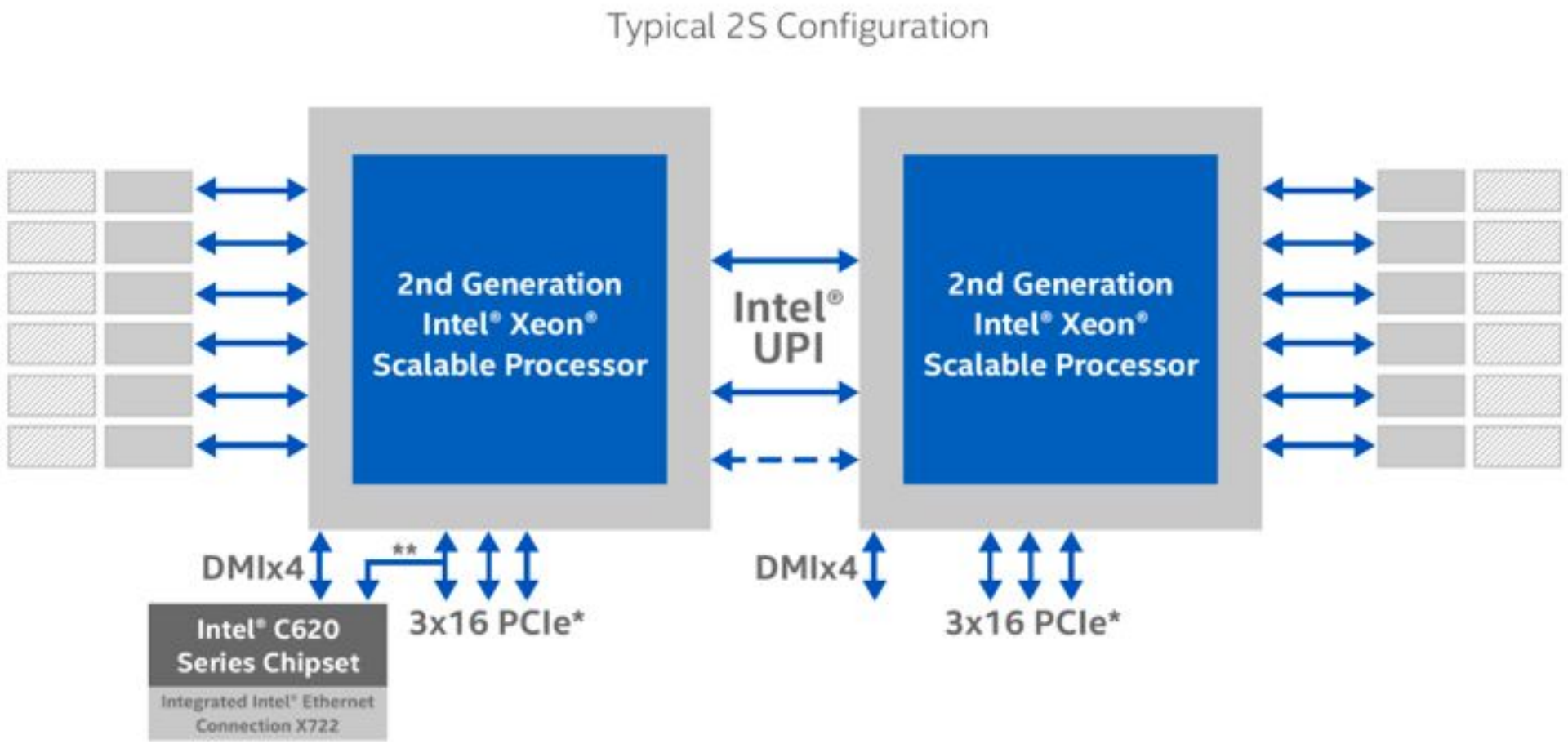
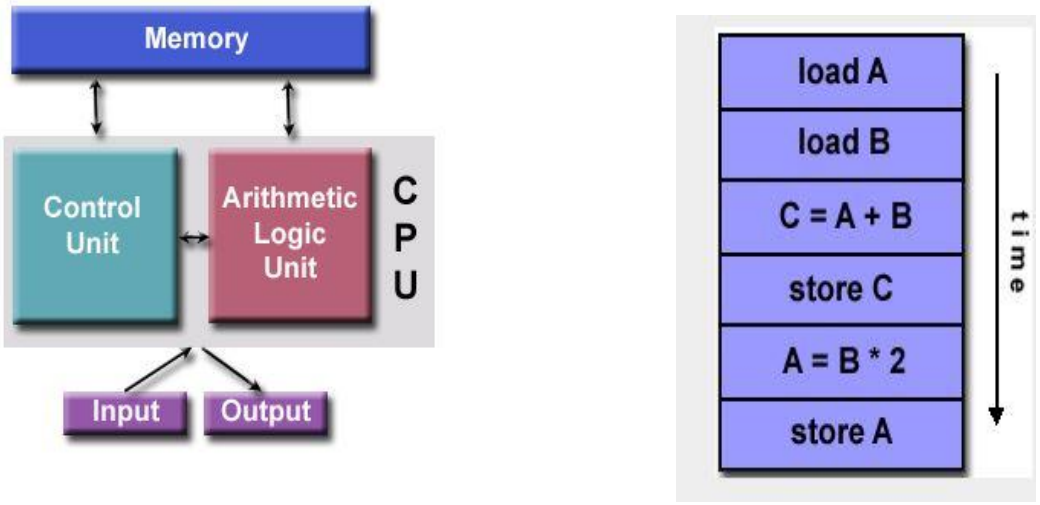
I - Características da Arquitetura de um Processador Moderno (exp: Cascade lake)



1- Arquitetura de um processador moderno continua sendo igual a que foi apresentada por "Von Neuman": onde o código fica armazenado na memória e as variáveis necessitam ser transferidas de/para memória de/para CPU.

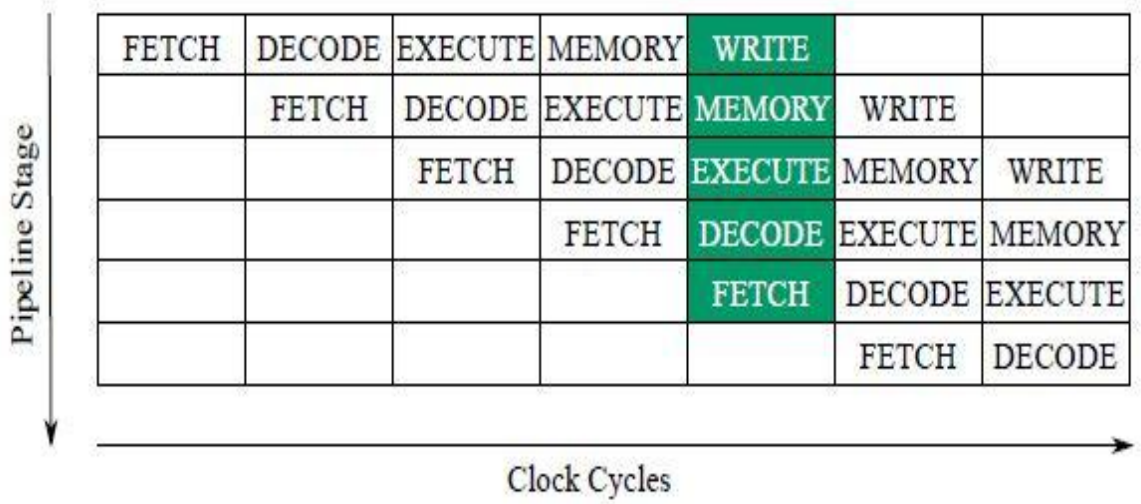


Arquitetura Von Neuman 1945
(Processador Sequencial)

2- As CPU's dos processadores modernos implementam "paralelismo pipeline" para aumentar a "vazão"de operações executadas por "clock"

INSTRUCTION-LEVEL PARALLELISM (ILP) WALL: PIPELINING

Pipelining – replication of hardware to run different stages of different instruction streams at the same time



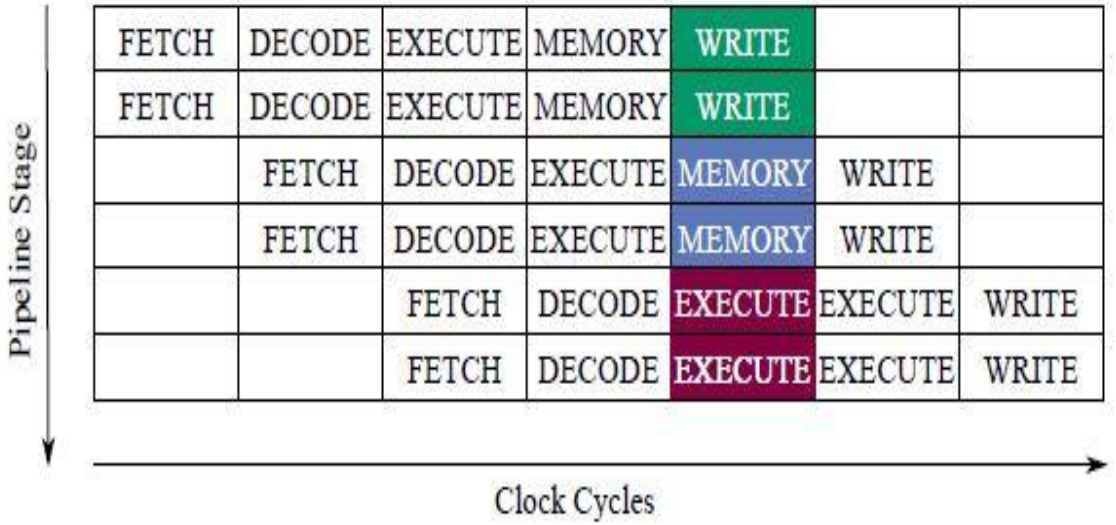
FETCH	DECODE	EXECUTE	MEMORY	WRITE		
	FETCH	DECODE	EXECUTE	MEMORY	WRITE	
		FETCH	DECODE	EXECUTE	MEMORY	WRITE
			FETCH	DECODE	EXECUTE	MEMORY
				FETCH	DECODE	EXECUTE
					FETCH	DECODE

Only so many pipeline stages, possible conflicts

3- As CPU's dos processadores modernos implementam "paralelismo superescalar" dentro do "paralelismo pipeline" para permitir que diversas instruções independentes sejam executadas em paralelo dentro da CPU.

INSTRUCTION-LEVEL PARALLELISM (ILP) WALL: SUPERSCALAR EXECUTION

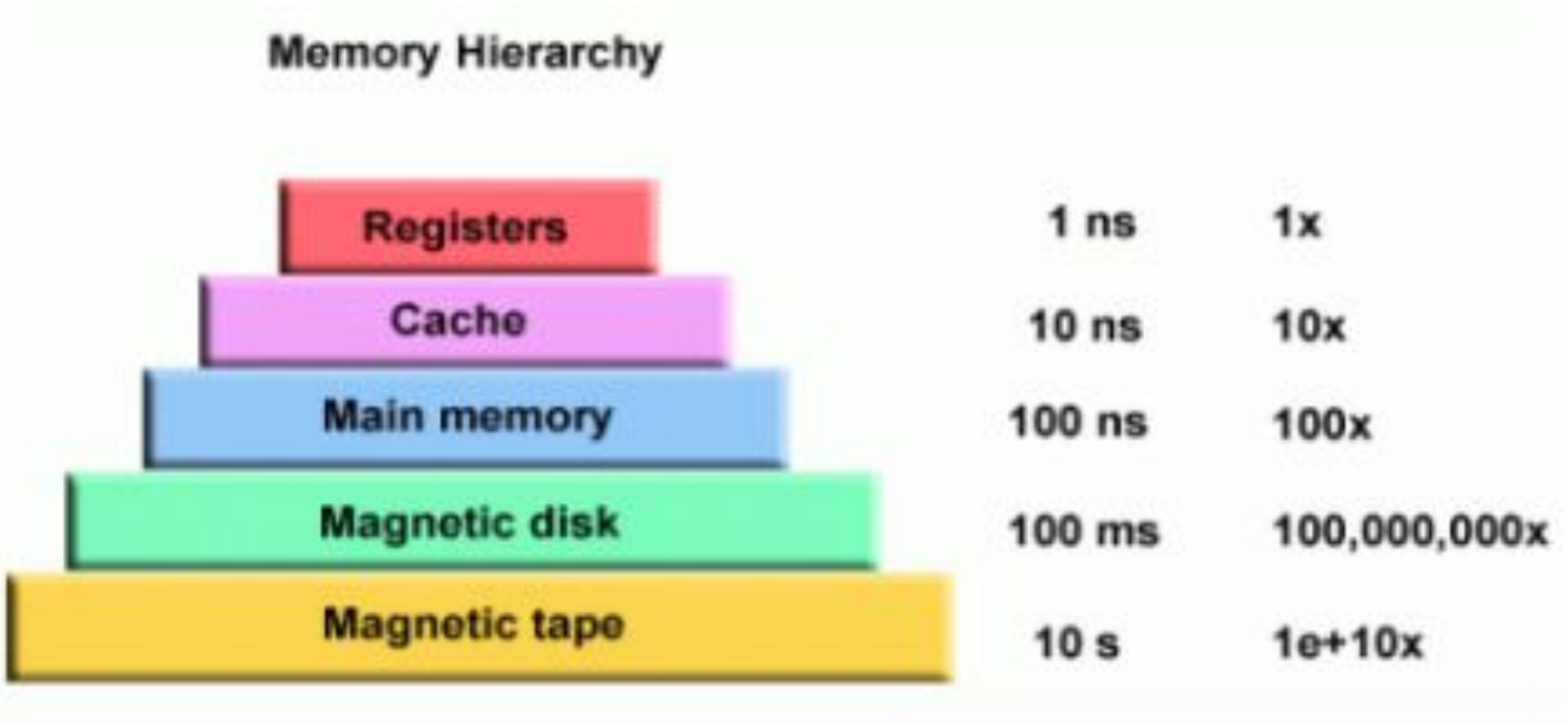
Superscalar Execution – hardware checks for independence of operations, pipelines multiple instructions in a cycle



Automatic search for independent instructions reqiures extra resources

4- Os processadores modernos implementam “hierarquia de memória” para movimentar os dados entre a CPU e a memória com “baixo custo”, utilizando algoritmos com “localidade espacial e localidade temporal.

Tempo de acesso ao dado conforme a tecnologia : "tempo de acesso menor tem custo maior”

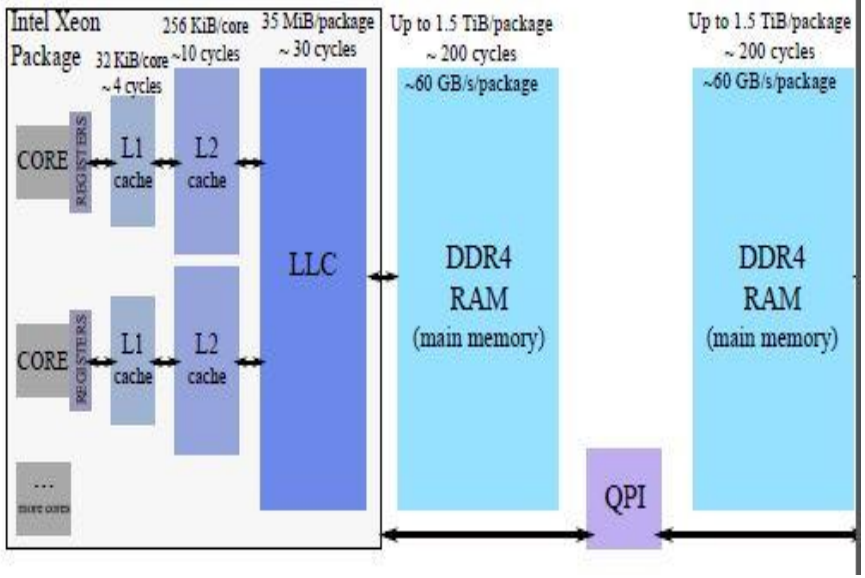


5- Os processadores modernos possuem um grande número de unidades de processamento, chamados de cores.

Hierarquia de memória em um processador multicore: cada core possui cache própria L1 e L2 e a cache LLC é compartilhada por todos os cores.

INTEL XEON CPU: MEMORY ORGANIZATION

- Hierarchical cache structure
- Two-way processors have NUMA architecture

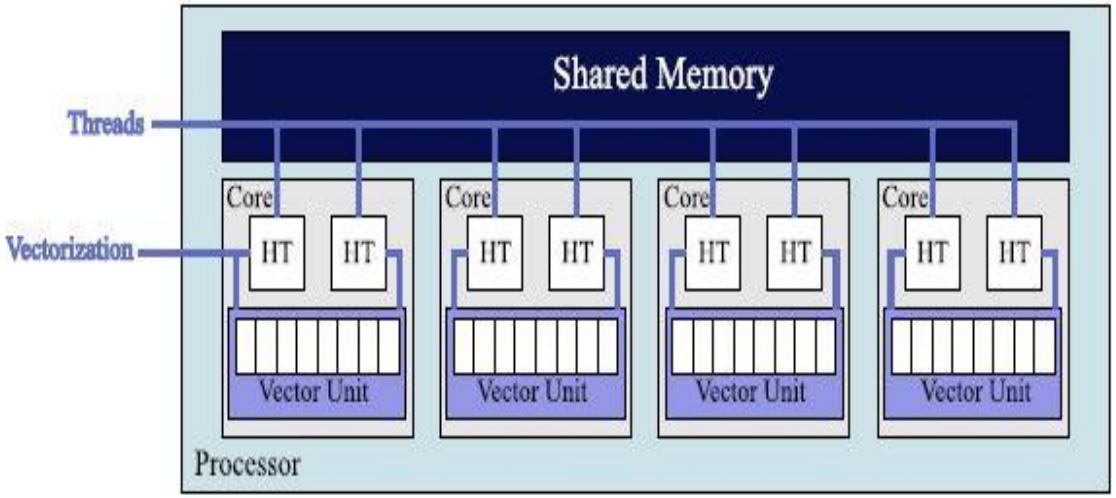


6- Os processadores modernos possuem cores e cada core possui instruções vetoriais".

PARALLELISM

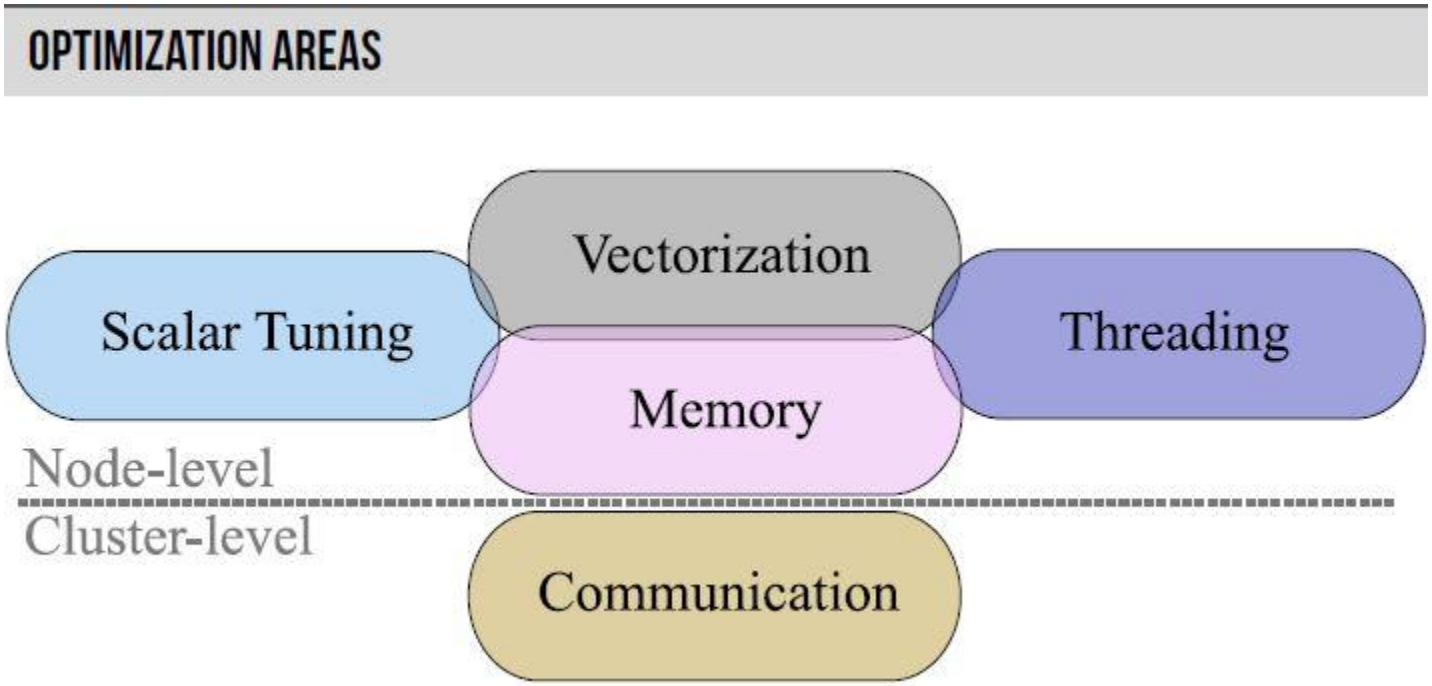
CORES – multiple instructions on multiple data elements (MIMD)

VECTORS – single instruction on multiple data elements (SIMD)



Unbounded growth opportunity, but **not automatic**

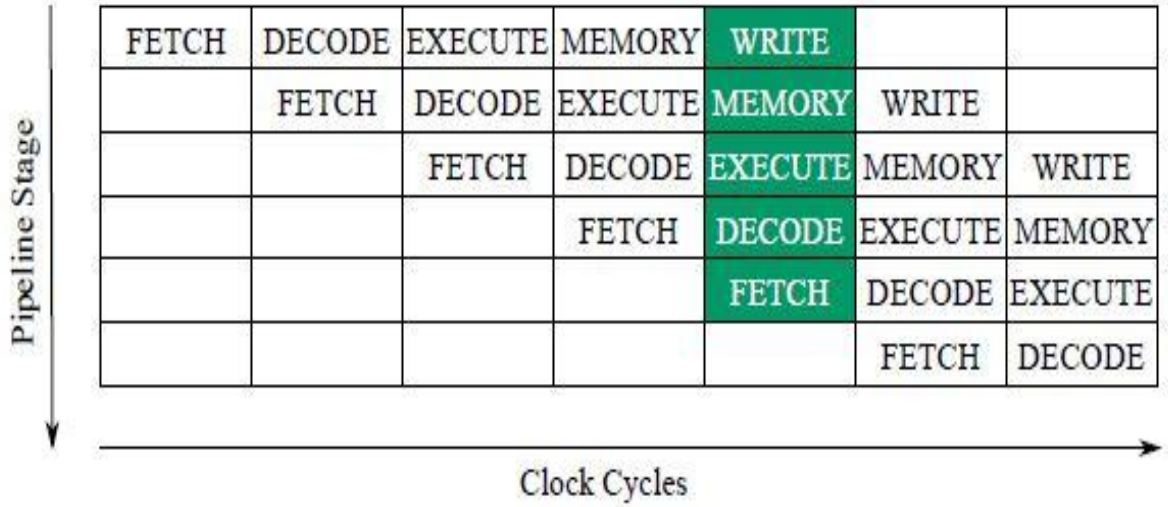
II- Podemos generalizar as otimizações que podem ser implementadas em um processador moderno nos 5 grupos abaixo:



II.1- Scalar Tunning: Problemas relacionados com o “paralelismo pipeline” :

INSTRUCTION-LEVEL PARALLELISM (ILP) WALL: PIPELINING

Pipelining – replication of hardware to run different stages of different instruction streams at the same time



FETCH	DECODE	EXECUTE	MEMORY	WRITE		
	FETCH	DECODE	EXECUTE	MEMORY	WRITE	
		FETCH	DECODE	EXECUTE	MEMORY	WRITE
			FETCH	DECODE	EXECUTE	MEMORY
				FETCH	DECODE	EXECUTE
					FETCH	DECODE

Only so many pipeline stages, possible conflicts

Branch prediction deve ser evitado dentro de um loop com operações vetoriais

REDUNDANT CODE IS OK

```
1  // Elegant, but bad for performance
2  for (ii = 0; ii < n; ii+=16) {
3    for (i = ii; i < ii+16; i++)
4      // Branch causes unnecessary
5      // masking of vector iterations
6      if (i < n) {
7        A[k*n + i] = ...
8      }
9  }
```

```
1  // Redundant code, but faster
2  const int nTrunc = n - n%16;
3  for (ii = 0; ii < nTrunc; ii+=16) {
4    for (i = ii; i < ii+16; i++)
5        A[k*n + i] = ...
6
7  for (i = nTrunc; i < n; i++)
8    A[k*n + i] = ...
9  }
```

```c
#include <stdio.h>
#include <time.h>

#define N 2000000


int main(){
int i, j, vet[N], k ;

k=1/N;


    for(j=0; j<N; j+=16 ){
        for(i=j; i< j+16; i++)
            if(i < N){
                vet[k*N+i]=1;
            }
    }

    for(j=0; j<N; j+=16 ){
        for(i=j; i< j+16; i++)
            if(i < N){
                vet[k*N+i] ++;
            }
    }


printf("\n FIM.... \n");

}
```

```c
#include <stdio.h>
#include <time.h>

#define N 2000000

int main(){
int i, j, vet[N], k ;


k=1/N;

const int nTrunc= N - N%16;



    for(j=0; j<nTrunc; j+=16 ){
        for(i=j; i< j+16; i++)
            vet[k*N+i] = 1;
    }
    for(i = nTrunc; i<N; i++)
        vet[k*N+i] = 1;

    for(j=0; j<nTrunc; j+=16 ){
        for(i=j; i< j+16; i++)
            vet[k*N+i]++;
    }
    for(i = nTrunc; i<N; i++)
        vet[k*N+i] ++;

printf("\n FIM.... \n");

}
```

CLOCK	*Com 'IF'*	*Sem 'IF'*
n=200	0.003 ms	0.006 ms
n=20.000	0.271 ms	0.241 ms
n= 2.000.000	15.506 ms	9.066 ms

```c
#define N 2000000

int main(){
int i, j, vet[N], k ;

k=1/N;


        for(j=0; j<N; j+=16 ){
                for(i=j; i< j+16; i++)
                        if(i < N){
                                vet[k*N+i] = 1;
                        }
        }
```

```c
#define N 2000000

int main(){
int i, j, vet[N], k ;


k=1/N;

const int nTrunc= N - N%16;


        for(j=0; j<nTrunc; j+=16 ){
                for(i=j; i< j+16; i++)
                        vet[k*N+i] = 1;
        }
        for(i = nTrunc; i<N; i++)
                vet[k*N+i] = 1;
```

Figura: Implementação com if x sem if

Utilizar operações que apresentam melhor desempenho: baixa latência e alta vazão no pipeline

PERFORMANCE OF VECTOR INSTRUCTIONS IN KNL

All values in cycles. Lower is better.

Instruction	Latency	1/Throughput
Most vector math and FMA	6	0.5
64-bit exp2a23, rcp28 and rsqrt28	7	2
32-bit exp2a23, rcp28 and rsqrt28	8	3
Floating-point division and sqrt	38	10
Simple integer math	2	2
32-bit scalar division	25	20
64-bit scalar division	40	30
Type conversion (same width)	2	1
Type conversion (different widths)	6	5

Utilizar instruções equivalentes que possuem melhor desempenho e instruções implementadas em hardware:

Common Subexpression Elimination.

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

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

Replace division with multiplication.

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

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

Use functions with Hardware support.

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

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

Problemas de precisão que podem ocorrer ao utilizar “Operações de Ponto Flutuante”:

Formato de representação digital de números reais usada nos computadores

Ao falar em números reais a visualização vinda à cabeça é:

Parte Inteira	Ponto ou Vírgula	Parte Fracionária

No entanto, essa representação custa caro, em termos de processamento e armazenamento ao computador havendo a necessidade de utilizar uma outra maneira que favoreça tais tarefas. Para trabalhar com a parte fracionária de forma satisfatória, usa-se a representação por pontos flutuantes.

Essa representação baseia-se no deslocamento da virgula de forma que se obtenha um número menor ou próximo de 1. Esse deslocamento é feito por meio de notação científica. Esclarecendo: o número 25,456 em notação corresponde ao $0{,}25456 \times 10^2$.

O exemplo acima tinha como base a decimal, no entanto o computador trabalha com a base 2 (binários – 0 e 1). Então um número binário 11,011 em notação corresponde ao $0{,}11011 \times 2^2$. Esse processo de transcrever um número em notação científica recebe o nome de normalização, portanto $0{,}11011 \times 2^2$ está normalizado.

De forma geral, representa-se um ponto flutuante da seguinte forma:

$$+\text{-}\ M \times B^{+\text{-}e}$$

Onde:

- M é a mantissa (parte fracionária)
- B é a base
- e é o expoente

Representação IEEE

- As mais diversas representações de ponto flutuante já foram propostas, mas …
- O padrão IEEE 754 atualmente é o mais utilizado:
  - Criado em 1985 como padrão para representação e aritmética em ponto flutuante
  - Implementado na grande maioria das CPUs

- Define três precisões:
  - Single precision (`float`) 32 bits (precisão 24 bits)
  - Double precision (`double`) 64 bits (precisão 53 bits)
  - Double extended precision 80 bits (precisão 63 bits)

  Obs: esta última somente em arquiteturas Intel-like

Padrão IEEE 754

- Forma numérica

$$(-1)^s \, M \, 2^E$$

  - Bit de sinal *s* determina se número é negativo ou positivo
  - Mantissa *M* é um valor fracionário no intervalo [1.0,2.0), na representação normalizada.
  - Expoente *E*
- Codificação

  - bit mais significativo é s
  - Campo `exp` codifica *E*
  - Campo `frac` codifica *M*

Precisão simples ocupa 32 bits e Precisão dupla ocupa 64 bits

IEEE 754: Precisões de Ponto Flutuante

- Tamanhos
  - **`float`**: `exp` = 8 bits, `frac` = 23 bits, `s` = 1 bit
    - Total: 32 bits
    - Faixa de valores: 2^{-126} até 2^{127}
  - **`double`**: `exp` =11 bits, `frac` = 52 bits, `s` = 1 bit
    - Total: 64 bits
    - Faixa de valores: 2^{-1022} até 2^{1023}
  - Precisão estendida: `exp` =15 bits, `frac` = 63 bits,`s` = 1 bit
    - Total: 80 bits
    - Faixa de valores: 2^{-16382} até 2^{16383}
    - 1 bit é desperdiçado

O resultado pode ser afetado quando a operação não utiliza a mesma precisão em ambos os lados:

CONSISTENCY OF PRECISION: CONSTANTS

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

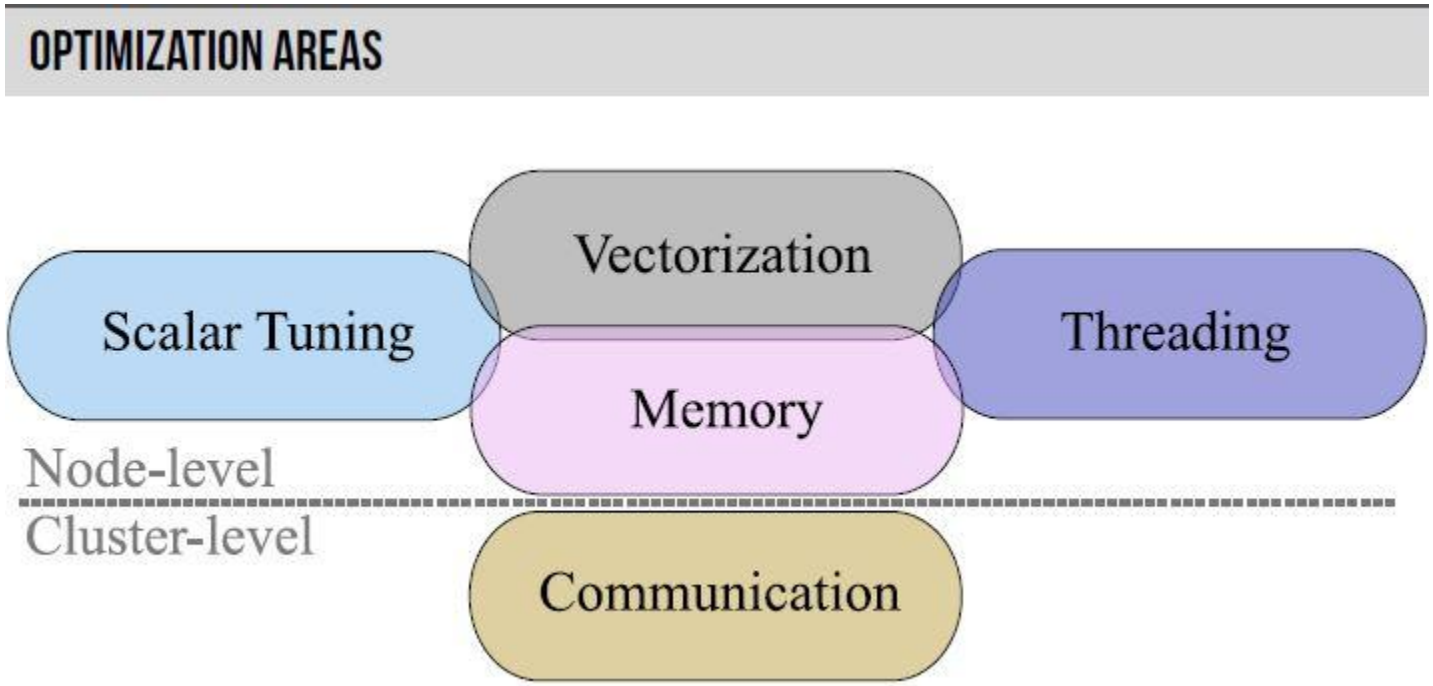
O mesmo problema pode ocorrer em operações com funções:

CONSISTENCY OF PRECISION: FUNCTIONS

```
1  // Bad: 3.14 is a double
2  float x = 3.14;
3
4  // Bad: sin() is a
5  // double precision function
6  float s = sin(x);
7
8  // Bad: round() takes double
9  // and returns double
0  long v = round(x);
1
2  // Bad: abs() is not from IML
3  // it takes int and returns int
4  int v = abs(x);
```

```
1  // Good: 3.14f is a float
2  float x = 3.14f;
3
4  // Good: sin() is a
5  // single precision function
6  float s = sinf(x);
7
8  // Good: lroundf() takes float
9  // and returns long
10 long v = lroundf(x);
11
12 // Good: fabsf() is from IML
13 // It takes and returns a float
14 float v = fabsf(x);
```

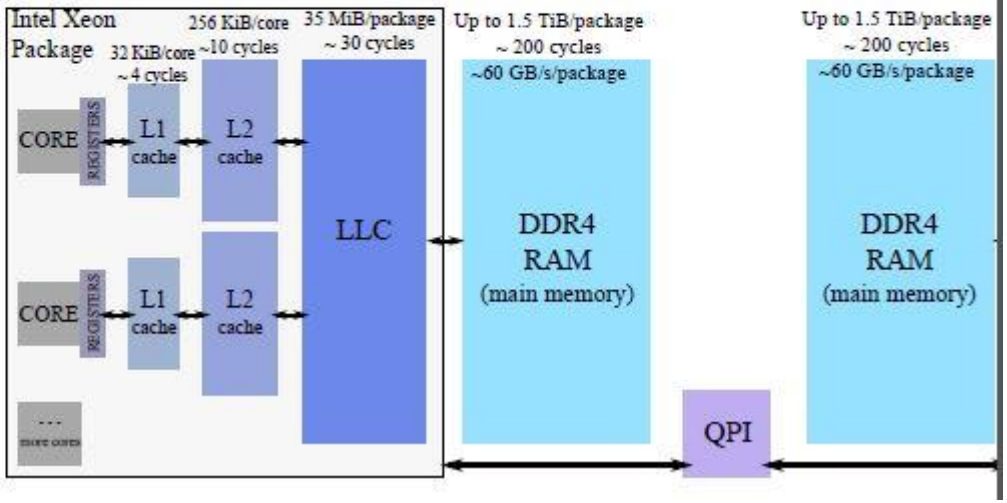
II.2- Memory Optimization



A Hierarquia de Memória afeta enormemente o desempenho: é importante acessar o dado de forma a obter localidade temporal e localidade espacial

INTEL XEON CPU: MEMORY ORGANIZATION

- Hierarchical cache structure
- Two-way processors have NUMA architecture



Localidade de referência é uma propriedade importante para o projeto de sistemas computacionais eficientes em diversos cenários reais. Nesses sistemas, o acesso aos recursos tende a não ser igualmente provável. Além disso, o projeto de sistemas pode considerar o padrão de acesso aos recursos como forma de aumentar o desempenho. Um exemplo de recurso que apresenta alta localidade de referência é a memória.

Existem, basicamente, dois tipos de localidade de referência: a temporal e a espacial. A localidade de referência temporal refere-se ao acesso de um mesmo recurso duas ou mais vezes em um curto intervalo de tempo. A localidade de referência espacial, refere-se ao acesso de dois recursos que estejam próximos em um curto intervalo de tempo ou também pode ser definida como a possibilidade de executar o programa ou recurso seguinte. O uso da hierarquia de memória, como caches, memória RAM e disco rígido, por exemplo, tem forte impacto sobre o desempenho dos computadores graças à localidade de referência.

Acessos à memória sem localidade espacial (com “stride” e “offset”) degradam o desempenho:

UNIT-STRIDE ACCESS

Unit-stride access is optimal:

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

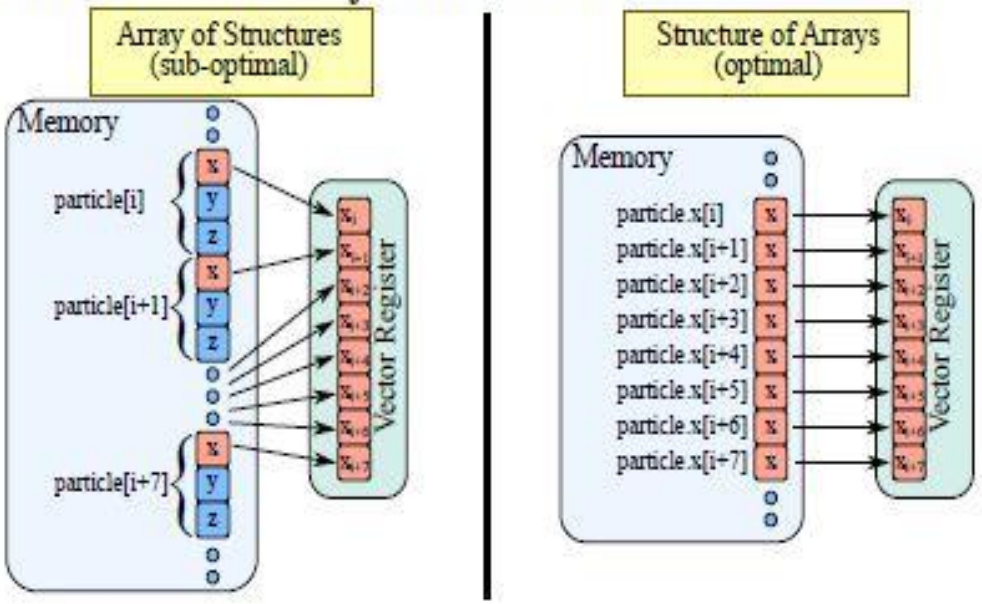
Non-unit stride is slower:

```
1 for (int i = 0; i < n; i++)
2   A[i*stride] += B[i];
```

Stochastic access may be vectorized (but not efficient):

```
1 for (int i = 0; i < n; i++)
2   A[offset[i]] += B[i];
```

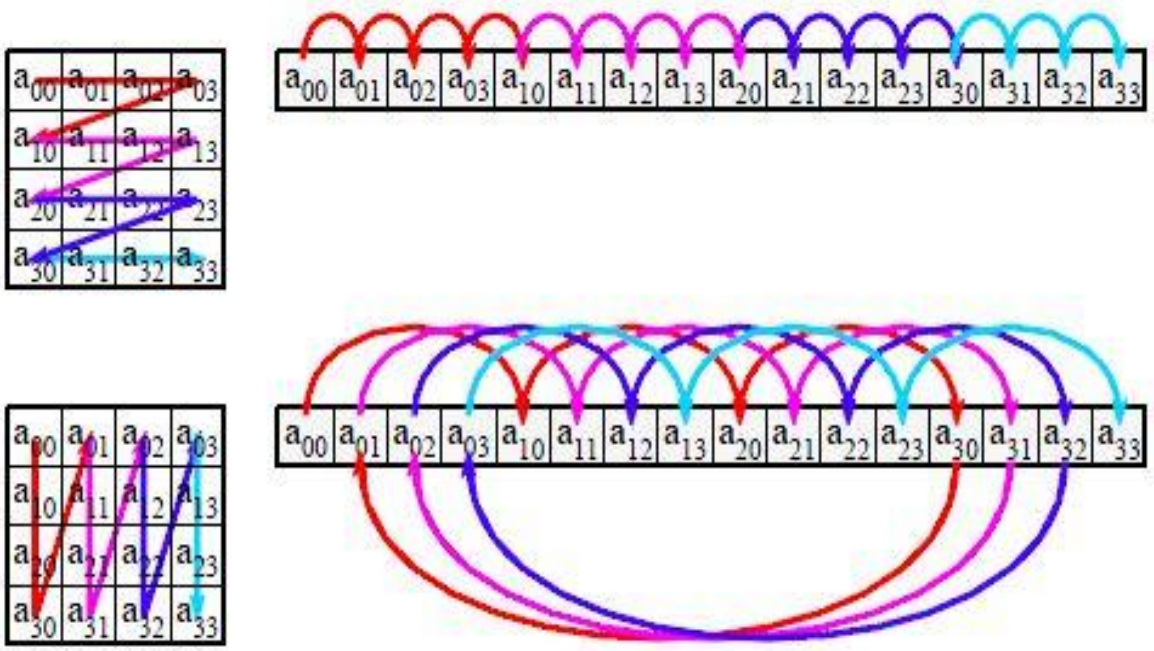
It may be a question of changing the order of loop nesting, but sometimes you need to modify data structures:



O loop deve percorrer a matriz evitando acessos de stride:

PRINCIPLE

Choose loop order to maintain unit-stride memory access



Compiler may or may not be able to automate loop permutation.

Multiplicação de matriz por vetor: C=AB

```
1 void Multiply (const double* const A, const double* const b, double
      * const c, const long n, const long m){
2     assert(n%10 == 0);
3
4     for(long i = 0; i < m; i++)
5         for(long j = 0; j < n; j++)
6             c[i] += A[i*n+j] * b[j];
7 }
```

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} & a_{19} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} & a_{29} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} & a_{39} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Multiplicação de matriz: Solução para percorrer a matriz com enlace interior percorrendo a variável que apresenta maior localidade espacial (mudar o percurso em k para j)

EXAMPLE: OVER-SIMPLIFIED MATRIX-MATRIX MULTIPLICATION

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4 #pragma vector aligned
5     for (int k = 0; k < n; k++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

After:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int k = 0; k < n; k++)
4 #pragma vector aligned
5     for (int j = 0; j < n; j++)
6       C[i*n+j]+=A[i*n+k]*B[k*n+j];
```

Exemplo de acesso á memória com “stride” no loop interior que pode ser transferido para o loop exterior

```c
#include <stdio.h>
#include <omp.h>

int main(){
    int n=128;

    int A[n*n];
    int B[n*n];
    int C[n*n];

    #pragma omp parallel for

        for (int i=0;i<n;i++){
            for(int j=0;j<n;j++)
            {

            #pragma vector aligned
                for (int k=0;k<n;k++){
                    C[i*n+j]+=A[i*n+k]*B[k*n+j];
                }
            }

    }
    printf("\n");
return 0;
}
```

```c
#include <stdio.h>
#include <omp.h>

int main(){
    int n=128;

    int A[n*n];
    int B[n*n];
    int C[n*n];

    #pragma omp parallel for
    for (int i=0;i<n;i++){
        for(int k=0;k<n;k++)


    #pragma vector aligned
    for (int j=0;j<n;j++){
        C[i*n+j]+=A[i*n+k]*B[k*n+j];
    }
    }
return 0;
}
```

Solução para aumentar a localidade temporal dos dados na cache

LOOP FUSION TECHNIQUE

Re-use data in cache by fusing loops in a data processing pipeline

```
1  MyData* data = new MyData(n);
2
3  for (int i = 0; i < n; i++)
4    Initialize(data[i]);
5
6  for (int i = 0; i < n; i++)
7    Stage1(data[i]);
8
9  for (int i = 0; i < n; i++)
10   Stage2(data[i]);
```

```
1  MyData* data = new MyData(n);
2
3  for (int i = 0; i < n; i++) {
4
5    Initialize(data[i]);
6
7    Stage1(data[i]);
8
9    Stage2(data[i]);
10 }
```

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance.

Códigos para testes

```cpp
#include <cstdlib>

int main() {
    int n = 1024*1024*1024;
    float* data;
    data = (float*)malloc(n*sizeof(float));

    for (int i = 0; i < n; i++)
        data[i] = i;

    for (int i = 0; i < n; i++)
        data[i] = 2*data[i];

    for (int i = 0; i < n; i++)
        data[i]++;
}
```

```cpp
#include <cstdlib>

int main() {
    int n = 1024*1024*1024;
    float* data;
    data = (float*)malloc(n*sizeof(float));

    for (int i = 0; i < n; i++) {
        data[i] = i;
        data[i] = 2*data[i];
        data[i]++;
    }
}
```

Resultados

- Windows 8.1
- Intel Core i7-4500U

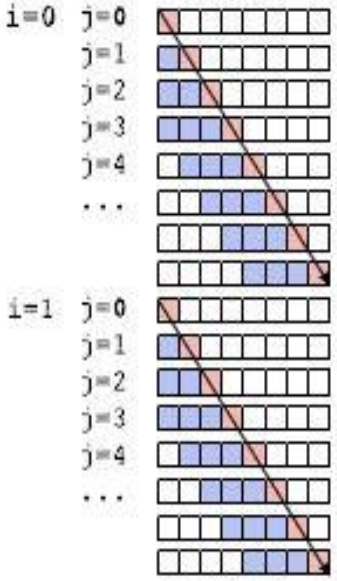
	Média (ms)	Desvio Padrão	N execuções
Com Loop Fusion	2760,2	122,9	5
Sem Loop Fusion	3593,6	171,3	5

O sistema de pré-busca, traz para a cache os dados a serem percorridos pelo enlace interior, se o número de dados (n) é muito grande os dados do percurso não cabem na cache e ocorre “cache miss” para cada acesso dentro do enlace interior. Quando o percurso é pequeno os dados "recentemente acessados" continuam armazenados na cache.

LOOP TILING: CACHE BLOCKING

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



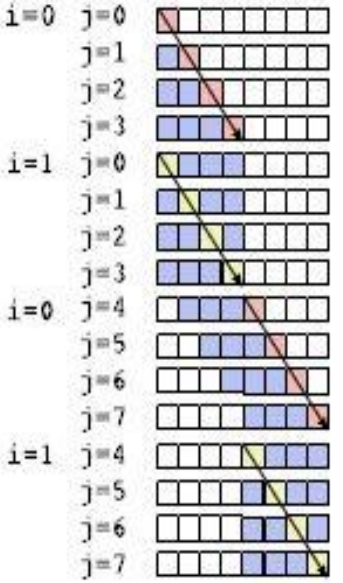
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

Tiled:

```
for (jj=0; jj<n; jj+=TILE)
  for (i=0; i<m; i++)
    for (j=jj; j<jj+TILE; j++)
      ...=...*b[j];
```



Passo a passo para implementar cache blocking:

LOOP TILING (CACHE BLOCKING) -- PROCEDURE

```
for (int i = 0; i < m; i++)  // Original code:
  for (int j = 0; j < n; j++)
    compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
// Step 1: strip-mine inner loop
for (int i = 0; i < m; i++)
  for (int jj = 0; jj < n; jj += TILE)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Same order of operation as original
```

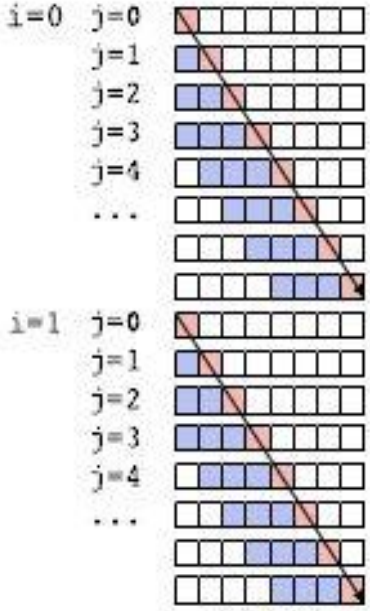
```
// Step 2: permute
for (int jj = 0; jj < n; jj += TILE)
  for (int i = 0; i < m; i++)
    for (int j = jj; j < jj + TILE; j++)
      compute(a[i], b[j]); // Re-use to j=jj sooner
```

Técnica Register Blocking aumenta o “cache hit” em 75%

LOOP TILING: REGISTER BLOCKING

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



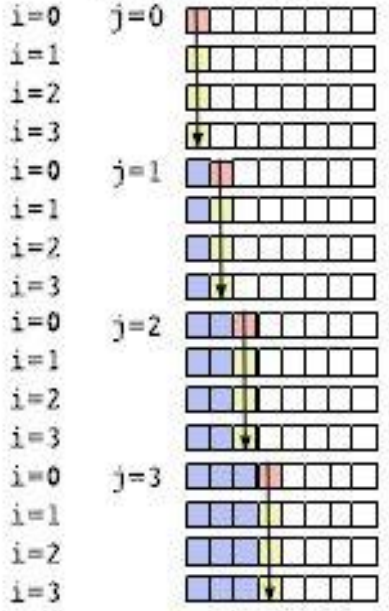
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

Tiled:

```
for (ii=0; ii<m; ii+=TILE)
  for (j=0; j<n; j++)
    for (i=ii; i<ii+TILE; i++)
      ...=...*b[j];
```



Passo a passo para implementar register blocking:

LOOP TILING (UNROLL-AND-JAM/REGISTER BLOCKING)

```
for (int i = 0; i < m; i++)  // Original code:
  for (int j = 0; j < n; j++)
    compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
// Step 1: strip-mine outer loop
for (int ii = 0; ii < m; ii += TILE)
  for (int i = ii; i < ii + TILE; i++)
    for (int j = 0; j < n; j++)
      compute(a[i], b[j]); // Same order of operation as original
```

```
// Step 2: permute and vectorize outer loop
for (int ii = 0; ii < m; ii += TILE)
#pragma simd
    for (int j = 0; j < n; j++)
      for (int i = ii; i < ii + TILE; i++)
        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

A implementação “unroll-and-jam” facilita a vetorização automática

LOOP TILING (UNROLL-AND-JAM) -- ALTERNATIVE IMPLEMENTATION

```
for (int i = 0; i < m; i++)   // Original code:
  for (int j = 0; j < n; j++)
    compute(a[i], b[j]); // Memory access is unit-stride in j
```

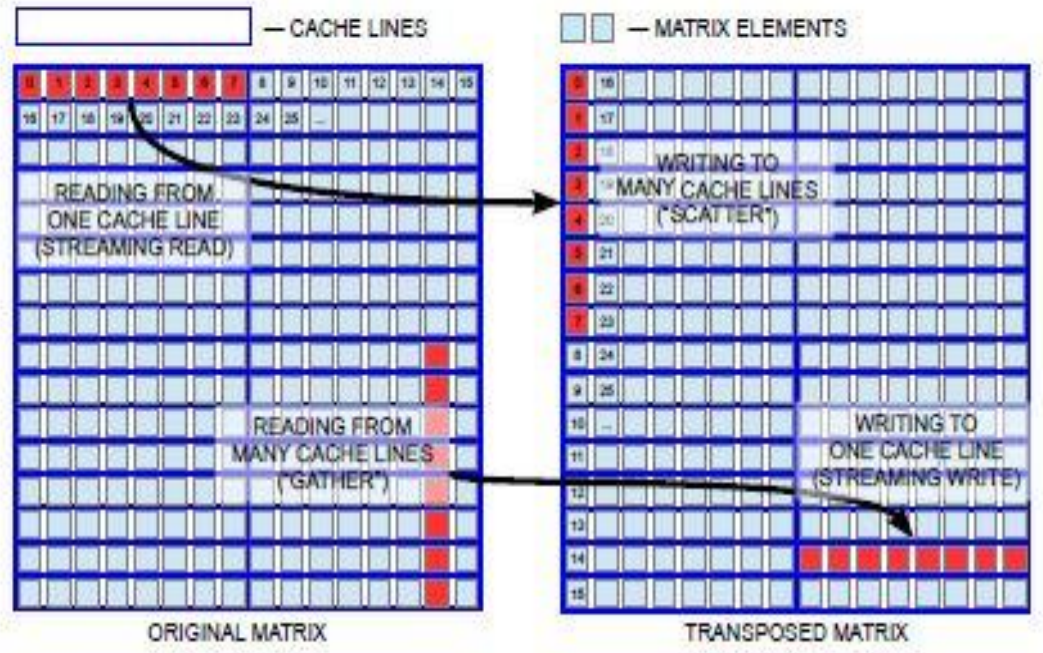
```
// Step 1: strip-mine both loops
for (int ii = 0; ii < m; ii += TILE)
  for (int i = ii; i < ii + TILE; i++)
    for (int jj = 0; jj < n; jj += VECLEN)
      for (int j = jj; j < jj + VECLEN; j++)
        compute(a[i], b[j]); // Same order of operation as original
```

```
// Step 2: permute middle two loops
for (int ii = 0; ii < m; ii += TILE)
  for (int jj = 0; jj < n; jj += VECLEN)
    for (int i = ii; i < ii + TILE; i++)
      for (int j = jj; j < jj + VECLEN; j++)
        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

Exemplo com código de transposição de matriz

LOOP TILING EXAMPLE: MATRIX TRANSPOSITION

$$B = A^T \quad \Leftrightarrow \quad B_{ij} = A_{ji}$$



Implementar os códigos abaixo com tile múltiplo de 64

MATRIX TRANSPOSITION

Before:

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++)
3   for (int j = 0; j < n; j++)
4     B[i*n + j] = A[j*n + i];
```

After:

```
1 const int tile = 200;
2 if (n%tile != 0) exit(1);
3
4 #pragma omp parallel for
5 for (int ii=0; ii<n; ii+=tile)
6  for (int jj=0; jj<n; jj+=tile)
7   for (int i=ii; i<ii+tile; i++)
8    for (int j=jj; j<jj+tile; j++)
9     B[i*n + j] = A[j*n + i];
```

Regras para aumentar a localidade espacial

PRINCIPLE

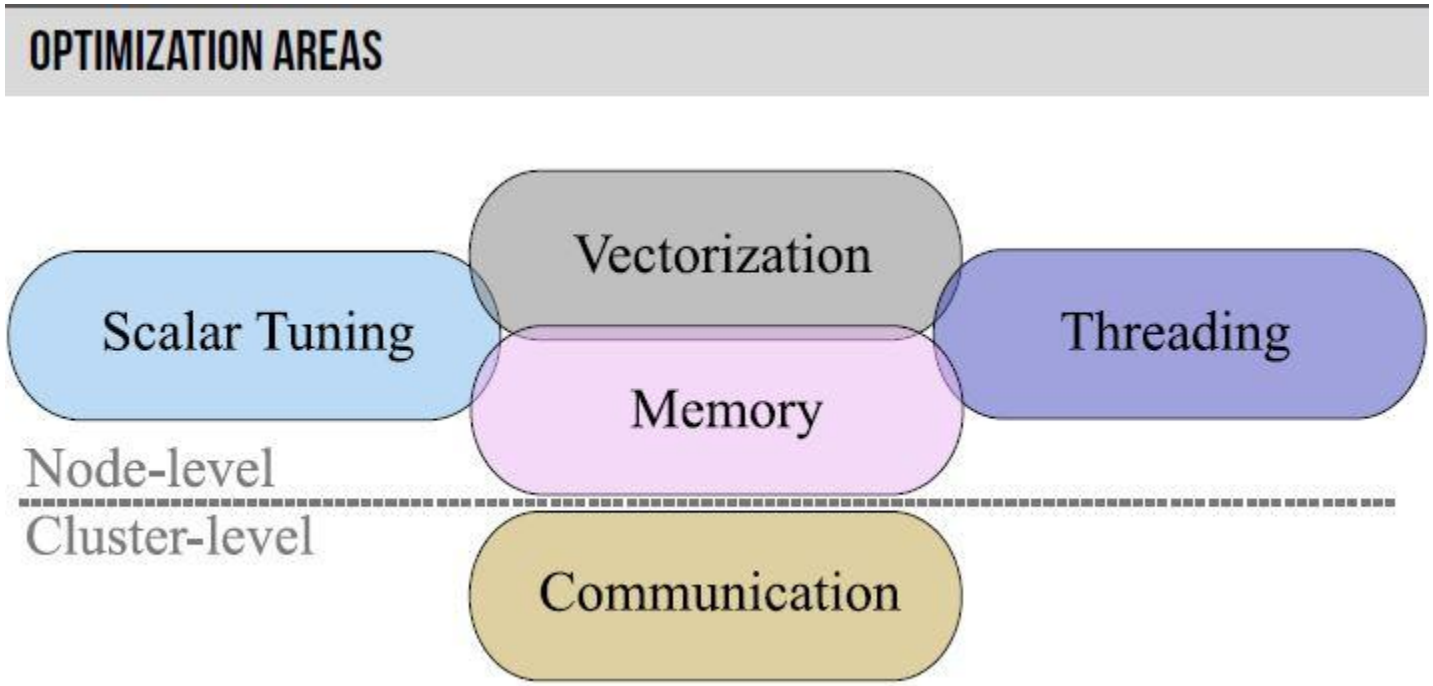
- For best spatial locality, order loops to get unit-stride
- At -O2 and above, the compiler may interchange loops
- In complex cases, investigate loop interchange manually
- May need to re-design data containers to get unit stride

ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^{\alpha})$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha > 1$. Good implementation can be compute-bound, poor one – memory-bound.	Matrix-matrix multiplication: $O\left(N^{3/2}\right)$ (N = amount of data in matrix), direct N-body calculation: $O\left(N^{2}\right)$
$O(N \log N)$	Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound	Fast Fourier transform, merge sort
$O(\log N)$	Always memory-bound.	Binary search

N = data size

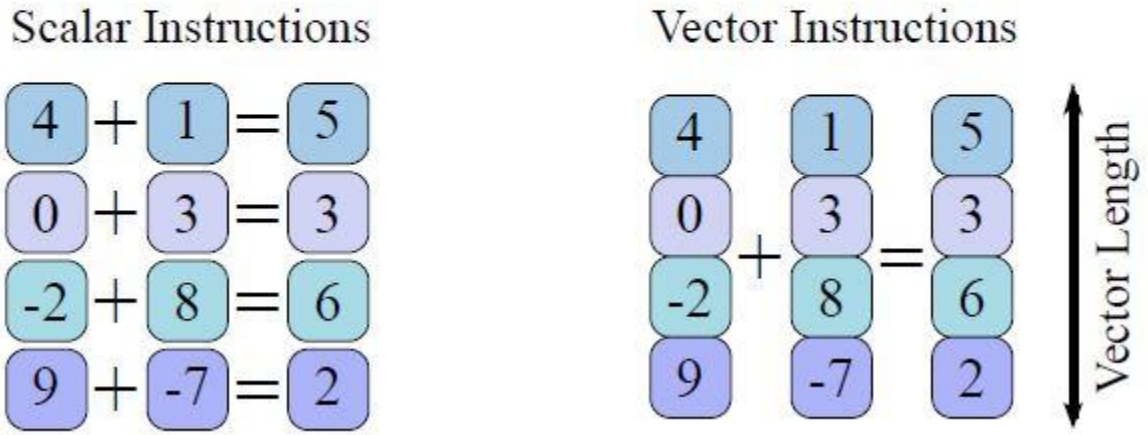
II.2- Vectorization Optimization



Vetorização: permite a execução de uma mesma instrução em vários dados

SHORT VECTOR SUPPORT

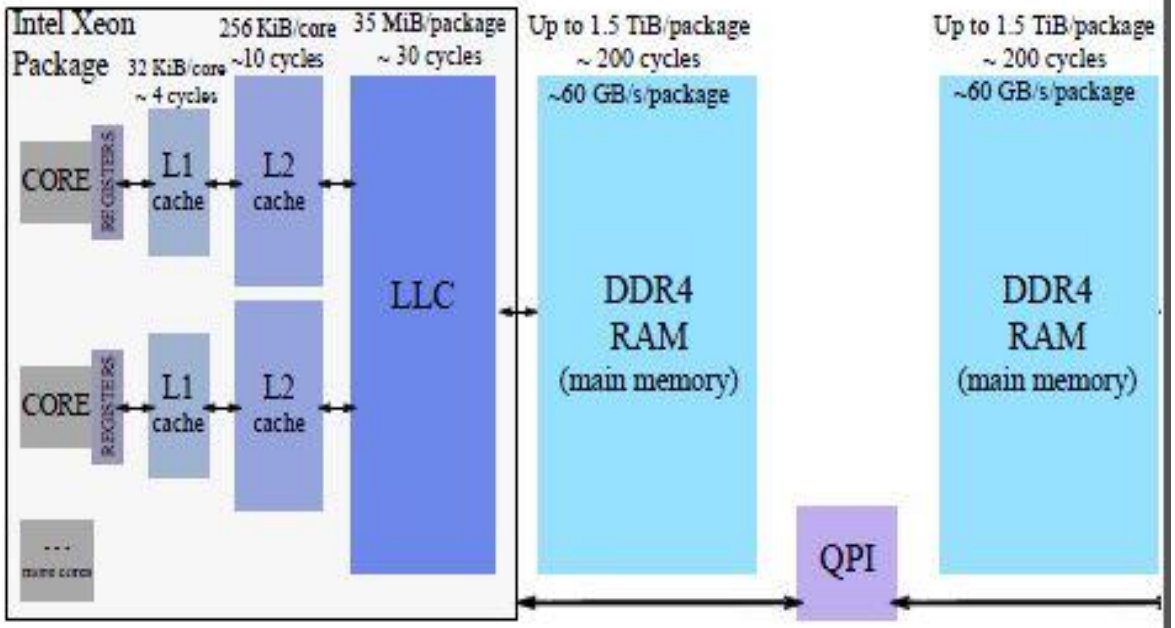
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.



Hierarquia de memória dos processadores da Intel utiliza transferência de linha de cache múltiplo de 64 bytes e a transferência para os registradores é em múltiplo de 16

INTEL XEON CPU: MEMORY ORGANIZATION

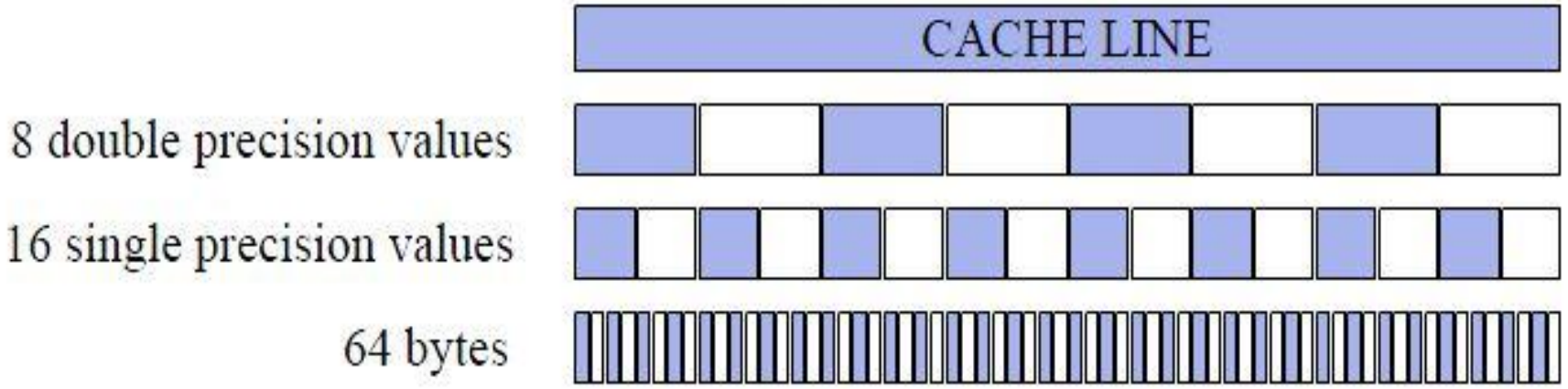
- Hierarchical cache structure
- Two-way processors have NUMA architecture



A transferência de um bloco de dados na arquitetura Intel é múltiplo de 64 bytes

CACHE LINES

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



Streaming SIMD Extensions

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.

Find sources: "Streaming SIMD Extensions" – news · newspapers · books · scholar · JSTOR *(June 2014) (Learn how and when to remove this template message)*

In computing, **Streaming SIMD Extensions** (**SSE**) is a single instruction, multiple data (SIMD) instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series of Central processing units (CPUs) shortly after the appearance of Advanced Micro Devices (AMD's) 3DNow!. SSE contains 70 new instructions, most of which work on single precision floating point data. SIMD instructions can greatly increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.

Intel's first IA-32 SIMD effort was the MMX instruction set. MMX had two main problems: it re-used existing x87 floating point registers making the CPUs unable to work on both floating point and SIMD data at the same time, and it only worked on integers. SSE floating point instructions operate on a new independent register set, the XMM registers, and adds a few integer instructions that work on MMX registers.

SSE was subsequently expanded by Intel to SSE2, SSE3, SSSE3, and SSE4. Because it supports floating point math, it had wider applications than MMX and became more popular. The addition of integer support in SSE2 made MMX a largely redundant code, though further performance increases can be attained in some situations[*when?*] by using MMX in parallel with SSE operations.

SSE was originally called **Katmai New Instructions** (**KNI**), Katmai being the code name for the first Pentium III core revision. During the Katmai project Intel sought to distinguish it from their earlier product line, particularly their flagship Pentium II. It was later renamed **Internet Streaming SIMD Extensions** (**ISSE**[1]), then SSE. AMD eventually added support for SSE instructions, starting with its Athlon XP and Duron (Morgan core) processors.

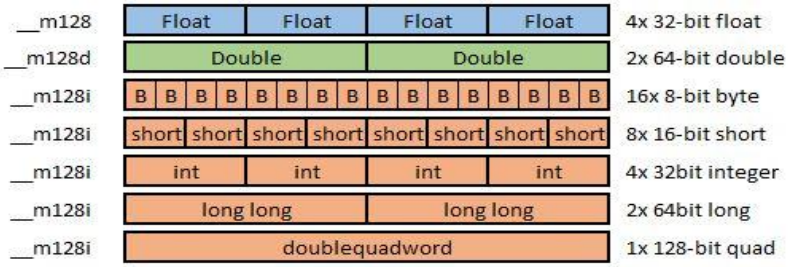
SSE & AVX Registers

SSE and AVX have 16 registers each. On SSE they are referenced as XMM0-XMM15, and on AVX they are called YMM0-YMM15. XMM registers are 128 bits long, whereas YMM are 256bit.

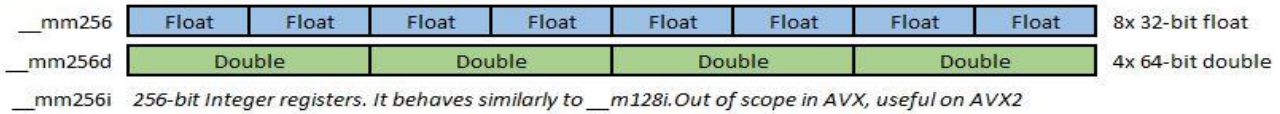
SSE adds three typedefs: `__m128` , `__m128d` and `__m128i` . Float, double (d) and integer (i) respectively.

AVX adds three typedefs: `__m256` , `__m256d` and `__m256i` . Float, double (d) and integer (i) respectively.

SSE Data Types (16 XMM Registers)



AVX Data Types (16 YMM Registers)



> **NOTE:** XMM and YMM overlap! XMM registers are treated as the lower half of the corresponding YMM register. This can introduce some performance issues when mixing SSE and AVX code.

Floating point datatypes (__m128, __m128d, __m256 and __m256d) have only one kind of data structure. Because of this, GCC allows for access to data components as an array. I.e: This is valid:

```
__m256 myvar = _mm256_set1_ps(6.665f); //Set all vector values to a single float
myvar[0] = 2.22f;                      //This is valid in GCC compiler
float f = (3.4f + myvar[0]) * myvar[7]; //This is valid in GCC compiler
```

__m128i and __m256i are unions, so the datatype needs to be referenced. I have not found a proper way to get the union declaration, so I use _mm_extract_epiXX() functions to retrieve individual data values from integer vectors.

https://software.intel.com/content/www/us/en/develop/articles/data-alignment-to-assist-vectorization.html

Data Alignment to Assist Vectorization

By Rakesh Krishnaiyer,
Published:09/07/2013 Last Updated:01/22/2021

Compiler Methodology for Intel® MIC Architecture

Data Alignment to Assist Vectorization

Overview

Data alignment is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase efficiency of data loads and stores to and from the processor. Without going into great detail, processors are designed to efficiently move data when that data can be moved to and from memory addresses that are on specific byte boundaries. For Intel® processors that support Intel® AVX-512 instructions (codenames: SKYLAKE-AVX512, ICELAKE-SERVER, etc.), memory movement is optimal when the data starting address lies on 64 byte boundaries. This is also true for the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) such as the Intel® Xeon Phi™ Coprocessor. Thus, it is desired to force the compiler to create data objects with starting addresses that are modulo 64 bytes.

Para obter informações sobre a arquitetura do processador:

DETECTING AVAILABLE INSTRUCTIONS

In the OS:

```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size    : 64
cache_alignment: 64
address sizes   : 46 bits physical, 48 bits virtual
...
```

In code (see also):

```
1  // Intel compiler
2  // preprocessor macros:
3
4  #ifdef __SSE__
5  // ...SSE code path
6  #endif
7
8  #ifdef __SSE4_2__
9  // ...SSE code path
10 #endif
11
12 #ifdef __AVX__
13 // ...AVX code path
14 #endif
```

Diretivas de compilação para fornecer a arquitetura do processador

TARGETING A SPECIFIC INSTRUCTION SET

`-x[code]` to target specific processor architecture
`-ax[code]` for multi-architecture dispatch

code	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Future Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3/E5/E7 v3, v4 family
AVX	Intel Xeon processor E3/E5 and E3/E5/E7 v2 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

A diretiva –qopt-report gera um arquivo com informações sobre a compilação, para verificar se ocorreu compilação automática

AUTOMATIC VECTORIZATION OF LOOPS

```
1  #include <cstdio>
2
3  int main(){
4    const int n=1024;
5    int A[n] __attribute__((aligned(64)));
6    int B[n] __attribute__((aligned(64)));
7
8    for (int i = 0; i < n; i++)
9      A[i] = B[i] = i;
10
11   // This loop will be auto-vectorized
12   for (int i = 0; i < n; i++)
13     A[i] = A[i] + B[i];
14
15   for (int i = 0; i < n; i++)
16     printf("%2d %2d %2d\n",i,A[i],B[i]);
17 }
```

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

(Linha, coluna)

Condicionais dentro de um enlace diminuem o desempenho da vetorização porque fazem com que sejam executadas várias versões dentro do pipeline

MOVE BRANCHES OUTSIDE OF LOOPS

```
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;
```

É mais eficiente fazer código “redundante” para retirar a condicional do enlace e permitir a execução de apenas uma versão para permitir vetorização. Observe que quando o percurso do enlace interior é múltiplo de 16 ocorre vetorização automática.

REDUNDANT CODE IS OK

```
1 // Elegant, but bad for performance
2 for (ii = 0; ii < n; ii+=16) {
3   for (i = ii; i < ii+16; i++)
4     // Branch causes unnecessary
5     // masking of vector iterations
6     if (i < n) {
7       A[k*n + i] = ...
8     }
9 }
```

```
1 // Redundant code, but faster
2 const int nTrunc = n - n%16;
3 for (ii = 0; ii < nTrunc; ii+=16) {
4   for (i = ii; i < ii+16; i++)
5       A[k*n + i] = ...
6
7 for (i = nTrunc; i < n; i++)
8   A[k*n + i] = ...
9 }
```

A diretiva " aligned" serve para informar que o endereço do dado está alinhado com o tamanho do vetor e deve ser utilizada para permitir a vetorização automática avisando ao compilador quando o índice do enlace interno possui stride múltiplo de 16, mas o compilador não tem como saber automaticamente:

https://software.intel.com/content/www/us/en/develop/articles/data-alignment-to-assist-vectorization.html

DATA ALIGNMENT HINTS

Programmer may promise to the compiler (under penalty of segmentation fault) that alignment has been taken care of:

```
// Promising that A[i*lda + 0] is aligned for every i
// and the same for every other array in this loop
#pragma vector aligned
    for (int j = 0; j < n; j++)
      A[i*lda + j] -= ...
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation and *peel loops*.

Quando o tamanho da matriz não é múltiplo do vetor o percurso do enlace interior não é múltiplo do tamanho do vetor, para possibilitar vetorização a solução está em aumentar o tamanho do enlace interior para ser múltiplo de 16:

PADDING MULTI-DIMENSIONAL CONTAINERS FOR ALIGNMENT

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

Incorrect:

```
1  // A - matrix of size (n x n)
2  // n is not a multiple of 16
3  float* A =
4   _mm_malloc(sizeof(float)*n*n, 64);
5
6  for (int i = 0; i < n; i++)
7    // A[i*n + 0] may be unaligned
8    for (int j = 0; j < n; j++)
9      A[i*n + j] = ...
```

Correct:

```
1  // ... Padding inner dimension
2  int lda=n + (16-n%16); // lda%16==0
3  float* A =
4   _mm_malloc(sizeof(float)*n*lda, 64);
5
6  for (int i = 0; i < n; i++)
7    // A[i*lda + 0] aligned for any i
8    for (int j = 0; j < n; j++)
9      A[i*lda + j] = ...
```

Código original com tamanho n do "enlace interior que não é múltiplo do tamanho do vetor

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   int main()
6   {
7       int n=18;
8       float* A= (float *)_mm_malloc(sizeof(float)*n*n,64);
9       #pragma vector aligned{
10      for (int i = 0; i < n ; i++)
11          for (int j = 0;j < n; j++)
12              A[i*n + j]= 1*2;
13      }
14      return 0;
15
16  }
17
```

Código que altera o percurso do "enlace interno para ser múltiplo do tamanho do vetor

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4
5
6   int main()
7   {
8       int n=18;
9       int lda=n + (16-n%16);
10      float *A= (float *)_mm_malloc(sizeof(float)*n*lda,64);
11      #pragma vector aligned{
12      for (int i=0; i < n ; i++)
13          for (int j = 0 ; j < n; j++)
14              A[i*lda+j]=1*2;
15      }
16      return 0;
17  }
18
```

“Strip Mining é uma técnica utilizada em um loop único para permitir “ Vetorização automática”

STRIP-MINING FOR VECTORIZATION

- Programming technique that turns one loop into two nested loops.
- Used to expose vectorization opportunities.

Original:

```
1 for (int i = 0; i < n; i++) {
2   // ... do work
3 }
```

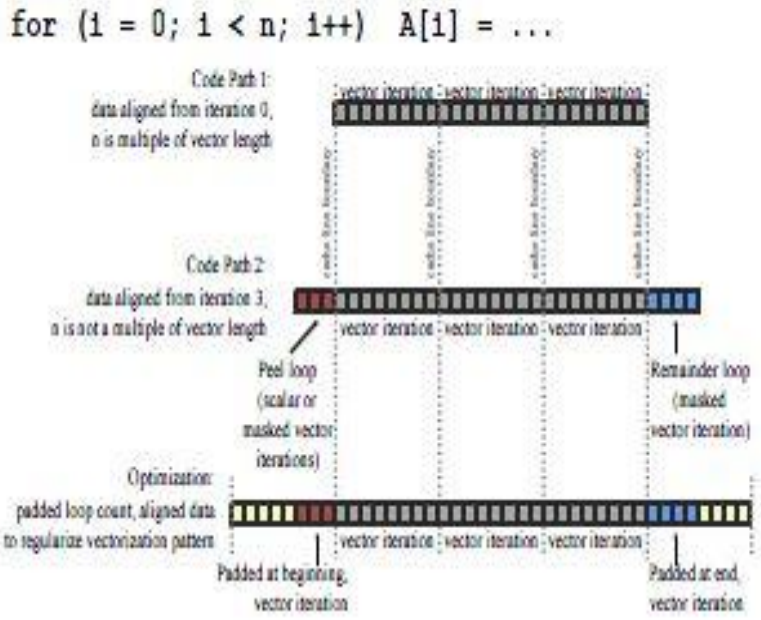
Strip-mined:

```
1 const int STRIP=1024;
2 const int nPrime = n - n%STRIP;
3 for (int ii=0; ii<nPrime; ii+=STRIP)
4   for (int i=ii; i<ii+STRIP; i++)
5     // ... do work
6
7 for (int i=nPrime; i<n; i++)
8   // ... do work
```

Para um vetorização eficiente:

LOOP WAS VECTORIZED, NOW WHAT?

1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**



SUMMARY

1. Vector-Friendly Data Structures
   - Use data structures that allow for unit-stride vector load.
2. Regularization of Vectorization Pattern
   - Align data to 64-byte boundaries
   - Pad data containers and loop bounds
3. Remove Run-time Checks
   - Disable run-time checks for alignment and aliasing with compiler hints
4. Strip-Mining for Vectorization
   - Use strip-mining expose vectorization opportunities.

II-4- Threading Optimization