FILTRAGEM COLABORATIVA COMO SERVIÇO UTILIZANDO PROCESSAMENTO NA GPU

Vinicius Dalto do Nascimento

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Geraldo Zimbrão da Silva

Rio de Janeiro
Abril de 2013

FILTRAGEM COLABORATIVA COMO SERVIÇO UTILIZANDO PROCESSAMENTO NA GPU

Vinicius Dalto do Nascimento

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Geraldo Zimbrão da Silva, D.Sc.

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof[a]. Jonice de Oliveira Sampaio, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

ABRIL DE 2013

Nascimento, Vinicius Dalto do

Filtragem Colaborativa como Serviço utilizando Processamento na GPU/Vinicius Dalto do Nascimento. – Rio de Janeiro: UFRJ/COPPE, 2013.

XIII, 75 p.: il.; 29, 7cm.

Orientador: Geraldo Zimbrão da Silva

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 62 – 63.

1. Filtragem Colaborativa. 2. GPU. 3. Threads. 4. Serviço. 5. Alto Desempenho. 6. CUDA. I. Silva, Geraldo Zimbrão da. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A meu querido avô José Nicolau,

meu exemplo.

Agradecimentos

Gostaria de agradecer primeiramente à minha esposa, pela paciência e compreensão que teve nos momentos em que tive trabalhando nesta dissertação. Em segundo, minha mãe, pois sem toda a sua dedicação e esforço, nunca teria me tornado a pessoa que sou e seguido a trajetória que segui. Ao meu orientador Geraldo Zimbrão, que sempre confiou em meu trabalho, conseguiu os recursos necessários para que o trabalho pudesse ser feito e me orientou nas decisões importantes. Ao professor Ricardo Farias, permitindo utilizar sua máquina para rodar os experimentos e colaborando ao me ajudar com seus conhecimentos em CUDA.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

FILTRAGEM COLABORATIVA COMO SERVIÇO UTILIZANDO PROCESSAMENTO NA GPU

Vinicius Dalto do Nascimento

Abril/2013

Orientador: Geraldo Zimbrão da Silva

Programa: Engenharia de Sistemas e Computação

Atualmente um algoritmo largamente utilizado em sistemas de recomendação é o *Filtro Colaborativo (FC)*, que tem o objetivo de recomendar itens ou prever a avaliação de um item para um usuário utilizando como base os dados dos usuários similares à ele. Alguns trabalhos tentaram reduzir o tempo de recomendação do FC utilizando paralelismo com threads na CPU *(Central Processing Unit)* [1] e até mesmo com threads na GPU *(Graphical Processing Unit)* [2]. Atualmente as GPUs modernas manipulam operações matemáticas com eficiência e suas estruturas de processamento paralelo as tornam mais capazes neste tipo de trabalho que CPUs normais. Com a larga utilização de GPUs, sendo até mesmo utilizadas nos top500 [3] computadores mais poderosos do mundo, este trabalho irá propor uma arquitetura que utilizará um FC como serviço tendo o processamento feito na GPU. Mostraremos uma arquitetura escalável, onde em nossos experimentos utilizaremos bases de dados relativamente grandes como a do Movielens [4] e Netflix [5]. Como a memória global das GPUs atualmente são relativamente menores que as memórias principais dos sistemas, tentaremos resolver o problema no qual as matrizes do algoritmo de FC não possuem espaço suficiente na memória da GPU. Utilizamos algumas tarefas apresentadas no artigo [1] para efeito de comparação de desempenho e poderemos verificar que realmente há um grande aumento de performance em nossa arquitetura utilizando GPU.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COLLABORATIVE FILTERING AS A SERVICE USING PROCESSING IN GPU

Vinicius Dalto do Nascimento

April/2013

Advisor: Geraldo Zimbrão da Silva

Department: Systems Engineering and Computer Science

Now a days an algorithm widely used in recommendation systems is Collaborative Filtering (CF), which aims to recommend items or provide a item recommendation for a user on using user data similar to him. Some studies have tried to shorten the recommendation of FC using parallelism with threads on the CPU (Central Processing Unit) [1] and even threads on the GPU (Graphical Processing Unit) [2]. Modern GPUs efficiently handle mathematical operations and parallel processing structure makes them more capable in this type of work that mainstream CPUs. With the widespread use of GPUs, even being used in TOP500 [3] most powerful computers in the world, this work proposes an architecture that uses CF as service having the processing done on the GPU. We present a scalable architecture, which we use in our experiments relatively large databases such as the Movielens [4] and Netflix [5]. The global memory of the GPUs today are relatively smaller than the main memory of systems, we tryed to solve the problem in which the matrix of the algorithm FC does not have enough space in GPU memory. We used some tasks presented in the article [1] for performance comparison and we can see that there is a really big performance boost in our architecture using GPU.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com o surgimento da Internet, diversos serviços foram disponibilizados, como por exemplo, pesquisas, troca de mensagens através do correio eletrônico e também para processos de compra, venda e troca de produtos através do comércio eletrônico. Com o passar dos anos as empresas começaram a utilizar massivamente a internet como canal de vendas, onde não somente a empresa, mas também os consumidores que foram beneficiados pela facilidade de acesso e também pelo baixo custo de manter um site.

A empresa que disponibiliza seu negócio na web aumenta o alcance de consumidores, expondo uma enorme quantidade de produtos e operando todos os dias do ano, fatos que dificilmente poderiam ser alcançados pelo método de venda tradicional. Os consumidores beneficiam-se com maior comodidade podendo comprar de qualquer lugar (trabalho, residência, etc.) a qualquer hora, ter acesso a muitas informações sobre os produtos como preço, características, opiniões de outros usuários, e navegar pela loja através de diferentes categorias de produtos. Este benefício, entretanto, pode vir a tornar-se um problema para o consumidor, que diante de tantas opções não consegue encontrar algo específico.

Devido a grande quantidade de produtos disponíveis no mercado existe a dificuldade das pessoas em escolherem entre uma grande variedade de produtos e serviços. Entre as várias alternativas que lhe são apresentadas, surgem os *sistemas de recomendação* (SR). Sistema de Recomendação é um tipo de técnica de filtragem da informação que procura recomendar informação e produtos (como filmes, vídeos, músicas, livros, páginas de internet, itens de consumo, etc.) que possam ser de in-

teresse para o usuário. A evolução destes sistemas e o fato deles trabalharem com grandes bases de informações permitiram que recomendações emergentes (não triviais) pudessem ser alcançadas, proporcionando ainda maior credibilidade que uma recomendação humana.

Os sistemas de recomendação coletam informações dos usuários para descobrir suas preferências e assim ofertar produtos e serviços que lhe sejam relevantes. Para isso, utilizam uma técnica de Filtragem de Informação (FI) para extrair as relações e similaridades existentes entre produtos, entre consumidores e entre produtos e consumidores. Dentre as diversas técnicas de filtragem pode ser citada principalmente a Filtragem Colaborativa (FC) que utiliza dados como histórico de compras, visualização ou avaliações de produtos, encontra relações existentes entre usuários, como interesses em comum (ou não), e também relações entre os itens, como compras conjuntas.

A filtragem colaborativa utiliza como base um histórico de avaliações que os usuários deram à um conjunto de itens. Esse histórico é guardado em uma forma matricial que será utilizada como entrada no algoritmo de filtro colaborativo. Essa matriz na maioria dos casos precisa ser armazenada em memória e possui um tamanho na ordem de gigabytes.

Com bases de dados crescentes e com milhares de usuários e itens, cada vez mais há necessidade de mais processamento para rodar os algoritmos de filtragem colaborativa para que se possa sugerir algo ao usuário. Os algoritmos de filtragem colaborativa utilizam muitas operações matriciais que podem ser paralelizadas e consequentemente diminuir o tempo de execução da aplicação. Atualmente, com o avanço dos processadores podemos encontrar diversas arquiteturas que utilizam mais de um núcleo. Para poder fazer uso do processamento paralelo, existem diversas soluções tecnológicas que podem ser utilizadas para paralelizar algoritmos na CPU, p.ex: *OpenMP* [6], *PThreads*[7] e até mesmo o *MPI (Message Passing Interface)*[8]. Atualmente não somente as CPUs, mas também as GPUs tiveram um grande avanço, tanto na parte do hardware aumentando seu número de processadores e a velocidade de acesso a sua memória, quanto no software onde era bem trabalhoso utilizar a GPU para uma tarefa científica. Atualmente uma CPU pode conter cerca de quatro e até mesmo seis núcleos. No entanto, uma GPU mediana atualmente pode conter

336 núcleos de processamento, como é o caso da Geforce GTX560 da NVidia por exemplo.

Houve um tempo no passado não tão distante, quando a computação paralela era vista como uma especialidade dentro do campo da ciência da computação. Essa percepção mudou de forma profunda nos últimos anos. O mundo da computação mudou ao ponto em que, atualmente longe de ser um exercício atípico, quase todo aspirante à programador cursa algum tipo de cadeira em programação paralela no curso de ciência da computação. A computação paralela vem cada vez mais se tornando comum para se resolver problemas grandes onde há necessidade de muito processamento.

A computação está evoluindo de "processamento centralizado" na CPU para "co-processamento" na CPU e GPU [9]. Para possibilitar esse novo paradigma de computação, a NVIDIA disponibilizou uma arquitetura de computação paralela que já está sendo suportada em placas gráficas da família GeForce, Quadro e Tesla. Esta tecnologia é chamada de *CUDA (Compute Unified Device Architecture)*. O CUDA foi recebido com entusiasmo na área da pesquisa científica. Por exemplo, agora o CUDA pode acelerar o AMBER [10], que é um programa de simulação de dinâmica molecular usado por mais de 60.000 pesquisadores em universidades e empresas farmacêuticas em todo o mundo para acelerar a descoberta de novos medicamentos.

Um indicador de adoção do CUDA é a grande utilização da GPU Tesla para computação em GPU. Os supercomputadores movidos com GPU da NVIDIA se apresentam hoje entre os 10 melhores supercomputadores do Top500 [3] e Green500rankings [11], sendo atualmente o primeiro colocado dos Top500. Atualmente a computação em GPU é totalmente suportada por todos os principais sistemas operacionais. Devido à todo esse crescimento da computação utilizando GPU, decidimos utilizar em nosso trabalho o paralelismo utilizando CUDA para realizar as operações alta demanda de processamento e com isso tentar aumentar as vazões de recomendações solicitadas ao nosso serviço de recomendação.

Em nosso trabalho iremos propor uma arquitetura onde utilizaremos um algoritmo de filtro colaborativo como serviço utilizando o processamento na GPU. Iremos utilizar grandes bases dados, como por exemplo, a do Netflix [5] e tentaremos mostrar um serviço escalável que utilize os recursos de processamento da

GPU. Utilizaremos como base de comparação o trabalho de LEVANDOSKI[1] onde foram feitos experimentos visando desempenho utilizando esse algoritmo. Para validar nossa arquitetura efetuaremos execuções de tarefas propostas no artigo citado e analisaremos os resultados com relação a performance.

É importante salientar que existem outras técnicas de recomendação que podem ser vista em [12]. Mas estamos interessados em estudar desempenho, em nosso trabalho focaremos no algoritmo de filtro colaborativo com base no item, que é um algoritmo que exige uma boa carga de processamento e operações matriciais, que são bons requisitos para se utilizar o processamento na GPU.

Nosso trabalho será estruturado em 5 capítulos e um anexo. O capítulo 1 será composto pela introdução aqui apresentada, no capítulo 2 iremos mencionar os principais trabalhos que foram utilizados como referência e os respectivos comentários sobre cada trabalho, sua contribuição para este trabalho e iremos citar o que fará parte deste trabalho e o que não fará. Além disso, iremos dissertar sobre o problema de executar um algoritmo de filtro colaborativo com base de dados grandes e a quantidade processamento que é necessária para sua execução. Para finalizar o capítulo, iremos dar uma breve explanação sobre os algoritmos utilizados e também sobre as arquiteturas que utilizam GPU e sobre o seu funcionamento.

No capítulo 3 iremos explicar qual foi a solução proposta além de como implementá-la. Explicaremos como será nossa arquitetura, a tecnologia que utilizamos, os principais algoritmos utilizados além dos principais problemas encontrados na implementação da solução.

O capítulo 4 será dedicado para a execução dos experimentos e a análise dos resultados. Iremos mostrar como foram os feitos os experimentos, os parâmetros utilizados e também o hardware utilizado na execução das tarefas. Mostraremos os resultados obtidos neste trabalho e também os resultados das tarefas que foram utilizadas no artigo citado no capítulo 2.

No capitulo 5 serão postas as nossas conclusões sobre a nossa arquitetura e sobre os resultados obtidos. Para encerrar o capítulo, iremos propor algumas sugestões de trabalhos futuros que poderiam completar este trabalho ou que não fazem parte do nosso escopo, mas que poderiam contribuir. E finalmente no anexo A poderão ser vistos os códigos fontes dos algoritmos mencionados no capítulo 3.

Capítulo 2

Definição do Problema

Atualmente com a crescente necessidade de recomendar algo ao usuário, tornou-se primordial recomendar com rapidez e qualidade já que isso pode ser a diferença entre um usuário comprar um produto ou não. Com a grande utilização do comércio eletrônico atualmente, há a necessidade de se gerar diversas recomendações por segundo para que se possa suprir a demanda de usuários acessando um site e navegando nos produtos oferecidos. Um algoritmo muito utilizado na recomendação de itens é o Filtro Colaborativo que utiliza o histórico de avaliações que os usuários deram aos produtos para encontrar uma similaridade entre usuários ou produtos e então dar uma recomendação ao usuário. A utilização deste algoritmo como um serviço seria interessante, pois este receberia diversas requisições por segundo solicitando recomendações para produtos ou listas de produtos para recomendar a um usuário. Para atender um grande site, este serviço deverá ter uma grande carga de trabalho e, além disso, deverá ser escalável pois poderá haver uma crescente demanda de usuários acessando o site o que acarretará em solicitações de recomendação ao serviço. Para resolver o problema da grande demanda de processamento utilizamos o processamento do serviço em GPUs. Atualmente as GPUs já faz parte do melhor supercomputador do mundo da lista dos Top500 e além de diversas empresas utilizarem o seu trabalho pesado de processamento nas GPUs.

Atualmente uns dos problemas da utilização de GPUs é a baixa quantidade de memória quando comparada as memórias principais de um computador. Apesar de serem bem mais rápidas que as memórias principais, por serem em menor quantidade, a escalabilidade de alguns algoritmos é dificultada, pois há a necessidade

de gerenciar a troca de dados entre as memórias até o limite de espaço da memória da GPU. Em nosso trabalho tentamos resolver o problema da alta demanda de recomendações utilizando a GPU e também tentamos resolver o problema da escalabilidade do algorítmo de filtro colaborativo utilizando a GPU com as suas limitações de memória. Para validar nossa arquitetura utilizamos tarefas utilizadas em outros trabalhos e comparamos os resultados de forma que possamos verificar se o tempo de resposta ao serviço será razoável.

Para resolver o problema apresentado, neste capítulo apresentaremos os principais trabalhos correlatos e os conceitos necessários para o entendimento de nossa solução. Abordaremos os principais conceitos sobre algoritmos de filtragem colaborativa e os principais conceitos de computação paralela utilizando a GPU.

2.1 Trabalhos Correlatos

Nesta seção iremos apresentar os principais trabalhos relacionados e alguns resultados que foram obtidos nestes trabalhos para que possamos utilizar como comparação. Alguns comentários serão feitos sobre os artigos com relação as considerações que serão feitas neste trabalho.

2.1.1 RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures.(Justin J. Levandoski, Michael D. Ekstrand, Michael J. Ludwig, Ahmed Eldawy, Mohamed F. Mokbel, and John T. Riedl, September, 2011)

No trabalho de [1] são apresentadas três arquiteturas alternativas para construção de sistemas de recomendação e o estudo de desempenho de cada uma delas. Uma dessas arquiteturas é uma implementação feita em Java [13] e utiliza threads para realizar processamento simultâneo e tentar reduzir o tempo de execução das tarefas propostas no artigo. As tarefas utilizadas no artigo são:

1 Inicialização. O objetivo é preparar a inicialização do sistema de recomendação para uso online. Na maioria dos sistemas, a inicialização significa construir

um modelo de recomendação com base nas classificações anteriores ou histórico de compras de uma base de usuários.

2 Lista de Recomendações. Esta tarefa simula um usuário navegando para um site de e-commerce (por exemplo, a Netflix), onde recebem um conjunto de recomendações a partir do sistema em uma home page. O objetivo desta tarefa é produzir para o usuário um conjunto de n itens recomendados. Esta tarefa testa a eficácia do sistema na execução do processo de geração de uma lista de recomendação para um usuário comum.

3 Recomendação para um conjunto de itens. Esta tarefa simula um usuário requisitando recomendações para apenas um conjunto específico de itens (por exemplo, filmes de comédia lançados após 1990). O objetivo desta tarefa é produzir um conjunto de n itens recomendados que correspondam uma ou mais restrições colocadas sobre os metadados de item (por exemplo, no ano de filme). Esta tarefa testa a eficiência do sistema em consultar metadados de item e produzir um conjunto de recomendações de n itens filtrados.

4 Recomendação Misturada. Esta tarefa simula um usuário fazendo um busca de texto e recebendo uma recomendação do item buscado. O objetivo desta tarefa é produzir um conjunto de n recomendações dos itens buscados e gerar uma média dos itens recomendados e do texto buscado.

5 Predição de Item. Esta tarefa simula um usuário navegando por um item específico (p.ex: filme, livro) onde o sistema de recomendação detecta o interesse do usuário pelo item. O objetivo desta tarefa é fazer com que o sistema de recomendação gere uma nota tentando prever a nota que o usuário daria ao item. Esta tarefa testa o desempenho do sistema na produção de predições para um único item.

6 Atualização de Item. Esta tarefa simula um novo item sendo adicionado ao sistema. O objetivo desta tarefa é fazer com que o item entre em circulação como candidato de recomendação o mais cedo possível depois que os usuários começaram a classificação dele ou clientes começaram a comprá-lo. Na maioria dos sistemas, esta tarefa vai exigir que ao ser incorporado o item, o modelo seja construído originalmente pela tarefa de inicialização.

Em nosso trabalho iremos utilizar as tarefas 1, 2 e 5 para que possamos comparar o tempo execução de nossa arquitetura com o artigo citado. As tarefas 3, 4 e 5 deste

trabalho nós não faremos uso, pois as três implementações apresentadas por este trabalho fazem uso de um SGBD *(Sistema Gerenciador de Banco de Dados)* para armazenar o dataset com as avaliações dos usuários e nessas tarefas são utilizados comandos SQL para filtragem de itens que não fazem parte do escopo de nosso trabalho.

2.1.2 A social network-aware top-N recommender system using GPU.(Li, Ruifeng and Zhang, Yin and Yu, Haihan and Wang, Xiaojun and Wu, Jiangqin and Wei, Baogang, 2011)

Neste artigo, foi proposto um algoritmo de recomendação paralela Top-N em CUDA, que combina a filtragem colaborativa e *trust-based*[1] para lidar com o problema do *cold start user*[2]. Então, com base neste algoritmo, foi apresentado um sistema paralelo recomendação de livros utilizando uma GPU para uma biblioteca digital chinesa chamada *CADAL* [14]. Neste trabalho o *dataset* utilizado (a biblioteca CADAL) não possui um tamanho considerável como o do Netflix Prize, e também não foi considerado o caso das GPUs não possuírem espaço suficiente em memória para armazenar as matrizes de recomendação e similaridades.

2.1.3 Item-Based Collaborative Filtering Recommendation Algorithms (Badrul Sarwar, George Karypis, Joseph Konstan, John Riedl, 2001)

Neste trabalho utilizado como referência, foram analisados diferentes tipos de algoritmos de recomendação baseados nos item. Foram analisadas diferentes técnicas para computar a similaridade entre itens. Foram feitas analises de algoritmos de predição baseados no item e diferentes modos de implementá-los. Além disso, foram realizados experimentos para comparar a qualidade da predição em cada algoritmo apresentado. Este trabalho nos serviu como base para entender os principais algorit-

[1] *trust-based: Relação de confiança estabelecida para ser utilizada como heurística entre os usuários.*

[2] *cold start user: Problema do usuário não possuir um histórico de recomendação.*

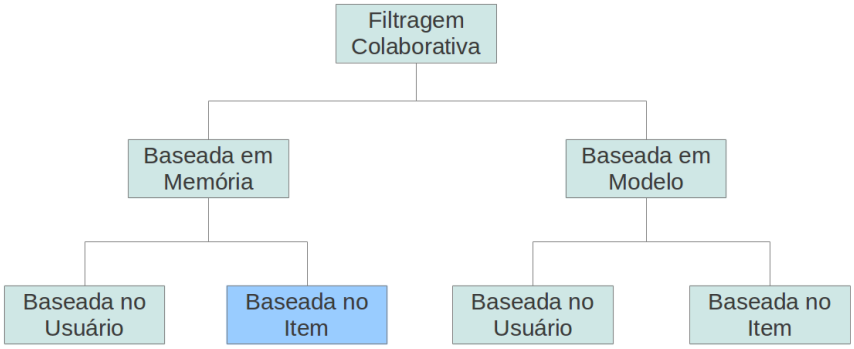

Figura 2.1: Divisão dos algoritmos de filtragem colaborativa.

mos de predição e seus conceitos, além conhecermos alguns dos principais problemas encontrados na área, que são a escalabilidade dos algoritmos e as grandes matrizes esparsas.

2.2 Algoritmo de Filtragem Colaborativa

A filtragem colaborativa (FC) é um tipo de sistema de recomendação que tem o objetivo de recomendar itens ou prever a avaliação de um item para um usuário, utilizando como base os dados os usuários similares a ele, ou seja, que mostraram os mesmos interesses que ele. A ideia dessa abordagem é que aqueles usuários que concordaram no passado tendem a concordar novamente no futuro. Por exemplo, se José gostou de assistir os filmes A e B, e Maria gostou de assistir os filmes A, B e C, então provavelmente José gostará de assistir o filme C. Nesse exemplo nós podemos perceber a importância de se calcular a similaridade entre os usuários ou itens para se utilizar no filtro colaborativo. Os métodos de filtragem colaborativa podem ser divididos em baseados em modelo e baseados em memória, como pode ser vistos na figura 2.1. O foco de nosso trabalho será baseado no algoritmo de filtro colaborativo baseado em memória, que será apresentado nas próximas seções.

O principal passo para execução do algoritmo de filtro colaborativo é a construção do modelo no caso do baseado em modelo e a construção da matriz de similaridades no caso do baseado em memória. Em seguida, com base no primeiro passo apresentado, será aplicado um algoritmo de recomendação, que será responsável por gerar a avaliação do item para um usuário. Nas próximas seções veremos mais detalhes

$$
R = \begin{array}{c|cccccc}
 & u_1 & u_2 & u_3 & u_4 & \dots & u_m \\
i_1 & 3 & 3 & 0 & 4 & \dots & 0 \\
i_2 & 5 & 0 & 0 & 4 & \dots & 0 \\
i_3 & 0 & 0 & 0 & 4 & \dots & 0 \\
i_4 & 4 & 5 & 0 & 4 & \dots & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots \\
i_n & 5 & 0 & 1 & 4 & \dots & 0
\end{array}
$$

Figura 2.2: Exemplo de Matriz de Avaliações Baseada no Item.

sobre os algoritmos citados em cada passo e também como implementá-los.

2.2.1 Filtragem Colaborativa Baseada em Memória

No algoritmo de filtro colaborativo baseado em memória, são definidas heurísticas com o intuito de prever notas para itens que não foram avaliados a partir de todo o conjunto de avaliações de itens feitas pelos usuários. O histórico de avaliações pode ser representado por uma matriz *item X usuário* **R** de dimensão n *x* m que indica a relação de n itens com m usuários. O filtro colaborativo baseado em memória pode ser utilizado com base no usuário ou com base no item. O baseado no usuário busca encontrar vizinhos, ou seja, um conjunto de usuários que possuam gostos similares ao usuário ativo, por exemplo, tendem a gostar de filmes de mesmas características. O baseado no item procura encontrar a similaridade entre os itens de forma que o usuário que gostou de um item também gostará de um item similar. Na prática, o baseado no usuário utilizará a matriz avaliações **R** como *usuário X item* e o baseado no item utilizará **R** como *item x usuário*. Em nosso trabalho iremos focar no filtro colaborativo baseado no item.

O conteúdo da matriz *R(i, u)* indica a avaliação que o item i recebeu do usuário u. A Figura 2.2 ilustra um exemplo da matriz *Item X Usuário*, onde as notas variam de 1 a 5, e 0 representa que o item não foi avaliado. Os itens em nosso trabalho serão considerados filmes que os usuário já viram. Exemplificando a interpretação da matriz mostrada na Figura 2.2, o usuário 2 avaliou o item 1 com nota 3, não avaliou os itens 2 e 3 e o item 4 foi avaliado com 5. No caso de um item não avaliado, pode-se utilizar um mecanismo de previsão de avaliações para estimar a avaliação que o usuário atribuiria ao item, a partir das avaliações que usuários semelhantes a ele fizeram do item. Para aplicação deste mecanismo de previsão é necessário que

se identifique os usuários vizinhos mais similares, ou seja, em nosso caso, usuários tenham interesse por filmes de mesmo gênero. Conhecendo a vizinhança dos usuários ou itens que desejamos prever, podemos aplicar o método de previsão de avaliações.

Criação do Modelo

Para prever a nota que um usuário irá atribuir a um filme, primeiro precisamos calcular a similaridade entre os itens, ou seja, nós precisamos atribuir pesos a todos os itens indicando o seu grau similaridade com o item ativo e assim criar uma matriz de similaridades que servirá de modelo. Para calcular a similaridade entre itens existem diversos métodos, os dois mais utilizados são o coeficiente de correlação de *Pearson* e o método do Cosseno. Em nosso trabalho iremos trabalhar somente com o método do cosseno, pois os trabalhos os quais estamos referenciando estão fazendo uso deste método. Como estamos interessados em comparar desempenho e não precisão, iremos utilizar somente este método para efeito de uma comparação justa. No método cosseno, dois usuários ou dois itens, dependendo em que o método de predição é baseado, são tratados como um vetor e a similaridade é a medida através do cálculo do cosseno entre os vetores, como pode ser visto na fórmula 2.1.

$$sim(i_p, i_q) = \frac{\vec{i_p}.\vec{i_q}}{||\vec{i_p}||.||\vec{i_q}||} \tag{2.1}$$

Para cada combinação de item (ou linha da matriz $\mathbf{R}$) é aplicada a fórmula 2.1. De forma geral, somente os itens abaixo da diagonal da matriz $\mathbf{S}$ serão calculados. Como a matriz é simétrica, os itens restantes serão espelhados com seus respectivos pares com exceção da diagonal que o valor será um. É uma prática comum truncar o modelo baseado no item armazenando somente os k itens mais similares ao item ativo, isso diminuirá o espaço em memória e terá um impacto mínimo na qualidade das recomendações [15]. Mas para que este truncamento seja feito, as linhas da matriz $\mathbf{S}$ precisarão ser ordenadas pela similaridade, o que também terá um custo de processamento que iremos detalhar mais a frente.

Geração das Recomendações

Suponha que um usuário u_q necessita de uma recomendação para um item i, então a predição representada por $P(u_q, i)$ pode ser calculada de acordo com a seguinte

equação de predição 2.2:

$$P(u_q, i) = \frac{\sum_{l \in L} sim(i,l) * r_{u_{q,l}}}{\sum_{l \in L} |sim(i,l)|} \tag{2.2}$$

Antes de iniciarmos o cálculo da predição, nós reduzimos cada lista de similaridades L para conter somente itens avaliados pelo usuário u_q. Então podemos definir a equação de predição como a soma de $r_{u_{q,l}}$, que é a avaliação do usuário para um item $l \in L$ que possui um peso dado por *sim(i,l)*, que é a similaridade de l com o item candidato i, que então é normalizado pela soma das similaridades entre i e l. Com este valor de predição, nós poderemos não somente tentar prever a nota que o usuário dará ao item, mas também gerar um lista de recomendações para o usuário com base na nota recomendada ordenada como veremos mais a frente.

2.2.2 Considerações Sobre os Algoritmos

Vimos nas seções anteriores como o algoritmo de FC baseado no item funciona, as estruturas de dados necessárias para sua implementação e os principais passos do algoritmo que precisam ser executados em sequência para que possamos gerar uma recomendação. Veremos nas próximas seções que por se tratar de operações matriciais e vetoriais poderemos utilizar computação paralela nos algoritmos para reduzir seu tempo de execução. Nas próximas seções iremos explicar os principais conceitos de computação paralela e de utilização da GPU necessários para o entendimento da solução deste trabalho.

2.3 Computação Paralela

Microprocessadores baseados em uma única unidade central de processamento (UCP), como as das famílias Intel Pentium e AMD Opteron proporcionaram rápidos aumentos de desempenho e redução nos custos de aplicações computacionais por mais de duas décadas. Esses microprocessadores trouxeram a capacidade de processar Giga (bilhão) de operações de ponto flutuante por segundo (GFLOPS) em computadores pessoais (*Desktops*) e cerca de milhares de GFLOPS em *clusters* de servidores. Este grande aumento de desempenho permitiu que diversas aplicações pudessem ter mais funcionalidades, melhores interfaces gráficas e gerar melhores re-

sultados. Os usuários, por sua vez, passaram a exigir melhorias no *software* uma vez que se tornou comum a evolução do *hardware*, criando um ciclo positivo para a indústria de computadores.

Com o passar dos anos, a maioria dos desenvolvedores de software tem contado com os avanços do hardware para aumentar a velocidade de suas aplicações subjacentes, o mesmo software simplesmente roda mais rápido à medida que cada nova geração de processadores é introduzido. Esta crescente evolução, no entanto, diminuiu desde 2003 devido ao consumo de energia e dissipação de calor, problemas que têm limitado o aumento da frequência do relógio e do nível de atividades produtivas que podem ser realizadas em cada período de relógio dentro de uma única CPU. Virtualmente, todos os fabricantes de microprocessadores passaram a utilizar modelos com múltiplas unidades de processamento, que podem ser conhecidas como *núcleos de processador*, que são utilizados em cada chip para aumentar o poder de processamento. Esta mudança tem exercido um enorme impacto sobre a comunidade de desenvolvedores de software.

Um programa que antes rodava sequencialmente, mesmo com o aumento do número de núcleos de um processador, não terá aumento de performance. É necessário outra abordagem de programação para que se possa tirar proveito dos novos recursos de hardware hoje disponíveis. As aplicações que irão continuar a tirar proveito da melhoria de desempenho a cada nova geração de microprocessadores serão os programas paralelos, em que várias *threads* de execução colaboram para completar o trabalho mais rápido. A prática da programação paralela não é um assunto novo. A comunidade de computação de alto desempenho tem desenvolvido programas paralelos ao longo de décadas. Estes programas são executados em grande escala, em computadores caros. Apenas alguns aplicativos de elite poderiam justificar o uso desses computadores caros, limitando assim a prática da programação paralela a um pequeno número de desenvolvedores de aplicativos. Agora que todos os novos microprocessadores são computadores paralelos, o número de aplicativos que devem ser desenvolvidos como programas paralelos aumentaram drasticamente. Há agora uma grande necessidade para os desenvolvedores de software aprender sobre programação paralela, que será um recurso que será largamente utilizado neste trabalho.

2.3.1 GPUs Como Computadores Paralelos

Desde 2003, a indústria de semicondutores tem investido em duas principais abordagens para o projeto de microprocessador. A abordagem multicore procura manter a velocidade de execução de programas sequenciais enquanto executam em vários núcleos. Os multicores começaram como dois processadores de núcleo, com o número de núcleos de aproximadamente dobrando a cada geração de processamento de semicondutores. Um exemplo atual é o recente Intel microprocessador® Core™ i7, que tem seis núcleos de processamento, cada um dos quais independentes, o processador em questão implementa o conjunto completo de instruções x86. O microprocessador suporta a tecnologia *hyperthreading* com duas *threads* de hardware e é projetado para maximizar a velocidade de execução de programas sequenciais.

No entanto, a trajetória dos processadores multicores ficou voltada principalmente para *throughput (vazão)* de execução das aplicações paralelas. As aplicações paralelas passaram a utilizar um conjunto de threads de forma colaborativa a fim de aumentar a vazão de trabalho de acordo com a quantidade de núcleos. Mas em paralelo com a evolução dos processadores, veio a evolução das GPUs, que passaram a ter centenas de núcleos mas com outra abordagem de processamento, tendo o foco nas instruções matemáticas. Com o lançamento do *CUDA (Compute Unified Device Architecture)*, que foi um facilitador da utilização dos recursos da GPU, uma outra abordagem de programação paralela surgiu aumentando mais ainda as opções dos desenvolvedores.

2.3.2 Compute Unified Device Architecture (CUDA)

Cinco anos após o lançamento da série 3 da linha de placas gráficas GeForce da NVIDIA, em 2006 foi lançado o CUDA 1.4 para placa gráfica GeForce 8800 GTX que foi a primeira GPU a utilizar CUDA. Esta arquitetura incluiu vários novos componentes projetados exclusivamente para computação na GPU e teve como objetivo aliviar muitas das limitações que impediram que os processadores gráficos anteriores de ser legitimamente úteis para computação de propósito geral. Ao contrário das gerações anteriores que os recursos de computação foram particionados em vértices e *pixel shaders*, a arquitetura CUDA incluiu um *shader pipeline* unificado, permitindo que cada unidade lógica e aritmética (ULA) sobre o chip possa ser empacotada por

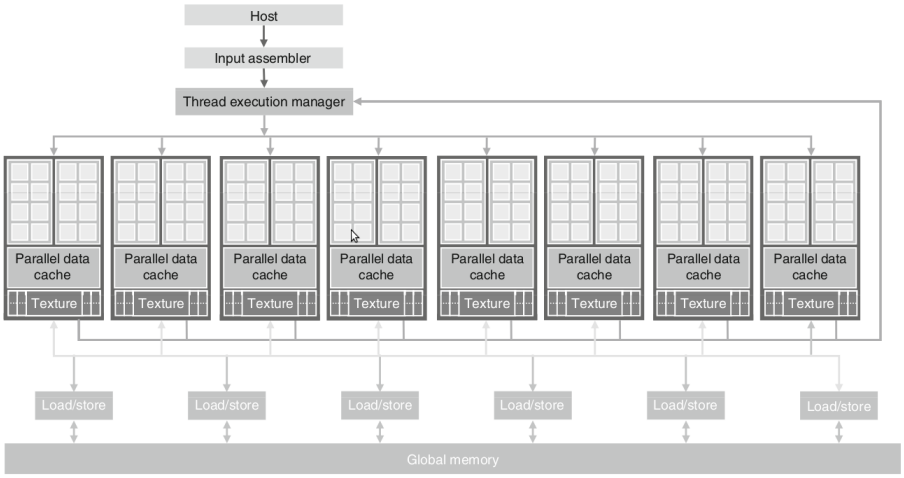

Figura 2.3: Arquitetura CUDA. Figura retirada do livro Programming Massively Parallel Processors A Hands-on Approach [16].

um programa que pretenda realizar computações de uso geral. A NVIDIA destina esta nova família de processadores gráficos para ser usada para computação de propósito geral, essas ULAs foram construídos para cumprir os requisitos da IEEE, como por exemplo a precisão simples aritmética de ponto flutuante e também foram projetadas para usar um conjunto instruções para a computação de uso geral e não especificamente para aplicações gráficas. Além disso, nas unidades de execução na GPU foram permitidas a leitura arbitrária e permissão de gravação na memória, bem como o acesso a um cache gerido por software conhecido como memória compartilhada. Todas estas características da arquitetura CUDA foram adicionadas a fim de criar uma GPU que pudesse superar as expectativas de computação, além de executar bem em tradicionais tarefas gráficas.

2.3.3 Arquitetura CUDA

A figura 2.3 mostra uma típica arquitetura de uma GPU que utiliza CUDA. Ela é organizada em uma matriz de *streaming multiprocessors* (**SM**s) altamente segmentada. Na figura 2.3, dois SMs formam um bloco de construção, no entanto, o número de SMs em um bloco de construção pode variar de acordo uma geração de GPUs CUDA para outra geração. Além disso, cada SM na figura 2.3 tem um número de *streaming processors* (**SP**s) que compartilha o controle lógico e o cache de instrução. Cada GPU atualmente pode possuir até 6 *gigabytes of double data rate* (GDDR)

DRAM, conhecido como **memória global** (*global memory*) como pode ser visto na figura 2.3. Estes GDDR DRAM diferem das DRAM do sistema da placa-mãe CPU, eles são essencialmente a memória que é usada exclusivamente para trabalhar na placa gráfica. Para aplicações executando diretamente na GPU as transferências de dados na memória da GPU são extremamente rápidas, no entanto quando há a necessidade de acessar a memória principal do conjunto placa-mãe CPU cuja latência aumenta, podendo ocasionar gargalos nas aplicações em alguns casos.

O G80, que introduziu a arquitetura CUDA tinha 86.4 GB/s de largura de banda de memória, além de uma largura de banda de comunicação de 8 GB/s com a CPU. Uma aplicação CUDA pode transferir dados da memória do sistema em 4 GB/s e ao mesmo tempo enviar dados de volta para a memória do sistema, a 4 GB/s. Ao todo, na combinação do dois há um total de 8 GB/s. A largura de banda de comunicação é muito mais lenta do que a largura de banda de memória e pode parecer uma limitação, no entanto, a largura de banda do barramento *PCI Express* é comparável a largura de banda do front-side bus (FSB), que é o canal de comunicação entre a CPU e a memória do sistema, por isso não é realmente o limitador que parece à primeira vista. É esperado que a largura de banda de comunicação evolua assim como a largura de banda entre a memória do sistema e a CPU evolua.

O massivamente paralelo chip G80 tem 128 SPs (16 SMs, cada um com 8 SPs). Cada SP tem uma unidade de multiplicar-somar (MAD) e uma unidade de multiplicação adicional. Com 128 SPs, que é um total de mais de 500 gigaflops. Além disso, funções especiais executam operações de ponto flutuante como, por exemplo, a raiz quadrada (SQRT), bem como outras funções transcendentais. Com 240 SPs, o chip GT200 excede um teraflops. Como cada SP utiliza massivamente threads, este pode executar milhares de threads por aplicativo. Um bom aplicativo funciona tipicamente com 5000 a 12.000 threads executando simultaneamente nesse chip. Para aqueles que estão acostumados a utilização de threads, note que os processadores da Intel suportam 2 ou 4 threads, dependendo do modelo da máquina, por núcleo. O chip G80 suporta até 768 threads por SM, que soma cerca de 12.000 threads para este chip. O mais recente chip GT200 suporta 1024 threads por SM e até cerca de 30.000 threads. Um detalhe importante é que duas ou até três dessas placas gráficas (precisam ser idênticas) podem ser combinadas no mesmo sistema através da tecno-

logia chamada SLI (*Scalable Link Interface*), essa combinação aumenta mais ainda os recursos de computação do sistema.

2.3.4 Estrutura de um Programa

Para iniciar o desenvolvimento em CUDA é necessário utilizar o SDK (*Software Development Kit*) disponibilizado pela NVIDIA, que é um pacote de desenvolvimento contendo um compilador chamado *nvcc*, que utiliza o compilador gcc (no caso do linux) para compilar os programas em CUDA. Pode-se imaginar CUDA como uma extensão para a linguagem C. Primeiramente é necessário configurar os drivers disponibilizados pela NVIDIA para a utilização do CUDA, em seguida o SDK pode ser instalado e após estes dois passos já é possível escrever ou adaptar algum programa escrito em C para a utilização do CUDA e aproveitar suas funcionalidades.

É importante ressaltar algumas nomenclaturas de um programa em CUDA, a CPU normalmente é chamada de *Host*, e o conceito de *host memory* se refere a memória RAM principal do computador. A GPU normalmente é chamada de *Device* e sua memória é chamada de *device memory* ou *memória global*. Os dados que estão na host memory podem ser transferidos via barramentos de dados PCI-e (PCI Express) para a device memory ou o caminho oposto também pode ser feito. A operação de transferência de dados sempre será coordenada pela CPU.

Para que um código paralelo execute na GPU é necessário fazer uso de funções chamadas *kernels*. Um kernel é uma função que é chamada pelo host e é executada no device, ou seja, é uma função chamada pela CPU que será executada nos SMs da GPU. Em algumas arquiteturas apenas um kernel pode ser chamado por vez, mas na atual arquitetura *Fermi*[17] da NVIDIA diversos kernels simultâneos podem ser chamados.

Existe uma grande diferença entre as threads convencionais que executam na CPU e as *CUDA-Threads*, que são as threads que rodam na GPU. As CUDA--Threads são muito menos onerosas com relação a sua criação e com a sua troca de contexto quando comparadas as threads da CPU. Milhares de CUDA-Threads podem ser criadas em poucos ciclos de processamento com um baixo overhead, a troca de contexto entre as threads tem praticamente custo zero segundo informação da NVIDIA.

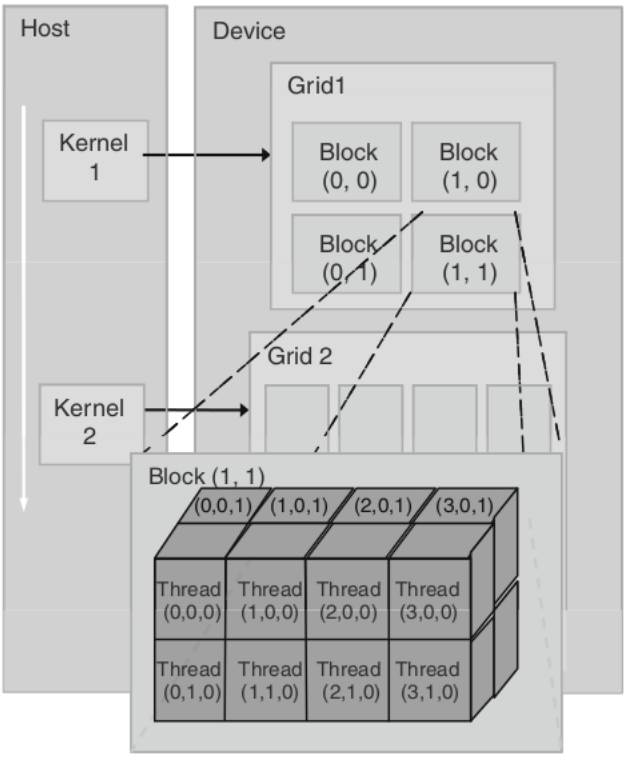

Figura 2.4: Organização das threads em CUDA.

[3]

As CUDA-Threads são organizadas em Grades (*Grids*), Blocos (*Blocks*) e *Threads*, onde cada unidade pode ser dividida em até três dimensões. Na figura 2.4 podemos como são organizadas as CUDA-Threads. Cada grid possui um conjunto de blocos e cada bloco possui um conjunto de threads. Essa hierarquia de grid, bloco e thread sempre será mantida, o que pode ser alterado na chamada do kernel são parâmetros que indicarão a quantidade grids, blocos e threads com relação a quantidade e dimensões que serão utilizados.

Para um melhor entendimento, iremos mostrar um simples exemplo de uma soma de vetores em um código sequencial em C e o código utilizando CUDA. O código da soma de vetores sequencial pode ser visto no código 2.1 e o código 2.2 faz a mesma operação, só que utilizando CUDA.

```
1 void soma_sequencial( int *a, int *b, int *c ) {
  int tid = 0; // CPU zero
3
  while (tid < N) {
```

[3]Figura 2.4 retirada do livro Programming Massively Parallel Processors A Hands-on Approach.

```
5       c[tid] = a[tid] + b[tid];
        tid += 1; // utilizamos apenas uma CPU, entao incrementaremos um
            por um
7     }
  }
```

Código 2.1: Soma de vetores sequencial.

```
1 __global__ void soma_cuda( int *a, int *b, int *c ) {
    // id da thread que executara o metodo kernel
3   // tid = (Quantidade de Blocos na Dimensao X do Grid) * (Posicao X do
        Bloco Atual) + Posicao X da Thread do Bloco Atual
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
5   while (tid < N) {
      c[tid] = a[tid] + b[tid];
7     tid += blockDim.x * gridDim.x;
    }
9 }
```

Código 2.2: Soma de vetores utilizando CUDA.

[4]

É importante ressaltar que no código 2.2, os grids, blocos e threads possuem apenas uma dimensão, que é a dimensão X. Cada thread recebe um id único na variavél *tid*, este id é calculado de acordo com o id da thread dentro do bloco e o id do bloco dentro do grid. É levada em consideração também, a dimensão que cada grid, bloco thread está utilizando, no exemplo citado estamos utilizando somente a dimensão X. A diretiva *__global__* é utilizada quando definimos o método kernel, ou seja, este será o método dará início a execução da aplicação na GPU.

Além disso, métodos kernel tem a capacidade de chamar outros métodos que estejam definidas no mesmo espaço de memória do dispositivo. Esses métodos são definidos da mesma forma que os métodos kernel, mas usando a diretiva *device* no lugar de *global*. A CPU somente pode chamar diretamente métodos kernel, e métodos kernel ou device podem chamar outros métodos device.

Definido o método kernel, agora os próximos passos serão alocar espaço em memória na GPU para os vetores, copiar os vetores para a memória global da GPU,

[4] O exemplo citado foi retirado e modificado do livro CUDA By Example: An Introduction to General-Purpose GPU Programming [9].

chamar o método kernel e em seguida copiar o resultado da memória global da GPU para memória principal do sistema. Os passos descritos podem ser vistos no código 2.3.

```
#define N (33 * 1024)

int main( void ) {
  int a[N], b[N], c[N];
  int *dev_a, *dev_b, *dev_c;
  // aloca memoria na GPU para os 3 vetores.
  cudaMalloc( (void**)&dev_a, N * sizeof(int) );
  cudaMalloc( (void**)&dev_b, N * sizeof(int) );
  cudaMalloc( (void**)&dev_c, N * sizeof(int) );
  // preenche os vetores a e b utilizando a CPU.
  for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i * i;
  }
  // copia os vetores a e b para a memoria da GPU.
  cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

  // chamada kernel especificando 128 blocos e 128 threads por bloco.
  soma_cuda<<<128,128>>>( dev_a, dev_b, dev_c );

  // copia o vetor c de volta da GPU para a CPU.
  cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

  // desaloca a memoria alocada na GPU.
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_c );
  return 0;
}
```

Código 2.3: Exemplo chamada do método kernel e cópia de dados.

Podemos observar que no exemplo apresentado que todas as threads trabalharam de forma totalmente independente, lendo da memória global e escrevendo na memória global. Threads de um mesmo bloco possui alguns recursos de colaboração entre

elas, como por exemplo, possuem um espaço comum de memória chamado *shared memory*, que é uma memória com uma latência muito menor que a da memória global. Existem outros recursos que também podem ser utilizados por threads de um mesmo bloco, podendo ser destacado a sincronização por barreira e as instruções atomicas.

2.3.5 Pinned Memory e Shared Memory

Outros recursos do CUDA que são importantes serem lembrados e que foram largamente utilizados neste trabalho são os de *pinned memory* e os de memória compartilhada (*shared memory*). No exemplo anteriormente mostrado, podemos ver que quando alocamos a memória na GPU utilizamos o comando *cudaMalloc* e na memória da CPU nós utilizamos o comando *malloc*. No entando, o CUDA nos oferece outro método para alocação de memória na CPU, o *cudaHostAlloc*. Mas por quê utilizar o *cudaHostAlloc* se o *malloc* do C sempre funcionou muito bem? No entanto, existe uma grande diferença entre a memória alocada pelo *malloc* e o *cudaHostAlloc*. A função *malloc* da biblioteca do C aloca o espaço na memória da CPU, mas em algum momento esses blocos podem ser movimentados para o disco, o que fará com que o tempo de transferência de dados entre a CPU e a GPU aumente drasticamente. Para resolver este problema nós utilizamos a *page-locked host memory* ou as vezes chamada de *pinned memory*, que é a memória alocada pela função *cudaHostAlloc*. Uma vez alocada a pinned memory, o sistema operacional garante que os blocos que ali foram alocados não serão movimentados. Com a garantia que os dados não serão movimentados, o sistema operacional permite que a aplicação acesse diretamente o endereço físico da memória, nesse caso, a GPU conhecendo o endereço físico ela pode utilizar o *direct access memory* (DMA) para trocar dados entre o host e o device. Utilizando esse recurso, além da CPU ficar livre, o gargalo da taxa de transferência entre a memória do host e a memória do device será a velocidade de transferência do barramento PCI-e. Para ser ter uma ideia, utilizando uma GeForce GTX 285 quando copiamos dados da host memory para a device memory quando utilizamos o comando *malloc* a taxa de transferência chega a 2.77 GB/s, mas quando utilizamos pinned memory a taxa de transferência chega a 5.11 GB/s. A princípio, pode parecer ótimo trocar todos os *malloc* por *cudaHostAlloc*, mas essa

função deve ser usada com cuidado, pois como a memória virtual não será utilizada, rapidamente a memória do sistema poderá se esgotar se a devida desalocação não for feita, causando erros de execução na aplicação.

Outro recurso de memória muito importante em CUDA é a *shared memory* ou memória compartilhada, que nada mais é que um cache gerenciado por software que pode ser acessado por threads de um mesmo bloco. Para fazer uso da memória compartilhada basta declarar __*shared*__ antes da declaração de uma variável. O acesso a memória compartilhada é muito mais veloz que o acesso a memória global. Através da memória compartilhada nós podemos efetuar diversas operações conhecidas na computação paralela, como por exemplo, barreiras de sincronização, operações de redução e etc. Mas não existem apenas vantagens, assim como na computação paralela, como são diversas threads acessando a mesma região de memória é necessário gerenciar manualmente os acessos simultâneos, isso pode ser feito utilizando variáveis com exclusões mútuas ou com instruções atômicas, que de fato reduzirá o desempenho da aplicação. O ideal para programas paralelos é que as threads acessem os dados de forma independente, fazendo com o desempenho seja melhor, mas em todo caso, no CUDA há ferramentas para tratar os casos acesso simultâneo.

A grande vantagem de se utilizar memória compartilhada é quando as threads de um mesmo bloco fazem vários acessos a dados na memória global que já foram lidos por uma thread do mesmo bloco. Quando uma thread faz um acesso a memória global ela poderá copiar este dado para memória compartilhada para que as outras threads não precisem ir até a memória global, isso reduzirá consideravelmente o tempo de execução da aplicação. Para se ter uma ideia, a latência para acessar a memória compartilhada está na mesma ordem de tempo de acesso aos registradores. Apesar da sua rápida velocidade o seu tamanho é bem limitado, sendo necessário gerenciar manualmente a transferência de dados entre memória global.

2.3.6 Arquitetura Fermi

A arquitetura Fermi foi lançada em abril de 2010 trazendo suporte para novas instruções para programas em C++, como alocação dinâmica de objeto e tratamento de exceções em operações de *try* e *catch*. Cada SM de um processador Fermi possui

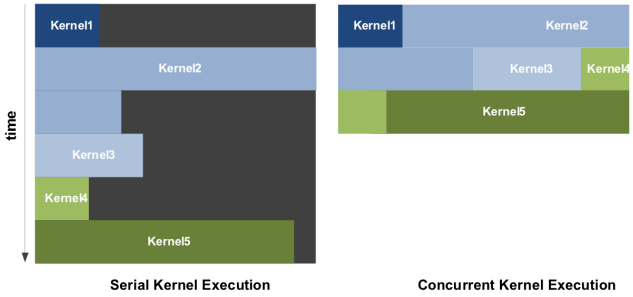

Figura 2.5: Comparação de execução de um kernel sequencial e um concorrente. Figura retirada do artigo [17] da NVIDIA.

32 CUDA cores. É possível fazer 16 operações de precisão dupla por SM em cada ciclo de clock. Diversas outras melhorias foram trazidas pela arquitetura Fermi, podemos mencionar como mais importantes a performance da precisão dupla que foi elevada, as operações atômicas agora executam 20 vezes mais rápido e a grande novidade que é utilizada neste trabalho, a execução de kernels concorrentes.

A arquitetura Fermi dá suporte a execução de kernels concorrentes, onde diferentes kernels de uma mesma aplicação podem executar na GPU ao mesmo tempo. Na figura 2.5 podemos ver um exemplo de comparação da execução sequencial com a concorrente. A execução de kernels concorrentes permite que aplicações executem um número de pequenos kernels para utilizarem todo o potencial da GPU. Kernels de diferentes aplicações irão executar sequencialmente. Neste trabalho veremos um exemplo de implementação deste recurso de paralelismo. Recentemente, a NVIDIA lançou a arquitetura *Keppler*, mas como já tínhamos iniciado nosso trabalho e era uma tecnologia bem recente não iremos utilizá-la.

2.3.7 Múltiplas GPUs

A utilização de mais de uma GPU é possível desde que a placa-mãe do sistema tenha suporte ao SLI, podendo então ser instaladas nos slots PCIe disponíveis duas ou até três placas gráficas idênticas, como pode ser vista na figura 2.6.

Para utilizar múltiplas GPUs em CUDA, primeiramente é necessário criar threads na CPU para que estas possam fazer as chamadas aos respectivos kernels que irão executar nas GPUs. Essas threads podem ser criadas utilizando um variedade de bibliotecas de criação de threads, neste trabalho nós utilizamos a biblioteca *CUTThread* que vem junto com o SDK disponibilizado pela NVIDIA e em alguns

Figura 2.6: Um PC com três placas Geforce 8800 GTX e uma fonte de 700W. Figura retirada do livro Programming Massively Parallel Processors A Hands-on Approach.

casos nós utilizamos a biblioteca *OpenMP* [6], que utiliza diretivas de compilação para paralelizar laços de repetição. O OpenMP será muito útil nos casos onde iremos fazer chamadas simultâneas ao kernel utilizando a arquitetura Fermi.

Em uma aplicação que irá utilizar múltiplas GPUs, cada thread da CPU responsável por uma GPU deverá se encarregar de "chavear" para a respectiva GPU através do comando *cudaSetDevice(deviceID)*, onde deviceID é o id da GPU que será utilizada. Para saber quantas GPUs estão disponíveis no sistema, basta utilizar o comando *cudaGetDeviceCount* e então será retornado quantidade de GPUs detectadas pelo sistema. Cada thread da CPU deverá também ser responsável por alocar e desalocar a memória na respectiva GPU além de gerenciar a troca de dados entre a CPU e a respectiva GPU. Cada GPU terá os mesmos recursos citados nas seções anteriores, com a diferença que agora as threads na CPU deverão gerenciar a divisão e a alocação de trabalho para cada GPU. Com esse recurso, nós podemos cada vez mais escalar nossas aplicações desenvolvidas em CUDA.

2.3.8 Algumas Bibliotecas e Outros Recursos

A quantidade recursos que o CUDA oferece são tão grandes que este trabalho não será suficiente para mencioná-los todos aqui, então nesta seção nós iremos mencionar alguns recursos e bibliotecas disponíveis em CUDA que nós utilizamos neste trabalho e se for o caso o porquê nós não utilizamos.

Um recurso comumente utilizado em CUDA são as *Streams*, que representam uma

fila de operações à serem executada na GPU em uma ordem específica. Com esse recurso nós podemos adicionar operações como, por exemplo, chamadas ao kernel, cópias de memória e início de eventos e de parada na stream. A ordem na qual cada operação é adicionada à stream é a ordem na qual ela será executada. Podemos imaginar cada stream como se fosse uma tarefa na GPU e a cada oportunidade a GPU as executará. Em nosso trabalho nós não utilizamos esse recurso, pois a complexidade de nosso código já estava se tornando muito alta e também como nós já estamos utilizando chamadas simultâneas ao kernel desempenho não seria tão relevante.

Um outro recurso importante que precisa ser mencionado e que simplifica a implementação em CUDA é o *Zero-Copy Host Memory*. Este recurso é utilizado em conjunto com o conceito de pinned memory, quando alocamos a memória no host com o comando *cudaHostAlloc* mas passando o parâmetro *cudaHostAllocMapped* o sistema operacional permite a GPU mapeie fisicamente os endereços de memória desta região, ou seja, permite que de dentro da função kernel seja possível acessar a região de memória do host. Esse recurso simplifica o desenvolvimento já que não será preciso utilizar comandos de cópia de dados do host para o device e vice-versa. Mas em nosso trabalho não utilizamos este recurso pela questão de desempenho, como iremos trabalhar com uma volume de dados grande, a cópia sequencial de blocos do host para device acaba sendo mais rápida do que acessos aleatórios de dentro do kernel à espaços de memória do host.

Acompanhando o SDK do CUDA 4.0 existe uma biblioteca chamada *Thrust* [18], cuja primeira versão foi disponibilizada para o público em maio de 2009 e foi desenvolvida pela própria NVIDIA. Utilizando a biblioteca Thrust é possível, por exemplo, instanciar vetores diretamente na memória do device e efetuar a cópia dos dados da host memory para a device memory em uma linha de código apenas, sem a necessidade de explicitamente realizar todas essas operações. Além disso, a biblioteca Thrust oferece uma série de algoritmos implementados de forma paralela na GPU prontos para serem utilizados. A thrust permite efetuar transformações lineares entre vetores e possui algoritmos de ordenação prontos, mas em nosso trabalho nos limitamos a utilizar somente o algoritmo de ordenação de arrays na GPU, as outras operações por questão de desempenho nós não iremos utilizar.

Outra biblioteca que possui operações básicas de álgebra linear está disponível no pacote do CUDA 4.0, o seu nome é *CUBLAS* e também foi desenvolvido pela NVIDIA. O CUBLAS é uma implementação da biblioteca *BLAS (Basic Linear Algebra Subprograms)* em CUDA. Esta permite ao usuário acessar os recursos computacionais da GPU, mas não é automaticamente paralelizável entre múltiplas GPUs. Para utilizar o CUBLAS, a aplicação precisa alocar as matrizes ou vetores necessários na memória da GPU, preenche-los com os dados, chamar a sequência de funções designada pelo CUBLAS e então os resultados calculados na GPU são copiados de volta para a memória do host. Apesar dessas facilidades, nós não utilizamos esta biblioteca devido à algumas peculiaridades de nosso problema que nos levariam a fazer diversas customizações, sendo mais viável implementar nossas próprias operações.

Recentemente a NVIDIA lançou o CUDA 5.0 que trouxe como principais melhorias as ferramentas de desenvolvimento, plugins para auxílio de debug e a melhora nos recursos de transferência de dados entre a GPU e os dipositivos PCI-e *(RDMA)*. Neste trabalho utilizaremos o CUDA 4.0, pois quando foi lançado o CUDA 5.0 já tínhamos executados nossos experimentos.

Capítulo 3

Metodologia

Neste capítulo iremos explicar detalhadamente nossa solução para o problema apresentado e sua implementação. Iremos apresentar uma arquitetura onde utilizaremos um FC com base no item como serviço utilizando os conceitos apresentados no capítulo anterior. Nós mostraremos como foi realizada a leitura dos *datasets* utilizados e, além disso, faremos uma análise prévia da distribuição das notas e o espaço que será ocupado em memória. A arquitetura que será apresentada será dividida em gerenciadores de serviço que terão funções similares ao algoritmo de FC apresentado no capítulo anterior. Para cada gerenciador nós mostraremos as suas funções, um pseudocódigo de alto nível e sua implementação utilizando CUDA que ficará no anexo A deste trabalho.

3.1 Arquitetura de um Filtro Colaborativo como Serviço

Atualmente diversos sites de comércio eletrônico (por exemplo, Amazon, Submarino, Americanas) utilizam algum sistema de recomendação para sugerir produtos à seus usuários. Seria grande valia se algum serviço pudesse sugerir de maneira precisa e rápida, produtos que sejam de interesse do usuário de forma que ele efetivamente acabe comprando. Manter uma lista de recomendação atualizada pode fazer com que a loja realize mais vendas e consequentemente tenha mais lucro. Para gerar uma lista de produtos correlacionados há a necessidade de muito processamento e essa operação pode levar muito tempo. Seria interessante se esse processamento

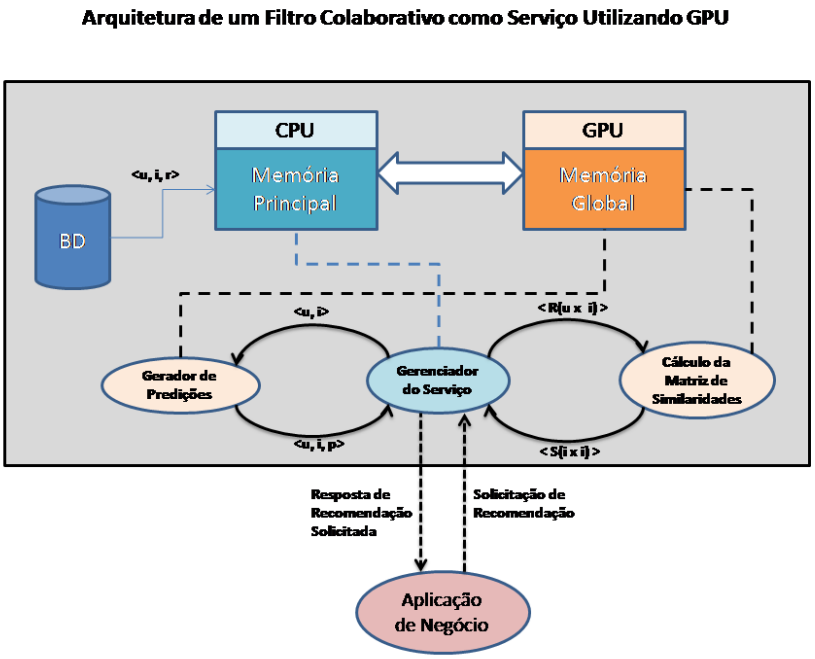

Figura 3.1: Arquitetura de funcionamento dos serviços de FC utilizando a GPU.

fosse feito em outro local e fosse retornada apenas a recomendação solicitada, desonerando o site de um carga desnecessária. Em nosso trabalho iremos propor uma arquitetura de filtro colaborativo sendo utilizado como serviço. Onde, toda vez que alguma aplicação necessite de algum tipo de recomendação esta poderá solicitar à um serviço e este responderá a requisição com uma listagem itens recomendados ou simplesmente a recomendação que um usuário poderá dar em um item. Por trás deste serviço existirão outros serviços que executarão parte do algoritmo de FC que auxiliarão esse *Gerenciador de Serviço*. O Gerenciador de Serviço irá utilizar o serviço de *Cálculo da Matriz de Similaridades* para atualizar a matriz que contém os dados sobre as similaridades dos itens e o serviço *Gerador de Predições* para solicitar predições de nota que um usuário poderá dar à um item ou simplesmente uma lista de recomendação de itens para um usuário. Cada serviço terá o seu trabalho feito em uma GPU de forma que o tempo de resposta possa ser diminuído. O funcionamento desta arquitetura pode ser visto na figura 3.1, onde u é o identificador do usuário, i é o identificador do item e p é a predição gerada pelo serviço. A matriz de avaliações lida do *dataset* é representada por R e a matriz de similaridades é representada por S .

3.1.1 Gerenciador do Serviço

Em nosso modelo nós propomos um *Gerenciador de Serviço* para receber as solicitações de recomendação de alguma aplicação externa que necessite deste serviço de recomendação de itens, além disso, este gerenciador ficará responsável por dar a resposta a aplicação que o solicitou. Também será responsabilidade deste serviço, a leitura dos dados históricos de recomendação de itens e usuários (*dataset*) e criação da matriz R. Para finalizar, a chamada para a criação da matriz de similaridades e as solicitações de predição ao *Gerador de Predições* também será atribuição deste serviço. No geral, este serviço além de ser a porta de entrada das requisições de recomendação, este também irá gerenciar a chamada aos demais serviços.

3.1.2 Cálculo da Matriz de Similaridades

Este serviço será responsável receber a matriz *Item X User* representada por R do *Gerenciador do Serviço* e efetuar o cálculo da matriz de similaridades Item X Item representada por S utilizando a GPU. Também será atribuição deste serviço, a divisão da matriz R em partes menores de forma que sejam suportadas no espaço da memória da GPU. As diversas cópias de dados entre a memória principal e a memória global da GPU e as chamadas ao *kernel* serão realizadas também por este serviço.

3.1.3 Gerador de Predições

Neste serviço será aplicado o algoritmo de geração de predição com base na matriz de similaridades. As requisições à este serviço irão acontecer de acordo com a quantidade de usuários que irão necessitar de recomendações. No caso de uma simples predição, é uma operação de tempo de reposta curto, mas no caso de uma listagem de recomendação baseada no algoritmo top-n pode ser uma operação de tempo bem maior, que poderia ser executada em paralelo na CPU ou na GPU. Imagine um caso onde vários usuários solicitam uma listagem de recomendação de itens, isso pode ter um tempo de reposta alto se o processamento for "online" e sequencial. Neste serviço iremos apresentar uma implementação paralela do algoritmo de predição tanto para uma listagem de usuários e itens onde uma lista de recomendações é retornada

quanto para uma lista de recomendações solicitada pelo algoritmo top-n.

3.2 Implementação

Nesta seção primeiramente iremos fazer uma análise prévia dos *datasets* que iremos utilizar e em seguida detalhar a implementação de cada serviço em nível de algoritmo e nível de código.

3.2.1 A Leitura do DataSet

Para implementarmos um algoritmo de filtro colaborativo, primeiramente há a necessidade de ler os arquivos onde se encontram os usuários, os itens e as avaliações dos itens feita pelos usuários. Em seguida, essas informações são carregadas em uma matriz na memória principal. Nessa matriz, na qual passamos a chamar de $\boldsymbol{R}$, as linhas serão representadas pelo código do item, as colunas serão representadas pelo código do usuário e o conteúdo da *matriz(i,u)* será o valor que representará a avaliação que o usuário u deu ao item i. Na maioria casos, grande parte da matriz será composta por zeros. Por exemplo, no dataset de dez milhões de avaliações do Movielens [4] existem 10.681 itens e 69.878 usuários, então a matriz **R** ocupará 746.366.918 posições onde apenas dez milhões de posições serão diferentes de zero, ou seja, 98,7% da matriz é igual a zero. Portanto, a matriz **R** do dataset do movielens ocupará cerca de 2.9 gigabytes da memória RAM se considerarmos o tipo da nota como *float* (4 bytes). O mesmo pode ser visto no dataset do netflix [5], neste existem 480.189 usuários e 17.770 filmes e a matriz ocupará cerca de 8.532.958.530 de posições, se considerarmos que cada posição ocupa 4 bytes (tamanho de um tipo *float*), a matriz ocupará cerca de 34 gigabytes na memória RAM. Para reduzir esse espaço drasticamente, iremos considerar a nota como um inteiro do tipo *char* que em C ocupa apenas 1 byte. Então a matriz de avaliações **R** do dataset do netflix passará a ocupar cerca de 8,5 gigabytes. Assim como no movielens grande parte da matriz **R** quando carregamos o dataset do Netflix é zero, cerca de 98,8% dos elementos da matriz. Para os elementos da matriz **R** que não são zero, podemos ver a distribuição de notas dos datasets do Movielens e Netflix nas figuras 3.2 e 3.3 respectivamente.

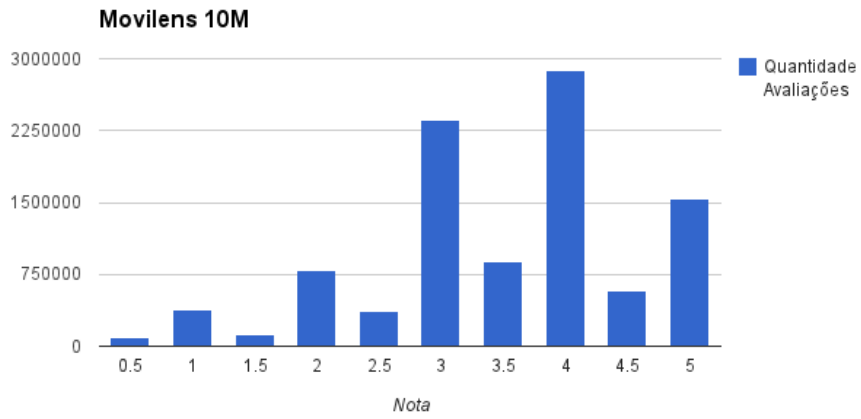

Figura 3.2: Histograma de Notas do Dataset do Movielens.

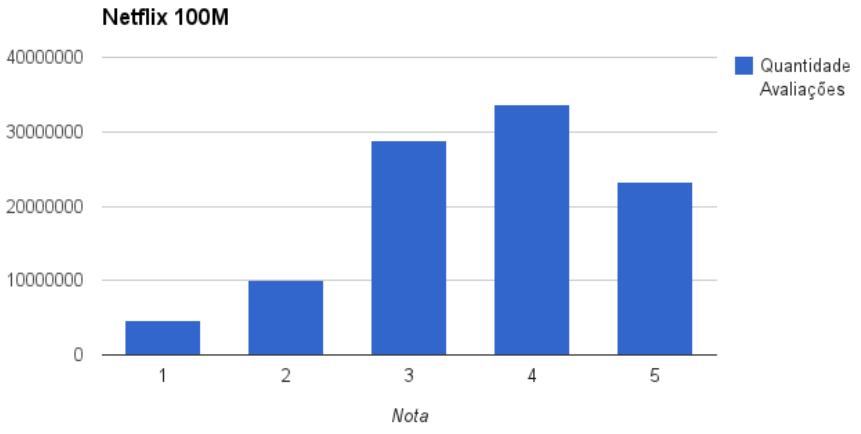

Figura 3.3: Histograma de Notas do Dataset do Netflix.

Neste trabalho poderíamos utilizar matrizes esparsas para armazenar esses dados já que poucas posições das matrizes são preenchidas com valores diferentes de zero. No entanto, estamos interessados em obter desempenho nas operações de matrizes e a utilização deste tipo de matriz acaba sendo ineficiente no ponto de vista de acesso aos dados da matriz. Para armazenar a nossa matriz $\boldsymbol{R}$ utilizamos apenas um array unidimencional que é bem eficiente para efetuar operações matemáticas na GPU.

3.2.2 Criação da Matriz de Similaridades

Com a matriz $\boldsymbol{R}$ carregada em memória, o próximo passo é calcular a matriz que possui as similaridades entre os itens a qual chamaremos de matriz $\boldsymbol{S}$. Para o cálculo desta matriz é necessário a interação entre dois laços de repetição que irão ocasionar em um algoritmo da ordem de $O(m.n^2/2)$ onde o n é o número de itens e m o número de usuários. Dentro desse laço de repetição há um custo com processamento matemático que irá se repetir diversas vezes como foi mostrado no capítulo anterior na explicação do algoritmo. Em nossa implementação iremos transferir este processamento dos laços para a GPU tentando evitar instruções do tipo *if* para que possamos ter o melhor desempenho possível. O pseudo código do algoritmo sequencial de criação da matriz de similaridades pode ser visto abaixo:

```
procedure cria_matriz_similaridades( S, R )
  Para i:=1 ate NItems
    Para j:=1 ate i
    {
      Para k:=1 ate NUsuarios
      {
          soma_ab += R[i][k] * R[j][k];
          soma_a += R[i][k] * R[i][k];
          soma_b += R[j][k] * R[j][k];
      }
      S[i][j] := soma_ab / raiz( soma_a * soma_b );
      S[j][i] := soma_ab / raiz( soma_a * soma_b );
    }
```

Com base em nosso código sequencial acima, implementamos o nosso código para execução paralela na GPU, mas antes precisávamos tomar alguns cuidados. Primeiramente, a memória da GPU é limitada, normalmente bem menor que a memória RAM da maioria dos computadores atualmente. Em alguns trabalhos este caso não foi considerado, ou seja, foi considerado que existiria espaço suficiente na memória global da GPU para a matriz $\boldsymbol{R}$. No nosso trabalho estamos considerando um algoritmo escalável, ou seja, a matriz $\boldsymbol{R}$ inteira pode não caber na memória global da GPU. Nesse caso, foi necessário fazer uma implementação onde fosse possível copiar parte da matriz $\boldsymbol{R}$ para memória global da GPU, executar o algoritmo paralelo na GPU e em seguida copiar o resultado de volta para memória principal na parte da matriz $\boldsymbol{S}$ a qual teve sua parte calculada.

Primeiramente para fazermos uso da GPU, foi necessário criar um método de chamada ao kernel, que é um método onde é iniciada a execução do programa na GPU. Este método pode ser visto abaixo através de seu pseudo-código e seu código no anexo A.1:

```
kernel_calcula_similaridade( linhaA[], linhasB[][],
       retSimilaridades, tamLinhaB, numLinhasB )
{
    memoria_compartilhada s_sum_aa;
    memoria_compartilhada s_sum_bb;
    memoria_compartilhada s_sum_ab;
    sum_aa := 0;
    sum_bb := 0;
    sum_ab := 0;
    vectorBase := 0;
    posA := 0;

    inicializa valores de s_sum_aa, s_sum_bb e s_sum_ab com zero;

    Para cada bloco B dentro do grid G
    calcular a similariade entre os vetores linhasA e linhasB[]:
    {
```

```
        Para cada thread T dentro do bloco B
        efetue as operações:
        {
            vectorBase := idBlocoB*tamLinhaB;

            Para pos := (vectorBase + idThreadT) ate
            pos < (vectorBase + tamLinhaB)
            incrementando pos por NroThreadsBlocoB:
            {
                posA := pos - vectorBase;
                sum_aa += A[posA]*A[posA];
                sum_bb += B[pos]*B[pos];
                sum_ab += A[posA]*B[pos];
            }
            s_sum_aa[idThreadT] := sum_aa;
            s_sum_bb[idThreadT] := sum_bb;
            s_sum_ab[idThreadT] := sum_ab;
        }

        Realizar o somatório dos arrays s_sum_aa, s_sum_bb e s_sum_ab
        através de uma operação de redução armazenando o resultado
        no item zero do array;

        Se idThreadT == 0 entao
            retSimilaridades[idBlocoB] :=
                    sqrt( s_sum_ab[0] / s_sum_aa[0] * s_sum_bb[0] );
    }
}
```

A ideia principal do método *kernel_ calcula_ similaridade* é realizar um produto escalar entre o vetor *linhaA* e os vetores de *linhasB* e o resultado dos produtos escalares armazenados em *retSimilaridades*. O algoritmo apresentado fará com que cada produto escalar seja calculado por um bloco, onde cada thread deste bloco

efetuará as operações matemáticas que neste caso são de multiplicação. Após as threads do bloco efetuarem as multiplicações do produto escalar, estas irão efetuar a soma dos resultados e gravar na posição zero dos vetores que estão na memória compartilhada. Para finalizar, as threads de *id* zero de cada bloco irão efetuar as operações de *sqrt* e divisão para gravar o resultado no vetor *retSimilaridades*.

Após a criação do método que faz o processamento de parte da matriz na GPU foi necessário dividir a matriz $\boldsymbol{R}$ em partes menores de forma que houvesse espaço na memória global da GPU para armazena-la. Para dividir a matriz $\boldsymbol{R}$ em partes menores e fazer as respectivas chamadas para o cálculo na GPU foi utilizado o pseudo-código abaixo e o código A.2 do anexo A supondo que temos disponível somente uma placa gráfica:

```
host_cria_matriz_similaridades( R[][], S[][] )
{
    Calcula tamanho da matriz R;
    Calcula em quantas partes serão necessarias dividir R;
    Para cada parte P de R faça:
    {
         Copia P para memoria global da GPU;
         Para i variando até o Número de Itens do Dataset faça:
         {
             Copia R[i] para memória da GPU;
             chama kernel_calcula_similaridade(
                                 R[i],
                                 P,
                                 parteS,
                                 número de usuários do dataset,
                                 numeros de linhas em P );

             Copia resultado da similaridade para S;
         }
    }
}
```

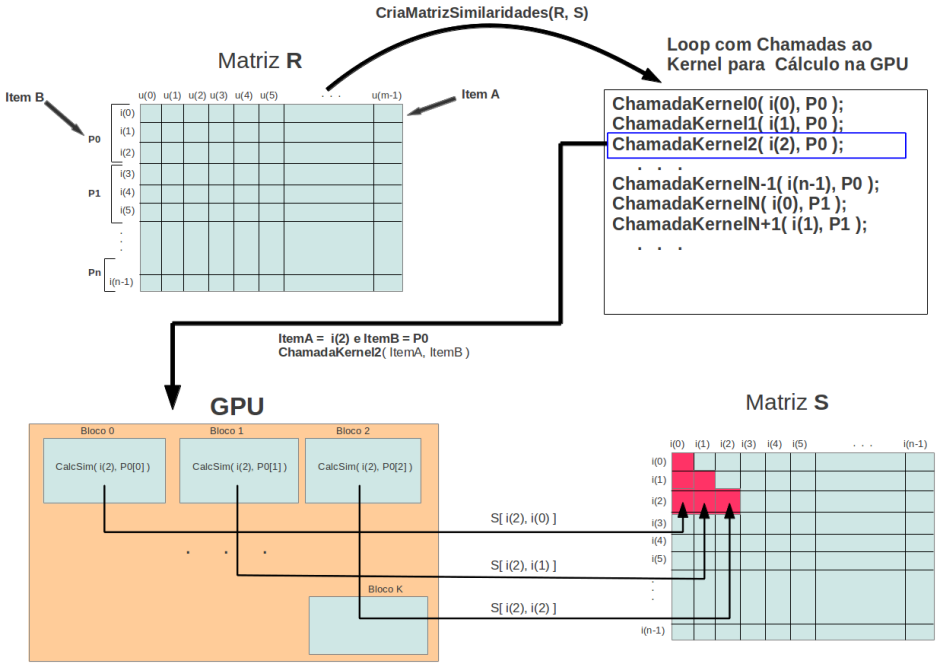

Figura 3.4: Passo a passo do algoritmo paralelo para cálculo da matriz de similaridades utilizando a GPU.

Após as diversas chamadas ao método que calcula a similaridade entre os vetores na GPU, a matriz de similaridades $\boldsymbol{S}$ estará criada. Um exemplo de execução do algoritmo pode ser visto na figura 3.4, supondo que na memória global da GPU só suporte três linhas da matriz $\boldsymbol{R}$ por vez, ou seja, na chamada ao método da GPU são passadas três linhas de $\boldsymbol{R}$ (chamada de item B) e mais uma linha de $\boldsymbol{R}$ (chamada de item A). Na GPU serão feito os cálculos de similaridades das combinações entre os itens A e B.

Na figura 3.4, o processo é iniciado com a chamada do método *host_cria_matriz_similaridades* sendo passados como parâmetros a matriz R e a matriz S que será retornada. Dentro deste método a matriz R será divida em partes (representada na figura por P0, P1, ..., Pn) e haverá um *loop* com chamadas ao kernel, que em nosso exemplo é o pseudocódigo *kernel_calcula_similaridade* e o código A.1 para calcular a similaridade das partes de R. O parâmetro *ItemA* apresenta o laço de repetição que sempre irá varrer todos os *id*'s dos itens do dataset, já o parâmetro *ItemB* representa os *id*'s dos itens que estão na parte da matriz R que foi passada como parâmetro. Pela divisão feita pelo algoritmo *kernel_calcula_similaridade*, na

GPU cada bloco ficará responsável por calcular a similaridade entre dois itens até que todas as combinações entre ItemA e ItemB termine, sendo cada cálculo gravado na respectiva parte da matriz *S*.

Podemos ressaltar que o caso que consideramos anteriormente foi onde utilizamos apenas uma placa gráfica, mas podemos considerar também o caso de possuirmos mais de uma placa gráfica. Neste caso, o método *calcula_similaridade* continuaria o mesmo, a diferença é que no método *cria_matrix_similaridades* teríamos que escalonar as chamadas ao método *calcula_similaridade* direcionando o processamento para as diferentes placas gráficas. Mas primeiramente, precisamos alocar o mesmo espaço que utilizávamos em uma placa gráfica em todas as outras que iremos utilizar. Em seguida, cada parte da matriz $\boldsymbol{S}$ será direcionada para uma placa gráfica. Um detalhe importante é que para cada placa gráfica deverá ser criada uma *Thread* na CPU, onde cada uma irá chamar o método *host_cria_matrix_similaridades_multigpu* mostrado no código A.3 do anexo A, tendo a sua execução independente em cada placa gráfica. O pseudocódigo abaixo mostra de forma simplificada como podemos utilizar mais de uma placa gráfica para criar a matriz $\boldsymbol{S}$. Ao inicializar o CUDA, cada placa gráfica recebe um *Id* que começa por zero, em nosso pseudocódigo esse id será representado pela variável *idDevice*.

```
host_cria_matriz_similaridades_multigpu( idDevice, R[][], S[][] )
{
    Calcula tamanho da matriz R;
    Calcula em quantas partes serão necessarias dividir R;
    Configura no CUDA o Id da placa que será utilizada;
    Para i:=idDevice até i < Nro de Partes com incremento de Nro de GPUs faça:
    {
         P := Parte da matriz R equivalente ao pedaço i;
         Copia P para memoria global da GPU;
         Para i variando até o Número de Itens do Dataset faça:
         {
             Copia R[i] para memória da GPU;
             chama kernel_calcula_similaridade(
                                 R[i],
```

```
                                    P,
                                    parteS,
                                    número de usuários do dataset,
                                    numeros de linhas em P );

            Copia resultado da similaridade para S;
        }
    }
}
```

Para melhor ilustrar, utilizaremos o exemplo anterior onde na memória global da GPU só tem suporte três linhas da matriz $\boldsymbol{S}$. A modificação do exemplo é que agora utilizaremos duas GPUs, neste caso cada GPU terá suporte para três linhas. Repare que na figura 3.5 a matriz $\boldsymbol{S}$ foi segmentada nas cores vermelho e azul que representam respectivamente os itens processados nas GPUs zero e um. Um detalhe importante de ser observado é que as chamadas aos métodos que executam na GPU são feitos por *Threads* criadas na CPU para que a execução possa ser simultânea.

Para facilitar o entendimento de como a matriz $\boldsymbol{S}$ é construída, a matriz da figura 3.6 representa a matriz $\boldsymbol{S}$ se ela tivesse tamanho 6x6, os itens em vermelho representam os itens calculados no primeiro laço do comando de repetição que divide a matriz em partes quando consideramos o exemplo onde somente três linhas da matriz $\boldsymbol{R}$ são suportadas na memória da GPU. Os itens em azul representam os itens calculados na execução da segunda repetição dos algoritmos mostrados.

Nesta seção conseguimos ver com detalhes como foi realizado o cálculo da matriz de similaridades que será utilizado nos experimentos, nos capítulos seguintes poderemos ver a questão da eficiência das implementações apresentadas e apresentaremos os resultados da implementação e uma conclusão.

3.2.3 Ordenação da Matriz de Similaridades

Para diminuir o tempo de execução do algoritmo e gerar uma recomendação com mais precisão, iremos ordenar as linhas da matriz $\boldsymbol{S}$ de forma que os itens de maior similaridade fiquem mais próximos da coluna zero. Para isso, foi necessária alterar a estrutura de dados de forma em que cada item da matriz fosse armazenado o *itemid*

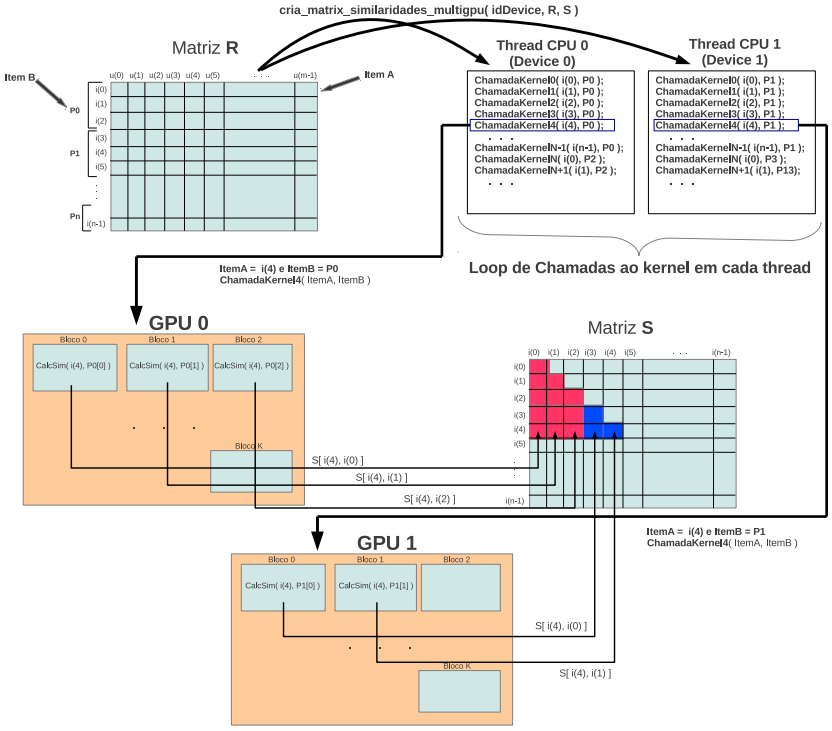

Figura 3.5: Passo a passo do algoritmo paralelo para cálculo da matriz de similaridades utilizando a múltiplas GPUs.

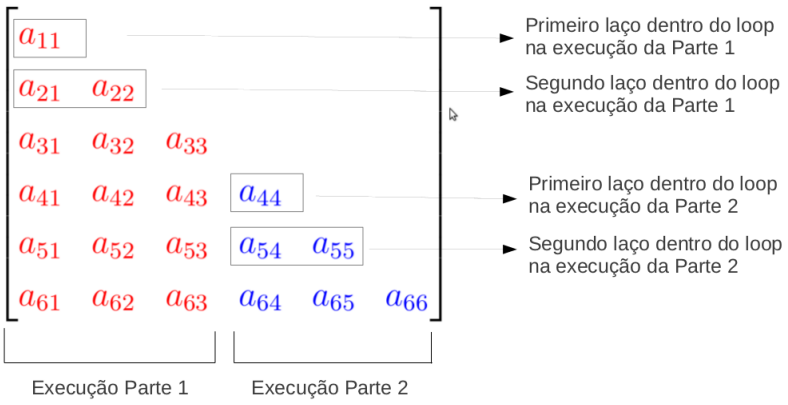

Figura 3.6: Exemplo de construção da matriz S divida em partes quando a memória da GPU não tem espaço suficiente.

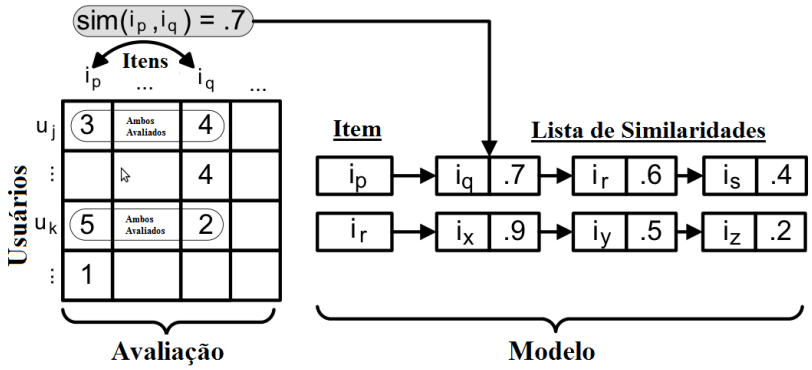

Figura 3.7: Transformação da Matriz R em uma lista ordenada. Figura adaptada do artigo de LEVANDOSKI[1]

e a similaridade do item, um exemplo pode ser visto na figura 3.7. Desta forma foi possível realizar a ordenação da matriz $\boldsymbol{S}$ linha por linha. Utilizamos um algoritmo paralelo que utiliza a GPU para ordenar arrays, este algoritmo pode ser encontrado na biblioteca *Thrust* que vem em conjunto com o pacote *SDK* disponibilizado pela *NVIDIA*. A utilização deste algoritmo nos poupará trabalho de implementar novamente um algoritmo de ordenação na GPU. Neste trabalho, assim como o no trabalho apresentado no artigo [1] iremos utilizar somente os 70 itens mais similares, mas iremos representar este valor pela constante TAM_LISTA_SIMILARIDADES, então a nossa nova matriz de similaridades $\boldsymbol{S}$ agora ficará como a dimensão de *NI-tems x TAM_LISTA_SIMILARIDADES*. Isso nos poupará espaço em memória, já que temos limitação no espaço em memória da GPU além de diminuir o tempo de execução do algoritmo.

3.2.4 Geração das Recomendações

Tendo calculada a matriz de similaridades $\boldsymbol{S}$, agora podemos iniciar o cálculo das recomendações que serão solicitadas em itens que ainda não foram avaliados. Para gerar as recomendações utilizaremos a equação 2.2, que para efeito de comparação é o mesmo método utilizado no artigo [1] para que os resultados possam ser comparados, outro métodos de predição podem ser vistos em [15]. Para efetuar o cálculo de uma recomendação, dado uma solicitação de recomendação que um usuário *userid* daria em um item *itemid*, a recomendação poderia ser calculada com o seguinte algoritmo:

```
procedure gera_recomendacao( userId, itemId, R,  S )
{
  soma_a := 0;
  soma_b := 0;
  nro_computados := 0;
  Para i:=1 ate NItens
  {
    /* Verifica se o Item foi avaliado pelo usuario */
    se R[ userID ][ S[itemId][i].itemId ] > 0 entao
    {
      sum_a += R[ userID ][ S[itemId][i].itemId ] * S[itemId][i].similaridade;
      sum_b += absoluto( S[itemId][i].similaridade );
      nro_computados++;
    }

    se nro_computados >= NRO_MAX_VIZINHOS entao
    {
      termina comando para;
    }

  }

  se sum_b == 0 entao
    retorna 0;
  senao
    retorna sum_a / sum_b;
}
```

No algoritmo podemos verificar que para efeito de cálculo só estamos considerando os itens que o usuário avaliou e estamos limitando a lista de itens avaliados para o tamanho representado por *NRO_MAX_VIZINHOS*. Podemos realizar este "corte" pois efetuamos a ordenação da matriz S de forma que as maiores similaridades fiquem mais próximas do início da linha.

3.2.5 Lista de Recomendação Gerada com Base no Top-n

Em nossa implementação a listagem de itens sugeridos ao usuário será baseada no algoritmo *Top-n*. O algoritmo ordena os itens que o usuário ainda não avaliou utilizando como parâmetro a recomendação gerada pelo algoritmo de filtro colaborativo mostrada na seção anterior. Para gerar uma lista de recomendação para o usuário, primeiramente é necessário saber todos os itens que o usuário ainda não avaliou. Em seguida, é preciso chamar o algoritmo de recomendação para cada item não avaliado e então ordenar a lista de recomendação gerada. O número de itens recomendados exibidos para o usuário no final não terá tanta importância no ponto de vista de custo de processamento, pois necessariamente todos os itens não avaliados terão que passar pelo algoritmo de recomendação, serem ordenados e só então os N itens com maior valor de recomendação serão exibidos. Na implementação utilizando *CUDA* cada bloco será responsável por executar o algoritmo mostrado na seção anterior e cada thread executará um laço de repetição dentro do algoritmo. O pseudocódigo simplificado em CUDA mostra cada passo e pode ser visto no método *kernel_ calcula_ topn* logo abaixo, sua implementação real pode ser vista no código A.4.

```
kernel_calcula_topn( idUsuario, S, R[Somente itens que idUsuario avaliou], Topn )
{
    memoria_compartilhada s_sum_a;
    memoria_compartilhada s_sum_b;

    Para cada bloco B dentro do grid G
    calcular a predição para o usuario idUsuario e o item R[idBlocoB] igual a zero:
    {
        Para cada thread T dentro do bloco B
        efetue a operações:
        {
          similaridade := S[idBlocoB][idThreadT].similaridade;
          rating := R[S[idBlocoB][idThreadT].item_id];

          s_sum_a[idThreadT] := rating*similaridade;
```

```
          s_sum_b[idThreadT] := absoluto( similaridade )*(1&&rating);
        {

        Realizar o somatório dos arrays s_sum_a e s_sum_b através de uma
        operação de redução armazenando o resultado no item zero do array;

        Se idThreadT == 0 entao
            Topn[idBlocoB] :=  s_sum_a[0] / s_sum_b[0];
    }
}
```

Como a arquitetura da placa gráfica que estamos utilizando nos permite chamadas simultâneas (Arquitetura *Fermi* [17]) ao método kernel e a quantidade de itens a serem predizidos tem um tamanho limitado, em nossa implementação fizemos chamadas simultâneas ao método *kernel_ calcula_ topn* utilizando threads da CPU assim como fizemos na implementação em CUDA do cálculo da matriz de similaridades. Esse recurso fará com que a vazão de geração de listas de recomendações aumente.

3.2.6 Geração de Previsão de Nota Por Demanda

O nosso serviço gerador de predições deverá estar preparado para receber uma grande quantidade de solicitações de recomendações por segundo e para isso deverá suportar solicitações de predições "independentes", diferentemente do caso da geração baseada no top-n em que todas as recomendações geradas no algoritmo é do mesmo usuário mudando apenas o item. Se tivéssemos utilizando um algoritmo sequencial, para cada solicitação de recomendação deveríamos fazer uma chamada ao método *gera_ recomendacao* mostrado na seção anterior e mesmo utilizando threads na CPU, uma grande quantidade seria necessária para que tivéssemos uma boa vazão de recomendação. Em nosso serviço gerador de recomendação utilizamos um algoritmo paralelo em CUDA onde uma lista de *(usuário, item)* é fornecida para o serviço e uma lista é retornada com as respectivas recomendações. O algoritmo em paralelo é similar ao algoritmo de geração de lista de recomendação baseada no top-n, a diferença é que como não é apenas um "usuário fixo", uma lista com

as recomendações de cada usuário precisa ser fornecida para o serviço. O pseudocódigo simplificado para o entendimento do algoritmo pode ser visto no método *kernel_ calcula_ lista_ predicoes* abaixo e a implementação pode ser vista no código A.5.

```
kernel_calcula_lista_predicoes( avaliacoesUsuarios[],
       S, usuarios[], itens[], tamLista, resul[] )
{
    memoria_compartilhada s_sum_a;
    memoria_compartilhada s_sum_b;

    blocoId := id do Bloco Atual;

    Equanto blocoId < tamLista
    {
        Para cada thread T dentro do bloco Atual
        efetue as operações:
        {
          similaridade := S[ itens[blocoId] ][idThreadT].similaridade;
          rating := avaliacoesUsuarios[ blocoId*NRO_ITENS_DATASET +
                              S[ itens[blocoId ][idThreadT].item_id ];

          s_sum_a[idThreadT] := rating*similaridade;
          s_sum_b[idThreadT] := absoluto( similaridade )*(1&&rating);
        }

        Realizar o somatório dos arrays s_sum_a e s_sum_b através de uma
        operação de redução armazenando o resultado no item zero do array;

        Se idThreadT == 0 entao
            resul[blocoId] :=  s_sum_a[0] / s_sum_b[0];

        blocoId += Nro de Threads Em um Bloco;
```

```
    }
}
```

Neste caso não iremos utilizar o recurso de chamadas simultâneas ao kernel pois toda a lista de predições será calculada em uma única chamada e então não há necessidade de chamadas simultâneas.

3.2.7 Questões Sobre a Implementação

Neste capítulo foi apresentado um modelo de filtro colaborativo como serviço onde o principal objetivo era tentar responder as requisições de recomendação no menor tempo possível. A partir deste modelo foi apresentado uma implementação paralelizada do algoritmo de filtro colaborativo utilizando a GPU. Outros trabalhos também tentaram paralelizar este algoritmo só que com uma abordagem diferente, em alguns foram utilizadas threads na CPU [1] e outros trabalhos que utilizaram GPU [2] não consideraram datasets grandes que não coubessem na memória global da GPU. Nos experimentos que serão apresentados no próximo capítulo, iremos mostrar o comportamento do algoritmo trabalhando em paralelo na GPU. Em alguns casos devido à particularidade do algoritmo, pode ocorrer não haver grande aumento de desempenho, como por exemplo, no caso da GPU, a existência de instruções de desvio no algoritmo é péssima para o desempenho. Então, em alguns pontos teremos que avaliar se vale realmente a pena utilizar um algoritmo que rode em paralelo na GPU, em paralelo na CPU ou se realmente não vale a pena paralelizar.

Capítulo 4

Experimentos e Resultados

Neste capítulo iremos apresentar os experimentos realizados utilizando a arquitetura proposta nos capítulos anteriores além da implementação dos serviços apresentados. Nós primeiramente apresentaremos o ambiente no qual nossa arquitetura foi executada, em seguida uma descrição das tarefas que foram executadas, os parâmetros que foram levados em conta na implementação e os datasets utilizados.

4.1 Experimentos

4.1.1 Ambiente de Execução dos Experimentos

Para realizar os testes, iremos utilizar como base um conjunto de tarefas que foram utilizadas no artigo [1] para avaliar o desempenho da implementação apresentada no trabalho citado.

Para realizar nossos experimentos utilizamos uma máquina com processador *Intel(R) Core(TM) i7* de 3.20GHz, com 16Gb de memória RAM e duas placas de video NVidia GeForce GTX 480 que possuem 480 núcleos de processamento e 1Gb de memória global cada placa. No artigo que iremos utilizar como comparação a maquina utilizada foi uma com processador *Intel(R) Xeon* com 4 núcleos de 3.0GHz e 48Gb de memória RAM.

Serão utilizadas duas bases de dados de recomendação, a primeira será a de 10M do movielens [19], que possui 69.878 usuários e 10.681 itens. A segunda base de dados será a de 100M do Netflix Prize [20] que possui 480.189 usuários e 17.770 itens. Nossos experimentos serão divididos em três tarefas, onde em cada tarefa

iremos detalhar seu funcionamento e quais tipos de implementação foram utilizadas nas próximas seções. Para cada experimento faremos 5 execuções e calcularemos a média do tempo.

4.1.2 Tarefas

Para avaliar o desempenho de nossa arquitetura e poder comparar com o desempenho de outros trabalhos nós precisamos criar tarefas para efetuar medições. Nesta seção iremos apresentar as tarefas utilizadas e seu funcionamento. Alguns parâmetros do algoritmo de filtro colaborativo foram utilizados para otimizar a execução e gerar um resultado com qualidade satisfatória assim como foi utilizado no artigo [1] e os parâmetros justificados em [15]. O parâmetro *Tamanho da Lista de Similaridades*, que são os N itens mais similares de um determinado item foi fixado em 75. O parâmetro *Quantidade de Vizinhos* foi fixado em 30, que são itens que serão utilizados para calcular a predição de um item, que pode ser visto na equação 2.2. Na próxima seção iremos mostrar como foram divididos os usuários e itens em nossas execuções.

Divisão dos Usuários e Itens em Quartis

Para efetuar a execução de algumas tarefas foi necessário dividir os usuários e itens em quartis. Cada quartil foi construído de acordo com a quantidade de avaliações dada ao item ou ao usuário. Por exemplo, o item que obteve a maior quantidade de avaliações ficará no primeiro quartil enquanto que o item que obteve o menor número de avaliações ficará no quarto quartil. A mesma analogia poderá ser feita aos usuários na divisão dos quartis. Para exemplificar iremos ilustrar o seguinte exemplo: Suponha que para um usuário pertencer ao primeiro quartil ele tenha que ter avaliado no mínimo 4.462 itens, para pertencer ao segundo quartil 2.390 itens, para o terceiro quartil 1.460 itens e para pertencer ao quarto quartil ter avaliado menos que 1.460 itens e mais que 30 itens. Na tabela 4.1 podemos ver como ficaria divido os quartis no exemplo citado.

Um detalhe importante é que a quantidade de usuários em cada quartil é igual a 2.5% da quantidade de usuários no dataset. No caso da divisão dos itens em quartis ao invés de usuários, a analogia é mesma, sendo apenas trocados os usuários

Divisão dos Quartis		
	Usuário	Quantidade de Avaliações
Quartil 1	500 632 732 ... 955	5.132 itens 5.050 itens 5.010 itens ... 4.462 itens
Quartil 2	1.032 852 356 ... 1.235	4.461 itens 4.460 itens 4.460 itens ... 2.390 itens
Quartil 3	1.568 1.851 2.357 ... 3.335	2.389 itens 2.388 itens 2.388 itens ... 1.460 itens
Quartil 4	633 2.489 1.253 ... 2.457	1.459 itens 1.459 itens 1.458 itens ... 1.200 itens

Tabela 4.1: Exemplo de como ficaria a divisão em quartis.

por itens e sendo utilizada a quantidade de avaliações que cada item recebeu. Nas próximas seções iremos apenas citar a divisão em quartis em alguns experimentos quando estas forem utilizadas.

Tarefa de Inicialização

Esta tarefa representa o processo de inicialização necessário para o sistema de recomendação executar. Para um sistema de recomendação baseado no item há a necessidade criar o modelo baseado nas similaridades dos itens, ou seja, uma matriz *Item x Item* onde seu conteúdo é a similaridade entre os itens. Para que essa matriz de similaridades possa ser construída é necessário efetuar um cálculo com base nos históricos de recomendações dos itens lidos dos datasets. A implementação detalhada desta tarefa pode ser vista na seção onde foi mostrada a implementação do serviço de criação da matriz de similaridades. Nesta tarefa será medido o tempo necessário para o modelo ser construído.

Como a carga de trabalho para esta tarefa é muito grande comparada às outras,

vale a pena dividir a execução desta tarefa em dois cenários, o primeiro utilizando o cálculo da matriz de similaridades executando em uma GPU e o segundo tendo o processamento do cálculo da matriz sendo dividido para duas GPUs. O grande desafio dessa tarefa é que como os datasets do Movielens e do NetFlix são consideravelmente grandes, tivemos que fazer uma implementação onde esses datasets pudessem ser parcialmente armazenados na memória global da GPU em forma de matrizes. Para que essa matriz *Item x Usuário* pudesse ser processada na GPU, foi necessário dividir esta em pedaços aproximadamente do tamanho do espaço livre na memória global da GPU. Desta forma, nosso algoritmo se tornou escalável dando a possibilidade que datasets maiores pudessem ser divididos e armazenados parcialmente na memória global da GPU. Podendo assim, serem processados e o seu resultado devolvido a memória principal.

No Multilens [21] esta tarefa foi implementada utilizando threads na CPU sendo o processamento divido entre as threads nos diferentes núcleos da CPU. O armazenamento da matriz *Item x Usuário* utilizado, ficou somente na memória principal não necessitando de nenhum tipo de adaptação como foi feito no caso da utilização da GPU. No capítulo de resultados poderemos comparar o desempenho desses cenários apresentados.

Tarefa de Geração de Lista de Recomendação para o Usuário

Esta tarefa simula um usuário navegando em um site de comércio eletrônico (ex: Netflix), onde este recebe uma lista de recomendações de um sistema por trás do site visitado. O objetivo desta tarefa é produzir um conjunto de n itens que serão recomendados ao usuário. Esta tarefa irá testar a eficiência do sistema em executar o processo de recomendação do algoritmo top-n.

Para executar esta tarefa, nós primeiro iremos dividir os usuários em quartis com base no número de filmes que cada usuário avaliou, assim como foi explicado no início do capítulo. Em seguida, iremos escolher aleatoriamente k usuários de cada quartil, onde k é igual a aproximadamente 2,5% de todos os usuários do dataset. Para cada usuário escolhido uma listagem de recomendação de itens é gerada com base no algoritmo top-n. Os valores reportados neste experimento são uma média do tempo gasto para geração da lista para um usuário ao longo de cada quartil.

É importante observar que como os usuários foram divididos em quartis, portanto a quantidade de trabalho será diferenciada para cada quartil. Suponha que um usuário do primeiro quartil foi sorteado e será necessário gerar uma lista de recomendação para este usuário. O primeiro quartil é onde estão os usuários que mais avaliaram, logo existirão poucos itens para serem geradas predições. Se um usuário do quarto quartil for sorteado, um maior número de predições deverá ser gerado para este usuário já que este avaliou menos itens que o usuário do primeiro quartil. Essa divisão será importante para que uma média justa possa ser gerada, onde a média que será calculada será de usuários que possuem aproximadamente o mesmo número de recomendações, logo aproximadamente a mesma quantidade de trabalho reduzindo assim o desvio padrão.

Com essa divisão em quartis, iremos apresentar nesta tarefa os resultados da geração da listagem de recomendações quando esta for executada na GPU e iremos comparar com os resultados apresentados pelo Multilens mantendo o mesmo cenário apresentado.

Tarefa de Geração de Recomendações por Demanda

Depois de ser apresentado com uma lista de recomendações, o nosso usuário do sistema está pronto para começar a olhar para os detalhes de filmes específicos. Esta tarefa simula um usuário navegando por filme para ver detalhes específicos. O trabalho do sistema é dizer ao usuário o quanto ele vai gostar do filme, prevendo a nota que o usuário irá dar ao filme (utilizando a equação de predição de item 2.2). A medição que será feita nesta tarefa será o tempo médio que o serviço leva para gerar uma predição, para isso uma grande quantidade de requisições deverá ser feita ao serviço.

Assim como na tarefa de geração de lista de recomendação, os usuários foram divididos em quartis, mas agora também iremos dividir os itens em quartis de acordo com quantidade de avaliações assim como foi explicado no início do capítulo. O experimento será feito com base na combinação entre quartis dos usuários e itens como o exemplo da tabela 4.2. O número de predições calculadas em cada combinação de quartil será de aproximadamente 2.5% do número de usuários do dataset.

Poderemos observar nessa tarefa, assim como na tarefa de geração de lista de

Combinação dos Quartis		
	Item	Usuário
i1, u1	item quartil 1 ...	usuário quartil 1 ...
i1, u2	item quartil 1 ...	usuário quartil 2 ...
i1, u3	item quartil 1 ...	usuário quartil 3 ...
i1, u4	item quartil 1 ...	usuário quartil 4 ...
i4, u1	item quartil 4 ...	usuário quartil 1 ...
i4, u2	item quartil 4 ...	usuário quartil 2 ...
i4, u3	item quartil 4 ...	usuário quartil 3 ...
i4, u4	item quartil 4 ...	usuário quartil 4 ...

Tabela 4.2: Exemplo da combinação entre quartis de itens e usuários.

recomendação, onde haverá uma diferença na quantidade trabalho que será realizado em cada combinação de quartil. Da mesma forma que na tarefa anterior, essa combinação de quartis fará como que o desvio padrão fique baixo, os usuários e itens sorteados em um mesmo quartil terão um número de avaliações próximo. No capítulo de resultados poderemos comparar o resultado da média obtida em cada combinação de quartil executando na GPU e a média da execução utilizando o Multilens.

4.2 Resultados

Em outros trabalhos, como o Multilens apresentado em [1], foram utilizadas tarefas similares para avaliar o desempenho de um sistema de recomendação. O Multilens é uma implementação feita em *Java* que possui diferentes métodos de recomendação, o trabalho citado alterou o seu código e focou a implementação no modelo baseado no item utilizando a fórmula de cosseno para cálculo da similaridade assim como foi feito em nosso trabalho. No Multilens foi utilizado uma implementação utilizando threads na CPU na construção do modelo e na predição de itens, logo estamos comparando nossos resultados com outra aplicação paralela. Para podermos ter uma melhor

ideia de desempenho, iremos mencionar aqui os resultados do trabalho citado para efeito de comparação. Os resultados que serão apresentados nas próximas seções são com relação as tarefas citadas na seção anterior. Os resultados de tempo que serão apresentados para comparação, serão o tempo execução utilizando o Multilens, os casos utilizando uma GPU e o caso utilizando duas GPUs.

4.2.1 Resultados da Tarefa de Inicialização

Esta é a tarefa de inicialização da matriz de similaridades S, a descrição do algoritmo sequencial e do algoritmo paralelo que roda na GPU foi explicada nos capítulos anteriores. Os resultados desta tarefa serão apresentados de acordo com tempo de execução e os diferentes casos onde o algoritmo foi aplicado.

Tabela 4.3: Tabela de resultados da Tarefa de Inicialização com tempo de execução em segundos.

DataSet	Multilens	GTX480	2xGTX480
10M Movielens	426	182	168
100M Netflix	3371	2881	1887

Podemos observar na tabela 4.3 os resultados das execuções da tarefa de inicialização. Quando verificamos os tempos de execução no dataset de 10M do Movielens podemos ver que a execução utilizando a placa geforce GTX480 obteve um tempo 2.3 vezes menor que o tempo do Multilens, ou seja, conseguimos diminuir cerca de 57% do tempo de execução. Mas, quando verificamos o tempo de execução utilizando duas placas GTX480 obtivemos um tempo 2.5 vezes menor ou uma redução de 61% do tempo de execução do Multilens. A diferença do tempo de execução entre utilizarmos uma GPU e duas GPUs não foi relativamente grande para este dataset. Isso pode ser explicado pela subutilização das threads na GPU, fazendo com que algumas threads ficassem ociosas no caso da utilização de duas placas gráficas. No caso da utilização de apenas uma placa gráfica o aproveitamento de utilização das threads foi melhor tendo um resultado mais próximo da utilização de duas placas gráficas.

Ainda na tabela 4.3 podemos analisar o resultado da execução do dataset de 100M do Netflix. Neste caso, utilizando apenas uma GPU tivemos uma redução no tempo de execução de aproximadamente de 15%. Mas quando comparamos com a execução utilizando duas GPUs obtivemos uma redução de 44% do tempo de

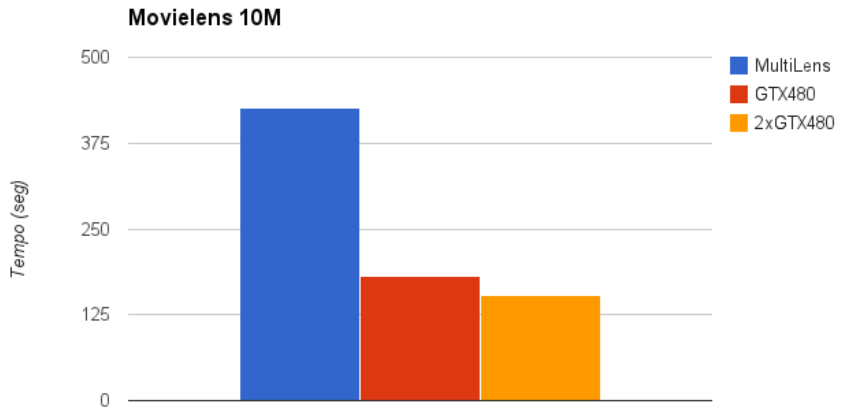

Figura 4.1: Gráfico com Tempo médio de execução da Tarefa de Inicialização utilizando o Movielens.

execução. Para efeito de uma melhor comparação podemos observar os gráficos das figuras 4.1 e 4.2.

Podemos observar nos gráficos das figuras 4.1 e 4.2 que quando comparamos o tempo de execução utilizando uma placa gráfica e duas placas gráficas para o dataset do Netflix a diferença é maior do que quando utilizamos o dataset do Movielens. Essa maior diferença de desempenho pode ser explicada pela quantidade de trabalho que aumentou diminuindo a ociosidade das threads no caso da utilização de duas placas gráficas. Com o melhor aproveitamento dos recursos da GPU podemos observar uma grande melhora desempenho consequentemente reduzindo o tempo de execução do serviço.

4.2.2 Resultados da Tarefa de Geração de Lista de Recomendação

Esta tarefa tem a função de gerar uma lista de itens como recomendação para um usuário. Para isso, para cada item que o usuário não avaliou é necessário gerar uma predição para este item e em seguida ordenar a lista de predições. Para medir o tempo, todos os usuários foram divididos em quatro quartis, onde as divisões dos quartis foram feitas como explicado nas seções anteriores. Para efetuar a medição de tempo utilizamos aleatoriamente 2.5% de usuários de cada quartil para efetuar

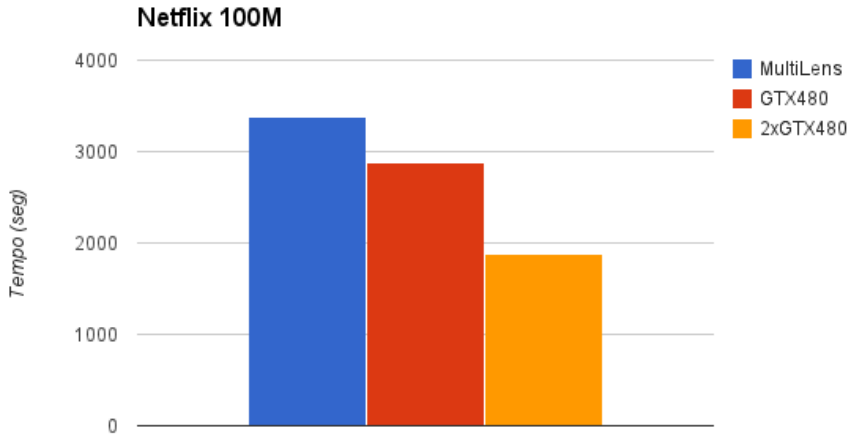

Figura 4.2: Gráfico com Tempo médio de execução da Tarefa de Inicialização utilizando o Netflix.

os testes e em seguida foram feita as médias de tempo.

Tabela 4.4: Tabela de resultados da tarefa de geração de lista de recomendação utilizando o dataset do Movielens com tempo de execução em milissegundos.

	Multilens	GTX480
Quartil 1	1	0.46
Quartil 2	1	0.45
Quartil 3	2	0.45
Quartil 4	4	0.44

Tabela 4.5: Tabela de resultados da tarefa de geração de lista de recomendação utilizando o dataset do Netflix com tempo de execução em milissegundos.

	Multilens	GTX480
Quartil 1	0.5	0.8
Quartil 2	1	0.74
Quartil 3	2.5	0.71
Quartil 4	7.5	0.71

Podemos observar nos gráficos nas figuras 4.3 e 4.4 que o tempo de execução nos primeiros quartis é menor para a execução na CPU. Isso ocorre devido ao espaço de busca ser menor, ou seja, os usuários desses quartis terem avaliado mais filmes do que os dos quartis seguintes, quanto mais filmes o usuário avaliou menos predições para este usuário precisam ser geradas. No caso da GPU o tempo de execução em todos os quartis parece ser constante, independentemente da quantidade de itens a serem avaliados. Isso pode ser explicado pela grande quantidade de threads que realizam a execução em paralelo. Se todas as threads estiverem ocupadas ou se poucas threds estiverem ocupadas o tempo de execução será próximo. Este fato ocorre

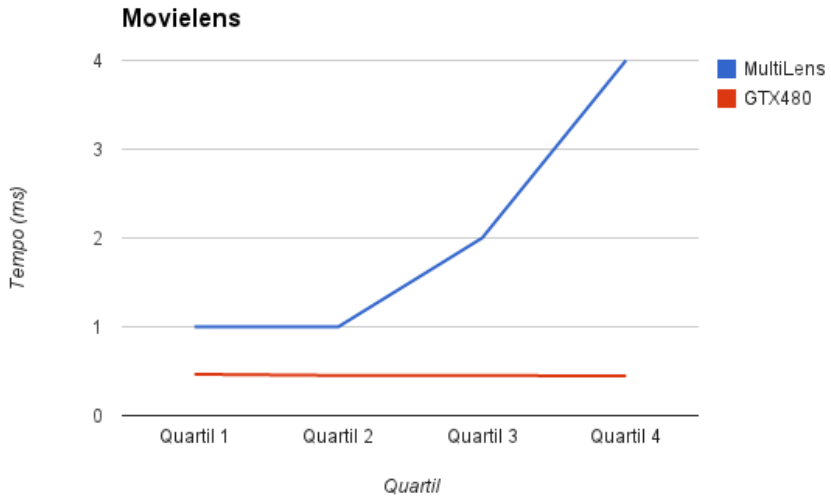

Figura 4.3: Gráfico com tempo médio de execução da tarefa de geração de lista de recomendação para o dataset do Movielens.

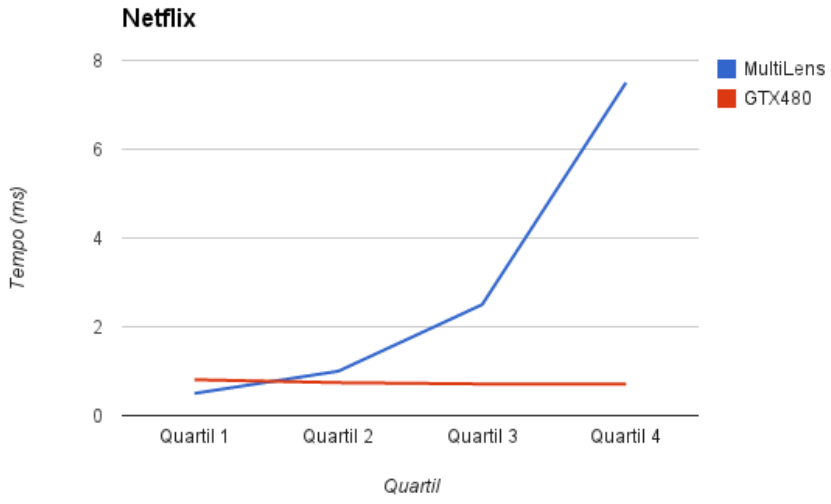

Figura 4.4: Gráfico com Tempo médio de execução da tarefa de geração de lista de recomendação para o dataset do Netflix.

porque as threads ociosas terão que esperar pelas threads que estão executando terminar, logo o tempo final dependerá da thread que demorou mais tempo. Como no desenvolvimento em CUDA é recomendado que não se utilizasse instruções de desvio no algoritmo, o tempo de execução das threads fica parecido dando a impressão de ser constante. No quartil um do gráfico 4.4 podemos verificar que o tempo de execução do Multilens foi melhor que do que utilizando a GPU, isso pode ser explicado devido a pouca quantidade de tarefas de desse quartil fazendo com que várias threads ficassem ociosas na GPU. No caso da execução na CPU, como são poucas threads executando simultaneamente e existem instruções de desvio evitando o processamento matemático, o tempo de execução acabou ficando menor. Este caso não acontece quando o trabalho das threads vai aumentando, como pode ser visto nas execuções dos outros quartis onde o número de avaliações dos usuários são menores. Com a utilização da arquitetura fermi, foram feitas chamadas simultâneas ao kernel aumentando a vazão e consequentemente diminuindo o tempo médio de geração das listas de recomendações.

4.2.3 Resultados da Tarefa de Geração de Recomendações por Demanda

A medição realizada nesta tarefa foi o tempo médio que o serviço levou para gerar uma predição. Para isso, uma grande quantidade de requisições foram feitas ao serviço, para cada combinação de quartil (quartil de item x quartil de usuário) o número de predições que foram feitas foi de aproximadamente 2.5% do número de usuários. A combinação de quartis de itens e usuários e os respectivos tempos utilizando os datasets do Movielens e do Netflix podem ser vistos nos gráficos nas figuras 4.5 e 4.6 respectivamente.

Tabela 4.6: Tabela de resultados da Tarefa de Geração de Recomendações por Demanda utilizando o dataset do Movielens com tempo de execução em milissegundos.

Comb. Quartil	Multilens	GTX480
i1,u1	0.05	0.011
i1,u2	0.05	0.010
i1,u3	0.055	0.010
i1,u4	0.06	0.010
i4,u1	0.06	0.010
i4,u2	0.06	0.010
i4,u3	0.06	0.010
i4,u4	0.06	0.010

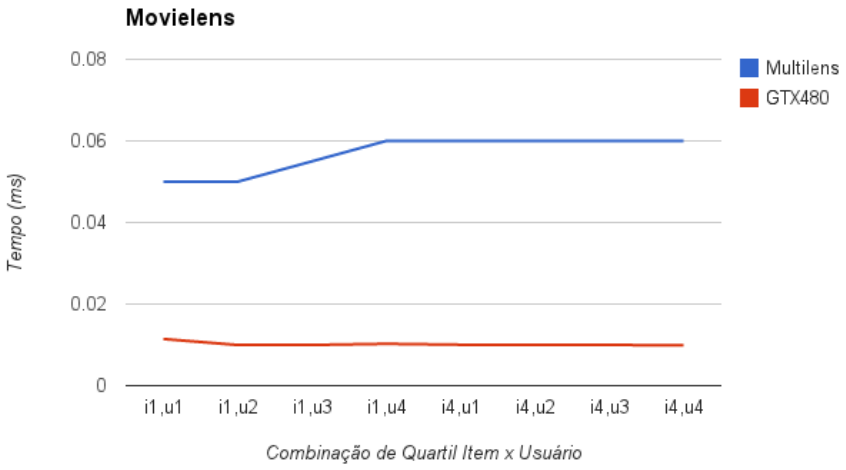

Figura 4.5: Gráfico com Tempo médio de uma predição gerada na Tarefa de Geração de Recomendações por Demanda para o dataset do Movielens.

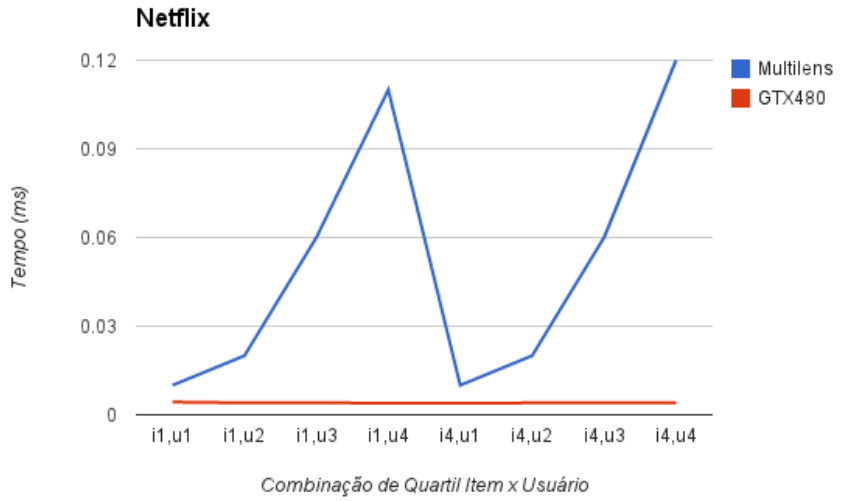

Figura 4.6: Gráfico com Tempo médio de uma predição gerada na Tarefa de Geração de Recomendações por Demanda para o dataset do Netflix.

Tabela 4.7: Tabela de resultados da Tarefa de Geração de Recomendações por Demanda utilizando o dataset do Netflix com tempo de execução em milissegundos.

Comb. Quartil	Multilens	GTX480
i1,u1	0.01	0.004
i1,u2	0.02	0.004
i1,u3	0.06	0.004
i1,u4	0.11	0.004
i4,u1	0.01	0.004
i4,u2	0.02	0.004
i4,u3	0.06	0.004
i4,u4	0.12	0.004

Assim como na tarefa de geração de lista de recomendação o menor quartil possui a maior quantidade de avaliações, ou seja, o quartil de itens 1 possui os itens com maior quantidade de avaliações enquanto que o quartil de itens 4 possui os itens com menor quantidade de avaliações. O mesmo acontece com os quartis de usuários, o quartil de usuário 1 possui os usuários que mais avaliaram itens enquanto que o quartil de usuários 4 possui os usuários que menos avaliaram itens.

Podemos observar nos gráficos das figuras 4.5 e 4.6 que o tempo médio das predições utilizando o Multilens basicamente foi dependente da quantidade de trabalho que cada quartil forneceu, ou seja, quanto mais avaliado o item ou quanto mais avaliações o usuário realizou, menor será o tempo de predição. No caso das execuções utilizando a GPU o tempo assim como na tarefa de geração de lista de recomendação, se manteve constante em todos os quartis. Esse tempo constante mais uma vez pode ser explicado pela ociosidade de algumas threads e conforme o aumento de trabalho ocorrido ao utilizar outros quartis, essa ociosidade foi diminuída mas o tempo de execução continuou a ser o da thread mais lenta apesar da melhora na utilização dos recursos da GPU.

Capítulo 5

Conclusões

Com relação a tarefa de inicialização utilizando o dataset do Movielens a performance utilizando CUDA ficou melhor, levando aproximadamente a metade do tempo do Multilens. A performance utilizando uma placa gráfica e duas placas gráficas foi similar, isso pode ser explicado pela relação entre quantidade de trabalho a ser realizado e as unidades de processamento da GPU. Provavelmente no caso onde foi utilizada uma GPU as threads foram mais bem utilizadas que no caso da utilização de duas GPUs, onde algumas threads ficaram ociosas equiparando o seu tempo de execução com o caso de uma GPU. Como a execução é em paralelo o tempo final de execução será o pior tempo de execução das threads executadas.

No caso da utilização do dataset do Netflix na tarefa de inicialização, a performance utilizando CUDA também ficou melhor, sendo aproximadamente 15% mais rápido utilizando somente um placa gráfica. Utilizando duas placas gráficas houve uma diminuição de cerca 44% do tempo de execução quando comparado com o Multilens. Com uma maior carga de trabalho, houve um melhor aproveitamento das duas placas gráficas diminuindo a quantidade de threads ociosas e consequentemente diminuindo o tempo de execução. Com um maior fluxo de trabalho, houve uma vazão maior no processamento das linhas das matrizes. Nossa implementação em CUDA se mostrou escalável, não importando o tamanho da matriz de ratings que será carregada em memória. Com o aumento da carga de trabalho, houve um melhor aproveitamento dos recursos da GPU e uma maior vazão de linhas processadas.

Com a divisão do dataset em quartis na tarefa de geração de lista de recomendação, conseguimos observar que no Multilens de acordo com quartil utilizado a

carga de trabalho pode variar devido a quantidade de itens não avaliados ou de usuários que não avaliaram o item. Utilizando CUDA o tempo de execução ficou praticamente constante em todos os quartis. O tempo ficou constante devido o tempo total de execução da geração da lista de recomendação depender do pior tempo das threads em execução. Conforme se altera o quartil, a carga de trabalho é alterada, diminuindo (ou aumentando) as threads ociosas e alterando a vazão de geração da lista de predições mas mantendo o tempo de execução. Com a utilização da arquitetura fermi, chamadas simultâneas ao kernel foram feitas aumentando a vazão e consequentemente diminuindo o tempo médio de geração das listas de recomendações.

Quando analisamos os resultados da tarefa de geração de recomendações por demanda, observamos que os resultados utilizando CUDA se mostraram mais rápidos e capazes atender uma grande demanda de requisições de predições. Os resultados do Multilens mostraram que de acordo com a combinação de quartil o tempo de execução de uma predição pode variar, devido a um item com poucas avaliações ou um usuário que avaliou pouco, isso pode influenciar no resultado. Em nossa implementação em CUDA, cada predição foi feita por um bloco da GPU e com as chamadas simultâneas ao kernel, conseguimos ter uma maior vazão na geração das recomendações sob demanda e consequentemente ter um menor tempo médio de geração da predição como pode ser visto nos resultados.

A nossa arquitetura utilizando a GPU se mostrou eficiente para atender grandes requisições de predições de itens e geração de listas de recomendações para usuários. Não foi necessário um hardware de ponta para chegar a esses resultados. A arquitetura se mostrou escalável, não sendo limitada pela memória da GPU. A utilização da GPU para operações matriciais é bastante eficaz, no entanto a utilização de instruções de desvios dentro do código CUDA pode fazer com que o desempenho caia drasticamente. O grande desafio em se escrever um código utilizando CUDA é fazer com que a aplicação utilize o máximo dos recursos disponíveis da GPU, tentando fazer com que o mínimo de threads fiquem ociosas durante os ciclos de processamento. Como podemos ver nos resultados apresentados, o resultado da execução sempre ficará dependente do pior tempo executado por uma thread do warp. O ideal é que se tente fazer com que todas tenham praticamente o mesmo tempo de

execução otimizando assim a vazão do trabalho realizado.

Em nosso trabalho podemos listar as principais contribuições que foram feitas:

1. A proposta e avaliação do algoritmo de filtro colaborativo baseado em memória sendo executado em uma GPU.

2. Um algoritmo para lidar com bases de dados relativamente grandes onde a matriz R de recomendação e a matriz S de similaridades não cabem na memória da GPU, ou seja, um algoritmo apresentado pode ser considerado escalável.

3. Conseguimos melhorar o tempo de execução do algoritmo em até 61% do tempo de um algoritmo que já rodava em paralelo na CPU.

4. Foi mostrado que não é necessário uma GPU de ponta para chegar a esses resultados quando comparado com uma CPU de custo relativamente alto.

Trabalhos Futuros

Com a experiência adquirida tentando resolver nosso problema, ao longo do caminho nos deparamos com algumas outras soluções que também poderiam ser apresentadas utilizando este tema. Outros trabalhos poderiam utilizar não apenas uma máquina com múltiplas GPUs, mas *clusters de GPUs* para serem utilizados por trás dos serviços. Para realizar este trabalho, poderia ser utilizado o *MPI (Message Passing Interface)* para distribuir os processos entre as máquinas na rede e executar o CUDA localmente em cada máquina.

Outra grande solução que poderia ser estudada, seria trabalhar com este mesmo problema utilizando matrizes esparsas e CUDA. Essa solução reduziria drasticamente o espaço ocupado em memória, tanto na CPU quanto na GPU. Apesar da queda de desempenho ao utilizar um estrutura de dados desse tipo, em algumas situações poderia ser importante a economia de espaço em detrimento do desempenho.

Os serviços de predição e geração de lista de recomendação apresentados neste trabalho, poderiam também ser implementados dando suporte à múltiplas GPUs e comparados com os resultados aqui apresentados. Com familiarização do CUDA, diversas outros problemas poderiam ser resolvidos utilizando este importante recurso.

Referências Bibliográficas

[1] LEVANDOSKI, J. J., EKSTRAND, M. D., LUDWIG, M., etal. "RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures." *PVLDB*, v. 4, n. 11, pp. 911–920, 2011. Disponível em: <http://dblp.uni-trier.de/db/journals/pvldb/pvldb4.html#LevandoskiELEMR11>

[2] LI, R., ZHANG, Y., YU, H., etal. "A social network-aware top-N recommender system using GPU." In: Newton, G., Wright, M., Cassel, L. N. (Eds.), *JCDL*, pp. 287–296. ACM, 2011. ISBN: 978-1-4503-0744-4. Disponível em: <http://dblp.uni-trier.de/db/conf/jcdl/jcdl2011.html#LiZYWWW11>.

[3] "Top500". 2013. Feb., 2013. Disponível em: <http://www.top500.org/>. Acesso em Fevereiro, 2013.

[4] "Movielens". 2013. Feb., 2013. Disponível em: <http://www.movielens.umn.edu/>. Acesso em Fevereiro, 2013.

[5] "Netflix". 2013. Feb., 2013. Disponível em: <http://www.netflix.com/>. Acesso em Fevereiro, 2013.

[6] "OpenMP". 2013. Feb., 2013. Disponível em: <http://openmp.org/>. Acesso em Fevereiro, 2013.

[7] "PThreads". 2013. Feb., 2013. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/>. Acesso em Fevereiro, 2013.

[8] "Message Passing Interface". 2013. Feb., 2013. Disponível em: <http://www.mcs.anl.gov/research/projects/mpi/>. Acesso em Fevereiro, 2013.

[9] SANDERS, J., KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[10] "AMBER". 2013. Feb., 2013. Disponível em: <http://ambermd.org/gpus/>. Acesso em Fevereiro, 2013.

[11] "Green500rankings". 2013. Feb., 2013. Disponível em: <http://www.green500.org/>. Acesso em Fevereiro, 2013.

[12] ADOMAVICIUS, G., TUZHILIN, A. "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions", *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, v. 17, n. 6, pp. 734–749, 2005.

[13] "Java". 2013. Feb., 2013, Disponível em: <http://www.java.com/>. Acesso em Fevereiro, 2013.

[14] "CADAL". 2013. Feb., 2013, Disponível em: <http://www.cadal.zju.edu.cn/>. Acesso em Fevereiro, 2013.

[15] SARWAR, B., KARYPIS, G., KONSTAN, J., etal. "Item-based collaborative filtering recommendation algorithms". In: *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pp. 285–295, New York, NY, USA, 2001. ACM. ISBN: 1-58113-348-0. doi: 10.1145/371920.372071. Disponível em: <http://doi.acm.org/10.1145/371920.372071>.

[16] KIRK, D. B., HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, fev. 2010. ISBN: 0123814723. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0

[17] "NVIDIA Corporation. Fermi computer architecture whitepaper." 2013. Feb., 2013. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute Acesso em Fevereiro, 2013.

[18] "CUDA Thrust". 2013. Feb., 2013. Disponível em: <https://developer.nvidia.com/thrust>. Acesso em Fevereiro, 2013.

[19] "Movielens Dataset". 2013. Feb., 2013. Disponível em: <http://www.grouplens.org/node/73>. Acesso em Fevereiro, 2013.

[20] "Net Flix Prize Dataset". 2013. Feb., 2013. Disponível em: <http://www.netflixprize.com/>. Acesso em Fevereiro, 2013.

[21] "Sistema de Recomendação Multilens". 2013. Feb., 2013. Disponível em: <http://knuth.luther.edu/~bmiller/dynahome.php?page=multilens>. Acesso em Fevereiro, 2013.

Apêndice A

Códigos Fontes

A.1 Código que calcula similaridade entre vetores na GPU

```
void calcula_similaridade(
    TRating *d_A, /* Linha do Item A da matriz. */
    TRating *d_B, /* vectorN Linhas da matriz que serqo calculadas com
       linha A. */
    float *d_COS, /* Vetor para armazenar os calculos das similaridades
        entre A e B. (p.ex: d_COS[1]=sim(A, B[1]), d_COS[2]=sim(A, B
       [2]), etc...*/
    int vectorN, /* Nro de Linhas de B. */
    int elementN /* Nro de elementos na Linha de B */
){
    /* Vetores na memoria compartilhada do bloco */
    __shared__ float accumResultAB[ACCUM_N];
    __shared__ float accumResultAA[ACCUM_N];
    __shared__ float accumResultBB[ACCUM_N];
    __shared__ float accumResultSUM[ACCUM_N];
    __shared__ int accumResultN[ACCUM_N];
    int posA  = 0;
    int andAB = 0;
    float sdA = 0;
    float sdB = 0;

    __syncthreads();
```

```
    accumResultAB[threadIdx.x] = 0;
    accumResultAA[threadIdx.x] = 0;
    accumResultBB[threadIdx.x] = 0;
    accumResultSUM[threadIdx.x] = 0;
    accumResultN[threadIdx.x] = 0;

    /**
     * Cada bloco ficara responsavel por calcular a similaridade
     * entre um par de linhas
     */
    for(int vec = blockIdx.x; vec < vectorN; vec += gridDim.x){
        int vectorBase = IMUL(elementN, vec);
        int vectorEnd  = vectorBase + elementN;

   /**
    * Cada Thread de um bloco ficara responsavel por realizar as
    * operacoes do calculo de similaridade entre um par de
    * elementos de uma linha.
    * Caso o numero de elementos da linha seja maior que o
    * nro de threads algumas threads terao que realizar
    * mais trabalho.
    */
        for(int iAccum = threadIdx.x; iAccum < ACCUM_N; iAccum +=
            blockDim.x){
            float sumAB = 0;
            float sumAA = 0;
            float sumBB = 0;
            float sumSUM = 0;
            int sumN = 0;

            for(int pos = vectorBase + iAccum; pos < vectorEnd; pos +=
                ACCUM_N)
            {
              posA = pos - vectorBase; //como se fosse um modulo %
              sdA = (float)d_A[posA];
              sdB = (float)d_B[pos];

              andAB = sdA * sdB;
                sumAB += andAB;
```

```
57                  andAB = 1 && andAB;

59                  sumAA += (andAB * sdA) * sdA;
                    sumBB += (andAB * sdB) * sdB;
61              }

63              accumResultAB[iAccum] = sumAB;
                accumResultAA[iAccum] = sumAA;
65              accumResultBB[iAccum] = sumBB;
            }
67
            /**
69      * Em cada bloco sera realizado uma operacao de reducao
        * tendo o resultado armazenado no item zero dos vetores.
71      */
            for(int stride = ACCUM_N >> 1; stride > 0; stride >>= 1)
73          {
                __syncthreads();
75              for(int iAccum = threadIdx.x; iAccum < stride; iAccum +=
                    blockDim.x)
                {
77                  accumResultAB[iAccum] += accumResultAB[stride + iAccum
                        ];
                    accumResultAA[iAccum] += accumResultAA[stride + iAccum
                        ];
79                  accumResultBB[iAccum] += accumResultBB[stride + iAccum
                        ];
                }
81          }

83          /* Cada bloco calculou a similaridade entre um par de linhas */
            if(threadIdx.x == 0)
85          {
              d_COS[vec] = accumResultAA[0] && accumResultBB[0] ?
87                __fdividef( accumResultAB[0], sqrt(accumResultAA[0] *
                      accumResultBB[0] ) ) : 0;
            }
89      }
}
```

Código A.1: Código que calcula similaridade entre vetores na GPU.

A.2 Código que divide a Matriz R em partes menores e faz a chamada para o cálculo na GPU

```
void cria_matrix_similaridades(
  float* R, /* Matriz Item x Usuario */
  float* S  /* Matriz de similaridades entre itens */
)
{
    float *dev_a;
    float *dev_b;
    float *dev_res_cos;
    float *result_cos;
    long itema, useru;
    float res;

    /* Calcula o nro de partes em que serÃ¡ dividida a matriz de acordo
        com o tamanho da mem da GPU. */
    int nPartes = (NItems*NUsers*sizeof(TRating))/(1024*1024) / 500;
    nPartes = nPartes ? nPartes : 1;

    /* Calcula quantas linhas serao copiadas por rodada. */
    int parteDasLinhas = NItems / nPartes;
    int inicio = 0;

    /* Calcula o resto das linhas se a divisao nao for inteira. */
    int resto = NItems % nPartes;
    nPartes = nPartes + (resto ? 1 : 0);

    for( int i=0; i < nPartes; i++ )
    {

  /* Copia parte da Matriz R para memoria global da GPU */
      cudaMemcpy( dev_b, &(R[inicio*NUsers]), parteDasLinhas*NUsers *
          sizeof(TRating), cudaMemcpyHostToDevice);
```

```
30
        for( itema=1;  itema < NItems; itema++ )
32      {
          if( m_item_numbers[itema] <= 0 )
34          continue;

36        int nLines = (itema - inicio)+1;
          if( nLines > 0 )
38        {
            if( nLines > parteDasLinhas )
40            nLines = parteDasLinhas;
          }
42        else
            nLines = itema;
44
          if( inicio < itema )
46        {
            cudaMemcpy( dev_a, &(R[itema*NUsers]), NUsers * sizeof(
                TRating), cudaMemcpyHostToDevice);
48
        /* chamada ao metodo que sera executado na GPU */
50          calcula_similaridade<<<nblocks,ACCUM_N>>>( dev_a, dev_b,
                dev_res_cos, dev_res_mean, nLines, NUsers );


52
        /**
54       * Copia o resultado do calculo de similaridade da matriz R
         * na memoria da GPU para memoria principal
56       */
            cudaMemcpy( result_cos, dev_res_cos, nLines * sizeof( float )
                , cudaMemcpyDeviceToHost);
58

60      /* Copia o resultado para a matriz S */
            for( int i=0; i < nLines; i++ )
62          {
                                res = result_cos[i];
64                              S[itema][inicio+i].similarity = res;
                                S[itema][inicio+i].item_id = inicio+i;
```

```
66                                  S[inicio+i][itema].similarity = res;
                                    S[inicio+i][itema].item_id = itema;
68              }
              }
70          }
    /* Incrementa nro de linhas ja calculadas na matriz */
            inicio += parteDasLinhas;
            if( resto && i == (nPartes-2)) //ultima interacao coloca
                somente o resto
74              parteDasLinhas = resto;
        }
76  }
```

Código A.2: Algoritmo que divide a Matriz R em partes menores e faz a chamada para o cálculo na GPU.

A.3 Código que calcula a similaridade entre vetores utilizando mais de uma GPU

```
    void* cria_matrix_similaridades_multigpu(
2     void *pvoidData /* Estrutura com os dados que foram alocados na
          memoria da GPU e o ID da placa grafica*/
    )
4   {
        DataStruct  *d = (DataStruct*)pvoidData;
6       cudaSetDevice( d->deviceID ); /* Configura qual placa grafica sera
            utilizada */

8       long itema, useru;
        float res;
10
        /* Calcula o nro de partes em que serÃ¡ dividida a matriz de acordo
            com o tamanho da mem da GPU. */
12      int nPartes = (NItems*NUsers*sizeof(TRating))/(1024*1024) / 500;
        nPartes = nPartes ? nPartes : 1;
14
        /* Calcula quantas linhas serao copiadas por rodada. */
```

```
16    int parteDasLinhas = NItems / nPartes;
      int inicio = 0;
18
      /* Calcula o resto das linhas se a divisao nao for inteira. */
20    int resto = NItems % nPartes;
      nPartes = nPartes + (resto ? 1 : 0);
22
      /* O loop foi divido em partes de acordo com o numero de placas
          graficas */
24    for( int i=d->deviceID; i < nPartes; i+=NDEVICES )
      {
26      cudaSetDevice( d->deviceID )
    /* Copia parte da Matriz R para memoria global da GPU */
28      cudaMemcpy( d->dev_b, &(R[inicio*m_nUserID]), parteDasLinhas*
            m_nUserID * sizeof(TRating), cudaMemcpyHostToDevice);

30        for( itema=1;  itema < m_nItemID; itema++ )
        {
32          if( m_item_numbers[itema] <= 0 )
              continue;
34
          int nLines = (itema - inicio)+1;
36        if( nLines > 0 )
          {
38          if( nLines > parteDasLinhas )
              nLines = parteDasLinhas;
40        }
          else
42          nLines = itema;

44        if( inicio <= itema )
          {
46          cudaSetDevice( d->deviceID );
            cudaMemcpy( d->dev_a, &(R[itema*m_nUserID]), m_nUserID *
                sizeof(TRating), cudaMemcpyHostToDevice);
48
            int nblocks = ceil( MAX_USER / ACCUM_N );
50        /* chamada ao metodo que sera executado na GPU */
```

```
            calcula_similaridade<<<nblocks,ACCUM_N>>>( d->dev_a, d->dev_b
                , d->dev_res_cos, d->dev_res_mean, nLines, m_nUserID );
52
        /**
54       * Copia o resultado do calculo de similaridade da matriz R
         * na memoria da GPU para memoria principal
56       */
            cudaMemcpy( d->result_cos, d->dev_res_cos, nLines * sizeof(
                float ), cudaMemcpyDeviceToHost);
58
        /* Copia o resultado para a matriz S */
60          for( int i=0; i < nLines; i++ )
            {
62            res = d->result_cos[i];
              S[itema][inicio+i].similarity = res;
64            S[itema][inicio+i].item_id = inicio+i;
              S[inicio+i][itema].similarity = res;
66            S[inicio+i][itema].item_id = itema;
            }
68        }
      }
70    /* Incrementa nro de linhas ja calculadas na matriz */
        inicio += parteDasLinhas;
72  /* ultima interacao coloca somente o resto */
        if( resto && i == (nPartes-NDEVICES))
74        parteDasLinhas = resto;
    }
76}
```

Código A.3: Algoritmo que calcula similaridade entre vetores utilizando mais de uma GPU.

A.4 Código que gera uma lista de recomendações baseada no algoritmo Top-n utilizando a GPU

```
// cada predicao eh feita por um bloco e cada thread executa uma
   operacao matematica
```

```
__global__ void cuda_topn(
    TRating* d_matrix_rating, // items que userid avaliou
    Item* d_matrix_sim, // matriz de similaridades
    int userid, // id do usuario
    float* d_pred // array com as predicoes que sera retornado
){
  int b = blockIdx.x;
  int strideb = gridDim.x;
  float sim, d;
  float rat;
  int index_rat = 0;
  int index_sim = 0;
  __shared__ Item s_matrix_sim[NRO_MAX_VIZINHOS];
  __shared__ float s_sum_a[NRO_MAX_VIZINHOS];
  __shared__ float s_sum_b[NRO_MAX_VIZINHOS];

  // bloco b, cada bloco calculara uma predicao
  // as threads do bloco b efetuarao o calculo matematico necessario na
      predicao
  while( b < MAX_ITEM )
  {
    // se ja tem recomendacao entao a predicao retornada eh -1 e pula
        para a proxima predicao
    if( d_matrix_rating[ b ] )
    {
      d_pred[b] = -1;
      b += strideb;
      continue;
    }
    // carrega para a memoria compartilhada as similaridades que estao
        na memoria global
    s_matrix_sim[threadIdx.x] = d_matrix_sim[b*NRO_MAX_VIZINHOS+
        threadIdx.x];
    s_sum_a[threadIdx.x] = 0.0;
    s_sum_b[threadIdx.x] = 0.0;
    __syncthreads();

    sim = s_matrix_sim[threadIdx.x].similarity;
    rat = (float)d_matrix_rating[s_matrix_sim[threadIdx.x].item_id];
```

```
38      // calcula a operacao matematica da predicao
        s_sum_a[threadIdx.x] = rat*sim;
40      s_sum_b[threadIdx.x] = fabs( sim )*(1&&rat);

42      // efetua uma operacao de reduce para calcular o somatorio
        for(int stride = NRO\_MAX\_VIZINHOS >> 1; stride > 0; stride >>= 1)
44      {
            __syncthreads();
46          for(int iAccum = threadIdx.x; iAccum < stride; iAccum +=
                blockDim.x)
            {
48        s_sum_a[iAccum] += s_sum_a[stride + iAccum];
          s_sum_b[iAccum] += s_sum_b[stride + iAccum];
50          }
        }
52
        // toda thread zero de um bloco grava o resultado da predicao no
            array de retorno
54      if( threadIdx.x == 0 )
        {
56        d = (float)(s_sum_b[0] == 0.0 ? 0.0 : __fdividef(s_sum_a[0] ,
              s_sum_b[0]));
          d_pred[b] = d;
58      }
        b += strideb;
60    }
}
```

Código A.4: Algoritmo gera uma lista de recomendações baseada no algoritmo Topn utilizando a GPU.

A.5 Código que calcula uma lista de recomendações para uma lista de pares (usuario, item) utilizando a GPU

```
/**
 * Calcula uma lista de predicoes.
 * cada predicao eh feita por um bloco e cada thread executa uma
    operacao matematica
 */
__global__ void cuda_lista_predicoes(
    TRating* d_matrix_rating, // array com as avaliacoes de cada
       usuario
    Item* d_matrix_sim, // matrix de similaridades
    int* userid, // lista de usuarios
    int* itemid, // lista de items
    int nLista, // numero de itens e usuarios a serem avaliados
    float* d_pred // array com o resultado das predicoes
){
  int b = blockIdx.x;
  int strideb = gridDim.x;
  float sim, d;
  float rat;
  int index_rat = 0;
  int index_sim = 0;
  __shared__ Item s_matrix_sim[COVER];
  __shared__ float s_sum_a[COVER];
  __shared__ float s_sum_b[COVER];

  // bloco b, cada bloco calculara uma predicao
  // as threads do bloco b efetuarao o calculo matematico necessario na
      predicao
  while( b <  nLista )
  {
          // carrega para a memoria compartilhada as similaridades que
             estao na memoria global
    s_matrix_sim[threadIdx.x] = d_matrix_sim[itemid[b]*COVER+threadIdx.
       x];
    s_sum_a[threadIdx.x] = 0.0;
    s_sum_b[threadIdx.x] = 0.0;
    __syncthreads();

    sim = s_matrix_sim[threadIdx.x].similarity;
```

```
    rat = (float)d_matrix_rating[ b*MAX_ITEM + s_matrix_sim[threadIdx.x
        ].item_id ];

    // calcula a operacao matematica da predicao
    s_sum_a[threadIdx.x] = rat*sim;
    s_sum_b[threadIdx.x] = fabs( sim )*(1&&rat);

    // efetua uma operacao de reduce para calcular o somatorio
    for(int stride = COVER >> 1; stride > 0; stride >>= 1)
    {
        __syncthreads();
        for(int iAccum = threadIdx.x; iAccum < stride; iAccum +=
            blockDim.x)
        {
      s_sum_a[iAccum] += s_sum_a[stride + iAccum];
      s_sum_b[iAccum] += s_sum_b[stride + iAccum];
        }
    }

    // toda thread zero de um bloco grava o resultado da predicao no
        array de retorno
    if( threadIdx.x == 0 )
    {
      d = (float)(s_sum_b[0] == 0.0 ? 0.0 : __fdividef(s_sum_a[0] ,
          s_sum_b[0]));
      d_pred[b] = d;
    }
    b += strideb;
  }
}
```

Código A.5: Algoritmo que calcula uma lista de recomendações para uma lista de (usuario, item) utilizando a GPU.