HPC Architecture Basics

Introduction to High Performance Computing 2019

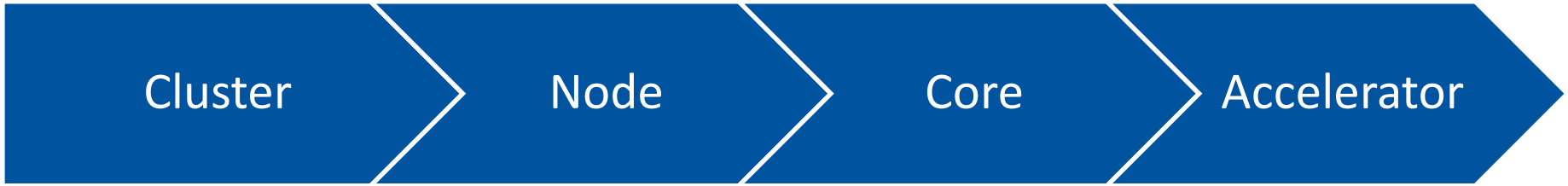
Dr. Christian Terboven

Levels of parallelism in hardware

- **A modern supercomputer may contain multiple levels of parallelism**
  - → Processor level parallelism: Superscalar, SIMD
  - → Node/Chip level: Several cores/processors run in parallel with access to the same memory
  - → System level: Several nodes run in parallel and are communicating over a network interconnect

- **Parallelism introduces overhead**
  - → Additional computational costs (cycles)
  - → Implementation (hours of work)
- **Overhead increases from processor to system level**

Agenda



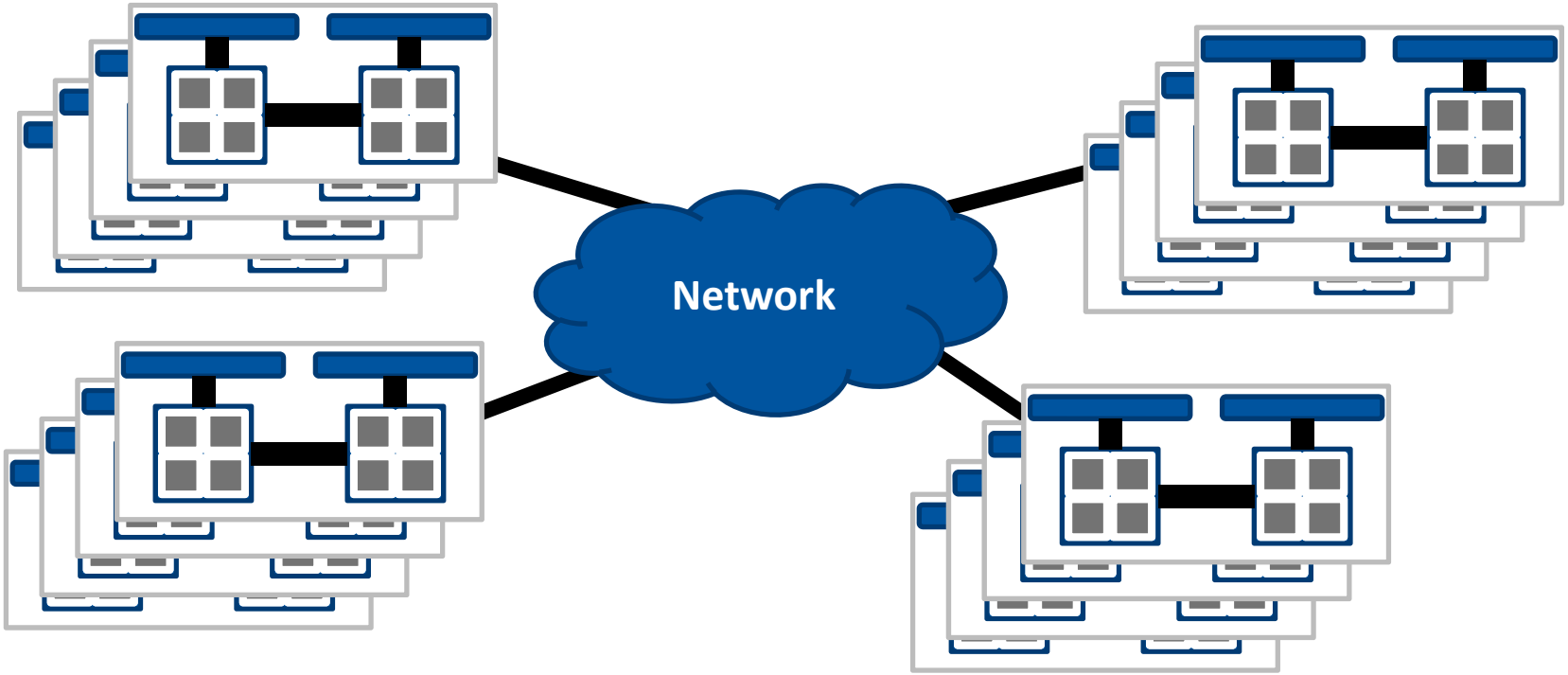
What is a Cluster?

Motivation

Clusters

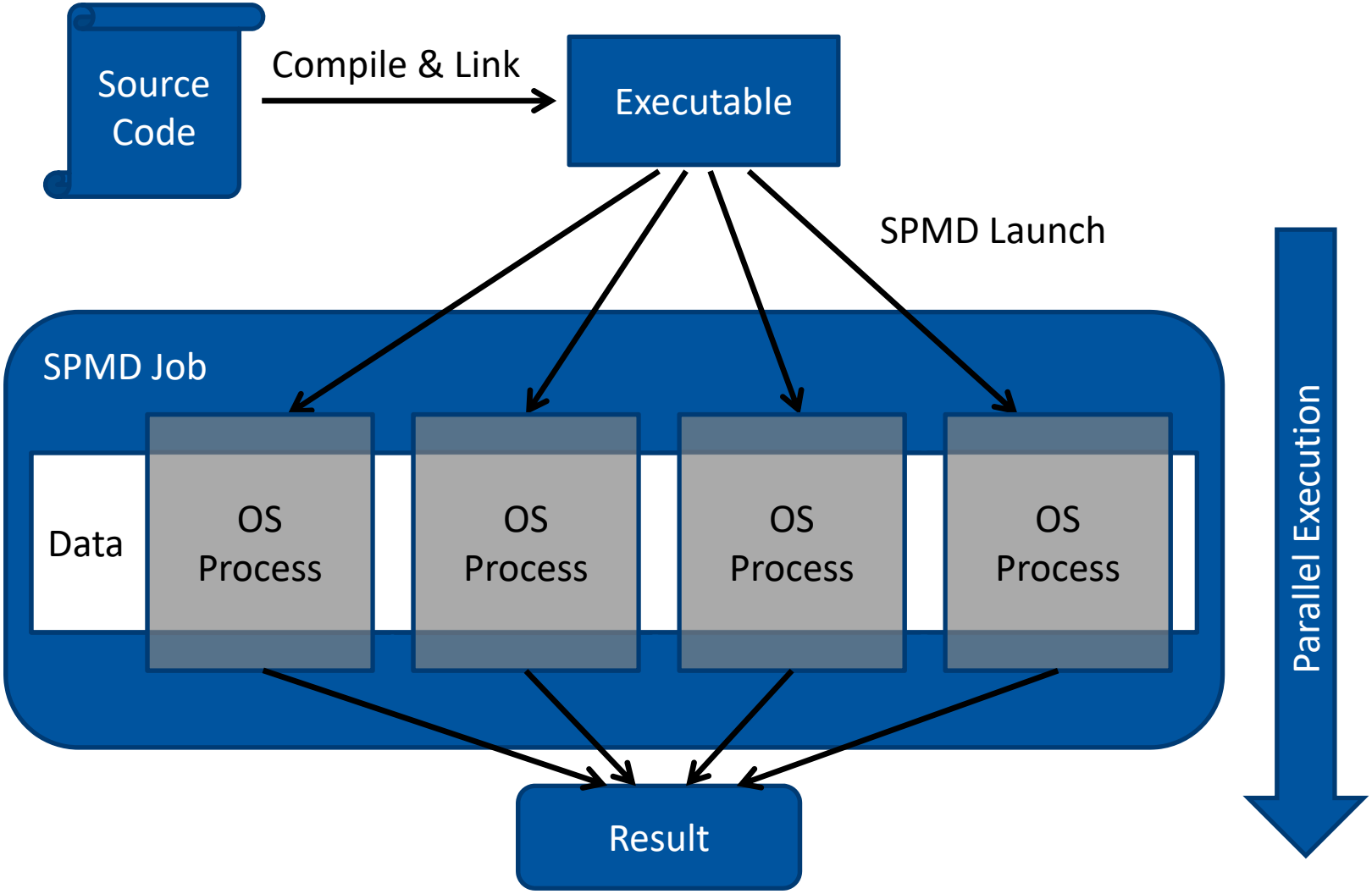
→ HPC market is dominated by distributed memory multicomputers (clusters)

→ Many nodes with no direct access to other nodes' memory



SPMD – Program Lifecycle

SPMD: Single Program Multiple Data

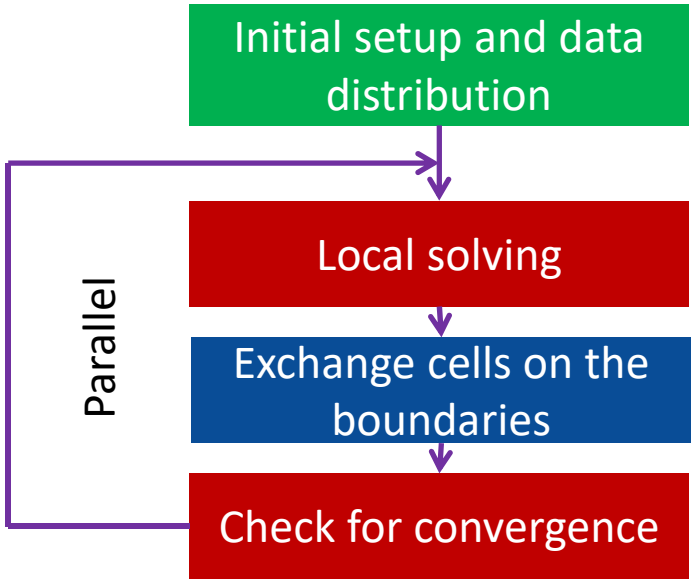


System level parallelism

- **System level parallelism**
  - → Overhead of parallelization: typically O(10,000+) cycles
  - → Workload needs to exceed O(10,000+) in computation time

- **Example: Domain decomposition in CFD: Mapping of 3D mesh to the processors**

- **Programming techniques**
  - → Data parallel approach
    - → Distribute data structures
  - → Parallel algorithms
  - → Explicit data exchange (MPI)



What is a Node in a Cluster?

Worksharing

- **Distribution of work among threads**

- **Most common use case: *for* loop**

  → Example: OpenMP

```
C/C++

int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

  → Distribution of loop iterations over all threads in a Team.

  → Scheduling of the distribution can be influenced.

- **Loops often account for most of a program‘s runtime!**

Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

Thread 2

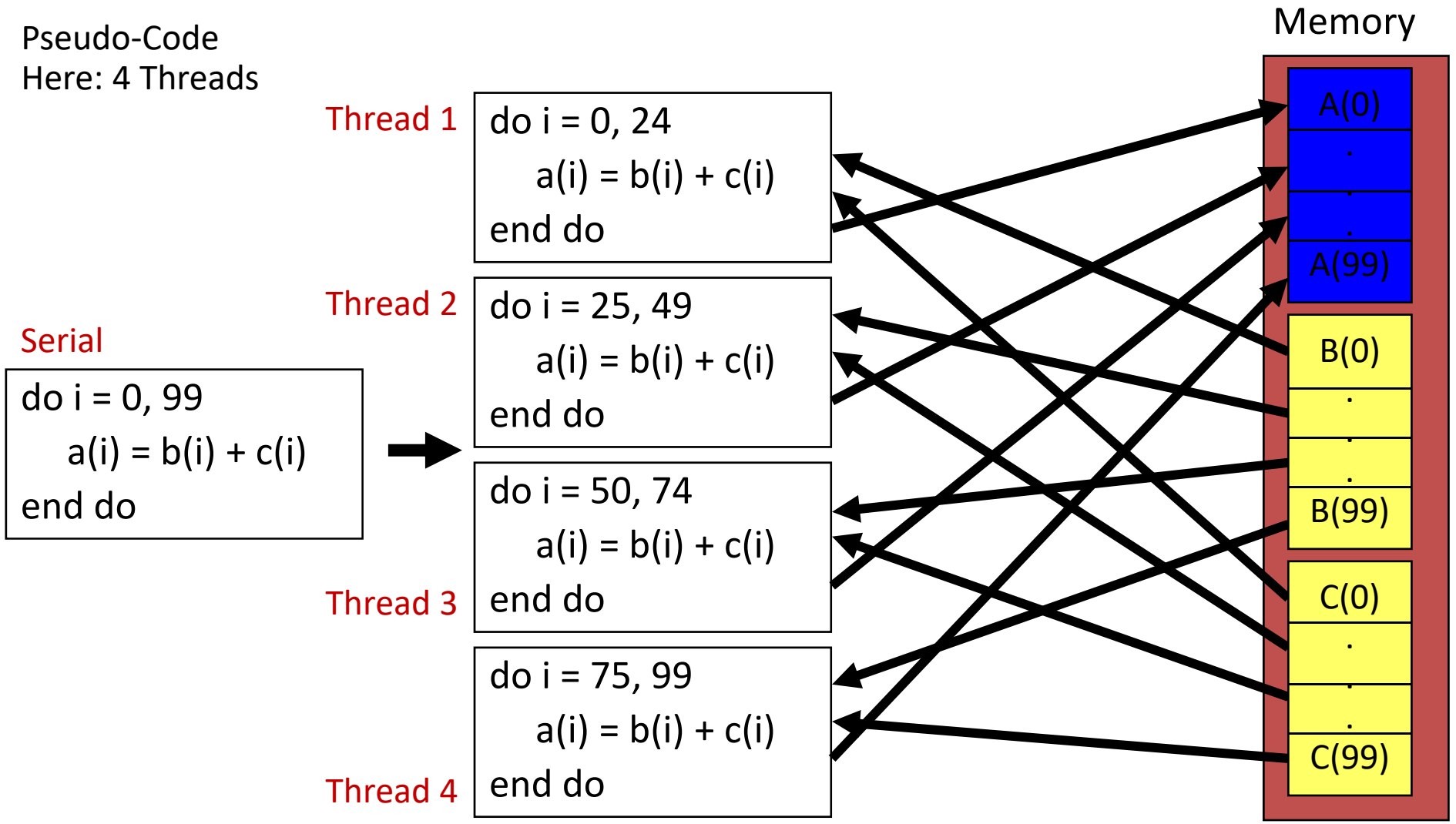
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```



Node level parallelism

- **Parallelization approach: threading**
  - → Thread administration/ synchronization overhead: O(100) – O(1000) cycles
  - → Workload should exceed O(100) – O(1000) cycles in computation time
- **Example: Matrix-Vector Multiplication (MVM)**
  - → Costs:
    - →inner loop: O(200)-O(1000)
    - →Parallelize outer loop!
- **Programming techniques**
  - → Auto-parallelization (by compiler)
    - →Works only in most simple cases
  - → Compiler directives (OpenMP)
  - → Explicit parallelization (pthreads, Win32 Threads)

```
for(i=0; i<N; ++i)
{
  C[i] = 0;
  for(j=0; j<N; ++j)
  {
    C[i] += A[i][j] * B[j];
  }
}
```

What is a Core?

Processor level parallelism

- **Parallelism at processor/ instruction level**
  - → Pipelining (overlap in execution: load, decode, execute)
  - → Superscalar (redundant arithmetical units: Multiplication, Addition, …)
  - → SIMD execution (e.g. 256 bit registers)

- **Example: Multiplication of two vectors**

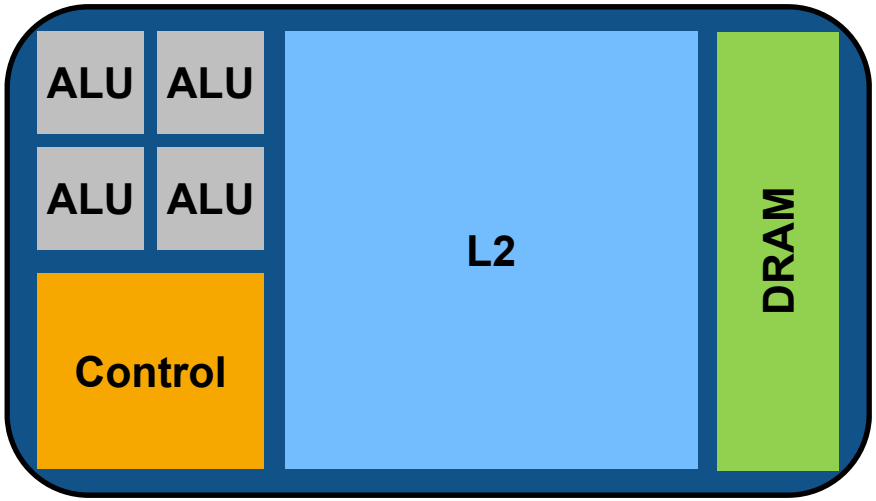
```
for(i=0; i<N; ++i)
{
  C[i] = A[i] * B[i];
}
```

- **Programming techniques**
  - → Exploitation of data or instruction level parallelism
  - → Code modifications: Unrolling, Cache reuse
  - → Compiler optimizations

What is a Accelerator in a Node?

Comparison CPU ⇔ GPU

- **Different design**



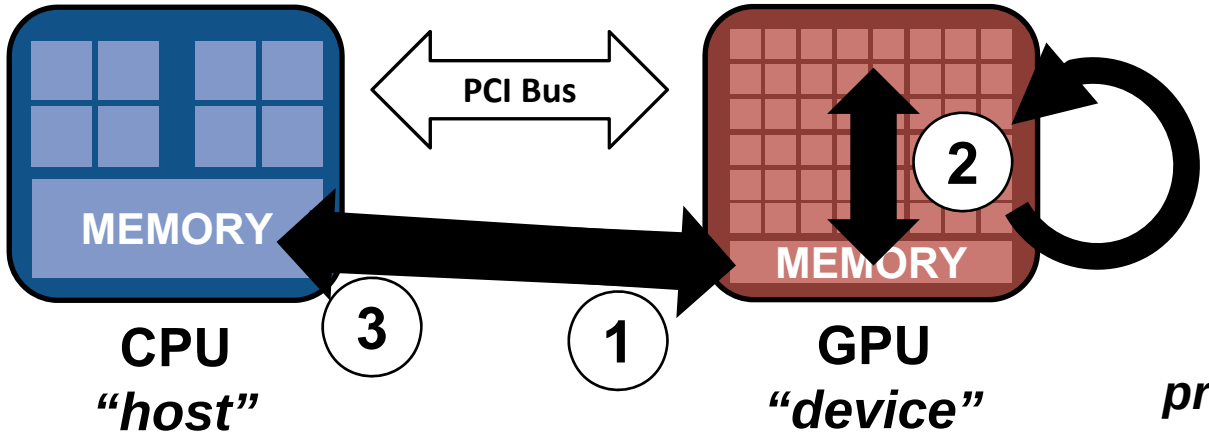
CPU

- → Optimized for low latencies
- → Huge caches
- → Control logic for out-of-order and speculative execution
- → **Targets on general-purpose applications**

GPU

- → Optimized for data-parallel throughput
- → Architecture tolerant of memory latency
- → More transistors dedicated to computation
- → **Suited for special kind of apps**

Offloading



We refer to "discrete GPUs" here.

- **Weak memory model**
  - → Host + device memory = separate entities
  - → No coherence between host + device
    - → Data transfers needed
- **Host-directed execution model**
  - → Copy input data from CPU mem. to device mem.
  - → Execute the device program
  - → Copy results from device mem. to CPU mem.

processing flow *(simplified)*