Modernização de código: estudo de caso utilizando multiplicação de matriz

Nickolas R. Machado, Juliana M N S Zamith

Departamento de Ciência da Computação – Universidade Federal Rural do Rio de Janeiro (UFRRJ)
Rio de Janeiro – RJ – Brasil

Resumo. *Operações aritméticas envolvendo estruturas matriciais são processos que consomem tempos significativos de execução dependendo do tamanho da instância a ser resolvida. Este trabalho propõe otimizar e modernizar um código que resolve o problema de multiplicação de matriz com objetivo de melhorar o seu desempenho. Para tanto, foram utilizadas técnicas de paralelização como instruções vetoriais (AVX), OpenMP e o uso de placas gráficas. Foram obtidos resultados que melhoram em até 735 vezes o tempo computacional dos códigos modernizados quando comparados com o não modernizado.*

1. Introdução

Multiplicação de matrizes faz parte da solução de problemas reais de diversas áreas, como: processamento de imagens e problemas nos setores medicinais e empresariais. Isso ocorre porque a multiplicação de matrizes é um dos passos essenciais para métodos de resolução de sistemas lineares e decomposições de matrizes, como, por exemplo a *Singular Value Decomposition* (*SVD),* que é usada, comumente, no tratamento de imagens e vídeos para comprimir e reduzir ruído das figuras.

Uma matriz é um conjunto de dados que são exibidos em formato de tabela, onde a posição de cada elemento é definida pelas coordenadas *i* e *j,* que indicam, respectivamente, a linha e coluna do elemento. Uma das operações básicas efetuada entre matrizes é a multiplicação. Para realiza-la, cada elemento da matriz resultante C_{ij} é dado pelo produto escalar da linha *i* de uma matriz *A* pela coluna *j* de uma matriz *B*. Outrossim, é válido ressaltar que para todos os elementos da linha de *A* sejam multiplicados, *B* precisa ter a quantidade de linhas igual a de colunas de *A*.

No âmbito computacional, o algoritmo que expressa a multiplicação de matrizes é altamente dependente de operações aritméticas e possui complexidade $O(n^3)$ em matrizes quadradas, o que significa ser necessário efetuar ao menos n^3 operações, no pior caso, para finalizar o trabalho, onde *n* é o número de linhas da matriz *B*.

Para aproveitar as arquiteturas atuais em sua totalidade deve-se paralelizar o código de modo que todas as unidades de processamento sejam usadas durante a execução. Ainda, deve-se fazer um bom uso da memória para evitar possíveis gargalos de acesso. Além disto, dispositivos, como a *GPU*, quando presentes nas máquinas, permitem a computação massiva de dados, aumentando o *speedup* da aplicação.

Este trabalho teve como objetivo utilizar algumas alternativas de modernização de código para acelerar a execução da multiplicação de matrizes. Os resultados mostram o ganho obtido a cada otimização, chegando a uma aceleração de até 735 vezes em relação a um código sem as devidas otimizações. As próximas seções apresentam as melhorias realizadas bem como os resultados obtidos.

2. Otimizações

Nesta seção são descritas as otimizações utilizadas. As mesmas foram escolhidas por terem alto potencial para a computação paralela e para melhorar o uso da memória que é excessivamente acessada na execução.

2.1. Otimização 1 (O1): melhorando o uso da memória.

Fazer o uso inteligente da memória *cache* pode tornar o código eficiente, tendo em vista que a *cache* permite um acesso mais rápido ao dado do que na memória principal. Pelo princípio da localidade, uma referência a memória principal causa a cópia não só do dado requisitado, mas também seus vizinhos [2]. Aproveitar a localidade para evitar falhas na cache pode melhorar muito o tempo de execução. Sendo assim, percorrer a matriz *B* por colunas pode invalidar o princípio da localidade, pois, cada dado está a *n* (número de colunas de *B*) itens de distância. Consequentemente, o valor procurado pode não estar entre os vizinhos carregados na iteração anterior. Todavia, ao transpor a matriz *B*, cada coluna da matriz é percorrida por linha, evitando a *cache miss*. Esta foi a primeira otimização aplicada ao código.

2.2. Otimização 2 (O2): registradores vetoriais *AVX*

A segunda otimização foi realizada através do uso de registradores *AVX256*. O uso destas instruções, além de reduzir o gargalo de memória, permite executar laços com uma quantidade menor de iterações. Uma vez que cada posição do vetor utiliza 32 *bits* para cada número, por meio do uso de registradores *AVX* de 256 *bits*, é possível realizar a mesma operação com 8 números simultaneamente, melhorando o desempenho[3]. Porém, não se traduz numa melhora de 8 vezes em razão do algoritmo possuir alguns passos a mais na preparação e na execução.

2.3. Otimização 3 (O3): paralelismo na *CPU* utilizando *OpenMP*

É intuitivo pensar que o uso dos múltiplos núcleos da *CPU* durante a execução do código possa melhorar o seu desempenho. Por essa premissa, foi feita, juntamente com as otimizações anteriores, a paralelização do código utilizando a biblioteca *OpenMP* [2]. Nesta otimização, cada linha matriz resultante *C* foi calculada por uma *thread*. A paralelização foi simples e realizada utilizando a diretiva *#parallel for* na execução dos *loops* da multiplicação.

2.4. Otimização 4 (O4): paralelismo na *GPU*

Por fim, a otimização que se apresentou mais eficiente foi utilizar a *GPU* para realizar os cálculos. Inicialmente usou-se a localização global por blocos e *threads* para abstrair o uso dos índices *i* e *j* [1]. Entretanto, tal otimização não explorava o potencial da *GPU*, fazendo com que o tempo de execução estivesse próximo aos tempos obtidos pela *CPU*. Por conseguinte, foi implementada a multiplicação usando *tiles*, um método que visa dividir a matriz principal em submatrizes e realizar as operações separadamente, de forma a efetuar as contas em blocos do tamanho das submatrizes. Além disso, cada *thread,* responsável por um elemento, copia os dados para uma memória local mais rápida e compartilhada, e então, realiza a multiplicação por partes a partir dos dados nessa memória[1] .

Outras alterações foram feitas, como transformar as matrizes em vetores para evitar o cálculo de endereços de acesso à memória e utilizar variáveis locais para guardar os resultados de operações frequentes em *loops*.

3. Resultados

Os testes foram realizados em uma máquina Intel Core i7 4790, com uma *GPU* NVIDIA GTX 970 com 16GB de *RAM* e *cores* com frequência de 3900Mhz. Os testes

estão descritos na Tabela 1, sendo os valores nas colunas *speedup* obtidos a partir do tempo de execução do tipo de otimização em relação a versão original para cada matriz. As matrizes foram geradas utilizando a função *rand()* com valores inteiros de 4 bytes.

Os resultados, quando utilizada a otimização O1, apresentam uma melhora de, aproximadamente, 3,5 vezes para matrizes da ordem de 4000x4000 em relação a versão original (sem nenhuma otimização). Vale ressaltar que, os dados já são gerados ou lidos na forma transposta, sendo assim, o tempo apresentado não inclui o tempo gasto para transpor a matriz. Acredita-se que o tempo foi reduzido pelo uso eficiente da *cache*.

Considerando a otimização O2, adicionando a O1 as instruções *AVX*, foi obtida uma melhora de 8,9 vezes em relação ao tempo quando comparada com a versão original e 2,5 vezes quando comparada com a O1, para matrizes de tamanho 4000. Para a O3 foram utilizados os quatro *cores* da máquina e criadas oito *threads*. O tempo de execução melhorou próximo de 4 vezes quando comparada com a O2.

Com relação à otimização O4, foi obtida melhora na ordem de 656 vezes em matrizes de dimensões 4000 em relação a versão original. Vale ressaltar que este tempo inclui o tempo de transferência entre *CPU* e *GPU*. Para esta otimização foi realizado um teste considerando uma matriz maior da ordem de 10.000 com o intuito de verificar a escalabilidade. O tempo de execução foi de 20s enquanto a versão original apresentou um tempo de 4h5m; uma redução de 1/735 vezes.

Tabela 1. Tempos de execução dos códigos por dimensão das matrizes

Tipo de otimização	**Tamanho da Matriz**							
	1000x1000		**2000x2000**		**3000x3000**		**4000x4000**	
	Tempo	***speedup***	**Tempo**	***speedup***	**Tempo**	***speedup***	**Tempo**	***speedup***
Original	6,58s		90,83s		348,56s		853,70s	
O1: Transposta	3,78s	1,74	30,19s	3,00	101,89s	3,42	241,33s	3,54
O2:AVX	1,44s	4,57	11,69s	7,77	40,47s	8,61	95,78s	8,91
O3: OpenMP	0,34s	19,35	2,75s	33,03	9,54s	36,54	22,45s	38,03
O4: CUDA	0,03s	219,3	0,18s	504,61	0,56s	622,43	1,30s	656,69

4. Conclusão

A partir das técnicas aqui apresentadas conclui-se que o uso de paralelismo, em *CPU* e *GPU*, o uso de instruções de vetorização e programação *cache friendly* reduz muito o tempo de computação. A multiplicação de matrizes foi escolhida para este trabalho inicial pois, é uma aplicação inerentemente paralela, evitando *overheads* de comunicação ou sincronização, sugerindo assim, um bom *speedup* em sua versão paralela. Pretende-se avaliar o uso das técnicas de modernização em aplicações que sugerem mais desafios em relação a paralelização para verificar o ganho de desempenho em nestas classes de algoritmos.

5. Referências

[1] Jason Sanders & Edward Kandrot, (2010) “CUDA by example : an introduction to general-purpose GPU programming”.

[2] Tim Mattson, (2012) “A “Hands-on” Introduction to OpenMP”.

[3] Intel Corporation, “Intrinsics Guide”.