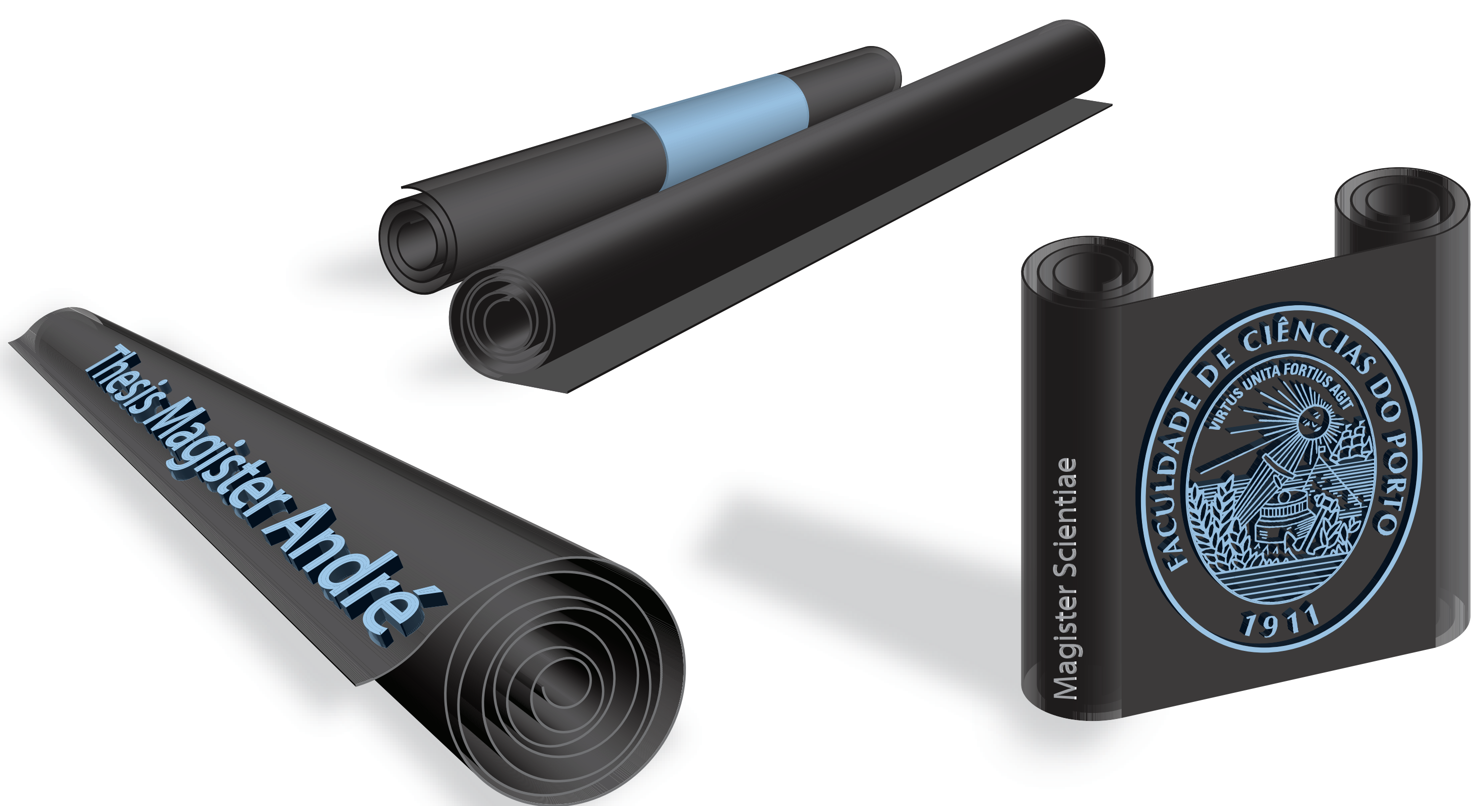
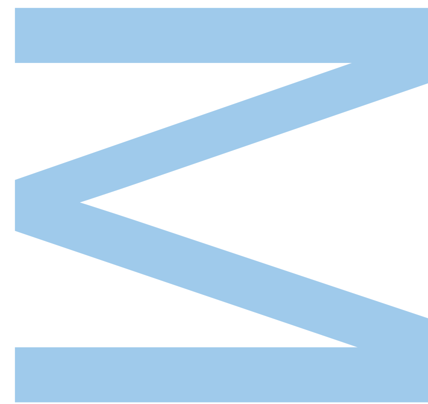


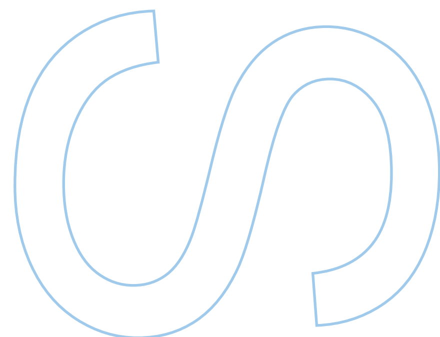
Paralelização de algoritmos de fatorização de matrizes para recomendação usando GPU

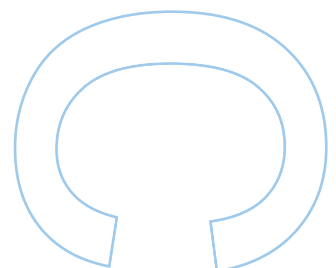


André Valente Rodrigues
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2014

Orientador
Alípio Mário Guedes Jorge

Coorientadora
Inês de Castro Dutra

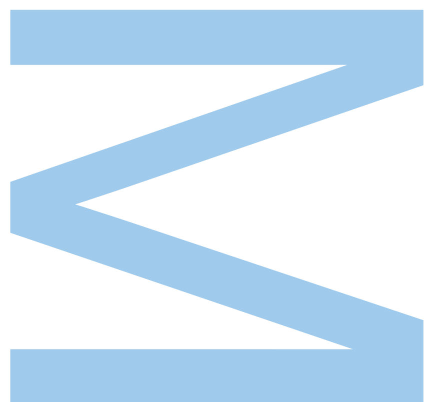
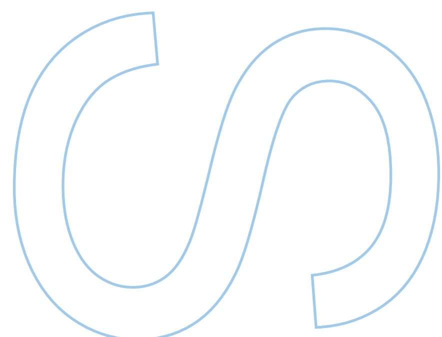
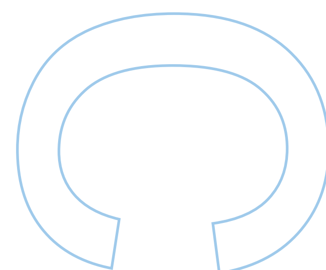




Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

FACULDADE DE CIÊNCIAS DO PORTO
VIRTUS UNITA FORTIUS AGIT
1911

Dedicatória

Para os meus pais, e todos aqueles que me apoiaram, dando estímulo, carinho, amor e energia para toda esta jornada que atinge o grau de Mestre.

Agradecimentos

Começo por expressar a minha gratidão aos meus orientadores, Professor Alípio Jorge e Professora Inês Dutra, pelo apoio, orientação, pela confiança que me incutiram, bem como agradecer pelas oportunidades que me deram, porquanto contribuíram muito para a minha carreira e sucesso académico.

Com o propósito de escrever esta tese e um artigo científico, agradeço também o financiamento dado pela FCT no âmbito do projeto Sibila - Scientific Grant (NORTE-07-0124-FEDER-000059).

Quero agradecer, ainda, aos meus amigos pelo tempo que disponibilizaram para me apoiarem moralmente, sobretudo ao Mestre Sérgio Oliveira Marques, físico/matemático, dotado de grande aptidão inata, que me ajudou na elaboração das demonstrações matemáticas, presentes nesta dissertação, relativas à minimização da expressão do algoritmo *ALS*.

Um agradecimento especial à Professora Fátima Sørås pela ajuda na revisão final dos textos.

Por fim, gostaria de agradecer aos meus pais por todo apoio que me têm dado ao longo da minha vida académica. Sem eles, dificilmente, teria alcançado mais uma etapa nesta longa jornada que é a minha vida.

Abstract

This thesis has a bidisciplinary character, as it is deals with the two areas of data mining and parallel systemns. Regarding the area of data mining, the recommendation systems and their respective algorithms are defined. Concerning the area of parallel systems, the distinction between classical systems and systems using graphics cards is clarified. Finally, involving both areas, we discuss the implementation of recommendation algorithms through parallel programming on General Purpose Graphics Processing Units (*GPGPU*), using *NVIDIA CUDA* (Compute Unified Device Architecture). Next, the thesis focuses on matrix factorization algorithms, particularly the algorithms Alternating Least Squares (*ALS*), Stochastic Gradient Descent (*SGD*) and Cyclic Coordinate Descent (*CCD*). We describe *GPGPU* implementations of the recommendation algorithms *CCD++* and *ALS*, which were designed for this thesis. The processing time and the predictive ability of the *GPGPU* implementations are compared with the same algorithms and implemented in existing multicore versions. The results show that it is possible to obtain good speedups in *GPGPU* (maximum 14.8). The speedups in *GPGPU* are better than those obtained with the multicore versions. The *GPU-CCD++* algorithm is faster than the *GPU-ALS*.

Resumo

Esta dissertação tem um carácter bidisciplinar, uma vez que abrange duas áreas de especialização: *data mining* e sistemas paralelos. Relativamente à área de *data mining*, apresenta-se a definição dos sistemas de recomendação e respetivos algoritmos. No que concerne à área de sistemas paralelos, esclarece-se a definição dos sistemas clássicos e de sistemas, utilizando placas gráficas. Por fim, envolvendo as duas áreas, é abordada a implementação de algoritmos de recomendação através da programação paralela em Unidades de Processamento Gráfico de Propósito Geral (*GPGPU*), utilizando, para esse efeito, *NVIDIA CUDA (Compute Unified Device Architecture)*. Seguidamente, a dissertação centra a sua atenção nos algoritmos de fatorização de matrizes, em particular os algoritmos *Alternating Least Squares (ALS)*, *Stochastic Gradient Descent (SGD)* e *Cyclic Coordinate Descent (CCD)*. Descrevem-se, posteriormente, as implementações *GPGPU* dos algoritmos de recomendação baseados em fatorização de matrizes *CCD++* e *ALS*, que foram concebidas para esta dissertação. O tempo de processamento e a capacidade preditiva das implementações *GPGPU* são comparados com os mesmos algoritmos em versões *multicore* existentes e implementadas. Os resultados mostram que é possível obter em *GPGPU* bons *speedups* (máximo 14.8). Os *speedups* em *GPGPU* são melhores em relação aos obtidos com as versões *multicore*. O algoritmo *GPU-CCD++* é mais rápido que o *GPU-ALS*.

Conteúdo

Dedicatória . . . vii
Agradecimentos . . . ix
Abstract . . . xi
Resumo . . . xiii
Conteúdo . . . xvii
Lista de Figuras . . . xix
Lista de Tabelas . . . xxi
Lista de Algoritmos . . . xxiii

1 Introdução **1**

2 Sistemas de recomendação **5**
2.1 Definição . . . 5
2.2 Aplicações . . . 6
2.3 Implementação . . . 7
2.4 Algoritmos baseados em fatorização de matrizes . . . 8
2.4.1 Noções de *Singular Value Decomposition (SVD)* . . . 8
2.4.2 *Alternating Least Squares (ALS)* . . . 13
2.4.3 *Stochastic Gradient Descent (SGD)* . . . 21
2.4.4 *Cyclic Coordinate Descent (CCD)* . . . 23
2.4.5 Comparações entre os algoritmos *ALS, SGD e CCD/CCD++* . . . 28
2.5 Avaliação . . . 29
2.6 Metodologias . . . 31
2.7 Sumário . . . 31

3 Sistemas paralelos **33**
3.1 Sistemas paralelos clássicos . . . 33
3.1.1 Programação em memória distribuída . . . 35
3.1.2 Programação em memória partilhada . . . 36

3.2 Sistemas paralelos usando *General Purpose Graphics Processing Unit (GPGPU)* 41
3.2.1 A arquitetura *Compute Unified Device Architecture (CUDA)* 43
3.2.2 O futuro das placas gráficas 47
3.3 Métricas de desempenho 49
3.4 Sumário 49

4 Paralelização de algoritmos de recomendação 51
4.1 Paralelização do algoritmo *ALS* 51
4.2 Paralelização do algoritmo *SGD* 52
4.2.1 Versão *Distributed Stochastic Gradient Descent (DSGD)* 52
4.2.2 Versão *Hogwild* do algoritmo *SGD* 53
4.3 Paralelização do algoritmo *CCD++* 53
4.4 Sumário 55

5 Paralelização para *GPU* de dois algoritmos de recomendação 57
5.1 Implementação do algoritmo *CCD++* em *CUDA* 58
5.2 Implementação do algoritmo *ALS* 59
5.2.1 Ponto de partida para a implementação e tratamento dos dados esparsos 60
5.2.2 Implementação *multicore* 64
5.2.3 Implementação *CUDA* 65
5.3 Utilização 66
5.4 Sumário 67

6 Metodologia experimental 69
6.1 Máquina *cracs-gpu* 70
6.2 Máquina *hyperthreading* 71
6.3 Conjunto de dados *Netflix* 72
6.4 Conjunto de dados *MovieLens 10M* 73
6.5 Sumário 73

7 Resultados 75
7.1 Resultados utilizando o conjunto de dados *Netflix* (Secção 6.3) 76
7.1.1 Testes em precisão *double* na versão original do *CCD++* da *LIBPMF* em *OpenMP* 76
7.1.2 Testes em precisão *float* na versão modificada do *CCD++* em *OpenMP* e em *CUDA* 77
7.1.3 Testes em precisão *float* no *ALS* em *OpenMP* e em *CUDA* 78
7.2 Resultados utilizando o conjunto de dados *MovieLens 10M* (Secção 6.4) 79
7.2.1 Testes em precisão *double* na versão original do *CCD++* da *LIBPMF* em *OpenMP* 79

7.2.2 Testes em precisão *float* na versão modificada do *CCD++* em *OpenMP* e em *CUDA* 80
7.2.3 Testes em precisão *float* no *ALS* em *OpenMP* e em *CUDA* 81
7.3 Análise do comportamento relativamente à parametrização dos algoritmos implementados 82
7.4 Análise do comportamento em *CUDA* dos algoritmos recorrendo a *profiling* visual 85
7.4.1 Análise do algoritmo *CCD++* 85
7.4.2 Análise do algoritmo *ALS* 88
7.4.3 Comparação entre *ALS* e *CCD++* 90
7.4.4 Erros conhecidos 91
7.4.5 Artigo submetido 91
7.5 Sumário 92

8 Conclusões **93**

A Instalação do ambiente de programação **95**
A.1 Instalação do *CUDA toolkit 6.0* no *Linux fedora 20* 95
A.2 Executar o projeto do *VS 2013* em *Windows* 97

B Configurações **99**
B.1 Configuração de um projeto em *C++* e *CUDA* com o *Microsoft Visual Studio Ultimate 2013* 99
B.2 Configuração de um *makefile* para compilar *C++* e *CUDA* em *Linux* 118

Bibliografia **119**

Lista de Figuras

2.1 Ilustração das matrizes *SVD* 8
2.2 Ilustração das matrizes de um modelo baseado em fatorização de matrizes 12
2.3 Ilustração da função com um mínimo global 22
2.4 Ilustração da função com mínimos locais 22
2.5 Ilustração do ruído estocástico 22

3.1 Exemplo de um fluxo de execução de uma aplicação em *OpenMP* 37
3.2 Exemplo de um fluxo de execução de um *for* parelelo em *OpenMP* 39
3.3 Foto do interior de um *GPU Kepler 2 GK110* 42
3.4 Ilustração de alto nível do *hardware CUDA* 44
3.5 Ilustração da conexão *NVLink* entre *GPUs* e *CPU* 48
3.6 Ilustração da conexão *NVLink* apenas entre *GPUs* 48

7.1 Comportamento do algoritmo *CCD++* relativamente a *k* 82
7.2 Análise comparativa do tempo de processamento entre os algoritmos *ALS* e *CCD++*, relativamente ao aumento de *k* 83
7.3 Análise comparativa do *RMSE* entre os algoritmos *ALS* e *CCD++*, relativamente ao aumento de *k* 84
7.4 *Timeline* para 1 iteração de *CCD++* no *GPU* 85
7.5 *Timeline* para 15 iterações de *CCD++* no *GPU* 86
7.6 Análise adicional *CCD++* com o *NVVP* 86
7.7 Informação proveniente dos sensores da placa gráfica com o algoritmo *CCD++* . 87
7.8 *Timeline* para 1 iteração de *ALS* no *GPU* 88
7.9 *Timeline* para 15 iterações de *ALS* no *GPU* 88
7.10 Análise adicional *ALS* com o *NVVP* 89
7.11 Informação proveniente dos sensores da placa gráfica com o algoritmo *ALS* . . . 90

Lista de Tabelas

2.1 Comparação dos algoritmos *ALS, SGD e CCD/CCD++* 28

4.1 Vantagens e desvantagens dos algoritmos a nível da paralelização 55

7.1 *Speedups* do *CCD++* da *LIBPMF* em *OpenMP* 76
7.2 *Speedups* na versão modificada do *CCD++* em *OpenMP* e em *CUDA* 77
7.3 *Speedups* no *ALS* em *OpenMP* e em *CUDA* 78
7.4 *Speedups* do *CCD++* da *LIBPMF* em *OpenMP* 79
7.5 *Speedups* na versão modificada do *CCD++* em *OpenMP* e em *CUDA* 80
7.6 *Speedups* no *ALS* em *OpenMP* e em *CUDA* 81
7.7 Comparação de *Speedups* entre o *CCD++* e o *ALS* em *OpenMP* e em *CUDA* . . 92

Lista de Algoritmos

1 *ALS* [66] 18
2 *SGD* [21, 75] 21
3 *CCD* [75] 25
4 *CCD++* [75] 28

5 Versão paralela do algoritmo *ALS* para sistemas *multicore* [78] 51
6 Versão *DSGD* do algoritmo *SGD* para sistemas *multicore* [21] 52
7 Versão *Hogwild* do algoritmo *SGD* para sistemas *multicore* [46, 79] 53
8 Versão paralela do algoritmo *CCD++* para sistemas *multicore* [75] 54

9 Versão *GPU* do algoritmo *CCD++* 59
10 Versão *GPU* do algoritmo *ALS* 65

Capítulo 1

Introdução

Os Sistemas de Recomendação surgiram a partir da necessidade da filtragem inteligente de dados e da definição automática de prioridades e preferências dos utilizadores por determinados itens. O primeiro sistema de recomendação, *Tapestry* [24], criado no início da década de 90, consistia num sistema experimental de filtragem inteligente de mensagens de *e-mail*. Foi neste momento, também, que surgiu o conceito de *collaborative filtering* que consiste em gerar recomendações, partindo da colaboração da comunidade de utilizadores.

Hoje em dia, com o crescimento da Internet, devido à grande variedade de produtos e serviços disponibilizados, os utilizadores têm dificuldade em escolher entre as várias alternativas que lhes são apresentadas. É neste contexto que os Sistemas de Recomendação são úteis. Tipicamente, são sistemas que, tendo como ponto de partida as preferências dos utilizadores de uma comunidade, cruzam a informação e agregam-na, de maneira a produzirem novas recomendações personalizadas dentro das preferências do utilizador. A ideia consiste num sistema que simula características sociais relacionadas com a partilha de conhecimento, como, por exemplo, perguntar a um amigo se ele conhece um filme interessante numa determinada área [54].

Muitos sistemas de recomendação são implementados, utilizando algoritmos baseados em fatorização de matrizes [35, 57]. Dada a matriz de interação utilizador-item A, o objetivo destes algoritmos é encontrar duas matrizes W e H de modo a que $W \times H^T$ resulte numa aproximação da matriz A. A matriz W representa os utilizadores segundo um conjunto de características latentes, e a matriz H representa os itens segundo um conjunto de características latentes. Considerando W e H, é fácil calcular as previsões para um utilizador i relativamente a um item j com o qual ainda não tenha interagido. O facto destes algoritmos de recomendação serem baseados em cálculos matriciais torna-os apropriados para paralelização.

Como é necessário gerar recomendações a partir de conjuntos de dados cada vez maiores, será do interesse dos provedores de serviços responder em tempo real a grandes números de pedidos

em simultâneo. Portanto, a solução para acelerar o processo, passa, primordialmente, por recorrer ao paralelismo. O paralelismo consiste em dividir um problema em vários subproblemas independentes que podem ser processados, simultaneamente, em vários processadores.

Pelas razões mencionadas, vários autores têm proposto soluções paralelas para os algoritmos baseados em fatorização de matrizes. Por exemplo, algoritmos populares como o *Alternating Least Squares (ALS)* e o *Stochastic Gradient Descent (SGD)* têm versões paralelas tanto para arquiteturas em memória distribuída como para memória partilhada [66, 75, 76, 80]. Recentemente, Yu *et al.* [75] demonstraram que os algoritmos baseados no método *Cyclic Coordinate Descent (CCD)* têm uma maior eficiência na regra de atualização relativamente ao *ALS* e, relativamente ao *SGD*, uma convergência mais estável. Utilizando por base o método de fatorização *CCD*, Yu *et al.* implementaram o algoritmo de recomendação *CCD++*, utilizando um sistema de memória distribuída com 256 nós (máquinas) e demonstraram que este é 20 vezes mais rápido que o *Distributed-ALS* e 40 vezes mais rápido que o *Distributed-SGD*.

Com o aumento da popularidade de *General Purpose Graphics Processing Units (GPGPU)*, e a sua adequação para programação paralela, algoritmos baseados em matrizes, geralmente, funcionam bem na plataforma supracitada. No contexto de *collaborative filtering*, para *GPGPU*, só é conhecida a implementação do algoritmo baseado em vizinhança de Zhanchun *et al.* [77]. Para esta dissertação, foi feita, pela primeira vez, a implementação para *GPGPU* de dois algoritmos baseados em fatorização de matrizes, não se conhecendo qualquer outra implementação deste género. Os algoritmos aqui implementados na versão *GPU* são o *ALS* e o *CCD++*. Para isso, utiliza-se o modelo de programação *CUDA* nos sistemas operativos *Windows* e *Linux*. Estas versões dos algoritmos são testadas com avaliações comummente encontradas na literatura. No *CCD++*, os resultados são comparados com a versão *multicore* de Yu *et al.* [75], e no *ALS* são comparados com a versão *multicore* deste algoritmo, também implementada nesta dissertação. Os melhores resultados, utilizando as versões *GPU-CCD++* e *GPU-ALS*, no que concerne às versões sequenciais (*single core*) atingem um *speedup* de 14.8 e 6.2, respetivamente. A versão *CUDA* mais rápida (*CCD++* em *Windows*) supera a versão de 32 *cores*. Relativamente às versões sequenciais, todos os resultados, utilizando *GPU* e *multicore*, geram recomendações exatamente com a mesma qualidade (*root mean squared error*).

Levando em consideração que o custo por *core* de uma máquina *GPGPU* com milhares de *cores* é muito inferior ao custo de uma *multicore*, facilmente se conclui que a versão *GPU-CCD++* permite um desempenho idêntico ou superior com um investimento menos elevado relativamente à versão *multicore*.

Esta dissertação está dividida em 8 capítulos. No Capítulo 1, é contextualizado o trabalho e são discutidos objetivos e contribuições. No Capítulo 2, são apresentados os conceitos fundamentais em sistemas de recomendação e são discutidas algumas técnicas utilizadas nestes sistemas. No Capítulo 3, são apresentados os sistemas paralelos clássicos e sistemas paralelos recorrendo a placas gráficas. No Capítulo 4, são apresentadas técnicas de paralelização *multicore* dos algoritmos descritos no Capítulo 2. No Capítulo 5, é descrita a paralelização em *GPU* dos algoritmos *ALS* e *CCD++*. No Capítulo 6, são explicados todos os detalhes relativos às

máquinas utilizadas e aos conjuntos de dados utilizados. No Capítulo 7, são apresentados os resultados obtidos nas experiências. Finalmente, no Capítulo 8, são discutidas as conclusões e trabalho futuro.

Capítulo 2

Sistemas de recomendação

Neste Capítulo apresentam-se as motivações, os conceitos e alguns algoritmos dos Sistemas de Recomendação. São também descritas as métricas e as metodologias de avaliação experimental.

Sempre que são mencionadas as palavras "itens"ou "item", pretende-se aludir ao que o sistema recomenda no contexto do problema. Podem referir-se a pessoas, a viagens, a textos, a músicas, a livros, a vídeos, a anúncios, a páginas de Internet, a produtos de uma loja virtual e a tudo o que é possível ser recomendado.

2.1 Definição

Os sistemas de recomendação são programas capazes de, inteligentemente, sugerir ao utilizador itens de sua preferência [9, 40, 54]. Geralmente, são utilizados em sites relacionados com músicas, vídeos, venda de artigos tangíveis ou intangíveis e tudo o que no contexto exploratório dos mesmos possa ser relevante dar a conhecer ao utilizador. Redes sociais como o *Facebook* também usam estes sistemas [2, 27].

Devido ao enorme número de itens que um site pode oferecer, é comum os utilizadores sentirem-se perdidos. Por isso, os sistemas de recomendação permitem ajudá-los nas suas decisões [54]. Tome-se como exemplo a *Amazon.com*, que implementa um Sistema de Recomendação para personalizar ao cliente a loja online [32]. É de realçar que existem sistemas de recomendação não personalizados, consistindo nas 10 seleções mais vistas da lista de livros ou CDs.

No dia a dia, é comum as pessoas trocarem ideias e sugestões relativamente a tomarem decisões, como, comprar um produto, ver um filme, etc. Normalmente, as pessoas escolhidas para a troca de ideias são amigos ou colegas que partilham gostos idênticos, ou pessoas que são

peritas numa área específica. É da natureza humana tomar decisões rotineiras com base nas experiências de conhecidos ou de pessoas em quem se confia. Alguns sistemas de recomendação tentam imitar, no meio digital, estas características sociais. A isto chama-se *collaborative filtering*. Parte-se do princípio racional de que, se o histórico de utilização do utilizador ativo é idêntico ao de outros utilizadores, existe concordância preferencial. Por isso, com base naquilo que outros utilizadores, com um gosto idêntico, viram, são recomendados itens que o utilizador ativo ainda não viu e, provavelmente, gosta [40, 42].

2.2 Aplicações

Os sistemas de recomendação, para além de ajudarem as pessoas nas suas tomadas de decisão, são também utilizados pelos provedores de serviços para:

1. **Aumentar a quantidade de itens vendidos** - Esta é uma das razões que mais motiva os provedores de serviços para a utilização de sistemas de recomendação, fruto da apresentação de itens que interessam ou induzem curiosidade no utilizador. Mesmo no caso de aplicações não comerciais, o simples facto de manter os utilizadores mais tempo no site, indiretamente acaba por gerar lucros [58].

2. **Vender uma maior diversidade de itens** - Esta aplicação consiste em ajudar o utilizador a encontrar itens, que, de outra forma, seriam difíceis ou mesmo impossíveis de encontrar, dando-lhe a conhecer itens agradáveis de que, possivelmente, nem fazia ideia da sua existência [19].

3. **Melhorar a satisfação dos utilizadores** - Um sistema de recomendação, bem implementado e desenhado, melhora a experiência de utilização. É importante ter um sistema que faça recomendações precisas, aliado a uma interface usável e agradável [34, 53].

4. **Aumentar a fidelização dos utilizadores** - O simples facto de promover a personalização do serviço, adaptando-o ao perfil de cada um dos utilizadores, faz com que estes se sintam bem tratados, promovendo, por conseguinte, a fidelização [58].

5. **Perceber quais os desejos dos utilizadores** - As empresas de grande dimensão são muito complexas a nível de gestão de stocks, principalmente, no caso da existência de vários armazéns dispersos geograficamente. A informação proveniente do sistema de recomendação pode incluir informação geográfica relativamente a preferências associadas a regiões, permitindo, desta forma, uma gestão mais eficaz dos stocks e uma melhor capacidade de resposta aos pedidos. Como consequência, obtém-se a redução de custos relacionados com a mão de obra [12, 33, 72].

2.3 Implementação

Um sistema de recomendação é o resultado da união multidisciplinar de várias áreas, tais como: interação pessoa máquina, paralelismo, utilização intensa de *data mining/machine learning* e todos os subcampos da inteligência artificial, podendo, também, envolver estudos relacionados com características sociais dos utilizadores, etc. Conforme o caso, pode até ser necessário técnicas inteligentes de recuperação de informação (*information retrieval*) tal como *text mining*. Por fim, é feito o tratamento da informação previamente extraída, de forma a conseguir-se prever itens ao gosto do utilizador, para, seguidamente, apresentar as recomendações de uma forma não invasiva que promova a aceitação por parte do mesmo [16, 22, 67, 70].

Para a conceção de um sistema de recomendação, existem várias técnicas com propósitos diferentes, a saber: *content-based, user-based, collaborative filtering, demographic, knowledge-based, community-based, hybrid recommender systems*, entre outras [7, 8, 39, 50, 55, 71]. Devido à popularidade em competições recentes, como a do *Netflix*[1], este trabalho focar-se-á nas técnicas de *collaborative filtering* que, com base em informação implícita, o histórico de utilização, ou classificação explícita, introduzida pelos utilizadores, criam uma vizinhança de pessoas com gosto idêntico. Estas técnicas permitem, ainda, recomendar ao utilizador itens que os seus vizinhos também gostam, sem necessitar de informação exógena por parte dos utilizadores ou itens [28]. Para implementar *collaborative filtering*, existem diversos algoritmos baseados em *data mining*. No entanto, devido à capacidade de escalabilidade, ao desempenho preditivo e aos resultados obtidos em competições, como a do *Netflix*[1], os algoritmos de fatorização de matrizes têm vindo a suscitar interesse de investigação [65]. Este trabalho focar-se-á nos algoritmos de recomendação baseados em fatorização de matrizes. Devido à natureza não estática dos dados, à medida que o tempo passa, os modelos ficam desatualizados. Existem, no entanto, duas maneiras de atualizar os modelos baseados em fatorização de matrizes: uma faz-se incrementalmente, a outra realiza-se em *batch*. Porém, devido ao desempenho dos algoritmos, o facto de ser incremental não é de importância absoluta; logo, neste trabalho não é abordada a atualização incremental. Os sistemas de recomendação baseados em *collaborative filtering* podem utilizar dois tipos de *feedback* do utilizador: valores numéricos ou valores binários. Considerando a competição do *Netflix*[1], neste trabalho serão utilizados valores numéricos, também designados por *ratings*.

Apesar de existirem duas formas possíveis de recuperar dados para sistemas de recomendação baseados em *collaborative filtering*: uma, explícita com base nas classificações dos utilizadores; a outra, implícita fundamentada pelos dados binários (histórico de utilização viu ou não viu), este trabalho irá utilizar, unicamente, as classificações explícitas dos utilizadores.

1 Consultar a Secção 6.3.

2.4 Algoritmos baseados em fatorização de matrizes

A fatorização de matrizes é um problema de otimização para o qual existem vários algoritmos. Os algoritmos variam entre a facilidade da implementação, a facilidade da paralelização, a parametrização, a convergência e a qualidade das previsões.

2.4.1 Noções de *Singular Value Decomposition (SVD)*

SVD é técnica de álgebra linear (ramo da matemática), que tem muitas aplicações relacionadas com compressão e representação de matrizes, portanto tem uma variedade grande de utilizações em ciência de computadores, e é comum surgir em compressão de imagens e vídeos. No contexto de *collaborative filtering*, permite resolver o problema de gerar modelos baseados em fatorização de matrizes. *SVD* consiste em decompor uma matriz A de dimensão $m \times n$ num produto de três matrizes tal como ilustrado na Figura 2.1 [1, 6, 57].

SVD do ponto de vista genérico

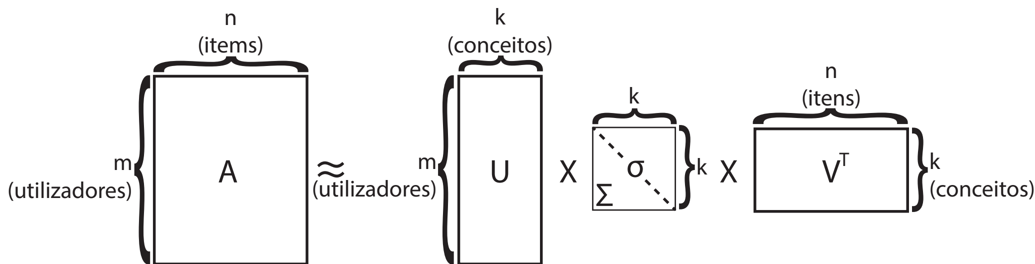


Figura 2.1: Ilustração das três matrizes geradas pela fatorização da matriz A, em que a matriz Σ é composta por zeros à exceção da diagonal σ onde residem os valores singulares, que são números positivos e servem para atribuir a força de cada um dos conceitos.

Para calcular as três matrizes da Figura 2.1, tomando como exemplo uma matriz A de dimensão 2×2, são aplicados os seguintes cinco passos:

$$A = \begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 2 & -1 \\ 2 & 1 \end{bmatrix}$$

1. Calcular AA^T e A^TA:

$$AA^T = \begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix}$$

$$A^T A = \begin{bmatrix} 2 & -1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 5 \end{bmatrix}$$

2. Calcular os valores próprios λ a partir de $|AA^T - \lambda I| = 0$:

$$\left| \begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right| = 0$$

$$\left| \begin{bmatrix} 8-\lambda & 0 \\ 0 & 2-\lambda \end{bmatrix} \right| = 0$$

$$det\left(\left| \begin{bmatrix} 8-\lambda & 0 \\ 0 & 2-\lambda \end{bmatrix} \right| \right) = 0$$

$$(8-\lambda)(2-\lambda) - 0 = 0$$

$$\lambda_1 = 8$$

$$\lambda_2 = 2$$

Obter os valores singulares σ para a diagonal da matriz Σ. Os valores singulares $\sigma_1, ..., \sigma_n$ são equivalentes a $\sqrt{\lambda_1}, ..., \sqrt{\lambda_n}$, respetivamente:

$$\sigma_1 = \sqrt{\lambda_1} = \sqrt{8}$$

$$\sigma_2 = \sqrt{\lambda_2} = \sqrt{2}$$

A matriz Σ fica:

$$\Sigma = \begin{bmatrix} \sqrt{8} & 0 \\ 0 & \sqrt{2} \end{bmatrix}.$$

3. Calcular U:

Primeiro é necessário calcular os vetores singulares de AA^T, recorrendo aos valores singulares λ previamente calculados:

$$(AA^T - \lambda I)x = 0$$

Para λ_1:

$$\left(\begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix} - \lambda_1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) x_1 = 0$$

$$\begin{bmatrix} 8-\lambda_1 & 0 \\ 0 & 2-\lambda_1 \end{bmatrix} x_1 = 0$$

$$\begin{bmatrix} 0 & 0 \\ 0 & -6 \end{bmatrix} x_1 = 0$$

A partir do *span*, é necessário arranjar um x_1 que seja verdadeiro para a igualdade acima; por isso, escolhe-se:

$$x_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Para λ_2:

$$\left(\begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix} - \lambda_2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) x_2 = 0$$

$$\begin{bmatrix} 8 - \lambda_2 & 0 \\ 0 & 2 - \lambda_2 \end{bmatrix} x_2 = 0$$

$$\begin{bmatrix} 6 & 0 \\ 0 & 0 \end{bmatrix} x_2 = 0$$

A partir do *span*, é necessário arranjar um x_2 que seja verdadeiro para a igualdade acima; por isso, escolhe-se:

$$x_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Os vetores singulares x_i têm de ser unitários ortonormais:

$$u_i = \frac{x_i}{||x_i||}$$

Por isso:

$$u_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix} u_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Como u_1 e u_2 são os vetores singulares esquerdos, U é obtida diretamente:

$$U = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

4. Calcular V:

   O processo é igual ao de calcular U, apenas com a diferença de ser com a expressão:

$$(A^T A - \lambda I)x = 0$$

   Com base no que foi calculado em U, adaptando a V, tem-se o seguinte *kernel*:

$$E_i = ker \begin{bmatrix} A^T A_{11} - \lambda_i & A^T A_{12} \\ A^T A_{21} & A^T A_{22} - \lambda_i \end{bmatrix}$$

Por isso, simplificando os passos, aplicando o *kernel* e escolhendo um dos possíveis vetores do *span*, tem-se:

$$E_1 = ker \begin{bmatrix} -3 & 3 \\ 3 & -3 \end{bmatrix} = span \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$E_2 = ker \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix} = span \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Visto que para obter os vetores singulares direitos é necessária uma base ortonormal, basta:

$$\overrightarrow{v_1} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\overrightarrow{v_2} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Com estes dois vetores singulares direitos é obtida a matriz V:

$$V = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

5. Por fim, com tudo calculado, é possível escrever:

$$A = U\Sigma V^T$$

$$\begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{8} & 0 \\ 0 & \sqrt{2} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Os valores singulares σ presentes na diagonal da matriz Σ, no final do cálculo, ficam organizados de forma decrescente a partir de Σ_{11}, permitindo, por isso, facilmente fazer aproximações a A. Quantos mais valores singulares forem considerados, mais exata é a aproximação a A. Visto que as matrizes U e V são reduzidas em conformidade, então, a matriz reconstruída $A_k = U_k S_k V_k'$ é a aproximação $rank - k$ da matriz A. A_k minimiza a norma de *Frobenius* [57] (dada por $||A - A_k||$)perante todas as matrizes $rank - k$. Por isso, esta técnica fornece as melhores aproximações de A com a classificação mais baixa possível ao nível da norma de *Frobenius* [57].

SVD no contexto de *collaborative filtering/UV-Decomposition*

No contexto de *collaborative filtering*, pretende-se um algoritmo capaz de gerar um modelo a partir de dados de aprendizagem incompletos, por isso, quando se fazem previsões, são obtidos dados completos, com aproximação. A utilização de *SVD* serve para capturar relações latentes entre clientes e itens. Neste espaço conjunto de fatores latentes de dimensionalidade k, as interações utilizador-item são modeladas como produtos internos. O espaço latente tenta

caracterizar cada um dos fatores k entre utilizadores e itens inferidos, automaticamente, a partir do conjunto de interações utilizador-item. Por exemplo, quando os produtos são filmes, os fatores podem ter medições óbvias de dimensões como comédia vs. drama, quantidade de ação, ou orientação para as crianças. No entanto, também existem dimensões menos bem definidas, tal como profundidade do desenvolvimento do caráter, 'estranheza' ou, até mesmo, dimensões não interpretáveis [57, 75].

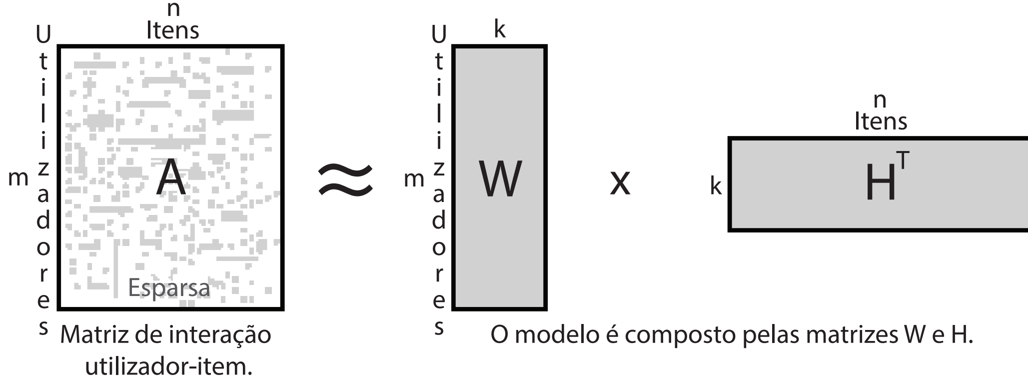


Figura 2.2: Ilustração das matrizes de um modelo baseado em fatorização de matrizes.

Devido à matriz A ser muito esparsa, grande parte é desconhecida, por isso a aplicação do *SVD* convencional é indefinida. O *SVD* só pode ser aplicado a matrizes completamente definidas. Para ser possível capturar o que foi supracitado, são necessárias alterações ao *SVD* original. A diferença é que, agora, pretende-se uma fatorização que não seja apenas reduzir o tamanho de uma matriz completa com recurso a aproximações, mas sim que seja capaz de produzir previsões, recorrendo a aproximações que partem de uma matriz muito esparsa. No *SVD*, as aproximações de U e V são obtidas a partir da matriz Σ, em *collaborative filtering*, as aproximações de W e H são obtidas a partir da Função 2.1. Para isso, sendo $A \in \mathbb{R}^{m \times n}$ a matriz de interação utilizador-item onde m é o número de utilizadores e n o número de itens, pretende-se gerar duas matrizes $W \in \mathbb{R}^{m \times k}$ e $H \in \mathbb{R}^{n \times k}$ em que $W \times H^T \approx A$. Portanto, os mínimos da Função 2.1 permitem obter W e H, minimizando o erro,

$$f(W, H) = \sum_{(i,j) \in \Omega} (A_{ij} - \omega_i^T h_j)^2 + \lambda(||W||_F^2 + ||H||_F^2). \tag{2.1}$$

Para obter os mínimos da Função (2.1), tem-se o Problema 2.2,

$$\min_{\substack{W \in \mathbb{R}^{m \times k} \\ H \in \mathbb{R}^{n \times k}}} \sum_{(i,j) \in \Omega} (A_{ij} - \omega_i^T h_j)^2 + \lambda(||W||_F^2 + ||H||_F^2). \tag{2.2}$$

Onde Ω é o conjunto de índices relativos às classificações observadas, i é mapeado diretamente pelo número de utilizador e j pelo número de item. O parâmetro λ é o de regularização, e serve para contornar o efeito de *overfitting*[2]. A norma de *Frobenius* é indicada por $||\star||_F$ que,

[2] *Overfitting* - Consiste num modelo demasiado ajustado aos dados de treino. O caso extremo acontece quando o modelo só apenas consegue prever o dados que aprendeu; neste caso, não tem utilidade.

neste contexto, se pode explicar a partir da matriz A e a sua aproximação $rank - k = A_k$. Sendo $E = A - A_k$, a norma de *Frobenius* consiste no seguinte somatório $||E||_F^2 = \sum_{i,j} |E_{i,j}|^2$. Por isso, quanto menor for o inteiro produzido pelo somatório, mais próxima é a matriz A_k da A [31, 43]. O vetor ω_i^T corresponde às linhas i da matriz W, e o vetor h_j corresponde às linhas j da matriz H. O objetivo é conseguir, a partir do calculo de WH^T, uma aproximação da matriz A, em que W e H são matrizes $rank - k$.

Do ponto de vista matemático, a representação dos dados é indiferente. Mas, geralmente, utiliza-se uma representação esparsa que é constituída por tuplos *i, j, classificação*. Os tuplos visualizam-se como *i=linha, j=coluna, classificação=valor*. Em virtude das dimensões da matriz A poderem ser muitíssimo elevadas e pelo facto de a matriz ser praticamente toda constituída por zeros (pares i, j que ainda não interagiram), nestes casos a representação esparsa é obrigatória. O que se pretende é prever os pares i, j que ainda não interagiram.

Para determinar o mínimo da Função (2.1), existem vários algoritmos. É possível com um único algoritmo obter diretamente os mínimos exatos W e H a partir da equação de minimização[3] da Função (2.1); no entanto, envolve cálculos muito complexos que não são viáveis, computacionalmente; por isso é um problema de otimização que tem várias formas de ser resolvido. Nesta dissertação, serão apresentados três algoritmos: *ALS, SGD e CCD/CCD++*. Os dois primeiros obtiveram bons resultados na competição promovida pela empresa *Netflix*[1]. O último, segundo os autores [75], consegue unir as vantagens dos dois primeiros.

2.4.2 *Alternating Least Squares (ALS)*

Como a Função (2.1) tem dois mínimos, a ideia deste algoritmo consiste em alternar a otimização entre cada um dos mínimos, otimiza W, mantendo H fixa, e otimiza H, mantendo W fixa. O algoritmo consegue, por isso, tornar a Função (2.1) não convexa numa solução convexa. Quando se fixa H e minimiza W, por forma a, independentemente das linhas de W, ser conseguido um ω_i^* ótimo, é deduzida da Função (2.1) a Função (2.3),

$$f(\omega_i) = \sum_{j \in \Omega_i} (A_{ij} - \omega_i^T h_j)^2 + \lambda ||\omega_i||^2, i = 1...n. \tag{2.3}$$

Para obter os mínimos da Função (2.3), tem-se o seguinte problema,

$$min_{\omega_i} \sum_{j \in \Omega_i} (A_{ij} - \omega_i^T h_j)^2 + \lambda ||\omega_i||^2. \tag{2.4}$$

Para resolver o Problema (2.4) é necessário recorrer à minimização da Função (2.3). Por isso, onde A é uma matriz $m \times n$, ω_i é um vetor linha de uma matriz $W \in \mathbb{R}^{m \times k}$, e h_j o vetor linha de uma matriz $H \in \mathbb{R}^{n \times k}$. O valor de Ω_i corresponde ao subconjunto do conjunto dos índices das colunas de A, de forma a que as entradas respetivas na linha i possuam valores.

[3] Equação de minimização - Em matemática, os pontos críticos de $f(x)$ são obtidos a partir da equação de minimização de $f(x)$, que é dada por $f'(x) = 0$ resolvida em ordem a x.

Exemplo para $m = n = 3$ e $k = 2$.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & - & a_{23} \\ - & a_{32} & - \end{bmatrix}$$

Neste exemplo, tem-se $\Omega_1 = \{1, 2, 3\}$, uma vez que a linha 1 possui todas as entradas definidas. Por outro lado, $\Omega_2 = \{1, 3\}$ e $\Omega_3 = \{2\}$.

Considerando agora k=2 faz-se

$$W = \begin{bmatrix} \omega_{11} & \omega_{12} \\ \omega_{21} & \omega_{22} \\ \omega_{31} & \omega_{32} \end{bmatrix}, H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \\ h_{31} & h_{32} \end{bmatrix}$$

pretende-se minimizar a Função (2.3) para ω_2, tem-se

$$f(\omega_2) = (A_{21} - \omega_{21}h_{11} - \omega_{22}h_{12})^2 + (A_{23} - \omega_{21}h_{31} - \omega_{22}h_{32})^2 + \lambda(\omega_{21}^2 + \omega_{22}^2)$$

calcula-se a derivada parcial em ordem a ω_{21}. Tem-se, então,

$$\frac{\partial f}{\partial \omega_{21}} = -2h_{11}(A_{21} - \omega_{21}h_{11} - \omega_{22}h_{12}) - 2h_{31}(A_{23} - \omega_{21}h_{31} - \omega_{22}h_{32}) + 2\lambda\omega_{21}$$

a equação $\frac{\partial f}{\partial \omega_{21}} = 0$ fica depois de expandir e passar os termos em A_{ij} para o segundo membro,

$$\omega_{21}(h_{11}^2 + h_{31}^2 + \lambda) + \omega_{22}(h_{11}h_{12} + h_{31}h_{32}) = h_{11}h_{12} + h_{31}h_{32}$$

por outro lado, tem-se

$$\frac{\partial f}{\partial \omega_{22}} = -2h_{12}(A_{21} - \omega_{21}h_{11} - \omega_{22}h_{12}) - 2h_{32}(A_{23} - \omega_{21}h_{31} - \omega_{22}h_{32}) + 2\lambda\omega_{23}$$

a equação $\frac{\partial f}{\partial \omega_{22}} = 0$ resultante simplifica-se, do mesmo modo, em

$$\omega_{21}(h_{12}h_{11} + h_{32}h_{31}) + \omega_{22}(h_{12}^2 + h_{32}^2) = h_{12}A_{21} + h_{32}A_{23}$$

colocando em sistema tem-se

$$\begin{cases} \omega_{21}(h_{11}^2 + h_{31}^2 + \lambda) + \omega_{22}(h_{11}h_{12} + h_{31}h_{32}) = h_{11}h_{12} + h_{31}h_{32} \\ \omega_{21}(h_{12}h_{11} + h_{32}h_{31}) + \omega_{22}(h_{12}^2 + h_{32}^2 + \lambda) = h_{12}A_{21} + h_{32}A_{23} \end{cases}$$

em forma matricial

$$\left(\begin{bmatrix} h_{11}^2 + h_{31}^2 & h_{11}h_{12} + h_{31}h_{32} \\ h_{12}h_{11} + h_{32}h_{31} & h_{12}^2 + h_{32}^2 \end{bmatrix} + \lambda I \right) \begin{bmatrix} \omega_{21} \\ \omega_{22} \end{bmatrix} = \begin{bmatrix} h_{11}A_{21} + h_{31}A_{23} \\ h_{12}A_{21} + h_{32}A_{23} \end{bmatrix}$$

onde I denota a matriz identidade. Ora,

$$\begin{bmatrix} h_{11}^2 + h_{31}^2 & h_{11}h_{12} + h_{31}h_{32} \\ h_{12}h_{11} + h_{32}h_{31} & h_{12}^2 + h_{32}^2 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{31} \\ h_{12} & h_{32} \end{bmatrix} \begin{bmatrix} h_{11} & h_{12} \\ h_{31} & h_{32} \end{bmatrix}$$

representando-se por $H_{\Omega_2}^T$ a submatriz de H cujos índices das linhas encontram-se em Ω_2, tem-se

$$H_{\Omega_2}^T = \begin{bmatrix} h_{11} & h_{31} \\ h_{12} & h_{32} \end{bmatrix}$$

e, portanto,

$$\begin{bmatrix} h_{11}^2 + h_{31}^2 & h_{11}h_{12} + h_{31}h_{32} \\ h_{12}h_{11} + h_{32}h_{31} & h_{12}^2 + h_{32}^2 \end{bmatrix} = H_{\Omega_2}^T H_{\Omega_2}$$

representando-se por a_2 o vetor que constitui a linha da matriz A com índice 2, então

$$a_2 = \begin{bmatrix} A_{21} \\ 0 \\ A_{23} \end{bmatrix}$$

portanto

$$\begin{bmatrix} h_{11}A_{21} + h_{31}A_{23} \\ h_{12}A_{21} + h_{32}A_{23} \end{bmatrix} = H^T a_2$$

o sistema escreve-se, finalmente,

$$(H_{\Omega_2}^T H_{\Omega_2} + \lambda I) \begin{bmatrix} \omega_{11} \\ \omega_{21} \end{bmatrix} = H^T a_2$$

o sistema admite uma solução caso

$$|H_{\Omega_2}^T H_{\Omega_2} + \lambda I| \neq 0$$

que se verifica sempre que $\lambda > 0$.

A fórmula de inversão permite escrever

$$\begin{bmatrix} \omega_{11} \\ \omega_{21} \end{bmatrix} = (H_{\Omega_2}^T H_{\Omega_2} + \lambda I)^{-1} H^T a_2$$

Generalização a um número arbitrário de dimensões: Seja então $A \in \mathbb{R}^{m \times n}, W \in \mathbb{R}^{n \times k} e H \in \mathbb{R}^{m \times k}$. Pretendemos calcular os pontos críticos da função

$$f(\omega_i) = \sum_{j \in \Omega_i} (A_{ij} - \boldsymbol{\omega}_i^T \boldsymbol{h}_j)^2 + \lambda ||\boldsymbol{\omega}_i||^2.$$

Definindo-se

$$\boldsymbol{\omega}_i^T = \begin{bmatrix}\omega_{i1} & \dots & \omega_{ik}\end{bmatrix}, \boldsymbol{h}_j = \begin{bmatrix} h_{j1} \\ \vdots \\ h_{jk}\end{bmatrix}$$

como as linhas i e j respetivamente das matrizes,

$$W = \begin{bmatrix}\omega_{11} & \dots & \omega_{1k} \\ \vdots & \ddots & \vdots \\ \omega_{n1} & \dots & \omega_{nk}\end{bmatrix}, H = \begin{bmatrix} h_{11} & \dots & h_{1k} \\ \vdots & \ddots & \vdots \\ h_{m1} & \dots & h_{mk}\end{bmatrix}.$$

Determina-se então a derivada $\frac{\partial f}{\partial \omega_{ri}}$ com $1 \leqslant r \leqslant n$. Os cálculos proporcionam

$$\frac{\partial f}{\partial \omega_{ri}} = -2 \sum_{j \in \Omega_i} h_{jr}(A_{ij} - \boldsymbol{\omega}_i^T \cdot \boldsymbol{h}_j) + 2\lambda\omega_{ri}.$$

A equação $\frac{\partial f}{\partial \omega_{ri}} = 0$ reduz-se a

$$\sum_{j \in \Omega_i} h_{jr}(\boldsymbol{\omega}_i^T \cdot \boldsymbol{h}_j) + 2\lambda\omega_{ri} = \sum_{j \in \Omega_i} h_{jr} A_{ij}.$$

Ora, como

$$\boldsymbol{\omega}_i^T \cdot \boldsymbol{h}_j = \sum_{l=1}^{k} \omega_{il} h_{jl}$$

também

$$\sum_{j \in \Omega_i} h_{jr}(\boldsymbol{\omega}_i^T \cdot \boldsymbol{h}_j) = \sum_{j \in \Omega_i} \sum_{l=1}^{k} \omega_{il} h_{jr} h_{jl} = \sum_{l=1}^{k} \omega_{il} \sum_{j \in \Omega_i} h_{jr} h_{jl}.$$

A equação $\frac{\partial f}{\partial \omega_{ri}} = 0$ escreve-se, portanto, na forma matricial como

$$(M + \lambda I)\boldsymbol{\omega}_i = H^T \boldsymbol{a}_i$$

onde a_i é o vetor correspondente à linha i da matriz A. A matriz M tem entradas

$$M = \begin{bmatrix} \sum\limits_{j \in \Omega_i} h_{j1}^2 & \dots & \sum\limits_{j \in \Omega_i} h_{j1} h_{jk} \\ \vdots & \ddots & \vdots \\ \sum\limits_{j \in \Omega_i} h_{jk} h_{j1} & \dots & \sum\limits_{j \in \Omega_i} h_{jk}^2 \end{bmatrix} = H_{\Omega_i}^T H_{\Omega_i}.$$

Se $|H_{\Omega_i}^T H_{\Omega_i} + \lambda I| \neq 0$ então os pontos críticos de $f(\omega_i)$ são

$$\boldsymbol{\omega}_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T \boldsymbol{a}_i.$$

onde $H_{\Omega_i}^T$ é a submatriz que se obtém de H, considerando apenas as linhas cujos índices

encontram-se em Ω_i.

Mostra-se que, de facto, tem-se sempre $|H_{\Omega_i}^T H_{\Omega_i} + \lambda I| \neq 0$. Ora, sabe-se que

$$|H_{\Omega_i}^T H_{\Omega_i} + \lambda I| = p(-\lambda)$$

sendo $p(x)$ o polinómio característico da matriz $H_{\Omega_i}^T H_{\Omega_i}$. Além disso, é verdade que

$$p(-\lambda) = \lambda^k + Tr(H_{\Omega_i}^T H_{\Omega_i})\lambda^{k-1} + Tr[2](H_{\Omega_i}^T H_{\Omega_i})\lambda^{k-2} + \ldots + |H_{\Omega_i}^T H_{\Omega_i}|,$$

representando $Tr[j](H_{\Omega_i}^T H_{\Omega_i})$ o traço generalizado de ordem j da matriz $H_{\Omega_i}^T H_{\Omega_i}$.

Para determinar o valor de $Tr[j](H_{\Omega_i}^T H_{\Omega_i})$ introduz-se a seguinte notação: dada a matriz $M \in \mathbb{R}^{n\times n}$, é representada por $M[A]$ a submatriz de M constituída pelas linhas e colunas cujos índices se encontram no conjunto A. Define-se ainda I_j^n como sendo o conjunto de todos os subconjuntos de $\{1, 2, \ldots, n\}$ com j elementos. Com esta notação, pode-se escrever

$$Tr[j](H_{\Omega_i}^T H_{\Omega_i}) = \sum_{\alpha \in I_j^n} |(H_{\Omega_i}^T H_{\Omega_i})[\alpha]| = \sum_{\alpha \in I_j^n} |(H_{\Omega_i}[\alpha; *])^T (H_{\Omega_i}[\alpha; *])|.$$

Analisada $H_{\Omega_i}[\alpha; *]$, a submatriz que resulta de H_{Ω_i} tem em consideração apenas os índices da matriz A que têm representação nas linhas. Este subconjunto α, constituído pelos índices anteriormente referidos, mapeia as linhas de H que geram a submatriz de H_{Ω_i}. Com base nas regras sobejamente conhecidas da álgebra linear $|AB| = |A| \cdot |B|$ e $|A^T| = |A|$, e baseando-se, também, na fórmula *identidade de Binet-Cauchy*, obtém-se

$$Tr[j](H_{\Omega_i}^T H_{\Omega_i}) = \sum_{\alpha \in I_j^n} |H_{\Omega_i}[\alpha; *]|^2 \geqslant 0.$$

Segue-se portanto

$$|H_{\Omega_i}^T H_{\Omega_i} + \lambda I| \geqslant \lambda^k > 0$$

como pretendíamos. Resta determinar a natureza do ponto crítico

$$\boldsymbol{\omega}_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T \boldsymbol{a}_i.$$

Para o efeito, é construída a matriz das segundas derivadas, isto é,

$$R_i = \left[\frac{\partial^2 f}{\partial \omega_{ri} \partial \omega_{si}} \right].$$

Ora,

$$\frac{\partial^2 f}{\partial \omega_{ri} \partial \omega_{si}} = 2 \frac{\partial f}{\partial \omega_{si}} \left[-\sum_{j \in \Omega_i} h_{jr} (A_{ij} - \boldsymbol{\omega}_i^T \cdot \boldsymbol{h}_j) + \lambda \omega_{ri} \right] = 2 \left(\sum_{j \in \Omega_i} h_{jr} h_{js} + \lambda \omega_{ri} \omega_{si} \partial_{rs} \right)$$

onde

$$\partial_{rs} = \begin{cases} 1, & r = s \\ 0, & r \neq s \end{cases}$$

corresponde ao símbolo de *Kronecker*. Então, R_i é dada precisamente por

$$R_i = H_{\Omega_i}^T H_{\Omega_i} + \lambda I.$$

Viu-se já que $|H_{\Omega_i}^T H_{\Omega_i} + \lambda I| > 0$. O mesmo artifício permite mostrar que todos os seus menores principais (ver página 494 de [43]) satisfazem a mesma inequação, sendo a matriz, por conseguinte, definida positiva. Está-se, portanto, na presença de um mínimo.

Tal como provado acima, neste momento com H fixa já se pode seguramente escrever,

$$\omega_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T a_i \tag{2.5}$$

que é um ponto crítico, visto que λ é sempre positivo, logo o vetor ω_i^* é mínimo de (2.3).

E para calcular H com W fixa com base na Equação (2.5) fica-se com

$$h_j^* = (W_{\Omega_j}^T W_{\Omega_j} + \lambda I)^{-1} W^T a_j \tag{2.6}$$

O objetivo é minimizar a Função (2.1), por isso, o algoritmo alterna entre W e H, até convergir segundo um número T fixo de iterações definido pelo utilizador [66, 75, 76]. De seguida apresenta-se o ALS no **Algoritmo 1**.

Algoritmo 1: *ALS* [66]

input : A, W, H, λ, T

1 **begin**
2 `inicializar`($H \leftarrow$ *(números aleatórios pequenos)*);
3 **for** *iter* $\leftarrow 1$ **to** T *Step* $= 1$ **do**
4 Calcular W que minimiza (2.1) mantendo H fixa, utilizando $\omega_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T a_i$;
5 Calcular H que minimiza (2.1) mantendo W fixa, utilizando $h_j^* = (W_{\Omega_j}^T W_{\Omega_j} + \lambda I)^{-1} W^T a_j$;

Exemplo do cálculo do vetor ω_0^* na primeira iteração para uma matriz A de classificações, de dimensões pequenas:

Sendo $m = 6\,utilizadores$ e $n = 5\,itens$, a matriz de interação utilizador-item $A^{6\times5}$:

$A =$

utilizador \ item	0	1	2	3	4
0	4	2	0	1	0
1	2	0	0	0	3
2	1	3	0	0	2
3	0	1	0	4	0
4	0	0	3	4	0
5	0	0	0	0	1

Para $k = 3$ e $\lambda = 0.1$, tem-se W inicializada a 0 e H inicializada com números aleatórios pequenos (de modo a garantir que não sejam feitas multiplicações por 0):

$W =$

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

$H =$

	0	1	2
0	0.2	0.7	0.3
1	0.1	0.9	0.5
2	0.4	0.5	0.3
3	0.3	0.1	0.2
4	0.2	0.6	0.8

Então ω_i representa o i vetor linha de W e h_j representa o j vetor linha de H. Com $i = 0$ para calcular ω_0^* é primeiramente necessário extrair os índices coluna j observados na linha $A^{0\times*} = a_0$ dados por Ω_0, por isso $\{h_j : j \in \Omega_0\} \rightarrow j = \{0, 1, 3\}$. Então, neste caso, a matriz H_{Ω_0} resultante fica:

$H_{\Omega_0} =$

	0	1	2
0	0.2	0.7	0.3
1	0.1	0.9	0.5
3	0.3	0.1	0.2

Neste ponto já se tem tudo o que é necessário para calcular $\omega_0^* = (H_{\Omega_0}^T H_{\Omega_0} + \lambda I)^{-1} H^T a_0$,

substituindo e calculando:

$$\omega_0^* = \left(\begin{bmatrix} 0.2 & 0.1 & 0.3 \\ 0.7 & 0.9 & 0.1 \\ 0.3 & 0.5 & 0.2 \end{bmatrix} \times \begin{bmatrix} 0.2 & 0.7 & 0.3 \\ 0.1 & 0.9 & 0.5 \\ 0.3 & 0.1 & 0.2 \end{bmatrix} + 0.1 \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1} \times$$

$$\begin{bmatrix} 0.2 & 0.1 & 0.4 & 0.3 & 0.2 \\ 0.7 & 0.9 & 0.5 & 0.1 & 0.6 \\ 0.3 & 0.5 & 0.3 & 0.2 & 0.8 \end{bmatrix} \times \begin{bmatrix} 4 \\ 2 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Leftrightarrow$$

$$\omega_0^* = \left(\begin{bmatrix} 0.14 & 0.26 & 0.17 \\ 0.26 & 1.31 & 0.68 \\ 0.17 & 0.68 & 0.38 \end{bmatrix} \times \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \right)^{-1} \times \begin{bmatrix} 1.3 \\ 4.7 \\ 2.4 \end{bmatrix} \Leftrightarrow$$

$$\omega_0^* = \left(\begin{bmatrix} 0.014 & 0.026 & 0.017 \\ 0.026 & 0.131 & 0.068 \\ 0.017 & 0.068 & 0.038 \end{bmatrix} \right)^{-1} \times \begin{bmatrix} 1.3 \\ 4.7 \\ 2.4 \end{bmatrix} \Leftrightarrow$$

$$\omega_0^* = \begin{bmatrix} 232.742 & 110.454 & -301.775 \\ 110.454 & 159.763 & -335.306 \\ -301.775 & -335.306 & 761.341 \end{bmatrix} \times \begin{bmatrix} 1.3 \\ 4.7 \\ 2.4 \end{bmatrix} \Leftrightarrow$$

$$\omega_0^* = \begin{bmatrix} 97.438 \\ 89.742 \\ -141.027 \end{bmatrix}$$

Substituindo ω_0^* em W, tem-se:

$W =$

	0	1	2
0	97.438	89.742	−141.027
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

Para calcular as restantes linhas $\omega_1^* \dots \omega_5^*$ de W, o raciocínio é o mesmo que para ω_0, sendo apenas alterado o índice da linha.

Após estarem calculadas todas as linhas de W, a partir de H fixa, é necessário calcular as linhas de H com W fixa; o raciocínio continua igual ao de calcular W. A única diferença é que, para calcular H, passa-se a trabalhar com as colunas da matriz A.

O processo de atualização para W e H tem de ser repetido T vezes. Normalmente, é escolhido um valor T fixo que se considere suficiente para a convergência do algoritmo. A

convergência é atingida quando numa nova iteração H e W continuam idênticas a H e W da iteração anterior.

2.4.3 *Stochastic Gradient Descent (SGD)*

O **Algoritmo 2** tem por base a utilização da fórmula do gradiente [5]. O gradiente é constituído por um vetor de derivadas que permite encontrar a direção da maior variação da Função. Por isso, para minimizar a Função (2.1), utiliza-se um múltiplo do gradiente da Função (2.1) em ordem a ω e outro em ordem a h, isto com base na fórmula do gradiente que permite escrever,

$$\omega_i \leftarrow \omega_i - \eta \left(\frac{\lambda}{|\Omega_i|}\omega_i - (A_{ij} - \omega_i^T h_j)h_j \right), \tag{2.7}$$

$$h_j \leftarrow h_j - \eta \left(\frac{\lambda}{|\bar{\Omega}_j|}h_j - (A_{ij} - \omega_i^T h_j)\omega_j \right), \tag{2.8}$$

em que valor de Ω_i corresponde aos índices das colunas de A com valores na linha i e o valor de $\bar{\Omega}_j$ corresponde ao dos índices das linhas de A com valores na coluna j. A parte estocástica é utilizada a cada iteração para escolher uma linha coluna (i, j) do conjunto Ω, por isso a escolha de índices (i, j) é aleatória. O parâmetro η define o avanço do gradiente a cada iteração, servindo, por isso, para controlar a velocidade de descida. A cada iteração, o vetor linha e o vetor coluna de A aos quais se aplicam as Equações (2.7) e (2.8), respetivamente, são escolhidos de maneira estocástica, sendo, por isso, afetada a convergência; logo, para os mesmos dados de treino, o tempo de aprendizagem varia [4, 5, 21, 75].

Algoritmo 2: *SGD* [21, 75]

input : A, W, H, λ, η

1 **begin**
2   `inicializar(`$W \leftarrow 0, H \leftarrow 0$`);`
3   **while** *not converged* **do**
4         $\omega_i' \leftarrow \omega_i - \eta \left(\frac{\lambda}{|\Omega_i|}\omega_i - (A_{ij} - \omega_i^T h_j)h_j \right)$;
5         $h_j \leftarrow h_j - \eta \left(\frac{\lambda}{|\bar{\Omega}_j|}h_j - (A_{ij} - \omega_i^T h_j)\omega_j \right)$;
6         $\omega_i \leftarrow \omega_i'$;

Na descida do gradiente, começa-se por um ponto qualquer da função obtida pela fórmula do gradiente e vai-se descendo por qualquer um dos lados até ao mínimo global. Na função da Figura 2.3, é fácil encontrar esse mínimo. A existência de vários mínimos locais, conforme ilustrado na Figura 2.4, traduz-se na principal fraqueza deste algoritmo. Neste caso, o algoritmo pode convergir num mínimo que não é o melhor por ser apenas local.

Os mínimos locais são contornados pela introdução de ruído estocástico, criando um efeito idêntico ao ilustrado na Figura 2.5.

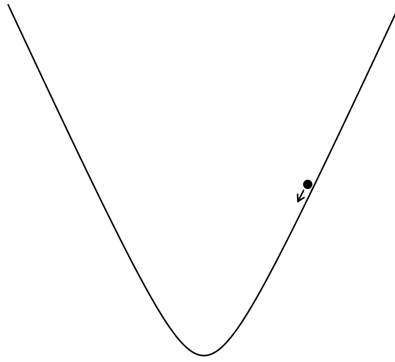


Figura 2.3: Função apenas com um mínimo global.

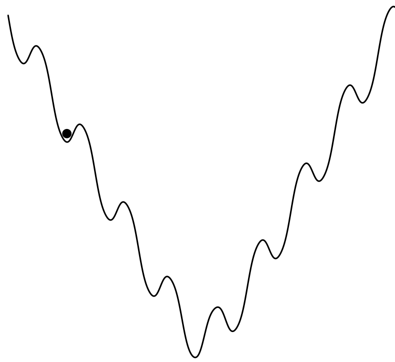


Figura 2.4: Função com mínimos locais.

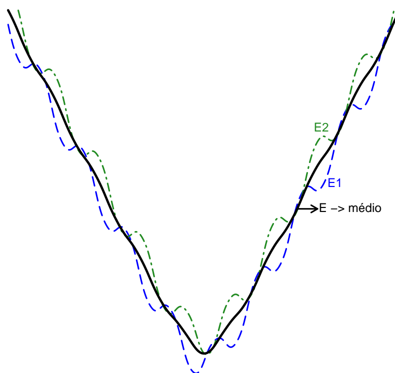


Figura 2.5: Função com os mínimos locais atenuados por ruído estocástico.

2.4.4 *Cyclic Coordinate Descent (CCD)*

Este algoritmo é muito semelhante ao *ALS*, só que, no caso do *ALS*, a atualização é feita sobre a minimização da Função (2.1) em ordem a H ou W. Já no que diz respeito ao *CCD*, a atualização é feita com a minimização da Função (2.1) em ordem a uma única posição de H ou W. Enquanto o *ALS* atualiza H de linha em linha e alterna para W, novamente de linha em linha até convergir, o *CCD* atualiza H de célula em célula e alterna para W também de célula em célula e assim sucessivamente, até convergir [30, 75]. Sendo ω_i o vetor linha referente à linha i de W, então ω_{it} representa o elemento da linha i na coluna t. Para tornar possível a atualização célula a célula, é necessário modificar o Problema (2.2) de forma a que apenas ω_{it} possa modificar o valor da única variável do problema z. Por isso, fixando todas as variáveis, reduz-se ao seguinte,

$$\min_{z} f(z) = \sum_{j \in \Omega_i} (A_{ij} - (\omega_i^T h_j - \omega_{it} h_{jt}) - z h_{jt})^2 + \lambda z^2, \tag{2.9}$$

visto que se trata de fatorização não negativa de matrizes, e a Função (2.9) é invariavelmente quadrática; logo é de concavidade para cima, contendo, garantidamente, um único mínimo. Por isso, basta minimizar a Função (2.9) em ordem a z e obtém-se,

$$z^* = \frac{\sum_{j \in \Omega_i} (A_{ij} - \omega_i^T h_j + \omega_{it} h_{jt}) h_{jt}}{\lambda + \sum_{j \in \Omega_i} h_{jt}^2}. \tag{2.10}$$

Calcular o mínimo z^* por (2.10) requer $O(|\Omega_i|k)$ iterações. No caso do valor de k ser um valor elevado, este passo pode ser otimizado para, logo a partir da primeira iteração, só passar a necessitar de $O(|\Omega_i|)$ iterações, bastando, para isso, manter uma matriz residual R tal que $R_{ij} \equiv A_{ij} - \omega_i^T h_j, \forall (i,j) \in \Omega$. Por isso, com R_{ij} obtido, a minimização $z*$ fica

$$z^* = \frac{\sum_{j \in \Omega_i} (R_{ij} + \omega_{it} h_{jt}) h_{jt}}{\lambda + \sum_{j \in \Omega_i} h_{jt}^2}. \tag{2.11}$$

Com z^* calculado, a atualização de ω_{it} e R_{ij} é feita da seguinte forma

$$R_{ij} \leftarrow R_{ij} - (z^* - \omega_{it}) h_{jt}, \forall j \in \Omega_i \tag{2.12}$$

$$\omega_{it} \leftarrow z^*. \tag{2.13}$$

Após atualizar cada uma das variáveis ω_{it} com (2.13) pertencentes a W, passa a ser necessário atualizar de forma semelhante as variáveis h_{jt} pertencentes a H, em que na primeira

iteração s^* é dado por

$$s^* = \frac{\sum_{i \in \bar{\Omega}_j} (A_{ij} - \omega_i^T h_j + \omega_{it} h_{jt}) \omega_{it}}{\lambda + \sum_{i \in \bar{\Omega}_j} \omega_{it}^2} \tag{2.14}$$

e a matriz residual R é novamente calculada da mesma forma com $R_{ij} \equiv A_{ij} - \omega_i^T h_j, \forall (i,j) \in \Omega$. Por conseguinte, a partir de primeira iteração com R_{ij} obtido, a minimização $s*$ fica

$$s^* = \frac{\sum_{i \in \bar{\Omega}_j} (R_{ij} + \omega_{it} h_{jt}) \omega_{it}}{\lambda + \sum_{i \in \bar{\Omega}_j} \omega_{it}^2}. \tag{2.15}$$

Com s^* calculado, a atualização de h_{jt} e R_{ij} é feita da seguinte forma

$$R_{ij} \leftarrow R_{ij} - (s^* - h_{jt}) \omega_{it}, \forall i \in \bar{\Omega}_j \tag{2.16}$$

$$h_{jt} \leftarrow s^*. \tag{2.17}$$

Tendo as regras de atualização dadas pelas Equações (2.12), (2.13), (2.16) e (2.17), torna-se possível aplicar qualquer sequência de atualização sobre as variáveis de W e H. Serão abordadas duas formas de fazer a atualização: primeiramente, a mais simples que é a *item/user-wise*, e seguidamente, a *feature-wise*.

Atualização *item/user-wise CCD*

Neste tipo de atualização, W e H são atualizadas da seguinte forma:

$$\overbrace{\underbrace{\omega_{11},\ldots,\omega_{1k}}_{\omega_1},\ldots,\underbrace{\omega_{m1},\ldots,\omega_{mk}}_{\omega_m}}^{W},\overbrace{\underbrace{h_{11},\ldots,h_{1k}}_{h_1},\ldots,\underbrace{h_{n1},\ldots,h_{nk}}_{h_n}}^{H}.$$

Na primeira iteração do **Algoritmo 3**, para a matriz residual R ser igual à matriz A, a matriz W é inicializada a 0. O parâmetro T define o número de iterações *CCD*.

Algoritmo 3: *CCD* [75]

```
   input  : A, W, H, λ, k, T
 1 begin
 2     inicializar(W ← 0, R ← A);
 3     for iter ← 1 to T Step = 1 do
 4         for i ← 1 to m Step = 1 do // ▷ Atualiza W.
 5             for t ← 1 to k Step = 1 do
 6                 obter z*, utilizando (2.11);
 7                 atualizar R e ω_it, utilizando (2.12) e (2.13);
 8         for j ← 1 to n Step = 1 do // ▷ Atualiza H.
 9             for t ← 1 to k Step = 1 do
10                 obter s*, utilizando (2.15);
11                 atualizar R e h_jt, utilizando (2.16) e (2.17);
```

A nível de complexidade, pode perceber-se que o cálculo de cada valor ω_{it} em W tem complexidade $O(|\Omega_i|)$ e que o cálculo de cada valor h_{jt} em H tem complexidade $O(|\bar{\Omega}_j|)$. Logo, a complexidade do **Algoritmo 3** é

$$O\left(\left(\sum_i |\Omega_i| + \sum_j |\bar{\Omega}_j|\right) k\right) = O(|\Omega| k).$$

Atualização *feature-wise CCD++*

Na atualização *item/user-wise*, H e W são percorridas pelas linhas; na atualização *feature-wise*, H e W são percorridas pelas colunas. O que garante a igualdade dos cálculos é dado pela observação de WH^T: o cálculo dos produtos internos de vetores linha de W e H é equivalente à soma dos produtos exteriores[4] entre colunas de W e H. Assumindo o exemplo de $A = \begin{bmatrix} 20 & 17 \\ 61 & 50 \end{bmatrix}$, com quatro classificações dadas por dois utilizadores a dois itens, e supondo uma fatorização de A com $k = 3$, as matrizes H e W são

$$W = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 7 & 8 \end{bmatrix} \text{ e } H = \begin{bmatrix} 1 & 5 & 3 \\ 3 & 4 & 2 \end{bmatrix}.$$

Então,

$$W \begin{bmatrix} 1 & 2 & 3 \\ 2 & 7 & 8 \end{bmatrix} \times H^T \begin{bmatrix} 1 & 3 \\ 5 & 4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 20 & 17 \\ 61 & 50 \end{bmatrix} \approx A$$

é equivalente a

$$W \begin{array}{c} \\ \omega_1 \\ \omega_2 \end{array} \begin{array}{ccc} \bar{\omega}_1 & \bar{\omega}_2 & \bar{\omega}_3 \\ \hline 1 & 2 & 3 \\ 2 & 7 & 8 \end{array} \text{ e } H \begin{array}{c} \\ h_1 \\ h_2 \end{array} \begin{array}{ccc} \bar{h}_1 & \bar{h}_2 & \bar{h}_3 \\ \hline 1 & 5 & 3 \\ 3 & 4 & 2 \end{array}$$

$$(\bar{\omega}_1 \otimes \bar{h}_1^T) + (\bar{\omega}_2 \otimes \bar{h}_2^T) + (\bar{\omega}_3 \otimes \bar{h}_3^T) = \begin{array}{c|cc} & 1 & 3 \\ \hline 1 & 1 & 3 \\ 2 & 2 & 6 \end{array} + \begin{array}{c|cc} & 5 & 4 \\ \hline 2 & 10 & 8 \\ 7 & 35 & 28 \end{array} + \begin{array}{c|cc} & 3 & 2 \\ \hline 3 & 9 & 6 \\ 8 & 24 & 16 \end{array} = \begin{bmatrix} 20 & 17 \\ 61 & 50 \end{bmatrix} \approx A.$$

A partir do exemplo acima, como $\bar{\omega}_t$ corresponde aos vetores coluna da matriz W, $\bar{h}_t^T$ equivale aos vetores coluna transpostos da matriz H. $\bar{h}_t$ é um vetor representado em coluna que necessita de ser transposto para indicar que passa a ter uma representação em linha. Logo, é fácil perceber a igualdade da seguinte expressão, em que WH^T pode ser representado como um somatório de k produtos exteriores

$$A \approx WH^T = \sum_{t=1}^{k} \bar{\omega}_t \bar{h}_t^T. \tag{2.18}$$

Com esta conclusão, em vez de W e H serem percorridas de utilizador em utilizador m e de item em item n, respetivamente, W e H passam a ser percorridas de característica latente em característica latente k. Então, $\bar{\omega}_t \in \mathbb{R}^m$, $\bar{h}_t \in \mathbb{R}^n$ e $t \in \mathbb{R}^k$.

Para tornar possível esta forma de pensar, passam a ser necessárias algumas alterações às funções originais do *CCD*. Sendo u^* e v^* os vetores a serem respetivamente injetados sobre $\bar{\omega}_t$

4 Produto exterior - Em inglês *outer product*. Trata-se de um caso especial do produto de Kronecker [69], porque, em vez de aplicado a duas matrizes, é aplicado a dois vetores. O produto exterior aplicado a dois vetores resulta sempre numa matiz, não tendo nada que ver com o muito conhecido produto externo/vetorial. Este último, aplicado a dois vetores, resulta sempre num vetor.

e $\bar{h}_t$, u^* e v^* podem ser calculados recorrendo ao seguinte problema

$$\min_{u \in \mathbb{R}^m, v \in \mathbb{R}^n} \sum_{(i,j) \in \Omega} (R_{ij} + \bar{\omega}_{ti}\bar{h}_{tj} - u_i v_j)^2 + \lambda(||u||^2 + ||v||^2), \tag{2.19}$$

onde $R_{ij} \equiv A_{ij} - \omega_i^T h_j, \forall (i,j) \in \Omega$ é a entrada residual de (i,j); mas nesta forma de atualização, ainda há a possibilidade de se fixarem mais valores pré-calculados, recorrendo a uma segunda matriz residual que é dada por $\hat{R}_{ij}$

$$\hat{R}_{ij} = R_{ij} + \bar{\omega}_{ti}\bar{h}_{tj}, \forall (i,j) \in \Omega. \tag{2.20}$$

Assim, o problema equivalente ao Problema (2.2) é reescrito com

$$\min_{u \in \mathbb{R}^m, v \in \mathbb{R}^n} \sum_{(i,j) \in \Omega} (\hat{R}_{ij} - u_i v_j)^2 + \lambda(||u||^2 + ||v||^2). \tag{2.21}$$

Para obter u^*, basta resolver o Problema (2.21) em ordem a u_i

$$u_i \leftarrow \frac{\sum_{j \in \Omega_i} \hat{R}_{ij} v_j}{\lambda + \sum_{j \in \Omega_i} v_j^2}, i = 1, \ldots, m, \tag{2.22}$$

e, para obter v^*, basta resolver o Problema (2.21) em ordem a v_j

$$v_j \leftarrow \frac{\sum_{i \in \bar{\Omega}_j} \hat{R}_{ij} u_j}{\lambda + \sum_{i \in \bar{\Omega}_j} u_i^2}, j = 1, \ldots, n. \tag{2.23}$$

Note-se que devido à independência dos vetores u^* e v^* em relação às restantes variáveis, para cada k, u^* e v^* apenas dependem um do outro para atingirem a máxima otimização. Logo, com esta hipótese, após o ambiente estar criado por $\hat{R}_{ij}$, basta aplicar *CCD* para otimizar as posições dos vetores u^* e v^*, alternando entre um e outro tantas vezes quantas as necessárias. Por isso, cada conjunto u^* e v^* apenas é visitado uma única vez pelas iterações *CCD*. Segundo [75], uma boa forma para inicializar u^* e v^* faz-se utilizando $\bar{\omega}_t$ e $\bar{h}_t$, respetivamente.

Por fim, u^* e v^* obtidos, basta atualizar $(\bar{\omega}_t, \bar{h}_t)$ e R_{ij} da seguinte forma

$$(\bar{\omega}_t, \bar{h}_t) \leftarrow (u^*, v^*), \tag{2.24}$$

$$R_{ij} \leftarrow \hat{R}_{ij} - u_i^* v_j^*, \forall (i,j) \in \Omega. \tag{2.25}$$

De seguida, apresenta-se o *CCD++* no **Algoritmo 4**.

Algoritmo 4: *CCD++* [75]

```
   input  : A, W, H, λ, k, T
1  begin
2      inicializar(W ← 0, R ← A);
3      for iter ← 1 ... Step = 1 do
4          for t ← 1 to k Step = 1 do
5              construir R̂, utilizando (2.20);
6              for inneriter ← 1 to T Step = 1 do // ▷ T iterações CCD
               para (2.21).
7                  atualizar u, utilizando (2.22);
8                  atualizar v, utilizando (2.23);
9              atualizar (ω̄_t, h̄_t) e R, utilizando (2.24) e (2.25);
```

2.4.5 Comparações entre os algoritmos *ALS, SGD e CCD/CCD++*

Tabela 2.1: Comparação dos algoritmos *ALS, SGD e CCD/CCD++*.

	ALS	*SGD*	*CCD/CCD++*
Complexidade por iteração	$O(\|\Omega\|k^2 + (m+n)k^3))$ [52]	$O(\|\Omega\|k)$	$O(\|\Omega\|k)$
Convergência	Estável	Sensível à parametrização	Estável

Considerando a Tabela 2.1, percebe-se que ambas as sequências de atualização *CCD/CCD++* conseguem juntar os pontos melhores entre o *ALS* e o *SGD*. É conclusivo dizer que *CCD/CCD++*, comparativamente, detêm a complexidade mais baixa aliada a uma convergência estável.

O algoritmo *ALS* minimiza vetor a vetor ω_i ou h_j por inteiro, $\overbrace{\omega_1, \ldots, \omega_m}^{W}, \overbrace{h_1, \ldots, h_n}^{H}$; já *CCD* minimiza ciclicamente posição a posição de ω_{it} ou h_{jt}. Logo, para minimizar cada um dos vetores, necessita de k iterações. No entanto, a expressão de minimização é consideravelmente menos elaborada; por conseguinte, o algoritmo *CCD++* torna-se muitíssimo rápido a otimizar as k posições em cada um dos vetores.

A sequência de atualização aplicada em *CCD++* permite ter um ambiente de atualização notavelmente mais estável que *CCD*. Esta observação justifica-se pelo facto de R_{ij} ser menos vezes atualizado, fazendo com que consiga aproveitar melhor o ambiente de cálculos pré-preparados, levando, consequentemente, a que a convergência seja mais rápida. O problema dos métodos baseados em *CCD* está relacionado com a manutenção de R_{ij}. Nos casos em que a matriz de entrada não é esparsa, R_{ij} torna-se impossível de manter. No entanto, no contexto de sistemas de recomendação, como as matrizes são muito esparsas, é fácil manter R_{ij} para problemas de larga escala.

A convergência do algoritmo *SGD* é sensível à parametrização η e tem problemas com os mínimos locais (ver Secção 2.4.3). Teoricamente, é percetível que os algoritmos *ALS* e *CCD/CCD++* acabam por ter vantagem por garantirem que convergem sempre no mínimo global.

2.5 Avaliação

Em virtude de haver necessidade de avaliar se o modelo gerado pelo algoritmo é bom ou não a prever as classificações dos itens, serão mencionadas algumas das métricas mais utilizadas no contexto de *collaborative filtering*. De um ponto de vista genérico, avaliar um modelo de um sistema de recomendação, baseado em classificações de utilizadores, é semelhante a avaliar um modelo associado a um problema de regressão comum, uma vez que a *variável objetivo* é numérica e contínua. Devido à heterogeneidade dos objetivos de cada sistema de recomendação, não há uma métrica perfeita, mas existem várias que podem ser escolhidas conforme a adequação do que se pretende que o sistema recomende. As métricas são importantes para possibilitar a escolha do algoritmo e o ajuste de parâmetros do Algoritmo [17, 59, 74].

Seguidamente, serão apresentadas algumas das métricas utilizadas para avaliar o desempenho preditivo do modelo gerado pelo algoritmo.

Root Mean Squared Error (RMSE)

A utilização da *RMSE* é muito comum como métrica de erro de propósito geral para previsões numéricas. Comparativamente com a *MAE (Mean Absolute Error)*, a *RMSE* amplifica e pune severamente os erros grandes, logo enfatiza o *recall*.[5] É **pior** recomendar um item que o utilizador não gosta do que **não** recomendar um que ele gosta; logo, o *recall* é a medida com maior importância. A métrica *RMSE*, geralmente, é das mais adequadas para sistemas de recomendação. Não é por acaso que foi a eleita na competição *Netflix Prize*[1] [11, 18, 59, 78].

Sendo n o número de linhas do conjunto de dados de teste, y é a coluna referente à *variável objetivo* que nos dados de teste é conhecida, e $\hat{y}$ é a coluna prevista pelo modelo. Assim,

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}. \tag{2.26}$$

[5] O *recall* pune os falsos negativos; por isso, é 100% quando acerta em todos os que são classificados como positivos. Isto acontece, mesmo que estejam a ser classificados muitos dos positivos como negativos. O *precision* pune os falsos positivos; por conseguinte, é 100% quando classifica todos os positivos que existem. Este facto acontece, mesmo que estejam a ser classificados muitos dos negativos como positivos.

Mean Squared Error (MSE)

A *MSE* é idêntica à *RMSE*; a diferença é a dimensionalidade do resultado. A interpretação da *RMSE* torna-se mais fácil, porque o resultado é da mesma dimensão dos valores previstos [17].

Sendo n o número de linhas do conjunto de dados de teste, y é a coluna referente à *variável objetivo* que nos dados de teste é conhecida, e $\hat{y}$ é a coluna prevista pelo modelo. Assim,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}^i)^2. \tag{2.27}$$

Mean Absolute Error (MAE) / Mean Absolute Deviation (MAD)

No contexto de *collaborative filtering*, a métrica *MAE* não é muito utilizada [41]. *MAE* é uma métrica que enfatiza o *precision*[5] [17, 68].

Sendo n o número de linhas do conjunto de dados de teste, y é a coluna referente à *variável objetivo* que nos dados de teste é conhecida, e $\hat{y}$ é a coluna prevista pelo modelo. Assim,

$$MAE/MAD = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}^i|. \tag{2.28}$$

Normalized Mean Absolute Error (NMAE)

A normalização é interessante quando se pretende fazer comparações do desempenho preditivo, entre conjuntos de dados com escalas diferentes. Assim, os resultados ficam compreendidos entre $[0, 1]$, independentemente da escala presente nos dados de teste [18, 25].

Sendo n o número de linhas do conjunto de dados de teste, y é a coluna referente à *variável objetivo* que nos dados de teste é conhecida, e $\hat{y}$ é a coluna prevista pelo modelo, r_{high} é o valor mais alto possível que a *variável objetivo* pode ter, e r_{low} é o valor mais baixo possível que a *variável objetivo* pode ter. Assim,

$$NMAE = \frac{1}{n \times (r_{high} - r_{low})}\sum_{i=1}^{n}|y_i - \hat{y}^i|. \tag{2.29}$$

2.6 Metodologias

Existem várias metodologias para estimar as métricas supracitadas, como, por exemplo, as apresentadas de seguida.

holdout

Esta metodologia consiste em dividir o conjunto de dados em duas partes, geralmente 80% para treinar o modelo e os restantes 20% para testar o modelo. Estas percentagens podem variar conforme o problema. A divisão no contexto de sistemas de recomendação, de forma geral, é feita, escolhendo aleatoriamente avaliações de alguns ou todos os utilizadores [17].

k-fold cross validation

Consiste em dividir o conjunto de dados em k partes iguais. Dividindo o conjunto de dados D, fica-se com k *folds* F_i; por isso, $D = \{F_1, F_2, \ldots, F_k\}$.

São gerados k modelos, em que os dados de treino para $i = 1$ são $D - F_1$ e os dados de teste são F_1 e assim sucessivamente até $i = k$. Resumidamente, esta metodologia consiste em treinar e testar, tendo por base o particionamento de dados mencionado. Por fim, são obtidos k testes e o resultado final é a média dos testes [17].

Avaliação *online*

Baseia-se em ter o sistema em pleno funcionamento, e verificar se os resultados das previsões são consistentes com o comportamento dos utilizadores [59]. Os métodos *holdout* e *k-fold cross validation* enquadram-se na avaliação *offline*. Nem sempre é possível testar num ambiente real.

2.7 Sumário

Neste Capítulo, foram explicadas as bases necessárias para a compreensão dos sistemas de recomendação. Começou-se por apresentar a definição e aplicação dos sistemas de recomendação. Foram, também, apresentadas várias técnicas de implementação destes sistemas. Foi, ainda, explicado que este trabalho focaliza-se nas técnicas de *collaborative filtering* baseadas em algoritmos de fatorização de matrizes. De seguida, foi dada uma explicação genérica do que é a fatorização de matrizes, e passou-se à explicação de 3 algoritmos utilizados em sistemas de recomendação (*ALS, SGD, CCD++*). Por fim, foram discutidas as métricas e metodologias que geralmente se utilizam para avaliar estes sistemas.

No próximo Capítulo, será discutida a motivação para os sistemas paralelos e quais os limites computacionais. Serão, também, apresentados os conceitos relacionados com sistemas paralelos clássicos e, por fim, com placas gráficas.

Capítulo 3

Sistemas paralelos

Devido à associação intrínseca do limite computacional às leis da física [38], a partir de um certo limite, é impossível aumentar a velocidade dos processadores. Com a tecnologia de processadores atual já não é possível aumentar a velocidade muito mais. No entanto, os limites mantém-se, mesmo utilizando outro tipo de tecnologias tal como processadores quânticos. Apesar da expectativa gerada à volta da descoberta da computação quântica, esta tem vindo a ser estudada há muito tempo, mas o desenvolvimento é lento e penoso [3, 60, 61]. A irreversibilidade lógica é outro problema que limita a computação. Se as operações lógicas fossem reversíveis, era possível criar um computador que só consumisse energia no seu arranque [37]. Isto acontece, porque, a partir da primeira operação lógica, não era necessário gastar mais energia, na medida em que, sendo a operação reversível, não haveria dissipação térmica nem eletromagnética. Logo, a mesma energia era aproveitada para a próxima operação e assim sucessivamente.

Paralelismo é uma estratégia para contornar os problemas de limitação física de um único processador. Permite o aumento do poder computacional, ao adicionar várias unidades de processamento; ao mesmo tempo também permite a resolução de problemas maiores, ao combinar as memórias das várias unidades de processamento.

3.1 Sistemas paralelos clássicos

Tradicionalmente, arquiteturas paralelas são divididas em dois grandes grupos: (1) máquinas com memória partilhada, onde todos os processadores têm acesso uniforme a um único espaço de endereçamento e (2) máquinas de memória distribuída, onde cada processador tem a sua unidade de memória independente. Este trabalho é focado em máquinas de memória partilhada. Os modelos de programação para estas arquiteturas são mais simples e, no contexto deste

trabalho, as implementações utilizadas, como base para comparação, são implementadas em memória partilhada.

Contudo, existem algoritmos que dependem da execução sequencial. Em certos casos pontuais, não é possível paralelizar obtendo vantagem, em virtude de cada linha de execução depender literalmente de toda a execução anterior.

Alguma nomenclatura útil utilizada em sistemas paralelos:

- **Processador -** Unidade física destinada ao processamento dos dados.
- ***Core* -** Núcleo de processamento paralelo de um processador. Um processador com quatro *cores* é capaz de executar simultaneamente quatro tarefas diferentes.
- **Processo -** Nome referente à execução de um programa. O processo está sempre associado ao *main thread*. Nos modelos de programação paralela comuns, os processos não partilham memória entre si. Mesmo assim, há possibilidade de criar vários processos que comuniquem entre si; no entanto, essa comunicação requer trabalho por parte do programador.
- ***Thread* -** Pedaço de código paralelo que é criado pelo processo. Um processo pode controlar várias *threads*, sendo a execução de cada *thread* delegada a um *core*. O custo em tempo de criar uma *thread* é muito inferior ao custo de criar um processo. No ambiente interno do processo, as *threads* criadas por ele dispõem de memória partilhada e sincronismo.

Usando o dobro dos recursos de *hardware*, é expectável que o programa execute em metade do tempo. Mas, nos sistemas paralelos, raramente isso acontece por causa do *overhead* associado. Geralmente, os tipos de *overhead* são os seguintes:

- ***Interprocess Interaction* -** Qualquer sistema paralelo requer comunicação entre os elementos de processamento. O tempo gasto na comunicação normalmente é a principal fonte de *overhead* [26].
- ***Idling* -** Os elementos de processamento, por vezes, são obrigados a esperar (ficam em vazio). Isto acontece devido a diversas razões, tal como carga não balanceada, sincronização e presença de componentes sequenciais no código. Por exemplo, quando um elemento de processamento trabalha numa zona sequencial, os restantes elementos têm de esperar [26].
- ***Excess Computation* -** Muitas vezes, as implementações sequenciais mais rápidas de certos algoritmos são impossíveis de paralelizar. Quando existe uma solução, esta pode residir num aumento do número de operações e numa gestão de memória menos eficiente. Por exemplo, o caso do algoritmo *Fast Fourier Transform* - na versão sequencial, é possível reutilizar os resultados de algumas das operações; na versão paralela, isso já não é possível [26].

3.1.1 Programação em memória distribuída

A programação paralela explícita requer que o programador programe toda a especificação necessária para as tarefas paralelas. Esta especificação envolve sincronização entre tarefas, comunicação dos resultados intermédios e garantia de que os dados requeridos para o processamento de cada um dos elementos estão carregados na memória do respetivo processador. Existem várias linguagens e bibliotecas que suportam a programação paralela explícita. O modelo usado em memória distribuída chama-se *message-passing programming paradigm* e é um dos modelos mais antigos usados na computação paralela. A origem deste modelo vem do início da computação paralela. A razão deve-se ao facto de ser um modelo que pode ser suportado com *hardware* relativamente simples [26]. Uma das bibliotecas mais conhecidas e usadas é a *Message Passing Interface (MPI)* e suporta as linguagens *Fortran*, *C/C++* e *Java* [23, 26]. Atualmente, este modelo de programação é muito utilizado para processamento multinó (várias máquinas ligadas por rede). Este modelo de programação paralela é baseado em *processos*.

No bloco de Código 3.1, é dado um exemplo muito simples de um programa em *MPI*. Este programa faz com que cada um dos processos imprima no ecrã a mensagem *"Hello World!"*. Quando os processos imprimem texto no ecrã, todos concorrem ao mesmo tempo pelo mesmo local de escrita. Por isso, a ordem em que o texto aparece pode não corresponder exatamente à altura em que o processo acabou.

Código 3.1: *Hello World* em *MPI*

```
#include <mpi.h>
#include <stdio.h>
main(int argc, char *argv[])
{
   int nprocs, myrank;
   char name[500];
   MPI_Init(&argc, &argv);//Inicia o ambiente MPI
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);//Conta o número de processos
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);//Identifica o número do processo que está a ser
       executado
   MPI_Get_processor_name(name, &length);//Identifica o nome do nó
   printf("Hello world from process %d of %d running on: %s\n",myrank, nprocs, name);
   MPI_Finalize();//Finaliza o ambiente MPI
}
```

Para compilar o Código 3.1, utiliza-se o comando *shell$ mpicc your_app.c -o your_app.*

Para executar uma aplicação num ambiente multinó, primeiro é necessário criar um ficheiro de texto com o nome de cada uma das máquinas segundo o seguinte formato:

shell$ cat my-hosts

node0 slots=2 max_slots=20

node1 slots=2 max_slots=20

De seguida, para executar, utiliza-se o seguinte comando *shell$ mpirun –hostfile my-hosts -np 8 your_app* em que *-np* define o número de processos e *–hostfile* define o nome do ficheiro com os nomes das máquinas. Usando o comando mencionado, para executar o Código 3.1, produz *output* semelhante ao seguinte:

Hello World I am rank 0 of 8 running on: node0

Hello World I am rank 1 of 8 running on: node0

Hello World I am rank 2 of 8 running on: node1

Hello World I am rank 3 of 8 running on: node1

Hello World I am rank 4 of 8 running on: node0

Hello World I am rank 5 of 8 running on: node0

Hello World I am rank 6 of 8 running on: node1

Hello World I am rank 7 of 8 running on: node1

Para não sair demasiado do escopo desta dissertação, mais pormenores sobre este modelo de programação e esta biblioteca são delegados para a literatura [23, 26, 48].

3.1.2 Programação em memória partilhada

Este modelo de programação necessita que todos os processadores partilhem o mesmo espaço de endereçamento na memória. Consequentemente, este modelo é focado na concorrência e na sincronização das *threads*. Criar *threads* só é possível em memória partilhada. Neste modelo de programação, existem vários mecanismos para gerir a partilha de dados, a concorrência e a sincronização. Geralmente, nos sistemas baseados em processos (memória distribuída), os dados são privados para cada um dos processos. Este tipo de proteção é importante no caso de sistemas para vários utilizadores, mas, geralmente, quando a cooperação dos vários processadores é para resolver um único problema, esta proteção pode ser relaxada. Devido à proteção que os processos têm, faz com que tenham um *overhead* maior em relação às *threads* [26].

Relativamente aos modelos de programação baseados em processos, os modelos de programação baseados em *threads* têm várias vantagens, tal como portabilidade, diminuição da latência na comunicação, escalonamento/balanceamento de carga, e são mais simples de programar. Por outro lado, necessitam de uma maior atenção no que concerne à escrita dos dados em variáveis partilhadas [26].

Grande parte das linguagens de programação suporta a criação de *threads*. Mas como este trabalho é focado na linguagem *C/C++*, para esta linguagem, mencionam-se duas bibliotecas.

A primeira é a *API*[1] *POSIX* que é conhecida por *Pthreads* e é baseada na criação explícita das *threads* pelo programador. Mais pormenores podem ser encontrados na literatura [10, 26]. A segunda é a *OpenMP*[2] [13, 26]. Esta é baseada em diretivas ao pré-processador.

Neste trabalho, nas versões em memória partilhada, o paralelismo é obtido pela utilização da biblioteca *OpenMP*. Por isso, é feita uma descrição dos pormenores considerados principais desta *API*.

Em *OpenMP*, todos os programas iniciam a execução com um processo, a *master thread*. A *master thread* executa sequencialmente até encontrar um construtor paralelo, altura em que cria um *team of threads*. O código delimitado pelo construtor paralelo é executado em paralelo pelo *master thread* e pelo *team of threads*. Ao completar a execução paralela, o *team of threads* sincroniza numa barreira implícita com o *master thread*. Quando o *team of threads* termina a execução, a *master thread* continua a execução sequencial até encontrar um novo construtor paralelo tal como na Figura 3.1. Este modelo é conhecido por *fork-join*.

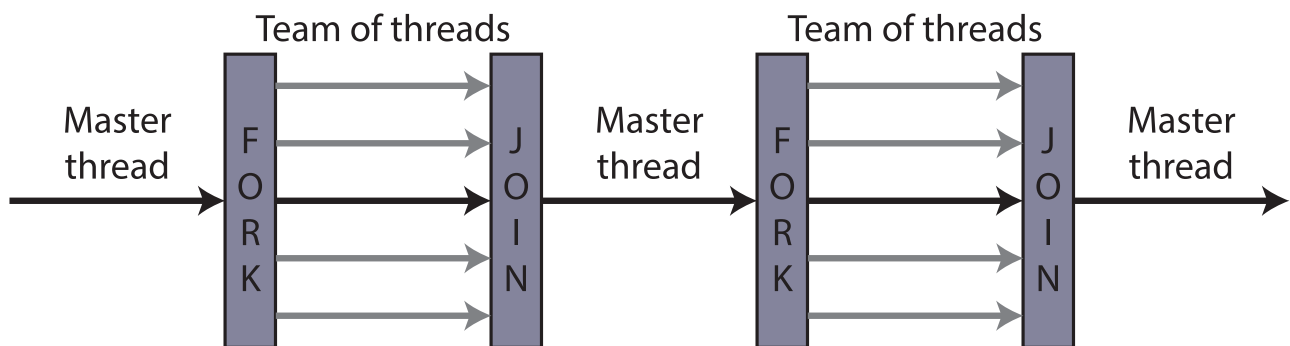


Figura 3.1: Exemplo de um fluxo de execução de uma aplicação em *OpenMP* que usa o modelo *fork-join*.

No bloco de Código 3.2 é mostrada uma estrutura simples para um programa em *OpenMP*.

Código 3.2: Estrutura base de um programa em *OpenMP*

```
#include <omp.h>
...
main(int argc, char *argv[])
{
   ... //região sequencial executada apenas pela master thread
   #pragma omp parallel //construtor paralelo do OpenMP
   {//master thread cria um team of threads
         ... //região paralela executada por todas as threads
   }//team of threads sincroniza com a master thread e termina
   ... //região sequencial executada apenas pela master thread
}
```

1 Application Programming Interface
2 Open Multi-Processing

Descrição de algumas das funções básicas do *OpenMP*:

- ***int omp_get_num_threads(void)*** **-** Retorna o número de *threads* que é utilizado nas regiões paralelas.
- ***int omp_get_max_threads(void)*** **-** Retorna o número máximo de *threads* que o sistema consegue processar em simultâneo (retorna o número de *cores*).
- ***int omp_set_num_threads(void)*** **-** Define o número de *threads* que será utilizado nas regiões paralelas.
- ***int omp_get_num_procs(void)*** **-** Retorna o número de processadores físicos existentes na máquina.
- ***int omp_get_thread_num(void)*** **-** Retorna o identificador da *thread* corrente. As N *threads* a executar numa região paralela são numeradas de 0 a $N-1$ e a *master thread* é sempre identificada pelo número 0.
- ***int omp_in_parallel(void)*** **-** Quando chamada de uma zona paralela retorna o 1, retorna 0, caso seja chamada a partir de uma zona sequencial.

Descrição de duas das diretivas principais do *OpenMP*:

- ***#pragma omp parallel [argumento1,argumento2,...]*** **-** Serve para indicar que o bloco de código passa a ser executado em paralelo. O número de *threads* é definido antes pela *master thread* com a chamada à função *omp_set_num_threads()*, ou, então, é definido pela utilização do argumento *num_threads*. Os argumentos principais desta diretiva são os seguintes:
  - ***if(expr)*** **-** Executa em paralelo, se *expr* for verdadeiro.
  - ***num_threads(val)*** **-** Define que o número de *threads* a ser executado seja igual ao valor *val*.
  - ***private(var1,var2,...)*** **-** Serve para definir que as variáveis têm acesso privado em cada uma das *threads*. O valor inicial é indefinido.
  - ***firstprivate(var1,var2,...)*** **-** Serve para definir que as variáveis têm acesso privado em cada uma das *threads*. As variáveis são iniciadas em cada uma das *threads* com o valor que tinham antes da região paralela.
  - ***shared(var1,var2,...)*** **-** Indica que as variáveis são partilhadas pelas *threads*. Por omissão, todas as variáveis são partilhadas.
  - ***reduction(operador:var)*** **-** A variável *var* é duplicada em cada uma das *threads* e o seu acesso passa a ser privado. No final da execução paralela, é aplicado o *operador* a cada uma das cópias e *var* recebe o resultado final. Alguns dos *operadores* possíveis são $+, -, *, /$.

- ***#pragma omp for [argumento1,argumento2,...]*** **-** É uma diretiva de *work-sharing* colocada antes de um ciclo *for* e serve para o paralelizar. À entrada desta diretiva, não existe nenhuma barreira implícita nem qualquer tipo de sincronismo. Apenas na saída, quando a região demarcada pela diretiva é completa, todas as *threads* sincronizam numa barreira implícita.

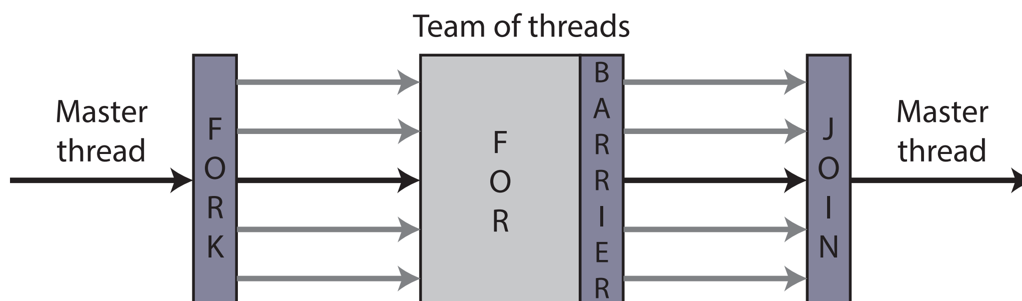


Figura 3.2: Exemplo de um fluxo de execução de um *for* paralelo em *OpenMP*.

Esta diretiva, para além de usar os argumentos mencionados para a diretiva *#pragma omp parallel*, utiliza alguns argumentos próprios. Um deles é o argumento *schedule(type,chunck)*. Este argumento é muito importante, porque serve para definir como é que as N iterações do ciclo devem ser divididas pelas T *threads* da região paralela, configurando, simultaneamente, o funcionamento do balanceamento da carga. O argumento *schedule* é dado pelos seguintes esquemas:

  - ***static*** **-** As iterações são divididas em conjuntos (*chunks*) $\frac{N}{T}$ iterações contínuas que são atribuídas estaticamente a cada *thread*. Este é o esquema atribuído por defeito.
  - ***static, C*** **-** As iterações são divididas em conjuntos de C iterações contínuas que são, depois, atribuídas a cada *thread*.
  - ***dynamic, C*** **-** As iterações são divididas em conjuntos de C iterações contínuas que são, depois, atribuídos dinamicamente a cada *thread*. Esta atribuição é feita sempre que o conjunto anterior acaba de executar. Por omissão, C é 1.
  - ***guided, C*** **-** Idêntico ao *dynamic*, mas as iterações são divididas em conjuntos proporcionais ao número de *threads* e ao número de iterações por executar, decrescendo de forma exponencial até um mínimo de C iterações contínuas por conjunto. Por omissão, C é 1. A ideia é ter um esquema adaptativo em que os conjuntos de iterações começam por ser maiores no início e menores no fim, diminuindo, assim, a probabilidade de uma *thread* ficar sem nada para fazer prematuramente.
  - ***runtime*** **-** O esquema a utilizar é escolhido em tempo de execução a partir da variável de ambiente *OMP_SCHEDULE*.

Seguidamente, serão apresentados alguns exemplos práticos do funcionamento desta biblioteca. Começa-se pelo exemplo clássico *Hello!*, de seguida apresenta-se *reduction* e, por fim, um ciclo *for* em paralelo para reduzir um vetor.

O programa apresentado no bloco de Código 3.3 produz o *output* seguinte:

Thread 2: Hello!

Thread 0: Hello!

Thread 1: Hello!

Thread -1: Bye!

Código 3.3: Exemplo de um programa em *OpenMP*

```
#include <stdio.h>
#include <omp.h>
#define NTHREADS 3

main(int argc, char* argv[]){
int tid = -1, n = NTHREADS;
#pragma omp parallel if(n>=1) num_threads(n) private(tid) shared(n)
   {
      tid = omp_get_thread_num();
      printf("Thread %d: Hello!\n", tid);
   }
   printf("Thread %d: Bye!\n", tid);
}
```

O programa apresentado no bloco de Código 3.4 produz o *output* seguinte:

Thread id: 3

Thread id: 2

Thread id: 1

Thread id: 4

The reduction result is: 10

Código 3.4: Exemplo de *reduction* em *OpenMP*

```
#include <stdio.h>
#include <omp.h>
#define NTHREADS 4

main(int argc, char* argv[]){
int sum=0;
#pragma omp parallel num_threads (NTHREADS) reduction(+:sum)
   {
      sum = omp_get_thread_num() + 1;
      printf("Thread id: %d\n", sum);
   }
   printf("\nThe reduction result is: %d\n", sum);
}
```

O programa apresentado no bloco de Código 3.5 produz o seguinte *output*:

The reduction result is: 34

Código 3.5: Exemplo de um *for* em paralelo com *reduction* em *OpenMP*

```
#include <stdio.h>
#include <omp.h>
#define NTHREADS 4

main(int argc, char* argv[]){
   int myVector[] = { 3, 2, 5, 6, 10, 8};
   int num_threads_old = omp_get_num_threads();
   int sum=0;
   omp_set_num_threads(NTHREADS);

#pragma omp parallel for reduction(+:sum) schedule(static,1)
   for (int x = 0; x < 6; ++x){
      sum += myVector[x];
   }
   printf("The reduction result is: %d\n", sum);
   omp_set_num_threads(num_threads_old);
}
```

Os programas que usam *OpenMP* têm de ser compilados com a opção *-fopenmp*, caso contrário todas as diretivas serão ignoradas.

3.2 Sistemas paralelos usando *General Purpose Graphics Processing Unit (GPGPU)*

Os microprocessadores das placas gráficas (*Graphics Processing Units (GPUs)*) têm muita capacidade de processamento, motivo pelo qual a sua utilização para processamento de propósito genérico (*General Purpose Graphics Processing Unit (GPGPU)*) tem vindo a ganhar cada vez mais notoriedade.

As ferramentas mais utilizadas na programação para estes dispositivos são *Compute Unified Device Architecture (CUDA)* [29, 56, 73] e *OpenCL* [45]. A diferença entre estas duas ferramentas reside no facto de *CUDA* ser proprietária e apenas funcionar em *GPUs NVIDIA*, enquanto *OpenCL* é *Open Source* e funciona em *GPUs NVIDIA*, *AMD* e *Intel*. Para esta dissertação, opta-se pela ferramenta *CUDA*, porque é desenhada especificamente para a arquitetura da *NVIDIA* e, por isso, tira o máximo proveito destas placas. A programação para *GPGPUs* consiste em usar o microprocessador gráfico (*GPU*) contido na placa gráfica para processar, para além de gráficos, programas genéricos que, normalmente, apenas se executam em processadores. A ideia é usar a placa gráfica como se fosse um processador (*CPU*).

Alguma nomenclatura útil para descrever o modelo de programação *CUDA*:

- ***host*** - Máquina principal onde se encontra a/s placa/s gráfica/s conectada/s; refere-se ao/s *NUMA node/s* (*NUMA node* é definido pelo conjunto: *socket* com o *CPU* instalado e respetiva memória *RAM*).

- ***device*** **-** Placa gráfica.

- **Código *kernel* -** Código que é enviado para ser processado no *GPU*.

- **Lançar o *kernel* -** Quando no *host* é feita uma chamada a uma função com código *kernel*, é neste momento que é definido o ambiente de execução com o número de *threads* por *bloco*. Note-se que, previamente, é necessário copiar para a memória do *device* os dados necessários à execução do código.

- **Funções de código *kernel* marcadas como __*global*__ -** Podem ser chamadas a partir do *host* ou a partir do *device*.

- **Funções de código *kernel* marcadas como __*device*__ -** Estas funções __*device*__- apenas podem ser chamadas por funções __*global*__ e dependem do ambiente de *blocos* e *threads* previamente criado.

- ***nvcc*** **-** Compilador desenvolvido especificamente pela *NVIDIA* para compilar o conteúdo dos ficheiros com código *kernel* e extensão *".cu"*.

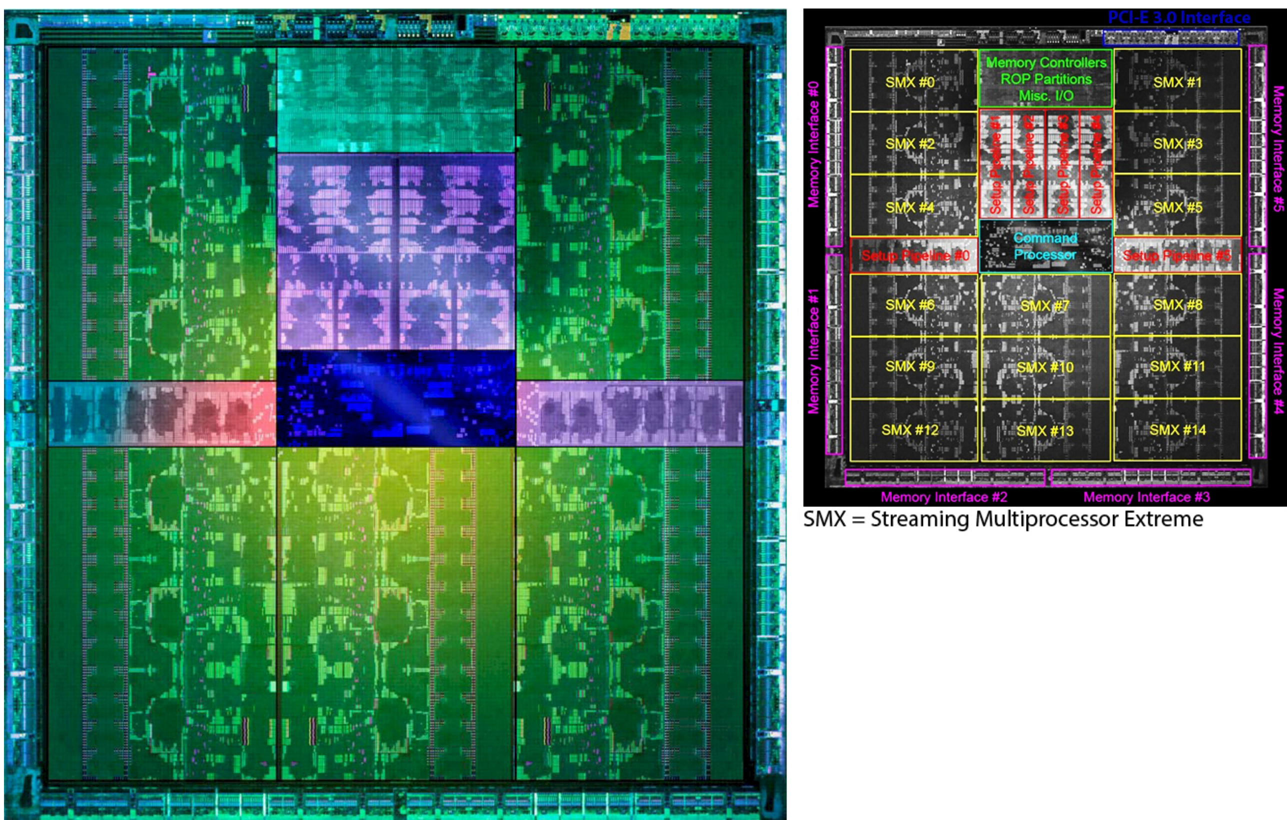


Figura 3.3: [3] Foto do interior de um *GPU Kepler 2 GK110*, neste caso com 15 *multiprocessadores* de 192 *cores* num total de $15 \times 192 = 2880$ *CUDA cores*. Do lado direito, encontra-se a descrição de alto nível de cada uma das unidades principais [49, 62].

3 Baseado em: http://technewspedia.com/more-details-of-nvidia-tesla-cgpu-k20-gk110/

3.2.1 A arquitetura *Compute Unified Device Architecture (CUDA)*

O modelo de programação *Compute Unified Device Architecture (CUDA)*, criado pela *NVIDIA*, é uma plataforma de computação paralela, implementada pelos microprocessadores gráficos (*GPUs*). Uma máquina *host* pode ter um microprocessador gráfico (*GPU*) ou vários microprocessadores gráficos (*GPUs*); a quantidade de microprocessadores gráficos (*GPUs*) disponíveis varia conforme o tipo e o número de placas gráficas instaladas. Geralmente, uma placa gráfica contém um microprocessador gráfico (*GPU*). Este modelo de programação é suportado apenas por *hardware* gráfico *NVIDIA*. Utilizando *CUDA*, os *GPUs* passam a poder ser utilizados como unidades para processamento de propósito geral (*General Purpose Graphics Processing Unit - GPGPU*). Por outras palavras, são utilizados não exclusivamente para gráficos mas sim para executar outras tarefas [29, 56, 73].

Para programar em *CUDA*, é necessário ter em conta um aspeto fundamental que é o *hardware*. Um *GPU* é constituído por vários *multiprocessadores multicore*. Neste contexto, é aqui que surge o conceito de *blocos* e *threads*. Cada *bloco* pode ser constituído por uma ou mais *threads*, e é lançado em cada *multiprocessador*. As *threads*, por sua vez, são lançadas nos *cores* do *multiprocessador*. Quando o número de *blocos* excede o número de *multiprocessadores*, é criada uma fila e, à medida que os *blocos* vão sendo concluídos, entram os *blocos* que estão na fila de espera. O mesmo acontece quando o número de *threads* excede o número de *cores* do *multiprocessador*. Apenas é necessário ter em conta que o número máximo de *threads* por *bloco* é 1024, para as arquiteturas *fermi* e *kepler* (*GPUs NVIDIA*). A nível de sincronização, no *device*, só é possível sincronizar as *threads* dentro de cada *bloco*; a sincronização de *blocos* não é possível, porque a implementação é dispendiosa tanto em desempenho como em custo de fabrico. Para sincronizar os *blocos*, é necessário usar o *host*. É fundamental ter em conta que a memória do *host* é separada da memória do *device*; por isso, cabe ao programador fazer a gestão explícita da memória. A memória do lado do *device* também tem divisões, porque, para além da memória principal, cada um dos *multiprocessadores* tem a sua própria memória (*cache*) que é muito mais rápida em relação à memória principal. Este aspeto é transparente ao programador, dado que, tal como na programação para *CPU*, a memória do *device* pode ser vista como uma só partilhada [29, 56, 73]. Uma das técnicas comuns para reduzir a leitura na memória global chama-se *coalesced memory access*, e consiste em ter as *threads* a lerem consecutivamente a mesma zona de memória, o que permite que o *hardware* junte todas as leituras numa só.

O trabalho é enviado para o *device* do seguinte modo: o fluxo é controlado por uma *thread* na plataforma do *host*, esta *thread*, primeiro copia os dados da memória do *host* para a memória do *device*, de seguida, invoca as *threads* do *device*, executando nelas o *kernel* para processar os dados. Cada *thread* no *device* tem um identificador único, que é utilizado para identificar qual é a parte dos dados que irá processar. Quando o trabalho termina, o *device* envia um sinal para a *thread* no *host*, esta copia os dados da memória do *device* para a memória do *host*, e o fluxo continua.

A Figura 3.4 ilustra como se dividem os *multiprocessadores* com os seus *cores* e respetivas memórias. Neste caso, é ilustrada uma placa *fermi* com 14 *multiprocessadores* de 32 *cores* cada, num total de $14 \times 32 = 448$ *CUDA cores*; no entanto, estes números variam conforme o modelo e arquitetura da placa.

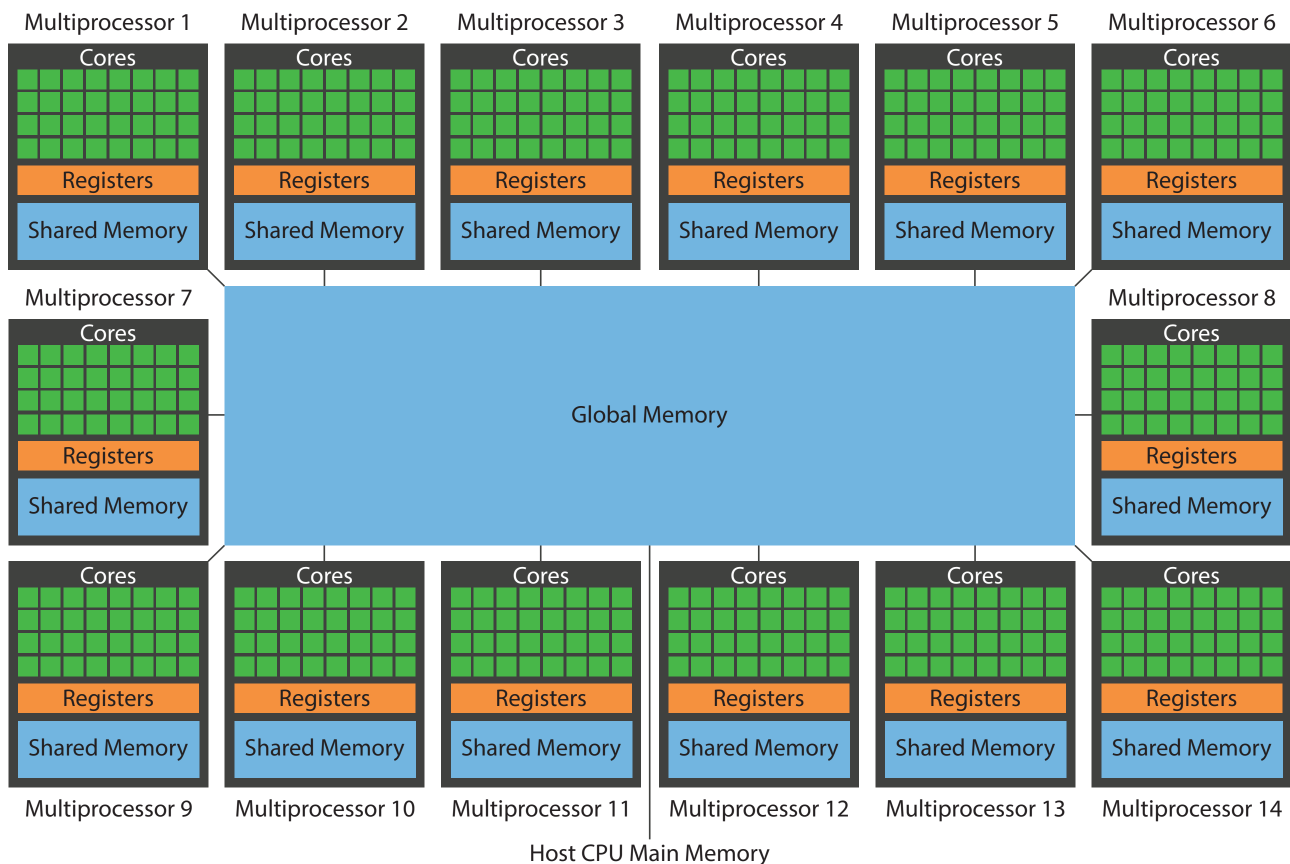


Figura 3.4: [4] lustração de alto nível do *hardware CUDA* numa arquitetura *fermi* em que cada *Streaming Multiprocessor (SM)* é de 32 *cores*.

Os algoritmos ideais para implementar em *CUDA* são aqueles que, nas suas partes paralelizáveis, não requerem sincronismo, devendo, também, ser associados a cálculos matriciais ou vetoriais. Implementar *reduction* em *CUDA*, para além de não ser trivial a eficiência, também é um problema. No caso do algoritmo original, se não se encaixar nestes padrões, é necessário reformulá-lo de forma a ter estas características. Claro está que isto acontece dentro das limitações impostas pelo problema que se pretende resolver. Em alguns algoritmos, a utilização de *CUDA* não é possível; no entanto, para os restantes algoritmos que encaixam neste modelo de programação os resultados são sempre excelentes. Por fim, a forma como o código *kernel* é escrito, também influencia o desempenho, porque em ambas as arquiteturas *fermi* e *kepler* a memória de $64KB$ associada a cada um dos *blocos* é consideravelmente mais rápida em relação à memória partilhada. Por isso, também é interessante na medida do possível, tirar proveito desta *cache* pequena. Devido à latência do *slot PCIe*[5], para possibilitar uma implementação

4 Baseado em: http://www.cs.rit.edu/~ark/lectures/cuda01/
5 Ranhura na *motherboard*, própria para encaixar placas.

eficaz, as transferências de dados *host to device* e *device to host* devem ser tanto quanto possível reduzidas. A arquitetura do *device* é 32*bit*; por conseguinte, é várias vezes mais rápido a efetuar cálculos em *single precision (float)* do que em *double precision (double)*.

O *nvcc* é o compilador desenvolvido pela *NVIDIA* específico para compilar ficheiros com código *kernel*. A linguagem *C* é a única suportada pelo *nvcc*, e a extensão dos ficheiros com código *kernel* é *".cu"*. Graças ao facto de o compilador de *C++* ser compatível com *C*, é possível criar um projeto em *C++* que consuma diretamente funções provenientes de ficheiros *".cu"*. Os ficheiros *cpp* são compilados pelo compilador de *C++*, e os ficheiros *".cu"* pelo *nvcc*. Contudo, *CUDA* não é apenas suportado por *C/C++*, porque qualquer linguagem, tal como *java, c#, python* entre outras, que suportam importação de pacotes, pode ser utilizada, desde que o pacote esteja escrito em *C/C++*.

No exemplo dado pelo programa apresentado no bloco de Código 3.6, o *kernel* é executado numa *thread* de um dos *blocos* da placa gráfica. O objetivo é mostrar o mecanismo básico de funcionamento para a execução de código *kernel*.

Código 3.6: Exemplo de um programa em *CUDA* que soma o valor de duas variáveis

```
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <device_launch_parameters.h>
#include <device_functions.h>
#include <stdio.h>

__global__ void add(int a, int b, int *c) {
   *c = a + b;
}

main(int argc, char* argv[]){
   cudaError_t cudaStatus;
   int c;
   int *dev_c;

   cudaStatus = cudaMalloc((void**)&dev_c, sizeof(int));//aloca memória na placa gráfica
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMalloc failed!");
   }

   add <<<1, 1 >>>(2, 7, dev_c);//chama o kernel add com um bloco de uma thread

   cudaStatus = cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);//copia para o
       host o valor do resultado
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMemcpy failed!");
   }
   printf("2 + 7 = %d\n", c);//imprime no ecrã "2 + 7 = 9"
   cudaFree(dev_c);//liberta a memória que foi alocada na placa gráfica
   return 0;
}
```

No exemplo dado pelo programa apresentado no bloco de Código 3.7, o *kernel* é executado com uma *thread* por *bloco* em um conjunto de 16 *blocos*. Por este motivo, são executadas 16 cópias do *kernel* e cada uma trabalha em zonas diferentes do vetor. Neste caso, cada uma das posições do vetor é copiada em paralelo. Logo, o vetor é todo copiado ao mesmo tempo. Neste exemplo, é mostrado o mecanismo de cópia *device to host*, *host to device*. O *output* produzido pelo programa é o seguinte:

The myVector values:1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

The copyVector current values:1944550800 32763 -2104831048 228 -2103077296 228 -2103160928 228 -2103160880 228 1 33 1 3 1 2

The copyVector final values:1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Código 3.7: Exemplo de um programa em *CUDA* para copiar vetores

```
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <device_launch_parameters.h>
#include <device_functions.h>
#include <stdio.h>

__global__ void copyVector(const int numItens, const int *first, int *returnVec){
   int ii = threadIdx.x + blockIdx.x * blockDim.x;
   for (size_t i = ii; i < numItens; i += blockDim.x*gridDim.x){
      returnVec[i] = first[i];
   }
}

main(int argc, char* argv[]){
   cudaError_t cudaStatus;
   int vecSize = 16,nBlocks=16,nthreadsPerblock=1;
   int myVector[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
   int *cpyVector;
   int *dev_myVector;
   int *dev_cpyVector;
   cpyVector = (int*)malloc(vecSize*sizeof(int));
   printf("The myVector values:");
   for (size_t x = 0; x < vecSize; ++x){
      printf("%d ", myVector[x]);
   }
   printf("\nThe copyVector current values:");
   for (size_t x = 0; x < vecSize; ++x){
      printf("%d ", cpyVector[x]);
   }
   cudaStatus = cudaMalloc((void**)&dev_myVector, vecSize*sizeof(int));
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMalloc failed!");
   }
   cudaStatus = cudaMalloc((void**)&dev_cpyVector, vecSize*sizeof(int));
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMalloc failed!");
   }

   cudaStatus = cudaMemcpy(dev_myVector, myVector, vecSize*sizeof(int),
    cudaMemcpyHostToDevice);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMemcpy failed!");
```

```
    }

    copyVector <<<nBlocks, nthreadsPerblock>>>(vecSize, dev_myVector, dev_cpyVector);

    cudaStatus = cudaMemcpy(cpyVector, dev_cpyVector, vecSize*sizeof(int),
         cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
       fprintf(stderr, "cudaMemcpy failed!");
    }

    printf("\nThe copyVector final values:");
    for (size_t x = 0; x < vecSize; ++x){
       printf("%d ", cpyVector[x]);
    }

    cudaFree(dev_myVector);
    cudaFree(dev_cpyVector);
    free(cpyVector);
    return 0;
}
```

Os restantes pormenores são delegados para a literatura [29, 56, 73].

3.2.2 O futuro das placas gráficas

No presente ano de 2014, a velocidade máxima de comunicação entre a memória do *GPU* e *CPU* depende da velocidade da *PCIe x16*, que na sua versão 3.0 atinge o máximo de *8 gigabyte* por segundo [51]. Mas, para o ano 2016, está anunciado o lançamento de uma nova tecnologia chamada *NVLink*, criada pela *NVIDIA* juntamente com a *IBM*. Relativamente aos *17 gigabyte* por segundo entre o *CPU* e a memória *DDR4* lançada agora em 2014 [63], esta tecnologia vai permitir uma comunicação muito mais rápida entre o *GPU* e o *CPU*. A velocidade de transferência de dados por *NVLink* irá ser compreendida entre os *80* e *200 gigabyte* por segundo [20, 44]. Note-se que, em 2016, a memória *standard* para *CPU* será a *DDR4*. Pelo que está anunciado, esta conexão revolucionária apenas irá estar disponível na plataforma *tesla*. Os únicos processadores que irão suportar a conexão *NVLink* serão os processadores *Power* desenvolvidos pela *IBM*. Estes processadores são específicos para processamento simétrico em servidores de larga escala [14, 15, 44].

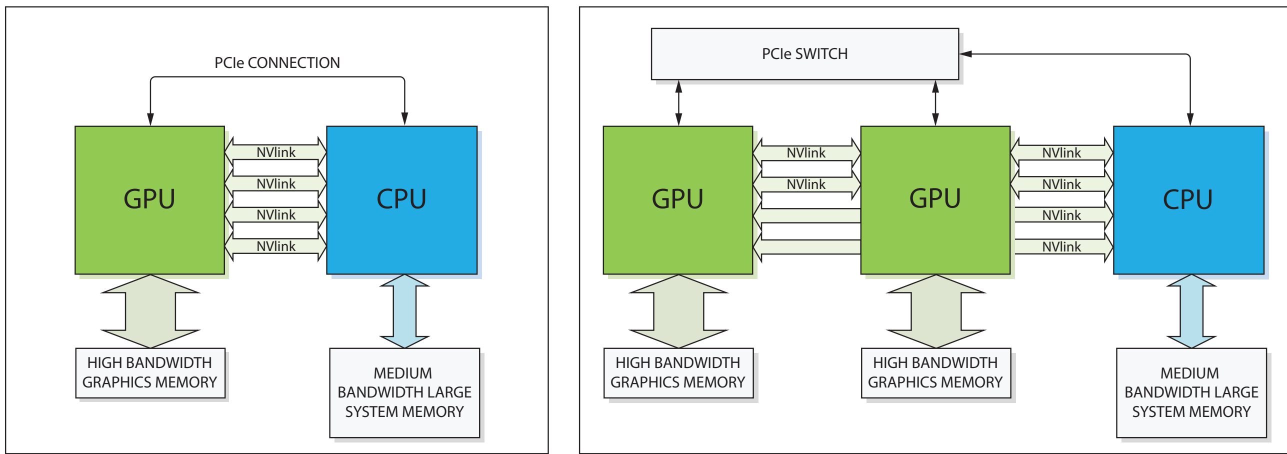


Figura 3.5: [6] Ilustração da conexão *NVLink* entre *GPUs* e *CPU*; do lado esquerdo, apenas um *GPU* conectado a um *CPU*; do lado direito, dois *GPUs* conectados a um *CPU*.

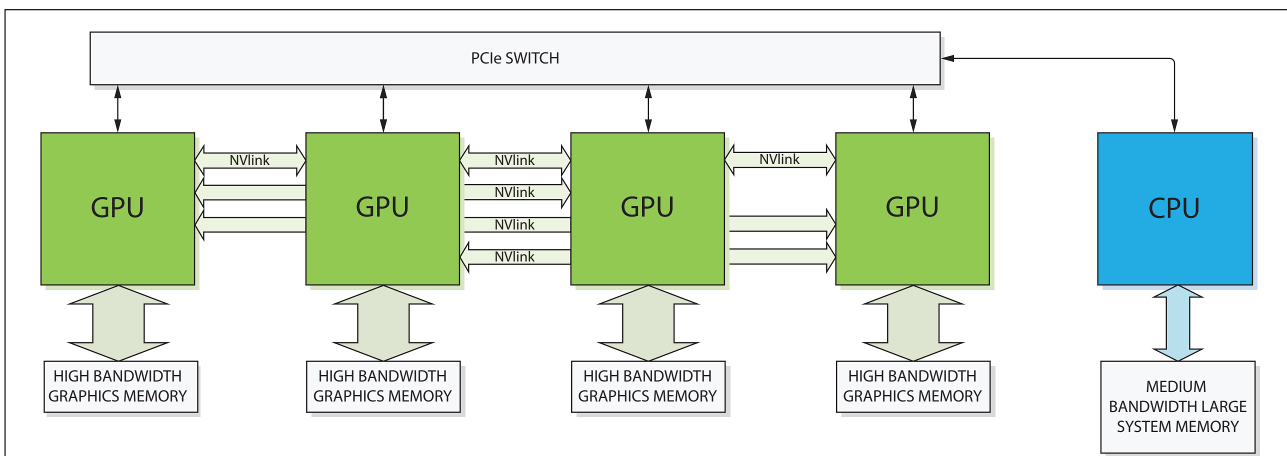


Figura 3.6: [6] No caso do *CPU* não suportar *NVLink*, tal como atualmente em 2014, toda a comunicação passa a ser pela *PCIe*. Mas entre *GPUs* continua a ser possível utilizar *NVLink*.

Outra tecnologia que está anunciada para 2016 é a *Stacked Memory*. Esta será a atualização da *GDDR5* existente nas placas atuais. Vai suportar velocidades de transferência de dados na ordem de 1 *terabyte* por segundo. A *Stacked Memory* é uma tecnologia que permite o empilhamento vertical de várias camadas de componentes *DRAM* sobre a placa. Comparativamente com a *GDDR5*, a *Stacked Memory*, para além de fornecer uma largura de banda várias vezes superior, em densidade, fornece mais do dobro da capacidade e quadruplica eficiência energética. Pelo facto de ser empilhada, permite um melhor aproveitamento do espaço, tornando possível colocar os *VRMs*[7] mais perto do *GPU* o que se traduz numa melhor eficiência. [20, 44]

Em 2016, o nome da arquitetura que irá suportar as tecnologias *NVLink* e *Stacked Memory*, será *Pascal*. E trará, para além de placas mais compactas e muito mais poderosas, uma eficiência energética, de longe, nunca antes alcançada.

[6] Imagem retirada de:
http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/

[7] *VRM* - Voltage Regulator Module

3.3 Métricas de desempenho

Existem várias métricas para avaliar o desempenho de um sistema paralelo. Na literatura [26], encontram-se métricas para avaliar o *overhead*, o *speedup*, a eficiência, os efeitos da granularidade, etc. Nesta dissertação, a métrica usada é o *speedup* S que é dado pela Fórmula 3.1, em que T_s é o tempo que demora a execução sequencial, T_p é o tempo que demora entre o início do processamento paralelo e o fim de todos os elementos de processamento

$$S = \frac{T_s}{T_p}. \tag{3.1}$$

O valor ideal do *speedup* é quando a eficiência E, que é dada pela Fórmula 3.2, tem valor $E = 1$. Embora seja possível obter este valor, em alguns casos, também, é possível $E > 1$. Nesta situação, o *speedup* é super-linear. Isto acontece quando são maximizados os acessos às memórias *cache* dos processadores. Ter mais unidades de processamento físicas aumenta a quantidade de *cache* disponível

$$E = \frac{S}{p}. \tag{3.2}$$

3.4 Sumário

Neste Capítulo, começou-se por falar um pouco do futuro e limitação dos sistemas informáticos, e apresentou-se a motivação para os sistemas paralelos. De seguida, apresentaram-se os sistemas paralelos clássicos; por fim, foi referido o modelo de programação *GPGPU* e métricas de desempenho.

No próximo Capítulo, será apresentada a paralelização *multicore* de 3 algoritmos de recomendação baseados em fatorização de matrizes (*ALS, SGD, CCD++*).

Capítulo

4

Paralelização de algoritmos de recomendação

Neste Capítulo, serão discutidas as estratégias de paralelismo dos algoritmos descritos no Capítulo 2. Embora o foco, como mencionado anteriormente, seja as arquiteturas de memória partilhada, no Capítulo seguinte, será discutida a paralelização em *GPGPUs*.

4.1 Paralelização do algoritmo *ALS*

A primeira utilização deste algoritmo para fatorização não negativa de matrizes foi publicada em 1994 por Paatero *et. al.* [47]. Devido às características inatas, desde sempre foi, um algoritmo paralelizável.

A estratégia de paralelização do algoritmo *ALS* consiste em dividir equitativamente as linhas das matrizes W e H pelas *threads*, tal como descrito no **Algoritmo 5** [78].

Algoritmo 5: Versão paralela do algoritmo *ALS* para sistemas *multicore* [78]

input : A, W, H, λ, T

1 **begin**

2   `inicializar(`$H \leftarrow$ *(números aleatórios pequenos)*`)`;

3   **for** *iter* $\leftarrow 1$ **to** T *Step* $= 1$ **do**

4   Calcular em paralelo W que minimiza (2.1), mantendo H fixa, utilizando $\omega_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T a_i$; `// Sincroniza`;

5   Calcular em paralelo H que minimiza (2.1), mantendo W fixa, utilizando $h_j^* = (W_{\Omega_j}^T W_{\Omega_j} + \lambda I)^{-1} W^T a_j$; `// Sincroniza`;

A cada *thread* corresponde um conjunto de linhas de W. Devido ao facto de a matriz H estar fixa, as linhas ω_i de W podem ser calculadas de forma independente. Supondo que se

pretendem usar d *threads*, são definidos d conjuntos disjuntos $S_1, ..., S_d$ de índices de linhas de W; então, $W_{S_{threadID}}$ é calculado em paralelo, sendo $threadID$ o número que identifica a *thread*. Por exemplo, para $d = 4$, os valores de $threadID$ variam entre $threadID = 1, ..., threadID = 4$. O mesmo raciocínio é aplicado para a matriz H.

4.2 Paralelização do algoritmo *SGD*

O *SGD* tem um carácter intrinsecamente sequencial, o que torna a sua paralelização mais complexa. No entanto, existem algumas implementações paralelas, tal como *DSGD* [21], *HOGWILD* [46, 79] entre outras. Não é por acaso que a sua primeira implementação paralela foi apresentada em 2010 [75, 80].

4.2.1 Versão *Distributed Stochastic Gradient Descent (DSGD)*

Distributed Stochastic Gradient Descent (DSGD) é uma estratégia de paralelismo proposta por Gemulla *et. al.* [21].

A implementação *DSGD* tem por base uma separação estratificada por utilizadores da matriz A. Por isso, A passa a ser um conjunto S de q estratos $S = \{A_1, \ldots, A_q\}$ em que o processamento de cada estrato $A_e \subseteq A$ pode ser feito paralelamente. Cada estrato A_e tem que ser *d-monomial*, o que significa que tem de poder ser particionado em d conjuntos não vazios $A_e^1, A_e^2, \ldots, A_e^d$, tal que $i \neq i'$ e $j \neq j'$, sempre que $(i,j) \in A_e^{b_1}$ e $(i',j') \in A_e^{b_2}$ para $b_1 \neq b_2$. Então a matriz de treino A_e é *d-monomial* apenas se for constituída por um estrato *d-monomial*. O conjunto de estratos tem de cobrir por inteiro o conjunto de dados de treino A, pelo que $\sum_{e=1}^{q} A_e = A$, para tal a sobreposição de estratos é permitida. O paralelismo é obtido pelo parâmetro d que define o número de *threads* pretendido, tal como descrito no **Algoritmo 6**.

Algoritmo 6: Versão *DSGD* do algoritmo *SGD* para sistemas *multicore* [21]

```
   input : A, W, H, λ, η, d
1  begin
2      inicializar(W ← 0, H ← 0);
       // Particionar Z/W/H em respetivamente d × d/d × 1/1 × d blocos;
3      while not converged do
4          for t ← 1 to d Step = 1 do
5              Gerar d blocos {A^{1j_1}, ..., A^{dj_d}};
6              for b ← 1 to d Step = 1 do // For em paralelo.
7                  Executar SGD em A^{bj_b}, utilizando as equações (2.7) e (2.8)
               // Sincroniza as threads.
```

O facto da matriz A estar dividida em estratos ou submatrizes permite que cada um desses estratos seja atualizado em paralelo. A cada iteração, as *threads* necessitam de ser sincronizadas [79].

A matriz A é esparsa, por conseguinte, os estratos só têm os valores provenientes das interações utilizador-item.

4.2.2 Versão *Hogwild* do algoritmo *SGD*

Esta versão parte da ideia de que a matriz de interação utilizador-item A é muito esparsa, e assume que entre duas amostras a atualização é equivalente a ser independente. A razão prende-se com o facto de entre duas amostras nunca ser atualizado o mesmo utilizador-item. Logo, as atualizações podem ser executadas em paralelo, e, com base nos pressupostos explicitados anteriormente, não necessitam de sincronização. Como não há garantias de atualização atómica, uma variável pode, ocasionalmente, ser atualizada por duas *threads*. Assim, logo poderá afetar a convergência, embora esta, geralmente, ocorra porque a matriz de interação utilizador-item A é muito esparsa [46, 79]. A versão *Hogwild* é apresentada no **Algoritmo 7**.

Algoritmo 7: Versão *Hogwild* do algoritmo *SGD* para sistemas *multicore* [46, 79]

```
  input : A, W, H, λ, η, d
1 begin
2   inicializar(W ← 0, H ← 0);
3   for b ← 1 to d Step = 1 do // For em paralelo.
4     while not converged do
5       Selecionar de A aleatoriamente uma instância a_{u,v};
6       Atualizar ω_u com a equação (2.7) e atualizar h_v com a equação (2.8);
```

Este algoritmo tem a vantagem de não necessitar de sincronização. Mas, devido à possibilidade de, por vezes, não convergir, deve ser utilizado com precaução.

4.3 Paralelização do algoritmo *CCD++*

No algoritmo *CCD++*, a solução aproximada é obtida atualizando u e v, alternadamente. Quando v é fixo por (2.22), cada variável u_i é atualizada independentemente. Portanto, a atualização de u pode ser dividida em m tarefas independentes e tratada em vários *cores*.

A título de exemplo, numa máquina com p *cores*, são definidas as partições dos índices linha de $W, \{1, \ldots, m\}$ como sendo $S = S_1, \ldots, S_p$. O vetor u fica decomposto em p vetores $u^1, u^2, \ldots, u^p$, sendo u^r o subvetor de u correspondente a S_r. Quando a matriz W é dividida em pedaços iguais $|S_1| = |S_2| = \ldots = |S_p| = \frac{m}{p}$, em virtude de o tamanho dos vetores linha (Ω_i), contidos em A, variar, surge um problema de balanceamento de carga. Neste caso, o trabalho exato para que cada r *core* atualize u^r é dado por $\sum_{i \in S_r} 4|\Omega_i|$, logo o trabalho não é o mesmo para todos os *cores*. Este é um dos problemas deste algoritmo que pode ser contornado, utilizando o escalonamento dinâmico fornecido pela maioria das livrarias de processamento paralelo, tal como *OpenMP* [75], entre outras.

Partindo dos aspetos referidos em [75], poder-se-á afirmar que, para cada subproblema, cada

core r constrói $\hat{R}$ com

$$\hat{R}_{ij} \leftarrow R_{ij} + \bar{\omega}_{ti}\bar{h}_{tj}, \forall(i,j) \in \Omega_{S_r}, \tag{4.1}$$

onde $\Omega_{S_r} = \cup_{i \in S_r}\{(i,j) : j \in \Omega_i\}$. Então, para cada *core* r constata-se

$$u_i \leftarrow \frac{\sum\limits_{j \in \Omega_i} \hat{R}_{ij} v_j}{\lambda + \sum\limits_{j \in \Omega_i} v_j^2}, \forall i \in S_r. \tag{4.2}$$

A atualização de H é feita com o mesmo princípio que em W por (4.2). Por isso, para p *cores* os índices linha de $H\{1, \ldots, n\}$ são particionados em $G = G_1, \ldots, G_p$. Então, para cada *core* r constata-se

$$v_j \leftarrow \frac{\sum\limits_{i \in \bar{\Omega}_j} \hat{R}_{ij} u_j}{\lambda + \sum\limits_{i \in \bar{\Omega}_j} u_i^2}, \forall j \in G_r. \tag{4.3}$$

Como todos os *cores* partilham a mesma memória, não é necessário haver comunicação para aceder a u e v. Portanto, após obter os vetores (u^*, v^*), a atualização de R e dos vetores $(\bar{\omega}_t^r, \bar{h}_t^r)$ é também implementada em paralelo pelos r *cores* da seguinte forma

$$(\bar{\omega}_t^r, \bar{h}_t^r) \leftarrow (u^r, v^r), \tag{4.4}$$

$$R_{ij} \leftarrow \hat{R}_{ij} - \bar{\omega}_{ti}\bar{h}_{tj}, \forall(i,j) \in \Omega_{S_r}. \tag{4.5}$$

A versão *multicore* do *CCD++*, anteriormente descrita, está representada no **Algoritmo 8**.

Algoritmo 8: Versão paralela do algoritmo *CCD++* para sistemas *multicore* [75]

```
   input  : A, W, H, λ, k, T
1  begin
2      inicializar(W ← 0, R ← A);
3      for iter ← 1 ... Step = 1 do
4          for t ← 1 to k Step = 1 do
5              em paralelo construir R̂ dividido por r cores utilizando (4.1);
6              for inneriter ← 1 to T Step = 1 do
7                  em paralelo atualizar u dividido por r cores, utilizando (4.2);
8                  em paralelo atualizar v dividido por r cores, utilizando (4.3);
9              em paralelo atualizar (ω̄_t^r, h̄_t^r), utilizando (4.4);
10             em paralelo atualizar R, utilizando (4.5);
```

4.4 Sumário

Neste Capítulo, foram apresentados os métodos de paralelização dos algoritmos (*ALS, SGD, CCD++*).

Na Tabela 4.1 é exposta uma comparação entre algoritmos apresentados neste Capítulo.

Tabela 4.1: Considerando a informação anteriormente apresentada, poder-se-á verificar que, a nível da paralelização, os algoritmos evidenciam aspetos mais/menos vantajosos.

<table>
<tr><th colspan="2">Algoritmo</th><th>Vantagens</th><th>Desvantagens</th></tr>
<tr><td colspan="2">ALS</td><td>• Convergência estável.
• Garantia de que as threads não acedem, para escrita, à mesma posição de memória (a escrita é atómica).</td><td>• Necessidade de sincronização e consequente consumo de recursos.</td></tr>
<tr><td rowspan="2">SGD</td><td>DSGD</td><td>• Garantia de que as threads não acedem, para escrita, à mesma posição de memória (a escrita é atómica).</td><td>• Necessidade de sincronização e consequente consumo de recursos.
• Consumo de recursos para gerar o conjunto de q estratos.
• A paralelização do SGD é complexa devido ao carácter intrinsecamente sequencial.</td></tr>
<tr><td>Hogwild</td><td>• Não necessita de pontos de sincronização.</td><td>• As threads podem ocasionalmente escrever na mesma posição de memória, implicando a não convergência.
• Necessita de precaução na utilização, porque pode nunca convergir.
• A paralelização do SGD é complexa devido ao carácter intrinsecamente sequencial.</td></tr>
<tr><td colspan="2">CCD++</td><td>• Convergência estável.
• Garantia de que as threads não acedem, para escrita, à mesma posição de memória (a escrita é atómica).</td><td>• Necessidade de sincronização e consequente consumo de recursos.
• Granularidade mais fina, necessitando de uma maior quantidade de pontos de sincronismo relativamente ao ALS.</td></tr>
</table>

No próximo Capítulo, serão apresentados os métodos de paralelização em *GPU* dos algoritmos *ALS* e *CCD++*.

Capítulo 5

Paralelização para *GPU* de dois algoritmos de recomendação

A tecnologia *CUDA* é desenvolvida pela *NVIDIA Corporation* e permite utilizar o *hardware* gráfico como máquina de processamento paralelo. Pelas razões apresentadas pelos autores de [75], o algoritmo *CCD++* demonstra ser uma boa solução de paralelização. Pelo facto de ser um algoritmo associado a cálculos matriciais e não necessitar de sincronização nas suas partes paralelizáveis, detém boas características que possibilitam uma implementação eficiente em *CUDA*. Contudo, devido à popularidade do algoritmo *ALS*, permanece interessante averiguar a sua potencialidade quando implementado em *CUDA*. Por isso, é também feita uma implementação de raiz do algoritmo *ALS* e, seguidamente, a sua paralelização.

No contexto de sistemas de recomendação paralelos baseados em fatorização de matrizes, existe muito trabalho feito, tal como demonstrado no Capítulo anterior. No entanto, geralmente, o que existe é para ambientes *multicore* ou *multinó*, não se conhecendo nenhuma implementação em *CUDA* ou para *GPU* dos algoritmos mencionados. Assim, pretende-se verificar até que ponto estes algoritmos podem ser implementados nessa tecnologia e analisar as vantagens ou desvantagens em relação a implementações *multicore*. Esta investigação poderá ser do interesse geral, na medida em que, relativamente aos *CPUs*, os *GPUs* têm um baixo consumo energético e o custo por *core* muito inferior.

5.1 Implementação do algoritmo *CCD++* em *CUDA*

LIBPMF[1] é uma biblioteca *open source* para *linux* elaborada pelos autores do algoritmo *CCD++*. Esta contém a implementação paralela do algoritmo *CCD++* [75]. A implementação da *LIBPMF* é em *C++* e é para ambientes *multicore* em memória partilhada; o paralelismo utiliza a biblioteca *OpenMP*.

Na *LIBPMF* existem dois projetos: O *pmf-train* que é para gerar o modelo para um conjunto de dados de treino. E o *pmf-predict* que é para medir o *RMSE* para um conjunto de dados de teste.

A implementação em *CUDA* do algoritmo *CCD++* tem por base a implementação da *LIBPMF*. Para converter para *CUDA* a versão paralela em *OpenMP* da *LIBPMF*, utiliza-se o projeto *pmf-predict*. Os passos tomados são os seguintes:

1. Alteração do código para ser compilável em *Windows* e *Linux*. Em *Windows*, o *IDE*[2] selecionado é o *Microsoft Visual Studio 2013*. No *Microsoft Visual Studio 2013*, o projeto é configurado para compilar *C++*, utilizando o compilador do *IDE*[2], e para compilar os ficheiros *".cu"*, utilizando o *nvcc*. Em *Linux*, como o código foi alterado para ser o mesmo que em *Windows*, não é utilizado nenhum *IDE*[2], apenas se recorre a um *makefile* e utiliza-se o compilador *g++* para compilar *C++* e o *nvcc* para compilar os ficheiros *".cu"*. Eventuais edições de código são realizadas com recurso a um editor de texto comum. Para finalizar este passo, é feita a análise do funcionamento do código da *LIBPMF*, com o intuito de saber e entender quais as funções que implementam o algoritmo *CCD++*.

2. Após entender o passo anterior, visto que na *LIBPMF* todas as funções estão em *double precision*, é feita uma reimplementação dessas funções para passarem a utilizar *float (single precision)*, dado que o *device* detém um comportamento muito mais eficaz em *float*.

3. Modificação das funções em *C++* que implementam o algoritmo *CCD++*, para passarem a ser compiladas em *extern "C"* [3]. Para que isto seja possível, é necessário fazer uma ponte que permita a passagem de dados entre o ambiente *C++* e *C*.

4. Alteração das funções em *C*, para ficarem de acordo com o modelo de programação *CUDA* utilizando a gestão de memória explícita.

Nos passos supracitados, em relação à implementação de Yu *et. al.* [75], é garantida a equivalência medida em *RMSE* (ver Secção 2.5).

O balanceamento da carga entre *threads*, na versão paralela do algoritmo *CCD++*, não é uniforme [75]. Este problema acontece, pelo facto de o número de classificações associado a cada

---

1 *LIBPMF* - Library for Large-scale Parallel Matrix Factorization

2 *IDE* - Integrated Development Environment é um software que reúne características e ferramentas de apoio ao desenvolvimento de software. Tem como objetivo agilizar e facilitar o processo de desenvolvimento.

3 Marcar uma função com *extern "C"* serve para indicar ao compilador de *C++* que essa função é para ser compilada em *C*; caso dentro dela exista *C++* em algum ponto, retorna erro de compilação

utilizador não ser homogéneo. No caso de uma implementação para um ambiente *multicore*, geralmente, as bibliotecas suportam escalonamento dinâmico; por conseguinte, é assegurado o facto de alguns *cores* não terminarem cedo demais o processamento. O escalonamento dinâmico em *CUDA* não é suportado pelas funções *kernel* [64], mas, sabendo que criar *threads* em *CUDA* é um procedimento barato, apesar de não ser a ideal, uma solução é executar muito mais *threads* em relação ao número de *cores* de cada um dos *multiprocessadores*.

Seguidamente apresenta-se a versão *GPU* do *CCD++* no **Algoritmo 9**.

Algoritmo 9: Versão *GPU* do algoritmo *CCD++*

input : A, W, H, λ, k, T

1 **begin**
2   `inicializar(`$W \leftarrow 0, R \leftarrow A$`);`
3   Aloca memória no *device* para as matrizes A e R e para os vetores u e v;
4   Copia as matrizes A e R do *host* para o *device*;
5   **for** $iter \leftarrow 1$ **to** T $Step = 1$ **do**
6   **for** $t \leftarrow 1$ **to** k $Step = 1$ **do**
7       $u \leftarrow \bar{\omega}_t$ and $v \leftarrow \bar{h}_t$;
8   Copia os vetores u e v do *host* para o *device*;
9   Chama o *kernel* para atualizar $\hat{R}$ no *device*, utilizando (4.1);
10   **for** $inneriter \leftarrow 1$ **to** T $Step = 1$ **do**
11   Chama o *kernel* para atualizar u e v no *device*, utilizando (4.2) e (4.3);
12   Copia os vetores u e v do *device* para o *host*;
13       $\bar{\omega}_t \leftarrow u$ e $\bar{h}_t \leftarrow v$;
14   Chama o *kernel* para atualizar $\hat{R}$ no *device*, utilizando (4.5);

5.2 Implementação do algoritmo *ALS*

Como não foi encontrada nenhuma implementação em *C/C++* do algoritmo *ALS*, foi necessário elaborar uma de raiz com base na informação disponibilizada nos artigos [66, 75, 76, 78].

5.2.1 Ponto de partida para a implementação e tratamento dos dados esparsos

Como ponto de partida para carregar e ordenar a representação esparsa da matriz A, utiliza-se a base da biblioteca *LIBPMF*. Como o algoritmo *CCD++* apenas necessita de percorrer a matriz A, coluna a coluna, na representação esparsa os dados são carregados e ordenados por coluna. No entanto, no *ALS*, também, é necessário que os dados sejam ordenados por linha. Logo, a solução foi criar um vetor de índices que, quando acedido contiguamente, mapeia as linhas de A, diretamente, nos dados ordenados por coluna. Seguidamente, serão explicados, em detalhe, os pormenores.

Assumindo a seguinte matriz $A^{6\times5}$:

$A =$

utilizador \ item	0	1	2	3	4
0	4	2	0	1	0
1	2	0	0	0	3
2	1	3	0	0	2
3	0	1	0	4	0
4	0	0	3	4	0
5	0	0	0	0	1

A representação esparsa de A ordenada por colunas em que o vetor *Utilizador* mapeia o índice de utilizador, o vetor *Item* mapeia o índice do item, e o vetor *Valor* contém a classificação respetiva. É dada por:

$A_{Sparse} =$ (5.1)

	Utilizador	*Item*	*Valor*
0	**0**	**0**	**4**
1	**0**	**1**	**2**
2	**0**	**3**	**1**
3	*1*	*0*	*2*
4	*1*	*4*	*3*
5	**2**	**0**	**1**
6	**2**	**1**	**3**
7	**2**	**4**	**2**
8	*3*	*1*	*1*
9	*3*	*3*	*4*
10	**4**	**2**	**3**
11	**4**	**3**	**4**
12	*5*	*4*	*1*

No caso do ficheiro de entrada para a *LIBPMF* o formato é o mesmo de A_{Sparse}, apenas os índices começam em 1. O facto de estarem ordenados por linhas ou colunas não é relevante.

Neste caso, A_{Sparse} está ordenada por linhas. Na Tabela (5.1), as linhas horizontais, os itálicos e os negritos servem para destacar a mudança de linha/utilizador na matriz A.

Primeiramente, na representação esparsa, é necessário saber qual o índice onde começa cada linha e cada coluna. Percorrendo a matriz A por linhas, a primeira linha começa no índice 0 e tem 3 valores, a segunda começa no índice $0 + 3 = 3$ e tem 2 valores, a terceira no índice $3 + 2$ e tem 3 valores, e assim sucessivamente tal como na Tabela (5.2). Para as colunas, o raciocínio é o mesmo, com a diferença de que a matriz A é percorrida por colunas tal como na Tabela (5.3).

	Vetor dos índices relativos ao início de cada uma das linhas
0	0
1	3
2	5
3	8
4	10
5	12
6	13

(5.2)

	Vetor dos índices relativos ao início de cada uma das colunas
0	0
1	3
2	6
3	7
4	10
5	13

(5.3)

A biblioteca *LIBPMF* ordena os utilizadores por colunas, os itens por linhas e os valores por colunas. A representação esparsa resultante fica:

	Utilizador	*Item*	*Valor*
0	**0**	**0**	**4**
1	**1**	**1**	**2**
2	**2**	**3**	**1**
3	*0*	*0*	*2*
4	*2*	*4*	*3*
5	*3*	**0**	*1*
6	**4**	**1**	**3**
7	*0*	**4**	*1*
8	*3*	*1*	*4*
9	*4*	*3*	*4*
10	**1**	**2**	**3**
11	**2**	**3**	**2**
12	**5**	*4*	**1**

(5.4)

Na Tabela (5.4) as linhas horizontais, os negritos e os itálicos servem para assinalar o seguinte relativamente à matriz A: no vetor *Utilizador*, assinalam a mudança de coluna/item; no vetor *Item*, assinalam a mudança de linha/utilizador; no vetor *Valor*, assinalam a mudança de coluna/item.

Para atualizar H, apenas é necessário aceder aos vetores *Utilizador* e *Valor* que já estão ordenados por colunas; não é necessário acrescentar nem alterar nada. Mas, para atualizar W, é necessário aceder aos vetores *Item* e *Valor*. Apesar do vetor *Item* estar devidamente ordenado por linhas, o vetor *Valor* está ordenado por colunas. Logo, é necessário acrescentar um vetor *Map* que faça o mapeamento dos índices, de forma a que no vetor *Valor* se possa aceder aos mesmos, estando estes devidamente ordenados por linhas (este vetor foi criado para o algoritmo *ALS*):

	Map		$Utilizador$	$Item$	$Valor$
0	**0**	–	**0**	**0**	**4**
1	**3**	–	**1**	**1**	**2**
2	**7**	–	**2**	**3**	**1**
3	*1*	–	*0*	*0*	*2*
4	*10*	–	*2*	*4*	*3*
5	**2**	–	*3*	**0**	*1*
6	**4**	–	**4**	**1**	**3**
7	**11**	–	*0*	**4**	*1*
8	*5*	–	*3*	*1*	*4*
9	*8*	–	*4*	*3*	*4*
10	**6**	–	**1**	**2**	**3**
11	**9**	–	**2**	**3**	**2**
12	*12*	–	**5**	*4*	**1**

Por exemplo, para identificar o valor de $Item_7$, faz-se $Valor_{Map_7}$: sendo $Map_7 = 11$, fica $Valor_{11} = 2$. Por isso, o valor de $Item_7$ é 2.

Para mapear $Item_7$ diretamente na matriz A, basta fazer $Utilizador_{Map_7}$. Sendo $Map_7 = 11$, fica $Utilizador_{11} = 2$. Uma vez que $Item_7 = 4$, logo, na coluna 4 e linha 2 da matriz A encontra-se um 2.

Nesta implementação, o vetor *Map* é calculado apenas percorrendo uma única vez o vetor *Utilizador*.

Com os problemas inerentes à representação esparsa resolvidos da forma citada, passa-se à implementação da equação $\omega_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T a_i$. Os passos para implementar o *ALS* estão citados na Subsecção 2.4.2. No entanto, é importante citar as técnicas utilizadas para otimizar a velocidade dos cálculos, a saber:

- Pelo facto de $H_{\Omega_i}^T H_{\Omega_i}$ resultar sempre numa matriz simétrica ($R = R^T$) definida positivamente:
  - Para calcular $H_{\Omega_i}^T H_{\Omega_i}$, sendo $H_{\Omega_i} = M$, para M_{ij}, a posição transposta correspondente é M_{ji}. Mas, como a matriz resultante R é simétrica, percorrendo os ij na parte triangular de cima, incluindo a diagonal, basta calcular R_{ij} e copiar o valor para a parte triangular de baixo R_{ji}.
  - Sendo $(H_{\Omega_i}^T H_{\Omega_i} + \lambda I) = SubMatrix$, para calcular a inversa de $SubMatrix$ $(SubMatrix)^{-1}$, utiliza-se a decomposição de *André Cholesky* por ser um dos métodos mais eficientes que permite reduzir o número de operações [36]. No entanto, inverter uma matriz continua a ter uma complexidade cúbica. Como $SubMatrix$ tem dimensões $k \times k$, logo aumentar k neste algoritmo tem complexidade cúbica.
- Para calcular $H^T a_i$, ignoram-se as multiplicações por zero; com a representação esparsa de a_i é simples ignorar os zeros.

5.2.2 Implementação *multicore*

Na versão *multicore*, utiliza-se *OpenMP* e o método de paralelização é o citado na Secção 4.1.

5.2.3 Implementação *CUDA*

Na versão *CUDA*, os dados são copiados uma única vez para a placa gráfica e só se utiliza o *host* para sincronizar. Como no *host* W e H são representadas de maneira bidimensional, estas matrizes são transformadas para uma representação a uma dimensão e, assim, transferidas para o *device* onde são tratadas a uma dimensão. No final, as matrizes W e H são copiadas do *device* para o *host*, e são novamente transformadas para voltar a ter representação bidimensional. Estas transformações só são feitas uma vez no início e outra no fim do processamento; por isso, o tempo gasto por elas é insignificante.

Seguidamente apresenta-se, no **Algoritmo 10**, a versão *GPU* do *ALS*.

Algoritmo 10: Versão *GPU* do algoritmo *ALS*

---

input : A, W, H, λ, T

1 Aloca memória no *device* para as matrizes A, W e H;

2 Copia as matrizes A, W e H do *host* para o *device*;

3 **begin**

4   `inicializar(`$H \leftarrow$ *(números aleatórios pequenos)*`)`;

5   **for** *iter* $\leftarrow 1$ **to** T *Step* $= 1$ **do**

6   Chama o *kernel* para atualizar W que minimiza (2.1) mantendo H fixa, utilizando $\omega_i^* = (H_{\Omega_i}^T H_{\Omega_i} + \lambda I)^{-1} H^T a_i$; `// Sincroniza no host`;

7   Chama o *kernel* para atualizar H que minimiza (2.1) mantendo W fixa, utilizando $h_j^* = (W_{\Omega_j}^T W_{\Omega_j} + \lambda I)^{-1} W^T a_j$; `// Sincroniza no host`;

8   Copia as matrizes W e H do *device* para o *host*;

5.3 Utilização

Primeiro, é necessário preparar a máquina para funcionar em *CUDA*. Para compilar o código e executar em *Linux*, consultar o Anexo A.1. Para compilar o código e executar em *Windows*, consultar o Anexo A.2.

A utilização da versão da biblioteca *LIBPMF cuda enabled* é semelhante à utilização definida originalmente; apenas acrescem alguns parâmetros ao argumento.

Na linha de comandos, executa-se *cuda-or-omp-pmf-train [options] data_dir [model_filename]*,

onde *[options]* é definido pelas seguinte opções:

'-k' valor: Define o valor de k nas matrizes do modelo; por defeito é 10.
'-n' threads: Define o número de *threads* utilizadas na versão *OpenMP*; por defeito é 4.
'-l' lambda: Define o parâmetro de regularização λ; por defeito é 0.1.
'-t' max_iter: Define o número de iterações *CCD*; por defeito é 5.
Opções acrescentadas:

'-ALS': Quando presente, utiliza o algoritmo *ALS*. Caso não esteja presente, utiliza o algoritmo *CCD++*.
'-Cuda': Quando presente, ativa a versão *CUDA enabled*. Caso não esteja presente, é executada a versão *OpenMP* em *float precision*. Não pode ser utilizado em conjunto com *'-runOriginal'*.
'-runOriginal': Quando presente, executa a versão original *OpenMP* em *double precision*. Não pode ser utilizado em conjunto com *'-Cuda'*.
'-nBlocks' valor: Define o número de *blocos* a utilizar em *CUDA*; por defeito é 16. Este parâmetro está associado a cada um dos *Streaming Multiprocessors*.
'-nThreadsPerBlock' valor: Define o número de *threads* por *bloco* a utilizar em *CUDA*; por defeito é 32. Este parâmetro está associado a cada uma das *threads* de cada um dos *Streaming Multiprocessors*.

O argumento *'data_dir'* serve para indicar o nome do diretório, com os ficheiros referentes aos dados de treino e teste.

O Formato dos dados em *'data data_dir'* é o seguinte: o diretório *'data_dir'* tem de conter três ficheiros, um chamado *'meta'*, outro para os dados de treino e outro para os dados de teste. O ficheiro *'meta'* contém as seguintes linhas:

1ª: *m n*
2ª: *num_training_ratings training_file_name*
3ª: *num_test_ratings test_file_name*

'm' corresponde ao número de utilizadores, *'n'* ao número de itens, *'num_training_ratings'* ao número de observações existentes no ficheiro de treino, *'training_file_name'* ao nome do

ficheiro de treino, *'num_test_ratings'* ao número de observações existentes no ficheiro de teste e *'test_file_name'* ao nome do ficheiro de teste. Para mais, pormenores ver o exemplo *'toy-example'* fornecido com a biblioteca *LIBPMF*.

5.4 Sumário

Neste Capítulo, começou-se por falar dos *GPUs* no contexto de sistemas de recomendação e da motivação para utilização em processamento paralelo. De seguida, foram apresentados os métodos de paralelização em *GPU* dos algoritmos *ALS* e *CCD++*. Por fim, foi explicada a forma como se utiliza o programa anexo a esta dissertação.

O Capítulo seguinte será iniciado com a discussão de que existem muitos modelos de placas gráficas e o seu desempenho varia muito de modelo para modelo. Ainda no âmbito deste mesmo Capítulo, será apresentada a metodologia experimental relativamente à escolha do *hardware* e dos conjuntos de dados para teste.

Capítulo

6

Metodologia experimental

Em testes de desempenho algorítmico que envolvem tempos de execução, o tipo de *hardware* é um fator que influencia os resultados obtidos. No caso da tecnologia *CUDA*, a capacidade de processamento das placas gráficas varia muito entre modelos. Além disso, a evolução das placas gráficas tem vindo a ser frenética. Por isso, para testar e utilizar programas em *CUDA*, o tipo de placa gráfica é um fator decisivo.

Devido à influência que o *hardware* assume na execução de experiências que envolvem tempo de execução de algoritmos, neste Capítulo, é apresentada uma descrição pormenorizada do *hardware* que é utilizado na análise experimental, facto pelo qual cada uma das máquinas utilizadas será descrita numa Secção.

Os conjuntos de dados afetam, também, o comportamento dos algoritmos. Aspetos como o tamanho, a distribuição dos dados por linhas e colunas bem como a relação entre o número de linhas e colunas são fatores importantes. Por este motivo, cada um dos conjuntos de dados é descrito numa Secção.

As métricas de avaliação utilizadas são o $speedup = \frac{sequentialTime}{parallelTime}$ (consultar Secção 3.3) e o *RMSE* (consultar Fórmula 2.26). A metodologia utilizada para estimar o *RMSE* é a *holdout* (consultar Secção 2.6).

6.1 Máquina *cracs-gpu*

Esta máquina é um servidor da Faculdade de Ciências da Universidade do Porto, com suporte *CUDA*.
Detalhes do sistema operativo:

- *Linux Fedora release 20 (Heisenbug).* Versão do *kernel*: *Linux version 3.13.9-200.fc20.x86_64 (mockbuild@bkernel01.phx2.fedoraproject.org) (gcc version 4.8.2 20131212 (Red Hat 4.8.2-7) (GCC)) #1 SMP Fri Apr 4 12:13:05 UTC 2014*

Detalhes do *hardware*:

- **Placa gráfica:** *Tesla C2050*
  - *Total Dedicated Memory 3GB GDDR5*
  - *# of CUDA Core 448*
  - *Single Precision floating point performance 1.03 Tflops*
  - *GPU Clock 1015 MHz*
  - *Architecture Fermi*
- **Processadores:** 2 × *Intel® Xeon® E5620*
  - *12M Cache*
  - *2.40 GHz*
  - *# of Cores 4*
  - *# of Threads 8*
  - *Max TDP*[1] *80W*
  - *Microarchitecture Nehalem*
  - *Socket Supported FCLGA1366*
- **Memória *RAM*:** *12GB*

[1] *TDP - Thermal Design Power*, representa a potência máxima que o *CPU* consegue atingir sem sobreaquecer durante períodos longos de utilização.

6.2 Máquina *hyperthreading*

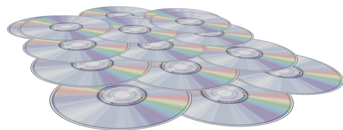
Detalhes do sistema operativo:

- *Linux Fedora release 20 (Heisenbug).* Versão do *kernel*: *Linux version 3.13.9-200.fc20.x86_64 (mockbuild@bkernel01.phx2.fedoraproject.org) (gcc version 4.8.2 20131212 (Red Hat 4.8.2-7) (GCC)) #1 SMP Fri Apr 4 12:13:05 UTC 2014*
- *Microsoft Windws 8.1 pro*

Detalhes do *hardware*:

- **Placa gráfica:** *Gainward GeForce GTX 580 Phantom*, preço = $600
  - *Total Dedicated Memory 3GB GDDR5*
  - *# of CUDA Core 512*
  - *Single Precision floating point performance 1.58 Tflops*
  - *GPU Clock 1544 MHz*
  - *Architecture Fermi*
- **Processadores:** 2 × *Intel® Xeon® X5550*, preço = 2 × $999 = $1998
  - *8M Cache*
  - *2.66 GHz*
  - *# of Cores 4*
  - *# of Threads 8*
  - *Max TDP*[1] *95W*
  - *Microarchitecture Nehalem*
  - *Socket Supported FCLGA1366*
- **Memória *RAM*:** *24GB 6 × 4GB HYNIX HMT151R7BFR4C-H9*
  - *DDR3 PC3-10600 1333MHz*
  - *ECC Registered DIMM Memory*
- ***Motherboard*:** *Tyan S7020WAGM2NR*
- **Disco rígido para o *Windows*:** *Samsung SSD 840 PRO 256GB MZ-7PD256BW*
  - Velocidade de leitura sequencial *540 MB/s*
  - Velocidade de escrita sequencial *520 MB/s*
- **Disco rígido para o *Linux*:** *Samsung SSD 840 PRO 128GB MZ-7PD128BW*
  - Velocidade de leitura sequencial *530 MB/s*
  - Velocidade de escrita sequencial *390 MB/s*

6.3 Conjunto de dados *Netflix*



A *Netflix* é uma empresa americana de aluguer *online* de filmes, séries e programas de televisão. O conjunto de dados *Netflix* foi publicado para a famosa competição *Netflix prize* que teve início em 2 de Outubro de 2006 e terminou a 21 de Setembro de 2009. O concurso consistia em atribuir um prémio de 1 milhão de dólares ao grupo que melhor resultado conseguisse, e a medir pelo *RMSE*, fosse, no mínimo, melhor 10% em relação ao sistema $Cinematch^{SM}$ existente na empresa. O prémio foi atribuído ao grupo *BellKor's Pragmatic Chaos* que conseguiu um melhoramento de 10.06% no *RMSE*.

- **Número de utilizadores:** Nos dados, os números de utilizador não são contíguos, variam entre 1 e 2.649.429. O número de utilizadores é 480.189.
- **Número de itens (filmes):** 17.770
- **Variação dos valores de classificação:** Variam entre 0 e 5, provenientes de uma escala de cinco estrelas.
- **Número de entradas (classificações) em representação esparsa:** 100.580.527 aproximadamente 100 milhões.
- **Dimensões da matriz não esparsa:** $A^{2.649.429 \times 17.770}$.

Para medir o resultado das previsões, é necessário dividir os dados num conjunto de treino e noutro de teste. Para isso, juntamente com os dados da *Netflix* está o ficheiro *probe.txt*, contendo uma sugestão de divisão dos dados. Por conseguinte, utilizando o *probe*, fica-se com

- **Número de entradas para treino (classificações), em representação esparsa:** 100.480.507.
- **Número de entradas para teste (classificações), em representação esparsa:** 100.020.

6.4 Conjunto de dados *MovieLens 10M*

Este conjunto de dados provém do site *MovieLens* que é especializado em recomendações de filmes. Quem utiliza o *MovieLens* recebe recomendações de filmes e contribui para o funcionamento do site com as suas classificações. Todos os utilizadores selecionados para este conjunto de dados votaram em pelo menos vinte filmes.

- **Número de utilizadores:** Nos dados, os números de utilizador não são contíguos, variam entre 1 e 71.567. O número de utilizadores é 69.878.
- **Número de itens (filmes):** 65.133.
- **Variação dos valores de classificação:** Variam entre 0 e 5, provenientes de uma escala de cinco estrelas.
- **Número de entradas (classificações) em representação esparsa:** 10.069.932 aproximadamente 10 milhões.
- **Dimensões da matriz não esparsa:** $A^{71.567 \times 65.133}$.

Para medir o resultado das previsões, é necessário dividir os dados num conjunto de treino e noutro de teste. O conjunto de dados de teste consiste numa amostragem estratificada por utilizador:

- **Número de entradas para <u>treino</u> (classificações), em representação esparsa:** 10.000.054.
- **Número de entradas para <u>teste</u> (classificações), em representação esparsa:** 69.878.

6.5 Sumário

Neste Capítulo, foram demonstradas as metodologias usadas na avaliação do desempenho dos algoritmos. Foi, igualmente, descrito o *hardware* utilizado bem como os conjuntos de dados usados na avaliação dos algoritmos.

No próximo Capítulo, serão apresentados os resultados obtidos, utilizando a metodologia descrita neste Capítulo.

Capítulo 7

Resultados

Utilizando a metodologia experimental descrita no Capítulo 6, são apresentados e comparados aqui os resultados obtidos em ambientes *multicore* e *CUDA*. Como este trabalho incide nas implementações em *CUDA*, é feita uma análise do comportamento de cada um dos algoritmos executados na placa gráfica.

Independentemente do conjunto de dados, salvo indicação contrária, em todos os testes, os parâmetros utilizados nos algoritmos *CCD++* e *ALS* são $k = 5$, $\lambda = 0.1$ e $T = 15$. Foi escolhido o valor $k = 5$, porque, como será mostrado adiante para o conjunto de dados *Netflix* (Secção 6.3), um k maior não melhora o *RMSE*. O λ não afeta a velocidade de processamento, portanto, não se opta por uma análise exaustiva do mesmo, preferindo utilizar-se o valor por defeito da *LIBPMF*. A escolha do valor elevado de T garante que o algoritmo convirja sempre. Em cada um dos testes apresentados, o tempo em segundos é obtido a partir da média de 10 execuções. Voltando à escolha de k, o facto de ser elevado não se traduz, obrigatoriamente, em valores de *RMSE* mais baixos; a escolha correta depende da análise das dimensões existentes nos dados. O valor ideal de k, geralmente, é igual ao número de dimensões dos dados. Discutir-se-á também a escalabilidade e valores de *RMSE* entre cada um dos algoritmos implementados.

Para os testes em *CUDA*, a definição do valor do número de *blocos* está relacionada com o número de *multiprocessadores* disponibilizados pela placa.

7.1 Resultados utilizando o conjunto de dados *Netflix* (Secção 6.3)

Para o conjunto de teste *probe*, não foram encontrados valores de referência para o *RMSE*. O que se sabe é que na competição organizada pela empresa *Netflix* (Secção 6.3), no conjunto de dados privado, o valor do *RMSE* que ganhou a competição foi 0.8567.

7.1.1 Testes em precisão *double* na versão original do *CCD++* da *LIBPMF* em *OpenMP*

Em todos os testes referidos na Tabela 7.1, nas previsões do conjunto de dados *probe*, o valor do *RMSE* é sempre 0.94.

Tabela 7.1: *Speedups* do *CCD++* da *LIBPMF* em *OpenMP*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 521,512s$	
2 *threads*	$\approx 316,701s$	$1,6$
8 *threads*	$\approx 136,2s$	$3,8$
16 *threads*	$\approx 126,81s$	$4,1$
32 *threads*	$\approx 136,023s$	$3,8$

Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 717,307s$	
2 *threads*	$\approx 407,873s$	$1,8$
8 *threads*	$\approx 179,499s$	$4,0$
16 *threads*	$\approx 166,746s$	$4,3$
32 *threads*	$\approx 161,48s$	$4,4$

Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 596,245$	
2 *threads*	$\approx 307,624s$	$1,9$
8 *threads*	$\approx 117,005s$	$5,0$
16 *threads*	$\approx 110,476s$	$5,3$
32 *threads*	$\approx 109,526s$	$5,4$

O *Windows* na linguagem *C/C++* é mais lento que o *Linux*.

Na máquina *cracs-gpu* (Secção 6.1), apesar da velocidade dos processadores ser ligeiramente menor, em virtude de terem uma *cache* maior, no geral, em tempos de processamento, os resultados são, ligeiramente, inferiores.

Contudo, na versão do *CCD++*, implementada para esta dissertação, utiliza-se a precisão decimal *single/float*. Os resultados são os mesmos relativamente ao *RMSE*, mas, especialmente em *Windows*, o desempenho é muito beneficiado por este tipo de dados. Utilizando *float* em vez de *double*, é atingido um *speedup* de 9.5 (com 32 *threads*) o que é mais do dobro relativamente à versão em precisão *double*.

7.1.2 Testes em precisão *float* na versão modificada do *CCD++* em *OpenMP* e em *CUDA*

Em todos os testes, nas previsões do conjunto de dados *probe*, o valor do *RMSE* é 0.94.

Tabela 7.2: *Speedups* na versão modificada do *CCD++* em *OpenMP* e em *CUDA*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 528,538s$	
2 *threads*	$\approx 309,707s$	1,7
8 *threads*	$\approx 111,968s$	4,7
16 *threads*	$\approx 98,1266s$	5,3
32 *threads*	$\approx 99,8027s$	5,2
CUDA* com 16 *blocos* de 512 *threads	**≈ 168,109s**	**3,1**
Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 1252,35s$	
2 *threads*	$\approx 540,973s$	2,3
8 *threads*	$\approx 181,501s$	6,9
16 *threads*	$\approx 131,881s$	9,5
32 *threads*	$\approx 131,661s$	9,5
CUDA* com 16 *blocos* de 512 *threads	**≈ 84,7718s**	**14,8**
Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 480,628s$	
2 *threads*	$\approx 247,219s$	1,9
8 *threads*	$\approx 85,8724s$	5,5
16 *threads*	$\approx 76,8607s$	6,3
32 *threads*	$\approx 74,9617s$	6,4
CUDA* com 14 *blocos* de 512 *threads	**≈ 350,21s**	**1,3**

O melhor *speedup* (14.8) é obtido na versão *CUDA* em *Windows*, porquanto, mostra que, relativamente ao preço, uma *multicore*, que custa mais do dobro do preço, consegue ser ultrapassada pela placa gráfica. Sendo assim, esta tecnologia tem grande potencialidade para a implementação de sistemas de recomendação baseados em fatorização de matrizes.

O *Windows* em precisão *float* na linguagem *C/C++* é mais lento que em *double* enquanto o *Linux* é, precisamente, o contrário. Na precisão *float*, o *Linux* é consideravelmente mais rápido que o *Windows*. Mas em *CUDA*, o *Windows* é duplamente mais rápido que o *Linux*, encontrando-se uma possível explicação na maior sofisticação do *driver Windows*.

Na máquina *cracs-gpu* (Secção 6.1), apesar da velocidade dos processadores ser ligeiramente menor, em virtude de terem uma *cache* maior, em tempos de processamento, os resultados são, ligeiramente, inferiores. A placa gráfica da máquina *cracs-gpu* (Secção 6.1), pelo facto da velocidade dos *cores* ser inferior e ter menos *cores*, é mais lenta, em relação à da máquina *hyperthreading* (Secção 6.2).

7.1.3 Testes em precisão *float* no *ALS* em *OpenMP* e em *CUDA*

Em todos os testes, representados na Tabela 7.3, nas previsões do conjunto de dados *probe*, o valor do *RMSE* é sempre $0,97$.

Tabela 7.3: *Speedups* no *ALS* em *OpenMP* e em *CUDA*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 429,539s$	
2 *threads*	$\approx 224,99s$	$1,9$
8 *threads*	$\approx 93,8716s$	$4,6$
16 *threads*	$\approx 98,3057s$	$4,3$
32 *threads*	$\approx 95,8294s$	$4,5$
CUDA* com 16 *blocos* de 512 *threads	**≈ 98,71s**	**4,4**
Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 665,74s$	
2 *threads*	$\approx 355,144s$	$1,9$
8 *threads*	$\approx 158,912s$	$4,2$
16 *threads*	$\approx 121,667s$	$5,5$
32 *threads*	$\approx 122,121s$	$5,5$
CUDA* com 16 *blocos* de 512 *threads	**≈ 107,214s**	**6,2**
Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 386,614s$	
2 *threads*	$\approx 196,25s$	$2,0$
8 *threads*	$\approx 76,6014s$	$5,0$
16 *threads*	$\approx 80,0473s$	$4,8$
32 *threads*	$\approx 80,5598s$	$4,8$
CUDA* com 14 *blocos* de 512 *threads	**≈ 149,663s**	**2,6**

A versão *CUDA* em *Windows* é mais lenta que em *Linux*. A razão está relacionada com a granularidade grossa dos *kernels* do *ALS*. Quando o conjunto de dados, do tamanho do *Netflix* (Secção 6.3), é executado numa placa não dedicada e dependendo da potência da mesma, pode gerar instabilidade no sistema (o *driver* necessita que o ecrã seja refrescado). Este facto explica o aumento do tempo. Sabendo que em relação ao *Linux*, o *Windows*, para manter o ambiente de trabalho, consome mais recursos gráficos, para atingir tempos de execução mais precisos, é necessário uma segunda placa gráfica dedicada apenas para *CUDA*.

7.2 Resultados utilizando o conjunto de dados *MovieLens 10M* (Secção 6.4)

O conjunto de dados *MovieLens 10M* (Secção 6.4) é demasiado pequeno para atingir um *speedup* significativo. Nos testes *multicore*, em geral, a partir das oito *threads*, o *speedup* começa a decair.

7.2.1 Testes em precisão *double* na versão original do *CCD++* da *LIBPMF* em *OpenMP*

Em todos os testes, representados na Tabela 7.4, nas previsões do conjunto de dados de teste, o valor do *RMSE* é sempre $2,572$.

Tabela 7.4: *Speedups* do *CCD++* da *LIBPMF* em *OpenMP*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 23,4608s$	
2 *threads*	$\approx 13,492s$	$1,7$
8 *threads*	$\approx 7,71895s$	$3,0$
16 *threads*	$\approx 8,58666s$	$2,7$
32 *threads*	$\approx 8,84259s$	$2,6$
Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 33,1817s$	
2 *threads*	$\approx 17,8441s$	$1,9$
8 *threads*	$\approx 9,27782s$	$3,6$
16 *threads*	$\approx 9,73639s$	$3,4$
32 *threads*	$\approx 9,7224s$	$3,4$
Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 29,5452$	
2 *threads*	$\approx 14,5926s$	$2,0$
8 *threads*	$\approx 7,38381s$	$4,0$
16 *threads*	$\approx 9,13583s$	$3,2$
32 *threads*	$\approx 8,91894s$	$3,3$

7.2.2 Testes em precisão *float* na versão modificada do *CCD++* em *OpenMP* e em *CUDA*

Em todos os testes, representados na Tabela 7.5, nas previsões do conjunto de dados de teste obtido com o método mencionado na Secção 6.4, o valor do *RMSE* é sempre $2,570$.

Tabela 7.5: *Speedups* na versão modificada do *CCD++* em *OpenMP* e em *CUDA*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 25,2886s$	
2 *threads*	$\approx 13,3587s$	$1,9$
8 *threads*	$\approx 5,69531s$	$4,4$
16 *threads*	$\approx 6,86471s$	$3,6$
32 *threads*	$\approx 7,24346s$	$3,4$
CUDA* com 16 *blocos* de 512 *threads	**≈ 25,0212s**	**1,0**

Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 61,5171s$	
2 *threads*	$\approx 29,5249s$	$2,1$
8 *threads*	$\approx 10,8432s$	$5,7$
16 *threads*	$\approx 10,0441s$	$6,1$
32 *threads*	$\approx 9,93356s$	$6,2$
CUDA* com 16 *blocos* de 512 *threads	**≈ 11,3952s**	**5,4**

Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 29,6824s$	
2 *threads*	$\approx 15,126s$	$2,0$
8 *threads*	$\approx 6,10678s$	$4,9$
16 *threads*	$\approx 7,06194s$	$4,2$
32 *threads*	$\approx 7,08091s$	$4,2$
CUDA* com 14 *blocos* de 512 *threads	**≈ 45,4349s**	**0,7**

7.2.3 Testes em precisão *float* no *ALS* em *OpenMP* e em *CUDA*

Em todos os testes, representados na Tabela 7.6, nas previsões do conjunto de dados de teste obtido com o método mencionado na Secção 6.4, o valor do *RMSE* é sempre $2,570$.

Tabela 7.6: *Speedups* no *ALS* em *OpenMP* e em *CUDA*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 20,8461s$	
2 *threads*	$\approx 11,1398s$	$1,9$
8 *threads*	$\approx 3,8365s$	$5,4$
16 *threads*	$\approx 4,35807s$	$4,3$
32 *threads*	$\approx 4,13933s$	$5,0$
CUDA* com 16 *blocos* de 512 *threads	**≈ 14,725s**	**1,4**

Máquina *hyperthreading* (Secção 6.2), SO: *Windows*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 34,6181s$	
2 *threads*	$\approx 17,461s$	$2,0$
8 *threads*	$\approx 9,64755s$	$3,6$
16 *threads*	$\approx 6,82955$	$5,1$
32 *threads*	$\approx 7,30159s$	$4,7$
CUDA* com 16 *blocos* de 512 *threads	**≈ 13,8062s**	**2,5**

Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*		
Tipo de teste	Tempo de execução	*Speedup*
1 *thread*	$\approx 20,3254s$	
2 *threads*	$\approx 11,4484s$	$1,8$
8 *threads*	$\approx 4,4823s$	$4,5$
16 *threads*	$\approx 4,33758s$	$4,7$
32 *threads*	$\approx 4,54055s$	$4,5$
CUDA* com 14 *blocos* de 512 *threads	**≈ 21,9959s**	**0,9**

7.3 Análise do comportamento relativamente à parametrização dos algoritmos implementados

Na versão *CUDA* do algoritmo *CCD++*, o *speedup* diminui à medida que o parâmetro k aumenta. Na versão em *OpenMP*, o *speedup* apesar de diminuir ligeiramente, é bastante estável tal como ilustrado na Figura 7.1.

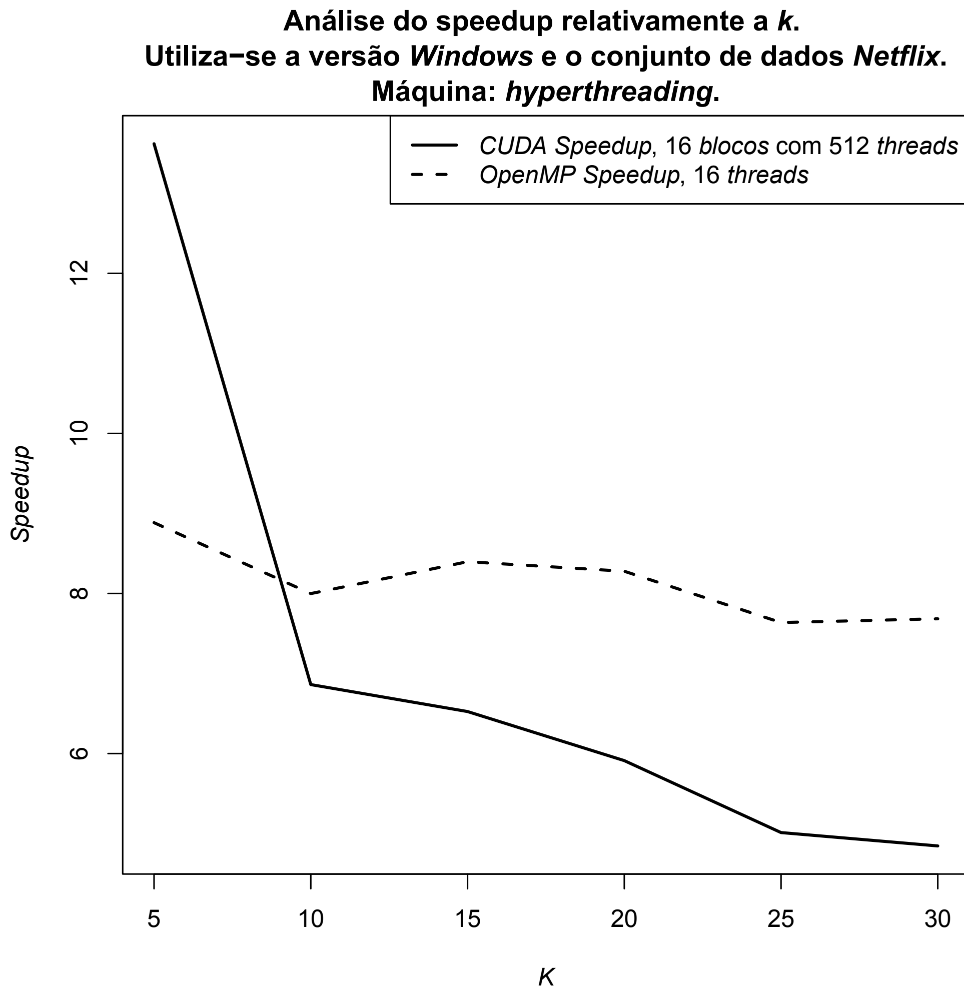


Figura 7.1: Comportamento do algoritmo *CCD++* relativamente a k. A parametrização é $\lambda = 0.1$, $T = 5$.

Para realizar os testes da Figura 7.1, a escolha do sistema operativo e da máquina justifica-se pelo conjunto de dados que obteve melhores resultados em *CUDA*. Para máquinas diferentes, o comportamento relativamente ao aumento de k é igual.

O facto da versão *CUDA* do algoritmo *CCD++* ser sensível ao aumento de k, prende-se com o número de cópias *host to device*, *device to host*. Quanto maior for k, relativamente ao número total de iterações, o número de cópias de dados aumenta.

Alterar k no algoritmo *ALS*, tanto em *CUDA* como em *OpenMP*, não interfere no *speedup*.

A Figura 7.2 mostra que o algoritmo *CCD++*, relativamente ao algoritmo *ALS*, é mais rápido para valores altos de k.

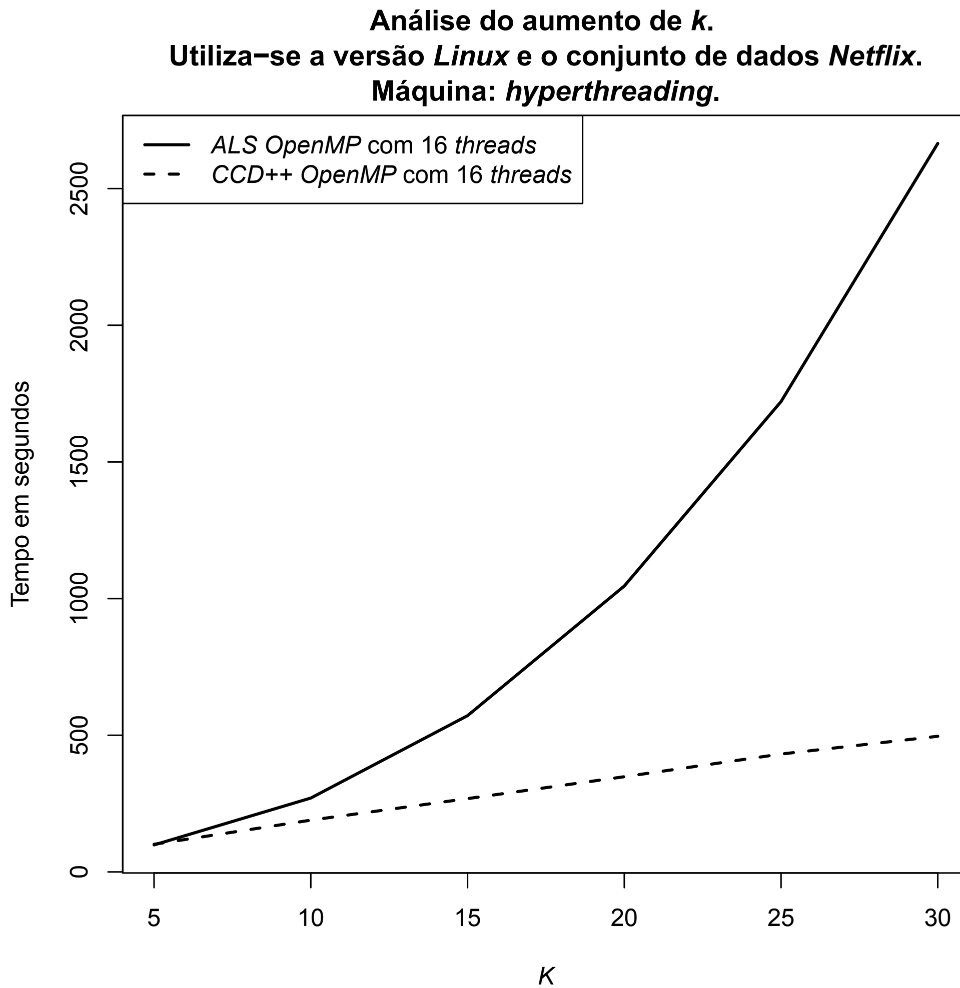


Figura 7.2: Análise comparativa do tempo de processamento entre os algoritmos *ALS* e *CCD++*, relativamente ao aumento de k. A parametrização é $\lambda = 0.1$, $T = 15$.

Para realizar os testes da Figura 7.2, a escolha do sistema operativo e da máquina deve-se ao conjunto que obteve melhores resultados em *OpenMP*. Para máquinas diferentes, o comportamento, relativamente ao aumento de k, é igual.

A Figura 7.3 mostra os valores de *RMSE*, relativamente aos testes realizados na Figura 7.2.

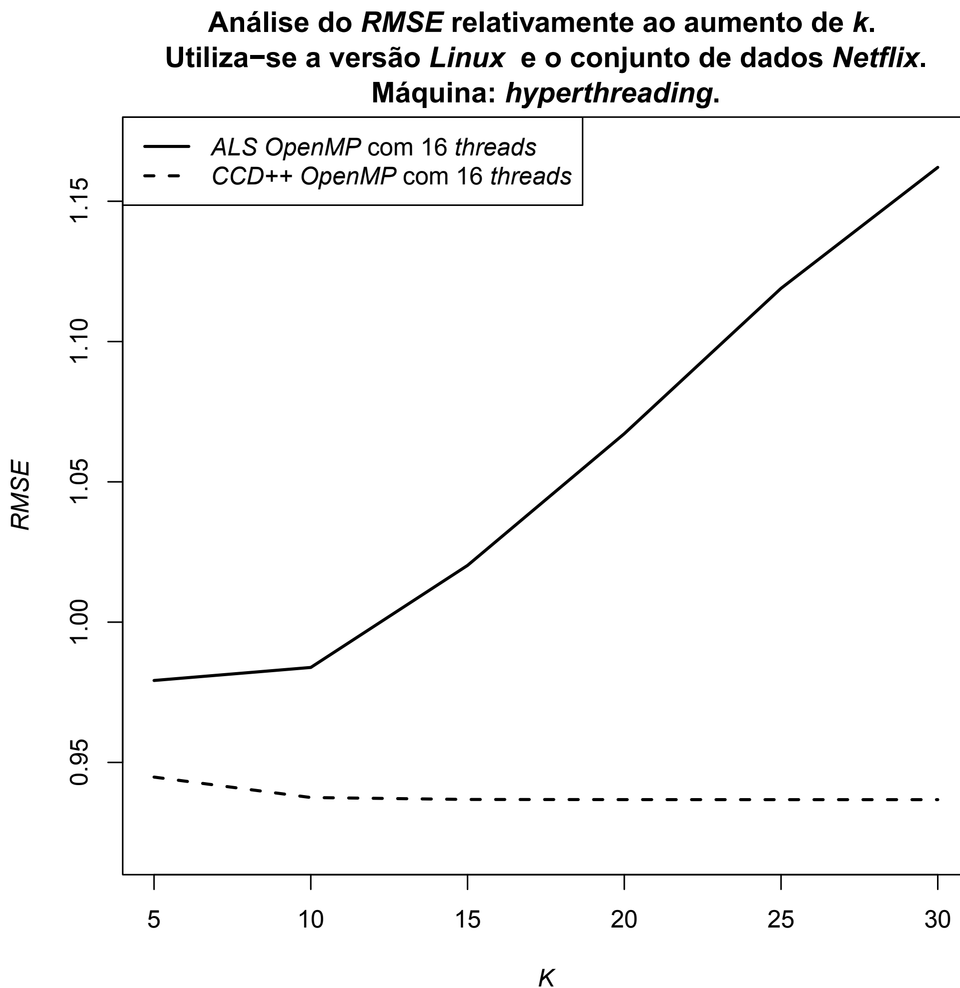
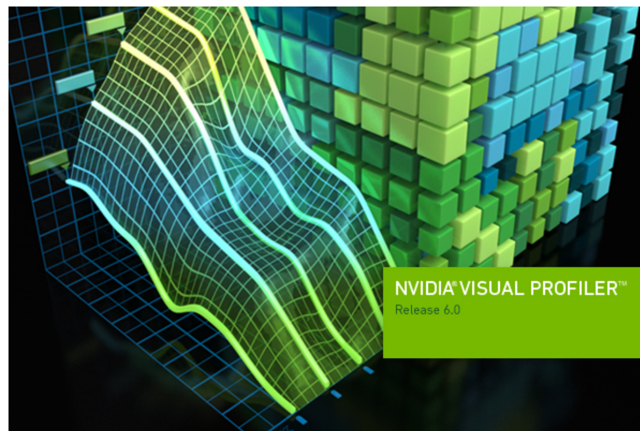


Figura 7.3: Análise comparativa do *RMSE* entre os algoritmos *ALS* e *CCD++*, relativamente ao aumento de k. A parametrização é $\lambda = 0.1$, $T = 15$.

Para um valor de k, superior a 10, o algoritmo *ALS* aumenta o *RMSE*, enquanto o algoritmo *CCD++* mantém-se estável. Tal como esperado, o algoritmo *CCD++*, normalmente, tem melhores resultados [75].

7.4 Análise do comportamento em *CUDA* dos algoritmos recorrendo a *profiling* visual

Para analisar visualmente o comportamento de cada um dos *kernels*, utiliza-se um conjunto de dados pequeno. Os detalhes deste conjunto de dados não são importantes, porque nesta análise o que se torna relevante é apenas investigar a forma como os *kernels* são executados, para perceber até que ponto eles estão a aproveitar o processamento disponível. Por outro lado, um conjunto de dados pequeno facilita a interpretabilidade da *timeline* visual, porque o número de eventos ao longo do tempo é muito inferior. Tanto os parâmetros como o conjunto de dados utilizados são os mesmos para os dois algoritmos *CCD++* e *ALS* em estudo. O programa utilizado é a versão *Linux* do *NVIDIA Visual Profiler (NVVP)*.



7.4.1 Análise do algoritmo *CCD++*

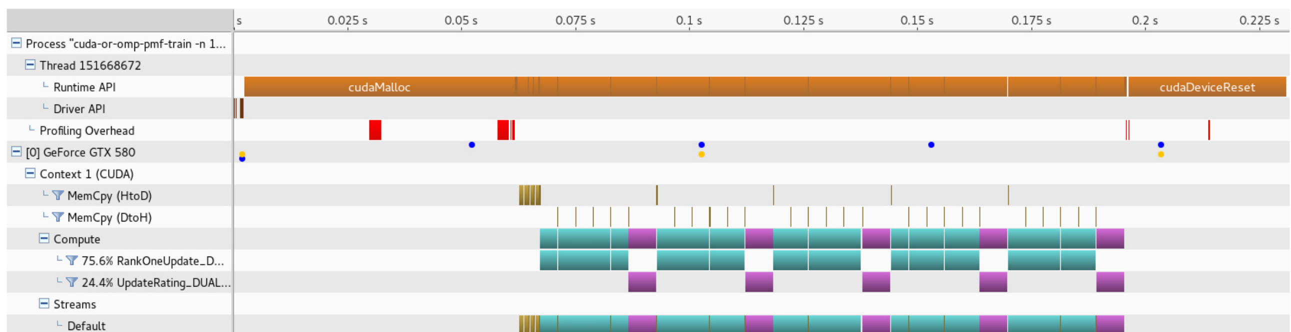


Figura 7.4: *Timeline* para 1 iteração de *CCD++* no *GPU*.

Na linha *[0] GeForce GTX 580*, os círculos azuis representam a variação da rotação da ventoinha da gráfica, e os círculos amarelos representam a variação da temperatura da gráfica. Quanto mais altos, maior é o valor lido pelos sensores. A linha *Compute* é expandida em duas linhas, uma para cada *kernel*. Pela análise da *Timeline*, verifica-se que a maior parte do tempo é despendido a processar os dois *kernels*. Assim que o espaço é alocado e os dados são copiados

para o *device*, a execução dos *kernels* é praticamente contígua. Apenas se veem algumas cópias de dados *device to host* (linha *MemCopy (DtoH)*), *host to device* (linha *MemCopy (HtoD)*) que são insignificantes, relativamente ao tempo de execução ocupado pelos *kernels*. Note-se que o *kernel* *RankOneUpdate* é executado várias vezes seguidas e o *UpdateRating* é executado k vezes. Pela *timeline*, percebe-se que $k = 5$.

Na Figura 7.5, é apresentada a visualização da execução com 15 iterações.

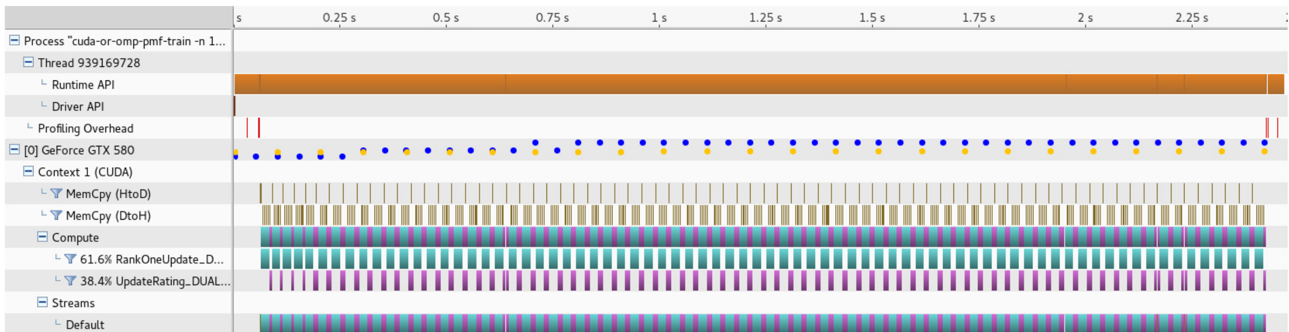


Figura 7.5: *Timeline* para 15 iterações de *CCD++* no *GPU*.

Nesta segunda *timeline*, em que o número de iterações é mais elevado, as cópias de dados durante a execução dos *kernels* *device to host*, *host to device*, têm linhas muito finas. Outro facto é que o tempo gasto é preenchido pelo processamento praticamente contíguo dos *kernels*.

Por fim, falta ainda saber até que ponto os recursos disponibilizados estão a ser inteiramente utilizados. Por isso, seguidamente, recorre-se a uma análise adicional que consiste em averiguar as mensagens de alerta emitidas pelo *NVVP*.

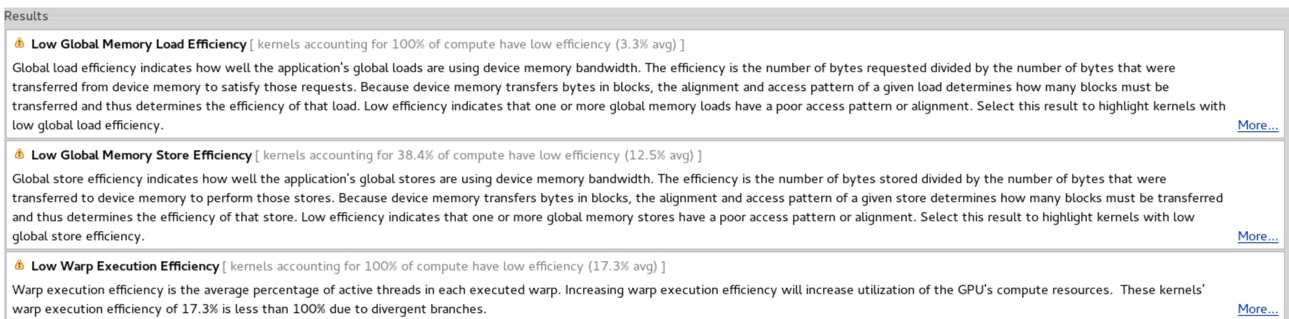


Figura 7.6: Análise adicional do *CCD++* com o *NVVP*.

Estas mensagens são independentes do número de iterações utilizadas. Mas o seu conteúdo pode ter ligeiras variações, conforme o conjunto de dados e o valor de k. Contudo, as mensagens são sempre as mesmas. Apenas as percentagens podem variar ligeiramente. Na Figura 7.6, são apresentadas algumas destas mensagens, produzidas após a execução das 15 iterações. As duas primeiras mostram a eficiência de acesso à memória global da placa gráfica, sendo a primeira relativa às leituras e a segunda às escritas. Segundo estas percentagens (3.3% para leituras e 12.5% para escritas), a eficiência das transferências é baixa. Isto poderia ser corrigido com uma modificação nas estruturas de dados, de forma a obter um melhor alinhamento dos dados às

linhas de memória. Mas, mesmo assim, dependendo do padrão de acesso aos dados durante a execução, esta modificação poderia não resultar em melhorias na eficiência. A terceira linha da Figura 7.6 diz respeito à eficiência da execução das *threads* num *"warp"* (*multiprocessador*).

Para analisar o comportamento mais detalhado dos sensores da placa gráfica, existe, para *Windows*, a aplicação *GPU-Z*.

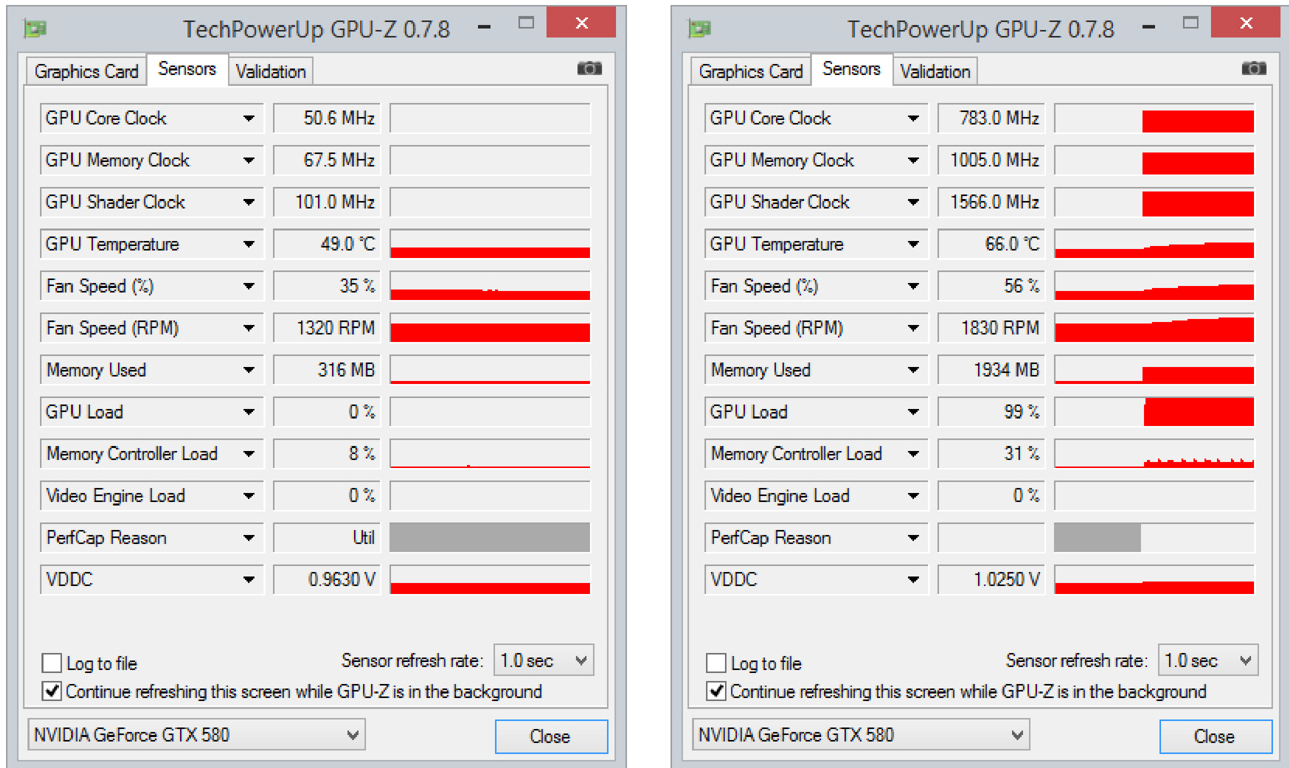


Figura 7.7: Informação proveniente dos sensores da placa gráfica com o algoritmo *CCD++*.

Desta análise, salienta-se a baixa carga na controladora de memória, o que corrobora aquilo que já foi dito anteriormente. O conjunto de dados utilizado para criar a Figura 7.7 é o *Netflix* (Secção 6.3). Na Figura 7.7, do lado esquerdo, encontra-se a placa gráfica em vazio e, do direito, encontra-se em pleno processamento dos *kernels*. A quantidade de memória consumida com este conjunto de dados é $1934 - 316 = 1618MB$.

7.4.2 Análise do algoritmo *ALS*

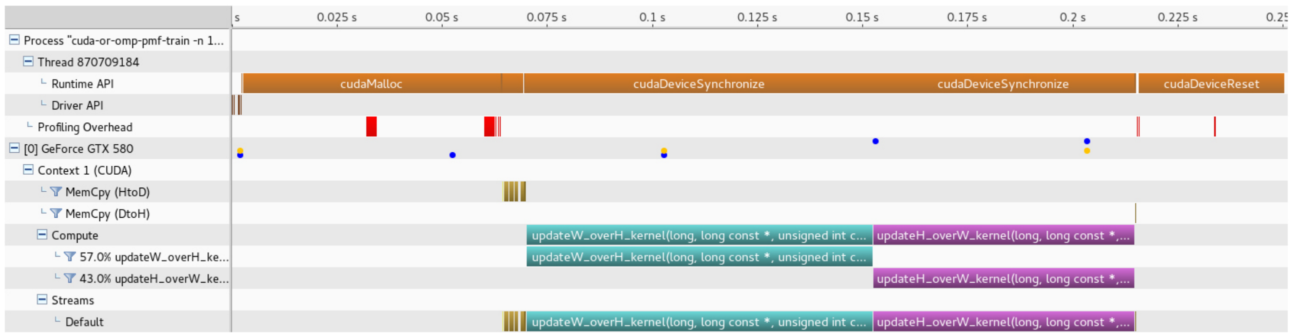


Figura 7.8: *Timeline* para 1 iteração de *ALS* no *GPU*.

Na linha *[0] GeForce GTX 580*, os círculos azuis representam a variação da rotação da ventoinha da gráfica, e os círculos amarelos representam a variação da temperatura da gráfica. Quanto mais altos, maior é o valor lido pelos sensores. A linha *Compute* é expandida em duas linhas, uma para cada *kernel*. Pela análise da *Timeline*, verifica-se que a maior parte do tempo é despendido a processar os dois *kernels*. Assim que o espaço é alocado e os dados são copiados para o *device*, a execução dos *kernels* é contígua. Apenas no final do processamento é que são copiadas para o *host* as matrizes W e H.

Passa-se à analise do comportamento para mais do que uma iteração.

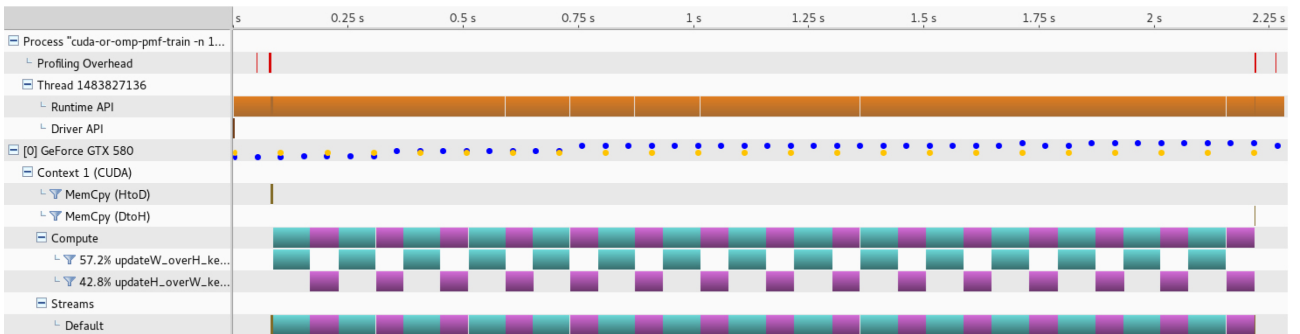


Figura 7.9: *Timeline* para 15 iterações de *ALS* no *GPU*.

Nesta segunda *timeline*, em que o número de iterações é mais elevado, à exceção do início e do fim da execução, não existem cópias de dados durante a execução dos *kernels device to host*, *host to device*.

Por fim, falta ainda saber até que ponto os recursos disponibilizados estão a ser inteiramente utilizados. Por isso, seguidamente, recorre-se a uma análise adicional, que consiste em averiguar as mensagens de alerta emitidas pelo *NVVP*.

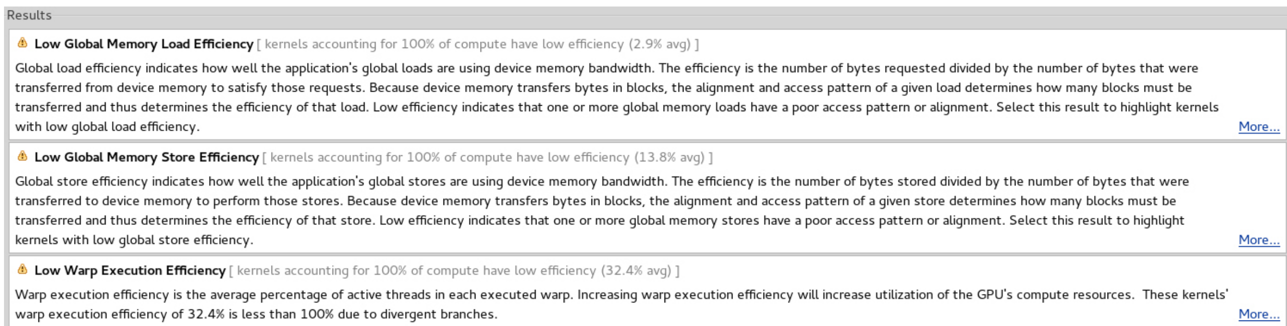


Figura 7.10: Análise adicional do *ALS* com o *NVVP*.

Estas mensagens são independentes do número de iterações utilizadas. Mas o seu conteúdo pode ter ligeiras variações, conforme o conjunto de dados e o valor de k. Contudo, as mensagens são sempre as mesmas. Apenas as percentagens podem variar ligeiramente. Na Figura 7.10, são apresentadas algumas destas mensagens, produzidas após a execução das 15 iterações. As duas primeiras mostram a eficiência de acesso à memória global da placa gráfica, sendo a primeira relativa às leituras e a segunda às escritas. Segundo estas percentagens (2.9% para leituras e 13.8% para escritas), a eficiência das transferências é baixa. Isto poderia ser corrigido com uma modificação nas estruturas de dados, de forma a obter um melhor alinhamento dos dados às linhas de memória. Mas, mesmo assim, dependendo do padrão de acesso aos dados durante a execução, esta modificação poderia não resultar em melhorias na eficiência. A terceira linha da Figura 7.10 diz respeito à eficiência da execução das *threads* num *"warp"* (*multiprocessador*).

Análise em *Windows* utilizando a aplicação *GPU-Z*.

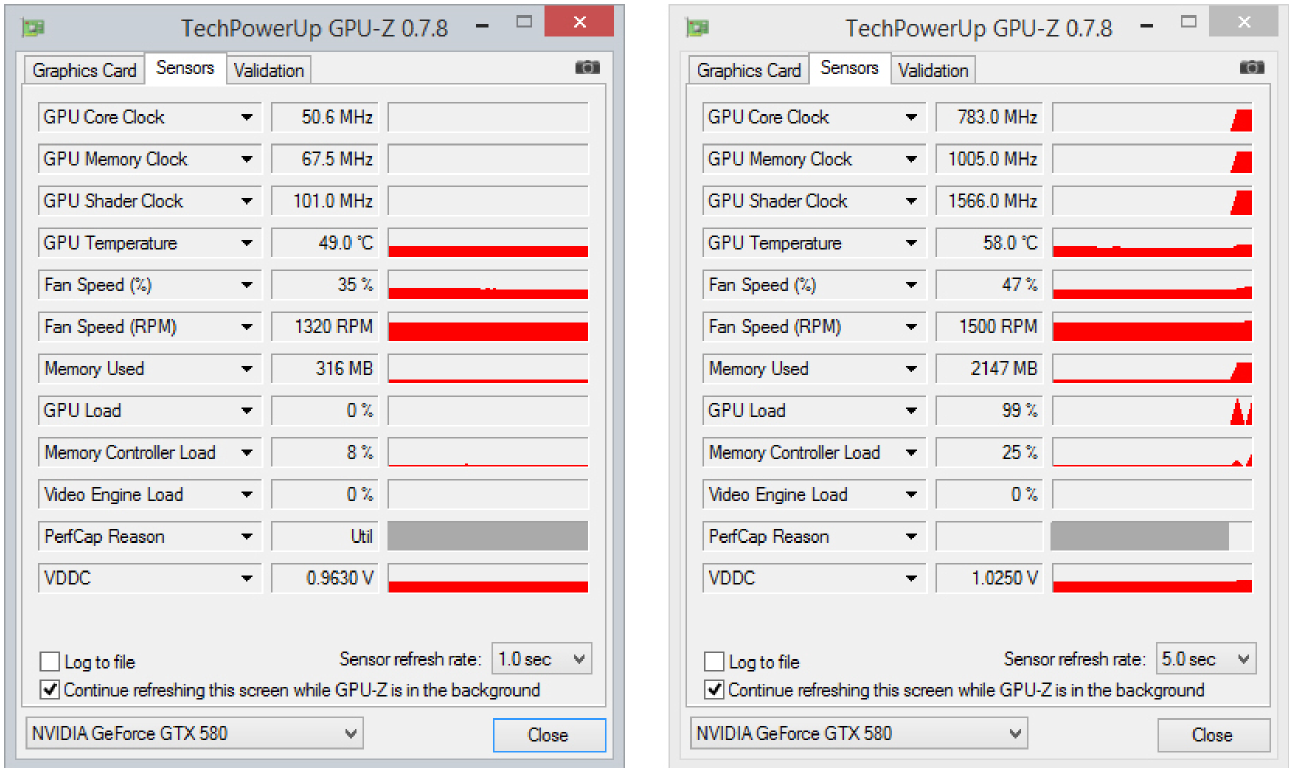


Figura 7.11: Informação proveniente dos sensores da placa gráfica com o algoritmo *ALS*.

Desta análise, salienta-se a baixa carga na controladora de memória, o que corrobora aquilo que já foi dito anteriormente. O conjunto de dados utilizado para criar a Figura 7.11 é o *Netflix* (Secção 6.3). Na Figura 7.11, do lado direito, encontra-se a placa gráfica em vazio e, do esquerdo, encontra-se em pleno processamento dos *kernels*. A quantidade de memória consumida com este conjunto de dados é $2147 - 316 = 1831MB$.

7.4.3 Comparação entre *ALS* e *CCD++*

A quantidade de trabalho atribuída a cada *kernel* no *ALS* é muito superior à do *CCD++*. Para uma iteração no *ALS*, cada um dos *kernels* apenas é executado uma vez. A nível de escalabilidade, o *CCD++* é melhor, porque necessita de alocar uma menor quantidade de memória. Enquanto o *CCD++*, apenas necessita alocar uma coluna de W e uma de H, o *ALS* necessita alocar W e H por inteiro. Um fator, também, importante está na quantidade de trabalho associada a cada *kernel*. No caso do *ALS*, para conjuntos de dados de grandes dimensões, o tempo que cada *kernel* demora a executar pode ser relativamente elevado. No algoritmo *ALS*, a granularidade dos *kernels* é muito mais grossa em relação ao *CCD++*.

A partir da análise adicional, conclui-se que ambos os algoritmos têm problemas com o acesso contíguo à memória. No entanto, a execução do algoritmo *ALS* consegue utilizar com maior eficiência os recursos disponibilizados, devido à não existência de cópias de dados *device to host*, *host to device*.

7.4.4 Erros conhecidos

"Runtime API error 6: the launch timed out and was terminated." - Este erro ocorre, quando a execução de um único *kernel* demora mais que oito a dez segundos. Pode surgir em conjuntos de dados de tamanho superior ao *Netflix* (Secção 6.3), sobretudo no algoritmo *ALS*, residindo a sua explicação na granularidade grossa dos *kenels* do *ALS*. Neste caso, é necessário instalar uma placa gráfica secundária sem nenhum ecrã conectado. É um problema que acontece tanto em *Windows* como em *Linux*, e é a forma que o *driver* tem para prevenir que o ecrã do computador não deixe de responder por longos períodos de tempo.

7.4.5 Artigo submetido

Com base no trabalho criado para esta dissertação e devido aos bons resultados obtidos nas experiências realizadas, foi submetido e aceite pelo júri um artigo científico com o título *Accelerating Recommender Systems using GPUs*, para a conferência *ACM SAC 2015 (Association for Computing Machinery Symposium on Applied Computing)*.

7.5 Sumário

Neste Capítulo, começou-se por falar dos parâmetros escolhidos e das respetivas razões. De seguida, foram apresentados os resultados e feita uma análise do comportamento dos algoritmos relativamente à parametrização. Por fim, foi realizada uma análise mais pormenorizada, recorrendo a *profiling* visual.

Os *spedups* obtidos nos algoritmos *CCD++* e *ALS* com o conjunto de dados *Netflix* (Secção 6.3) podem ser vistos, sumariamente, lado a lado na Tabela 7.7.

Tabela 7.7: Comparação de *Speedups* entre o *CCD++* e o *ALS* em *OpenMP* e em *CUDA*.

Máquina *hyperthreading* (Secção 6.2), SO: *Linux*			
Tipo de teste	*CCD++ double*	*CCD++ float*	*ALS float*
2 *threads*	$1,6$	$1,7$	$1,9$
8 *threads*	$3,8$	$4,7$	$4,6$
16 *threads*	$4,1$	$5,3$	$4,3$
32 *threads*	$3,8$	$5,2$	$4,5$
CUDA* com 16 *blocos* de 512 *threads		**3,1**	**4,4**

Máquina *hyperthreading* (Secção 6.2), SO: *Windows*			
Tipo de teste	*CCD++ double*	*CCD++ float*	*ALS float*
2 *threads*	$1,8$	$2,3$	$1,9$
8 *threads*	$4,0$	$6,9$	$4,2$
16 *threads*	$4,3$	$9,5$	$5,5$
32 *threads*	$4,4$	$9,5$	$5,5$
CUDA* com 16 *blocos* de 512 *threads		**14,8**	**6,2**

Máquina *cracs-gpu* (Secção 6.1), SO: *Linux*			
Tipo de teste	*CCD++ double*	*CCD++ float*	*ALS float*
2 *threads*	$1,9$	$1,9$	$2,0$
8 *threads*	$5,0$	$5,5$	$5,0$
16 *threads*	$5,3$	$6,3$	$4,8$
32 *threads*	$5,4$	$6,4$	$4,8$
CUDA* com 14 *blocos* de 512 *threads		**1,3**	**2,6**

No Capítulo seguinte, serão apresentadas as conclusões e o trabalho futuro.

Capítulo 8

Conclusões

Esta dissertação mostra a vantagem da utilização de *GPUs* na implementação de algoritmos de recomendação baseados em fatorização de matrizes. São comparadas as versões sequenciais e *multicore* dos algoritmos *ALS* e *CCD++* com a versão *GPU*, verificando-se que existe um melhor desempenho a nível da velocidade/tempo de processamento. Pelo conhecimento proveniente da investigação para esta dissertação, neste trabalho, é apresentada a primeira implementação de *ALS* e de *CCD++* em *CUDA*.

Relativamente a um servidor *multicore*, as vantagens de uma implementação *CUDA* são o baixo consumo de energia, o baixo preço por *core*, relativamente aos *CPUs*, e a libertação dos *cores* do *CPU* para outras tarefas. Atualmente, os computadores integram várias *PCI slots* que podem ser utilizadas para instalar um *GPU* ou vários *GPUs*. Por isso, é relativamente simples aumentar a capacidade computacional do *hardware* existente.

As placas gráficas de gama posterior à utilizada no suporte experimental desta dissertação têm uma capacidade superior de processamento e densidade de memória. Futuramente, pretende-se testar nestas novas placas gráficas os algoritmos implementados. Pretende-se, também, testar conjuntos de dados (*data sets*) maiores, existindo, porém, problemas relacionados com a limitação da memória das placas gráficas. No entanto, para conjuntos de dados com o dobro do tamanho do *Netflix*, por exemplo, não constitui qualquer problema para a densidade de memória das placas utilizadas neste trabalho (consultar a Subsecção 7.4.1). Será possível eliminar os problemas associados a este limite da memória, se se implementar uma gestão de memória que suporte paginação e que permita trabalhar com conjuntos de dados de tamanhos superiores ao da memória disponível. Tomando, ainda, como exemplo o conjunto de dados *Netflix*, que obteve um *speedup* de 14.8, verificou-se que este aumento da velocidade foi melhor do que o *speedup* melhor obtido na literatura para máquinas *multicore* [75].

Considerando a simplicidade e a facilidade inerentes à implementação do algoritmo *SGD*,

aquando da investigação realizada para a elaboração desta dissertação, houve a preocupação de dedicar algum tempo de estudo a este algoritmo. No entanto, devido a factos encontrados na literatura [75], nomeadamente, a convergência instável, os mínimos locais e a menor capacidade preditiva (relativamente ao *ALS* e *CCD++*), optou-se pela não implementação do mesmo. Futuramente, para quem for usar os algoritmos implementados neste trabalho, é aconselhável utilizar uma placa gráfica dedicada para poder ter o melhor desempenho possível.

Para trabalho futuro, é necessário investigar formas de melhorar o balanceamento de carga e a gestão da memória do *GPU* (paginação?), a fim de permitir executar aplicações de maior dimensão, experiências em mais conjuntos de dados, e de possibilitar a análise em maior detalhe dos fatores que afetam o desempenho dos algoritmos e a relação destes fatores com a arquitetura dos *GPUs*, compiladores, etc.

Anexo A

Instalação do ambiente de programação

A.1 Instalação do *CUDA toolkit 6.0* no *Linux fedora 20*

Lista de comandos para instalar o *CUDA toolkit 6.0* no *Linux fedora 20*). O *driver* instalado por esta lista de comandos foi desenvolvido para as seguintes series de placas, *GeForce 8/9/200/300/400/500/600/700 series*.

- Atualizar o *kernel* do *Linux*:

  $ su -
  # yum update kernel* selinux-policy*
  # reboot

- Instalar o *driver* da *NVIDIA* para a placa:

  $ su -
  # yum localinstall –nogpgcheck
  http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-$(rpm -E
  %fedora).noarch.rpm
  http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-$(rpm
  -E %fedora).noarch.rpm
  # yum install akmod-nvidia xorg-x11-drv-nvidia-libs kernel-devel acpid
  # mv /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r)-nouveau.img
  # dracut /boot/initramfs-($uname - r).img$(uname -r)
  # reboot
  $ su -
  # yum install vdpauinfo libva-vdpau-driver libva-utils
  # exit

- Restaurar o *plymouth*:

  $ su -
  # vi /etc/default/grub
  No ficheiro editado acima, identificar e comentar com "#" a linha "*#GRUB_TERMINAL_OUTPUT="console"*", e adicionar as linhas abaixo:

  GRUB_TERMINAL_OUTPUT="gfxterm"
  GRUB_GFXMODE="1280x800x32"
  GRUB_GFXPAYLOAD_LINUX="keep"
  GRUB_VIDEO_BACKEND="vbe"
  GRUB_FONT_PATH="/boot/grub2/fonts/unicode.pf2"

  # yum install plymouth-theme-solar
  # plymouth-set-default-theme –list
  # plymouth-set-default-theme solar
  # /usr/libexec/plymouth/plymouth-update-initrd
  # cp /boot/grub2/grub.cfg /boot/grub2/grub.cfg.bkp
  # grub2-mkconfig -o /boot/grub2/grub.cfg
  # reboot

- Instalar o *CUDA toolkit 6.0*:

  $ su -
  # yum install perl-Env
  # yum install environment-modules.x86_64
  # yum install wget make gcc-c++ freeglut-devel libXi-devel libXmu-devel mesa-libGLU-devel
  # exit
  $ cd Downloads/
  $ wget http://developer.download.nvidia.com/compute/cuda/6_0/rel/installers/cuda_6.0.37_linux_64.run
  $ chmod +x cuda_6.0.37_linux_64.run
  $ sudo ./cuda_6.0.37_linux_64.run –silent -toolkit -toolkitpath=/usr/local/cuda-6.0 -samples -samplespath=/usr/local/cuda-6.0/samples -override
  $ cd
  $ vi .bashrc
  No ficheiro "*.bashrc*" editado acima, adicionar as duas seguintes linhas:

  PATH=$PATH:/usr/local/cuda-6.0/bin; export PATH
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-6.0:/usr/local/cuda-6.0/lib64:/usr/local/cuda/lib; export LD_LIBRARY_PATH

  $ exit
  $ su -

```
# cd /etc/ld.so.conf.d/
# touch cuda.conf
# echo /usr/local/cuda-6.0/lib >> cuda.conf
# echo /usr/local/cuda-6.0/lib64 >> cuda.conf
# ldconfig
# exit
```

- Comandos de verificação:

```
$ lspci -v | grep -i nvidia
$ nvidia-settings -q NvidiaDriverVersion
$ cat /proc/driver/nvidia/version
$ lsmod | grep -i nvidia
$ nvcc -V
$ nvidia-smi
```

A.2 Executar o projeto do *VS 2013* em *Windows*

Para executar o projeto do *VS 2013*, é necessário instalar os seguintes programas pela ordem abaixo:

1. Instalar o último *driver* da placa gráfica.
2. Instalar o *Microsoft Visual Studio Ultimate 2012*
3. Instalar o *Microsoft Visual Studio Ultimate 2013*
4. Instalar o *CUDA toolkit 6.0*

Para executar o executável compilado fornecido, basta ter o último *driver* da placa gráfica instalado.

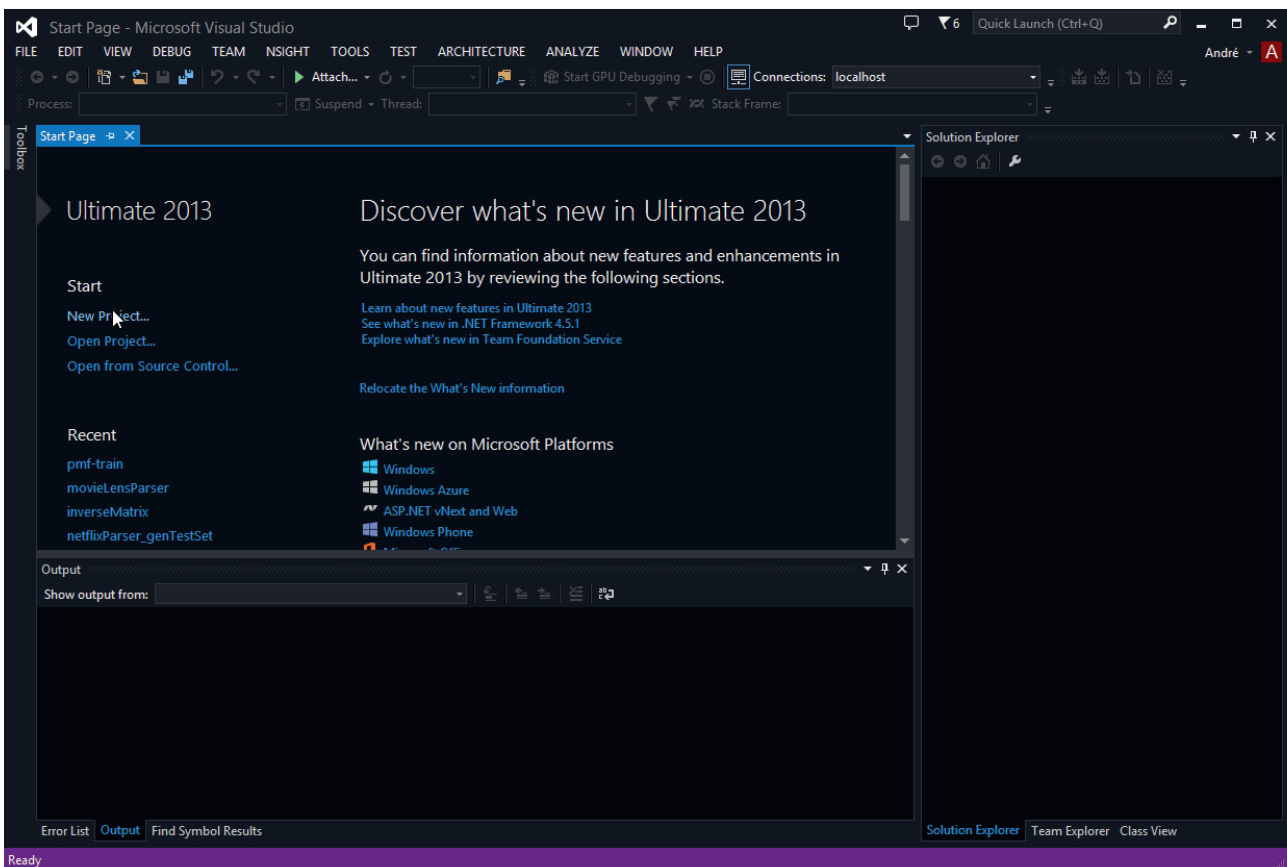
Anexo

B

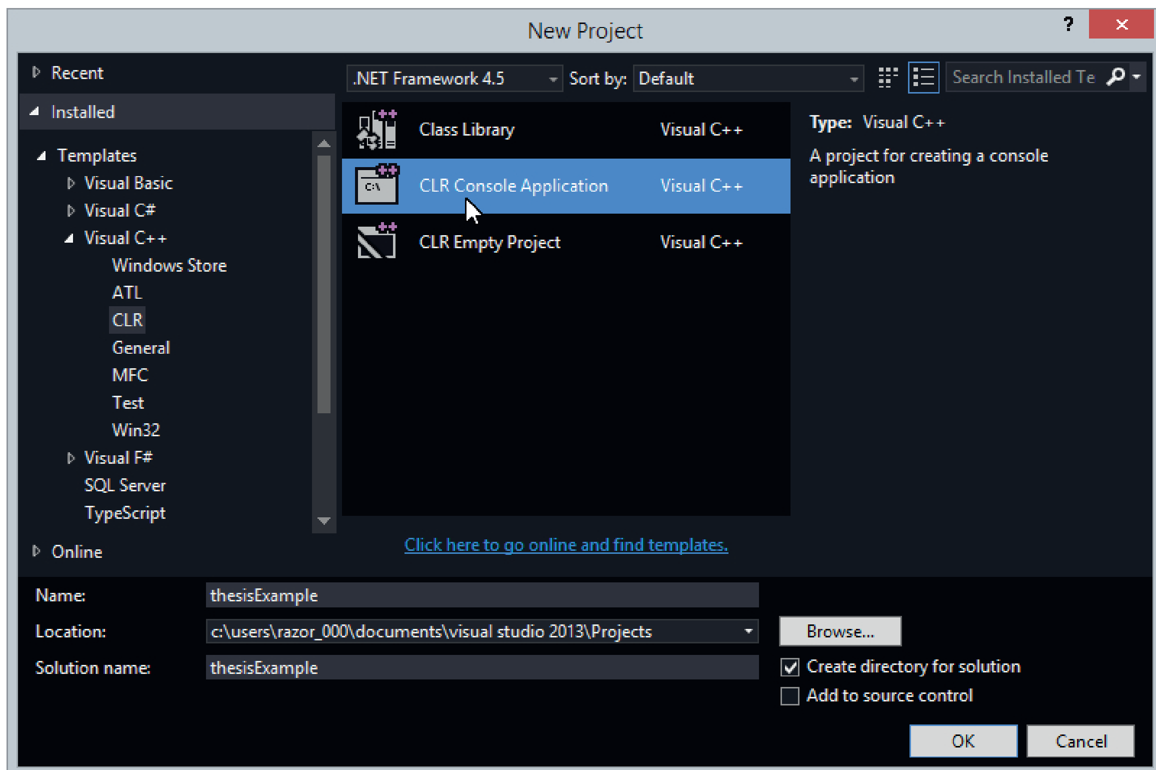
Configurações

B.1 Configuração de um projeto em *C++* e *CUDA* com o *Microsoft Visual Studio Ultimate 2013*

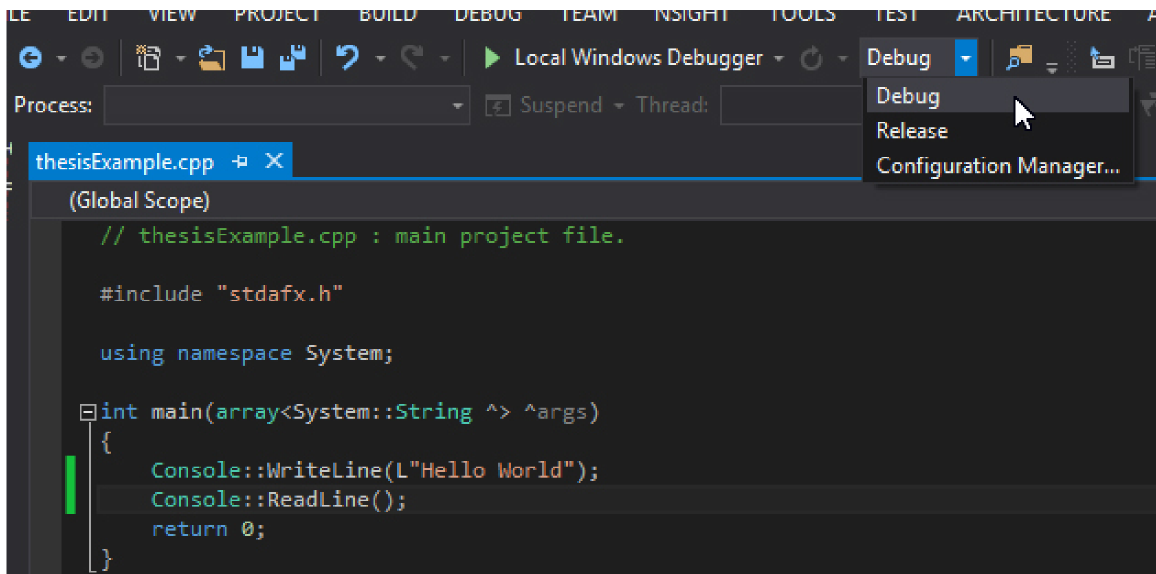
1. Abrir o *Microsoft Visual Studio Ultimate 2013*:



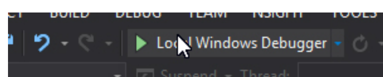
2. Clicar em *New project*. Na *dialog box*, escolher as seguintes opções e dar um nome ao projeto:



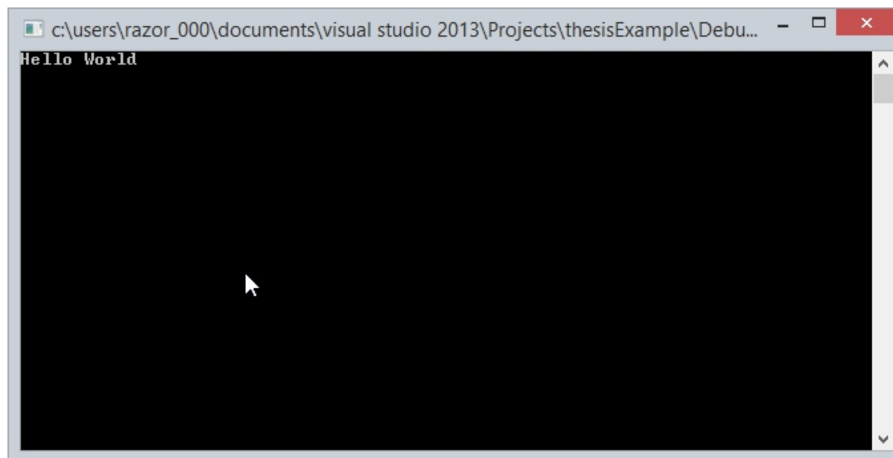
3. Selecionar o modo *debug*:



4. Para executar o projeto, clicar em:

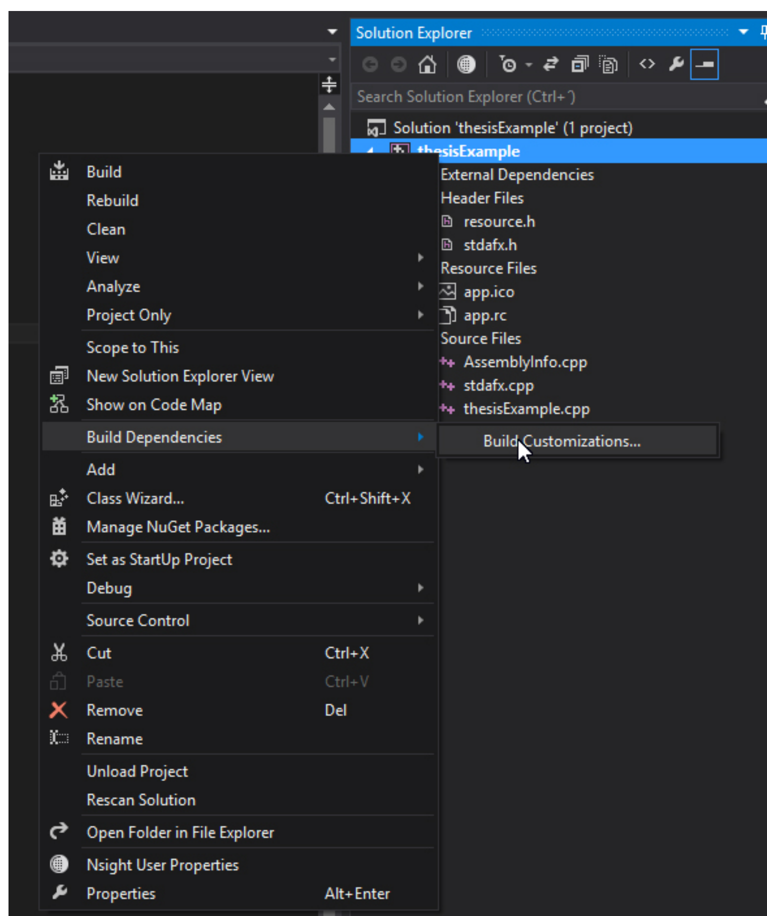


5. O resultado é o seguinte:

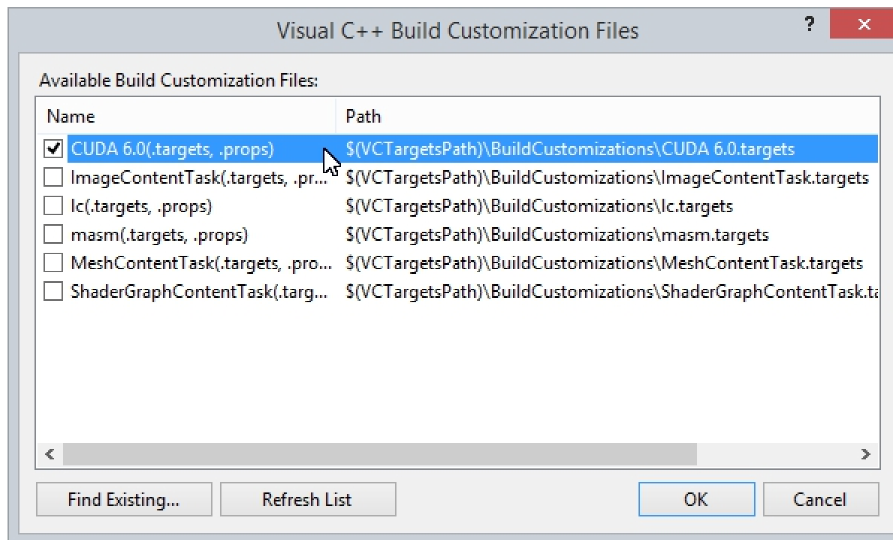


6. Configurar o projeto para funcionar em *CUDA* no modo *debug* requer os seguintes passos:

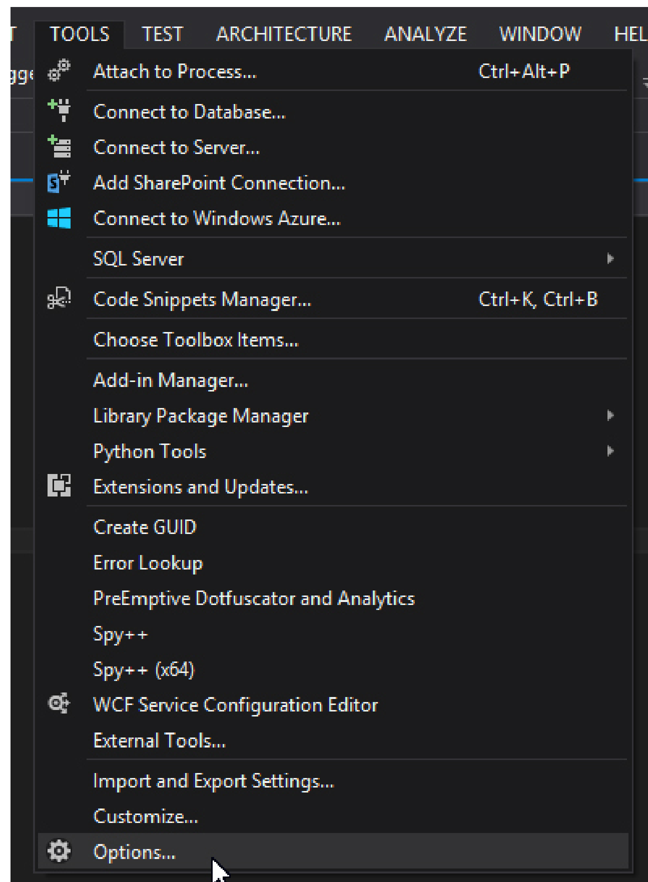
   (a) No *Solution Explorer*, clicar com o botão direito do rato sobre o nome do projeto e selecionar *Build Dependencies –> Build Customisations* (caso o *Solution Explorer* não esteja visível fazer *CTRL+ALT+L*):



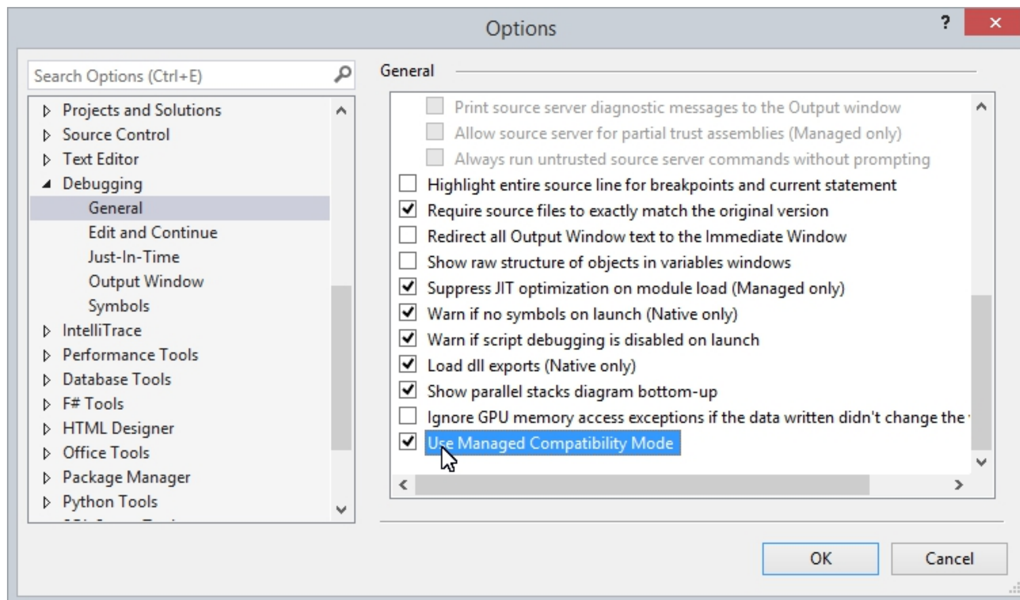
(b) Na *dialog box*, colocar um visto na opção selecionada e clicar em *OK*:



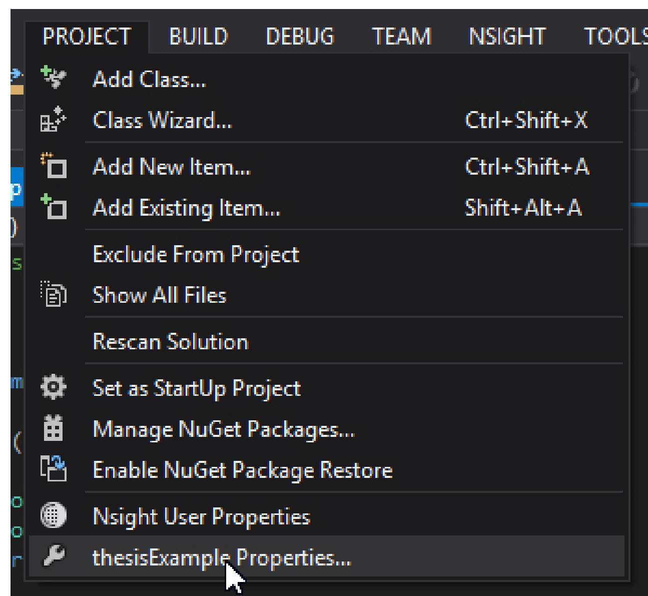
(c) Ir ao menu *Tools* –> *Options*:



(d) Na *dialog box*, colocar um visto na opção selecionada e clicar em *OK*:

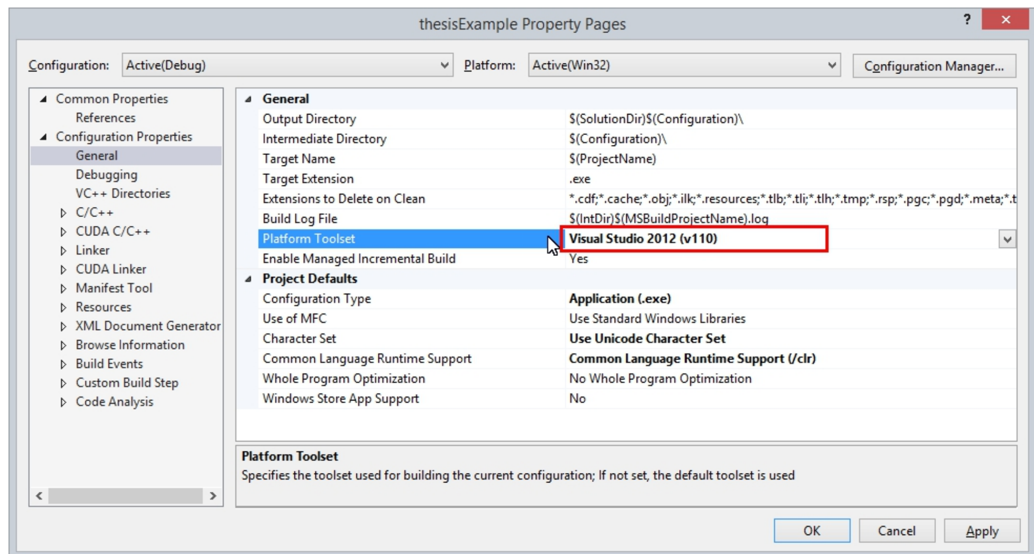


(e) Ir ao menu *Project –> projName Properties...*:

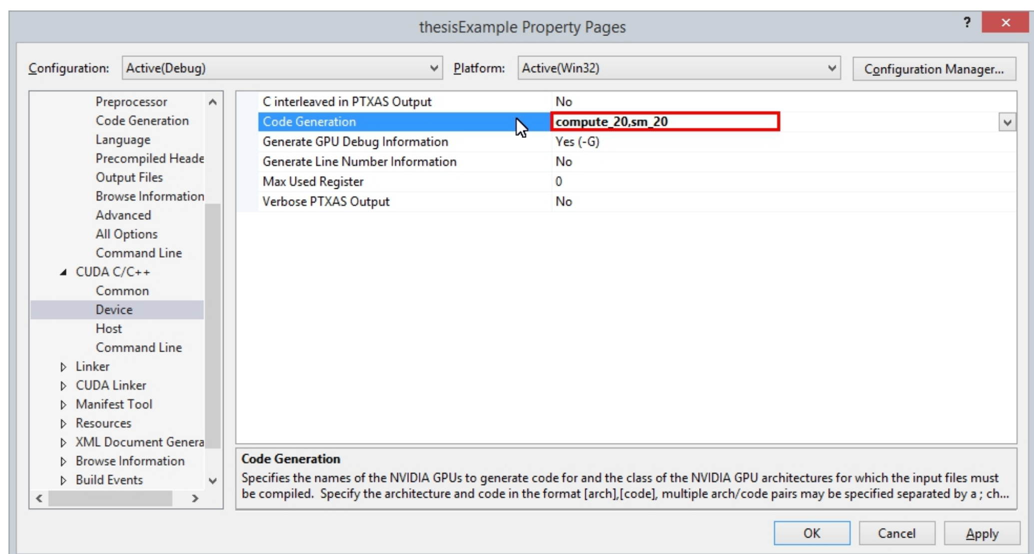


(f) Na *dialog box*, fazer as seguintes configurações:

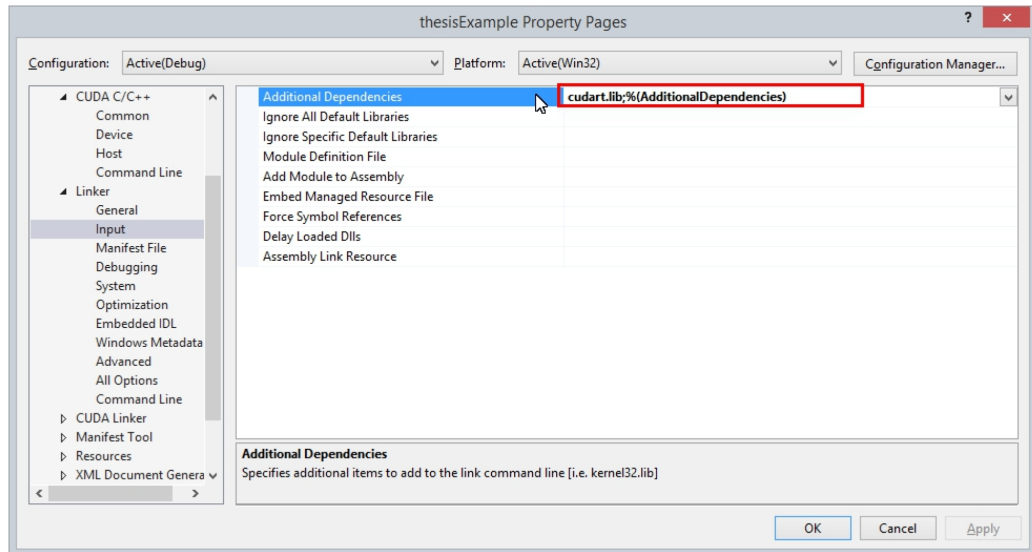
- Modificar o que está marcado com o retângulo vermelho:



- Modificar o que está marcado com o retângulo vermelho:



- Na opção marcada com o retângulo vermelho, escrever "*cudart.lib;%(AdditionalDependencies)*". Clicar em *Apply*, *OK* (apenas clicar em *OK* não é suficiente, porque as alterações são ignoradas):



7. Código para testar:

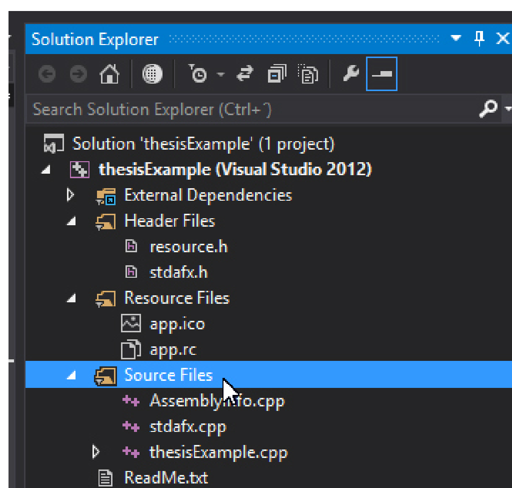
   (a) Abrir ou selecionar o ficheiro com o nome do projeto "*projectName.cpp*". Caso não esteja aberto, é necessário procurar no *Solution Explorer*:

```
thesisExample.cpp
(Global Scope)
// thesisExample.cpp : main project file.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    Console::ReadLine();
    return 0;
}
```

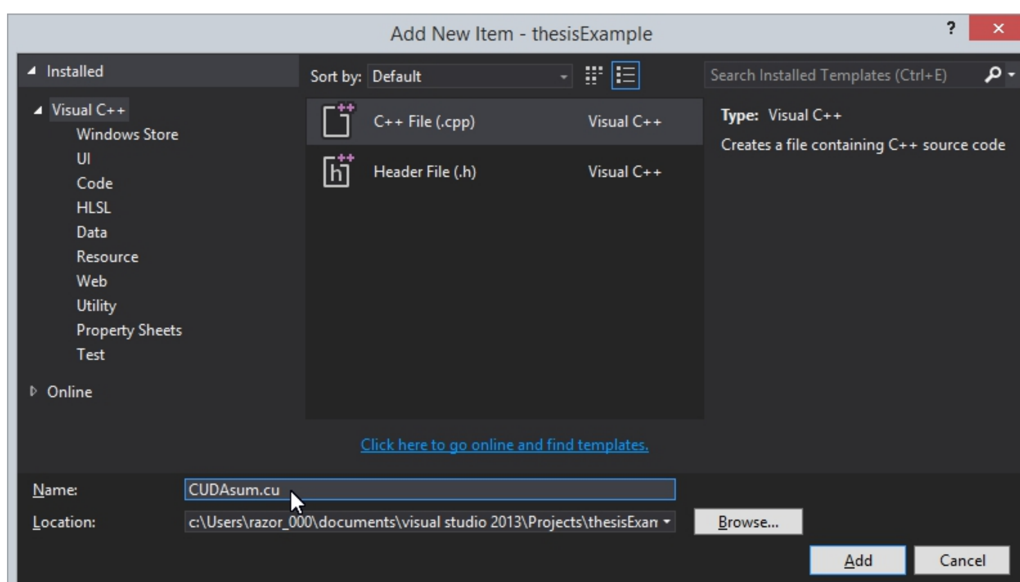
(b) No ficheiro com o nome do projeto "*projectName.cpp*", apagar tudo e colar o seguinte código:

```
// thesisExample.cpp : main project file.

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
# include <omp.h>
extern "C"  void kernel_wrapper(int vecSize, int nBlocks, int nthreadsPerblock, int
    *myVector, int * &cpyVector);

int main(int argc, char * argv[])
{
   int vecSize = 16, nBlocks = 16, nthreadsPerblock = 1;
   int myVector[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
   int * cpyVector;
   cpyVector = (int *)malloc(vecSize * sizeof (int));

   printf("The myVector values :");
   for (size_t x = 0; x < vecSize; ++x){
      printf("%d ", myVector[x]);
   }
   printf("\n\nThe copyVector current values :");
   for (size_t x = 0; x < vecSize; ++x){
      printf("%d ", cpyVector[x]);
   }

   kernel_wrapper(vecSize, nBlocks, nthreadsPerblock, myVector, cpyVector);

   printf("\n\nThe copyVector final values :");
   for (size_t x = 0; x < vecSize; ++x){
      printf("%d ", cpyVector[x]);
   }

   printf("\n\nPress one key to quit...");
   free(cpyVector);
   getchar();
  return 0;
}
```

(c) No *Solution Explorer*, selecionar a pasta *Source Files* e fazer *Ctrl+Shift+A*.



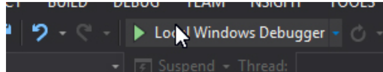
(d) É aberta uma *dialog box* para acrescentar um novo ficheiro. Este ficheiro tem que ter extensão "*.cu*". De seguida, tendo o nome dado com a extensão pretendida, clicar em *Add*. No caso do exemplo, o nome dado é "*CUDAsum.cu*":



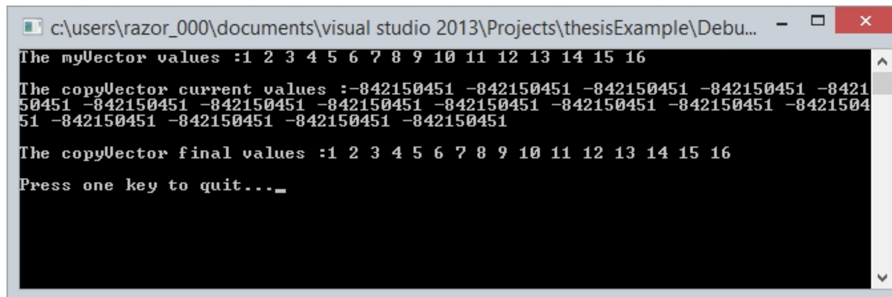
(e) No ficheiro "*CUDAsum.cu*", colar o seguinte código:

```
# include <cuda.h>
# include <cuda_runtime_api.h>
# include <device_launch_parameters.h>
# include <device_functions.h>
# include <stdio.h>

cudaError_t NVcode(int vecSize, int nBlocks, int nthreadsPerblock, int *myVector,
     int * &cpyVector);
__global__ void copyVector(const int numItens, const int *first, int * returnVec);
__global__ void copyVector(const int numItens, const int *first, int * returnVec){
   int ii = threadIdx.x + blockIdx.x * blockDim.x;
   for (size_t i = ii; i < numItens; i += blockDim.x* gridDim.x){
      returnVec[i] = first[i];
   }
}

extern "C" void kernel_wrapper(int vecSize, int nBlocks, int nthreadsPerblock, int
     *myVector, int * &cpyVector){
   cudaError_t cudaStatus = NVcode(vecSize, nBlocks, nthreadsPerblock, myVector,
       cpyVector);
   cudaStatus = cudaDeviceReset();
}

cudaError_t NVcode(int vecSize, int nBlocks, int nthreadsPerblock, int *myVector,
     int * &cpyVector){
   cudaError_t cudaStatus;
   int * dev_myVector;
   int * dev_cpyVector;
   cudaStatus = cudaMalloc((void **)& dev_myVector, vecSize * sizeof (int));
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, " cudaMalloc failed ! %s\n", cudaGetErrorString(cudaStatus));
      goto Error;
   }
   cudaStatus = cudaMalloc((void **)& dev_cpyVector, vecSize * sizeof (int));
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, " cudaMalloc failed ! %s\n", cudaGetErrorString(cudaStatus));
      goto Error;
   }
   cudaStatus = cudaMemcpy(dev_myVector, myVector, vecSize * sizeof (int),
      cudaMemcpyHostToDevice);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, " cudaMemcpy failed ! %s\n", cudaGetErrorString(cudaStatus));
      goto Error;
   }
   copyVector << <nBlocks, nthreadsPerblock >> >(vecSize, dev_myVector,
       dev_cpyVector);
   cudaStatus = cudaMemcpy(cpyVector, dev_cpyVector, vecSize * sizeof (int),
      cudaMemcpyDeviceToHost);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, " cudaMemcpy failed ! %s\n", cudaGetErrorString(cudaStatus));
      goto Error;
   }
   cudaFree(dev_myVector);
   cudaFree(dev_cpyVector);
Error:
   cudaFree(dev_myVector);
   cudaFree(dev_cpyVector);
   return cudaStatus;
}
```

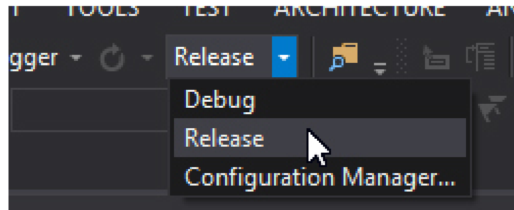
(f) Para executar o projeto em *C++* que usa *CUDA*, basta clicar em:



(g) O resultado da execução é o seguinte:

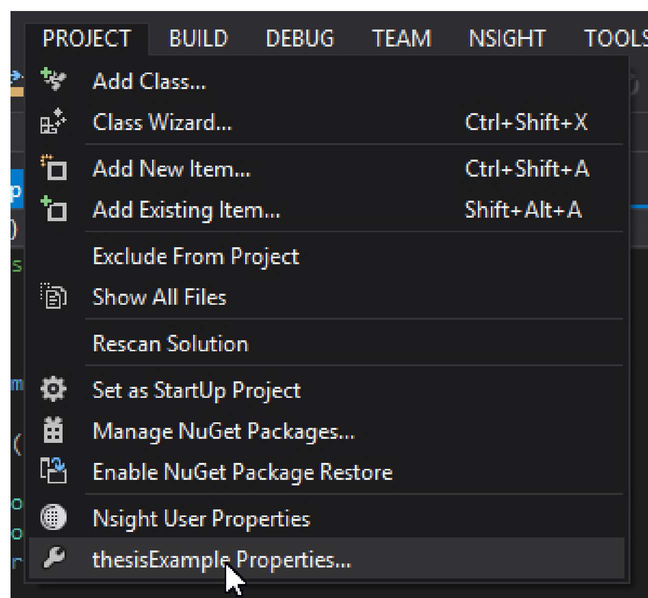


8. Agora é necessário que o projeto compile e execute no modo *release*. Por isso, seleciona-se o modo *release*:



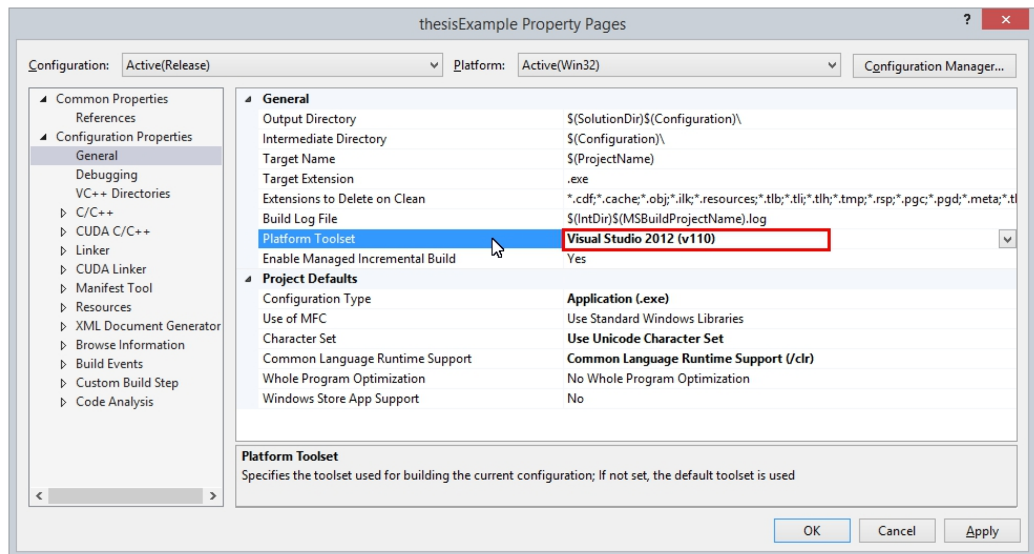
9. Para compilar e executar no modo *release*, após garantir que o modo *release* está selecionado, é necessária a seguinte configuração:

   (a) Ir ao menu *Project –> projName Properties...*:

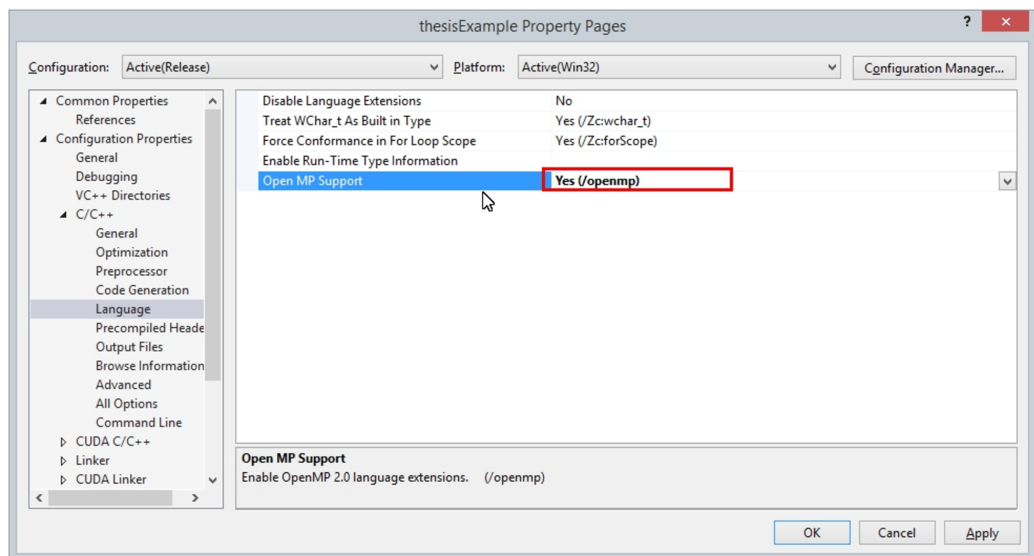


(b) Na *dialog box*, fazer as seguintes configurações:

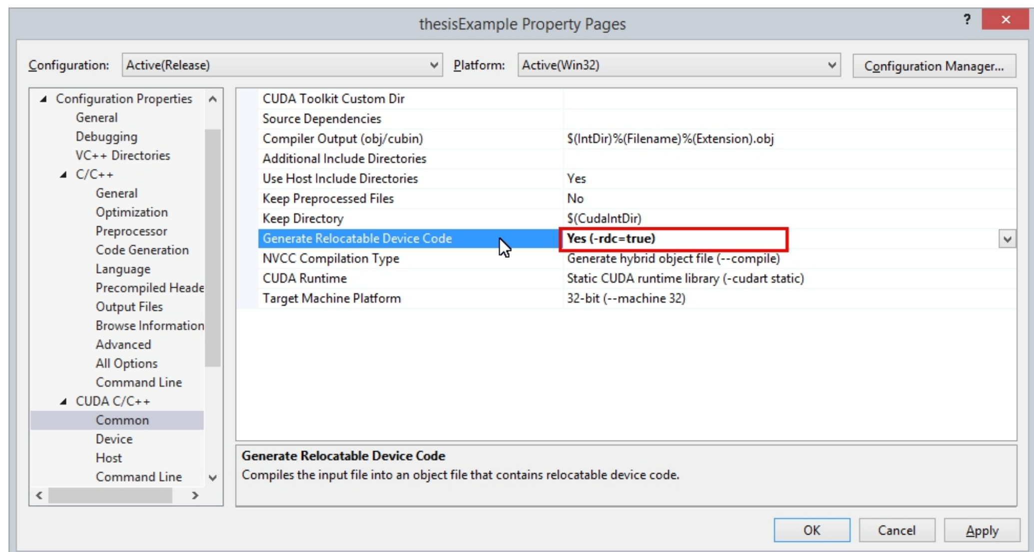
- Modificar o que está marcado com o retângulo vermelho:



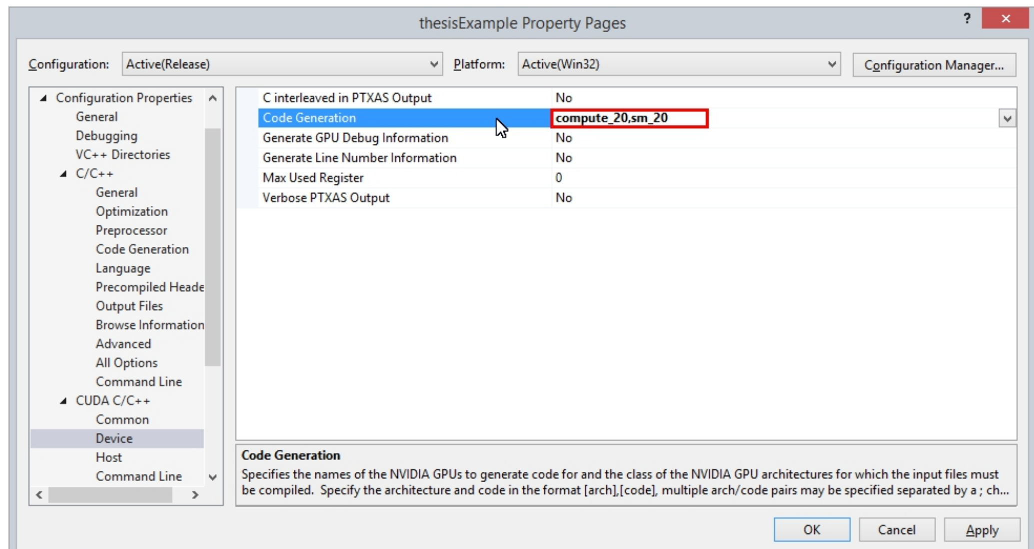
- No caso de se pretender compilar também com suporte para *OpenMP*, modificar o que está marcado com o retângulo vermelho:



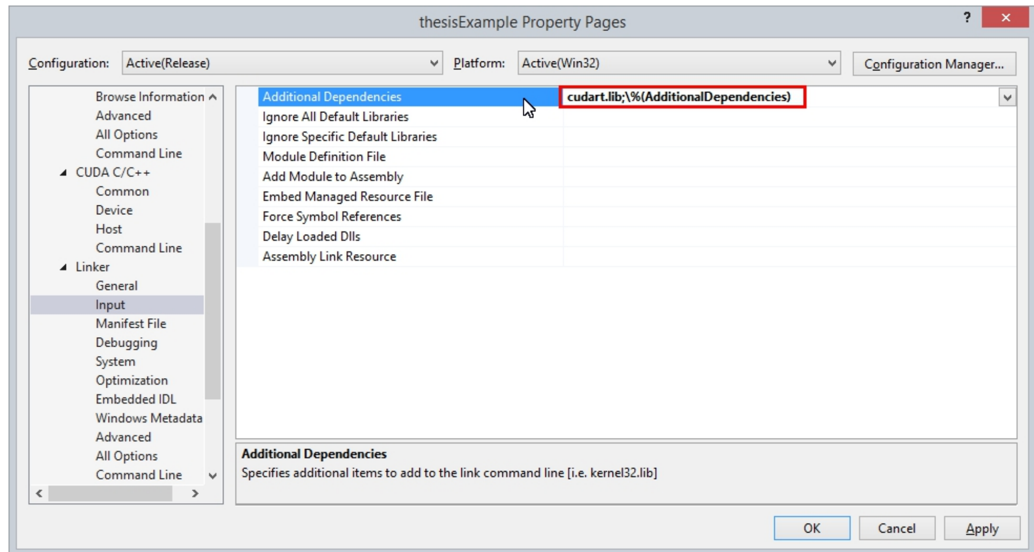
- Modificar o que está marcado com o retângulo vermelho:



- Modificar o que está marcado com o retângulo vermelho:

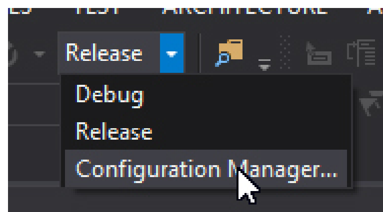


- Na opção marcada com o retângulo vermelho, escrever "*cudart.lib;%(AdditionalDependencies)*". Clicar em *Apply*, *OK* (apenas clicar em *OK* não é suficiente, porque as alterações são ignoradas):



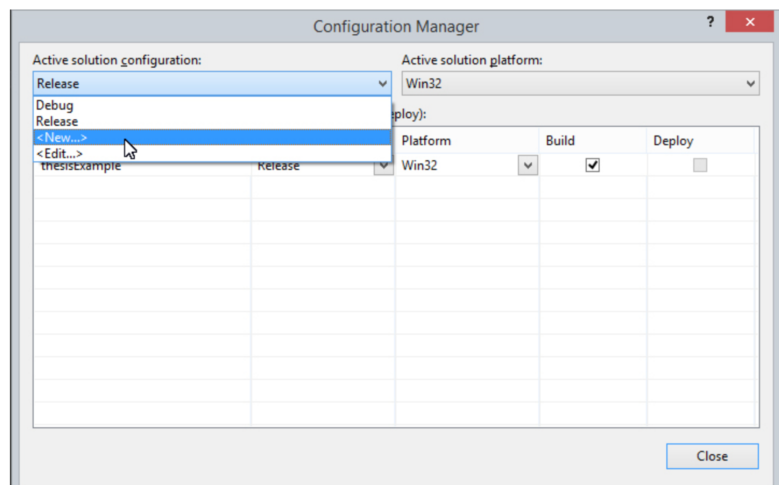
10. Para trabalhar com *big data*, é necessário ter o projeto compilado em *64 bit*. Para isso, são necessárias as seguintes configurações:

    (a) Com o modo *release* ativo, clicar em *Configuration Manager...*:

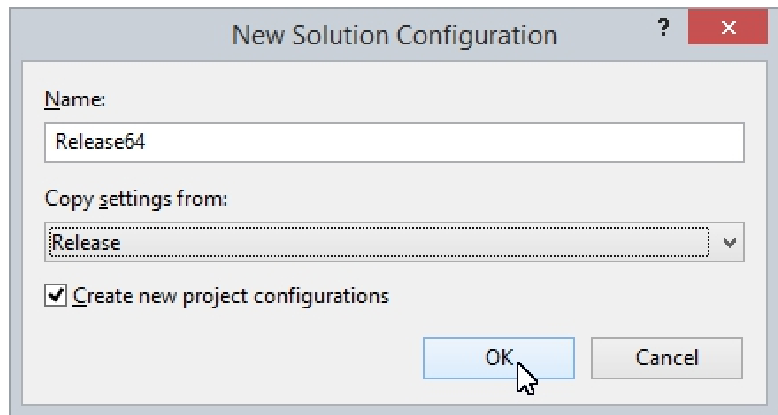


    (b) Na *dialog box* do *configuration manager*, fazer as seguintes configurações:

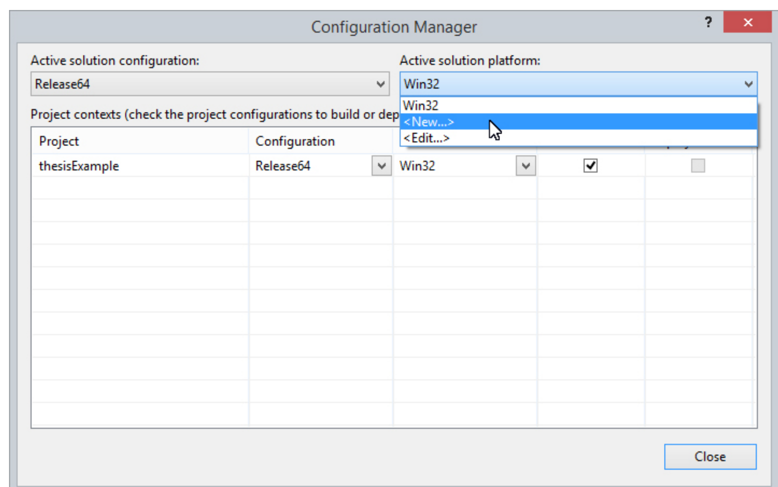
    - Em *Active Solution Configuration*, clicar em <*New...*>:



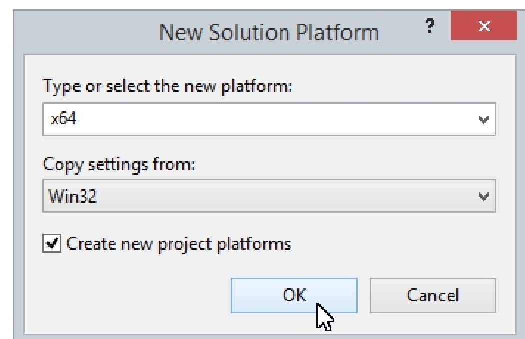
- Usar a seguinte configuração e clicar em *OK*:



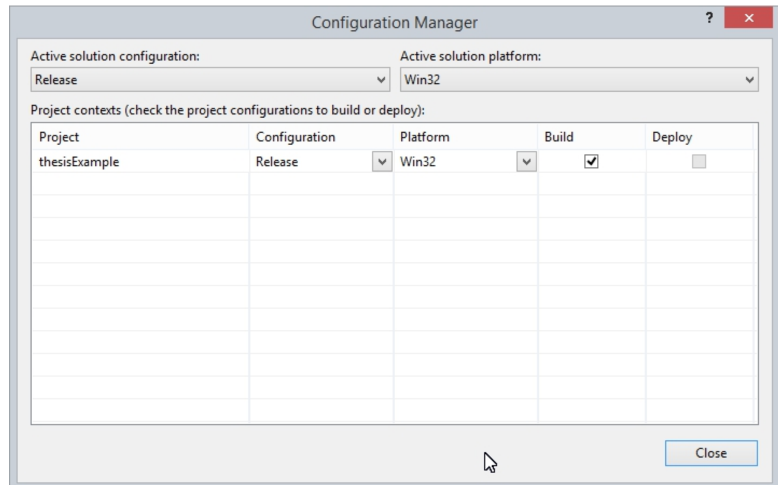
- Em *Active Solution Platform*, clicar em *<New...>*:



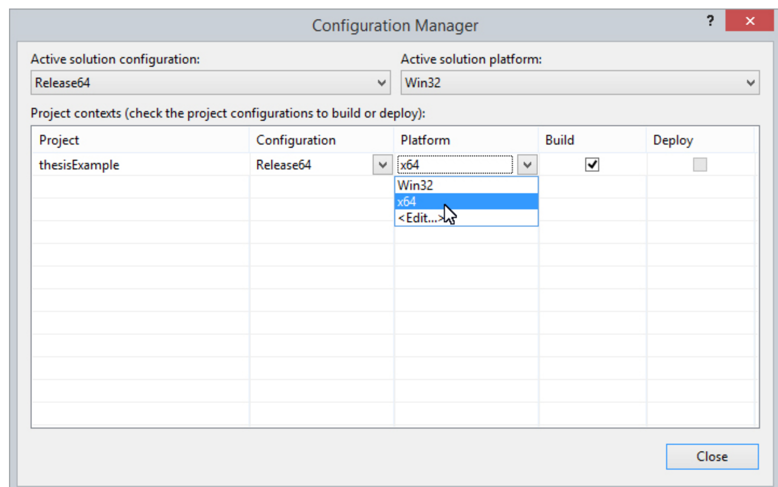
- Usar a seguinte configuração e clicar em *OK*:



- Em *Active Solution Configuration*, selecionar *Release* e garantir que *Active Solution Platform* esteja na opção *Win32*:

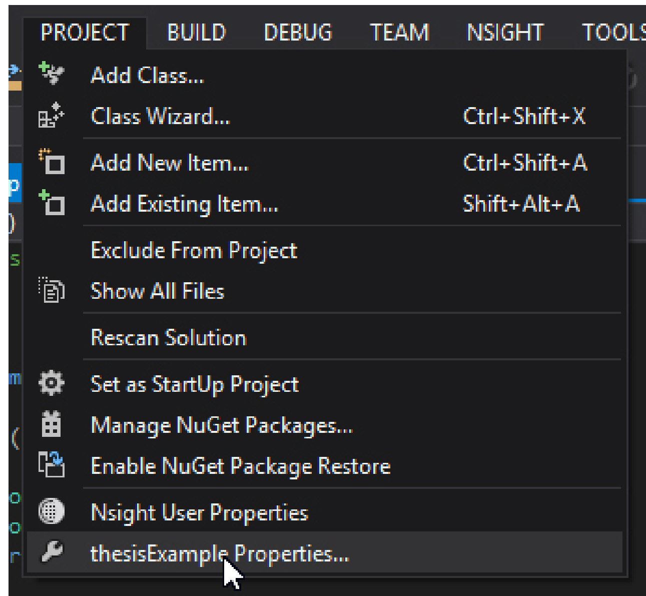


- Em *Active Solution Configuration*, selecionar *Release64* e garantir que *Platform* esteja na opção *x64*. Por fim, clicar em *Close*:



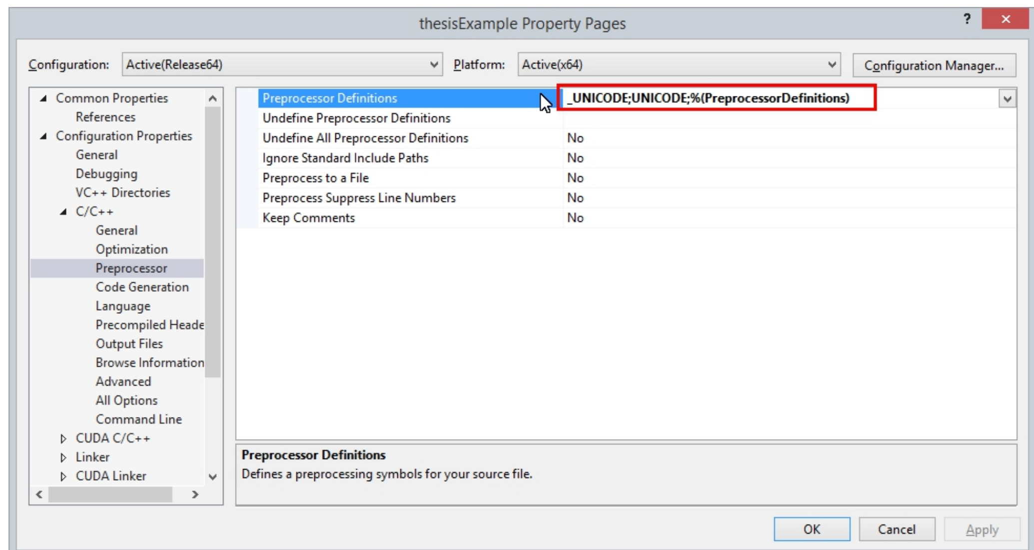
(c) Fechar o *Microsoft Visual Studio 2013*, voltar a abrir e carregar o projeto em que se estava a trabalhar.

(d) Ir ao menu *Project –> projName Properties...*:

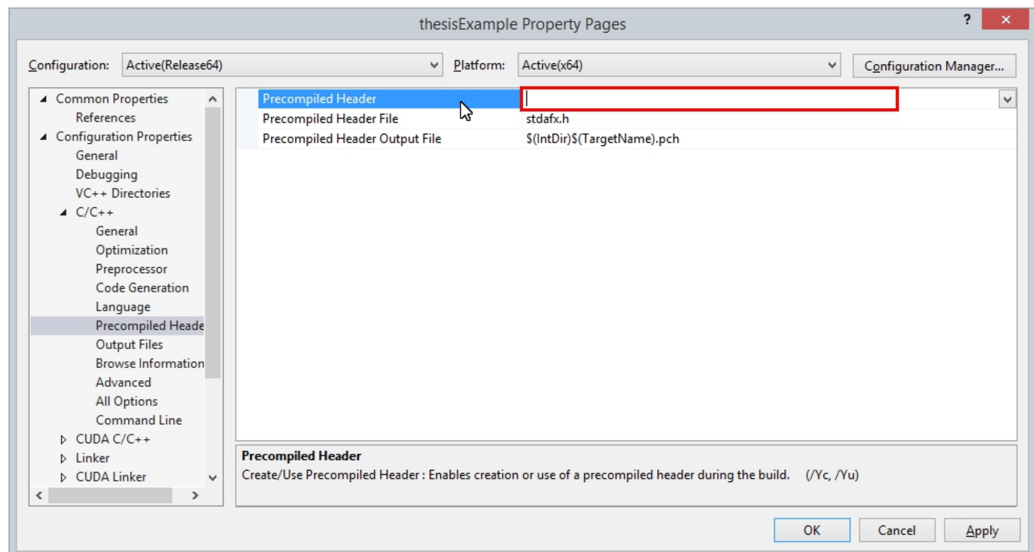


(e) Na *dialog box*, fazer as seguintes configurações:

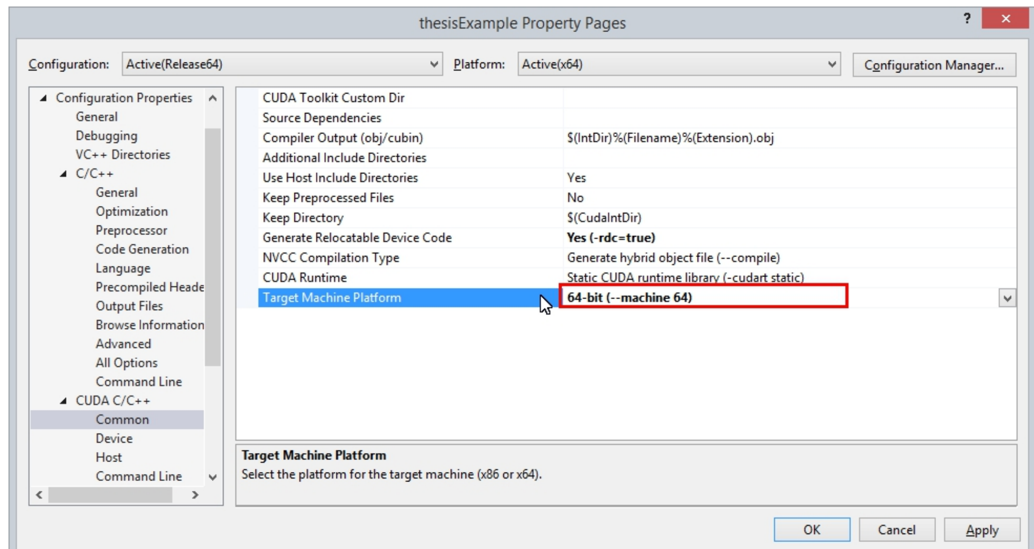
- Na opção marcada com o retângulo vermelho, escrever "*_UNICODE;UNICODE;%(PreprocessorDefinitions)*":



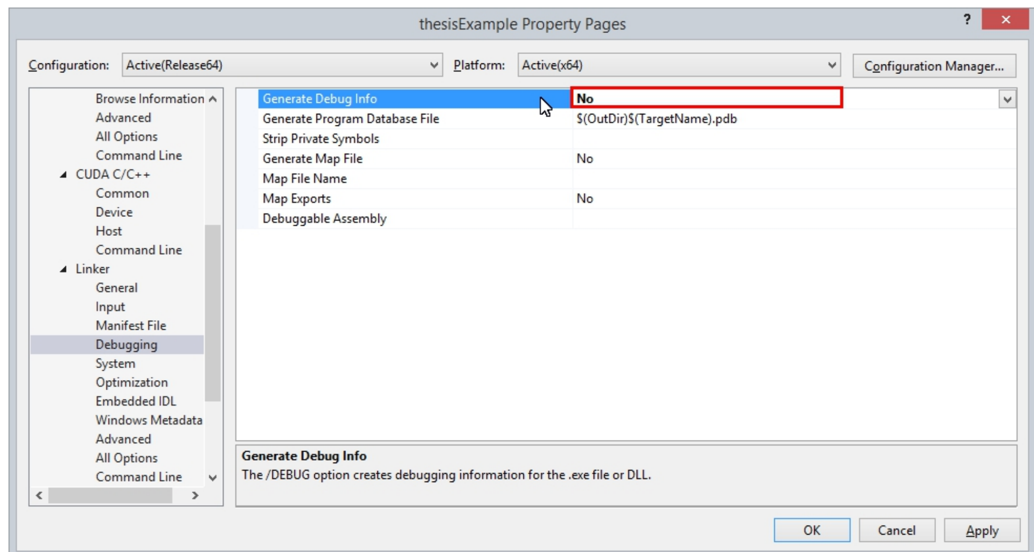
- Apagar o que estiver escrito dentro do retângulo vermelho:



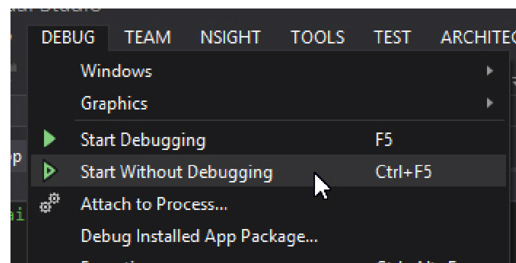
- Modificar o que está marcado com o retângulo vermelho:



- Modificar o que está marcado com o retângulo vermelho:



(f) Quando se pretende executar a aplicação com um conjunto de dados grande, deve-se assegurar que o modo *Release64* está ativo, e deve-se, também, executar o projeto da seguinte forma:



Este tipo de configuração serve para criar um projeto completo, para todas as plataformas, que seja capaz de ter funções em *C++* que consumam *CUDA C*. Garante, também, o funcionamento correto do *debugging*. O exemplo de código dado mostra como se pode fazer a passagem de dados entre plataformas.

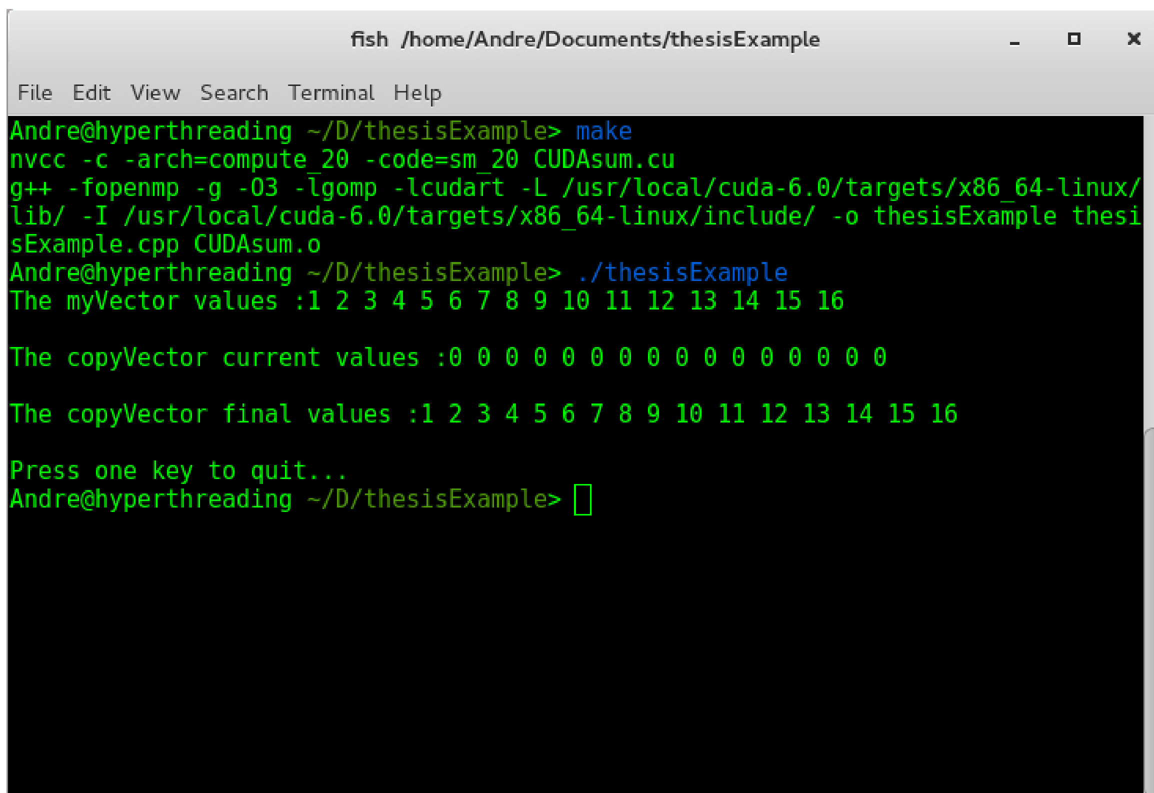
Assume-se que, primeiramente, todos os componentes foram instalados, tal como mencionado no Anexo A.2.

B.2 Configuração de um *makefile* para compilar *C++* e *CUDA* em *Linux*

O código apresentado no Anexo B.1 é compilável em *Linux*. Os ficheiros são *“thesisExample.cpp”* e *“CUDAsum.cu”*. Após remover no ficheiro *“thesisExample.cpp”* o *include*, “<#*include “stdafx.h”*>”, basta ter o seguinte *makefile*:

```
CXX=g++
nvcc=nvcc -c -arch=compute_20 -code=sm_20
CXXFLAGS=-fopenmp -g -O3 -lgomp -lcudart -L /usr/local/cuda-6.0/targets/x86_64-linux/lib/ -I
    /usr/local/cuda-6.0/targets/x86_64-linux/include/
VERSION=1.4

all: thesisExample

thesisExample: CUDAsum.o
        ${CXX} ${CXXFLAGS} -o thesisExample thesisExample.cpp CUDAsum.o

CUDAsum.o:
        ${nvcc} CUDAsum.cu
```

O resultado na consola é o seguinte:



Bibliografia

[1] H. Andrews and C. Patterson. Singular value decompositions and digital image processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 24(1):26–53, Feb 1976. Cited in page 8.

[2] Enkh-Amgalan Baatarjav, Santi Phithakkitnukoon, and Ram Dantu. Group recommendation system for facebook. In *Proceedings of the OTM Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems*, OTM '08, pages 211–219, Berlin, Heidelberg, 2008. Springer-Verlag. Cited in page 5.

[3] Paul Benioff. Quantum mechanical hamiltonian models of turing machines. *Journal of Statistical Physics*, 29(3):515–546, 1982. Cited in page 33.

[4] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012. Cited in page 21.

[5] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems 20*, pages 161–168, 2008. Cited in page 21.

[6] Otto Bretscher. *Linear Algebra With Applications*. Pearson Education, Boston, 2013. Cited in page 8.

[7] Robin Burke. Knowledge-based recommender systems. In *Encyclopedia of Library and Information Systems*, page 2000. Marcel Dekker, 2000. Cited in page 7.

[8] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, November 2002. Cited in page 7.

[9] Robin Burke. The adaptive web. In Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *Lecture Notes In Computer Science, Vol. 4321.*, chapter Hybrid Web Recommender Systems, pages 377–408. Springer-Verlag, Berlin, Heidelberg, 2007. Cited in page 5.

[10] D.R. Butenhof. *Programming with POSIX Threads*. Pearson Education, 1993. Cited in page 37.

[11] T. Chai and R. R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? *Geoscientific Model Development Discussions*, 7(1):1525–1534, 2014. Cited in page 29.

[12] Thanarat H. Chalidabhongse and Chayaporn Kaensar;. A Personalized Stock Recommendation System using Adaptive User Modeling. In *International Symposium on Communications and Information Technologies. ISCIT '06*, pages 463–468, October 2006. Cited in page 6.

[13] R. Chandra. *Parallel Programming in OpenMP*. High performance computing. Morgan Kaufmann Publishers, 2001. Cited in page 37.

[14] IBM Corporation. Hardware deep dive. In *IBM Power Systems*, 2013. Consultado em 20/06/2014 http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf. Cited in page 47.

[15] IBM Corporation. Power8. In *Power Roadmap*, 2013. Consultado em 20/06/2014 https://www-950.ibm.com/events/wwe/grp/grp030.nsf/vLookupPDFs/Tour%20P8%20Charts/$file/Tour%20P8%20Charts.pdf. Cited in page 47.

[16] Dan Cosley, Shyong K. Lam, Istvan Albert, Joseph A. Konstan, and John Riedl. Is seeing believing?: How recommender system interfaces affect users' opinions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 585–592, New York, NY, USA, 2003. ACM. Cited in page 7.

[17] Paolo Cremonesi, Roberto Turrin, Eugenio Lentini, and Matteo Matteucci. An evaluation methodology for collaborative recommender systems. In *Proceedings of the 2008 International Conference on Automated Solutions for Cross Media Content and Multi-channel Distribution*, AXMEDIS '08, pages 224–231, Washington, DC, USA, 2008. IEEE Computer Society. Cited in pages 29, 30, and 31.

[18] Michael D. Ekstrand, John T. Riedl, and Joseph A. Konstan. Collaborative filtering recommender systems. *Found. Trends Hum.-Comput. Interact.*, 4(2):81–173, February 2011. Cited in pages 29 and 30.

[19] Daniel M. Fleder and Kartik Hosanagar. Recommender systems and their impact on sales diversity. In *Proceedings of the 8th ACM Conference on Electronic Commerce*, EC '07, pages 192–199, New York, NY, USA, 2007. ACM. Cited in page 6.

[20] Denis Foley. Nvlink, pascal and stacked memory: Feeding the appetite for big data. In *Parallel Forall*, 2014. Consultado em 20/06/2014 http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/. Cited in pages 47 and 48.

[21] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM. Cited in pages xxiii, 21, and 52.

[22] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 625–628, Washington, DC, USA, 2005. IEEE Computer Society. Cited in page 7.

[23] Vladimir Getov, Paul Gray, and Vaidy Sunderam. Mpi and java-mpi: Contrasts and comparisons of low-level communication performance. In *In Supercomputing*, 1999. Cited in pages 35 and 36.

[24] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992. Cited in page 1.

[25] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, July 2001. Cited in page 30.

[26] A. Grama. *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003. Cited in pages 34, 35, 36, 37, and 49.

[27] Jianming He. *A Social Network-based Recommender System*. PhD thesis, UCLA, Los Angeles, CA, USA, 2010. AAI3437557. Cited in page 5.

[28] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, January 2004. Cited in page 7.

[29] R. Hochberg. Matrix multiplication with cuda-a basic introduction to the cuda programming model. *Shodor*, 2012. Cited in pages 41, 43, and 47.

[30] Cho-Jui Hsieh and Inderjit S. Dhillon. Fast coordinate descent methods with variable selection for non-negative matrix factorization. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1064–1072, New York, NY, USA, 2011. ACM. Cited in page 23.

[31] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 263–272, Washington, DC, USA, 2008. IEEE Computer Society. Cited in page 13.

[32] Dietmar Jannach. Techniques for fast query relaxation in content-based recommender systems. In *Proceedings of the 29th Annual German Conference on Artificial Intelligence*, KI'06, pages 49–63, Berlin, Heidelberg, 2007. Springer-Verlag. Cited in page 5.

[33] Chayaporn Kaensar and Thanarat Chalidabhongse. An adaptive recommendation system for personalized stock trading advice using artificial neural networks. *ICCAS*, pages 931–934, 2005. Cited in page 6.

[34] Bart P. Knijnenburg and Martijn C. Willemsen. Understanding the effect of adaptive preference elicitation methods on user satisfaction of a recommender system. In *Proceedings of the Third ACM Conference on Recommender Systems*, RecSys '09, pages 381–384, New York, NY, USA, 2009. ACM. Cited in page 6.

[35] Yehuda Koren and Robert Bell. Advances in collaborative filtering. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 145–186. Springer US, 2011. Cited in page 1.

[36] A Krishnamoorthy and D. Menon. Matrix inversion using cholesky decomposition. In *Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA), 2013*, pages 70–72, Sept 2013. Cited in page 64.

[37] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5(3):183–191, July 1961. Cited in page 33.

[38] S. Lloyd. Ultimate physical limits to computation. *Nature*, 406:1047–1054, 2000. Cited in page 33.

[39] Tariq Mahmood and Francesco Ricci. Towards learning user-adaptive state models in a conversational recommender system. In Alexander Hinneburg, editor, *LWA*, pages 373–378. Martin-Luther-University Halle-Wittenberg, 2007. Cited in page 7.

[40] Tariq Mahmood and Francesco Ricci. Improving recommender systems with adaptive conversational strategies. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, HT '09, pages 73–82, New York, NY, USA, 2009. ACM. Cited in pages 5 and 6.

[41] Matthew R. McLaughlin and Jonathan L. Herlocker. A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 329–336, New York, NY, USA, 2004. ACM. Cited in page 30.

[42] Frank McSherry and Ilya Mironov. Differentially private recommender systems: Building privacy into the net. In *Proceedings of the 15th ACM SIGKDD International Conference*

on *Knowledge Discovery and Data Mining*, KDD '09, pages 627–636, New York, NY, USA, 2009. ACM. Cited in page 6.

[43] Carl D. Meyer, editor. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. Cited in pages 13 and 18.

[44] George Millington. Nvidia launches world's first high-speed gpu interconnect, helping pave the way to exascale computing. In *NEWS RELEASE*, 2014. Consultado em 20/06/2014 http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx. Cited in pages 47 and 48.

[45] A. Munshi, B. Gaster, T.G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL. Pearson Education, 2011. Cited in page 41.

[46] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *In NIPS*, 2011. Cited in pages xxiii, 52, and 53.

[47] Pentti Paatero and Unto Tapper. Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5(2):111–126, 1994. Cited in page 51.

[48] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997. Cited in page 36.

[49] Donald Paul. More details of nvidia tesla cgpu k20 ”gk110”. In *Hardware News*, 2012. Consultado em 30/05/2014 http://technewspedia.com/more-details-of-nvidia-tesla-cgpu-k20-gk110/. Cited in page 42.

[50] Michael J. Pazzani and Daniel Billsus. The adaptive web. In Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *Lecture Notes In Computer Science, Vol. 4321.*, chapter Content-based Recommendation Systems, pages 325–341. Springer-Verlag, Berlin, Heidelberg, 2007. Cited in page 7.

[51] http://www.pcisig.com/specifications/pciexpress/resources/ PCI-SIG. Pci express® 3.0. In *Frequently Asked Questions*, 2014. Consultado em 20/06/2014 http://www.pcisig.com/specifications/pciexpress/resources/ PCIe_3.0_External_FAQ_Nereus_9.20.pdf. Cited in page 47.

[52] István Pilászy and Domonkos Tikk. Computational complexity reduction for factorization-based collaborative filtering algorithms. In Tommaso Noia and Francesco Buccafurri, editors, *E-Commerce and Web Technologies*, volume 5692 of *Lecture Notes in Computer Science*, pages 229–239. Springer Berlin Heidelberg, 2009. Cited in page 28.

[53] Lara Quijano-Sanchez, Juan A. Recio-Garcia, Belen Diaz-Agudo, and Guillermo Jimenez-Diaz. Social factors in group recommender systems. *ACM Trans. Intell. Syst. Technol.*, 4(1):8:1–8:30, February 2013. Cited in page 6.

[54] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, March 1997. Cited in pages 1 and 5.

[55] Shaghayegh Sahebi and William Cohen. Community-based recommendations: a solution to the cold start problem. In *Workshop on Recommender Systems and the Social Web (RSWEB), held in conjunction with ACM RecSys?11*, October 2011. Cited in page 7.

[56] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. Cited in pages 41, 43, and 47.

[57] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John T. Riedl. Application of dimensionality reduction in recommender system – a case study. In *IN ACM WEBKDD WORKSHOP*, 2000. Cited in pages 1, 8, 11, and 12.

[58] J. Ben Schafer, Joseph Konstan, and John Riedl. Recommender systems in e-commerce. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 158–166, New York, NY, USA, 1999. ACM. Cited in page 6.

[59] Guy Shani and Asela Gunawardana. Evaluating recommendation systems. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 257–297. Springer US, 2011. Cited in pages 29 and 31.

[60] Seung Woo Shin, Graeme Smith, John A. Smolin, and Umesh Vazirani. How ”quantum”is the d-wave machine?, 2014. Cited in page 33.

[61] Daniel R. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26:116–123, 1994. Cited in page 33.

[62] Ryan Smith. Gk110: The gpu behind tesla k20. In *AnandTech*, 2012. Consultado em 30/05/2014 http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last/3. Cited in page 42.

[63] Heeyoung Son and Seong-Ae Park. Sk hynix developed the world's first highest density 128gb ddr4 module. In *Press Release*, 2014. Consultado em 20/06/2014 http://www.skhynix.com/en/pr_room/news-data-view.jsp?search.seq=2329&search.gubun=0014. Cited in page 47.

[64] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic scheduling on gpus. *ACM Trans. Graph.*, 31(6):161:1–161:11, November 2012. Cited in page 59.

[65] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Matrix factorization and neighbor based algorithms for the netflix prize problem. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 267–274, New York, NY, USA, 2008. ACM. Cited in page 7.

[66] Gábor Takács and Domonkos Tikk. Alternating least squares for personalized ranking. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 83–90, New York, NY, USA, 2012. ACM. Cited in pages xxiii, 2, 18, and 59.

[67] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson International Edition. Pearson Addison Wesley, 2006. Cited in page 7.

[68] Luis Torgo and Rita Ribeiro. Precision and recall for regression. In *Proceedings of the 12th International Conference on Discovery Science*, DS '09, pages 332–346, Berlin, Heidelberg, 2009. Springer-Verlag. Cited in page 30.

[69] Charles F. Van Loan. The ubiquitous kronecker product. *J. Comput. Appl. Math.*, 123(1-2):85–100, November 2000. Cited in page 26.

[70] Mark van Setten, Sean M. McNee, and Joseph A. Konstan. Beyond personalization: The next stage of recommender systems research. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pages 8–8, New York, NY, USA, 2005. ACM. Cited in page 7.

[71] Yuanyuan Wang, Stephen Chi-Fai Chan, and Grace Ngai. Applicability of demographic recommender system to tourist attractions: A case study on trip advisor. In *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*, WI-IAT '12, pages 97–101, Washington, DC, USA, 2012. IEEE Computer Society. Cited in page 7.

[72] Brad Williams and Tracy Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking &Amp; Computing*, MobiHoc '02, pages 194–205, New York, NY, USA, 2002. ACM. Cited in page 6.

[73] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. Cited in pages 41, 43, and 47.

[74] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. Cited in page 29.

[75] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and InderjitS. Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, pages 1–27, 2013. Cited in pages xxiii, 2, 12, 13, 18, 21, 23, 25, 27, 28, 52, 53, 54, 57, 58, 59, 84, 93, and 94.

[76] D. Zachariah, M. Sundin, M. Jansson, and S. Chatterjee. Alternating least-squares for low-rank matrix reconstruction. *Signal Processing Letters, IEEE*, 19(4):231–234, April 2012. Cited in pages 2, 18, and 59.

[77] Gao Zhanchun and Liang Yuying. Improving the collaborative filtering recommender system by using gpu. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, pages 330–333, Oct 2012. Cited in page 2.

[78] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proc. 4th Int'l Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008. Cited in pages xxiii, 29, 51, and 59.

[79] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 249–256, New York, NY, USA, 2013. ACM. Cited in pages xxiii, 52, and 53.

[80] Martin A. Zinkevich, Alex Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, pages 2595–2603, 2010. Cited in pages 2 and 52.