AGRUPAMENTO VIA SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA CUDA

Marcelo Signorelli Mendes

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

Orientadores: Adilson Elias Xavier
Sergio Barbosa Villas-Boas

Rio de Janeiro
Setembro de 2012

AGRUPAMENTO VIA SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA CUDA

Marcelo Signorelli Mendes

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

Examinada por:

Prof. Adilson Elias Xavier, D.Sc.

Prof. Sergio Barbosa Villas-Boas, Ph.D.

Prof. Ricardo Farias, Ph.D.

Prof. Carmen Lucia Tancredo Borges, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

SETEMBRO DE 2012

Mendes, Marcelo Signorelli

Agrupamento via Suavização Hiperbólica com Arquitetura CUDA/Marcelo Signorelli Mendes. – Rio de Janeiro: UFRJ/COPPE, 2012.

XIII, 64 p.: il.; 29, 7cm.

Orientadores: Adilson Elias Xavier

Sergio Barbosa Villas-Boas

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 64.

1. Suavização. 2. Clusterização. 3. Paralelismo. I. Xavier, Adilson Elias *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Em memória de meu pai,

Ronaldo de Souza Mendes

Agradecimentos

Agradeço a minha mãe Maria Helena, meu irmão Flavio e minha tia Sonia por tudo que fizeram ao longo de todos esses anos, sempre me incentivando a alcançar meus objetivos.

A meu orientador e amigo Adilson Elias Xavier, pelos conhecimentos transmitidos, pela generosidade e pela oportunidade de concluir esse trabalho.

Ao professor e co-orientador Sergio Barbosa Villas-Boas, por toda a experiência passada e pelos conselhos que me ajudaram muito ao longo de meu mestrado.

A minha namorada Priscila dos Santos Abonante, pelo apoio, carinho e paciência.

Aos meus amigos do Controlab, Henrique Serdeira, Julio Tadeu e ao Professor Ernesto Lopes.

Aos meus amigos de LABOTIM, Jesus Ossian e Renan Vicente.

As pessoas que não estão mais entre nós mas que contribuíram de forma extraordinária no meu desenvolvimento pessoal e profissional. Em especial minhas avós Mizzi de Souza Mendes e Dina Olivieri Signorelli, minha antiga orientadora, Professora Eliana Prado Lopes Aude e meu pai, Ronaldo de Souza Mendes, a quem dedico integralmente esse trabalho.

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

AGRUPAMENTO VIA SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA CUDA

Marcelo Signorelli Mendes

Setembro/2012

Orientadores: Adilson Elias Xavier

Sergio Barbosa Villas-Boas

Programa: Engenharia de Sistemas e Computação

A modelagem matemática para a formulação de problemas de agrupamento segundo o critério de mínima soma de quadrados produz um problema do tipo min-sum-min com a característica de ser fortemente não diferenciável. Utilizando-se a técnica de suavização hiperbólica, é possível transformar o problema original em uma sequência de problemas diferenciáveis de minimização irrestrita com dimensão menor que o problema inicial. Este método é conhecido como Xavier Clustering Method (XCM). Neste trabalho, é introduzida uma nova implementação para o método, denominada XCM-GPU, baseada na arquitetura Nvidia CUDA, para paralelizar partes do algoritmo na unidade de processamento gráfico (GPU) do computador. Para validar esta nova implementação, os resultados do XCM original e do XCM-GPU foram comparados e foi possível verificar que, sem perda alguma de precisão, é possível atingir tempos de execução até 13 vezes menores na versão paralela, demonstrando que, para grandes conjuntos de dados, os benefícios dessa implementação são aparentes.

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

HYPERBOLIC SMOOTHING CLUSTERING WITH CUDA ARCHITECTURE

Marcelo Signorelli Mendes

September/2012

Advisors: Adilson Elias Xavier

Sergio Barbosa Villas-Boas

Department: Systems Engineering and Computer Science

The mathematical modelling for the formulation of clustering problems according to the criterion of minimum sum of squares leads to a min-min-sum like problem with the characteristic of being strongly non-differentiable. Using the technique of hyperbolic smoothing, it is possible to transform the original problem into a sequence of differentiable unconstrained minimization problems with lower dimension than the initial problem. This method is known as Xavier Clustering Method (XCM). In this work it is introduced a new implementation for the method, called XCM-GPU, based on Nvidia CUDA architecture to parallelize portions of the algorithm in the computer's graphics processing unit (GPU). To validate this new implementation, the results of original XCM and XCM-GPU were compared and it was verified that, without any loss of accuracy, it is possible to achieve run times up to 13 times smaller in the parallel version which shows that, for large data sets, the benefits of this implementation are clear.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

A constante busca por uma crescente eficiência de processamento computacional vem produzindo novas tecnologias que, quando aplicadas a problemas de diversos tipos, promovem soluções de igual ou superior qualidade com custos reduzidos, seja em tempo de processamento ou até mesmo no consumo de energia necessário. Com o conceito de programação paralela não foi diferente. Já há algum tempo é possível criar programas (mais eficientes que seus equivalentes sequenciais) baseados nesse modelo, mas até o momento não era comum um programa paralelo possuir, por exemplo, uma centena de unidades de processamento (threads) em um mesmo processador.

Em fevereiro de 2007 foi lançado pela empresa Nvidia a primeira versão do SDK (Software Development Kit) CUDA, um acrônimo para *Compute Unified Device Architecture*. Este kit de desenvolvimento permite que seja possível desenvolver códigos que são executados em paralelo na unidade de processamento gráfico do computador, ou GPU na sigla em inglês.

Até então, a GPU era um processador especializado para atender as demandas de tempo real para processamento de vídeo e gráficos em 3 dimensões. Com o lançamento deste kit de desenvolvimento de software, o conceito de GPU evoluiu para GPGPU, uma abreviação de *General-Purpose Graphics Processor Unit*. Em uma GPGPU é possível realizar cálculos de aplicações que seriam tradicionalmente feitos na CPU em novo modelo, com características de processamento altamente

paralelo, multi-thread, multi-core (ou seja, com muitos núcleos de processamento) com altíssimo poder computacional e largura de banda para acesso a memória tremendamente alta [2].

A Figura 1.1, extraída de [2], mostra a evolução da capacidade computacional em GFLOPS (Giga Floating Points Operations per Second) de GPGPUS (em verde) comparado-a com a mesma medida de CPU's convencionais, em azul (1 GFLOP = 10^9 FLOPS) que ilustra muito bem a capacidade computacional dessa arquitetura. Utilizando-se dessa tecnologia, é possível desenvolver programas que usam centenas ou até mesmo milhares de threads que são executadas de forma paralela em uma ou mais GPGPU's presentes no computador.

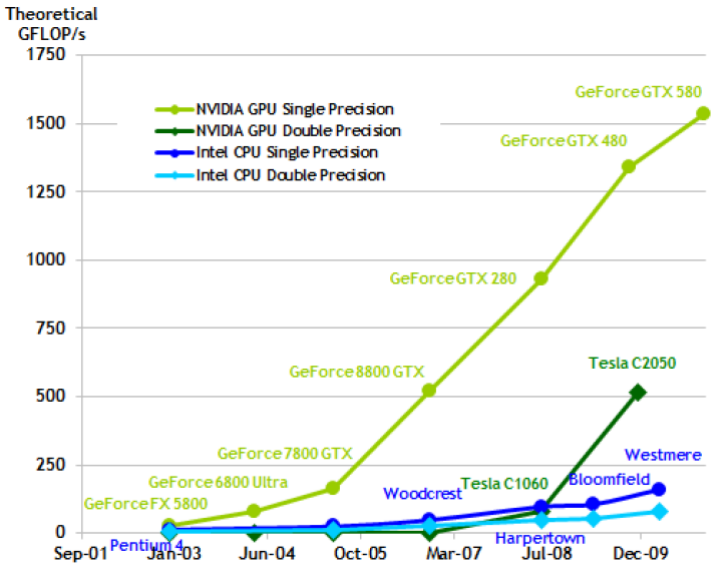


Figura 1.1: Comparação de GFLOPS entre GPGPU e CPU.

O principal objetivo deste trabalho é aplicar esta nova arquitetura à resolução de problemas de Análise de Agrupamento (Cluster Analysis) formulados segundo o critério de mínimas somas de quadrados, identificando os principais pontos do algoritmo onde é possível fazer uso do conceito de programação massivamente paralela para obter mais eficiência nos cálculos.

Um problema de agrupamento consiste basicamente em, dado um conjunto de m observações em um espaço multidimensional e q conjuntos (ou clusters), definir quais observações pertencerão a um determinado conjunto seguindo algumas regras

previamente definidas.

A formulação deste problema de agrupamento como uma mínima soma de quadrados produz um problema de Otimização Global não-diferenciável da classe NP-Hard [1]. No entanto, foi proposto por [7] uma forma alternativa de resolução desse tipo de problema aplicando-se uma técnica conhecida como *Suavização Hiperbólica*. Com este conceito, o problema original pode ser transformado para ser resolvido em uma série de problemas diferenciáveis irrestritos com dimensão relativamente baixa, tirando-se total proveito de métodos de minimização aplicáveis a problemas desse tipo.

Desta forma, surgiu o *Hyperbolic Smoothing Clustering Method*, ou *Xavier Clustering Method*, doravante denominado XCM. Os resultados computacionais do XCM, apresentados em [6] e [9], mostram que o método é extremamente eficiente, robusto e preciso, em alguns casos até mesmo encontrando soluções melhores que as encontradas na literatura. Na implementação previamente introduzida, as minimizações irrestritas foram calculadas por meio de um algoritmo *Quasi-Newton*, com fórmula de atualização BFGS. A principal vantagem dessa família de métodos é que o cálculo da matriz hessiana não é necessário, mas o cálculo do gradiente e do valor da função objetivo em si são. É exatamente neste ponto que reside a principal motivação para este trabalho: Desenvolver o XCM-GPU, que implementa o cálculo da função objetivo, em paralelo, em um ou mais GPGPU's disponíveis para aumentar a eficiência computacional da minimização e promover um maior ganho de performance no algoritmo. O cálculo do gradiente também é realizado de forma semelhante.

Os resultados computacionais mostram que, sem perda alguma de precisão, são obtidos ganhos consideráveis de performance.

Esta tese está organizada da seguinte forma: No capítulo 2 é apresentado um breve resumo do conceito de análise de agrupamento, assim como alguns conceitos sobre métodos de minimização irrestrita. Ainda no capítulo 2 são apresentados os detalhes da arquitetura CUDA. No capítulo 3 é o problema de agrupamento é definido sob a ótica da mínima soma de quadrados, e o XCM é apresentado. No capítulo

4 é introduzido o XCM-GPU, detalhando a forma como o método foi implementado. No capítulo 5 são apresentados os resultados computacionais, principalmente comparando-se as implementações sequencial e paralela do método e no capítulo 6 tem-se as conclusões sobre o trabalho.

Capítulo 2

Revisão Bibliográfica

Neste capítulo é feita uma breve revisão sobre os principais temas empregados neste trabalho. Primeiramente são abordados conceitos de análise de agrupamento e em seguida são apresentados alguns conceitos sobre algoritmos de minimização. Por último, a arquitetura CUDA é detalhada os principais conceitos necessários ao seu entendimento são descritos.

2.1 Penalização Hiperbólica

A penalização hiperbólica (*hyperbolic penalty*) é uma técnica de processamento numérico com várias aplicações, tais como clustering [9][11], covering[10], packing, hub-location e outras. Essa técnica foi originalmente proposta por Adilson Xavier. O mais antigo documento escrito que menciona "penalização hiperbólica" é a tese de mestrado de Adilson Xavier [7], em março de 1982, cuja imagem de folha de rosto é mostrada na figura 2.1.

Em novembro de 1982, o mesmo tema foi publicado na conferência conjunta "XV Simpósio Brasileiro de Pesquisa Operacional" e "I Congresso Latino-Americano de Pesquisa Operacional e Engenharia de Sistemas". A figura 2.2 mostra essa publicação.

O mais antigo documento em inglês mencionando "hyperbolic penalty" é o artigo [8], em 2001. A figura 2.3 mostra essa publicação.

PENALIZAÇÃO HIPERBÓLICA: UM NOVO MÉTODO
PARA RESOLUÇÃO DE PROBLEMAS DE OTIMIZAÇÃO

Adilson Elias Xavier

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.).

Aprovada por:

João Ricardo Rodrigues Hermes de Araújo
(Presidente)

Antonio Alberto Fernandes de Oliveira

Luiz Fernando Loureiro Legey

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 1982

Figura 2.1: Capa da tese de mestrado de Adilson Xavier, em 1982, o mais antigo documento mencionando "penalização hiperbólica"

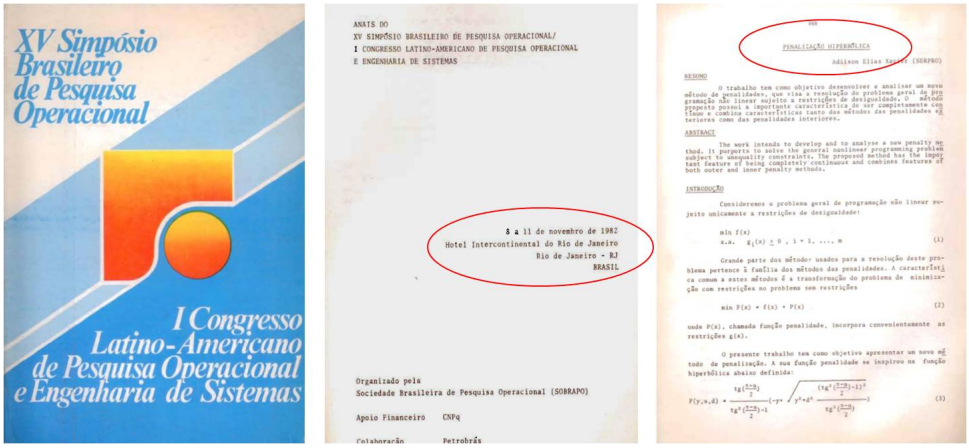
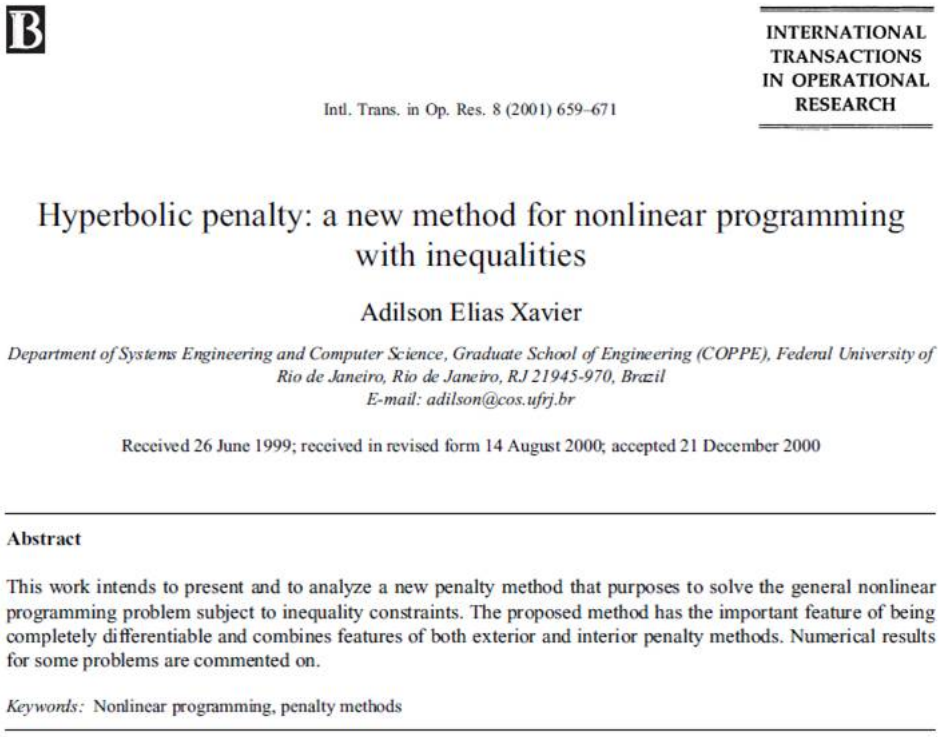


Figura 2.2: "penalização hiperbólica" publicada em congresso



Intl. Trans. in Op. Res. 8 (2001) 659–671

INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Hyperbolic penalty: a new method for nonlinear programming with inequalities

Adilson Elias Xavier

Department of Systems Engineering and Computer Science, Graduate School of Engineering (COPPE), Federal University of Rio de Janeiro, Rio de Janeiro, RJ 21945-970, Brazil
E-mail: adilson@cos.ufrj.br

Received 26 June 1999; received in revised form 14 August 2000; accepted 21 December 2000

Abstract

This work intends to present and to analyze a new penalty method that purposes to solve the general nonlinear programming problem subject to inequality constraints. The proposed method has the important feature of being completely differentiable and combines features of both exterior and interior penalty methods. Numerical results for some problems are commented on.

Keywords: Nonlinear programming, penalty methods

Figura 2.3: Primeira publicação mencionando "hyperbolic penalty"

A partir da técnica original de penalização hiperbólica, foi proposto pelo mesmo autor Adilson Xavier a técnica de clusterização baseada em *suavização hiperbólica*.

A suavização hiperbólica é um conceito derivado da penalização hiperbólica e, quando aplicado a problemas de agrupamento, tem-se um método de clusterização que será chamado nesse trabalho de XCM (Xavier Clustering Method) [9], definido no capítulo 3.

2.2 Análise de Agrupamento

A Análise de Agrupamento (Cluster Analysis, ou Clustering) consiste basicamente em definir, dado um conjunto de observações, agrupamentos onde observações pertencentes a um mesmo grupo são mais similares entre si do que observações de outros grupos, segundo um critério específico pré-definido. Problemas de análise de agrupamento são encontrados nas mais diversas aplicações, como, por exemplo, processamento gráfico de imagens e até mesmo em outros ramos de ciência como

medicina e química.

Pode-se entender como observações um conjunto de pontos definidos em um espaço euclidiano de dimensão p. A primeira parte da figura 2.4 ilustra esse conceito, com 6 pontos definidos no $\mathbb{R}^2$. A segregação destas observações em um número q pré-definido de conjuntos, ou *clusters* é o principal objetivo da análise de agrupamento. Uma possível solução para este conjunto de observações com 3 clusters pode ser visto na segunda parte da Figura 2.4. É comum utilizar pontos como centros de gravidade, ou *centroids*, que definem o posicionamento de um cluster no espaço das observações. Esta ideia está ilustrada na parte 3 da Figura 2.4.

Dois principais objetivos devem ser levados em consideração: Homogeneidade e Separação. O objetivo da homogeneidade dita que observações em um mesmo grupo sejam similares entre si, enquanto que o objetivo da separação dita que observações em grupos distintos sejam o mais diferentes quanto possível. O critério definido *a priori* mais simples e intuitivo que pode ser considerado é o da minima soma de quadrados.

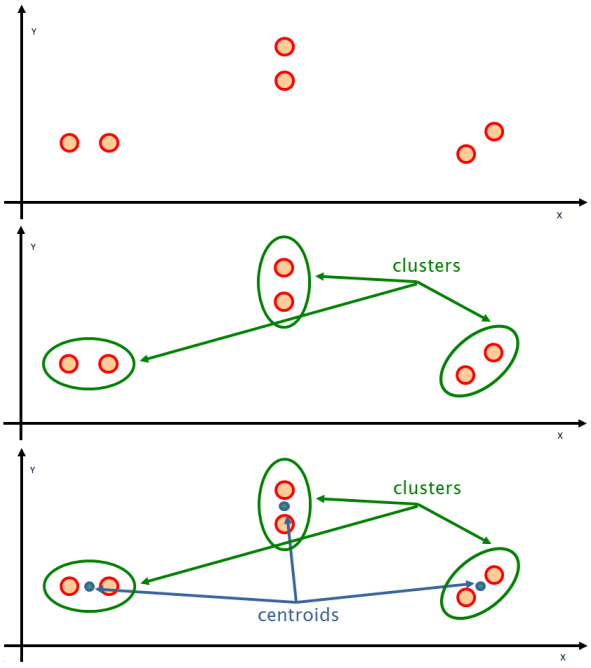


Figura 2.4: Exemplo de observações, clusters e centroids no $\mathbb{R}^2$.

No entanto, a modelagem matemática de um problema de clusterização com critério de mínima soma de quadrados leva à formulação de um problema NP-Hard

de otimização global não diferenciável e não convexo, com um grande número de minimizadores locais.

A ideia principal deste tipo de formulação é muito simples: Deseja-se encontrar o posicionamento de q pontos no espaço de dimensão p das m observações de forma que o somatório das menores distâncias euclidianas de cada uma das m observações a cada um dos q centroids seja o menor possível.

Para se alcançar esse objetivo, um posicionamento inicial para os q centroids é definido segundo um critério qualquer, por exemplo, considerando o ponto médio das m observações e uma dispersão baseada no desvio padrão somada a um número aleatório. Em seguida, para cada observação, é calculado o quadrado da distância euclidiana para todos os q centroids, tomando-se a menor dentre todas as q distâncias calculadas. O somatório dessas m menores distâncias é o valor da função objetivo o qual se deseja minimizar. Na próxima etapa da execução, a posição dos q centroids é modificada (o posicionamento das m observações sempre permanece fixo) e todas as menores distâncias são novamente calculadas. O processo é repetido até que não se consiga mais diminuir o somatório das menores distâncias.

É possível modificar a modelagem matemática do problema original, suavizando-o através de uma técnica conhecida como *Suavização Hiperbólica*. Os detalhes dessa transformação são apresentados no capítulo 3. Essa estratégia de resolução leva à criação de problemas diferenciáveis irrestritos, podendo-se tirar proveito de métodos de minimização irrestritos baseados na primeira derivada.

Existem outros estrategias para resolver problemas de agrupamento, como por exemplo, métodos de partição e métodos hierárquicos. Esses métodos e suas características não serão abordados neste trabalho.

2.3 Conceitos de Otimização

Otimização é uma ferramenta importante na ciência de decisão e na análise de sistemas físicos. Para usá-la, é preciso primeiro identificar algum objetivo, uma medida quantitativa do desempenho do sistema em estudo. O processo de identificação de

objetivos, variáveis e restrições para um dado problema é conhecido como modelagem. Quando um modelo for formulado, um algoritmo de otimização pode ser usado para encontrar a sua solução.

Não existe um algoritmo de otimização universal. Pelo contrário, existem inúmeros algoritmos, cada um dos quais é adaptado a um tipo particular de problema de otimização. Muitas vezes, é de responsabilidade do desenvolvedor do modelo escolher um algoritmo que é adequado para a sua aplicação específica.

Após um algoritmo de otimização ser aplicado ao modelo, deve-se ser capaz de reconhecer se obteve-se êxito na tarefa de encontrar uma solução. Em muitos casos, existem elegantes expressões matemáticas conhecidas como *condições de otimalidade* para a verificação se o atual conjunto de variáveis é de fato a solução do problema.

Outra característica sempre presente nos algoritmos de otimização é o fato de serem algoritmos iterativos. Eles começam com uma estimativa inicial dos valores ideais das variáveis e geram uma sequência de estimativas até chegar a uma solução. A estratégia utilizada para se deslocar de uma iteração para a próxima distingue um algoritmo de outro.

Além dessas características, usualmente os algoritmos de minimização devem possuir as seguintes propriedades:

- Robustez - Eles devem ter um bom desempenho em uma ampla variedade de problemas de sua classe, para escolhas razoáveis das variáveis iniciais.
- Eficiência - Eles não devem exigir muito tempo computacional ou espaço de armazenamento.
- Precisão - Devem ser capazes de identificar uma solução com precisão, sem ser excessivamente sensível a erros nos dados ou a erros aritméticos de arredondamento.

Esses objetivos podem ser conflitantes. Por exemplo, um método que converge rapidamente para a solução pode exigir uma capacidade de armazenamento muito

elevada. Por outro lado, um método robusto também pode ser mais lento. Comparações entre a taxa de convergência e requisitos de armazenamento, ou entre robustez e velocidade e assim por diante, são questões centrais na na resolução de problemas de otimização numérica.

Dentro do conceito de otimização, muitos outros temas devem ser considerados, tais como: Otimização restrita e irrestrita, otimização global e local e convexidade. Este estudo está focado em Otimização Irrestrita.

Na otimização irrestrita, procura-se minimizar uma função objetivo que depende de variáveis reais, sem restrições nos valores dessas variáveis. Geralmente, o objetivo máximo é encontrar um minimizador global da função, ou seja, o ponto onde a função atinge seu valor mínimo.

Pode parecer que a única maneira de descobrir se um ponto X^* é um mínimo local é examinar todos os pontos na sua vizinhança imediata, para se certificar de que nenhum deles tem um valor menor. No entanto, quando a função f é suave, existem maneiras muito mais eficientes e práticas de se identificar mínimos locais.

Em particular, se uma função objetivo f é duas vezes continuamente diferenciável, pode-se de dizer que X^* é um minimizador local, examinando apenas o gradiente $\nabla f(X^*)$ e a Hessiana $\nabla^2 f(X^*)$. A ferramenta matemática usada para estudar minimizadores de funções suaves é o teorema de Taylor.

Teorema de Taylor: Suponha que $f : \mathbb{R}^n \to \mathbb{R}$ é continuamente diferenciável e que $p \in \mathbb{R}^n$. Então

$$f(x+p) = f(x) + \nabla f(x+tp)^T p \tag{2.1}$$

para algum $t \in (0,1)$.

Além disso, se f é duas vezes continuamente diferenciável, então

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp)^T p \tag{2.2}$$

para algum $t \in (0,1)$.

Condições necessárias para otimalidade são obtidas assumindo que X^* é um minimizador local e, em seguida, provando fatos sobre $\nabla f(X^*)$ e $\nabla^2 f(X^*)$.

Condições Necessárias de Primeira Ordem: Se X^* é um minimizador local e f é continuamente diferenciável em uma vizinhança aberta de X^*, então $\nabla f(X^*) = 0$.

Condições Necessárias de Segunda Ordem: Se X^* é um minimizador local de f e $\nabla^2 f$ é contínua em uma vizinhança aberta de X^*, então $\nabla f(X^*) = 0$ e $\nabla^2 f(X^*)$ é semidefinida positiva.

Condições Suficientes de Segunda Ordem: Suponha que $\nabla^2 f$ é contínua em uma vizinhança aberta de X^* e que $\nabla f(X^*) = 0$ e $\nabla^2 f(X^*)$ é positiva definida. Então X^* é um minimizador local estrito de *f*.

Todos os algoritmos de minimização exigem um ponto de partida, que é geralmente denotado por x^0. Começando em x^0, os algoritmos geram uma sequência de iterações x^k que terminam quando alguma regra de parada estabelecida é satisfeita. Por exemplo, pode-se assumir uma regra de parada quando a diferença entre o valor da iteração x^k e x^{k-1} for menor que um determinado limite.

Ao decidir como passar de uma iteração x^k para a próxima, os algoritmos usam informações sobre o valor da função f em x^k e possivelmente informações de iterações anteriores $x^{k-1}, \ldots, x^1, x^0$. Estas informações são usadas para encontrar uma nova iteração x^{k+1} com um valor de f menor do que o valor de f em x^k.

Há duas estratégias fundamentais para passar do ponto x^k atual para uma nova iteração x^{k+1}. Na estratégia de *busca de linha*, o algoritmo escolhe uma direção p_k e busca ao longo desta direção, a partir do ponto x^k atual, um novo ponto com um valor de função menor. A distância que deve ser percorrida ao longo da direção p_k pode ser encontrada resolvendo-se o seguinte problema de minimização unidimensional para se encontrar uma medida α, definida como *comprimento de passo*:

$$minimizar\ f(x_k + \alpha p_k), \alpha > 0 \tag{2.3}$$

Ao encontrar uma solução exata para este problema, o maior benefício a partir da direção p_k terá sido obtido. No entanto, a minimização exata é computacionalmente cara e desnecessária. Em vez disso, algoritmos de busca em linha geram um número limitado de comprimentos de passo até que se encontre uma solução que se aproxima da solução ótima do problema.

No novo ponto, ou seja, na solução exata ou aproximada do problema 2.3, uma nova direção de pesquisa e um novo comprimento de passo são computados, e o processo é repetido. Resumidamente, a cada iteração de um método de busca em linha, uma direção de pesquisa p_k é calculada e então o algoritmo "decide"o quanto se mover ao longo dessa direção.

O sucesso de um método de busca em linha depende de escolhas efetivas tanto da direção de busca p_k quanto do comprimento de passo α_k. No cálculo do comprimento de passo α_k, existe uma questão de custo-benefício. O ideal é escolher α_k para se obter uma substancial redução no valor da função objetivo mas, ao mesmo tempo, não se deseja gastar muito tempo fazendo essa escolha.

Estratégias práticas para se implementar o método de busca em linha inexata (ou seja, quando a solução ótima do problema 2.3 não é encontrada) para se identificar um comprimento de passo que produza uma redução adequada no valor de f a um custo mínimo são normalmente empregadas. Uma condição de parada, conhecida como *Regra de Armijo* para uma busca em linha inexata usualmente utilizada estipula que α_k deve primeiramente fornecer uma redução suficiente na função objetivo f da seguinte forma:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \ 0 < c_1 < 1 \tag{2.4}$$

Outra condição de parada para busca de linha inexata são as chamadas *Condições Wolfe*:

$$
\begin{aligned}
& f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \\
& \nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k \\
& 0 < c_1 < c_2 < 1
\end{aligned} \tag{2.5}
$$

Na regra de Wolfe, além da imposição de uma redução no valor da função objetivo, também é imposta uma restrição ao novo valor do gradiente da função no ponto $x_k + \alpha_k p_k$.

Em outra estratégia algorítmica, conhecida como *Região de Confiança*, as informações adquiridas sobre a função objetivo f são usadas para construir um modelo m_k cujo comportamento próximo do ponto atual x_k é semelhante ao comportamento real de *f*.

Como o modelo m_k pode não ser uma boa aproximação de f quando $x*$ está longe de x_k, a ideia é restringir a procura por um minimizador local de m_k para alguma região em torno de x_k. Em outras palavras, encontra-se o passo de minimização candidato p resolvendo-se o subproblema :

$$
minimizar\ p\ em\ m_k(x_k + p) \tag{2.6}
$$

onde $x_k + p$ está dentro da região de confiança.

Se a solução encontrada não produzir uma redução suficiente no valor da função f, pode-se concluir que a região de confiança é muito grande, reduzi-la e voltar a resolver o problema.

O modelo m_k é geralmente definido como uma função quadrática da forma

$$
m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T B_k p \tag{2.7}
$$

Onde f_k, ∇f_k e B_k são um escalar, um vetor e uma matriz, respectivamente.

A matriz B_k é a Hessiana $\nabla^2 f_k$ ou alguma aproximação dessa matriz.

Em um certo sentido, as abordagens busca em linha e região de confiança diferem

na ordem em que são escolhidas a direção e distância a ser percorrida para a próxima iteração. A busca em linha começa pela fixação de uma direção p_k para, em seguida, identificar uma distância adequada a ser percorrida, ou seja, o comprimento do passo α_k.

No método da região de confiança, primeiro é escolhido uma distância máxima (o raio da região de confiança k) e, em seguida, procura-se uma direção e um tamanho de passo que alcancem a maior de redução possível no valor de f com esta restrição de distância. Se essa etapa revelar-se insatisfatória, o raio da região de confiança k é reduzido e uma nova tentativa de se encontrar um ponto que diminua o valor da função objetivo é feita.

Uma importante direção de busca é chamada de *Direção de Newton*. Essa direção é derivada da aproximação da série de Taylor de segunda ordem para $f(x_k + p)$. Métodos que usam a direção de Newton tem uma taxa rápida de convergência local, tipicamente quadrática.

A principal desvantagem da direção de Newton é a necessidade do cálculo da Hessiana $\nabla^2 f(x)$. A computação explícita desta matriz de segundas derivadas é, na maioria das vezes, um processo caro e propenso a erros de arredondamento.

Como alternativa, existem as chamadas direções *Quasi-Newton* de busca, as quais fornecem uma opção atraente na medida em que não requerem computação explícita da matriz hessiana e ainda assim atingem uma taxa de convergência superlinear. No lugar da verdadeira matriz Hessiana $\nabla^2 f(x)$, é utilizada uma aproximação da mesma, usualmente definida como B_k, que é atualizada após cada iteração para se levar em conta o conhecimento adicional sobre a função objetivo e seu gradiente adquiridos durante a iteração atual.

As atualizações na matriz B_k fazem uso do fato de que as mudanças no gradiente fornecem informações sobre a segunda derivada da função objetivo ao longo da direção de busca. A nova aproximação para a Hessiana B_{k+1} é calculada para satisfazer a seguinte condição, conhecida como *equação da secante*:

$$
\begin{aligned}
&B_{k+1}s_k = y_k \; onde \\
&s_k = x_{k+1} - x_k \; e \\
&y_k = \nabla f_{k+1} - \nabla f_k
\end{aligned}
\tag{2.8}
$$

Tipicamente, requisitos adicionais são impostos em B_{k+1}, tais como simetria (motivada pela simetria da matriz Hessiana exata) e uma restrição onde a diferença entre aproximações sucessivas B_k e B_{k+1} tenha um baixo posto. A aproximação inicial B_0 deve ser escolhida pelo desenvolvedor do modelo.

Uma das fórmulas mais populares para atualizar a aproximação da matriz hessiana B_k é a fórmula BFGS, nomeada a partir das iniciais de seus inventores, Broyden, Fletcher, Goldfarb e Shanno, que é definida por

$$
B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{2.9}
$$

A direção de busca quasi-Newton é dada por B_k substituindo a matriz hessiana exata na fórmula:

$$
p_k = -B_k^{-1} \nabla f_k \tag{2.10}
$$

Algumas implementações práticas de métodos Quasi-Newton evitam a necessidade de se fatorar B_k a cada iteração atualizando a inversa da matriz B_k em vez da matriz B_k em si.

Após a atualização da matriz b_k, a nova iteração é dada por:

$$
x_{k+1} = x_k + \alpha_k p_k \tag{2.11}
$$

onde o comprimento do passo α_k é escolhido para satisfazer as condições de Wolfe, por exemplo.

A inversa da matriz B_k, chamada de H_k, é atualizada através da seguinte fórmula:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.12}$$

onde

$$\rho_k = \frac{1}{y_k^T s_k} \tag{2.13}$$

Uma versão simplificada do algoritmo BFGS pode ser definida como:

Algoritmo 1 Algoritmo simplificado do método BFGS

Dado um ponto inicial x_0, uma tolerância $\epsilon > 0$ e uma aproximação para a matriz hessiana H_0;

$k \leftarrow 0$
while $\|\nabla f_k\| > \epsilon$ **do**
  $p_k \leftarrow -H_k \nabla f_k$ {Calcular a direção de busca p_k}
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$ {Onde α_k é calculado a partir de um procedimento de busca em linha que satisfaça as condições de Wolfe}
  $s_k \leftarrow x_{k+1} - x_k$
  $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$
  Calcular H_{k+1} *usando* (2.12)
  $k \leftarrow k + 1$
end while

O método BFGS tem taxa de convergência superlinear.

Os métodos Quasi-Newton não são diretamente aplicáveis a grandes problemas de otimização porque as aproximações a matriz Hessiana ou a sua inversa são geralmente densas. Nestes casos, usam-se métodos Quasi-Newton com memória limitada. Estes métodos são úteis para resolver problemas de grande porte cuja matriz Hessiana não pode ser computada a um custo razoável ou é demasiadamente densa para ser manipulada facilmente.

A implementação desses métodos mantém aproximações simples de matrizes Hessianas, e ao invés de armazenar plenamente essa matriz de aproximação densa, são armazenados apenas poucos vetores que representam as aproximações de forma implícita. Apesar destes modestos requisitos de armazenamento, métodos de memória limitada muitas vezes conseguem uma taxa de rendimento aceitável de convergência, normalmente linear.

Para a implementação do XCM, o algoritmo de minimização L-BFGS foi utilizado. A ideia principal do método L-BFGS é usar informações de curvatura apenas a partir das iterações mais recentes para construir a aproximação da matriz Hessiana. Um dos principais pontos fracos do método L-BFGS é que ele muitas vezes converge lentamente, o que geralmente leva a um número relativamente grande de avaliações da função objetivo. Além disso, o método é altamente ineficiente em problemas mal condicionados, especificamente nos casos onde a matriz Hessiana contém uma ampla distribuição de autovalores.

Existem implementações livres do algoritmo L-BFGS, como por exemplo nas bibliotecas ALGLIB, GSL - GNU Scientific Library e LBFGS. Essas três implementações foram testadas com um exemplo clássico teste para minimização multidimensional, a função de Rosenbrock.

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{2.14}$$

Este caso de teste amplamente conhecido possui seu ponto de mínimo em $(1, 1)$ e o valor da função f nesse ponto é 0. O ponto de partida tradicional, x_0, é $(-1.2, 1)$. A primeira opção de implementação do XCM foi com a Biblioteca GSL pela sua facilidade de utilização. No entanto, após alguns testes operacionais foi observado que a biblioteca LBFGS apresentou melhores resultados computacionais, sendo a biblioteca escolhida para a implementação final do XCM que serve como base para o desenvolvimento do XCM-GPU.

2.4 A Arquitetura CUDA

No final de 2006, a empresa NVidia introduziu o conceito da arquitetura CUDA (Compute Unified Device Arquitechture) e em fevereiro de 2007 foi lançada a primeira versão do SDK, ou Software Development Kit. O termo CUDA se refere a dois principais conceitos: a arquitetura massivamente paralela de GPU's e ao modelo de programação paralela desenvolvido para ser utilizado nessas modernas GPU's, de

forma a utilizar o poder de processamento dessa arquitetura de forma fácil e eficiente. O foco principal desse desenvolvimento é criar um modelo heterogêneo de computação, onde aplicações utilizam processamento sequencial na CPU e paralelo na GPU.

O conceito de programação em GPU existe desde o início da década passada. As GPU's do início dos anos 2000 foram projetadas para produzir uma cor para cada pixel na tela usando unidades aritméticas programáveis, conhecidos como *pixel shaders*. A ideia empregada nos experimentos de programação em GPU da época era de manipular os dados de entrada para que significassem algo diferente dos dados convencionais para cálculo de cores. Em seguida, os pixel shaders eram programados para executar os cálculos sobre esses dados e os resultados eram retornados como se fossem cores, mas na realidade tinham outro significado, referente aos dados de entrada.

Esse modelo de programação se mostrou extremamente complicado e com diversos inconvenientes. Grande parte das aplicações de computação científica, por exemplo, não podiam ser empregadas devido a capacidade limitada para operações de ponto flutuante (alguns pixel shaders nem suportavam esse tipo de operação). A ausência de bons métodos de debug para o código de GPU também dificultava esse tipo de desenvolvimento.

No entanto, esse modelo demonstrou a elevada capacidade de processamento aritmético das GPU's, e abriu o caminho para o desenvolvimento da arquitetura CUDA. Em novembro de 2006 foi lançada a primeira placa de vídeo com suporte a CUDA na GPU, onde cada unidade lógica e aritmética presente pode ser utilizada para computação de uso geral, ou *general purpose computing*. Duas ou até três placas idênticas podem ser combinadas no mesmo sistema, através da tecnologia chamada SLI, ou *scalable link interface*, como pode ser visto na figura 2.1 abaixo. Essa combinação aumenta ainda mais o poder de processamento do sistema pois todas as GPU's presentes podem ser usadas em conjunto.

A evolução das funcionalidades e do poder de processamento das GPU's vem

Figura 2.5: GeForce 8800 GTX, em configuração SLI

sendo extraordinária. Em um primeiro momento, apenas operações de ponto flutuante de precisão simples eram suportadas e foram implementadas atendendo aos requerimentos da IEEE para essa especificação. O suporte para operações de ponto flutuante com precisão dupla foi apresentado em 2008. Como as especificações se modificaram bastante ao longo do tempo, o conceito de compatibilidade de computação, ou *compute capability*, foi adotado para classificar diferentes modelos de GPU's. Por exemplo, a família de GPU's com compute capability 1.0, 1.1 e 1.2 suportam apenas operações de ponto flutuante simples, enquanto que GPU's com compute capability acima de 1.3 já suportam cálculos com precisão dupla.

A utilização, via software, da arquitetura CUDA é muito simples em linguagens de programação como C, por exemplo. Pode-se imaginar CUDA como um conjunto pequeno de extensões para a linguagem C. Uma vez configurados os drivers do sistema e instalado o SDK, é possível rapidamente aplicar CUDA incrementalmente a um programa em C para utilizar as funcionalidades disponíveis.

É importante destacar a principal característica de um sistema heterogêneo de CPU e GPU: essas unidades de processamento são completamente distintas, cada uma possuindo sua área de memória. A CPU é normalmente chamada de *Host*, e o conceito de *host memory* se refere à memória RAM principal do computador, onde

programas sequenciais são executados. A GPU é chamada de *Device* e sua memória é definida como *device memory*. Essa é a memória disponível na placa de vídeo do sistema. Dados que estão na host memory podem ser transferidos para a device memory via barramento PCI-e (PCI Express) e vice-versa. Esse tipo de operação de transferência de dados é sempre comandado pela CPU.

Dessa forma, partes sequenciais de um programa são executados na CPU, e as porções paralelas de código são executadas na CPU em funções chamadas *kernels*. Um kernel é simplesmente uma função que é chamada pelo host e roda no device, ou seja, é chamada pela CPU e roda em paralelo nas unidades de processamento da GPU. Apenas um kernel pode ser executado em um device por vez, e essa execução é iniciada disparando-se muitas threads concorrentes, que executam o mesmo kernel em paralelo.

Existem algumas distinções entre threads convencionais de CPU e as chamadas *CUDA-Threads*, que são as threads que rodam na GPU. Cuda-Threads são muito mais "leves"do que threads normais no que se refere a criação das mesmas e em trocas de contexto. Por exemplo, milhares de cuda-threads podem ser criadas em poucos ciclos de processamento. Dessa forma, o overhead de criação de threads é muito baixo, e mesmo uma implementação de kernel muito simples pode levar a ganhos consideráveis de performance. Como a troca de contexto também é muito rápida nessa arquitetura, no momento que uma thread "parar"esperando um dado da memória, uma outra thread passa a ser executada praticamente sem custo computacional.

A título de clareza, pode-se imaginar um exemplo onde deve-se somar um vetor A com um vetor B e guardar o resultado em um vetor C (este exemplo encontra-se descrito detalhadamente em [5]). Uma simples função em C que implementa essa operação pode ser vista no código 2.1. Nesse caso de implementação sequencial, os elementos do vetor A são somados ao elemento do vetor B e guardados no vetor C a cada iteração do loop. Uma implementação paralela desse código, rodando em CPU com 2 threads por exemplo, poderia ser implementada de forma que uma

thread somasse os elementos em posições pares do vetor e a outra os elementos em posições ímpares. Seguindo esse conceito, na arquitetura CUDA, cada cuda-thread seria usada de forma semelhante. No entanto, para atingir tal objetivo é necessário transformar a função *add* em um kernel, ou seja, a função que será executada por todas as cuda-threads.

```
#define N   10

void add( int *a, int *b, int *c )
{
  for (i=0; i < N; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```

Código 2.1: Um exemplo sequencial de soma de vetores

A implementação de um kernel é tão simples quanto adicionar a palavra chave *global* antes do nome da função, como pode ser visto no código 2.2. Um ou mais *kernels* de execução podem ser definidos em um mesmo programa. Além disso, kernels têm a capacidade de chamar outras funções que estejam definidas no espaço de memória do dispositivo. Essas funções são definidas da mesma forma que kernels, mas usando o qualificador *device* no lugar de *global*.

```
__global__ void add( int *a, int *b, int *c )
{
  //Codigo do kernel
}
```

Código 2.2: Definição de um *Kernel*

O próximo passo consiste em movimentar os dados dos vetores A e B da *host memory*, ou seja, da memória principal do computador, para a *device memory*, que é a memória da placa de vídeo onde as cuda-threads podem operar. Antes de efetuar essa movimentação de dados, é necessário primeiramente alocar espaço na *device memory* para que os dados possam ser copiados. Existem funções especiais que permitem alocar e liberar espaço (cudaMalloc e cudaFree, respectivamente) na memória do device que funcionam de forma muito parecida com os seus equivalentes

da própria linguagem C. Dessa forma, deve-se alocar espaço para os dois vetores de entrada de dados (A e B) e também para o vetor C, do resultado. Uma vez realizada a soma dos vetores, o espaço alocado na *device memory* deve ser liberado.

A movimentação dos dados é realizada através da função cudaMemcpy. Um dos parâmetros de entrada dessa função é a "direção"para onde os dados devem ser copiados. No caso, a direção de cópia pode ser da *host memory* para a *device memory*, ou vice-versa. Nesse exemplo, os vetores A e B seriam copiados da *host memory* para a *device memory* e o resultado final (o vetor C), seria copiado da *device memory* para a *host memory*, de forma a ser usado no restante do programa.

```
#define N   10

int main( void )
{
  int a[N], b[N], c[N];
  int *dev_a, *dev_b, *dev_c;
  // Aloca memoria na GPU
  cudaMalloc( (void**)&dev_a, N * sizeof(int) );
  cudaMalloc( (void**)&dev_b, N * sizeof(int) );
  cudaMalloc( (void**)&dev_c, N * sizeof(int) );
  // Copia os vetores a and b para a GPU
  cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

  // Chamada do kernel seria nesse ponto

  // Copia o vetor C da memoria da GPU para a memoria do host
  cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

  // Libera a memoria alocada na GPU
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_c );
}
```

Código 2.3: Alocação e movimentação de dados

O código 2.3 mostra como são feitas as chamadas para todas essas funções, menos a chamada do kernel propriamente dita. Quando um kernel é iniciado, ele é executado como um vetor de cuda-threads paralelas. Cada thread roda o mesmo kernel em um device. Para que as threads atuem sobre dados diferentes, cada thread recebe uma identificação, ou índice, que é usado para fins de endereçamento de memória, por exemplo. Se o kernel do exemplo anterior for executado em 10 cuda-

threads, cada cuda-thread possuirá uma identificação única e dessa forma pode usar essa referência para acessar uma posição específica do vetor. Assim, a cuda-thread com identificação 0 pode ler o elemento 0 do vetor A, em seguida ler o elemento 0 do vetor B, efetuar a soma dos dois elementos e guardar o resultado no elemento 0 do vetor C.

Se cada cuda-thread existisse de forma totalmente independente de outra cuda-thread, a cooperação entre as cuda-threads seria extremamente limitada. Dessa forma, cuda-threads podem ser agrupadas em conjuntos, ou *blocos* de threads. Assim, cuda-threads dentro de um mesmo bloco possuem as seguintes características:

- podem cooperar entre si
- possuem um espaço comum de memória chamada *shared memory*
- podem ser sincronizadas com barreiras

A chamada de um *kernel* é feita como um grid de blocos de threads. Cada bloco possui a sua identificação no grid e, como foi dito anteriormente, cada cuda-thread possui a sua identificação dentro do bloco. Esse tipo de implementação é extremamente escalável, já que o hardware do *device* é livre para alocar os blocos de acordo com os processadores disponíveis. Um conjunto de 8 blocos pode rodar em um *device* com 2 processadores (com 4 blocos por processador) e pode ser transparentemente portado para um *device* com 4 processadores, nesse caso executando dois blocos por processador. Em ambos os casos, nenhuma alteração no código fonte do programa é necessária se o mesmo estiver devidamente parametrizado. Em qualquer chamada a um kernel, é necessário definir uma configuração de execução, cuja sintaxe de invocação possui dois parâmetros adicionais, que são o número de blocos de execução e o número de cuda-threads por bloco de execução. A sintaxe de definição de uma configuração de execução pode ser visto no código 2.4. Nesse exemplo, N blocos de execução seriam invocados na GPU e cada bloco teria apenas 1 cuda-thread.

O kernel completo do exemplo de soma de vetores pode ser visto no código 2.5. A variável *blockIdx* é definida automaticamente, de forma sequencial a partir de zero,

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

Código 2.4: Sintaxe de configuração de execução de um kernel

para enumerar os blocos de execução que foram criados a partir da configuração de execução definida pelo programador. De forma semelhante, existem outras variáveis que podem ser usadas para identificar uma cuda-thread dentro de um grid de blocos de execução. No exemplo de soma de vetores, como cada bloco utiliza apenas uma cuda-thread, cada cuda-thread pode ser identificada pelo número do bloco a que ela pertence. Internamente, uma grid de blocos pode ter uma ou mais dimensões. No exemplo foi utilizada uma grid de dimensão 1 e por isso a componente x da variável blockIdx foi empregada para identificar os blocos. Em um grid de dimensão 2, por exemplo, tanto a componente x quanto a componente y da variável blockIdx podem ser usadas para identificar um bloco.

```
__global__ void add( int *a, int *b, int *c )
{
  int tid = blockIdx.x;
  if (tid < N)
    c[tid] = a[tid] + b[tid];
}
```

Código 2.5: Código do kernel de exemplo

Como foi descrito anteriormente, uma vez que cada cuda-thread tenha a sua identificação, essa identificação pode ser utilizada para fazer o acesso a memória para buscar um ou mais elementos específicos que se deseja processar. Além da busca dos dados e da operação de soma propriamente dita, é feita uma verificação se o acesso a memória está sendo feito em uma área de memória efetivamente alocada. Essa verificação é considerada uma boa prática de programação, para evitar que uma configuração de execução "errada"faça com que as cuda-threads acessem memória não alocada. No exemplo, isso poderia acontecer se mais de 10 blocos de execução fossem invocados.

O fluxo de execução do exemplo de soma de vetores com CUDA pode ser resumido nos seguintes passos:

- Alocar espaço na memória da GPU (device memory) para receber uma cópia dos vetores A e B
- Copiar os dados da memória da CPU (host memory) para a memória da GPU
- Chamar o kernel com uma configuração de execução definida em relação ao número de blocos e ao número de cuda-threads por bloco
- Copiar o resultado do processamento do kernel de volta para a host memory
- Desalocar os vetores da device memory

Este procedimento, apesar de extremamente resumido, é a base para toda a implementação do XCM-GPU. Existem diversos outros aspectos da arquitetura CUDA, como as características dos tipos de memórias disponíveis e formas de "perguntar" ao device algumas propriedades (como o número de compatibilidade de versão do device, por exemplo) que tiveram de ser estudados para se conseguir uma correta implementação do método.

A introdução à arquitetura CUDA apresentada aqui visa apenas descrever os principais conceitos necessários ao entendimento do XCM-GPU. A documentação disponível para download a partir do próprio site da empresa Nvidia é bastante compreensiva e completa, sendo fortemente recomendada para um entendimento mais abrangente do assunto.

Além da documentação disponível, existe a biblioteca de templates Thrust (que a partir da versão 4.0 da arquitetura passou a vir integrada no próprio pacote de instalação) que facilita demasiadamente o uso da arquitetura CUDA. A primeira versão dessa biblioteca se fez disponível ao público em 26 de maio de 2009 e foi desenvolvida pelos próprios funcionários da empresa NVidia, que desenvolveu a arquitetura CUDA. Com o uso de Thrust é possível, por exemplo, instanciar vetores diretamente na memória do device e efetuar a cópia dos dados da host memory para a device memory em uma linha de código apenas, sem a necessidade de explicitamente realizar todas essas operações. Além disso, a biblioteca Thrust oferece uma série de

algoritmos implementados de forma paralela na GPU prontos para serem utilizados, como pode ser visto no código 2.6, disponível em http://thrust.github.com/ e reproduzido abaixo:

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
int main(void)
{
  // aloca um vetor de 100.000 posicoes na host memory
  thrust::host_vector<int> h_vec(100000);
  // preenche o vetor com numeros aleatorios
  thrust::generate(h_vec.begin(), h_vec.end(), rand);
  // copia o vetor da host memory para a device memory
  thrust::device_vector<int> d_vec = h_vec;
  // ordena o vetor no device em paralelo
  thrust::sort(d_vec.begin(), d_vec.end());
  // copia o vetor ordenado de volta para a host memory
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
  return 0;
}
```

Código 2.6: Exemplo de uso da biblioteca de templates Thrust

Nesse exemplo, um vetor de 100.000 posições é gerado e preenchido com números aleatórios. Em seguida, esse vetor é copiado para a device memory, ordenado em paralelo com o algoritmo disponível na biblioteca e então copiado de volta para a GPU.

A implementação final do XCM-GPU utiliza todos os conceitos aqui descritos, e alguns detalhes adicionais serão apresentados no capítulo 4 onde a metodologia de implementação é discutida em detalhes.

Capítulo 3

O Método XCM

O método de agrupamento via suavização hiperbólica tradicional, apresentado inicialmente em [6] e definido aqui como XCM, *Xavier Clustering Method*, utiliza a formulação do problema de agrupamento baseado no critério de mínima soma de quadrados como ponto de partida. Este critério corresponde à minimização da soma de quadrados das distâncias das observações para o ponto definido como centro do conjunto, ou *cluster*.

O principal problema associado a essa formulação é a criação de um problema de otimização global não-diferenciável e não-convexo. A ideia principal do XCM é *suavizar* o problema criado a partir dessa formulação. A técnica de suavização aplicada foi desenvolvida a partir de uma adaptação do método de penalização hiperbólica, originalmente apresentado em [7].

Para uma análise da metodologia de resolução, é necessário definir o problema de agrupamento como um problema *min-sum-min*, para em seguida transformar o problema e aplicar a técnica de suavização hiperbólica. As etapas de formulação do problema, até a definição do problema final utilizado no XCM-GPU são definidas nas seções a seguir. Como essa formulação serve de base para todo o XCM-GPU, ela é apresentada aqui apenas em razão da completude do trabalho. Todo esse desenvolvimento já foi apresentado em [6] e [9], sendo o crédito devidamente destinado aos autores.

3.1 Análise de Agrupamento como um problema min-sum-min

Seja $S = s_1, s_2, ..., s_m$ um conjunto de m pontos, ou observações, em um espaço euclidiano de dimensão n que devem ser agrupados em q conjuntos. O número q é pré-definido, e um dos parâmetros de entrada do algoritmo.

Seja $x_i, i = 1, ..., q$ um conjunto de pontos centrais dos *clusters*, onde cada x_i é chamado de *centroid* e cada $x_i \in \mathbb{R}^n$.

O conjunto dos q centroids é representado por $X \in \mathbb{R}^n q$. Para cada observação s_j, é calculada a distância dela aos q centroids, e a menor distância é definida como

$$z_j = \min_{x_i \in X} \parallel s_j - x_i \parallel_2 \tag{3.1}$$

Considerando uma posição específica dos q centroids, pode-se definir a mínima soma de quadrados das distâncias como:

$$D(X) = \sum_{j=1}^{m} z_j^2 \tag{3.2}$$

Dessa forma, o posicionamento ótimo dos centros dos centroids deve fornecer o melhor resultado para essa medida. Assim, se X^* é definido como o posicionamento ótimo, então o problema pode ser definido como:

$$X^* = \underset{X \in R^{nq}}{\operatorname{argmin}} D(X) \tag{3.3}$$

onde X é o conjunto de pontos dos centros dos q centroids.

Assim, utilizando as equações 3.1, 3.2 e 3.3 acima, chega-se na seguinte definição:

$$X^* = \underset{X \in R^{nq}}{\operatorname{argmin}} \sum_{j=1}^{m} \min_{x_i \in X} \parallel s_j - x_i \parallel_2^2 \tag{3.4}$$

Esse problema pode ser transformado da seguinte forma:

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\,a: \; z_j = \min_{i=1..q} \parallel s_j - x_i \parallel_2,\; j = 1, \ldots, m \end{aligned} \tag{3.5}$$

Considerando a sua definição em 3.1, cada z_j deve necessariamente satisfazer o conjunto de desigualdades de 3.6,

$$z_j - \parallel s_j - x_i \parallel_2 \leq 0, i = 1, \ldots, q \tag{3.6}$$

dado que z_j já é a menor distância entre a observação j e um centroid qualquer.

Substituindo-se as igualdades do problema 3.5 pelas desigualdades de 3.6, obtem-se o problema relaxado

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\,a: \; z_j - \parallel s_j - x_i \parallel_2 \leq 0,\; j = 1, \ldots, m,\; i = 1, \ldots, q \end{aligned} \tag{3.7}$$

Como as variáveis z_j não são limitadas inferiormente, a solução ótima do problema 3.7 é $z_j = 0, j = 1, \ldots, m$. Assim, o problema 3.7 não é equivalente ao problema 3.5. É necessário, portanto, modificar o problema 3.7 para se obter a equivalência desejada. Para isso, é necessário definir a função φ da seguinte forma:

$$\varphi(y) = \max\{0, y\} \tag{3.8}$$

Pode-se observar que, se as desigualdades em 3.6 são válidas, também são igualmente válidas as restrições de 3.9:

$$\sum_{i=1}^{q} \varphi(z_j - \parallel s_j - x_i \parallel_2) = 0,\; j = 1, \ldots, m \tag{3.9}$$

Na figura 3.1, extraída de [6], pode-se observar o gráfico das três primeiras par-

celas do somatório da equação 3.9 como função de z_j, onde $d_i = \| s_j - x_i \|_2$ e considerando-se as distâncias d_i ordenadas em ordem crescente segundo os índices.

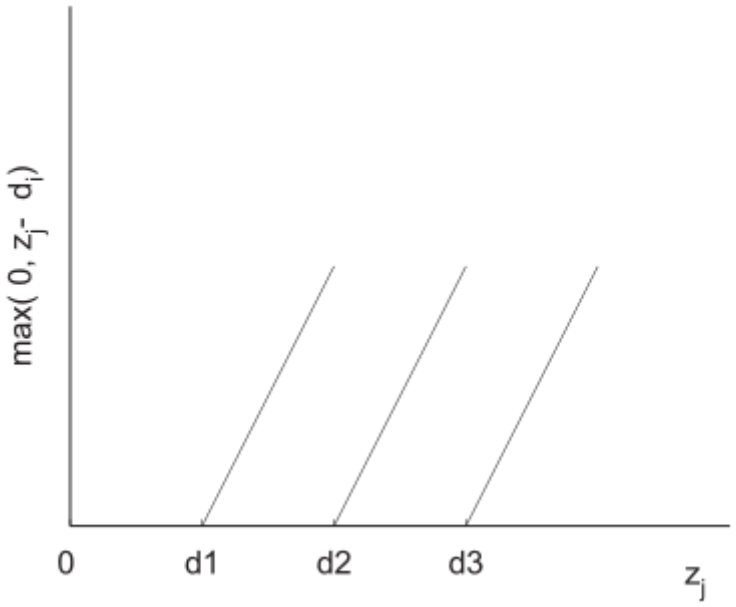


Figura 3.1: Três primeiras parcelas do somatório da equação 3.9

Com a substituição das desigualdades do problema 3.7 pela equação 3.9, seria obtido um problema equivalente porém com a mesma propriedade indesejável que as variáveis $z_j = 0, j = 1, \ldots, m$ ainda não são limitadas inferiormente. No entanto, como a função objetivo do problema 3.7 forçará os valores das variáveis $z_j = 0, j = 1, \ldots, m$ a receber os menores valores possíveis, pode-se pensar em limitar inferiormente essas variáveis ao se considerar $>$ ao invés de $=$ na equação 3.9. Dessa forma, chega-se ao problema não-canônico 3.10.

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\,a: \sum_{i=1}^{q} \varphi(z_j - \| s_j - x_i \|_2) > 0,\ j = 1, \ldots, m \end{aligned} \tag{3.10}$$

Para recuperar a formulação canônica, as desigualdades no problema 3.10 são perturbadas e, dessa forma, obtem-se o seguinte problema modificado 3.11:

$$
\begin{aligned}
&minimizar \sum_{j=1}^{m} z_j^2 \\
&sujeito\,a: \sum_{i=1}^{q} \varphi(z_j - \parallel s_j - x_i \parallel_2) \geq \varepsilon,\ j = 1, \ldots, m
\end{aligned}
\tag{3.11}
$$

para $\varepsilon > 0$. Como o conjunto viável de soluções do problema 3.10 é o limite do conjunto viável de soluções do problema 3.11 quando $\varepsilon \to 0_+$ pode-se, então, resolver o problema 3.10 através da resolução de uma sequência de problemas iguais a 3.11 para uma sequência de valores decrescentes de ε que se aproximam de 0 pela direita.

A prova que o valor da solução ótima do problema 3.4 está arbitrariamente próximo do valor da solução do problema 3.10 está disponível em [6].

Analisando o problema 3.11, é possível verificar que a definição da função ε impõe a ele uma estrutura não diferenciável de difícil solução. Nesse ponto é aplicado o conceito de suavização, de forma a transformar o problema 3.11 para que sua solução numérica seja simplificada, sem distanciá-lo do problema original.

Assim, utilizando-se do conceito de suavização, deve-se definir uma função ϕ da seguinte forma:

$$
\phi(y, \tau) = \frac{(y + \sqrt{y^2 + \tau^2})}{2} \tag{3.12}
$$

para $y \in \mathbb{R}$ e $\tau > 0$.

Pode-se notar que a função ϕ possui as seguintes características:

(a) $\phi(y, \tau) > \varphi(y),\ \forall \tau > 0$;

(b) $\lim_{\tau \to 0} \phi(y, \tau) = \varphi(y)$;

(c) $\phi(y, \tau)$ é uma função convexa crescente que pertence à classe de funções C^∞ na variável y.

Assim, a função ϕ consiste em uma aproximação da função φ definida na equação 3.8. Utilizando-se das mesmas convenções definidas na apresentação da

figura 3.1, as três primeiras parcelas componentes da equação 3.9 e a correspondente suavização aproximada, obtida a partir da equação 3.12 são mostradas na figura 3.2.

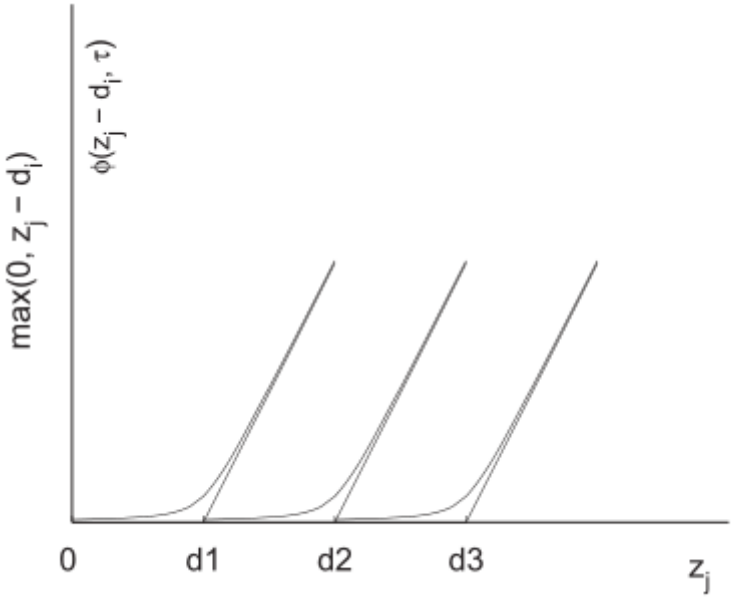


Figura 3.2: Três primeiras parcelas do somatório da equação 3.9 com suas equivalentes suavizações

Ao substituir a função ϕ pela função φ no problema 3.11, obtem-se o seguinte problema:

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\,a: \sum_{i=1}^{q} \phi(z_j - \parallel s_j - x_i \parallel_2, \tau) \geq \varepsilon,\ j = 1, \ldots, m \end{aligned} \tag{3.13}$$

Para se obter um problema completamente diferenciável, ainda é necessária a suavização da distância euclidiana $\parallel s_j - x_i \parallel_2$ do problema 3.13. Dessa forma, define-se a função θ como abaixo:

$$\theta(s_j, x_i, \gamma) = \sqrt{\sum_{l=1}^{n} (s_j^l + x_i^l)^2 + \gamma^2} \tag{3.14}$$

para $\gamma > 0$.

A função θ possui as seguintes propriedades:

(a) $\lim_{\gamma \to 0} \theta(s_j, x_i, \gamma) = \| s_j - x_i \|_2$;

(b) θ é uma função que pertence à classe de funções C^{∞}.

Ao substituir a distância euclidiana $\| s_j - x_i \|_2$ do problema 3.13 pela função θ, obtém-se o problema diferenciável 3.15, definido da seguinte forma:

$$\begin{aligned} & minimizar \sum_{j=1}^{m} z_j^2 \\ & sujeito\, a: \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) \geq \varepsilon,\ j = 1, \ldots, m \end{aligned} \tag{3.15}$$

As propriedades das funções ϕ e θ permitem buscar uma solução para o problema 3.10 através da resolução de uma sequência de problemas como definidos em 3.15 para valores cada vez menores de ε, τ, e γ, até que uma regra de parada seja atingida. Os valores de ε, τ, e γ devem tender a zero sempre pela direita, e nunca devem chegar a ser iguais a zero.

Como $z_j \geq 0, j = 1 \ldots, m$ a minimização da função objetivo irá tentar reduzir ao máximo esses valores. Por outro lado, dado qualquer conjunto de centroids $x_i, i = 1 \ldots, q$ e observando-se a propriedade (c) da função de suavização hiperbólica ϕ, as restrições do problema 3.15 são funções monotonicamente crescentes em z_j. Então, essas restrições sempre estarão ativas e o problema 3.15 será equivalente ao problema 3.16 abaixo:

$$\begin{aligned} & minimizar\, f(x) = \sum_{j=1}^{m} z_j^2 \\ & sujeito\, a: \ h_j(z_j, x) = \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) - \varepsilon = 0,\ j = 1, \ldots, m \end{aligned} \tag{3.16}$$

A dimensão do problema 3.16 é $(nq + m)$. Como, em geral, o número de observações m é grande, o problema 3.16 possui um número muito grande de variáveis.

No entanto, uma característica interessante desse problema é que ele possui uma estrutura separável, porque cada variável z_j aparece em apenas uma restrição de igualdade. Assim, como a derivada parcial de $h(z_j, x)$ em relação a $z_j, j = 1 \ldots, m$ não é igual a zero, é possível usar o teorema da função implícita para calcular cada componente $z_j, j = 1 \ldots, m$ como função das variáveis dos centroids $x_i, i = 1 \ldots, q$. Dessa forma, o problema irrestrito 3.17 é obtido:

$$minimizar\, f(x) = \sum_{j=1}^{m} z_j(x)^2 \tag{3.17}$$

Cada z_j do problema 3.17 resulta do cálculo da raiz da equação 3.18 abaixo:

$$h_j(z_j, x) = \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) - \varepsilon = 0,\ j = 1, \ldots, m \tag{3.18}$$

Novamente, devido a propriedade (c) da função de suavização hiperbólica, cada termo ϕ de 3.18 é estritamente crescente em relação a variável z_j e, dessa forma, pode-se concluir que as equações de 3.18 possuem uma única raiz. Além disso, considerando-se novamente o teorema da função implícita, as funções $z_j(x)$ têm todas as derivadas em relação as variáveis $x_i, i = 1 \ldots, q$. Logo, é possível calcular o gradiente da função objetivo do problema 3.17 como:

$$\nabla f(x) = \sum_{j=1}^{m} 2 z_j(x) \nabla z_j(x) \tag{3.19}$$

Onde

$$\nabla z_j = -\nabla h_j(z_j, x) / \frac{\partial h_j(z_j, x)}{\partial z_j} \tag{3.20}$$

e $\nabla h_j(z_j, x)$ é obtido das equações 3.12 e 3.14. $\partial h_j(z_j, x)/\partial z_j$ vem da equação 3.18.

Dessa forma, é fácil resolver o problema 3.17 usando métodos de minimização baseados na primeira derivada, como apresentado no capítulo 2. Além desse fato, vale ressaltar que o problema 3.17 está definido no espaço de dimensão (nq). Como o número de centroids, q, é relativamente pequeno em relação ao número de observações e, em geral, é um número pequeno, o problema 3.17 pode ser considerado um problema de dimensão pequena, principalmente quando comparado ao problema 3.16.

O algoritmo do XCM pode ser resumido da seguinte forma:

Algoritmo 2 Algoritmo XCM simplificado

Inicialização: Defina os valores iniciais de $x^0, \varepsilon^1, \tau^1, \gamma^1$ e os valores $0 < \rho_1 < 1, 0 < \rho_2 < 1, 0 < \rho_3 < 1$;
  $k \leftarrow 1$
  while Uma regra de parada não for atingida **do**
    Resolva o problema (17) com $\varepsilon = \varepsilon^k, \tau = \tau^k, \gamma = \gamma^k$ começando no ponto inicial x^{k-1} e seja x^k a solução obtida.
    $\varepsilon^{k+1} \leftarrow \rho_1 \varepsilon^k$
    $\tau^{k+1} \leftarrow \rho_2 \tau^k$
    $\gamma^{k+1} \leftarrow \rho_3 \gamma^k$
    $k \leftarrow k + 1$
  end while

Assim como em outros métodos iterativos, a solução para o problema de agrupamento é obtida, em teoria, através da resolução de uma sequencia infinita de problemas de otimização. No método XCM, cada problema a ser resolvido é irrestrito e de baixa dimensão. Outro ponto que merece destaque é que o algoritmo faz com que os valores de τ e γ se aproximem de zero. Então, as restrições dos sub-problemas resolvidos tendem àquelas do problema 3.11 e, de forma simultânea, a redução do valor de ε (também tendendo a zero) faz com que o problema 3.11 se aproxime do problema original 3.10.

Diversas regras de parada podem ser empregadas, como, por exemplo, observando-se se os valores da função objetivo não sofreu alteração significativa após um determinado número de iterações. A mesma regra pode ser aplicada aos

valores das coordenadas dos pontos dos centroids.

Devido às propriedades de continuidade de todas as funções envolvidas, a sequência $x^1, x^2, \ldots, x^k$ de valores ótimos dos problemas suavizados tendem ao valor ótimo de 3.1. Os resultados computacionais mostraram que, após poucas iterações, o valor mínimo da função objetivo é atingido e uma solução para o posicionamento dos centroids é encontrada.

Capítulo 4

O Método XCM-GPU

4.1 Conceitos Básicos

A ideia para a implementação do método XCM-GPU surgiu após uma análise do método XCM original. A transformação do problema 3.16 no problema 3.17, além de gerar uma grande redução no número de variáveis do problema, permite a formulação do mesmo como uma função que é definida por um somatório de parcelas independentes. Esta ideia pode ser observada na equação 4.1 abaixo, onde cada cálculo de $z_j(x)^2$ depende apenas de x, ou seja, da solução da iteração atual do método.

$$minimizar\, f(x) = \sum_{j=1}^{m} z_j(x)^2 = z_1(x)^2 + z_2(x)^2 + z_3(x)^2 + \ldots + z_m(x)^2 \qquad (4.1)$$

A equação 3.18 representa o cálculo de cada $z_j(x)$. O somatório de $f(x)$ possui m parcelas e, em geral, para aplicações reais, o número m (a quantidade total de observações) é grande. Na resolução de 3.18 necessita-se encontrar a raiz da equação e dessa forma são calculadas m raízes. No entanto, não é necessário encontrar a raiz de z_1 para que o cálculo da raiz de z_2 seja feito, e assim por diante.

Esse é o principal ponto explorado pelo XCM-GPU. A base do método é resolver na GPU, usando cuda-threads (definidas no capítulo 2) distintas, as diversas parcelas

referentes ao cálculo de $f(x)$.

O mesmo conceito aplica-se ao gradiente da função f. A equação 3.19 define o gradiente da função f também como uma soma de parcelas independentes e, apesar da maior complexidade do cálculo, o mesmo conceito de resolução paralela se aplica.

Excetuando-se os passos de inicialização (discutidos na seção 4.1) necessários à programação em GPU, a implementação sequencial e a implementação paralela possuem as mesmas características. As principais diferenças residem na forma do cálculo da função objetivo e seu gradiente onde, na versão em GPU, ganhos de performance consideráveis são obtidos. Uma vez calculado o valor de $f(x)$ e de $\nabla f(x)$, esses valores são repassados à função de minimização irrestrita, que foi implementada da mesma forma tanto na versão sequencial quanto na versão paralela.

Um outro ponto do algoritmo XCM que permite uma implementação paralela é na criação da chamada *lista de associações*. Uma vez calculada a solução final X^*, é necessário associar as observações a um, e apenas um, cluster seguindo a regra estabelecida para o agrupamento. Considerando a miníma soma-de-quadrados como exemplo, uma observação estará associada ao centroid que estiver mais perto considerando-se a menor distância euclidiana como referência. Como existem q centroids, cada observação deve calcular q distâncias e se associar ao centroid mais próximo. Levando-se em consideração que o posicionamento dos centroids já está definido e é visível para todas as observações, a observação 1 pode descobrir a que centroid está associada de forma independente da observação 2, e assim por diante. No entanto, a implementação paralela da lista de associações não foi feita para que a comparação dos resultados de desempenho sejam puramente focadas na implementação paralela do cálculo da função objetivo e do gradiente da mesma.

4.2 Inicialização do XCM-GPU

Tanto o algoritmo XCM quanto o algoritmo XCM-GPU possuem uma etapa de inicialização, onde os valores dos parâmetros são defindos e a estimativa inicial dos centroids é calculada. No entanto, o XCM-GPU possui uma etapa adicional de

inicialização que se faz necessária para a correta configuração do algoritmo paralelo.

Esta pode ser subdividida nas seguintes ações:

4.2.1 Alocação de dados na memória do device

Antes da cópia efetiva de dados da memória RAM do host para a memória global do device, é necessário alocar espaço de memória da GPU para que essa cópia possa ser feita. Uma vez alocado o espaço para as principais variáveis, usando algumas das funções mostradas no capítulo 2, a cópia dos dados pode ser realizada.

Vale notar que, como o conjunto de pontos das observações é fixo e imutável ao longo da execução do algoritmo, não é necessário copiar esses dados a cada chamada de kernel. Isso é possível devido a uma característica da memória global da arquitetura CUDA, que é persistente ao longo de diversas chamadas de kernel. Dessa forma, a movimentação de dados entre a memória do host e do device é bastante reduzida uma vez que essa etapa inicial seja feita. Em geral, apenas a solução da iteração anterior e alguns outros parâmetros (como os valores de gamma, tau e epsilon por exemplo) precisam ser movimentados entre as memórias em cada chamada de kernel.

Como exemplo, pode-se imaginar um problema de dimensão 2 com 100 centroids ($p = 2$, $q = 100$). Nesse caso, a cada chamada de kernel, pouco mais de 200 valores do tipo double devem ser movimentados de uma memória a outra, considerando-se um valor double para cada dimensão dos centroids e mais alguns outros parâmetros do algoritmo. Uma vez que a execução do kernel tenha terminado, apenas o valor da função objetivo deve ser movimentado no sentido inverso, assim como o vetor gradiente da função que possui $p * q$ posições. Assim, o custo inicial de movimentação dos dados é alto, mas é consideravelmente menor no momento de consecutivas chamadas de execução do kernel.

4.2.2 Cálculo do número e do tamanho do bloco de cuda-threads

O algoritmo XCM-GPU possui apenas um parâmetro de entrada a mais que o XCM-GPU original, que representa o número de cuda-threads a ser usado na execução. Foram utilizados sempre valores da potência de 2 para o número de cuda-threads. Na grande maioria dos casos de teste, foram utilizadas 4096 cuda-threads na computação, pois optou-se por não investigar profundamente o impacto de diferentes números de cuda-threads na performance do XCM-GPU.

Além disso, como explicado no capítulo 2, é necessário calcular quantos blocos de cuda-threads serão criados. Para isso, existem funções no toolkit CUDA que permitem buscar informações de características da GPU disponível para uso. Por exemplo, o uso da função cudaGetDeviceProperties permite descobrir, entre outras propriedades, o nome da GPU, qual versão de capacidade de computação (compute capability) e o número de multiprocessadores disponíveis.

Um multiprocessador pode executar mais de um bloco de cuda-threads, mas um bloco não pode ser executado em mais de um multiprocessador. Dessa forma, para simplificar a implementação do método, foi definido inicialmente que o número de blocos será igual ao número de multiprocessadores disponíveis na GPU quando o número de cuda-threads passado como parâmetro pelo usuário for maior que 32 (no caso da GPU disponível para o desenvolvimento, 32 é o número de multiprocessadores disponíveis).

Assim, resta o cálculo do tamanho do bloco, que é feito dividindo-se o número de cuda-threads passado como parâmetro pelo número de blocos previamente definido.

Por exemplo, se o usuário passar como parâmetro de entrada 64 cuda-threads, e usando-se o valor default igual a 32 para o número de blocos, 32 blocos serão criados com 2 cuda-threads em cada bloco. 128 cuda-threads implicam na criação de 32 blocos de 4 threads cada e assim sucessivamente. Vale ressaltar que o número mínimo de cuda-threads não é igual a 32. Caso seja passado um valor menor que 32, o número de blocos pode passar a ser o número de cuda-threads informado pelo

usuário. Se, por exemplo, for passado como parâmetro 4 cuda-threads, 4 blocos com uma cuda-thread em cada podem ser criados.

4.2.3 Balanceamento de Carga

Após a movimentação dos dados para a memória do device e a definição do número de blocos e do tamanho do bloco a ser utilizado, é necessário efetuar o balanceamento de carga das cuda-threads. Essa etapa consiste em definir exatamente a quantidade de trabalho que cada cuda-thread irá executar. A ideia aqui é que cada cuda-thread tenha a mesma quantidade de trabalho de qualquer outra cuda-thread ou, no máximo, uma unidade de trabalho a mais para se executar. No caso, uma unidade de trabalho representa o cálculo de um z_j.

Duas situações pode ser usadas como exemplo, uma onde o número de observações é maior que o número de cuda-threads e outra onde o contrário acontece. Na primeira situação, pode-se tomar como exemplo um caso onde o número de observações é igual a 66 e temos 32 cuda-threads para o cálculo. Ocorre que as primeiras duas cuda-threads irão executar o cálculo de 3 observações, enquanto que as outras 30 cuda-threads irão executar apenas duas computações. Na segunda situação, pode-se imaginar 4 observações para as mesmas 32 cuda-threads. Nesse caso, as 4 primeiras cuda-threads farão um cálculo cada, enquanto que todas as outras não efetuarão nenhuma computação.

4.3 O algoritmo do XCM-GPU

Uma vez que todas as etapas de inicialização estejam concluídas, o algoritmo XCM-GPU é iniciado. Na versão paralela, a chamada da biblioteca de minimização irrestrita L-BFGS para o cálculo da função objetivo e de seu gradiente são substituídas por uma função que recebe todos os parâmetros e invoca o *kernel* de execução na GPU. As chamadas de kernel tomam como base o endereçamento das cuda-threads e o valor recebido na etapa de balanceamento de carga para efetuar apenas o cálculo

dos z_j's pelos quais a cuda-thread é responsável. Dessa forma, não é necessário nenhum controle de concorrência aos dados, pois cada cuda-thread acessa somente os endereços de memória a ela destinados.

Quando as cuda-threads terminarem sua execução, as diversas parcelas do cálculo da função objetivo f e de seu gradiente estarão "espalhadas"pela memória global da GPU. No caso da função objetivo, apenas um valor deve ser retornado para a memória do host. Este conceito está associado à ideia de *Redução*.

Redução é um problema muito comum, tanto para algoritmos paralelos como sequenciais. Como exemplo, suponha que deseja-se adicionar os números 1, 2, 10 e 12. A fim de fazer isso, em paralelo, uma thread pode calcular o valor de 1+2, e uma outra thread pode calcular o valor de 10+12. Após esses cálculos estrem completos, uma outra thread pode adicionar os resultados das adições anteriores. O número total de adições permanece o mesmo, mas as duas primeiras adições são feitas "ao mesmo" tempo. Quando o número de adições se torna muito grande, o benefício da redução se torna aparente.

Em relação a função objetivo, apenas uma operação de redução é necessária. No entanto, no caso da função gradiente, diversas reduções (mais precisamente, $p * q$ reduções, onde p é a dimensão do espaço do problema e q é o número de centroids) são necessárias para que esse vetor seja corretamente calculado.

Uma versão resumida do algoritmo XCM-GPU está presente no algoritmo 3 abaixo:

Algoritmo 3 Algoritmo XCM-GPU simplificado

Inicialização: Defina os valores iniciais de $x^0, \varepsilon^1, \tau^1, \gamma^1$ e os valores $0 < \rho_1 < 1, 0 < \rho_2 < 1, 0 < \rho_3 < 1$; Efetue a inicialização do XCM-GPU

$k \leftarrow 1$

while Uma regra de parada não for atingida **do**

Copie a solução x^{k-1} e os parâmetros $\varepsilon^k, \tau^k, \gamma^k$ para a memória da GPU.

Resolva o problema (17) chamando o kernel de execução com $\varepsilon = \varepsilon^k, \tau = \tau^k, \gamma = \gamma^k$ utilizando a solução inicial x^{k-1} . Seja x^k a solução obtida.

$\varepsilon^{k+1} \leftarrow \rho_1 \varepsilon^k$

$\tau^{k+1} \leftarrow \rho_2 \tau^k$

$\gamma^{k+1} \leftarrow \rho_3 \gamma^k$

$k \leftarrow k + 1$

end while

4.4 Arquitetura Orientada a Objetos para Comparação de Algoritmos

Um dos objetivos centrais desse trabalho é fazer a comparação entre o desempenho dos algoritmos XCM e XCM-GPU. Para essa finalidade, foi proposto nesse trabalho uma arquitetura orientada a objetos onde se escreve os códigos comuns do XCM e do XCM-GPU uma única vez na classe IXCM (I de interface). Essa arquitetura é chamada de "Arquitetura Orientada a Objetos para Comparação de Algoritmos".

Na classe base IXCM foi implementado o código comum entre XCM e XCM-GPU. Dentre os códigos comuns estão o método run (que executa o cálculo principal), diversos métodos auxiliares, os casos de teste, os atributos gerais do método XCM e outros.

Na classe derivada XCM foi implementada a versão sequencial do XCM, e na classe derivada XCM-GPU foi implementada a versão GPU do XCM. Essa mesma arquitetura orientada a objetos será usada em outros trabalhos (em andamento), em que serão usadas outras técnicas de paralelismo para o XCM, tais como OpenMP, MPI e Trebuchet.

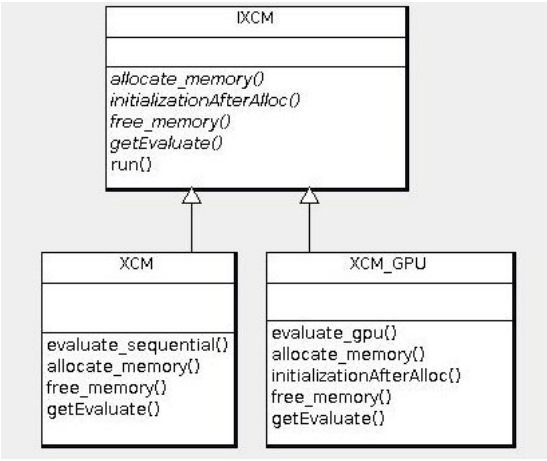


Figura 4.1: Diagrama de Classes da Arquitetura Orientada a Objetos para Comparação de Algoritmos

Os métodos virtuais da classe base IXCM estão listados abaixo.

1. allocate_memory

Esse método é usado para alocar memória usada no algoritimo e salvar a referência dessa alocação nos ponteiros, que são membros da classe IXCM. Devido a requisições do tipo de paralelismo usado, pode ser necessário um tipo diferente de alocação de memória. Por isso esse método deve ser virtual.

2. initializationAfterAlloc

   Esse método é chamado após a alocação de dados, leitura de disco e preenchimento dos dados de observação. No caso de XCM-GPU esse método é necessário, para copiar os dados de observação para a memória do dispositivo (GPU).

3. free_memory

   Esse método é chamado para se liberar a memória. Como a alocação de dados pode variar dependendo da implementação do algoritmo XCM, a liberação de memória também precisa ser customizada.

4. getEvaluate

   Esse método é usado para retornar o ponteiro de função "evaluate" que será usada como argumento da função global LBFGS. O método não "run" pergunta qual o ponteiro da função "evaluate" chamando o método "getEvaluate", e chama o LBFGS com esse valor. O resultado é que cada implementação (sequencial ou paralela) do XCM pode implementar como quiser o cálculo da função de custo e de sua derivada. Par isso, basta escrever o código da função estática que realiza esse cálculo e retornar o ponteiro dessa função pelo método getEvaluate. No caso da classe XCM, o retorno é o método estático evaluate_sequential. No caso da classe XCM-GPU o retorno é o método estático evaluate_gpu.

Capítulo 5

Resultados Computacionais

Os resultados computacionais apresentados foram obtidos através da implementação do XCM-GPU no servidor *coyote* do Labotim, o Laboratório de Otimização do PESC/COPPE/UFRJ. A biblioteca LBFGS, disponível em [3], foi utilizada para calcular as minimizações irrestritas. O sistema operacional utilizado foi o OpenSuse 11.2, com a versão 4.0 da arquitetura CUDA. A visualização dos resultados foi gerada através do software Octave.

Nos testes operacionais, os pontos iniciais do algoritmo iterativo foram definidos usando-se o ponto médio das observações para definir um ponto base. Em seguida, os outros pontos do conjunto X inicial foram gerados através de uma perturbação nesse ponto base multiplicando-se um número aleatório com distribuição uniforme entre zero e um com o desvio-padrão das observações e um fator de expansão proporcional a ordem de grandeza dos valores das coordenadas das observações.

Vale ressaltar que esse método para a escolha do ponto inicial foi puramente intuitiva e, utilizando-se uma heurística mais desenvolvida é possível obter uma estimativa muito melhor para os pontos iniciais. Mesmo assim, os resultados apresentados são extremamente satisfatórios.

Todos os fatores do algoritmo foram inicializados com valor igual a 0.5.

5.1 Problemas de Pequeno Porte

Para verificar a correta implementação do algoritmo XCM-GPU (e da própria implementação base do XCM), dois pequenos casos de teste foram utilizados. O primeiro é um exemplo extremamente simples, capaz de ser resolvido rapidamente sem o uso de um computador. Este teste consiste na definição de 4 pontos no $\mathbb{R}^2$, localizados nas coordenadas $(0,1), (1,3), (6,0)$ e $(8,0)$ que devem ser agrupados em dois conjuntos distintos.

Esse caso de teste pode ser visto na figura 5.1. Dessa forma, temos m, o número de observações, igual a 4, n, a dimensão do espaço das observações, igual a 2 e q, o número de centroids, igual a 2.

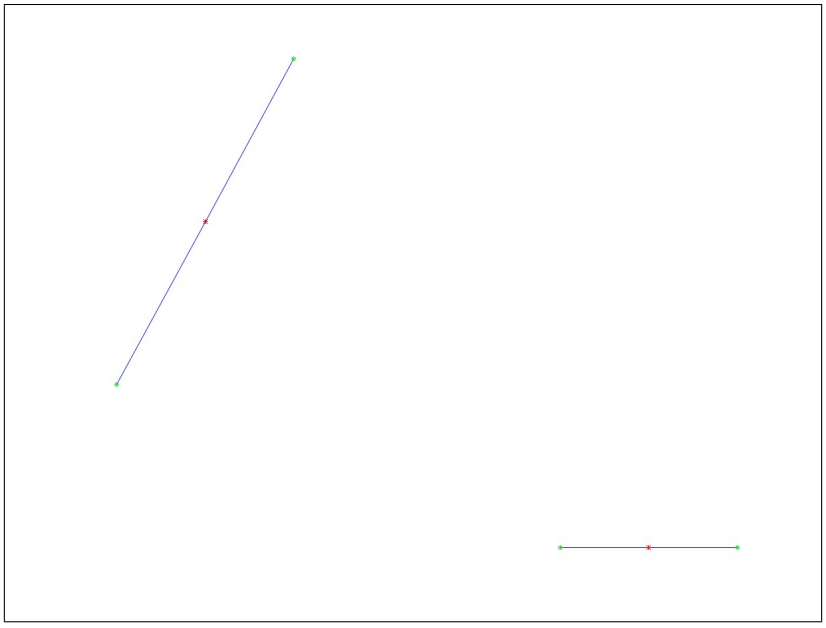


Figura 5.1: Caso de teste extremamente simples.

O esquema de cores definido nas figuras resultantes dos casos de teste é definido da seguinte forma:

- Pontos em verde marcam o posicionamento das observações.
- Pontos em vermelho marcam o posicionamento calculado dos centroids.
- As linhas em azul fazem as ligações entre as observações e os centroids a que estão associados.

Todo o desenvolvimento do XCM-GPU foi feito utilizando esse caso de teste como verificação da implementação, levando em consideração a acurácia do resultado. Em questão de performance, dado que o problema é extremamente simples, a versão sequencial XCM leva vantagem sobre o XCM-GPU. O tempo do cálculo sequencial é de 0.16 segundos, enquanto que a versão paralela levou 0.52 segundos para ser calculada utilizando-se 4 cuda-threads.

O caso de teste seguinte, denominado problema de Demyanov é comumente encontrado na literatura. Esse problema consiste de 41 observações ($m = 41$) no $\mathbb{R}^2$ ($p = 2$) que devem ser agrupados em três conjuntos ($q = 3$). A figura 5.2, mostra os resultados desse caso de teste, para $q = 3$. Na figura 5.3, o mesmo caso de teste resolvido para $q = 6$ pode ser visto.

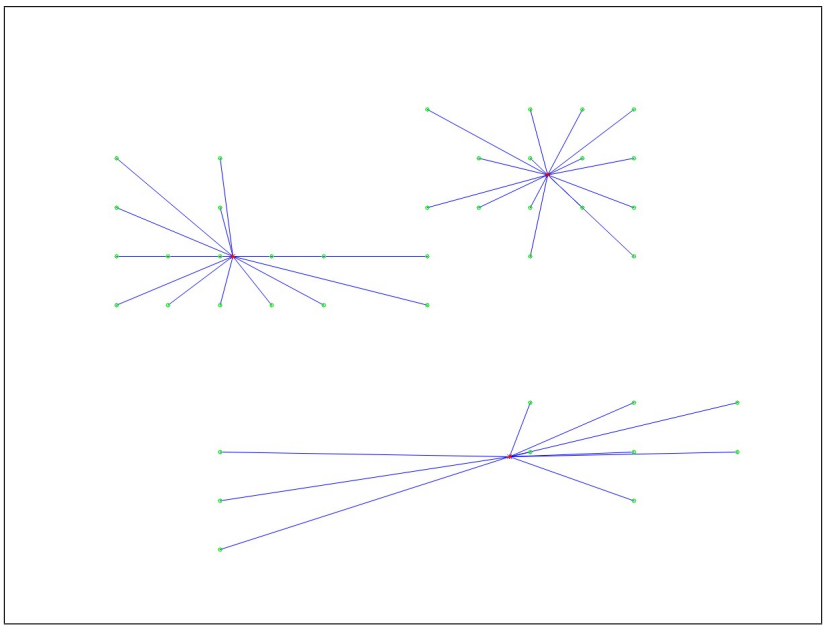


Figura 5.2: O caso de teste do Problema de Demyanov (q = 3).

No problema de Demyanov, o valor da função objetivo calculado é de 99.4167, tanto na versão sequencial quanto na paralela, sendo esse valor igual ao valor ótimo da solução do problema de acordo com a literatura. Nota-se, portanto, que não há perda de precisão entre as implementações sequencial e paralela. O tempo sequencial de resolução é de 0.15 segundos e o tempo de execução paralela utilizando-se 4 cuda-threads é de 2.19 segundos. Como a quantidade de cálculos é relativamente pequena, é natural que a versão sequencial tenha um desempenho melhor que a versão paralela

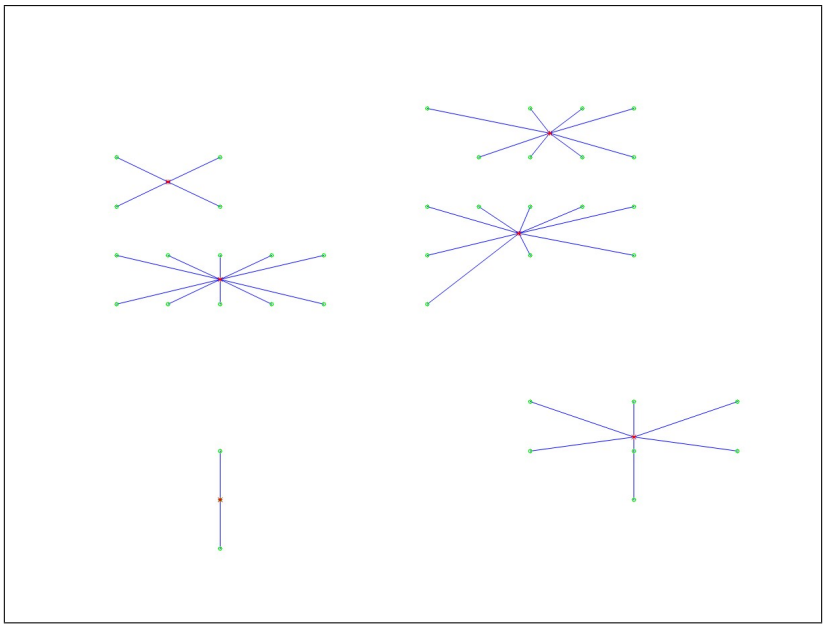


Figura 5.3: O caso de teste do Problema de Demyanov (q = 6).

dado o *overhead* associado a esse tipo de implementação.

5.2 Problemas de Médio e Grande Porte

Após a validação dos resultados dos problemas de pequeno porte, foram considerados os problemas de tamanho médio, com 3038 e 15112 observações e de grande porte, com 85900 observações. O problema de 3038 observações é relacionado ao problema clássico do caixeiro viajante da literatura, disponibilizado na biblioteca TSPLIB [4]. No caso, cada observação representa uma cidade em um plano euclidiano com $p = 2$. As imagens resultantes do caso de teste com 3038 observações podem ser vistas a seguir, nas figuras 5.4, 5.5, 5.6, 5.7, 5.8, 5.9.

Os resultados das comparações de tempo de execução entre as implementações sequencial e paralela podem ser vistas na tabela 5.1. O tempo da implementação paralela foi calculado usando-se 4096 cuda-threads em todos os casos de teste. Na figura 5.10 pode-se observar um gráfico comparando os tempos de execução das implementações sequencial e paralela.

Os resultados do caso de teste para o problema com 15112 observações podem ser vistos nas figuras 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, também considerando 5,

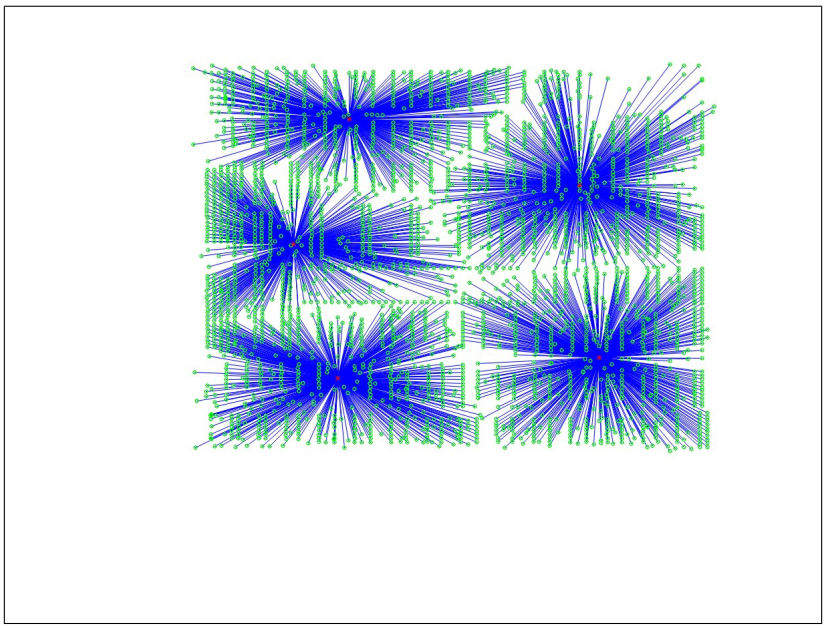


Figura 5.4: O caso de teste do Problema TSPLIB-3038 (q = 5).

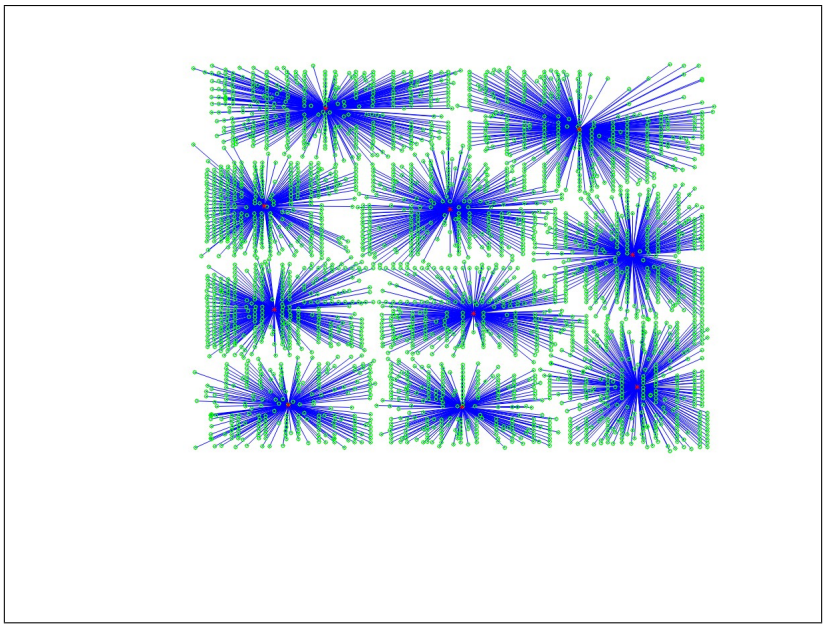


Figura 5.5: O caso de teste do Problema TSPLIB-3038 (q = 10).

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	2.19	1.85	1.18
10	8.99	2.53	3.55
15	13.5	3.23	4.18
20	26.9	6.71	4.01
25	50.43	9.3	5.42
30	70.86	10.32	6.87

Tabela 5.1: Resultados de Tempo e Speed Up para o problema TSPLIB-3038

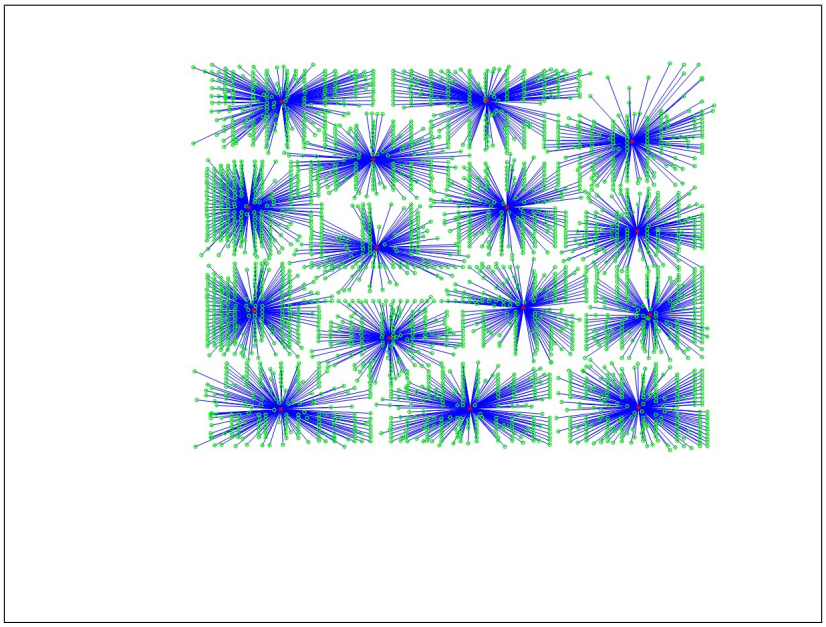


Figura 5.6: O caso de teste do Problema TSPLIB-3038 (q = 15).

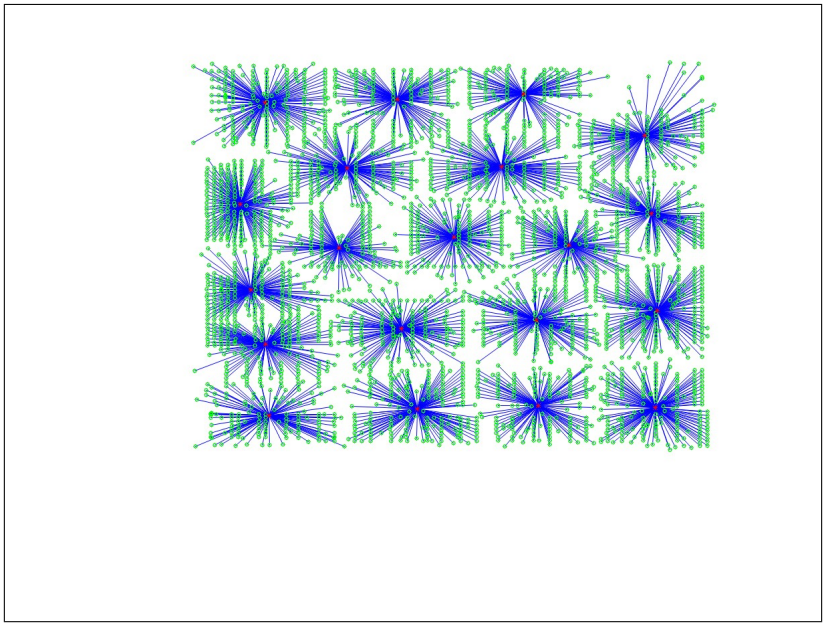


Figura 5.7: O caso de teste do Problema TSPLIB-3038 (q = 20).

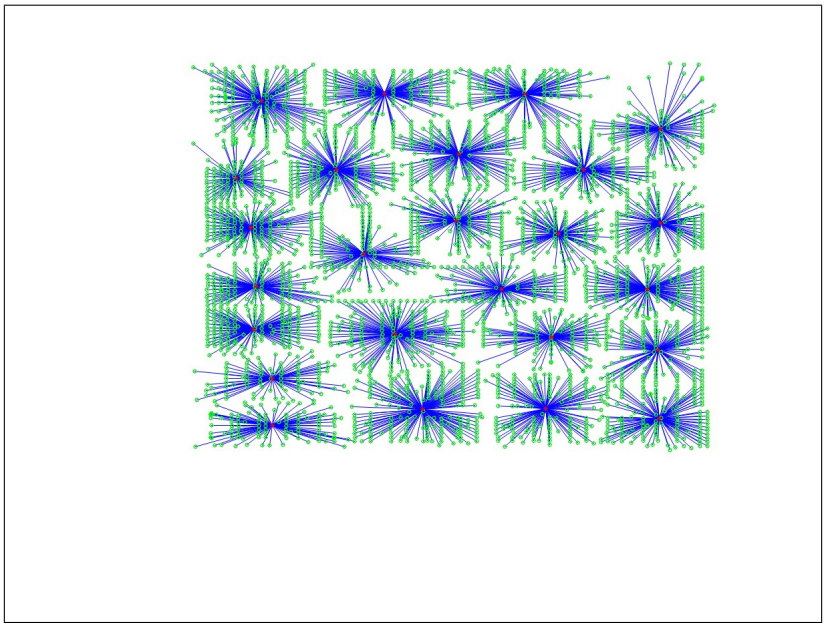


Figura 5.8: O caso de teste do Problema TSPLIB-3038 (q = 25).

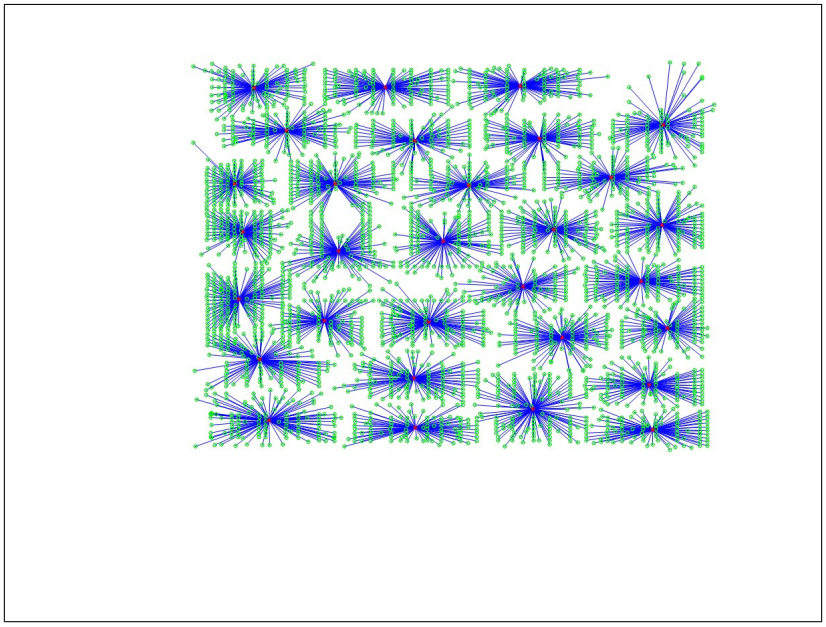


Figura 5.9: O caso de teste do Problema TSPLIB-3038 (q = 30).

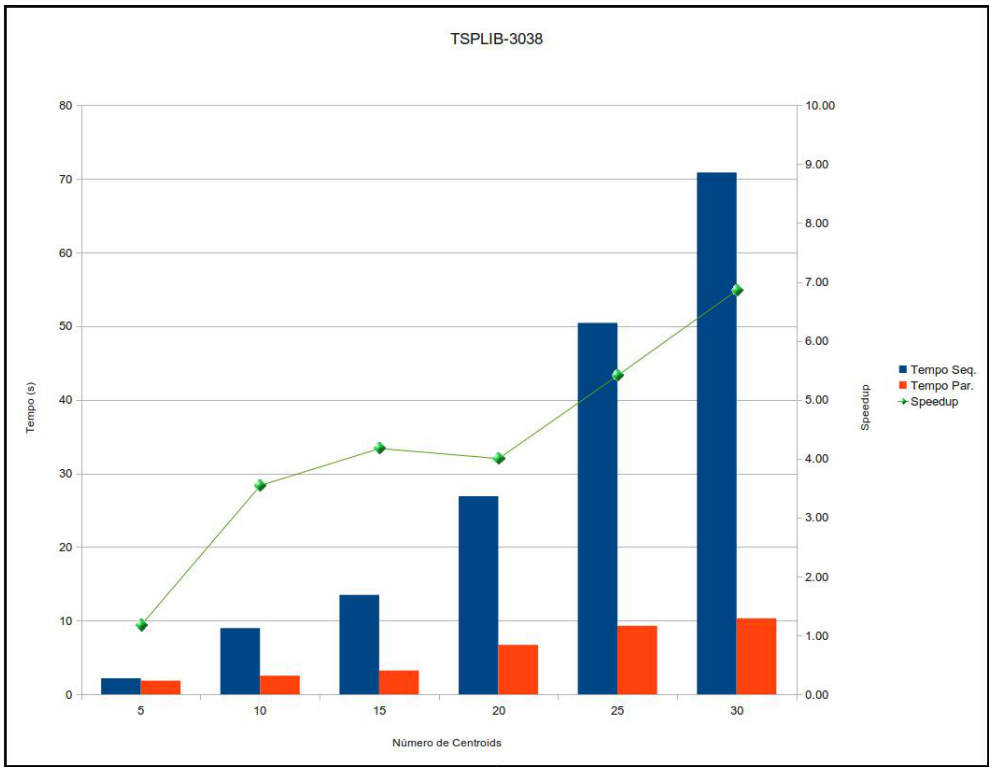


Figura 5.10: Gráfico dos resultados do problema TSPLIB-3038.

10, 15, 20, 25 e 30 centroids, respectivamente.

Os resultados das comparações de tempo de execução entre as implementações sequencial e paralela podem ser vistos na tabela 5.2. Como no caso anterior, as mesmas 4096 cuda-threads foram usadas na medição de tempo. Na figura 5.17 é possível observar o gráfico com os resultados de tempo.

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	11.18	2.49	4.49
10	39.64	4.47	8.87
15	67.94	10.43	6.51
20	131.25	14.8	8.87
25	216.83	22.51	9.63
30	273.73	28.89	9.47

Tabela 5.2: Resultados de Tempo e Speed Up para o problema 15112

Para o problema com 85900 observações, foram medidos os tempos como nos casos de teste anteriores, mas devido a grande densidade do pontos, foram geradas apenas as imagens dos resultados da computação de 5 e 10 centroids, como pode ser visto nas figuras 5.18 e 5.19. Os resultados das medições de tempo encontram-se na

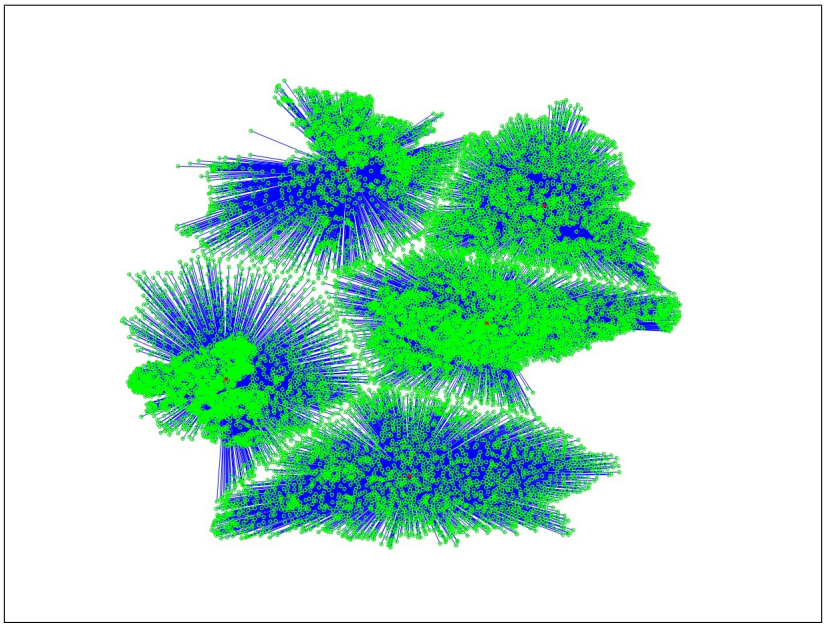


Figura 5.11: O caso de teste do Problema com 15112 observações (q = 5).

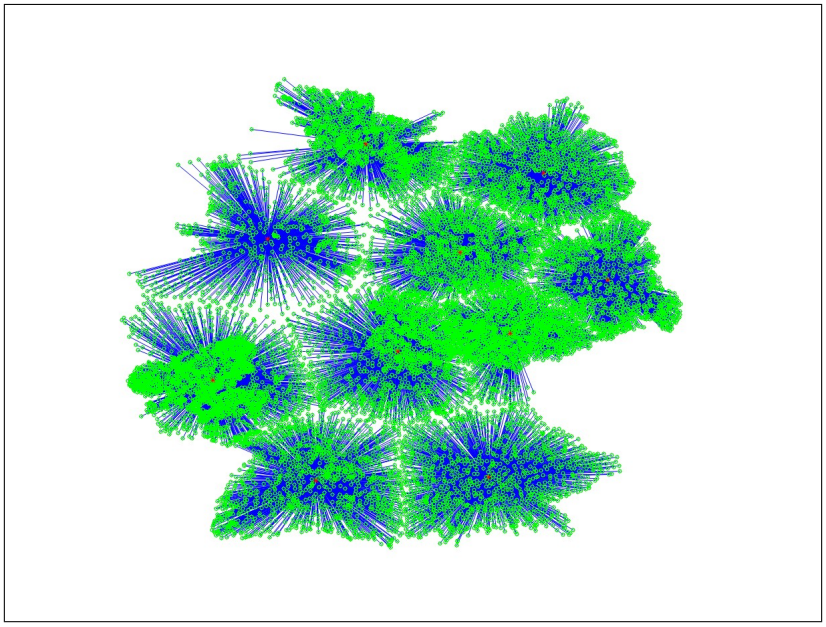


Figura 5.12: O caso de teste do Problema com 15112 observações (q = 10).

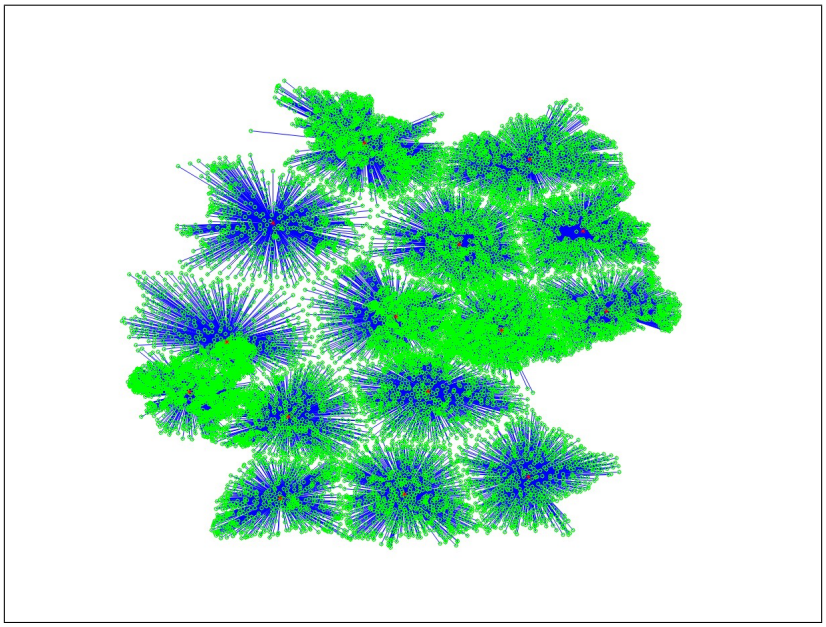


Figura 5.13: O caso de teste do Problema com 15112 observações (q = 15).

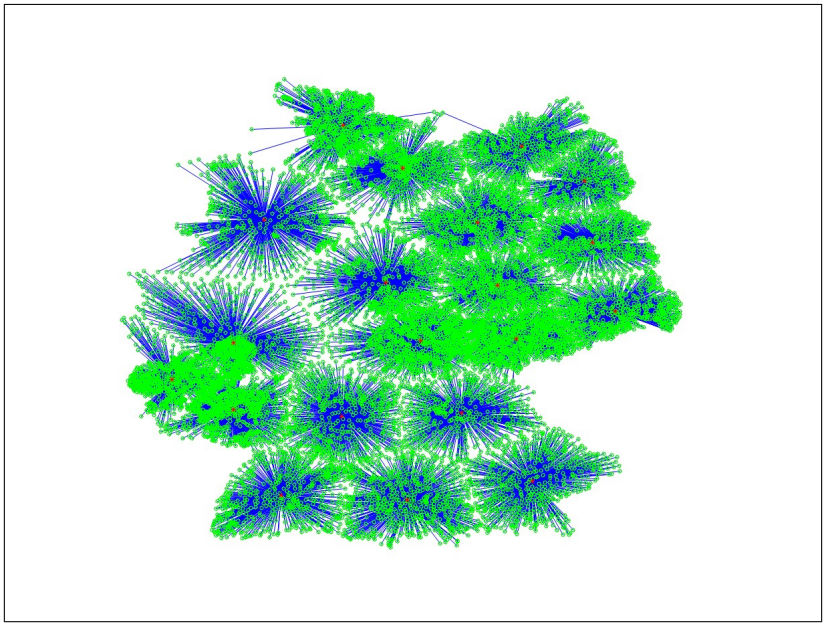


Figura 5.14: O caso de teste do Problema com 15112 observações (q = 20).

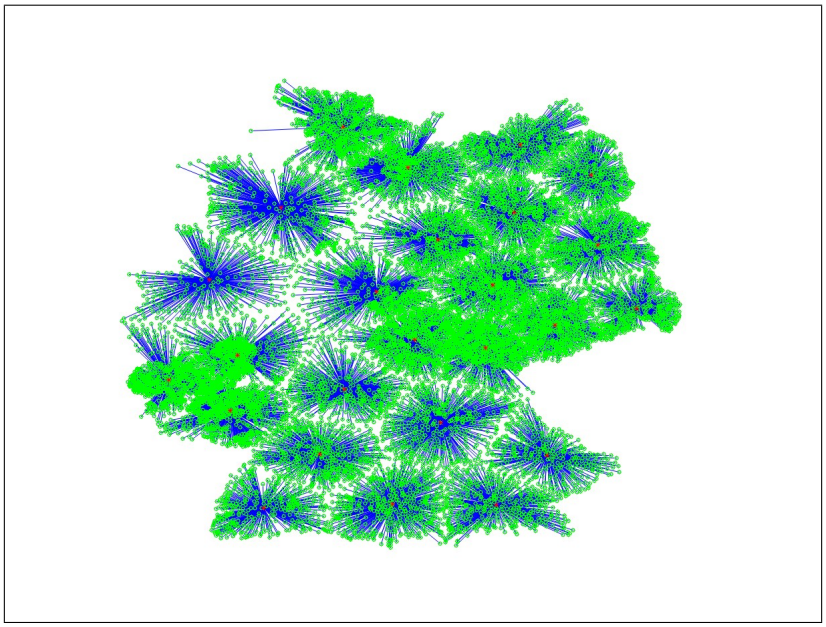


Figura 5.15: O caso de teste do Problema com 15112 observações (q = 25).

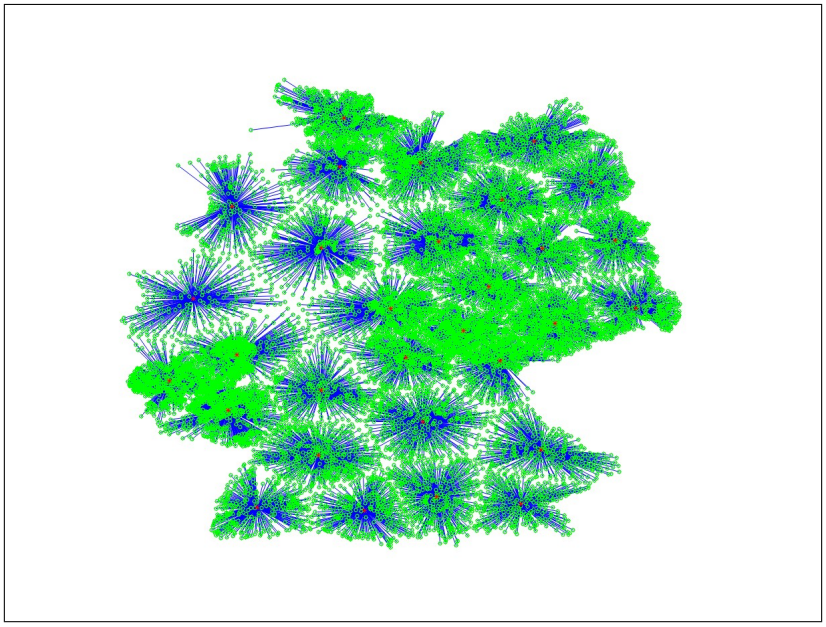


Figura 5.16: O caso de teste do Problema com 15112 observações (q = 30).

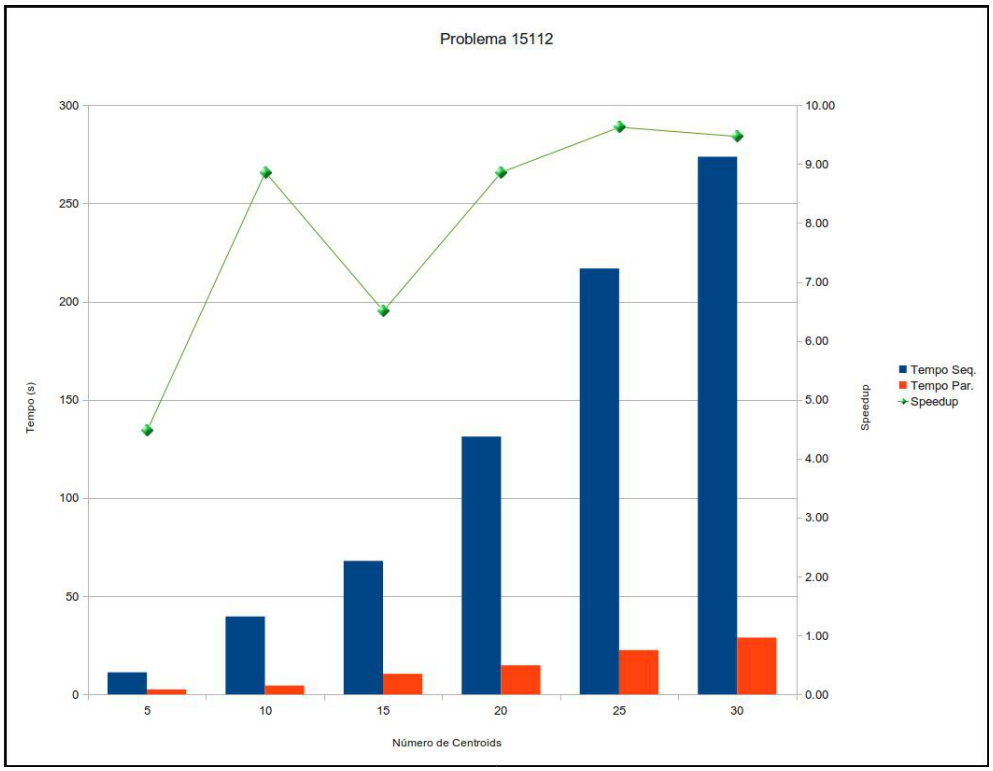


Figura 5.17: Gráfico dos resultados do problema 15112.

tabela 5.3. O gráfico com a comparação dos tempos de execução para esse problema é visto na figura 5.20.

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	63.53	6.39	9.94
10	233.26	19.88	11.73
15	682.37	51.54	13.24
20	964.74	90.13	10.70
25	1663.8	131.63	12.64
30	2577.29	189.75	13.58

Tabela 5.3: Resultados de Tempo e Speed Up para o problema 85900

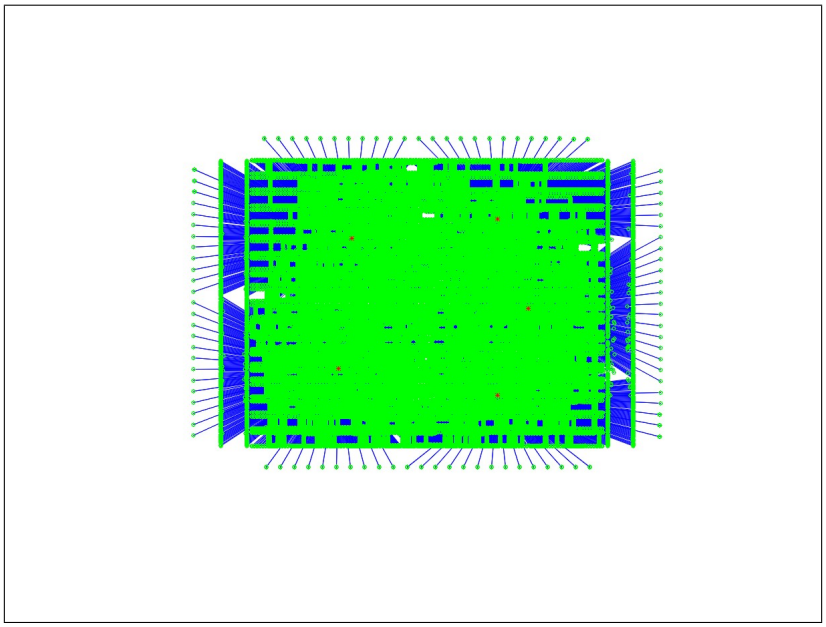


Figura 5.18: O caso de teste do Problema com 85900 observações (q = 5).

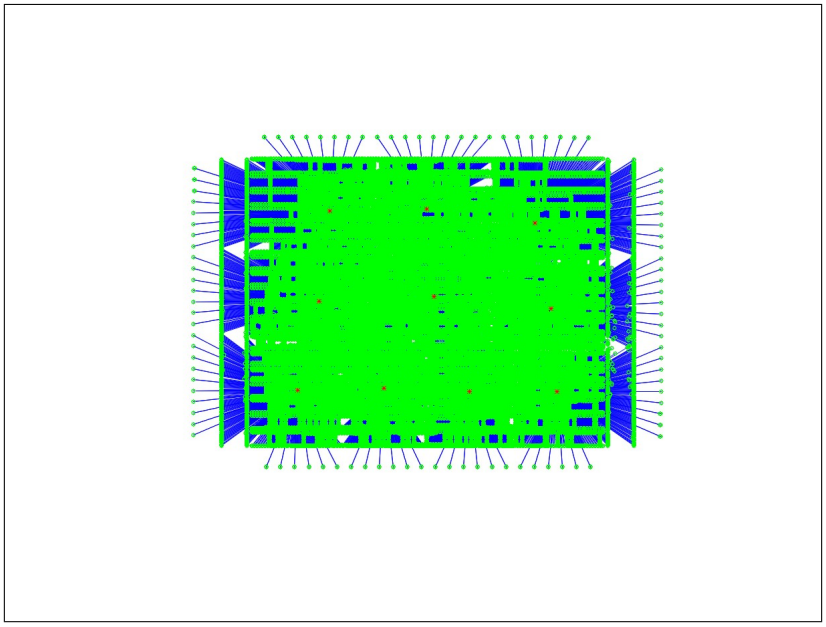


Figura 5.19: O caso de teste do Problema com 85900 observações (q = 10).

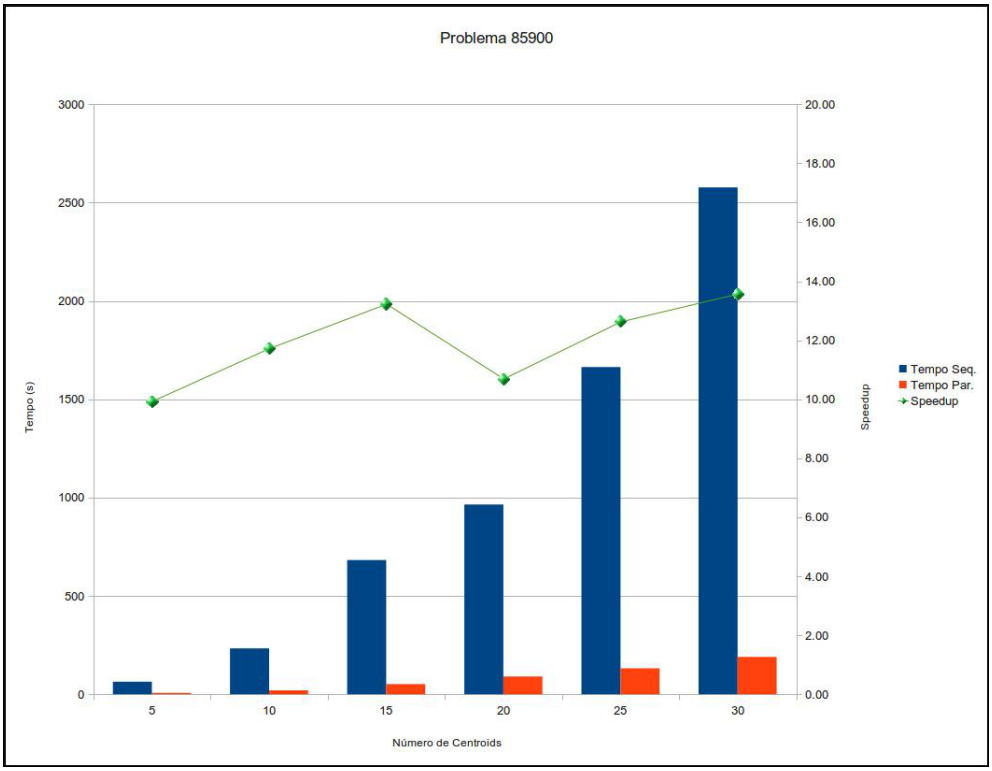


Figura 5.20: Gráfico dos resultados do problema 85900.

Capítulo 6

Conclusão

O problema de agrupamento (*clustering*), formulado segundo o critério de mínima de soma de quadrados, é um problema numérico muito importante, útil e de difícil solução (NP-Hard).

O método de clusterização proposto por Xavier, conhecido por XCM (Xavier Clustering Method), introduz um melhora significativa na resolução do problema de agrupamento, com excelentes resultados em relação aos métodos tradicionais.

Em uma análise detalhada do XCM, as suas partes paralelizáveis foram identificadas. Nesta tese optou-se por explorar a GPU como a arquitetura de hardware para a computação paralela dessas partes do algoritmo XCM. Dessa forma, foi proposto o método que chamamos de XCM-GPU.

Para a comparação dos resultdos do XCM com o XCM-GPU, foi proposta uma "Arquitetura Orientada a Objetos para Comparação de Algoritmos", em que numa classe base IXCM foi implementado o código comum entre XCM e XCM-GPU. Na classe derivada XCM foi implementada a versão sequencial do XCM, e na classe derivada XCM-GPU foi implementada a versão GPU do XCM. Essa mesma arquitetura orientada a objetos será usada em outros trabalhos (em andamento), em que serão usadas outras técnicas de paralelismo para o XCM, tais como OpenMP, MPI e Trebuchet.

Os resultados experimentais, como esperado, mostraram grade Speed Up da versão XCM-GPU em relação a versão XCM. Nos testes realizados, o maior Speed

Up foi maior que 13 vezes. Ressalte-se que não ocorreu perda de precisão numérica na versão paralela.

A experiência adquirida na realização desse trabalho permite enumerar algumas recomendações gerais para o exercício da atividade de desenvolvimento de software científico:

- Pode-se usar orientação a objetos (OO) para computação científica orientada a desempenho, desde que se observe alguns cuidados.
  - Os arrays mono devem ser implementados como arrays canônicos (double*).
  - Os arrays multi-dimensionais devem ser implementados como arrays canônicos (double*), com formula de inteiros para conversão de indices multi-dimensionais em indices mono-dimensionais.
  - Deve-se evitar de estruturas de dados complexas (como as classes vector ou deque de C++), especialmente para armazenamento de dados em ponto flutuante (double).
  - É permitido o uso de estruturas OO (herança, late bind e outras), desde que não se atue na camada de representação de arrays de dados. Isso é o que foi feito nas classes IXCM , XCM e XCM-GPU, para a comparação direta de desempenho entre os algorítmos sequencial e paralelo.
- O método XCM requer a chamada de uma rotina externa de otimização irrestrita. Na implementação desta tese, foi utilizada a rotina LBFGS. Existe mais de uma implementação possível para LBFGS, e o desempenho final é fortemente dependente da escolha dessa rotina.

  Neste trabalho, foram analizadas três opções (LBGFS do Nocedal, GSL-BFGS e ALGLIB). A opção que mostrou melhores resultados foi a primeira. Essa foi a opção usada para gerar os resultados experimentais.

- Existem no mercado a venda GPU's que implementam em hardware o ponto flutuante em precisão simples apenas (float). A precisão numérica e o próprio desempenho do método é muito prejudicado no caso de ser executado em GPU's que não possuem suporte a números de ponto flutuante com precisão dupla (double). Durante grande parte do desenvolvimento do trabalho, esse foi o principal obstáculo para se obter ganhos de performance consideráveis, já que o único hardware disponível não possuía essa precisão. Em uma fase posterior do trabalho, em 2012, com um computador novo instalado no Labotim, cujo modelo é conhecido como "coyote", o trabalho pode ter prosseguimento de forma bem mais produtiva.

- O desenvolvimento de software para GPU é muito sofisticado, e requer depuração minuciosa (inclusive pelo fato de que a ferramenta de debug não funciona para o código que é executado pela GPU).

Em 2009, GPU's com suporte a precisão dupla tinham custos relativamente altos, mas a evolução constante dessa tecnologia já permite encontrar GPU's com precisão dupla a custos acessíveis. A idealização desse trabalho, pioneiro na utilização da arquitetura CUDA na linha de pesquisa de Otimização do PESC/COPPE/UFRJ, motivou a especificação e aquisição do hardware com suporte a CUDA no computador denominado "coyote" do Laboratório de Otimização (LABOTIM). O ambiente de desenvolvimento em CUDA foi totalmente implementado para esse trabalho, e está disponível para que outros pesquisadores possam desenvolver seus trabalhos com CUDA.

6.1 Trabalhos Futuros

O XCM-GPU ainda permite desenvolvimentos futuros, buscando-se ainda mais melhora de performance. A implementação feita nesse trabalho não tirou pleno proveito das potencialidades da GPU. Abaixo citam-se algumas técnicas que podem ser usadas com objetivo de melhorar a performance do XCM-GPU, que podem ser usadas

em pesquisas futuras.

- o conceito de memória local de GPU's pode ser aplicado em futuras implementações ou até mesmo a utilização em mais de uma GPU pode ser empregada na obtenção de resultados ainda melhores.

- Como a capacidade de processamento em GPU é enorme, pode-se ainda considerar problemas muito maiores do que os apresentados neste trabalho, compostos até mesmo por centenas de milhares de pontos. Inclusive, pode-se vislumbrar problemas onde a capacidade de armazenamento da memória da GPU não é suficiente para armazenar todos os dados das observações.

- Outras soluções podem ser empregadas de forma a criar um método de paralelismo misto, utilizando-se threads na CPU e na GPU para buscar mais eficiência na resolução dos problemas de agrupamento.

Referências Bibliográficas

[1] P. Brucker. On the complexity of clustering problems. *Optimization and Operations Research*, 1978.

[2] Nvidia Corporation. *CUDA C Programming Guide*. 4 edition, 2011.

[3] Jorge Nocedal and Naoaki Okazaki. http://www.chokkan.org/software/liblbfgs/.

[4] G. Reinelt. Tsplib:a traveling salesman library. *ORSA Journal of Computing*, page 376?384, 1991.

[5] J. Sanders and E. Kandrot. *CUDA By Example*. 1 edition, 2010.

[6] L.C.F. Sousa. Desempenho computacional do método de agrupamento via suavização hiperbólica. *M.Sc. Thesis, COPPE/UFRJ, Rio de Janeiro, RJ*, 2005.

[7] A. E. Xavier. Penalização hiperbólica: Um novo método para resolução de problemas de otimização. *M.Sc. Thesis, COPPE/UFRJ, Rio de Janeiro, RJ*, 1982.

[8] A. E. Xavier. Hyperbolic penalty: a new method for nonlinear programming with inequalities. *International Transactions in Operational Research*, 8 (6):659–671, 2001.

[9] A. E. Xavier. The hyperbolic smoothing clustering method. *Pattern Recognition*, 43 (3):731–737, 2010.

[10] A. E. Xavier and A. A. F. Oliveira. Optimal covering of plane domains by circles via hyperbolic smoothing. *Journal of Global Optimization*, 31 (3)(3):493–504, 2005.

[11] A. E. Xavier and V. L. Xavier. Solving the minimum sum-of-squares clustering problem by hyperbolic smoothing and partition into boundary and gravitational regions. *Pattern Recognition*, 44 (1):70–77, 2011.