Relatório Técnico: Implementação e Avaliação da Função DGEMM

João Chamarelli, Pedro Tuttmann, Antônio Magalhães, Brenno Santos, Gabriel Pereira

[1]DEL - Arquitetura de Computadores - EEL580

Prof. Diego Leonel Dutra

1. Introdução

O presente trabalho tem como objetivo estudar, implementar e comparar diferentes versões da função DGEMM (Double-precision General Matrix Multiply), amplamente utilizada em aplicações científicas e de engenharia para a multiplicação de matrizes de precisão dupla. Foram desenvolvidas e testadas diversas variações da DGEMM, tanto em Python quanto em C, explorando desde a implementação mais ingênua (triple loop) até abordagens otimizadas que utilizam vetorização (SIMD), cache blocking e paralelismo com OpenMP. O foco da análise recai sobre a eficiência computacional de cada implementação, mensurada pelo tempo médio de execução e pelo desvio padrão em diferentes tamanhos de matrizes. A comparação sistemática entre as versões visa demonstrar a relevância de boas práticas de otimização de código para o desempenho de aplicações intensivas em cálculo.

2. Capítulo 1

2.1. DGEMM Triple-Loop

Essa é a implementação mais simples da multiplicação de matrizes, escrita em Python. Ela utiliza três laços aninhados para iterar pelos elementos das matrizes A e B e calcular os elementos da matriz C. A estrutura da função permanece inalterada, mas foram adicionadas duas funcionalidades auxiliares ao redor dela:

- Uma função para gerar matrizes quadradas aleatórias com valores reais entre 0 e 10.

- Um bloco principal que utiliza ‘sys.argv‘ para permitir que o usuário defina o tamanho da matriz pela linha de comando, e mede o tempo de execução com a biblioteca ‘time‘.

Essa estrutura permite testar o desempenho da multiplicação com diferentes tamanhos de entrada de forma flexível e automatizada.

Implementação do código em Python:

```python
import random
import time
import sys

def dgemm(n, A, B):
    C = [[0.0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

def gerar_matriz_aleatoria(n):
    return [[random.uniform(0, 10) for _ in range(n)] for _ in range(n)]

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Uso: python script.py <tamanho_da_matriz>")
        sys.exit(1)
```

```
    try:
        n = int(sys.argv[1])
    except ValueError:
        print("Erro: o tamanho da matriz deve ser um numero inteiro.")
        sys.exit(1)

    A = gerar_matriz_aleatoria(n)
    B = gerar_matriz_aleatoria(n)

    inicio = time.time()
    C = dgemm(n, A, B)
    fim = time.time()

print(f"Tempo de execucao para matriz {n}x{n}: {fim - inicio:.4f} segundos")
```

Implementação do código em C:

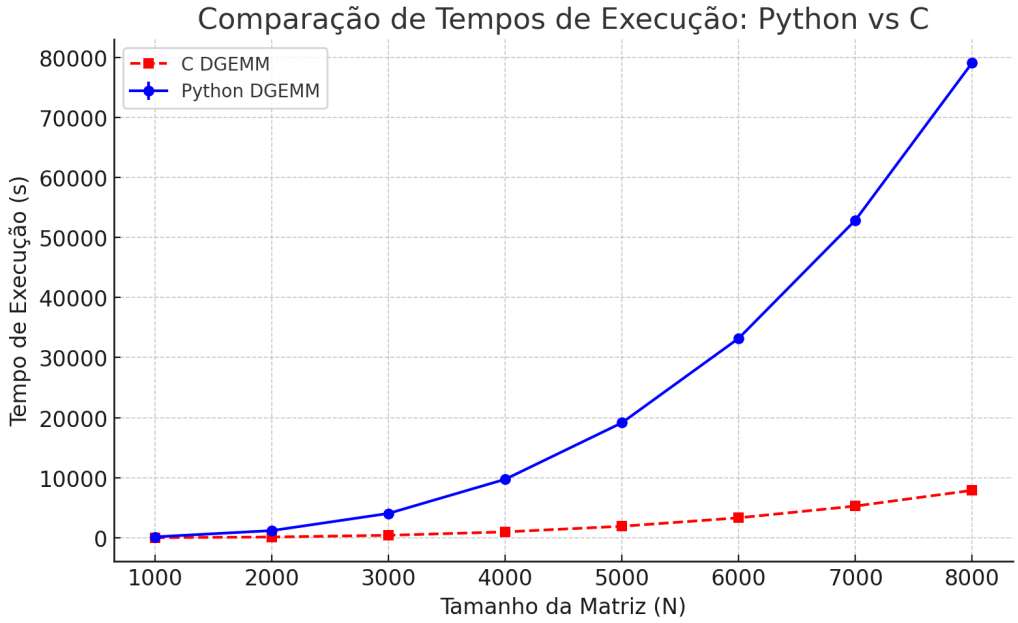
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

double** aloca_matriz(int n) {
    double** mat = (double**) malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        mat[i] = (double*) calloc(n, sizeof(double));
    }
    return mat;
}

void preenche_matriz(double** mat, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            mat[i][j] = (double)((i * j) % 100);
}

void libera_matriz(double** mat, int n) {
    for (int i = 0; i < n; i++)
        free(mat[i]);
    free(mat);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Uso: %s <tamanho_da_matriz>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

    double** A = aloca_matriz(n);
```

```
    double** B = aloca_matriz(n);
    double** C = aloca_matriz(n);

    preenche_matriz(A, n);
    preenche_matriz(B, n);

    clock_t inicio = clock();
    dgemm_basico(n, A, B, C);
    clock_t fim = clock();

    double tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
    printf("Tempo de execucao (dgemm_basico): %.4f segundos\n", tempo);

    libera_matriz(A, n);
    libera_matriz(B, n);
    libera_matriz(C, n);

    return 0;
}
```

2.2. Análise do tempo de execução - DGEMM 3

Tabela 1. Tempos de execução em Python e C

Tamanho (N)	Tempo Médio - Python (s)	DP - Python (s)	Tempo - C (s)
1000	137,60	0,83	9,6631
2000	1178,90	5,01	111,0590
3000	4038,50	19,40	398,5973
4000	9728,20	17,35	965,2479
5000	19138,10	32,94	1903,9901
6000	33199,70	44,58	3307,8563
7000	52879,70	46,33	5269,7992
8000	79084,40	23,50	7882,8155



Comparando por alto (apenas uma execução do código para cada n variando de 1000 a 8000) os resultados do código em C, podemos observar uma diferença clara entre as duas linguagens, sendo Python interpretada e C compilada, de desempenho em relação a um programa que utiliza a mesma implementação de um multiplicador de matriz.

3. Capítulo 2 e 3

3.1. Análise

O primeiro passo para o desenvolvimento do projeto foi compreender o funcionamento da função DGEMM e suas diferentes implementações. Em seguida, migramos o código das duas versões da função para o ambiente Visual Studio Code (VSCode). Criamos também um script gerador de matrizes aleatórias, com o objetivo de automatizar a entrada de dados, uma vez que, para este projeto, seriam utilizadas matrizes com dimensões grandes, inviabilizando a digitação manual.

Posteriormente, desenvolvemos um script para medição de tempo de execução de cada programa. Essa medição era essencial para comparar a eficiência entre as implementações de DGEMM. No entanto, inicialmente não percebemos que o código responsável pela medição de tempo esperava um parâmetro do tipo int, enquanto a segunda implementação de DGEMM utilizava o tipo size_t. Ao invés de duplicar o código de medição, decidimos padronizar o tipo de entrada para size_t, inclusive na primeira versão, por ser um tipo sem sinal (unsigned) e mais apropriado para representar tamanhos de estruturas de dados. Apesar de a dimensão utilizada neste projeto não exceder os limites de int, a adoção de size_t é uma prática mais segura e moderna.

A seguir, tentamos implementar um script em Python para automatizar a compilação e execução dos arquivos C. No entanto, nos deparamos com o erro undefined reference to WinMain@16, o qual indicava que o GCC estava tentando compilar o código como se fosse um programa com interface gráfica, esperando a função WinMain em vez da função padrão main. Tentamos contornar o problema adicionando a flag -mwindows, mas o erro persistiu. Diante disso, optamos por criar o script de inicialização na própria linguagem C. Nesse script, configuramos o programa para aceitar, via linha de comando, o valor de n (dimensão da matriz) e a versão da DGEMM a ser utilizada, tornando o sistema flexível e de fácil uso.

3.2. Corrigindo a primeira implementação

Inicialmente, observamos que a execução repetida da função não estava iniciando um novo processo, o que permitia o uso de cache pelo sistema operacional, distorcendo os resultados das medições de tempo. Para solucionar esse problema, foi necessário garantir que cada execução ocorresse de forma isolada, impedindo o reaproveitamento de dados armazenados em cache.

Como o projeto envolvia duas versões distintas da função (dgemm1 e dgemm2), criamos executáveis independentes para cada uma, assegurando a separação entre as execuções. Além disso, utilizamos um script Python com o módulo subprocess, que permitia iniciar um novo processo a cada execução. Dessa forma, garantimos maior precisão e confiabilidade nas medições de desempenho, eliminando interferências causadas por execuções anteriores.

3.3. Compreendendo as implementações

- DGEMM 1:

  Implementação padrão da multiplicação de matrizes (DGEMM). Para calcular cada elemento da matriz resultado C, soma-se o produto dos elementos da linha de A com os da coluna correspondente de B, formando assim cada linha de C. Por mais que formemos as linhas C, na memória, os endereços consecutivos correspondem aos elementos da coluna da matriz, por isso adota-se o termo coluna-major para caracterizar esse tipo de armazenamento.

  A matriz C é passada como argumento da função por motivos de desempenho e eficiência: ao delegar a alocação de memória ao chamador, evita-se o custo elevado de alocar

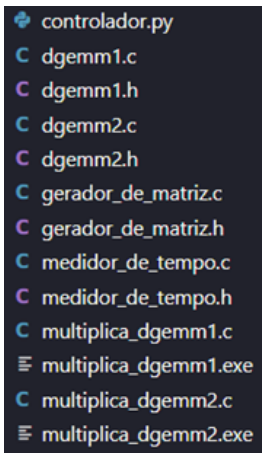


Figura 1. Diretório utilizado na implementação dos DGEMM 1 e 2

(e realocar) memória dentro da função. Essa abordagem segue a convenção da biblioteca BLAS (Basic Linear Algebra Subprograms), onde as matrizes de entrada e saída são sempre fornecidas como argumentos, permitindo maior controle e desempenho em ambientes de alto desempenho e computação científica.

```c
#include <stddef.h>

void dgemm1(size_t n, double* A, double* B, double* C) {
    for (size_t i = 0; i < n; i++)
        for (size_t j = 0; j < n; j++) {
            double sum = 0.0;
            for (size_t k = 0; k < n; k++)
                sum += A[i*n + k] * B[k*n + j];
            C[i*n + j] = sum;
        }
}

```

Primeira versão do DGEMM do livro "Computer Organization and Design"

- DGEMM 2:

  A ideia principal é processar múltiplos elementos simultaneamente, aproveitando o paralelismo de dados para acelerar significativamente a computação. Ao invés de uma multiplicação escalar, é utilizada a multiplicação vetorial.

  Para calcular cada elemento da matriz resultado C, assim como na implementação padrão, soma-se o produto dos elementos da linha de A com os da coluna correspondente de B. Porém, nesta versão, ao invés de calcular um único valor por vez, o código processa quatro elementos da matriz C simultaneamente, ou seja, quatro elementos de uma mesma coluna de C, correspondentes a quatro linhas consecutivas. Essa implementação percorre a matriz C no sentido horizontal (das linhas), assim como na primeira implementação, mas ao invés de compor apenas um elemento de cada linha por vez, vai compondo de 4 em 4.

  A função utiliza registradores do tipo__m256d, que armazenam quatro valores double (64 bits cada) em um único registrador de 256 bits. A instrução _mm256_load_pd carrega quatro doubles consecutivos da memória para o registrador, e a _mm256_store_pd grava os resultados de volta. Já a _mm256_broadcast_sd replica um único valor (um escalar double) para os quatro slots do registrador, permitindo multiplicar esse valor com um vetor completo (broadcasting).

  Esse paralelismo permite que, em vez de calcular C[i][j] linha por linha, o código calcule C[i][j], C[i+1][j], C[i+2][j], e C[i+3][j] de uma vez só, otimizando o uso da CPU e reduzindo o número de ciclos por operação.

Um cuidado que tivemos com essa implementação foi verificar o AVX do processador que seria utilizado para o projeto, já que a versão desse componente definiria o tamanho dos registradores. No caso da versão do livro “Computer Organization And Design – RiscV Edition”, ele usa um tipo _m256d, que significa que, segundo esse modelo do DGEMM, o processador estaria utilizando um registrador de 256 bits que armazena doubles, no caso, 4 deles, uma vez que cada double apresenta 64 bits. Essa é a razão pela qual essa versão do dgemm multiplica de 4 em 4 elementos. Essa dinâmica ocorre por causa do tipo de registrador SIMD AVX usado: __m256d, que comporta 4 doubles (64 bits cada). Então o código é escrito para tirar proveito da paralelização vetorial, fazendo operações em 4 elementos de uma vez.

```
#include <stddef.h>
# include <x86intrin.h>
void dgemm2 (size_t n, double* A, double* B, double* C)
{
    for ( size_t i = 0; i < n; i+=4 )
        for ( size_t j = 0; j < n; j++ ) {
            __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
            for( size_t 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 */
    }
}

```

Segunda versão do DGEMM do livro "Computer Organization and Design"

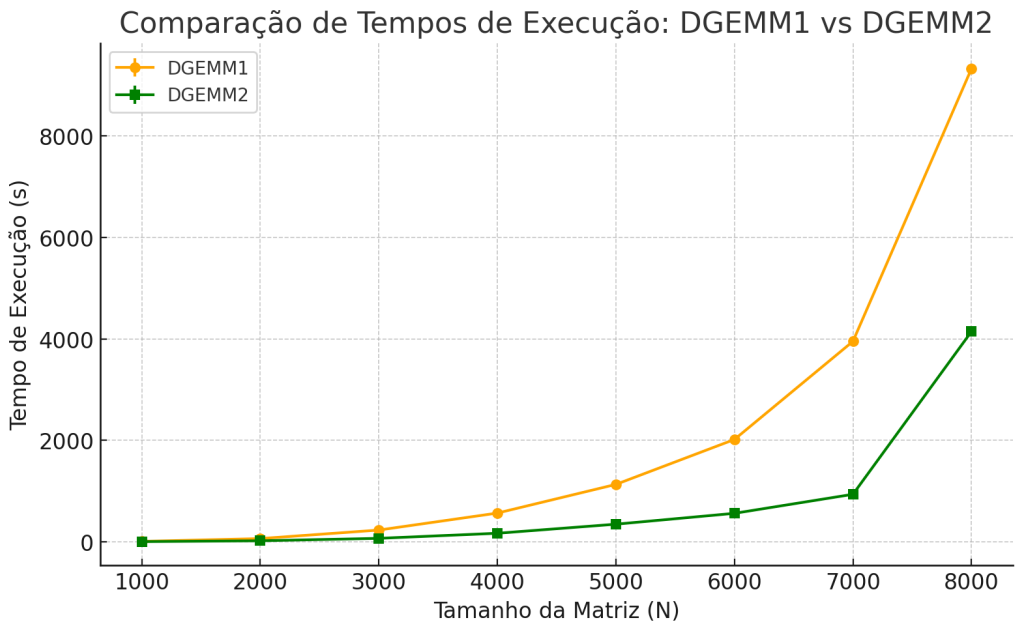
3.4. Análise do tempo de execução - DGEMM 1 e 2

Com o objetivo de observar a diferença de performance entre as versões do DGEMM, decidimos compilar, primeiramente, os programas sem nenhuma flag de otimização, e executar o binário alternando o n (tamanho das linhas e colunas das matrizes a serem multiplicadas) de 1000 até 8000. Para cada n, executamos 10 vezes os programas, com o intuito de obter uma média entre os valores. Esses foram os tempos levados pelas implementações para executar o programa:

Tabela 2. Tempos de execução de DGEMM1 e DGEMM2

N	DGEMM1 (s)	DP DGEMM1 (s)	DGEMM2 (s)	DP DGEMM2 (s)
1000	6,9	0,275	2,1	0,318
2000	62,0	1,147	18,5	0,611
3000	229,9	2,87	66,7	1,80
4000	564,7	3,89	166,7	2,01
5000	1130,0	5,54	345,6	4,88
6000	2016,1	10,26	559,8	8,56
7000	3951,0	13,27	935,2	8,56
8000	9328,6	38,2	4145,0	26,37

A partir desses resultados, comprovamos o comportamento já esperado da segunda implementação perante a primeira, sendo a DGEMM 2 mais eficiente no quesito performance, graças ao conceito de paralelização vetorial incorporado nesse modelo. Observamos, também, que essa discrepância de tempo de execução se intensifica conforme aumentamos N, o que nos diz que, quanto maior for a tarefa, mais importante é desenvolver um programa que valorize o desempenho.



4. Capítulo 4, 5 e 6

4.1. Iniciando a análise - DGEMM 4, 5 e 6

4.1.1. Estrutura der arquivos no diretório

Abaixo está a explicação de cada arquivo necessário para a execução das versões DGEMM 4, 5 e 6.

- dgemm4.c / dgemm5.c / dgemm6.c: Contêm as implementações das funções de multiplicação de matrizes. A versão 4 aplica otimizações com acesso por blocos e organização da ordem de loops. A versão 5 adiciona uso de SIMD com AVX-512. A versão 6 incorpora paralelismo usando OpenMP.

- multiplica_dgemm4.c / multiplica_dgemm5.c / multiplica_dgemm6.c: Arquivos com a função main(). Eles recebem o valor de n via linha de comando, alocam dinamicamente as matrizes A, B e C, preenchem-nas com valores fixos e invocam a função dgemm correspondente.

- multiplica_dgemm4.exe / multiplica_dgemm5.exe / multiplica_dgemm6.exe: Executáveis compilados a partir dos arquivos .c anteriores. São utilizados para medir o desempenho das implementações.

- executa_dgemm.py: Script auxiliar em Python que recebe um executável e um valor de n, mede o tempo de execução usando a biblioteca ‘time‘.

- controlador_dgemm456.py: Script principal em Python que recebe o valor de n via ‘argv‘, executa os três binários (multiplica_dgemm4, 5 e 6), e imprime os tempos individuais de execução no terminal.

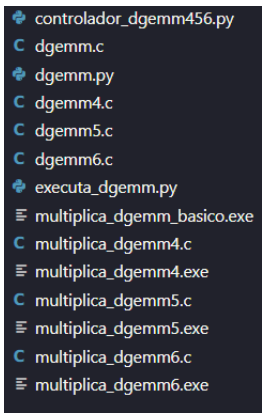


Figura 2. Diretório utilizado na implementação dos DGEMM 4, 5 e 6

4.1.2. Processo de compilação do DGEMM 4 e 5

As versões DGEMM 4 e 5 não utilizam paralelismo explícito, o que permite que sejam compiladas diretamente com o compilador GCC padrão. Ambas utilizam otimizações como blocagem de cache e reordenação dos loops, sendo que a versão 5 também explora SIMD com AVX-512. Os comandos de compilação utilizados foram:

4.1.3. Processo de compilação do DGEMM 6 (OpenMP)

Para a versão DGEMM 6, foi incluído paralelismo por meio da biblioteca OpenMP, utilizando diretivas como ‘#pragma omp parallel for‘. No entanto, ao tentar compilar com o MinGW tradicional, foi gerado o seguinte erro:’cannot find -lpthread’

Esse erro indica ausência da biblioteca POSIX Threads no ambiente, essencial para a execução paralela. A solução foi utilizar o ambiente MSYS2, que fornece um terminal de desenvolvimento completo com suporte a OpenMP por padrão. Os passos seguidos foram:

- Instalação do MSYS2 a partir de https://www.msys2.org.;
- Execução do terminal correto: ’MSYS2 MinGW 64-bit’;
- Atualização dos pacotes com ‘pacman -Syu‘ (reinício) e ‘pacman -Su‘;
- Instalação do GCC com OpenMP: ‘pacman -S mingw-w64-x86_64-gcc‘;
- Verificação de suporte com: ‘echo | gcc -fopenmp -dM -E - | grep _OPENMP‘;

A compilação foi feita com sucesso usando o comando:

’gcc -O2 -mavx512f -fopenmp -o multiplica_dgemm6.exe multiplica_dgemm6.c dgemm6.c’

O uso do MSYS2 garantiu que o paralelismo funcionasse corretamente, aproveitando múltiplos núcleos do processador para acelerar a multiplicação.

4.2. Compreensão das implementações

4.2.1. DGEMM 4 - SIMD com unroll (sem cache-blocking)

Esta versão utiliza registradores AVX-512 para realizar operações vetoriais com 4 elementos do tipo double ao mesmo tempo. A técnica de unrolling reduz o overhead de controle de loop. Não há uso de cache blocking, portanto, embora mais rápida que a versão ingênua, ela ainda não é ideal para matrizes muito grandes.

Cache Blocking: Cache blocking (também conhecido como loop blocking ou tiling) é uma técnica de otimização de desempenho utilizada principalmente em computação científica e progra-

mação de baixo nível, cujo objetivo é melhorar o uso da memória cache da CPU durante o acesso a dados em estruturas como matrizes.

A memória RAM é lenta comparada ao processador. Por isso, os processadores usam memória cache, que é bem mais rápida, mas pequena. Quando acessamos dados grandes (como uma matriz), se o código não for bem organizado, ele pode estar constantemente trazendo e descartando dados do cache — isso é ineficiente.

É uma forma de reorganizar os loops de um algoritmo (geralmente com matrizes) para trabalhar com blocos menores de dados que cabem no cache. Isso reduz cache misses (acessos à RAM quando o dado não está no cache) e aumenta o desempenho.

```
#include <x86intrin.h>
#define UNROLL (4)

void dgemm (int n, double* A, double* B, double* C) {
    for (int i = 0; i < n; i += UNROLL * 8)
        for (int j = 0; j < n; ++j) {
            __m512d c[UNROLL];
            for (int r = 0; r < UNROLL; r++)
                c[r] = _mm512_load_pd(C + i + r * 8 + j * n);

            for (int k = 0; k < n; ++k) {
                __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B + j * n + k));
                for (int r = 0; r < UNROLL; r++)
                    c[r] = _mm512_fmadd_pd(_mm512_load_pd(A + n * k + r * 8 + i)
    , bb, c[r]);
            }

            for (int r = 0; r < UNROLL; r++)
                _mm512_store_pd(C + i + r * 8 + j * n, c[r]);
        }
}
```

4.2.2. DGEMM 5 - Cache blocking com SIMD

Essa versão melhora o desempenho utilizando cache blocking, que divide as matrizes em blocos menores, otimizando o uso da cache. Além disso, utiliza instruções SIMD para processar múltiplos elementos por vez, resultando em maior desempenho, especialmente em matrizes grandes.

```
#include <x86intrin.h>
#define UNROLL (4)
#define BLOCKSIZE 32

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) {
            __m512d c[UNROLL];
            for (int r = 0; r < UNROLL; r++)
                c[r] = _mm512_load_pd(C + i + r * 8 + j * n);

            for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B + j * n + k));
                for (int r = 0; r < UNROLL; r++)
```

```
                    c[r] = _mm512_fmadd_pd(_mm512_load_pd(A + n * k + r * 8 + i)
    , bb, c[r]);
            }

            for (int r = 0; r < UNROLL; r++)
                _mm512_store_pd(C + i + r * 8 + 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);
}
```

4.2.3. DGEMM 6 - Cache blocking com SIMD + OpenMP

Essa versão combina as vantagens do cache blocking e SIMD com a paralelização do OpenMP, permitindo que blocos diferentes sejam processados por múltiplas threads simultaneamente. É a versão mais rápida, aproveitando totalmente a capacidade da CPU moderna.

```
#include <x86intrin.h>
#define UNROLL (4)
#define BLOCKSIZE 32

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) {
            __m512d c[UNROLL];
            for (int r = 0; r < UNROLL; r++)
                c[r] = _mm512_load_pd(C + i + r * 8 + j * n);

            for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B + j * n + k));
                for (int r = 0; r < UNROLL; r++)
                    c[r] = _mm512_fmadd_pd(_mm512_load_pd(A + n * k + r * 8 + i)
    , bb, c[r]);
            }

            for (int r = 0; r < UNROLL; r++)
                _mm512_store_pd(C + i + r * 8 + 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);
}
```

4.3. Análise do tempo de execução - DGEMM 4, 5 e 6

Assim como foi feito anteriormente com as outras implementações do DGEMM, rodamos cada código 10 vezes para conseguirmos o efeito comparativo entre as diferentes abordagens de multiplicação de matrizes.

Tabela 3. Tempos de execução de DGEMM4, DGEMM5 e DGEMM6 para diferentes tamanhos de matriz

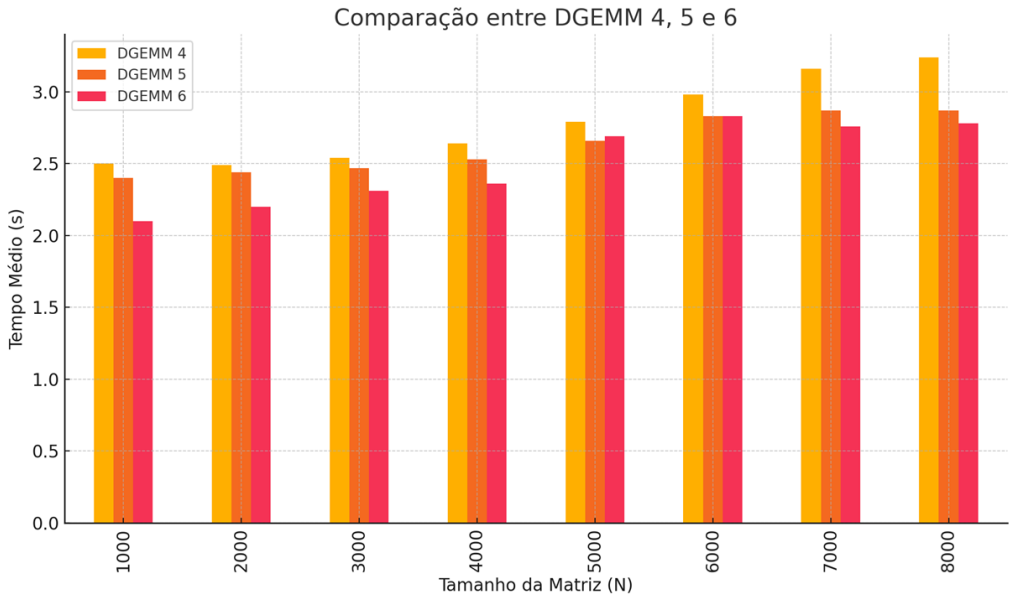
N	DGEMM 4	σ	DGEMM 5	σ	DGEMM 6	σ
1000	2,50	0,29	2,40	0,14	2,10	0,50
2000	2,49	0,18	2,44	0,17	2,20	0,22
3000	2,54	0,28	2,47	0,19	2,31	0,20
4000	2,64	0,34	2,53	0,28	2,36	0,15
5000	2,79	0,29	2,66	0,21	2,69	0,24
6000	2,98	0,31	2,83	0,20	2,83	0,14
7000	3,16	0,42	2,87	0,24	2,76	0,19
8000	3,24	0,36	2,87	0,19	2,78	0,17

4.4. Comparação entre resultados do DGEMM 4, 5, e 6

As versões DGEMM 4, 5 e 6 foram projetadas com técnicas de otimização crescentes:

- A DGEMM 4 utiliza vetorização com AVX-512 e unrolling básico;
- A DGEMM 5 adiciona cache blocking à vetorização, otimizando o uso de memória;
- A DGEMM 6 inclui ainda paralelismo com OpenMP, explorando múltiplos núcleos.

Apesar das diferenças, os tempos médios de execução entre essas três versões são bastante próximos, todos abaixo de 3.3 segundos mesmo para matrizes de dimensão 8000. Ainda assim, a DGEMM 6 se mostrou levemente mais eficiente na maioria dos casos, confirmando que a combinação de paralelismo e vetorização traz benefícios adicionais, mesmo que modestos para entradas desse tamanho.



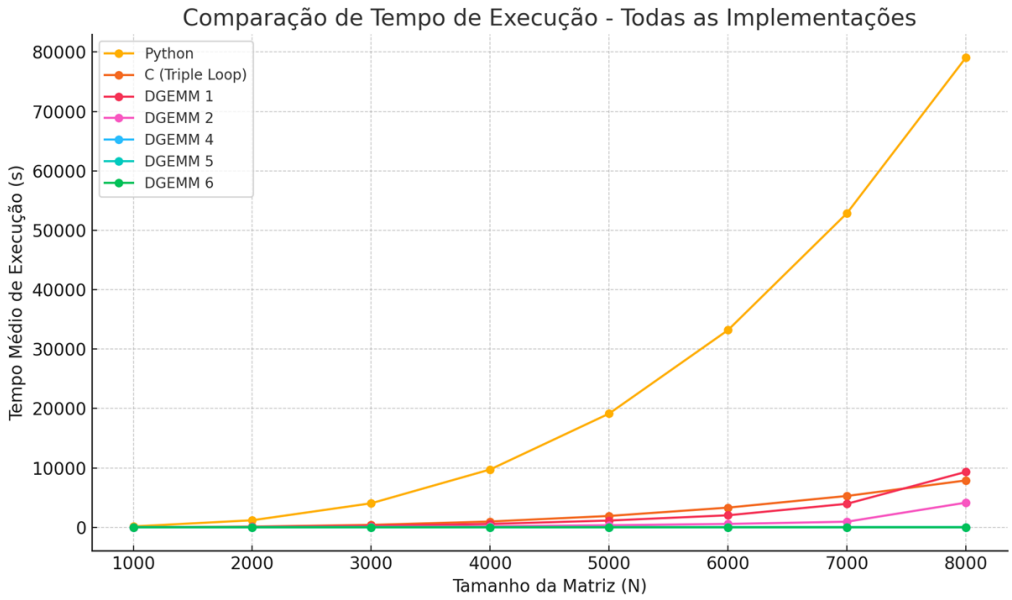
4.5. Comparação entre todas as DGEMM

Ao comparar todas as implementações — desde a versão Python até a DGEMM 6 em C — observamos uma curva de otimização progressiva. A DGEMM em Python representa a abordagem menos eficiente, adequada apenas para propósitos didáticos. O salto para C já representa uma

melhora massiva, mesmo sem otimizações explícitas. As versões otimizadas (DGEMM 2 a 6) demonstram como técnicas como vetorização, cache blocking e paralelismo afetam significativamente o desempenho. Em particular:

- A transição da DGEMM 1 para a DGEMM 2 traz um ganho médio de mais de 2 vezes graças ao uso de AVX.
- Da DGEMM 2 para as versões 4–6, o desempenho estabiliza e os ganhos se tornam incrementais, mas importantes para cargas maiores ou aplicações reais.

O estudo deixa claro que, em computação científica, o refinamento da implementação — mesmo sem alterar o algoritmo base — pode ter impacto decisivo na viabilidade de execução de grandes volumes de dados.

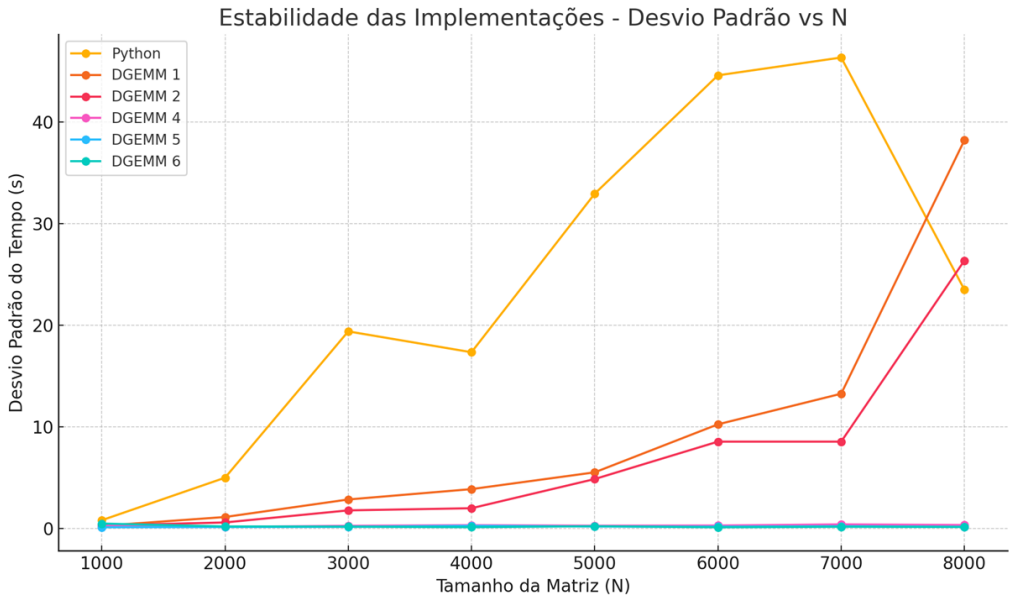


4.6. Considerações sobre estabilidade e confiabilidade

O gráfico evidencia a variação nos tempos de execução (em segundos) para diferentes implementações da função DGEMM conforme o tamanho da matriz cresce. Um desvio padrão menor indica que a execução é mais estável e previsível, o que é desejável em aplicações críticas.

Algumas observações sobre os dados de estabilidade e confiabilidade são:

- Python apresentou uma instabilidade moderada, com desvios crescendo significativamente até matrizes de dimensão 6000 e um leve recuo em 8000. Isso reflete não só a ineficiência da linguagem para tarefas intensivas, mas também a sua sensibilidade ao ambiente de execução (como garbage collection e interpretador);
- **DGEMM 1 e DGEMM 2** mostram aumento gradual do desvio padrão à medida que o tamanho da matriz cresce, o que é esperado em algoritmos que não utilizam paralelismo nem otimizações profundas de cache. Ainda assim, o DGEMM 2 se mostra consistentemente mais estável que o DGEMM 1;
- **DGEMM 4, 5 e 6** apresentam desvios padrão muito baixos e relativamente constantes, mesmo com o aumento do tamanho da matriz. Isso indica altíssima estabilidade de desempenho e evidencia a eficiência de técnicas como **SIMD, cache blocking e paralelismo com OpenMP**.;
- **DGEMM 6**, apesar de ser a versão mais complexa, é também uma das mais consistentes em termos de estabilidade, demonstrando que o uso de múltiplas threads não compromete a previsibilidade da execução.



4.7. Conclusão

As versões otimizadas (4, 5 e 6) não apenas são mais rápidas, como também muito mais confiáveis. Em cenários onde estabilidade e repetibilidade do tempo de execução são tão importantes quanto a velocidade (como benchmarks ou sistemas de tempo real), essas versões se destacam amplamente.

5. Referências bibliográficas

- Livro didático - "Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 2nd. Patterson, D. A. and Hennessy, J. L.. Morgan Kaufman, 2021, ISBN : 978-0-12-820331-6"
- Aulas do prof. Diego Leonel Cadette Dutra
- Site da disciplina - `https://www.compasso.ufrj.br/disciplinas/eel580`
- Site utilizado para a instalação do MSYS2 com suporte a OpenMP, usado na implementação do DGEMM 6 - `https://www.msys2.org/`

Apêndice: Conteúdo Completo dos Arquivos

A. GitHub do projeto

`https://github.com/pedrotuttman/DGEMM.git`

B. Diretório 1

controlador.py

```python
import subprocess
import os
import time

def executar_versao(versao, n):
    # Define o caminho completo para os execut veis
    if versao == 'dgemm1':
        executavel = os.path.join(os.getcwd(), 'multiplica_dgemm1.exe')
    elif versao == 'dgemm2':
        executavel = os.path.join(os.getcwd(), 'multiplica_dgemm2.exe')
```

```
    else:
        print(f"Versao {versao} nao reconhecida.")
        return

    # Exibe a mensagem de multiplicacao para a versao
    print(f"Multiplicando com {versao}, n = {n}...")

    # Registra o tempo de inicio
    start_time = time.time()

    try:
        # Executa o subprocesso com o caminho completo e passando apenas o argumento n
        subprocess.run([executavel, str(n)], check=True)
    except subprocess.CalledProcessError as e:
        print(f"Erro ao executar o programa: {e}")
        return

    # Calcula o tempo de execucao
    end_time = time.time()
    execution_time = end_time - start_time

    # Exibe o tempo de execucao
    print(f"Tempo de execucao: {execution_time:.6f} segundos")

def main():
    if len(os.sys.argv) != 2:
        print("Uso: python controlador.py <tamanho das matrizes (n)>")
        return

    n = int(os.sys.argv[1])  # Tamanho da matriz (n)

    # Executa a versao dgemm1
    executar_versao('dgemm1', n)

    # Executa a versao dgemm2
    executar_versao('dgemm2', n)

if __name__ == "__main__":
    main()
```

Listing 1. Conteúdo de `controlador.py`

dgemm1.h

```
#ifndef DGEMM1_H
#define DGEMM1_H

void dgemm1(size_t n, double* A, double* B, double* C);

#endif
```

Listing 2. Conteúdo de `dgemm1.h`

dgemm2.h

```
#ifndef DGEMM2_H
#define DGEMM2_H

#include <stddef.h>  // para size_t

// Funcao de multiplicacao de matrizes otimizada com AVX (DGEMM versao 2)
```

```c
void dgemm2(size_t n, double* A, double* B, double* C);

#endif
```

Listing 3. Conteúdo de `dgemm2.h`

gerador_de_matriz.c

```c
#include <stdlib.h>
#include <stddef.h>
#include "gerador_de_matriz.h"

double* gerar_matriz(size_t n) {
    double* matriz = (double*) malloc(n * n * sizeof(double));
    for (size_t i = 0; i < n * n; i++) {
        matriz[i] = (double)rand() / RAND_MAX;
    }
    return matriz;
}
```

Listing 4. Conteúdo de `gerador_de_matriz.c`

gerador_de_matriz.h

```c
#ifndef GERADOR_DE_MATRIZ_H
#define GERADOR_DE_MATRIZ_H

#include <stddef.h>

double* gerar_matriz(size_t n);

#endif
```

Listing 5. Conteúdo de `gerador_de_matriz.h`

medidor_de_tempo.c

```c
#include <stdio.h>
#include <time.h>
#include "medidor_de_tempo.h"

double medir_tempo(void (*funcao)(size_t, double*, double*, double*), size_t n,
    double* A, double* B, double* C) {
    clock_t start_time = clock(); // Hora de in cio

    funcao(n, A, B, C); // Chama a fun   o

    clock_t end_time = clock(); // Hora de t rmino

    return (double)(end_time - start_time) / CLOCKS_PER_SEC;
}
```

Listing 6. Conteúdo de `medidor_de_tempo.c`

medidor_de_tempo.h

```c
#ifndef MEDIDOR_DE_TEMPO_H
#define MEDIDOR_DE_TEMPO_H

#include <stddef.h>

double medir_tempo(void (*funcao)(size_t, double*, double*, double*), size_t n,
    double* A, double* B, double* C);
```

```

#endif
```

Listing 7. Conteúdo de `medidor_de_tempo.h`

C. Diretório 2

controlador_dgemm456.py

```python
import sys
from executa_dgemm import executar_dgemm

if len(sys.argv) < 2:
    print("Uso: python controlador_dgemm456.py <tamanho_da_matriz>")
    sys.exit(1)

try:
    n = int(sys.argv[1])
except ValueError:
    print("Erro: o tamanho da matriz deve ser um n mero inteiro.")
    sys.exit(1)

executaveis = [
    "multiplica_dgemm4.exe",
    "multiplica_dgemm5.exe",
    "multiplica_dgemm6.exe"
]

for exe in executaveis:
    executar_dgemm(exe, n)
```

Listing 8. Conteúdo de `controlador_dgemm456.py`

executa_dgemm.py

```python
import subprocess
import time

def executar_dgemm(executavel, n):
    inicio = time.time()
    subprocess.run([executavel, str(n)])
    fim = time.time()
    print(f"Tempo de execu  o ({executavel}, n={n}): {fim - inicio:.4f}
    segundos")
```

Listing 9. Conteúdo de `executa_dgemm.py`

multiplica_dgemm4.c

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

void dgemm(int n, double* A, double* B, double* C);  // Prot tipo

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Uso: %%s <tamanho_da_matriz>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

```

```
    double* A = (double*) malloc(n * n * sizeof(double));
    double* B = (double*) malloc(n * n * sizeof(double));
    double* C = (double*) malloc(n * n * sizeof(double));

    if (!A || !B || !C) {
        printf("Erro ao alocar mem ria\n");
        return 1;
    }

    for (int i = 0; i < n * n; i++) {
        A[i] = 1.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    dgemm(n, A, B, C);

    free(A);
    free(B);
    free(C);

    return 0;
}
```

Listing 10. Conteúdo de `multiplica_dgemm4.c`

multiplica_dgemm5.c

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

void dgemm(int n, double* A, double* B, double* C);  // Prot tipo

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Uso: %%s <tamanho_da_matriz>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

    double* A = (double*) malloc(n * n * sizeof(double));
    double* B = (double*) malloc(n * n * sizeof(double));
    double* C = (double*) malloc(n * n * sizeof(double));

    if (!A || !B || !C) {
        printf("Erro ao alocar mem ria\n");
        return 1;
    }

    for (int i = 0; i < n * n; i++) {
        A[i] = 1.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    dgemm(n, A, B, C);

    free(A);
    free(B);
    free(C);
```

```
34
35      return 0;
36  }
```

Listing 11. Conteúdo de `multiplica_dgemm5.c`

multiplica_dgemm6.c

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

void dgemm(int n, double* A, double* B, double* C);  // Prototipo

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Uso: %s <tamanho_da_matriz>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);

    double* A = (double*) malloc(n * n * sizeof(double));
    double* B = (double*) malloc(n * n * sizeof(double));
    double* C = (double*) malloc(n * n * sizeof(double));

    if (!A || !B || !C) {
        printf("Erro ao alocar memoria\n");
        return 1;
    }

    for (int i = 0; i < n * n; i++) {
        A[i] = 1.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    dgemm(n, A, B, C);

    free(A);
    free(B);
    free(C);

    return 0;
}
```

Listing 12. Conteúdo de `multiplica_dgemm6.c`