Session 2: Memory hierarchy

COMP52315: performance engineering

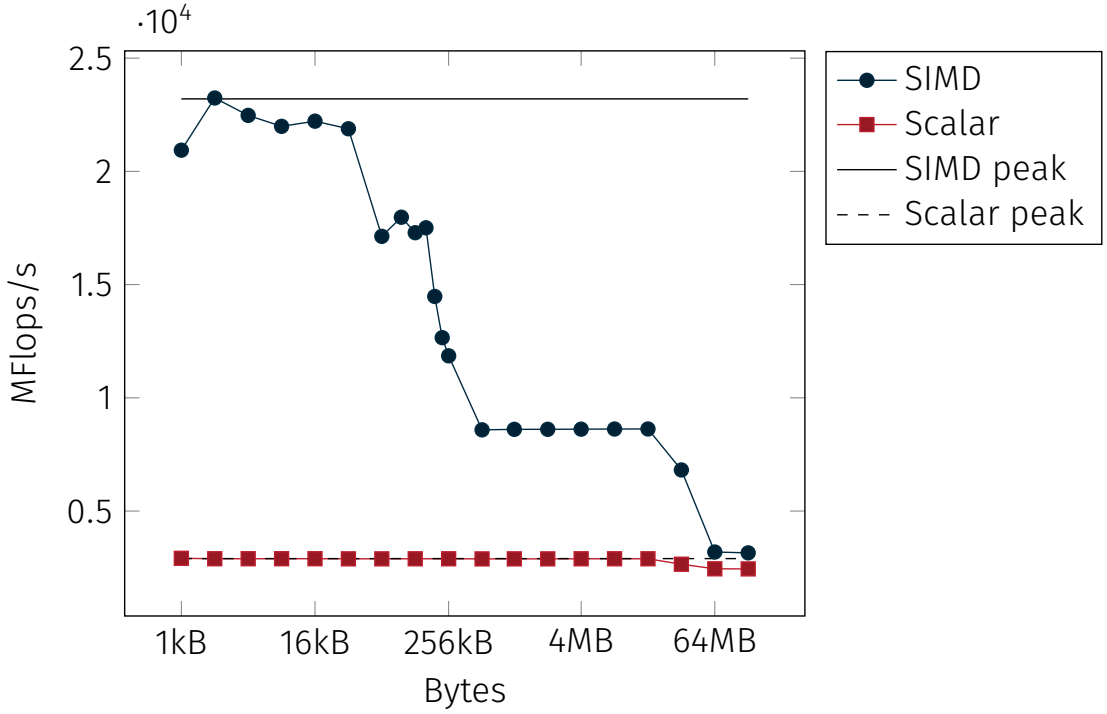
Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Sum reduction benchmark

In exercise 1, you hopefully produced a plot similar to this one.

Notice how the SIMD code has four distinct performance plateaus, whereas the scalar code only really has two.



Performance peak

- Broadwell chips can issue up to one ADD (scalar or vector) per cycle.
- Peak clock speed is 2.9GHz for this hardware.

Question

Why does the vectorised code not achieve theoretical peak for all vector sizes?

$2.9 \times 8 = 16 + 7.2 = 23.2$ Gflops/s.

Performance peak

- Broadwell chips can issue up to one ADD (scalar or vector) per cycle.
- Peak clock speed is 2.9GHz for this hardware.

Question

Why does the vectorised code not achieve theoretical peak for all vector sizes?

Lack of hardware resource

Recall that as well as worrying about instruction throughput, we have to think about data transfers.

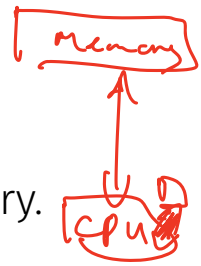
$\Rightarrow$ need to consider the memory hierarchy.

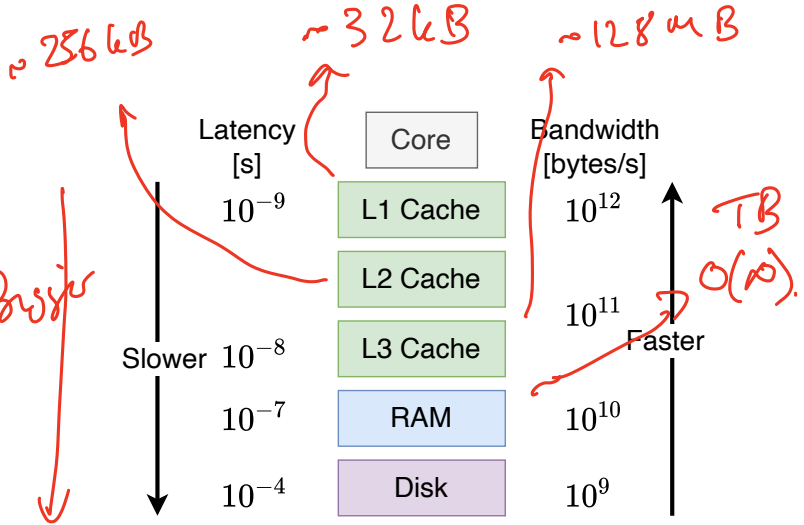
Memory hierarchy

Can either build *small* and *fast* memory
or

large and *slow* memory.

Not possible to have large and fast: physics gets in the way.

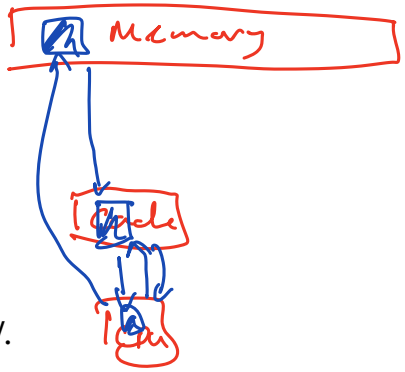




⇒ Purpose of many optimisations is to refactor algorithms to keep data in fast memory.

See `https://colin-scott.github.io/personal_website/research/interactive_latency.html` for more detail on latencies

Main idea for caches



- Add hierarchy of small, fast memory.
- Keeps a copy of *frequently used* data, speeding up access
- Typically not possible to *a priori* know which data will be needed frequently.

$\Rightarrow$ Caches rely on *principle of locality*

Principle of locality

- Normally impossible to decide before execution exactly which data will be needed “frequently”.
- In practice, most programs (could) exhibit *locality* of data access.
- Optimised algorithms will attempt to *exploit* this locality.

Temporal locality

If I access data at some memory address, it is likely that I will do so again “soon”.

Spatial locality

If I access data at some memory address, it is likely that I will access neighbouring addresses.

Temporal locality

- The first time we access an address, it is loaded from main memory *and* stored in the cache.
- We pay a (small) penalty for the first load (storing is not free).
- But subsequent accesses to that address use the copy in the cache, and are much faster.

$N >> 16$

Sum reduction

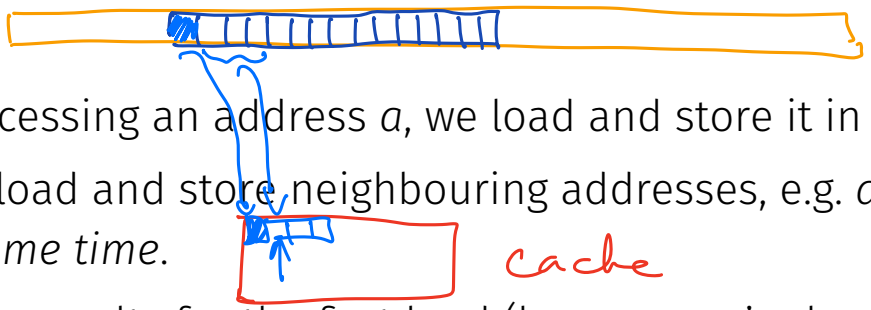
```
float s[16] = 0
for (i = 0; i < N; i++)
  s[i%16] += a[i];
```

$s[0] = \sum_{i=0}^{N/16} a[i*16]$

$s[1] = \sum_{i=0}^{N/16} a[i*16+1]$

Access to 16 entries of s exhibits temporal locality. Makes sense to keep all of s in cache.

Spatial locality



- When accessing an address a, we load and store it in the cache.
- We also load and store neighbouring addresses, e.g. $a + 1$, $a + 2$, $a + 3$ *at the same time*.
- We pay a penalty for the first load (because we're loading more data).
- Hope that next load is for $a + 1$, then access will be fast.

Sum reduction

```
float s[16] = 0
for (i = 0; i < N; i++)
  s[i%16] += a[i];
```

Access to a exhibits spatial locality. Makes sense when loading `a[i]` to also load `a[i+1]` (it will be used in the next iteration).

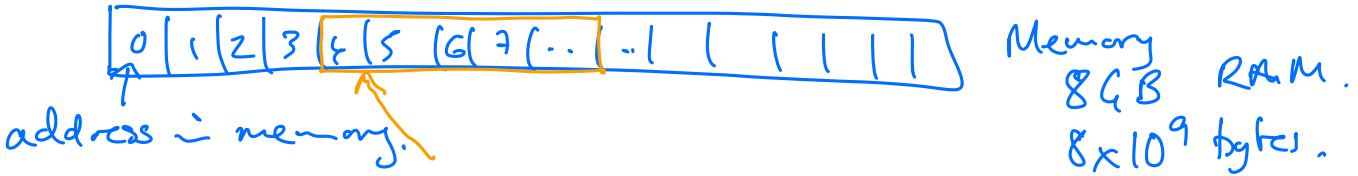
Designing a cache

Important questions

1. When we load data into the cache, where do we put it?
2. If we have an address, how do determine if it is in the cache?
3. What do we do when the cache becomes full?

- (1) & (2) are intimately related.

Putting data in a cache

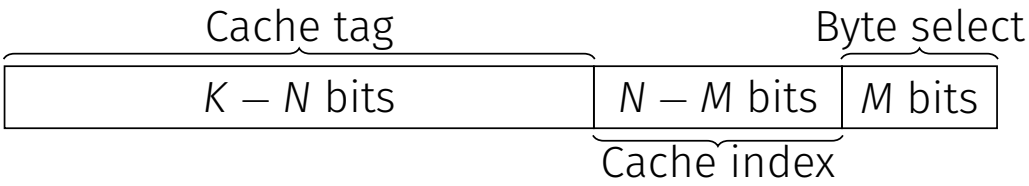


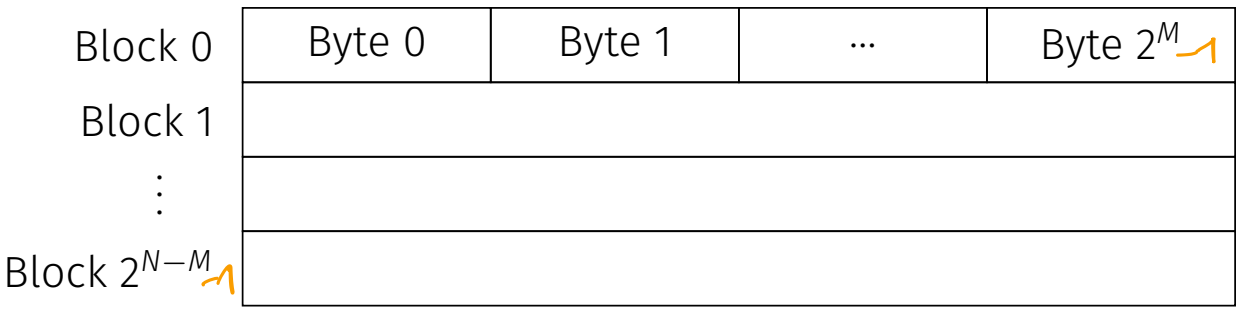
- Each datum uniquely referenced by its *address*, K bits. Usually $K = 32$ or $K = 64$.
- Need to turn this large address into a cache location.

Direct mapped caches

- Cache can store 2^N bytes.
- Divided into *blocks* each of 2^M bytes.
- Each address references one byte.
- Use N bits of the address, to select which slot in the cache to use.

Direct mapped caches: indexing





- **Byte select**: Use lowest M bits to select correct byte in block.
- **Cache index**: Use next $N - M$ bits to select correct block.
- **Cache tag**: Use remaining $K - N$ bits as a key.

Choice of block size

- Data is loaded one *block* at a time (also called *cache lines*).
- Immediately exploits *spatial locality*.
- Larger blocks are not always better.
- Almost all modern CPUs use 64byte block size.

Corollary

Cache-friendly algorithms work on cache line sized chunks of data.

Direct mapped caches: eviction

- What happens if two addresses have the same low bit pattern?
- We have a *conflict*.
- Resolution: newest loaded address wins.
- This is a *least recently used* (LRU) eviction policy.

What can go wrong?

64x4 bytes. 64 cache lines. registers.

```
int a[64], b[64], r = 0;
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 64; j++)
        r += a[j] + b[j];
```

Total size of a & b is 512 bytes. So they should all fit in cache.

- 1KB cache
- 32 byte block size
- So $N = 10$, $M = 5$.
  32 blocks in the cache.

Conflicts reduce *effective* cache size

```
for (int j = 0; j < 64; j++)
    r += a[j] + b[j];
```

block 0

block byte

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

```
for (int j = 0; j < 64; j++)
   r += a[j] + b[j];
```

$\int = 32$

$a_{0:7}$	$a_{8:15}$	$a_{16:23}$	$a_{24:31}$
$b_{0:7}$	$b_{8:15}$	$b_{16:23}$	$b_{24:31}$

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

```
for (int j = 0; j < 64; j++)
   r += a[j] + b[j];
```

$b_{32:39}$	$a_{8:15}$	$a_{16:23}$	$a_{24:31}$
$a_{32:39}$			
$b_{0:7}$	$b_{8:15}$	$b_{16:23}$	$b_{24:31}$

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

```
for (int j = 0; j < 64; j++)
   r += a[j] + b[j];
```

$b_{32:39}$	$b_{40:47}$	$a_{16:23}$	$a_{24:31}$
$a_{32:39}$	$a_{40:47}$		
$b_{0:7}$	$b_{8:15}$	$b_{16:23}$	$b_{24:31}$

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

```
for (int j = 0; j < 64; j++)
   r += a[j] + b[j];
```

$b_{32:39}$	$b_{40:47}$	$b_{48:55}$	$a_{24:31}$
$a_{32:39}$	$a_{40:47}$	$a_{48:55}$	
$b_{0:7}$	$b_{8:15}$	$b_{16:23}$	$b_{24:31}$

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

Now when we go round again a[0:31] are not in the cache.

```
for (int j = 0; j < 64; j++)
   r += a[j] + b[j];
```

$b_{32:39}$	$b_{40:47}$	$b_{48:55}$	$b_{56:63}$
$a_{32:39}$	$a_{40:47}$	$a_{48:55}$	$a_{56:63}$
$b_{0:7}$	$b_{8:15}$	$b_{16:23}$	$b_{24:31}$

```
&a[00] = b..._00000_00000 => line 0, byte offset 0
&a[01] = b..._00000_00100 => line 0, byte offset 4
&a[02] = b..._00000_01000 => line 0, byte offset 8
&a[03] = b..._00000_01100 => line 0, byte offset 12
&a[04] = b..._00000_10000 => line 0, byte offset 16
&a[05] = b..._00000_10100 => line 0, byte offset 20
&a[06] = b..._00000_11000 => line 0, byte offset 24
&a[07] = b..._00000_11100 => line 0, byte offset 28
...
&b[00] = b..._11100_00000 => line 28, byte offset 0
&b[01] = b..._11100_00100 => line 28, byte offset 4
&b[02] = b..._11100_01000 => line 28, byte offset 8
&b[03] = b..._11100_01100 => line 28, byte offset 12
&b[04] = b..._11100_10000 => line 28, byte offset 16
&b[05] = b..._11100_10100 => line 28, byte offset 20
&b[06] = b..._11100_11000 => line 28, byte offset 24
&b[07] = b..._11100_11100 => line 28, byte offset 28
...
```

Cache thrashing

What can go wrong?

```
int A[64], B[64], r = 0;
for (int i = 0; i < 100; i++)
   for (int j = 0; j < 64; j++)
       r += A[j] + B[j];
```

- 1KB cache
- 32 byte block size
- So $N = 10$, $M = 5$.
  32 blocks in the cache.

Cache thrashing

- We need $2 \cdot 64 \cdot 4 = 512$ bytes to store A and B in cache. This only requires 16 blocks, so our cache is large enough.
- But if the addresses match in the low bits, we will try and store to same locations.
- In worst case, every load of `B[j]` evicts `A[j]`, and vice versa.

Cache associativity

Direct mapped

- Each block from main memory maps to exactly one cache line.
- LRU eviction policy (new data overwrite old).

Fully associative

- Each byte from main memory can maps to any cache line.
- Most flexible, but also expensive.

k-way set associative

- k "copies" of a direct mapped cache. Each block from main memory maps to one of k cache lines, called *sets*.
- Typically use LRU eviction.
- Usual choice: $N \in \{2, 4, 8, 16\}$.
- Skylake has $N = 8$ for L1, $N = 16$ for L2, $N = 11$ for L3.

Most common.

Exercise: cache bandwidth

- Let's try and do this in the round again.
- Goal is to benchmark the memory bandwidth as a function of vector size to see what we observe.
- We will use the results to explain the observations of the sum reduction benchmark.

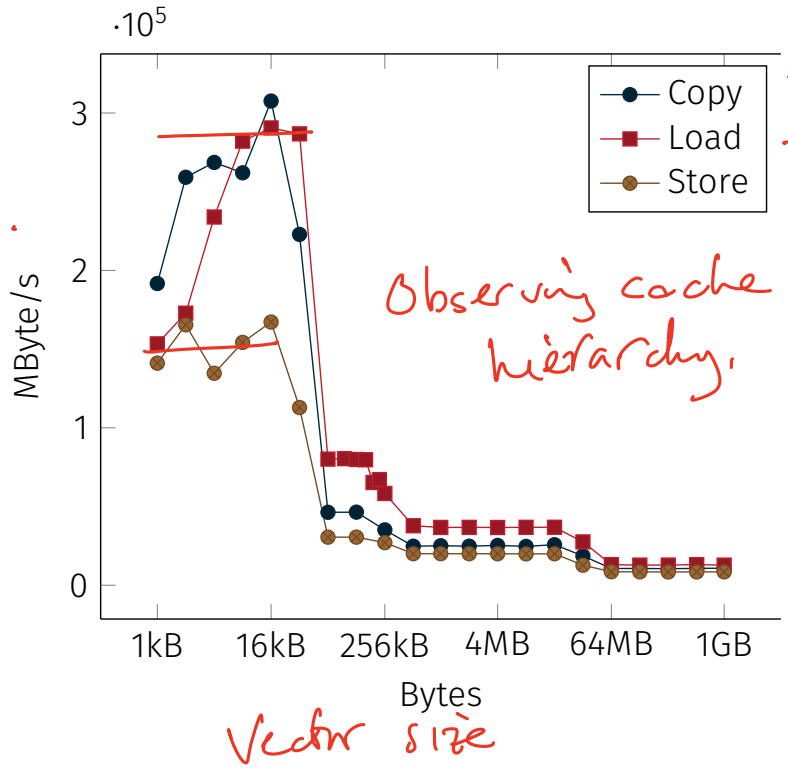
$\Rightarrow$ over to you.

`teaching.wence.uk/comp52315/exercises/exercise02/`

Results

You hopefully produced a plot similar to this one.

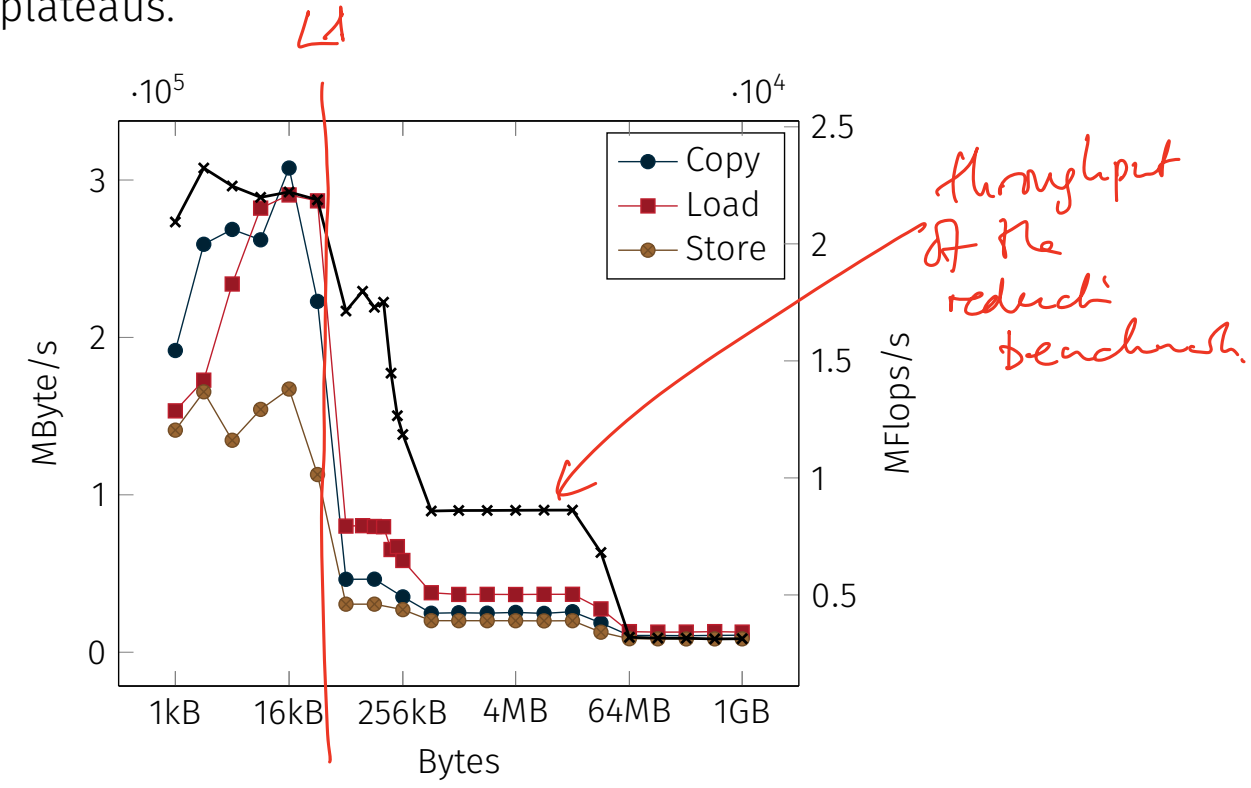
I added the floating point throughtput of the sum reduction so we can compare the plateaus.



Results

You hopefully produced a plot similar to this one.

I added the floating point throughtput of the sum reduction so we can compare the plateaus.



Interpretation

1 ADD per cycle. → Needs 8 floats/cycle.

- Vectorised addition requires 1 32Byte load/cycle (for the 8 floats)
- Accumulation parameter held in a register.

each float is 4 bytes. clock frequency 2.9 GHz.

⇒ requires sustained load bandwidth of $32 \times 2.9 = 92.8$GByte/s

- From L1 (less than 32kB) we see sustained bandwidth of around 300GByte/s ⇒ floating-point throughput is limit.

throughput limited by 1 ADD/cycle.

- L2 (less than 256kB) provides around 80GByte/s or around 27Bytes/cycle ⇒ 6.75 floats/cycle ⇒ peak is around 19GFlops/s.

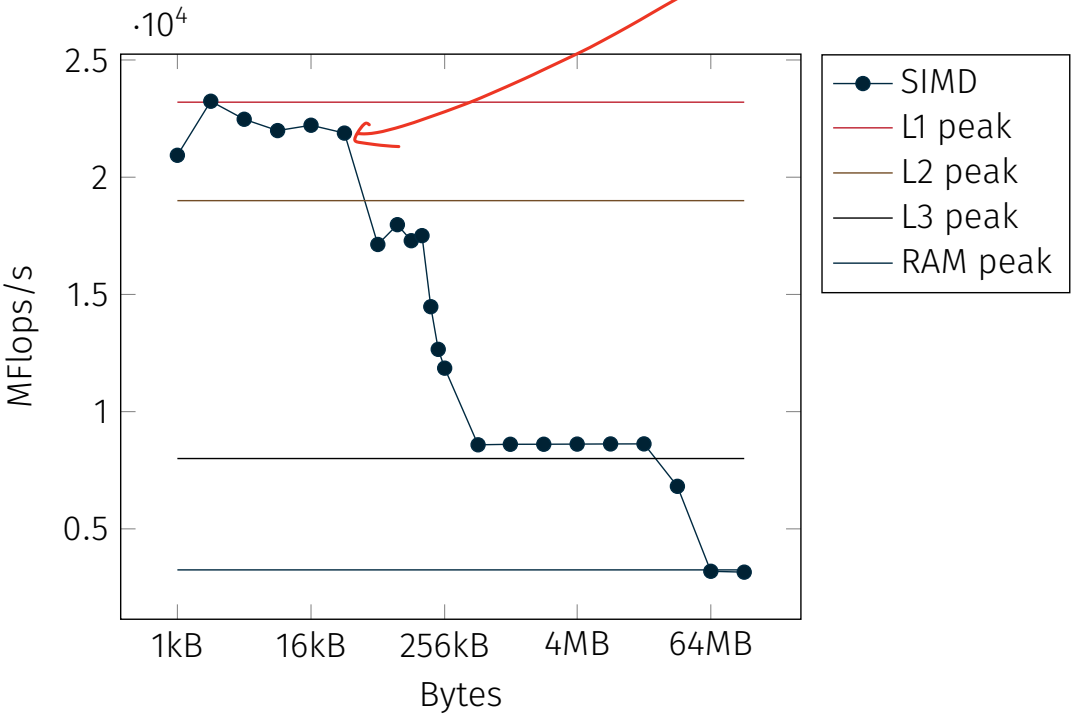
$6.75 \times 2.9 \rightarrow \sim 19$

- L3 (less than 30MB) provides around 36GByte/s or around 12Bytes/cycle ⇒ 2.75 floats/cycle ⇒ peak is around 8GFlops/s.
- Main memory provides around 13GByte/s or around 4.5Bytes/cycle ⇒ 1.1floats/cycle ⇒ peak is around 3.25GFlops/s.

2 limits: speed of ADD (1/cycle)
or how fast we can get data to CPU
→ dependent on size of vector.

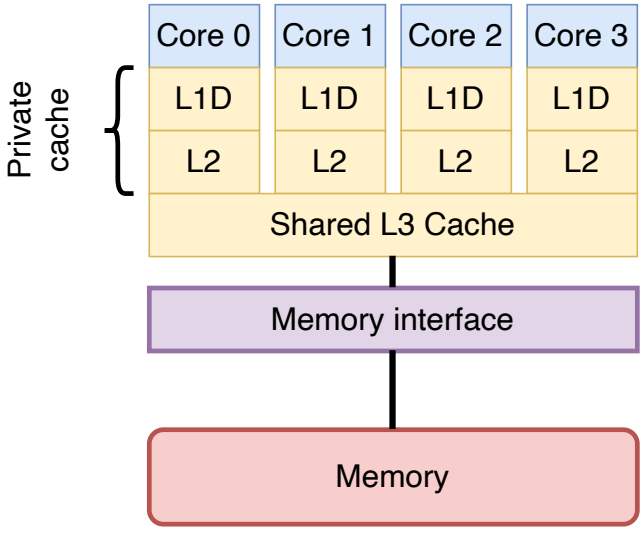
Adding bandwidth-induced limits

Not bad for a pen-and-paper exercise.



Memory/node topology

`likwid-topology` reports an ASCII version of diagrams like this.



More than one core

- So far, just looked at performance when we use a single core.
- In practice, most scientific computing algorithms will be parallel

$\Rightarrow$ How does this affect the performance?

Scalable vs. Saturating

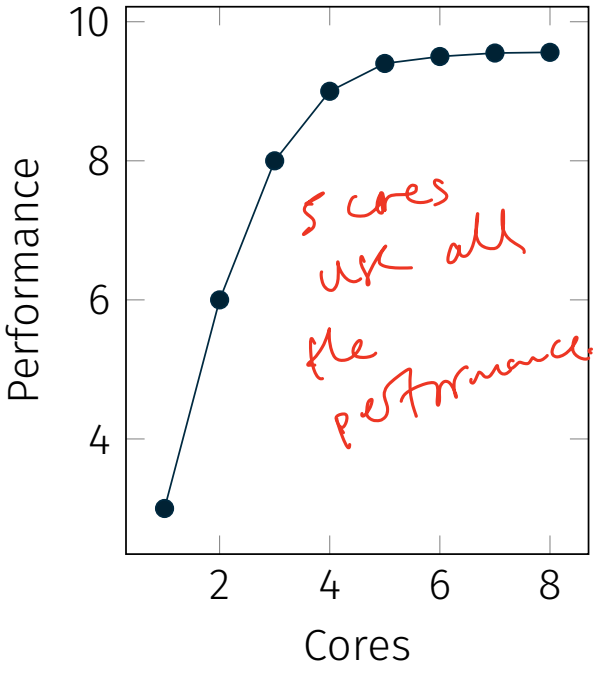
CPU cores are a *scalable* resource.

Adding a second core doubles the number of floating point operations we can perform.

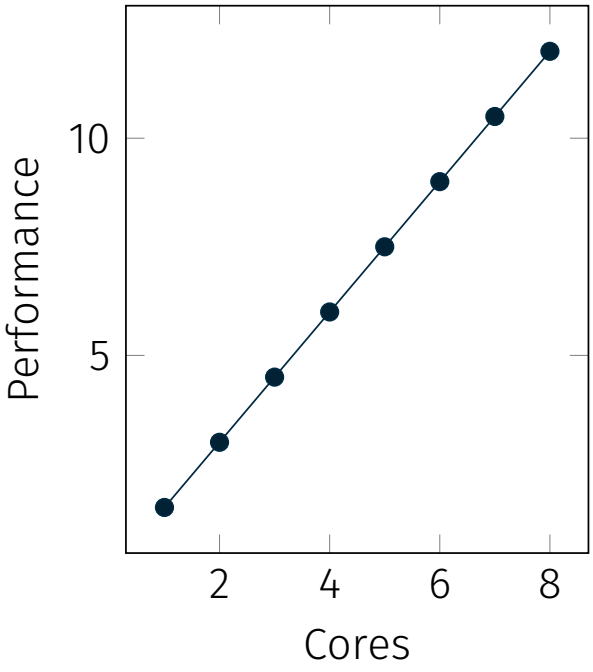
Memory bandwidth is a *saturating resource*. Outside of L2 cache (L3, main memory), CPU cores compete for the same resource.

Scalable vs. Saturating

Shared resources might show saturating performance



Parallel resources show scalable performance



Exercise: memory bandwidth saturation

- Goal is to benchmark the memory bandwidth for different vector sizes as a function of number of cores
- Will then look at scaling of sum reduction with cores

$\Rightarrow$ over to you.

Exercises at

`teaching.wence.uk/comp52315/exercises/exercise03/`

Results in next slides

→ sanity up soon!

Conclusions on hardware architecture

Performance considerations

- How many instructions are required to implement an algorithm
- How efficiently those instructions are executed on a processor
- The runtime contribution of the data transfers

Complex "topology" of hardware

- Many layers of parallelism in modern hardware
- Sockets: around 1-4 CPUs on a typical motherboard
- Cores: around 4-32 cores in a typical CPU
- SIMD/Vectorisation: typically 2-16 single precision elements in vector registers on CPUs
- Superscalar execution: typically 2-8 instructions per cycle

⟹ Getting the most out of hardware is difficult.

Challenges for program development

- We will focus most of our efforts on SIMD and some superscalar execution here.
- An ongoing challenge is that most programming models do not offer a lot of explicit access to parallelism.

$\Rightarrow$ will look at mechanisms to convince compilers to “do the right thing”.