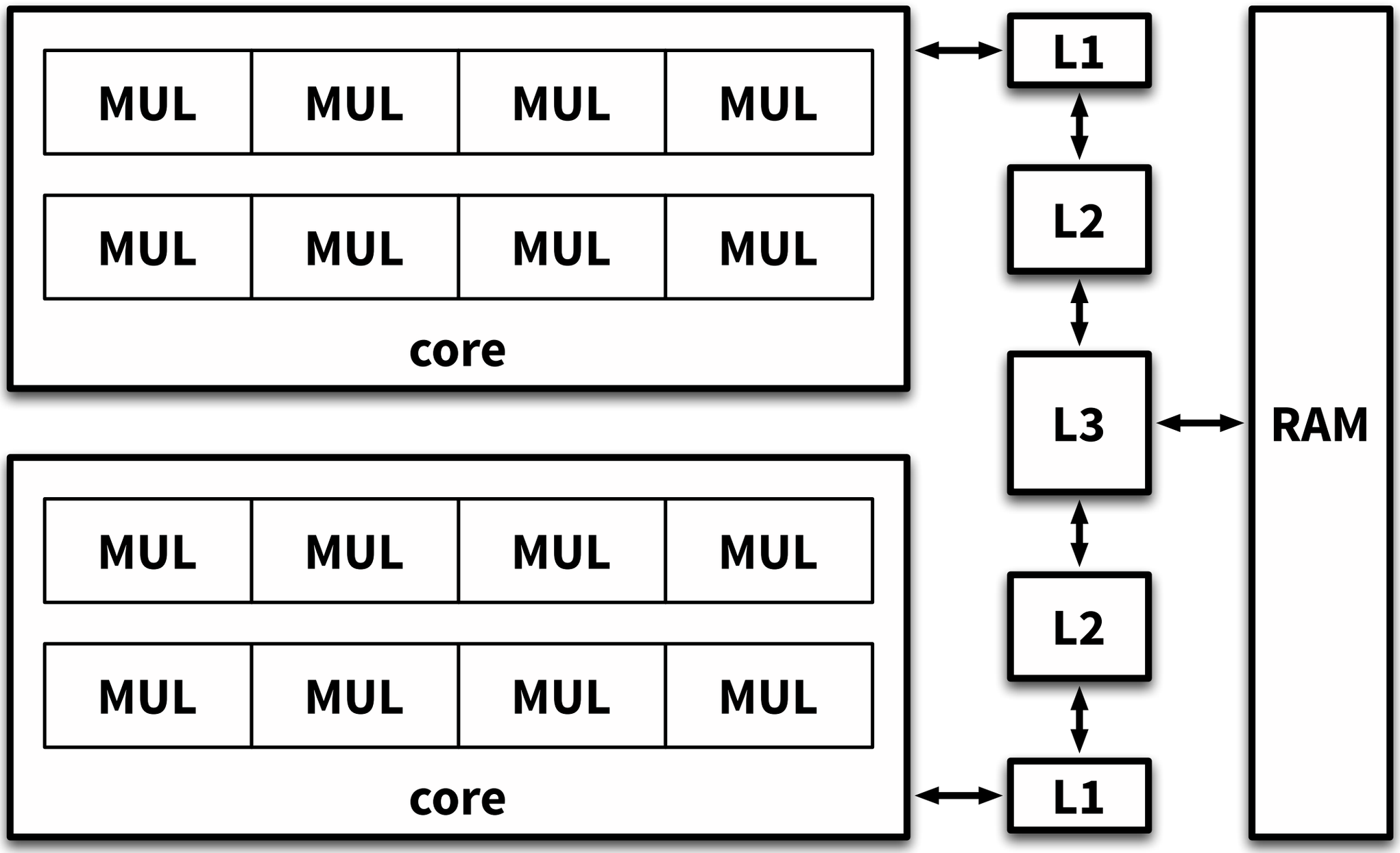

MUL
MUL
MUL
MUL
MUL
MUL
MUL
MUL
core
MUL
MUL
MUL
MUL
MUL
MUL
MUL
MUL
core
L1
L2
L3
L2
L1
RAM

Today performance comes from parallelism

- **Old kind of progress:**
  - new computers can perform individual operations *faster* than old computers
- **New kind of progress:**
  - new computers have *more parallel units* than old computers

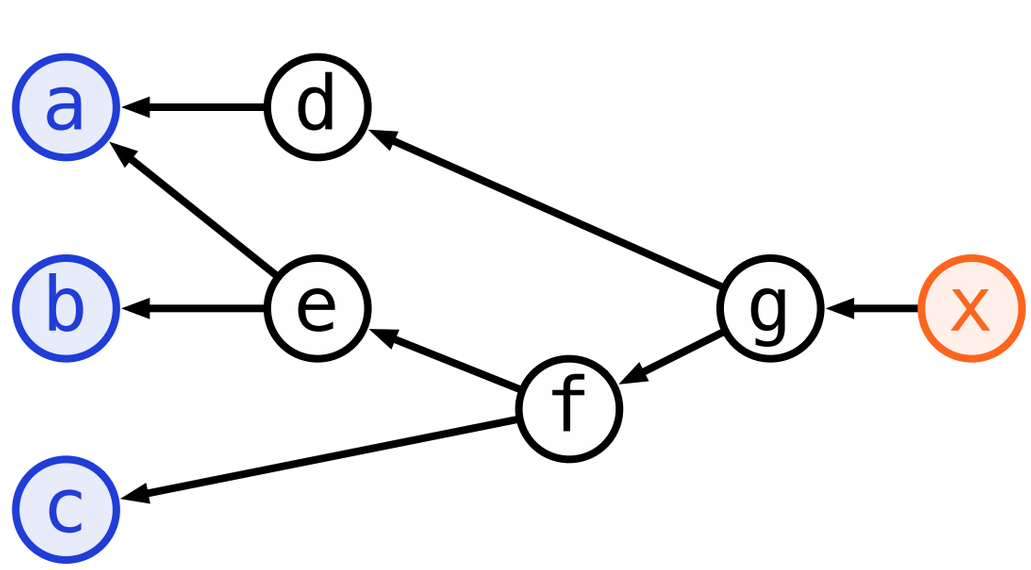
Exploiting parallelism requires independence

- **You can exploit parallelism only if your algorithm has *independent* operations**
  - `a0 *= b0; a1 *= b1; a2 *= b2; ...`
- **Long chain of dependencies: no opportunities for parallelism**
  - `a *= b0; a *= b1; a *= b2; ...`

Key concepts

Dependency graph, independence, depth

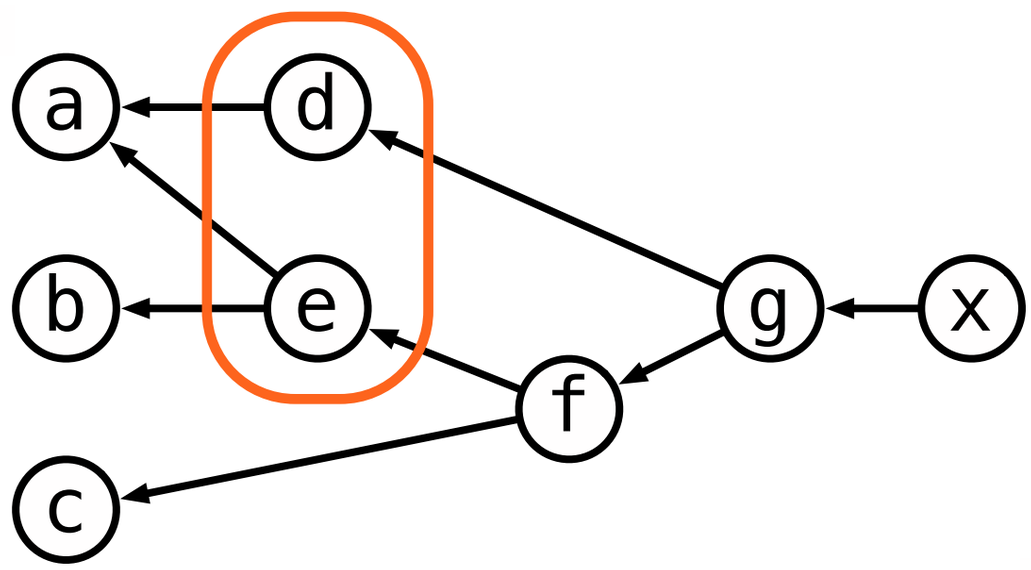
```
d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
```



Key concepts

Dependency graph, *independence*, depth

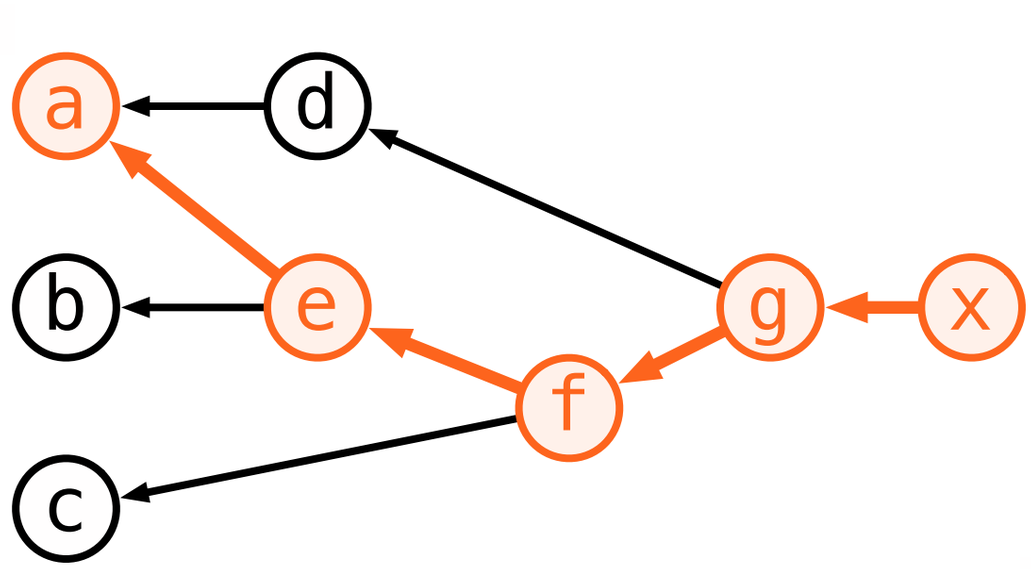
```
d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
```



Key concepts

Dependency graph, independence, *depth*

```
d = op1(a)
e = op2(a, b)
f = op3(c, e)
g = op4(d, f)
x = op5(g)
```



Two main kinds of parallel hardware

- **Pipelining**
  - operation A starts at time 0, ready at time 5
  - operation B starts at time 1, ready at time 6…
- **Multiple parallel units**
  - can perform operation A and operation B at the same time

Pipelining in hardware

- **Arithmetic units in CPU and cores in GPU**
- **Memory accesses, disk accesses**
- **Data transfer CPU–GPU**
- **Data transfer over network…**

Multiple parallel units in hardware

- **Multiple cores, multiple processors, multiple GPUs, multiple arithmetic units …**
- **Wide registers, vector operations**
- **Wide memory bus, cache lines**
- **Computer clusters**

Exploiting pipelining in software

- **CPU arithmetic, memory accesses: automatic for *independent operations***
- **GPU arithmetic, memory accesses: have *lots of threads* ready for execution**
- **Data transfers, disk access, networking…: *large data blocks*, *asynchrony*, *threading***

Exploiting parallel units in software

- **CPU multiprocessor, multicore:**
  use *threads* (e.g. OpenMP)

- **CPU arithmetic units:**
  automatic for *independent operations*

- **CPU vector operations:**
  use explicitly *vector types*

Exploiting parallel units in software

- **GPU multiprocessor, multicore: use *lots of threads***
- **Multi-GPU: switch GPUs in your code**
- **GPU + CPU: asynchronous GPU operations**

Exploiting parallel units in software

- **Wide memory bus: organise data so that memory references are *localised***
- **Clusters: use e.g. MPI to *exchange messages* between computers**

It is not just multithreading!

- **E.g. on my laptop:**
  - independent operations: ≈ **10** times faster
  - vector operations: ≈ **4–8** times faster
  - multiple threads: ≈ **2** times faster

Avoiding memory bottleneck

- ***Arithmetic intensity***
  - how much useful work do we do
    per one read from main memory to caches?
  - how much useful work do we do
    per one read from caches to registers?

Avoiding memory bottleneck

- **Arithmetic intensity**
- ***Cache blocking*** **may help here**
  - plan what to keep in caches
  - use registers explicitly as a cache
  - GPU: use shared memory explicitly as a cache

Demystifying hardware

- **Hardware does many things automatically**
  - caches, out-of-order execution, pipelining…
- **Do not just blindly *assume* that all this helps with your code**
- ***Design* your code so that all this helps!**

Demystifying hardware

- **Measure the performance**
- **Check the specifications of the hardware**
- **Do the math — is your code efficient?**
- **Understand *why* this happens!**

Demystifying hardware

- **Ballpark figures and back-of-the-envelope calculations are often sufficient**

- **Understand *orders of magnitude***
  - 10^{12} units of data >> your RAM
  - 10^{12} operations << one minute

(very roughly)

10 TB: network storage

1 TB: desktop SSD

100 GB: laptop SSD

10 GB: main memory

1 GB: GPU memory

…

1 MB: L3 cache

100 KB: L2 cache

10 KB: L1 cache, shared memory

1 KB: all registers together

100 B: one vector register, cache line

10 B: one scalar register

Basic primitive: parallel for loop

```
for (i = 0; i < n; ++i) {
    x[i] = y[i] * z[i];
}
```

Basic primitive: parallel for loop

// **Pipelining**: automatic!

```
for (i = 0; i < n; ++i) {
    x[i] = y[i] * z[i];
}
```

Basic primitive: parallel for loop

// **OpenMP**: explicit pragma

```
#pragma omp parallel for
for (i = 0; i < n; ++i) {
    x[i] = y[i] * z[i];
}
```

Basic primitive: parallel for loop

// **SIMD**: pack data in vectors, use vector types
// (assuming here that n is a multiple of 8)

```
for (i = 0; i < n / 8; ++i) {
    vecx[i] = vecy[i] * vecz[i];
}
```

Basic primitive: parallel for loop

// **CUDA**: multiple blocks

```
i = blockIdx.x;
x[i] = y[i] * z[i];
...
kernel<<<n, 1>>>();
```

Basic primitive: parallel for loop

// **CUDA**: multiple threads
// (assuming here that n is small)

```
i = threadIdx.x;
x[i] = y[i] * z[i];
...
kernel<<<1, n>>>();
```

Basic primitive: parallel for loop

// **CUDA**: multiple blocks and multiple threads
// (assuming here that n is a multiple of 32)

```
i = threadIdx.x + blockIdx.x * 32;
x[i] = y[i] * z[i];
...
kernel<<<n/32, 32>>>();
```

Basic primitive: parallel for loop

- **A vector operation is just a parallel for loop**
- **CUDA kernel launch is just a parallel for loop**
- **Universal: virtually any tool for any kind of parallel programming supports at least parallel for loops** (in some form)

More details that might matter…

- **False sharing:**
  - thread A reading and writing x[0], x[2], x[4] …
  - thread B reading and writing x[1], x[3], x[5] …
  - everything in the cache happens in full *cache lines* (64 bytes)
  - x[2k] and x[2k+1] in the same cache line

More details that might matter...

- **False sharing:** avoid
- **Register renaming:**
  - `a = x[0]; ++a; y[0] = a;`
    `a = x[1]; ++a; y[1] = a;`
  - `a = x[0]; ++a; y[0] = a;`
    `b = x[1]; ++b; y[1] = b;`

More details that might matter...

- **False sharing:** avoid
- **Register renaming:** helps you
- **Branch prediction:**
  - CPU makes a guess for each *conditional jump*
  - misprediction: penalty ≈ length of pipeline

More details that might matter...

- **False sharing:** avoid
- **Register renaming:** helps you
- **Branch prediction:** avoid hard-to-predict jumps
- **Virtual memory:** extra indirection

More details that might matter...

- **Typically, such details are important only when trying to use more than 50% of available CPU power**
- **Seemingly efficient C++ code may use less than 1 % of available CPU power!**
  - e.g. CP1 vs. CP4 vs. theoretical limits

Always matters a lot: independence

- **Compilers and hardware take care of lots of tiny low-level details pretty well**

- **Just make sure you have *independent* operations in your code so that there are *opportunities* for efficient parallel execution**

Take-home message

- **With some attention to parallelism, you can often *easily* speed up computation by a factor of 10 or more**

- **It can be as easy as:**
  - avoiding some unnecessary dependencies
  - throwing in e.g. #pragma omp parallel for

Reminder: careful with data races!

- **These are always fine:**
  - multiple threads reading, no thread writing
  - only one thread writing and reading
- **These *always* require some extra care:**
  - one thread writes, another thread reads
  - many threads write

What else is there

- **Concurrency:**
- what happens under the hood?
- how can one efficiently implement e.g. OpenMP critical sections?

What else is there

- **Concurrency**
- **Distributed computing:**
  - multiple computers
  - clouds, clusters …
  - MapReduce, MPI …

What else is there

- **Concurrency**
- **Distributed computing**
- **External memory:**
  - processing data sets much larger than RAM
  - cf. cache blocking

What else is there

- **Concurrency**
- **Distributed computing**
- **External memory**
- **Design and analysis of parallel algorithms**

What else is there

CSE-E5430 **Scalable Cloud Computing**

This week

Challenge!

Course feedback!