Caching

Ned Nedialkov

McMaster University
Canada

CS/SE 4F03
January 2016

Outline

Caches

- Cache is memory that can be accessed faster than main memory
- Can be located on the same chip as the CPU or on a separate chip
- Locality: access to one location is followed by an access to a nearby location
- Spatial locality: if a memory location is accessed, then a nearby location is likely to be accessed in the near future
- Temporal locality: data are referenced repeatedly in a small time window
  Special case of spatial locality: accessing the same data

- Memory access operates on blocks of data and instructions
- Called cache blocks or cache lines
- Typically 8 to 16 times as much informations as a single memory location
- Example
  - Cache lines stores 16 floats, say a[0], a[1], . . . , a[15]
  - Each can be accessed fast
  - If a[16] is accessed, cache miss. Another cache line is brought into memory

- The cache is usually divided into levels, e.g. L1, L2, L3
- L1 is smaller and faster than L2, L2 is smaller and faster than L3, and L3 is smaller and faster than main memory
- Data in L1 is usually (but not always) stored in L2 and so on
- When accessing an instruction or data, the CPU works its way down the cache hierarchy
  - checks L1, if not in L1 checks L2 . . .
  - if the data is available: cache hit; otherwise cache miss
  - it can happen e.g. a cache miss in L1 and a cache hit in L2
- If a cache miss occurs, the CPU may stall waiting for the data

- When writing into a cache, the data in the cache and in the main memory are inconsistent
- Write-through cache: a line is written into main memory as soon as the data in the cache is updated
- Write-back cache: a line is written into main memory when it is replaced in the cache

Cache mappings

- Fully associative cache: a new line can be placed at any location in the cache
- Direct mapped cache: each line has a unique location in the cache
- *n*-way set associative cache: each cache line is placed in one of *n*-different locations

Example: cache mappings

- Main memory: 16 lines, 0, 1, . . . 15
- Cache: 4 lines, 0, 1, 2, 3
- Fully associative: each line from memory can go into any of the cache lines
- Direct mapped: we can map using division by 4 and taking the remainder
- 2-way set associative
  - we have two sets of cache lines
  - set 0 contains lines 0 and 1
  - set 1 contains lines 2 and 3

We can map

memory line	cache line
0	0 or 1
1	2 or 3
2	0 or 1
3	2 or 3
4	0 or 1
⋮	⋮

- When more than one line can be mapped into a cache, which one to replace?
- E.g. assume line 0 is in location 0 and line 2 is in location 1
- Where to store line 4? It can be in 0 or 1
- Most common: evict the least recently used

Example: impact on performance

Which loop would execute faster?

```
for (i=0; i<n; i++)
     for (j=0; j<n; j++)
        y[j] += A[i][j]*x[j]
```

```
for (j=0; j<n; j++)
     for (i=0; i<n; i++)
        y[j] += A[i][j]*x[j]
```

Consider the following C program

```
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>
#include <math.h>

double getTime() {
  struct rusage usage;
  getrusage(RUSAGE_SELF, &usage);
  struct timeval time;
  time = usage.ru_utime;
  return time.tv_sec+time.tv_usec/1e6;
}

#define N 10000
double A[N][N], x[N], y[N];
int Y[N];

int main()
{
  int i,j;
```

```
double time;

// generate random entries
for (i=0; i<N; i++){
  x[i]=drand48();
  for (j=0; j<N; j++) A[i][j]=drand48();
}

// process by rows
for (i=0; i<N; i++)     y[i]=0;
time = getTime();
for (i=0; i<N; i++) {
  for (j=0; j<N; j++)
      y[i] += A[i][j]*x[j];
}
printf("Accessing␣by␣rows.␣Time␣is␣%␣.2e\n",
       getTime()-time);
```

```
  // process by columns
  for (i=0; i<N; i++) Y[i] = 0;
  time = getTime();
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      Y[i] += A[i][j]*x[j];

  printf("Accessing␣by␣cols.␣Time␣is␣%␣.2e\n",
         getTime()-time);

  return 0;
}
```

On Mac OSX system

- 1.7GHz Intel Core i5
- Two cores
- L2 Cache (per Core): 256 KB
- L3 Cache: 3 MB
- 4GB 1333MHz DDR3

Compiled with gcc version 4.2.1 with -O3, this program outputs

```
Accessing by rows. Time is  1.32e-01
Accessing by cols. Time is  2.20e+00
```

About 16.7 times faster if processed by rows!

Latency and bandwidth

- Latency: the delay between the CPU issuing a request for a memory item and the arrival of the item
  - measured in nanosecond or clock cycles
    1 ns = 10^{-9} seconds
- Bandwidth the rate at which data is obtained from memory
  - measured in bytes (kilo, mega, giga) per second

Example: effect of memory latency on performance

Assume a processor

- operating at 1 GHz
  1 ns clock, 1 ns = 10^{-9} seconds
- with two multiply-add units
- can execute 4 instructions per cycle
  i.e. 4 instructions every 1 ns
- Peak performance
  - 4 instructions every 1 ns or 4 instructions every 10^{-9} seconds
  - in 1 second, $4/10^{-9} = 4 \times 10^9$ operations
  - 4 GFLOPS
- FLOPS: Floating-Point Operations per Second

Now assume

- The processor is connected to a DRAM with 100 ns latency and no caches
- Every time a memory request is made, the processor waits 100 cycles

Consider computing a dot product

- One multiply and one add on each two elements
  - 2 floating-point operations and 2 data fetches
  - 1 FLOP per data fetch
  - 1 FLOP every 100 cycles or 100 ns
  - In 10^9 cycles we have $10^9/100 = 10 \times 10^6$ FLOPS
- 10 MFLOPS

Example: impact of caches

- Consider 1GHz processor with a 100 ns latency DRAM
- Assume a 32 KB cache with 1 ns latency
- Consider multiplying two 32×32 matrices A and B
  - the cache can store A and B and the result $C = AB$
- To fetch the matrices into cache we fetch 2 matrices each of $32 \times 32 = 1024$ or 1K words
  Total is 2K words
  - one word every 100 ns $= 10^{-7}$ seconds
  - 2K words in $\approx 2000 \times 10^{-7}$s $= 200 \times 10^{-6}$ s $= 200\ \mu$s
    1 μs $= 10^{-6}$ seconds

- Multiplying 2 matrices takes $\approx 2n^3$ operations
- $2 \times 32^3 = 2^{16} = 64$K operations
- The processor can do 4 operations per cycle
- We can multiply them in $64/4 = 16$K cycles
- 16K cycles $\times\ 10^{-9}s = 16\ \mu$s
- Total time is $200 + 16 = 216\ \mu$s
- That is, we do 64K FLOPS in 216×10^{-6} s
- In one second $64\text{K}/(216 \times 10^{-6}) \approx 303 \times 10^6$ FLOPS
- 303 MFLOPS

Example: impact of memory bandwidth

- Memory bandwidth (MB): the rate at which data is moved between the processor and memory
- Depends on the bandwidth of the memory bus and the memory units
- A common technique to increase MB is to increase the size of a memory block

- In the dot product example, we fetch one word every 100 cycles: peak performance is 10 MFLOPS
- Assume that a request for a word returns 4 words, cache line
- That is, we obtain 4 words every 100 ns, 8 words every 200 ns
- We have 8 FLOPS every 200 ns, or 1 FLOP every 25 ns
- 1 FLOP every 25×10^{-9} s
- In one second, $1/(25 \times 10^{-9}) = 0.04 \times 10^{9} = 40 \times 10^{6}$
- 40 MFLOPS

- In this example, we assumed 4 bytes are obtained at once
- In practice, first word is retrieved in 100 ns and each next word is obtained on a subsequent bus cycle
- 4 words are available after $100 + 3 \times$ (memory bus cycle)
- Assume a data bus at 200MHz
  Bus cycle $1/(200 \times 10^6) = 0.005 \times 10^{-6} = 5$ ns for each next word
- 4 words are obtained in $100 + 3 \times 5 = 115$ ns
- 8 FLOPS each 230 ns
- 1 FLOP each 230/8 ns $= 230/(8 \times 10^{-9})$ s
- In one second, $8/230 \times 10^9 \approx 34.8 \times 10^6$
- 34.8 MFLOPS
- Not the latency did not change, only the block size changed