OpenMP Programming for Parallel/Vector Computing

Lecture 1: Introduction to Intel CPUs

Mike Giles

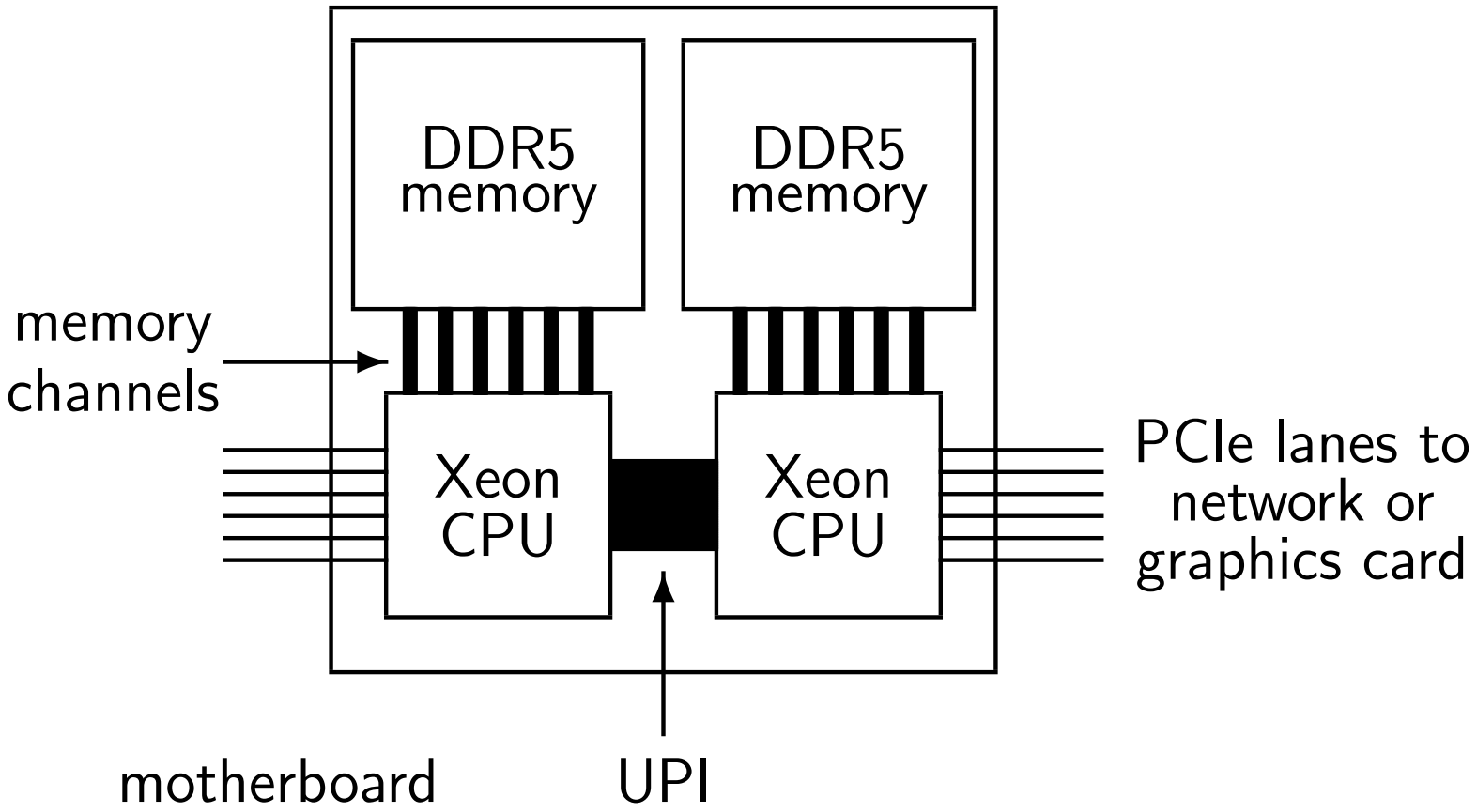
Mathematical Institute

Overview

- lecture 1: current Intel hardware
- lecture 2: an introduction to OpenMP, with application to a simple PDE solver
- lecture 3: more advanced OpenMP, with application to a Monte Carlo solver

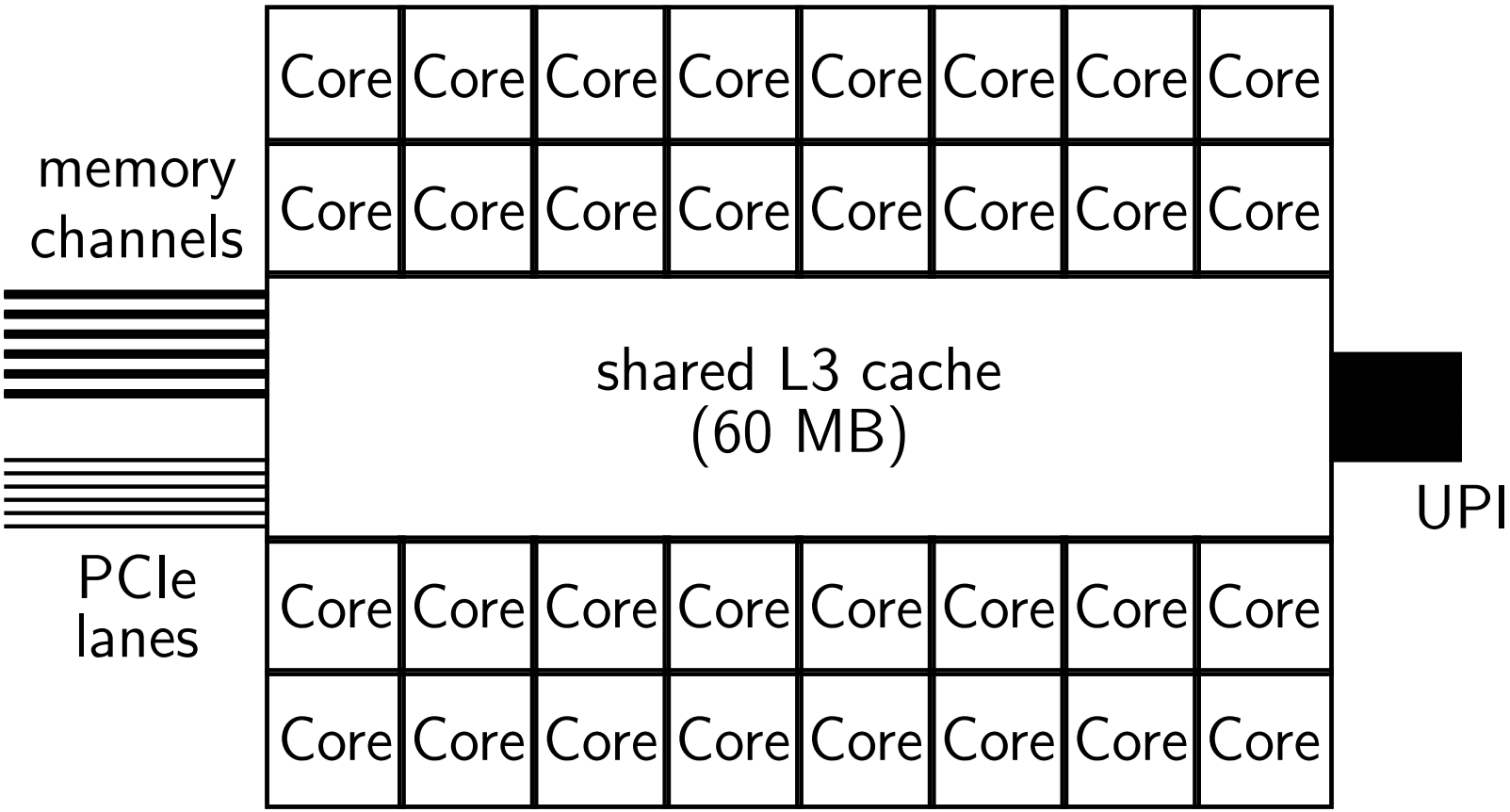
System view

A typical server has 2 multi-core Intel Xeon server chips, connected to a large amount of memory (DDR5) as well as network cards and perhaps a graphics card or two.

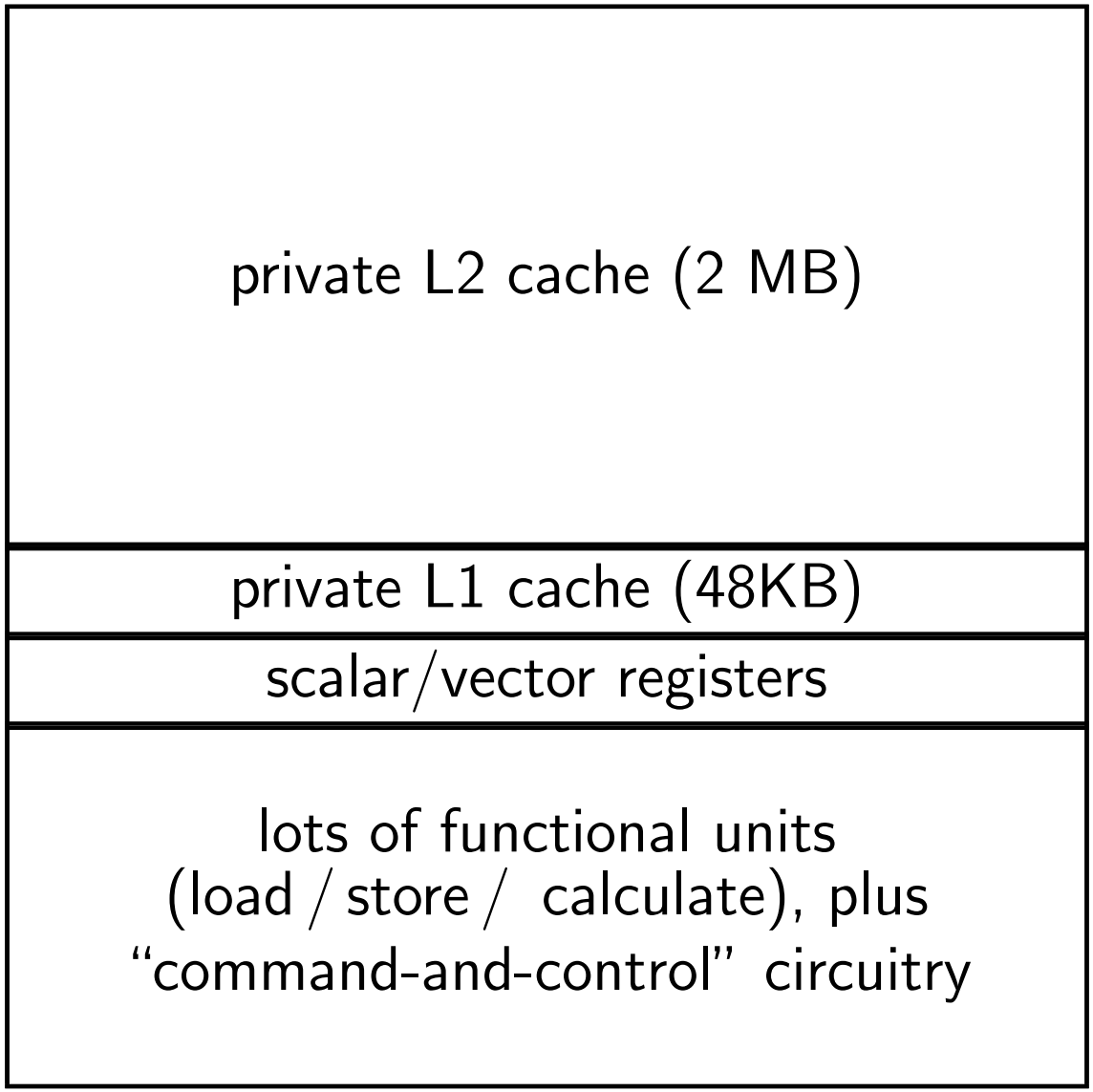


CPU view

Example: 32-core Intel Xeon Gold 6538Y+ processor (around $4000), part of the "Emerald Rapids" family of CPUs



Core view


private L2 cache (2 MB)
private L1 cache (48KB)
scalar/vector registers
lots of functional units
(load / store / calculate), plus
"command-and-control" circuitry

Core

Each core is

- **superscalar**
- with multiple **pipelined** functional units
- including **AVX512 vector units**
- with **out-of-order execution**
- and **branch prediction**
- and optional **hyperthreading**

Now to explain all of those buzzwords . . .

Superscalar

This just means that more than one instructions can be issued (started) every clock cycle.

I think the cores in current Intel Xeon CPUs can issue up to 8 different instructions each clock cycle, including a combination of

- load/store operations (moving data between L1 cache and registers, both scalar and vector)
- floating point operations (scalar or vector)
- integer operations (scalar or vector)

https://en.wikipedia.org/wiki/Superscalar_processor

AVX512 vector units

The latest Xeon server cores have 32 AVX512 vector registers, each of which can hold 8 `double` or 16 `float` variables.

The AVX512 vector unit can add two vector registers to give

$$c := a + b$$

where all three are vectors, not scalars – multiplication is similar.

It can even do a fused multiply-add (FMA)

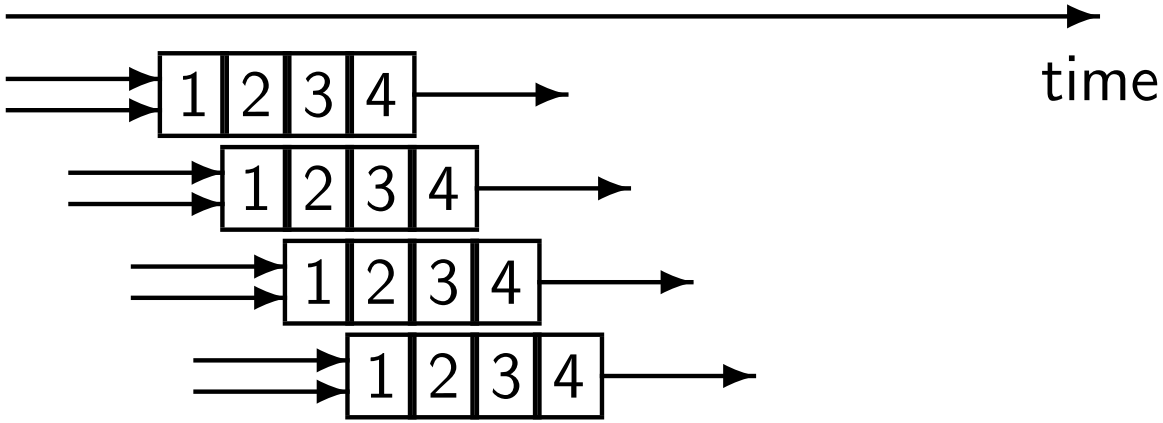
$$c := (a * b) + c$$

so you get two vector operations in one instruction

(It can also use a mask, e.g. to only add elements 0, 1, 3, 4, 6; there are 8 mask vector registers for this)

Pipelined units

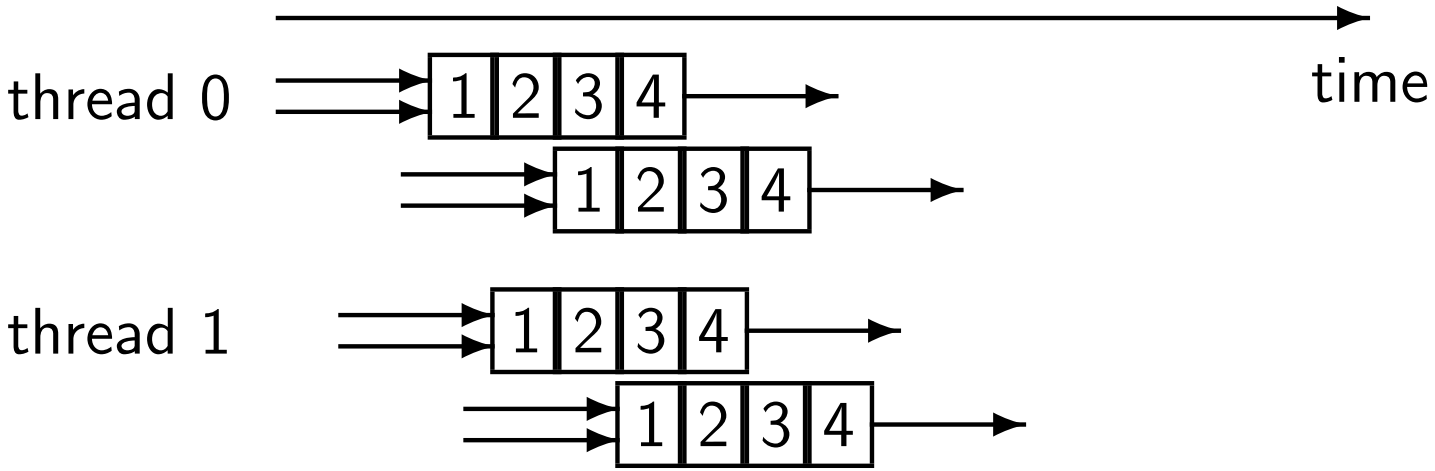
- scalar and vector operations are performed in multiple stages (3-6 for most floating point operations?) with overlapping execution



- **latency** is number of cycles for first instruction; **throughput** is number of additional cycles for next
- note that this may require later instructions to wait for inputs from earlier instructions

Hyperthreading

- this is an optional operating system setting which leads to two hardware threads per core, operating on alternate clock cycles
- each has their own set of registers, but they have to share the L1 and L2 cache
- the possible benefit is better use of pipelined units



an output is available as an input to the next-but-one instruction from the same thread

Out-of-order execution

Because of pipelines, clock cycles may be wasted if a previous instruction has not yet finished.

Much worse than this, a load operation may take 100's of cycles to fetch data from the main memory – potentially a huge waste of computation.

In out-of-order execution the core's control unit looks at a "window" of about 200 instructions, and will execute them in a different order if it's valid and the inputs are ready.

This adds **hugely** to the complexity of the core.

Branch prediction

Because of pipelining, code branching due to conditional tests can be expensive, since the test has to be evaluated to know what to do next.

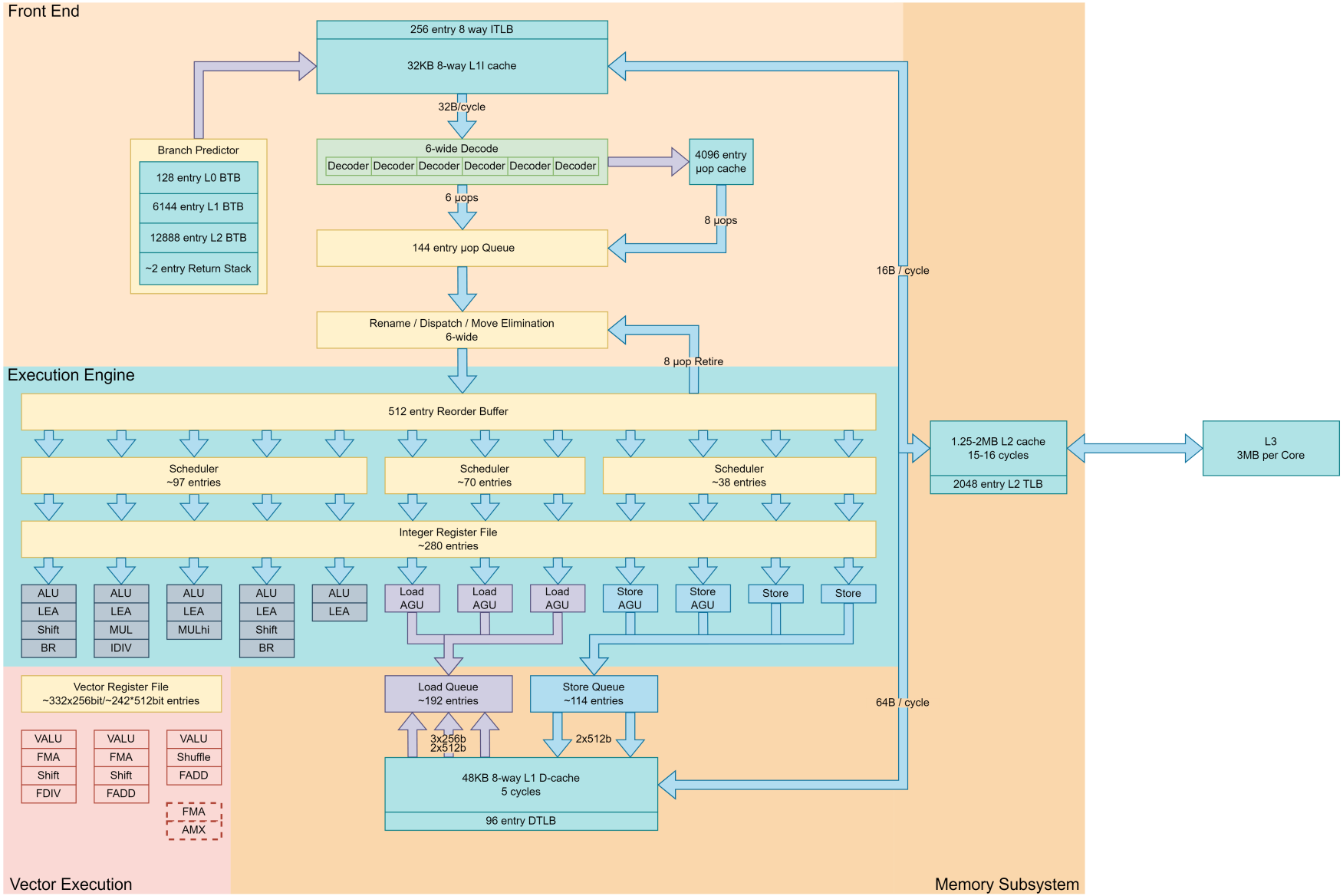
Branch prediction remembers what happened last time at this branch, guesses it will be the same this time, and works on that assumption.

There's some cleanup if the guess was wrong.

This improves performance, but again adds to the complexity of the core. The "command-and-control" circuitry is much more extensive than the floating point calculation hardware, but that balance has improved with the long AVX512 vector units.

Core block diagram

Golden Cove



Potential Performance

The Xeon Gold 6538Y+ has 32 cores, each with 2 AVX512 units, running at 2.2GHz, so the peak double precision (DP) performance is:

$$\underbrace{32}_{\#cores} \times \underbrace{2}_{\#AVXs/core} \times \underbrace{8}_{vector\ length} \times \underbrace{2}_{2\ ops/cycle} \times \underbrace{2.2\,\mathrm{GHz}}_{clock\ freq} \approx 2.25\,\mathrm{TFlops}$$

This one CPU is comparable to the fastest supercomputer from 25 years ago (Intel ASCI Red) but requires use of all the cores, and all of the vector units

Compiler often uses vector length 4 (why?) – half the performance

Without any vectorisation, lose a factor 8

Potential Performance

I think the corresponding total bandwidth from L1 caches into vector registers is

$$\underbrace{32}_{\#cores} \times \underbrace{2}_{\#loads/cycle} \times \underbrace{64\mathrm{B}}_{register\ size} \times \underbrace{2.2\,\mathrm{GHz}}_{clock\ freq} \approx 9\,\mathrm{TB/s}$$

and the bandwidth to the main DDR5 memory is

$$\underbrace{8}_{\#channels} \times \underbrace{8\mathrm{B}}_{channel\ width} \times \underbrace{5200\,\mathrm{MT/s}}_{transfers/second} \approx 330\,\mathrm{GB/s}$$

Recap

There are many levels of parallelism here:

- multiple CPU chips
- multiple cores in each CPU chip
- multiple functional units (superscalar)
- pipelines (overlapping execution)
- vector units
- hyperthreading

The compiler and the hardware will take care of most things, but to get close to full performance the programmer has to help too, and has to understand to some extent what is going on in the hardware.

Moving data

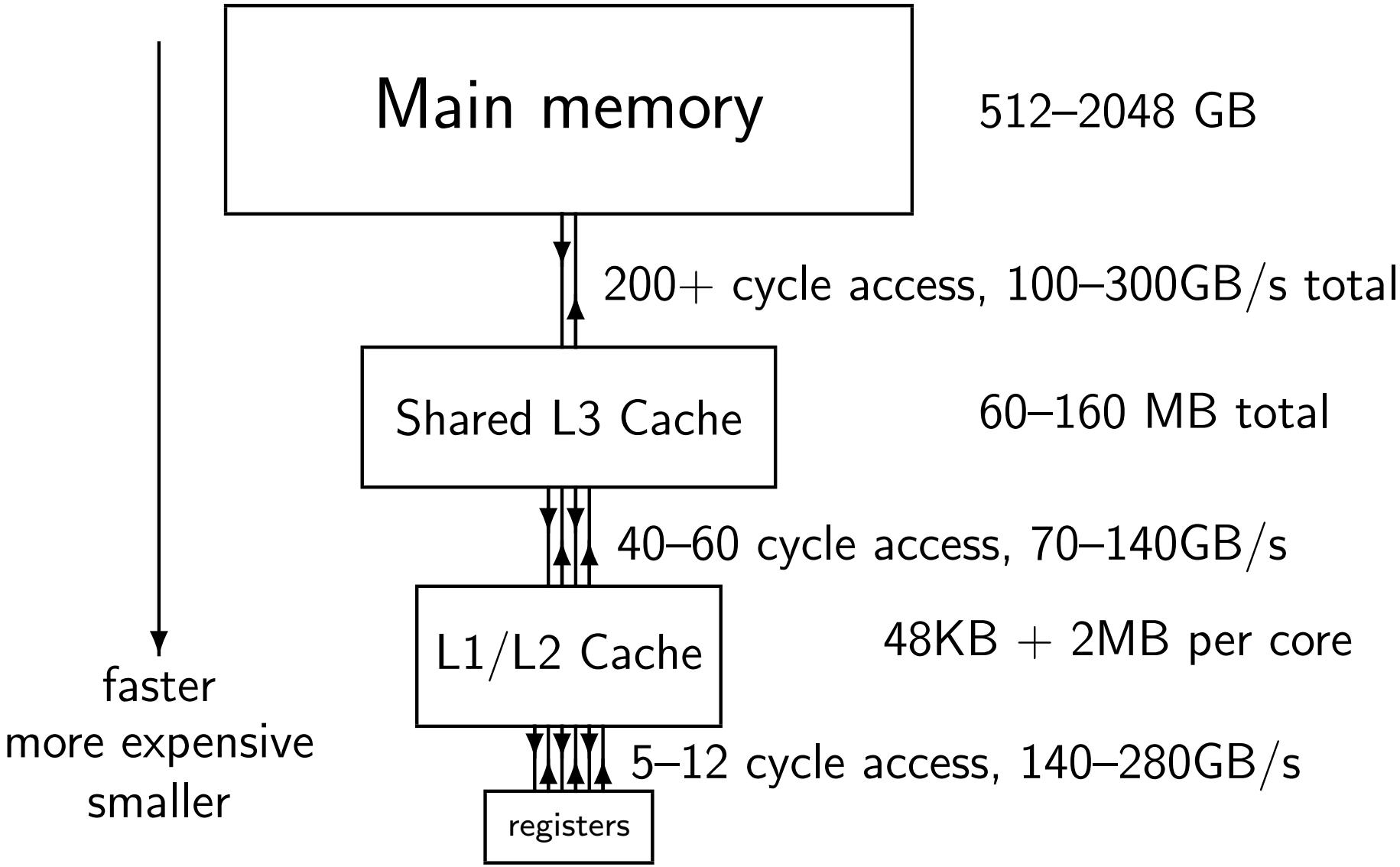
So far we have focussed on performing calculations (e.g. addition and multiplication).

Increasingly, this is an old-fashioned view. Now, the focus is on moving the required data.

In terms of both time and energy consumption, moving data usually costs more than performing calculations, and modern algorithms are being designed to minimise the amount of data movement.

Understanding data movement is **very** important to achieving good OpenMP performance.

Memory Hierarchy



Memory Hierarchy

Execution speed relies on exploiting data *locality*

- temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache
- spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a 'wide' bus (like a multi-lane motorway)

This wide bus is only way to get high bandwidth

Caches

The cache line is the basic unit of data transfer; standard size is 64 bytes $\equiv$ 512 bits $\equiv$ 8 `double` or 16 `float` items.

With a single cache, when the CPU loads data into a register:

- it looks for line in cache
- if there (hit), it gets data
- if not (miss), it gets entire line from main memory, displacing an existing line in cache (usually least recently used)

When the CPU stores data from a register:

- same procedure

There is a natural generalisation to multiple levels of cache

Importance of Locality

Typical server:
2 TFlops (assuming full vectorisation)
256 GB/s bandwidth to main memory
64 bytes/line

256GB/s $\equiv$ 4G line/s $\equiv$ 32G double/s

At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines $\Longrightarrow$ 1.3 GFlops

If all 8 variables/line are used, then this increases to around 10 GFlops.

To get up towards 2TFlops needs temporal locality, re-using data already in the cache.

Importance of Locality

Data reuse matters even within a single core.

Typical core:
50 GFlops (assuming full vectorisation)
128 GB/s bandwidth L1 – L2 cache
64 bytes/line

$$128\text{GB/s} \equiv 2\text{G line/s} \equiv 16\text{G double/s}$$

At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines $\Longrightarrow$ 0.7 GFlops

If all 8 variables/line are used, this increases to around 5 GFlops
if data is in L2 cache

To get up to 50GFlops needs reuse of data in L1 cache or registers.

Additional info

Complexities: 1) where can a particular line reside in cache?

Fully associative:

- each line can be anywhere
- hard to implement quickly if cache is large

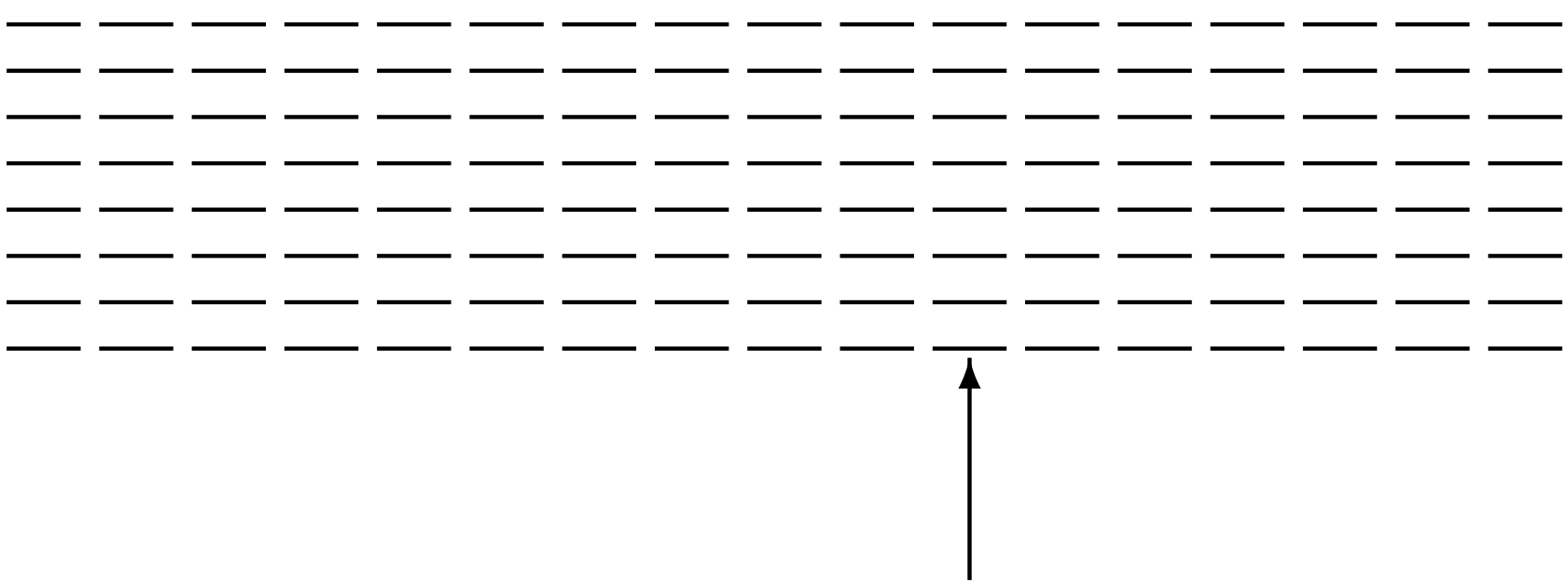
Direct mapped:

- each line has only one possible location
- very rapid
- displaced lines may still be needed, resulting in more cache misses for a given cache size

Additional info

Usual compromise: set associative cache in which each line can be anywhere within a subset of the cache

I think the Raptor Cove cores in Intel's Emerald Rapids CPUs use 12-way set associative for L1, 10-way for L2, 12-way for L3



set of possible locations for a particular cache line

Additional info

Complexities: 2) what happens when a cache line is modified?

Write-through cache:

- modified line is immediately written to higher level (cache or main memory)
- higher level stays up-to-date
- generates lots of memory traffic

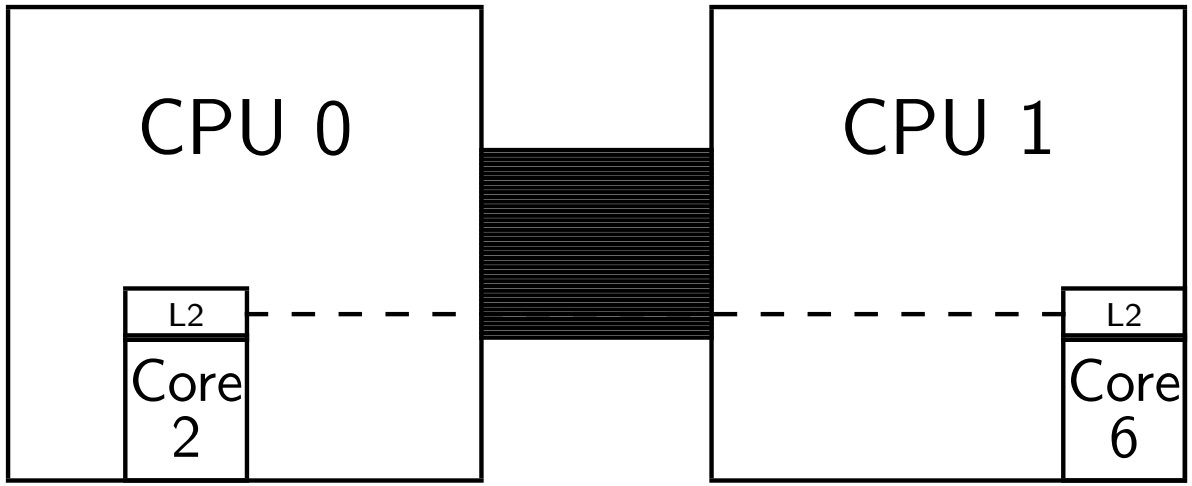
Write-back cache:

- modified line is only written to higher level (cache or main memory) when it gets displaced from the cache
- much less memory traffic
- main memory may not have latest values – potential problem for parallel computing

Intel uses write-back caches at all levels.

Multithreaded execution

New problem due to write-back caches: cache coherency



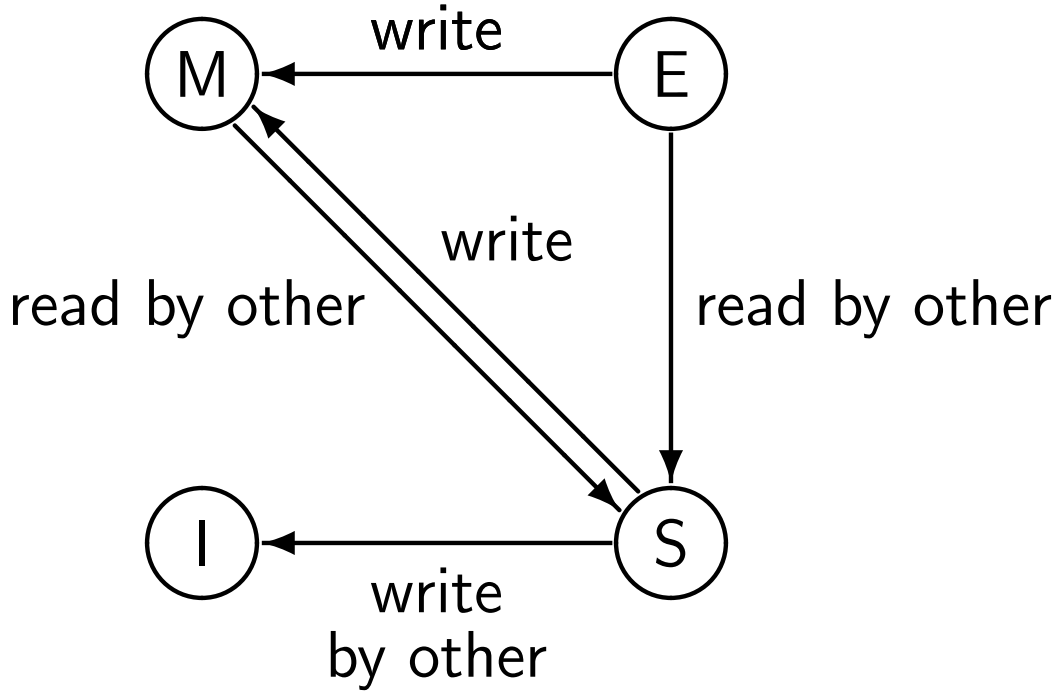
Suppose a thread on Core 2 of CPU 0 loads and modifies variable X in its level 2 cache, and then a thread on Core 6 of CPU 1 loads X?

There is a special link (Snoop Filter) between all of the caches so that the Core 2/CPU 0 cache controller spots the request and responds instead of the main memory. There are challenges with maintaining this cache coherency as core counts increase.

MESI cache coherency protocol

A cache line can be in one of 4 states:

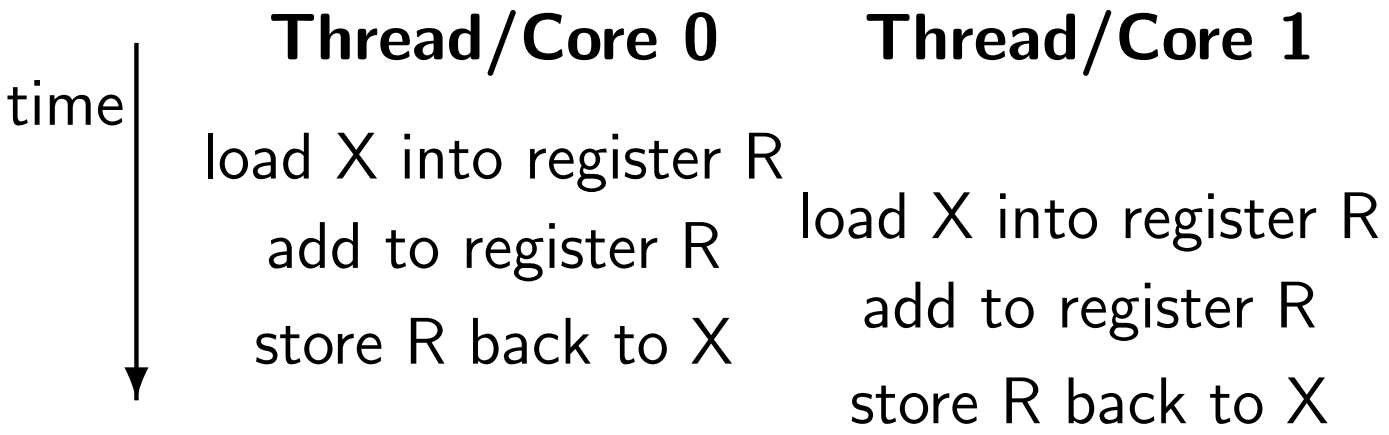
- Modified: sole owner of modified line
- Exclusive: sole owner, not modified
- Shared: shared ownership, not modified
- Invalid: incorrect data



MESI cache coherency protocol

This ensures that a read obtains the latest version of the data, **but** it doesn't solve the following problem.

Suppose Core 0 and Core 1 add to X at roughly the same time. We can get the following situation:



In the end, the contribution from Core 0 has been lost. It is the responsibility of the programmer to avoid this!

Fortunately, OpenMP will help a lot with this.

MESI cache coherency protocol

There's another very annoying problem, sometimes referred to as **false sharing**.

What happens if Core 0 repeatedly updates X, and Core 1 repeatedly updates Y?

It doesn't look like a problem, but if they are in the same cache line then the two cores will fight over ownership; each needs it (temporarily) to modify its variable.

This can lead to very poor performance, and again the programmer is responsible for avoiding this. However, in this case there is no help from OpenMP.

Why bother with parallel programming?

Suppose you have 72 cores, and 1 program to run – parallel programming will give you the answer in the shortest time

Now suppose you have 72 cores, and you have 72 programs to run.

You have two extreme choices

- run all 72 jobs at the same time, each one using 1 core
- run the 72 jobs sequentially, one after another, using 72 cores in parallel for each job

plus various options in between.

What should you do, and why?

Why bother with parallel programming?

A helpful experiment: compare time for 1 job to time for 72 jobs running at same time.

If the 72 jobs run in the same time, this is probably your best option.

But they probably won't, because they are sharing

- main memory
- L3 cache

The first of these may be the most significant; DDR5 memory is expensive, and there may not be enough to run 72 programs each with a lot of data.

The downside of parallelisation is the hassle of parallel programming, and the overheads of handling multiple threads.

Final comments

- the latest Intel Xeon server chips are very powerful
- to achieve the best performance, code has to be multithreaded (to use multiple cores) and vectorised (to use AVX512 units)
- we will see that OpenMP helps with both of these, but there are major pitfalls to be avoided on the data access side
- the danger is that performance is severely limited by data bandwidth in the cache hierarchy and to/from main memory
- see course webpage for links to further information