Processamento Vectorial

Arquitetura de Computadores

Licenciatura em Engenharia Informática

Luís Paulo Santos

Material de Apoio

- "*Computer Systems: a Programmer's Perspective*"; Randal E. Bryant, David R. O'Hallaron--Pearson (3rd ed., 2016)
  Web aside: http://csapp.cs.cmu.edu/3e/waside/waside-simd.pdf

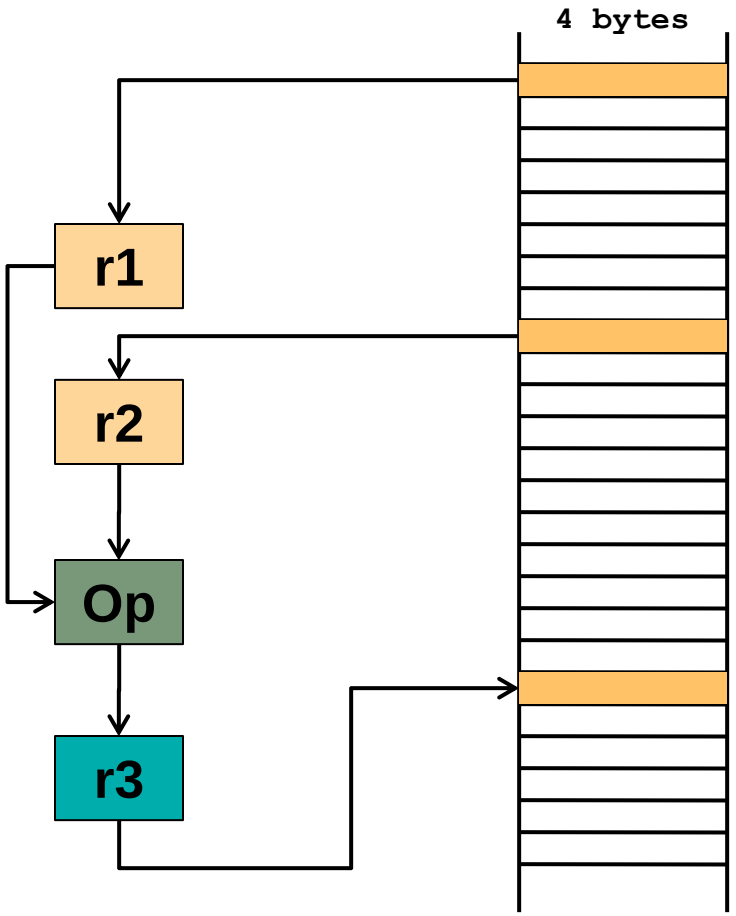
- "*Computer Organization and Design: The Hardware / Software Interface*" David A. Patterson, John L. Hennessy; 5th Edition, 2013
  - Secções 3.7 e 3.8 (págs. 224 a 228)
  - Secção 6.3 (págs. 509 a 515) – SISD, MIMD, SIMD, SPMD and Vector

Processamento Escalar

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

```
for (i=0 ; i < SIZE ; 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 SIZE, %ecx
  jl loop
```



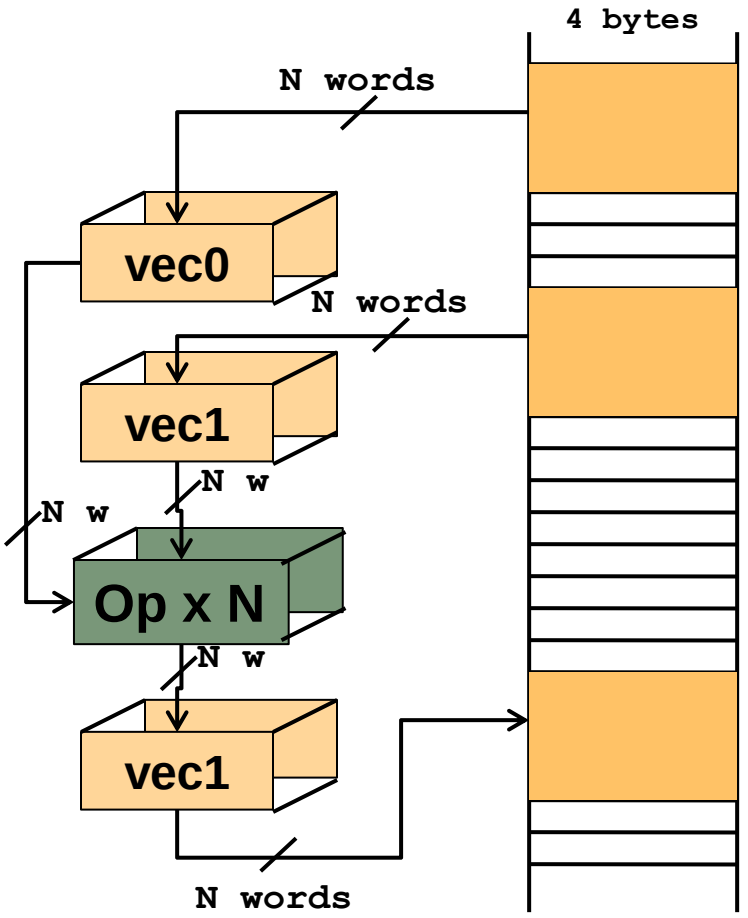
Processamento Vectorial

DATA LEVEL PARALLELISM

Cada instrução processa **N** elementos do conjunto de dados

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

```
loop:
  mov.v (%esi, %ecx, 4), %vec0
  mov.v (%edi, %ecx, 4), %vec1
  add.v %vec0, %vec1
  mov.v %vec1, (%ebx, %ecx, 4)
  addl $N, %ecx
  cmpl SIZE, %ecx
  jl loop
```



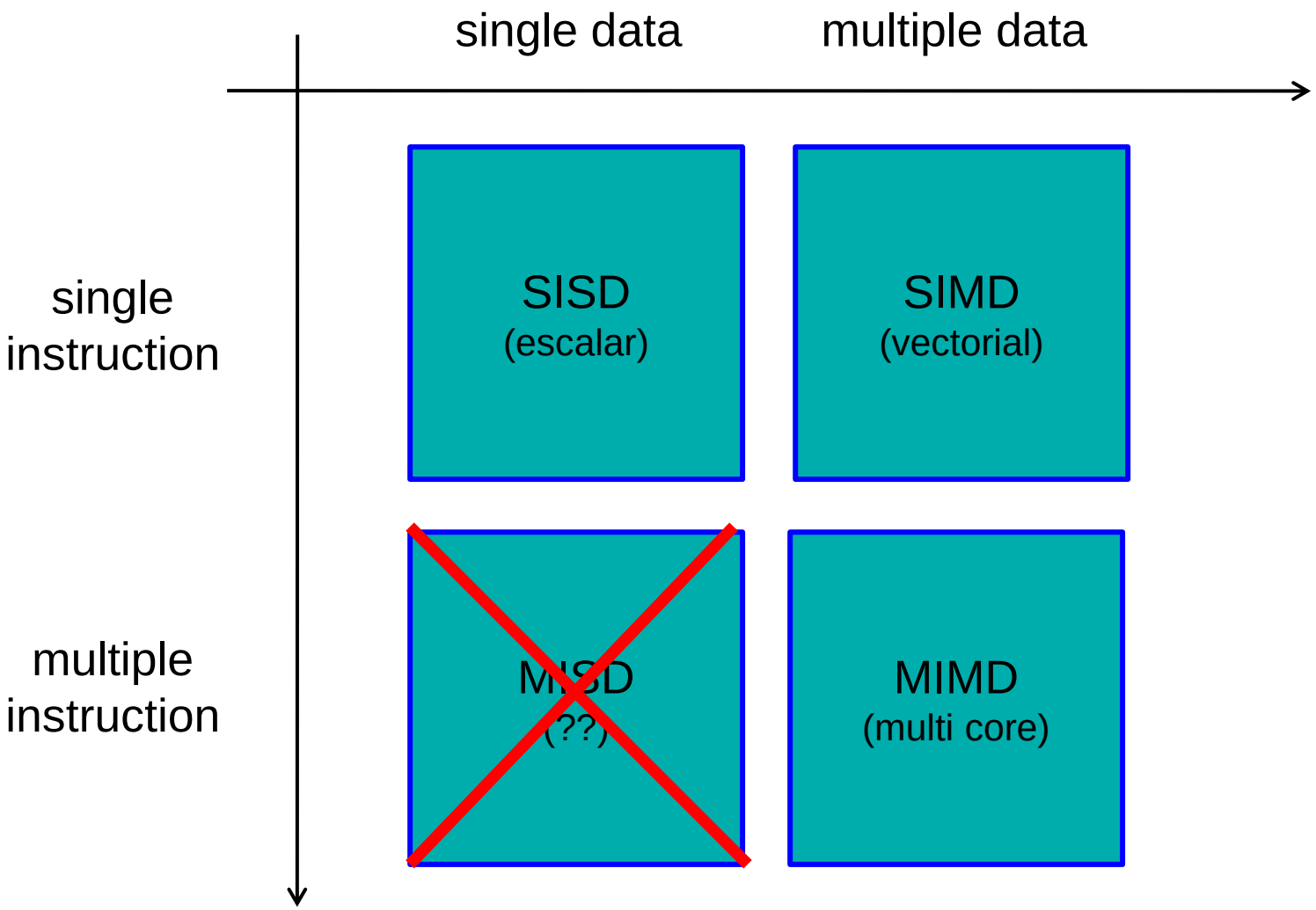
Processamento Vectorial

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

- **#I** – reduz, cada instrução processa **N** elementos de dados

- **CPI**
  - as unidades funcionais vectoriais realizam as **N** operações em paralelo, contribuindo para **manter o CPI**
  - **mas** a quantidade de dados a transferir de e para a memória por unidade de tempo (por instrução / ciclo do relógio) aumenta, contribuindo para **aumentar o CPI**

Paralelismo – Taxonomia de Flynn



Intel SSE

1994 – Pentium II e Pentium with MMX – MultiMedia eXtensions
8 registos de 64 bits (%mm0 .. %mm7) que mapeiam nos registos de vírgula flutuante (%st0 .. %st7) ; apenas operações sobre inteiros

1995 – Introdução de Streaming Simd Extensions (SSE) no Pentium III
8 novos registos de 128 bits (%xmm0 .. %xmm7) e operações FP

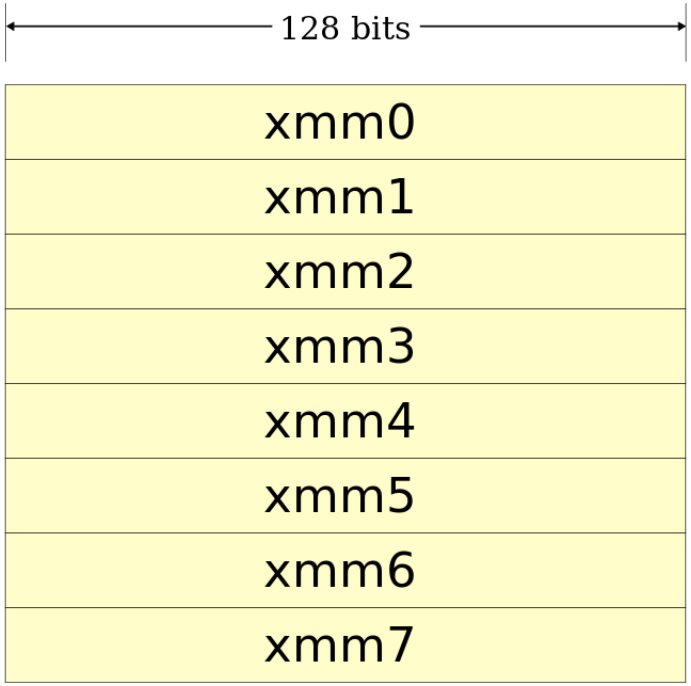
2000 – Introdução de SSE2 no Pentium IV

2004 - Introdução de SSE3 no Pentium IV HT

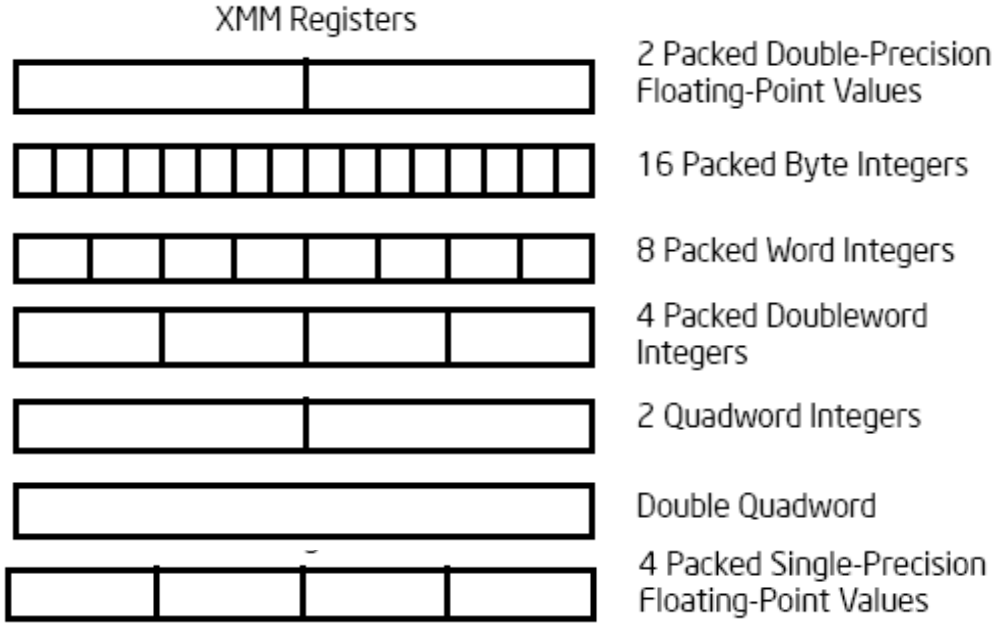
2007 - Introdução de SSE4

Intel SSE - Streaming SIMD Extensions

- SSE adiciona à arquitectura Intel 8 registos de 128 *bits: %xmm0 .. %xmm7*
- adiciona ainda instruções para operar sobre vectores de vários tipos de dados

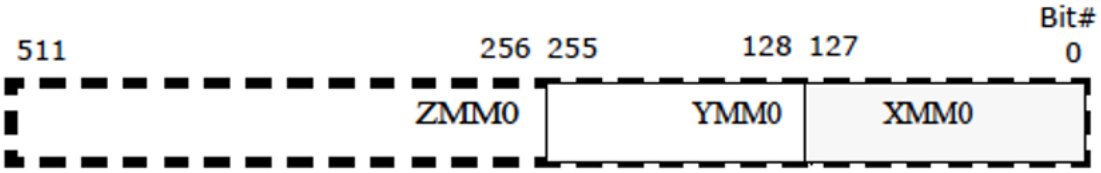


Tipos de Dados



Intel Advanced Vector Extensions (AVX)

- Intel AVX – 16 registos `YMM0..YMM15` de 256 *bits*
  incluindo todas as operações e tipos de dados SSE, referido como AVX128 (Sandy Bridge, 2011)
- Intel AVX2 – suporte para inteiros e *Fused-Multiply-Add (FMA)*
  (Haswell, 2013)
- Intel AVX512 – 32 registos `ZMM0..ZMM31` de 512 *bits*
  (Knights Landing, Xeon Phi, June, 2016; Intel Xeon Skylake)



YMMx : 8 SPFP (ou 8 int se AVX2)

SPFP7	SPFP6	SPFP5	SPFP4	SPFP3	SPFP2	SPFP1	SPFP0

YMMx : 4 DPFP

DPFP3	DPFP2	DPFP1	DPFP0

Instruções AVX: Notação

- O AVX introduz a notação de 3 operandos, previamente inexistente no x86:
  - 0 operandos: `ADD` - operandos no topo da stack
  - 1 operando: `ADD <op1>` - usa implicitamente um registo: o acumulador
  - 2 operandos: `ADD <op1>, <op2>` - x86: `<op2> = <op2> + <op1>`
  - 3 operandos: `ADD <op1>, <op2>, <op3>` - RISC, AVX: `<op3> =<op2> +<op1>`

- A operação de adição (`ADD`) é usada como exemplo
- `%xmm?` registos de 128 *bits*, `%ymm?` registos de 256 *bits* , `%zmm?` registos de 512 *bits*
- `m128, m256, m512` referem operandos em memória: 128, 256 e 512 *bits*
- Instruções AVX com o prefixo `V` usam o formato de três operandos:
  - `VADDPS %[x|y|z]mm?, %[x|y|z]mm? / m[128|256|512], %[x|y|z]mm?`
    - adiciona os 2 operandos da esquerda guarda o resultado no operando da **DIREITA**

Instruções AVX: Notação

- Instruções com o sufixo `S` operam sobre valores em vírgula flutuante precisão simples; o sufixo `D` indica vírgula flutuante precisão dupla:
  - `VADDP`**`S`** `%ymm?, %ymm? / m256, %ymm?` – realiza **8** operações em **SP**FP
  - `VADDP`**`D`** `%ymm?, %ymm? / m256, %ymm?` – realiza **4** operações em **DP**FP
- Muitas instruções admitem a forma escalar, isto é, apenas realizam **UMA** operação sobre o valor armazenado nos *bits* menos significativos dos operandos.
  O penúltimo caracter pode tomar os valores `S` ou `P`, para indicar operação escalar (**UMA ÚNICA OPERAÇÃO**) ou vectorial, respectivamente:
  - `VADD`**`P`**`S %ymm?, %ymm? / m256, %ymm?` – realiza **8** operações em SPFP
  - `VADD`**`P`**`D %ymm?, %ymm? / m256, %ymm?` – realiza **4** operações em DPFP
  - `VADD`**`S`**`S %ymm?, %ymm? / m256, %ymm?` – realiza **1** operação em SPFP
  - `VADD`**`S`**`D %ymm?, %ymm? / m256, %ymm?` – realiza **1** operação em DPFP

  Esta notação permite substituir o conjunto de instruções de vírgula flutuante classico do x86, orientado a uma stack de 8 registos, por uma alternativa mais flexível

Instruções AVX: Transferência de Dados

- `VMOV[A|U]P[S|D]`

  Mover vectores, representando valores SPFP ou DPFP (sufixo `S` ou `D`), de endereços alinhados ou não (modificador `A` ou `U`)
  - `VMOVUPD m256,%ymm?` – move 4 DPFP de memória (endereço não alinhado) para `%ymm?`
  - `VMOVAPD %xmm?, m128` – move 2 DPFP de `%xmm?` para memória (endereço alinhado)

- **Alinhamento:** um bloco de dados com **B** *bytes*, diz-se alinhado, se o endereço inicial desse bloco em memória é múltiplo de **B**.
- Acessos alinhados são **significativamente** mais eficazes do que acessos não alinhados.
- `AVX2` permite o uso de instruções `A` (*aligned*) com acessos não alinhados, com penalização no desempenho. `SSE` e `AVX` resulta numa excepção.

Instruções AVX: Operações FP

Instruções	Operandos
VADD[S|P][S|D] VSUB[S|P][S|D] VMUL[S|P][S|D] VDIV[S|P][S|D] VSQRT[S|P][S|D] VMAX[S|P][S|D] VMIN[S|P][S|D] VAND[S|P][S|D] VOR[S|P][S|D] ...	[S|P] : escalar ou vectorial ? [S|D] : SPFP ou DPFP ? Endereços em memória alinhados O resultado não pode ser em memória

Exemplo AVX

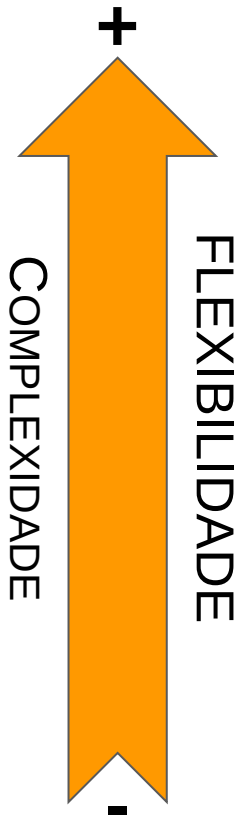
```
float a[1000] __attribute__ ((aligned(32)));
float b[1000] __attribute__ ((aligned(32)));
float r[1000] __attribute__ ((aligned(32)));

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

```
func:
  …
  movl 8(%rbp), %rdx    # n
  movl 12(rbp), %rax    # a
  movl 16(rbp), %rbx    # b
  movl 20(rbp), %rsi    # r
  movl $0, %rcx
ciclo:
  vmovaps (%rax, %rcx, 4), %ymm0
  vmulps (%rbx, %rcx, 4), %ymm0, %ymm1
  vmovaps %ymm1, (%rsi, %rcx, 4)
  addl $8, %rcx
  cmpl %rdx, %rcx
  jl 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)

Para detalhes ver **Intel Intrinsics Guide** (https://software.intel.com/sites/landingpage/IntrinsicsGuide/#)

Compiler Intrinsics

- As funções e tipos de dados definidos como *intrinsics* são acessíveis incluindo os *headers* apropriados:

xmmintrin.h	Streaming SIMD Extensions	SSE
emmintrin.h	Streaming SIMD Extensions 2	SSE2
pmmintrin.h	Streaming SIMD Extensions 3	SSE3
smmintrin.h	Streaming SIMD Extensions 4 (vector math)	SSE4.1
nmmintrin.h	Streaming SIMD Extensions 4 (string processing)	SSE4.2
immintrin.h	Advanced Vector Extensions 1 e 2	AVX2

Tipos de Dados	
__m64	Vector de 64 bits – inteiros (MMX)
__m128	Vector 128 *bits* – 4 FP SP (SSE)
__m256	Vector 256 *bits* – 8 FP SP (AVX)

Compiler Intrinsics

Operações Aritméticas (single FP)		
Pseudo-função	Descrição	Instrução
`__m256 _mm_add_ps (__m256, __m256)`	Adição	`VADDPS`
`__m256 _mm_sub_ps (__m256, __m256)`	Subtracção	`VSUBPS`
`__m256 _mm_mul_ps (__m256, __m256)`	Multiplicação	`VMULPS`
`__m256 _mm_div_ps (__m256, __m256)`	Divisão	`VDIVPS`
`__m256 _mm_sqrt_ps (__m256)`	Raiz Quadrada	`VSQRTPS`
`__m256 _mm_rcp_ps (__m256)`	Inverso	`VRCPPS`
`__m256 _mm_rsqrt_ps (__m256)`	Inverso Raiz Quadrada	`VRSQRTPS`

Compiler Intrinsics

Movimento de Dados (single FP)		
Pseudo-função	Descrição	Instrução
`__m256 _mm256_load_ps (float *)`	Carrega vector de memória para registo (alinhado 32)	`VMOVAPS`
`__m256 _mm_broadcast_ps (float *)`	Carrega 1 FP de memória para os 8 elementos do registo YMM	`VBROADCASTSS`
`_mm256_store_ps (float *, __m256)`	Escreve registo em vector de memória (alinhado 32)	`VMOVAPS`
`__m256 _mm256_set1_ps (float)`	Todos os 8 elementos do registo YMM são iniciados com o mesmo float	`---`

Compiler Intrinsics: Exemplo 1

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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

```
#include <immintrin.h>
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

func() {
  for (int i=0 ; i<SIZE ; i+=8) {
    __m256 mb = _mm256_load_ps (&b[i]);
    __m256 ma = _mm256_load_ps(&a[i]);
    __m256 mc = _mm256_add_ps (ma, mb);
    _mm256_store_ps (&c[i], mc);
} }
```

Compiler Intrinsics: Exemplo 2

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

float alfa;

func() {
  for (int i=0 ; i<
    c[i] = alfa * a
} }
```

```
#include <immintrin.h>
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));
float alfa;
func() {
  __m256 m_alfa = _mm256_broadcast_ps (&alfa);
  for (int i=0 ; i<SIZE ; i+=8) {
    __m256 mb = _mm256_load_ps (&b[i]);
    __m256 ma = _mm256_load_ps(&a[i]);
    ma = _mm256_mul_ps(ma, m_alfa);
    __m256 mc = _mm256_add_ps (ma, mb);
    _mm256_store_ps (&c[i], mc);
} }
```

Compiler Intrinsics: Exemplo 3

```
#include <math.h>
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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

```
#include <ia32intrin.h>
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

func() {
  __m256 cinco = _mm256_set1_ps (5.);
  for (int i=0 ; i<SIZE ; i+=8) {
    __m256 mb = _mm256_sqrt_ps(_mm256_load_ps (&b[i]));
    __m256 ma = _mm256_load_ps(&a[i]);
    __m256 mr = _mm256_mul_ps (cinco, _mm256_add_ps (ma, mb);
    _mm256_store_ps (&c[i], mr);
} }
```

Auto-vectorização

- O compilador pode vectorizar o código

- Comando gcc:

```
gcc –O3 –march=…
Ou
gcc –ftree-vectorize –march=….
```

Auto-vectorização

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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

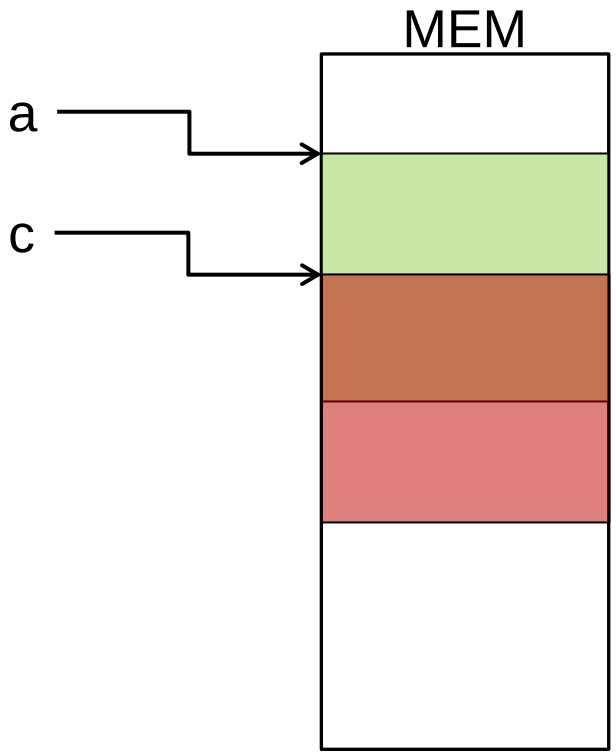
```
loop:
  xor %eax, %eax
.L1:
  vmovaps a(%eax), %ymm0
  vaddps b(%eax), %ymm0, %ymm0
  vmovaps %ymm0, c(%eax)
  add $32, %eax
  cmp $4000000, %eax
  jl .L1
  ret
```

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 gera 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
NÂO EXISTE QUALQUER OUTRA REFERÊNCIA
para a zona de memória acedida a partir desse apontador.
Logo não existe a possibilidade de *aliasing*

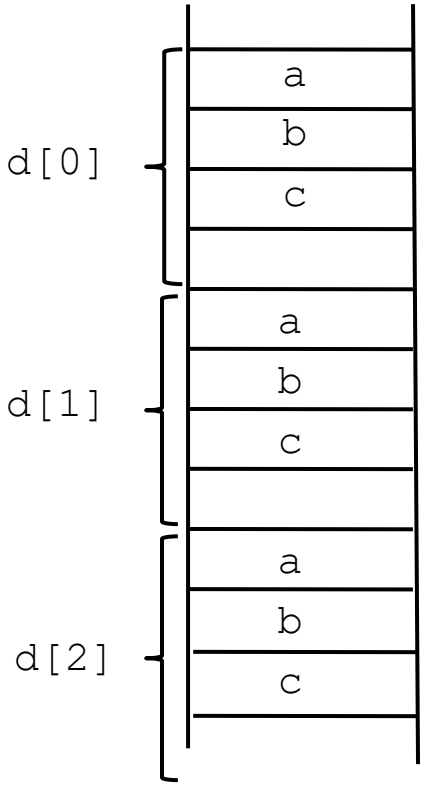
Bloqueadores Auto-vectorização: dados não contíguos

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

MYDATA d[SIZE] __attribute__ ((aligned(32)));

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 não vectorizável!

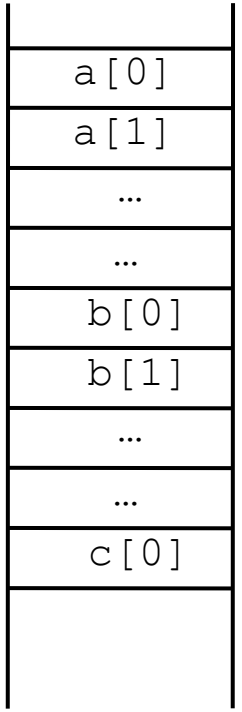


Bloqueadores Auto-vectorização: dados não contíguos

```
#define SIZE 1000000
struct {
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));
} 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 1000000
struct {
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));
} d;

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

```
loop:
  xor %eax, %eax
.L1:
  vmovaps d(%eax), %ymm0
  vaddps d+4000000(%eax), %ymm0
  vmovaps %ymm0, d+8000000(%eax)
  add $32, %eax
  cmp $4000000, %eax
  jl .L1
  ret
```

AoS versus SoA

- o mapeamento das estruturas de dados como AoS é mais natural para o programador e resulta em código mais legível
- a utilização de SoA resulta numa melhor utilização da memória (menos *padding*), maior localidade especial e permite vectorização

Alguma linguagens de programação recentes permitem indicar ao compilador qual o mapeamento desejado sem alterar a indexação da informação a tratar: ODIN (https://odin-lang.org/docs/overview/#soa-data-types)

```
Vector3 :: struct {x, y, z: f32}
N :: 2

v_aos: [N]Vector3
v_aos[0].x = 1 v_aos[0].y = 4
v_aos[0].z = 9
v_aos[1] = {0, 3, 4}
fmt.println(v_aos)
```

```
// Same syntax as AOS
//treat as if it was an array

v_soa: #soa[N]Vector3
v_soa[0].x = 1 v_soa[0].y = 4
v_soa[0].z = 9
v_aos[1] = {0, 3, 4}
fmt.println(v_aos)
```

Bloqueadores Auto-vectorização: *stride*

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));

loop () {
  for (int i=0 ; i< SIZE ; i+=2) {
    a[i] = a[i] + 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: *uncountable loops*

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

loop () {
  for (int i=0 ; i< SIZE ; i++) {
    float 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
float a[SIZE] __attribute__ ((aligned(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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

Algumas estruturas condicionais simples realizáveis como uma máscara*:* Código vectorizável nesses casos

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(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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(32)));
float b[SIZE] __attribute__ ((aligned(32)));
float c[SIZE] __attribute__ ((aligned(32)));

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

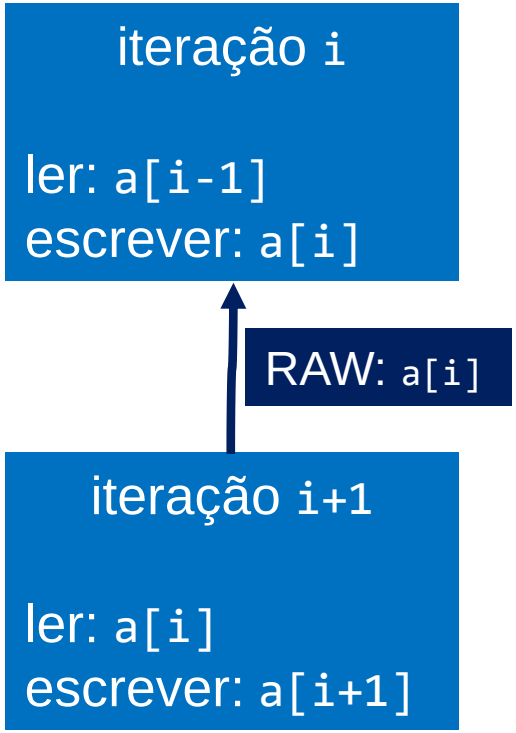
Invocação de funções intrínseca dentro do ciclo:
Código vectorizável!

Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));

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!

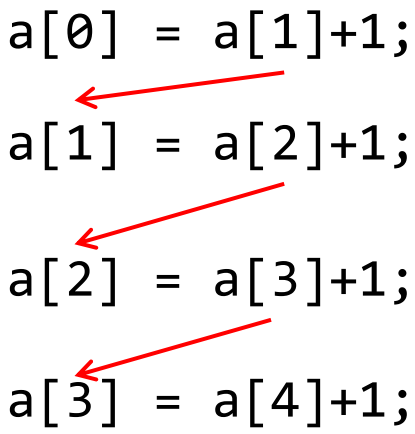


Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));

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 vectorizável!



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* <= 0 não há dependências RaW : ciclo pode ser vectorizado

`for (i=1 ; i < SIZE ; i++) { a[i] = 2 * a[i-1]; }`	**`for (i=0 ; i < SIZE-1 ; i++) { a[i] = 2 * a[i+1]; }`**
`d = i - (i-1) = 1` `d>0 => RaW`	`d = i - (i+1) = -1` `d<0 => WaR`

- 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
float a[SIZE]
  __attribute__ ((aligned(32)));

loop () { float 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
float a[SIZE]
  __attribute__ ((aligned(16)));

loop () { float 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!

Bloqueadores Auto-vectorização: dependências

```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));

loop () {
  for (int i=9 ; i< SIZE ; i++) {
    a[i] = a[i-9] + 1;
} }
```

Máquina AVX: largura das unidades funcionais W= 8

```
d = i – (i-9) = 9
d>0 => RaW

Mas d>W ; 9 > 8
```

Código vectorizável

Processamento Vectorial: Linhas de Orientação

- Usar ciclos “for” contáveis: pontos únicos de entrada e saída;
- Evitar estruturas condicionais; no entanto, máscaras são vectorizáveis;
- 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)
  - Alinhar os dados a múltiplos de 32 (Intel® AVX)