

# *Processamento Vectorial*

Arquitetura de Computadores  
Lic. em Engenharia Informática

# Processamento Escalar

Cada instrução processa apenas **um** elemento do conjunto de dados

```
for (i=0 ; i < N ; i++) {  
    c[i] = a[i] + b[i];  
}
```

loop:

```
movl (%esi, %ecx, 4), %eax
```

```
movl (%edi, %ecx, 4), %edx
```

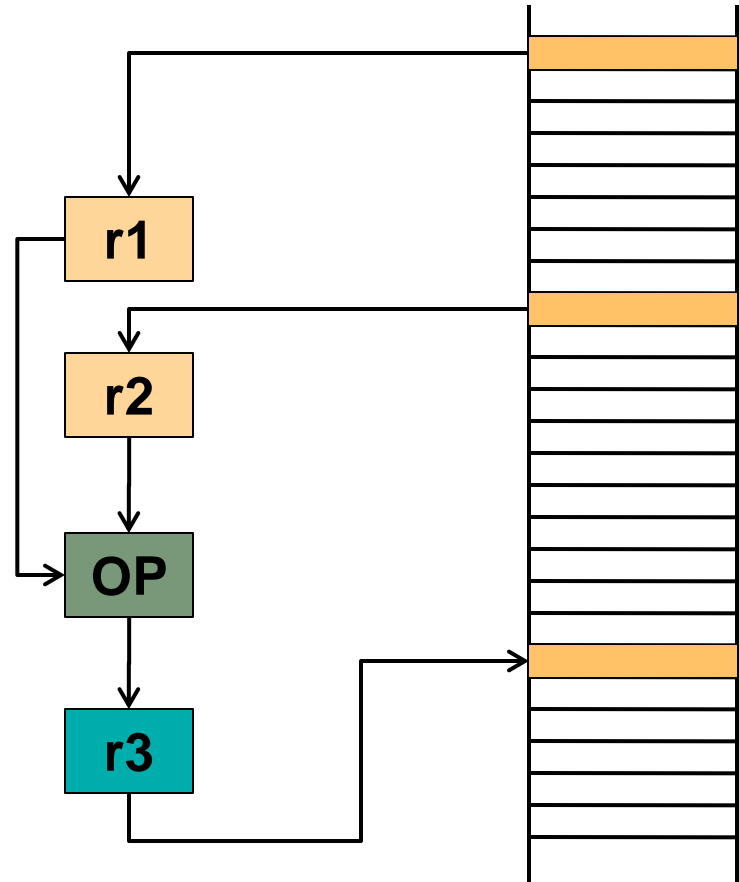
```
addl %eax, %edx
```

```
movl %edx, (%ebx, %ecx, 4)
```

```
incl %ecx
```

```
cmpl N, %ecx
```

```
j1 loop
```



# Processamento Vectorial

Cada instrução processa **n** elementos do conjunto de dados (n=4 neste exemplo)

```
for (i=0 ; i < N ; i++) {  
    c[i] = a[i] + b[i];  
}
```

loop:

```
mov.v (%esi, %ecx, 4), %xmm0
```

```
mov.v (%edi, %ecx, 4), %xmm1
```

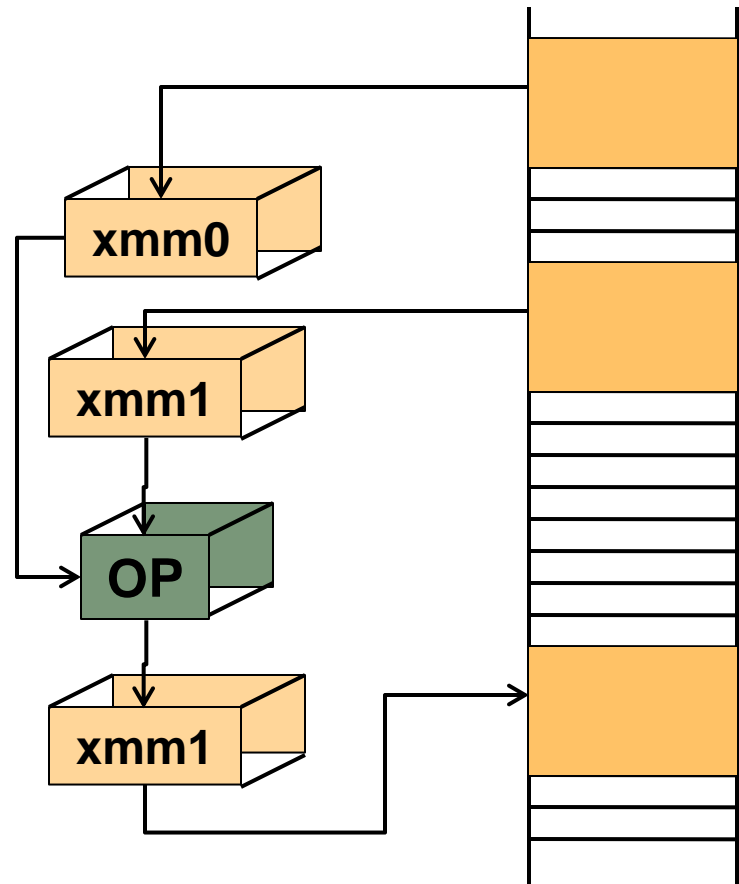
```
add.v %xmm0, %xmm1
```

```
mov.v %xmm1, (%ebx, %ecx, 4)
```

```
addl $4, %ecx
```

```
cmpl N, %ecx
```

```
j1 loop
```



# Processamento Vectorial

$$T_{EXEC} = CPI * \#I / f$$

- Ganho 1

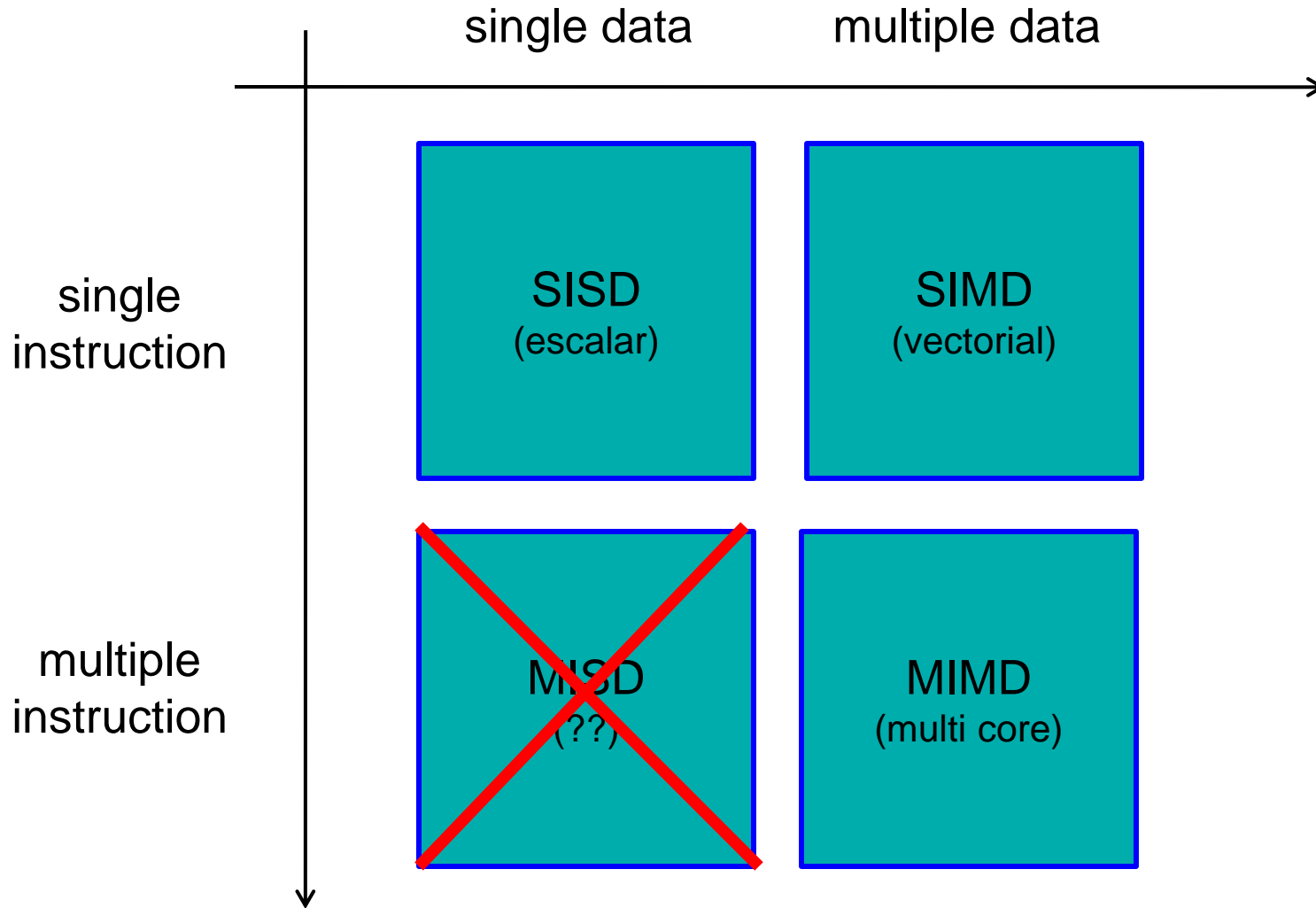
Cada instrução processa  $n$  elementos de dados, logo  
**reduz o número de instruções**

Escreva uma equação aqui.

- Ganho 2

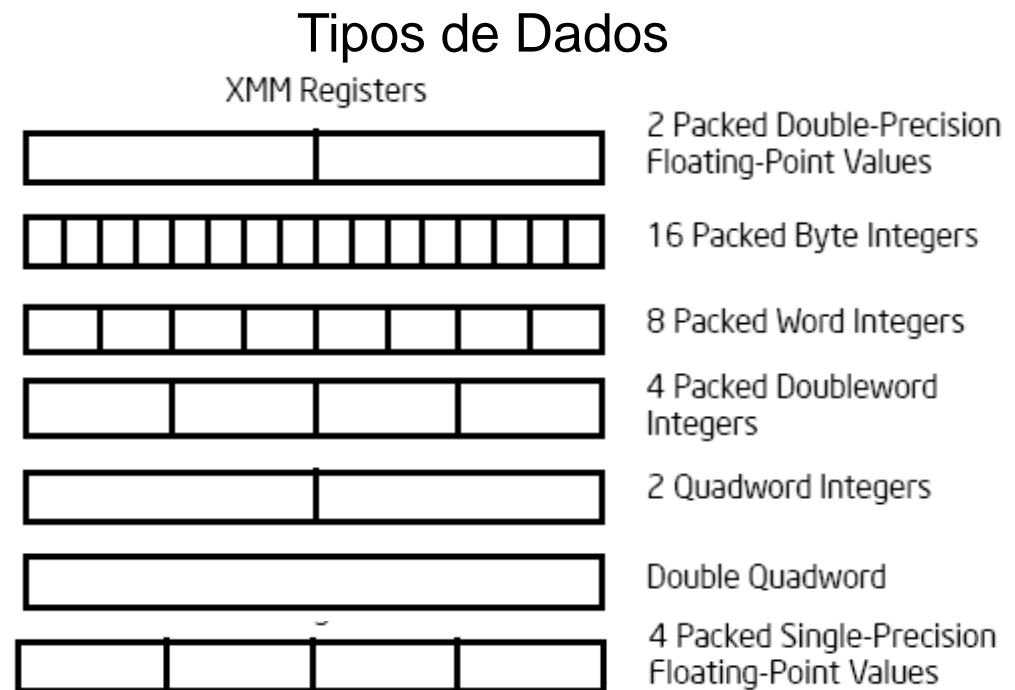
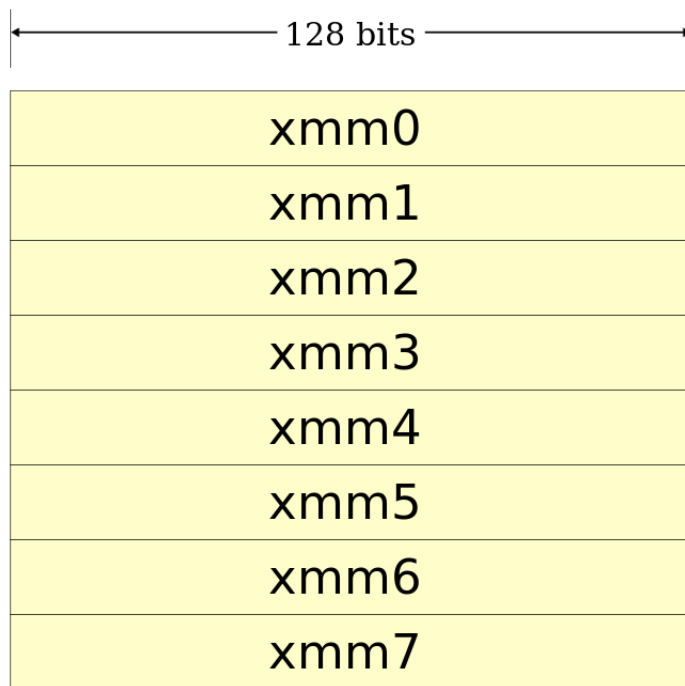
**Se** as unidades funcionais realizarem as  $n$  operações em paralelo,  
**reduz o CPI**

# Parelelismo – Taxonomia de Flynn



# Intel SSE - Streaming SIMD Extensions

- SSE adiciona à arquitetura Intel 8 registros de 128 *bits*: *%xmm0* .. *%xmm7*
- adiciona ainda instruções para operar sobre vetores de vários tipos de dados



# *Intel SSE*

1994 – Pentium II e Pentium with MMX –

Intel introduz a primeira extensão SIMD ao conjunto de instruções (MMX - MultiMedia eXtensions)

1995 – Introdução de Streaming Simd Extensions (SSE) no Pentium III

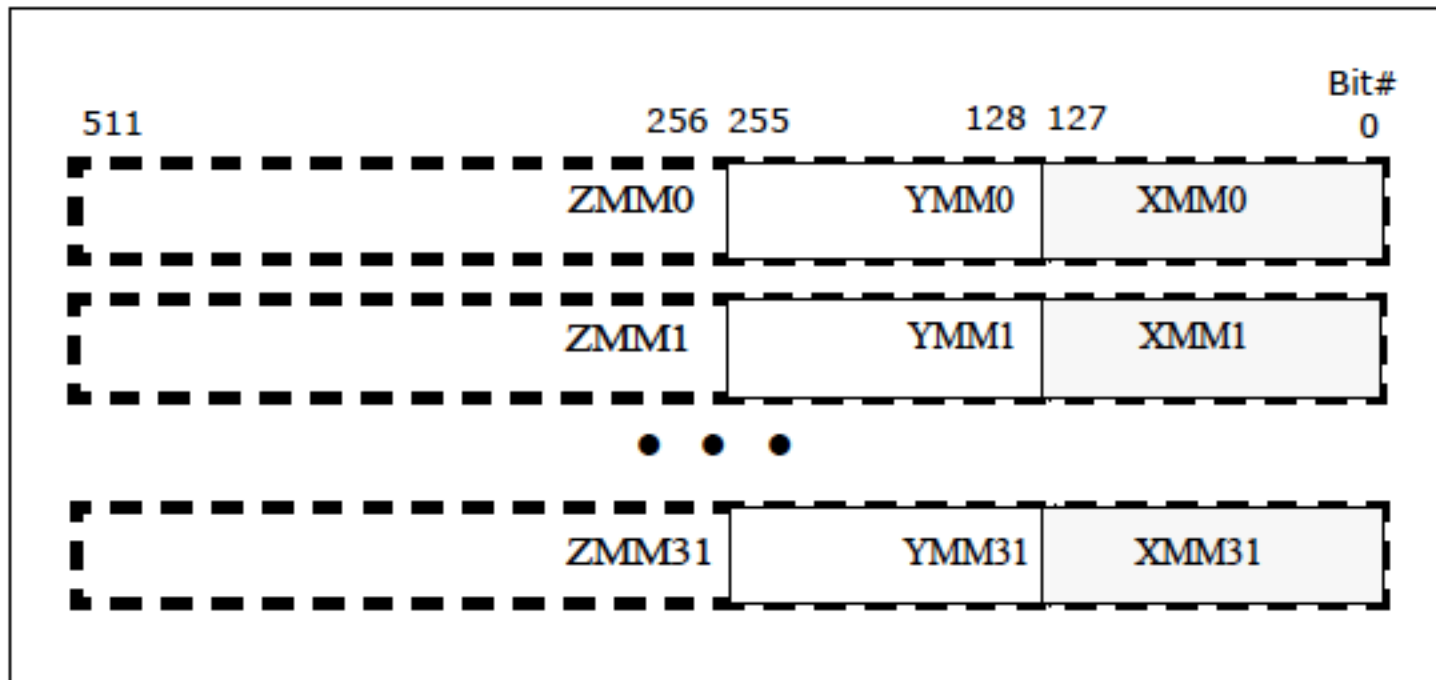
2000 – Introdução de SSE2 no Pentium IV

2004 - Introdução de SSE3 no Pentium IV HT

2007 - Introdução de SSE4

# Intel Advanced Vector Extensions (AVX)

- Intel AVX (2011) – registros YMM0 . . YMM15 de 256 *bits*
- Intel AVX512 (2015) – registros ZMM0 . . ZMM31 de 512 *bits*





# Instruções SSE: Transferência de Dados

Instruções	Operandos orig, dest	Obs.
MOVDQA	xmm/m128, xmm	Mover 2 palavras quádruplas (2*8 bytes) Apenas para inteiros
MOVDQU	xmm, xmm/m128	A - addr alinhado M16; U – addr não alinhado
MOVAP[S   D]	xmm/m128, xmm	Mover 4 FP precisão simples ou 2 FP precisão dupla
MOVUP[S   D]	xmm, xmm/m128	A – addr alinhado M16 U – addr não alinhado

- **Alinhamento SSE:**

Os acessos à memória é mais eficiente se o endereço for alinhado, isto é, se o bloco de 16 *bytes* a aceder se iniciar num endereço múltiplo de 16

# Instruções SSE: Operações Inteiras

Instruções	Operandos orig, dest	Obs.
PADD? PSUB? PAND? POR?	xmm/m128, xmm	Adição, subtração, conjunção ou disjunção do tipo de dados indicado Operação realizada sobre o número de elementos determinado pelo registo+tipo de dados Endereços em memória alinhados O resultado não pode ser em memória

? = B | W | D | Q

B – byte

D – 4 bytes

W – 2 bytes

Q – 8 bytes

# Instruções SSE: Operações FP

Instruções	Operandos orig, dest	Obs.
ADDP? SUBP? MULP? DIVP? SQ RTP? MAXP? MINP? ANDP? ORP?	xmm/m128, xmm	Operação sobre o tipo de dados indicado  Operação realizada sobre o número de elementos determinado pelo tipo de dados (S = 4 ; D = 2)  Endereços em memória alinhados  O resultado não pode ser em memória

? = S | D

S – precisão simples

D – dupla precisão

# Exemplo SSE

```
float a[100], b[100], r[100];

func (int n, float *a, float *b, float *r) {
    int i;
    for (i=0 ; i<n ; i++)
        r[i] = a[i] * b[i];
}
```

func:

...

movl 8(%ebp), %edx

movl 12(%ebp), %eax

movl 16(%ebp), %ebx

movl 20(%ebp), %esi

movl \$0, %ecx

ciclo:

movaps (%eax, %ecx, 4), %xmm0

mulps (%ebx, %ecx, 4), %xmm0

movaps %xmm0, (%esi, %ecx, 4)

addl \$4, %ecx

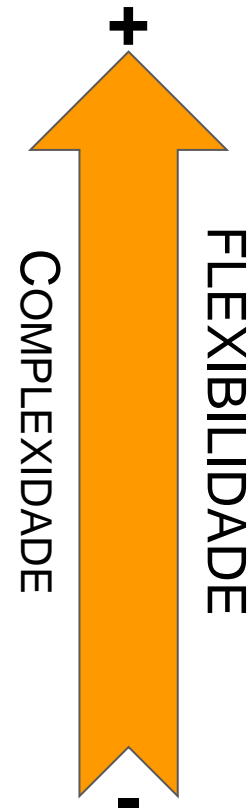
cmpl %edx, %ecx

jle ciclo

...

# Processamento Vectorial - desenvolvimento

- *Assembly*
  - Utilização directa de instruções *assembly*
- *Compiler Intrinsics*
  - Pseudo-funções disponibilizadas pelo compilador que permitem o desenvolvimento explícito de código vectorial a um nível semântico mais elevado que o *assembly*
- *Auto Vectorização*
  - Vectorização pelo compilador



# *Compiler Intrinsics*

- *Compiler intrinsics* são pseudo-funções que expõem funcionalidades do CPU incompatíveis com a semântica da linguagem de programação usada (C/C++ neste caso)

# Compiler Intrinsics

- As funções e tipos de dados definidos como *intrinsics* são acessíveis incluindo o *header* `<ia32intrin.h>`

Tipos de Dados	
__m64	Vector de 64 bits – inteiros (MMX)
__m128	Vector 128 <i>bits</i> – 4 FP SP (SSE)
__m128d	Vector 128 <i>bits</i> – 2 FP DP (SSE2)
__m128i	Vector 128 <i>bits</i> – inteiros (SSE2)

# Compiler Intrinsics

Operações Aritméticas (single FP)		
Pseudo-função	Descrição	Instrução
<code>__m128 _mm_add_ps (__m128, __m128)</code>	Adição	ADDPS
<code>__m128 _mm_sub_ps (__m128, __m128)</code>	Subtracção	SUBPS
<code>__m128 _mm_mul_ps (__m128, __m128)</code>	Multiplicação	MULPS
<code>__m128 _mm_div_ps (__m128, __m128)</code>	Divisão	DIVPS
<code>__m128 _mm_sqrt_ps (__m128)</code>	Raiz Quadrada	SQRTPS
<code>__m128 _mm_rcp_ps (__m128)</code>	Inverso	RCPPS
<code>__m128 _mm_rsqrt_ps (__m128)</code>	Inverso da Raiz Quadrada	RSQRTPS



# *Compiler Intrinsics*

## **Movimento de Dados (single FP)**

Pseudo-função	Descrição	Instrução
<code>__m128 _mm_load_ps (float *)</code>	Carrega vector de memória para registo (alinhado 16)	MOVAPS
<code>_mm_store_ps (float *, __m128)</code>	Escreve registo em vector de memória (alinhado 16)	MOVAPS

# Compiler Intrinsics

## Set (single FP)

Pseudo-função	Descrição	Instrução
<code>__m128 _mm_set1_ps (float)</code>	Carrega 1 constante para os 4 elementos do registo	Várias
<code>__m128 _mm_set_ps (float, float, float, float)</code>	Carrega 4 constantes para os 4 elementos do registo	Várias
<code>__m128 _mm_setzero_ps (f)</code>	Coloca os 4 elementos do registo a zero	Várias

# Compiler Intrinsics

## Comparação (single FP)

Pseudo-função	Descrição	Instrução
<code>__m128 _mm_cmpeq_ps (__m128, __m128)</code>	Põe a 1 se iguais	CMPEQPS
<code>__mm_cmp[lt, le, gt, gneq, nlt, ngt, nle, nge]</code>		

A comparação é feita elemento a elemento dos registos %xmm,  
Sendo o resultado um registo %xmm com o elemento correspondente a 0 ou 1

# Compiler Intrinsics: Exemplo 1 – Intel C compiler

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

func() {
    for (int i=0 ; i< SIZE ; i++) {
        c[i] = a[i] + b[i];
    } }
```

```
#include <ia32intrin.h>
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

func() {
    for (int i=0 ; i<SIZE ; i+=4) {
        __m128 mb = _mm_load_ps (&b[i]);
        __m128 ma = _mm_load_ps (&a[i]);
        __m128 mc = _mm_add_ps (ma, mb);
        _mm_store_ps (&c[i], mc);
    } }
```

# Compiler Intrinsics: Exemplo 2 - Intel C compiler

```
#include <math.h>
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));
```

```
func() {
    for (int i=0 ; i< SIZE ; i++) {
        r[i] =
    } }
```

```
#include <ia32intrin.h>
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

func() {
    __m128 cinco = _mm_set1_ps (5.);
    for (int i=0 ; i<SIZE ; i+=4) {
        __m128 mb = _mm_sqrt_ps(_mm_load_ps (&b[i]));
        __m128 ma = _mm_load_ps(&a[i]);
        __m128 mr = _mm_mul_ps (cinco, _mm_add_ps (ma, mb);
        _mm_store_ps (&c[i], mr);
    } }
```

# Auto-vectorização

- O compilador pode vectorizar o código
- Comando gcc:

```
gcc -O3 -ftree-vectorizer-verbose=n -march=...
```

# Auto-vectorização

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        c[i] = a[i] + b[i];
    } }
```

```
loop:
    xor %eax, %eax
.L1:
    movaps a(%eax), %xmm0
    addps b(%eax), %xmm0
    movaps %xmm0, c(%eax)
    add $16, %eax
    cmp $4000000, %eax
    jl .L1
    ret
```

# Auto-vectorização

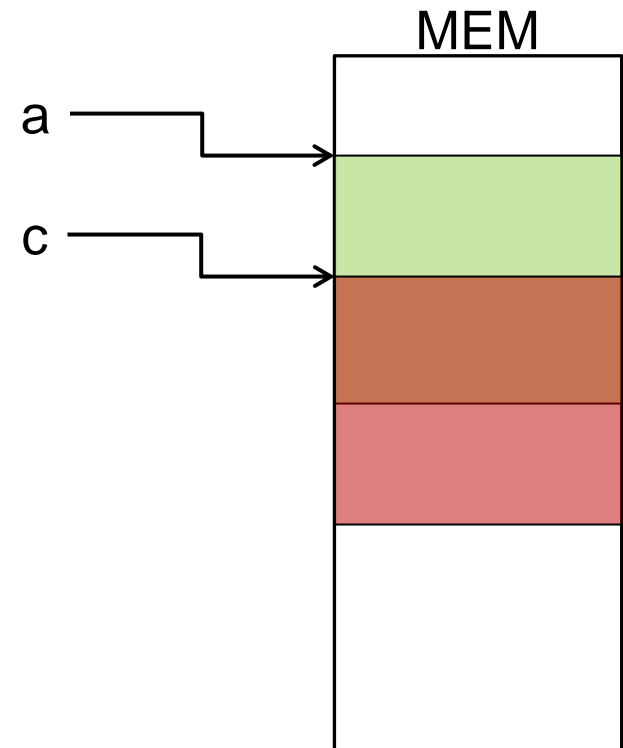
```
loop (float *a, float *b, float *c, const int S) {  
    for (int i=0 ; i< S ; i++) {  
        c[i] = a[i] + b[i];  
    }  
}
```

Possibilidade de ***aliasing***, isto é:  
as regiões de memória apontadas  
pelos diferentes apontadores podem-  
se sobrepor!

***versioning***, isto é:

O compilador versões escalares e  
vectoriais do ciclo e código para  
verificar o *aliasing*.

Em *runtime* é escolhida a versão mais  
apropriada do ciclo





# Auto-vectorização

```
loop ( float * __restrict__ a, float * __restrict__ b,  
      float * __restrict__ c, const int S) {  
    for (int i=0 ; i< S ; i++) {  
        c[i] = a[i] + b[i];  
    } }
```

O qualificador `__restrict__` indica ao compilador que durante a existência daquele apontador

**NÃO EXISTE QUALQUER OUTRA REFERÊNCIA**

para a zona de memória acedida a partir desse apontador.

Logo não existe a possibilidade de *aliasing*

# Auto-vectorização

```
loop ( float * __restrict__ a, float * __restrict__ b,  
       float * __restrict__ c, const int S) {  
    for (int i=0 ; i< S ; i++) {  
        c[i] = a[i] + b[i];  
    } }
```

Alinhamento a múltiplos de 16 dos vetores?

Resulta na utilização de instruções para acessos não alinhados:  
mais ineficientes.

```
loop ( float * __restrict__ a, float * __restrict__ b,  
       float * __restrict__ c, const int S) {  
a = __builtin_assume_aligned (a,16);  
b = __builtin_assume_aligned (b,16);  
c = __builtin_assume_aligned (c,16);  
    for (int i=0 ; i< S ; i++) {  
        c[i] = a[i] + b[i];  
    } }
```

# Bloqueadores Auto-vectorização: dados contíguos

```
#define SIZE 1000000
typedef struct {float a, b, c, pad;} MYDATA;

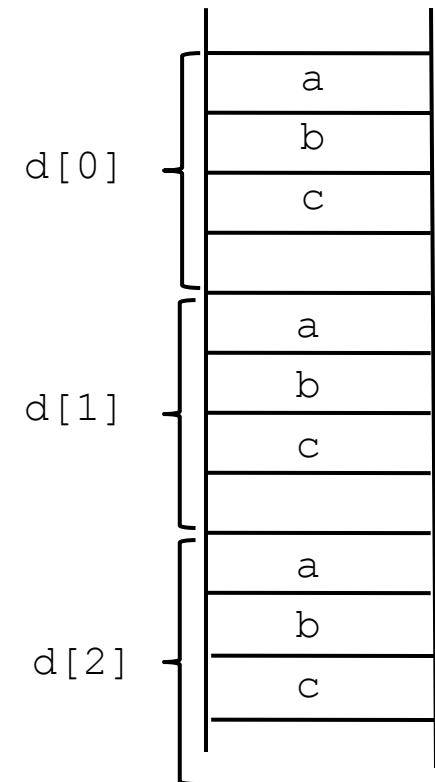
MYDATA d[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        d[i].c = d[i].a + d[i].b;
    }
}
```

Array of Structures (AoS) :

os vários elementos do mesmo campo não são armazenados consecutivamente em memória.

Código potencialmente não vectorizável!



# Bloqueadores Auto-vectorização: dados contíguos

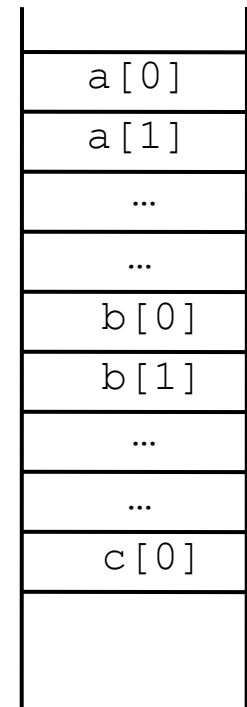
```
#define SIZE 1000000
struct {
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));
} d;

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        d.c[i] = d.a[i] + d.b[i];
    }
}
```

## Structures of Arrays (SoA) :

os vários elementos do mesmo campo são armazenados consecutivamente em memória.

Código vectorizável!



# Bloqueadores Auto-vectorização: dados contíguos

```
#define SIZE 10000000
struct {
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));
} d;

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        d.c[i] = d.a[i] + d.b[i];
    } }
```

```
loop:
    xor %eax, %eax
.L1:
    movaps d(%eax), %xmm0
    addps d+4000000(%eax), %xmm0
    movaps %xmm0, d+8000000(%eax)
    add $16, %eax
    cmp $4000000, %eax
    jl .L1
    ret
```

# Bloqueadores Auto-vectorização: *uncountable loops*

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; a[i]!=0 && i< SIZE ; i++) {
        c[i] = a[i] + b[i];
    } }
```

O número de iterações não pode ser computado  
(*uncountable loop*):  
Código não vectorizável!

# Bloqueadores Auto-vectorização: condições

```
#define SIZE 1000000
int a[SIZE] __attribute__((aligned(16)));
int b[SIZE] __attribute__((aligned(16)));
int c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        int s = a[i] + b[i];
        if (s<0) {c[i] = s;}
        else if (s==0) {c[i] = -10;}
        else {c[i] = -s;}
    } }
```

Estruturas condicionais:  
Código não vectorizável!

# Bloqueadores Auto-vectorização: condições

```
#define SIZE 1000000
int a[SIZE] __attribute__((aligned(16)));
int b[SIZE] __attribute__((aligned(16)));
int c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        int s = a[i] + b[i];
        c[i] = (s < 0 ? s : 0);
    } }
```

Estruturas condicionais realizável como uma máscara: Código vectorizável!

## **NOTA:**

s é calculado para todos os elementos do vector.

usando uma máscara só é atribuído aqueles elementos de c para os quais s é <0!



# Bloqueadores Auto-vectorização: funções

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        c[i] = myfunc(a[i]) + b[i];
    }
}
```

Invocação de funções dentro do ciclo:  
Código não vectorizável!

# Bloqueadores Auto-vectorização: funções

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));
float b[SIZE] __attribute__((aligned(16)));
float c[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE ; i++) {
        c[i] = absf(a[i]) + b[i];
    }
}
```

Invocação de funções intrínseca dentro do ciclo:  
Código vectorizável!

# Bloqueadores Auto-vectorização: *stride*

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=1 ; i< SIZE ; i+=2) {
        a[i] = a[i-1] + 1;
    } }
```

*Stride* != 1

Acessos **não contíguos**, mas **ordenados**.

Compilador pode não vectorizar o código.

Código (mesmo vectorial) menos eficiente, devido a acessos a memória e reduzida localidade espacial.

# Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=1 ; i< SIZE ; i++) {
        a[i] = a[i-1] + 1;
    }
}
```

Dependência *read after write* (RaW)!

Como  $i$  cresce, o valor de  $a[i+1]$  é alterado na próxima iteração anterior!

Código não vectorizável!

$a[1] = a[0] + 1;$

$a[2] = a[1] + 1;$

$a[3] = a[2] + 1;$

$a[4] = a[3] + 1;$

# Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
float a[SIZE] __attribute__((aligned(16)));

loop () {
    for (int i=0 ; i< SIZE-1 ; i++) {
        a[i] = a[i+1] + 1;
    }
}
```

Dependência write *after read* (WaR)!

Como  $i$  cresce, o valor de  $a[i+1]$  só será alterado na próxima iteração!

Código vetorizável!

$a[0] = a[1] + 1;$



$a[1] = a[2] + 1;$



$a[2] = a[3] + 1;$



$a[3] = a[4] + 1;$

# Bloqueadores Auto-vectorização: dependências

- Distância da dependência : diferença entre o índice de escrita e o índice de leitura

$$d = c^W - c^R$$

- Se  $d \leq 0$  não há dependências RaW : ciclo pode ser vectorizado

<pre>for (i=1 ; i &lt; SIZE ; i++) {     a[i] = 2 * a[i-1]; } </pre>	<pre>for (i=0 ; i &lt; SIZE-1 ; i++) {     a[i] = 2 * a[i+1]; } </pre>
<pre>d = i - (i-1) = 1 d&gt;0 =&gt; RaW</pre>	<pre>d = i - (i+1) = -1 D&lt;0 =&gt; WaR</pre>

- Nota: o sinal da distância deve respeitar a ordem de iteração.  
Isto é, se o índice for decrementado então  $d = -(c^W - c^R)$

# Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
int a[SIZE]
    __attribute__((aligned(16)));

loop () { int c;

    for (int i=1 ; i< SIZE ; i++) {
        c = a[i-1]*2 ;
        a[i] = (c >0 ? c : 1);
    } }
```

$$d = c^W - c^R = i - (i - 1) = 1$$

$a[1] = a[0]*2 : 1;$

$a[2] = a[1]*2 : 1;$

*read after write*

Código NÃO vectorizável!

```
#define SIZE 1000000
int a[SIZE]
    __attribute__((aligned(16)));

loop () { int c;

    for (int i=SIZE -1 ; i>0; i--) {
        c = a[i-1]*2 ;
        a[i] = (c >0 ? c : 1);
    } }
```

$$d = -(c^W - c^R) = -i + i - 1 = -1$$

$a[SIZE-1] = a[SIZE-2]*2:1;$

$a[SIZE-2] = a[SIZE-3]*2:1;$

*Write after read*

Código vectorizável!

# Processamento Vectorial: Linhas de Orientação

- Usar ciclos “for” contáveis: pontos únicos de entrada e saída;
- Evitar estruturas condicionais; máscaras são vectorizáveis, mas resultam na realização de trabalho inútil;
- Evitar dependências, especialmente do tipo “read-after-write”
- Evitar a utilização de apontadores e prevenir *aliasing*
- Usar acessos à memória eficientes:
  - Ciclo mais aninhado com *stride* 1 (dados consecutivos)
  - Minimizar acessos indirectos
  - Alinhar os dados a múltiplos de 16 (Intel® SSE)