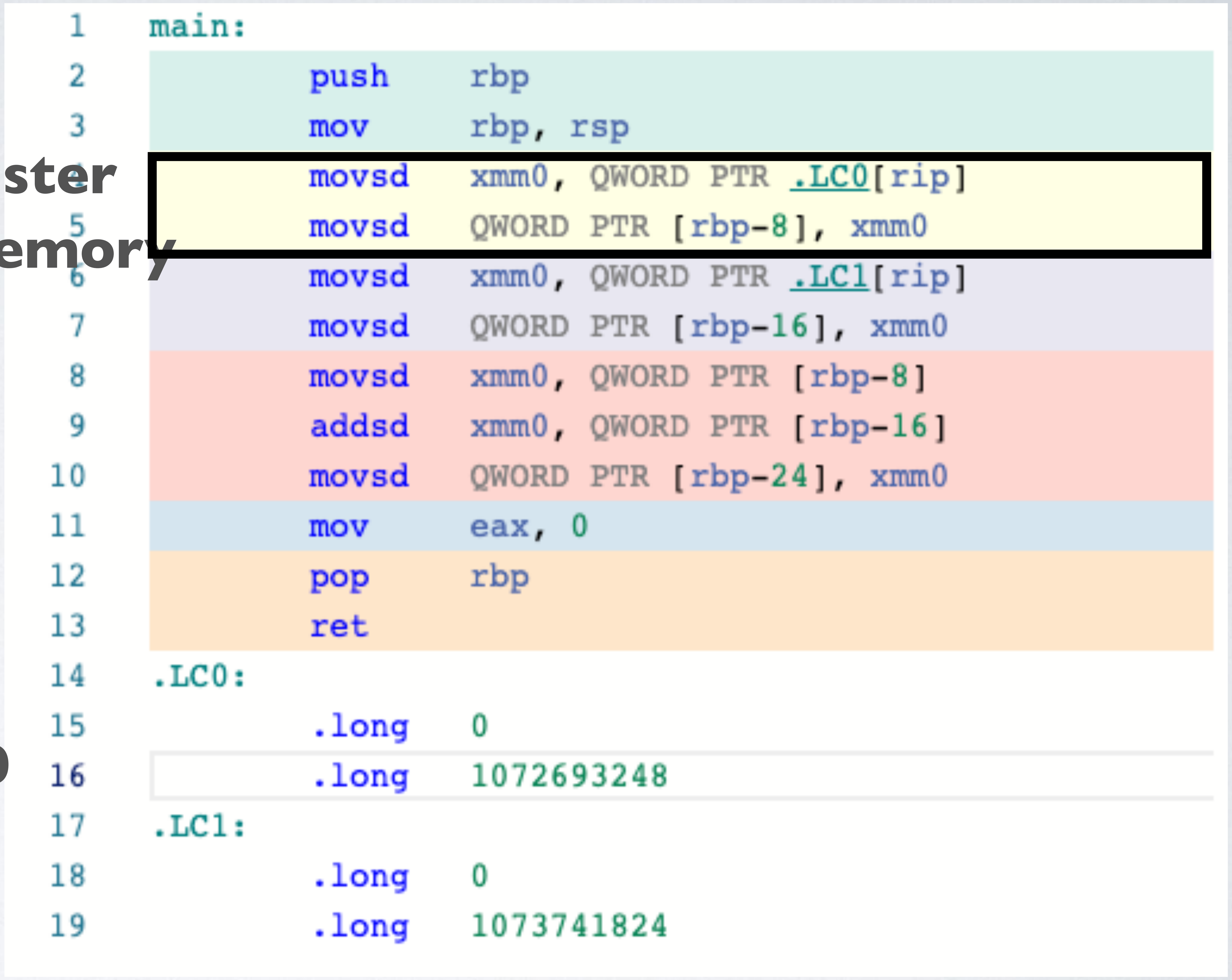
```asm
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```

Binary representation of 1.0

Binary representation of 2.0

Copy value from .LC0 to register
Copy value from register to memory

Binary representation of 1.0

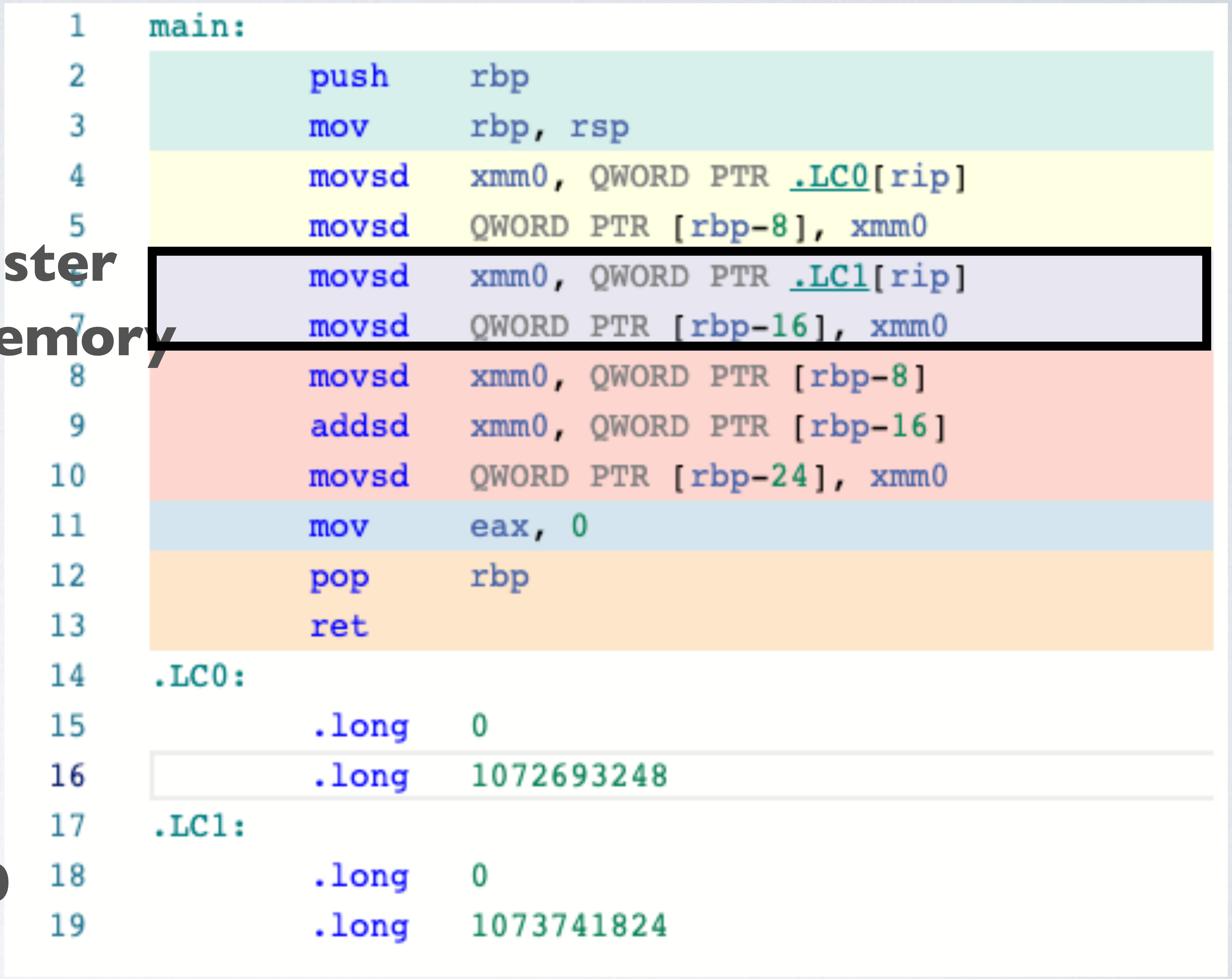


```
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```

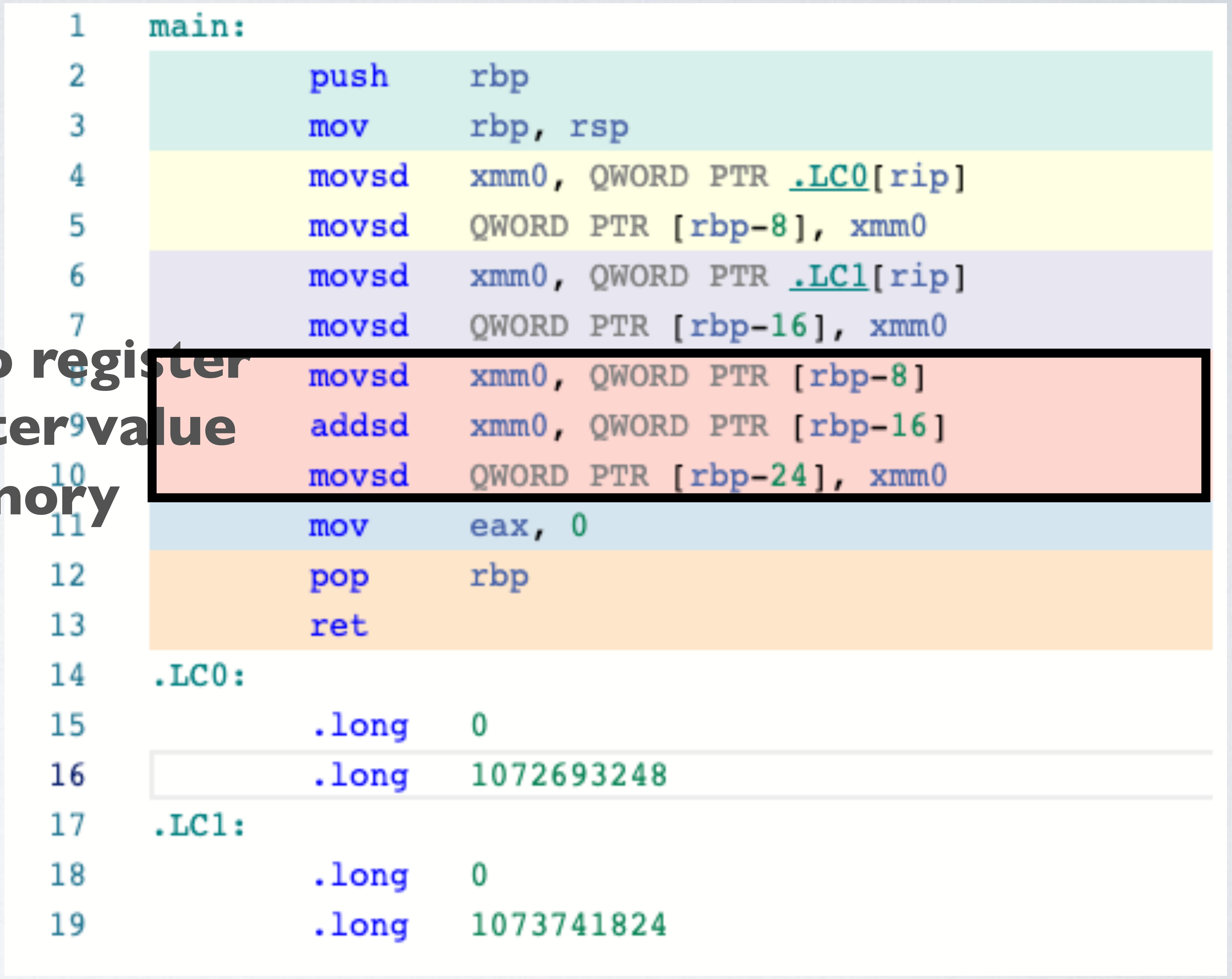
Copy value from .LC1 to register

Copy value from register to memory

Binary representation of 2.0



```
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```



```
main:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   QWORD PTR [rbp-8], xmm0
        movsd   xmm0, QWORD PTR .LC1[rip]
        movsd   QWORD PTR [rbp-16], xmm0
        movsd   xmm0, QWORD PTR [rbp-8]
        addsd   xmm0, QWORD PTR [rbp-16]
        movsd   QWORD PTR [rbp-24], xmm0
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   0
        .long   1072693248
.LC1:
        .long   0
        .long   1073741824
```

Copies value from memory to register
Adds memory value to register value
Copies the result to memory

WHAT IS A CPU?

- That looks complicated, but all it is does is simply:
  - Fetch data from memory into processor registers
  - Perform arithmetic operations
  - Store the value of processor registers back to memory
- *Operates on a single data item at a time*

By Pstrahl via Wikipedia - CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=151972144

WHAT IS A CPU?

- *Operates on a single data item at a time*
- *Optimized for low latency memory fetches*
- The CPU is called a "scalar" processor
- Modern CPUs have multiple cores, but each core operates independently on its own stream of sequential instructions

By Pstrahl via Wikipedia - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=151972144

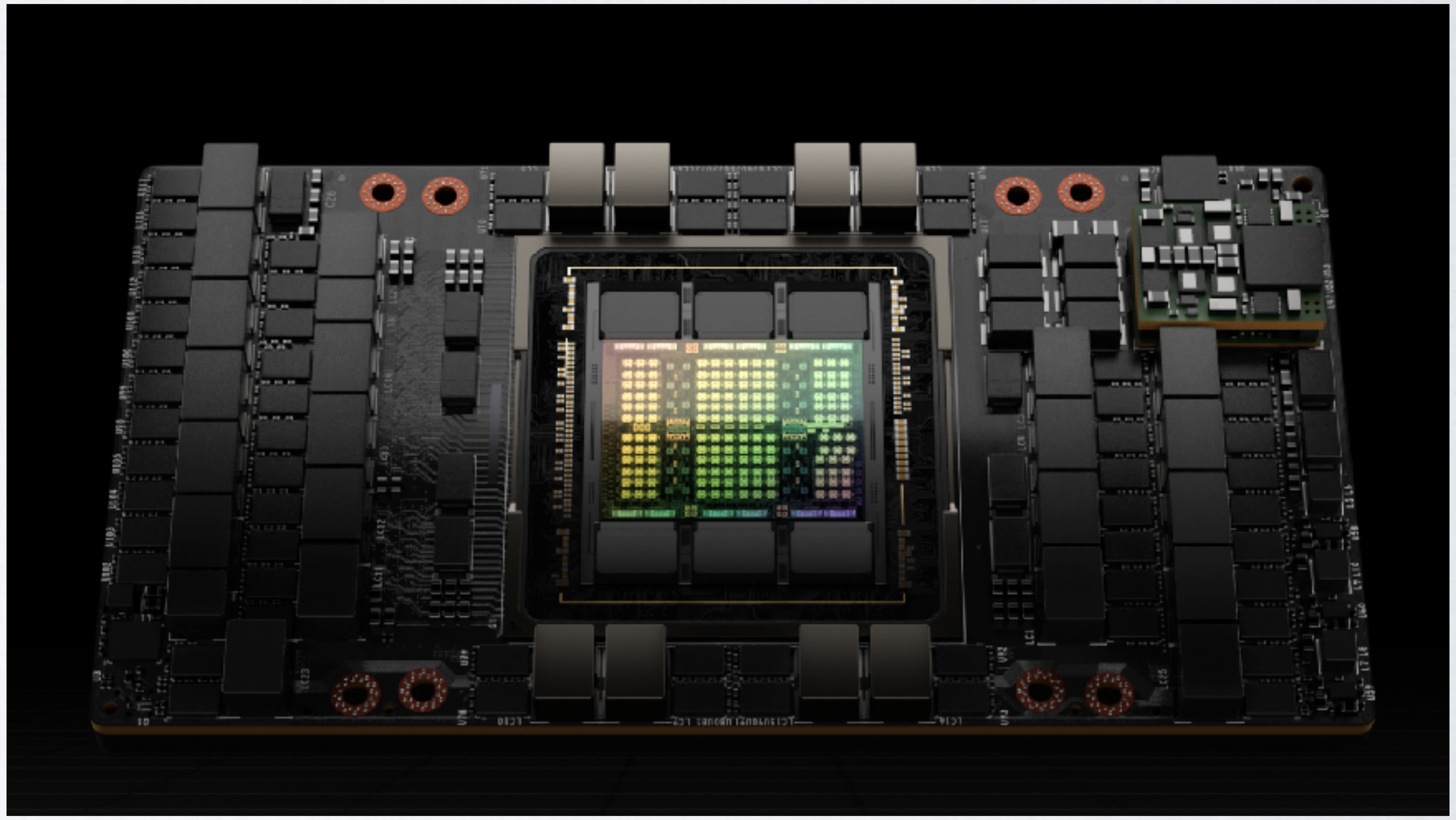
BUT, WAIT!

- Modern processors (>1990s) internally re-order instructions and execute them out of order
  - Critical to high performance of all CPUs today
  - Huge engineering effort to maintain the illusion of sequential execution
  - This is very expensive in both transistors and power

By Pstrahl via Wikipedia - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=151972144

WHAT IS A GPU?

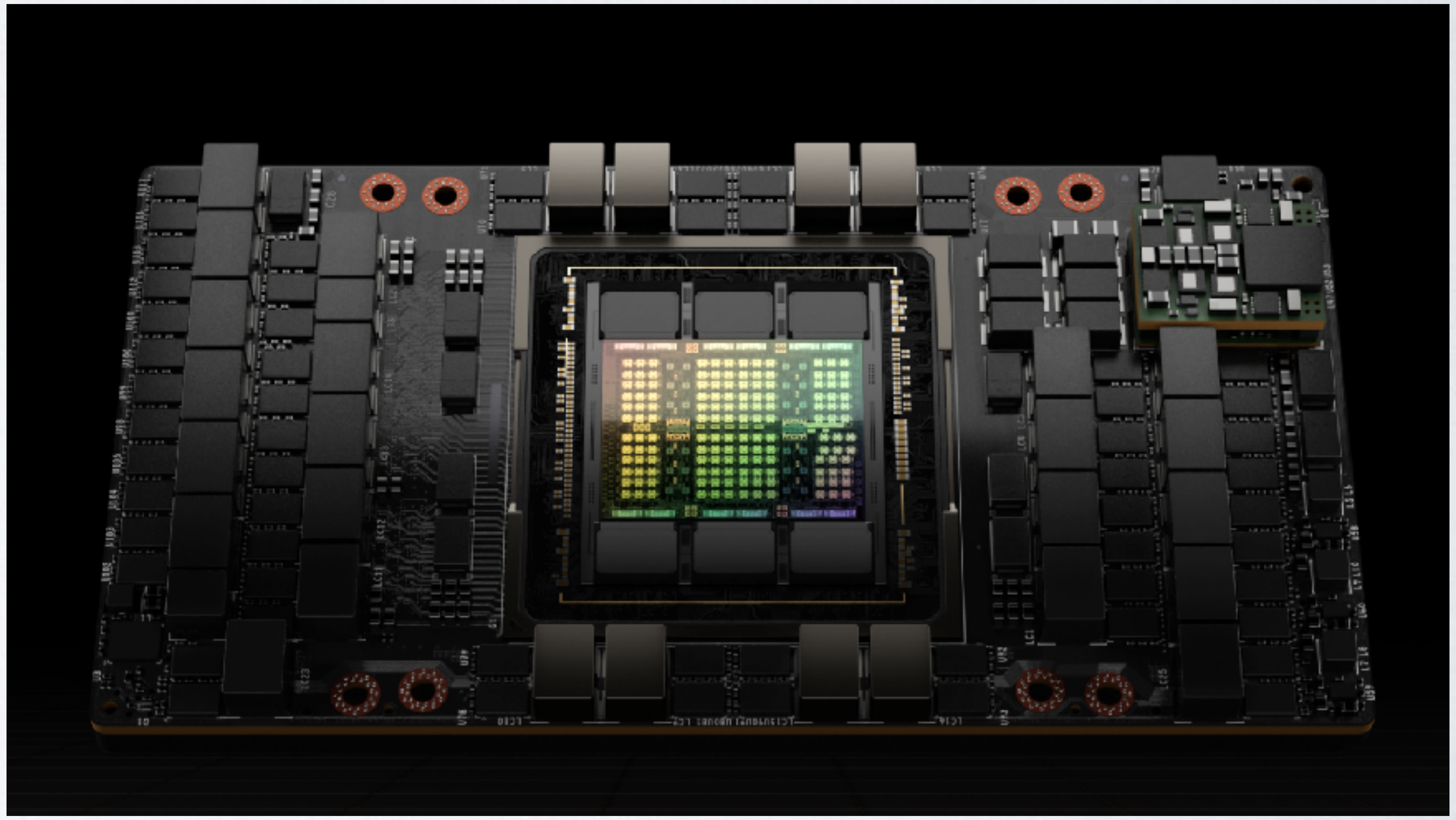
- The "graphics processing unit"
- Executes one (or more!) *compute kernels* simultaneously
- Has a complicated execution model (*not* reducible to a linear stream of sequential instructions)
- Operates on many data items in parallel



NVIDIA Hopper GPU

WHAT IS A GPU?

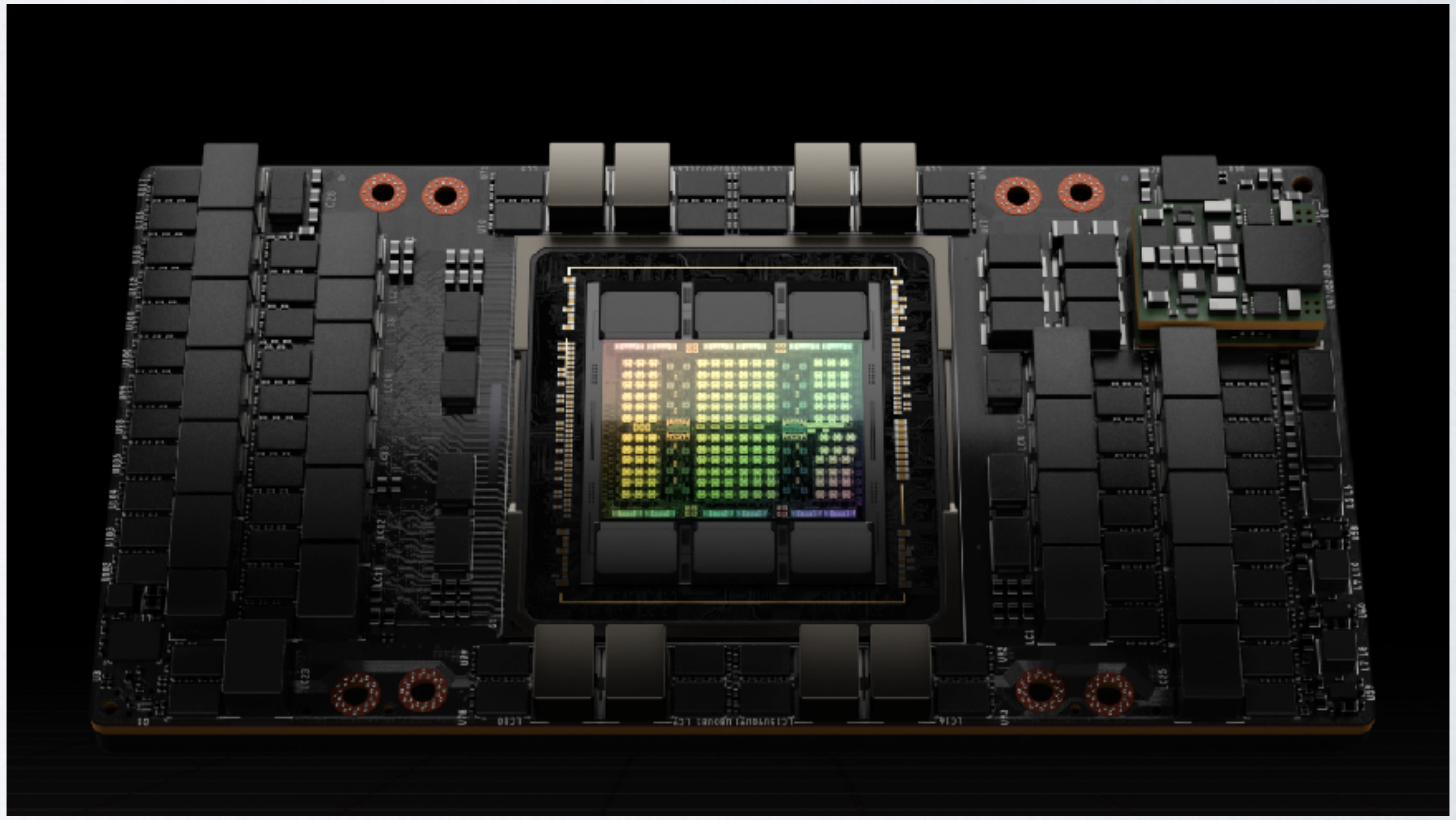
- A "vector" processor
- Operates with the same operation on many data items in parallel (analogous to adding two vectors)



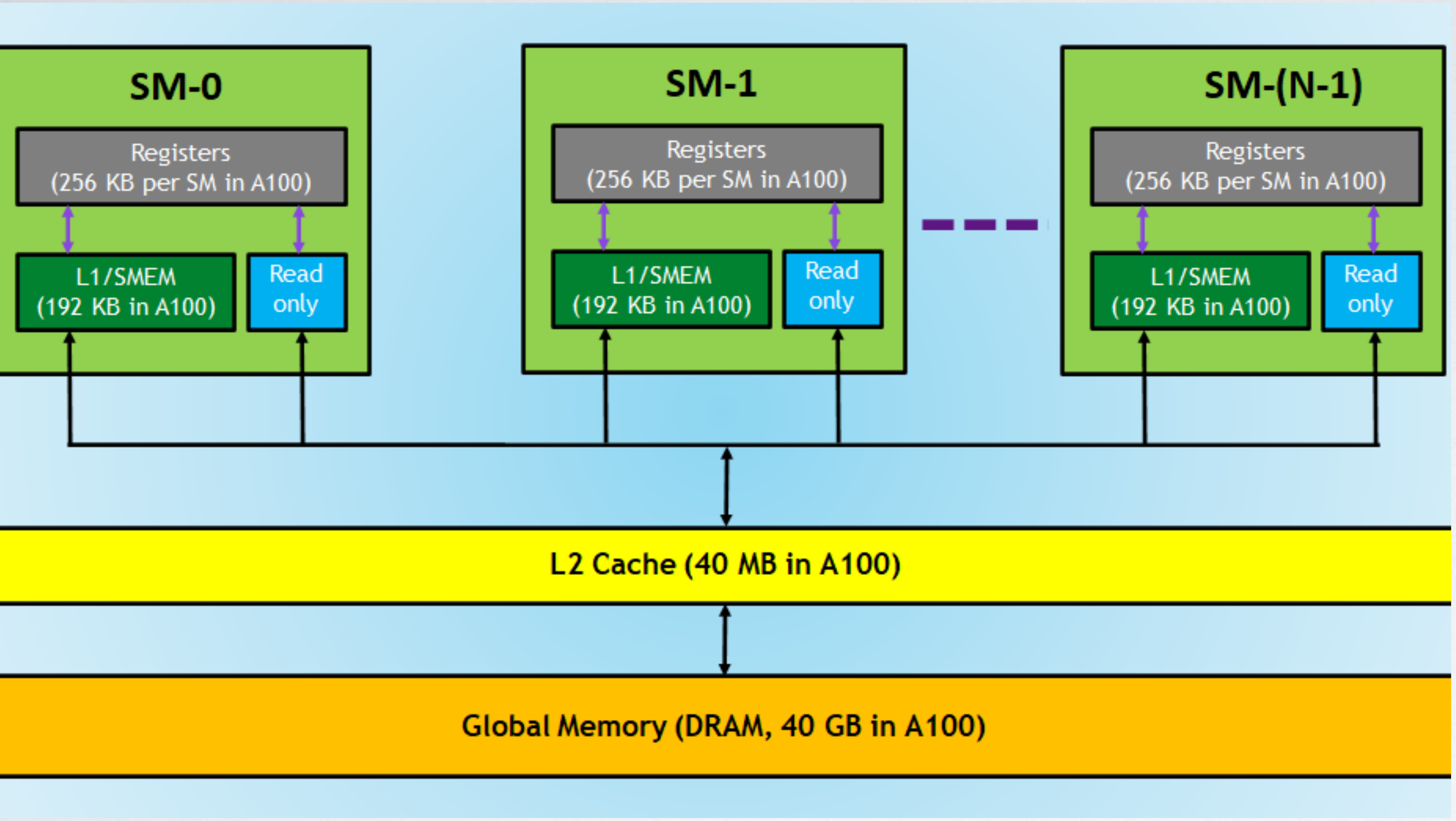
NVIDIA Hopper GPU

WHAT IS A GPU?

- Does not internally re-order instructions
    - More transistors can be spent on real computing
    - Uses less power per unit of computation
- Larger human effort to program efficiently


NVIDIA Hopper GPU

Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/



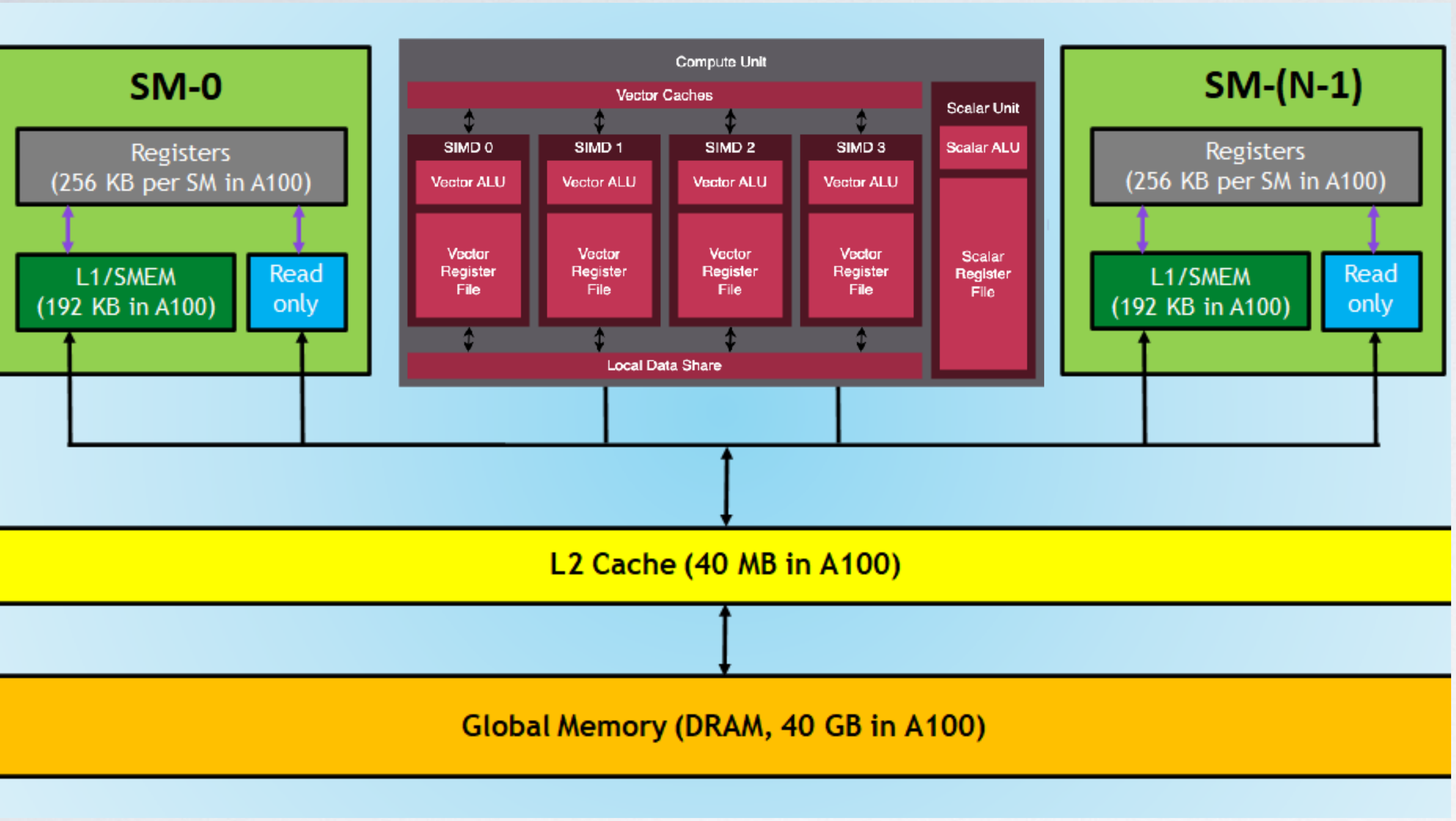
NVIDIA:
Streaming multiprocessor

AMD:
Compute unit

High-bandwidth memory
(Physically attached to GPU)

SIMPLIFIED BLOCK DIAGRAM OF A GPU

Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/



NVIDIA:
Streaming multiprocessor

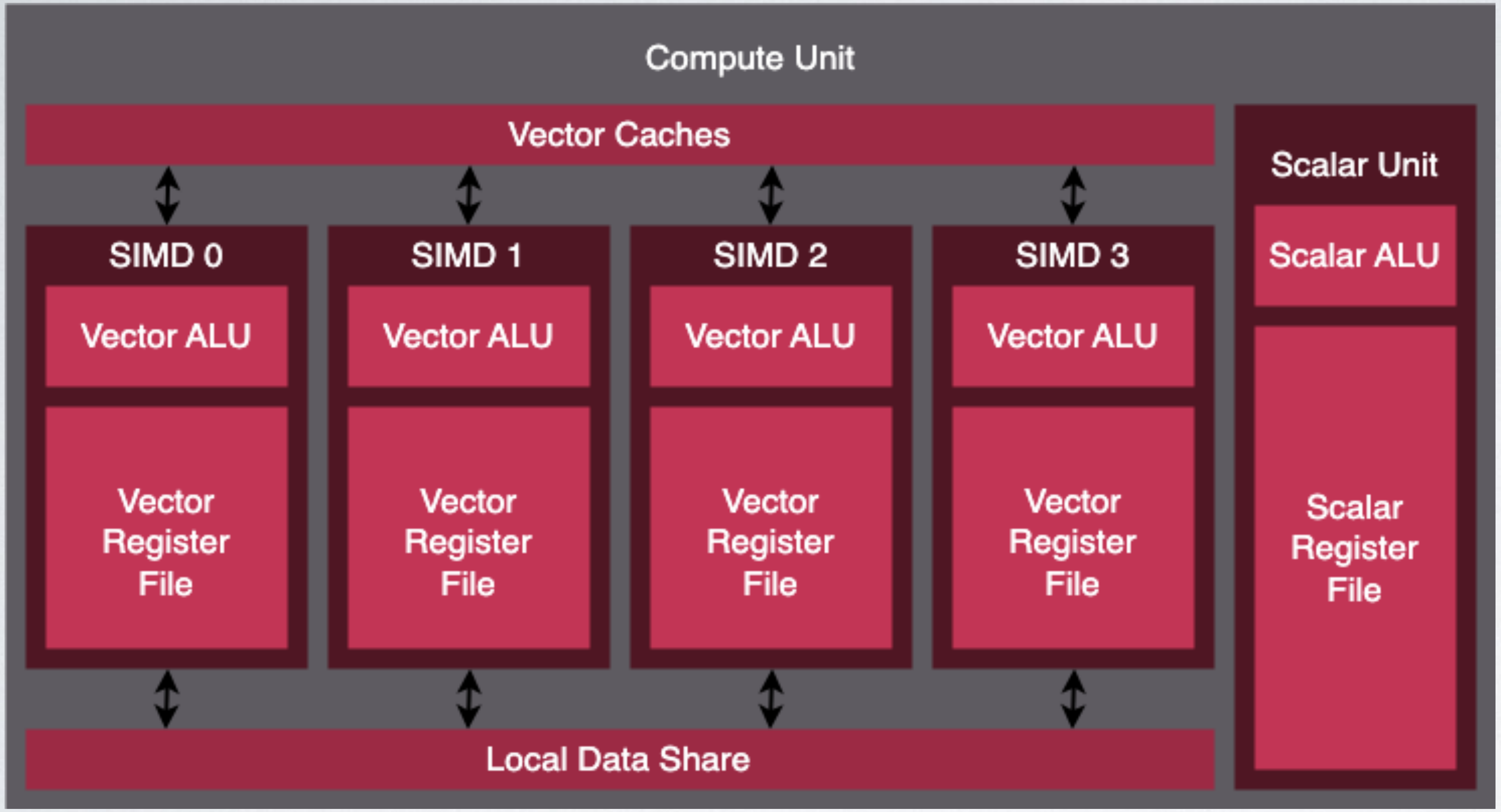
AMD:
Compute unit

High-bandwidth memory
(Physically attached to GPU)

SIMPLIFIED BLOCK DIAGRAM OF A GPU

Source: https://rocm.docs.amd.com/projects/HIP/en/latest/understand/hardware_implementation.html

NVIDIA: Streaming multiprocessor AMD: Compute unit

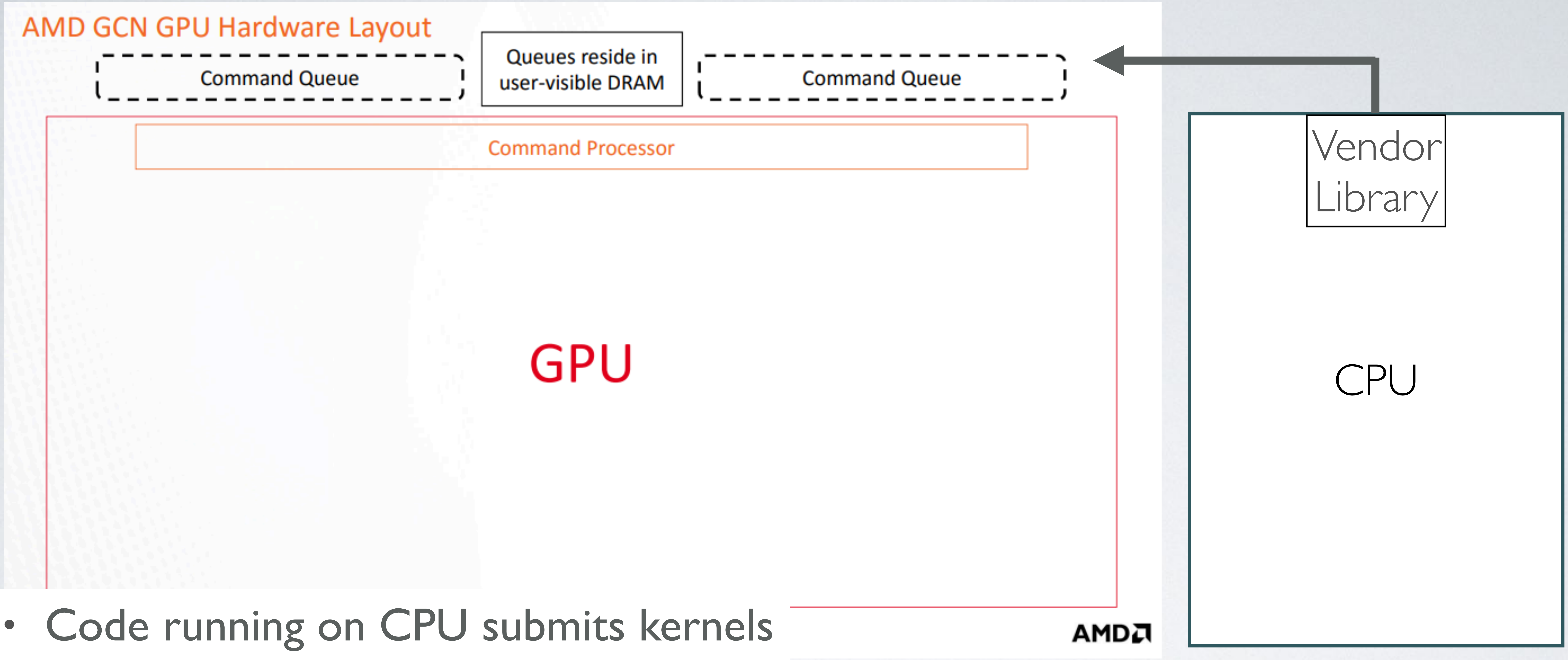


Vector ALU:
Performs arithmetic operations on vector registers

Vector register:
32- or 64-component vector that can be accessed with single clock cycle latency

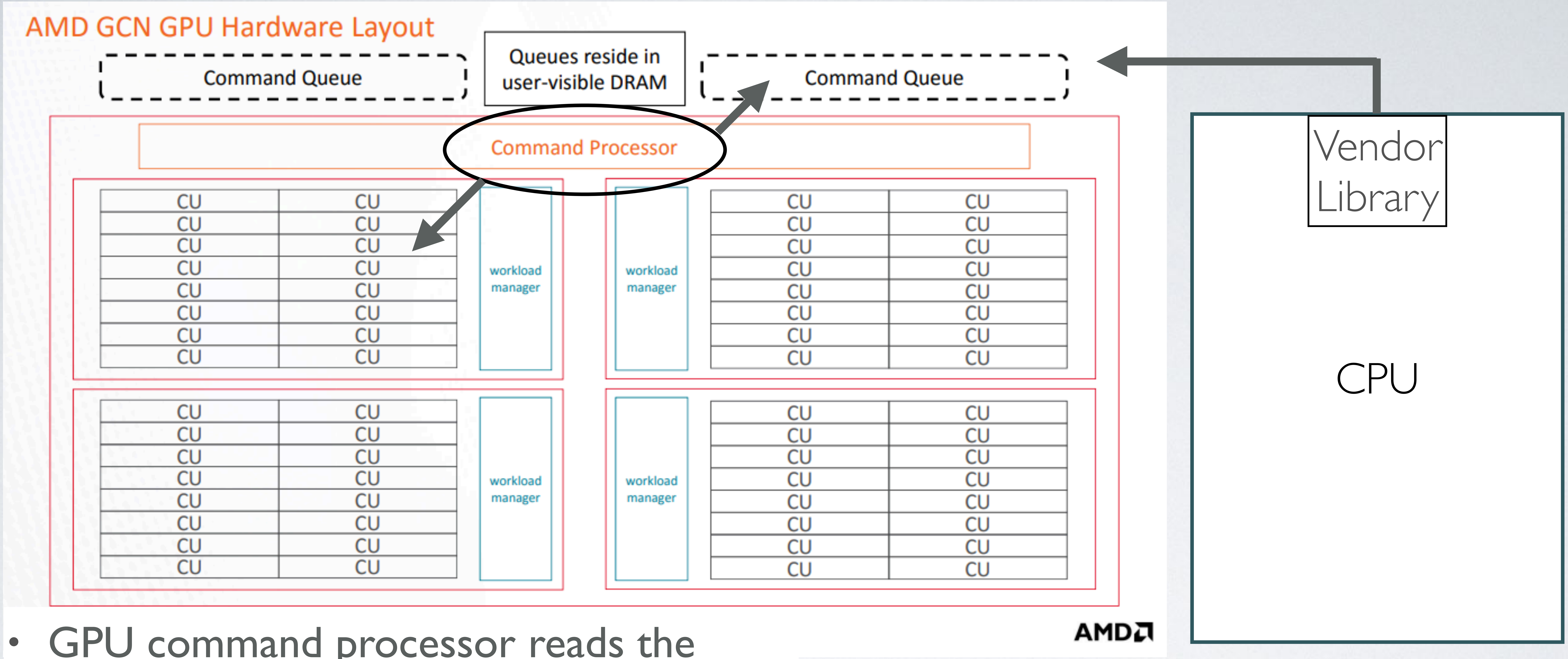
SIMPLIFIED BLOCK DIAGRAM OF A SM/CU

Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf

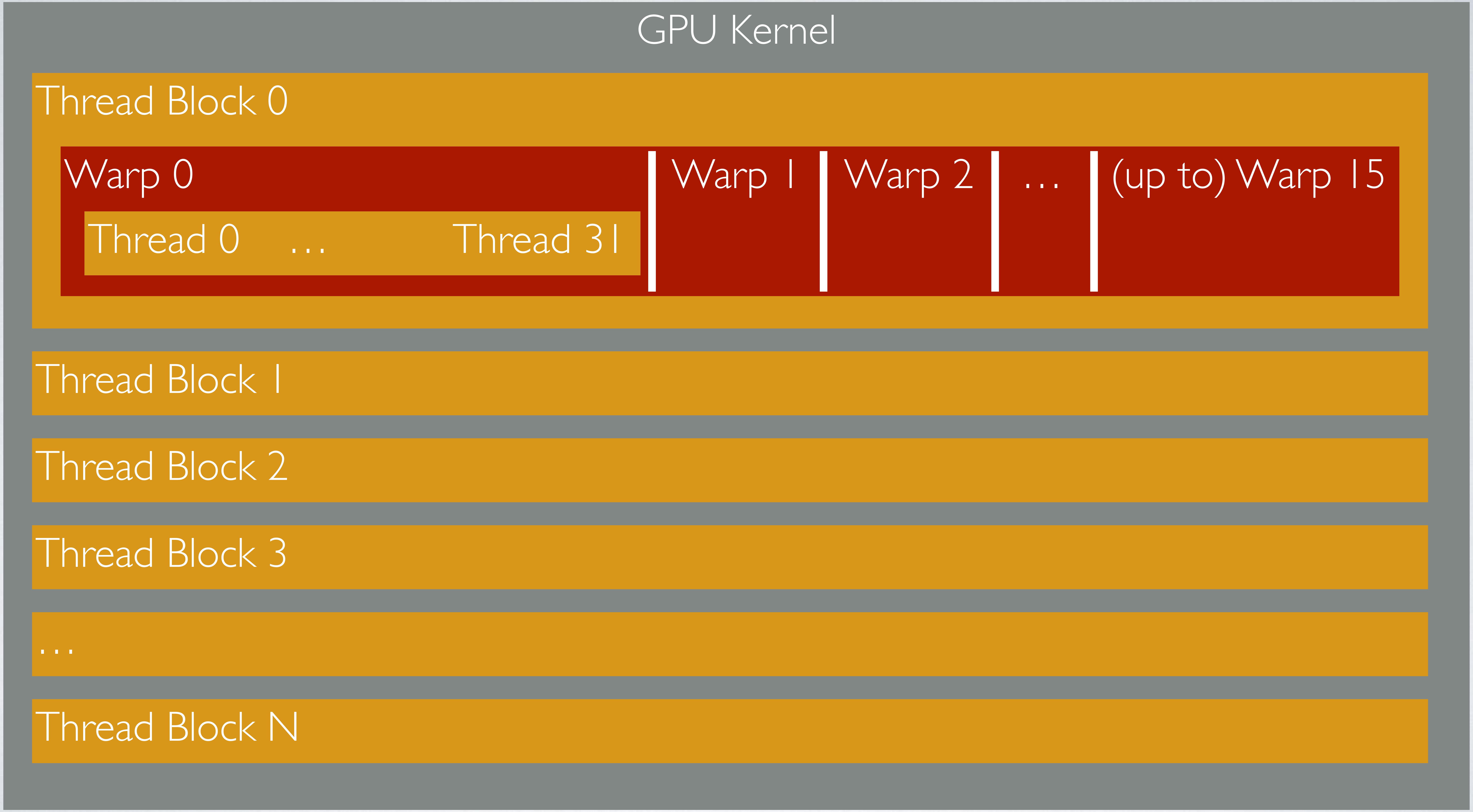


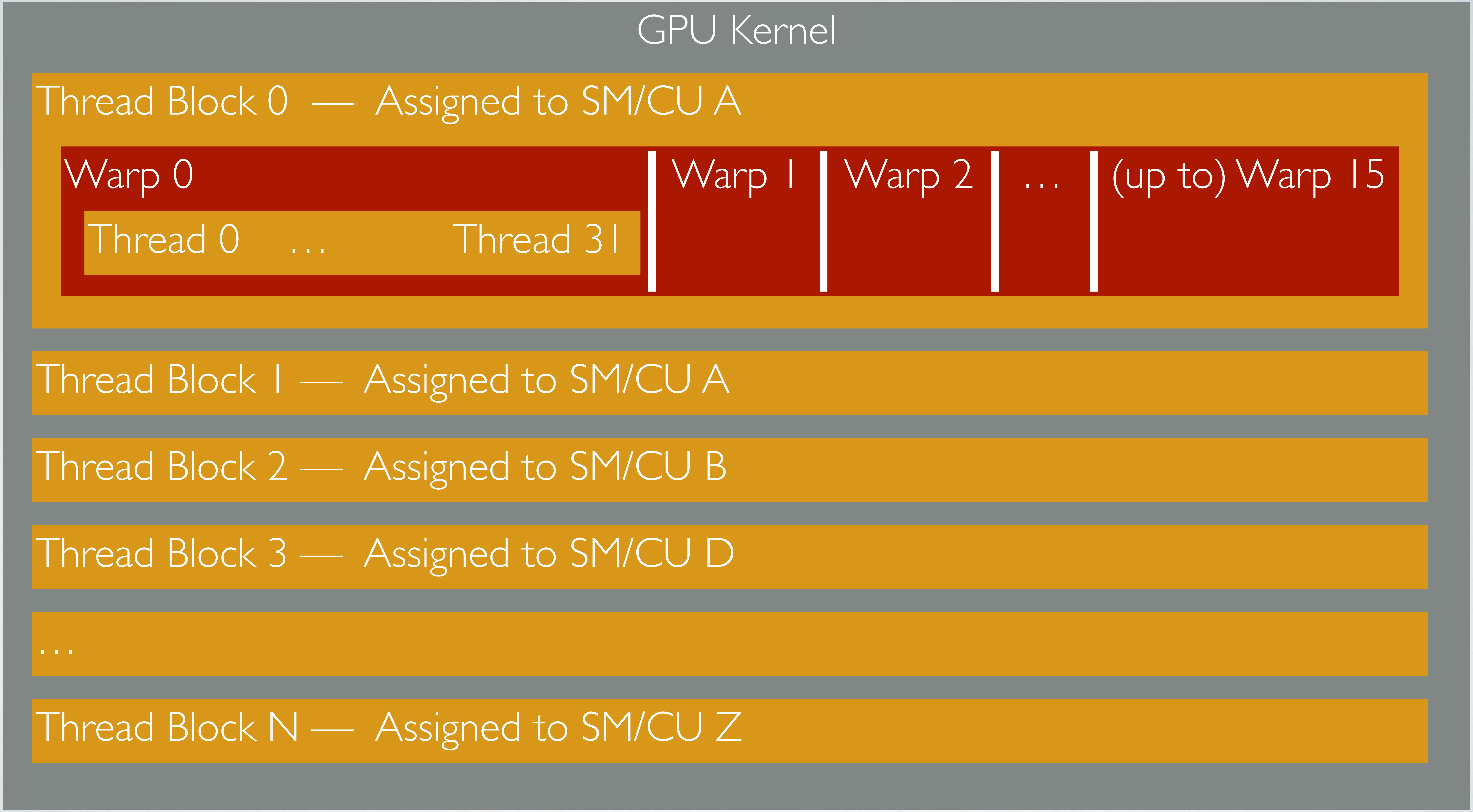
- Code running on CPU submits kernels to the GPU by calling a vendor library
- The vendor library writes the kernel launch request to a special area of GPU memory

Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf



- GPU command processor reads the kernel launch request, and then schedules thread blocks on individual SMs/CUs
- Individual SMs/CUs then independently execute these thread blocks


GPU Kernel
Thread Block 0
Warp 0
Thread 0
…
Thread 31
Warp 1
Warp 2
…
(up to) Warp 15
Thread Block 1
Thread Block 2
Thread Block 3
…
Thread Block N


GPU Kernel
Thread Block 0 — Assigned to SM/CU A
Warp 0
Thread 0 … Thread 31
Warp 1
Warp 2
…
(up to) Warp 15
Thread Block 1 — Assigned to SM/CU A
Thread Block 2 — Assigned to SM/CU B
Thread Block 3 — Assigned to SM/CU D
…
Thread Block N — Assigned to SM/CU Z

Example: Thread blocks from GPU kernel A and GPU kernel B execute simultaneously on SM/CUs A, B, D, and Z

GPU Kernel A
Thread Block 0 — Assigned to SM/CU A
Warp 0
Thread 0 . . .
. . .
Thread Block 1 — Assigned to SM/CU A
Thread Block 2 — Assigned to SM/CU B
Thread Block 3 — Assigned to SM/CU D
. . .
Thread Block N — Assigned to SM/CU Z
GPU Kernel B
Thread Block 0 — Assigned to SM/CU A
Warp 0
Thread 0 . . .
. . .
Thread Block 1 — Assigned to SM/CU A
Thread Block 2 — Assigned to SM/CU B
Thread Block 3 — Assigned to SM/CU D
. . .
Thread Block N — Assigned to SM/CU Z

Example 8-wide vector registers:

	Lane	Lane	Lane	Lane	Lane	Lane	Lane	Lane
v1	1.0	2.5	1.1	7.0	0.1	0.2	10.	104.
v2	80.	0.4	14.	2.3	3.14	5.12	19.	4.
v3	81.	2.9	15.1	9.3	3.24	5.32	29.	108.

⋮

Example vector instruction: **v3 ← v1 + v2**

Vector register:

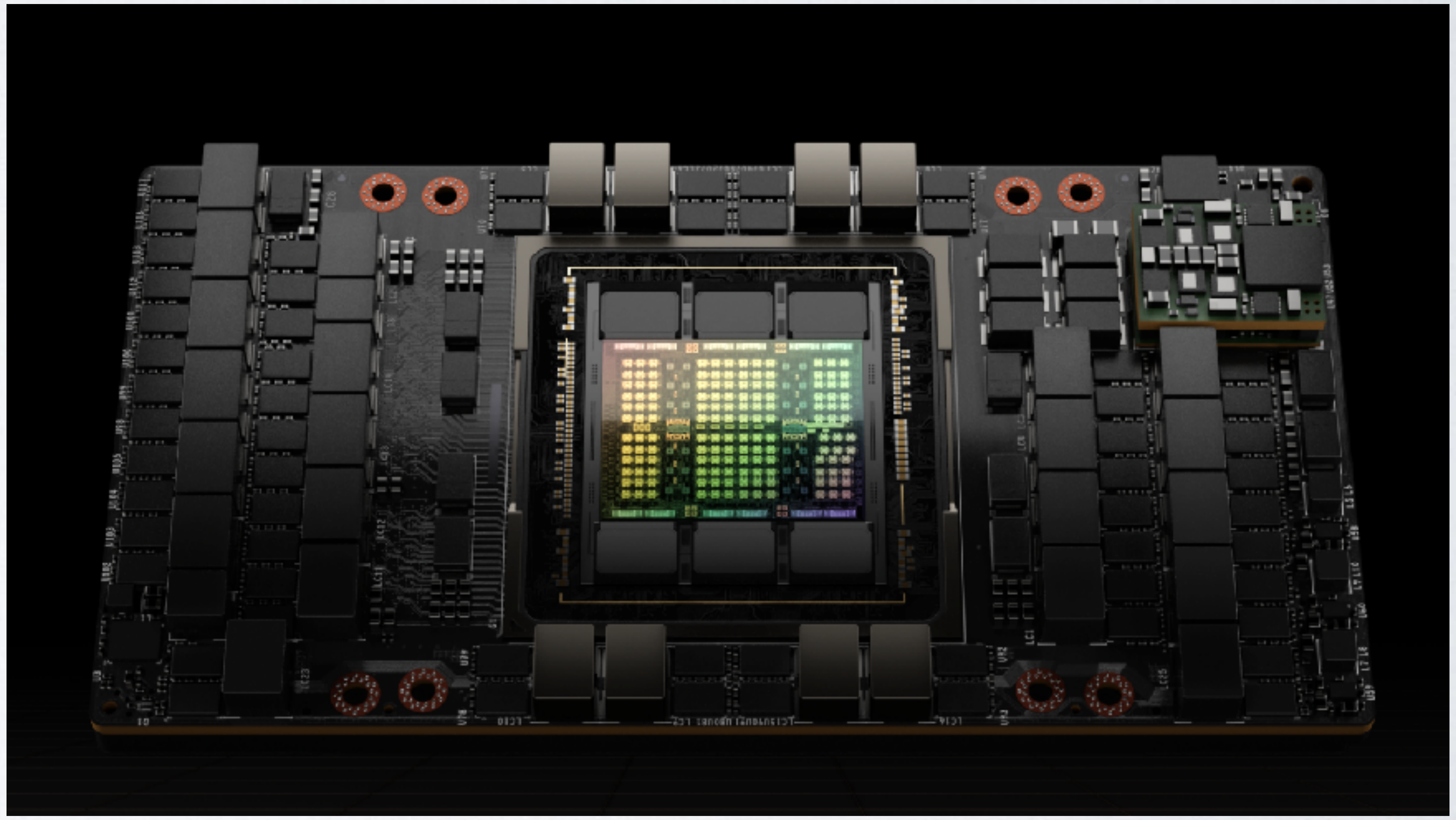
32- or 64-component vector that can be accessed with single clock cycle latency

- same data type (e.g., all floating point numbers)
- operate on all elements simultaneously

Each GPU thread (see previous slide) operates on a different vector component

WHAT IS A GPU?

- A bunch of CPUs that work really fast

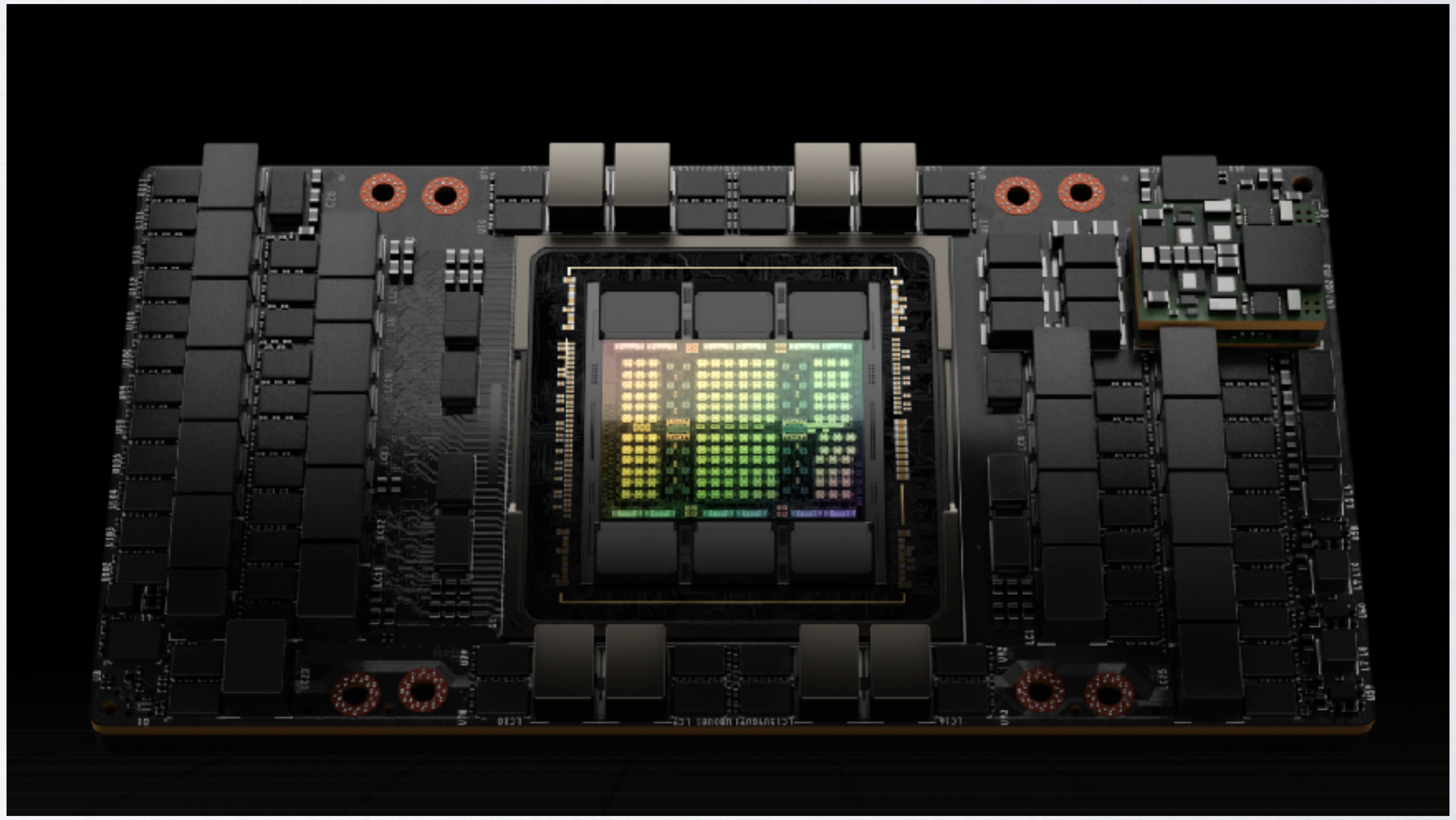


NVIDIA Hopper GPU

WHAT IS A GPU?

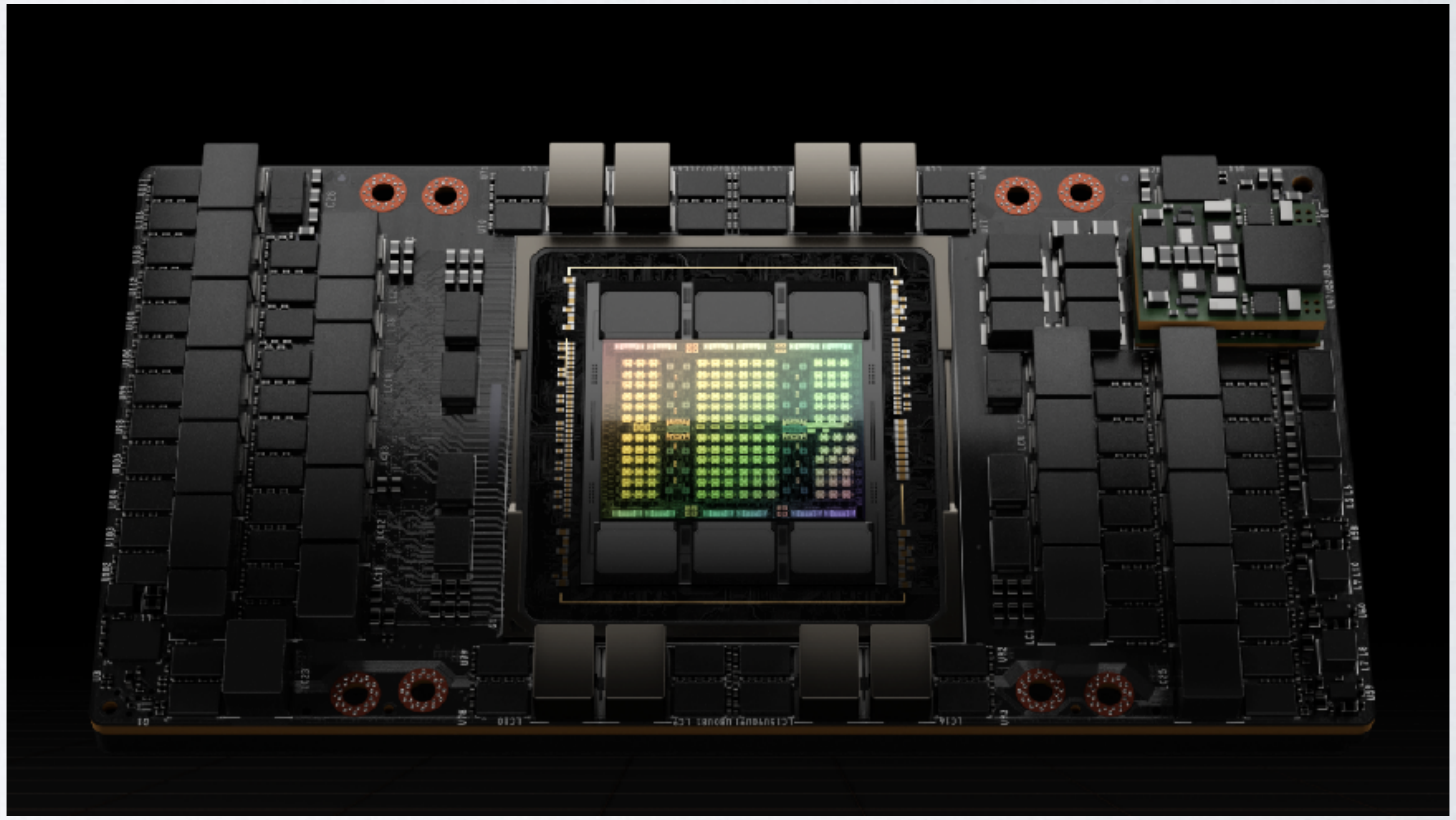
- ~~A bunch of CPUs that work really fast~~

NO!!


NVIDIA Hopper GPU

WHAT IS A GPU?

- A "vector" processor (i.e., adding vectors)
- High bandwidth: very high bandwidth read/write from the separate on-package GPU memory (about 1 order of magnitude higher BW than CPU main memory)
- High latency: several μs wait time for GPU kernels to start running
- "Simpler" hardware design than CPUs
- Significantly more complicated programming model

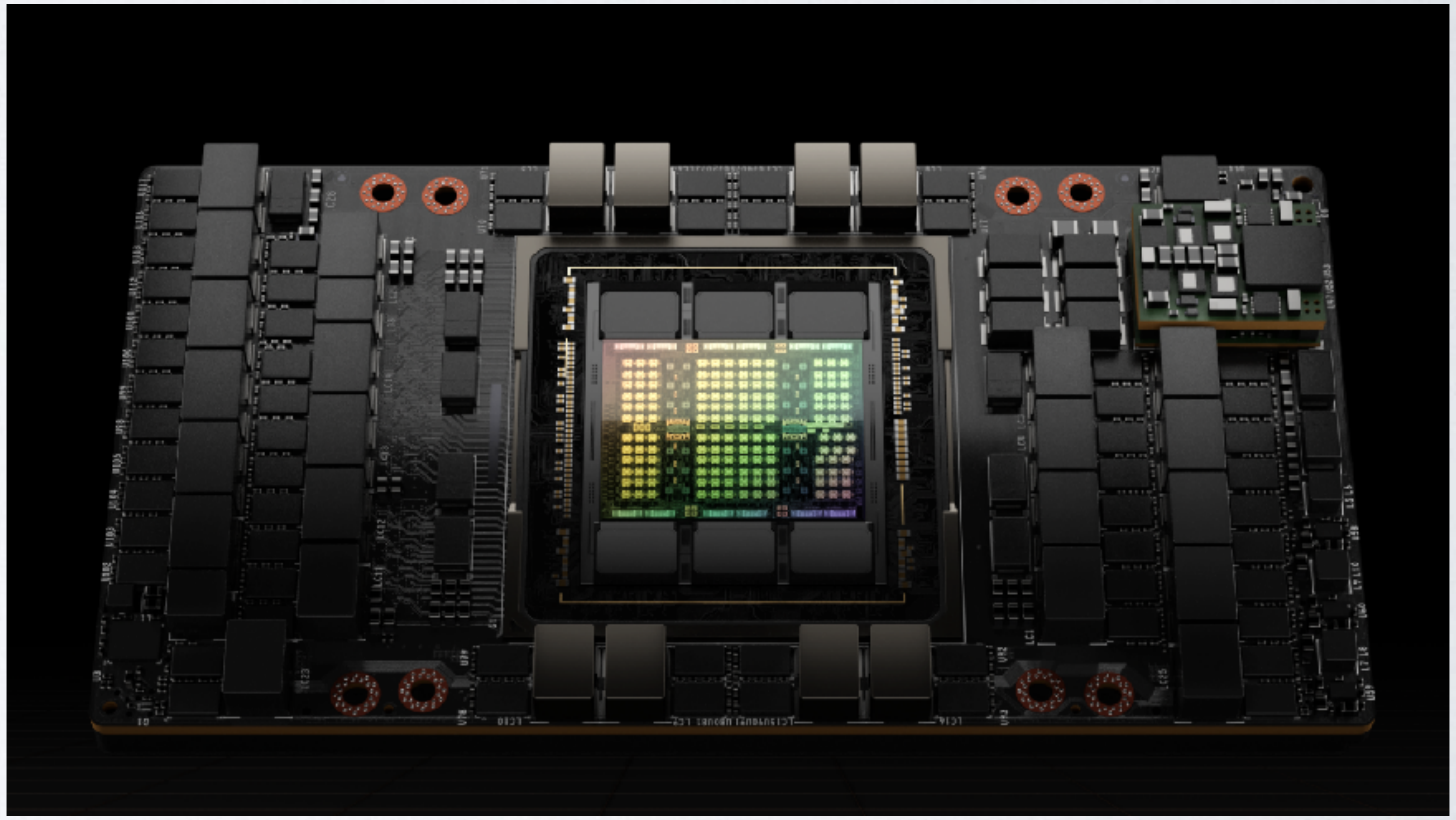


NVIDIA Hopper GPU

WHY ARE GPUS "FAST"?

- Wrong answer:
They have lots of cores.

- Right answer:
They are fast for certain computations that have high data parallelism:
  - Parallel processing designed for high throughput
  - Much higher memory bandwidth than CPUs


NVIDIA Hopper GPU

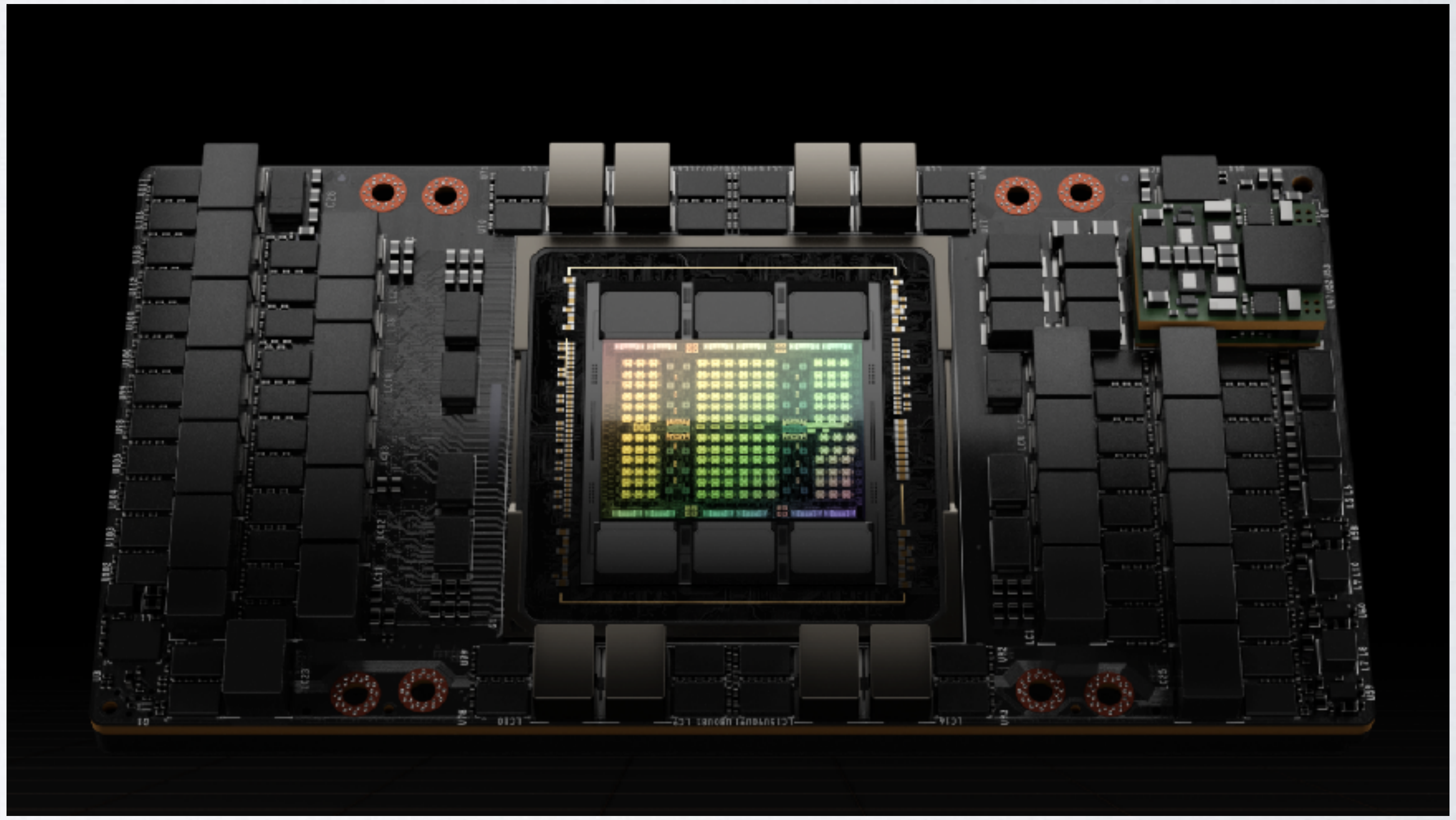
A CPU IS LIKE A TAXI

It gets a small amount of data processed with low latency, at the cost of low throughput.

By Pstrahl via Wikipedia - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=151972144

A GPU IS LIKE A SUBWAY

It gets a huge amount of data processed with high throughput, at the cost of having high latency.



NVIDIA Hopper GPU

WHAT CAN GPUS DO?

✓

- GPUs can:
  - Multiply/add/divide numbers in parallel
  - Fetch lots of data from ordered locations in memory
  - Perform direct memory copies to/from other GPUs

WHAT CAN'T GPUS DO?

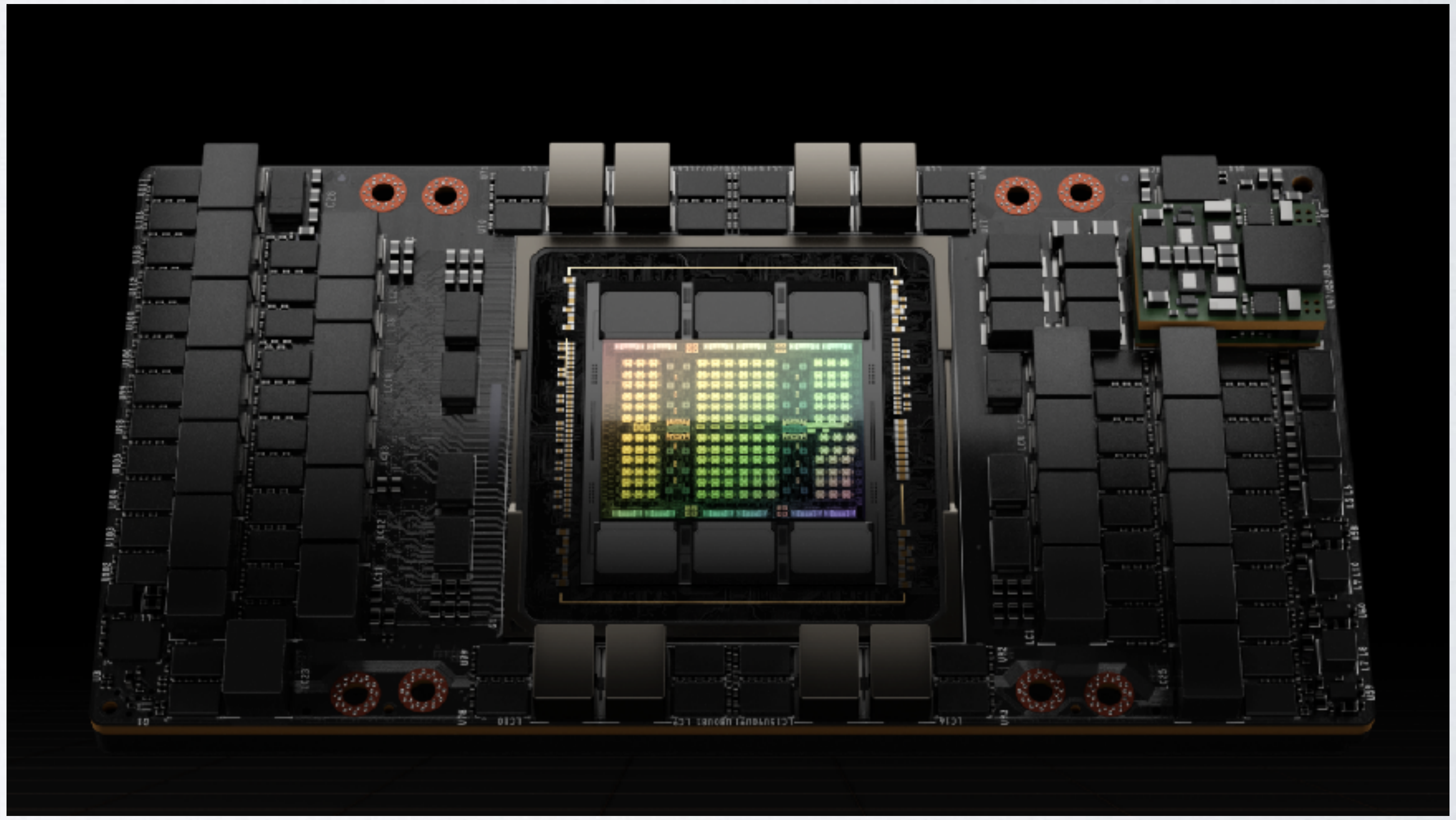
- GPUs can:
  - Multiply/add/divide numbers in parallel
  - Fetch lots of data from ordered locations in memory
  - Perform direct memory copies to/from other GPUs

- GPUs *can't*:
  - Efficiently read unpredictable/irregular locations in memory
  - Efficiently perform operations on a small amount of data
  - Perform I/O
  - Make MPI calls

SHOULD I COMPUTE ON THE GPU?

- *Ask yourself:*
  - Do I have >10^6 data items to process each iteration/timestep?
  - Can I keep my entire data set *(including temporary/work arrays)* in GPU memory for the entire duration of my computation?
  - Can I access my data items in a regular order in memory?

- *If yes to <u>all</u>, then it's a good candidate for running on the GPU*



NVIDIA Hopper GPU

SHOULD I COMPUTE ON THE GPU?

- *Ask yourself:*
  - Do I have >10^6 data items to process each iteration/timestep?

    Required to saturate SM/CUs with a sufficient number of threads + amortize launch overhead
  - Can I keep my entire data set *(including temporary/work arrays)* in GPU memory for the entire duration of my computation?

    Required to avoid cost of transferring data from CPU memory to GPU memory
  - Can I access my data items in a regular order* in memory?

    Required to achieve full bandwidth transfers from GPU memory
- *If yes to all, then it's a good candidate for running on the GPU*

**Strictly, memory accesses must coalesce.*

WHAT APPLICATIONS PERFORM WELL ON GPUS?

- PDE solvers on structured grids
  - Hydrodynamics
  - Solid mechanics / finite element analysis
  - Climate / weather simulation
- Dense linear algebra
- Particle-in-cell (PIC) plasma codes
- All-pairs N-body

WHAT APPLICATIONS PERFORM WELL ON GPUS?

✓✓

- PDE solvers on structured grids
  - Hydrodynamics
  - Solid mechanics / finite element analysis
  - Climate / weather simulation
- Dense linear algebra
- Particle-in-cell (PIC) plasma codes
- All-pairs N-body

✓

- Monte Carlo particle transport

WHAT APPLICATIONS *DON'T* PERFORM WELL ON GPUS?*

- PDE solvers on grids with irregular/ arbitrary connectivity (i.e., fully unstructured meshes)
- Sparse linear algebra with irregular matrix structure
- Tree-based N-body

**Yes, there are exceptions, but this is typically true.*

WHAT APPLICATIONS ARE A MIXED BAG ON GPUS?*

✗

- PDE solvers on grids with irregular/ arbitrary connectivity (i.e., fully unstructured meshes)
- Sparse linear algebra with irregular matrix structure
- Tree-based N-body

?

- Fast Fourier transform (FFT)
  - Historically poor performance, but much better on new GPUs
- Sparse linear algebra with regular matrix structure
  - Requires a large number of matrix elements per GPU (at least ~200^3)

**Yes, there are exceptions, but this is typically true.*

WHAT WE LEARNED

- "What is exascale supercomputing?"
- "What is a CPU?"
- "What is a GPU?"
- How GPUs execute programs
- What kinds of algorithms and applications perform well on GPUs due to their unique architectural features

Practical exercises

- **Go here: https://github.com/BenWibking/cmse822-gpu-exercises**

Rosetta Stone for GPUs

C/C++	CUDA (NVIDIA GPUs)	HIP (AMD GPUs)
malloc()	cudaMalloc()	hipMalloc()
free()	cudaFree()	hipFree()
	cudaMemcpy()	hipMemcpy()