SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

RELATÓRIO TP MULTIPLICAÇÃO DE MATRIZES

Disciplina: Introdução à Programação Paralela
Autores: Gabriel Melo Silva, Gabriela Ramalho Santos
Professor Orientador: Anolan Yamile Milanes Barrientos

1 Introdução

O trabalho consiste na implementação de uma rotina para multiplicação de matrizes quadradas, que tenha o melhor desempenho possível. Inicialmente um método implementado de forma ingênua, após sucessivas otimizações espera-se que o tempo de execução do programa diminua, tenha poucas falhas de cache e seja relativamente competitivo com o algoritmo de referência.

2 Metodologia

Um método ingênuo que efetua a multiplicação de matrizes, é apresentado a seguir.

```
1  void square_dgemm (int n, double* A, double* B, double* C){
2    /* For each row i of A */
3    for (int i = 0; i < n; ++i){ /
4      /* For each column j of B */
5      for (int j = 0; j < n; ++j) {
6      /* Compute C(i,j) */
7         double cij = C[i+j*n];
8              for( int k = 0; k < n; k++ ){
9                cij += A[i+k*n] * B[k+j*n];
10             }
11             C[i+j*n] = cij;
12     }
13   }
14 }
```

A operação é realizada multiplicando cada linha i de A por cada coluna j de B, resultando em um elemento na posição ij da coluna C. Veja que dessa forma preenchemos C linha por linha. Entretanto, em nosso programa de análise de desempenho benchmark.c, as matrizes estão armazenadas em forma de coluna, ou seja, n elementos adjacentes na memória formam uma coluna de uma matriz qualquer de ordem n. Por exemplo, seja A = $\begin{pmatrix} a1 & a4 & a7 \\ a2 & a5 & a8 \\ a3 & a6 & a9 \end{pmatrix}$, uma matriz quadrada de ordem 3. Em nosso programa ela estaria armazenada em apenas 1 dimensão, com um vetor A1 = $\begin{pmatrix} a1 & a2 & a3 & a4 & a5 & a6 & a7 & a8 & a9 \end{pmatrix}$.

Para matrizes de tamanho muito grande, a forma de multiplicação linha x coluna, pode resultar em várias falhas de cache. Essa é uma consequência do princípio de localidade espacial, que parte do pressuposto de que se um dado é acessado, dados cujos cujos endereços são próximos a este, tenderão a ser acessados também. Sabendo disso, almejando reduzir as falhas de cache, ao invés de preenchermos C linha por linha, iremos preencher coluna por coluna. Para isso, inventemos a ordem dos loops na rotina, de forma a fixar uma coluna a ser trabalhada. Sendo assim:

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

Essa mudança deu certo, pois foi observado que no método implementado, houve um aumento no valor de peak. Nos testes realizados quando era executado a versão original do código era encontrado um Peak médio de 9.9003, com a troca o valor ficou na média de 10.0315.

Percebe-se que na multiplicação de matrizes, uma mesma linha é multiplicada diversas vezes para encontrar os valores de elementos de posições diferentes da matriz resultante. Em busca de minimizar quantas vezes cada elemento de uma determinada linha ou coluna sera multiplicado, dividimos a matriz em blocos, para que a matriz resultante seja a soma dos resultados calculados por cada bloco da matriz. Além disso, ao dividir a matriz em blocos do tamanho da linha de cache, o problema de falha de cache também é minimizado. Dessa forma, segue o programa dividido em blocos:

```
const char* dgemm_desc = "Simple blocked dgemm";

#if !defined(BLOCK_SIZE)
#define BLOCK_SIZE 32
#endif

#define min(a,b) (((a)<(b))?(a):(b))

static void do_block (int lda, int M, int N, int K, double* A, double* B, double*
    C)
{
  /* For each row i of A */
  for (int i = 0; i < M; ++i){
    /* For each column j of B */
    for (int j = 0; j < N; ++j) {
      /* Compute C(i,j) */
      double cij = C[i+j*lda];
      for (int k = 0; k < K; ++k)
        cij += A[i+k*lda] * B[k+j*lda];
        C[i+j*lda] = cij;
      }
    }
}

/* This routine performs a dgemm operation
 *  C := C + A * B
 * where A, B, and C are lda-by-lda matrices stored in column-major format.
 * On exit, A and B maintain their input values. */
void square_dgemm (int lda, double* A, double* B, double* C)
{
```

```
30    /* For each block-row of A */
31    for (int i = 0; i < lda; i += BLOCK_SIZE)
32      /* For each block-column of B */
33      for (int j = 0; j < lda; j += BLOCK_SIZE)
34        /* Accumulate block dgemms into block of C */
35        for (int k = 0; k < lda; k += BLOCK_SIZE){
36        /* Correct block dimensions if block "goes off edge of" the matrix */
37          int M = min (BLOCK_SIZE, lda-i);
38          int N = min (BLOCK_SIZE, lda-j);
39          int K = min (BLOCK_SIZE, lda-k);
40          /* Perform individual block dgemm */
41          do_block(lda, M, N, K, A + i + k*lda, B + k + j*lda, C + i + j*lda);
42        }
43  }
```

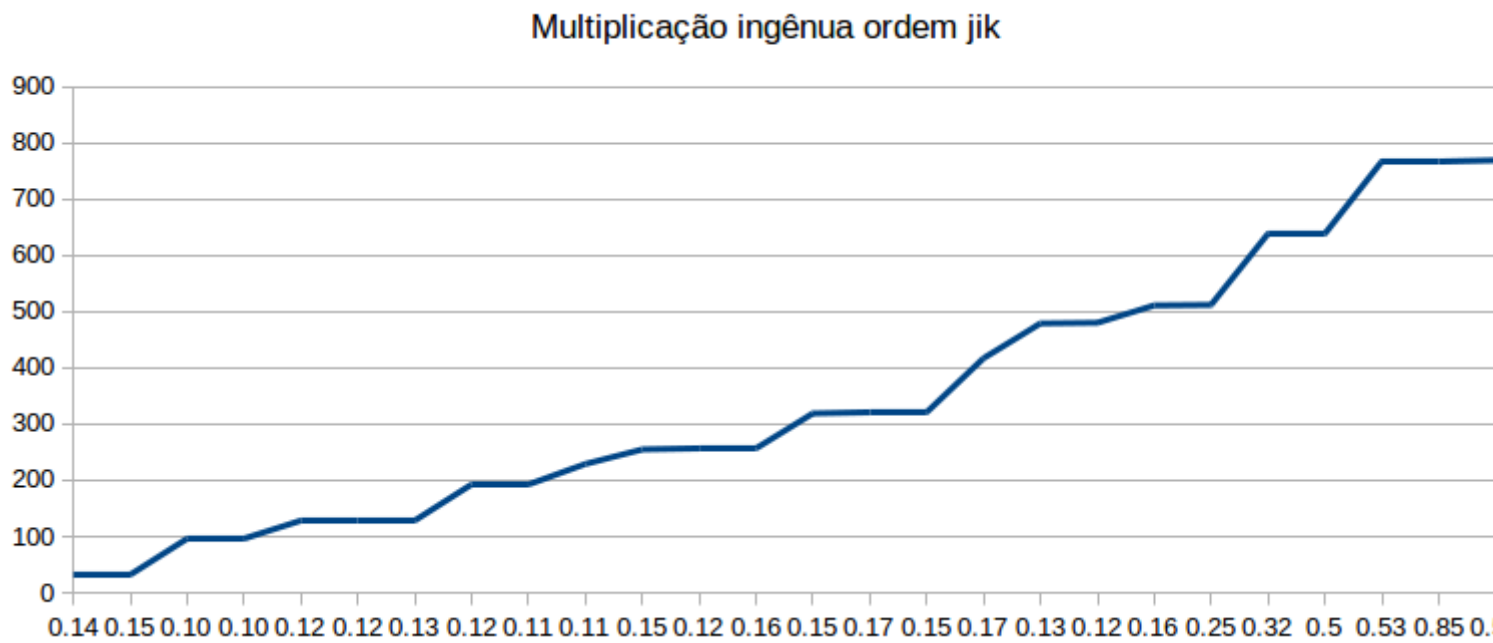
A implementação de blocos na ordem dos laços ijk em relação à melhor implementação ingênua (ordem jki) aumentou em 20% a porcentagem média de Peak.

3 Resultados computacionais

Os testes foram executados no cluster disponibilizado para turma, com processador Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, com 8GB de memória RAM,e com uma cache com tamanho de linha de 32 K. Para medir o desempenho do programa foi utilizado o código disponibilizado para o trabalho, que fazia o calculo de Mfloap/s de operações eram realizados, e a porcentagem do processador que usado na execução do processo. Para medir a otimização do programa, procuramos melhorar a intensidade computacional, que quer dizer dimunir o número busca de dados na mémoria principal. Primeiro fizemos teste dos laços sem a divisão em blocks. A média dos peaks encontrados para as ordens ijk, jik e jki para tamanhos de matrizes variados de múltiplos de 32 +- 1 até 769 foram respectivamente, 9.39089, 8.97356 e 10.0315.

O desempenho de cada combinação dos laços referentes as variâveis ijk pode ser visto graficamente em 1, 2, 3.

Figure 1: Gráfico de Tamanho x Tempo de multiplicação de matrizes ingênua com ordem dos laços j,k,i



Depois foi usado a melhoria da divisão por blocos, a tabela 1 mostra o comparativo entre testes feitos com trocas de laços. A média dos peaks encontrados para as ordens ijk, jik e jki foram respectivamente, 8,785, 10,9825

Figure 2: Gráfico de Tamanho x Tempo de multiplicação de matrizes ingênua com ordem dos laços i,j,k

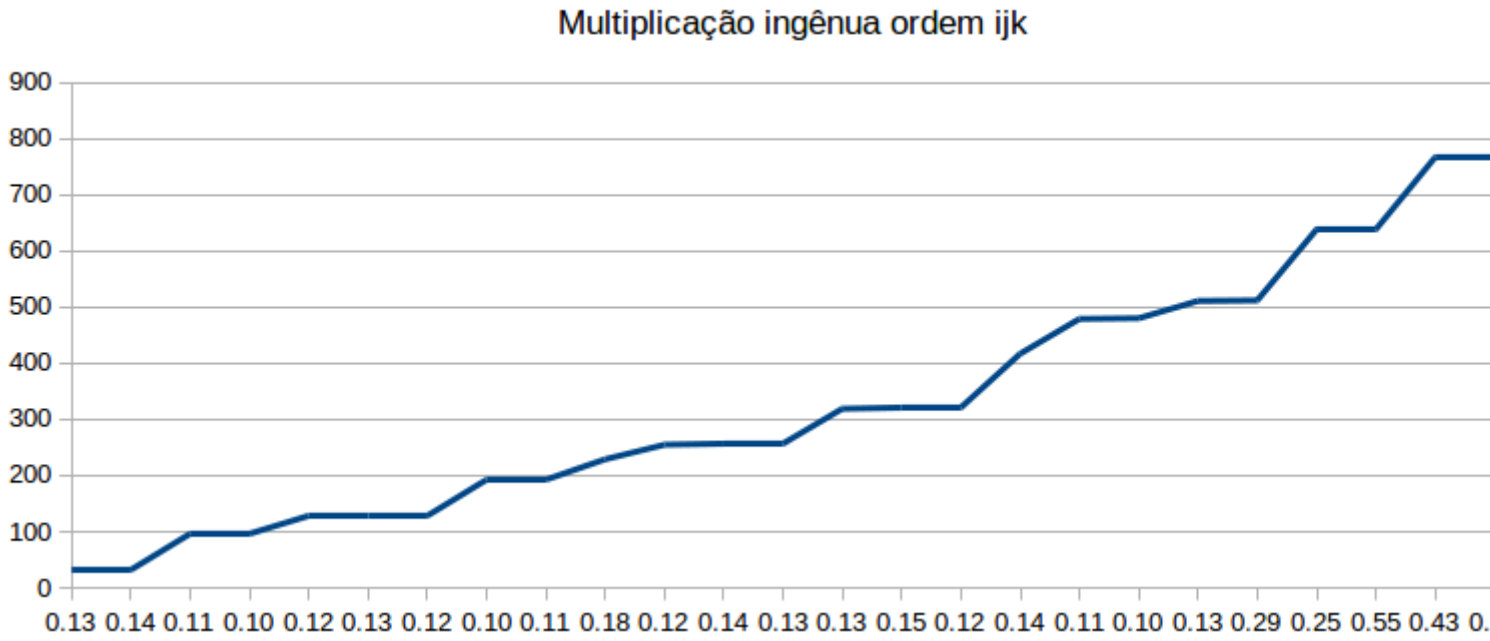
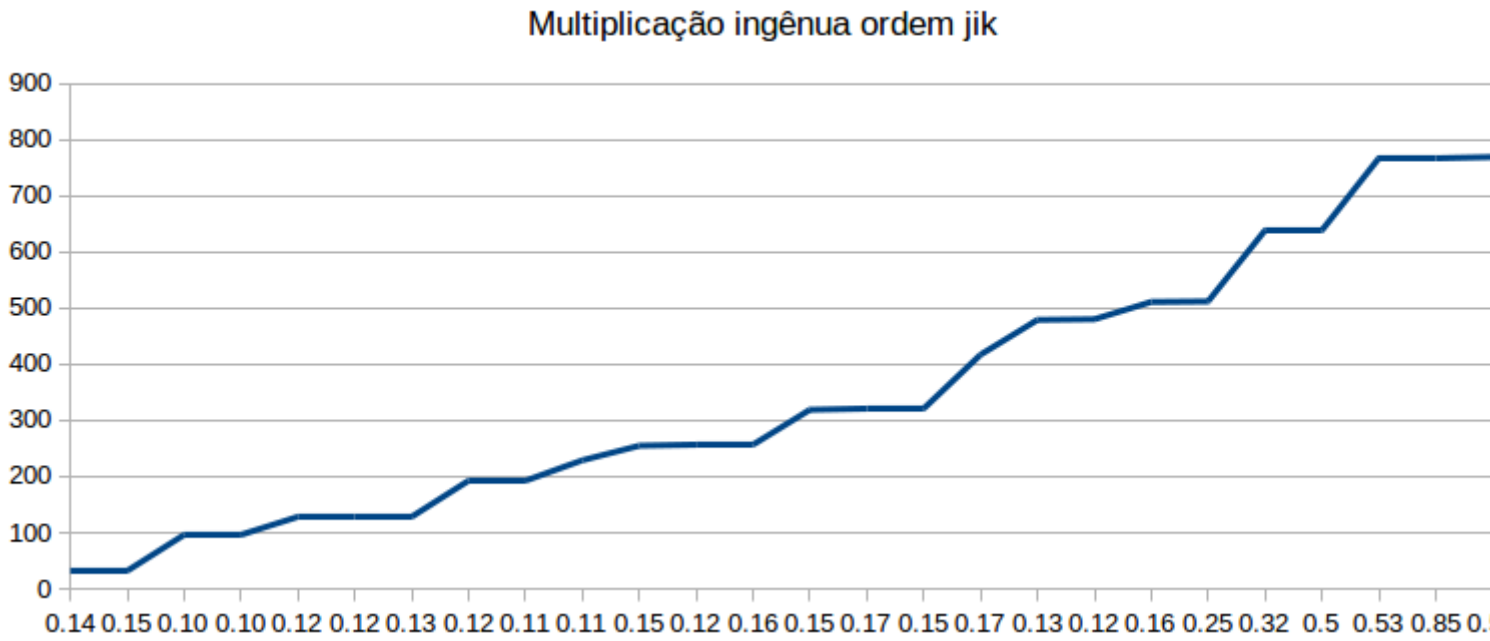


Figure 3: figura: Gráfico de Tamanho x Tempo de multiplicação de matrizes ingênua com ordem dos laços j,k,i



e 7,055, para tamanhos 400,1600,3600,6400. Para os tamanhos variados de 32+-1, a média deles foi: 12.0649 (ijk), 12.4942(jik) e 7.79559(jki).

Table 1: Tamanho de entradas e tempo de execução para ordens de laços diferentes

Tamanho	Tempo ijk	Tempo jik	Tempo jki
400	0.17	0.13	**0.11**
1600	4.57	**3.54**	5.75
3600	48.05	**39.33**	66.17
6400	354.25	**259.72**	358.24

Observamos que o ordem de laços jik obteve melhor tempo de execução para os tamanhos de matrizes 1600,3600 e 6400. Isso se deve ao fato de preenchermos a matriz resultante em colunas, o que acarreta em menos falhas de cache já que a matriz está armazenada em column-major.

4 Conclusão

Para fazer a otimização de multiplicação de matrizes em um código linear, a melhor estratégia vista foi a divisão da matriz em blocos, mas depois disso as melhores opções encontradas para otimização foram a troca da ordem de leitura das matrizes. A partir dos testes feitos, foi encontrado que a melhor combinação seria jik.