Técnicas de Otimização Computacional em um Algoritmo de Multiplicação de Matrizes

Sherlon Almeida da Silva[1], Matheus da Silva Serpa [2], Claudio Schepke[1]

[1]Laboratório de Estudos Avançados – Universidade Federal do Pampa (UNIPAMPA)
97546-550, Alegrete – RS – Brasil

sherlonalmeida@alunos.unipampa.edu.br, claudioschepke@unipampa.edu.br

[2]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970, Porto Alegre – RS – Brasil

msserpa@inf.ufrgs.br

Resumo. *Muitas aplicações usam matrizes como forma de representação de dados. O processamento de operações matriciais de grande ordem exige recursos computacionais eficientes. Nesse contexto, o objetivo deste trabalho foi otimizar um algoritmo de multiplicação de matrizes utilizando técnicas de loop como: interchange, unrolling e tiling. Os resultados mostram um ganho de até 259 vezes na versão super otimizada paralela em uma arquitetura de 16 núcleos.*

1. Introdução

Algumas aplicações desenvolvidas a partir de modelos matemáticos exigem muitos recursos computacionais para prover soluções rápidas e precisas. Para obtenção de resultados em tempo aceitável é necessário investir em diferentes técnicas de otimização e processamento. Como o tempo de execução sequencial é alto, pode-se melhorá-lo com a execução paralela usando vários núcleos de processamento. Também a utilização eficaz da memória *cache*, que é uma memória rápida localizada no processador, para facilitar o acesso aos dados uma vez que o acesso à memória principal é mais demorado.

A análise do código e reestruturação do mesmo ajuda a reduzir o tempo de execução. O aproveitamento dos princípios de localidade espacial e temporal podem proporcionar alto ganho de desempenho [Kowarschik and Weiß 2003]. Nesse contexto, o objetivo deste trabalho foi encontrar soluções eficientes para multiplicações de matrizes usando a interface de programação de aplicação *OpenMP* para gerenciar o paralelismo e o ótimo uso de *cache* com a exploração dos princípios de localidade. Foi testado o armazenamento por indexação simples e dupla e a execução nos compiladores **gcc** e **icc**, assim como a escalabilidade através do aumento do número de *threads*.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta as técnicas utilizadas. Na Seção 3 são apresentados detalhes sobre os algoritmos utilizados e os métodos de extração de desempenho. Os resultados experimentais são discutidos na Seção 4. Por fim, a Seção 5 expõe a conclusão deste artigo.

2. Metodologia

Para a análise das otimizações em produtos matriciais foram avaliados os resultados de quatro versões de um algoritmo de multiplicação de matrizes, sendo elas: *Naive*,

Naive+Unrolling, *Tiling* e *Tiling+Unrolling*. Cada algoritmo foi executado na versão sequencial e paralela, também em suas versões contígua (indexação simples) e não-contígua (indexação dupla) e nos compiladores *gcc* e *icc*.

A versão *Naive* é o algoritmo convencional de multiplicação de matrizes, possui três laços de repetição que operam multiplicando as linhas da matriz A pelas colunas da matriz B. A *Naive+Unrolling* corresponde à matriz convencional, porém são calculadas várias partes de elementos por iteração, ganhando com a localidade espacial. A versão *Tiling* fragmenta a matriz original em matrizes menores, facilitando o uso da localidade temporal na memória *cache* pois mantém as submatrizes em *cache* para efetuar a multiplicação, reduzindo o número de *cache misses*. A versão *Tiling+Unrolling* explora os ganhos de desempenho oferecidos pela combinação das duas otimizações avaliadas.

As implementações utilizaram a técnica de *Loop Interchange* para considerar todas as combinações dos laços de repetição. Nas versões *Naive* e *Naive+Unrolling* foram avaliadas as 6 combinações possíveis dos laços **ijk**, responsáveis por controlar a multiplicação. Já para as versões *Tiling* e *Tiling+Unrolling* o número de combinações seria 720 para os laços **ijkxyz**. Neste caso foram selecionadas apenas 12 combinações, sendo a variação de **ijk** e de **xyz**. As combinações **ikj** e **ikjxzy** foram escolhidas para os demais testes pois obtiveram os menores tempos de execução.

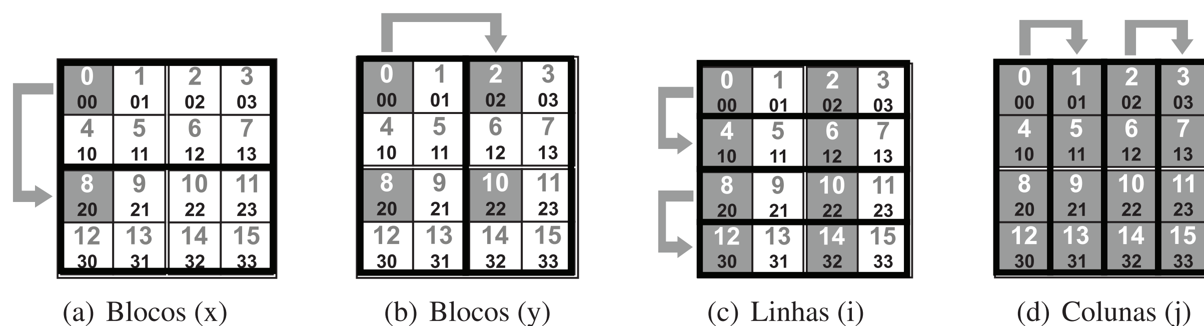
Os resultados apresentados são a média de 15 execuções de matrizes quadradas de tamanho: 512, 1024, 2048 e 4096. Quanto maior a matriz, mais estável o desvio. Para a ordem 4096 o desvio manteve-se abaixo de 5%. Diferentes tamanhos de blocos e desenrolamento foram utilizados. Os tamanhos de blocos escolhidos são potências de base 2, e que cabem totalmente em *cache* L1. Além disso, foram usadas *flags* de otimização do compilador, sendo que a flag -O2 obteve maior redução no tempo da aplicação. O ambiente utilizado foi uma *workstation*, com dois processadores Intel Xeon E5-2650, cada processador tem 8 *cores* físicos, permitindo a execução de 32 *threads* com *Hyper-Threading*. Cada *core* tem uma *cache* L1 privada de 32 KB e uma *cache* L2 privada de 256 KB. O último nível de *cache* é o L3 com 20 MB compartilhados.

3. Técnicas de Otimização

A alta latência de acesso à memória principal é um dos limitadores de desempenho. A memória *cache* serve para minimizar o impacto dessa latência, uma vez que ela armazena dados utilizados pelo processador baseando-se nos princípios de localidade espacial e temporal. Através da modificação de algoritmos convencionais de multiplicação de matrizes para execução em arquiteturas *multi-core*, a análise de distribuição dos dados à nível hierárquico, como a memória *cache*, viabiliza o ganho de desempenho para esta arquitetura [Jacquelin et al. 2009].

No código *Tiling* o deslocamento das operações ocorre conforme ilustra a Figura 1. As variáveis **x** e **y** percorrem os blocos linhas e colunas. Nesses blocos as variáveis **i** e **j** percorrem as linhas e colunas. Há o aproveitamento dos dados da *cache* reduzindo o acesso a memória principal para matrizes que não podem ser armazenadas totalmente em *cache*, como as de ordem superior a 2048. Entretanto, matrizes menores como 512 e 1024 não ganham significativamente com a *Tiling* pois elas cabem totalmente em *cache*.

A utilização eficaz da memória *cache* é fundamental para uma execução veloz. Além disso a execução paralela permite a divisão de tarefas entre os núcleos agilizando



(a) Blocos (x) (b) Blocos (y) (c) Linhas (i) (d) Colunas (j)

Figura 1. Deslocamento dos elementos da matriz.

Tabela 1. Tempo de execução (segundos) de cada versão.

Algoritmo	**Entrada**			
	512	1024	2048	4096
Naive (Sequencial - ijk - gcc - O2)	0.89	4.25	102.26	915.47
Naive + Unrolling (Paralela - ikj - icc - O2 - 32 *threads*)	0.21	0.57	1.01	3.53
Speedup	4.23	7.45	101.24	259.33

a execução possivelmente tantas vezes quanto o número de *threads* utilizadas.A partir disso, foi variado o número de *threads* em 4, 8, 16 e 32 para verificar a escalabilidade. A alocação contígua (indexação simples) ou não-contígua (indexação dupla) permite ganho de desempenho dependendo da ordem da matriz e do compilador utilizado. Testes realizados mostram que pode haver ganho de até 2 vezes com indexação dupla no *gcc*.

4. Resultados e Discussão

Os resultados mostram que o ganho da versão *Tiling+Unrolling* entre execuções sequenciais superiores a 2048 dá-se nos compiladores *gcc* e *icc* pelo princípio de localidade da *cache*, pois aumenta o número de acertos de dados em *cache*. Tal análise foi feita com a ferramenta *Perf* coletando o número de *cache misses* e mensurando tal redução de faltas. O compilador *icc* se mostrou mais eficiente, seus resultados obtiveram menores tempos de execução, por isso utilizou-se seus dados como referência na Tabela 1.

As técnicas utilizadas neste trabalho vem sendo aprimoradas. O último ganho obtido foi de 19.57 vezes de uma versão paralela otimizada sobre uma versão sequencial não otimizada para uma arquitetura de 6 núcleos [da Silva et al. 2016]. Os resultados atuais mostram um ganho de 259.33 vezes de uma versão paralela otimizada sobre uma versão sequencial não otimizada. Estes dados são mostrados na Tabela 1, a qual apresenta os tempos de execução para a ordem 512, 1024, 2048 e 4096. O *tile* e *unroll* utilizados nos experimentos da Tabela 1 correspondem aos menores tempos de execução. O melhor caso, com 3.53 segundos, utilizou a versão *Naive+Unrolling* com *unroll* 2 e execução paralela em 32 *threads*. Este melhor caso justifica-se uma vez que os elementos da matriz são melhor distribuídos entre as *threads* do que a *Tiling* e *Tiling+Unrolling*.

O ganho de desempenho referente a *Tiling+Unrolling* foi através da modificação do código, uma vez que o particionamento em blocos facilita o acerto de dados na *cache*, pois toda a submatriz a ser multiplicada cabe na memória *cache*. A execução paralela

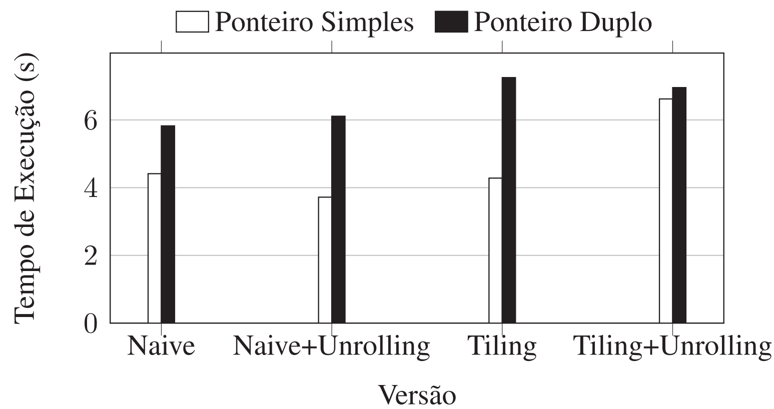


Figura 2. Tempo de Execução (Segundos).

melhora o tempo de execução uma vez que divide a tarefa em execuções simultâneas em diferentes cores. O ganho total de 259.33 vezes, foi consequência desta série de otimizações. Primeiramente foi feita a otimização sequencial usando *Loop Tiling* e *Loop Interchange*. Após, as versões do algoritmo foram paralelizadas. Por fim a otimização de alocação por indexação simples ou dupla e a utilização de um compilador mais eficiente. A Figura 2 mostra o tempo de uma matriz de ordem 4096 executada em 32 *threads*.

5. Conclusão e Trabalhos Futuros

Devido a necessidade de computações mais rápidas para prover resultados em tempo de execução aceitável, cada etapa de otimização foi eficaz e nota-se o ganho de desempenho de até 259 vezes após estas otimizações. Desta forma a utilização de diversas técnicas de otimização possibilita resultados cada vez mais velozes. A combinação do melhor aproveitamento da memória *cache* com a distribuição em diferentes núcleos de processamento e um compilador eficiente possibilita um alto ganho de desempenho.

Como trabalhos futuros podem ser avaliadas técnicas como vetorização de dados, execuções em GPUs (*Graphics Processing Units*), e também a aplicação das melhores versões obtidas em algoritmos que utilizam matrizes para análise de dados.

Agradecimentos

Este trabalho foi realizado com recursos do PROBIC/FAPERGS 2016 e pela agência de fomento CNPq via Edital Universal Processo N. 457684/2014-3.

Referências

da Silva, S. A., Serpa, M., and Schepke, C. (2016). Técnicas de Otimização Loop Unrolling e Loop Tiling em Multiplicações de Matrizes Utilizando OpenMP. In *Workshop de Iniciação Científica do WSCAD 2016 (XVII Simpósio em Sistemas Computacionais de Alto Desempenho) WSCAD-WIC 2016*, pages 13–18, Aracajú. SBC.

Jacquelin, M., Marchal, L., and Robert, Y. (2009). Complexity Analysis and Performance evaluation of Matrix Product on Multicore Architectures. In *2009 International Conference on Parallel Processing*, pages 196–203. IEEE.

Kowarschik, M. and Weiß, C. (2003). An Overview of cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, pages 213–232. Springer.