TRABALHO PRÁTICO 0:
Produto de Kronecker

Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

27 de setembro de 2012

Resumo. *Esse relatório descreve como foi solucionado o problema de calcular o produto de Kronecker, também conhecido como produto tensorial ou produto direto. Será descrito também a modelagem do problema e a solução proposta para tal. Finalmente será detalhado a análise de complexidade dos algoritmos, os testes utilizados para comprovar tais análises e uma breve conclusão do trabalho implementado.*

1. INTRODUÇÃO

O objetivo do trabalho é implementar um programa que receba várias instâncias de matrizes e retorne o produto de Kronecer de cada instância.

O produto de Kronecker, também conhecido como produto tensorial ou produto direto, consiste em uma operação entre duas matrizes de tamanhos abritrários resultando em uma matriz de bloco. Em outras palavras, essa matriz resultante pode ser seccionada em submatrizes.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve como foi feita a modelagem e manipulação das matrizes. A Seção 3 descreve rapidamente qual foi a solução proposta além do método utilizado para gerar o produto tensorial entre as instâncias de matrizes. A Seção 4 trata de detalhes específicos da implementação do trabalho: quais os arquivos utilizados; como é feita a compilação e execução; além de detalhar o formato dos arquivos de entrada e saída. A Seção 5 contém a avaliação experimental, quantificando o tempo de execução de cada operação com matrizes de diversos tamanhos. A Seção 6 conclui o trabalho.

2. MODELAGEM

Inicialmente, para trabalhar com matrizes, foi criada uma estrutura que contém a informação de quantas linhas e colunas a matriz tem, além de conter todos os elementos da matriz.

```
struct Matriz
    int col, lin;
    int ** matriz;
 Matriz;
```

A complexidade dessa estrutura pode ser considerada como $O(n)$, sendo n o tamanho da matriz bidimensional, ou seja, o produto entre a quantidade de linhas e colunas que ela possui.

Como foi detalhado na especificação, a matriz foi modelada para ser alocada e desalocada dinâmicamente, através dos comandos *malloc* e *free*. Para confirmar que este processo de dinâmico de memória estava ocorrendo como esperado, foi utilizado o comando *valgrind* para verificar qualquer tipo de vazamento de memória.

Além disso, vários procedimentos foram criados para manipular as matrizes em um módulo específico. Essas funções são:

criaMatriz Cria a matriz
mallocaMatriz Maloca todas as linhas e colunas da matriz
destroiMatriz Dá free em todos os vetores alocados pela matriz
leMatrizes Lê todas as matrizes do arquivo e as insere em um vetor de matriz
preencheMatriz Preenche a matriz m com os valores da matriz do arquivo.
imprimeMatriz Imprime todos os valores da matriz
imprimeMatrizNoArquivo Imprime todos os valores da matriz no arquivo

3. SOLUÇÃO PROPOSTA

A solução proposta aqui foi varrer o arquivo de entrada e armazenar todas as matrizes em um array. Posteriormente, esse array de matrizes é percorrido e, então, é calculado o produto de Kronecker entre cada instância. Por último, essas matrzies resultado são escritas no arquivo.

Foi criado um módulo apenas para o produto de Kronecker. Este módulo possui apenas um procedimento, o **produtoKronecker**, que, efetivamente, calcula o produto tensorial entre duas matrizes e insere o resultado em um array.

3.1. Algoritmos

3.1.1. Kronecker

O produto de Kronecker consiste em uma operação entre duas matrizes de dimensões arbitrárias que resulta em uma matriz de bloco. A ideia do algoritmo é que cada elementro da matriz **A** multiplique todos os elementos da matriz **B** gerando diversos blocos, que, eventualmente, irão gerar a matriz de blocos **C**.

A seguir uma definição do produto de Kronecker [Schafer 1996]:

Seja **A** uma matriz *m x n* e **B** uma matriz *p x q*, o produto de Kronecker **C = A** $\bigotimes$ **B**, também conhecido como produto tensorial, é uma matriz $(mp) * (nq)$, a qual elementos são definidos como:

$c_{\alpha\beta} = a_{ij}b_{kl}$,

onde

$\alpha = p(i-1) + k$

$\beta = q(j-1) + l$.

Uma única diferença em relação à definição do produto de Kronecker de [Schafer 1996], foi a posição do resultado na matriz **C** do produto de dois valores das matrizes **A** e **B**. Como na nossa situação, o primeiro elemento se encontra na posição *(0,0)*, e não *(1,1)*, como esperado, então a posição do elemento na matriz **C** teve de ser alterada para o seguinte:

$$c_{\alpha\beta} = a_{ij}b_{kl},$$

onde

$$\alpha = (p * i + k) - 1$$

$$\beta = (q * j + l) - 1.$$

O algoritmo percorre as linhas e colunas da matriz **A** e da matriz **B** para realizar todas as operações. Então, podemos definir que sua complexidade terá um limite superior de $O(n^4)$.

4. IMPLEMENTAÇÃO

4.1. Código

4.1.1. Arquivos .c

- **tp0.c** Arquivo principal do progrma, lê todas as instâncias de matrizes do arquivo de entrada, realiza o produto de Kronecker e insere cada resultado no arquivo de saída.
- **matriz.c** Contém todas as funções de manipulação, leitura e escrita de matrizes.
- **kronecker.c** Contém a função que efetivamente calcula o produto de Kronecker.
- **arquivos.c** Um tipo abstrado de dados de manipulação de arquivos, contendo funções de abertura, leitura, escrita e fechamento.

4.1.2. Arquivos .h

- **matriz.h** Biblioteca que define as funções relativas à matrizes, além de definir a estrutura que é utilizada a todo momento.
- **kronecker.h** Biblioteca que define a função que calcula o produto de Kronecker.
- **arquivos.h** Definição da das funções utilizadas para ler, escrever e fechar corretamente um arquivo.

4.2. Compilação

O programa deve ser compilado através do compilador GCC através de um makefile ou do seguinte comando:

```
gcc -Wall -Lsrc src/tp0.c src/matriz.c src/kronecker.c src/arquivos.c -o tp0
```

4.3. Execução

A execução do programa tem como parâmetros:

- Um arquivo de entrada contendo várias instâncias de matrizes.
- Um arquivo de saída que irá receber o resultado do produto de Kronecker de cada instância de matriz.

  O comando para a execução do programa é da forma:

```
./tp0 <arquivo_de_entrada> <arquivo_de_saída>
```

4.3.1. Formato da entrada

A primeira linha do arquivo de entrada contém o valor *k* de instâncias (pares de matrizes) que o arquivo contém. A próxima linha contém as dimensões *m* e *n* da matriz A_1. As próximas *m* linhas contém os elementos de cada linha de de A_1 separados por um espaço. Em seguida, as dimensões e os elementos da matriz B_1 são especificadas da mesma forma. E assim sucessivamente nas instâncias seguintes.

A seguir dois pares de matrizes de exemplo:

$$A_1 = \begin{bmatrix} 3 & 7 & 4 \end{bmatrix} \quad B_1 = \begin{bmatrix} 0 \\ 7 \\ 6 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 6 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$$

Esse arquivo de entrada tem a seguinte configuração:

```
2
1 3
3 7 4
3 1
0
7
6
1 1
6
2 1
6
6
```

4.3.2. Formato da saída

O arquivo de saída tem a mesma configuração, sendo a primeira linha o valor *k*, seguido das *k* matrizes.

A seguir o resultado do produto de Kronecker das instâncias de matrizes definidas anteriormente:

$$C_1 = \begin{bmatrix} 0 & 0 & 0 \\ 21 & 49 & 28 \\ 18 & 42 & 24 \end{bmatrix} \quad C_2 = \begin{bmatrix} 36 \\ 36 \end{bmatrix}$$

Esse arquivo de saída tem a seguinte configuração:

```
2
3 3
0 0 0
21 49 28
18 42 24
2 1
36
36
```

5. AVALIAÇÃO EXPERIMENTAL

Foram gerados arquivos de vários tipos para testar, depois foi usado o comando *time* para calcular o tempo de execução de cada configuração de teste diferente. Nas próximas duas subseções vamos detalhar os scripts que foram criados para gerar esses arquivos de testes, além de demonstrar os resultados de cada teste.

5.1. Scripts

Primeiro, foi criado um script, em Python, para gerar arquivos de entrada para o trabalho. Esse programa recebe como parâmetro o número de instâncias a ser criado e o número máximo de linhas e colunas que essas matrizes terão.

Posteriormente foi feito outro script para gerar um lotes de arquivos de entrada e já armazenar o tempo de processamento de cada um.

Primeiro, o script que gera o arquivo de entrada:

```
import random,sys

k = int(sys.argv[1]) # pares de matrizes
maxX = int(sys.argv[2]) # tamanho máximo de linhas da matriz
maxY = int(sys.argv[3]) # tamanho máximo de colunas da matriz

print k

for i in range(k)*2: # Multiplica por 2 pois são pares de matrizes
    m = maxX
    n = maxY
    print m,n
    for j in range(m):
        for l in range(n):
            print random.randint(0,100),
        print ''
```

Segundo, o script que utiliza o primeiro para fazer um *batch* de testes:

```
import sys,os,random

ent = 'entrada/'
sai = 'saida/'
tempo = 'tempo/'
script = './testes/entrada.py '
arqAppend = ' >> '
arqNew = ' > '


for i in range(100):
  i+=1
  i*=5
  #teste1 = script+' 1 '+str(i)+' '+str(i)+' ' # Matrizes quadradas
  #teste2 = script+' 1 10 '+str(i) # linhas fixas em 10
  #teste3 = script+' 1 '+str(i)+' 10' # colunas fixas em 10
  teste = script+' '+str(i)+' 50 50' # linhas e colunas fixas em 50 +
                                     nº de instâncias variando
  arqEnt= 'ent'+str(i)+'.txt'
  arqSai= 'sai'+str(i)+'.txt'
  arqTmp= 'tmp'+str(i)+'.txt'
  os.system(teste+arqNew+ent+arqEnt)
  os.system('/usr/bin/./time -o '+tempo+arqTmp+' ./tp0 '+arqEnt+' '+
                                     arqSai)
```

5.2. Resultado

O primeiro teste realizado foi o seguinte: foi fixado o número de linhas e colunas das matrizes para 50. Ou seja, cada matriz tinha 2.500 elementos, e as matrizes resultado tinham $6.250.000$ elementos cada uma.

Na Figura 1, podemos ver que o tempo de processamento cresceu de uma maneira esperada. Como a complexidade do algoritmo é definida basicamente pelo tamanho das matrizes, então uma variação do número de instâncias resulta em um crescimento praciamente constante no tempo de processamento do programa.

Na Figura 2 foi feito um teste com matrizes quadradas. Tanto a matriz **A**, quanto a matriz **B** variaram o tamanho de $1 * 1$ para $100 * 100$. Ou seja, num momento tinham apenas um elemento e no último tinham 10.000 elementos. A matriz **C**, no final desse teste, possuía $100.000.000$ elementos. O tempo de processamento aqui já cresceu de maneira exponencial, como era de se esperar por causa da complexidade exponencial do produto de Kronecker.

Os últimos dois testes foram configurados da seguinte maneira: fixamos as colunas e variamos as linhas e vice-versa.

No primeiro, Figura 3, fixamos as colunas das matrizes em 10 e variamos as linhas até 500, ou seja, a matriz **C** começou com tamanho $1 * 100$ (100 elementos) e terminou com um tamanho de $250.000 * 100$ ($25.000.000$ elementos); no segundo, Figura 4, fixamos as linhas das matrizes em 10 e variamos as colunas até 500. O tempo de processamento desses testes se encontra nas figuras abaixo:

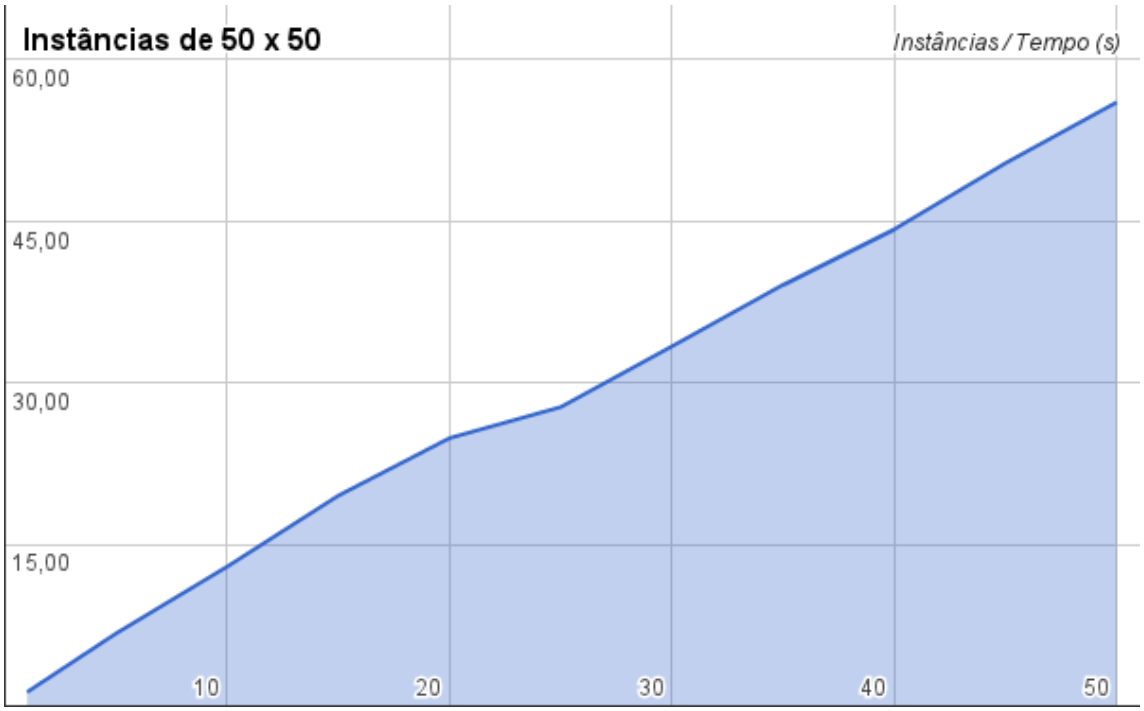

Figura 1. A(50,50) B(50,50)

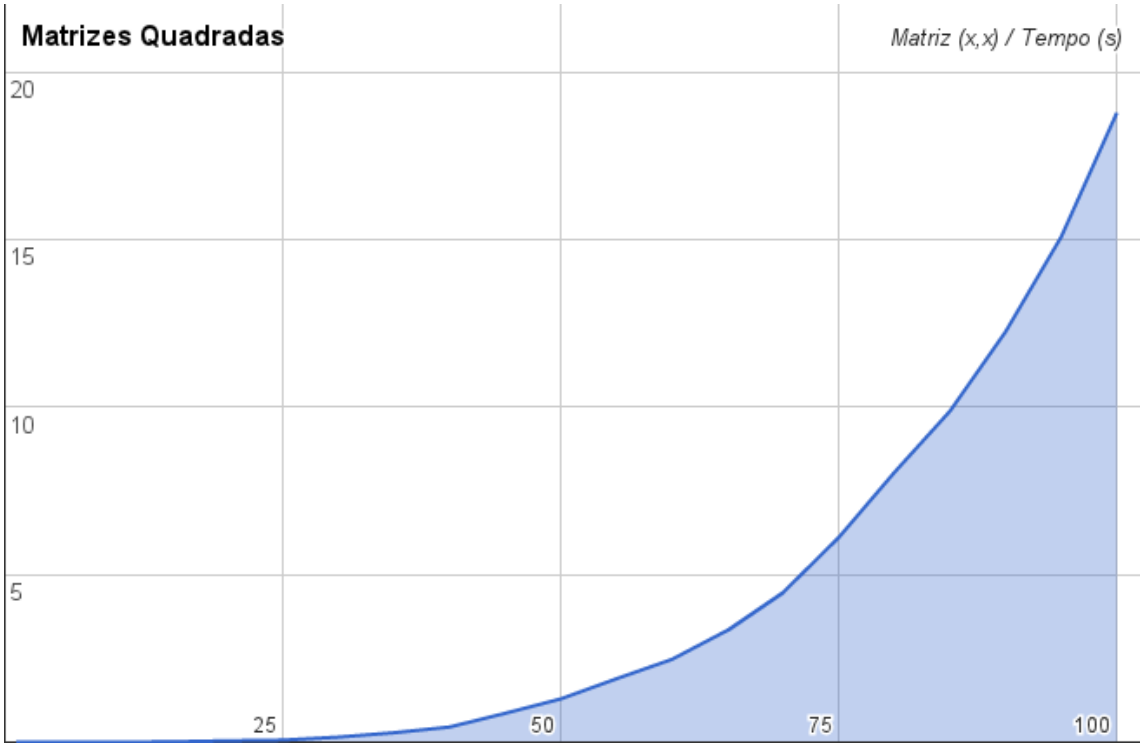

Figura 2. Matrizes Quadradas A(x,x) B(x,x)

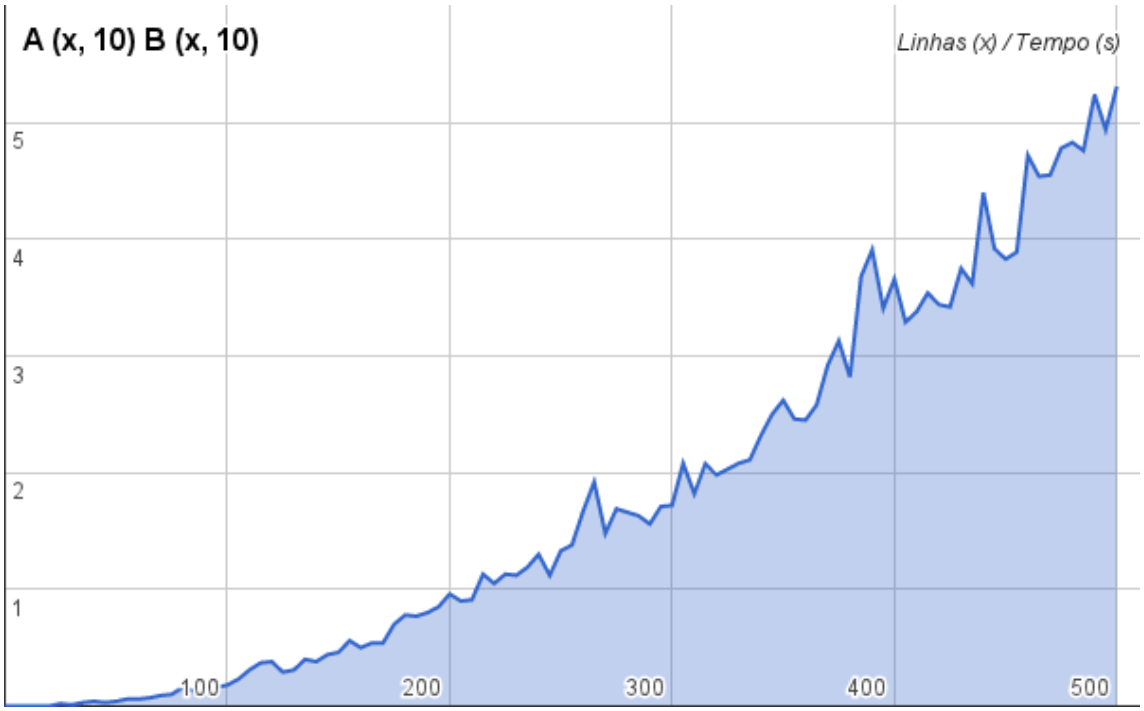

Figura 3. Linhas - A(x,10) B(x,10)

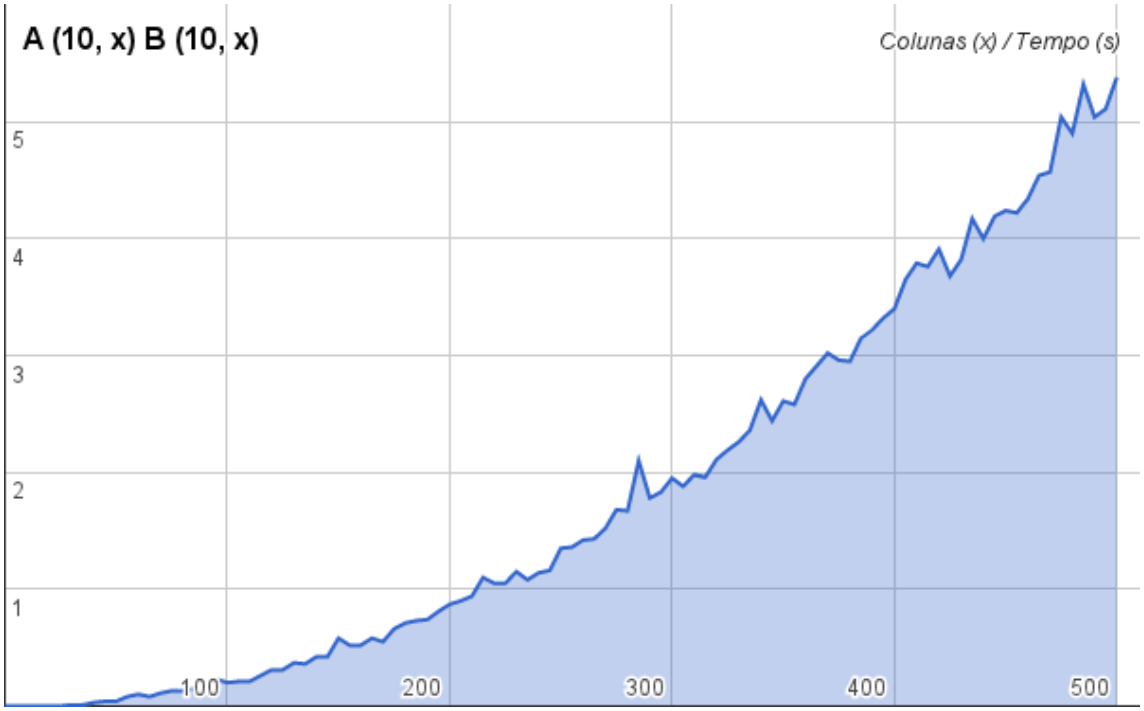

Figura 4. Colunas - A(10,x) B(10,x)

6. CONCLUSÃO

O problema de calcular o produto de Kronecker das matriz foi um problema sem muitas complicações. As matrizes foram alocadas dinâmicamente com sucesso, de acordo com o *Valgrind*. Além disso, um padrão de código bem modularizado foi seguido, para que os módulos possam ser reutilizados futuramente.

O código também já foi construído de uma maneira em que, caso futuramente seja necessário paralelizá-lo, poucas mudanças no código precisarão ser feitas para tal.

Os testes foram bem sucedidos pois o tempo de processamento do algoritmo em diversas situações ocorreu exatamente como esperado.

As primitivas básicas da linguagem C, como alocação dinâmica de memória e manipulação de arquivos, foram bem exploradas no trabalho prático. Além disso, procurei construir uma documentação bem detalhada sobre toda a estrutura e funcionamento do programa construído.

Como primeiro trabalho da disciplina, acredito que seus objetivos foram cumpridos.

Referências

Schafer, R. D. (1996). *An Introduction to Nonassociative Algebras.*