Universidade Federal do Rio de Janeiro
Departamento de Engenharia Eletrônica
Escola Politécnica
EEL580 - Arquitetura de Computadores

Relatório da Atividade Prática
DGEMM

Grupo
Bernardo Brandão Pozzato Carvalho Costa - 123289593
Gabriel Schmitz Corrêa Rizawinsk - 123225573
Guilherme En Shih Hu - 123224674
Luís Miguel de Freitas Costa - 123510368
Maria Victoria França Silva Ramos - 123311073

Rio de Janeiro, 05 de junho de 2024

Conteúdo

1 Introdução

A computação tem as matrizes como parte fundamental do seu funcionamento, já que elas são responsáveis desde a estruturação de dados e representação dos mesmos até a modelagem e simulação de problemas mais complexos.

Devido à grande relevância das matrizes na composição da computação, diversas tentativas de aprimorar a eficiência das matrizes e otimizar seu uso foram feitas. Um desses exemplos é o DGEMM, sigla para "Double-precision General Matrix Multiply- uma função da biblioteca BLAS (Basic Linear Algebra Subprograms)-, responsável por realizar multiplicações de matrizes de ponto flutuantes de dupla precisão, independente de seu formato, de forma muito mais eficiente do que a simples multiplicação criando matrizes de zeros, atribuindo os valores dados e colocando os resultados da multiplicação em outra matriz iniciada com zeros.

Além disso, devido à grande importância de sempre ampliar a eficiência da resolução de problemas computacionais, mostra-se necessária, também, uma adaptação dos códigos em busca de se adequar aos componentes de hardware visando obter uma execução com menor tempo. Quanto melhor a adaptação, aproximando-se ao máximo da microarquitetura disponível, maior a eficiência da implementação.

A partir da necessidade de sempre melhorar o desempenho das atividades computacionais, o livro referência da disciplina de Arquitetura de Computadores, "COMPUTER ORGANIZATION AND DESIGN RISC-V EDITION", apresenta seis seções denominadas "Going Faster", uma por capítulo, que mostram possíveis melhorias focadas na otimização do uso de hardware em um código de multiplicações de matrizes em busca de maximizar sua performance, a fim de reduzir os tempos de execução, utilizando a função DGEMM de base.

Utilizando essas seções do livro, esse trabalho implementa as sugestões do livro-texto, anotando os tempos de execução e justificando as causas das melhorias no desempenho. Além disso, foram realizados testes em outras linguagens de programação para comparar seu desempenho em relação às linguagens propostas pelo livro. Com os dados coletados, é possível analisar as linguagens utilizadas e as otimizações implementadas em cada uma delas, a fim de concluir sobre a forma mais eficiente de realizar as multiplicações de matrizes.

2 Seções "Going Faster"do livro

2.1 Capítulo 1

Começamos os testes a partir de uma multiplicação não otimizada em Python, que é uma linguagem rápida para definir a forma de multiplicar as matrizes, mas que demora muito para gerar o resultado da multiplicação. Ou seja, esse é o passo inicial do trabalho, com a forma menos otimizada e mais demorada das opções do livro para se obter uma multiplicação de duas matrizes com número de linhas e colunas sendo múltiplos de 2.

A multiplicação matricial feita a partir do Python será a base de comparação. Por ser uma linguagem mais distante do hardware, ela demora muito mais que as demais para realizar a multiplicação. A cada capítulo é sugerida uma nova forma de aprimorar o código e incrementar a performance em relação ao Python.

2.2 Capítulo 2

Para a primeira melhoria no desempenho do código, a linguagem será trocada de Python para C. Para isso fizemos o DGEMM e usamos matrizes de uma dimensão e endereços aritméticos. Essa versão de código não tem nenhuma otimização em C e já apresenta melhora significativa, se comparada ao Python.

A melhora significativa na mudança de linguagem se deve à substituição do interpretador (do Python) pelo Compilador (do C) e ao fato de que as declarações de tipo no C permitem a construção de um código mais eficiente do que em Python.

2.3 Capítulo 3

Usando as ideias de paralelismo no "data-level"apresentadas no capítulo, implementa-se nessa seção a primeira otimização no código: o uso de "subword parallelism". Essa técnica consiste em usar parte do registrador de forma separada e paralela, dividindo as tarefas e executando simultaneamente as diversas partes (subwords).

No caso especificamente do código de multiplicações de matrizes, o compilador vai usar 8 "double-precision floating-point values"por vez, fazendo, também, 8 produtos e 8 somas ao mesmo tempo. Há, além disso, uma troca do scalar double(sd) para o parallel double (pd). Como são executadas 8 vezes mais operações no mesmo intervalo de tempo, a melhora de desempenho chega próxima de 8 vezes.

2.4 Capítulo 4

Após a mudança do paralelismo, implementa-se otimizações a partir dos conceitos do quarto capítulo, que aborda, sobretudo, o pipeline. Para isso, a nova otimização será o "instruction-level parallelism", que realiza a divisão do loop em 4. Isso gera um aumento de performance, que, se combinado à melhoria do capítulo anterior, ultrapassa a performance do Python em mais de 8 vezes.

A melhoria na performance se deve ao fato de que ao segmentar o loop, é possível usar a emissão múltipla de instruções, fazendo mais trabalho simultaneamente. Ademais, o hardware pode realizar uma execução fora de ordem, fato que também auxilia o incremento de performance pelo fato das instruções desordenadas serem adequadas de forma a executar o código mais rapidamente.

2.5 Capítulo 5

O capítulo apresenta uma análise das memórias dos computadores e a sua importância na arquitetura, seohretudo devido a sua hierarquia, que deve ser respeitada para melhorar a performance. A partir desses conceitos, implementa-se a penúltima otimização do código de multiplicação matricial, o cache blocking.

O cache blocking aumenta a eficiência do código para qualquer entrada, mas possui uma mudança mais significativa conforme as entradas vão ficando cada vez maiores (a técnica tem mais eficácia em matrizes de 4k do que de 2k, proporcionalmente). Tal técnica consiste em minimizar o acesso à memória principal (RAM), que é mais lenta, e maximizar o acesso à memória cache, já que ela tem maior velocidade. Para isso, a matriz fica dividida em blocos menores para que caibam na memória cache de modo que as operações aritméticas possam ser realizadas de forma mais eficaz devido ao acesso rápido dessa memória.

2.6 Capítulo 6

Por fim, no capítulo 6, o livro trata dos multiprocessadores, computadores com microprocessadores multicore que possibilitam a execução simultânea de um ou mais programas dentro desses núcleos no chamado do processamento paralelo. Dessa forma, é apresentado e discutido o funcionamento dos multithreading no hardware, processamento gráfico, multiprocessadores de memória, topologias de network e de repasse de mensagem, mostrando como ocorre sua atividade e desempenhho desde o cliente até a nuvem (cloud).

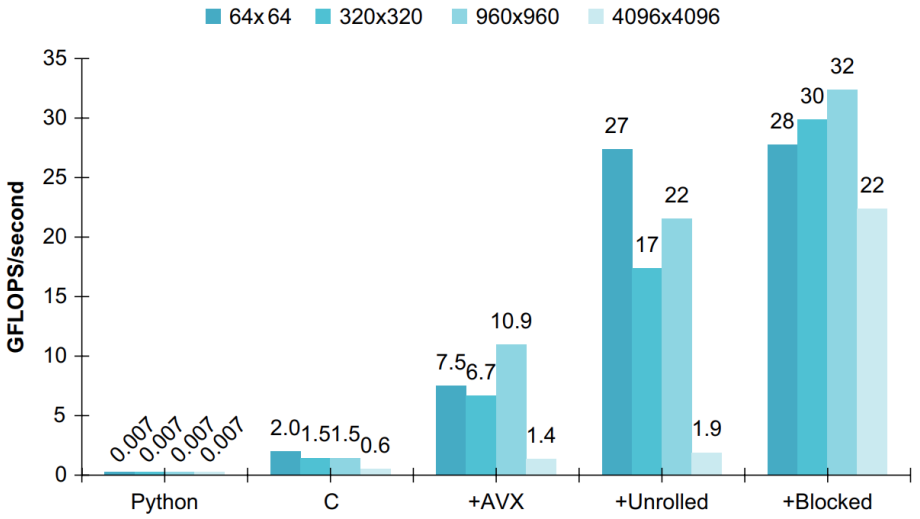

FIGURE 5.49 Performance of multiple versions of DGEMM as change matrix size measured in billion floating point operations per second (GFLOPS/second). The fully optimized code is 14–32 times faster than the C version in Chapter 2. Python runs at 0.007 GFLOPS/second for all matrix sizes. The Intel i7 hardware speculates by prefetching from the L3 to L1 and L2 caches, which is why the benefits of blocking are not as high as on some microprocessors.

Figura 1: Imagem do livro, página 488, que contém a comparação de performance da execução da multiplicação de matrizes sob diversas formas de otimização.

Com isso, é proposta e abordada a última melhoria para otimizar o DGEMM, baseada no thread-level parallelism. Tal aprimoramento é feito por meio do uso dos múltiplos processadores e dos "multiple threads"do computador durante a operação da multiplicação de matrizes, de modo que o processo a ser executado possui seu trabalho dividido pelos núcleos do computador.

Embora essa técnica seja extremamente eficiente para grandes entradas, e o livro discute com bastante ênfase como ocorre uma grande otimização do tempo de execução da função de multiplicação de matrizes conforme se aumenta a quantidade de núcleos e threads, o uso do multiprocessamento traz prejuízos em entradas menores, aumentando o tempo de execução. Assim, verifica-se que é nessa etapa que ocorre o maior nível de otimização, e esse trabalho mostrará experimentalmente a diferença de tempo que o uso de todas essas camadas de otimizações pode contribuir para o aumento de velocidade na execução da multiplicação de matrizes, semelhante ao que o livro faz na Figura 1.

3 Códigos utilizados e suas adaptações em relação ao livro

Segue aqui o link do site "github.com"que contém todos os códigos criados e utilizados na execução da multiplicação de matrizes:

https://github.com/GuilhermeHu/DGEMM/tree/main/Principal.

A seguir, será explicado como ocorreu a implementação geral de cada código e das funções criadas.

3.1 Going Faster 1

O Going Faster 1 utiliza a função DGEMM escrita na linguagem Python, fazendo o comum algoritmo que se conhece para realização de multiplicações de matrizes a partir de 3 loops de *for*. Em nosso código, utilizamos tal função do DGEMM e foram adicionadas também uma função main que recebe como input o tamanho da matriz quadrada que se objetiva calcular, um randomizador de elementos que constrói as matrizes A e B que serão usados nos cálculos, e o trecho de código responsável pela medição de tempo. Note que o randomizador usa a biblioteca *random* para gerar elementos aleatórios em formato float, e a medição de tempo foi feita por meio da biblioteca *time*, a partir de seu método *perf_ counter*, que realiza medições de tempo bastante precisos.

Tendo o código principal feito, foi criado outro código em Python, o benchmarking, que, basicamente, apresenta todos os valores de teste que deverão ser recebidos pelo arquivo main para realização dos testes de tempo. Com isso, foi possível fazer a medição de todos os valores sem precisar ficar colocando manualmente o input toda vez que o arquivo terminava de executar uma operaçõ de multiplicação matricial.

```
def DGEMM(A, B, C):
  for i in range(n):
    for j in range(n):
      for k in range(n):
        C[i][j] += A[i][k] * B[k][j]
```

Figura 2: Função "DGEMM", que realiza a multiplicação matrizes, na linguagem Python.

```python
def main():

  # Geração de matrizes aleatórias
  A = [[random.random() for _ in range(n)] for _ in range(n)]
  B = [[random.random() for _ in range(n)] for _ in range(n)]
  C = [[0 for _ in range(n)] for _ in range(n)]

  time_start, time_start_cpu = 0, 0
  time_start, time_start_cpu = perf_counter(), time.process_time()
  DGEMM(A,B,C)
  time_end, time_end_cpu = perf_counter(), time.process_time()

  print("Total time = %fms" % ((time_end - time_start)*1000))
  print("Total time = %fms" % ((time_end_cpu - time_start_cpu)*1000))
  # print("Total time = %f s" % (time_end - time_start))
  # print("A = ", A)
  # print("B = ", B)
  # print("C = ", C)

main()
```

Figura 3: Função "*Main*"do código em Python, que contém o construtor das matrizes e a medição do tempo de execução da função DGEMM.

3.2 Going Faster 2

A partir do Going Faster 2, o livro passa a escrever a função DGEMM na linguagem C, que é muito mais rápida que o Python e apresenta muitos recursos para otimização do código na perspectiva tanto do hardware, quanto do compilador.

Semelhante ao que foi realizado no Python, a função DGEMM desse Going Faster era composta basicamente de 3 loops de *for* com a realização de suas operações de soma entre cada elemento de coluna e linha das matrizes operandos. Sendo assim, a necessidade propriamente dita de adaptação desse código era de criar uma função main que recebesse como input o tamanho da matriz que se objetiva realizar os cálculos, criasse tais matrizes com números float e alocasse espaço na memória para elas e, por fim, que executasse tal função fazendo simultaneamente a medição de seu tempo de execução.

```
void dgemm (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 * N) + j] += A[(i * N) + k] * B[(k * N) + j];
    }
}
```

Figura 4: Função DGEMM, que opera a multiplicação de matrizes, em linguagem C. Inicialmente, não havendo otimizações de hardware, o DGEMM é basicamente um conjunto de *for* que acessa cada elemento da matriz resultado e calcula seu valor a partir das matrizes operandos.

```
void printMatrix(double* matrix, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j) {
            printf("%8.2lf ", matrix[(i * N) + j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

Figura 5: Função " *printMatrix*", usada para verificação da corretude dos cálculos da multiplicação.

```
void make_rand_matrix(size_t n, double A[][N], double B[][N]){
    srand(GetTickCount());
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j) {
            A[i][j] = (double)rand() / RAND_MAX * 1000000.0;
            B[i][j] = (double)rand() / RAND_MAX * 1000000.0;
        }
    }
}
```

Figura 6: Função "*make_rand_matrix*", que realiza a atribuição de elementos randômicos, em formato float, às matrizes operandos A e B.

```
LARGE_INTEGER frequency;
QueryPerformanceFrequency(&frequency);

for (int i = 0; i < 10; ++i) {
    make_rand_matrix(n, A, B);

    LARGE_INTEGER start, end;
    double executionTime;

    QueryPerformanceCounter(&start);

    // Chama a função de multiplicação de matriz
    dgemm(n, (double *)A, (double *)B, (double *)C);

    QueryPerformanceCounter(&end);

    // Calcula o tempo de execução em segundos
    executionTime = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;
```

Figura 7: A medição de tempo foi feita com a função específica "*QueryPerformanceCounter*" do API do Windows, que mede o tempo de execução por meio da frequência de unidades de contadores de alta resolução.

Uma observação importante é que é possível otimizar o tempo de execução do código a partir de jeitos diferentes de compilação do arquivo. Tais otimizações são chamadas de O1, O2 e O3, e podem ser feitas na hora de se compilar o arquivo adicionando uma flag com tal otimização. Dessa forma, nosso grupo decidiu executar todas essas otimizações por parte do compilador nessa sessão de Going Faster, uma vez que, a partir do próximo, que trata do uso do AVX, a compilação em otimização O3 passa a ser necessária.

3.3 Going Faster 3

No Going Faster 3, o livro traz a primeira otimização relacionada ao hardware para o código do DGEMM: o uso do AVX, que faz paralelismo das subwords. Para criação desse código, a base da função main e as funções de inicialização de matriz, print e medição de tempo foram idênticas as utilizadas anteriormente no Going Faster 2. Em contrapartida, a única mudança foi feita na função DGEMM, que ,agora, indica ao compilador o tamanho de bits que será armazenado por variável,

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

Figura 8: Função DGEMM com a otimização de AVX.

3.4 Going Faster 4

No Going Faster 4, foi apresentada a técnica do ILP, no qual é feito unroll sobre o loop da multiplicação de matrizes, visando aumentar a quantidade de instruções que o processador pode trabalhar simultaneamente. Isso ocorre porque, ao desdobrar um laço, várias de suas iterações são explicitamente escritas dentro do corpo do laço, reduzindo a sobrecarga de controle de laço e permitindo que o compilador (ou hardware) encontre mais paralelismo entre as instruções.

O código criado para a execução dessa seção foi diferente do proposto pelo livro. Em razão do livro trabalhar com um processador de 86 bits, seu código foi moldado para execução nesse tipo de arquitetura, porém, como o computador utilizado por nosso grupo é de 64 bits, foi necessária uma adaptação. De forma geral, ao invés de trabalhar com unroll de tamanho 32 (8*4), utilizamos um de tamanho 16 (4*4), além de fornecer ao compilador um tamanho de bits de 256 (4*64).

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

                        for (size_t k = 0; k < n; k++){
                                __m256d bb = _mm256_broadcast_sd(A + (i * N) + k);
                                for (int r = 0; r < UNROLL; r++){
                                        c[r] = _mm256_add_pd(c[r], _mm256_mul_pd(bb,
                                                _mm256_load_pd(B + (N * k) + j + 4 * r)));
                                }
                        }
                        for (int r = 0; r < UNROLL; r++)
                                _mm256_store_pd(C + (i * N) + j + r * 4, c[r]);
                }
}
```

Figura 9: Função DGEMM com a otimização de ILP.

3.5 Going Faster 5

O Going Faster 5 traz a técnica do cache blocking. Código do DGEMM igual ao do livro, exceto pelas alterações feitas no Going Faster anterior, como o tamanho do unroll e o tamanho de bits das variáveis.

```
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++)
    for (int j = sj; j < sj + BLOCKSIZE; j += 4 * UNROLL) {
      __m256d c[UNROLL];
      for (int r = 0; r < UNROLL; r++)
        c[r] = _mm256_load_pd(C + (i * N) + j + r * 4); //[ UNROLL];

      for (int k = sk; k < sk + BLOCKSIZE; k++)
      {
        __m256d bb = _mm256_broadcast_sd(A + (i * N) + k);
        for (int r = 0; r < UNROLL; r++)
          c[r] = _mm256_add_pd(c[r], _mm256_mul_pd(bb, _mm256_load_pd(B + (N * k) + j + 4 * r)));
      }
      for (int r = 0; r < UNROLL; r++)
        _mm256_store_pd(C + (i * N) + j + r * 4, c[r]);
    }
}

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

Figura 10: Função DGEMM com a otimização de CB.

3.6 Going Faster 6

No Going Faster 6 é proposto a otimização do multiprocessador, que diminui o tempo de execução da multiplicação matricial a partir da divisão do trabalho entre os núcleos do processador. Dessa forma, seu código é praticamente idêntico ao do Going Faster 5, havendo apenas uma diferença entre eles que é uma linha de código a mais dentro da função DGEMM que é a responsável por possibilitar o uso dos diversos núcleos do computador. Vale lembrar que foi necessário, na compilação, utilizar uma flag a mais, sendo ela a responsável por ativar o OpenMP para execução da divisão do trabalho para cada núcleo do multiprocessador.

```
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++)
    for (int j = sj; j < sj + BLOCKSIZE; j += 4 * UNROLL) {
      __m256d c[UNROLL];
      for (int r = 0; r < UNROLL; r++)
        c[r] = _mm256_load_pd(C + (i * N) + j + r * 4); //[ UNROLL];

      for (int k = sk; k < sk + BLOCKSIZE; k++)
      {
        __m256d bb = _mm256_broadcast_sd(A + (i * N) + k);
        for (int r = 0; r < UNROLL; r++)
          c[r] = _mm256_add_pd(c[r], _mm256_mul_pd(bb, _mm256_load_pd(B + (N * k) + j + 4 * r)));
      }
      for (int r = 0; r < UNROLL; r++)
        _mm256_store_pd(C + (i * N) + j + r * 4, c[r]);
    }
}

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

Figura 11: Função DGEMM com a otimização de multiprocessador. Note que a única diferença entre o código do Going Faster 6 com o do 5 é a linha logo em seguida da função DGEMM.

4 Resultados e comparação dos dados

Os códigos criados e discutidos na seção anterior foram executados no computador de um dos integrantes do grupo, dentro do próprio VScode, de modo a coletar os tempos de execução da multiplicação de matrizes para cada código e para diferentes tamanhos de matrizes. As especificações do computador utilizado para executar os códigos são as seguintes:

- Dado do processador: 13th Gen Intel(R) Core(TM) i9-13950HX 2.20 GHz 24/32
- RAM instalada: 16,0 GB (utilizável: 15,7 GB)
- Tipo de sistema: Sistema operacional de 64 bits, processador baseado em x64

```
PS C:\Users\guilh\OneDrive\VScode\Arq comp\C 5 (multiprocessador)> ./DGEMM_6.exe
Insira o valor de n: 1024
Tempo de execucao do ciclo 1: 0.014324 segundos
Tempo de execucao do ciclo 2: 0.010829 segundos
Tempo de execucao do ciclo 3: 0.011811 segundos
Tempo de execucao do ciclo 4: 0.011345 segundos
Tempo de execucao do ciclo 5: 0.012048 segundos
Tempo de execucao do ciclo 6: 0.011874 segundos
Tempo de execucao do ciclo 7: 0.011506 segundos
Tempo de execucao do ciclo 8: 0.011421 segundos
Tempo de execucao do ciclo 9: 0.013051 segundos
Tempo de execucao do ciclo 10: 0.010766 segundos
Media dos tempos de execucao: 0.011897 segundos
Desvio padrao dos tempos de execucao: 0.001018 segundos
PS C:\Users\guilh\OneDrive\VScode\Arq comp\C 5 (multiprocessador)>
```

Figura 12: Com o loop predefinido no código, era possível obter 10 tempos de execução seguidos a partir de uma execução e uma entrada

Sobre as matrizes, foi definido pelo grupo que os tamanhos que seriam computados seriam de: 1k (1024), 2k (2048), 4k (4096) e 8k (8192). Além disso, foi decidido que, para cada tamanho de matriz, seriam feitas 10 execuções do DGEMM, sendo isso feito para todos os códigos de otimização. De um modo geral, os códigos criados pelo grupo já continham um loop de *for* que realiza 10 iterações do código da multiplicação, dessa forma, bastava informar como input o tamanho das matrizes operandos e o código já estava programado para printar 10 tempos de execuções seguidos.

Porém, não bastou a execução do código apenas uma vez para cada código e tamanho de matriz. Isso porque, a fim de obter os valores mais acurados e precisos possíveis, o código era rodado mais 2 ou 3 vezes, e desses resultados, era pego o conjunto de dados que possuía o menor tempo médio e menor desvio padrão entre os valores resultantes.

A seguir está o link das tabelas que contém os resultados encontrados pelo grupo na execução dos códigos com suas respectivas otimizações:

- *https://docs.google.com/spreadsheets/d/1BQIK__3G-HsydQAq2BYdlIVK6E1mCGT2f6K1Ktf6VnU/edit?usp=sharing*

Nas subseções adiante, serão mostradas as tabelas de cada tamanho de matriz e seus respectivos gráficos, mostrando o tempo médio de execução da multiplicação matricial:

4.1 1024

1024x1024					
	Python	C-1 (sem otim.)	C-1 (O1)	C-1 (O2)	C-1 (O3)
1	81848	3400,016	2445,469	2205,022	1174,413
2	82705	3396,899	2445,043	2187,672	1179,552
3	83211	3408,365	2458,613	2185,841	1174,479
4	89366	3404,187	2447,918	2189,58	1170,379
5	83437	3404,912	2454,469	2170,702	1176,575
6	82757	3405,994	2446,906	2185,152	1167,545
7	83093	3405,135	2437,452	2179,559	1176,516
8	82272	3415,518	2442,001	2180,265	1176,527
9	83064	3407,038	2459,369	2181,267	1182,696
10	82765	3396,312	2472,444	2230,202	1180,957
MÉDIA (ms)	*83451,8*	*3404,4376*	*2450,9684*	*2189,5262*	*1175,9639*
Desvio Padrão	*2128,936761*	*5,68198743*	*10,3103762*	*16,79442894*	*4,604360529*
	C-2 (AVX)	C-3 (ILP)	C-4 (CB)	C-5 (multi. proc.)	
1	876,368	200,487	90,057	14,742	
2	882,676	189,761	85,546	12,207	
3	858,987	192,326	92,142	12,336	
4	858,858	193,178	87,58	10,39	
5	871,975	198,83	92,162	11,084	
6	864,984	194,636	87,373	13,142	
7	878,011	192,793	86,678	10,643	
8	890,896	193,464	87,432	10,9	
9	885,81	190,48	87,607	12,034	
10	874,739	194,566	89,335	12,138	
MÉDIA (ms)	*874,3304*	*194,0521*	*88,5912*	*11,9616*	
Desvio Padrão	*10,85988096*	*3,361372122*	*2,256179406*	*1,310917084*	

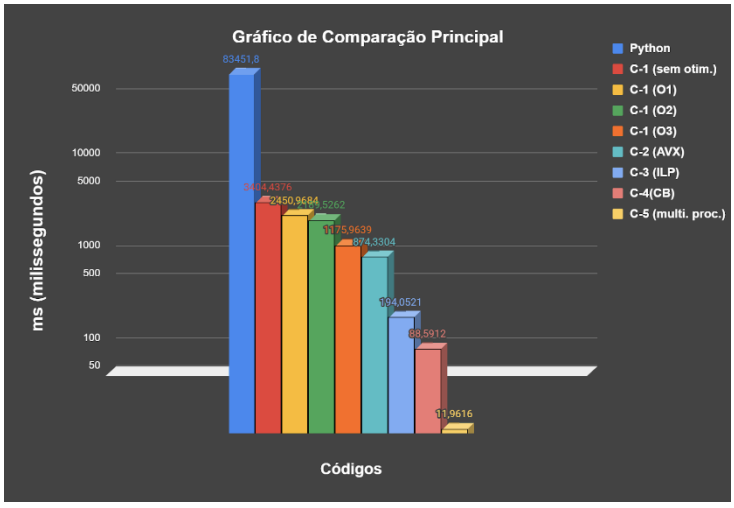

Figura 13: Tempos médios de execução da multiplicação de matrizes de tamanho 1024 para cada otimização proposta pelo livro, em milissegundos.

4.2 2048

2048x2048					
	Python	C-1 (sem otim.)	C-1 (O1)	C-1 (O2)	C-1 (O3)
1	670588	55461,566	38832,419	36588,792	22972,604
2	676084	55669,39	38669,303	36711,425	23235,941
3	672761	55879,483	38685,128	36543,592	23634,657
4	662531	55836,602	38550,371	36559,823	23668,912
5	668836	55820,25	38625,071	36499,373	24029,919
6	678771	55675,05	38537,977	36638,816	24051,15
7	672856	55601,152	38650,842	36704,719	24596,396
8	661368	55596,58	38706,644	36461,81	23566,527
9	667351	55931,311	38802,623	36598,771	24952,119
10	675614	55821,256	38837,159	36631,12	24964,537
MÉDIA (ms)	*670676*	*55729,264*	*38689,7537*	*36593,8241*	*23967,2762*
Desvio Padrão	*5731,858492*	*150,5010764*	*107,3456388*	*81,39121653*	*687,6429719*
	C-2 (AVX)	C-3 (ILP)	C-4 (CB)	C-5 (multi. proc.)	
1	8468,106	2781,172	722,71	100,891	
2	8615,018	2782,49	713,298	107,271	
3	8642,223	2794,674	703,47	105,827	
4	8593,18	2776,7	711,553	107,408	
5	8703,578	2811,639	729,569	106,208	
6	8651,594	2775,314	709,039	99,605	
7	8592,156	2762,843	728,794	103,969	
8	8644,871	2760,441	713,953	102,797	
9	8706,003	2832,555	716,884	106,176	
10	8644,35	2800	707,677	107,043	
MÉDIA (ms)	*8626,1079*	*2787,7828*	*715,6947*	*104,7195*	
Desvio Padrão	*67,74418685*	*22,33582144*	*8,810809296*	*2,784384961*	

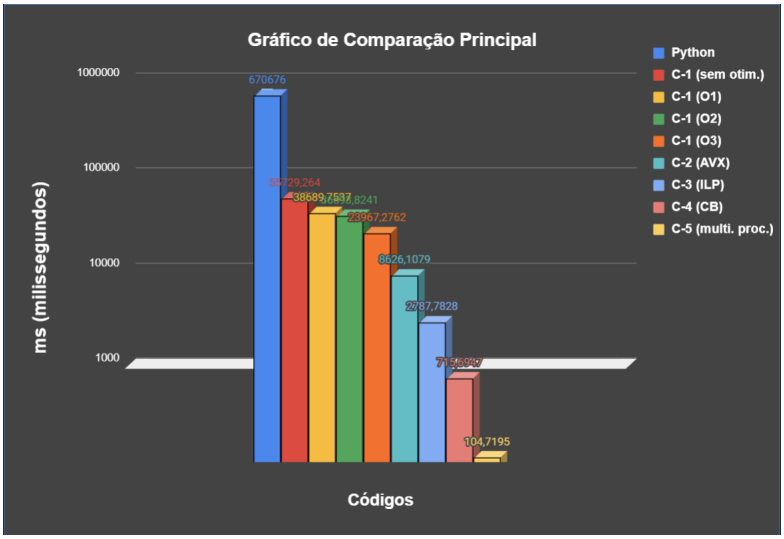

Figura 14: Tempos médios de execução da multiplicação de matrizes de tamanho 2048 para cada otimização proposta pelo livro, em milissegundos.

4.3 4096

4096x4096					
	Python	C-1 (sem otim.)	C-1 (O1)	C-1 (O2)	C-1 (O3)
1	7211620	898611,966	534814,118	426009,416	241057,469
2	7250955	896574,411	536719,238	425967,707	239913,138
3	7352173	896265,477	532477,152	426102,045	239661,128
4	7405961	899611,251	531343,164	425874,412	240936,325
5	7290199	897190,877	531012,781	426179,522	240386,431
6	7485953	894229,271	531986,472	425829,122	239089,678
7	7459194	891808,674	531612,94	425823,163	241330,368
8	7246334	895868,182	532299,775	426264,005	240548,839
9	7464002	893571,332	532138,702	426274,023	240599,539
10	7352173	898442,093	530996,913	425664,659	240342,883
MÉDIA (ms)	*7351856,4*	*896217,3534*	*532540,1255*	*425998,8074*	*240386,5798*
Desvio Padrão	*99884,01863*	*2444,767804*	*1830,971389*	*204,7335392*	*678,9713709*
	C-2 (AVX)	C-3 (ILP)	C-4 (CB)	C-5 (multi. proc.)	
1	102687,971	45834,829	6150,812	966,445	
2	103551,435	45497,743	6057,374	976,133	
3	103200,81	45686,792	6010,163	975,067	
4	102532,47	45315,14	6047,797	977,809	
5	104595,525	45398,518	6062,622	975,023	
6	103957,792	45806,373	6004,763	968,167	
7	102855,578	45630,254	5971,005	964,533	
8	104074,342	45767,134	5988,308	976,283	
9	104502,905	45161,647	5982,697	987,232	
10	103660,255	45367,977	6008,189	987,629	
MÉDIA (ms)	*103561,9083*	*45546,6407*	*6028,373*	*975,4321*	
Desvio Padrão	*733,0102568*	*231,8593899*	*53,3975355*	*7,799627063*	

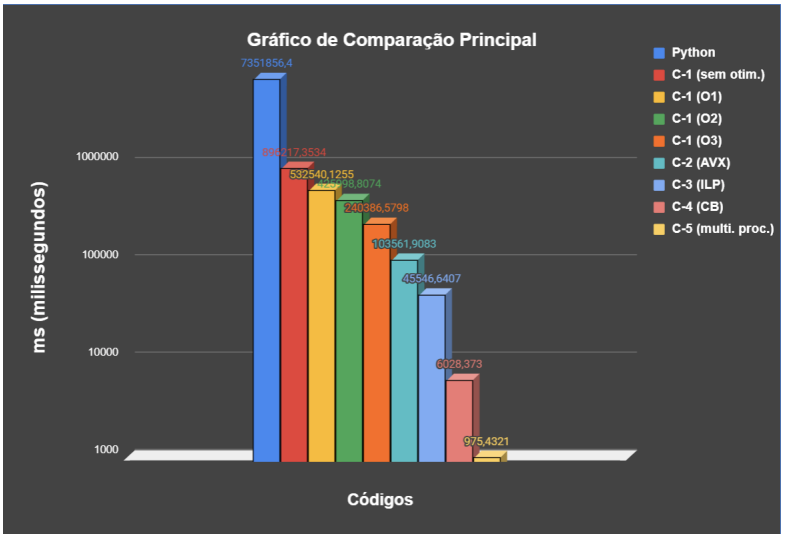

Figura 15: Tempos médios de execução da multiplicação de matrizes de tamanho 4096 para cada otimização proposta pelo livro, em milissegundos.

4.4 8192

8192x8192					
	Python	C-1 (sem otim.)	C-1 (O1)	C-1 (O2)	C-1 (O3)
1	69391908	9343037,19	6463863,643	5783509,161	3196128,365
2	69729570	9330193,86	6485233,09	5778852,728	3205287,52
3	73341272	9335836,379	6473825,994	5777200,675	3205290,201
4	70448721	9342994,117	6473960,168	5779395,451	3207322,993
5	69895131	9323667,416	6471185,917	5745367,788	3204547,207
6	71004726	9385068,689	6472236,511	5737619,455	3202856,895
7	70770621	9360860,927	6465738,997	5736666,729	3204825,835
8	72077843	9380411,636	6466240,105	5735132,194	3203954,239
9	71898041	9385139,764	6466736,699	5732646,885	3204178,2
10	71022077	9389573,771	6466712,427	5734997,505	3204103,698
MÉDIA (ms)	*70957991*	*9357678,375*	*6470573,355*	*5754138,857*	*3203849,515*
Desvio Padrão	*1211511,955*	*25533,88094*	*6297,215569*	*22330,29873*	*2952,090042*
	C-2 (AVX)	C-3 (ILP)	C-4 (CB)	C-5 (multi. proc.)	
1	1616626,275	409883,949	65253,686	9576,017	
2	1617141,563	410923,304	58992,522	9387,689	
3	1615918,636	411072,985	59012,81	9420,513	
4	1615673,077	410761,654	58884,567	9301,488	
5	1617893,041	409107,9	58854,681	9392,93	
6	1614904,058	408940,864	58758,898	9365,108	
7	1617054,113	408757,045	58870,027	9940,572	
8	1615599,438	408712,561	58897,148	9998,422	
9	1615889,234	408416,265	58974,743	9969,417	
10	1615324,182	408201,95	59076,45	9846,158	
MÉDIA (ms)	*1616202,362*	*409477,8477*	*59557,5532*	*9619,8314*	
Desvio Padrão	*938,3602089*	*1092,000694*	*2003,515785*	*285,3628475*	

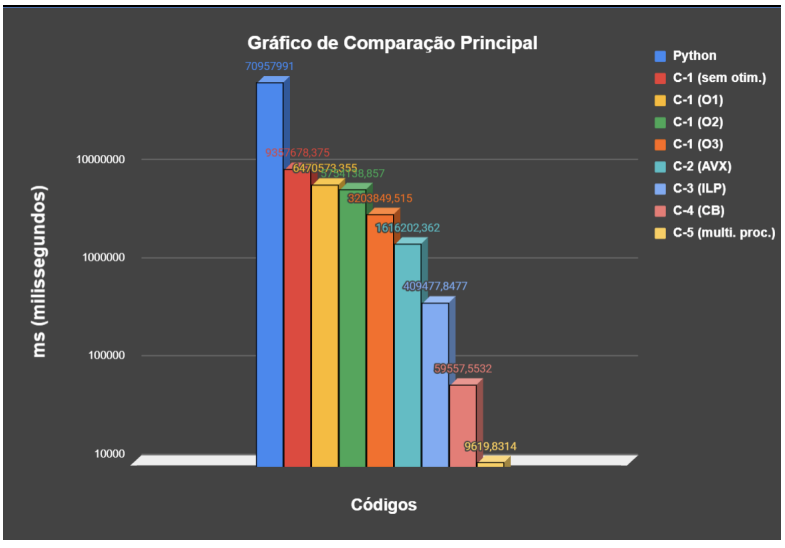

Figura 16: Tempos médios de execução da multiplicação de matrizes de tamanho 8192 para cada otimização proposta pelo livro, em milissegundos.

4.5 Análise geral

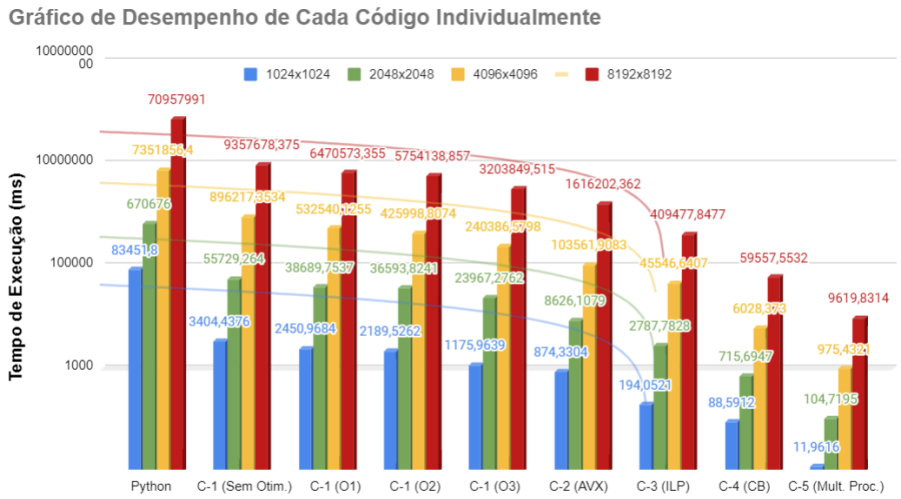

Figura 17: Gráficos geral de todo o desempenho médio de cada código e cada tamanho de matriz.

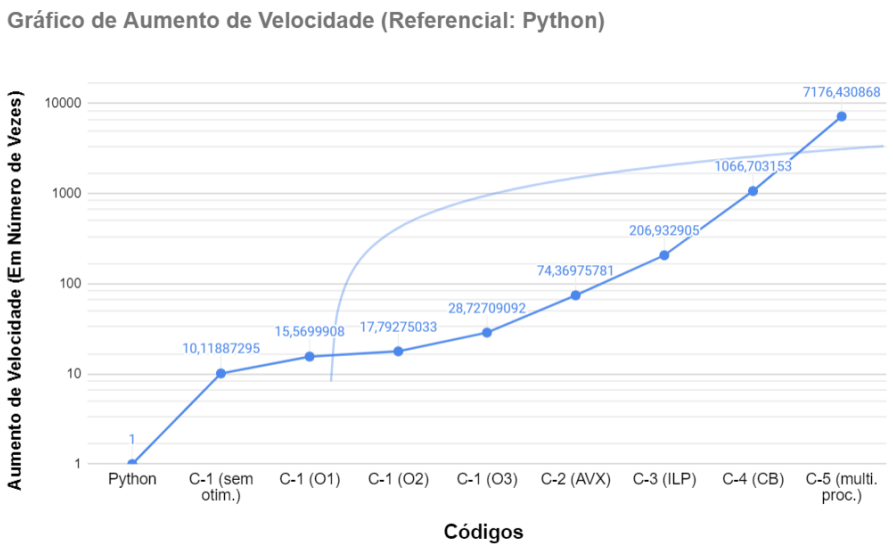

Figura 18: Gráfico do aumento de velocidade com referencial no Python.

A partir das informações de performance fornecidas pelo livro e os dados obtidos através dos testes realizados pelo grupo, pode-se fazer uma comparação de modo a verificar se os dados experimentais se aproximam dos dados propostos na teoria.

Para a primeira otimização, feita com todas as etapas completas na troca da linguagem de Python para C propostas pelo livro, ele propõe uma melhora da performance de 175 vezes. Pela observação dos dados, essa mudança proporcionou um aumento próximo de 28,73 vezes na velocidade.

A análise da otimização proposta pelo Capítulo 3 permite notar uma melhora de 2,6 vezes no desempenho em relação ao capítulo anterior. O livro define, teoricamente, uma melhora 7,8 vezes na mesma comparação.

A mudança no desempenho com as ideias do Capítulo 4 foi de, aproximadamente, 2,8 vezes a velocidade de execução do capítulo anterior, enquanto que o livro defende um aumento de 1,8 vezes a partir das mesmas alterações.

Para as modificações do Capítulo 5, os dados experimentais propõem um incremento de 5,2 vezes na velocidade. O livro argumenta que o aumento seria de 1,5 vezes.

Por fim, no último capítulo, foi obtida uma melhora de 6,7 vezes, enquanto o livro aborda um aumento, mais significativo, de quase 12 vezes.

Com isso, conclui-se que, apesar de haver uma diferença na melhora esperada a partir da teoria do livro e o incremento analisado a partir dos dados coletados (em alguns capítulos nossos dados superaram o aumento de desempenho proposto e em outros capítulos houve um desempenho inferior ao esperado) é notório que cada uma das alterações propostas pelo livro são capazes de influenciar positivamente no desempenho do código, ampliando sua proximidade com o hardware disponível.

5 Códigos extras

Além das linguagens e otimizações utilizadas pelo livro para a realização do DGEMM, nosso grupo também realizou a execução da multiplicação de matrizes nas linguagens JavaScript e Swift, e usando a biblioteca de otimização Numpy do Python. Os parâmetros de medições utilizados na análise dos códigos associados ao livro foram mantidos: mesmo tamanhos de matrizes, quantidade de dados por tamanho e PC utilizado.

Vale ressaltar que foram feitos dois códigos para o JavaScript e para o Swift, ambos seguindo o mesmo padrão: um deles é um código de multiplicação de matrizes normal, sem otimização; enquanto o outro possui otimizações, contendo bibliotecas próprias da linguagem que agilizem o DGEMM.

Link: *https://github.com/GuilhermeHu/DGEMM/tree/main/Extras*

5.1 1024

1024x1024					
Extras	Swift Não Otimizado	JavaScript Não Otimizado	JavaScript Otimizado	Swift Otimizado	Numpy
1	200629,173	2579,426625	1721,581208	190,1539564	17,092
2	201788,51	2662,745333	1715,875333	197,7969408	14,3073
3	200652,238	2552,266333	1671,025375	192,6039457	14,1867
4	199769,886	2663,893458	1637,468708	192,7599907	14,731
5	204729,846	2580,238583	1634,945291	189,3860102	14,5878
6	201459,163	2641,624208	1747,49175	192,1700239	14,4319
7	200501,315	2545,496459	1643,849959	194,8570013	14,172
8	200461,156	2586,887209	1644,507958	187,8238916	14,4269
9	200262,676	2580,261083	1693,792542	190,969944	13,8676
10	202968,6041	2681,463208	1705,794583	195,0139999	14,0771
MÉDIA (ms)	*201322,256708145*	*2607,43025*	*1681,633271*	*192,3535705*	*14,58803*
Desvio Padrão	*1506,85676*	*49,95618085*	*40,66937654*	*2,968613867*	*0,914917762*

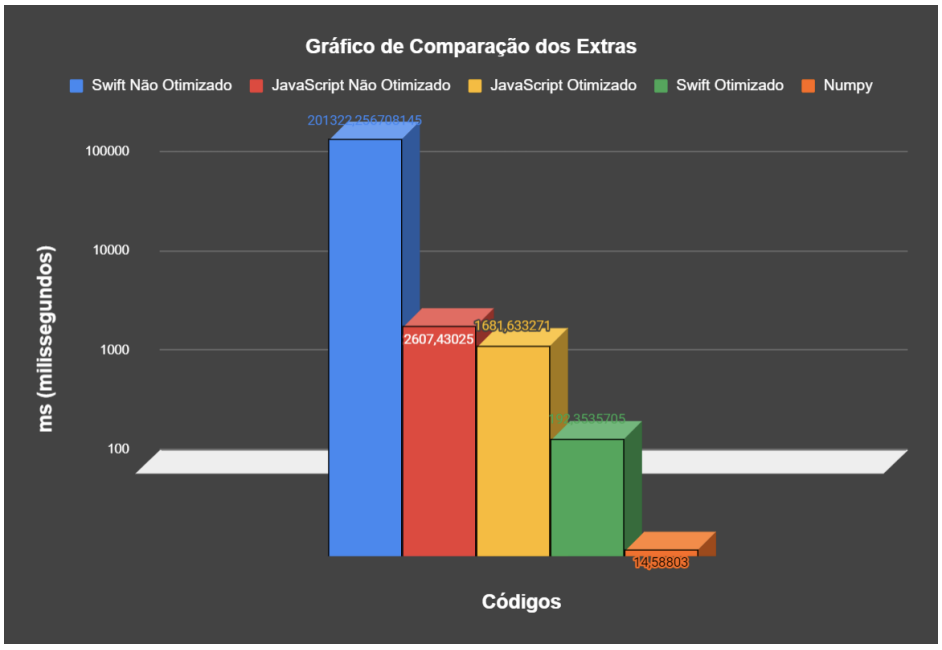

Figura 19: Tempos médios de execução da multiplicação de matrizes de tamanho 1024 para cada otimização extra feita pelo grupo, em milissegundos.

5.2 2048

Extras	Swift Não Otimizado	JavaScript Não Otimizado	JavaScript Otimizado	Swift Otimizado
1	1720364,904	54491,54342	34201,91617	789,4020081
2	1722979,722	57527,20913	32477,21221	830,5090666
3	1728165,992	61458,80438	33849,50646	828,2370567
4	1689646,847	59708,75821	34640,67283	817,0410395
5	1711691,478	59723,60413	36609,45058	826,597929
6	1703646,387	60704,94275	35876,76808	819,8350668
7	1720104,501	62065,9235	36526,76088	792,1099663
8	1697267,424	60495,97058	36106,3805	829,2599916
9	1734219,054	59439,06779	34211,8095	814,4088984
10	1772861,636	55696,73992	34099,35646	795,7890034
MÉDIA (ms)	*1720094,794*	*59131,25638*	*34859,98337*	*814,3190026*
Desvio Padrão	*23206,09727*	*2473,046341*	*1358,292799*	*16,07323138*

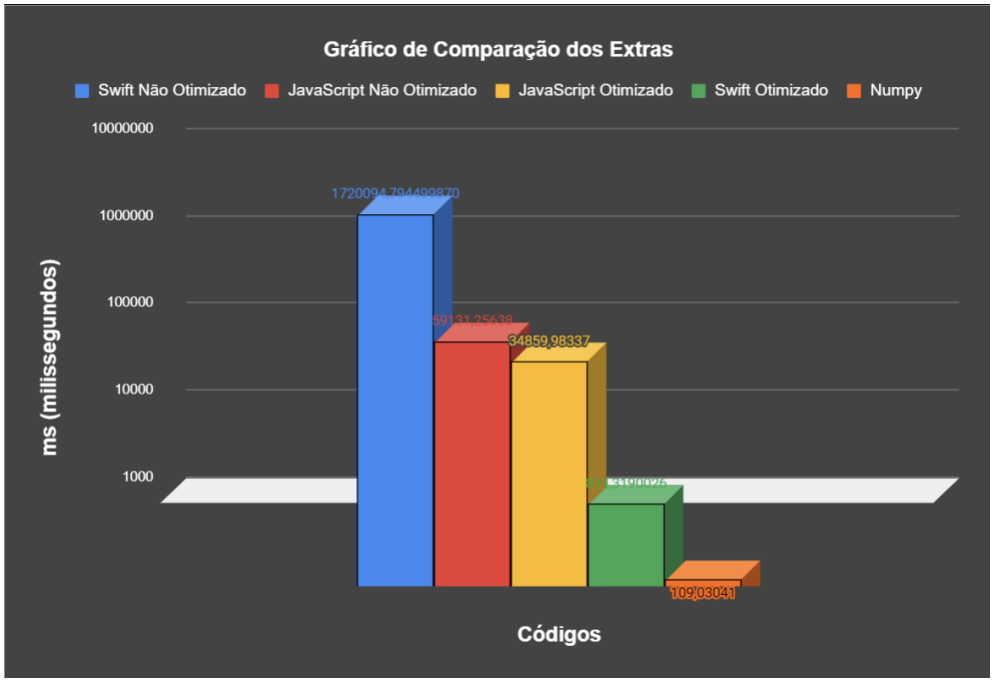

Figura 20: Tempos médios de execução da multiplicação de matrizes de tamanho 2048 para cada otimização extra feita pelo grupo, em milissegundos.

5.3 4096

Extras	Swift Não Otimizado	JavaScript Não Otimizado	JavaScript Otimizado	Swift Otimizado	Numpy
1	77357791,51	692539,9231	449480,8554	3597,591996	946,3436
2	81982688,91	693768,4341	439051,819	3497,012019	983,4415
3	51499243,04	686045,2307	431293,8995	3631,466985	920,491
4	79171153,32	676786,3024	436769,3805	3520,897985	883,4694
5	54139836,18	712255,576	441445,1853	3471,03107	878,6627
6	50159554,96	676196,8378	432445,3263	3586,102009	883,2251
7	76317013,33	669696,5988	430453,991	3550,33195	893,0501
8	46861915,63	676929,9612	445593,3104	3519,091964	883,1007
9	59947463,92	662216,9485	434040,2568	3614,868999	876,9292
10	61274912,47	667203,8428	434472,687	3429,412961	890,536
MÉDIA (ms)	*63871157,33*	*681363,9655*	*437504,6711*	*3541,780794*	*903,92493*
Desvio Padrão	*13524164,22*	*15001,9928*	*6345,557349*	*65,93351232*	*35,49913684*

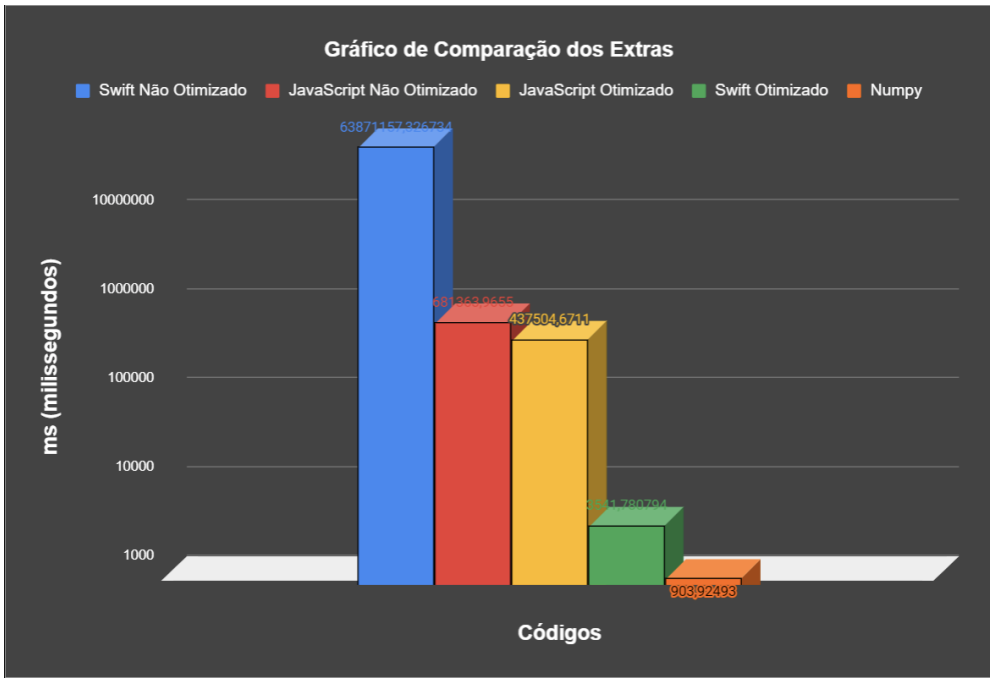

Figura 21: Tempos médios de execução da multiplicação de matrizes de tamanho 4096 para cada otimização extra feita pelo grupo, em milissegundos.

5.4 8192

Extras	Swift Não Otimizado	JavaScript Não Otimizado	JavaScript Otimizado	Swift Otimizado	Numpy
1	325270362,7	5784672,727	3977248,727	16960,08599	6887,8802
2	355003390,4	5321452,218	3421395,112	16019,44196	6889,6141
3	393994670,9	5838202,85	4149222,676	16188,82704	6777,2559
4	345351727	6261344,775	3663789,77	16165,74299	6929,1721
5	285073549,3	5320802,056	3317530,236	16337,75294	6932,2009
6	302401404,4	5710895,678	4106785,278	15655,85101	6942,6739
7	336538724,7	5290789,453	3472237,577	16806,43499	6940,3518
8	378151662,3	5955124,568	3572837,434	16216,89701	7521,1048
9	379856867,5	6033543,212	3194685,91	16276,41404	7306,044
10	309234699	5401234,568	3599632,557	15837,82494	7388,0562
MÉDIA (ms)	*341087705,8*	*5691806,21*	*3647536,528*	*16246,52729*	*7051,43539*
Desvio Padrão	*36220729,39*	*343344,3313*	*329190,0753*	*395,82757*	*253,896981*

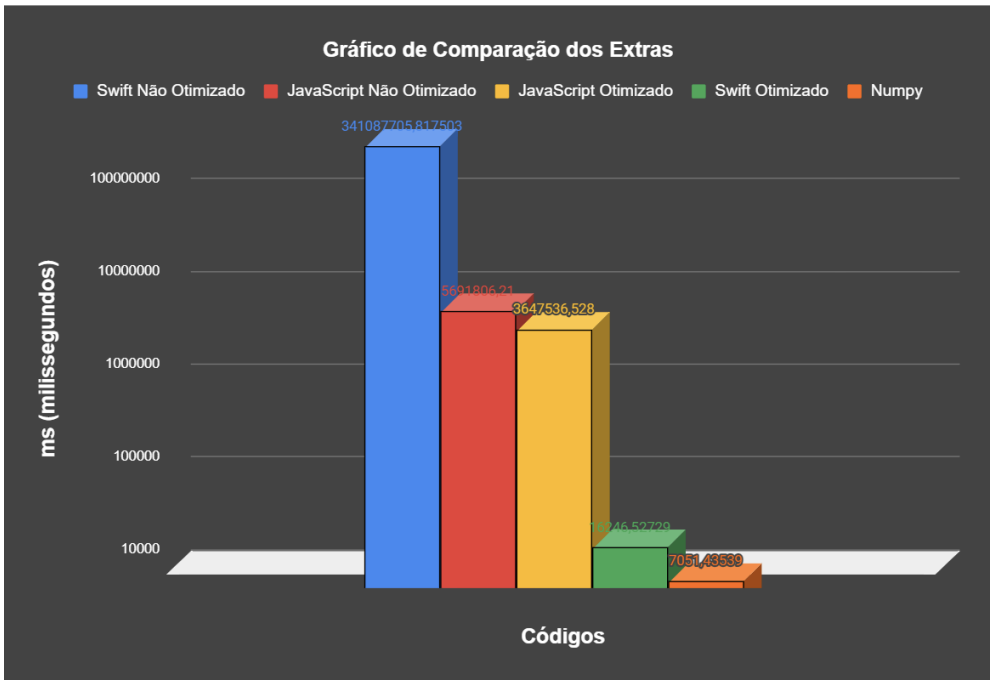

Figura 22: Tempos médios de execução da multiplicação de matrizes de tamanho 8192 para cada otimização extra feita pelo grupo, em milissegundos.

5.5 Análise geral

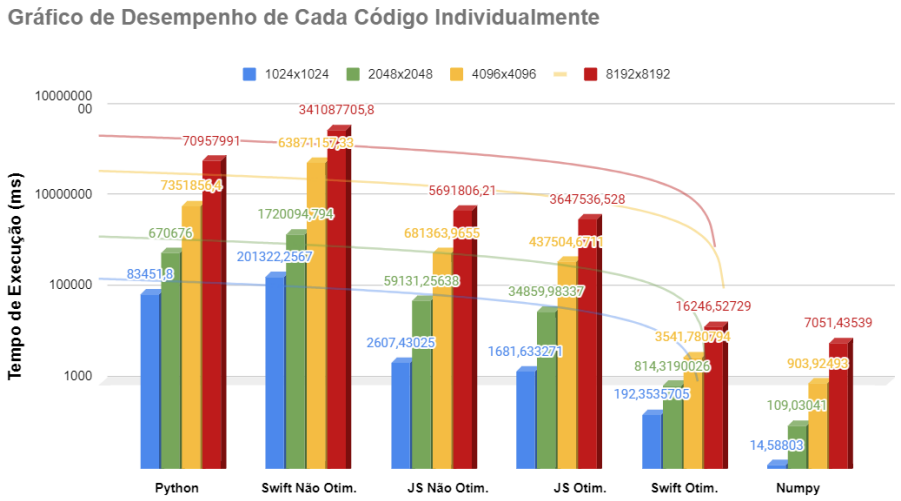

Figura 23: Gráficos geral de todo o desempenho médio de cada código e cada tamanho de matriz.

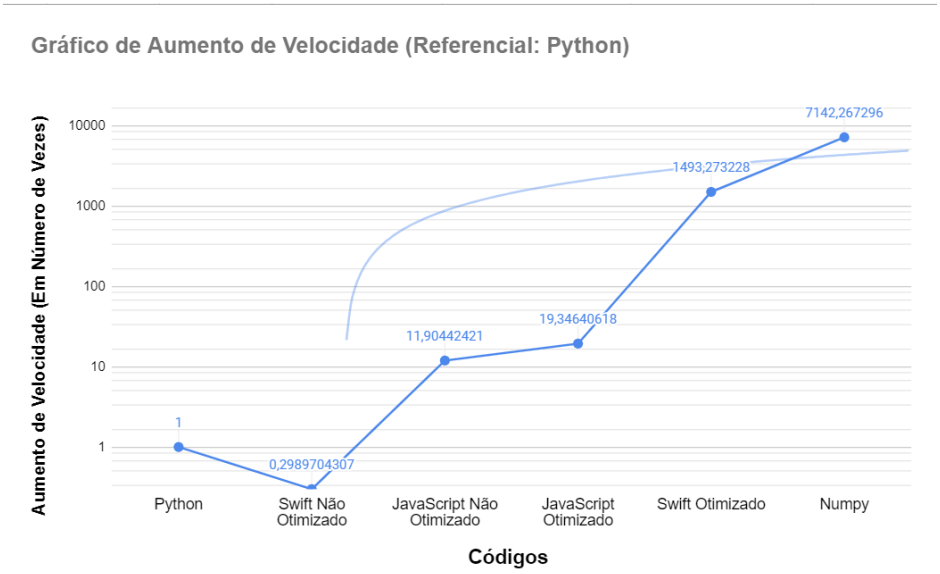

Figura 24: Gráfico do aumento de velocidade com referencial no Python.

O código de Swift não otimizado apresenta um desempenho duas vezes melhor do que o Python, tendo um desempenho ruim se comparado aos demais.

Ao realizar uma otimização deste código, usando bibliotecas da linguagem, é possível aumentar o desempenho significativamente. Tal melhoria faz ele ficar 1,69 vezes mais lento que a versão de melhor otimização fornecida pelo livro.

Para a segunda linguagem extra, o JavaScript, foi realizada a execução da versão não otimizada e otimizada, também. Para a primeira versão, há uma performance, aproximadamente, 2 vezes pior do que a otimização final proposta pelo Capítulo 2 (C-1 O3) do livro.

Já para a otimização usando bibliotecas da linguagem, o código fica próximo do desempenho da otimização final proposta pelo Capítulo 2 (C-1 O3).

6 Conclusão

A partir de todas as discussões apresentadas ao longo desse relatório, é possível perceber que há uma relação direta entre ampliar a proximidade do hardware do computador e melhorar o desempenho da execução do código.

Com isso, é possível concluir que, para que se obtenha o melhor resultado possível durante a execução dos códigos, mostra-se necessária uma maior afinidade do código desenvolvido pelo humano com o hardware disponível para a execução, de forma a influenciar positivamente na performance dependendo da adaptação do código. Além disso, quanto maior o nível de especificação do código em relação ao hardware, maior o ganho de performance.

7 Apêndice

- GitHub com o código da projeto:
  https://github.com/GuilhermeHu/DGEMM
- Tabelas dos dados encontrados nas medições de tempo dos códigos:
  https://docs.google.com/spreadsheets/d/1BQIK__3G-HsydQAq2BYdlIVK6E1mCGT2f6K1Ktf6VnU/edit?usp=sharing