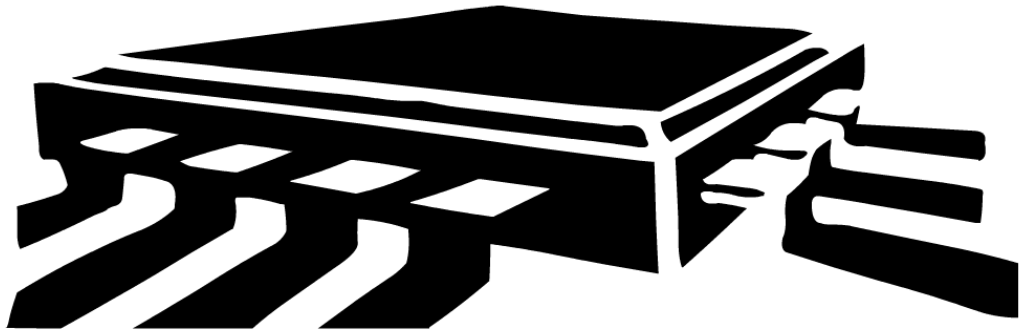

O que é Vetorização (SIMD)? As instruções das famílias MMX e SSE.

Pedro Garcia

http://www.sawp.com.br

12 de Fevereiro de 2010

Níveis de Paralelização

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD) – Vetorização
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

O que é Vetorização?

- Processamento de uma única instrução é aplicado à vários dados ordenados em forma de um vetor.
- Muito utilizado para modelos de dados que utilizam operações sobre dados vetores (Álgebra Linear): Soma de vetores, multiplicação por escalar, produto interno, etc. **Normalmente encontrado em vários problemas de multimídia (imagem, som, vídeo)**.
- A ideia é utilizar um único ciclo de processamento de instrução para processar todo o conjunto de dados sobre um vetor, gerando outro vetor: **S**ingle **I**nstruction, **M**ultiple **D**ata.

SIMD

- A forma simples de operar vetorização é atuar sobre tipos de dados inteiros, além de ter uma grande aplicabilidade em problemas de multimídia (processar cores RGB, por exemplo).
- As operações são realizadas sobre dados “empacotados” (*packed*).
- As duas principais famílias de instruções vetoriais da arquitetura Intel são: MMX e SSE.
  - MMX: desenvolvida para o Pentium II foi a primeira tecnologia desenvolvida para suportar SIMD em x86.
  - SSE: Enquanto a tecnologia MMX suportava vetorização apenas para aritmética inteira, SSE veio para expandir este recurso para aritmética de ponto flutuante.

Tecnologias SIMD da Intel

- MMX:
  - Modelo de Execução **SIMD** para dados do tipo **inteiro**
  - Primeira geração de tecnologia vetorial
  - Como tudo que é da Intel (e normalmente, de mercado), foi **mantida para evitar problema de incompatibilidade de aplicações antigas em máquinas novas**.

Tecnologias SIMD da Intel

- MMX:
  - Modelo de Execução **SIMD** para dados do tipo **inteiro**
  - Primeira geração de tecnologia vetorial
  - Como tudo que é da Intel (e normalmente, de mercado), foi **mantida para evitar problema de incompatibilidade de aplicações antigas em máquinas novas**.
- SSE:
  - Introduzida para prover **SIMD** na aritmética de ponto flutuante.
  - Implementada no Pentium III, já utilizava um conjunto de registradores próprios (XMM de 128 bits).
  - Processava tipos de ponto flutuantes de precisão simples (float) **apenas**.

Tecnologias SIMD da Intel

Tecnologias SIMD da Intel

- MMX + SSE:
  - Introduzida para prover **SIMD** na aritmética de ponto flutuante.
  - **Pentium III age**: SSE foi introduzida inicialmente para prover vetorização para aritmética de ponto flutuante, a de inteiros ficava a cargo das instruções MMX.

Tecnologias SIMD da Intel

- MMX + SSE:
  - Introduzida para prover **SIMD** na aritmética de ponto flutuante.
  - **Pentium III age**: SSE foi introduzida inicialmente para prover vetorização para aritmética de ponto flutuante, a de inteiros ficava a cargo das instruções MMX.
- SSE2:
  - A segunda implementação foi introduzida no Pentium 4.
  - Principal mudança: suporte adicional de tipos (SSE começa a suportar inteiros).
    - Inteiros: Opera sobre 16 bytes (byte), 8 shorts (word), 4 ints (doubleword) e 2 longs (quadword).
    - Ponto Flutuante: opera sobre 4 floats de 32 bits ou 2 doubles de 64 bits.

Tecnologias SIMD da Intel

- MMX + SSE:
  - Introduzida para prover **SIMD** na aritmética de ponto flutuante.
  - **Pentium III age**: SSE foi introduzida inicialmente para prover vetorização para aritmética de ponto flutuante, a de inteiros ficava a cargo das instruções MMX.
- SSE2:
  - A segunda implementação foi introduzida no Pentium 4.
  - Principal mudança: suporte adicional de tipos (SSE começa a suportar inteiros).
    - Inteiros: Opera sobre 16 bytes (byte), 8 shorts (word), 4 ints (doubleword) e 2 longs (quadword).
    - Ponto Flutuante: opera sobre 4 floats de 32 bits ou 2 doubles de 64 bits.
- SSE3:
  - Adiciona novas instruções para processar os dados incluídos no SSE2.
  - A família SSE, desde então, suporta toda classe de dados vetorizáveis, tornando a MMX obsoleta.

Tecnologias SIMD da Intel

Não utilize MMX em seus novos programas. [1]

[1] MMX, assim como todo recurso obsoleto da Intel, foi mantido para permitir que programas escritos com esta tecnologia se mantivessem compatíveis nas novas máquinas.

Uma breve história sobre a FPU

- A FPU é formada por **11** registradores
  - **3** de controle de 16 bits
    - Control
    - Status
    - Tags
  - **8** de dados com 80 bits
    - st0
    - st1
    - ...
    - st7

- Os registradores de dados operam como uma **pilha**. Ou seja, após a instrução operar sobre os dados, em geral, o resultado fica armazenado no registrador **`st0`**. Além disso, em geral, as operações de troca de dados entre a FPU e a memória são feitas apenas com o registrador **st0**. Quando queremos retirar algum dado que não esteja neste registrador precisamos reordenar estes dados na memória.

MMX - **M**ulti**M**edia e**X**tension

- A principal proposta da tecnologia MMX é operar sobre dados do tipo **inteiro**.
- Simula registradores de **64 bits**, aproveitando os **8** registradores de 80 bits da *FPU Stack* para armazenar os dados vetorizados (empacotados).
- Provê três tipos de dados *"packed"* (todos inteiros):
  - "empacota" 8 *bytes* dentro do registrador de 64 bits (1 *byte* = 8 bits)
  - "empacota" 4 *words* dentro do registrador de 64 bits (1 *word* = 16 bits)
  - "empacota" 2 *doublewords* dentro do registrador de 64 bits (1 *doubleword* = 32 bits)

MMX - **M**ulti**M**edia e**X**tension

- Os registradores da tecnologia MMX são referenciados através dos nomes **mm** seguidos de seu número. Ou seja, são os registradores MMX: **%mm0, %mm1, %mm2,** . . ., **%mm7**.
- Como os dados devem ser enviados diretamente para os registradores de 64 bits, ao invés de passar os dados dos registradores da CPU – %eax, %ebx, etc – é melhor passar os dados vetoriais diretamente da memória para os registradores MMX.

MMX - Usando MMX

Para utilizar a arquitetura MMX, é preciso apenas seguir os seguintes passos:

1. “Empacotar” os valores inteiros.

MMX - Usando MMX

Para utilizar a arquitetura MMX, é preciso apenas seguir os seguintes passos:

1. “Empacotar” os valores inteiros.
2. Carregar os valores empacotados dentro do registrador MMX.

MMX - Usando MMX

Para utilizar a arquitetura MMX, é preciso apenas seguir os seguintes passos:

1. “Empacotar” os valores inteiros.
2. Carregar os valores empacotados dentro do registrador MMX.
3. Realizar a operação vetorial desejada.

MMX - Usando MMX

Para utilizar a arquitetura MMX, é preciso apenas seguir os seguintes passos:

1. “Empacotar” os valores inteiros.
2. Carregar os valores empacotados dentro do registrador MMX.
3. Realizar a operação vetorial desejada.
4. “Desempacotar” os valores (retornar do registrador para uma região de memória.

MMX - Usando MMX

Exemplo do processo descrito

```
1  .section .data
2     unpack1:
3        .int 1000
4     unpack3:
5        .int 2000
6     unpack2:
7        .int -1024
8     unpack4:
9        .int -2048
10
11 .section .bss
12     .lcomm pack1, 8       # 8 bytes == 64 bits == size of MMX reg.
13     .lcomm pack2, 8
14     .lcomm unpackresult, 8
15
16 .section .text
17 .globl _start
```

MMX - Usando MMX

Exemplo do processo descrito (cont. 1)

```
1  _start:
2      # (1) pack values
3      movl $0, %edi
4      movl unpack1, pack1(, %edi, 4)
5      addl $4, %edi
6      movl unpack2, pack1(, %edi, 4)
7
8      movl $0, %edi
9      movl unpack3, pack2(, %edi, 4)
10     addl $4, %edi
11     movl unpack4, pack2(, %edi, 4)
12
13     # (2) Load the packed values to MMX registers
14     movq pack1, %mm0
15     movq pack2, %mm1
16
17     # (3) Perform SIMD MMX operation on packed data
18     paddd %mm1, %mm0      # ‘‘Packed ADD Double’’, %mm0 := %mm0 + %mm1
19
20     # (4) Unpack values in memory
21     movq %mm0, unpackresult
```

SSE - Streaming SIMD Extension

- SSE:
  - A principal proposta da tecnologia SSE é realizar operações vetoriais sobre dados de ponto flutuante.
  - Ao invés de reaproveitar registradores de outros circuitos, a arquitetura ganhou um conjunto de **oito** novos registradores de **128 bits**.
  - A referência aos registradores SSE é feita da mesma forma que para os MMX. Contudo, os registradores SSE são denominados XMM, seguido de seu número. Ou seja, são os registradores SSE: **%xmm0, %xmm1,** ..., **%xmm7** [2]

[2] O nome **xmm** dos registradores SSE vêm de e**X**tended **M**ulti**M**edia

Verificando o Teorema de De Morgan usando MMX

Cabeçalhos

```
1  .section .data
2  .align 16
3  head:
4      .asciz "\x1b[H\x1b[2JTrue Tables from De Morgan Theorem (With MMX).\
5              \nAuthor: Pedro Garcia."
6  first_demorgan:
7      .asciz "\n1th Thorem: !(A & B) == !A | !B\n"
8  second_demorgan:
9      .asciz "\n2th Theorem: !(A | B) == !A & !B\n"
10 first_true_table_head:
11     .asciz "| A      B |  !(A.B)  |  !A + !B |\n"
12 second_true_table_head:
13     .asciz "| A      B |  !(A+B)  |  !A . !B |\n"
14 true_table_fmt:
15     .asciz "| %2d    %2d |    %2d    |     %2d   |\n"
16 A:
17     .short 0, 0, -1, -1
18 B:
19     .short 0, -1, 0, -1
20 TRUE:
21     .short -1, -1, -1, -1
22 .section .bss
23     .lcomm column1, 8
24     .lcomm column2, 8
```

Verificando o Teorema de De Morgan usando MMX

Função main

```
1  .section .text
2  .globl main
3  main:
4      pushl   %ebp
5      movl    %esp, %ebp
6      call    _preamble
7
8      # 1th De Morgan Theorem
9      call _first
10
11     # 2th De Morgan Theorem
12     call _second
13
14     leave
15     ret
```

Verificando o Teorema de De Morgan usando MMX

_first

```
1  _first:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $8, %esp
5
6      movq    A, %mm0
7      movq    B, %mm1
8
9      # calcula !(A & B)
10     movq    %mm0, %mm2          # %mm2 := A
11     pand    %mm1, %mm2          # %mm2 := (A & B)
12     pxor    TRUE, %mm2          # %mm2 := !(A & B) == (A & B) ^ 1
13
14     # calcula !A | !B
15     movq    %mm0, %mm3          # %mm3 := A
16     pxor    TRUE, %mm3          # %mm3 := %mm3 ^ 1 == !A
17     movq    %mm1, %mm4          # %mm4 := B
18     pxor    TRUE, %mm4          # %mm4 := %mm4 ^ 1 == !B
19     por     %mm4, %mm3          # %mm3 := %mm3 | %mm4 == !A | !B
```

Verificando o Teorema de De Morgan usando MMX

_first (continuacao)

```
1     # get only the signal
2     psrlw   $15, %mm2
3     psrlw   $15, %mm3
4
5     # send results to buffer (used to print results)
6     movq    %mm2, column1
7     movq    %mm3, column2
8
9     call _print_first        # print the first De Morgan theorem tables
10
11    leave
12    ret
```

Verificando o Teorema de De Morgan usando MMX

_print_first

```
1  _print_first:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $64, %esp
5
6      movl    $first_demorgan, (%esp)
7      call    puts
8
9      movl    $first_true_table_head, (%esp)
10     call    printf
11
12     movl    $true_table_fmt, (%esp)
13     movl    $0, %edi
```

Verificando o Teorema de De Morgan usando MMX

_print_first (continuação)

```
1   loop:
2     movl    $0, %eax
3     movl    $0, %ebx
4     movl    $0, %ecx
5     movl    $0, %edx
6
7     movsx   A(, %edi, 2), %ea    # move 16 bits to a 32 bits register
8     movsx   B(, %edi, 2), %ebx
9     movsx   column1(, %edi, 2), %ecx
10    movsx   column2(, %edi, 2), %edx
11
12    shrl    $31, %eax            # select only the significant bit
13    shrl    $31, %ebx            # if 1, is true, if 0, false
14
15    movl    %eax, 4(%esp)
16    movl    %ebx, 8(%esp)
17    movl    %ecx, 12(%esp)
18    movl    %edx, 16(%esp)
19
20    call    printf
21
22    inc %edi
23    cmpl $4, %edi
24    jne loop
```

Verificando o Teorema de De Morgan usando MMX

O código para o segundo teorema de De Morgan apenas troca as instruções `por` por `pand` e vice-versa.

Saída dos dados

$ ./demorgan

```
True Tables from De Morgan Theorem (With MMX).\n
Author: Pedro Garcia.

1th Thorem: !(A & B) == !A | !B

| A      B |  !(A.B)  |  !A + !B |
|  0     0 |     1    |      1   |
|  0     1 |     1    |      1   |
|  1     0 |     1    |      1   |
|  1     1 |     0    |      0   |

2th Theorem: !(A | B) == !A & !B

| A      B |  !(A+B)  |  !A . !B |
|  0     0 |     1    |      1   |
|  0     1 |     0    |      0   |
|  1     0 |     0    |      0   |
|  1     1 |     0    |      0   |
```

Produto interno de dois vetores usando SSE.

Cabeçalhos

```
1  .section .data
2  head:
3      .asciz "\x1b[H\x1b[2JPerform the dot product between two vectors A and B.\
4              \nAuthor: Pedro Garcia.\n"
5  block:
6      .int 4
7  fmt:
8      .asciz "Dot product: A.B = %f\n"
9  A:
10     .asciz "A = [-1.2, -1.3, -1.14, 00.1, 1.2, 2, 4, 7]"
11 B:
12     .asciz "B = [12, 1.4, -1.1, 100, 12, 13, 1, 0]\n"
13
14 .align 32
15 vectorA:
16     .float -1.2, -1.3, -1.14, 00.1, 1.2, 2, 4, 7
17 .align 32
18 vectorB:
19     .float 12, 1.4, -1.1, 100, 12, 13, 1, 0
20
21 .section .bss
22     .lcomm tempbuffer, 32
23     .lcomm result, 4
```

Produto interno de dois vetores usando SSE.

Função main

```
1  .section .text
2  .globl main
3
4  main:
5      pushl   %ebp
6      movl    %esp, %ebp
7      call    _preamble
8
9      movl    $0, %edi
10     movups  vectorA(,%edi,4), %xmm0
11     movups  vectorB(,%edi,4), %xmm1
12     mulps  %xmm1, %xmm0
13
14     addl    $4, %edi
15     movups  vectorA(,%edi,4), %xmm2
16     movups  vectorB(,%edi,4), %xmm3
17     mulps  %xmm3, %xmm2
```

Produto interno de dois vetores usando SSE.

Função main (continuação)

```
1     movl    $0, %edi
2     movups  %xmm0, tempbuffer(,%edi,4)
3     addl    $4, %edi
4     movups  %xmm2, tempbuffer(,%edi,4)
5
6     movl    $0, %edi
7
8     finit
9     fldz
10
11    loop:
12      fadd    tempbuffer(,%edi,4)
13
14      inc %edi
15      cmpl $8, %edi
16      jne loop
17
18    call _print
19
20    leave
21    ret
```

Produto interno de dois vetores usando SSE.

_preamble

```
1  _preamble:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $8, %esp
5
6      movl    $head, (%esp)
7      call    puts
8
9      leave
10     ret
```

Produto interno de dois vetores usando SSE.

_print

```
1  _print:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $64, %esp
5
6      movl    $A, (%esp)
7      call puts
8
9      movl    $B, (%esp)
10     call puts
11
12     movl    $fmt, (%esp)
13     fstl    4(%esp)
14     call    printf
15
16     breka:
17     nop
18
19     leave
20     ret
```

Saída dos dados

$ ./dotproduct

```
Perform the dot product between two vectors A and B.\n
Author: Pedro Garcia.

A = [-1.2, -1.3, -1.14, 00.1, 1.2, 2, 4, 7]
B = [12, 1.4, -1.1, 100, 12, 13, 1, 0]

Dot product: A.B = 39.434000
```

Dúvidas,

sugestões,

reclamações?

Referências

- IDOETA, I.V. *Elementos de Eletrônica Digital*. 35 ed. Érica. São Paulo, 1998. **ISBN 85-7194-019-3**.
- BLUM, R. *Professional Assembly Language*. Wiley Publishing. Indianápolis (EUA), 2005. **ISBN 0-7645-7901-0**.
- Sun Microsystems. *x86 Assembly Language Reference Manual of the SolarisTM x86*. 4150 Network Circle, Santa Clara, CA 95054 (EUA), 2005. **Part No: 817–5477–10**