Scientific and Technical Computing

Hardware and Code Optimization

Lars Koesterke

UT Austin, 10/22/19

Quiz review

General

Terminology

Concepts

Overview

Primary components

Limitations: clock tick, power consumption

Concurrency: 5 ‘items’

- Data streams & prefetching
- Pipelining & vectorization
- Caches

Data Streams and Prefetching

Data streams and prefetching

Fill data-pipeline between memory and CPU

How long does it take to ‘move’ data?

Efficiency: strided access, random access

All 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

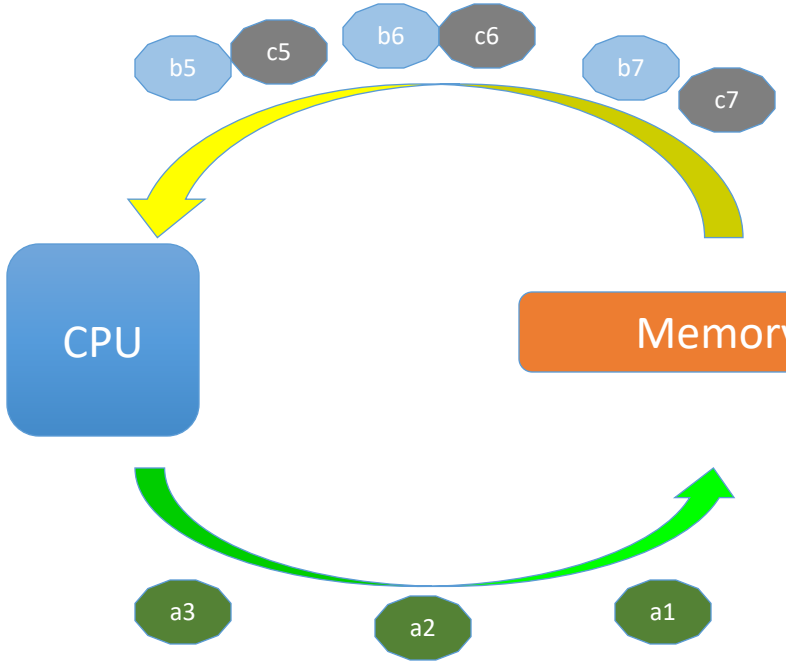
Single transaction is limited in speed

Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

Data Streams

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



1. How much data has to be ‘en route’?

Operation: 3 cycles
Data transfer: 300 cycles
Ratio: 1/100

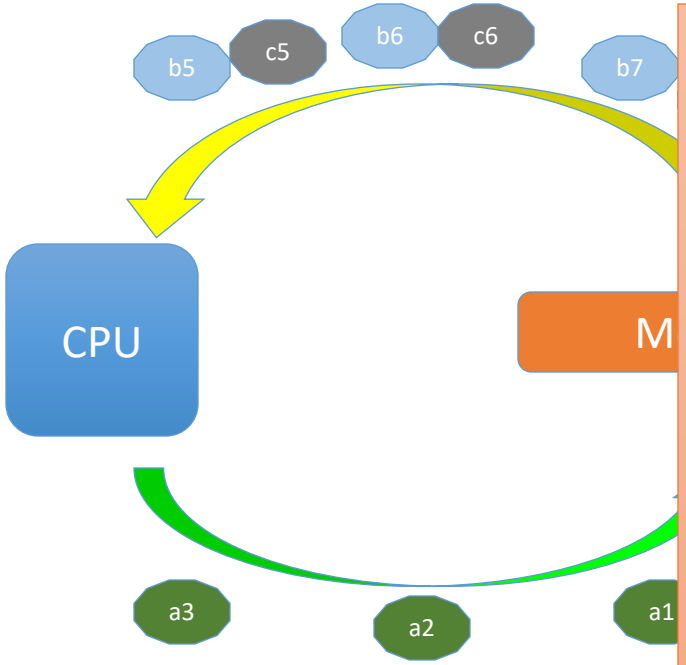
100 elements of a, b, and c

2. How many results (a) do we get per cycle? What is the speedup?

1 result every 3 cycles
Speedup is 200×

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)

Defaults:

- Software prefetching is off
- Hardware prefetching is on

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

Pipelining

How many cycles does it take for ‘add’ and ‘mult’?

How many results per cycle?

All 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

Caches

Mapping: direct, fully associative, set-associative

Mapping: address ‘calculation’

Eviction schemes: **FIFO, LRU**, random

Multi-level caches

MO(E)SI protocol

False sharing

Cache thrashing

All 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

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 latency and increase bandwidth

Pipelining computation (add/mult) to increase throughput

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

Cache: So far

What we have addressed so far

Why caches?
Cache blocking
Address mapping
Associativity
- fully associative, set-associative, direct mapping

Remaining problem

For each ‘cell’ in the cache we store the
- data (that’s what we are interested in)
- where the data came from (i.e. the address in main memory)

50% is useful (to us)
50% is book keeping

We can do better! But how?

Requirements

Know what is stored (full address)
Know how ‘old’ the data is

Finding data

Compare address with address in cache
Two possibilities
1. Match: load data from cache
2. No match: Load form memory, evict oldest data, store new data in place

Recap

Topics that we have addressed so far

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

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

One more hardware feature to cover (unrelated to caches)

And then we will start looking into ‘writing fast code’

Vectorization

Vector lanes

Masks

Efficiency

Strided access, random access

All 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

Recap: Vectorization

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

Code optimization

Hoisting operations outside of loops

Avoid artificial dependencies

Make code vectorizable

Avoid high-stride memory access

Stride-1 is best

Replace costly ops by cheaper ones

Minimize number of operations

All 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

Saving Power with Multiple Cores

Power = CV^2F

Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Cap = 2.2c
Volts = 0.6v
Freq = 0.5f
Power = $0.4cv^2f$

about 40% of the original consumption

Refresher

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

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

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

RAW

Can these loops be vectorized?

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

WAR

Definition of ‘WAR’, ‘RAW’, and ‘WAW’ at:
cvw.cac.cornell.edu/vector/coding_dependencies

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

WAW