Insper

01 - Introdução a SIMD

SuperComputação - 2018/2

Igor Montagner, Luciano Soares

Objetivos de aprendizado:

- Conhecer principais pontos da supercomputação
- Programar para arquiteturas vetoriais
- Habilitar auto vetorização e conhecer suas limitações

Tarefa 1 - Instruções SIMD em Assembly

Nesta primeira tarefa iremos compilar alguns códigos em C++ para Assembly usando as opções `-ffast-math -ftree-vectorize -mavx` do `gcc`. Estas opções habilitam a *autovetorização* de código. Ou sejam elas analisam o código e procuram substituir loops e acessos sequenciais a vetores por instruções SIMD que façam o mesmo trabalho.

A flag `-mavx` indica que o código de máquina gerado pode utilizar instruções SIMD da arquitetura *AVX* (e de sua predecessora, SSE), que usa registradores de 128 bits nomeados `%xmm0` até `%xmm7` e de 256 bits nomeados `%ymm0` até `%ymm15`. Ou seja, posso armazenar, em um registrador `%xmm0`

- _____ chars;
- _____ shorts;
- _____ ints;
- _____ longs;
- _____ floats;
- _____ doubles;

Para registradores `%ymm` é só dobrar os valores acima.

Toda instrução SIMD opera sobre todos os elementos guardados ao mesmo tempo. Ou seja, ao executar uma instrução SIMD de soma de variáveis `int` no registrador `%xmm0` estarei somando _____ variáveis em uma só instrução.

Vamos agora analisar o código Assembly de uma função simples que soma todos elementos de um vetor.

```
// Original: Nicolas Brailovsky

#define SIZE (400)
long sum(int v[SIZE]) throw() {
    int s = 0; // este exemplo eh didatico. soma de ints deveria ser long ;)
    for (unsigned i=0; i<SIZE; i++) s += v[i];
    return s;
}
```

Primeiro, compile este código para Assembly sem SIMD.

> $ g++ -S -c -O2 tarefa1.cpp -o /dev/stdout | c++filt

Agora, compile o mesmo programa habilitando a autovetorização.

> $ g++ -S -c -O2 -ftree-vectorize -mavx tarefa1.cpp -o /dev/stdout | c++filt

Discussão 1: Você consegue identificar onde os códigos diferem?

Tarefa 2 - Autovetorização em loops

Nesta tarefa iremos trabalhar com as opções de autovetorização do `gcc` para entender como escrever código que possa ser otimizado automaticamente.

Exercício 1

Escreva uma função `main` que gera um vetor de tamanho `10.000.000` contendo números aleatórios uniformemente distribuídos entre `-10` e `10`. Use as funções do cabeçalho `<random>`.

Exercício 2

Escreva uma função `double soma_positivos1(double *a, int n)` que soma todos os números positivos do vetor `a`. Adicione uma chamada a esta função no seu `main` e use as funções do cabeçalho `<chrono>` para medir o tempo de execução da sua função.

Compile com e sem as otimizações SIMD e verifique se há diferença no tempo de execução.

Exercício 3

O auto vetorizador suporta uma série de padrões de codificação relativamente abrangente (lista completa). Porém, códigos que são vetorizados de maneira idêntica possuem desempenho bastante diferente quanto a vetorização não está

habilitada. Faça uma função `double soma_positivos2(double *a, int n)` que faz o mesmo que a função anterior, mas usando agora o operador ternário `(cond)?expr_true:expr_false` ao invés de um `if`. (Se você fez com o operador ternário acima faça com `if`). Houve diferença de desempenho na versão SIMD? E na versão sem SIMD?

Exercício 4

Qual versão da função anterior você usaria se seu código fosse executado em processadores de baixo custo (Intel Celeron) ou muito antigos (mais de 5 anos)? E se o plano for executar em processadores novos?

Tarefa 3 - Avaliação de desempenho

Nas últimas duas tarefas vimos como usar as opções do compilador para gerar instruções SIMD, tornando nossos programas mais eficientes. Nesta tarefa iremos quantificar esta diferença de desempenho usando as funções presentes no arquivo *funcs.cpp*. Você deverá produzir um relatório comentando os ganhos de desempenho obtidos e abordando os seguintes pontos.

1. A partir de qual tamanho de array o código vetorizado mostra ganhos de desempenho expressivos?
2. Qual é o ganho de desempenho esperado? Leve em conta a arquitetura usada e o tipo de dados usado.
3. Os ganhos de desempenho são consistentes com o esperado?
4. Quais tipos de operações resultam em maior ganho de desempenho?

Uma boa análise de desempenho usa informações visuais (gráficos) e testa de maneira abrangente o espaço de parâmetros estudados. Além disto, todo relatório gerado deverá ser reprodutível. Ou seja, ele é acompanhado de um programa que permite reexecutar os testes reportados e de uma documentação mostrando como o programa funciona e como usá-lo para gerar o relatório e os gráficos contidos nele.

Requisitos:

1. Seu programa deverá estar em *C++* e usar a classe `std::chrono::high_resolution_clock` para medir os tempos.
2. Você deve testar tanto a arquitetura *AVX* (usando `-mavx`) quando *AVX2* (usando `-mavx2`). Olhe o código Assembly gerado e comente por que uma é mais rápida que a outra.
3. Seu programa deverá ser apresentável e auto contido. Realizar os testes não pode envolver mudar o código fonte na mão e recompilar.

Dica: você não precisa fazer tudo em C++. Você pode, por exemplo, carregar os dados de desempenho e escrever o relatório em um Jupyter Notebook ou mesmo gerar gráficos no Excel e depois colocá-los no texto.