Socket - last

Esempio multi: un esempio piu' realistico.

Flops

FLOPS e' un'abbreviazione di *Floating Point Operations Per Second* e indica il numero di operazioni in virgola mobile eseguite in un secondo dalla CPU.

Ad esempio nel caso del prodotto classico tra matrici, vengono eseguite $2*N^3$ operazioni, quindi ad esempio:

$$[\text{flops}] = 2*N^3 / \text{tempo}$$

Esercizio: Provate a scrivere un programma che esegua un prodotto matrice matrice (stampera' i flops).

Flops

Usando un programma di moltiplicazione matrice matrice se provo a cacolare i MFlops (= 1000000 flops) che riesco ad ottenere vedo:

```
$ gcc -O0 -o mm.1 mm.1.c
$ ./mm.1

Tempo impiegato per inizializzare 0.040000 s.
Tempo per prodotto classico 26.230000 s.
Tempo totale 26.270000 s.

Mflops ----------------> 81.746618
Controllo -------------> 268364458.846206
```

Valore Teorico della CPU usata circa 4 Gflops

Flops

Diminuendo le dimesioni delle matrici le cose cambiano sostanzialmente, perche' ?

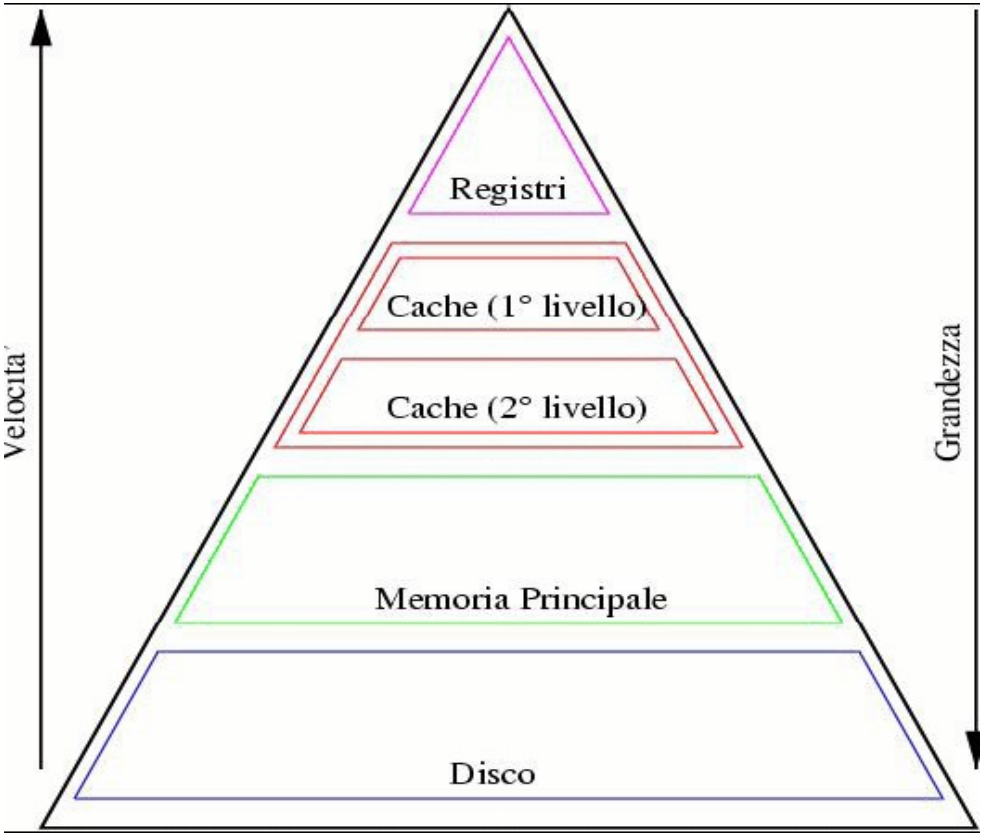
```
$ gcc -O0 -o mm.2 mm.2.c
$ ./mm.2
Tempo impiegato per inizializzare 0.000000 s.
Tempo per prodotto classico 1.200000 s.
Tempo totale 1.200000 s.

Mflops ----------------> 208.333333
Controllo -------------> 31275564.357626
```

Cosa e' la cache ?

Cache

La gerarchia di memoria nei processori attuali e' composta da diventi livelli di memoria, caratterizzati ciascuno da velocita' di accesso ai dati inversamente proporzionali alla dimensione: piu' sono grandi queste aree e maggiore e' il tempo richiesto per recuperare i dati in esso contenuti.



Cache

Livello	Dimensioni	Tempo di accesso
Registri	32 bit	1 ciclo
Cache L1	64 KB	1-10 cicli
Cache L2	4 MB	10-100 cicli
RAM	> 1 GB	> 100 cicli
Disco	> 10 GB	> 10000 cicli

Cache

In generale da 1 a 3 livelli di memoria cache sono installati. La memoria cache e' caratterizzata da diventi tempi di accesso e dimensioni, a seconda che essa sia all'interno del chip (on chip) o fuori dal chip (off-chip), e dalla tecnologia con cui sono realizzate le celle di memoria. (cat /proc/cpuinfo per vedere le dimensioni della cache)

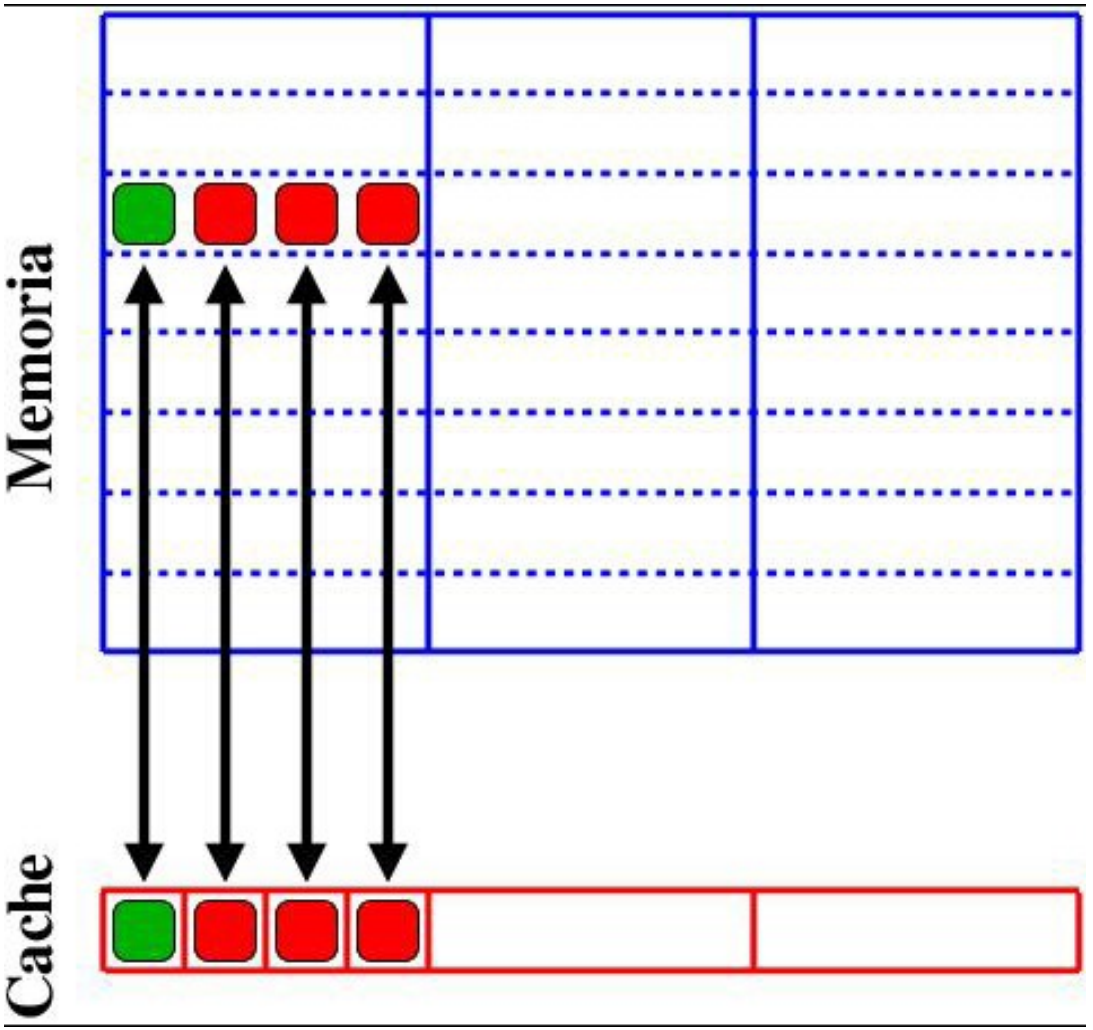
L'uso e il vantaggio di una cache si basa sul principio di localita' ovvero che un programma tende a riutilizzare dati ed istruzioni usate recentemente. Una conseguenza e una regola del pollice che dice che in genere il 90 % del tempo totale viene impiegato ad eseguire il 10% delle istruzioni [2]. Per essere piu precisi l'uso della cache e' vantaggioso quando di un codice si sfrutta sia la localita' spaziale che la localita temporale dei dati.

Cache

- Localita' temporale: una volta che viene usato un generico dato "a" questo verra riusato piu volte in un breve arco di tempo
- Localita' spaziale: se si usa il generico dato "a" allora anche il dato "b", che e conservato nella locazione di memoria contigua a quella di "a" verra' usato a breve.

Il fatto che nella maggior parte dei casi si lavori essenzialmente su un piccolo sottoinsieme dei dati totali permette di copiare nella cache L1, che ha latenza di pochi cicli, solo quei dati necessari alla CPU. Inoltre, grazie alla localita' spaziale, ogni volta che carico un dato dalla memoria alla cache non prendo solo quel dato ma anche altri dati contigui, ovvero la cosiddetta linea di cache, che e composta da 64-128 byte. Quando si richiede un dato non viene spostato solo quello ma anche i dati contigui.

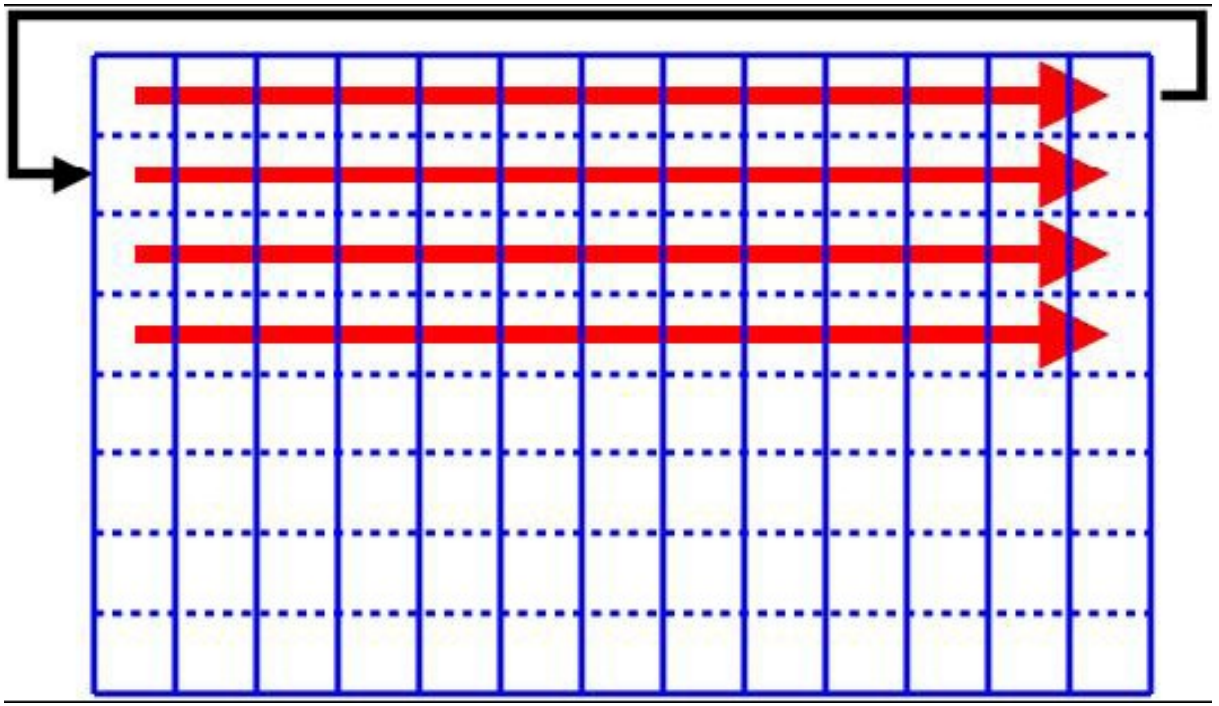
Cache



Array in C

La matrice A[i][j] di n*n elementi verra scritta in memoria, elemento dopo elemento, secondo l'indice piu' esterno. La matrice sara quindi memorizzata come una lunga la di elementi cosi' ordinati:

A[1][1],A[1][2],,...A[1][n],A[2][1],...A[n][n]



Array in C

Vediamo adesso mm.3:

```
$ ./mm.3
Tempo impiegato per inizializzare 0.050000 s.
Tempo per prodotto classico 6.860000 s.
Tempo totale 6.910000 s.

Mflops ----------------> 310.779110
Controllo -------------> 268364458.846206
```

Array in C

Vediamo dunque il perche' di tanta differenza, tra mm.1 ed mm.3:

```
$ diff mm.3.c mm.1.c
35,36c35,36
<      for (k=0; k<N; k++)
<        for (j=0; j<N; j++)
---
>      for (j=0; j<N; j++)
>        for (k=0; k<N; k++)
```

Pipeline

Oltre la cache. Si prenda in esame una generica operazione, come un operazione floating point: questa puo' essere divisa logicamente in fasi diffenti (allineamento matissa, somma, normalizzazione e arrotondamento) si considera completata solo dopo esser passata per tutti le fasi.

Dividendo una generica operazione in tre fasi, come ad esempio:

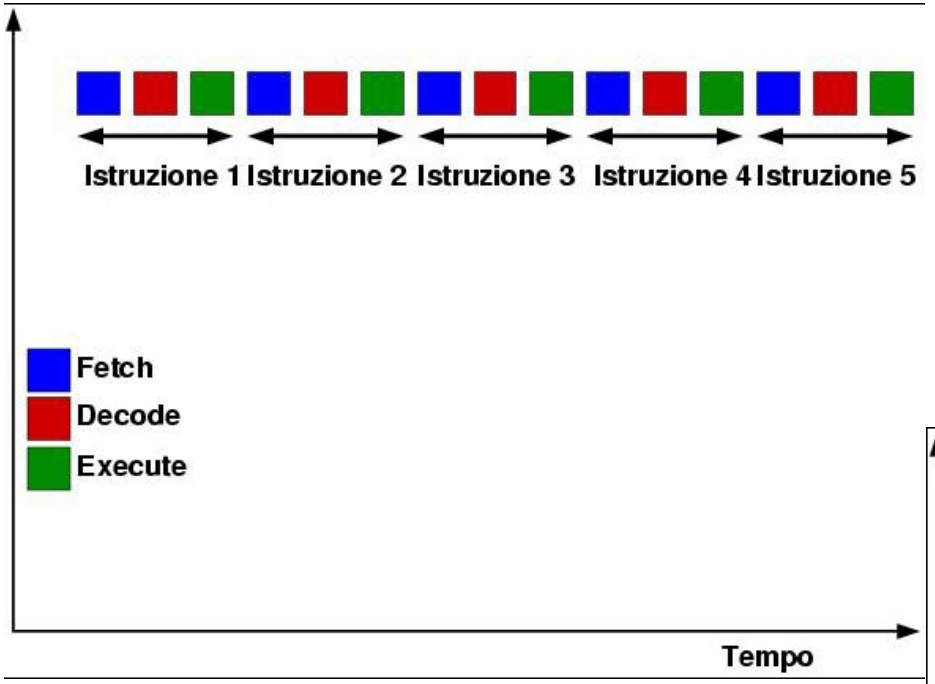
- Fetch: fase in cui si recupera il dato;
- Decode: fase in cui si decodificano le operazioni da fare;
- Execute: esecuzione dell'operazione.

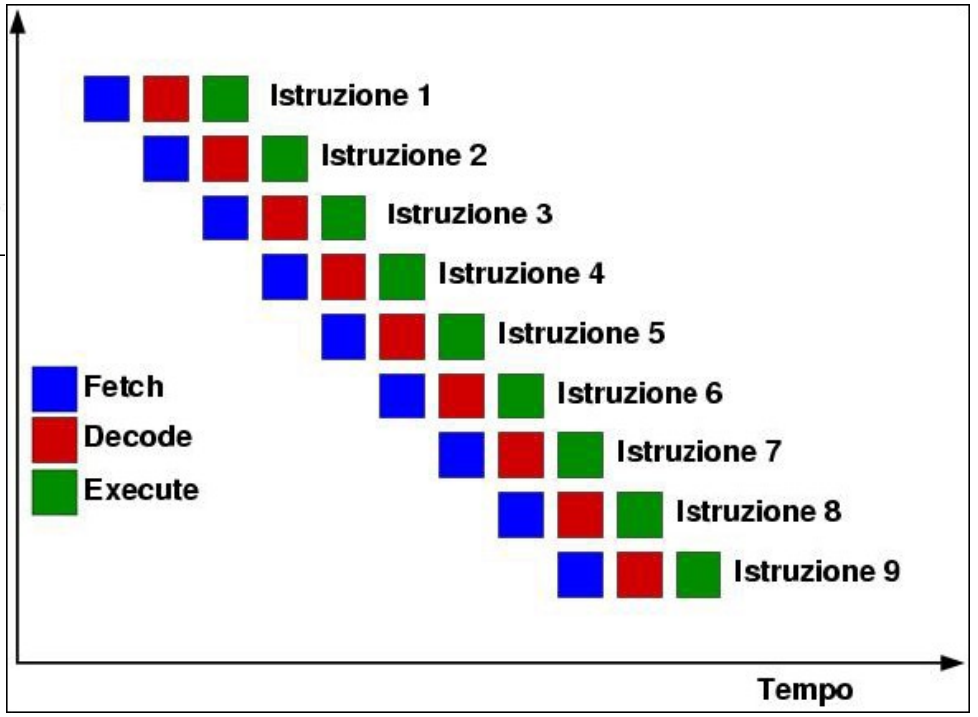
e nell'ipotesi che impieghino lo stesso tempo, si puo' classificare le CPU in base a come riescono a sfruttare le diffenti fasi.

Pipeline

- CPU sequenziale
-
- CPU pipelined: Facendo in modo che i tre passi che compongono l'operazione utilizzino parti diffenti ed indipendenti tra di loro del chip nello stesso ciclo si puo' fare il fetch di un'operazione, il decode di un'altra e l'execute di una terza senza che si diano fastidio l'un l'altra, purche' ovviamente non ci siano dipendenze tra queste tre istruzioni.In questo modo si ha un'architettura pipelined, che riesce a sovrapporre l'esecuzione di passi diffenti di istruzioni diffenti

Pipeline


Istruzione 1
Istruzione 2
Istruzione 3
Istruzione 4
Istruzione 5
Fetch
Decode
Execute
Tempo


Istruzione 1
Istruzione 2
Istruzione 3
Istruzione 4
Istruzione 5
Istruzione 6
Istruzione 7
Istruzione 8
Istruzione 9
Fetch
Decode
Execute
Tempo

Loop Unrolling

```
$ ./mm.4
Tempo impiegato per inizializzare 0.040000 s.
Tempo per prodotto classico 10.950000 s.
Tempo totale 10.990000 s.

Mflops ----------------> 195.403426
Controllo -------------> 268364458.846206
```

Loop Unrolling

```
$ diff mm.4.c  mm.1.c
12d11
<
35,37c34,36
<   for (i=0; i<N; i++) {
<     for (j=0; j<N; j +=8) {
<       for (k=0; k<N; k++) {
---
>   for (i=0; i<N; i++)
>     for (j=0; j<N; j++)
>       for (k=0; k<N; k++)
39,48d37
<         c[i][j+1] = c[i][j+1] + a[i][k] * b[k][j+1];
<         c[i][j+2] = c[i][j+2] + a[i][k] * b[k][j+2];
<         c[i][j+3] = c[i][j+3] + a[i][k] * b[k][j+3];
<         c[i][j+4] = c[i][j+4] + a[i][k] * b[k][j+4];
<         c[i][j+5] = c[i][j+5] + a[i][k] * b[k][j+5];
<         c[i][j+6] = c[i][j+6] + a[i][k] * b[k][j+6];
<         c[i][j+7] = c[i][j+7] + a[i][k] * b[k][j+7];
<       }
<     }
```

Cache and Loop Unrolling

```
$ ./mm.all
Tempo impiegato per inizializzare 0.050000 s.
Tempo per prodotto classico 5.630000 s.
Tempo totale 5.680000 s.

Mflops ----------------> 378.078107
Controllo -------------> 268364458.846206
```

Flags del compilatore

```
$ gcc  -O3 -mtune=core2 -o mm.5 mm.5.c

$ ./mm.5
Tempo impiegato per inizializzare 0.030000 s.
Tempo per prodotto classico 10.980000 s.
Tempo totale 11.010000 s.

Mflops ----------------> 195.048469
Controllo -------------> 268364458.846206
```

Non sempre usare il massimo dell'ottimizzazione disponibile incrementa le performances, non sempre il compilatore da solo riesce a risolvere i problemi relati ad un codice non scritto bene.

Padding

Allineamento dei dati in memoria. Altro argomento importante e' come i dati vengono allineati in memoria. I dati vengono spostati tra i vari livelli di memoria tramite il cosidetto bus di memoria, ovvero un collegamento caratterizzato da una frequenza che indica quanti pacchetti di dati vengono istradati per unit di tempo e da un'ampiezza che indica le dimensioni del pacchetto stesso inviato.

Il bus e' in realta' allineato con la memoria, ovvero pu muovere tutto il blocco di dati compresi tra l'indirizzo 0 e n-1, o quello tra l'indirizzo n e 2*n-1 e cosi' via, per bus di ampiezza di n bit e per leggere un dato compreso tra l'indirizzo n-4 e n+4 deve leggere 2 volte: prima il blocco compreso tra 0 e n-1 e poi tra n e 2n-1. Per cui se il dato non e' allineato, ovvero il suo indirizzo non e' un multiplo della sua dimensione (e.g. singola o doppia precisione) potrebbero essere necessarie piu' operazioni di lettura/scrittura. (cenni di data padding, usando ad esempio dell struct C).

Padding

```
#include <stdio.h>
#define DBGI(a) printf(#a " ==> %d\n", a)
int main ()
{
  struct {
    char a;
    int b;
    double c;
  } pad;

  DBGI(sizeof(char));
  DBGI(sizeof(int));
  DBGI(sizeof(double));
  DBGI(sizeof(pad));

  return 0;
}
```

Padding

```
$ gcc  -O0 -o padding padding.c

$ ./padding
sizeof(char) ==> 1
sizeof(int) ==> 4
sizeof(double) ==> 8
sizeof(pad) ==> 16
```