Embedded computing for scientific and industrial imaging applications

Lecture 7 - Computer architecture, cache, optimization

How fast are computers?

How fast are computers?

- Kilo = thousand (10^3)
- Mega = million (10^6)
- Giga = billion (10^9)
- Tera = trillion (10^{12})
- Peta = 10^{15}
- Exa = 10^{18}

Processor speeds usually measured in Gigahertz these days.

Hertz means “machine cycles per second”.

One operation may take a few cycles.

So a 1 GHz processor (10^9 cycles per second) can do > 100, 000, 000 floating point operations per second (> 100 Megaflops).

The Cray-1 computer

- World's first "supercomputer"
- Sold to Los Alamos National Laboratory, National Center for Atmospheric Research (NCAR), etc. starting in 1976
- Price: up to $8.8 million
- Speed: 80-100 Mflops
- Memory: 8MB

Not long ago counting flops was the best way to measure performance for scientific computing.

How fast are computers?

Not long ago counting flops was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.
If A and B are $n \times n$ then so is C.
Each element c_{ij} is the inner product of ith row of A with jth column of B. Requires n multiplications and $n - 1$ additions to compute c_{ij} .n^2 elements in C ⇒ Requires $O(n^3)$ floating point ops total.
Note: $n = 10{,}000$ ⇒ $n^3 = 10^{12}$ (> 1, 000 seconds on 1 GHz processor)

But these days, the bottleneck is often

getting data to and from the processor!

Note that each element of A, B is used n times.

Memory Hierarchy

(Main) Memory: “Fast” memory that is hopefully large enough to contain all the programs and data currently running.
(But not nearly fast enough to keep up with CPU.) Typically 4 – 16 GB.
Recall GB = gigabyte = 10^9 bytes = 8×10^9 bits.

For example, 1GB holds a single 10, 000 × 10, 000 matrix of floating point values (8 bytes each), or 125 matrices that are each 1000 × 1000.

Hard Drive: Slower memory that contains data (including photos, video, music, etc.) and all programs you might want to use.
Typically 128 – 1024 GB. (Slower but cheaper.)

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer. On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

$$\Rightarrow \text{at most 4GB of memory can be addressed.}$$

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

$$2^{64} = 1.84 \times 10^{19} \text{ distinct addresses}$$

$$\Rightarrow \text{could address an exabyte of memory.}$$

CPU and registers

CPU (central processor unit)

Executes instructions such as add or multiply.

Takes data from registers, performs operations, stores back to registers.

- Transferring between registers and processor is very fast.
- Different types of registers, e.g.
  - Integer, floating point
  - instruction registers
  - address registers
- Generally a very small number of registers.
- Data and instructions must be transferred between other memory and registers as needed.

Memory Hierarchy

Between registers and memory there are 2 or 3 levels of cache, each larger but slower.

- Registers: access time 1 cycle
- L1 cache: a few cycles
- L2 cache: ~ 10 cycles
- (Main) Memory: ~ 250 cycles
- Hard drive: 1000s of cycles

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Exploit parallelism to hide latency and increase throughput.

Exploit parallelism to hide latency and increase throughput.

Even a “single core” machine has lots of things going on at once.

For example:

- Pipelined operations
- Executing / fetching / storing
- Prefetching future instructions
- Prefetching data into cache

5-stage instruction pipeline for RISC machine

- F = Instruction Fetch,
- ID = Instruction Decode,
- EX = Execute,
- MEM = Memory access,
- WB = Register write back.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Instruction pipelining https://en.wikipedia.org/wiki/Instruction_pipelining

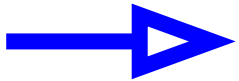
Reducing memory latency

Reduce memory fetches by reusing data in cache as much as possible. Requires temporal locality.

Very simple example: if length much larger than cache size,

```
for (i = 0; i < length; i++)
{
     z = z + pX[i];
}
for (i = 0; i < length; i++)
{
     w = w + 3.0 * pX[i];
}
```

0.09789 ms



```
for (i = 0; i < length; i++)
{
     z = z + pX[i];
     w = w + 3.0 * pX[i];
}
```

0.06525 ms

Check `$CLASS_REPO/codes/07_codes`

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.
A cache line typically holds 64 or 128 consecutive bytes (8 or 16 double precision floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Another cache line will be brought from higher up the hierarchy, and may displace some variables in cache.
Those cache lines will first have to be written back to memory.

Bottom line:
Good to do lots of work on each set of data while in cache, before it has to be written back.

Organize algorithm for Temporal locality.

Spatial locality

Also good to organize algorithm so data that is consecutive in memory is used together when possible.

If data you need is scattered through memory, many cache lines will be needed and will contain data you don’t need.

This is called spatial locality.

Spatial locality

Suppose A is $n \times n$ matrix,
D is $n \times n$ diagonal matrix with diagonal elements d_i.

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.
Which is better ?? Same number of flops!

```
for (i = 0; i < n; i++)
{
     for (j = 0; j < n; j++)
     {
          b[i][j] = d[i] * a[i][j];
     }
}
```

6.42944 ms

```
for (j = 0; j < n; j++)
{
     for (i = 0; i < n; i++)
     {
          b[i][j] = d[i] * a[i][j];
     }
}
```

29.10824 ms

Check `$CLASS_REPO/codes/07_codes`

More about cache

Simplied model of one level direct mapped cache.
32-bit memory address: 4.3×10^9 addresses
Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.
9 low order bits of memory address determine cache address.

For a memory fetch:

- Determine cache address, check if this holds desired words from memory.
- If so, use it.
- If not, check “dirty bit” to see if has been modified since load.
- If so, write to memory before loading new cache line.

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.
Cache line holds 64 bytes = 8 floats. 4096/8 = 512 cache lines per column of matrix.

Loading one column of matrix will fill up cache lines
0,1,2,...,511.

Second column will go back to cache line 0.
But all elements in cache have been used before this happens, Prefetching can be done by optimizing compiler.

Worse — Going across the rows:
The first 8 elements of column 1 go to cache line 0.
The first 8 elements of column 2 also map to cache line 0.
Similarly for all columns. The rest of cache stays empty.

More about cache

If cache holds more lines:

1024 lines ⇒

first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

In practice cache is often set associative: small number of cache addresses for each memory address.

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary. “Premature optimization is the root of all evil” (Don Knuth)
- If so, determine which parts of code need to be improved and spend effort on these sections.
- Use optimized software such as BLAS, LAPACK.

References

- Applied Mathematics 483/583 - High Performance Scientific Computing at University of Washington