Schedule --- ‘Hardware’ segment of the class

Homework deadline

- 11/16 (late in the day): part 1
- 11/23 (2pm!): part 2

Class schedule

- 11/4 Class material cont’d; time for questions regarding the homework
- 11/9 (today) Quiz review; time for questions regarding the homework
- 11/11 Quiz: 2 hours, starting at the beginning of normal class time
  - Quiz is designed for 1 hour. 2 hours will give everybody enough time to complete the task
  - I will be sitting in during the first hour (60 minutes, not the whole class period) to answer questions
- 11/23 Quiz and homework return

Recap ‘Data Streams’

Bandwidth

So far we have talked about data streams and how to improve bandwidth

Recap ‘Data streams’

- 300 cycles to move data between main memory to the CPU

Assume no streams for the example: a = b + c

- Average bandwidth = 1 ‘word’ per 200 cycles (0.005 wpc)
    - ‘a’, ‘b’, and ‘c’ are 4-byte or 8-byte words

wpc: our unit

- Moving ‘b’ and ‘c’ from memory to CPU: 2 words per 300 cycles
- Moving ‘a’ from CPU to memory: 1 word per 300 cycles
- Average: 3 words per 600 cycles = 0.005 wpc

round-trip: 600 cycles

Summary

Part 1

Keep in mind that we have mostly discussed concepts

Many relevant details of the 3 major hardware features are still missing
We have not discussed the ramifications for code design

Particularly our discussion of caches is very much incomplete!
(and we will continue with caches in the next section)

Main bottlenecks

Transfer the data between memory and CPU

Actual computation

Major technologies to increase **concurrency**

Streaming of data between memory and CPU to hide memory latency and increase memory bandwidth

Pipelining computation (add/mult) to increase compute throughput

Caches to re-use data in order to decrease pressure on the Memory-to-CPU connection
and to also to hide memory latency and to increase memory bandwidth

Concurrency

A single action (flops, memory operation, etc.) can only be so fast

Clock speed, power consumption, speed of light

Increasing the '**concurrency**' is the best (maybe only) way to increase performance substantially

Our Computer: CPU, Cache, Memory, ‘Connection’

CPU
1. Pipelined operation

System designed to get 1 opc

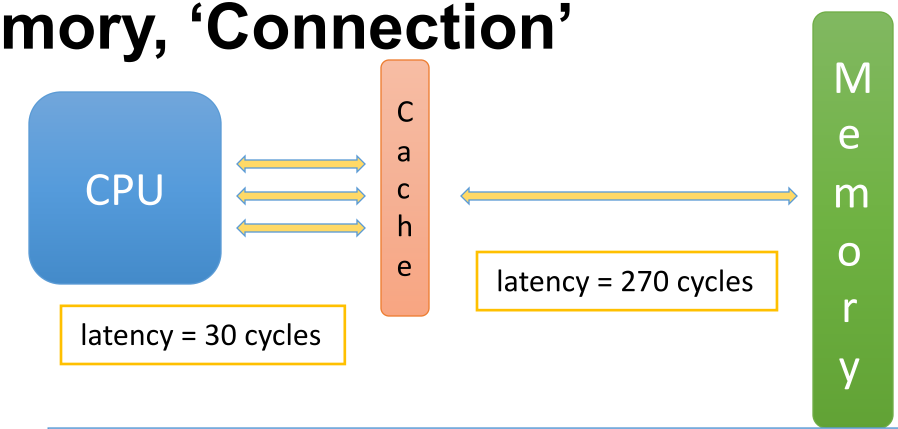
Memory
1. Data streams

System designed to support 1 wpc (for one row)

Caches
1. Managed by run-time
2. Cache size (for stencil update)

System designed for ‘enough’ bandwidth to support 2 rows
Size: at least 3×n words

Our computer has been somewhat ‘hypothetical’ so far
We have designed the specs so that we get ‘optimal’ performance for a stencil update



Requirement: Size of the cache = 3×n

n could be any number, any large number

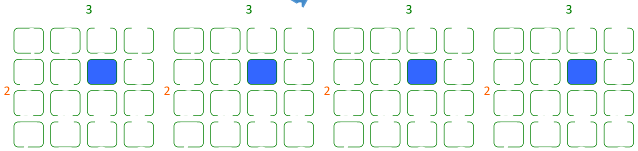
Size of cache in hardware certainly not adjustable
Also differences between chip generations

Cache Associativity

Let’s make up some address space notation

- Address has 6 digits
- Each digit holds a value between 1 and 4

Example: Cache with 4-way associativity



We encounter addresses in this order

What if I [illegible] it like this

```
1414 23
1414 24
1414 31
4433 11
1414 32
1414 33
1211 23
4433 23
…
```

An element of data may be cached in 1 of 4 locations

FIFO: replace the ‘oldest’ data element (out of the 4)

Advantages of Caches with Associativity

1. Ineffective mapping alleviated (somewhat)
2. Compare addresses only for few (4) locations
3. Eviction policy FIFO or LRU schedule
4. Keep ‘age’ for few (4) entries

(In principle, ‘random’ eviction could work, too)

Recap

Topics that we have addressed so far

Data streams

Pipelining

Caches

- Why?
- Cache blocking (software)
- Address mapping
- Eviction policy (FIFO, LRU, etc.)
- Associativity
    - fully associative, set-associative, direct mapping
- Storage efficiency (for addresses)
    - Cache lines
- Cache coherency (MOSI protocol)
- Shared-memory architecture
- False sharing

One more hardware feature to cover (unrelated to caches)

And then we will start looking into 'writing fast code'

All these topics come with these questions:

- What is it?
- Why do we need it?
- How is it implemented?
- Is there a specific trick?
- How does this increase concurrency?

Overarching idea

Single transaction is limited in speed

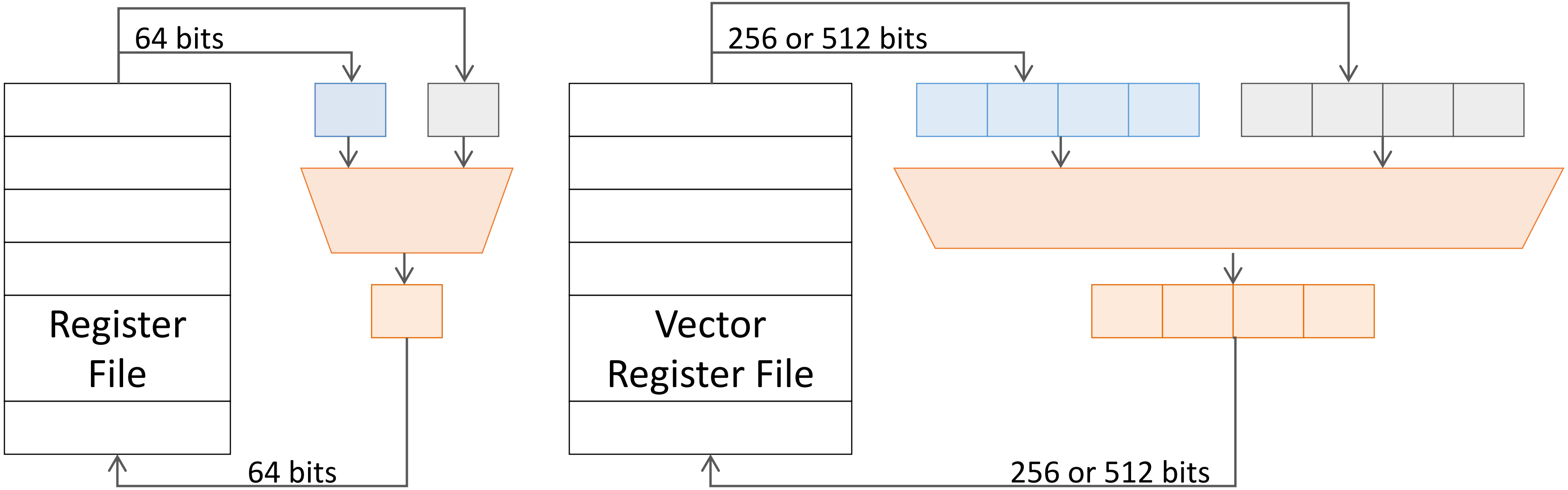
Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

Vector Hardware

a(i) = b(i) + c(i)

Example for vector length = 4 words



Scalar Unit

Input: 2 words (single or double precision)
Output: 1 word
Operations: 1 operation ➔ 1 result

Vector Unit

Input: 2 cache lines
Output: 1 cache line
Operations: 1 operation ➔ 8/16 results (dp/sp)

Strided Memory Access

```
do i=1, m
   a(1*i) = b(1*i) + c(1*i)
end do
```

Stride 1 access

```
do i=1, m
   a(2*i) = b(2*i) + c(2*i)
end do
```

Stride 2 access

```
do i=1, m
   a(8*i) = b(8*i) + c(8*i)
end do
```

Stride 8 access

Which access is best?
Lower strides are better
Stride-1 is best

Which one is worse?
Higher strides are worse
Stride-8, and stride-n are worst

```
do i=1, m
   a(n*i) = b(n*i) + c(n*i)
end do
```

Stride ‘n>8’ access

Lower ‘effective’ memory bandwidth

Lower number of results per numerical vector operation

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes

Loop iterations are independent

```
do i=2, n-1
  a(i) = a(i-1) + a(i+1)
end do
```

No

Loop iterations are not independent

```
for ( int i=0; i<n; i++ ) {
  temp = a[i] + 2.;
  a[i] = b[i];
  b[i] = temp;
}
```

Yes

Compiler resolves dependencies for scalars like temp, and also takes care of unnamed constants

Vector Lanes

All elements!

```
do i=1, n
  a(i) = b(i) + c(i)
end do
```

Complete cache lines are loaded
Unwanted results are not stored back to register

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)

+

c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)

1	1	1	1	1	1	1	1

=

a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

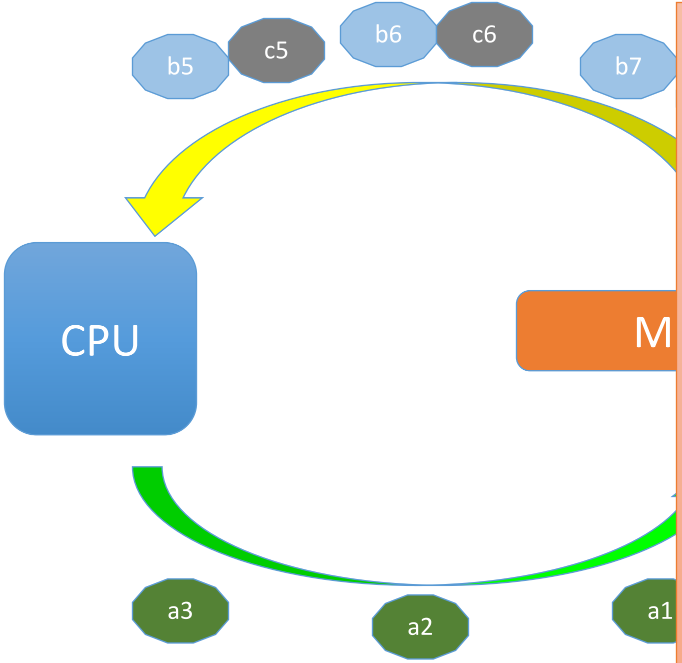
Load b(1:8)
Load c(1:8)
Compute: add
Store a(1:8) – full mask

Total: 4 instructions
8 results

Mask: All results copied back

Prefetching

a(i) = b(i) + c(i)



The computer does not know the code and cannot infer anything from the pattern in the code.

However, it can analyze the pattern from previous memory access. For example, data is requested cache line by cache line.

This is called prefetching
Prefetch instructions are added either

- during execution by the hardware (hardware prefetching)
- or to the assembly code by the compiler (software prefetching)

Typical defaults (x86 architecture):

- Software prefetching is off
- Hardware prefetching is on

Prefetching fills the data streams
Unwanted data (that is not useful) is ignored

Recap: Vectorization & Prefetching

Vectorization: When, how, why?
Under what circumstances can loops be vectorized?
Vector lanes
Strided data access
Vectorization efficiency
Data transfer efficiency
Prefetching
Multi dimensional arrays: strice-1 v. stride-n
Pointers and ‘array overlap’; mostly in C

All these topics come with these questions:

- What is it?
- Why do we need it?
- How is it implemented?
- Is there a specific trick?
- How does this increase concurrency?

Overarching idea

Single transaction is limited in speed

Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

Optimizations in Code

Cost of a division: maybe 60 cycles (depends highly on precision and accuracy)

```
do i=1, n
  a(i) = b(i) + c(i)/x
enddo
```

→

```
xi = 1. / x
do i=1, n
  a(i) = b(i) + c(i) * xi
enddo
```

Note that the following is not exactly (bit-wise) the same

```
z1 = y / x

z2 = y * (1./x)
```

Do no assume that z1 is exactly/bit-wise z2

If you change the code, or if you allow the compiler to change the code,

you will get a slightly different result (last bits will vary)

Compiler options allow you to control (to some degree)

- Whether operations in code are replaced by cheaper operations
- Whether operations are ‘executed’ to the fullest accuracy (which is expensive)
  - The default is some ‘reasonable’ compromise between accuracy and speed

Many optimization techniques will change the rounding bits.

Hence, bit-wise reproducibility is not the goal

Refresher

```
do i=1, n
   a(i) = a(i)   + b(i)
enddo
```

```
do i=1, n
   a(i) = a(i-1) + c(i)
enddo
```

RAW

```
do i=1, n
   a(i) = a(i+1) + c(i)
enddo
```

WAR

```
do i=1, n
   a(i) = b(i) + c(i)
   d(i) = a(i) + e(i)
   a(i) = f(i) + g(i)
enddo
```

WAW

Do you get the same result when running forward and backward?

Can these loops be vectorized?

Definition of ‘WAR’, ‘RAW’, and ‘WAW’ at:

cvw.cac.cornell.edu/vector/coding_dependencies