SESSION 2

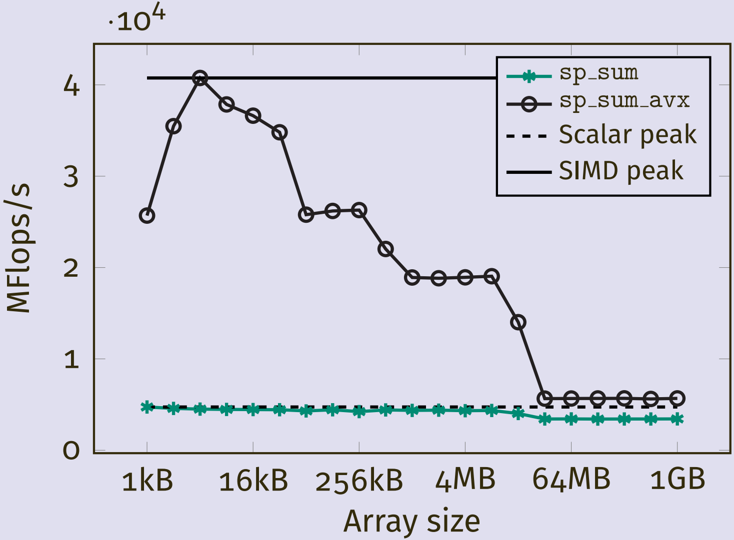
MEMORY HIERARCHY

Massimiliano Fasi

Durham University

Sum reduction benchmark (Exercise 1)

- SIMD: 4 plateaus
- scalar: 3 plateaus



Performance peak

Variability

This is due to CPU Boosting.

Performance peak

Variability

This is due to CPU Boosting.

Question

SIMD code does not achieve theoretical peak for all sizes. Why?

Performance peak

Variability

This is due to CPU Boosting.

Question

SIMD code does not achieve theoretical peak for all sizes. Why?

Hardware bottlenecks

Performance peak

Variability

This is due to CPU Boosting.

Question

SIMD code does not achieve theoretical peak for all sizes. Why?

Hardware bottlenecks

- Cannot be instruction throughput.
- Memory bandwidth decreases with vector size

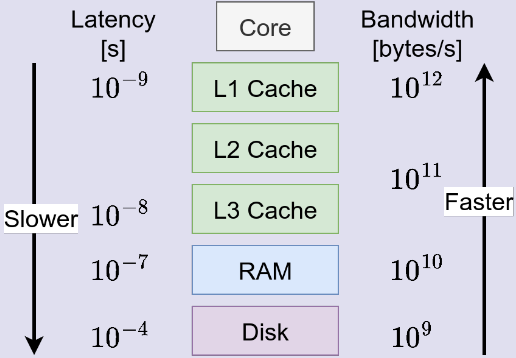
Memory hierarchy

Two types of memory:

- *small* and *fast*
- *large* and *slow*

Large and fast is impossible:

$\Rightarrow$ physics gets in the way.



Optimisation: refactor algorithms to keep data in fast memory.

Check Colin Scott's page for more detail on latencies.

Cache memory: overview

Features

- Hierarchy of small, fast memory.
- Keep a copy of *frequently used* data for faster access

Cache memory: overview

Features

- Hierarchy of small, fast memory.
- Keep a copy of *frequently used* data for faster access

Issues

- Frequently accessed data not known *a priori*
- Only heuristics are possible $\Rightarrow$ *princple of locality*

Principle of locality

- Frequently accessed data often unknown before execution
- In practice, most programs exhibit *locality* of data access.
- Optimised algorithms attempt to *exploit* this locality.

Principle of locality

- Frequently accessed data often unknown before execution
- In practice, most programs exhibit *locality* of data access.
- Optimised algorithms 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

On **first access** to a new address, the data is:

- loaded from main memory to registers
- stored in cache

Temporal locality

On **first access** to a new address, the data is:

- loaded from main memory to registers
- stored in cache

Trade-off solution:

- Small performance penalty for first access (storing is not free)
- Subsequent accesses use cached copy and are much faster.

Spatial locality

On **first access** to a new address, the data is:

- loaded from main memory to registers
- stored in cache
- neighbouring addressed are also stored in cache

Spatial locality

On **first access** to a new address, the data is:

- loaded from main memory to registers
- stored in cache
- neighbouring addressed are also stored in cache

Trade-off solution:

- Large performance penalty for first access
- Subsequent accesses to neighbouring data will be fast

Example: sum reduction

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

- Temporal locality
  - 16 entries of `s` are accessed repeatedly
  - Makes to keep all of `s` in cache
- Spatial locality
  - Contiguous entries of `a` are accessed
  - When loading `a[i]` it makes sense to load `a[i+1]` too.

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?

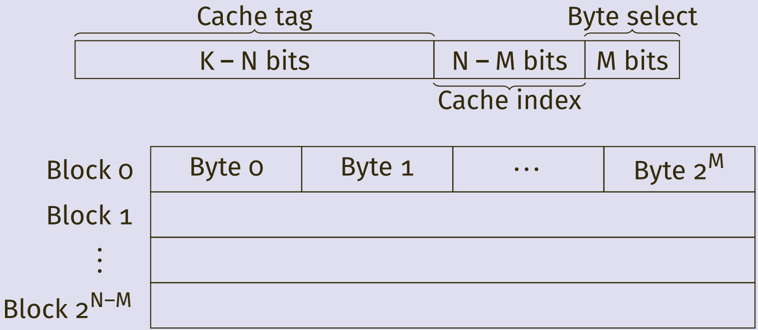
- Each datum uniquely referenced by its K-bit *address*
- Need to turn this large memory address into a cache location
- K is typically large ($2^{32}/2^{64}$ addresses)

Direct mapped cache

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

Simplest solution: injection from RAM to cache

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 cache line size

- Data is loaded one *cache line* at a time
- Immediately exploits *spatial locality*
- Larger cache lines are not always better
- Almost all modern CPUs use 64-byte size

Rule of thumb

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

Direct mapped caches: eviction

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

Direct mapped caches: eviction

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

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

Conflicts reduce *effective* cache size

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


```
&a[00] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

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

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

```
&a[00] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 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] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

```c
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] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 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] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 28, byte offset 28
...
```

Conflicts reduce *effective* cache size

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

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

```
&a[00] = ... 00000 00000 => block  0, byte offset 0
&a[01] = ... 00000 00100 => block  0, byte offset 4
&a[02] = ... 00000 01000 => block  0, byte offset 8
&a[03] = ... 00000 01100 => block  0, byte offset 12
&a[04] = ... 00000 10000 => block  0, byte offset 16
&a[05] = ... 00000 10100 => block  0, byte offset 20
&a[06] = ... 00000 11000 => block  0, byte offset 24
&a[07] = ... 00000 11100 => block  0, byte offset 28
...
&b[00] = ... 11100 00000 => block 28, byte offset 0
&b[01] = ... 11100 00100 => block 28, byte offset 4
&b[02] = ... 11100 01000 => block 28, byte offset 8
&b[03] = ... 11100 01100 => block 28, byte offset 12
&b[04] = ... 11100 10000 => block 28, byte offset 16
&b[05] = ... 11100 10100 => block 28, byte offset 20
&b[06] = ... 11100 11000 => block 28, byte offset 24
&b[07] = ... 11100 11100 => block 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.

- 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.
- If low bits of addresses match, same cache lines are mapped.
- In the worst case, every load of `B[j]` evicts `A[j]`, and vice versa.

Cache associativity

- Direct mapped
  - Each RAM *block* maps to exactly one cache line.
  - LRU eviction policy (new data overwrite old)

Cache associativity

- Direct mapped
  - Each RAM *block* maps to exactly one cache line.
  - LRU eviction policy (new data overwrite old)
- Fully associative
  - Each RAM *byte* can map to any cache line
  - Data is stored in first unused cache line
  - If all lines are used, overall LRU one is replaced
  - Most flexible, but also mostexpensive

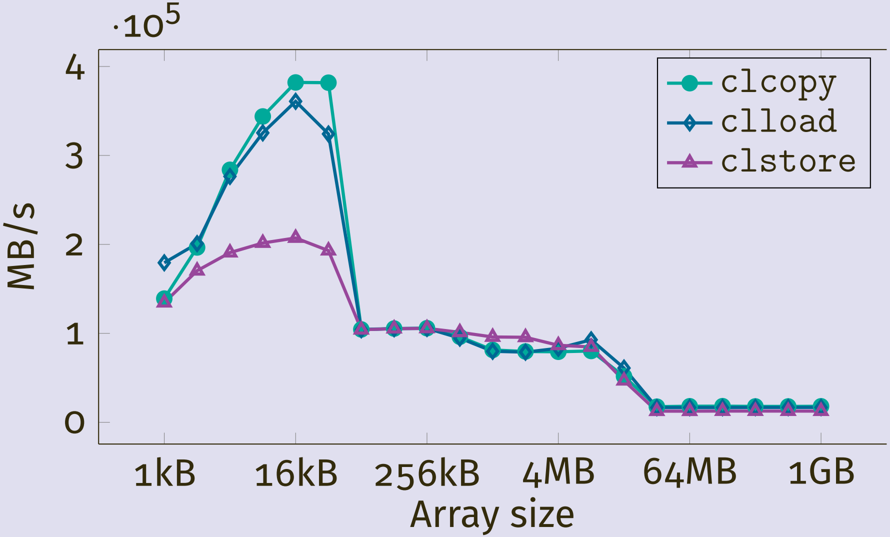
k-way set associative cache

- k “copies” of a direct mapped cache.
- Each block from main memory maps to 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.

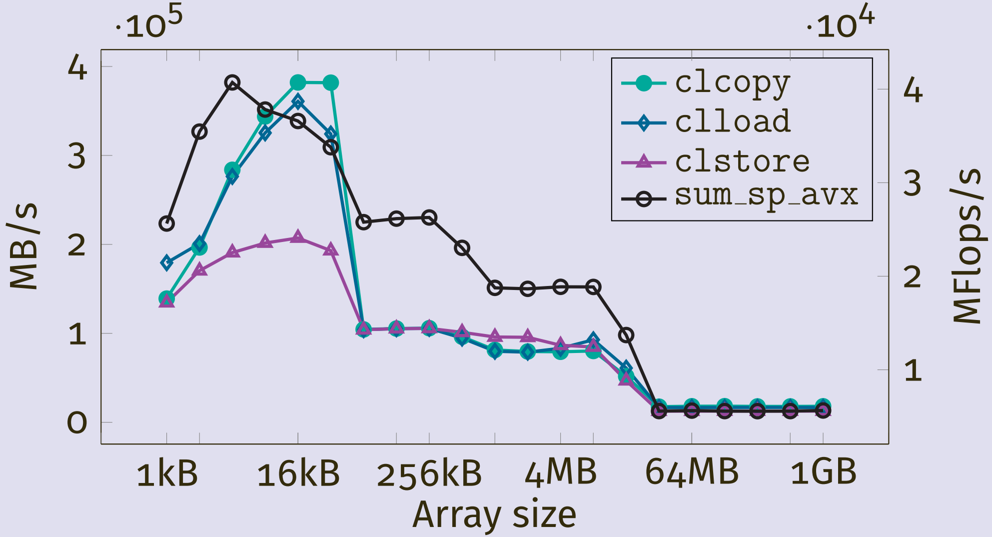
Exercises 2/3: memory bandwidth/saturation

1. Split into small groups
2. Make sure one person per group has access to Hamilton
3. Benchmark memory bandwidth as a function of vector size
4. You can use the bash script from last week.
5. Ask questions!

Exercise 2: results



Exercise 2: results



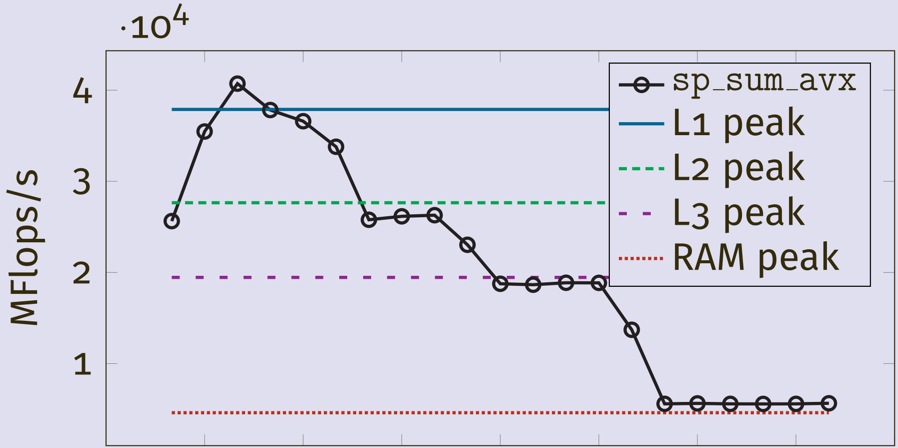
Interpretation

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

$\Rightarrow$ requires sustained load bandwidth of $4 \cdot 35 = 148$GB/s

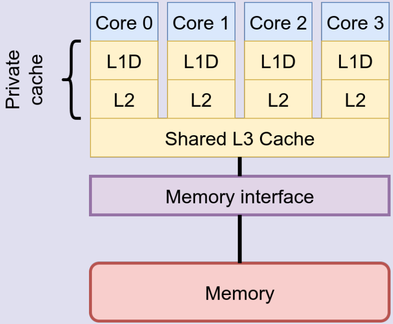
- From L1 (less than 32kB) we see sustained bandwidth of around 370GB/s or 90B/Flop $\Rightarrow$ 22 float/Flop $\Rightarrow$ floating-point throughput is limit.
- L2 (less than 512kB) provides around 100GB/s or around 25B/Flop $\Rightarrow$ 6.25 floats/Flop $\Rightarrow$ peak is around 27GFlop/s.
- L3 (less than 16MB) provides around 78GB/s or around 18B/Flop $\Rightarrow$ 4.45 floats/cycle $\Rightarrow$ peak is around 19GFlop/s.
- Main memory provides around 17.5GB/s or around 4B/Flop $\Rightarrow$ 1float/cycle $\Rightarrow$ peak is around 4.5GFlop/s.

AVX throughput with bandwidth-induced limits

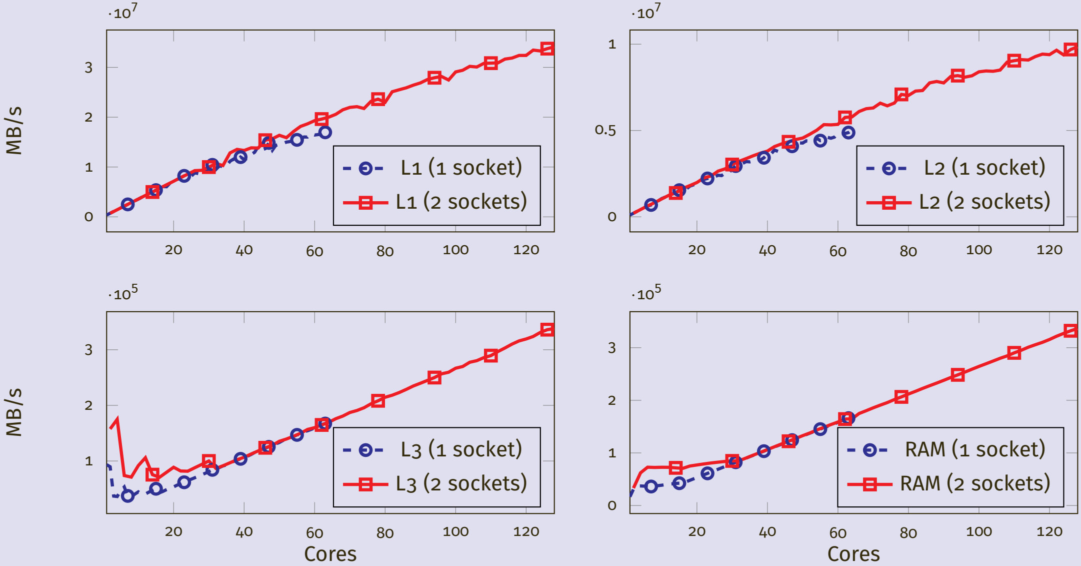


Memory/node topology

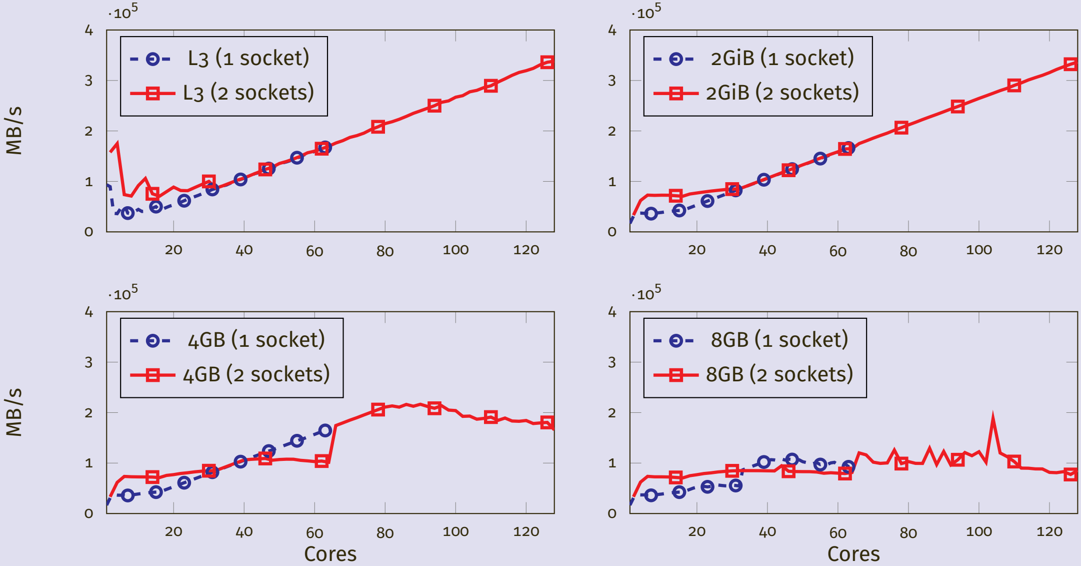
`likwid-topology` reports an ASCII version of diagrams like this.



Exercise 3: results



Exercise 3: results (updated)



Conclusions on hardware architecture

Performance considerations

- How many instructions are required
- How efficiently a processor can execute those instructions
- The runtime contribution of the data transfers

Conclusions on hardware architecture

Performance considerations

- How many instructions are required
- How efficiently a processor can execute those instructions
- 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
- Vectorisation: 2-16 `float`s per vector registers
- Superscalar execution: typically 2-8 instructions per cycle