Lecture 3:
Single processor architecture and memory

David Bindel

30 Jan 2014
Logistics

- Raised enrollment from 75 to 94 last Friday.
- Current enrollment is 90; C4 and CMS should be current?
- HW 0 (getting to know C4) due Tuesday.
- HW 1 (tuning matrix multiply) out Tuesday.
  - Teams of 2–3 (or 2–4)
  - Look for complementary groups!
Just for fun

http://www.youtube.com/watch?v=fKK933KK6Gg

Is this a fair portrayal of your CPU?

(See Rich Vuduc’s talk, “Should I port my code to a GPU?”)
The idealized machine

- Address space of named words
- Basic operations are register read/write, logic, arithmetic
- Everything runs in the program order
- High-level language translates into “obvious” machine code
- All operations take about the same amount of time
The real world

- Memory operations are *not* all the same!
  - Registers and caches lead to variable access speeds
  - Different memory layouts dramatically affect performance

- Instructions are non-obvious!
  - Pipelining allows instructions to overlap
  - Functional units run in parallel (and out of order)
  - Instructions take different amounts of time
  - Different costs for different orders and instruction mixes

Our goal: enough understanding to help the compiler out.
Prelude

We hold these truths to be self-evident:
  1. One should not sacrifice correctness for speed
  2. One should not re-invent (or re-tune) the wheel
  3. Your time matters more than computer time
Less obvious, but still true:
  1. Most of the time goes to a few bottlenecks
  2. The bottlenecks are hard to find without measuring
  3. Communication is expensive (and often a bottleneck)
  4. A little good hygiene will save your sanity
     - Automate testing, time carefully, and use version control
Today, a play in two acts:\(^1\)

1. Act 1: One core is not so serial
2. Act 2: Memory matters

\(^1\)If you don’t get the reference to *This American Life*, go find the podcast!
Act 1

One core is not so serial.
Parallel processing at the laundromat

- Three stages to laundry: wash, dry, fold.
- Three loads: darks, lights, underwear
- How long will this take?
Parallel processing at the laundromat

- **Serial version:**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>wash</td>
<td>dry</td>
<td>fold</td>
<td>wash</td>
<td>dry</td>
<td>fold</td>
<td>wash</td>
<td>dry</td>
<td>fold</td>
</tr>
</tbody>
</table>

- **Pipeline version:**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Dinner?</th>
<th>Cat videos?</th>
<th>Gym and tanning?</th>
</tr>
</thead>
<tbody>
<tr>
<td>wash</td>
<td>dry</td>
<td>fold</td>
<td>wash</td>
<td>dry</td>
<td>fold</td>
<td>wash</td>
<td>dry</td>
</tr>
</tbody>
</table>
Pipelining

- Pipelining improves *bandwidth*, but not *latency*
- Potential speedup = number of stages
  - But what if there’s a branch?
- Different pipelines for different functional units
  - Front-end has a pipeline
  - Functional units (FP adder, FP multiplier) pipelined
  - Divider is frequently not pipelined
Single Instruction Multiple Data

Old idea had a resurgence in mid-late 90s (for graphics)

Now short vectors are ubiquitous...
  ▶ C4 instructional: 128 bits (two doubles) in a vector (SSE4.2)
  ▶ Newer CPUs: 256 bits (four doubles) in a vector (AVX)
  ▶ And then there are GPUs!

Alignment often matters
Example: My laptop

MacBook Pro 15 in, early 2011.

- Intel Core i7-2635QM CPU at 2.0 GHz. 4 core / 8 thread.
- AVX units provide up to 8 double flops/cycle (Simultaneous vector add + vector multiply)
- Wide dynamic execution: up to four full instructions at once
- Operations internally broken down into “micro-ops”
  - Cache micro-ops – like a hardware JIT?!

Theoretical peak: 64 GFlop/s?
Punchline

- Special features: SIMD instructions, maybe FMAs, ...
- Compiler understands how to utilize these *in principle*
  - Rearranges instructions to get a good mix
  - Tries to make use of FMAs, SIMD instructions, etc
- In practice, needs some help:
  - Set optimization flags, pragmas, etc
  - Rearrange code to make things obvious and predictable
  - Use special intrinsics or library routines
  - Choose data layouts, algorithms that suit the machine
- Goal: You handle high-level, compiler handles low-level.
Act 2

Memory matters.
Roofline model

My machine

- Theoretical peak flop rate: 64 GFlop/s
- Peak memory bandwidth: 21.3 GB/s
- Arithmetic intensity = flops / memory accesses
- Example: Sum several million doubles (AI = 1) – how fast?
- So what can we do? Not much if lots of fetches, but...
Cache basics

Programs usually have \textit{locality}

- \textit{Spatial locality}: things close to each other tend to be accessed consecutively

- \textit{Temporal locality}: use a “working set” of data repeatedly

Cache hierarchy built to use locality.
Cache basics

- Memory latency = how long to get a requested item
- Memory bandwidth = how fast memory can provide data
- Bandwidth improving faster than latency

Caches help:
- Hide memory costs by reusing data
  - Exploit temporal locality
- Use bandwidth to fetch a cache line all at once
  - Exploit spatial locality
- Use bandwidth to support multiple outstanding reads
- Overlap computation and communication with memory
  - Prefetching

This is mostly automatic and implicit.
Cache basics

- Store cache lines of several bytes
- Cache hit when copy of needed data in cache
- Cache miss otherwise. Three basic types:
  - **Compulsory** miss: never used this data before
  - **Capacity** miss: filled the cache with other things since this was last used – working set too big
  - **Conflict** miss: insufficient associativity for access pattern

- **Associativity**
  - Direct-mapped: each address can only go in one cache location (e.g. store address xxxx1101 only at cache location 1101)
  - $n$-way: each address can go into one of $n$ possible cache locations (store up to 16 words with addresses xxxx1101 at cache location 1101).

Higher associativity is more expensive.
Teaser

We have $N = 10^6$ two-dimensional coordinates, and want their centroid. Which of these is faster and why?

1. Store an array of $(x_i, y_i)$ coordinates. Loop $i$ and simultaneously sum the $x_i$ and the $y_i$.

2. Store an array of $(x_i, y_i)$ coordinates. Loop $i$ and sum the $x_i$, then sum the $y_i$ in a separate loop.

3. Store the $x_i$ in one array, the $y_i$ in a second array. Sum the $x_i$, then sum the $y_i$.

Let’s see!
Caches on my laptop (I think)

- 32 KB L1 data and memory caches (per core), 8-way associative
- 256 KB L2 cache (per core), 8-way associative
- 6 MB L3 cache (shared by all cores)
A memory benchmark (membench)

for array $A$ of length $L$ from 4 KB to 8MB by 2x
for stride $s$ from 4 bytes to $L/2$ by 2x
time the following loop
  for $i = 0$ to $L$ by $s$
    load $A[i]$ from memory
membench on my laptop – what do you see?
membench on my laptop – what do you see?

- Vertical: 64B line size ($2^5$), 4K page size ($2^{12}$)
- Horizontal: 32K L1 ($2^{15}$), 256K L2 ($2^{18}$), 6 MB L3
- Diagonal: 8-way cache associativity, 512 entry L2 TLB
The moral

Even for simple programs, performance is a complicated function of architecture!

- Need to understand at least a little to write fast programs
- Would like simple models to help understand efficiency
- Would like common tricks to help design fast codes
  - Example: blocking (also called tiling)