Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
SSC0951 - Desenvolvimento de Código Otimizado

Atividade 4

Vetorização

Docente: Profa. Dra. Sarita Mazzini Bruschi

Danilo Alves - 10408390
Eduardo Amaral - 11735021
Rafael de Almeida - 11872028

Dezembro
2022

1 Introdução

Neste experimento foi analisado o impacto do uso de técnicas de vetorização por meio de funções *intrinsics* em um programa. O programa utilizado como base das análises é uma simples multiplicação de matrizes a fim de fazer um uso intenso de acesso à memoria e processamento de dados na forma vetorial. *Intrinsics* são funções codificadas em assembly que permitem usar chamadas de função e variáveis C/C++ no lugar de instruções de assembly. Além disso, as *intrinsics* fornecem acesso a instruções que não podem ser geradas usando as construções padrão das linguagens C e C++. As funções *intrinsics* permitem o programador explorar funcinalidades SIMD dos processadores.

2 Descrição do experimento

Para a presente análise, foram utilizadas três versões diferentes do mesmo programa. A primeira delas, sem nenhuma vetorização, foi compilada utilizando o *gcc* com a flag `-O1`, que não inclui otimizações relacionadas a vetorização. A segunda versão, além das flags `-O1` e `-ffast-math`, foi utilizada a flag explícita de vetorização `-ftree-vectorize`. Já ao compilar a última versão, a qual utilizamos as funções *intrinsics* para vetorizar o código explicitamente, utilizamos a flag `-O1` e habilitamos a possibilidade de se usar as extensões AVX com `-mavx`.

Para cada versão (sem vetorização, com vetorização automática pelo compilador e com vetorização explícita por meio de instruções *intrinsics*), foram realizadas dez execuções com diferentes tamanhos de entrada (10, 25, 50, 100, 250, 500, 1000, 2500, 5000, 7500), coletando o tempo médio dessas execuções, o número médio de *misses* de cache L1 e o número médio de *loads* de cache L1.

A função responsável pela multiplicação das matrizes sem vetorização é a seguinte:

```
double *matrixMultiplication(double *A, double *B, int N) {
    int i, j, k;
    double sum;

    double *C = (double *)calloc(N * N, sizeof(double));

    for (i = 0; i < N; i++) {
        for (k = 0; k < N; k++) {
            for (j = 0; j < N; j++) {
                C[flat2d(i, j, N)] += A[flat2d(i, k, N)] * B[flat2d(k, j, N)];
```

```
            }
        }
    }

    return C;
}
```

A função responsável pela multiplicação das matrizes com vetorização explícita por meio de *intrinsics* é a seguinte:

```
double *matrixMultiplication(double *A, double *B, int N) {
    int i, j, k;
    __m256d tmp, a, b;

    double *C = (double *)calloc(N * N, sizeof(double));

    for (i = 0; i < N; i++) {
        for (k = 0; k < N; k++) {

            a = _mm256_set1_pd(A[flat2d(i, k, N)]);

            for (j = 0; j < N - 4; j += 4) {
                b = _mm256_loadu_pd(&B[flat2d(k, j, N)]);
                tmp = _mm256_add_pd(_mm256_mul_pd(a, b),
                                    _mm256_loadu_pd(&C[flat2d(i, j, N)]));
                _mm256_storeu_pd(&C[flat2d(i, j, N)], tmp);
            }

            for (; j < N; j++) {
                C[flat2d(i, j, N)] += A[flat2d(i, k, N)] * B[flat2d(k, j, N)];
            }
        }
    }

    return C;
}
```

O repositório contendo todo o código utilizado para a realização do experimento pode ser encontrado em:
`https://github.com/rolimans/vectorizationExperiments`

2.1 Software e hardware utilizados

Para essa análise, o seguinte conjunto de software e hardware foram utilizados:

- Software:

Sistema operacional:	GNU/Linux
Kernel:	5.15.71-1-MANJARO
Compilador:	gcc 12.2.0
Linker:	GNU ld (GNU Binutils) 2.39.0
perf:	5.19.g3d7cb6b04c3f

- Informações sobre a CPU:

Arquitetura:	x86_64
Byte Order:	Little Endian
CPU(s):	4
Nome modelo:	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Thread(s) por core:	2
Core(s) por socket:	2
Socket(s):	1

- Caches (soma de todas):

L1d:	64 KiB (2 instâncias)
L1i:	64 KiB (2 instâncias)
L2:	512 KiB (2 instâncias)
L3:	3 MiB (1 instância)

- Informações sobre a memória

2x8GiB	SODIMM DDR4 Synchronous 2133 MHz (0,5 ns)
16GiB	Swap

3 Resultados

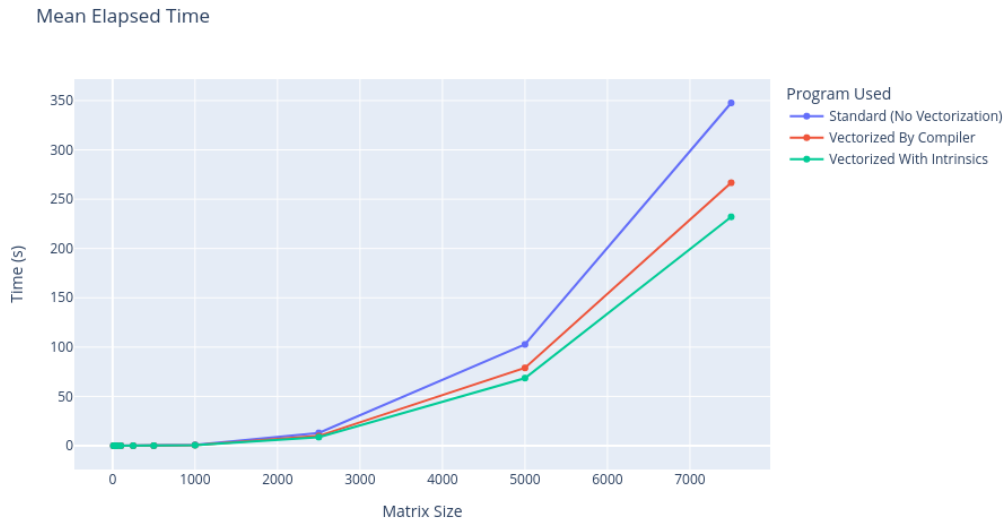

Figura 1: Tempo médio de execução

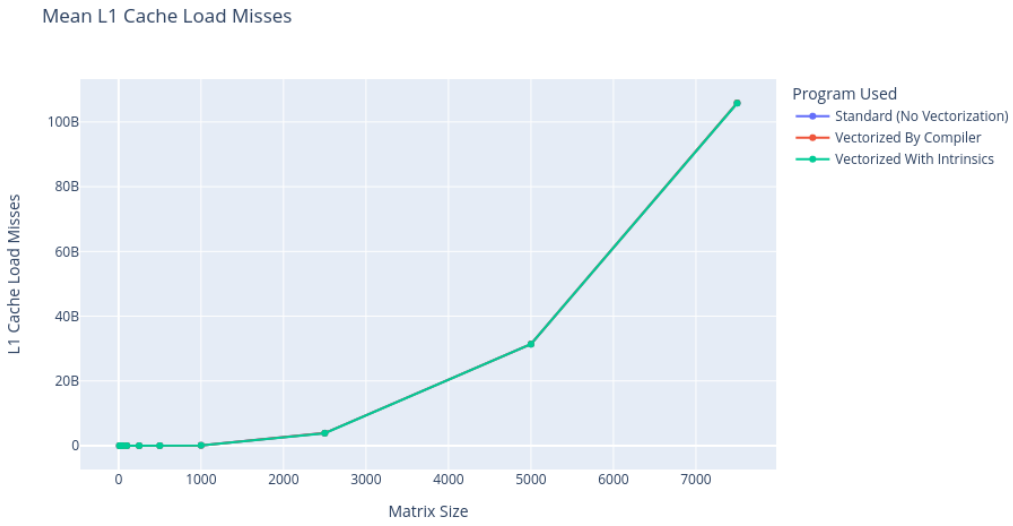

Figura 2: Média de *misses* de cache L1

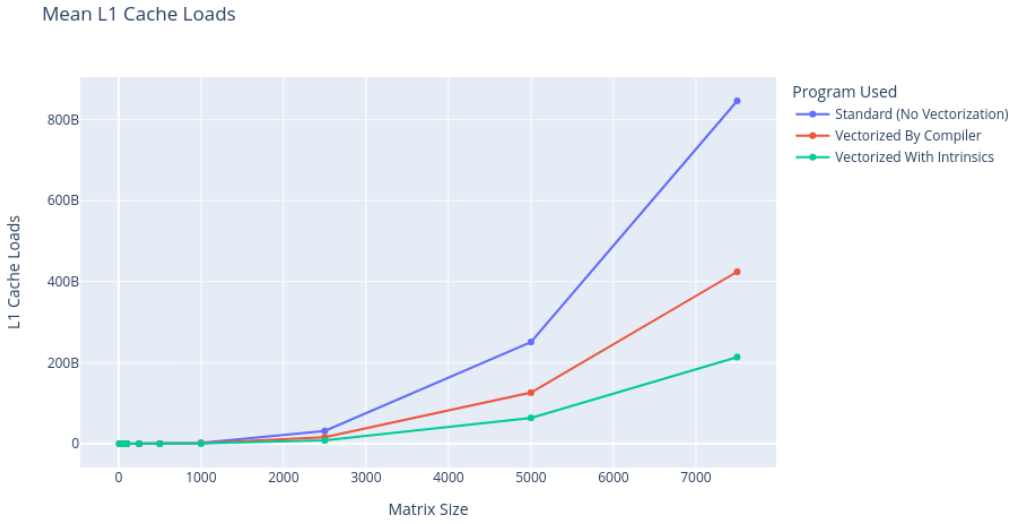

Figura 3: Média de *loads* de cache L1

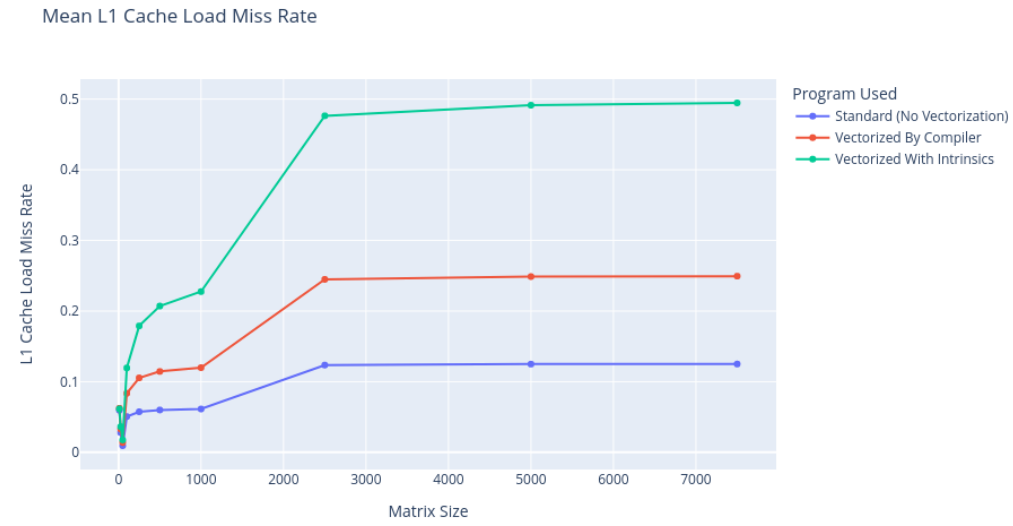

Figura 4: Razão *miss* / *load* de cache L1

4 Conclusão

É possível notar que o tempo de execução sem nenhuma vetorização foi sempre pior para todos os experimentos. A versão com a vetorização feita au-

tomaticamente pelo compilador performa melhor, mas ainda não consegue ser melhor que a versão feita com *intrinsics*. Tal melhoria de desempenho está associada com o maior número de dados processados por instrução das versões vetorizadas. A versão vetorizada com *intrinsics*, por exemplo, processa quatro elementos de cada matriz por instrução.

O número de Cache *Loads* feitos por cada versão segue a mesma tendência do tempo médio, mas é interessante notar que o número de *misses* permanece o mesmo entre elas.

A razão pela qual acredita-se que a versão vetorizada tem menos L1 Cache *Loads* do que a versão não vetorizada é porque ela é capaz de carregar dados da memória e realizar cálculos nesses dados com mais eficiência.

Na versão não vetorizada, cada iteração do loop mais interno executa uma única operação de multiplicação e adição em dois elementos das matrizes de entrada. Isso significa que, para realizar essas operações, os dois elementos acarretam em *loads* de cache L1.

Na versão vetorizada, cada iteração do loop mais interno usa uma única variável `__m256d` para executar quatro operações de multiplicação e adição em oito elementos das matrizes de entrada. Isso significa que o mesmo número de operações pode ser executado usando menos acessos à memória, reduzindo o número de *loads* de cache L1.

Além disso, a versão vetorizada usa as instruções `_mm256_loadu_pd` e `_mm256_storeu_pd` para carregar e armazenar dados da memória, o que pode ser mais eficiente do que as instruções padrão de carregamento e armazenamento usadas na versão não vetorizada. Isso reduz ainda mais o número de *loads* de cache L1.

Apesar da redução no número de *loads* de cache L1, o número de misses de cache L1 é o mesmo em ambas as versões porque ambas as versões acessam a memória em um padrão semelhante e, portanto, sofrem de evictions de cache L1 semelhantes.

Com o mesmo número de *misses* de cache e menos *loads*, a versão vetorizada possui uma Miss Rate de cache L1 mais alta.

Usar um vetor alinhado e instruções vetoriais que operam em variáveis alinhadas pode diminuir o número de faltas de cache L1 em alguns casos, o que reduziria a Miss Rate de cache L1.