UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

BRENO LEITE
RA - 192863

A criação de um algoritmo híbrido para a multiplicação de cadeias de matrizes utilizando OpenMP e CUDA

PROJETO FINAL

CAMPINAS
2017

1 INTRODUÇÃO

Muitos problemas na área da computação e matemática utilizam de alguma forma a multiplicação de matrizes. Temos exemplos dentro da álgebra, como as cadeias de Markov, onde uma determinada cadeia pode representar probabilidades de mudanças entre estados de um determinado objeto em um período de tempo (SANTOS, 2002). A multiplicação de matrizes também é muitas vezes utilizada para a solução de sistemas lineares, dentre outras inúmeras utilidades no meio científico.

O grande problema é que a multiplicação dessa cadeia de matrizes pode ser muito cara computacionalmente, por esse motivo este trabalho tem como intuito apresentar técnicas de paralelização para multiplicar cadeias de matrizes. Neste trabalho serão utilizados paradigmas de programação paralela, juntamente com ferramentas como OpenMP e CUDA para a criação de um algoritmo paralelo híbrido para resolver o problema de multiplicação de cadeias.

2 O PROBLEMA

Como o custo computacional de multiplicar cadeias de matrizes é normalmente elevado, tem-se um esforço para descobrir métodos mais eficazes de fazê-lo. Sabe-se que a multiplicação de duas matrizes $A_{(n\times r)}$ e $B_{(r\times m)}$ gera uma matriz $C_{(n\times m)}$, onde n e m são as dimensões da nova matriz. O número de operações necessárias para multiplicar as matrizes $A_{(n\times r)}$ e $B_{(r\times m)}$ é dado pela equação $n.m.r$, portanto, é totalmente dependente da dimensão das mesmas.

A multiplicação de matrizes é uma operação associativa, ou seja, as matrizes $A\ B\ C$ podem ser multiplicadas em ordens diferentes e mesmo assim obter o mesmo resultado. Existe uma ordem de multiplicação das matrizes de tal forma que o número de operações escalares de multiplicação é mínimo. A Tabela 1 mostra um exemplo de diferenciação na ordem utilizando a seguinte cadeia de matrizes: $A_{(5\times 2)}\ B_{(2\times 2)}\ C_{(2\times 2)}$.

Parentização	Custo
$((AB)\ C)$	$5.2.2 + 5.2.2 = 40$
$(A\ (BC))$	$5.2.2 + 2.2.2 = 28$

Tabela 1 – Diferentes parentizações e seus custos em operações escalares.

Podemos perceber que a parentização $(A\ (BC))$ possui o custo mínimo de operações escalares. A diferença entre o mínimo e os outros resultados pode ser muito significante quando o número de matrizes da cadeia cresce. Devido a isto, encontrar a melhor parentização é uma tarefa muito importante na multiplicação de cadeias de matrizes.

Neste trabalho iremos paralelizar um algoritmo que utiliza conceitos de programação dinâmica para encontrar a melhor parentização para uma determinada cadeia

de matrizes, este algoritmo é explicado no capítulo 15.2 do livro do Cormen (CORMEN, 2009). O mesmo tem complexidade de tempo $O(n^3)$, onde n é o número de matrizes de uma determinada cadeia.

Além de encontrar a melhor cadeia, este trabalho também tem como objetivo multiplicar as matrizes da cadeia de forma paralela. Esse processo não é tão complexo, porém tem um alto custo de memória.

3 PARALELIZAÇÃO

Este capítulo tem como objetivo demostrar quais técnicas de paralelização foram aplicadas, assim como as mesmas foram implementadas. O capítulo será dividido em três partes, onde serão explicados o *profiling*, a paralelização para encontrar a melhor parentização e a paralelização da multiplicação da cadeia de matrizes.

3.1 PROFILING

Para o *profiling* do programa foi utilizado a ferramenta *gprof*, o *profiling* do algoritmo serial não foi uma tarefa muito difícil pois o código é bem pequeno e fácil de encontrar os *HotSpots*.

O importante do *profile* foi notar que dependendo do número de matrizes e suas dimensões o algoritmo podia ter um *HotSpot* diferente, ou seja, quando havia um grande número de matrizes o tempo gasto para encontrar a melhor parentização era o maior. Já quando o número de matrizes era pequeno e as dimensões das mesmas eram grandes, o tempo de multiplicar a cadeia de matrizes era o mais longo.

Outro fato importante percebido é que para entradas com o número de matrizes e dimensões similares o custo de multiplicar as matrizes é maior, o motivo disto é o alto número de *cache misses* na multiplicação das matrizes. Neste trabalho não vamos nos preocupar com esse problema, mas já existem diversos estudos no meio cientifico relatando esse problema, assim como propondo soluções (LAM; ROTHBERG; WOLF, 1991). Uma solução simples é utilizar a matriz transposta no momento da multiplicação, porém, o custo para transformar essa matriz também teria que entrar no computo do tempo para verificar o quanto essa forma resolveria o problema. Alguns trabalhos tratam esse problema com o uso de uma *shared memory*, neste trabalho não aplicamos nenhuma dessas técnicas.

Devido a essa dependência do *HotSpot* com a entrada, foi decidido que ambos os processos deviam ser paralelizados. Ou seja, este trabalho trata da paralelização do algoritmo para encontrar a melhor parentização de determinada cadeia de matrizes, assim como a paralelização da reconstrução dessa parentização e sua multiplicação.

3.2 PARENTIZAÇÃO ÓTIMA DE UMA CADEIA DE MATRIZES

Conceitos de programação dinâmica são utilizados para resolver o problema de encontrar a melhor parentização para determinada cadeia de matrizes, desta forma, uma tabela é construída para encontrar a solução. Essa matriz é preenchida pelas diagonais,onde cada diagonal representa uma parentização diferente para cada item. A Figura 1 mostra um exemplo de uma matriz utilizada na solução do problema.

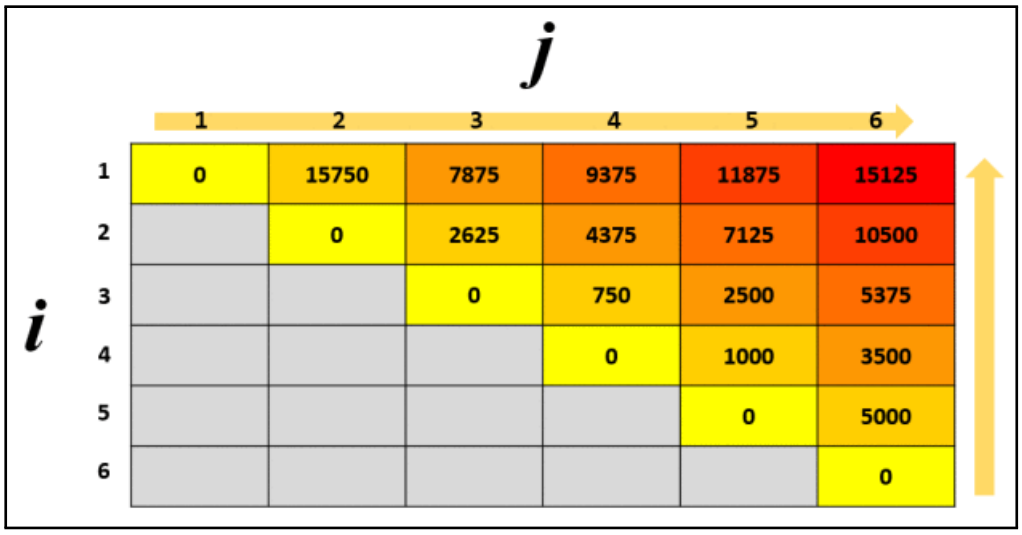

Figura 1 – Matriz de programação dinâmica para o problema
Fonte – Yong (2009)

Inicialmente a matriz começa com a sua diagonal preenchida com zeros, e a cada iteração do algoritmo uma nova diagonal é preenchida utilizando dos resultados da diagonal anterior. Mais informações sobre o funcionamento do algoritmo podem ser encontradas no capítulo 15.2 do livro do Cormen (2009).

A ferramenta OpenMP foi utilizada para paralelizar este algoritmo, a ideia da paralelização é aproveitar que as diagonais não tem dependência entre sí. Ou seja, o *loop* das diagonais é *DOALL*. Desta forma, cada diagonal é processada em paralelo iterativamente.

É interessante notar que o número de elementos em cada diagonal diminui em cada interação, o próprio OpenMP toma conta deste ajuste quando chamado dividindo o número de elementos do *loop* pelo número de *threads* utilizadas. Assim que o algoritmo começa uma *pool* de *threads* é criada e para cada iteração do algoritmo os elementos são divididos entre as *threads*. Essa *pool* tem como intuito reduzir o *overhead* de criar *threads* a cada iteração.

3.3 MULTIPLICAÇÃO DE CADEIA DE MATRIZES

A parelização desta parte do problema foi feita utilizando CUDA, o motivo desta escolha é que algoritmos em CUDA para multiplicação de matrizes são am-

plamente conhecidos. Porém, apesar de existir diversos algoritmos para multiplicar matrizes em CUDA, a grande maioria deles são projetados para multiplicar duas matrizes e não uma cadeia de matrizes.

Neste trabalho além da multiplicação da matriz ser paralela, a multiplicação de toda a cadeia de matrizes também é feita em paralelo. Isso significa que encontrada uma melhor cadeia da seguinte forma $((AB)(CD))$ a multiplicação das matrizes AB é feita em paralelo com a multiplicação das matrizes CD.

Cada uma das multiplicações é feita em um *kernel* CUDA, e esse mesmo *kernel* é chamado em várias vezes em sequência para todas as matrizes que não tem dependência de outros resultados. Os *kernels* utilizam blocos de 16×16 e o número *threads* de cada bloco depende da dimensão da matrizes. Essa configuração foi utiliza pois foi a mais comum encontrada na literatura. Além de que essas configurações são boas para a GPU que foi utilizada para os teste, que foi a NVIDIA Kepler K40 do servidor parsusy.

A forma com que duas matrizes são multiplicadas utilizando CUDA é trivial e não vai ser explicada neste trabalho, mais informações sobre esse processo podem ser vistas em Hochberg (2012).

A dependência entre as multiplicações das matrizes pode ser representada por uma árvore, a Figura 2 mostra um exemplo desta árvore para a seguinte cadeia: $(A(BC)(DE))$.

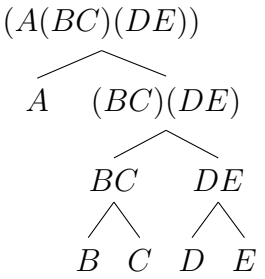

Figura 2 – Árvore de dependências

O algoritmo executa essa árvore da direita para a esquerda e das folhas para a raiz. Para cada nível da árvore n *kernels* CUDA são chamados, e então o algoritmo aguarda todos eles serem resolvidos para passar para o próximo nível da árvore.

Cada matriz é copiada apenas uma vez para a GPU, a única resposta que é copiada de volta para a CPU é a resposta final. Ou seja, no exemplo da Figura 2 as matrizes D e E são copiadas para a GPU, mas a matriz DE não é copiada de volta para a CPU, apenas o resultado final $(A(BC)(DE))$ é copiado de volta para a CPU quando o último nível da árvore é atingido.

4 METODOLOGIA

Este capítulo tem como intuito mostrar a metodologia para os testes feitos neste trabalho, explicando as entradas utilizadas e as formas com que os mesmos foram analisados durante o desenvolvimento do trabalho.

Primeiramente a Tabela 2 define os arquivos de entrada que foram utilizados para os testes.

Arquivo	**Nº de matrizes**	**Dim das matrizes**	**Arquivo**	**Nº de matrizes**	**Dim das matrizes**
arq1.in	1000	50 - 100	arq4.in	500	50 - 100
arq2.in	2000	50 - 100	arq5.in	500	100 - 150
arq3.in	3000	50 - 100	arq6.in	500	150 - 200
Grupo 1			**Grupo 2**		

Tabela 2 – Descrição dos arquivos de entrada utilizados nos testes.

Os arquivos foram divididos em dois grupos, o Grupo 1 são arquivos onde o número de matrizes da cadeia é variado, enquanto as dimensões das matrizes variam entre 50 e 100. Já o Grupo 2 mantém o número de matrizes e varia as dimensões das matrizes, nota-se que a dimensão exata das matrizes são geradas aleatoriamente e 50-100 significa que a matriz de menor dimensão tem 50 e a de maior tem 100.

Esses arquivos serão testados de três formas diferentes, uma comparando somente o *speedup* em relação a encontrar a melhor parentização. Outra levando em consideração que já temos a melhor parentização e comparando o resultado apenas da multiplicação da cadeia, e por ultimo a junção de ambos em um mesmo algoritmo.

Para os testes serão utilizados 2, 4 e 8 *threads*, é importante perceber que o número de *threads* não influencia no algoritmo de multiplicação da cadeia, o mesmo influência apenas no algoritmo de encontrar a melhor parentização.

Todos os tempos foram adquiridos utilizando o servidor parsusy, obtendo uma média de 10 execuções para cada caso de teste.

5 RESULTADOS

Este capítulo tem como objetivo mostrar os resultados obtidos com o algoritmo paralelo criado neste trabalho.

5.1 PARENTIZAÇÃO ÓTIMA DE UMA CADEIA DE MATRIZES

Nesta seção será mostrado os resultados da parte do algoritmo que é responsável em encontrar a melhor parentização possível para determinada cadeia de matrizes. Ou seja, esta seção analisa exclusivamente como o OpenMP se saiu em relação ao algoritmo serial. A Figura 3 mostra a variação do *SpeedUp* em relação ao número de *threads*.

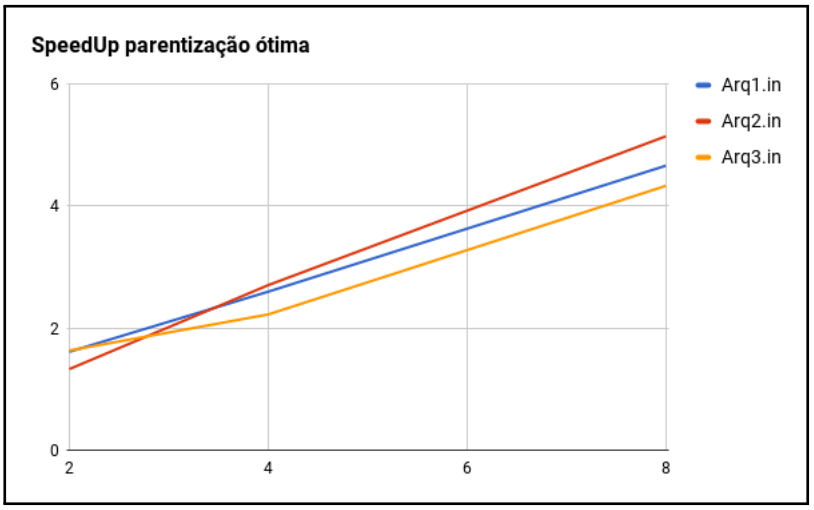

Figura 3 – Variação do *SpeedUp* pelo número de *threads*

A Figura 3 mostra que o algoritmo obteve um resultado de cerca entre 2 e 5 de *SpeedUp*, dependendo da entrada. Percebe-se que o aumento do número de *threads* aumenta o desempenho. Esse aumento continua até 8 *threads*, depois disso o *SpeedUp* atinge seu patamar máximo e começa a reduzir.

Um fato interessante, é que o aumento do número de matrizes não faz com que o *SpeedUp* aumente. Podemos ver no gráfico da Figura 3 que o arquivo 3 teve o menor *SpeedUp*, valores maiores que estes não foram testados por motivos que serão descritos no capítulo relacionado aos problemas encontrados.

5.2 MULTIPLICAÇÃO DE UMA CADEIA ÓTIMA DE MATRIZES

Esta seção tem como objetivo mostrar o desempenho da multiplicação feita na parentização ótima encontrada pelo algoritmo mostrado na seção anterior. Nesta parte vamos comparar apenas os resultados da multiplicação, na próxima seção será analisado o funcionamento dos dois algoritmos em conjunto. A Figura 4 faz uma comparação do tempo em segundos consumido pela GPU para executar a multiplicação utilizando a parentização ótima para as matrizes dos arquivos da Tabela 2.

Pode-se perceber para a multiplicação da cadeia o número de matrizes e a dimensão das mesmas tem uma grande influência no tempo de processamento, conforme esse número cresce a tendencia é que o *SpeedUp* cresça. A Tabela 3 mostra os *SpeedUp* obtidos pelos tempos mostrados na Figura 4.

Arquivo	SpeedUp	Arquivo	SpeedUp
arq1.in	0.3864	arq4.in	11.9539
arq2.in	10.7203	arq5.in	14.0967
arq3.in	10.997	arq6.in	49.3169

Tabela 3 – *SpeedUp* multiplicação de cadeias

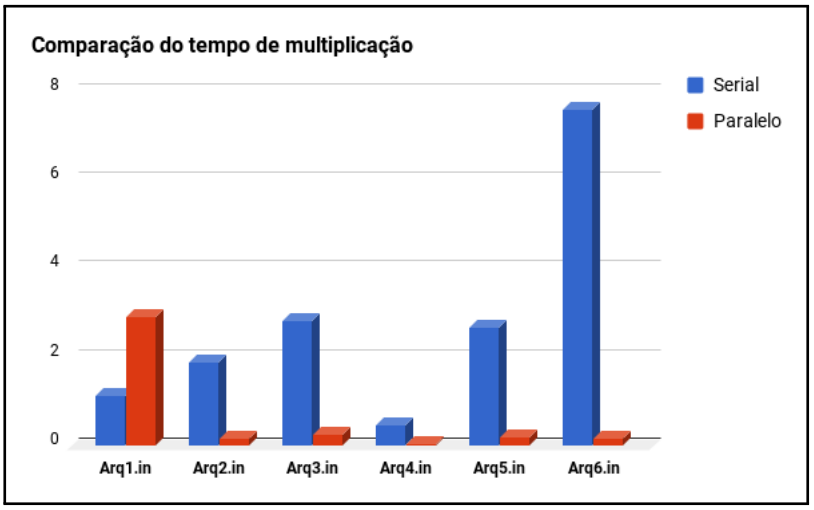

Figura 4 – Tempo em segundos para multiplicar cadeias dos arquivos

Algo importante de se notar é que o *SpeedUp* entre os arquivos 1 e 4 foram muito diferentes, mesmo com os dois tendo a mesma estrutura de arquivos de entrada. O motivo disto é que a forma que a cadeia está no arquivo 1 é mais difícil de ser multiplicada do que no arquivo 4. O número total de multiplicações foram 273731256 e 134559854 para os arquivos 1 e 4 respectivamente.

Como a cadeia é mais difícil, o tempo para percorrer a árvore mostrada na Figura 2 é maior, o que torna o algoritmo paralelo ruim para o arquivo 1. Esse tempo é irrelevante quando o número de multiplicações necessárias cresce muito.

5.3 ALGORITMO HÍBRIDO OPENMP + CUDA

Esta seção irá mostrar os resultados obtidos da junção dos dois algoritmos mostrados anteriormente, a Figura 5 mostra o *SpeedUp* obtido para cada entrada dependendo do número de *threads*.

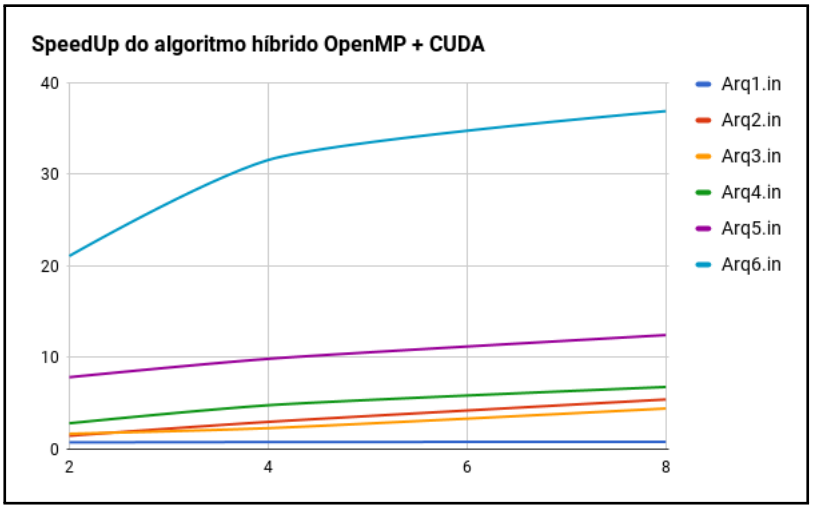

Figura 5 – Algoritmo híbrido para multiplicar cadeia de matrizes

Pode-se perceber que o algoritmo desenvolvido teve um excelente desempenho, dependendo do número de matrizes e da dimensão das mesmas o *SpeedUp* chegou ultrapassar 30 utilizando 8 *threads*.

Percebe-se também que conforme o tamanho do problema aumenta o *SpeedUp* melhora, isso se deve pois o custo do *overhead* é compensado pela solução paralela do problema. Esse aumento é ainda maior quando as dimensões das matrizes são aumentadas, o motivo desta alta melhora é o processamento paralelo tanto da multiplicação dessas matrizes como também a forma com que a parentização e executada em paralelo. Não foi possível verificar até onde esse crescimento acontece, o motivo será explicado no próximo capítulo.

6 DIFÍCULDADES ENCONTRADAS NO DESENVOLVIMENTO

A principal dificuldade encontrada foi a alta demanda por memória de acordo com o aumento do tamanho da entrada. Dado os recursos físicos disponíveis para a execução desse software, não pode-se executar testes em escalas maiores., vários métodos tiveram que ser implementados com intuito de economizar e gerenciar melhor a memória disponível. Outro problema foi a falta de um algoritmo serial já implementado, boa parte do algoritmo serial teve que ser implementado pois ambos os problemas são tratados normalmente separadamente.

7 CONCLUSÃO

O trabalho mostra um algoritmo híbrido que utiliza conceitos de programação paralela, juntamente com ferramentas como OpenMP e CUDA para otimizar um problema muito conhecido na área científica. O algoritmo criado mostrou-se capaz de obter um *SpeedUp* de quase 40 em determinadas condições, além de ter um bom rendimento no geral.

A criação do algoritmo foi um processo muito interessante, onde foi envolvido conceitos aprendidos dentro da sala de aula. Assim como uma mistura desses conceitos para resolver um mesmo problema da melhor maneira possível.

Pode-se perceber também que o número de ferramentas e conceitos aprendidos na disciplina nos proporciona um leque de possibilidades muito útil, e necessário, para resolver problemas que podemos vir a enfrentar no futuro.

REFERÊNCIAS

CORMEN, T. H. **Introduction to algorithms**. [S.l.]: MIT press, 2009.

HOCHBERG, R. **Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model**. 2012. Disponível em: <https://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf>. Acesso em: 11 Ago. 2017.

LAM, M. D.; ROTHBERG, E. E.; WOLF, M. E. The cache performance and optimizations of blocked algorithms. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 1991. v. 19, n. 2, p. 63–74.

SANTOS, R. J. Um curso de geometria analítica e álgebra linear. **Belo Horizonte: Imprensa Universitária**, 2002.

YONG, L. Z. **Matrix Chain Multiplication with C++ code – PART 2: Implementation**. 2009. Disponível em: <https://bruceoutdoors.wordpress.com/2015/11/19/matrix-chain-multiplication-with-c-code-part-2-implementation/>. Acesso em: 12 Jun. 2017.