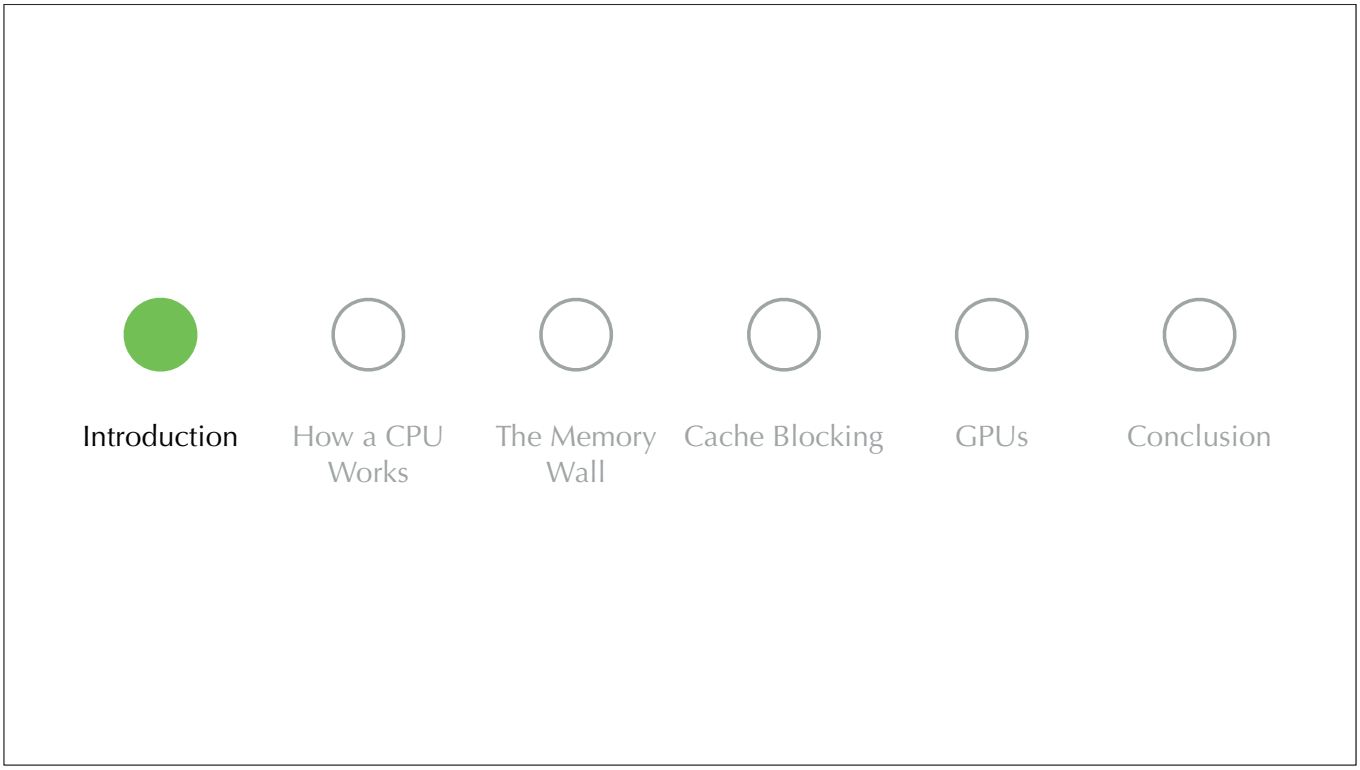
Elements of Modern Computing Hardware for Computational Fluid Dynamics

F.D. Witherden

Department of Ocean Engineering
Texas A&M University



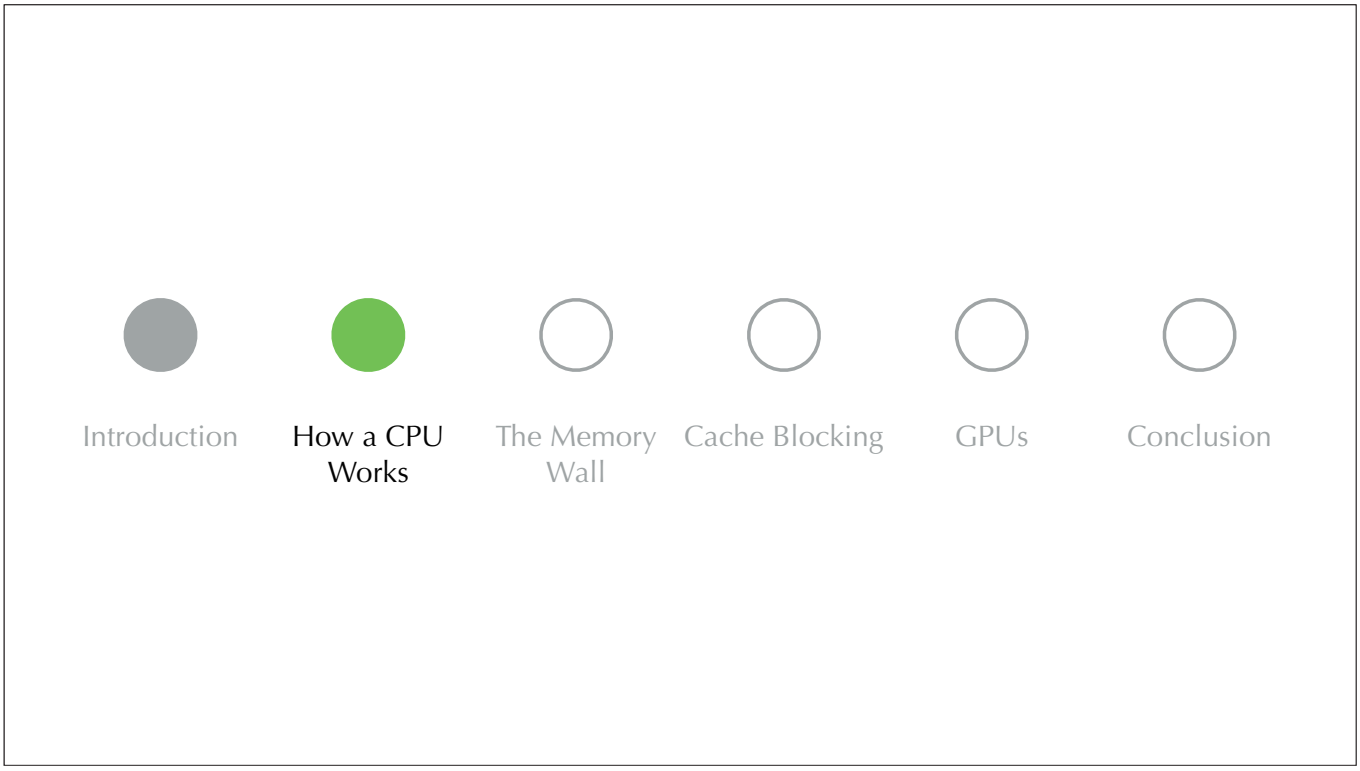
Introduction



- Computational fluid dynamics (CFD) is the bedrock of several high-tech industries.

Introduction

- However, over the last decade—on a cost basis—the performance of many industrial CFD codes has plateaued.
- In this presentation we will **investigate the root cause** of this and review alternative coding paradigms and hardware that can **get solver performance back on track**.



How a CPU Works

- CPUs perform work by **executing a series of simple instructions**.
- These instructions **manipulate data stored in registers**.
- A register is a **small region** of ultra-fast memory located on the chip itself.

How a CPU Works

- The data itself can be broken up into two categories:

```
42
-1912
```

Integers
(32- or 64-bit)

```
3.14159
-0.88
```

Floating point numbers
(32- or 64-bit)

How a CPU Works

- The rate at which a processor can execute instructions is determined by its **clock speed**.
- This is usually somewhere between **2 and 5 GHz**.
- We remark here that **power consumption** scales with approximately the **cube of the clock speed**.

How a CPU Works

- This relationship places **practical limits** on how high a chip can be clocked and still be power efficient.
- The solution here is to **increase the amount of work** we do per clock cycle.

How a CPU Works

- One issue is that many instructions, especially those operating on floating point data, take **multiple cycles to return a result**.
- A solution to this is **pipelining** which enables a new instruction to start execution before the current one has finished.

How a CPU Works

- Consider evaluating $f = a + b + c + d$ as:

```
add f, a, b
add f, f, c
add f, f, d
```

- Now, let us assume `add` takes **two cycles** to complete:

6 cycles total

How a CPU Works

- As our code is currently structured it **does not matter** if our processor is pipelined or not: execution will **always take 6 cycles**.
- However, this can be resolved by rearranging our operations.

How a CPU Works

- Lets try $f = (a + b) + (c + d)$ as:

```
add f, a, b
add t, c, d
add f, f, t
```

- Again assuming `add` takes **two cycles** to complete:

Without pipelining:  6 cycles total

With pipelining:  5 cycles total

How a CPU Works

- Going beyond pipelining another a second strategy is that of **superscalar execution**.
- Here, we duplicate functional units enabling us to execute **multiple independent instructions per cycle**.

How a CPU Works

- If our processor can **simultaneously issue** independent `add` instructions then things get even better still:

```
add f, a, b
add t, c, d
add f, f, t
```

Pipelined superscalar:  4 cycles total

How a CPU Works

- Decoding and scheduling a large number of instructions however is a **power intensive operation**.
- As such the practical limit for modern high-end processors is around **eight instructions per cycle**.

How a CPU Works

Processor	Instruction Set	Issue Width (Instructions / Cycle)	Max Clock Speed (GHz)
Intel Golden Cove	x86-64	6	5.8
AMD Zen 4	x86-64	6	5.4
Apple Firestorm	AARCH64	8	3.2
Fujitsu A64FX	AARCH64	4	2.2

How a CPU Works

- For numerical applications the key operation is the **floating point operation** or FLOP (+ or − or ∗).
- To improve efficiency most architectures support a **fused multiply-add** instruction (FMA) which computes:

$$c \leftarrow a \cdot b + c \quad \text{(two FLOPs)}.$$

How a CPU Works

- The best means of further improving performance is to increase the **amount of work done by each instruction**.
- This can be accomplished by having the instructions operate on **small vectors in lieu of simple scalars**.

How a CPU Works

- Also known as **single instruction multiple data** (SIMD) typical vector lengths are between 128- and 512-bits.
- SIMD capabilities are a core part of all recent processor architectures.

How a CPU Works

- Increasing the vector length is a simple means of **improving peak performance**.
- However, not all codes can fully utilise large vectors.
- As such general purpose processors are yet to **go beyond 512-bits**.

How a CPU Works

Processor	Vector Width	Multiply-Add Rate (Per Cycle)	Max DP FLOPs (Per Cycle)
Intel Golden Cove	512-bit	2 MADD	32
AMD Zen 4	512-bit	1 MADD 1 ADD	24
Apple Firestorm	128-bit	4 MADD	16
Fujitsu A64FX	512-bit	2 MADD	32

How a CPU Works

- Having reached the practical limit of what is possible for a single general purpose core, the simplest means of improving performance is to **replicate them**.
- This leads us to multi-core chips with the number of cores on a single package being **between 8 and 128**.

How a CPU Works

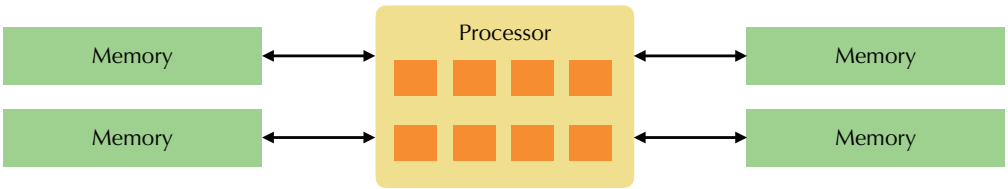
- A typical processor has either 16 or 32 **general purpose integer registers** and either 16 or 32 **vector registers**.
- Clearly, this is **not sufficient** to contain all of the data needed for any non-trivial problem.

How a CPU Works

- The solution here is to attach some memory to our processor.
- This is usually some kind of **dynamic memory** which is **cheap** and has **reasonable densities**.

How a CPU Works

- Memory is usually connected to the CPU through traces (wires) on a circuit board.



How a CPU Works

- This places practical limits on the latency and bandwidth of main memory.
- Specifically latency is usually **~50 ns** and bandwidth for an **eight channel DDR4** configuration is **~250 GiB/s**.

The Memory Wall

- To put these numbers into perspective a six-issue core running at 3 GHz can execute almost **1,000 instructions in 50 ns**!
- If we can **dual-issue 512-bit FMA's** this is about the same amount of time as is needed to perform 4,800 double precision floating point operations.

The Memory Wall

- Now, let us consider bandwidth.
- Consider a function to perform the following **'AXPY' operation**:

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y},$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar.

The Memory Wall

- This simple vector addition operation is a **building-block of many linear algebra kernels**.
- Running through our vectors each loop iteration requires us to load a component of $\mathbf{x}$ and $\mathbf{y}$ and write a component of $\mathbf{y}$.

The Memory Wall

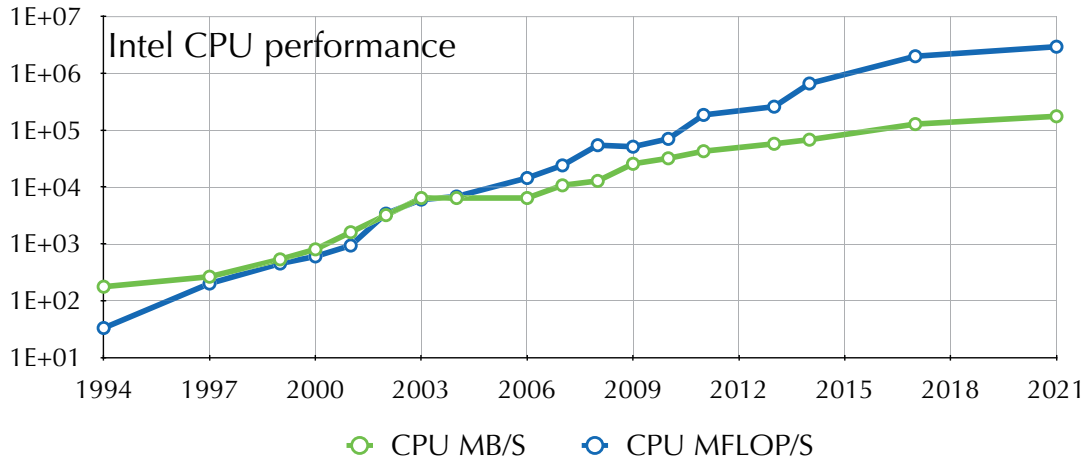
- On a 2 GHz core with 512-bit vectors that can sustain **two loads** and **one store** per cycle our bandwidth requirements are:

$$\left\{2 \times \frac{512}{8} + \frac{512}{8}\right\} \times 2 \cdot 10^9 = 358 \text{ GiB/s!}$$

The Memory Wall

- A **single core** hence needs **one and a half times** more bandwidth than our entire memory setup can provide.
- This memory bandwidth, however, is **shared amongst all of the cores on the chip**.

The Memory Wall



The Memory Wall

- Although it is possible to increase memory bandwidth it is **not economical at scale**.
- Most general purpose (non-HPC) applications are **not bandwidth limited** and thus it is not worth the extra expense and power.

Cache Blocking

- The standard approach for reducing the bandwidth requirements for a scheme is **kernel fusion**.

```
for (int i = 0; i < n; i++)
    a[i] += b[i];

for (int i = 0; i < n; i++)
    a[i] += c[i];
```

Bandwidth ~ 6*n*

```
for (int i = 0; i < n; i++)
    a[i] += b[i] + c[i];
```

Bandwidth ~ 4*n*

Cache Blocking

- Fusion is not a panacea however.
- Kernels become more difficult to write, test and maintain.
- Also requires access to the source since one **can't fuse across library calls**.

Cache Blocking

- An alternative to fusion on CPUs is cache blocking.
- Idea is to break up our loops into small blocks **b** such that the outputs **remain resident in cache**.

```
for (int j = 0; j < n; j += b) {
    for (int i = j; i < j + b; i++)
        a[i] += b[i];

    for (int i = j; i < j + b; i++)
        a[i] += c[i];
}
```

Cache Blocking

- Key advantage is that it enables existing **tried, tested, and optimised kernels to be used**—only now we call them more frequently with different starting offsets and smaller element counts.
- Not a new idea; has been used by BLAS for decades.

Cache Blocking

Intel Sapphire Rapids Xeon 2 Ghz / 56 cores	Capacity (KiB)	Latency (Cycles)	Bandwidth (Bytes / cycle)	Net Bandwidth (GiB/s)
L1 (Private per core)	48	5	128	13,351
L2 (Private per core)	2,048	14	~50	5,215
L3 (Shared)	1,920 (per core) 107,968 (56 cores)	88	< 32	< 1,000

Cache Blocking

- Effectiveness depends on the working set of the application relative to the size of the cache being blocked for.
- When solving the **Euler equations** using DG on a $p = 4$ hexahedra storing **U** and **F**(**U**) for **eight elements** requires 160 KB.

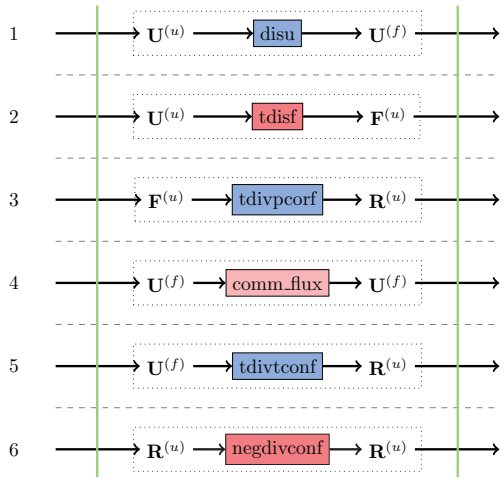
Cache Blocking

- Baseline arrangement for evaluating $\partial_t \mathbf{U} = -\nabla \cdot \mathbf{F}(\mathbf{U})$.

Matrix multiplication

Pointwise operation

Pointwise operation (indirect)

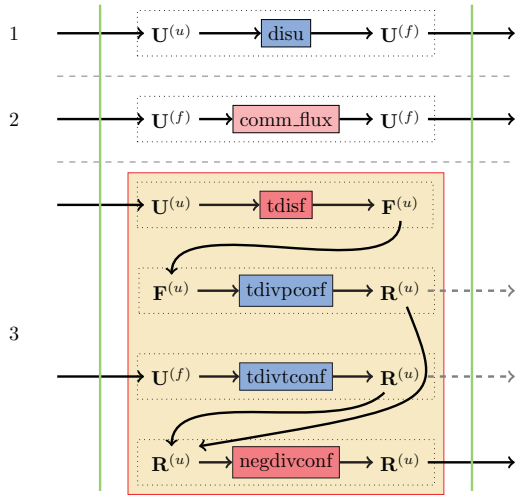


Cache Blocking

- Scheduling designed to maximise **overlapping of MPI communication with computation**.
- Net main memory bandwidth for one RHS eval:
  - ~59 KiB / curved $p = 3$ hex;
  - ~107 KiB / curved $p = 4$ hex.

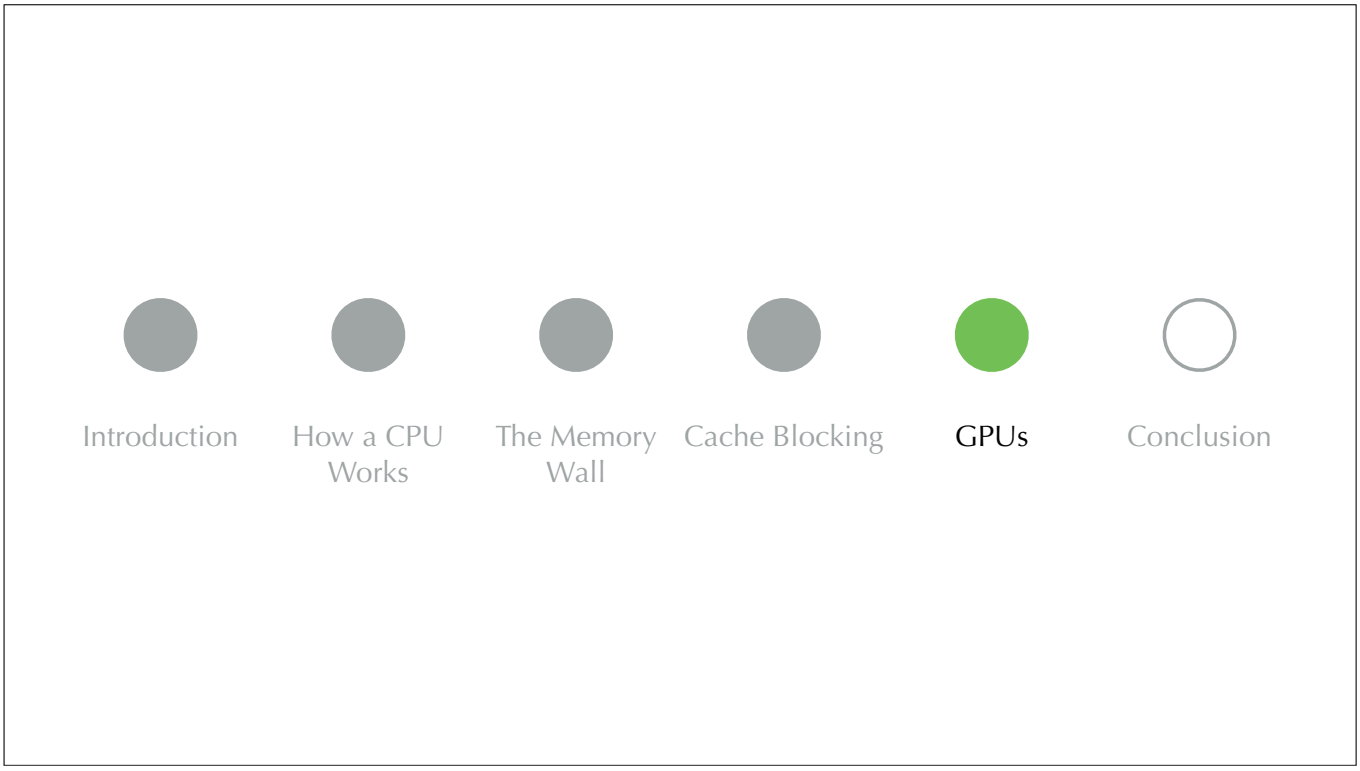
Cache Blocking

- Net main memory bandwidth for one RHS eval:
  - ~29 KiB / curved $p = 3$ hex;
  - ~49 KiB / curved $p = 4$ hex.



Cache Blocking

- This represents a twofold reduction in bandwidth!
- For Navier–Stokes the **reduction is closer to threefold** due to additional opportunities for data reuse.



GPUs

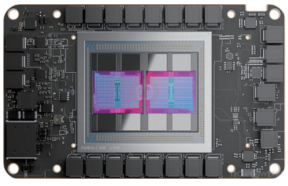
- All of the cache, wide issue width, and advanced execution capabilities in CPUs **consume large amounts of power and area**.
- GPUs remove this functionality in lieu of more execution resources enabling **super peak performance per Watt**.

GPUs

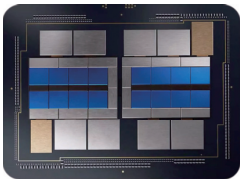
- This makes them more efficient, but also **more difficult to program**, as the hardware is doing less work for you.
- Moreover, the **minimum problem size** required to fully utilise a GPU is typically much larger than is required by a CPU.

GPUs

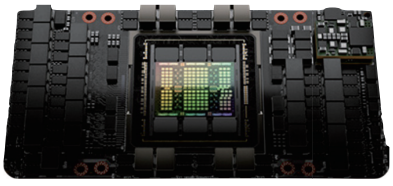
- Examples GPUs for **high-performance computing** include:



AMD MI250X



Intel Max



NVIDIA H100

GPUs

	Clock Speed (GHz)	Power (W)	DP TFLOP/s (Vector/Matrix)	Ratio (W per TFLOP/s)
Intel Sapphire Rapids (56 cores)	2.00	350	3.6 3.6	97.7 97.7
NVIDIA H100 (132 cores)	1.98	700	34.0 66.9	20.9 10.5
AMD Mi250X (2 × 110 cores)	1.70	560	47.9 95.7	11.7 5.9

GPUs

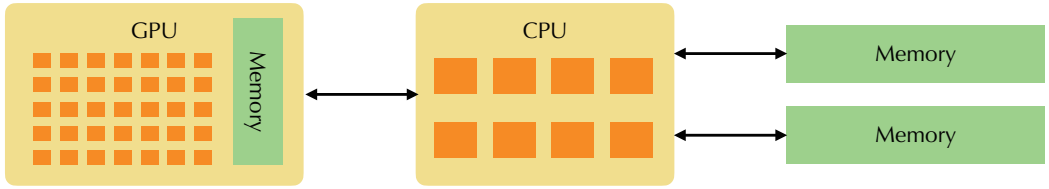
- GPUs also typically come with **high bandwidth memory**.
- However, this **comes at the cost of capacity**, which can be a problem for some (typically implicit) solvers.
- Furthermore, as cache blocking is not practical on GPUs they often **make less efficient use of bandwidth**.

GPUs

	Memory Type	Memory Capacity (GiB)	Memory Bandwidth (TiB /s)
Intel Sapphire Rapids (One Socket)	DDR5	1,536	0.25
NVIDIA H100	HBM3	80	3.0
AMD Mi250X	HBM2e	128 (2 × 64)	3.2 (2 × 1.6)

GPUs

- At the moment GPU memory is **usually managed separately to that of the host**.



PCIe 5
~63 GiB/s

DDR5
~250 GiB/s

GPUs

- Thankfully, there is a strong trend towards **fully unified memory** which will eliminate this issue.
- The first such HPC GPU doing this is the upcoming **AMD MI300A**, but we can expect other vendors to follow suit.
- The transfer problem is solved!

GPUs

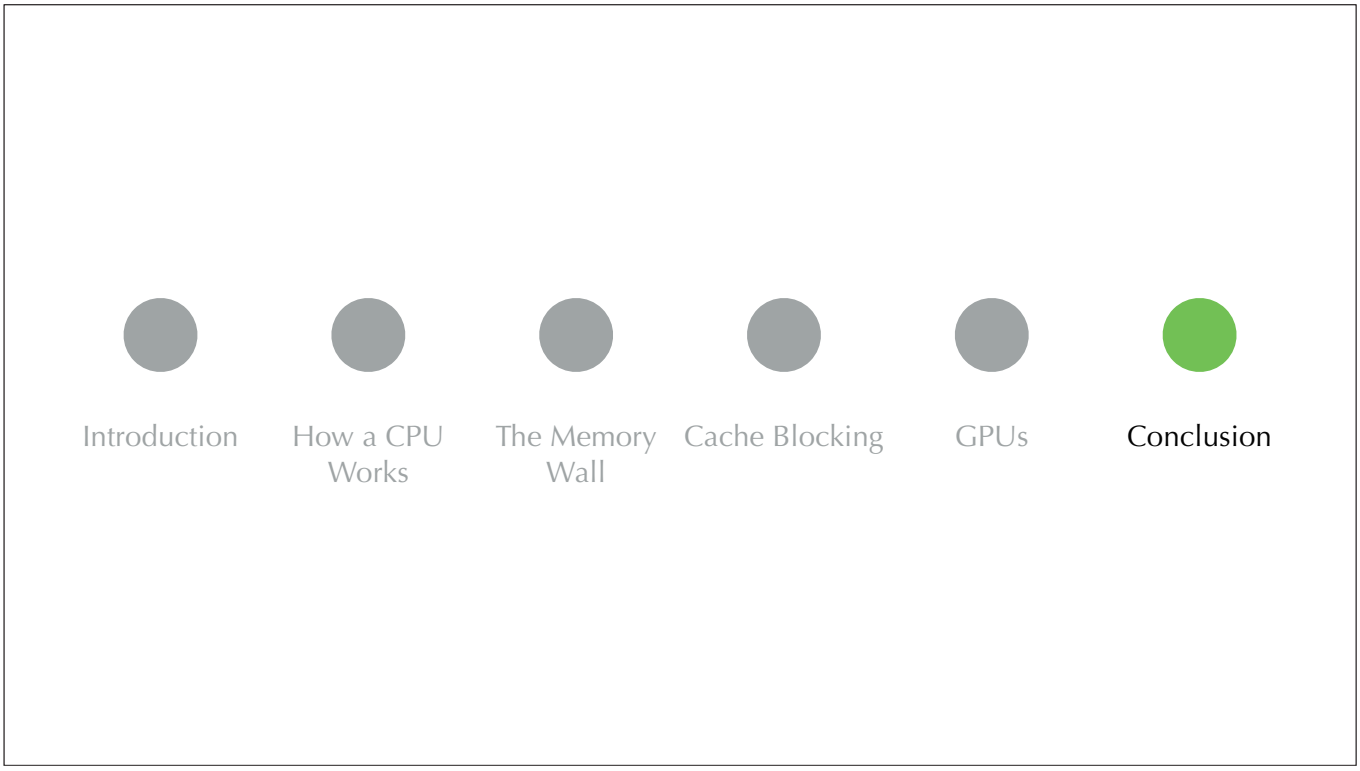
- Practically, the biggest downside of GPUs is the use of **vendor-specific programming languages**:
  - NVIDIA: CUDA.
  - AMD: HIP.
  - Intel: OpenCL and oneAPI.

GPUs

- This makes it difficult to achieve **performance portability** and can lead to **vendor lock-in**.
- Irrespective of which environment one uses there is one common problem: **kernel launch latency**.
- This makes it difficult to port codes **function-by-function** even if memory is unified.

GPUs

- As such porting a code to GPUs is a substantial undertaking and a lot of work is often required before observing any performance gains.
- Often it is easier to **rewrite a code from scratch**, e.g., Nek5000 to nekRS.



Conclusion

- Computing hardware for CFD—and HPC in general—is at an inflection point.
- Performance per Watt requirements means that **GPUs are probably here to stay**…
  - …but you'll probably need to **rewrite your code**.