Trabalho sobre Desempenho

Alunos:
Lucas Menduiña
Victor Miguez
Augusto Santos
Eduardo Chagas

Professora:
Juliana Mendes Nascente e Silva Zamith

Universidade Federal Rural do Rio de Janeiro
13/03/2023

Neste relatório, serão detalhadas as otimizações realizadas pelo grupo e os resultados obtidos a partir de benchmarks onde os códigos foram introduzidos e testados.

Primeiramente, será importante ressaltar a importância da vetorização: por que fazer? Ela irá permitir executar operações em grande conjunto de dados de maneira mais eficiente do que se fossem realizadas de forma serial. Em vez de fazer iterações sobre cada elemento do conjunto de dados e aplicar a operação, a vetorização permite que a operação seja aplicada a todos os elementos do conjunto de uma vez só, aproveitando as instruções de processamento paralelo disponíveis na maioria dos processadores modernos.

Dentre as diversas vantagens que acompanham a vetorização, pode-se incluir:

- **Velocidade:** a vetorização pode tornar o código mais rápido, já que reduz o número de instruções executadas pelo processador.
- **Legibilidade:** a vetorização pode tornar o código mais fácil de ler e entender, especialmente se houver trabalho com conjuntos de dados muito grandes.
- **Escalabilidade:** a vetorização permite que o código seja executado em grandes conjuntos de dados, tornando-o mais escalável.

A seguir, serão apresentados os desempenhos de dois programas distintos, cada um com sua respectiva versão: serial e otimizada. Também será apresentado o ganho de desempenho (speedup) após a implementação das otimizações na versão 2 de cada problema proposto, além da comparação dos índices de cache miss nos casos de testes que possuem como entrada 2048x2048 e 4096x4096. Os testes de performance foram realizados em uma máquina com as seguintes especificações:

- Intel i5 10400f
- DDR4 2x8 GBytes 2666 MHz
- **Cache L1** Data 6x32 KBytes
- **Cache L1** Inst 6x32 KBytes
- **Cache L2** 6x256 KBytes
- **Cache L3** 12 MBytes
- SSD 500 GBytes NVMe, Leitura: 3500 MB/s , Escrita: 2100 MB/s

O primeiro problema abordado foi o caso da multiplicação de matrizes, onde o primeiro código é isento de otimizações e o segundo possui melhorias como vetorização por meio das instruções SIMD (Single Instruction, Multiple Data), além de fazer acesso da matriz por linhas e não colunas (evitando cache misses e garantindo maior benefício das localidades temporal e espacial).

O segundo problema busca otimizar a verificação da ocorrência de uma matriz identidade, dada uma matriz qualquer. No primeiro código, também não houve qualquer melhoria implementada. Já no segundo, houve refinamento de

lógica de programação, acrescido da otimização por meio de instruções AVX2 (Advanced Vector Extensions 2).

O acesso a linhas em uma matriz pode otimizar buscas e reduzir a quantidade de stalls de memória porque os dados são armazenados de forma contínua na memória. Isso significa que, quando um programa lê uma linha da matriz, vários valores adjacentes também são carregados na cache. Se o programa precisar acessar outra linha imediatamente após, há uma boa chance de que essa linha também esteja armazenada na cache, permitindo um acesso rápido e eficiente.

Por outro lado, se o programa estiver acessando valores em locais aleatórios na memória, pode ser necessário buscar cada valor na memória principal, resultando em uma maior taxa de cache miss e, portanto, uma redução no desempenho. Além disso, o acesso aleatório à memória pode levar a uma maior quantidade de stalls de memória, onde o processador fica ocioso enquanto aguarda os dados serem buscados na memória principal.

Os testes de desempenho a seguir foram realizados no software Intel Vtune e possuem como finalidade comparar o tempo de execução de duas versões de um mesmo programa. Ambas as versões seguem uma mesma linha de raciocínio, embora possuam diferentes formas de implementação. A versão 1 foi implementada de forma serial, sem nenhum tipo de melhoria. Já a versão 2 foi implementada com instruções AVX2. A AVX2 é uma extensão de vetorização para o conjunto de instruções Intel x86, que permite que instruções SIMD (Single Instruction, Multiple Data) sejam executadas em vetores de 256 bits. O uso da AVX2 pode levar a melhorias significativas de desempenho em algoritmos que podem ser vetorizados e apresentam alta paralelização, resultando em menor latência e maior taxa de transferência para a CPU.

Veja abaixo os desempenhos da multiplicação de matrizes:

Tam. da Matriz	**TM Versão 1**	**TM Versão 2**	**SpeedUp(V2/V1)**
128 x 128	0.008*s*	0.039*s*	~0.02*x*
256 x 256	0.0064*s*	0.028*s*	~0.23*x*
512 x 512	0.595*s*	0.099*s*	~6.02*x*
1024 x 1024	7.003*s*	0.749*s*	~9.35*x*
2048 x 2048	85.967*s*	9.695*s*	~8.90*x*
4096 x 4096	941.367*s*	75.098*s*	~12.5*x*

Tabela de SpeedUp entre matriz não-vetorizada e matriz vetorizada (multiplicação de matrizes).

Diretivas de compilação

```
#include<stdio.h>
#include<immintrin.h>
#include<stdlib.h>
```

Veja abaixo o número de referências feitas a cada nível da memória cache e seus respectivos índices de cache miss e hit no problema da multiplicação de matrizes:

Para monitorar e medir o desempenho de todos os níveis da memória cache, foi utilizado o software Valgrind. Ele busca executar o programa e simular, em tempo de execução, as referências a cada nível da cache, bem como apontar falhas de dados e instruções (cache misses).

Legenda

L1- cache de dados L1
LL - (Last Level) L3

1. ***'D' cache reads*** (Dr, igual ao número de leituras na memória), D1 cache read misses (D1mr), and LL cache data read misses (DLmr).
2. ***'D' cache writes*** (Dw, igual ao número de escritas na memória), D1 cache write misses (D1mw), and LL cache data write misses (DLmw).

Linha de comando para teste :

```
$ valgrind --tool=cachegrind ./identidadeAVX2 4096
```

Teste 1 **: Multiplicação de Matrizes 4096 x 4096**

#	Versão 1 (Serial)	Versão 2 (Otimizada)
Dados Refs	$1.580 * 10^9$	$210.5 * 10^9$
D1 misses	$81.6 * 10^9$	$5.9 * 10^9$
LLd misses	$4.3 * 10^9$	$4.6 * 10^9$
D1 miss rate	5.2%	2.8%
LLd miss rate	0.3%	2.2%

Teste 2 **: Multiplicação de Matrizes 2048 x 2048**

#	Versão 1 (Serial)	Versão 2 (Otimizada)
Dados Refs	$197.7 * 10^9$	$26.4 * 10^9$
D1 misses	$8.7 * 10^9$	$687 * 10^6$
LLd misses	$539 * 10^6$	$539 * 10^6$
D1 miss rate	4.4%	2.6%
LLd miss rate	0.3 %	2.0%

Teste 3 **: Multiplicação de Matrizes 512 x 512**

#	Versão 1 (Serial)	Versão 2 (Otimizada)
Dados Refs	$9 * 10^9$	$426 * 10^6$
D1 misses	$158 * 10^6$	$11.4 * 10^6$
LLd misses	$51.6 * 10^3$	$51.6 * 10^3$
D1 miss rate	5.1%	2.7%
LLd miss rate	0.0%	0.0%

Versão 1 - Serial

```
    for(int i=0 ; i<LIN ; i++)
    {
        for(int j=0 ; j<COL ; j++)
        {
            for(int k=0 ; k<LIN ; k++)
            {
                mc[i][j] += ma[i][k] * mb[k][j] ;
            }
        }
    }

return 0 ;
}
```

Neste trecho de código, o qual se refere à versão 1 (sem otimização) da multiplicação de matrizes, haverá operações sendo realizadas de maneira serial (não-vetorizada), que irá gerar grande custo computacional excedente, além de maior número de instruções realizadas.

Versão 2 - Otimizada

```
for(int i = 0 ; i<LIN ; i++)
{
    for(int j = 0 ; j<COL ; j+=32)
    {
        __m256 soma256_0 = _mm256_setzero_ps();
        __m256 soma256_1 = _mm256_setzero_ps();
        __m256 soma256_2 = _mm256_setzero_ps();
        __m256 soma256_3 = _mm256_setzero_ps();

        for(int k = 0 ; k<LIN ; k+=2)
        {
            // Vetor com elementos iguais.
            __m256 Elemento_LIN = _mm256_set1_ps(matriz_A[i][k]) ;

            __m256 vect00 = _mm256_loadu_ps((float*)&matriz_B[k][j]) ;
            __m256 vect01 = _mm256_loadu_ps((float*)&matriz_B[k][j+8]) ;
            __m256 vect02 = _mm256_loadu_ps((float*)&matriz_B[k][j+16]) ;
            __m256 vect03 = _mm256_loadu_ps((float*)&matriz_B[k][j+24]) ;

            __m256 prod00 = _mm256_mul_ps(Elemento_LIN , vect00) ;
            __m256 prod01 = _mm256_mul_ps(Elemento_LIN , vect01) ;
            __m256 prod02 = _mm256_mul_ps(Elemento_LIN , vect02) ;
            __m256 prod03 = _mm256_mul_ps(Elemento_LIN , vect03) ;

            soma256_0 = _mm256_add_ps(soma256_0 , prod00) ;
            soma256_1 = _mm256_add_ps(soma256_1 , prod01) ;
            soma256_2 = _mm256_add_ps(soma256_2 , prod02) ;
            soma256_3 = _mm256_add_ps(soma256_3 , prod03) ;

            //REUTILIZAÇÃO DE REGISTRADORES
            Elemento_LIN = _mm256_set1_ps(matriz_A[i][k+1]) ;

            vect00 = _mm256_loadu_ps((float*)&matriz_B[k+1][j]) ;
            vect01 = _mm256_loadu_ps((float*)&matriz_B[k+1][j+8]) ;
            vect02 = _mm256_loadu_ps((float*)&matriz_B[k+1][j+16]) ;
            vect03 = _mm256_loadu_ps((float*)&matriz_B[k+1][j+24]) ;

            prod00 = _mm256_mul_ps(Elemento_LIN , vect00) ;
            prod01 = _mm256_mul_ps(Elemento_LIN , vect01) ;
            prod02 = _mm256_mul_ps(Elemento_LIN , vect02) ;
            prod03 = _mm256_mul_ps(Elemento_LIN , vect03) ;

            soma256_0 = _mm256_add_ps(soma256_0 , prod00) ;
            soma256_1 = _mm256_add_ps(soma256_1 , prod01) ;
            soma256_2 = _mm256_add_ps(soma256_2 , prod02) ;
            soma256_3 = _mm256_add_ps(soma256_3 , prod03) ;

        }
            _mm256_storeu_ps((float*)&matriz_C[i][j] , soma256_0) ;
            _mm256_storeu_ps((float*)&matriz_C[i][j+8] , soma256_1) ;
            _mm256_storeu_ps((float*)&matriz_C[i][j+16] , soma256_2) ;
            _mm256_storeu_ps((float*)&matriz_C[i][j+24] , soma256_3) ;
    }
}
```

Neste caso, retirado da segunda versão (otimizada), foram aplicadas instruções SIMD que garantirão que operações sejam aplicadas a todos os elementos de um conjunto (vetor, matriz) de uma só vez. Dessa forma, processadores modernos serão beneficiados pelo paralelismo.

No trecho de código acima, foram utilizados registradores AVX2. Estes registradores possuem tamanho de 256 bits cada um, isto é, na otimização implementada, foram armazenados ou operados 8 floats (32 bits) de uma só vez.

No código melhorado, o algoritmo busca replicar cada elemento da linha da matriz A, A(i,k) dentro de um registrador AVX2 e multiplicar pela linha de índice k da matriz B, B(k,j). Ao final de cada iteração de k, os valores obtidos serão somados (acumulados), para posteriormente serem armazenados na linha i da matriz resultante. Um exemplo minimalista será mostrado, para elucidar o passo a passo :

Matriz A

1	2
3	4

Matriz B

5	6
7	8

Elemento de A (j==1) * Linha de B (i==1)

1	1

x

5	6

=

5	6

Matriz A

1	2
3	4

Matriz B

5	6
7	8

Elemento de A (j==2) * Linha de B (i==2)

2	2

x

7	8

=

14	16

Matriz C
Primeira linha :

5	6

+

14	16

=

19	22

Matriz C

19	22
-	-

Assim, foi obtida a primeira linha da matriz resultante, fugindo do método tradicional de linhas x colunas, proposto na versão serial. Este método busca diminuir o número de instruções e iterações, além de aumentar a assertividade de dados na memória cache.

Veja abaixo os desempenhos da verificação da ocorrência de uma matriz identidade:

Tam. da Matriz	TM Versão 1	TM Versão 2	SpeedUp
128 x 128	0.022*s*	0.020*s*	1.10*x*
256 x 256	0.080*s*	0.026*s*	~ 3.08*x*
512 x 512	0.710*s*	0.103*s*	~ 6.90*x*
1024 x 1024	7.200*s*	0.777*s*	~ 9.27*x*
2048 x 2048	87.459*s*	10.720*s*	~ 8.16*x*
4096 x 4096	959.293*s*	79.593*s*	~ 12.05*x*

Tabela de SpeedUp entre matriz não-vetorizada e matriz vetorizada (matriz identidade).

Veja abaixo o número de referências feitas a cada nível da memória cache e seus respectivos índices de cache miss e hit no problema da multiplicação de matrizes:

Para monitorar e medir o desempenho de todos os níveis da memória cache, foi utilizado o software Valgrind. Ele busca executar o programa e simular, em tempo de execução, as referências a cada nível da cache, bem como apontar falhas de dados e instruções (cache misses).

Legenda

L1- cache de dados L1
LL - (Last Level) L3

1. ***'D' cache reads*** (Dr, igual ao número de leituras na memória), D1 cache read misses (D1mr), and LL cache data read misses (DLmr).
2. ***'D' cache writes*** (Dw, igual ao número de escritas na memória), D1 cache write misses (D1mw), and LL cache data write misses (DLmw).

Linha de comando para teste :

```
$ valgrind --tool=cachegrind ./identidadeAVX2 4096
```

Teste 1 **: Verificar identidade 4096 x 4096**

#	Versão 1 (serial)	Versão 2 (Otimizada)
Dados Refs	$1.582 * 10^9$	$207 * 10^9$
D1 misses	$81 * 10^9$	$5.5 * 10^9$
LLd misses	$4.3 * 10^9$	$4.5 * 10^9$
D1 miss rate	5.2%	2,7%
LLd miss rate	0.3%	2,2%

Teste 2 **: Verificar identidade 2048 x 2048**

#	Versão 1 (serial)	Versão 2 (Otimizada)
Dados Refs	$198 * 10^9$	$26.3 * 10^9$
D1 misses	$10.2 * 10^9$	$697 * 10^6$
LLd misses	$542 * 10^6$	$548.5 * 10^6$
D1 miss rate	5.2%	2.7%
LLd miss rate	0.3%	2.1%

Teste 3 **: Verificar identidade 512 x 512**

#	Versão 1 (serial)	Versão 2 (Otimizada)
Dados Refs	$3.1 * 10^9$	$419.6 * 10^6$
D1 misses	$159 * 10^6$	$11 * 10^6$
LLd misses	$51.6 * 10^3$	$51.6 * 10^3$
D1 miss rate	5.1%	2.6%
LLd miss rate	0.0%	0.0%

Versão 1 - serial

```
    for(int i=0 ; i < TAM ; i++)
    {
        for(int j=0 ; j < TAM ; j++)
        {
            for(int k=0 ; k < TAM ; k++)
            {
                mc[i][j] += mi[i][k]*mb[k][j] ;
            }
        }
    }

    for(int i=0 ; i<TAM ; i++)
    {
        for(int j=0 ; j<TAM ; j++)
        {
            if(mc[i][j] != mb[i][j])
            {
                return -1 ;
            }
        }
    }
return 0 ;
}
```

Utilizando a definição(*) da matriz identidade, podemos afirmar que, para quaisquer matrizes arbitrárias A e B, se A*B = A, então B é uma matriz identidade. Dessa forma, conclui-se que a matriz identidade é o elemento neutro da multiplicação matricial.

O trecho implementado utiliza duas lógicas que requerem recursos computacionais elevados, além de um grande número de instruções. O primeiro custo computacional significativo é no produto matricial entre duas matrizes, como mencionado no problema anterior sobre a multiplicação de matrizes. Além disso, após realizar a multiplicação, a matriz resultante “mc” deve ser comparada com a matriz “mb” elemento por elemento, para verificar sua igualdade. Se as matrizes forem iguais, então “mi” é uma matriz identidade.

<u>Versão 2 - Otimizada</u>

A versão 2 utiliza a multiplicação de matrizes para verificar se uma matriz arbitrária se caracteriza como **<u>identidade</u>**. Portanto, na primeira parte do código, foi utilizada a mesma lógica do produto matricial apresentado anteriormente, na versão otimizada do problema 1. Entretanto, uma lógica adicional foi inserida para tal afirmação, veja a seguir:

```
__m256i all_one = _mm256_set1_epi32(1) ;
for(int i=0 ; i<LIN ; i++)
{
    for(int j=0 ; j<COL ; j+=32)
    {
        __m256i sum = _mm256_setzero_si256() ;
        //MATRIZ RESULTANTE
        __m256i mc_8ints_0 = _mm256_loadu_si256((__m256i*)&mc[i][j]) ;
        __m256i mc_8ints_1 = _mm256_loadu_si256((__m256i*)&mc[i][j + 8]) ;
        __m256i mc_8ints_2 = _mm256_loadu_si256((__m256i*)&mc[i][j + 16]) ;
        __m256i mc_8ints_3 = _mm256_loadu_si256((__m256i*)&mc[i][j + 24]) ;

        //MATRIZ ALEATORIA
        __m256i mb_8ints_0 = _mm256_loadu_si256((__m256i*)&mb[i][j]) ;
        __m256i mb_8ints_1 = _mm256_loadu_si256((__m256i*)&mb[i][j + 8]) ;
        __m256i mb_8ints_2 = _mm256_loadu_si256((__m256i*)&mb[i][j + 16]) ;
        __m256i mb_8ints_3 = _mm256_loadu_si256((__m256i*)&mb[i][j + 24]) ;

        //SE OS PEDAÇOS SAO IGUAIS, ENTAO SUB DA ZERO
        __m256i verif_iguais_0 = _mm256_sub_epi32(mc_8ints_0 , mb_8ints_0) ;
        __m256i verif_iguais_1 = _mm256_sub_epi32(mc_8ints_1 , mb_8ints_1) ;
        __m256i verif_iguais_2 = _mm256_sub_epi32(mc_8ints_2 , mb_8ints_2) ;
        __m256i verif_iguais_3 = _mm256_sub_epi32(mc_8ints_3 , mb_8ints_3) ;

        sum = _mm256_add_epi32(sum , verif_iguais_0) ;
        sum = _mm256_add_epi32(sum , verif_iguais_1) ;
        sum = _mm256_add_epi32(sum , verif_iguais_2) ;
        sum = _mm256_add_epi32(sum , verif_iguais_3) ;

        int IF_ZERO_SET_ONE ;
        IF_ZERO_SET_ONE = _mm256_testc_si256(all_one , sum) ;

        if(IF_ZERO_SET_ONE != 1)
            exit(1) ;

    }
}
```

Um pequeno exemplo pode ser utilizado para tornar a lógica mais clara, para os que não estão familiarizados com instruções SIMD:

Matriz A

1	2
3	4

Matriz B

1	0
0	1

‘R’ a matriz resultante.

Logo, pela definição mencionada anteriormente (*), a matriz identidade é o elemento neutro da multiplicação de matrizes. Dessa forma, tem-se A*B = R , onde R == A. Portanto, (A - R) = 0, onde 0 é a matriz nula. Segue o algoritmo realizado :

1. A subtração de partes, 8 a 8 elementos, das linhas da matriz ‘mb’ com as de ‘mc’ é acumulada no registrador ‘sum’ ;
2. Por fim, a instrução ***_mm256_testc_si256()*** verifica o vetor ‘sum’, bit a bit, e retorna 1 se este for nulo ;
3. Se for nulo, então teste a próxima linha, repetindo os passos anteriores.
4. Senão, pare! A matriz não é identidade.

Dessa forma, feitas todas as iterações, conclui-se que a matriz ‘mi’ é uma matriz identidade.

O trecho de código da versão 2 (Otimizada) evidencia a importância da vetorização em otimizações de desempenho em aplicações computacionais. Em vez de realizar uma iteração serial e operar elemento por elemento, a vetorização permite tratar um grande conjunto de elementos simultaneamente, diminuindo o número de iterações necessários para realizar um cálculo, comparações, loads e stores.

Essa abordagem é conhecida como processamento em paralelo e pode resultar em um desempenho significativamente melhor em comparação com a execução serial. Além disso, a utilização de registradores AVX2 permite aproveitar ao máximo a capacidade do hardware moderno, que é projetado para processar grandes conjuntos de dados em paralelo.

Conclusão

Os problemas apresentados possuem como objetivo processar, por meio de operações matriciais, se uma matriz arbitrária se caracteriza como matriz identidade e realizar a multiplicação de matrizes. Vários testes foram realizados acerca do problema, com a finalidade de comparar duas implementações distintas do mesmo programa, uma serial e uma otimizada. O tempo de execução foi calculado para ambos os programas e comparados em diferentes casos de testes (128x128, 256x256, 512x512, 1024x1024, 2048x2048, 4096x4096). Em seguida, foi monitorado, com auxílio do software Valgrind, o acesso à cada nível da memória cache. Este monitoramento busca elucidar o número de cache misses para cada implementação e, posteriormente, concluir qual dos programas custará mais recursos computacionais para buscar os dados e instruções nos níveis inferiores da hierarquia.

Para ambos os problemas, quando comparados os desempenhos da memória cache entre suas versões, dada uma mesma entrada, fica evidente que a vetorização implementada, além de reduzir o número de instruções realizadas, aumenta a assertividade dos dados e instruções nos níveis da memória cache, ou seja, o cache hit torna-se mais frequente. Por outro lado, a análise mostra que para todos os testes, o cache miss da versão serial é maior, bem como seu miss rate em todos os níveis.

Além dos índices de acesso à cache, cada versão foi comparada para calcular quanto de speedup a versão 2 (otimizada), munida das instruções SIMD, ganhou em relação a versão 1, dada uma mesma entrada. Em ambos os problemas, observa-se valores de speedup que podem atingir até 12x, dependendo do tamanho da entrada.

Considerações finais

Em síntese, a vetorização de dados utilizando instruções SIMD é uma técnica poderosa que traz inúmeros benefícios para a otimização de desempenho em aplicações computacionais. Essa técnica permite que operações matemáticas complexas sejam realizadas de forma mais eficiente, utilizando recursos de hardware modernos, tais como registradores AVX2, que são capazes de processar grandes conjuntos de dados simultaneamente.

Além disso, a vetorização também pode resultar em uma redução significativa no uso de memória e uma simplificação na implementação de algoritmos, tornando o código mais fácil de entender e manter.

Com o avanço da tecnologia e o aumento da quantidade de dados a serem processados, a vetorização se tornou cada vez mais importante no mundo atual, permitindo que as aplicações computacionais possam utilizar de maneira mais eficiente os recursos disponíveis e fornecer um desempenho mais rápido e eficiente.

Dessa forma, a vetorização se tornou uma técnica fundamental em diversas áreas, como processamento de imagens, reconhecimento de fala, inteligência

artificial, entre outras. Portanto, compreender e aplicar essa técnica de otimização de desempenho é fundamental para manter a competitividade em um mundo cada vez mais digital e conectado.

FIM

Referências:

Code Modernization: Bringing Codes Into the Parallel Age (hpcwire.com)
Memory Usage View (intel.com)
Crunching Numbers with AVX and AVX2 - CodeProject
Programar em C/Pré-processador - Wikilivros (wikibooks.org)
Intel® Intrinsics Guide