

DGEMM - Going Faster

Carolina Santiago
Gustavo Roxo
Zuilho Segundo

Engenharia de Computação e Informação
Universidade Federal do Rio de Janeiro - UFRJ
Brasil
Julho 2023

Sumário

| | | |
|----------|----------------------------------|----------|
| 1 | Introdução | 2 |
| 2 | Criação dos Dados | 2 |
| 3 | Going Faster - Capítulo 1 | 2 |
| 4 | Going Faster - Capítulo 2 | 3 |
| 5 | Going Faster - Capítulo 3 | 3 |
| 6 | Going Faster - Capítulo 4 | 4 |
| 7 | Going Faster - Capítulo 5 | 5 |
| 8 | Going Faster - Capítulo 6 | 6 |
| 9 | Experimentos e Testes | 7 |

1 Introdução

O seguinte relatório tem por objetivo descrever a exploração dos códigos de Double-precision General Matrix Multiply (DGEMM) propostos pelo livro Computer Organization and Design RISC-V. O trabalho consiste na evolução das ferramentas utilizadas para a realização da tarefa, começando com uma linguagem de alto nível, no caso Python, e chegando até um código em C que se aproveita da utilização dos registradores e dos múltiplos cores de um processador.

Primeiro, é preciso entender a motivação por trás disso. Dentro da computação e dos problemas do mundo real, muitas coisas envolvem operações com matrizes. Se pensamos em uma aplicação moderna, Redes Neurais não são nada mais do que um conjunto gigantesco de matrizes sendo multiplicadas e atualizadas a cada iteração. Dentro desse contexto, se faz necessário que sejamos capazes de realizar essas operações de maneira eficiente, para que a multiplicação de matrizes muito grandes seja realizada de maneira rápida. Para que isso seja possível é possível conhecer as estruturas do processador e como podemos utilizá-las para resolver o problema.

Uma parte importante sobre a experimentação é o computador em que os códigos foram executado. No caso desse relatório, o computador utilizado possui um processador 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz e 8GB de RAM.

2 Criação dos Dados

Para a criação das matrizes, foi utilizado o seguinte código em C, que gera matrizes aleatórias a cada execução do código:

```
void randomize_matrix(double *A, int m, int n){
    srand(time(NULL));
    int i, j;
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            A[i * n + j] = (double)(rand() % 100) + 0.01 * (rand() % 100);
            if (rand() % 2 == 0) A[i * n + j] *= 1.0;
        }
    }
}
```

Ele recebe o endereço da matriz A e suas dimensões, inserindo a cada iteração os valores nas entradas da matriz.

3 Going Faster - Capítulo 1

Nessa etapa, foi proposto a utilização de um código em Python para a realização da multiplicação das matrizes. Para esse código, não utilizamos nenhuma biblioteca na multiplicação, apenas para a geração das matrizes aleatórias para a multiplicação. Segue o código utilizado:

```
t0 = time.time()
for i in range(tam):
    for j in range(tam):
        for k in range(tam):
            c[i][j] += m[i][k] * n[k][j]
t1 = time.time()
tempo = t1 - t0
```

Como esperado, essa multiplicação demorou bastante. Tendo sido executada em 44480.73 segundos, ou 12 horas, 21 minutos e 20 segundos.

Por exemplo, utilizando a biblioteca numpy, que implemente operações matemáticas de maneira mais eficiente, se utilizando de recursos de C. Teríamos o mesmo código rodando em 3.33 segundos.

```

a = np.array(m)
b = np.array(n)

#Multiplication
start = time.time()
c = np.dot(a, b)
end = time.time()

```

4 Going Faster - Capítulo 2

Para essa próxima etapa, realizamos a mudança de linguagem, de Python para C, uma linguagem de mais baixo nível. Isso significa que ela está muito mais próxima da linguagem da máquina, sendo capaz de realizar operações de maneira mais eficiente.

Como modificações nessa etapa do código, é utilizado uma versão unidimensional das matrizes, além de se aproveitar da vantagem de utilização dos ponteiros de memória em C, que possuem melhor desempenho que acesso direto a memória.

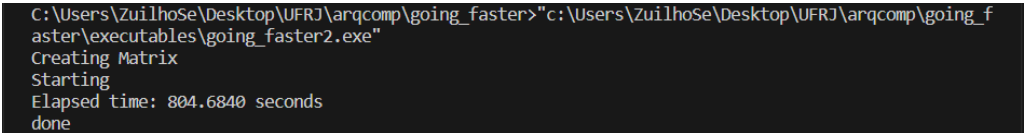
Foram feitas algumas modificações no código, para que funcionasse com o processador do computador em uso. Segue o código:

```

void dgemm (int n, double* A, double* B, double* C){
    int i, j, k;
    for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j){
        double cij = C[i+j*n]; /* cij = C[i][j] */
        for(k = 0; k < n; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
        C[i+j*n] = cij; /* C[i][j] = cij */
    }
}

```

Nessa versão, ainda iteramos sobre cada elemento usando os *for*, porém as modificações acima mencionadas geram um ganho de performance, rodando em 13 minutos e 24 segundos.



```

C:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster>"c:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster\executables\going_faster2.exe"
Creating Matrix
Starting
Elapsed time: 804.6840 seconds
done

```

Figura 1: DGEMM - Cap2

5 Going Faster - Capítulo 3

De maneira simplificada, nessa etapa nos utilizamos operações SIMD (Single Instruction, Multiple Data) que são importadas junto das funções AVX da biblioteca `x86intrin.h`.

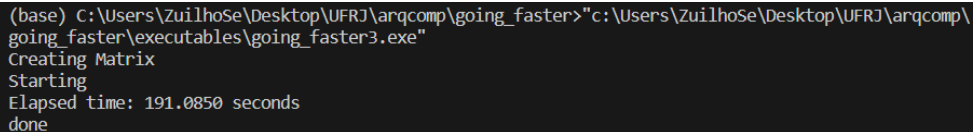
São percorridos blocos com 4 elementos, e as operações são realizadas em cada um dos blocos, utilizando as operações vetoriais do AVX. Dentro do loop mais interno, são realizadas as seguintes operações:

- Carregar o valor atual de $C[i][j]$ em um registrador SIMD de 256 bits ($c0$);
- Realizar um loop sobre a dimensão k e acumular o resultado da multiplicação de cada elemento de $A[i][k]$ com o respectivo elemento de $B[k][j]$, utilizando operações SIMD.
- Armazenar o valor atualizado de $C[i][j]$ de volta na memória.

Foram feitas algumas modificações no código, para que funcionasse com o processador do computador em uso. Segue o código:

```
void dgemm(size_t n, double* A, double* B, double* C){
    size_t i, j, k;
    for(i = 0; i < n; i+=4)
        for(j = 0; j < n ; j ++){
            __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]*/
            for(k = 0; k < n; k++)
                c0 = _mm256_add_pd(c0, /* c0 += A[i][k] * B[k][j] */
                    _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
                        _mm256_broadcast_sd(B+k+j*n)));
            _mm256_store_pd(C+i+j*n,c0); /* C[i][j] = c0*/
        }
}
```

Com essa nova implementação, tivemos a multiplicação acontecendo em aproximadamente 1/4 do tempo, em 3 minutos e 11 segundos.



```
(base) C:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster>"c:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\
going_faster\executables\going_faster3.exe"
Creating Matrix
Starting
Elapsed time: 191.0850 seconds
done
```

Figura 2: DGEMM - Cap3

6 Going Faster - Capítulo 4

Nessa etapa, começamos a utilizar a técnica de UNROLLING que foi aprendida no capítulo. Definimos a constante UNROLL com valor 4, sendo esse o número de vezes que o loop interno será desenrolado. Isso significa que em vez de realizarmos as operações para cada elemento individualmente, o loop interno é executado para blocos de 4 elementos simultaneamente.

Aqui, criamos um vetor $c[\text{UNROLL}]$, composto de 4 registradores `'__m256d'`, o que nos permite realizar quatro operações de maneira simultânea. Dentro do loop mais interno, em vez de carregar e armazenar os elementos individualmente, agora utilizamos loops para processar os 4 elementos de uma vez. Os registradores SIMD $c[r]$ são carregados com os valores atuais de $C[i+r*4][j]$. Em seguida, é realizado um loop sobre a dimensão k e, para cada elemento de $A[n*k+r*4+i]$, é realizado o cálculo com o respectivo elemento de $B[k][j]$, armazenando o resultado em $c[r]$. Por fim, é feito um loop para armazenar os 4 elementos de c de volta em $C[i+r*4][j]$.

Foram feitas algumas modificações no código, para que funcionasse com o processador do computador em uso. Segue o código:

```
void dgemm(size_t n, double* A, double* B, double* C){
    for(size_t i = 0; i < n; i+= UNROLL * 4)
        for(size_t j = 0; j < n ; j ++){
            __m256d c[4];
            for (int r =0; r < UNROLL; r++)
                c[r] = _mm256_load_pd(C+i+r*4+j*n);

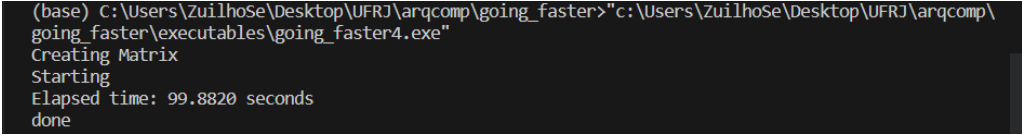
            for(int k = 0; k < n ; k ++){
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for(int r =0; r <UNROLL; r++)
                    c[r] = _mm256_add_pd(c[r],
                        _mm256_mul_pd(_mm256_load_pd(A+n*k+r*4+i),b));
            }
        }
}
```

```

        for(int r = 0 ; r < UNROLL; r++)
            _mm256_store_pd(C+i+r*4+j*n, c[r]);
    }
}

```

Com essa nova implementação, tivemos a multiplicação acontecendo em aproximadamente metade do tempo em comparação ao anterior, em 1 minutos e 39 segundos.



```

(base) C:\Users\ZuillhoSe\Desktop\UFRJ\arqcomp\going_faster>"c:\Users\ZuillhoSe\Desktop\UFRJ\arqcomp\
going_faster\executables\going_faster4.exe"
Creating Matrix
Starting
Elapsed time: 99.8820 seconds
done

```

Figura 3: DGEMM - Cap4

7 Going Faster - Capítulo 5

Nessa etapa, além do UNROLLING, utilizamos a divisão em blocos da matriz. Essa divisão em blocos, permite a explicitação de um acesso eficiente à memória cache, melhorando o desempenho. A função "do_block" é chamada para executar em cima de cada um dos blocos separados dentro da função "dgemm".

Dentro da função "do_block", os loops externos percorrem os blocos da matriz de entrada C, e para cada bloco, um novo conjunto de loops aninhados é executado para processar os elementos desse bloco.

Os registradores SIMD c são inicializados e carregados com os valores atuais de C. Em seguida, são realizados loops sobre as dimensões k e r, semelhante às implementações anteriores, para calcular a multiplicação de matrizes em blocos menores.

No final, os resultados são armazenados de volta na matriz C. A função "dgemm" utiliza loops aninhados para percorrer os blocos da matriz e chama a função "do_block" para processar cada bloco. Essa abordagem de divisão em blocos permite otimizar o uso da memória cache e realizar o cálculo de forma mais eficiente.

Foram feitas algumas modificações no código, para que funcionasse com o processador do computador em uso. Segue o código:

```

void do_block (int n, int si, int sj, int sk, double *A, double *B, double *C){
    for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*8 )
        for ( int j = sj; j < sj+BLOCKSIZE; j++) {
            __m256d c[UNROLL];
            for ( int r = 0; r < UNROLL; r++)
                c[r] = _mm256_load_pd(C+i+r*4+j*n);
            for (int k = sk; k < sk+BLOCKSIZE; k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int r = 0; r < UNROLL; r++)
                    c[r] = _mm256_add_pd(c[r], _mm256_mul_pd(_mm256_load_pd(A+n*k+r*4+i), b));
            }
            for ( int r = 0; r < UNROLL; r++)
                _mm256_store_pd(C+i+r*4+j*n, c[r]);
        }
}

void dgemm (int n, double* A, double* B, double* C){
    for(int sj = 0; sj < n; sj += BLOCKSIZE)
        for(int si = 0; si < n; si += BLOCKSIZE)
            for(int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}

```

Com essa nova implementação, tivemos a multiplicação acontecendo em aproximadamente 1/10 do tempo em comparação ao anterior, em 10.9 segundos.

```
(base) C:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster>"c:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster\executables\going_faster5.exe"
Creating Matrix
Starting
Elapsed time: 10.9570 seconds
done
```

Figura 4: DGEMM - Cap5

8 Going Faster - Capítulo 6

Nessa etapa, adicionamos diretivas de OpenMP para paralelização do processamento dos blocos da matriz.

A diretiva `#pragma omp parallel for` é adicionada antes do loop externo em `dgemm`. Essa diretiva indica ao compilador que o loop seguinte (`for`) pode ser executado em paralelo por várias threads. Cada thread executará uma parte do loop, dividindo o trabalho entre elas.

Dessa forma, os blocos da matriz `C` são processados em paralelo por diferentes threads, melhorando o desempenho em sistemas com múltiplos núcleos ou processadores.

Essa abordagem é possível porque a multiplicação de matrizes em cada bloco é uma tarefa independente, não havendo dependência de dados entre eles. Portanto, é possível dividir o trabalho entre as threads sem conflitos.

Foram feitas algumas modificações no código, para que funcionasse com o processador do computador em uso. Segue o código:

```
void do_block (int n, int si, int sj, int sk, double *A, double *B, double *C){
    for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*8 )
        for ( int j = sj; j < sj+BLOCKSIZE; j++) {
            __m256d c[UNROLL];
            for ( int r = 0; r < UNROLL; r++)
                c[r] = _mm256_load_pd(C+i+r*4+j*n);
            for (int k = sk; k < sk+BLOCKSIZE; k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int r = 0; r < UNROLL; r++)
                    c[r] = _mm256_add_pd(c[r], _mm256_mul_pd(_mm256_load_pd(A+n*k+r*4+i), b));
            }
            for ( int r = 0; r < UNROLL; r++ )
                _mm256_store_pd(C+i+r*4+j*n, c[r]);
        }
}

void dgemm (int n, double* A, double* B, double* C){
    #pragma omp parallel for
    for(int sj = 0; sj < n; sj += BLOCKSIZE)
        for(int si = 0; si < n; si += BLOCKSIZE)
            for(int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

Com essa nova implementação, não tivemos um ganho considerável comparado ao anterior, reduzindo em apenas 0.3 segundos, rodando na sua totalidade em 10.64 segundos.

```
(base) C:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster>"c:\Users\ZuilhoSe\Desktop\UFRJ\arqcomp\going_faster\executables\going_faster6.exe"  
Creating Matrix  
Starting  
Elapsed time: 10.6450 seconds  
done
```

Figura 5: DGEMM - Cap6

9 Experimentos e Testes

Decidimos realizar os testes na última implementação que é a que possui o melhor tempo de execução e é um conjunto de todas as técnicas que foram desenvolvidas até aqui.

O primeiro teste foi realizado no tempo de execução para diferentes tamanhos de matrizes. Como esperado, o tempo continua crescendo, de maneira muito rápida, já que o número de elementos aumenta por 4, quando dobramos as dimensões da matriz.

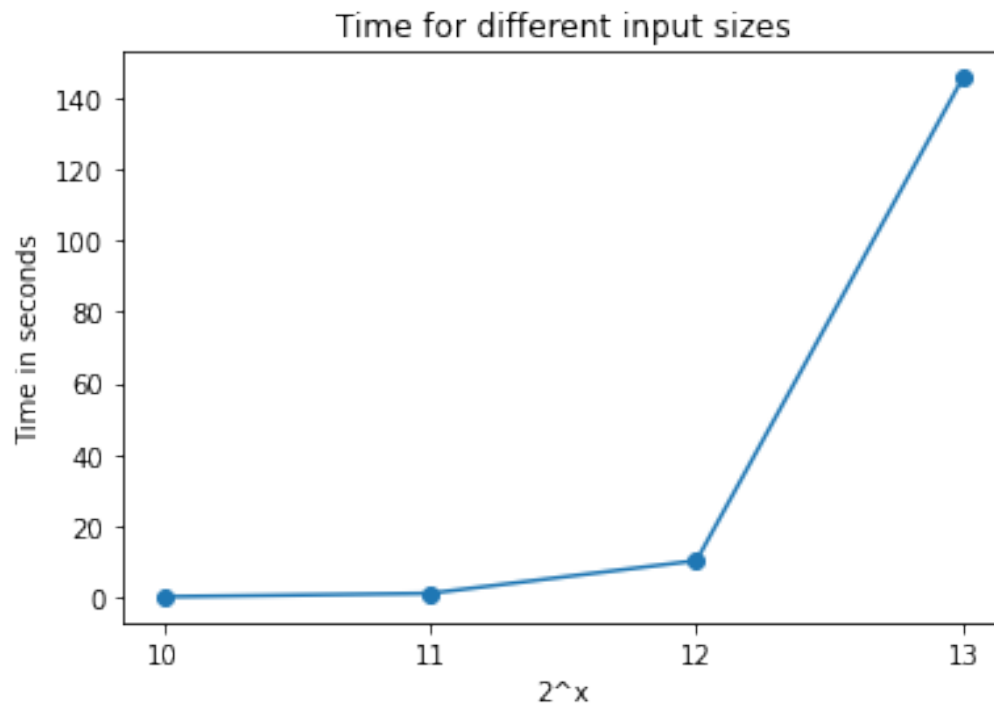


Figura 6: Tempo x Tamanho

O segundo teste foi realizado modificação do tamanho dos blocos. Como pode ser visto pelo gráfico, para valores maiores do que 32, o tempo volta a subir, apesar de ter caído conforme iam crescendo antes. Sendo assim, nos parece que o bloco de tamanho 32 é ótimo para esse problema sendo executado na arquitetura que foi executado.

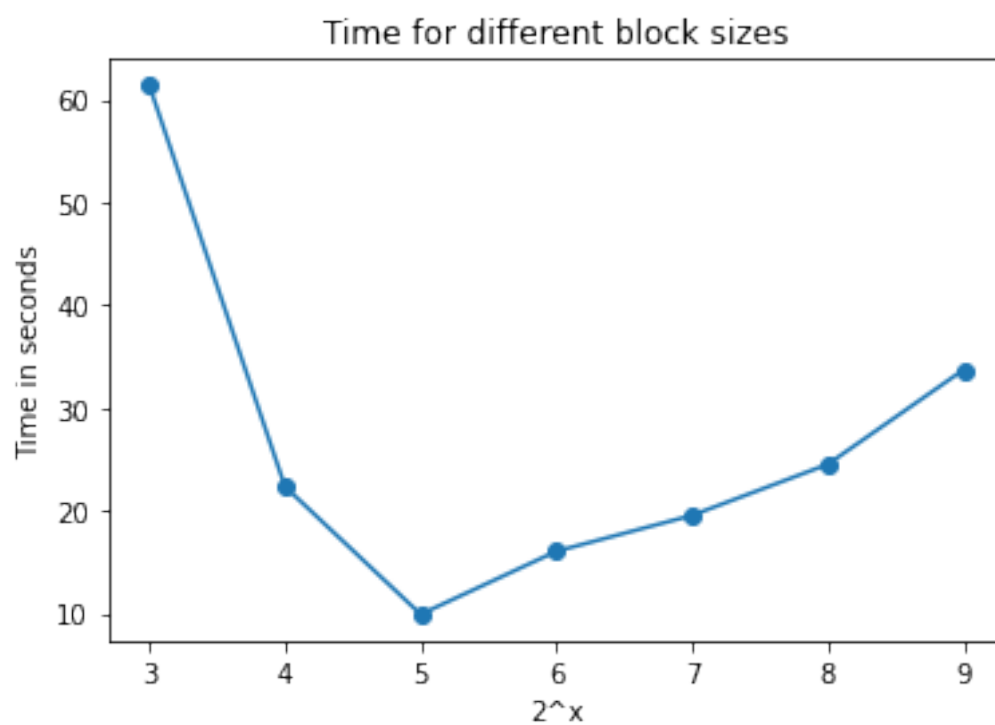


Figura 7: Tempo x Tamanho do Bloco