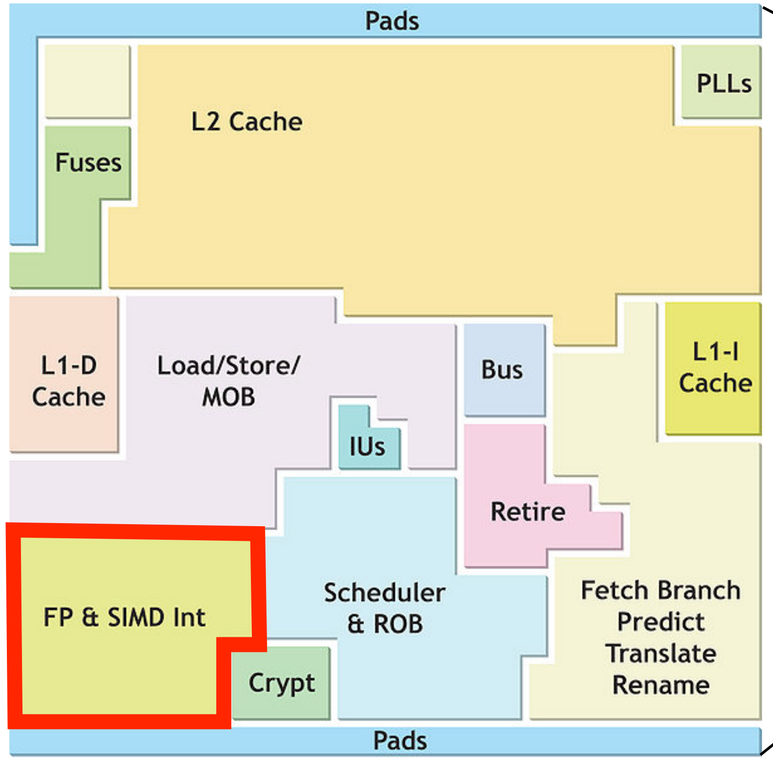
Lecture 3: Single-processor Computing Summary

CMSE 822: Parallel Computing

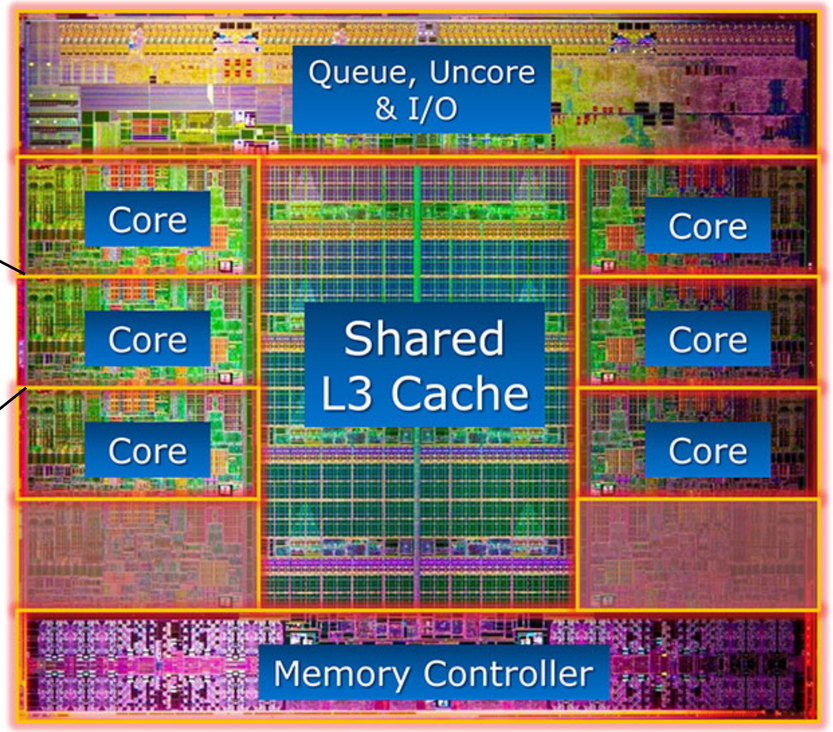
Prof. Sean M. Couch

Anatomy of a Computation

A CPU



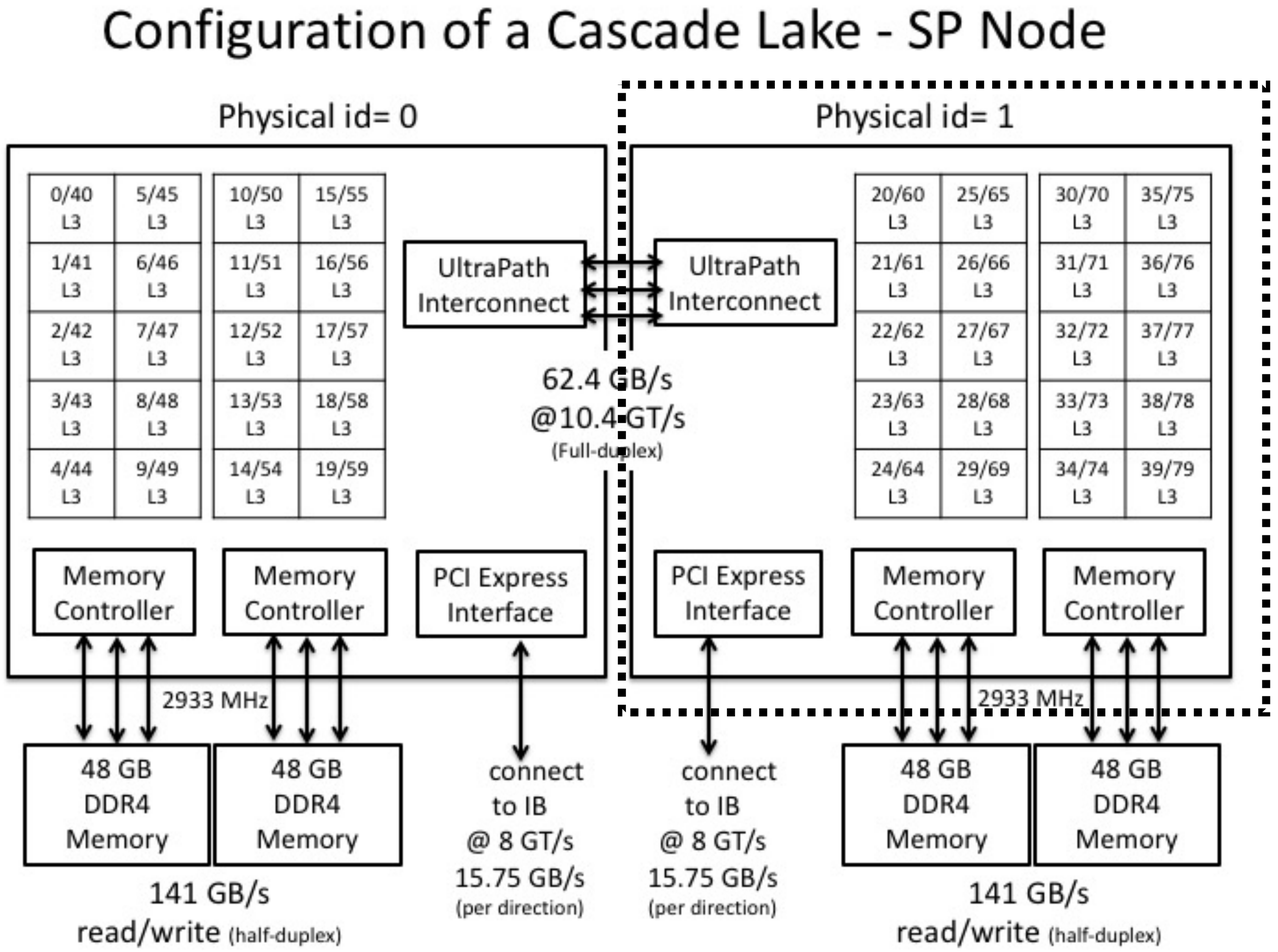
Intel® Core™ i7-3960X Processor Die Detail



Single-CPU computing is parallel!

Anatomy of a Computation

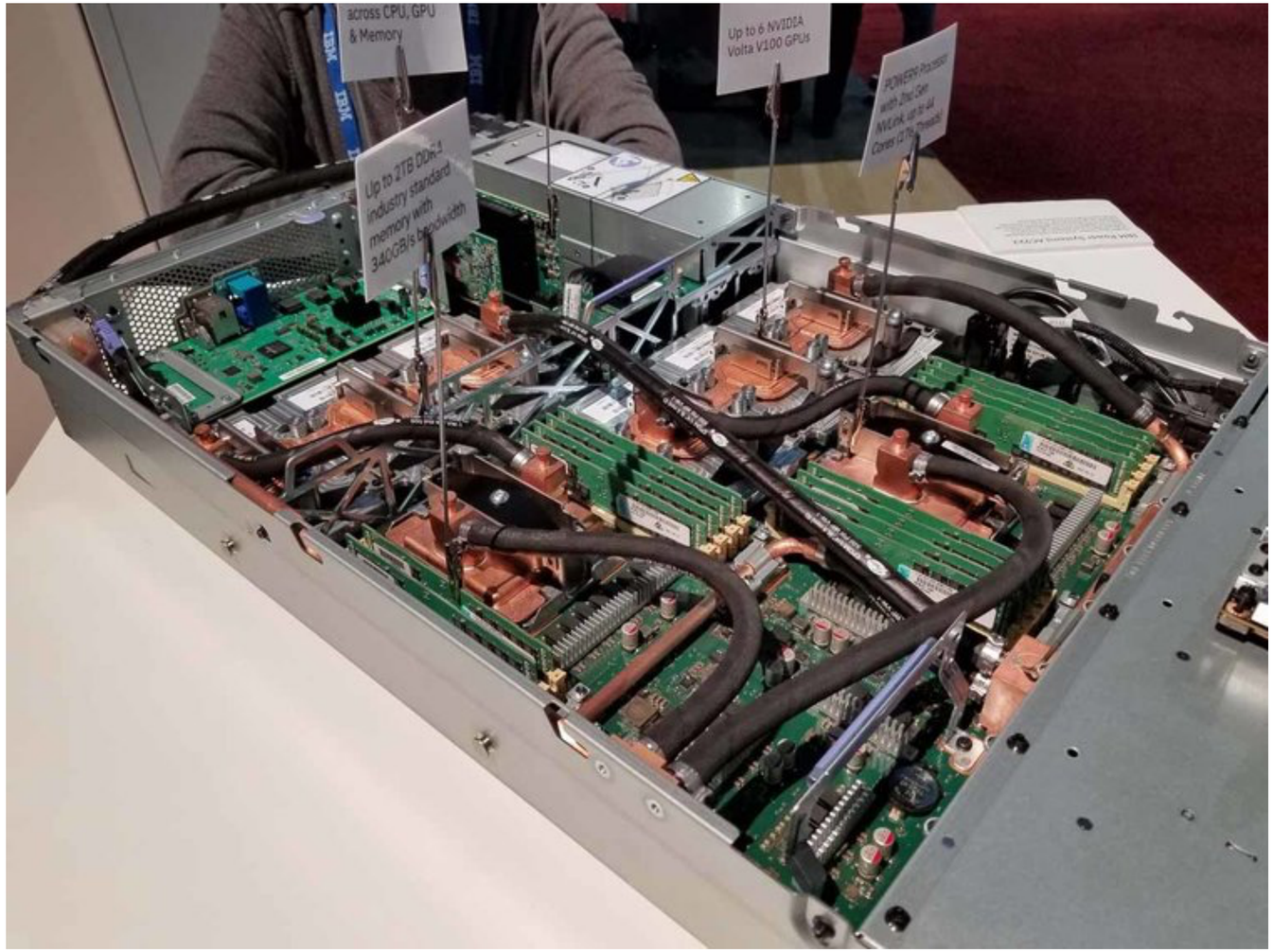
A Node



"socket"

Anatomy of a Computation

A Node



Summit, ORNL

Anatomy of a Computation

A Node

Summit, ORNL

Anatomy of a Computation

Node-to-node Interconnect

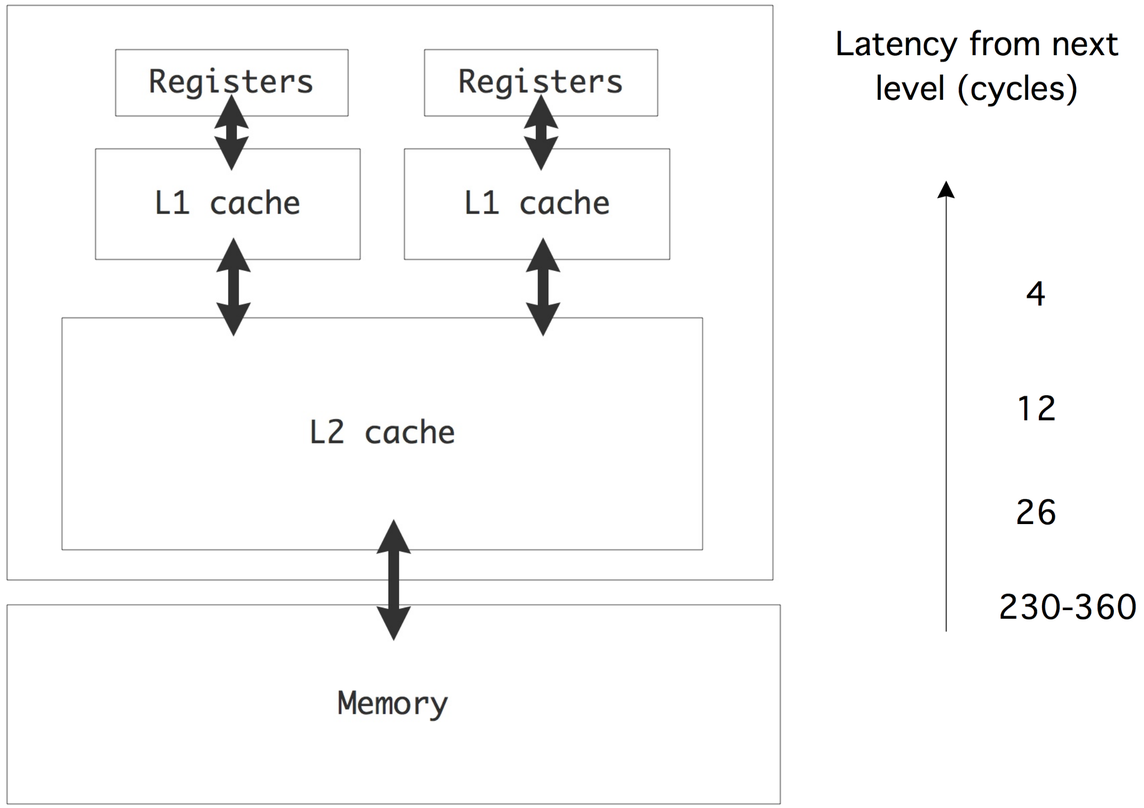
Anatomy of a Computation

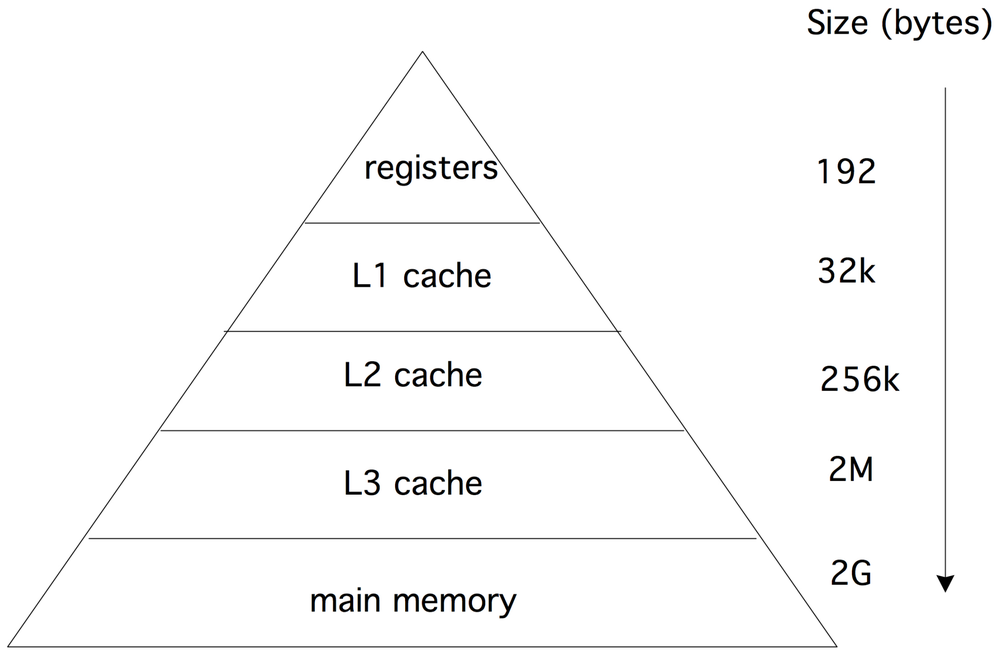
Cluster

Summit, ORNL

Memory hierarchy

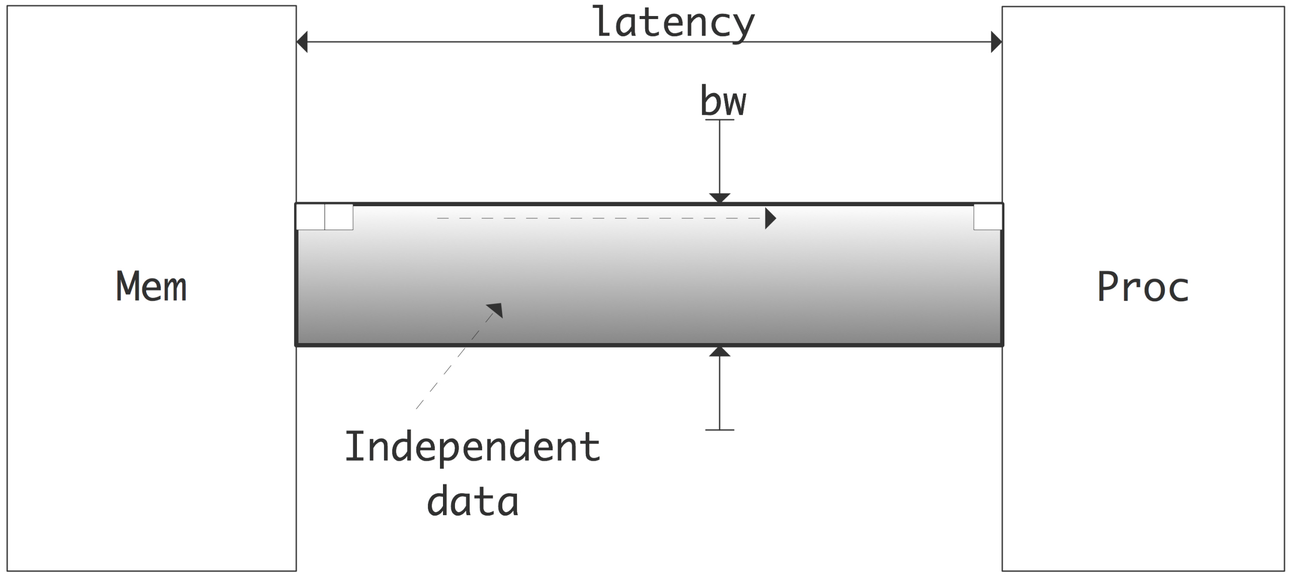
Often the limiter of performance…





Memory hierarchy

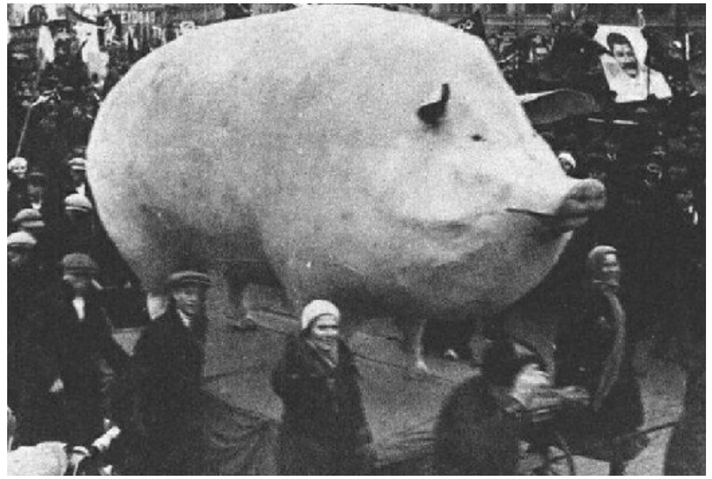
Need to feed the beast (er, CPU)



Little's Law: Concurrency = Bandwidth x Latency

Memory hierarchy

Absolute unit



What is the fundamental unit of memory movement?

a. page

b. word

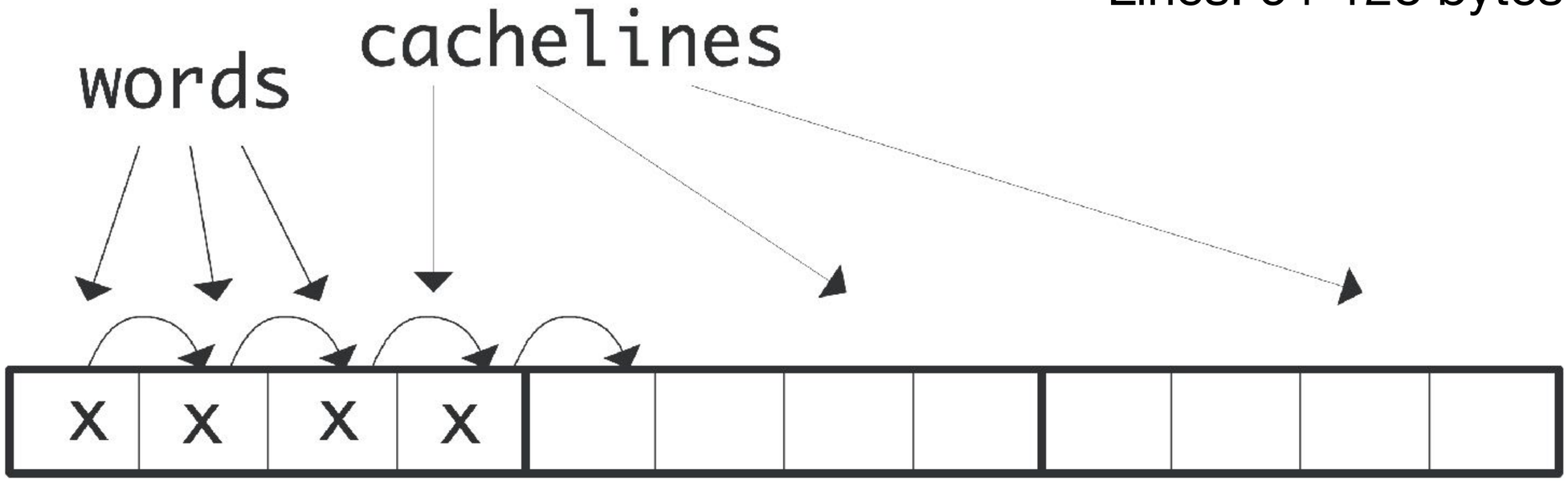
c. line

d. byte

Memory hierarchy

Cache line

Lines: 64-128 bytes

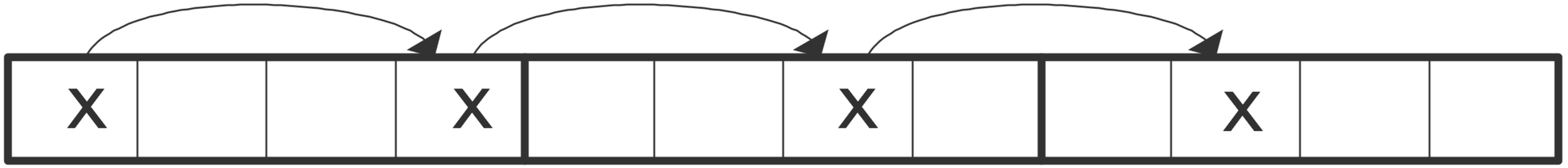


Words: usually 64 bits

Memory hierarchy

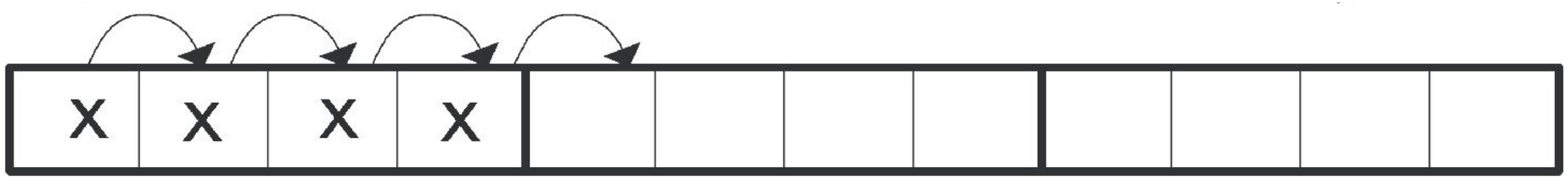
Strided access

Stride stride




```
for (i=0; i<N; i+=stride)
  ... = ... x[i] ...
```

Stride 1




```
for (i=0; i<N; i++)
  ... = ... x[i] ...
```

Memory hierarchy

Reuse is key to performance!

- Compulsory cache miss: first time memory is referenced
- Capacity cache miss: cache not big enough to fit problem
- Conflict cache miss: data mapped to same cache location as another
- Invalidation cache miss: another core changed value at memory address

Memory hierarchy
False sharing

```
  local_results = new double[num_threads];
#pragma omp parallel
{
  int thread_num = omp_get_thread_num();
  for (int i=my_lo; i<my_hi; i++)
    local_results[thread_num] = ... f(i) ...
}
global_result = g(local_results)
```

- Cores access and alter data in same *cache line*

Exercise 1.14

Matrix-matrix Multiply

Exercise 1.14. The matrix-matrix product, considered *as operation*, clearly has data reuse by the above definition. Argue that this reuse is not trivially attained by a simple implementation. What determines whether the naive implementation has reuse of data that is in cache?

Caches can only hold a finite amount of data. Once a row of A and a column of B take up more than the size of the cache, their elements will be flushed between iterations of the outer loop.

In-class Group Exercise

Exercise 1.16 - Spatial and temporal locality

Consider the following pseudocode of an algorithm for summing n numbers $x[i]$ where n is a power of 2:

```
for s=2,4,8,...,n/2,n:
  for i=0 to n-1 with steps s:
    x[i] = x[i] + x[i+s/2]
sum = x[0]
```

Analyze the spatial and temporal locality of this algorithm, and contrast it with the standard algorithm

```
sum = 0
for i=0,1,2,...,n-1
  sum = sum + x[i]
```

In-class Group Exercise

Exercise 1.17 - Reuse

Exercise 1.17. Consider the following code, and assume that `nvectors` is small compared to the cache size, and `length` large.

```
for (k=0; k<nvectors; k++)
  for (i=0; i<length; i++)
    a[k,i] = b[i] * c[k]
```

How do the following concepts relate to the performance of this code:

- Reuse
- Cache size
- Associativity

Would the following code where the loops are exchanged perform better or worse, and why?

```
for (i=0; i<length; i++)
  for (k=0; k<nvectors; k++)
    a[k,i] = b[i] * c[k]
```

Homework 2

Group discussion

- https://cmse822.github.io/assignments/hw2.html