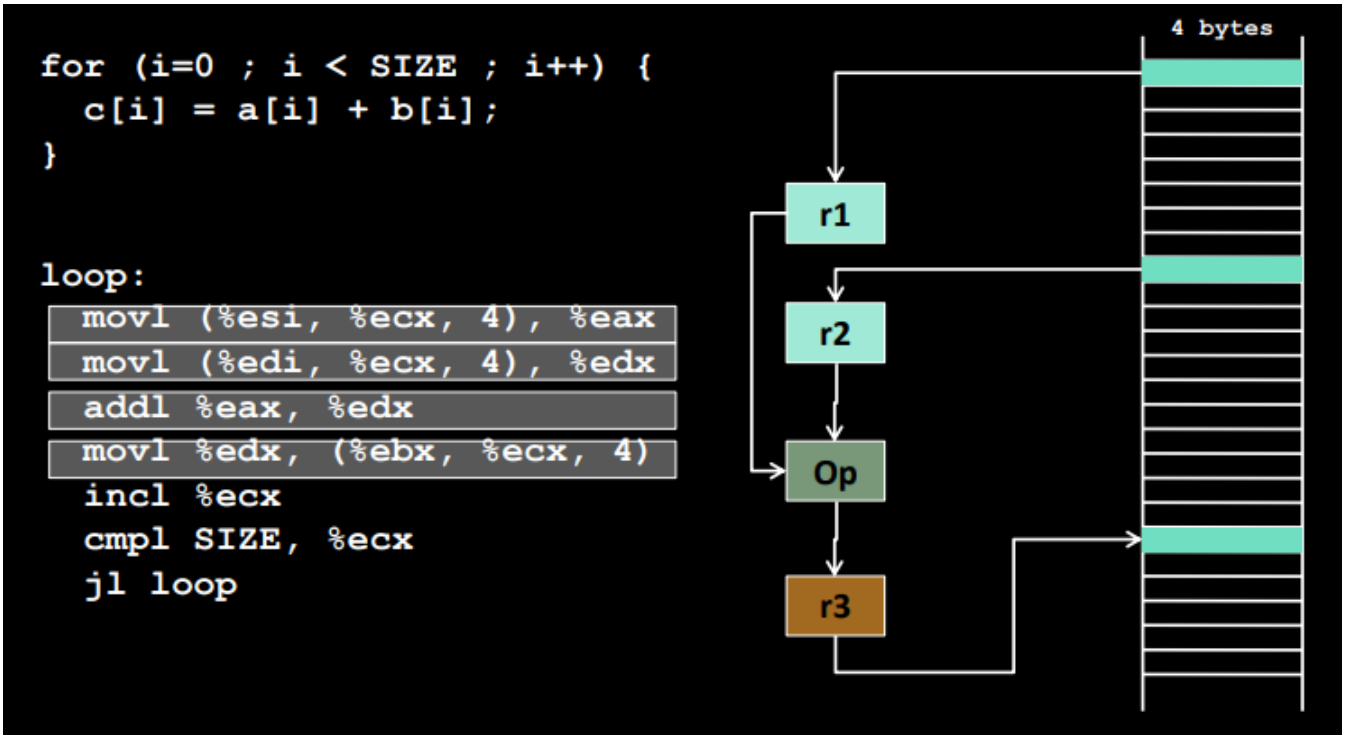
Processamento Vetorial

Data: 19-12-2022
PDF: 06-SuperEscalaridade.pdf
Tags: #ARQC #SoftwareEngineering #C

Processamento Escalar

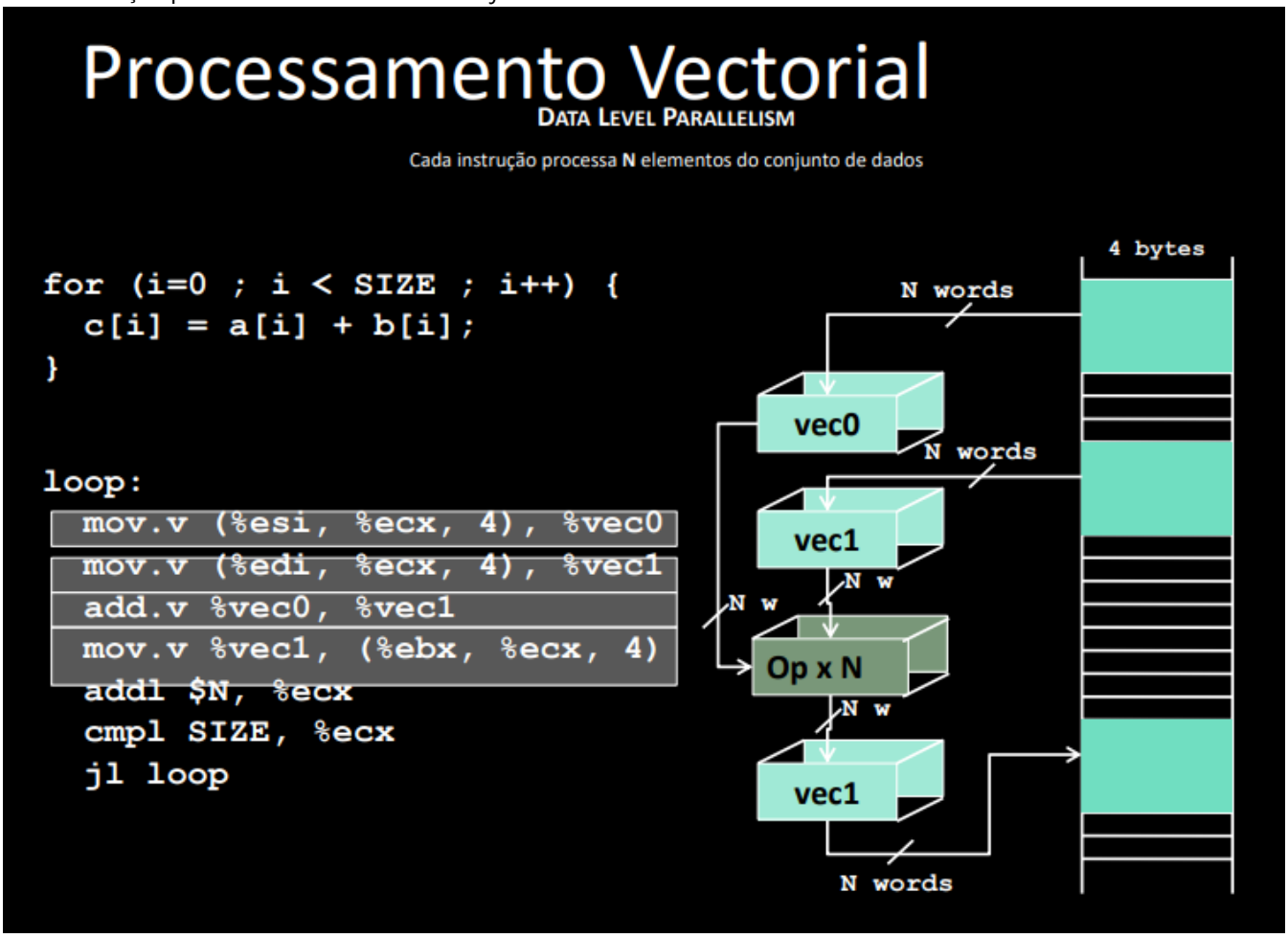
Cada instrução processa apenas **um elemento** do conjunto de dados.



Guarda o *int* $a[i]$ em *eax* e o *int* $b[i]$ em edx. De seguida, guarda o valor em $c[i]$.

Processamento Vetorial

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



O processador executa as instruções descritas anteriormente em paralelo.
Ou seja, atualizar valores em registos e somas em simultâneo.

Assim, o número de instruções reduz, visto que cada instrução executa N elementos de dados em vez de apenas um. Para além disso, o CPI tende a aumentar, porque:

- as unidades funcionais vetoriais realizam as N operações em paralelo, contribuindo para **manter** o CPI
- a quantidade de dados a transferir de e para a memória por unidade de tempo aumenta, contribuindo para aumentar o CPI

Taxonomia de Flynn

- SISD (Escalar: Single Instruction Single Data): Fluxo único de instruções sobre um único conjunto de dados.
- **SIMD** (Vetorial: Single Instruction Multiple Data): Fluxo único de instruções em múltiplos conjuntos de dados.
- MISD (Multiple Instruction Single Data): Fluxo múltiplo de instruções em um único conjunto de dados.
- MIMD (Multicore: Multiple Instruction Multiple Data): Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.

Intel SSE - Streaming SIMD Extensions

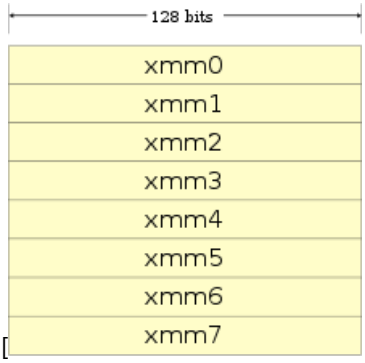
> ⓘ Info
>
> **Streaming SIMD Extensions** (**SSE**) is a single instruction, multiple data (SIMD) instruction set extension to the x86 architecture, designed by Intel and introduced in 1999

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 largely redundant, though further performance increases can be attained in some situations by using MMX in parallel with SSE operations.

Registers

SE originally added **eight new 128-bit registers** known as `XMM0` through `XMM7`. The AMD64 extensions from AMD (originally called *x86-64*) added a further eight registers `XMM8` through `XMM15`, and this extension is duplicated in the Intel 64 architecture. There is also a new 32-bit control/status register, `MXCSR`. The registers `XMM8` through `XMM15` are accessible only in 64-bit operating mode.

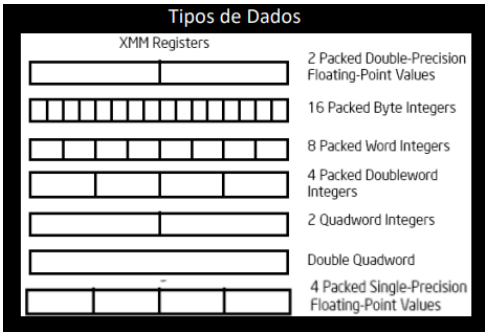


SSE used only a single data type for XMM registers:

- four 32-bit single-precision floating-point numbers

SSE2 would later expand the usage of the XMM registers to include:

- two 64-bit double-precision floating-point numbers or
- two 64-bit integers or
- four 32-bit integers or
- eight 16-bit short integers or
- sixteen 8-bit bytes or characters.



Because these 128-bit registers are additional machine states that the operating system must preserve across task switches, they are disabled by default until the operating system explicitly enables them. This means that the OS must know how to use the `FXSAVE` and `FXRSTOR` instructions, which is the extended pair of instructions that can save all x86 and SSE register states at once. This support was quickly added to all major IA-32 operating systems.

Intel Advanced Vector Extensions (AVX)

Notação

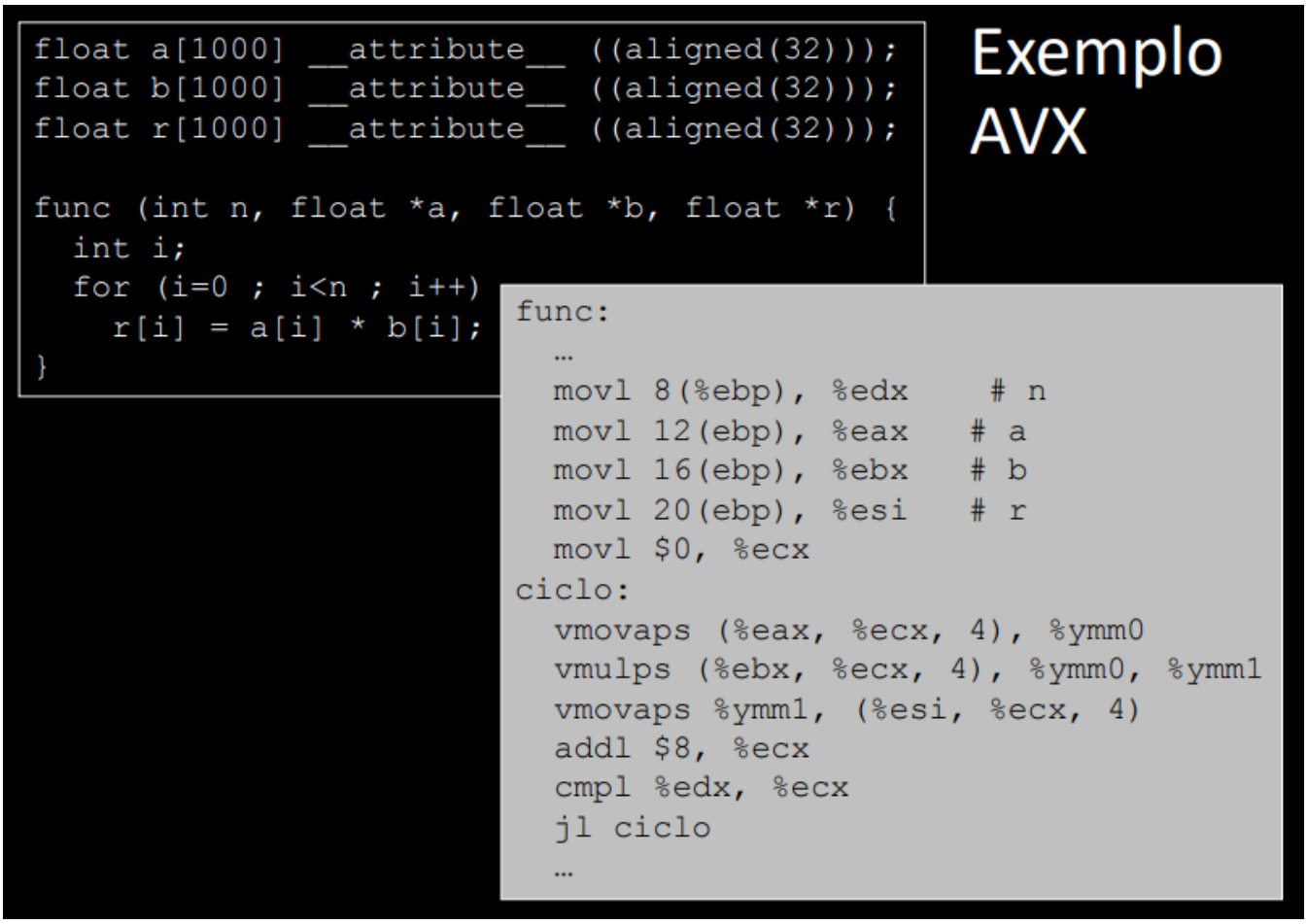
- `%xmm?` refere registos de 128 bits, e `%ymm?` registos de 156 bits
- `m128` refere 128 *bits* (16 *bytes*) em memória, `m256` refere 32 bytes em memória
- Instruções sem o prefixo `V` operam sobre quantidades de 128 bits e usam o formato de dois operandos:
  - `ADDPS %xmm? / m128, %xmm?`
    - adiciona o operando da esquerda com o da direita e guarda o resultado no operando da **direita**.

- Instruções com o prefixo `V` operam sobre quantidades de 128 ou 256 bits e usam o formato de três operandos:
  - `VADDPS %xmm?, %xmm? / m128, %xmm?`
  - `VADDPS %ymm?, %ymm? / m256, %ymm?`
    - adiciona os 2 operandos da esquerda e guarda o resultado no operando da **direita**.
- Instruções com o sufixo `S` operam sobre valores em vírgula flutuante precisão simples; o sufixo `D` indica vírgula flutuante precisão dupla:
  - `VADDPS %ymm?, %ymm? / m256, %ymm?` - realiza 8 operações em SPFP
  - `VADDPD %ymm?, %ymm? / m256, %ymm?` - realiza 4 operações em DPFP
- 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 `P` (uma única operação) ou vetorial `S`, respetivamente:
  - `VADDPS %ymm?, %ymm? / m256, %ymm?`
    - realiza 8 operações em SPFP (usa os 32 bytes dos registos)
  - `VADDPD %ymm?, %ymm? / m256, %ymm?`
    - realiza 4 operações DPFP (usa dos 32 bytes dos registos)
  - `VADDSS %ymm?, %ymm? / m256, %ymm?`
    - realiza 1 operação em SPDP (usa 4 bytes (0 .. 3) dos registos)
  - `VADDSD %ymm?, %ymm? / m256, %ymm?`
    - realiza 1 operação em DPDP (usa 8 bytes (0 .. 7) dos registos)

Transferência de dados

- `[V] MOV [A|U] P [S|D]`
  - Mover quantidades de 128 ou 256 bits (prefixo `V`), 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?`
    - `MOVAPD %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 exceção.

Instruções	Operandos
[V]ADD[S|P][S|D] [V]SUB[S|P][S|D] [V]MUL[S|P][S|D] [V]DIV[S|P][S|D] [V]SQRT[S|P][S|D] [V]MAX[S|P][S|D] [V]MIN[S|P][S|D] [V]AND[S|P][S|D] [V]OR[S|P][S|D] ...	Sem V? : xmm/m128, xmm Com V? : [x|y]mm, [x|y]mm/m[128|256], [x|y]mm [S|P] : escalar ou vectorial ? [S|D] : SPFP ou DPFP ? Endereços em memória alinhados O resultado não pode ser em memória



Processamento vetorial - desenvolvimento

- **Assembly**: utilização direta de instruções assembly
- **Compilier Intrinsics**: pseudo-funções disponibilizadas pelo compilador que permitem o desenvolvimento explícito de código vetorial a um nível semântico mais elevado que o assembly
- **Auto vetorização**: vetorização pelo compilador

Compiler Intrinsics

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

`gcc -march=haswell -O3`

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)

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

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	---

Video

Exemplos

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);
} }
```

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

float alfa=2.3f;

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 =2.3f;
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);
} }
```

```
#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-vetorização

O compilador pode vetorizar o código:
`gcc -O3 -march=...` ou `gcc -ftree-vectorize -march=...`.

```
#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
```

```
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 vetoriais do ciclo e código para verificar o aliasing. Em *runtime* é escolhida a versão mais apropriada do ciclo.

```
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*.

Bloqueadores Auto-vetorização

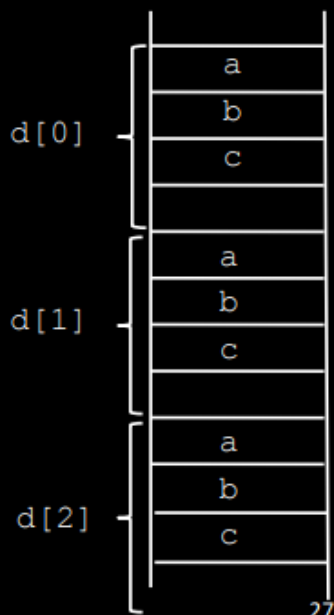
Dados não contíguos

```
#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;
} }
```

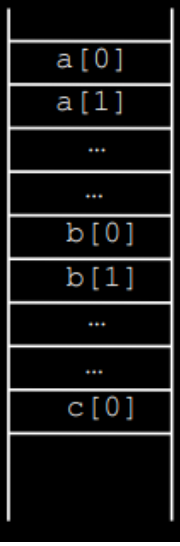
Array of Structures (AoS) :
os vários elementos do mesmo campo não são armazenados consecutivamente em memória.
Código não vectorizável!



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!

```
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
```

Stride

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

loop () {
  for (int i=0 ; i< SIZE ; i+=2) {
    a[i] = a[i] + 1;
} }
```

$Stride \neq 1 \implies$ Acessos **não contíguos**, mas **ordenados**.
Compilador pode não vetorizar o código.
Código (mesmo vetorial) menos eficiente, devido a acessos a memória e reduzida localidade espacial.

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, logo o código não é vetorizável.

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 {c[i] = 0.f;}
} }
```

Estruturas condicionais $\Longrightarrow$ Código não vetorizável.

```
#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 vetorizável nesses casos.

> Note
>
> `s` é calculado para todos os elementos do vetor.
> Usando uma máscara só é atribuído aqueles elementos de c para os quais s é menor do que 0.

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 $\Longrightarrow$ código não vetorizável.

```
#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];
} }
```

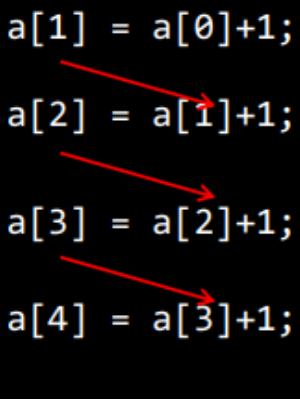
Caso especial: Invocação de funções intrínsecas dentro do ciclo $\Longrightarrow$ Vetorizável.

Dependências

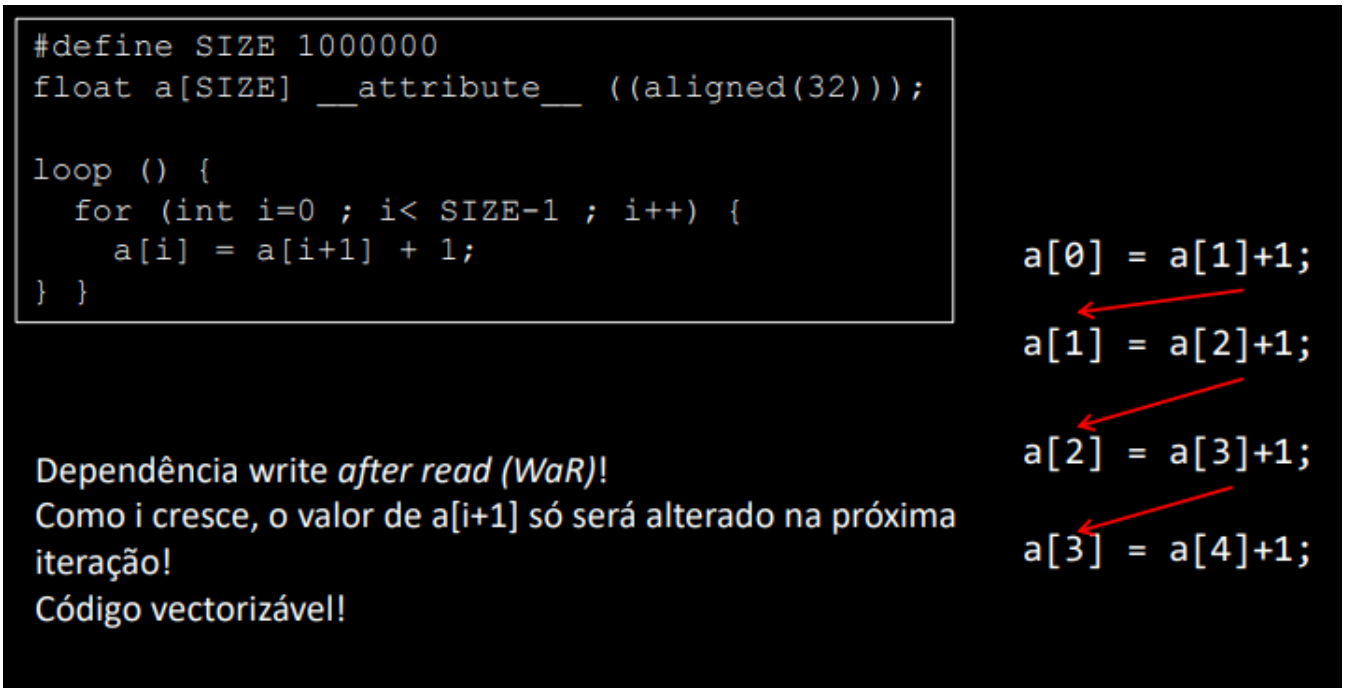
```
#define SIZE 1000000
float a[SIZE] __attribute__ ((aligned(32)));

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 iteração anterior!
Logo os processamentos de a[i] e a[i-1] **não podem ocorrer em paralelo**.
Código não vectorizável!



As instruções do cilco não podem ocorrer em paralelo visto que as instruções dependeriam de valores que estão a ser computados.



Como $a[i+1]$ só é alterado na próxima iteração, este dado pode ser acedido em instruções em paralelo.

- 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 vetorizado.

`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`

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

```
#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!

```
#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