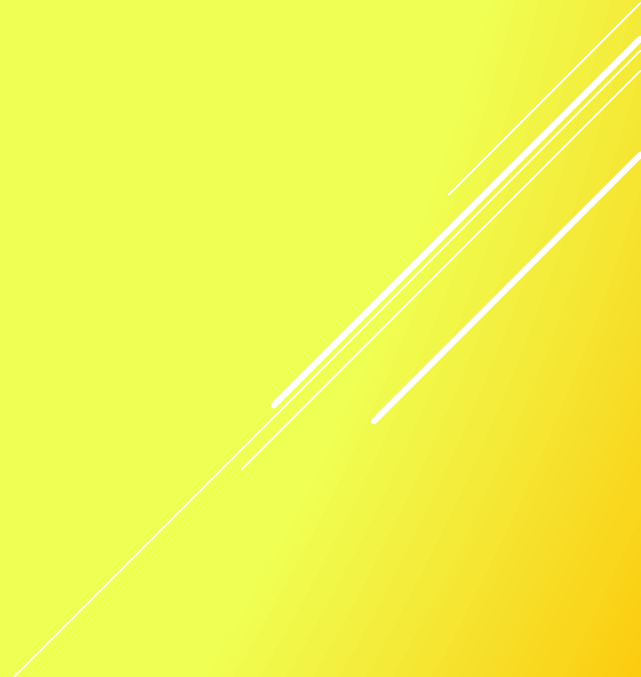


OTTIMIZZAZIONE E TECNICHE DI PARALLELIZZAZIONE

INTRODUZIONE

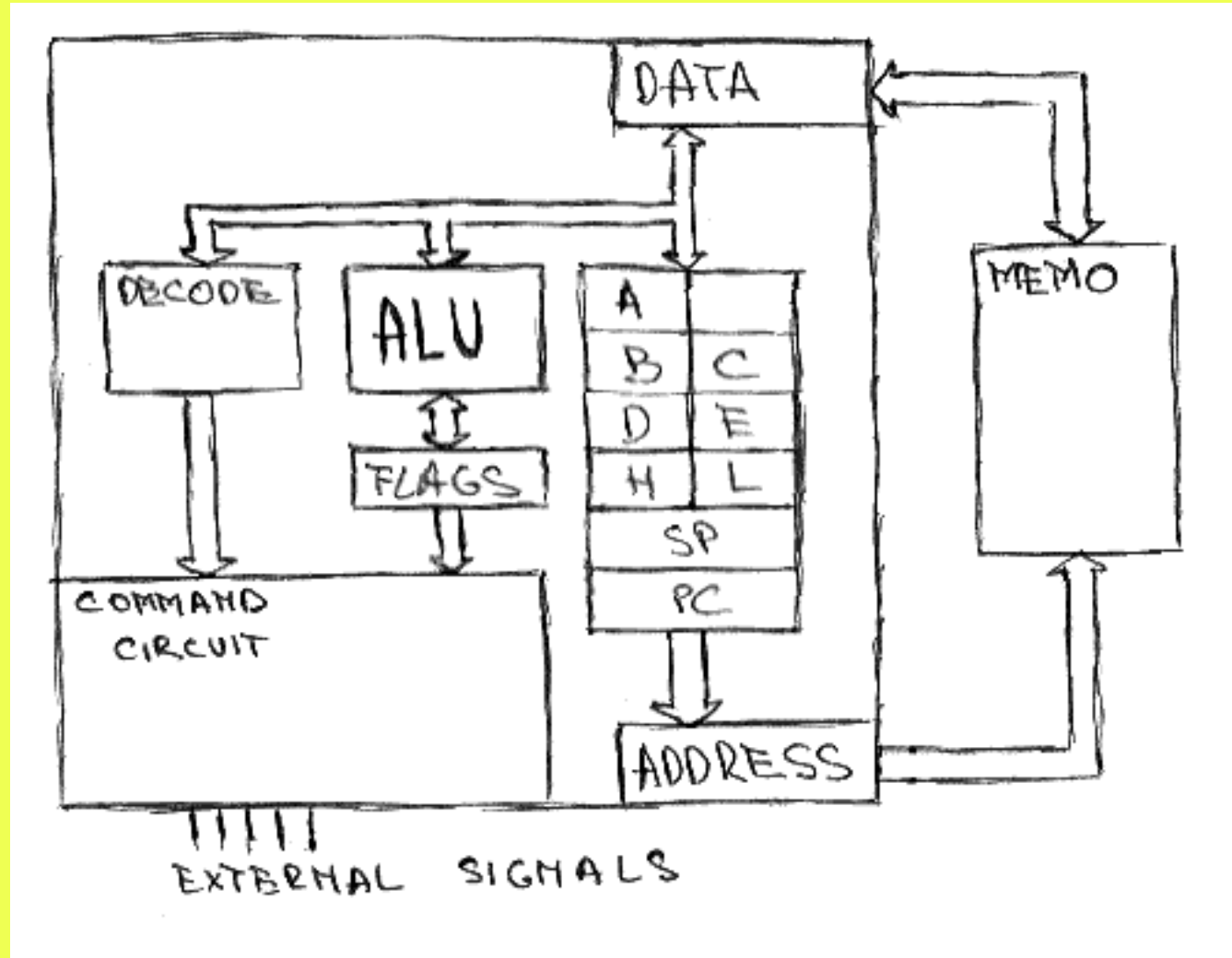
- ▶ Tecniche di calcolo efficienti
 - ▶ Vettorializzazione delle operazioni aritmetiche
 - ▶ Parallelizzazione automatica con OpenMP
- 

INTRODUZIONE

- ▶ *"Don't worry about performance, say 97% of the time. Premature Optimization is the Root of all Evil."*
- ▶ -- Donald Knuth, autore di "The Art Of Computer Programming" (1968)
- ▶ *"Always code for the programmer first and the computer second. If there is a performance difference, after the compiler has cast its expert eye over your code, AND you can measure it AND it matters - then you can change it."*

INTRODUZIONE

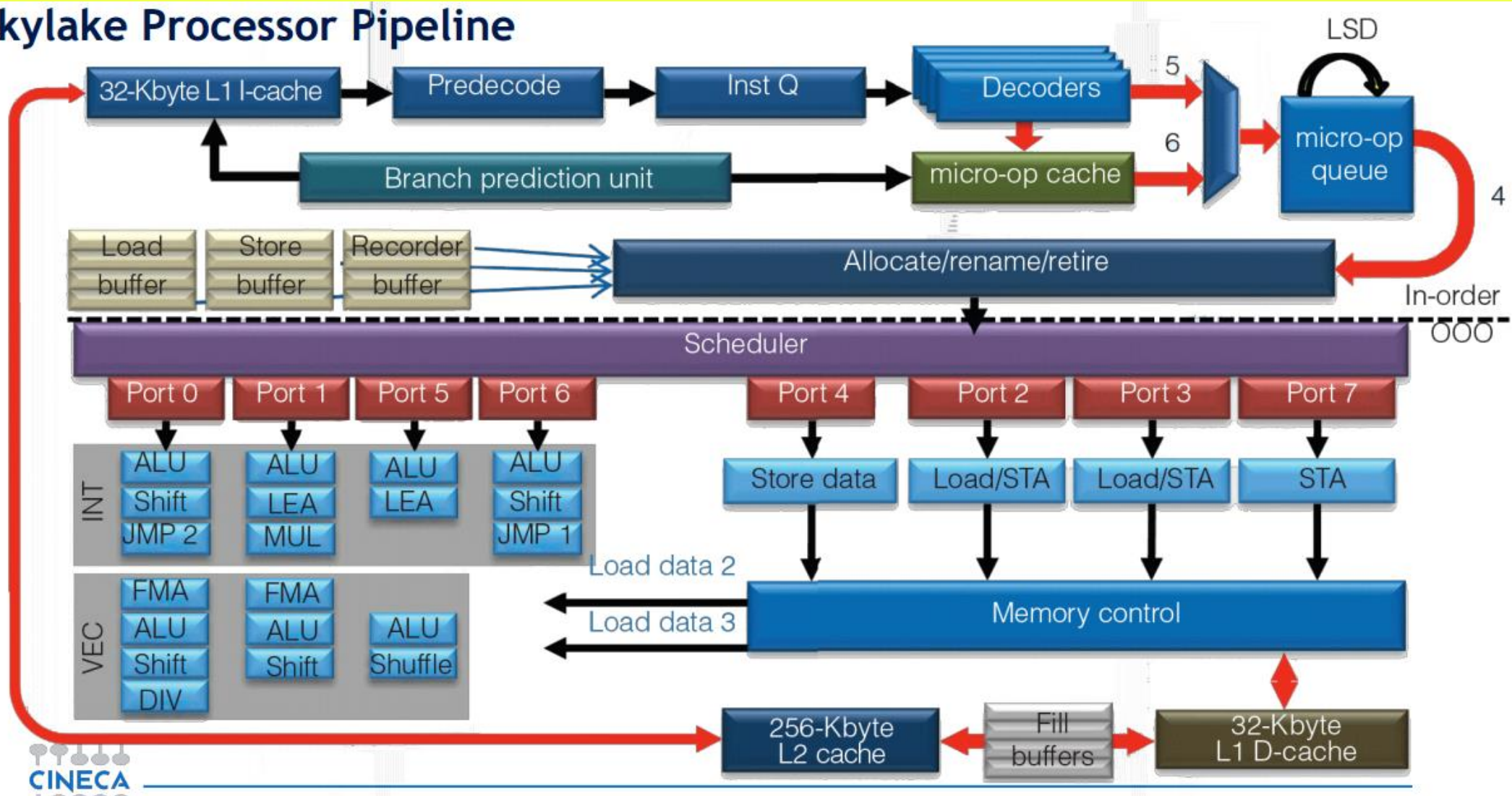
- Architettura modello di una CPU



INTRODUZIONE

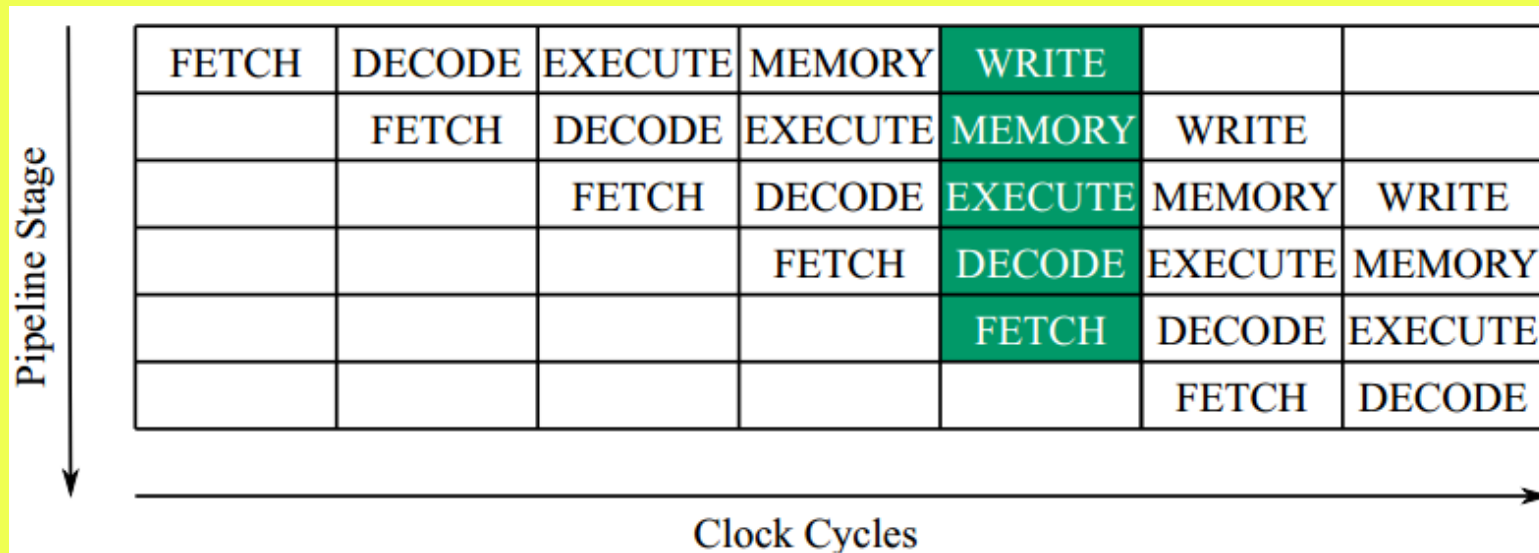
- Architettura di un core Intel Skylake

Skylake Processor Pipeline



BIT FIDDLING

- ▶ Evitare branching
- ▶ Pipelining: replicazione in hw di diversi stage di esecuzione, in modo da poter eseguire diversi stream di istruzioni contemporaneamente



- ▶ Branch può rompere la catena ed invalidare la pipeline.
- ▶ Risolto con meccanismi di branch prediction

BIT FIDDLING

- ▶ Evitare branching

- ▶ `int x;`
`bool a;`

- ▶ `if (x < 27)`
 `a = true;`
`else`
 `a = false;`

BIT FIDDLING

- ▶ Evitare branching

- ▶

```
int x;  
bool a;
```

- ▶

```
if (x < 27)  
    a = true;  
else  
    a = false;
```

- ▶

```
a = (x < 27);
```


BIT FIDDLELING

- ▶ `uint32_t v = 27;`
- ▶ `uint32_t f;`
- ▶ `f = (v & (v - 1)) == 0;`

BIT FIDDLING

```
▶ uint32_t v = 27;  
▶ uint32_t f;  
▶ f = (v & (v - 1)) == 0;
```

```
▶ 27 = 0b0001 1011  
   26 = 0b0001 1010
```

```
▶ &   = 0b0001 1010
```

BIT FIDDLING

- ▶ `uint32_t v = 32;`
- ▶ `uint32_t f;`
- ▶ `f = (v & (v - 1)) == 0;`
- ▶ `32 = 0b0010 0000`
`31 = 0b0001 1111`
- ▶ `& = 0b0000 0000`

BIT FIDDLING

- ▶ `uint32_t v = 128;`
- ▶ `uint32_t f;`
- ▶ `f = (v & (v - 1)) == 0;`
- ▶
128 = 0b1000 0000
127 = 0b0111 1111
- ▶ `& = 0b0000 0000`
- ▶ L'espressione rileva se v è una potenza di 2
- ▶ <https://graphics.stanford.edu/~seander/bithacks.html>

BIT FIDDLING

- ▶ Da Quake III: Arena:
- ▶ `float Q_rsqrt(float number) {`
- ▶ `long i;`
- ▶ `float x2, y;`
- ▶ `const float threehalfs = 1.5F;`
- ▶ `x2 = number * 0.5F;`
- ▶ `y = number;`
- ▶ `i = * (long *) &y; // evil floating point bit level hacking`
- ▶ `i = 0x5f3759df - (i >> 1); // what the fuck?`
- ▶ `y = * (float *) &i;`
- ▶ `y = y * (threehalfs - (x2 * y * y)); // 1st iteration`
- ▶ `// y = y * (threehalfs - (x2 * y * y)); // 2nd iteration, this can be removed`
- ▶ `return y;`
- ▶ `}`

BIT FIDDLING

- ▶ Come funziona?
- ▶ https://en.wikipedia.org/wiki/Fast_inverse_square_root
- ▶ Molto più complesso di quanto sembri.
- ▶ Attribuito erroneamente a John Carmack; trucco conosciuto già da metà anni '90

OTTIMIZZAZIONE DEL CODICE

- ▶ Ottimizzazione manuale? Di norma è meglio lasciare al compilatore l'onere della "micro-ottimizzazione".
- ▶ X es: divisioni e moltiplicazioni per potenze di 2 vengono convertite in automatico in operazioni con bitshift a destra o sinistra
- ▶ Concentrarsi su ottimizzazione algoritmica e delle strutture dati
- ▶ Tuttavia, alcune pratiche sono più corrette di altre:
- ▶ https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Code_optimization
- ▶ <https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf>

OTTIMIZZAZIONE DEL CODICE

- ▶ Postfix vs. prefix increment
- ▶ Differenza di prestazioni tra ++i e i++ in un ciclo?
`for(int i = 0; i < 100; i++)` vs. `for(int i = 0; i < 100; ++i)`
- ▶ Entrambe le espressioni equivalgono a $i = i + 1$, ma gli operatori sono implementati in maniera diversa, perché devono comportarsi diversamente
- ▶ Sono operatori, quindi vere e proprie funzioni: hanno un parametro in ingresso (in questo caso, i) e un return value
- ▶ i++ : valuta i, lo incrementa di 1 e restituisce il vecchio valore prima dell'incremento
- ▶ ++i : valuta i, lo incrementa di 1 e restituisce il nuovo valore incrementato

OTTIMIZZAZIONE DEL CODICE

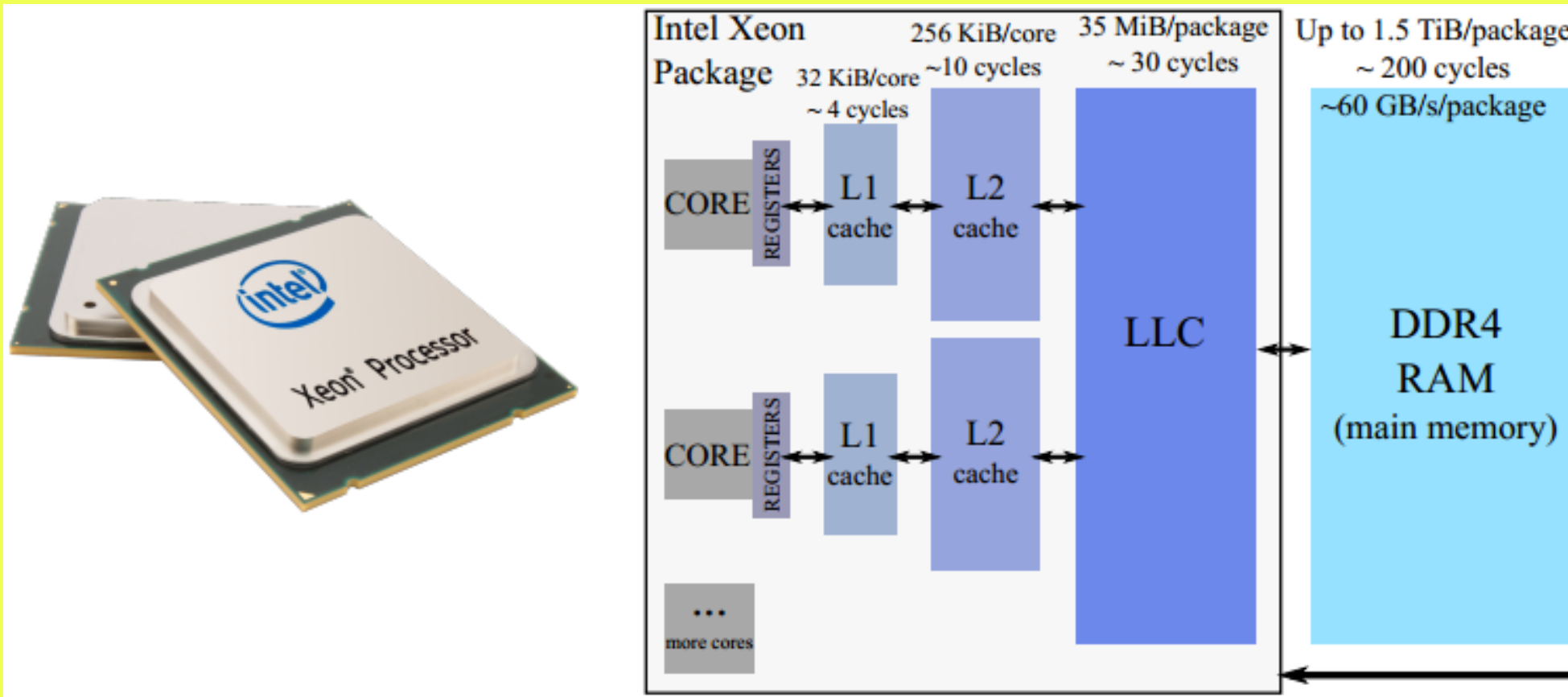
- ▶ Due implementazioni equivalenti possono essere queste:
- ▶

```
int postfix(int & operand) {  
    int tempOperand(operand);  
    operand = operand + 1;  
    return tempOperand;  
}
```
- ▶

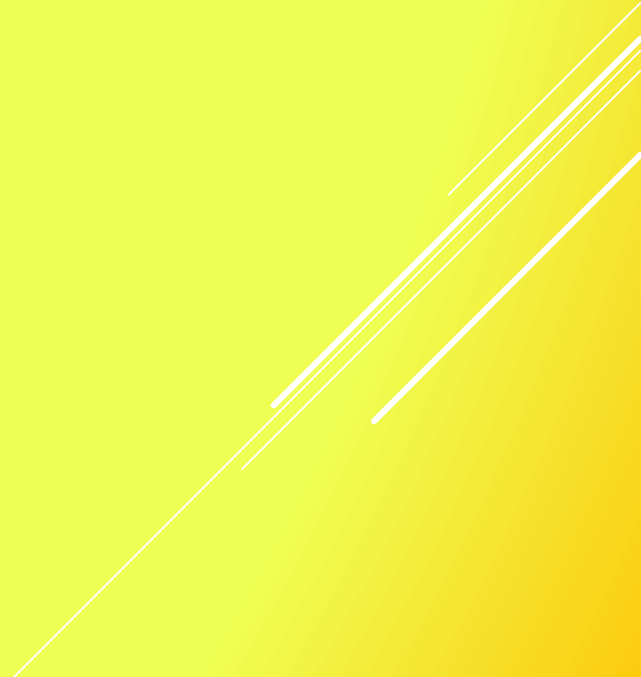
```
int prefix(int & operand) {  
    operand = operand + 1;  
    return operand;  
}
```
- ▶ Su tipi base, le prestazioni sono pressoché equivalenti, ma su oggetti complessi, come gli iteratori?

OTTIMIZZAZIONE DEL CODICE

- ▶ Altre pratiche di ottimizzazione in disuso perché inefficienti: look-up table
- ▶ Ora il collo di bottiglia in HPC è l'accesso ai dati in memoria RAM, mentre la computazione è "gratis"...



OTTIMIZZAZIONE DEL CODICE

- ▶ Possibili aree di ottimizzazione del codice:
 - ▶ - Ottimizzazione scalare (codice compier-friendly)
 - ▶ - Vettorializzazione automatica
 - ▶ - Pattern di accesso alla memoria
 - ▶ - Multi-thread
- 

OPZIONI DI COMPILAZIONE

- ▶ -O<n>: livello di ottimizzazione

Default optimization level -O2

- ▶ optimization for speed
- ▶ automatic vectorization
- ▶ inlining
- ▶ constant propagation
- ▶ dead-code elimination
- ▶ loop unrolling

Optimization level -O3

- ▶ aggressive optimization
- ▶ loop fusion
- ▶ block-unroll-and-jam
- ▶ if-statement collapse
- ▶ *may or may not be better than -O2*

- ▶ Per gcc: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

OPZIONI DI COMPILAZIONE

- ▶ -O<n>: livello di ottimizzazione
- ▶ -g: generazione del codice di debug
- ▶ Intel icc:
 - ▶ -fp-model <model>: tipologia di istruzioni floating point (strict, precise, fast1, fast2)
 - ▶ -x <code>: target architecture
- ▶ Gcc / G++
 - ▶ -m<code>: target architecture

OTTIMIZZAZIONE SCALARE

- Consiste nella scrittura di codice che non sia troppo convoluto, in modo che il compilatore riesca ad ottimizzarlo efficientemente, unite a pratiche di buona programmazione

Common Subexpression Elimination.

```
1 for (int i = 0; i < n; i++) {  
2     A[i] /= B;  
3 }
```

```
1 const float Br = 1.0f/B;  
2 for (int i = 0; i < n; i++)  
3     A[i] *= Br;
```

Replace division with multiplication.

```
1 for (int i = 0; i < n; i++) {  
2     P[i] = (Q[i]/R[i])/S[i];  
3 }
```

```
1 for (int i = 0; i < n; i++) {  
2     P[i] = Q[i]/(R[i]*S[i]);  
3 }
```

Use functions with Hardware support.

```
1 double r = pow(r2, -0.5);  
2 double v = exp(x);  
3 double y = y0*exp(log(x/x0)*  
4               log(y1/y0)/log(x1/x0));
```

```
1 double r = 1.0/sqrt(r2);  
2 double v = exp2(x*1.44269504089);  
3 double y = y0*exp2(log2(x/x0)*  
4               log2(y1/y0)/log2(x1/x0));
```

OTTIMIZZAZIONE SCALARE

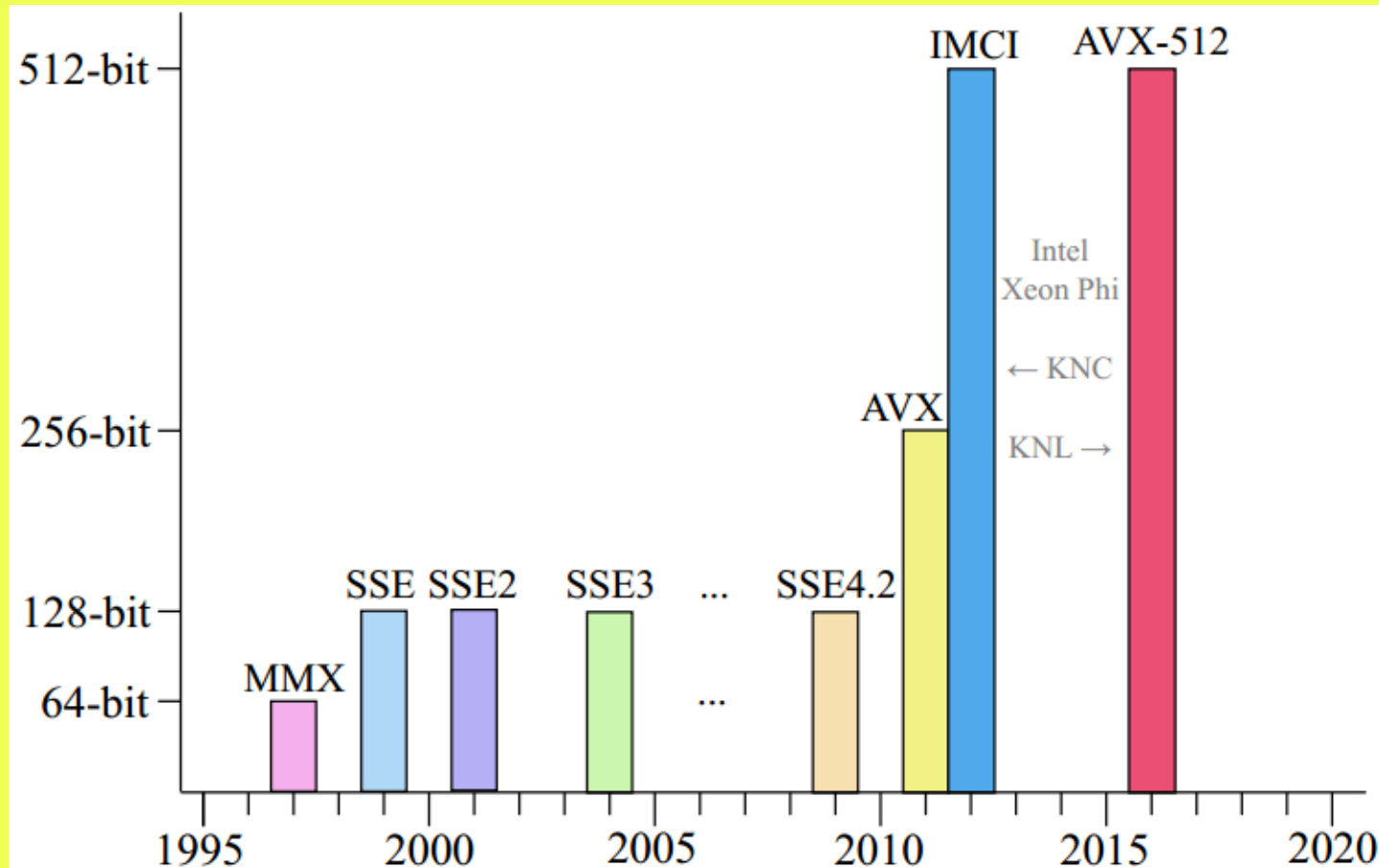
- Consiste nella scrittura di codice che non sia troppo convoluto, in modo che il compilatore riesca ad ottimizzarlo efficientemente, unite a pratiche di buona programmazione

```
1 // Elegant, but bad for performance
2 for (i = 0; i < n; i++) {
3     if (i == 0) {
4         // Absorbing boundary
5         B[i] = 0.0;
6     } else if (i == n - 1) {
7         // Injection at boundary
8         B[i] = A[i] + 1.0;
9     } else {
10        // Diffusion between boundaries
11        B[i] = 0.25*(A[i-1] +
12                    2.0*A[i] + A[i+1]);
13    }
14 }
```

```
1 // Moving branches out of loops
2
3
4 // Absorbing boundary
5 B[i] = 0.0;
6
7 for (i = 1; i < n - 1; i++) {
8     // Diffusion between boundaries
9     B[i] = 0.25*(A[i-1] + 2.0*A[i] +
10                 A[i+1]);
11 }
12
13 // Injection at boundary
14 B[n-1] = A[n-1] + 1.0;
```

VETTORIALIZAZIONE

- ▶ Come sono fatti i registri delle attuali CPU?
- ▶ Contengono più di un singolo `int32_t`, in particolare...



VETTORIALIZZAZIONE

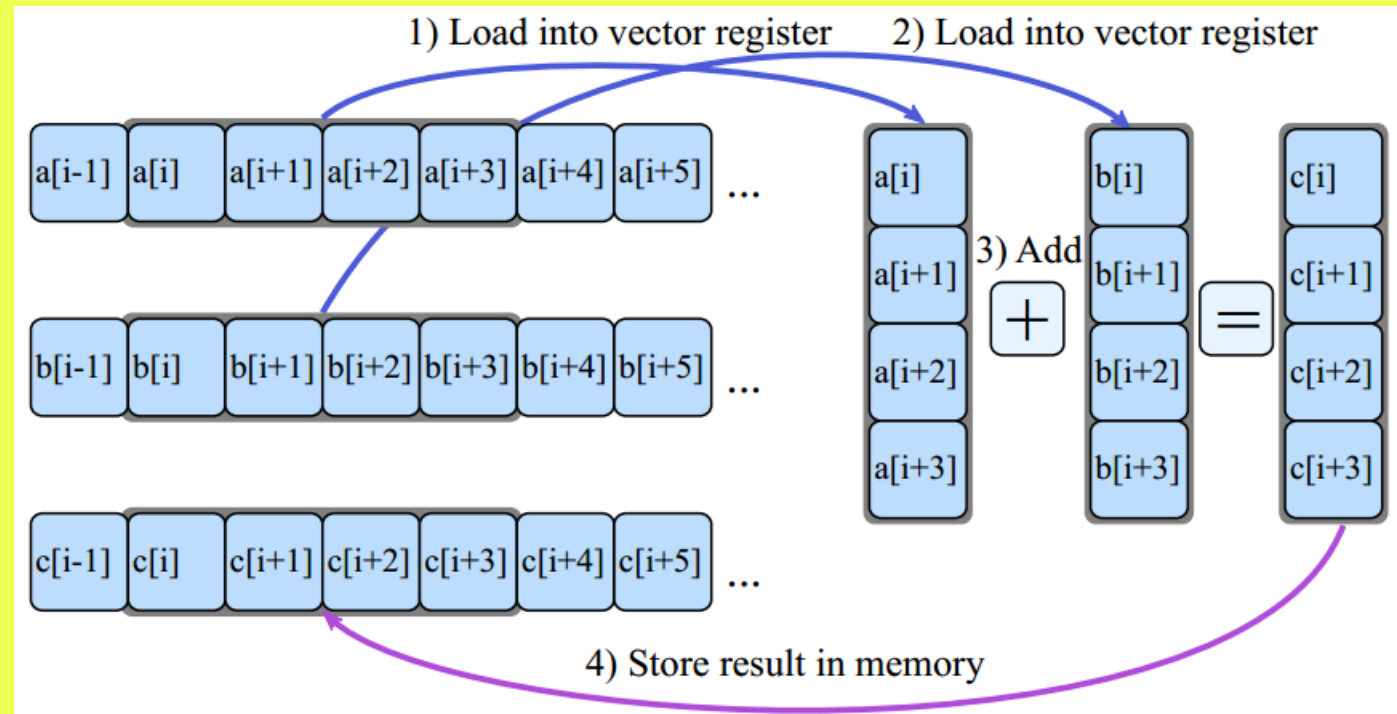
- ▶ Per sfruttare al meglio questi registri, bisogna usare il set di istruzioni per i vettori
- ▶ Compilatori mettono a disposizione macro intrinsec che richiamano istruzioni vettoriali per una determinata architettura
- ▶ Problema: non portabile!

Scalar Instructions

$$\begin{array}{rcl} 4 & + & 1 = 5 \\ 0 & + & 3 = 3 \\ -2 & + & 8 = 6 \\ 9 & + & -7 = 2 \end{array}$$

Vector Instructions

$$\begin{array}{rcl} 4 & & 1 & & 5 \\ 0 & + & 3 & = & 3 \\ -2 & & 8 & = & 6 \\ 9 & & -7 & = & 2 \end{array} \quad \begin{array}{c} \uparrow \\ \text{Vector Length} \\ \downarrow \end{array}$$



VETTORIALIZIZZAZIONE

- Es: calcolo dell'integrale

$$I(a, b) = \int_0^a \frac{1}{\sqrt{x}} dx$$

Rectangle method:

$$\begin{aligned}\Delta x &= \frac{a}{n}, \\ x_i &= (i+1)\Delta x, \\ I(a, b) &= \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x).\end{aligned}$$

```
1 float Integrate(const float a,  
2                 const int N) {  
3     const float dx = a/float(n);  
4     float S = 0.0f;  
5     for (int i = 0; i < n; i++) {  
6         const float xi = dx*float(i+1);  
7         S += 1.0f/sqrtf(xi) * dx;  
8     }  
9     return S;  
10 }
```

```
1 float Integrate(const float a, const int n) {  
2     __m128 dx = _mm_set1_ps(a/float(n));  
3     __m128 S = _mm_set1_ps(0.0f);  
4     for (int i = 0; i < n; i += 4) {  
5         __m128i ip1 =  
6             _mm_set_epi32(i+4, i+3, i+2, i+1);  
7         __m128 ip1f = _mm_cvtepi32_ps(ip1);  
8         __m128 xi = _mm_mul_ps(dx, ip1f);  
9         __m128 fi = _mm_rsqrt_ps(xi);  
10        __m128 dS = _mm_mul_ps(fi, dx);  
11        S = _mm_add_ps(S, dS);  
12    }  
13    ConverterType c;  
14    c.v = S;  
15    return c.f[0] + c.f[1] + c.f[2] + c.f[3];  
16 }
```

VETTORIALIZAZIONE

- Soluzione: sotto opportune condizioni, i compilatori vettorializzano automaticamente le operazioni aritmetiche

```
1 #include <stdio>
2
3 int main(){
4     const int n=8;
5     int i;
6     int A[n] __attribute__((aligned(64)));
7     int B[n] __attribute__((aligned(64)));
8
9     // Initialization
10    for (i=0; i<n; i++)
11        A[i]=B[i]=i;
12
13    // This loop will be auto-vectorized
14    for (i=0; i<n; i++)
15        A[i]+=B[i];
16
17    // Output
18    for (i=0; i<n; i++)
19        printf("%2d %2d %2d\n", i, A[i], B[i]);
20 }
```

```
vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(14,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(14,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(14,3)]
LOOP END
...
vega@lyra% ./a.out
0 0 0
1 2 1
2 4 2
3 6 3
4 8 4
5 10 5
6 12 6
7 14 7
```

VETTORIALIZZAZIONE

- ▶ Anche GCC supporta vettorializzazione automatica, specificando l'architettura:

GCC \geq 4.9.1 supports AVX-512 instruction set.

```
user@knl% g++ -v
gcc version 4.9.2 (GCC)
user@knl% g++ foo.cc -mavx512f -mavx512er -mavx512cd -mavx512pf
```

Basic automatic vectorization support: add -O3.

```
// ... foo.cc ... //
for(int i = 0; i < n; i++)
    B[i] = A[i] + B[i];
```

- ▶ Scrivere codice "pulito", affinché il compilatore rilevi automaticamente le parti da vettorializzare – evitare le dipendenze tra i dati

```
1 float *a, *b;
2 for (int i = 1; i < n; i++)
3     a[i] += b[i]*a[i-1]; // dependence on the previous element
```

- ▶ E' possibile forzare la vettorializzazione tramite la direttiva `#pragma simd`

VETTORIALIZAZIONE

- E' possibile forzare la vettorializzazione tramite la direttiva `#pragma ivdep`

Assumed vector dependence: when compiler cannot determine whether vector dependence exists, auto-vectorization fails:

```
1 void mycopy(int n,  
2             float* a, float* b) {  
3     for (int i = 0; i < n; i++)  
4         a[i] = b[i];  
5 }
```

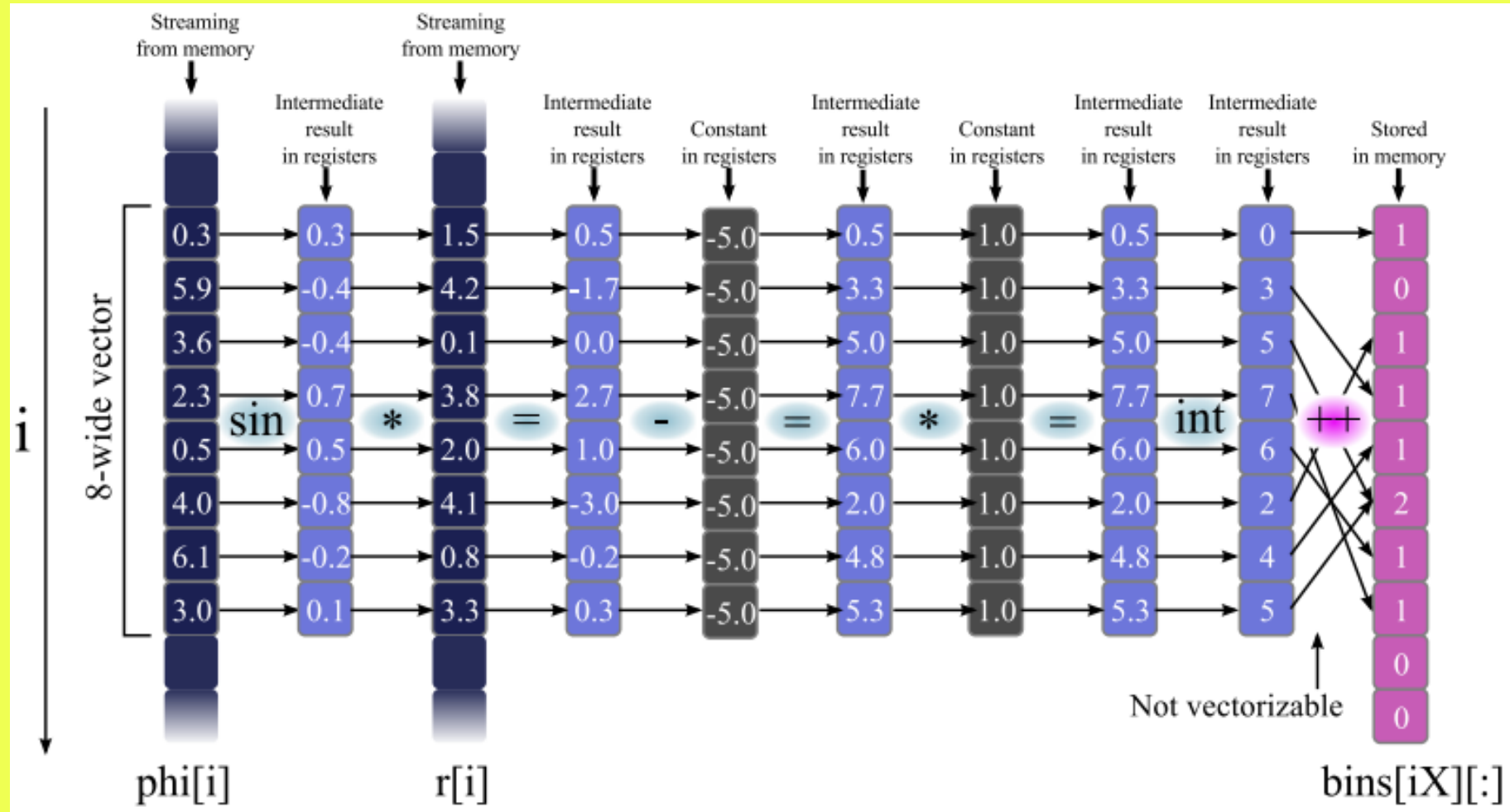
```
vega@lyra% icpc -c vdep.cc -qopt-report \  
> -qopt-report-phase:vec  
vega@lyra% cat vdep.optrpt  
...  
remark #15304: loop was not  
vectorized: non-vectorizable loop  
instance from multiversioning  
...
```

```
1 void mycopy(int n,  
2             float* a, float* b) {  
3     #pragma ivdep  
4     for (int i = 0; i < n; i++)  
5         a[i] = b[i];  
6 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \  
> -qopt-report-phase:vec  
vega@lyra% cat vdep.optrpt  
...  
LOOP BEGIN at vdep.cc(4,1)  
<Multiversioned v2>  
remark #15300: LOOP WAS VECTORIZED  
LOOP END
```

PATTERN DI ACCESSO ALLA MEMORIA

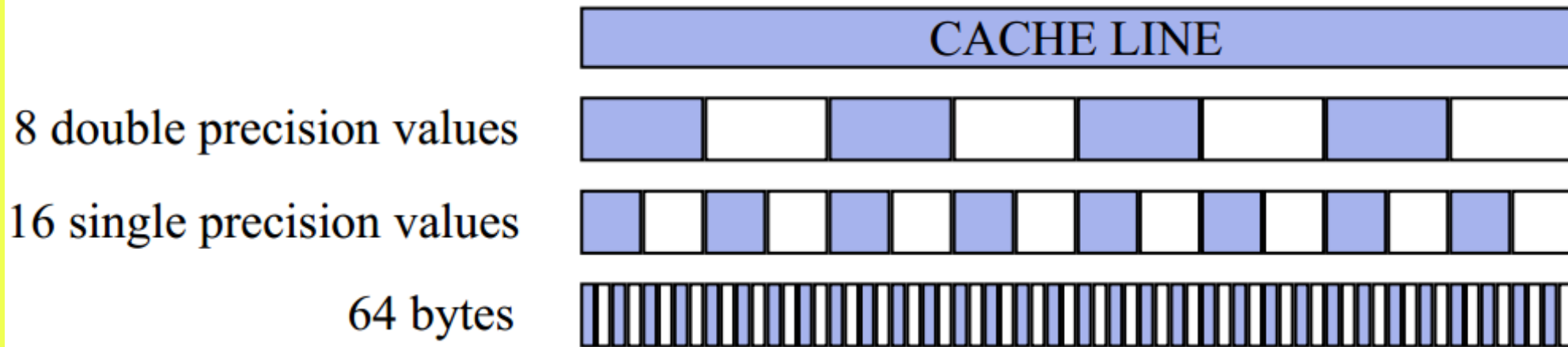
- Vettorializzazione può essere anche molto complessa:



PATTERN DI ACCESSO ALLA MEMORIA

- ▶ Come vengono prelevati i dati dalla RAM? Cache line

- ▶ Minimal block of data transferred between memory and cache
- ▶ 64 bytes long in Intel Architecture
- ▶ Aligned on 64-byte boundaries in memory



- ▶ CPU controlla che il dato richiesto sia presente in una delle cache; in caso contrario, parte il prelevamento dalla RAM
- ▶ Dati prelevati in blocchi di 64 byte e copiati nella cache L3, poi propagati nella L2 e L1

PATTERN DI ACCESSO ALLA MEMORIA

- ▶ Per minimizzare il numero di accessi alla memoria, i dati devono essere allineati
- ▶ "Array of structs" vs "struct of arrays"

```
▶ const int n = 10000;  
  struct {  
    double a;  
    double b;  
    double c;  
  } particle[n];
```

```
▶ for (int i = 0; i < n; ++i) {  
    particle[i].a = particle[i].b + particle[i].c;  
}
```


PATTERN DI ACCESSO ALLA MEMORIA

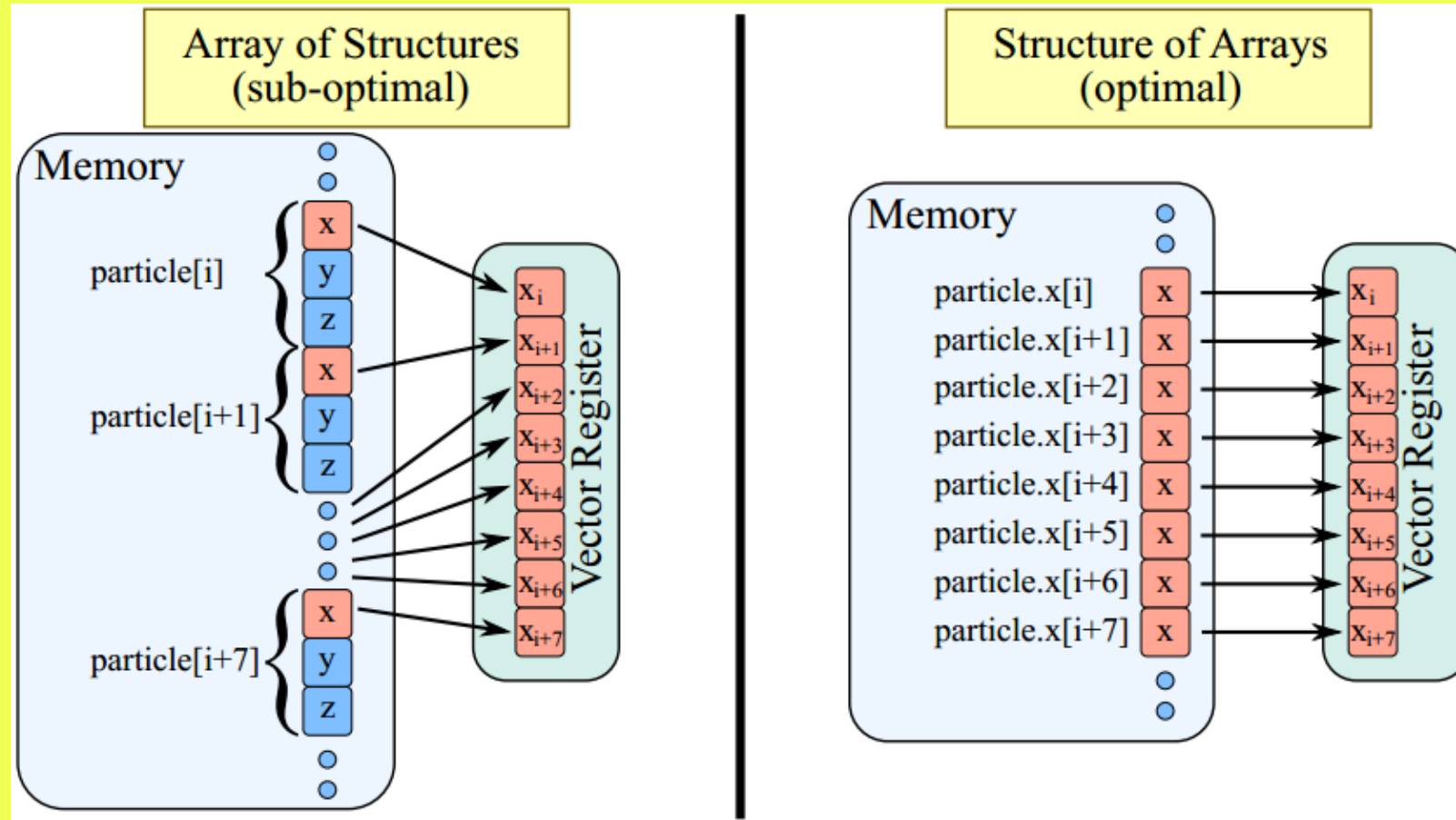
- ▶ Per minimizzare il numero di accessi alla memoria, i dati devono essere allineati
- ▶ "Array of structs" vs "struct of arrays"

```
▶ const int n = 10000;  
  struct {  
    double a[n];  
    double b[n];  
    double c[n];  
  } particle;
```

```
▶ for (int i = 0; i < n; ++i) {  
    particle.a[i] = particle.b[i] + particle.c[i];  
}
```

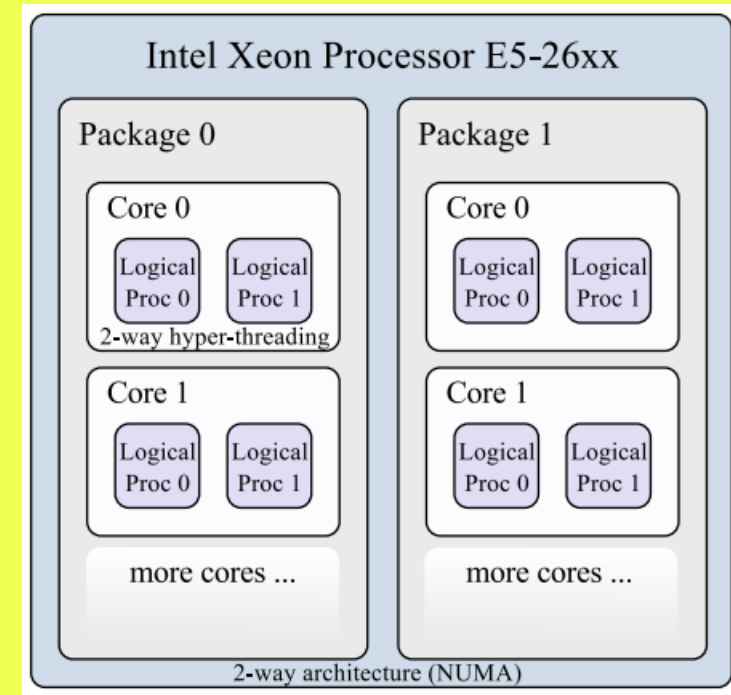
PATTERN DI ACCESSO ALLA MEMORIA

- Per minimizzare il numero di accessi alla memoria, i dati devono essere allineati



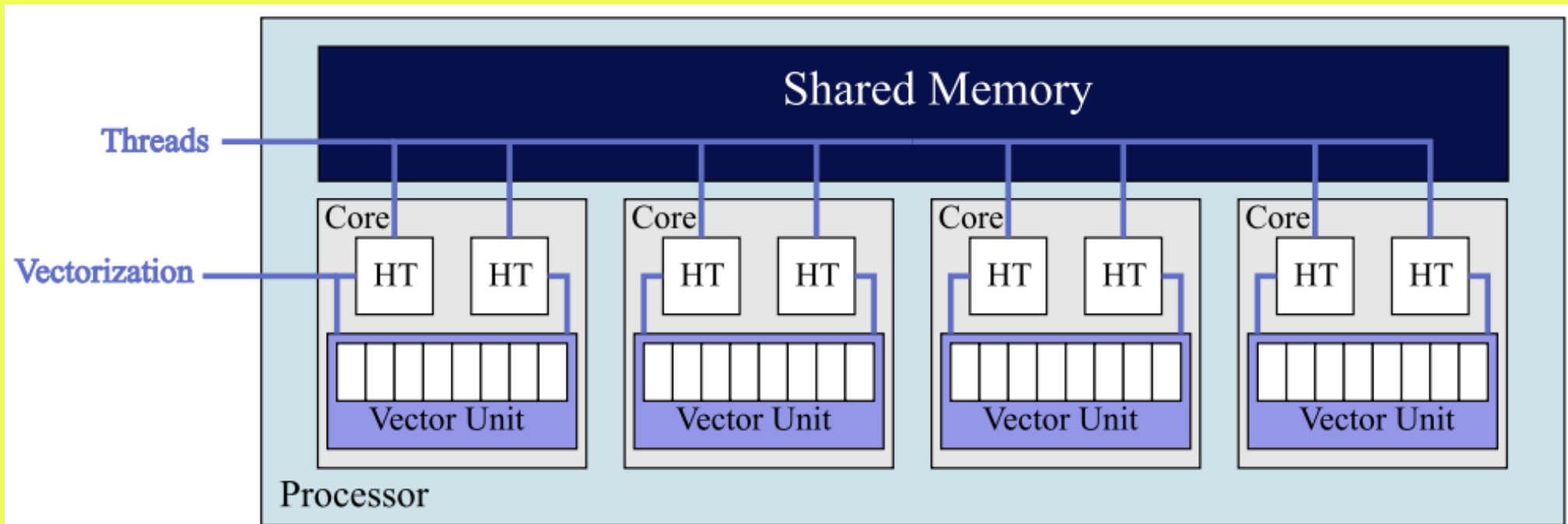
MULTITHREADING

- ▶ Processori attuali sono multicore
- ▶ Intel Xeon Platinum può essere contenere fino a 28 core
- ▶ Acceleratori Intel Xeon Phi fino a 72
- ▶ Adesso stiamo lavorando su macchine con 68 core * 4 hyperthread



MULTITHREADING

- Espressione del parallelismo dei task



Utilize cores: run multiple threads/processes (MIMD)

Utilize vectors: each thread (process) issues vector instructions (SIMD)

MULTITHREADING

- ▶ Espressione del parallelismo dei task attraverso i thread
- ▶ Thread: stream di istruzioni che condividono uno spazio di memoria comune
- ▶ Per ottenere speed-up: distribuisco l'esecuzione dei thread su più core della macchina, in modo che vengano eseguiti in parallelo
- ▶ In generale, thread possono eseguire task di natura molto diversa, pur collaborando all'esecuzione dello stesso programma
- ▶ Programmazione multithread è complicata: problemi di sincronizzazione, dead lock, race condition...
- ▶ Fino a C++11, non era standardizzato nel linguaggio (pthread su Linux, Threads su Windows...)

MULTITHREADING

- ▶ Espressione del parallelismo dei task attraverso i thread
- ▶ OpenMP (Open Multi Processing): computing-oriented framework for shared-memory programming
- ▶ E' un insieme di librerie (API) e di feature del compilatore: introduce un sistema di notazione per programmi sequenziali che specifica come i task debbano essere distribuiti tra i thread e come questi vengano eseguiti. Inoltre specifica un insieme di regole per l'accesso alla memoria condivisa
- ▶ Basato su direttive che dicono al compilatore come parallelizzare il codice sequenziale
- ▶ `#pragma omp ...`

MULTITHREADING

► Hello world OpenMP:

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5      // This code is executed by only 1 thread
6      const int nt=omp_get_max_threads();
7      printf("OpenMP with %d threads\n", nt);
8
9      #pragma omp parallel
10     {
11         // This code is executed in parallel
12         // by multiple threads
13         printf("Hello World from thread %d\n",
14               omp_get_thread_num());
15     }
16 }
```

```
vega@lyra% icpc -qopenmp hello_omp.cc
vega@lyra% export OMP_NUM_THREADS=5
vega@lyra% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
```

MULTITHREADING

- ▶ Un area da parallelizzare è introdotta da `#pragma omp parallel`
- ▶ Altre opzioni della direttiva suggeriscono al compilatore come deve avvenire la parallelizzazione, nonché quali variabili debbano essere condivise dai thread
- ▶

```
int A, B;  
#pragma omp parallel private(A) shared(B)  
{  
    //Ogni thread possiede una copia locale di A, ma B è condivisa  
}
```
- ▶

```
int B;  
#pragma omp parallel shared(B)  
{  
    int A; // dichiarata nello scope parallelo: sempre privata  
}
```


MULTITHREADING

- ▶ Il massimo della potenzialità di OpenMP viene espresso con la direttiva `#pragma omp parallel for`
- ▶ Questa direttiva dice al compilatore di parallelizzare automaticamente (quando possibile) i cicli `for`. Le iterazioni vengono distribuite sui thread disponibili
- ▶

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    std::cout << "L'iterazione " << i << " è eseguita dal thread " \
    << omp_get_thread_num() << std::endl;
}
```

MULTITHREADING

- ▶ Modificare un programma sequenziale diventa "abbastanza facile". Ad es:

```
▶ int A[1000];  
   int B[1000];  
   int somma[1000];
```

```
<... inizializzazione degli array...>
```

```
for (int i = 0; i < 1000; i++) {  
    somma[i] = A[i] + B[i];  
}
```

MULTITHREADING

- ▶ Modificare un programma sequenziale diventa "abbastanza facile". Ad es:

```
▶ int A[1000];  
   int B[1000];  
   int somma[1000];
```

```
<... inizializzazione degli array...>
```

```
#pragma omp parallel for  
for (int i = 0; i < 1000; i++) {  
    somma[i] = A[i] + B[i]; // Somma distribuita nei thread  
}
```

MULTITHREADING

- ▶ OpenMP fornisce dei costrutti molto semplici per la parallelizzazione...

```
#pragma omp parallel – create threads  
#pragma omp for – process loop with threads  
#pragma omp task/taskyield – asynchronous tasks  
#pragma omp critical/atomic – mutexes  
#pragma omp barrier/taskwait – synchronization points  
#pragma omp sections/single – blocks of code for individual threads  
OMP_* – environment variables, omp_*( ) – functions
```

MULTITHREADING

- ▶ ...ma non rimuove tutti i problemi legati al mondo della programmazione multithread, in particolare:
 - ▶ - Race condition
 - ▶ - Deadlock
 - ▶ - Gestione delle sezioni critiche
 - ▶ - Gestione della condivisione della memoria
- ▶ sono questioni che devono essere affrontate dal programmatore in fase di progettazione e implementazione.
- ▶ Strumenti di profilazione ed analisi del codice aiutano a trovare bug e inconsistenze nel codice parallelo (ad es. Valgrind su Linux)

SPEED-UP

- Usando tutte le tecniche presentate...

