CS420 – Lecture 3

Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Example

P1: Program assigns random values to consecutive entries in an array of length 5,000,000. 10 measurements are made.

```
#define N 5000000
int a[N];
int main()
{
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<int> dis(0, N);
  uint64_t i, j, k;
  for (k = 0; k < 10; k++) {
    auto start = std::chrono::high_resolution_clock::now();
    for(i = 0; i < N; i++) {
      a[i] = dis(gen);
    }
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration =
      std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
    std::cout << duration.count() << "ms ";
  }
  std::cout << std::endl;
```

10 runs:
227ms 186ms 184ms 186ms 185ms 186ms 184ms 186ms 185ms 184ms

third level optimization.

g++ -std=c++11 -O3 -o rand rand.cpp
73ms 63ms 55ms 51ms 50ms 50ms 51ms 53ms 52ms 55ms

Example

P2: Assigns values to random entries in an array of length 5,000,000.

```
for (k = 0; k < 10; k++) {
  auto start = std::chrono::high_resolution_clock::now();
  for(i = 0; i < N; i++) {
    a[dis(gen)] = i;
  }
  auto stop = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
  std::cout << duration.count() << "ms ";
}
std::cout << std::endl;
```

Example

P2: Assigns values to random entries in an array of length 5,000,000.

```
for (k = 0; k < 10; k++) {
  auto start = std::chrono::high_resolution_clock::now();
  for(i = 0; i < N; i++) {
    a[dis(gen)] = i;
  }
  auto stop = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
  std::cout << duration.count() << "ms ";
}
std::cout << std::endl;
```

245 ms

Example

P3: Same number of assignments as the second program but using a shorter array of length 50,000.

```
int M = 100;
uint64_t new_N = N / M;
std::uniform_int_distribution<int> new_dis(0, new_N);
for (k = 0; k < 10; k++) {
  auto start = std::chrono::high_resolution_clock::now();
  for (j = 0; j < M; j++) {
    for(i = 0; i < new_N; i++) {
      a[new_dis(gen)] = i;
    }
  }
  auto stop = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
  std::cout << duration.count() << "ms ";
}
std::cout << std::endl;
```

52 ms

Example

$$[a[0] \dots a[M-1]] \times \begin{bmatrix} B[0][0] & \dots & B[0][N-1] \\ \dots & \dots & \dots \\ B[M-1][0] & \dots & B[M-1][N-1] \end{bmatrix} = [c[0] \dots c[N-1]]$$

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 10000
int a[N], B[N][N], c[N];
int main() {
 int i, j, k;
 time_t time;
```

```
 for(k = 0; k < 10;k++) {
  time = clock();
  for(j = 0; j < N; j++)
   for(i = 0; i < N; i++)
    c[j] += a[i] * B[i][j];
  time = clock() - time;
  printf("%lu ",
    1000*time/CLOCKS_PER_SEC);
 }
 printf("/n");
}
```

In C/C++ matrices are stored in row-major order:
`B[0][0],B[0][1],...,B[0][N-1],B[1][0],B[1][1],...`
`B[1][N-1],B[2][0],...,B[N-1][N-1]`

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
  c[j] += a[i]*B[i][j];
```

faster because it accesses continuous memory

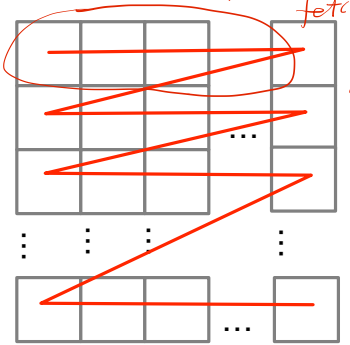
```
for (j=0;j<N;j++)
 for (i=0;i<N;i++)
  c[j] += a[i]*B[i][j];
```

relate to cache.

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
  c[j] += a[i]*B[i][j];
```

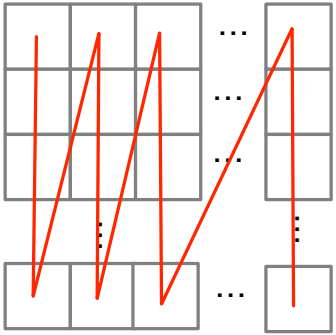
Matrix traversed in row-major order

spacial locality property: fetch a chunk of data then store in cache.



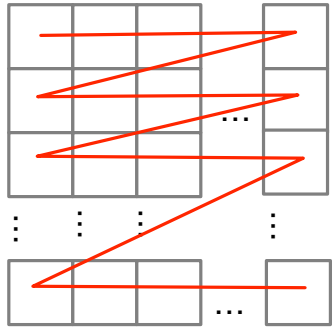
```
for (j=0;j<N;j++)
 for (i=0;i<N;i++)
  c[j] += a[i]*B[i][j];
```

matrix traversed in column-major order



```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
  c[j] += a[i]*B[i][j];
```

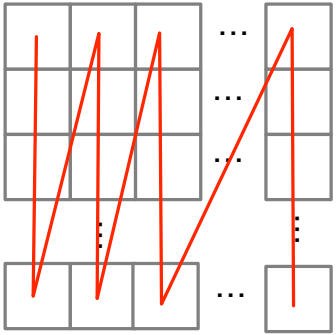
Matrix traversed in row-major order



Running time 34.9 ± 2

```
for (j=0;j<N;j++)
 for (i=0;i<N;i++)
  c[j] += a[i]*B[i][j];
```

matrix traversed in column-major order



Running time 821.4 ± 10.9

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

  Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them)

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them) # Registers is limited.
2. Load data earlier than needed for it to have time to arrive into register before it is used

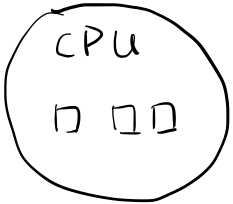
Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them)
2. Load data earlier than needed for it to have time to arrive into register before it is used
3. Caches

Caches



Store frequently used data.

SRAM (expensive)

in chip

access data from caches is quick.

caches size is small. (why)

what is capacitor? need to keep refreshing memory.

DRAM (cheap)

off chip

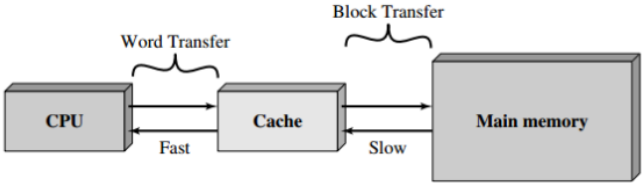
MM

Cache

- Can build large DRAM memory (off chip), but access time is high and bandwdith is low.
- Can build fast SRAM memory (on chip), but capacity is small and power consumption is high
- **Idea:** Use both to provide what looks like memory with capacity of DRAM and speed of SRAM
  - Keep frequently accessed data in cache

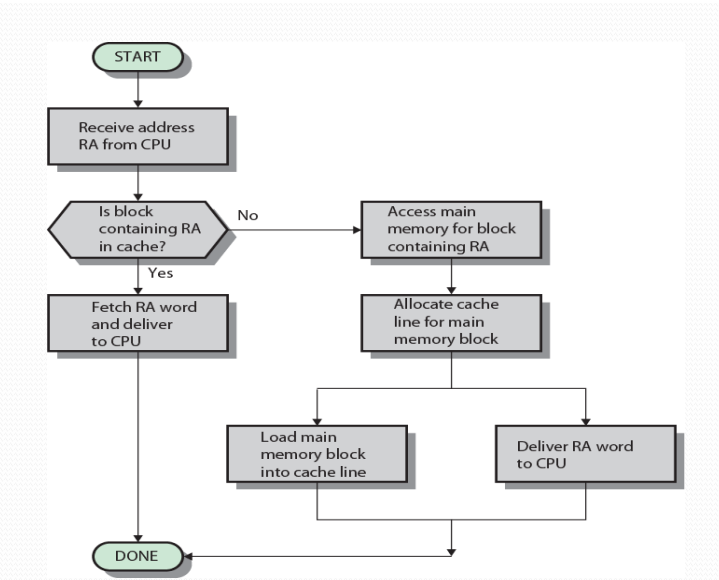
Computer Board

Cache



- Main memory is slow. In order to get higher bandwidth, data is moved from memory to cache in large blocks (typically 64 bytes) – *cache lines*.
- Data is moved from cache to registers in single words (8 bytes).
- A load first looks for the loaded word in cache; if it is not there, it is brought from memory to cache

Flow chart for a LOAD



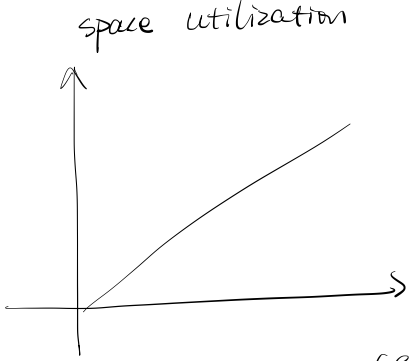
- How do we search the cache?
- What do we do if the cache is full?

Cache organization

Typical numbers

- Cache line = 64 bytes = 8 words (size of block transfered from memory)
- Cache size = 64 KB = 1024 cache lines
- Physical memory address is 48 bits – memory is byte addressable.
  - Typically, physical address is shorter than virtual address (i.e., value of pointer – 64 bits). We will not discuss virtual to physical address. translation.

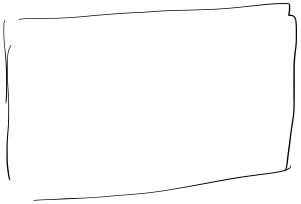
Direct mapped



Set associative mapping

tag

search in caches.



data

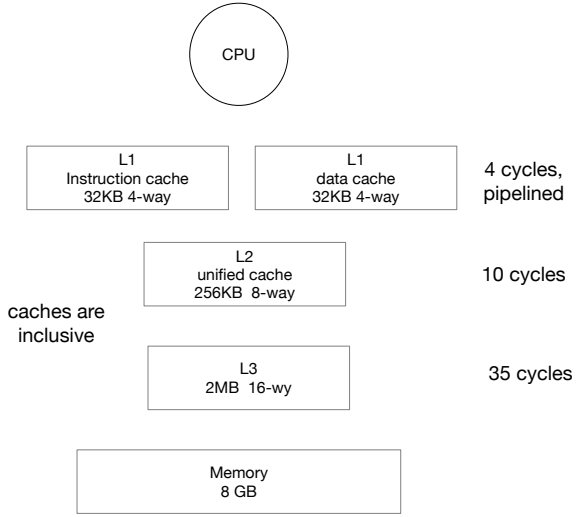
no need to search in memory.

Ideal cache: Fully associative cache

- The entire cache is one associativity set: A cache line can be stored anywhere in the cache.
- Least Recently Used (LRU): Always evict the line that was not accessed for the longest time.
- Not practical: Would need to compare all tags
- Practical caches are an approximation of a fully associative, LRU cache
- One can analyze code behavior assuming "ideal" caches

a[100] ... a[3]=88; print(a[3])

Can have multiple cache levels



Typical numbers

Prefetching

```
...
for (i=0;i<M;i++)
 for (j=0;j<N;i++)
  c[j] += a[i]*B[i][j];
...
```

- Accesses to `B` and `c` are to consecutive locations in memory
- The hardware will notice that after a few accesses and decide that the pattern will continue
- It will *prefetch* cache lines it guesses will be accessed in the near future
- Thus, "hiding" memory latency (loads will be serviced from cache)
- This does not change the results of the computation, but will often improve performance
- But wrong guesses will increase load on memory bus and pollute the cache

A more complex processor