Extensões vetoriais do gcc

Paulo Matias

Arquitetura de Computadores II - 2010

Instituto de Física de São Carlos
Universidade de São Paulo

Programa de Aperfeiçoamento de Ensino
PAE - CAPES

1 Introdução

A introdução de arquiteturas SIMD (*Single Instruction Multiple Data*) foi uma das primeiras medidas historicamente efetivas para paralelizar aplicações de computação científica, pois seu uso é conceitualmente intuitivo e natural para físicos e outros cientistas, e a presença de operações vetoriais é intensa em praticamente todas as aplicações da área.

Entretanto, linguagens de programação largamente usadas, como a linguagem C, tradicionalmente não permitem expressar cálculos em notação vetorial, sendo necessário expressar as operações "manualmente", com o uso de loops. Para poder se aproveitar de arquiteturas SIMD, então, o compilador precisa detectar loops que possam ser expressos em termos de operações vetoriais (processo chamado de "vetorização automatica").

Entretanto, o processo de vetorização automatica não é perfeito. Portanto, o ideal seria que a linguagem C permitisse expressar operações vetoriais diretamente, que poderiam então ser traduzidas facilmente para instruções SIMD do processador.

Os desenvolvedores do gcc tiveram essa mesma ideia. Eles criaram extensões para a linguagem C que permitem expressar algumas operações vetoriais. Vale lembrar que essas extensões não pertencem ao padrão da linguagem C. Elas são específicas do gcc.

2 Programação SIMD portável

Começaremos apresentando uma forma de criar programas com extensões vetoriais que possam ser compilados em qualquer arquitetura de processador que o gcc suporte. Por exemplo, na arquitetura Intel, o gcc compilará o programa para se aproveitar das instruções SSE e MMX. Na arquitetura PowerPC, o mesmo

programa será compilado para utilizar as instruções Altivec. Em um microcontrolador que não tenha unidade SIMD (por exemplo um ARM Cortex-M0), o gcc transformará essas operações em loops, permitindo que o mesmo código possa ser utilizado (porém, obviamente, ele ficará mais lento que em uma arquitetura que suporte SIMD).

2.1 Declarando um tipo vetorial

Antes de declarar variáveis vetoriais, é necessário declarar um tipo vetorial. O tipo deve especificar de que tipo é cada elemento do vetor (`char`, `short`, `int`, `long long`, `float` ou `double`) e quantos elementos tem o vetor.

O número de elementos não precisa ser um número suportado nativamente pela arquitetura. Por exemplo, na arquitetura Intel, o conjunto de instruções SSE suporta trabalhar com vetores de quatro elementos do tipo `float`, entretanto caso o programa trabalhe com vetores de 16 elementos, o gcc quebrará automaticamente cada operação em operações realizadas sobre quatro vetores de quatro elementos. Porém, vale ressaltar que o gcc exige que o número de elementos dos vetores seja sempre uma potência de dois.

Geralmente se declara o tipo vetorial em uma `union` com um array tradicional da linguagem C, para permitir que elementos do vetor sejam acessados individualmente, por exemplo para definir seus valores iniciais (antes de realizar operações), ou para consultar seus valores finais (após realizar operações). Essa é atualmente a única forma de fazer esse tipo de acesso individual de forma portável no gcc.

2.1.1 Exemplo: declarando um tipo vetorial portável de oito elementos do tipo `int`

```
typedef union {
    int __attribute__((vector_size(8*sizeof(int)))) vec;
    int elem[8];
} intvec8;
```

2.1.2 Exemplo: declarando um tipo vetorial portável de quatro elementos do tipo `float`

```
typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;
```

2.2 Declarando uma variável vetorial

Uma vez declarado um tipo vetorial, para declarar uma variável vetorial basta utilizar o tipo previamente declarado.

2.2.1 Exemplo: declarando três variáveis do tipo `floatvec4`

```c
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b, c;
    /* ... */
}
```

2.3 Definindo valores iniciais para os vetores

Para definir valores iniciais para elementos de vetores declarados como ensinado aqui, basta acessar os elementos individualmente por meio da notação `variavel.elem[indice]`.

2.3.1 Exemplo: definindo valores iniciais fixos

```c
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b, c;

    a.elem[0] = 1.345;
    a.elem[1] = 2.498;
    a.elem[2] = 9.438;
    a.elem[3] = 3.384;

    b.elem[0] = 5.398;
    b.elem[1] = 4.929;
    b.elem[2] = 0.321;
    b.elem[3] = 7.123;

    c.elem[0] = 1.000;
    c.elem[1] = 2.000;
    c.elem[2] = 3.000;
    c.elem[3] = 4.000;
```

```
    /* ... */
}
```

2.4 Realizando operações vetoriais

Para realizar operações, basta aplicar um operador suportado sobre vetores acessados com a notação `variavel.vec`. Os operadores suportados são:

- Soma (`var1.vec + var2.vec`).
- Subtração (`var1.vec - var2.vec`).
- Multiplicação elemento-a-elemento (`var1.vec * var2.vec`).
- Divisão elemento-a-elemento (`var1.vec / var2.vec`).
- Resto de divisão elemento-a-elemento (`var1.vec % var2.vec`).
- Troca de sinal (`-var1.vec`).
- XOR bit-a-bit (`var1.vec ^ var2.vec`).
- OR bit-a-bit (`var1.vec | var2.vec`).
- AND bit-a-bit (`var1.vec & var2.vec`).
- NOT bit-a-bit (`~var1.vec`).

2.4.1 Exemplo: somando `a` com `b` e multiplicando por `c`

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b, c;

    a.elem[0] = 1.345;
    a.elem[1] = 2.498;
    a.elem[2] = 9.438;
    a.elem[3] = 3.384;

    b.elem[0] = 5.398;
    b.elem[1] = 4.929;
    b.elem[2] = 0.321;
    b.elem[3] = 7.123;
```

```
    c.elem[0] = 1.000;
    c.elem[1] = 2.000;
    c.elem[2] = 3.000;
    c.elem[3] = 4.000;

    a.vec = (a.vec + b.vec) * c.vec;

    /* ... */
}
```

2.5 Acessando os resultados finais

Depois de realizadas operações vetoriais, utilize novamente a notação `variavel.elem[indice]` para acessar os resultados finais.

2.5.1 Exemplo: exemplo completo, exibindo os resultados

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b, c;

    a.elem[0] = 1.345;
    a.elem[1] = 2.498;
    a.elem[2] = 9.438;
    a.elem[3] = 3.384;

    b.elem[0] = 5.398;
    b.elem[1] = 4.929;
    b.elem[2] = 0.321;
    b.elem[3] = 7.123;

    c.elem[0] = 1.000;
    c.elem[1] = 2.000;
    c.elem[2] = 3.000;
    c.elem[3] = 4.000;

    a.vec = (a.vec + b.vec) * c.vec;

    printf("%.3f %.3f %.3f %.3f\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);
```

```
    return 0;
}
```

Para compilar o exemplo, utilize:

```
gcc -march=native -O2 exemplo.c -o exemplo
```

Ao executar o programa (`./exemplo`), a saída obtida será:

```
6.743 14.854 29.277 42.028
```

3 Programação SIMD específica para arquitetura Intel

Há operações SIMD disponíveis que não podem ser expressas por nenhum dos operadores padrão do C. Para utilizá-las sem ter de escrever uma função toda em Assembly, o gcc disponibiliza um conjunto de funções denominadas *built-ins*, que são traduzidas para uma instrução da arquitetura durante a compilação. O nome de cada *built-in* contém o nome da instrução para a qual ela será traduzida. Note que o uso de *built-ins* não é portável - o compilador não é capaz de gerar código equivalente em outras arquiteturas.

Trataremos aqui apenas dos *built-ins* disponíveis para arquitetura Intel. É importante notar também que certos *built-ins* só estarão disponíveis a partir de certas linhas de processador. Por exemplo, as instruções do conjunto SSSE3 só estão disponíveis em processadores da linha Core2 e posteriores. Caso os *built-ins* correspondentes a essas instruções sejam utilizados em seu código, ele só funcionará em processadores que as suportem.

Para verificar no Linux quais conjuntos de instruções SIMD o seu processador suporta, utilize o comando:

```
grep flags /proc/cpuinfo
```

O principal propósito desta seção é dar alguns exemplos de como essas instruções podem ser utilizadas facilmente em um código em linguagem C. Esta seção não apresenta todos os *built-ins* disponíveis. Para isso, consulte a documentação do gcc. Para detalhes do comportamento das instruções, consulte os manuais da Intel.

3.1 Soma e subtração saturada de vetores de inteiros (MMX)

O operador de soma do C tem o seguinte comportamento quando uma soma estoura, ou seja, quando o resultado não cabe no número de bits disponível para o tipo de variável utilizado:

```c
#include <stdio.h>
int main() {
    short a = 16384, b = 16384;
    printf("%hd\n", a + b);
    return 0;
}
```

O programa acima imprime o valor `-32768`. Isso acontece porque o tipo `short`, na arquitetura Intel x86, tem 16 bits, e pode armazenar (com sinal) no máximo o valor $2^{15} - 1 = 32767$. Como a soma dos dois números excede em um esse valor máximo, volta-se a contar a partir do menor valor possível ($-2^{15} = -32768$).

Entretanto, em alguns casos pode ser útil fazer com que a soma (ou subtração) sature. Ou seja, caso atingido o valor máximo, que não ocorra esse comportamento, mas sim que o valor permaneça no máximo. E caso atingido o valor mínimo, que o valor permaneça no mínimo. Para isso, o conjunto de instruções MMX fornece as operações de soma e subtração saturadas:

- `__builtin_ia32_paddsb` - recebe dois vetores de oito elementos do tipo `char` e retorna sua soma saturada.
- `__builtin_ia32_paddsw` - recebe dois vetores de quatro elementos do tipo `short` e retorna sua soma saturada.
- `__builtin_ia32_psubsb` - recebe dois vetores de oito elementos do tipo `char` e retorna sua subtração saturada.
- `__builtin_ia32_psubsw` - recebe dois vetores de quatro elementos do tipo `short` e retorna sua subtração saturada.
- `__builtin_ia32_paddusb` - recebe dois vetores de oito elementos do tipo `unsigned char` e retorna sua soma saturada.
- `__builtin_ia32_paddusw` - recebe dois vetores de quatro elementos do tipo `unsigned short` e retorna sua soma saturada.
- `__builtin_ia32_psubusb` - recebe dois vetores de oito elementos do tipo `unsigned char` e retorna sua subtração saturada.
- `__builtin_ia32_psubusw` - recebe dois vetores de quatro elementos do tipo `unsigned short` e retorna sua subtração saturada.

Nota: Em todas as instruções que operem sobre vetores de inteiros, adicionando-se `128` ao final do nome do *built-in*, é possível trabalhar com vetores com o dobro de elementos que o especificado, desde que o processador suporte SSE além de MMX. Por exemplo, com `__builtin_ia32_paddsw128` é possível trabalhar com vetores de oito elementos do tipo `short` em vez de quatro.

3.1.1 Exemplo: soma saturada de vetores de inteiros do tipo `short`

```
#include <stdio.h>

typedef union {
    short __attribute__((vector_size(4*sizeof(short)))) vec;
    short elem[4];
} shortvec4;

int main() {
    shortvec4 a, b;

    a.elem[0] = 16384;
    a.elem[1] = 17664;
    a.elem[2] = 18944;
    a.elem[3] = 20480;

    b.elem[0] = 16384;
    b.elem[1] = 16384;
    b.elem[2] = 16384;
    b.elem[3] = 16384;

    a.vec = __builtin_ia32_paddsw(a.vec, b.vec);

    printf("%hd %hd %hd %hd\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `32767 32767 32767 32767`.

3.2 Comparação entre vetores de inteiros (MMX)

Essas instruções aplicam uma comparação a cada par de elementos de um par de vetores. Sempre que a comparação for verdadeira, o elemento correspondente no vetor resultante terá todos seus bits ligados, e quando a comparação for falsa, o elemento será zero.

- `__builtin_ia32_pcmpeqb` - recebe dois vetores de oito elementos do tipo `char` e verifica se os pares de elementos são iguais.
- `__builtin_ia32_pcmpeqw` - recebe dois vetores de quatro elementos do tipo `short` e verifica se os pares de elementos são iguais.
- `__builtin_ia32_pcmpeqd` - recebe dois vetores de dois elementos do tipo `int` e verifica se os pares de elementos são iguais.
- `__builtin_ia32_pcmpgtb` - recebe dois vetores de oito elementos do tipo `char` e verifica se o primeiro elemento de cada par é maior que o segundo.

- `__builtin_ia32_pcmpgtw` - recebe dois vetores de quatro elementos do tipo `short` e verifica se o primeiro elemento de cada par é maior que o segundo.
- `__builtin_ia32_pcmpgtd` - recebe dois vetores de dois elementos do tipo `int` e verifica se o primeiro elemento de cada par é maior que o segundo.

3.2.1 Exemplo: comparação de "maior que" entre vetores de inteiros

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    int elem[4];
} intvec4;

int main() {
    intvec4 a, b;

    a.elem[0] = 10;
    a.elem[1] = 20;
    a.elem[2] = 30;
    a.elem[3] = 40;

    b.elem[0] = 10;
    b.elem[1] = 10;
    b.elem[2] = 40;
    b.elem[3] = 30;

    a.vec = __builtin_ia32_pcmpgtd128(a.vec, b.vec);

    printf("%08x %08x %08x %08x\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `00000000 ffffffff 00000000 ffffffff`.

3.3 Rearranjo de vetores de inteiros (MMX)

Existem instruções que permitem realizar certos rearranjos predefinidos de elementos em vetores de inteiros. Quando o rearranjo desejado é um dos rearranjos disponíveis, é vantajoso utilizar essas instruções em vez de movimentar os dados "manualmente", pois assim o rearranjo será realizado em uma única instrução, e dessa forma executado bem mais rápido.

- `__builtin_ia32_punpckhbw` - recebe dois vetores de oito elementos do tipo `char` e retorna um vetor constituído da segunda metade dos elementos

do primeiro vetor e da segunda metade dos elementos do segundo vetor, intercalados.

- `__builtin_ia32_punpckhwd` - recebe dois vetores de quatro elementos do tipo `short` e retorna um vetor constituído da segunda metade dos elementos do primeiro vetor e da segunda metade dos elementos do segundo vetor, intercalados.
- `__builtin_ia32_punpckhdq` - recebe dois vetores de dois elementos do tipo `int` e retorna um vetor constituído da segunda metade dos elementos do primeiro vetor e da segunda metade dos elementos do segundo vetor, intercalados.
- `__builtin_ia32_punpcklbw` - recebe dois vetores de oito elementos do tipo `char` e retorna um vetor constituído da primeira metade dos elementos do primeiro vetor e da primeira metade dos elementos do segundo vetor, intercalados.
- `__builtin_ia32_punpcklwd` - recebe dois vetores de quatro elementos do tipo `short` e retorna um vetor constituído da primeira metade dos elementos do primeiro vetor e da primeira metade dos elementos do segundo vetor, intercalados.
- `__builtin_ia32_punpckldq` - recebe dois vetores de dois elementos do tipo `int` e retorna um vetor constituído da primeira metade dos elementos do primeiro vetor e da primeira metade dos elementos do segundo vetor, intercalados.
- `__builtin_ia32_packsswb` - recebe dois vetores de quatro elementos do tipo `short` e retorna um vetor de oito elementos do tipo `char` constituído de ambos os vetores de entrada justapostos, saturando os elementos caso ocorra estouro.
- `__builtin_ia32_packssdw` - recebe dois vetores de dois elementos do tipo `int` e retorna um vetor de quatro elementos do tipo `short` constituído de ambos os vetores de entrada justapostos, saturando os elementos caso ocorra estouro.
- `__builtin_ia32_packuswb` - recebe dois vetores de quatro elementos do tipo `unsigned short` e retorna um vetor de oito elementos do tipo `unsigned char` constituído de ambos os vetores de entrada justapostos, saturando os elementos caso ocorra estouro.

3.3.1 Exemplo: intercalando a segunda metade de vetores de inteiros

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
```

```
    int elem[4];
} intvec4;

int main() {
    intvec4 a, b;

    a.elem[0] = 10;
    a.elem[1] = 20;
    a.elem[2] = 30;
    a.elem[3] = 40;

    b.elem[0] = 50;
    b.elem[1] = 60;
    b.elem[2] = 70;
    b.elem[3] = 80;

    a.vec = __builtin_ia32_punpckhdq128(a.vec, b.vec);

    printf("%d %d %d %d\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `30 70 40 80`.

3.3.2 Exemplo: justapondo dois vetores de `int` em um vetor de `short`

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    int elem[4];
} intvec4;

typedef union {
    short __attribute__((vector_size(8*sizeof(short)))) vec;
    short elem[8];
} shortvec8;

int main() {
    intvec4 a, b;
    shortvec8 c;

    a.elem[0] = 100;
    a.elem[1] = 2000;
    a.elem[2] = 30000;
```

```
    a.elem[3] = 400000;

    b.elem[0] = -50;
    b.elem[1] = -600;
    b.elem[2] = -7000;
    b.elem[3] = -80000;

    c.vec = __builtin_ia32_packssdw128(a.vec, b.vec);

    printf("%hd %hd %hd %hd %hd %hd %hd %hd\n",
           c.elem[0], c.elem[1], c.elem[2], c.elem[3],
           c.elem[4], c.elem[5], c.elem[6], c.elem[7]);

    return 0;
}
```

A saída do programa acima é:
`100 2000 30000 32767 -50 -600 -7000 -32768`.

3.4 Deslocamento (shift) de vetores de inteiros (MMX)

As seguintes instruções realizam o deslocamento para a esquerda ou para a direita de elementos contidos em vetores de inteiros, e seriam equivalentes aos operadores `<<` e `>>` do C. Entretanto, por motivos que desconhecemos, esses operadores não foram implementados (pelo menos até o momento) na extensão vetorial portável do gcc, fazendo-se necessário o uso de *built-ins* específicos de arquitetura em vez de operadores.

No caso de *built-ins* que recebam como argumentos um vetor e um escalar (um único inteiro), o comportamento é o esperado - o escalar especifica o número de bits a serem deslocados em cada elemento do vetor. Já em *built-ins* que recebam como argumentos dois vetores, o segundo vetor comportar-se-á como um escalar - apenas seu primeiro elemento será levado em consideração pela instrução.

- Deslocamento lógico para a esquerda - equivalente a uma multiplicação por 2^n sobre tipos `signed` ou `unsigned`, onde n é o número de bits deslocados.
  - `__builtin_ia32_psllw` - recebe dois vetores de quatro elementos do tipo `short`.
  - `__builtin_ia32_psllwi` - recebe um vetor de quatro elementos do tipo `short` e um escalar.
  - `__builtin_ia32_pslld` - recebe dois vetores de dois elementos do tipo `int`.
  - `__builtin_ia32_pslldi` - recebe um vetor de dois elementos do tipo `int` e um escalar.

  - `__builtin_ia32_psllq` - recebe dois "vetores" de um elemento do tipo `long long`.
  - `__builtin_ia32_psllqi` - recebe um "vetor" de um elemento do tipo `long long` e um escalar.

- Deslocamento lógico para a direita - equivalente a uma divisão por 2^n sobre tipos `unsigned`, onde n é o número de bits deslocados.
  - `__builtin_ia32_psrlw` - recebe dois vetores de quatro elementos do tipo `short`.
  - `__builtin_ia32_psrlwi` - recebe um vetor de quatro elementos do tipo `short` e um escalar.
  - `__builtin_ia32_psrld` - recebe dois vetores de dois elementos do tipo `int`.
  - `__builtin_ia32_psrldi` - recebe um vetor de dois elementos do tipo `int` e um escalar.
  - `__builtin_ia32_psrlq` - recebe dois "vetores" de um elemento do tipo `long long`.
  - `__builtin_ia32_psrlqi` - recebe um "vetor" de um elemento do tipo `long long` e um escalar.

- Deslocamento aritmético para a direita - equivalente a uma divisão por 2^n sobre tipos `signed`, onde n é o número de bits deslocados.
  - `__builtin_ia32_psraw` - recebe dois vetores de quatro elementos do tipo `short`.
  - `__builtin_ia32_psrawi` - recebe um vetor de quatro elementos do tipo `short` e um escalar.
  - `__builtin_ia32_psrad` - recebe dois vetores de dois elementos do tipo `int`.
  - `__builtin_ia32_psradi` - recebe um vetor de dois elementos do tipo `int` e um escalar.

3.4.1 Exemplo: deslocamento à direita de um vetor de inteiros

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    int elem[4];
} intvec4;

int main() {
    intvec4 a, b;
```

```
    a.elem[0] = 0x10;
    a.elem[1] = 0x100;
    a.elem[2] = 0x1000;
    a.elem[3] = 0x10000;

    b.elem[0] =  4;
    b.elem[1] =  8; /* ignorado */
    b.elem[2] = 12; /* ignorado */
    b.elem[3] = 16; /* ignorado */

    a.vec = __builtin_ia32_psrld128(a.vec, b.vec);

    printf("0x%x 0x%x 0x%x 0x%x\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `0x1 0x10 0x100 0x1000`.

3.4.2 Exemplo: equivalente ao anterior, porém mais inteligente

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    int elem[4];
} intvec4;

int main() {
    intvec4 a;

    a.elem[0] = 0x10;
    a.elem[1] = 0x100;
    a.elem[2] = 0x1000;
    a.elem[3] = 0x10000;

    a.vec = __builtin_ia32_psrldi128(a.vec, 4);

    printf("0x%x 0x%x 0x%x 0x%x\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `0x1 0x10 0x100 0x1000`.

3.5 Média entre vetores de inteiros (SSE)

- `__builtin_ia32_pavgb` - recebe dois vetores de oito elementos do tipo `char`, e retorna um vetor contendo as médias entre pares de elementos.
- `__builtin_ia32_pavgw` - recebe dois vetores de quatro elementos do tipo `short`, e retorna um vetor contendo as médias entre pares de elementos.

3.5.1 Exemplo

```
#include <stdio.h>

typedef union {
    short __attribute__((vector_size(8*sizeof(short)))) vec;
    short elem[8];
} shortvec8;

int main() {
    shortvec8 a, b;

    a.elem[0] = 1;
    a.elem[1] = 4;
    a.elem[2] = 9;
    a.elem[3] = 13;
    a.elem[4] = 16;
    a.elem[5] = 50;
    a.elem[6] = 123;
    a.elem[7] = 789;

    b.elem[0] = 3;
    b.elem[1] = 8;
    b.elem[2] = 12;
    b.elem[3] = 15;
    b.elem[4] = 23;
    b.elem[5] = 100;
    b.elem[6] = 456;
    b.elem[7] = 123;

    a.vec = __builtin_ia32_pavgw128(a.vec, b.vec);

    printf("%hd %hd %hd %hd %hd %hd %hd %hd\n",
           a.elem[0], a.elem[1], a.elem[2], a.elem[3],
           a.elem[4], a.elem[5], a.elem[6], a.elem[7]);

    return 0;
}
```

A saída do programa acima é: `2 6 11 14 20 75 290 456`.

3.6 Soma das diferenças absolutas entre vetores de inteiros (SSE)

O *built-in* `__builtin_ia32_psadbw` recebe dois vetores (a e b) de oito elementos do tipo `char` e retorna um "vetor" de um elemento do tipo `long long` contendo $\sum_i |a_i - b_i|$.

3.6.1 Exemplo

```
#include <stdio.h>

typedef union {
    char __attribute__((vector_size(8*sizeof(char)))) vec;
    char elem[8];
} charvec8;

typedef union {
    long long __attribute__((vector_size(1*sizeof(long long)))) vec;
    long long elem[1];
} longlongvec1;

int main() {
    charvec8 a, b;
    longlongvec1 c;

    a.elem[0] = 1;
    a.elem[1] = 4;
    a.elem[2] = 9;
    a.elem[3] = 13;
    a.elem[4] = 16;
    a.elem[5] = 50;
    a.elem[6] = 123;
    a.elem[7] = 12;

    b.elem[0] = 3;
    b.elem[1] = 8;
    b.elem[2] = 12;
    b.elem[3] = 15;
    b.elem[4] = 23;
    b.elem[5] = 100;
    b.elem[6] = 98;
    b.elem[7] = 123;

    c.vec = __builtin_ia32_psadbw(a.vec, b.vec);
```

```
    printf("%lld\n", c.elem[0]);

    return 0;
}
```

A saída do programa acima é: 204.

3.7 Máximos e mínimos entre vetores de inteiros (SSE)

- `__builtin_ia32_pmaxub` - Recebe dois vetores de oito elementos do tipo `unsigned char` e retorna um vetor contendo o maior elemento entre cada par de elementos.
- `__builtin_ia32_pmaxsw` - Recebe dois vetores de quatro elementos do tipo `short` e retorna um vetor contendo o maior elemento entre cada par de elementos.
- `__builtin_ia32_pminub` - Recebe dois vetores de oito elementos do tipo `unsigned char` e retorna um vetor contendo o menor elemento entre cada par de elementos.
- `__builtin_ia32_pminsw` - Recebe dois vetores de quarto elementos do tipo `short` e retorna um vetor contendo o menor elemento entre cada par de elementos.

3.7.1 Exemplo: máximos entre vetores de inteiros

```
#include <stdio.h>

typedef union {
    short __attribute__((vector_size(8*sizeof(short)))) vec;
    short elem[8];
} shortvec8;

int main() {
    shortvec8 a, b;

    a.elem[0] = 12;
    a.elem[1] = 4;
    a.elem[2] = 9;
    a.elem[3] = 13;
    a.elem[4] = 16;
    a.elem[5] = 50;
    a.elem[6] = 123;
    a.elem[7] = 12;
```

```
    b.elem[0] = 3;
    b.elem[1] = 8;
    b.elem[2] = 12;
    b.elem[3] = 15;
    b.elem[4] = 23;
    b.elem[5] = 100;
    b.elem[6] = 98;
    b.elem[7] = 123;

    a.vec = __builtin_ia32_pmaxsw128(a.vec, b.vec);

    printf("%hd %hd %hd %hd %hd %hd %hd %hd\n",
           a.elem[0], a.elem[1], a.elem[2], a.elem[3],
           a.elem[4], a.elem[5], a.elem[6], a.elem[7]);

    return 0;
}
```

A saída do programa acima é: `12 8 12 15 23 100 123 123`.

3.8 Extração do primeiro bit de cada byte de um vetor de inteiros (SSE)

O built-in `__builtin_ia32_pmovmskb` recebe um vetor de oito elementos do tipo `char` e extrai o primeiro bit de cada um desses elementos, retornando um escalar (inteiro) contendo os bits justapostos.

3.8.1 Exemplo: usando extração do primeiro bit para comparação rápida entre vetores de inteiros

```
#include <stdio.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    char __attribute__((vector_size(16*sizeof(char)))) charvec;
    int elem[4];
} intvec4;

int main() {
    intvec4 a, b;

    a.elem[0] = 10;
    a.elem[1] = 20;
    a.elem[2] = 30;
    a.elem[3] = 40;
```

```
    b.elem[0] = 1;
    b.elem[1] = 2;
    b.elem[2] = 3;
    b.elem[3] = 4;

    a.vec = __builtin_ia32_pcmpgtd128(a.vec, b.vec);
    if(__builtin_ia32_pmovmskb128(a.charvec) == 0xffff)
        printf("Todos os elementos de a sao maiores "
               "que os correspondentes de b.\n");
    else
        printf("Algum elemento de a eh menor "
               "ou igual ao correspondente de b.\n");

    return 0;
}
```

A saída do programa acima é: `Todos os elementos de a sao maiores que os correspondentes de b.`

3.9 Comparação entre primeiro elemento de vetores de ponto flutuante (SSE)

Os *built-ins* a seguir recebem dois vetores de quatro elementos do tipo `float` e comparam o primeiro elemento de cada um deles, retornando `0` (falso) caso a comparação for falsa e `1` (verdadeiro) se a comparação for verdadeira.

Isso é mais rápido que fazer a comparação usando acesso individual (com `var1.elem[0] < var2.elem[0]`, por exemplo) pois não envolve movimentação do primeiro elemento para outros registradores nem acessos de memória.

- `__builtin_ia32_ucomieq` - comparação de "igual".
- `__builtin_ia32_ucomineq` - comparação de "diferente".
- `__builtin_ia32_ucomilt` - comparação de "menor que".
- `__builtin_ia32_ucomile` - comparação de "menor ou igual".
- `__builtin_ia32_ucomigt` - comparação de "maior que".
- `__builtin_ia32_ucomige` - comparação de "maior ou igual".

Caso o processador suporte SSE2, poderão ser usados também *built-ins* equivalentes que trabalham com vetores de dois elementos do tipo `double`, denominados:

- `__builtin_ia32_ucomisdeq`
- `__builtin_ia32_ucomisdneq`
- `__builtin_ia32_ucomisdlt`

- `__builtin_ia32_ucomisdle`
- `__builtin_ia32_ucomisdgt`
- `__builtin_ia32_ucomisdge`

3.9.1 Exemplo: comparando o primeiro elemento de vetores de `floats`

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 10.93;
    a.elem[1] = 20.12;
    a.elem[2] = 30.05;
    a.elem[3] = 40.19;

    b.elem[0] = 1.928;
    b.elem[1] = 200.261;
    b.elem[2] = 300.598;
    b.elem[3] = 400.028;

    if(__builtin_ia32_ucomigt(a.vec, b.vec))
        printf("a.elem[0] > b.elem[0]\n");
    else
        printf("a.elem[0] <= b.elem[0]\n");

    return 0;
}
```

A saída do programa acima é: `a.elem[0] > b.elem[0]`.

3.10 Comparação entre vetores de ponto flutuante (SSE)

Estão disponíveis também instruções que comparam vetores inteiros, e não somente o primeiro elemento. Elas recebem dois vetores de quatro elementos do tipo `float` e, em tese, retornariam um vetor de quatro elementos do tipo `int` contendo `0xffffffff` quando a comparação fosse verdadeira e `0` quando fosse falsa. Entretanto, devido a uma inconsistência do gcc com a documentação, o vetor é retornado como se fosse um vetor de quatro `floats`. Para contornar isso, pode-se adicionar um vetor de `ints` na mesma `union` do vetor de `floats`, como será mostrado no exemplo.

- `__builtin_ia32_cmpeqps` - comparação de "igual".
- `__builtin_ia32_cmpneqps` - comparação de "diferente".
- `__builtin_ia32_cmpltps` - comparação de "menor que".
- `__builtin_ia32_cmpleps` - comparação de "menor ou igual".
- `__builtin_ia32_cmpgtps` - comparação de "maior que".
- `__builtin_ia32_cmpgeps` - comparação de "maior ou igual".

Nota: substituindo o sufixo `ps` por `pd`, tem-se instruções equivalentes para trabalhar com vetores de dois elementos do tipo `double`, que estão disponíveis em processadores que suportem SSE2.

3.10.1 Exemplo: comparação de "maior que" entre vetores de `floats`

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
    int __attribute__((vector_size(4*sizeof(int)))) ivec;
    int ielem[4];
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 10.93;
    a.elem[1] = 20.12;
    a.elem[2] = 30.05;
    a.elem[3] = 40.19;

    b.elem[0] = 1.928;
    b.elem[1] = 200.261;
    b.elem[2] = 300.598;
    b.elem[3] = 400.028;

    a.vec = __builtin_ia32_cmpgtps(a.vec, b.vec);

    printf("%08x %08x %08x %08x\n", a.ielem[0], a.ielem[1],
                                    a.ielem[2], a.ielem[3]);

    return 0;
}
```

A saída do programa acima é: `ffffffff 00000000 00000000 00000000`.

3.10.2 Exemplo: verificando se todos os elementos de um certo vetor de `floats` são menores que os de outro

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
    int __attribute__((vector_size(4*sizeof(int)))) ivec;
    int ielem[4];
    char __attribute__((vector_size(16*sizeof(char)))) cvec;
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 10.93;
    a.elem[1] = 20.12;
    a.elem[2] = 30.05;
    a.elem[3] = 40.19;

    b.elem[0] = 100.928;
    b.elem[1] = 200.261;
    b.elem[2] = 300.598;
    b.elem[3] = 400.028;

    a.vec = __builtin_ia32_cmpltps(a.vec, b.vec);
    if(__builtin_ia32_pmovmskb128(a.cvec) == 0xffff)
        printf("Todos os elementos de a sao menores "
               "que os correspondentes de b.\n");
    else
        printf("Algum elemento de a eh maior ou "
               "igual ao correspondente de b.\n");

    return 0;
}
```

A saída do programa acima é: `Todos os elementos de a sao menores que os correspondentes de b.`

3.11 Máximos e mínimos entre vetores de ponto flutuante (SSE)

- `__builtin_ia32_maxps` - Recebe dois vetores de quatro elementos do tipo `float` e retorna um vetor contendo o maior elemento entre cada par de elementos.

- `__builtin_ia32_minps` - Recebe dois vetores de quatro elementos do tipo `float` e retorna um vetor contendo o menor elemento entre cada par de elementos.

Nota: substituindo o sufixo `ps` por `pd`, tem-se instruções equivalentes para trabalhar com vetores de dois elementos do tipo `double`, que estão disponíveis em processadores que suportem SSE2.

3.11.1 Exemplo: máximos entre vetores de ponto flutuante

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 12.125;
    a.elem[1] = 4.256;
    a.elem[2] = 9.724;
    a.elem[3] = 13.647;

    b.elem[0] = 3.464;
    b.elem[1] = 8.746;
    b.elem[2] = 12.624;
    b.elem[3] = 15.643;

    a.vec = __builtin_ia32_maxps(a.vec, b.vec);

    printf("%.3f %.3f %.3f %.3f\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `12.125 8.746 12.624 15.643`.

3.12 Rearranjo de vetores de ponto flutuante (SSE)

- `__builtin_ia32_movhlps` - recebe dois vetores de quatro elementos do tipo `float`, e retorna o primeiro vetor com seus dois primeiros elementos trocados pelos dois últimos elementos do segundo vetor.
- `__builtin_ia32_movlhps` - recebe dois vetores de quatro elementos do tipo `float`, e retorna o primeiro vetor com seus dois últimos elementos trocados pelos dois primeiros elementos do segundo vetor.

- `__builtin_ia32_unpckhps` - recebe dois vetores de quatro elementos do tipo `float`, e retorna um vetor contendo os dois primeiros elementos de cada um dos vetores, intercalados.
- `__builtin_ia32_unpcklps` - recebe dois vetores de quatro elementos do tipo `float`, e retorna um vetor contendo os dois últimos elementos de cada um dos vetores, intercalados.
- `__builtin_ia32_shufps` - recebe dois vetores de quatro elementos do tipo `float` e um inteiro escalar, e retorna um vetor cujos dois primeiros elementos serão do primeiro vetor (escolhidos pelos quatro bits menos significativos do inteiro) e os dois últimos serão do segundo vetor (escolhidos pelos quatro bits seguintes do inteiro); o inteiro deve ser uma constante pois ele é codificado na instrução como um imediato.

3.12.1 Exemplo: utilizando o `shufps`

```c
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 12.125;
    a.elem[1] = 4.256;
    a.elem[2] = 9.724;
    a.elem[3] = 13.647;

    b.elem[0] = 3.464;
    b.elem[1] = 8.746;
    b.elem[2] = 12.624;
    b.elem[3] = 15.643;

    /* 0xd4 em binario: 11 01 01 00 */
    a.vec = __builtin_ia32_shufps(a.vec, b.vec, 0xd4);

    printf("%.3f %.3f %.3f %.3f\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `12.125 4.256 8.746 15.643`.

3.13 Conversão entre vetores de inteiros e de ponto flutuante (SSE)

- `__builtin_ia32_cvtpi2ps` - recebe um vetor de quatro elementos do tipo `float` e um vetor de dois elementos do tipo `int`, e retorna o primeiro vetor com seus dois primeiros elementos substituídos pelos elementos do vetor de inteiros convertidos para ponto flutuante.

- `__builtin_ia32_cvtsi2ss` - recebe um vetor de quatro elementos do tipo `float` e um escalar inteiro, e retorna o vetor com seu primeiro elemento substituído pelo escalar convertido para ponto flutuante.

- `__builtin_ia32_cvttps2pi` - recebe um vetor de quatro elementos do tipo `float` e retorna um vetor de dois elementos do tipo `int`, contendo os dois primeiros elementos do vetor recebido convertidos (com saturação) para inteiros.

- `__builtin_ia32_cvttss2si` - recebe um vetor de quatro elementos do tipo `float` e retorna um escalar inteiro, igual ao primeiro elementos do vetor recebido convertido (com saturação) para inteiro.

3.13.1 Exemplo: convertendo o primeiro elemento de um vetor de ponto flutuante para inteiro

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a;

    a.elem[0] = 12.125;
    a.elem[1] = 4.256;
    a.elem[2] = 9.724;
    a.elem[3] = 13.647;

    printf("%d\n", __builtin_ia32_cvttss2si(a.vec));

    return 0;
}
```

A saída do programa acima é: 12.

3.14 Operações aritméticas adicionais (SSE)

Além das operações que podem ser efetuadas utilizando as extensões portáveis do gcc (como soma, subtração, multiplicação e divisão), estão disponíveis algumas operações adicionais nos *built-ins* para arquitetura Intel:

- `__builtin_ia32_rcpps` - recebe um vetor de quatro elementos do tipo `float` e retorna um vetor contendo o inverso ($\frac{1}{x}$) de cada elemento calculado de forma aproximada porém rápida. É garantido um erro relativo máximo de 1.5×2^{-12}.
- `__builtin_ia32_rsqrtps` - recebe um vetor de quatro elementos do tipo `float` e retorna um vetor contendo o inverso da raíz quadrada ($\frac{1}{\sqrt{x}}$) de cada elemento calculado de forma aproximada porém rápida. É garantido um erro relativo máximo de 1.5×2^{-12}.
- `__builtin_ia32_sqrtps` - recebe um vetor de quatro elementos do tipo float e retorna um vetor contendo a raíz quadrada ($\sqrt{x}$) de cada elemento.

3.14.1 Exemplo: inverso da raíz quadrada de um vetor de ponto flutuante

```c
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a;

    a.elem[0] = 4.0;
    a.elem[1] = 16.0;
    a.elem[2] = 64.0;
    a.elem[3] = 256.0;

    a.vec = __builtin_ia32_rsqrtps(a.vec);

    printf("%.2e %.2e %.2e %.2e\n", a.elem[0], a.elem[1], a.elem[2], a.elem[3]);

    return 0;
}
```

A saída do programa acima é: `5.00e-01 2.50e-01 1.25e-01 6.25e-02`.

3.15 Soma horizontal de vetores (SSSE3)

Em processadores com suporte ao SSSE3, é possível realizar a operação de soma horizontal de vetores. É dado o nome de *horizontal* a essa operação pois ela trabalha com pares de elementos pertencentes ao mesmo vetor, em vez de cada elemento ser de um dos vetores, como ocorre nas outras operações (que podem ser chamadas de *verticais*).

Cada operação de soma horizontal recebe dois vetores de um certo comprimento, e retorna um vetor de mesmo comprimento contendo a soma de cada um dos pares de elementos formados do primeiro ao último elemento do primeiro vetor, seguida da soma de cada um dos pares de elementos formados do último ao primeiro elemento do segundo vetor.

- `__builtin_ia32_haddpd` - trabalha com vetores de dois elementos do tipo `double`.
- `__builtin_ia32_haddps` - trabalha com vetores de quatro elementos do tipo `float`.
- `__builtin_ia32_phaddd` - trabalha com vetores de dois elementos do tipo `int`.
- `__builtin_ia32_phaddw` - trabalha com vetores de dois elementos do tipo `short`.
- `__builtin_ia32_phaddsw` - equivalente à anterior, mas aplica soma saturada em vez de soma comum.

3.15.1 Exemplo: soma de todos os elementos de vetores de ponto flutuante

```
#include <stdio.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

int main() {
    floatvec4 a, b;

    a.elem[0] = 1.01;
    a.elem[1] = 2.02;
    a.elem[2] = 3.03;
    a.elem[3] = 4.04;

    b.elem[0] = 5.05;
    b.elem[1] = 6.06;
```

```
    b.elem[2] = 7.07;
    b.elem[3] = 8.08;

    a.vec = __builtin_ia32_haddps(a.vec, b.vec);
    a.vec = __builtin_ia32_haddps(a.vec, a.vec);

    printf("%.2f %.2f\n", a.elem[0], a.elem[1]);

    return 0;
}
```

A saída do programa acima é: `10.10 26.26`.

4 Alguns exemplos de aplicações

4.1 Método de Monte Carlo para o cálculo de π

É possível paralelizar facilmente a geração de números aleatórios utilizando instruções SIMD, o que beneficia muito problemas que sejam resolvidos pelo método de Monte Carlo. Utilizaremos o exemplo mais clássico de aplicação do método de Monte Carlo, o cálculo de π, para mostrar como o uso de instruções SIMD pode acelerá-lo.

4.1.1 Código em C sem uso de SIMD

```
#include <stdio.h>
#include <math.h>

int main() {
    /* Sementes aleatorias */
    unsigned int m_w = 4127241766ul, m_z = 2188914947ul;
    unsigned int i, j = 0;
    const unsigned int num_iters = 10000000;
    const float normalize_uint = 1./4294967295.;
    float x, y;
    for(i = 0; i < num_iters; i++) {
        /* Gerador de Marsaglia */
        m_z = 36969 * (m_z & 65535) + (m_z >> 16);
        m_w = 18000 * (m_w & 65535) + (m_w >> 16);
        x = normalize_uint*((m_z << 16) + m_w);
        /* Gerador de Marsaglia */
        m_z = 36969 * (m_z & 65535) + (m_z >> 16);
        m_w = 18000 * (m_w & 65535) + (m_w >> 16);
        y = normalize_uint*((m_z << 16) + m_w);
        /* Verifica se caiu dentro do circulo */
        if(sqrt(x*x + y*y) < 1.)
```

```
            j++;
    }
    /* Exibe o resultado, lembrando que lancamos dardos somente
     * sobre um dos quadro quadrantes.
     */
    printf("pi = %.8f\n", (4.*j)/num_iters);
    return 0;
}
```

4.1.2 Código em C com uso de SIMD

```
#include <stdio.h>
#include <math.h>

typedef union {
    int __attribute__((vector_size(4*sizeof(int)))) vec;
    unsigned int elem[4];
} uintvec4;

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
    char __attribute__((vector_size(16*sizeof(char)))) cvec;
} floatvec4;

int main() {
    unsigned int i, j = 0;

    /* Calcularemos 2 pares (x,y) por iteracao, portanto o numero de iteracoes
     * deve ser a metade
     */
    const unsigned int num_iters = 10000000/2;

    /* A conversao de float para int tera de ser com sinal, portanto a faixa
     * de variacao sera a metade que se fosse sem sinal.
     */
    const float normalize_int = 2./4294967295.;

    uintvec4 m_w, m_z;
    uintvec4 a1, a2, b0;
    floatvec4 normvec, xy, onevec;
    unsigned int mask;

    /* Sementes aleatorias */
    m_w.elem[0] = 4127241766ul;
    m_z.elem[0] = 2188914947ul;
```

```
    m_w.elem[1] =  234248649ul;
    m_z.elem[1] = 3381995595ul;
    m_w.elem[2] =  635236254ul;
    m_z.elem[2] = 3785659006ul;
    m_w.elem[3] = 3325200839ul,
    m_z.elem[3] = 1498055452ul;

    /* Parametros do gerador de Marsaglia */
    a1.elem[0] = a1.elem[1] = a1.elem[2] = a1.elem[3] = 36969;
    a2.elem[0] = a2.elem[1] = a2.elem[2] = a2.elem[3] = 18000;
    b0.elem[0] = b0.elem[1] = b0.elem[2] = b0.elem[3] = 65535;

    /* Vetores para normalizacao e comparacao */
    normvec.elem[0] = normvec.elem[1] = normvec.elem[2] = normvec.elem[3] = normalize_int;
    onevec.elem[0] = onevec.elem[1] = 1.;

    for(i = 0; i < num_iters; i++) {
        /* Gerador de Marsaglia */
        m_z.vec = a1.vec * (m_z.vec & b0.vec) + __builtin_ia32_psrldi128(m_z.vec, 16);
        m_w.vec = a2.vec * (m_w.vec & b0.vec) + __builtin_ia32_psrldi128(m_w.vec, 16);
        /* Passo final do gerador e conversao de int para float */
        xy.vec = __builtin_ia32_cvtdq2ps(__builtin_ia32_pslldi128(m_z.vec, 16) + m_w.vec);
        /* Normalizacao para numeros aleatorios entre 0 e 1 */
        xy.vec *= normvec.vec;
        /* Quadrado de cada elemento do vetor */
        xy.vec *= xy.vec;
        /* Soma entre elementos vizinhos */
        xy.vec = __builtin_ia32_haddps(xy.vec, xy.vec);
        /* Raiz quadrada de cada elemento do vetor */
        xy.vec = __builtin_ia32_sqrtps(xy.vec);
        /* Comparacao de "menor que um" */
        xy.vec = __builtin_ia32_cmpltps(xy.vec, onevec.vec);
        mask = __builtin_ia32_pmovmskb128(xy.cvec);
        /* Soma um para cada um dos dois resultados que tenha sido menor que um */
        j += (mask & 0x1) + ((mask & 0x10)>>4);
    }

    /* Exibe o resultado, lembrando que para cada iteracao foram lancados dois dardos */
    printf("pi = %.8f\n", (2.*j)/num_iters);

    return 0;
}
```

4.1.3 Avaliação de desempenho

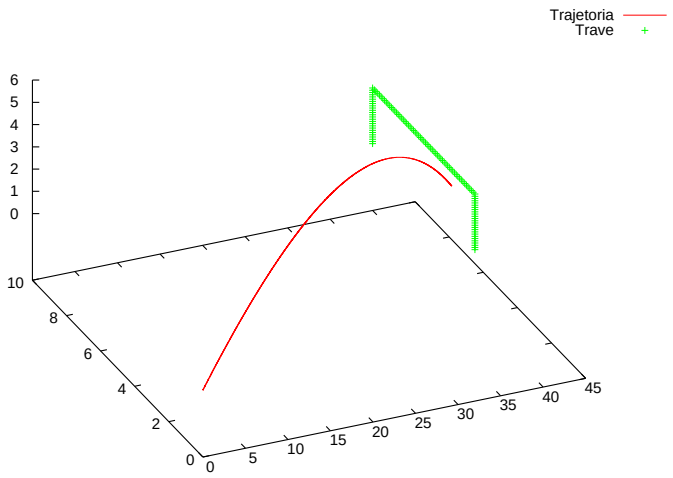
```
$ gcc -O2 -o pi pi.c -lm -Wall -march=native -mfpmath=sse
$ gcc -O2 -o pi_simd pi_simd.c -lm -Wall -march=native -mfpmath=sse
$ time ./pi
pi = 3.14123120

real    0m1.039s
user    0m1.024s
sys     0m0.008s
$ time ./pi_simd
pi = 3.14156000

real    0m0.429s
user    0m0.416s
sys     0m0.000s
```

Houve um *speed-up* de cerca de 2.4x com o uso de instruções SIMD. Considerando que, na versão SIMD, são calculados dois pontos (x, y) aleatórios por iteração em vez um único ponto, o *speed-up* obtido foi muito satisfatório.

4.2 Chute de uma bola de futebol



Esse é um exemplo no qual o desempenho obtido com o uso de instruções SIMD não é satisfatório. Isso ocorre pois é necessário acessar elementos individuais dos vetores com frequência durante o cálculo.

4.2.1 Código em C sem uso de SIMD

```c
#include <stdio.h>
#include <assert.h>
#include <math.h>

static inline void writetxyz(FILE *fp, float t, float x, float y, float z) {
    fwrite(&t, sizeof(float), 1, fp);
    fwrite(&x, sizeof(float), 1, fp);
    fwrite(&y, sizeof(float), 1, fp);
    fwrite(&z, sizeof(float), 1, fp);
}

int main() {
    FILE *fp;
    float mb, beta0_mb, theta, phi;

    /* Velocidade inicial da bola (m/s) */
    const float v0 = 100./3.6;
    /* Velocidade angular de rotacao da bola (rad/s) */
    const float w = 10.;
    /* Fatores para calculo da resistencia do ar (S.I.) */
    const float a1 = 4.e-3, a2 = 6.e-3;
    const float vd = 35.;
    const float delta = 5.;
    /* Aceleracao da gravidade (m/s^2) */
    const float g = 10.;
    /* Passo de integracao */
    const float delta_t = 0.00001;
    /* Vertices superiores da trave (x1,y1,z1) e (x1,y2,z1) */
    const float x1 = 40., y1 = 4., z1 = 2.5, y2 = 10.;

    /* Tempo atual */
    float t;
    /* Coordenadas da bola */
    float x, y, z;
    /* Componentes da velocidade da bola */
    float vx, vy, vz;
    float nvx, nvy, nvz;
    /* Velocidade da bola */
    float v;
    /* Fator de resistencia do ar */
    float gamma2_mb;

    /* Tempo maximo de simulacao (s). Evita que o programa trave
     * caso um usuario insano transforme a bola em um bumerangue
```

```
 * alterando o valor de beta0_mb.
 */
const float t_max = 180.;

/* Massa da bola (kg) */
scanf("%f", &mb);
/* Fator de efeito Magnus beta0/mb */
scanf("%f", &beta0_mb);
/* Angulos de lancamento (graus) */
scanf("%f%f", &theta, &phi);

/* Graus -> radianos */
theta = (theta / 180.) * M_PI;
phi   = (phi   / 180.) * M_PI;

/* Parametros iniciais */

t = 0.;

x = 0.;
y = 0.;
z = 0.;

v  = v0;
vx = v0 * sin(theta) * cos(phi);
vy = v0 * sin(theta) * sin(phi);
vz = v0 * cos(theta);

/* Computacao */
assert((fp = fopen("chute_out.dat", "wb")) != NULL);
writetxyz(fp,t,x,y,z);

while((x < x1) && (t < t_max)) {
    gamma2_mb = a1 + a2/(1. + exp((v - vd)/delta));
    nvx = vx - (gamma2_mb*v*vx + beta0_mb*w*vy)*delta_t;
    nvy = vy - (gamma2_mb*v*vy - beta0_mb*w*vx)*delta_t;
    nvz = vz - (g + gamma2_mb*v*vz)*delta_t;

    vx = nvx;
    vy = nvy;
    vz = nvz;

    v = sqrt((vx*vx) + (vy*vy) + (vz*vz));

    x += vx*delta_t;
    y += vy*delta_t;
```

```
        z += vz*delta_t;

        /* Faz a bola quicar elasticamente ao bater no chao */
        if(z < 0) {
            z = fabsf(z);
            vz = -vz;
        }

        t += delta_t;
        writetxyz(fp,t,x,y,z);
    }

    /* Verifica se foi feito gol */
    if((x > x1) && (z < z1) && (y > y1) && (y < y2)) {
        puts("1");
    }
    else {
        puts("0");
    }

    return 0;
}
```

4.2.2 Código em C com uso de SIMD

```
#include <stdio.h>
#include <assert.h>
#include <math.h>

typedef union {
    float __attribute__((vector_size(4*sizeof(float)))) vec;
    float elem[4];
} floatvec4;

static inline void writetxyz(FILE *fp, float t, floatvec4 vec) {
    fwrite(&t, sizeof(float), 1, fp);
    fwrite(vec.elem, sizeof(float), 3, fp);
}

int main() {
    FILE *fp;
    float mb, beta0_mb, theta, phi;

    /* Velocidade inicial da bola (m/s) */
    const float v0 = 100./3.6;
    /* Velocidade angular de rotacao da bola (rad/s) */
```

```
const float w = 10.;
/* Fatores para calculo da resistencia do ar (S.I.) */
const float a1 = 4.e-3, a2 = 6.e-3;
const float vd = 35.;
const float delta = 5.;
/* Aceleracao da gravidade (m/s^2) */
const float g = 10.;
/* Passo de integracao */
const float delta_t = 0.00001;
/* Vertices superiores da trave (x1,y1,z1) e (x1,y2,z1) */
const float x1 = 40., y1 = 4., z1 = 2.5, y2 = 10.;

/* Tempo atual */
float t;
/* Coordenadas da bola */
floatvec4 vecr;
/* Componentes da velocidade da bola */
floatvec4 vecv;
/* Velocidade da bola */
floatvec4 vecvmod;

/* Tempo maximo de simulacao (s). Evita que o programa trave
 * caso um usuario insano transforme a bola em um bumerangue
 * alterando o valor de beta0_mb.
 */
const float t_max = 180.;

/* Vetores auxiliares */
floatvec4 vecrf;    /* vecr final */
floatvec4 vecrchao; /* vecr do chao */
floatvec4 vecdt;    /* vetor com delta_t repetido */
floatvec4 vectmp;

/* Massa da bola (kg) */
scanf("%f", &mb);
/* Fator de efeito Magnus beta0/mb */
scanf("%f", &beta0_mb);
/* Angulos de lancamento (graus) */
scanf("%f%f", &theta, &phi);

/* Graus -> radianos */
theta = (theta / 180.) * M_PI;
phi   = (phi   / 180.) * M_PI;

/* Parametros iniciais */
```

```
t = 0.;

vecr.elem[0] = 0.;
vecr.elem[1] = 0.;
vecr.elem[2] = 0.;

vecvmod.elem[0] = v0;
vecvmod.elem[1] = v0;
vecvmod.elem[2] = v0;

vecv.elem[0] = v0 * sin(theta) * cos(phi);
vecv.elem[1] = v0 * sin(theta) * sin(phi);
vecv.elem[2] = v0 * cos(theta);
vecv.elem[3] = 0.;

vecrf.elem[0] = x1;
vecrchao.elem[0] = 0.;

vecdt.elem[0] = delta_t;
vecdt.elem[1] = delta_t;
vecdt.elem[2] = delta_t;
vecdt.elem[3] = 0.;

/* Computacao */
assert((fp = fopen("chute_out.dat", "wb")) != NULL);
writetxyz(fp,t,vecr);

while((__builtin_ia32_ucomilt(vecr.vec, vecrf.vec)) && (t < t_max)) {
    vectmp.elem[0] = vectmp.elem[1] = vectmp.elem[2] =
        a1 + a2/(1. + exp((vecvmod.elem[0] - vd)/delta));
    vectmp.vec *= vecvmod.vec;
    vectmp.vec *= vecv.vec;
    vectmp.elem[0] += beta0_mb*w*vecv.elem[1];
    vectmp.elem[1] -= beta0_mb*w*vecv.elem[0];
    vectmp.elem[2] += g;

    vecv.vec -= vectmp.vec*vecdt.vec;

    vecvmod.vec = vecv.vec*vecv.vec;
    vecvmod.vec = __builtin_ia32_haddps(vecvmod.vec, vecvmod.vec);
    vecvmod.vec = __builtin_ia32_haddps(vecvmod.vec, vecvmod.vec);
    vecvmod.vec = __builtin_ia32_sqrtps(vecvmod.vec);

    vecr.vec += vecv.vec*vecdt.vec;

    /* Faz a bola quicar elasticamente ao bater no chao */
```

```
        vectmp.vec = __builtin_ia32_movlhps(vectmp.vec, vecr.vec);
        if(__builtin_ia32_ucomilt(vecr.vec, vecrchao.vec)) {
            vecr.elem[2] = fabsf(vecr.elem[2]);
            vecv.elem[2] = -vecv.elem[2];
        }

        t += delta_t;
        writetxyz(fp,t,vecr);
    }

    /* Verifica se foi feito gol */
    if((vecr.elem[0] > x1) && (vecr.elem[2] < z1) &&
       (vecr.elem[1] > y1) && (vecr.elem[1] < y2)) {
        puts("1");
    }
    else {
        puts("0");
    }

    fclose(fp);
    return 0;
}
```

4.2.3 Avaliação de desempenho

```
$ gcc -O2 -o chute chute.c -lm -Wall -march=native -mfpmath=sse
$ gcc -O2 -o chute_simd chute_simd.c -lm -Wall -march=native -mfpmath=sse
$ time ./chute < param.txt
1

real    0m0.476s
user    0m0.216s
sys     0m0.264s
$ time ./chute_simd < param.txt
1

real    0m0.432s
user    0m0.168s
sys     0m0.260s
```

Houve um *speed-up* bastante baixo com o uso de instruções SIMD, de cerca de 1.1x.