UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Pós-Graduação em Ciência da Computação

Thiago Alexandre Domingues de Souza

Uma Solução Paralela de Agrupamento de Dados em GPU

UNESP
2017

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Pós-Graduação em Ciência da Computação

Thiago Alexandre Domingues de Souza

Uma Solução Paralela de Agrupamento de Dados em GPU

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

Orientador:
Prof. Dr. Aleardo Manacero Jr.

UNESP
2017

Thiago Alexandre Domingues de Souza

Uma Solução Paralela de Agrupamento de Dados em GPU

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

BANCA EXAMINADORA:

Prof. Dr. Aleardo Manacero Jr.
UNESP – São José do Rio Preto
Orientador

Prof. Dr. Alexandro José Baldassin
UNESP – Rio Claro

Prof. Dr. Paulo Sérgio Lopes de Souza
USP – São Carlos

Aos meus pais.

Agradecimentos

Agradeço primeiramente a Deus, por conceder o privilégio de chegar até aqui.

Aos meus pais, Eduvirges e José Antônio, que sempre estiveram ao meu lado, me incentivando e oferecendo todo suporte, amor e carinho ao longo desta jornada.

A todos os familiares, que sempre contribuíram e me apoiaram.

À minha esposa, Tatiana, pelo constante amor, carinho, compreensão e paciência, principalmente nos momentos difíceis desta caminhada.

Ao meu orientador, Prof. Dr. Aleardo Manacero Jr., pelos ensinamentos que levarei por toda a vida. Também agradeço pela confiança e compreensão, pois a distância física que nos separa e o intervalo até meu retorno aos bancos acadêmicos não impediram que acreditasse em mim.

À Profª. Drª. Renata Spolon Lobato, que desde os primeiros anos de graduação sempre incentivou minha busca pelo conhecimento.

Aos colegas do GSPD (Grupo de Sistemas Paralelos e Distribuídos), pelo auxílio para que este projeto fosse desenvolvido. Em especial ao Lucas Cazarote, Gabriel Covello e Mário Camara, pelas incontáveis assistências ao longo deste trabalho.

À IBM e aos colegas de trabalho, pela autonomia e flexibilidade em minhas atividades laborais, que permitiram a condução deste mestrado.

Meus sinceros agradecimentos a todas pessoas e instituições que contribuíram para a minha formação acadêmica, profissional e cidadã.

"Everything around you that you call life was made up
by people that were no smarter than you"

Steve Jobs

Resumo

A indústria de tecnologia da informação tem permitido uma explosão de dados coletados nos últimos anos. Isso ocorreu, entre outros fatores, pela expansão do acesso à rede por meio de uma infinidade de equipamentos. Uma análise detalhada dos dados armazenados pode, por exemplo, extrair informações valiosas sobre o comportamento dos indivíduos, permitindo uma relação personalizada de acordo com os interesses dos usuários. Essa tarefa pode ser feita usando algoritmos de agrupamento de dados. Porém, esse é um processo que requer grande esforço computacional tanto pela ordem de complexidade dos algoritmos existentes como pelos crescentes volumes processados. Nesse contexto, execuções sequenciais não são viáveis e sua paralelização é o caminho natural. Isso exige remodelar algoritmos para explorar o potencial de plataformas massivamente paralelas, de acordo com as particularidades da arquitetura alvo. Neste trabalho se propõe uma implementação paralela do algoritmo *Fuzzy Minimals* para GPU, como uma solução de alto desempenho e baixo custo para contornar dificuldades frequentes no agrupamento de dados. Com o objetivo de avaliar o desempenho de nossa solução, também desenvolvemos versões paralelas em MPI e OpenMP. Nossos experimentos mostram que a solução para GPU alcança resultados expressivos com um baixo custo, mantendo uma precisão significativa.

Palavras-chave: Agrupamento de dados, algoritmos paralelos, lógica *fuzzy*, GPU

Abstract

IT industry has witnessed an explosion of data collected for the past few years. This took place, among other factors, due to the expansion of network access through several devices. For example, a detailed analysis of the stored data can extract some valuable information about human behaviors, allowing a customized experience that matches the interests of users . This task can be performed by clustering algorithms. However, this is a time-consuming process due to the asymptotic complexity of existing algorithms and the increasing volumes of data processed. In this context, sequential executions are not feasible and their parallelization is the natural path. This requires redesigning algorithms to take advantage of massively parallel platforms according to the particularities of targeted architectures. In this paper, it is proposed a novel parallel implementation of the Fuzzy Minimals algorithm on GPU, as a high-performance low-cost solution for common clustering issues. In order to evaluate the performance of our implementation, we have also designed parallel versions using MPI and OpenMP. Our experiments show that our parallel solution on GPU can achieve a high performance at a low cost, preserving a significant accuracy.

Keywords: Clustering, Parallel, Fuzzy Logic, GPU

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os avanços na tecnologia da informação têm permitido coletar grandes quantidades de dados. Previsões indicam que esse volume deve aumentar substancialmente nos próximos anos, já que o tráfego na Internet em 2020 deve ser 95 vezes maior que o registrado em 2005 (3). Uma análise criteriosa desses dados pode extrair informações valiosas sobre eles. Grandes conjuntos de dados têm sido amplamente utilizados para identificar tendências e recomendar produtos ou serviços, proporcionando um tratamento individualizado aos usuários (4).

Algoritmos de classificação desempenham um papel importante nesse contexto. Se os dados estão previamente rotulados, essa classificação é frequentemente denominada como aprendizado supervisionado. Do lado oposto, o aprendizado não-supervisionado, obtido sem conhecimento prévio sobre os dados, apresenta os maiores desafios da área. Esse método é geralmente utilizado por algoritmos de agrupamento de dados (*data clustering*).

O agrupamento de dados pode ser definido como a tarefa de agrupar objetos em *clusters* (grupos) em razão de sua semelhança. Intuitivamente, objetos de um mesmo grupo possuem maior semelhança entre si do que objetos de grupos distintos. Esse método tem aplicações nos mais variados campos de pesquisa, como biologia (5), física (6), economia (7) e química (8).

A organização de objetos em grupos exige identificar as características fundamentais que os descrevem. Por exemplo, os carros podem ser agrupados em função de sua potência, consumo de combustível, tamanho, preço, etc. Cada um desses atributos possui um valor numérico, que normalmente é armazenado em um vetor. Assim, a semelhança entre objetos é avaliada por meio de métricas que estabeleçam a distância entre eles, isto é, objetos mais próximos têm maior similaridade, logo são atribuídos a um mesmo grupo.

1.1 Motivação

Agrupamento de dados é um problema que exige grande esforço computacional. Em geral, os algoritmos da área são baseados na minimização de uma função objetivo que mede a semelhança entre os objetos. Além disso, os volumes crescentes de dados analisados têm impacto significativo no desempenho dessas aplicações, tornando inviável sua execução sequencial. Por exemplo, em (9) é proposta uma solução paralela de agrupamento de dados em grafos usando o *framework* Apache Giraph, que reduz seu tempo de execução de aproximadamente 133 minutos para cerca de 16 minutos. Essa redução pode representar a análise de mais conjuntos de dados ao final do dia. Já em outras aplicações, em que o tempo de execução é crítico, como sistemas de detecção de intrusão ou de previsão do tempo, isso pode viabilizar a análise dos resultados em tempo hábil. Por esse motivo, a paralelização é o caminho natural desses algoritmos.

Para superar esses desafios, diversos algoritmos de agrupamento de dados foram propostos, com especial atenção ao seu desempenho (2). Existe uma vasta literatura em melhorias em torno do algoritmo *K-Means* (10), devido a sua simplicidade e eficiência em agrupar certos conjuntos de dados. As primeiras soluções paralelas eram baseadas no modelo de trocas de mensagens, porém diversas outras abordagens também já foram propostas. Mais recentemente, alguns trabalhos avaliaram outras técnicas e plataformas como OpenMP (11), GPUs (12) e Intel Xeon Phi (13).

Entretanto, essas soluções não alteram o comportamento do *K-Means*, principalmente em relação à atribuição dos objetos aos agrupamentos. Esses algoritmos oferecem um particionamento do tipo *hard*, isto é, cada objeto pertence a somente um grupo. Esse aspecto pode ser um problema quando os grupos não são bem definidos, como, por exemplo, em processamento de imagens ou reconhecimento de padrões, em que os objetos estão sobrepostos.

Diante dessas circunstâncias, algoritmos de agrupamento de dados que adotam a lógica *fuzzy*, como o FCM (14, 15), desempenham um papel importante. De modo semelhante ao *K-Means*, o FCM exige uma definição prévia sobre o número de agrupamentos no conjunto de dados, além disso, está sujeito à escolha dos centróides iniciais. Essas características têm um impacto exponencial no desempenho do algoritmo à medida que aumenta o tamanho do problema, pois diversas execuções são necessárias até que uma solução desejável seja encontrada.

Com o objetivo de contornar essas dificuldades, um novo algoritmo, chamado *Fuzzy Minimals*, foi proposto (16, 17, 18). Esse algoritmo, como o próprio nome sugere, utiliza a lógica *fuzzy* para encontrar agrupamentos sem qualquer conhecimento sobre o número de grupos, o tamanho ou o formato do conjunto de dados. Além disso, os

autores demonstram que essa solução satisfaz os requisitos esperados por bons algoritmos de classificação, como escalabilidade, adaptabilidade e estabilidade na presença de ruídos (19).

Uma versão paralela do algoritmo *Fuzzy Minimals* já foi proposta em (20), porém a solução foi desenvolvida usando o programa Matlab. Embora esse ambiente seja eficiente para processar equações, não é apropriado para executar programas com laços e testes condicionais. Nesse sentido, programas compilados alcançam maiores desempenhos, pois não precisam de interpretadores.

Dentre as plataformas adequadas para resolver problemas que exigem grande esforço computacional, as GPUs têm recebido grande destaque. Diversas aplicações que demandam alto grau de paralelismo têm explorado o alto desempenho e baixo custo desses equipamentos para acelerar suas soluções, alcançando execuções de até algumas centenas de vezes mais rápidas que sua versão sequencial (21). Apesar desses resultados expressivos, alcançá-los não é uma tarefa simples. É preciso conhecer a arquitetura e a organização desses equipamentos, além de suas bibliotecas, para explorar todos os recursos disponíveis neste ambiente.

1.2 Objetivos

Considerando os desafios encontrados na classificação de dados, tanto do ponto de vista de volume de informações a serem processadas quanto em relação às características dos algoritmos existentes, este trabalho tem como objetivo desenvolver uma solução paralela de alto desempenho para o problema de agrupamento de dados. Para isso, foi desenvolvida uma solução inédita, dentro da bibliografia pesquisada, do algoritmo *Fuzzy Minimals*. A solução proposta utiliza a plataforma GPU, pois este equipamento combina um alto desempenho com baixo custo por FLOP *(FLoating Point Operations Per Second)*. A solução foi avaliada tanto em relação ao método sequencial como implementações em MPI e OpenMP.

1.3 Organização

Este trabalho está organizado da seguinte maneira. No Capítulo 2 é realizada a fundamentação teórica em torno de GPUs. Já no Capítulo 3, são discutidos os conceitos relacionados ao agrupamento de dados. O desenvolvimento do projeto proposto é descrito no Capítulo 4. No Capítulo 5 são apresentados os testes e resultados obtidos. Por fim, as conclusões e trabalhos futuros são descritos no Capítulo 6.

Capítulo 2

GPU

A indústria do entretenimento tem exercido papel fundamental, a partir da década de 1970, no desenvolvimento de placas de processamento gráficas. Inicialmente, fabricantes de jogos utilizavam sistemas dedicados conhecidos como *arcade*, cujo circuito impresso incorporava o *software* na própria placa aceleradora, permitindo um melhor desempenho, mas restringindo o número de jogos por sistema. Até o final da década de 1990, fabricantes do setor utilizavam placas aceleradoras 2D, e posteriormente 3D, para melhorar o funcionamento de seus produtos, com algumas características primitivas de processamento de imagens, como desenho de linhas, preenchimento de polígonos e renderização de objetos previamente transformados (22).

Os aceleradores gráficos representaram um avanço no processamento gráfico, mas não auxiliavam a CPU nas atividades que demandam alto poder computacional como transformação e iluminação de objetos. As placas aceleradoras não utilizavam um processador integrado capaz de realizar operações complexas, assim, todo cálculo e processamento necessários para gerar a imagem eram feitos pela CPU, ficando a placa de vídeo responsável apenas pelo envio dos dados ao monitor.

O termo GPU (*Graphics Processing Unit*) foi popularizado em 1999 pela fabricante Nvidia, por meio de sua placa GeForce 256. O grande avanço desse modelo foi a incorporação de um *chip* com capacidade de processamento de 10 milhões de polígonos por segundo (23). Com isso, o *hardware* gráfico passou de um simples dispositivo de memória para uma unidade configurável, altamente paralelizada, permitindo implementar um *pipeline* gráfico, capaz de realizar transformação, iluminação, rasterização, texturização e demais etapas no próprio processador da placa gráfica.

Houve uma significativa evolução com a introdução de GPUs com capacidade de processamento próprios. Cálculos pesados de processamento gráfico que antes eram feitos pela CPU passaram a ser realizados na própria GPU, aumentando consideravel-

mente a velocidade de aplicações gráficas em tempo real. Apesar do avanço, seu *hardware* era limitado no desenvolvimento de programas, pois apesar de ser configurável, o programador não tinha controle sobre o código executado na placa.

Esse cenário sofreu uma mudança expressiva em 2001, com o lançamento da GeForce série 3 pela Nvidia. O modelo foi um divisor de águas na tecnologia de GPUs (22). Pela primeira vez, o mercado lançava uma GPU com arquitetura totalmente programável por meio da implementação do padrão DirectX 8. Nessa arquitetura operações aritméticas intensas eram realizadas usando como parâmetro as coordenadas do ponto ou vértice, sua cor, textura e demais componentes. Uma consequência disso é que o processamento de cores poderia ser substituído por qualquer computação numérica.

Não demorou muito para pesquisadores perceberem que a GPU poderia ser utilizada em aplicações de propósito geral, além de exibir objetos graficamente. No entanto, isso era feito de forma rudimentar, utilizando um truque para iludir a GPU. O programador normalmente utilizava bibliotecas gráficas como DirectX ou OpenGL, fornecia os dados numéricos que deveriam ser processados, e a GPU realizava todos os cálculos necessários e retornava o resultado como se fosse uma renderização comum.

O uso de GPUs, apesar de seu poder de processamento, arquitetura altamente paralela e baixo custo ainda era limitado. Havia sérias restrições como a limitação de leitura e escrita de dados na memória da GPU, a ausência de operações lógicas (*OR, AND, XOR e NOT*), a falta de conformidade com o padrão IEEE em operações de ponto-flutuante e a inexistência de precisão dupla (24). E, além de conhecer a linguagem de programação, o desenvolvedor precisava ter conhecimentos em processamento gráfico, o que aumentava o tempo de aprendizado.

Esses desafios tornavam a programação com objetivos gerais em GPU bastante restrita, e em alguns casos, proibitiva. Com o objetivo de reduzir esses desafios, a Nvidia lança em 2006 a série GeForce 8, com suporte à plataforma de computação paralela CUDA (*Compute Unified Device Architecture*). Este projeto implementava os padrões IEEE em operações de ponto-flutuante para processamento de uso geral e possuía suporte total à leitura e escrita na memória da GPU. Além disso, permitia que programadores desenvolvessem seus códigos sem a necessidade de utilizar bibliotecas gráficas como OpenGL e DirectX. Em razão disso surgiram novas perspectivas neste modelo de computação paralela, que veio a se chamar GPU *Computing*.

2.1 GPU *Computing*

As CPUs foram concebidas para resolver problemas de forma intrinsecamente sequencial. Mas mesmo em processadores seriais havia a sensação de paralelismo, pois as instruções de processos concorrentes eram executadas em velocidades tão elevadas, que criavam a ilusão conhecida como pseudo-paralelismo. Ao longo de sua evolução foram adicionadas técnicas reais de paralelismo como *pipelines* de instruções, múltiplas *threads* e mais recentemente núcleos de processamento foram incorporados ao seu circuito integrado. No entanto, a prioridade no desenvolvimento de CPUs sempre foi aumentar a frequência do processador e sua paralelização tinha menor relevância.

Ao contrário das CPUs, o processamento gráfico surgiu para resolver problemas massivamente paralelos, pois polígonos, cores, texturas e renderizações de objetos distintos são independentes e, consequentemente, podem ser processados em paralelo. O alto desempenho de GPUs em operações aritméticas também é fundamental, à medida que processam bilhões de *pixels* por segundo e suportam execuções em tempo real. Além disso, nessas aplicações o *throughput* é mais importante do que a latência, pois o olho humano opera na escala de milissegundos, enquanto o processador executa em nanossegundos. Isso permite *pipelines* gráficos profundos, com latência de centenas de milhares de ciclos (25).

Os aspectos que conduziram o desenvolvimento da GPU também são importantes para resolução de problemas não gráficos, em especial problemas paralelizáveis que demandam alto desempenho. As melhorias agregadas às plataformas de desenvolvimento, com suporte à programação de propósito geral, permitiram que pesquisadores utilizassem esses dispositivos para solucionar diversos problemas dos campos científico e comercial, habilitando o que se conhece como GPU *Computing*.

Recentemente as aplicações utilizando GPU *Computing* têm tido muito destaque, devido aos resultados positivos alcançados. O desempenho superior dessas aplicações é obtido caso aproveitem toda sua capacidade e tendo em consideração as particularidades de sua arquitetura. Pode-se citar exemplos de sua aplicação em áreas como genética (26), meteorologia (27), criptografia (28), redes neurais (29), etc.

Apesar desses resultados, nem toda aplicação alcança desempenhos satisfatórios em GPU. Um dos problemas mais comuns é o *branch divergence*, que ocorre quando *threads* de um mesmo *warp* executam caminhos diferentes. Nesse caso, é realizada a serialização do trecho do código, reduzindo o desempenho do programa. Outro problema acontece em aplicações que fazem uso intenso de acesso à memória em posições não contínuas, resultando em múltiplas transações para realizar essas operações.

2.2 Plataformas de Desenvolvimento

Inicialmente, programadores utilizavam linguagens de montagem para executar suas aplicações em GPU. Isso aumentava significativamente os esforços para implementar seus projetos, além de complicar a depuração e manutenção do código. Para reduzir esse esforço, fabricantes e a comunidade *open source* desenvolveram diversas soluções, como o Microsoft HLSL, o Nvidia CG e o OpenGL *Shading Language*. Essas bibliotecas reduziram a complexidade de codificação, permitindo a escrita de códigos com sintaxe parecida com a linguagem C.

Essas bibliotecas facilitavam o desenvolvimento, mas exigiam que o programador conhecesse detalhes e limitações específicas do *hardware*. Além disso, elas exigiam o uso de expressões gráficas como linhas, vértices, texturas, etc. Claramente esses desafios não eram fáceis para um programador comum. Logo, algumas iniciativas acadêmicas propuseram soluções de mais alto nível como o Brook (30) e o Sh (31), que facilitavam o desenvolvimento em GPU. Essas iniciativas influenciaram tecnologias predominantes atualmente, como CUDA, OpenCL e DirectCompute.

A plataforma CUDA, desenvolvida pela Nvidia exclusivamente para suas GPUs, disponibiliza bibliotecas e ferramentas para diversas linguagens. Essa solução conquistou rapidamente uma ampla adoção devido a suas funcionalidades e pelo alto desempenho de seu *hardware* em relação a seus competidores. Apesar de orientada ao fabricante, boa parte de suas funções são intercambiáveis entre as diferentes linguagens disponíveis.

O OpenCL (*Open Computing Language*) é um padrão aberto mantido pelo mesmo consórcio responsável pelo OpenGL. Essa biblioteca tem suporte à computação paralela em plataformas heterogêneas, isto é, não é restrito a nenhum fabricante e, diferentemente de CUDA, não é limitado a apenas arquiteturas de GPUs. Assim, também é possível utilizá-lo em CPUs, DSPs (*Digital Signal Processors*), FPGAs (*Field-Programmable Gate Arrays*), etc.

DirectCompute, desenvolvido pela Microsoft, é um componente DirectX responsável por GPU *computing*. Esse padrão também oferece suporte a múltiplos fabricantes, mas está restrito ao ambiente Windows de desenvolvimento. Esse fator limita sua utilização no meio científico, tendo em vista que grande parte das pesquisas são conduzidas em servidores Unix-*like*. As GPUs da Nvidia têm alcançado muito destaque em trabalhos acadêmicos em computação de alto desempenho, com a plataforma CUDA sendo a escolha natural para esses equipamentos, devido ao suporte do fabricante. Esses motivos influenciaram a escolha por GPUs da Nvidia e o uso da plataforma CUDA neste trabalho.

2.3 Comparação entre GPU e CPU

Uma métrica bastante utilizada para medir o desempenho de processadores é conhecida como FLOPS (*FLoating Point Operations Per Second*), isto é, o número de operações de ponto-flutuante realizadas por segundo. Em termos de FLOPS teóricos, isto é, considerando apenas as especificações técnicas do *hardware*, o desempenho de GPUs supera qualquer CPU disponível no mercado (1). A diferença de desempenho entre esses dispositivos pode ser explicada pelo contraste entre suas microarquiteturas. Enquanto a CPU dedica a maior parte de sua infraestrutura para armazenamento em *cache* e controle de fluxo, a GPU utiliza mais transistores para processamento de dados, por meio de unidades de lógica e aritmética, conforme ilustrado na Figura 2.1:

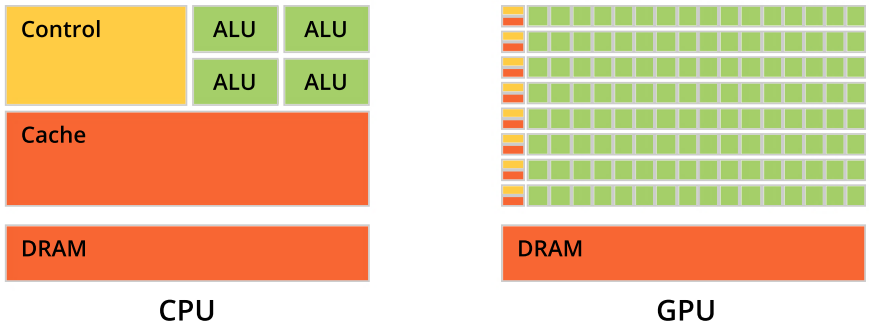


Figura 2.1: Arquitetura da CPU e GPU (1)

A CPU alcança seu desempenho máximo em séries de tarefas ordenadas, com significativa quantidade de acessos à leitura e escrita de dados armazenados em *cache*. Isso é resultado do seu padrão de desenvolvimento, que sempre priorizou o aumento da frequência de relógio. Porém, essa abordagem esbarrou no crescente consumo de energia, chegando ao limite quando a densidade de potência se aproximou a do núcleo de reatores nucleares (32). Impossibilitado de resfriar as altas temperaturas, a solução encontrada para melhorar seu desempenho foi adicionar núcleos de processamento.

De maneira oposta, as GPUs sempre priorizaram o paralelismo, fazendo uso de muitos núcleos menores e suporte a milhares de *threads*. A título de referência, os processadores Intel Core i7 possuem no máximo 4 núcleos de processamento e executam até 8 *threads*. Enquanto isso, GPUs com a arquitetura Maxwell da Nvidia suportam até 3072 Cuda *cores* (33), cada um com capacidade para realizar operações aritméticas com inteiros ou pontos-flutuantes, além da capacidade de hospedar 49152 *threads*.

2.4 Arquitetura da GPU

Atualmente existem três grandes fabricantes de GPUs: Intel, AMD e Nvidia. A Intel lidera o mercado com seu *hardware* integrado e de baixo desempenho. Já no segmento de alta capacidade destacam-se AMD e Nvidia. A comunidade científica tem dado bastante atenção para as GPUs da Nvidia, principalmente sob a plataforma CUDA, que é adotada neste projeto. Os detalhes e terminologias das GPUs variam de acordo com o fabricante, portanto, neste trabalho esses termos referem-se aos equipamentos da Nvidia.

A organização típica de uma GPU da Nvidia compreende matrizes de tamanho variável conhecidas como *Streaming Multiprocessors* (SMs). Cada SM possui núcleos de processamento com *pipelines* completos de operações de lógica e aritmética (*ALU*) e de ponto flutuante (*FPU*) chamados CUDA *cores*.

Quando a GPU recebe diversas *threads*, é feito o particionamento em grupos de 32 *threads*, chamados pela Nvidia como *warps* – esse valor é o mesmo entre todas GPUs desse fabricante. Na sequência o escalonador global distribui blocos de *threads* entre os SMs disponíveis. A partir de então, no nível do SM, o escalonador *warp scheduler* distribui os *warps* entre as unidades de processamento. As *threads* que compõem um *warp* iniciam simultaneamente a partir do mesmo endereço de programa, mas armazenam seu contador de instruções e mantém o estado de seus registradores separados, de modo a permitir sua execução paralela.

Um *warp* executa uma instrução comum por vez, portanto, a eficiência máxima é obtida quando todas as 32 *threads* concordam com seu caminho de execução. Se houver divergência, o *warp* executa serialmente cada trajeto e quando todos terminarem sua execução, as *threads* convergem para o mesmo caminho de execução. O sincronismo entre *threads* ocorre apenas dentro de um mesmo *warp*, pois diferentes *warps* executam independentemente de existir um caminho comum ou não (1).

Durante a execução, o *warp scheduler* pode realizar a troca de contexto entre *warps*. Esse processo tem um custo computacional muito baixo, pois o contexto de execução (PC - *Program Counter*, registradores, etc) de cada *warp* permanece armazenado durante todo o seu ciclo de vida.

A arquitetura presente nas GPUs estabelece uma hierarquia de acesso a diversos níveis de memória. Cada nível possui tamanhos e velocidades distintas, permitindo um melhor desempenho de acordo com sua categoria. Maiores detalhes serão explorados ao longo deste capítulo.

2.4.1 Organização das *Threads*

As *threads* da GPU são organizadas hierarquicamente em uma estrutura que agrupa *threads* em blocos com até três dimensões, conforme mostrado na Figura 2.2. A escolha do tamanho do bloco deve ser feita com muita atenção, pois esse fator tem impacto direto sobre seu desempenho. Assim, sua dimensão deve ser ajustada de forma a maximizar a ocupação da GPU. Também é importante destacar que todas as *threads* de um bloco executam e compartilham os recursos de um mesmo SM, criando um limite máximo de *threads* por bloco. Atualmente, este valor é de 1024 *threads*.

A comunicação entre *threads* de um mesmo bloco é feita a partir de funções de sincronismo, principalmente por meio de barreiras, que impedem o prosseguimento até que todas as *threads* do bloco estejam no mesmo ponto. Outra maneira bastante comum de cooperação é por meio da memória compartilhada de *threads* do mesmo bloco. Já a comunicação entre blocos distintos deve ser evitada. É possível trocar informações entre blocos usando a memória global, no entanto, isso pode causar efeitos colaterais como *deadlocks*.

As execuções dos blocos são independentes entre si, isto é, podem executar em qualquer ordem, concorrente ou serial. Isso proporciona maior escalabilidade do código, pois permite que o escalonador distribua os blocos de acordo com o número de SMs disponíveis na arquitetura.

De forma análoga, os blocos são agrupados em um *grid* e executam na mesma GPU. Além disso, todos os blocos do *grid* devem ter tamanhos semelhantes. Utilizando essa hierarquia podemos descobrir o identificador da *thread*, por meio de variáveis embutidas que representam sua posição no bloco. Em blocos de uma dimensão, seu identificador é a própria posição do vetor, já em blocos de duas ou três dimensões, são utilizadas suas coordenadas x, y e z, dependendo da dimensão utilizada, para construir um identificador único. Esse identificador permite, por exemplo, reconhecer as *threads* e controlar o que será processado durante sua execução.

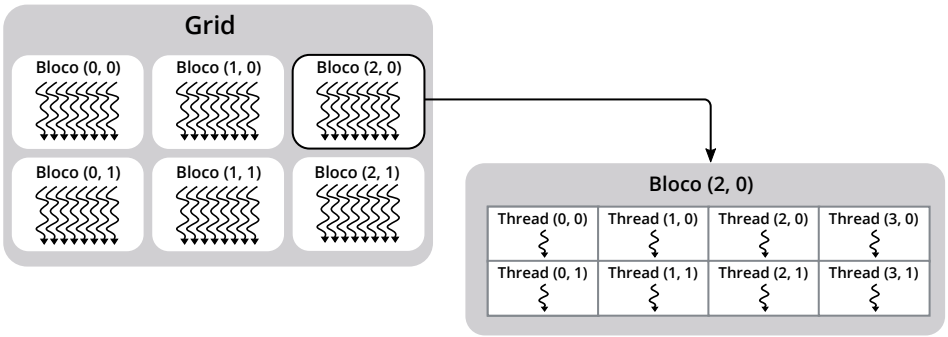


Figura 2.2: Organização das *threads* na GPU (adaptado de (1))

2.4.2 Hierarquias de Memórias

Assim como para a CPU, a organização do sistema de memória na GPU também é influenciada por fatores como velocidade, tamanho e custo. Similarmente, quanto mais próximo às unidades de processamento, maior é a velocidade, mas também maiores são os custos associados. Há portanto, uma necessidade de encontrar um ponto de equilíbrio entre desempenho e custo.

Numa GPU cada nível é acessível em função da disposição hierárquica de *threads* e a localização da informação. Desta maneira, *threads* individuais têm acesso a registradores e à memória local. Blocos de *threads* podem utilizar a memória compartilhada e quando o escopo são todos os blocos é possível acessar a memória global. Os detalhes da arquitetura e sua organização variam de acordo com construção do *hardware*, mas de modo geral se pode utilizar a Figura 2.3 como referência em alto nível do acesso aos diferentes níveis de memória.

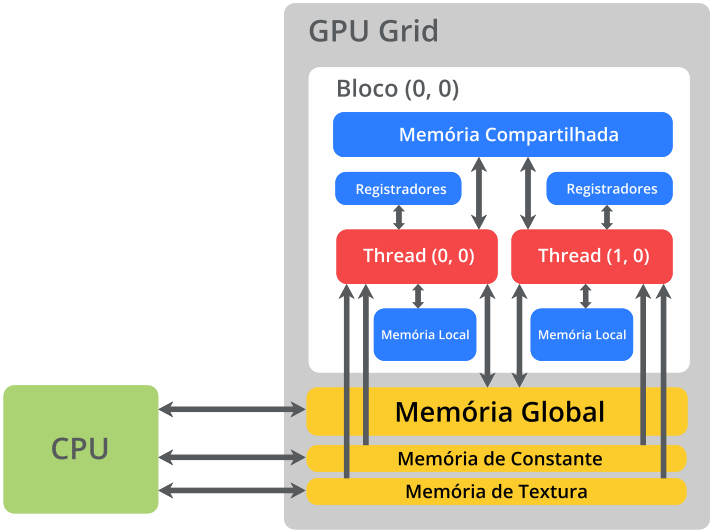


Figura 2.3: Organização da memória na GPU (adaptado de (1))

Registradores

Os registradores estão disponíveis em cada SM e fornecem o acesso mais rápido encontrado na hierarquia de memórias da GPU. Teoricamente o acesso pode ser feito em apenas um ciclo de relógio. Cada *thread* pode alocar um conjunto de registradores exclusivo para armazenar suas variáveis internas e atualmente esse limite é de 255 registradores por *thread*. Normalmente, o compilador tenta armazenar declarações de variáveis escalares em registradores. Se não for possível ocorre um transbordamento para a memória local, o que aumenta a latência de acesso. Vale destacar que à medida

que aumenta o número de registradores alocados por *thread*, diminui o número de *threads* em execução por SM, caso não existam recursos suficientes para todas as *threads*. Portanto, o compilador tem que ponderar entre o paralelismo e os recursos disponíveis para otimizar sua execução.

Memória Compartilhada

Essa memória de leitura/escrita é particionada entre todos os blocos residentes no SM. Deste modo, *threads* do mesmo bloco têm acesso aos mesmos dados e podem estabelecer pontos de sincronismo. Além disso, sua localização *on-chip* em cada multiprocessador garante um alto desempenho, cuja latência é entre 20 a 30 vezes menor que a memória global. Sua desvantagem é o seu tamanho reduzido, atualmente em 96Kb, o que restringe sua utilização.

Diferentemente da alocação de registradores, a memória compartilhada é controlada explicitamente pelo programador e sua utilização é feita por meio de declarações de seu uso. Isso permite distribuir o armazenamento entre registradores e a memória compartilhada de forma a otimizar o programa como um todo.

Em algumas arquiteturas, como a Fermi e a Kepler, a memória compartilhada e a *cache* L1 ocupam a mesma unidade. A distribuição do espaço nesses ambientes é feita de acordo com a utilização da memória compartilhada, e o restante do espaço é reservado para *cache* L1. A partir da arquitetura Maxwell há unidades exclusivas para a memória compartilhada, e neste caso, a *cache* L1 ocupa a mesma placa também utilizada como memória de textura. Um resumo dessas variações é ilustrado na Tabela 2.1.

Arquitetura	**Cache L1**	**Memória Compartilhada**
Fermi	unid. compartilhada	unid. compartilhada
Kepler	unid. compartilhada	unid. compartilhada
Maxwell	unid. exclusiva	unid. exclusiva
Pascal	unid. exclusiva	unid. exclusiva

Tabela 2.1: Mudanças dos níveis de memória entre as arquiteturas

Memória Local

Essa memória é na prática uma abstração de acesso à memória global. Neste nível cada *thread* também tem acesso exclusivo à sua memória local. A localização física dessa memória é fora do *chip* do processador, portanto, sua latência e largura de banda estão sujeitos ao desempenho da memória global, normalmente entre 400 e 800 ciclos.

Esse nível de memória é utilizado para armazenar variáveis que ocupam muito espaço, ou quando não há registradores livres suficiente (*register spilling*).

Memória Global

Conforme pressuposto por seu nome, o escopo de acesso dessa memória é global, isto é, todas as *threads*, independentemente do bloco associado, têm acesso a essa unidade. Esse nível fornece a maior capacidade de armazenamento na hierarquia de memórias da GPU, em torno de 6Gb. Contudo, a alta capacidade é penalizada pela latência de acesso, o que pode prejudicar o desempenho do sistema.

O armazenamento na memória global também é explícito, usando funções que alocam e liberam seu espaço. Normalmente, este nível de memória é utilizado como uma etapa de transferência de grandes quantidades de dados entre a CPU e a GPU, embora seja necessário ressaltar que essas transferências devem ser feitas com bastante critério, pois têm impacto significativo no sistema.

Com o objetivo de aumentar a vazão da memória global, as arquiteturas mais recentes implementam em *hardware* o acesso à memória global via armazenamento *cache* L1/L2. Desta maneira, se os dados já estiverem em *cache*, seu acesso será muito mais rápido, caso contrário, é feita a busca na memória global e posteriormente armazenado em *cache*.

Memória de Textura

Nível de memória somente de leitura, inicialmente projetada para armazenar dados no contexto de aplicações gráficas, daí o nome memória de textura, evoluiu para também suportar aplicações de propósito geral. Assim como a memória de constante, este nível também está localizado fora da unidade de processamento, e igualmente possui acesso via *cache* com até 48Kb de armazenamento. Sua organização é otimizada para obter desempenho superior em algumas operações específicas como, por exemplo, a interpolação linear de texturas, mas outras aplicações que fazem uso do princípio da localidade também são beneficiadas.

Memória de Constante

Memória somente de leitura utilizada para armazenar constantes compartilhadas entre *threads*. Nas GPUs atuais, essa memória ocupa 64Kb fora do *chip* do processador, mas cada SM possui 10Kb de *cache* dedicados a essa unidade. Isso permite que

acessos a constantes tenham baixa latência e alto *throughput*, comparável à velocidade de acesso aos registradores.

2.5 Acesso eficiente à memória

Entre os diversos níveis de memória em GPU, a memória global está presente na maior parte das aplicações, pois é a porta de entrada entre o *host* e o *device*. Portanto, qualquer aplicação em GPU irá acessar essa memória pelo menos uma vez ao longo do seu ciclo de vida, quando transferir informações entre os dispositivos. Além disso, ela é muito utilizada pois oferece a maior capacidade de armazenamento entre todos os níveis de memória.

A vantagem de ter a disposição um grande espaço de armazenamento como a memória global vem acompanhada de um alto custo computacional. A partir do *device*, esse custo só não é maior que o acesso à memória do *host*. Portanto, além de planejar o uso dos diferentes níveis de memória, para reduzir esse gargalo, também é preciso conhecer como é feita a busca na memória, pois eventualmente ela será utilizada. Nesse sentido, uma técnica bastante utilizada é chamada agrupamento de memória (*memory coalescing*.)

Os detalhes dessa técnica variam entre gerações diferentes de GPU, mas de modo geral, podem ser explicados da seguinte forma. Um conjunto de *threads*, chamado *warp*, executa a mesma instrução. Quando o *hardware* identifica instruções de acesso à memória, ele verifica se os acessos são feitos a posições contíguas de memória. Caso isso ocorra, as instruções são agrupadas em apenas uma transação, reduzindo o número de acessos. Portanto, caso a *thread* i acesse a posição n, a *thread* $i + 1$ acesse a posição $n + 1$, e assim sucessivamente, o número de transações necessárias para acessar a memória será reduzido.

Essa técnica é ilustrada na Figura 2.4. Nela, 32 *threads* acessam uma região contínua de memória, com 32 palavras de quatro *bytes* cada. Essa estratégia é utilizada neste projeto sempre que possível, tanto na leitura dos objetos fornecidos quanto na escrita dos protótipos encontrados.

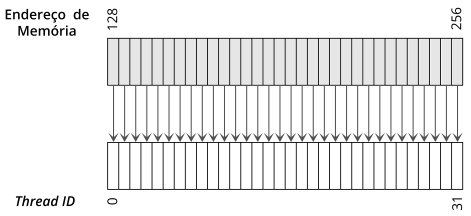


Figura 2.4: Busca eficiente de dados da memória

2.6 Ocupação

A otimização do código executado na GPU exige, além de algoritmos eficientes, um conhecimento básico quanto à sua arquitetura e sobre as restrições técnicas da geração do *hardware*. Assim, para alcançar um alto desempenho é necessário otimizar o uso dos recursos disponíveis e, ao mesmo tempo, manter a utilização do processador em níveis satisfatórios para obter todo o potencial de processamento.

Nesse contexto, uma métrica bastante utilizada é a ocupação, descrita como a razão entre o número de *warps* ativos por multiprocessador em função do número máximo possível. Por meio dessa medida, é preciso encontrar uma ocupação adequada para o programa. Isso não implica necessariamente em alcançar a máxima ocupação possível, pois a partir de certo ponto não existe ganho significativo em desempenho. Porém, se não houver ao menos uma ocupação suficiente para esconder a latência de acesso à memória, seu desempenho pode obter resultados abaixo do desejado. A ocupação da GPU é influenciada diretamente por três fatores:

- **Uso de Registradores:** a alocação de registradores é feita por blocos de *threads*, portanto, à medida que *threads* utilizam muitos registradores, menos blocos são possíveis por multiprocessador, reduzindo a ocupação geral na GPU. Atualmente, alguns equipamentos disponibilizam 65536 registradores de 32 *bits* e 64 *warps* por SM, isto é, 2048 *threads* ($64 \times 32 = 2048$). Assim, em uma configuração possível com 100% de ocupação, cada *thread* pode utilizar no máximo 32 registradores ($65536 \div 2048 = 32$).

- **Tamanho do Bloco:** a especificação mais recente permite 32 blocos e 64 *warps* residentes por SM. Então, uma configuração com blocos de 16 posições ($16 \times 32 = 512$ *threads*) resulta em apenas 16 *warps* ($512 \div 32 = 16$) de 64 possíveis, ou seja, atinge somente 25% de sua ocupação. Normalmente, o número de *threads* é um múltiplo de 32 para melhorar a divisão entre os *warps*, assim, valores como 128, 192, 256, 384 e 512 *threads* são frequentemente utilizados.

- **Uso de Memória Compartilhada:** a demanda por memória compartilhada também restringe a ocupação da GPU, pois existe um limite dessa memória por SM. Portanto, à medida que aumenta sua alocação, diminui o número de blocos possíveis por multiprocessador. As GPUs atuais têm capacidade para armazenar 96Kb em memória compartilhada por SM, e cada bloco deve ter no máximo 48Kb.

2.7 CUDA

A plataforma CUDA surgiu em um contexto desafiador na programação de propósito geral utilizando GPUs. Os objetivos dessa iniciativa eram uniformizar a programação usando esses equipamentos, atender aos padrões estabelecidos pelo IEEE e eliminar o uso de expressões gráficas no desenvolvimento do código, de modo que a programação nesse ambiente fosse simples o suficiente para estar ao alcance de qualquer programador comum.

Para utilizar essa plataforma o programador precisa instalar os *drivers* da placa e o *software* CUDA *toolkit*. O *toolkit* instala o ambiente de desenvolvimento CUDA, composto pelo compilador *nvcc*, bibliotecas de computação paralela e ferramentas de otimização e depuração do código. Há suporte a diversas linguagens de programação, porém a linguagem escolhida neste trabalho foi C++.

2.7.1 Estrutura de Programação

Aplicações escritas com a plataforma CUDA adotam um modelo de programação heterogêneo, em que códigos que executam em paralelo convivem com segmentos de computação sequencial em um mesmo arquivo binário, conforme indicado na Figura 2.5. No entanto, a execução das *threads* nessa plataforma acontece em um dispositivo externo ao hospedeiro, que realiza o processamento sequencial. Por isso, a GPU também é conhecida como *device*, enquanto a CPU é chamada de *host*.

Devido à separação entre esses ambientes, o *device* possui armazenamento e endereçamento de memória separados do *host*. Isso tem impacto direto no acesso aos diversos níveis de armazenamento em GPU, por meio de chamadas explícitas a funções disponíveis na biblioteca CUDA. Portanto, o programador é responsável pela alocação, transferência e liberação de memória entre os dois ambientes.

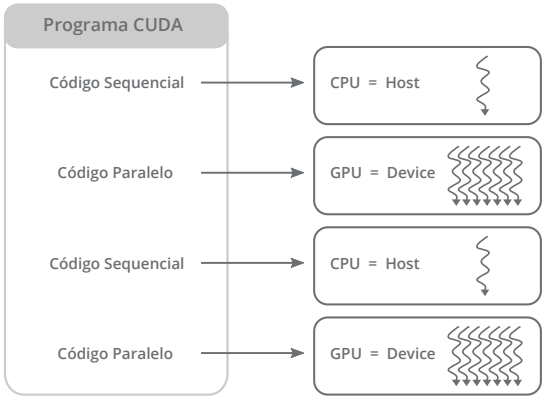


Figura 2.5: Estrutura de um programa CUDA (adpatado de (1))

2.7.2 *Kernel*

Um componente chave em CUDA é conhecido como *kernel*. Ele representa uma função chamada a partir do *host* que executa no *device* e retorna *void*. Logo, é a porta de entrada para a execução no *device*. A partir do *kernel* só é possível chamar funções que executam explicitamente no *device*. Do lado do *host*, é permitido executar funções que executam exclusivamente no *host* ou o *kernel*. Além disso, todo *kernel* utiliza uma sintaxe especial, que identifica o número de blocos e de *threads* por bloco que devem executar na GPU.

A sintaxe utilizada pelo *kernel*, mostrada no Exemplo 2.1, utiliza o qualificador `__global__` para informar ao compilador que o código deve ser gerado para o *device*. A sintaxe fornece flexibilidade na escolha da organização das *threads*, assim o programador tem autonomia sobre o código que irá executar no *device*, bem como a quantidade de *threads* que irá processar seu algoritmo.

```
__global__ void myKernel() {
  // implementação da função
}

int main() {
   // kernel com 8 blocos e 256 threads por bloco
   myKernel<<< 8, 256 >>>();
   ...
   // declaração em 2 dimensões
   dim3 threadsPerBlock(32,32);
   // kernel com 4 blocos e 32x32 threads por bloco
   myKernel<<< 4, threadsPerBlock >>>();
   ...
}
```

Exemplo 2.1: Sintaxe de um *kernel*

Os demais qualificadores de funções disponíveis em CUDA são mostrados na Tabela 2.2. O qualificador padrão é usado para chamar funções a partir do *host* e também executam nesse ambiente. Já funções que usam o qualificador `__device__` devem ser invocadas e executam na GPU.

Qualificador	**Invocação**	**Execução**	**Tipo de Retorno**
__global__	*host* / *device*	*device*	void
__device__	*device*	*device*	sem restrições
__host__	*host*	*host*	sem restrições

Tabela 2.2: Qualificadores de funções em CUDA

2.7.3 Memória

A programação usando CUDA expõe ao programador os diversos níveis de memória disponíveis na GPU. Cada nível tem como particularidade sua latência e a sua capacidade de armazenamento, portanto, fica sob responsabilidade do desenvolvedor fazer as escolhas que melhor atendam às suas necessidades. A maior parte desses níveis devem ser declarados explicitamente, conforme o Exemplo 2.2.

```
__constant__ int fib[8] = {1, 1, 2, 3, 5, 8, 13, 21}; // constante
__device__ int A[50000]; // memória global

__global__ void myKernel() {
  int i = 0; // registrador
  double B[1000]; // memória local
  __shared__ int C[100]; // memória compartilhada
}
```

Exemplo 2.2: Sintaxe dos principais níveis de memória

De modo geral, os programas em CUDA seguem uma estrutura muito parecida, definida pelo Exemplo 2.3. Primeiramente, é feita a alocação dinâmica do espaço na memória global, usando a função `cudaMalloc`. Em seguida, os dados de entrada são transferidos do *host* para o *device*, por meio da função `cudaMemcpy`. Na sequência, o *kernel* é invocado, passando como parâmetro um ponteiro para a região de memória alocada no *device*. Quando o *kernel* termina sua execução os dados devem ser transferidos no sentido inverso, isto é, do *device* para o *host*. Por fim, é feita a liberação do espaço alocado, por meio da função `cudaFree`.

```
__global__ void myKernel(int *dev_a) { ... }

int main() {
int *dev_a, a[N];
   ...
   // aloca vetor com nBytes no device
   cudaMalloc((void **) &dev_a, nBytes);
   // copia nBytes do host para device (dev_a = a)
   cudaMemcpy(dev_a, a, nBytes, cudaMemcpyHostToDevice);

   // invoca kernel e passa alocação da mem. global como parâmetro
   myKernel<<<blocks, threads_per_block >>>(dev_a);

   // copia resultado do device para host (a = dev_a)
   cudaMemcpy(a, dev_a, nBytes, cudaMemcpyDeviceToHost);
   //libera espaço reservado na memória global
   cudaFree(dev_a);
}
```

Exemplo 2.3: Alocação e transferência de dados

2.7.4 Variáveis Internas

Todo *kernel* tem acesso às variáveis internas `blockIdx` e `threadIdx`. Inicializadas pelo CUDA *runtime*, essas variáveis possuem coordenadas x, y e z que representam, respectivamente, a localização do bloco no *grid* e a posição da *thread* no bloco. Além disso, há também variáveis internas que armazenam a dimensão do grid e do bloco – `gridDim` e `blockDim`. Essas variáveis são frequentemente utilizadas, por exemplo, na divisão do processamento entre as *threads*.

2.7.5 Sincronização

Sincronização é um dos fatores mais importantes em programação paralela. Esse é o momento em que as *threads* podem trocar informações e acessar dados consistentes, processados pelas demais *threads*. Em CUDA, *threads* de um mesmo bloco podem ser sincronizadas. Para garantir que elas estejam em um mesmo ponto é utilizado um mecanismo conhecido como barreira, usando a função `__syncthreads()`. Esse método faz com que cada *thread* espere até que todas alcancem este mesmo local, para então permitir seu prosseguimento.

Uma demonstração do uso de barreiras pode ser visto no Exemplo 2.4. Nela é feito o cálculo do produto interno de dois vetores A e B. Cada *thread* calcula o produto da *i-ésima* posição e, em seguida, a função `__syncthreads()` garante que todas *threads* tenham terminado até o ponto de sua declaração. A certeza que todas *threads* alcançaram o ponto, permite que uma das *threads*, neste caso a *thread* com índice 0, realize a soma dos produtos e armazene a resposta.

```
__global__ void dotProduct(int *A, int *B, int *output) {
    __shared__ int tmp[N];
    int index = threadIdx.x;

    tmp[index] = a[index]*b[index];

    // threads esperam até que todas estejam neste ponto
    __syncthreads();

    if (index == 0) {
        int sum = 0;
        for (int i=0; i < N; i++)
             sum += tmp[i];
        // resposta
        *output = sum;
    }
}
```

Exemplo 2.4: Sincronização de *threads*

Instruções atômicas

Um problema comum em ambientes paralelos é conhecido como condição de corrida. Isso ocorre quando múltiplas *threads* acessam uma região de memória compartilhada de maneira não ordenada. Por exemplo, considere o Exemplo 2.5, suponha que diversas *threads* acessam essa função em paralelo e incrementam o valor da variável compartilhada. Neste caso, não é possível determinar se a leitura de uma *thread* foi feita antes ou depois da escrita de outra *thread*, tornando o resultado imprevisível.

```
__global__ void increment(int *addr){
  int tmp = *addr;
  tmp = tmp + 1;
  *addr = tmp;
}
```

Exemplo 2.5: Condição de corrida

Para solucionar esse problema é necessário assegurar que operações em regiões críticas sejam feitas de forma atômica, ou seja, independentemente do número de *threads* em execução, quando é feita a escrita do valor, o ambiente deve garantir que nenhuma outra *thread* esteja acessando essa região compartilhada. Assim, não é permitido escritas concorrentes, e a leitura deve ser ocorrer somente após a escrita.

A plataforma CUDA disponibiliza funções atômicas para operações lógicas, aritméticas e de troca de valores. Todas essas funções atuam sobre regiões de memória visíveis entre as *threads*, normalmente variáveis armazenadas em memória global ou compartilhada, e executam as operações sem interferência das demais *threads*. Algumas funções atômicas são ilustradas na Tabela 2.3:

Função	**Descrição**
`atomicAdd(int *ptr, int val)`	adiciona valor ao conteúdo de *ptr*
`atomicSub(int *ptr, int val)`	subtrai valor ao conteúdo de *ptr*
`atomicExch(int *ptr, int val)`	troca conteúdo de *ptr* por *val*
`atomicOr(int *ptr, int val)`	operação lógica *OR* entre conteúdo de *ptr* e *val*
`atomicAnd(int *ptr, int val)`	operação lógica *AND* entre conteúdo de *ptr* e *val*
`atomicXor(int *ptr, int val)`	operação lógica *XOR* entre conteúdo de *ptr* e *val*

Tabela 2.3: Instruções atômicas em CUDA

2.7.6 Compilação

A compilação neste ambiente pode ser vista como uma extensão à linguagem alvo escolhida, por meio de funções especiais que operam diretamente na GPU. O compilador *nvcc* separa os trechos que devem ser computados pela GPU dos demais segmentos que serão executados pela CPU, de modo que as partes que devem ser processadas pela GPU são compiladas pelo *nvcc* e os demais segmentos são gerados pelo compilador da linguagem alvo utilizada.

Pela evolução constante desses equipamentos não seria possível conciliar a compatibilidade do código binário entre gerações distintas sem prejudicar a performance do programa. Em razão disso, o *nvcc* realiza a compilação em duas etapas, garantindo compatibilidade e otimização independentemente da arquitetura. A primeira etapa da compilação gera uma representação virtual da arquitetura, com os requisitos mínimos necessários para sua execução. Já a segunda etapa, gera código binário para a arquitetura real de execução.

Diante da dificuldade em conhecer a *priori* o ambiente de execução, principalmente em plataformas heterogêneas, o compilador posterga ao máximo a geração do código binário. Assim, exceto quando as arquiteturas virtual e real são solicitadas explicitamente, o compilador utiliza a técnica conhecida como *Just-in-Time Compilation*, que gera o código binário somente no momento de sua execução.

A vantagem dessa técnica é a otimização do código gerado para a arquitetura alvo. A desvantagem, porém, é o tempo extra gasto para carregar a aplicação. Contudo, isso pode ser reduzido através do armazenamento em *cache* do código binário gerado, evitando a compilação entre execuções subsequentes.

Os comandos do Exemplo 2.6 realizam a compilação do arquivo *hello.cu* e produzem a saída *hello*. O primeiro comando não especifica as arquiteturas virtual e real, portanto, um valor mínimo é assumido pelo compilador. Já o segundo comando é equivalente ao primeiro, pois declara a arquitetura virtual mínima. O terceiro comando define ambas arquiteturas. Por fim, o último comando, especifica uma arquitetura virtual e múltiplas arquiteturas reais.

```
nvcc hello.cu -o hello
nvcc hello.cu --gpu-architecture=compute_20 -o hello
nvcc hello.cu --gpu-architecture=compute_30 --gpu-code=sm_30 -o hello
nvcc hello.cu --gpu-architecture=compute_20 --gpu-code=sm_20,sm_30,sm_50 -o hello
```

Exemplo 2.6: Compilação de programa CUDA usando *nvcc*

2.8 Considerações Finais

Neste capítulo realizou-se uma fundamentação teórica sobre GPUs, para melhor compreender o ambiente de desenvolvimento do projeto. Do ponto de vista de *hardware*, foram apresentados detalhes dessa arquitetura, como unidades de processamento, níveis de memória, principais especificações, etc. Por outro lado, do ponto de vista lógico, foram discutidos conceitos de estrutura de programação nesse ambiente, como organização de *threads*, blocos, desenvolvimento por meio do *kernel* e fatores que influenciam a ocupação da GPU.

Capítulo 3

Agrupamento de Dados

Agrupamento de dados, também conhecido pelos termos *data clustering* ou *cluster analysis*, é a tarefa de agrupar objetos em grupos ou *clusters*, utilizando critérios de semelhança entre eles. Intuitivamente, objetos de um mesmo grupo têm maior similaridade entre si do que com objetos de outros grupos. Nesse sentido, o conhecimento do problema é fundamental na organização e classificação dos dados. Uma aplicação prática é a classificação dos seres vivos, divididos em reino, filo, classe, ordem, família, gênero e espécie. Essa taxonomia permite separar os organismos com características semelhantes, para melhor compreender suas particularidades e o seu papel na biodiversidade do planeta.

Um dos métodos adotados em agrupamentos é o de aprendizado não-supervisionado. Nele, os dados fornecidos não são previamente rotulados, isto é, os grupos são formados sem nenhuma informação a *priori* sobre qualquer padrão. Por outro lado, há também metodologias de classificação que utilizam aprendizado supervisionado, ou seja, existe um conhecimento prévio sobre o conjunto de treinamento. Por fim, o método semi-supervisionado é um modelo intermediário entre os dois paradigmas anteriores, ou seja, apenas uma parte do conjunto de dados é rotulada. Entre esses tipos de classificação, o aprendizado não-supervisionado, utilizado por algoritmos de agrupamento, apresenta os maiores desafios na identificação dos dados (34).

Embora o agrupamento de dados seja aplicado principalmente na área de reconhecimento de padrões, tanto para processamento de imagens como para análise de dados, essa técnica também é vastamente empregada nos mais variados campos, como economia (7), física (6), química (8), psicologia (35), arqueologia(36) e muitos outros. Isso justifica o grande interesse por pesquisas na área.

3.1 Conceitos Iniciais

Para compreender os procedimentos envolvidos em agrupamento de dados é preciso antes apresentar seus conceitos básicos.

3.1.1 Objeto ou Padrão

Um item do conjunto de dados é definido normalmente como objeto ou padrão. Computacionalmente, um item do conjunto de dados pode ser armazenado em um vetor com n posições, $x = (x_1, ..., x_n)$, em que cada valor x_i corresponde a um atributo do padrão. Um exemplo prático são as informações nutricionais de um alimento, mostradas na Figura 3.1. O objeto é o alimento em questão e os atributos são seus nutrientes, representados pelos valores de carboidratos, proteínas, sódio e cálcio.

X =

Carboidratos	Proteínas	Sódio	Cálcio
9.3g	6.2g	143mg	240mg

Figura 3.1: Representação de um objeto

3.1.2 Matriz de Dados

O agrupamento de objetos com múltiplos atributos utiliza uma matriz para armazenar os dados. Deste modo, uma matriz X de ordem $n \times m$ armazena n objetos com m atributos cada. A Matriz 3.1 armazena três objetos, um por linha, cada um com quatro atributos, em suas colunas.

$$X = \begin{bmatrix} 9.3 & 6.2 & 143 & 240 \\ 52 & 11 & 467 & 178 \\ 23 & 1.1 & 1 & 5 \end{bmatrix} \tag{3.1}$$

3.1.3 Similaridade

Um aspecto muito importante em algoritmos de agrupamento é a relação de similaridade entre os dados. Similaridade é um conceito subjetivo aos olhos do observador, cujos significado e interpretação requerem conhecimento sobre o problema (34). Sob a perspectiva visual e de acordo com a complexidade do problema, um observador comum pode agrupar objetos, sem muitas dificuldades, em até três dimensões. A partir daí essa tarefa começa a ser desafiadora para o olhar humano. Logo, é preciso definir métricas que permitam estabelecer a relação de agrupamento dos objetos. Entre essas regras podemos destacar:

- **Distância Euclideana:** é a métrica mais comum, dada pela menor distância entre dois pontos p e q no espaço $\mathbb{R}^n$. Matematicamente, a distância entre os pontos pode ser calculada pela Equação 3.2. Alternativamente, com o objetivo de reduzir o processamento da raiz quadrada, a norma Euclideana também é frequentemente utilizada.

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}. \tag{3.2}$$

- **Distância Mahalanobis:** que é calculada a partir do ponto x em relação ao ponto médio μ e a matriz covariância inversa S^{-1}, conforme a Equação 3.3. Na prática, essa métrica é bastante utilizada pois oferece um modelo estatístico, a partir do cálculo da matriz covariância, que permite determinar a similaridade entre as amostras.

$$D_m(x) = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}. \tag{3.3}$$

- **Distância *Manhattan* ou *city block*:** é a soma absoluta entre dois pontos usando as coordenadas cartesianas, sendo equivalente a calcular a distância entre dois pontos considerando os blocos de uma cidade, conforme a Equação 3.4:

$$d(x,y) = \sum_{i}^{n} |x_i - y_i|. \tag{3.4}$$

3.1.4 Lógica *Fuzzy*

A lógica clássica, fundamental no surgimento e desenvolvimento da computação, admite apenas dois valores, verdadeiro ou falso. Nela, não há espaço para incertezas, isto é, cada sentença tem apenas um valor absoluto. Assim, não é possível concluir que uma proposição seja parcialmente verdadeira, pois não existe uma terceira alternativa. Na prática, contudo, nem sempre é possível concluir que uma afirmação seja completamente verdadeira ou falsa. Em meteorologia, por exemplo, apesar da evolução das medições, não é possível afirmar com 100% de exatidão, todos os dias, se irá chover ou não.

Para resolver esses problemas de incerteza foi preciso construir uma nova teoria, conhecida como lógica *fuzzy* ou nebulosa. Esse termo surgiu em 1965, a partir dos estudos sobre os conjuntos *fuzzy* realizados por Lotfi Zadeh (37). Nessa teoria, além de definir operações em conjuntos *fuzzy*, foi introduzido o conceito *grau de pertinência*, normalmente no intervalo $[0, 1]$, que indica o quanto um determinado elemento pertence a um conjunto. Dessa forma, é possível realizar uma transição suave entre conjuntos distintos. Essa transição pode representar, por exemplo, as inúmeras tonalidades encontradas entre olhos claros e escuros.

Diversas aplicações práticas fazem uso da lógica *fuzzy* no dia a dia. Os controladores de aparelhos de ar condicionado, por exemplo, utilizam transições de temperatura, como visto na Figura 3.2(b), para determinar o quão quente está o ambiente, e assim, decidir se é preciso ligar a ventilação, a refrigeração ou ambos. Por outro lado, se fosse utilizada a lógica *booleana*, ilustrada na Figura 3.2(a), a transição entre os estados seria abrupta, ou seja, no período de transição, não seria possível tomar as decisões mais eficientes para atingir a temperatura esperada.

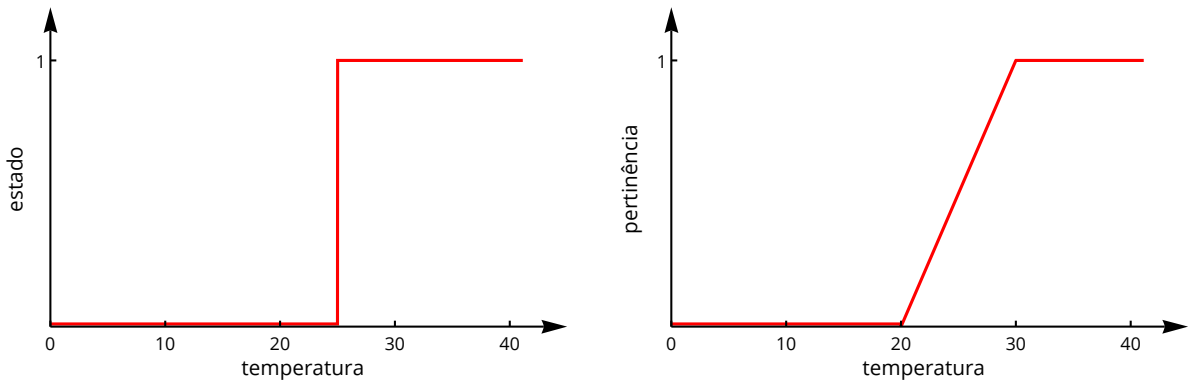


Figura 3.2: Comparação entre lógica clássica e *fuzzy*

3.2 Técnicas de Agrupamento

As diversas técnicas de agrupamento de dados podem ser divididas sob vários aspectos. Uma taxonomia bastante adotada foi proposta por Jain *et al.* em (2). De modo geral, segundo essa organização, ilustrada na Figura 3.3, os algoritmos de agrupamento podem ser basicamente divididos em duas categorias: hierárquicos e particionais. No modelo particional, o algoritmo divide os dados em subconjuntos independentes e não-sobrepostos. Por outro lado, o modelo hierárquico cria agrupamentos aninhados organizados como uma árvore, assim, cada nó é um agrupamento com sub-agrupamentos.

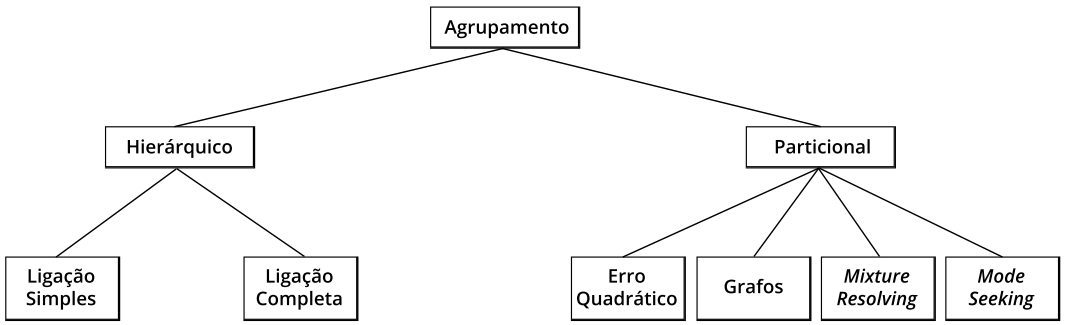


Figura 3.3: Taxonomia dos algoritmos de agrupamento (2)

Antes de uma discussão sobre as técnicas de agrupamento é preciso destacar os aspectos que podem afetar todas as metodologias. Independentemente da técnica adotada, as principais abordagens são:

- **Aglomerativa *vs.* Divisiva:** a abordagem aglomerativa começa com cada objeto em um agrupamento distinto e realiza a união dos grupos até satisfazer um critério de parada. Por outro lado, a estratégia divisiva começa com todos padrões em apenas um agrupamento e realiza a sua divisão até encontrar a solução desejada.
- ***Hard vs. Fuzzy*:** um agrupamento do tipo *hard* atribui um objeto a somente um agrupamento. Segundo o modelo *fuzzy*, existem graus de pertinência de cada objeto em relação aos grupos, ou seja, um padrão pode pertencer a múltiplos conjuntos. Um agrupamento do tipo *fuzzy* pode ser transformado em *hard* ao atribuir cada objeto ao grupo com maior grau de pertinência.
- **Determinística *vs.* Estocástica:** os agrupamentos produzidos em execuções sucessivas do mesmo algoritmo podem ser iguais, segundo a técnica determínistica, ou podem sofrer uma variação, se utilizar a abordagem estocástica.

3.2.1 Agrupamento Hierárquico

Os algoritmos que utilizam agrupamento hierárquico têm como resultado um diagrama chamado dendograma. Nessa estrutura cada nível representa a similaridade entre os grupos, portanto, quanto maior a distância entre os agrupamentos, menor sua similaridade. Uma característica importante dessa estrutura é a flexibilidade na escolha do número de agrupamentos. A escolha do nível do dendograma determina o número de conjuntos obtidos. Em razão disso, à medida que aumenta o seu nível, menor é a quantidade de grupos resultantes. O dendograma da Figura 3.4(b), obtido a partir dos pontos da Figura 3.4(a), foi cortado nos pontos tracejados. Nesse nível, cinco agrupamentos distintos são encontrados.

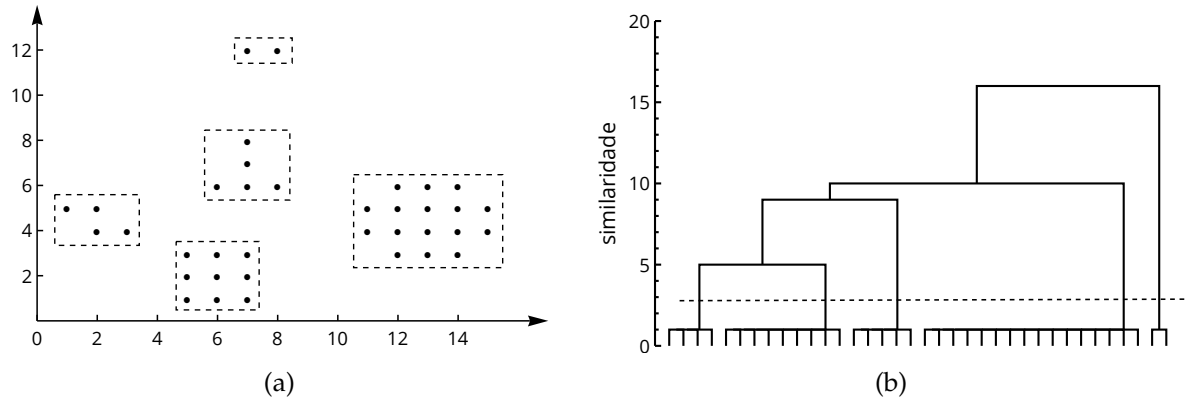


Figura 3.4: Agrupamento hierárquico usando ligação simples a partir do conjunto de pontos

Em linhas gerais, a diferença fundamental entre os algoritmos de agrupamento hierárquico é o modo como é feita a ligação entre os diferentes grupos. Existem diversas variações, porém os métodos mais utilizados são os de ligação simples e ligação completa (2):

- **Ligação Simples:** a distância entre dois grupos é a menor distância entre todos os pares de objetos de ambos agrupamentos. Essa abordagem produz longos agrupamentos, pois o critério de união escolhe os menores padrões, portanto, há uma avaliação local dos grupos.
- **Ligação Completa:** é o oposto da ligação simples, a distância entre os grupos é a maior entre todos os pares de ambos agrupamentos. Como resultado, essa técnica produz agrupamentos compactos com diâmetros menores, pois é feita uma análise global sobre os padrões.

Algoritmo de Johnson

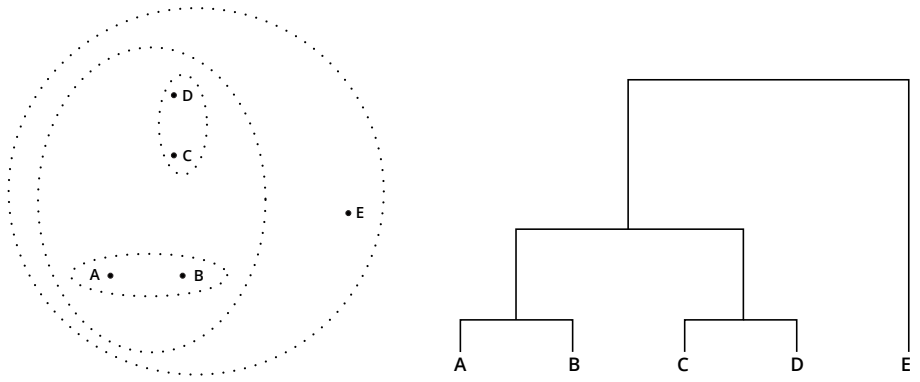
Uma implementação de agrupamento hierárquico foi proposta por Stephen Johnson (38). Sua desvantagem, como de qualquer algoritmo de agrupamento hierárquico, é seu custo computacional. Nesta implementação, ilustrada no Algoritmo 1, é utilizada a ligação simples e tem complexidade $O(n^2)$. O algoritmo utiliza uma matriz de distâncias D para criar agrupamentos entre os padrões mais próximos. A dimensão dessa matriz é reduzida em uma linha e uma coluna a cada iteração, pois os objetos mais próximos são intercalados. Esse processo ocorre até que um único agrupamento contenha todos os objetos.

Algoritmo 1 Etapas do algoritmo de Johnson

Entrada: conjunto de dados X
Saída: dendograma dos dados

procedimento JOHNSONSINGLELINKAGE(X)
    Inicialize a matriz D com as distâncias entre cada par de objetos (X_i, X_j).
    faça
        Encontre os objetos X_r e X_s mais próximos e forme um novo grupo
        Atualize a matriz de distâncias D, para todo objeto k
            $D[k,(r,s)] = min(D[k,r], D[k,s])$
        Intercale linhas e colunas r e s da matriz D com seus valores mínimos
    enquanto todos os objetos não estiverem em apenas um agrupamento
    retorne dendograma dos dados
fim procedimento

Uma abstração desse algoritmo pode ser vista na Figura 3.5(a). Os objetos mais próximos são colocados no mesmo agrupamento, representado por uma elipse. A cada iteração, um novo agrupamento incorpora objetos próximos ou grupos criados anteriormente, dependendo do que estiver mais próximo. Deste modo, os níveis hierárquicos são construídos e formam o dendograma mostrado na Figura 3.5(b).



(a) (b)

Figura 3.5: Hierarquização de acordo com o algoritmo de Johnson

3.2.2 Agrupamento Particional

Ao contrário de algoritmos hierárquicos, a metodologia particional têm como resultado apenas uma partição dos dados. Portanto, não é obtida uma estrutura do tipo dendograma por esses algoritmos. A ausência desse diagrama visual é a principal dificuldade em escolher o número de agrupamentos encontrados. Porém, algoritmos particionais são preferidos em aplicações que demandam grandes volumes de dados – dendogramas são impraticáveis a partir da análise de alguns milhares de padrões (39).

De modo geral, a técnica particional produz agrupamentos a partir da otimização de critérios definidos localmente, em um subconjunto dos dados, ou globalmente, considerando todos os padrões. A busca exaustiva de todos os agrupamentos possíveis dos dados é computacionalmente inviável, pois o número de combinações cresce exponencialmente à medida que aumenta o tamanho do problema. Em razão disso, as soluções mais adotadas executam iterativamente diversas vezes, a partir de diferentes estados iniciais, e a melhor configuração obtida é escolhida como resposta (2). Como visto na taxonomia apresentada na Figura 3.3, os principais critérios utilizados por algoritmos particionais são:

- **Erro Quadrático:** critério mais utilizado em algoritmos particionais, o objetivo dessa função é minimizar o erro quadrático entre cada padrão e o centróide ao qual o objeto pertence. Por essa característica, o método apresenta bons resultados em agrupamentos compactos e isolados. O algoritmo *K-means* (10) é a solução mais popular que utiliza essa estratégia.

- **Grafos:** técnica baseada na construção de uma árvore de espalhamento mínima (*Minimum Spanning Tree*) dos dados. Usando essa estrutura de dados é assegurado o peso mínimo da árvore, portanto, a remoção das arestas com os maiores pesos segmenta a árvore em agrupamentos distintos (40).

- ***Mixture Resolving:*** assume que os agrupamentos podem ser obtidos em uma das várias distribuições de probabilidade. A maior parte dos trabalhos usando essa abordagem pressupõe que os componentes individuais da mistura são Gaussianas. Alguns trabalhos utilizando essa técnica podem ser encontrados em (41, 42).

3.3 Algoritmo *K-means*

Embora tenha sido proposto há muitos anos, o algoritmo *K-means* (10) ainda é uma das soluções mais implementadas de agrupamento de dados. Seus pontos positivos são a facilidade de implementação e a eficiência do método, principalmente em agrupamentos bem definidos.

A metodologia é baseada na Equação 3.5, em que x_i representa os objetos do conjunto e μ_i é a média do agrupamento c_k. Seu objetivo é minimizar o erro quadrático entre cada padrão e o ponto médio do grupo (centróide). Porém, a minimização dessa função objetivo é um conhecido problema NP-difícil (43).

$$J(C) = \sum_{k=1}^{K} \sum_{x_i \in c_k} \|x_i - \mu_k\|^2. \tag{3.5}$$

O algoritmo é chamado *K-means* pois precisa conhecer a *priori* o número K de agrupamentos que devem ser encontrados, apesar de não existir um critério matemático bem estabelecido para encontrar esse valor. A estratégia mais comum é a execução do método com valores distintos de K e a solução mais representativa é escolhida. Além disso, diferentes inicializações de centróides podem gerar agrupamentos distintos, devido a sua convergência local. A solução para esse problema é executar o método múltiplas vezes com o mesmo K, com o resultado com o menor erro quadrático devendo ser escolhido. Seu funcionamento pode ser visto no Algoritmo 2.

Algoritmo 2 Etapas do algoritmo *K-means*

Entrada: conjunto de dados X com n objetos e k agrupamentos
Saída: dados particionados em k grupos

1: **procedimento** KMeans(X, k)
2: Defina k centróides iniciais ($c_1, c_2, ..., c_k$)
3: **faça**
4: Calcule a distância entre X_i e c_j, $\forall i \in [1, n], \forall j \in [1, k]$
5: Atribua o elemento X_i ao centróide c_j mais próximo
6: Recalcule os novos centróides ($c_1, c_2, ..., c_k$)
7: **enquanto** elementos mudarem de centróide
8: **retorne** dados particionados
9: **fim procedimento**

A construção dos agrupamentos usando o algoritmo *K-means* também é sensível à métrica usada para calcular a distância entre os objetos. A distância Euclideana é a medida mais utilizada e tem influência no formato esférico dos grupos. Contudo, outras métricas também já foram utilizadas, como a distância Mahalanobis, para encontrar agrupamentos elipsoidais (44).

O funcionamento do algoritmo *K-means*, usando a distância Euclideana, é mostrado nos itens da Figura 3.6. Os dados fornecidos são ilustrados na Figura 3.6(a) e os centróides iniciais estão em destaque na Figura 3.6(b). Note que a escolha dos centróides na primeira iteração influencia a escolha dos grupos. Porém, à medida que o algoritmo executa, os centróides são deslocados para o ponto médio dos agrupamentos esperados, visto nas Figuras 3.6(c) , 3.6(d) e 3.6(e), até alcançar sua última iteração, Figura 3.6(f).

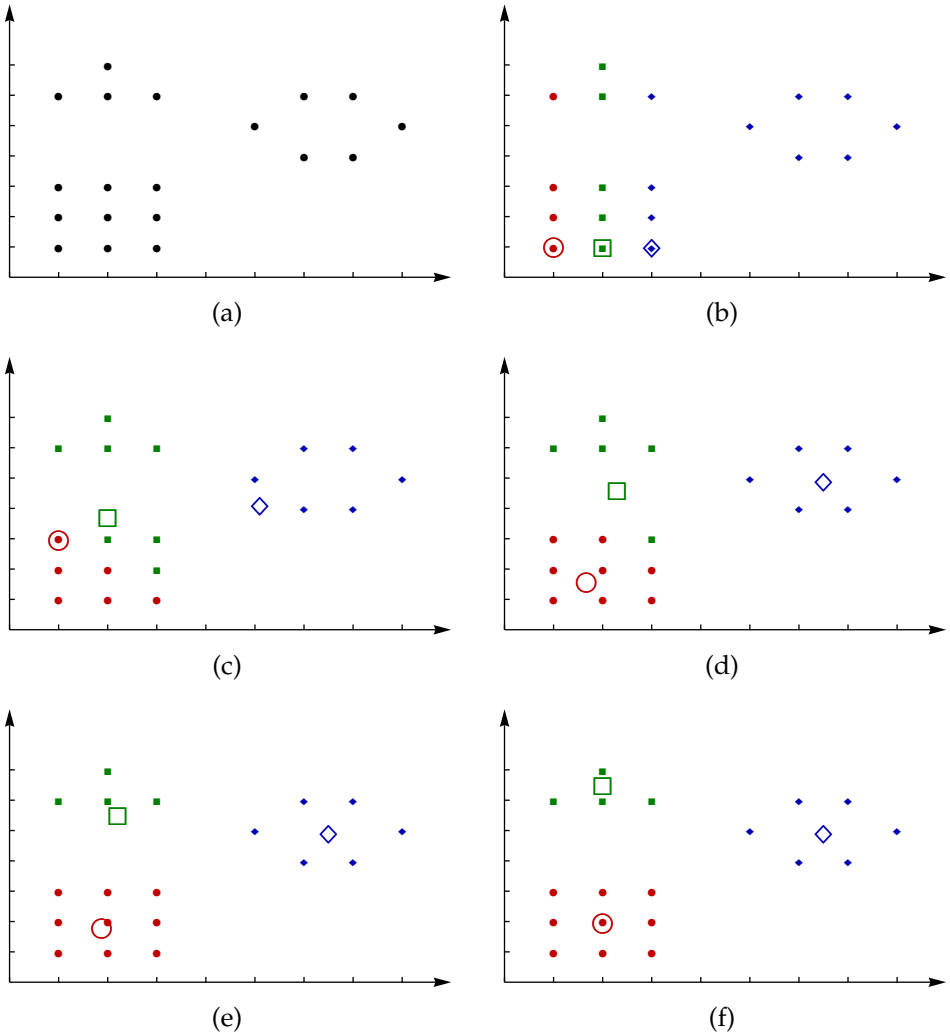


Figura 3.6: Execução do algoritmo *K-means*

3.4 Algoritmo *Fuzzy C-Means*

Algoritmos tradicionais de agrupamento como o *K-means*, adotam uma postura em que cada objeto pertence a um único grupo, sendo uma técnica do tipo *hard clustering*. Uma desvantagem desse método é que não são levadas em consideração as ambiguidades que surgem na classificação dos dados. Nesse sentido, os algoritmos que utilizam lógica *fuzzy* desempenham um papel importante no agrupamento de dados. Essa técnica explora a incerteza característica da lógica *fuzzy* para determinar os graus de pertinência dos objetos em relação aos grupos. Assim, quanto maior o grau de pertinência, maior é a probabilidade do elemento pertencer ao grupo.

Um dos algoritmos mais conhecidos de agrupamento de dados que utiliza a lógica *fuzzy* é chamado *Fuzzy C-Means* (FCM) (14, 15). O método é baseado no algoritmo *K-means* e também busca minimizar o erro quadrático da função objetivo. A Equação 3.6, a seguir, descreve a formulação matemática do FCM:

$$J = \sum_{i=1}^{n}\sum_{j=1}^{k} u_{ij}^{m}\|x_i - c_j\|^2 \quad 1 \leqslant m < \infty, \tag{3.6}$$

$$\text{sujeito à: } \begin{cases} 0 \leqslant u_{ij} \leqslant 1 \\ \sum\limits_{j=1}^{k} u_{ij} = 1, \forall i \in \{1, ..., n\} \\ 0 < \sum\limits_{i=1}^{n} u_{ij} < n, \forall j \in \{1, ..., k\}, \end{cases}$$

em que n é a quantidade de objetos, k é o número de agrupamentos, x_i representa cada objeto do conjunto de dados, c_j é o centro do j-*ésimo* grupo e u_{ij} o grau de pertinência do objeto x_i em relação ao grupo c_j. A matriz de pertinência u_{ij} e os centro do grupos c_j, podem ser calculados pela Equação 3.7:

$$u_{ij} = \frac{1}{\sum\limits_{s=1}^{k}\left(\frac{\|x_i - c_j\|}{\|x_i - c_s\|}\right)^{\frac{2}{m-1}}}, \qquad c_j = \frac{\sum\limits_{i=1}^{n} u_{ij}^{m} x_i}{\sum\limits_{i=1}^{n} u_{ij}^{m}}. \tag{3.7}$$

A estrutura geral do algoritmo FCM, mostrada no Algoritmo 3, é muito semelhante ao método *K-means*. Porém, o FCM incorpora maior custo computacional nos cálculos da matriz de pertinência e dos centros dos agrupamentos. Em relação a convergência, o FCM calcula a diferença absoluta entre iterações consecutivas da tabela de pertinência. Assim, a solução é encontrada quando a diferença entre as execuções atinge um limiar ε pré-estabelecido. O algoritmo também utiliza o número de iterações m como critério de parada, caso um valor inferior a ε não seja atingido.

Algoritmo 3 Etapas do algoritmo FCM

Entrada: conjunto de dados X com n objetos e k agrupamentos
Saída: matriz de pertinência dos objetos

1: **procedimento** FCM(X, k)
2: Defina os centros iniciais ($c_1, ..., c_k$)
3: **faça**
4: Calcule as distâncias $\|x_i - c_j\|$, $\forall i \in [1, n], \forall j \in [1, k]$
5: Calcule a matriz de pertinência
6: Calcule os novos centros dos grupos ($c_1, ..., c_k$)
7: **enquanto** critério de convergência não for satisfeito e não exceder m iterações
8:
9: **retorne** matriz de pertinência
10: **fim procedimento**

3.5 *Fuzzy Minimals*

Apesar da popularidade do algoritmo FCM, sua maior dificuldade é seu tempo de execução, que cresce exponencialmente em função do tamanho do problema. Assim como o *K-means*, o FCM também precisa conhecer previamente o número de agrupamentos em que o conjunto de dados deve ser dividido. Isso tem impacto direto no desempenho do algoritmo, pois é preciso executá-lo diversas vezes, com quantidades distintas de agrupamentos, até encontrar uma classificação adequada.

Diante dessas dificuldades, Flores-Sintas *et al.* propôs uma solução chamada *Fuzzy Minimals* (16). O algoritmo busca por representantes dos agrupamentos, chamados protótipos, sem qualquer necessidade de conhecimento prévio sobre o número de grupos. Os autores também demonstram em seu trabalho que essa solução satisfaz as principais características de um bom algoritmo de classificação, tais como adaptabilidade ao encontrar grupos de diferentes tamanhos e formatos, independência de dados sob a distribuição dos objetos, estabilidade na presença de ruídos e escalabilidade para lidar com crescentes volumes de dados (17).

A fundamentação matemática deste método pode ser encontrada em (16), (17) e (18). De modo geral, o problema pode ser enunciado do seguinte modo:

Seja $X = \{x_1, x_2, ..., x_n\} \subset \mathbb{R}^F$ um conjunto de n objetos com F dimensões, o método busca por protótipos a partir da minimização da função objetivo definida pela Equação 3.8:

$$J(v) = \sum_{x \in X} \frac{d_{xv}^2}{1 + r^2 d_{xv}^2} \ , \tag{3.8}$$

em que d_{xv}^2 é a norma Euclideana entre o ponto x e o ponto v, r é o fator que mede a isotropia no conjunto de dados, isto é, o momento a partir do qual a homogeneidade dos dados é quebrada e um novo grupo deve ser criado. O fator r, definido pela Equação 3.9, normaliza a distância Euclideana, tornando a densidade média da amostra igual a 1.

$$\frac{\sqrt{|C^{-1}|}}{n r^F} \sum_{x \in X} \frac{1}{1 + r^2 d_{xm}^2} \ = \ 1, \tag{3.9}$$

em que $|C^{-1}|$ é o determinante da matriz covariância inversa e d_{xm}^2 é a norma Euclideana entre x e m. Esse fator pode ser calculado usando algoritmos que resolvem equações não-lineares, como o método de Newton-Raphson, bissecção, etc.

A função objetivo, definida pela Equação 3.8, pode ser reescrita por meio da Equação 3.10, em que o termo μ_{xv} (Equação 3.11) mede o grau de pertinência do objeto x ao protótipo v. Esse termo é utilizado tanto na identificação dos protótipos quanto na vinculação de cada elemento do conjunto X aos protótipos identificados.

$$J(v) = \sum_{x \in X} \mu_{xv} d_{xv}^2 \ , \tag{3.10}$$

$$\mu_{xv} = \frac{1}{1 + r^2 d_{xv}^2} \ . \tag{3.11}$$

O algoritmo *Fuzzy Minimals* é um processo iterativo que minimiza a função objetivo por meio da Equação 3.12. Os valores mínimos encontrados por essa equação são protótipos que representam os agrupamentos. Esses protótipos estão a pelo menos um ε pré-estabelecido de distância entre si. Essa é uma das características que permitem que não seja necessário estipular o número de agrupamentos no conjunto de dados. Na prática, os protótipos são a saída produzida pelo algoritmo, pois eles representam grupos encontrados e, a partir deles, é possível classificar todos os elementos do conjunto, calculando a pertinência entre os objetos e os protótipos.

$$v = \frac{\sum\limits_{x \in X} \mu_{xv}^2 \, x}{\sum\limits_{x \in X} \mu_{xv}^2} \,. \tag{3.12}$$

De modo geral, as etapas deste método são ilustradas na Figura 3.7. Os objetos e o fator r são entradas para o algoritmo, a saída são os protótipos encontrados.

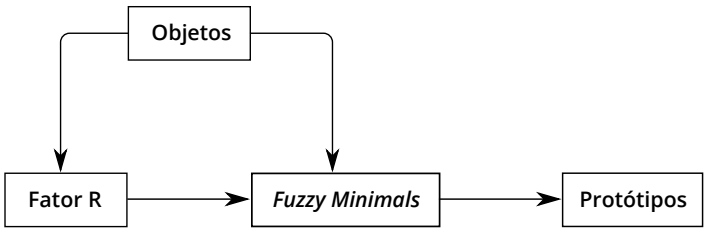


Figura 3.7: Diagrama do método *Fuzzy Minimals*

No Algoritmo 4 são apresentadas em alto nível as etapas do algoritmo *Fuzzy Minimals*. O procedimento recebe como entrada um vetor X com F dimensões, que representam as características do objeto, e o fator r, calculado previamente. O algoritmo calcula o valor v para cada objeto de entrada enquanto a diferença entre execuções sucessivas for maior que um ε_1. Se o valor encontrado não estiver ao menos ε_2 de distância de um valor adicionado anteriormente, esse é incluído no vetor V como um dos protótipos encontrados. Em outras palavras, ε_2 estabelece a distância mínima entre os protótipos. Sob o ponto de vista computacional, é notório que o laço iniciado na linha 5 demanda um grande esforço, pois a cada iteração é calculada a distância entre o elemento e todos os outros objetos.

Algoritmo 4 Etapas do algoritmo *Fuzzy Minimals*

Entrada: conjunto de dados X e fator r
Saída: protótipos identificados

procedimento FUZZYMINIMALS(X, r)
    $V = \{\} \subset \mathbb{R}^F$
    para k = 1 **até** N **faça**
        $v_0 = x_k$; $t = 0$; $E_0 = 1$
        enquanto $E_{(t)} \geqslant \varepsilon_1$ **faça**
            $t = t + 1$
            $\mu_{xv} = \dfrac{1}{1 + r^2 d_{xv}^2}$, usando v_{t-1}
            $v_t = \dfrac{\sum\limits_{x \in X} \mu_{xv}^2 \, x}{\sum\limits_{x \in X} \mu_{xv}^2}$
            $E_{(t)} = \sum\limits_{\alpha=1}^{F} (\, v_{(t)}^{\alpha} - v_{(t-1)}^{\alpha} \,)^2$
        fim enquanto
        se $\sum\limits_{\alpha=1}^{F} (\, v^{\alpha} - w^{\alpha} \,)^2 > \varepsilon_2, \forall w \in V$ **então**
            $V = V + \{v\}$
        fim se
    fim para
    retorne V
fim procedimento

3.6 Paralelizações em Agrupamento de Dados

Recentemente, os algoritmos de agrupamento de dados têm recebido muito destaque, entre outros motivos, pelo crescente interesse no comportamento das pessoas. Essas informações são extraídas de grandes conjuntos de dados, que ficaram popularmente conhecidos como *big data*. A análise desses dados, por meio de algoritmos de agrupamento de dados, requer um grande esforço computacional, tanto pelo volume de dados processados, quanto pela ordem de complexidade dessas soluções. Portanto, a paralelização desses algoritmos é inevitável.

De modo geral, as paralelizações mais comuns da área organizam os dados de dois modos. Primeiro, o conjunto de dados inteiro é avaliado por todos os processos. Nesse caso, é possível que os processos não se comuniquem entre si, pois todos têm acesso global aos dados. Segundo, os dados são particionados e cada processo opera em partições separadas. Já nesse caso, em geral, é necessária uma maior cooperação entre os processos. Isso pode ocorrer em uma fase intermediária ou ao final do algoritmo, garantindo que os agrupamentos produzidos representem o conjunto de dados inteiro, e não somente sua partição.

Existe uma vasta literatura sobre a paralelização dos algoritmos de agrupamento de dados. Devido à sua popularidade, o algoritmo *K-means* foi alvo das primeiras paralelizações da área (45), utilizando o modelo de troca de mensagens. Sob o ponto de vista de programação, o paradigma mestre/escravo é frequentemente utilizado (46, 47, 48, 49, 50). Além disso, outras implementações também exploram técnicas e plataformas mais recentes como OpenMP (51), CUDA (52) e *MapReduce* (53). Além disso, há muitos trabalhos sobre a paralelização do FCM (54, 55, 56) e de algoritmos de agrupamento hierárquico (57, 45, 58, 59).

Esses trabalhos evidenciam a importância da paralelização dos algoritmos de agrupamento de dados. Evidentemente, o desempenho de uma determinada solução está sujeito às condições do problema, como quantidade de dados analisados e tempo de execução aceitável. Assim, o que se espera dessas paralelizações é que encontrem respostas para problemas que eram impraticáveis, devido ao tempo excessivo de execução. Na prática esse tempo de execução depende do tipo de problema, por exemplo, a análise de dados de estações meteorológicas exige que o processamento esteja concluído a tempo para informar a população sobre eventos que irão ocorrer, sob o risco de produzir informações que já estão sendo constatadas. Por fim, mesmo aplicações com menos restrições de tempo de execução são beneficiadas pelo desempenho de soluções paralelas, pois podem processar mais dados no mesmo intervalo de tempo.

3.7 Considerações Finais

Este capítulo apresentou uma revisão bibliográfica sobre técnicas de agrupamento de dados. Isso permite uma melhor compreensão do escopo do projeto, contextualizando os desafios encontrados na área. Nesse sentido, as principais técnicas e algoritmos de agrupamento foram exploradas, destacando suas vantagens e desvantagens. Entre esses algoritmos encontrou-se uma proposta interessante chamada *Fuzzy Minimals*, que soluciona algumas dificuldades encontradas na área. Esse algoritmo é objeto de estudo deste projeto e maiores detalhes serão discutidos no próximo capítulo

Capítulo 4

Fuzzy Minimals em GPU

Os avanços na tecnologia da informação permitiram uma explosão no volume de dados armazenados. Cerca de 2,5 quintilhões de *bytes* (2,5 ExaBytes) são criados diariamente e 90% dos dados armazenados atualmente foram produzidos nos últimos dois anos (60). Esses dados são criados a partir de dispositivos tradicionais como computadores, mas também nos mais diversos equipamentos eletrônicos, como celulares, *tablets*, relógios inteligentes, TVs, etc. O uso de algoritmos de agrupamento para classificação e análise de dados permite extrair informações valiosas sobre o comportamento e as preferências das pessoas, auxiliando tomadas de decisão mais eficazes.

Conforme discutido no capítulo anterior, os algoritmos de agrupamento de dados têm um alto custo computacional, devido tanto à ordem de complexidade dos algoritmos disponíveis, quanto à quantidade de dados a serem analisados. Em função disso, um caminho natural para aumentar seu desempenho é a sua paralelização. Nesse sentido, muitos trabalhos da área têm explorado o paralelismo por meio de bibliotecas como MPI, OpenMP e CUDA (61, 62, 63).

A paralelização do algoritmo *Fuzzy Minimals* já foi objeto de estudo em (20), alcançando um *speedup* de até 10 vezes. Essa implementação, contudo, foi desenvolvida utilizando o *software* Matlab, que embora seja eficiente para resolver equações, não tem desempenho adequado para programas contendo laços, testes condicionais, etc. Nesse sentido, programas compilados apresentam um melhor desempenho, pois não necessitam de interpretadores. Atualmente, dentro da bibliografia pesquisada, não existe nenhuma implementação paralela em GPU do algoritmo *Fuzzy Minimals*. Esta é, portanto, uma oportunidade para investigar o algoritmo sob uma perspectiva ainda não explorada. Assim, ao longo deste capítulo, serão discutidos os principais aspectos no desenvolvimento dessa solução.

4.1 Paralelização do Método

De modo geral, a solução desenvolvida neste trabalho pode ser dividida nos itens descritos a seguir, que serão discutidos com maior profundidade ao longo deste capítulo. De acordo com o diagrama mostrado na Figura 4.1, os objetos particionados e o Fator R são passados como parâmetro para método *Fuzzy Minimals*, que executa em paralelo na GPU. O método retorna protótipos, e a partir deles é feito o agrupamento hierárquico. Em relação à versão clássica do *Fuzzy Minimals*, a paralelização do algoritmo, além de transformar o método para uma execução paralela em GPU, adiciona duas etapas: particionamento dos dados e agrupamento hierárquico.

- **Cálculo do Fator R:** esse valor global mede a isotropia do conjunto de dados completo, sendo o mesmo tanto para uma execução sequencial quanto para qualquer execução paralela.
- **Particionamento dos dados:** o conjunto de dados é dividido em partições, de modo que cada partição tenha acima de 10% dos dados, garantindo que cada partição tenha representantes significativos.
- ***Fuzzy Minimals* em GPU:** cada bloco de *threads* em GPU acessa somente uma partição dos dados, portanto, o mapeamento entre blocos em GPU e partições é de um para um. Em cada bloco, *threads* buscam localmente em sua partição por representantes dos agrupamentos, chamados protótipos.
- **Agrupamento hierárquico dos protótipos:** as *threads* de blocos distintos podem encontrar protótipos semelhantes aos identificados em outras partições, devido à distribuição dos dados. Para eliminar esse efeito colateral, os protótipos semelhantes são aglutinados usando agrupamento hierárquico.

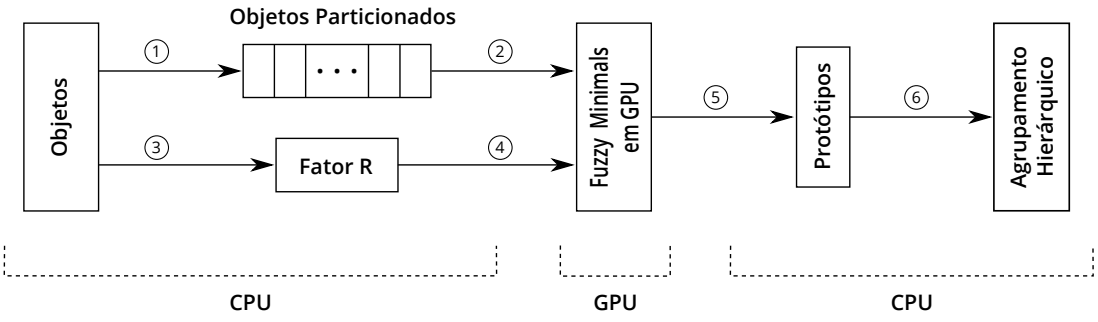


Figura 4.1: Diagrama da paralelização do método *Fuzzy Minimals* em GPU

4.2 Estrutura de dados

Um objeto, sob o ponto de vista de agrupamento de dados, representa um elemento x_i do conjunto $X = \{x_1, x_2, ..., x_n\} \subset \mathbb{R}^F$ com n objetos F-dimensionais, sendo que cada dimensão descreve um atributo do objeto. Computacionalmente, cada objeto pode simplesmente ser armazenado em um vetor. Essa facilidade inicial, porém, diminui a legibilidade do código conforme aumentam as operações realizadas sobre os objetos, e consequentemente, desvia o foco do objetivo do programa.

Com o objetivo de aumentar a legibilidade do código e incentivar a reutilização, cada objeto é representado por um tipo abstrato de dados, definido como *Object*. Cada objeto desse tipo armazena um vetor x com F posições. A vantagem dessa técnica é a utilização de uma operação disponível em linguagens orientadas a objetos, conhecida como sobrecarga de operadores. Por meio dessa operação é possível reescrever operações aritméticas básicas para também realizarem operações em tipos complexos como *Object*. Parte dessa estrutura de dados é mostrada no Algoritmo 5.

Algoritmo 5 Estrutura do tipo $Object$

```
1:  estrutura OBJECT contém
2:      x[ ] = ∅
3:      procedimento OPERATOR + ( Object a )
4:          Object p = ∅
5:          para i = 0 até F faça
6:              p.x[i] = a.x[i] + x[i]
7:          fim para
8:          retorne p
9:      fim procedimento
10: fim estrutura
```

Um exemplo simples, usando a operação de adição, é mostrado na Figura 4.2. Neste caso, o atributo i de um objeto é somado ao atributo i de outro objeto. Caso não fosse utilizada a sobrecarga de operadores, toda operação entre objetos deveria percorrer explicitamente o vetor, para realizar a operação desejada. Em vez disso, é utilizado um operador simples de adição, que implicitamente percorre o vetor de atributos, e portanto, elimina o uso de trechos desnecessários ao longo do código.

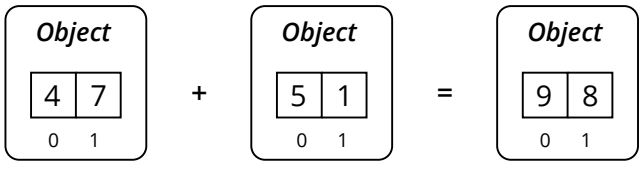


Figura 4.2: Soma de objetos utilizando sobrecarga de operador

4.3 Fator R

O Fator R, apresentado na Equação 3.8, exerce um papel importante na identificação dos agrupamentos, pois assume que existe uma homogeneidade no conjunto de dados. A partir do momento em que essa condição não é válida, é necessário criar um novo agrupamento ou associar o objeto a um outro protótipo. Em razão disso, esse é um valor global, isto é, deve considerar todos elementos do conjunto de dados para encontrar os protótipos. Essa etapa não é paralelizada, portanto, a mesma função é chamada pelas implementações serial e paralela do algoritmo *Fuzzy Minimals*.

O cálculo do Fator R envolve diversas operações matemáticas mas, de modo sucinto, é proporcional ao determinante da matriz covariância inversa, que multiplica a equação. Nas próximas subseções é destacado o método utilizado neste trabalho para calcular a matriz covariância e a resolução de equações não-lineares.

4.3.1 Matriz Covariância

A variância em um conjunto de dados mede o desvio dos elementos em relação à média em apenas uma dimensão. Por outro lado, a covariância, mede quanto as dimensões variam entre si em relação à média. Isso implica que a covariância entre uma dimensão e ela mesma é a própria variância. Portanto, a diagonal da matriz covariância é, na prática, a variância na dimensão. Outra propriedade interessante da matriz covariância é sua simetria em relação à diagonal principal, isto é, o elemento (x, y) é equivalente ao (y, x). Em relação à ordem da matriz, essa será quadrada em função do número de dimensões analisadas, isto é, a matriz covariância de d dimensões terá ordem $d \times d$.

A organização de uma matriz covariância com duas dimensões x e y é mostrada na Equação 4.1. Neste exemplo, por se tratar de duas dimensões, a matriz terá ordem 2×2. Em relação ao cálculo de cada elemento da matriz, o mesmo deve ser feito por meio da Equação 4.2 para quaisquer dimensões x e y, em que n representa o número de elementos do conjunto.

$$C = \begin{pmatrix} cov(x,x) & cov(x,y) \\ cov(y,x) & cov(y,y) \end{pmatrix} \tag{4.1}$$

$$cov(x,y) = \frac{\sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})}{n-1} \tag{4.2}$$

O procedimento que calcula a matriz covariância é descrito no Algoritmo 6. Este recebe como parâmetro os objetos do conjunto de dados e o seu tamanho. Inicialmente, a média dos objetos é calculada utilizando a sobrecarga do operador adição, ou seja, a soma das dimensões individuais é feita implicitamente. Em seguida, os valores dos elementos a partir da diagonal superior são calculados e o mesmo resultado é armazenado na posição simétrica.

Algoritmo 6 Cálculo da Matriz Covariância

Entrada: Objetos $object$ e sua quantidade n
Saída: Matriz covariância

```
1:  procedimento COVARIANCE(object[ ], n)
2:      para i = 1 até n faça
3:          mean = mean + object[i]
4:      fim para
5:      mean = mean / n
6:      para i = 1 até F faça
7:          para j = i até F faça
8:              sum = 0
9:              para t = 0 até n faça
10:                 sum += (object[t].x[i] − mean.x[i]) × (object[t].x[j] − mean.x[j])
11:             fim para
12:             cov[i][j] = cov[j][i] = sum / ( n − 1 )
13:         fim para
14:     fim para
15:     retorne cov
16: fim procedimento
```

4.3.2 Método de Newton-Raphson

Como é sabido, algumas classes de equações, por exemplo, polinômios de grau dois, podem ser resolvidos por meio de fórmulas bem definidas. Porém, no caso de polinômios de maior grau, encontrar as raízes reais não é uma tarefa trivial (64), principalmente quando resolvidas computacionalmente. Existem diversos métodos numéricos para encontrar raízes reais em polinômios. Independentemente do método, eles podem ser divididos em duas fases:

- **Fase I:** localização das raízes ou determinação de um intervalo que tenha a raiz.
- **Fase II:** melhoria do resultado encontrado, por refinamentos sucessivos, até que a raiz esteja dentro de uma precisão pré-estabelecida.

O método de Newton-Raphson é uma das técnicas mais populares na resolução de equações não lineares. Formalmente, podemos definir o método como segue. Seja $f(x)$ uma função bem definida e δ a raiz da equação $f(\delta) = 0$. Inicialmente, x_0 é uma estimativa de δ, em seguida, uma nova aproximação x_1 é produzida, a partir da intersecção da reta tangente ao polinômio $f(x)$ em x_0 com o eixo das abscissas. Algebricamente, o coeficiente angular da reta tangente no ponto é a derivada da equação no ponto. O método executa as aproximações enquanto a diferença entre x_n e x_{n+1} for maior que um ε pré-definido. Uma generalização do método é mostrada na Equação 4.3, em que $f'(x)$ é a derivada da função $f(x)$.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \ . \tag{4.3}$$

Graficamente é mais fácil compreender o método, como visto na Figura 4.3, usando um x_0 como estimativa inicial. A reta que tangencia $f(x)$ em x_0 produz uma nova estimativa x_1. A tendência é que a diferença entre x_n e x_{n+1} seja cada vez menor. Deste modo, a cada iteração o resultado é melhorado, até que seja pequeno o suficiente. Quando isso ocorrer, o valor encontrado representa a raiz da equação.

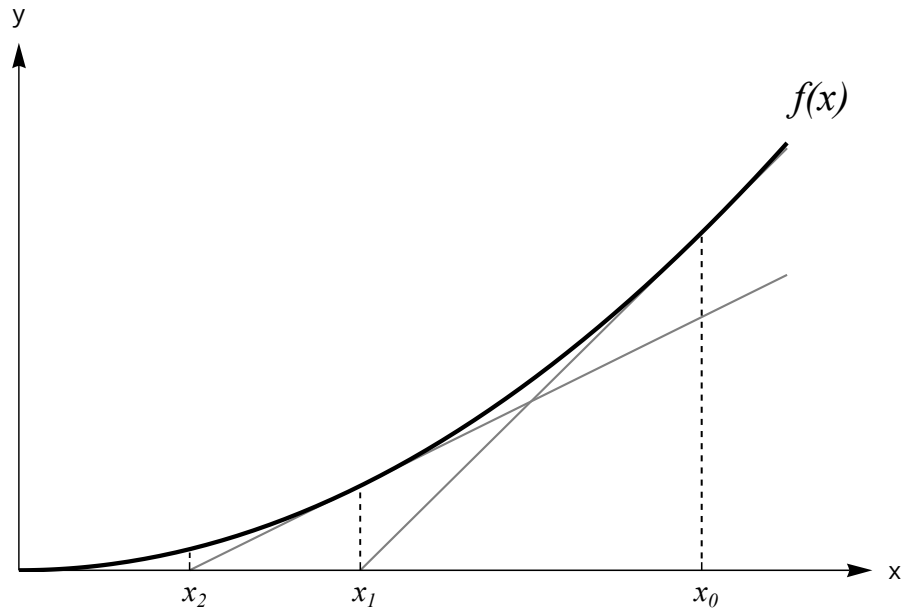


Figura 4.3: Iterações do método de Newton-Raphson

A convergência do método de Newton-Raphson é garantida para um certo intervalo $[a, b]$ que contém a raiz $f(x)$, desde que $f(x)$ e $f'(x)$ sejam contínuas nesse intervalo. Também é preciso atenção em relação a $f'(x)$, pois esse termo é um divisor na equação, portanto, seu resultado deve ser diferente de zero.

O método de Newton-Raphson é mostrado no Algoritmo 7. A função recebe como parâmetro a estimativa inicial x_0 e o resultado é refinado enquanto a diferença entre valores consecutivos é maior que a constante ε, definida previamente. O código chama duas funções $f(x_0)$ e $d(x_0)$, que calculam o valor da função e sua derivada no ponto x_0. No cálculo do Fator R, a função $f(x)$ é calculada usando a Equação 4.4, e sua derivada $f'(x)$ é determinada pela Equação 4.5, onde $|C^{-1}|$ é o determinante da matriz covariância inversa, F representa o número de dimensões do objeto e d^2_{xm} é a norma Euclideana entre x e m.

$$\frac{\sqrt{|C^{-1}|}}{n}\left(\sum_{x\in X}\frac{1}{r^F + r^{F+2}d^2_{xm}}\right) - 1 = 0. \tag{4.4}$$

$$\frac{\sqrt{|C^{-1}|}}{n}\left(\sum_{x\in X}\frac{-\left(\, Fr^{F-1} + (F+2)d^2_{xm}r^{F+1}\,\right)}{\left(\, r^F + r^{F+2}d^2_{xm}\,\right)^2}\right) = 0. \tag{4.5}$$

Algoritmo 7 Método de Newton-Raphson

Entrada: estimativa inicial x_0
Saída: raiz aproximada do polinômio

1: **procedimento** NewtonRaphson(x_0)
2: *diff* $= 1.0$
3: **enquanto** *diff* $\geqslant \varepsilon$ **faça**
4: $x_1 = x_0 - f(x_0) \,/\, d(x_0)$
5: *diff* $= |\, x_1 - x_0 \,|$
6: $x_0 = x_1$
7: **fim enquanto**
8: **retorne** x_0
9: **fim procedimento**

4.4 Particionamento dos Dados

Um aspecto importante na paralelização do algoritmo *Fuzzy Minimals* é o particionamento dos dados, isto é, a divisão do conjunto de dados em partes aproximadamente iguais, de modo que cada processo tenha uma carga de trabalho equivalente. Timón *et al.* (20) mostra empiricamente que cada partição deve ter acima de 10% do conjunto de dados para que o algoritmo encontre protótipos significativos. Alternativamente, caso cada partição contenha objetos que representam agrupamentos distintos, ou seja, partições completamente independentes entre si, não é necessário assegurar um percentual mínimo. Porém, essa situação não ocorre com frequência, pois na prática os dados são disponibilizados sem nenhuma separação prévia.

Garantir que cada partição tenha objetos significativos do conjunto de dados não é uma tarefa simples, pois depende do volume de dados e do modo como é feita a distribuição entre as partições. Algumas partições podem ter múltiplos representantes, enquanto outras podem ter menos objetos com as mesmas características, conforme ilustrado na Figura 4.4. Um modo de melhorar a distribuição dos objetos pode ser feito, por exemplo, utilizando uma função de distribuição aleatória uniforme. Porém, essa alternativa está sujeita à qualidade da função empregada. Assim, este trabalho utiliza uma outra alternativa. Nela os objetos são ordenados de forma crescente e, em seguida, distribuídos entre as partições. Deste modo, as partições têm maior uniformidade, aumentando as chances de alocar representantes significativos em cada partição.

É importante destacar que a ordem dos objetos em cada partição não altera o resultado final do algoritmo. O que pode influenciar o resultado são os objetos adicionados em cada partição. Portanto, pode haver uma variação nos protótipos identificados dependendo de quais objetos foram adicionados nas partições.

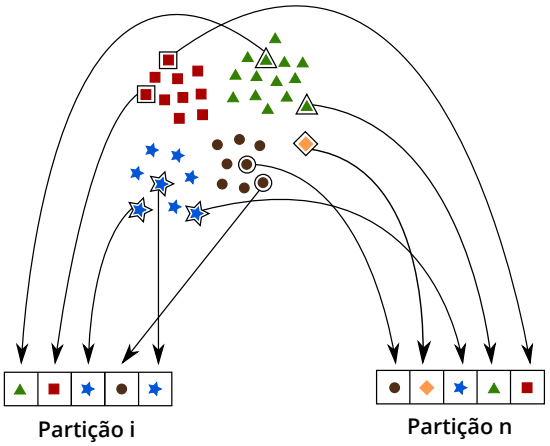


Figura 4.4: Distribuição dos objetos entre as partições

4.5 Agrupamento Hierárquico

Em razão da distribuição dos dados entre as partições, protótipos semelhantes podem ser encontrados isoladamente por *threads* trabalhando em partições distintas. Uma maneira de evitar protótipos parecidos como solução do algoritmo, é a aplicação de algoritmos de agrupamento hierárquico sobre os protótipos identificados. Deste modo, protótipos parecidos são aglutinados e apenas soluções relevantes são retornadas. Apesar do desempenho ruim de algoritmos de agrupamento hierárquico, esses podem ser aplicados na aglutinação de protótipos, uma vez que o número de protótipos é, em geral, muito inferior ao número total de objetos.

O algoritmo de agrupamento hierárquico de Johnson, discutido na subseção 3.2.1, esconde o custo computacional e a dificuldade de suas operações. Uma dessas dificuldades é como deve ser feita a intercalação entre dois objetos. Uma solução possível é armazenar o resultado da intercalação em um dos objetos e excluir o outro. Na prática, a exclusão de uma linha e uma coluna da matriz exigiria a reorganização da matriz inteira a cada iteração. Isso é impraticável conforme aumenta o número de objetos analisados.

Para melhor compreender a solução desenvolvida neste projeto considere quatro objetos $A = (2, 0)$, $B = (3, 0)$, $C = (8, 0)$ e $D = (10, 0)$. Assim, pelo algoritmo de Johnson, em um nível do dendograma, os objetos A e B devem formar um grupo e os objetos C e D outro conjunto. No próximo nível, esses conjuntos formam um novo agrupamento, conforme visto na Figura 4.5. A solução desenvolvida também inicializa a matriz de distância entre os objetos, conforme a Figura 4.6(a), mas não reorganiza as linhas e colunas da matriz. Para que isso ocorra é utilizado um vetor de índices, que identifica as posições na tabela que ainda não foram removidas. Outra vantagem dessa estratégia é que não é necessário percorrer a tabela inteira, bastando percorrer os itens contidos nesse vetor.

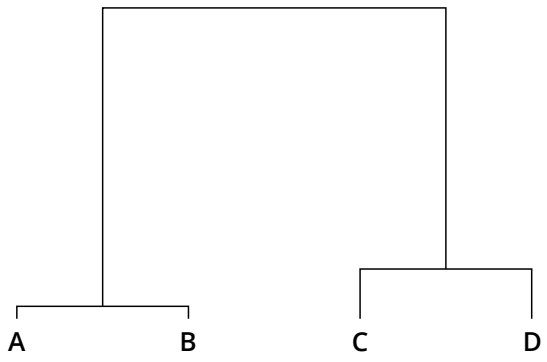


Figura 4.5: Dendograma obtido a partir dos objetos fornecidos

A solução desenvolvida é ilustrada na Figura 4.6. Considerando linhas e colunas distintas, a menor distância na Figura4.6(a) é encontrada na primeira linha, segunda coluna. Então, essas linhas/colunas devem ser intercaladas. Neste exemplo, o menor valor entre elas é armazenado na segunda linha/coluna, conforme visto na Figura 4.6(b) – as posições que não serão utilizadas novamente estão em destaque. Vale mencionar que quando uma posição deve ser removida do vetor de índices, ela é substituída pelo último elemento do vetor e seu tamanho é decrementado. O mesmo procedimento ocorre na Figura 4.6(c), que armazena a intercalação entre as linhas/colunas três e quatro. Por fim, todos objetos foram intercalados na Figura 4.6(d).

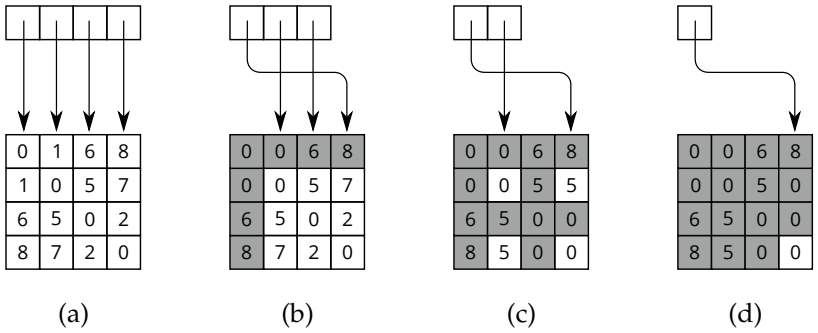


Figura 4.6: Intercalação de linhas e colunas sem reorganizar tabela

As etapas da solução desenvolvida são mostradas no Algoritmo 8. Neste algoritmo, o valor mínimo entre as linhas/colunas será armazenado na matriz D, cujas posições válidas estão guardadas no vetor de índices $v[s]$. Por outro lado, o valor armazenado em $v[r]$ será removido do vetor v. Assim, a cada iteração, será percorrido um elemento a menos, que foi removido do vetor v.

Algoritmo 8 Algoritmo de Johnson implementado

Entrada: conjunto de dados X
Saída: dendograma dos dados

1: **procedimento** JohnsonSingleLinkage(X)
2: Inicialize a matriz D com as distâncias entre cada par de objetos (X_i, X_j).
3: Inicialize o vetor v com os índices da matriz D
4: **faça**
5: Percorra o vetor v e encontre o objetos mais próximos $D[v[r]][v[s]]$, onde $r \neq s$
6: Atualize a matriz de distâncias D, para todo objeto k
7: $D[k][v[s]] = min(D[k][v[r]], D[k][v[s]])$
8: $D[v[s]][k] = min(D[v[s]][k], D[v[r]][k])$
9: Remova $v[r]$ do vetor v
10: **enquanto** todos os objetos não estiverem em apenas um agrupamento
11: **retorne** dendograma dos dados
12: **fim procedimento**

4.6 *Parallel Fuzzy Minimals on GPU* (PFMGPU)

A implementação em CUDA deste trabalho, chamada PFMGPU, pode ser dividida em duas etapas: programa principal e *kernel*. O programa principal faz a leitura dos dados, inicializa variáveis e demais operações necessárias para a execução do algoritmo. Já o *kernel* implementa a versão paralela em GPU do algoritmo *Fuzzy Minimals*. É importante destacar que o código desenvolvido não utiliza nenhum componente específico de uma arquitetura da Nvidia. Embora tenha sido compilado para a arquitetura Fermi, é possível compilá-lo para qualquer placa desse fabricante. Os detalhes da solução desenvolvida são descritos a seguir.

4.6.1 Considerações sobre a paralelização para GPU

A paralelização de um código, independentemente da plataforma utilizada, exige esforços adicionais em relação à programação sequencial. Questões como condição de corrida, sincronização, controle de acesso e compartilhamento de dados são necessários para garantir o funcionamento correto do algoritmo e o retorno de respostas válidas. Além disso, a paralelização para GPU também deve levar em consideração outros aspectos para explorar todo o potencial desse equipamento, como o uso dos diferentes níveis de memória, a ocupação dos SMs e a escolha do número de *threads*. Esses fatores aumentam as dificuldades no desenvolvimento de soluções para essa plataforma.

A arquitetura da GPU favorece aplicações que seguem o modelo SPMD (*Single Program Multiple Data*), devido a organização e a comunicação entre as unidades de processamento, armazenamento e controle. Nesse modelo, todas as *threads* executam o mesmo programa, mas podem seguir caminhos diferentes no código. Na solução desenvolvida, cada posição no vetor de dados é processada por apenas uma *thread*, conforme ilustrado na Figura 4.7, simplificando a paralelização e evitando o reprocessamento. Outra vantagem dessa estratégia é a eficiência de acesso à memória em posições contínuas, conforme discutido na Seção 2.5.

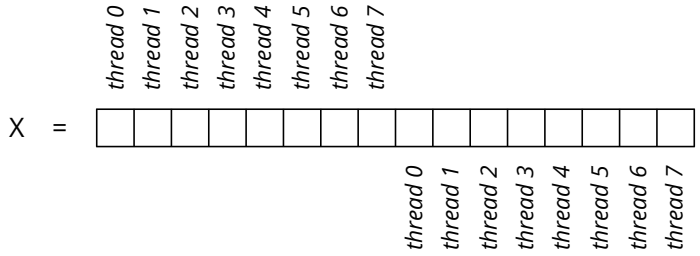


Figura 4.7: Execução das *threads* em trecho do conjunto de dados

O maior custo computacional do algoritmo *Fuzzy Minimals* ocorre no refinamento dos protótipos. Após essa etapa é preciso verificar se não existe um protótipo com pelo menos um ε de distância em relação às soluções encontradas anteriormente. Depois de verificar essa condição, o protótipo é adicionado ao vetor de respostas. Na paralelização do algoritmo, essa etapa deve garantir o sincronismo entre as *threads* para que haja troca de informações consistentes. Normalmente, isso pode ser feito de duas maneiras:

- **Semáforos:** por meio de semáforos é possível garantir que apenas uma *thread* esteja em uma região crítica da memória, evitando condições de corrida. Deste modo, cada *thread* pode ter acesso exclusivo ao vetor de respostas para verificar se não existe outro protótipo com pelo menos um ε de distância do valor encontrado pela *thread* atual. Assim, o semáforo garante que não existam *threads* fazendo leitura e escrita no vetor de soluções.
- **Barreiras:** esse mecanismo garante que todas *threads* estejam no mesmo ponto antes de continuar o processamento. Isso permite que os dados estejam em um estado consistente, portanto, não há risco que uma determinada *thread* esteja acessando outra região de memória ou processando outro trecho do código. Deste modo, uma *thread* pode acessar o vetor de respostas e verificar com segurança quais resultados obedecem ao critério de distância estabelecido.

Em CUDA não há uma implementação nativa de semáforos. Embora seja possível implementá-los por meio de instruções atômicas, o *framework* incentiva o uso de barreiras, fornecendo uma função específica para esse propósito. Em razão disso, os blocos de *threads* são sincronizados usando barreiras. Assim, ao final de cada iteração, cada *thread* deve armazenar o possível protótipo encontrado em uma região de memória exclusiva, mas visível entre as *threads* do bloco. Neste caso foi utilizada a memória compartilhada, devido a sua menor latência e a capacidade adequadada para o armazenamento desses valores. Após o armazenamento de cada *thread* na memória compartilhada, uma barreira garante que todas as *threads* estejam no mesmo ponto, e então uma *thread* verifica quais valores podem ser adicionados ao vetor de respostas.

Essa estratégia facilita a paralelização do código, pois não é preciso implementar e controlar os semáforos que acessam regiões críticas. O uso de barreiras permite que os dados estejam consolidados, assim apenas uma *thread* faz a verificação das soluções. Outra vantagem está relacionada ao desempenho superior de implementações nativas, pois empregam instruções específicas para a plataforma e não acrescentam a sobrecarga de funções desenvolvidas.

4.6.2 Programa Principal

O programa principal, descrito no Algoritmo 9, primeiramente define o número de blocos e de *threads* por bloco que serão utilizados pelo *kernel* (linhas 2 e 3). Cada bloco irá processar uma partição do conjunto de dados, sendo que é preciso que cada partição tenha pelo menos 10% dos dados. Para garantir que isso ocorra com certa folga é recomendado utilizar até oito blocos. O número de *threads* pode ser o máximo permitido na arquitetura, desde que exista no mínimo um objeto por *thread*. Isso é possível pois cada *thread* enxerga todos os dados do bloco. Se cada *thread* tivesse acesso a somente parte do bloco também seria recomendado utilizar no máximo oito *threads*.

Algoritmo 9 Programa principal da implementação PFMGPU

```
1:  procedimento MAIN
2:      b = número de blocos
3:      t = número de threads
4:      Leia objetos e armazene em X
5:      Distribua objetos de X entre as b partições
6:      Calcule o Fator R e armazene em r
7:      Aloque espaço dos objetos no device em devX
8:      Aloque espaço aos protótipos no device em devV
9:      Copie dados de X para devX
10:     Chame o kernel:
11:       PFMGPU <<< b, t >>> (r, devX, devV)
12:     Copie dados de devV em V
13:     Classifique hierarquicamente os protótipos em V e armazene em C
14:     Mostre os protótipos em C
15:     Libere espaços alocados
16: fim procedimento
```

Após a inicialização dessas variáveis, os objetos são lidos a partir de um arquivo de entrada (linha 4). Esses objetos devem ser distribuídos de modo que cada bloco tenha representantes significativos do conjunto de dados (linha 5). Neste trabalho isso é feito ordenando o vetor e distribuindo os objetos entre as partições. Essa alternativa é mostrada na Figura 4.8, na qual os objetos são divididos em duas partições lógicas, isto é, sem a necessidade de criar vetores separados. Isso ocorre pois apenas um vetor com todos os dados é passado para o *kernel*, e fica a cargo de cada bloco identificar sua partição. Cada partição tem representantes significativos do conjunto de dados como um todo. Independentemente da semelhança entre as partições, os protótipos identificados serão aglutinados na etapa de agrupamento hierárquico, portanto, não haverá protótipos duplicados como resposta final.

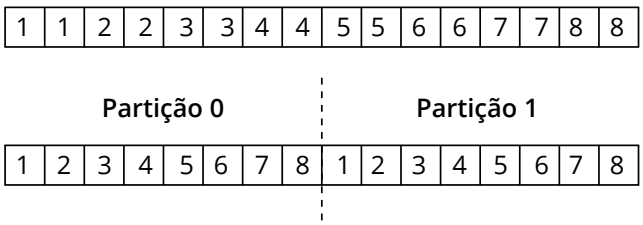


Figura 4.8: Distribuição dos dados entre partições

Após a distribuição dos dados é calculado o fator r usando o conjunto de dados inteiro (linha 6). De fato, o fator r é um valor global que se aplica a todas as partições. Também é importante destacar que a organização do conjunto de dados não interfere no cálculo desse valor. Além disso, a organização de uma partição em específico não altera os protótipos identificados pelo bloco de *threads*, pois elas irão verificar todos os objetos contidos na partição. O que influencia a identificação dos protótipos é se a partição tem representantes significativos dos objetos, sendo irrelevante o modo como eles estão organizados em uma partição.

Em seguida deve ser executada a função CUDA de alocação de espaço no *device*, para armazenar objetos e protótipos encontrados (linhas 7 e 8). Após essa etapa, é possível transferir os dados do *host* para o *device* (linha 9). Somente então é possível invocar o *kernel* (linhas 10 e 11), que na prática é o código que irá executar o algoritmo em paralelo. O *kernel* é executado com o número de blocos e de *threads* definidos previamente, além disso, são passados como parâmetro o fator r, um ponteiro para os objetos armazenados no *device* e outro para guardar os protótipos identificados em cada bloco.

Qualquer chamada ao *kernel* ocorre de modo assíncrono, isto é, o controle do programa retorna à função chamadora. Porém, a transferência dos protótipos armazenados no *device* só acontece após a conclusão do *kernel*, garantindo que os dados estejam em um estado consistente. Em posse dos protótipos identificados pelos blocos de *threads*, é feito o agrupamento hierárquico, para evitar valores semelhantes como resposta do algoritmo. Por fim, os protótipos encontrados são exibidos e os espaços em memória alocados são liberados.

4.6.3 *Kernel*

O *kernel* implementado é mostrado no Algoritmo 10 e seu funcionamento é descrito a seguir. A função recebe como parâmetro o fator r, o conjunto de dados com todos objetos e um vetor para armazenar os protótipos encontrados. Esse vetor deve ser passado como parâmetro da função, pois o *kernel* não retorna nenhum tipo de dado. Também é importante destacar que como esses parâmetros estão armazenados no *de-*

vice, para acessá-los a partir do *host* é preciso fazer uma transferência explícita. Isso ocorre, por exemplo, para transferir os protótipos encontrados.

Passar todos objetos ao *kernel* permite que cada bloco identifique o intervalo de sua partição em X. No algoritmo, esse intervalo é definido pelas variáveis $blockStart$ e $blockEnd$. A divisão dos objetos, e consequentemente seus limites, é definida de acordo com o número de blocos em que o *kernel* foi invocado. Na Figura 4.9 os objetos são divididos em duas partições e cada *thread* armazena variáveis locais com o início e final de seu bloco – $blockStart$ e $blockEnd$. Deste modo, nenhuma *thread* busca dados além dessa região.

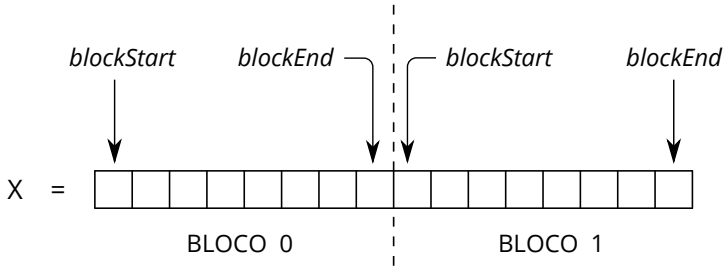


Figura 4.9: Divisão do conjunto de dados entre os blocos de *threads*

Após cada *thread* definir o limite de seu bloco são declaradas variáveis que identificam a *thread* no bloco e a quantidade de *threads* no bloco. Na prática, são utilizadas variáveis internas, disponíveis pela biblioteca CUDA, chamadas `threadIdx` e `blockDim`. Em seguida, na linha 6, são declaradas variáveis compartilhadas. Os valores armazenados por essas variáveis são visíveis por todas as *threads* de um mesmo bloco, isto é, *threads* de blocos distintos não têm acesso aos mesmos dados.

Em seguida, é realizado o principal laço do programa, a partir da linha 7. Cada *thread* executa o laço em função do início do seu bloco e do identificador de sua *thread*. Os incrementos são feitos em razão do número de *threads* do bloco. Dessa forma, cada posição no intervalo da partição de dados será processada por apenas uma *thread*, não havendo recálculo na posição. Essa operação é ilustrada na Figura 4.7. Nela, oito *threads* executam sobre um trecho da partição. Observe que qualquer *thread i* não executa na posição seguinte, mas sim na posição $i + n$, em que n é o número de *threads* no bloco.

O laço entre as linhas 10 e 17 é semelhante ao *Fuzzy Minimals* tradicional, exceto que neste caso ele ocorre em paralelo. Também vale destacar que os intervalos de cálculo nos somatórios são referentes ao intervalo da partição de X e não do conjunto completo.

Quando as *threads* terminarem o laço entre as linhas 10 e 17, elas irão armazenar o resultado calculado na variável compartilhada *tmp*, na posição correspondente ao seu

Algoritmo 10 *Kernel* do PFMGPU

Entrada: fator r, conjunto de dados X e vetor V para armazenamento de protótipos.
Saída: como qualquer *kernel*, não possui tipo de retorno.

1: **procedimento** PFMGPU($r,\ X[\,],\ V[\,]$)
2: $blockStart =$ índice inicial do bloco em relação a X
3: $blockEnd\ =$ índice final do bloco em relação a X
4: $id =$ identificador da *thread* no bloco
5: $dim =$ número de *threads* por bloco
6: $\_\_shared\_\_\ V_s[\,],\ tmp[\,],\ size = 0$
7: **para** $k = blockStart + id;\ k < blockEnd;\ k$ += dim **faça**
8: $v_{(0)} = X[k]$
9: $t = 0$; $E_0 = 1$
10: **enquanto** $E_{(t)} \geqslant \varepsilon_1$ **faça**
11: $t = t + 1$
12: $\mu_{xv} = \dfrac{1}{1 + r^2 d_{xv}^2}$, usando v_{t-1}
13:
14: $v_t = \dfrac{\sum\limits_{x \in X} \mu_{xv}^2\, x}{\sum\limits_{x \in X} \mu_{xv}^2}$
15:
16: $E_{(t)} = \sum\limits_{\alpha=1}^{F} (\ v_{(t)}^{\alpha} - v_{(t-1)}^{\alpha}\ )^2$
17: **fim enquanto**
18: $tmp[id] = v_{(t)}$
19: $\_\_syncthreads()$
20: **se** id == 0 **então**
21: **para** $i = 0$ **até** dim **faça**
22: **se** $\sum\limits_{\alpha=1}^{F} (\ tmp[i]^{\alpha} - w^{\alpha}\ )^2 > \varepsilon_2, \forall w \in V_s$ **então**
23: $V_s = V_s + \{tmp[i]\}$
24: $size = size + 1$
25: **fim se**
26: **fim para**
27: **fim se**
28: **fim para**
29: $\_\_syncthreads()$
30: **para** $i = id;\ i\ <\ size;\ i$ += dim **faça**
31: $V[i] = V_s[i]$
32: **fim para**
33: **fim procedimento**

índice. Isso ocorre pois somente uma *thread* do bloco, trecho entre as linhas 20 e 27, irá verificar se não existem protótipos parecidos e guardar a resposta. Deste modo evita-se a condição de corrida, tanto na verificação quanto no armazenamento do protótipo.

Após o final do laço mais externo, na linha 29, é utilizada uma barreira para garantir que todas as *threads* tenham terminado. Em seguida, os protótipos são copiados da memória compartilhada em V_s para a memória global em V, utilizando *memory coalescing*, encerrando a execução do *kernel*.

4.7 Considerações Finais

Neste capítulo apresentou-se as etapas do desenvolvimento deste projeto. Para tal, foi preciso realizar uma fundamentação teórica em torno do algoritmo *Fuzzy Minimals* tradicional, com destaque para os conceitos matemáticos em que o método é baseado. Em seguida, foram discutidas as estratégias e dificuldades de paralelização do algoritmo, como a distribuição e o particionamento dos dados, o acesso eficiente à memória e a sua implementação em CUDA.

Capítulo 5

Testes e Resultados

5.1 Considerações Iniciais

Neste capítulo serão apresentados os testes e os resultados obtidos com o trabalho desenvolvido. Esses resultados permitem uma análise crítica sobre eles, avaliando a eficácia da paralelização do método, tanto do ponto de vista de validade dos resultados quanto do seu desempenho.

Com o objetivo de ampliar a análise dos resultados, todos os testes foram realizados em quatro implementações distintas do método: sequencial, paralelas em CUDA, MPI e OpenMP. As implementações em MPI e OpenMP seguem os mesmos critérios adotados na versão desenvolvida em CUDA em relação ao particionamento dos dados, agrupamento hierárquico, etc. Essa implementação também será discutida neste capítulo. A implementação em OpenMP não produziu resultados atrativos e apenas parte desses resultados são apresentados ao final do capítulo

O principal objetivo dos testes realizados é mostrar a eficiência e o desempenho das implementações sob diversos agrupamentos. Para isso foram realizados quatro testes. O primeiro busca validar os resultados encontrados pelas diferentes implementações, sem considerar seu tempo de execução. Esse teste também é importante para familiarizá-lo sobre como os protótipos são encontrados quando há particionamento dos dados. Já os demais testes avaliam os resultados e o desempenho obtidos de conjuntos com 10, 20 e 40 mil objetos. Em todos os testes são mostrados os protótipos e os dendogramas correspondentes que geraram esses resultados.

5.2 *Parallel Fuzzy Minimals in MPI* (PFMMPI)

De modo geral, avaliações de desempenho comparam uma implementação sequencial com a solução paralela em uma determinada arquitetura. Neste trabalho o foco é na paralelização do algoritmo *Fuzzy Minimals* em GPU. Porém, com o objetivo de aumentar o escopo da avaliação, também foi desenvolvida uma solução usando trocas de mensagens (MPI – *Message Passing Interface*), chamada PFMMPI (*Parallel Fuzzy Minimals in MPI*), utilizando a paralelização em GPU como referência, ou seja, as mesmas estratégias empregadas na paralelização em GPU (particionamento dos dados, cálculo dos protótipos e agrupamento hierárquico) também se aplicam nesta solução.

Diversos fatores facilitam o desenvolvimento do código usando o modelo de trocas de mensagens. A implementação em MPI, por exemplo, não precisa se preocupar em explorar diversos níveis de memória, como ocorre em CUDA. Além disso, a sincronização é mais simples, pois existe apenas um processo por CPU. Assim, apenas um processo mestre deve aguardar a conclusão dos demais processos.

Essa solução é mostrada no Algoritmo 11. Inicialmente, nas linhas 2 e 3 são declaradas as variáveis que armazenam o número de processos que executarão o algoritmo e o identificador do processo em execução. Em seguida, na linha 4, apenas o processo com identificador 0 irá ler os objetos (linha 5), distribuí-los entre partições (linha 6) e calcular o fator r (linha 7). Na sequência, o processo 0 irá enviar aos demais processos o fator r, o tamanho de sua partição e seus objetos. Os demais processos, isto é, os processos com identificadores entre 1 e $n - 1$, receberão os dados enviados pelo processo 0 (linha 13).

Na linha 15 todos os processos têm à sua disposição os dados necessários para a execução da função *FuzzyMinimals*. Diferentemente da implementação em CUDA dessa função, que exige, por exemplo, ajustes no código para manipular os blocos de *threads*, a implementação em MPI chama uma cópia idêntica da versão sequencial, mostrada no Algoritmo 4.

Ao final da execução da função *FuzzyMinimals*, na linha 17 os processos de 1 a $n - 1$ enviam os protótipos armazenados em seus vetor local V ao processo 0. Já o processo 0, entre as linhas 19 e 21, recebe dos demais processos os protótipos enviados. Finalmente, na linha 24, o processo 0 realiza o agrupamento hierárquico dos protótipos.

Algoritmo 11 Programa principal da implementação PFMMPI

```
1: procedimento MAIN
2:     p = número de processos
3:     rank = identificador do processo
4:     se rank == 0 então
5:         Leia n objetos e armazene em X
6:         Distribua objetos de X entre as p partições
7:         Calcule o Fator R e armazene em r
8:         size = n/p
9:         para k = 1 até k < p faça
10:            Send(r, size, X[size * k])
11:        fim para
12:    senão
13:        Receive(r, size, X)
14:    fim se
15:    V = FuzzyMinimals(X, r)
16:    se rank > 0 então
17:        Send(V);
18:    senão
19:        para k = 1 até k < p faça
20:            V = V + Receive(V)
21:        fim para
22:    fim se
23:    se rank == 0 então
24:        Classifique hierarquicamente os protótipos em V e armazene em C
25:    fim se
26: fim procedimento
```

5.3 Ambiente de Testes

Os experimentos deste trabalho foram realizados no laboratório do Grupo de Sistemas Paralelos e Distribuídos (GSPD) da UNESP, composto por oito máquinas semelhantes, com as configurações a seguir:

- **Sistema Operacional:** Debian 3.16
- **Processador:** Intel Core i7 3.4Ghz
- **Memória RAM:** 16 Gb
- **GPU:** Nvidia GeForce GTS 450 (microarquitetura Fermi)

Os testes do algoritmo sequencial e da versão paralela para GPU utilizaram apenas uma dessas máquinas. Já os testes da versão paralela em MPI utilizaram um processo por máquina, com até as oito máquinas.

5.4 Teste de Acurácia

O principal objetivo deste teste é verificar se o algoritmo realmente encontra os agrupamentos esperados, independentemente do tempo de execução entre a versão sequencial e paralela. De fato, não se justifica o desenvolvimento de uma solução, por mais rápida que seja, sem que apresente os resultados corretos. Portanto, primeiramente é preciso garantir que as implementações sequencial e paralela retornem resultados confiáveis.

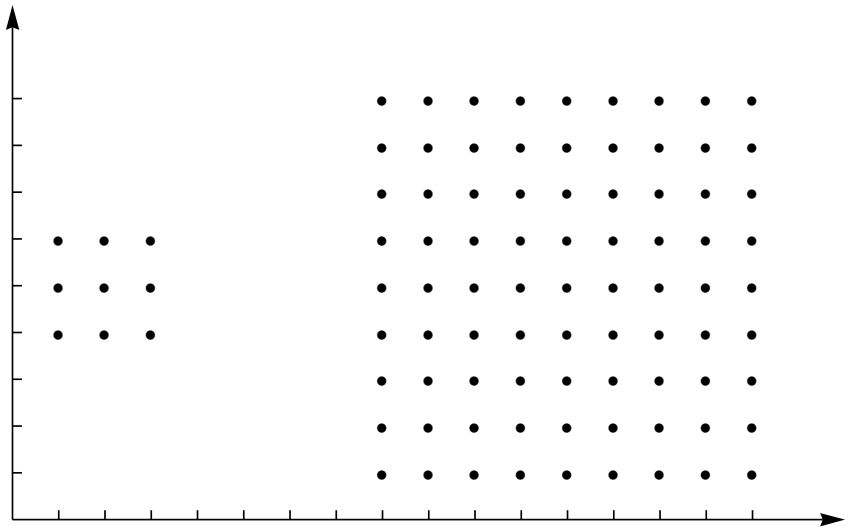


Figura 5.1: Exemplo com 90 elementos compostos de dois grupos

Evidentemente, inúmeros testes podem ser construídos nesse sentido. É uma boa prática, contudo, utilizar exemplos bem difundidos na área, para facilitar eventuais discussões e trabalhos futuros. Um teste bastante empregado foi extraído de (15). Nele, 90 elementos estão organizados em dois grupos bem definidos, um grupo com nove elementos e outro com 81, conforme mostrado na Figura 5.1. O resultado esperado, tanto para a versão sequencial quanto a paralela, é a identificação de protótipos que representem os dois conjuntos. Independentemente do valor numérico dos protótipos, o principal objetivo é separar os dados em dois grupos.

5.4.1 Execução Sequencial

A implementação sequencial do algoritmo para este teste calculou o fator r igual a 0.8927 e encontrou dois protótipos: $P_1 = (2.12, 4.96)$ e $P_2 = (11.43, 5.42)$. A partir dessas informações, é possível calcular a pertinência de cada objeto em relação aos protótipos, usando a Equação 3.11 descrita no Capítulo 3. Uma amostra dos objetos e sua pertinência aos protótipos é mostrada na Tabela 5.1. Note que a pertinência de cada objeto é maior para um determinado protótipo. Assim, o objeto deve estar vinculado ao protótipo com maior valor.

Objeto	$\mathbf{P_1}$	$\mathbf{P_2}$	**Objeto**	$\mathbf{P_1}$	$\mathbf{P_2}$	**Objeto**	$\mathbf{P_1}$	$\mathbf{P_2}$
(1, 4)	0.36437	0.01118	(8, 4)	0.03413	0.08310	(11, 4)	0.01548	0.36205
(1, 5)	0.49978	0.01137	(8, 5)	0.03502	0.09464	(11, 5)	0.01566	0.77242
(1, 6)	0.35074	0.01135	(8, 6)	0.03401	0.09352	(11, 6)	0.01546	0.70369
(2, 4)	0.56947	0.01359	(9, 4)	0.02534	0.13607	(12, 4)	0.01257	0.34975
(2, 5)	0.98775	0.01386	(9, 5)	0.02583	0.17002	(12, 5)	0.01269	0.71850
(2, 6)	0.53688	0.01384	(9, 6)	0.02527	0.16644	(12, 6)	0.01256	0.65866
(3, 4)	0.42349	0.01685	(10, 4)	0.01952	0.23481	(13, 4)	0.01041	0.21977
(3, 5)	0.61814	0.01727	(10, 5)	0.01981	0.35826	(13, 5)	0.01049	0.32438
(3, 6)	0.40519	0.01723	(10, 6)	0.01948	0.34273	(13, 6)	0.01040	0.31160

Tabela 5.1: Tabela de pertinência da execução serial de uma amostra dos objetos

Calculando a pertinência para todos objetos verifica-se que cada elemento possui maior probabilidade para apenas um protótipo, isto é, não existem valores semelhantes de pertinência para um mesmo objeto. Os agrupamentos encontrados, mostrados na Figura 5.2, destacam que cada objeto foi corretamente atribuído ao seu grupo. Além disso, a localização dos protótipos também é ilustrada na figura.

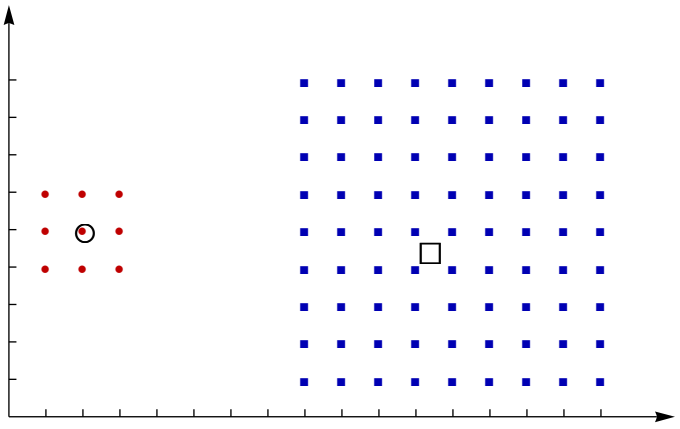


Figura 5.2: Resultado do agrupamento usando o algoritmo sequencial

5.4.2 Execução do PFMGPU com um bloco de *threads*

O fator r é um parâmetro global, que leva em consideração todos os objetos do conjunto de dados. Portanto, independentemente se o método *Fuzzy Minimals* executa de maneira sequencial ou paralela, o valor passado como parâmetro é exatamente igual. Na prática, essa função é idêntica tanto no código sequencial em C++ quanto na versão que utiliza a biblioteca CUDA. Assim, o valor encontrado para o fator r foi o mesmo, isto é, 0.8927.

Um caso especial da implementação proposta é a execução paralela com apenas um bloco de *threads*. Como os dados estão em apenas uma partição é esperado que o resultado seja exatamente igual à execução sequencial, pois cada *thread* enxerga todos os dados e calcula seu valor independentemente das demais. Portanto, a execução com apenas um bloco, com qualquer número de *threads*, deve sempre retornar os mesmos resultados. Neste exemplo, o *kernel* foi invocado com um bloco com 90 *threads*, portanto, uma *thread* por objeto do conjunto de dados.

A classificação dos agrupamentos é feita através dos protótipos, que na prática são a saída do algoritmo. Se os protótipos são iguais, a tabela de pertinência e, consequentemente o gráfico com os agrupamentos também serão semelhantes. Nesta execução paralela, o algoritmo encontrou exatamente os mesmos protótipos da execução sequencial, $P_1 = (2.12, 4.96)$ e $P_2 = (11.43, 5.42)$, conforme esperado. De fato, a tabela de pertinência, mostrada na Tabela 5.2, também é semelhante à execução sequencial.

Os resultados obtidos neste teste retornaram os valores esperados, tanto em relação ao fator r quanto aos protótipos identificados. Isso garante que não houve interferências entre as *threads* em execução do algoritmo.

Objeto	$\mathbf{P_1}$	$\mathbf{P_2}$	**Objeto**	$\mathbf{P_1}$	$\mathbf{P_2}$	**Objeto**	$\mathbf{P_1}$	$\mathbf{P_2}$
(1, 4)	0.36437	0.01118	(8, 4)	0.03413	0.08310	(11, 4)	0.01548	0.36205
(1, 5)	0.49978	0.01137	(8, 5)	0.03502	0.09464	(11, 5)	0.01566	0.77242
(1, 6)	0.35074	0.01135	(8, 6)	0.03401	0.09352	(11, 6)	0.01546	0.70369
(2, 4)	0.56947	0.01359	(9, 4)	0.02534	0.13607	(12, 4)	0.01257	0.34975
(2, 5)	0.98775	0.01386	(9, 5)	0.02583	0.17002	(12, 5)	0.01269	0.71850
(2, 6)	0.53688	0.01384	(9, 6)	0.02527	0.16644	(12, 6)	0.01256	0.65866
(3, 4)	0.42349	0.01685	(10, 4)	0.01952	0.23481	(13, 4)	0.01041	0.21977
(3, 5)	0.61814	0.01727	(10, 5)	0.01981	0.35826	(13, 5)	0.01049	0.32438
(3, 6)	0.40519	0.01723	(10, 6)	0.01948	0.34273	(13, 6)	0.01040	0.31160

Tabela 5.2: Tabela de pertinência da execução paralela de uma amostra dos objetos

5.4.3 Execução do PFMGPU com n blocos de *threads*

Aplicações em GPU que utilizam apenas um bloco de *threads* não exploram todo o potencial do *hardware*. Isso ocorre pois as *threads* do bloco executam em apenas um *Streaming Multiprocessor* (SM), enquanto os demais SMs permanecem ociosos durante todo tempo. Daí a importância de utilizar mais de um bloco, sempre que possível, em aplicações que demandam alto desempenho.

Neste trabalho dois motivos explicam a melhora de desempenho ao utilizar mais de um bloco. Primeiro, mais blocos utilizarão o poder de processamento do *hardware*. Segundo, os dados são divididos em partições, de modo que cada bloco busque por protótipos em apenas uma das partições, consequentemente, quanto mais partições, menor é o custo para encontrar os protótipos. Porém, a quantidade de blocos deve ser escolhida com atenção, pois quanto maior é o particionamento dos dados, menor é a precisão do método, pois menos objetos significativos estão em cada partição. Um particionamento eficiente deve conter acima de 10% dos dados.

Os protótipos encontrados por cada bloco de *threads* são mostrados na Tabela 5.3. Note que para um mesmo número de blocos, mas com diferentes números de *threads*, os protótipos encontrados são idênticos, conforme esperado. Isso confirma que o número de *threads* não interfere no resultado final. Por outro lado, a variação de blocos influencia a identificação dos protótipos, pois o particionamento implica em uma busca de protótipos local na partição, conforme explicado anteriormente.

Blocos	***Threads* por Bloco**	**Protótipos encontrados por cada bloco**
2	16	Bloco 0: $(1.57, 4.78), (8.72, 1.76), (11.13, 6.27)$
		Bloco 1: $(3.00, 4.80), (9.90, 3.84), (14.12, 7.32)$
2	32	Bloco 0: $(1.57, 4.78), (8.72, 1.76), (11.13, 6.27)$
		Bloco 1: $(3.00, 4.80), (9.90, 3.84), (14.12, 7.32)$
4	8	Bloco 0: $(1.06, 4.56), (8.07, 1.08), (9.94, 6.59), (14.93, 8.92)$
		Bloco 1: $(1.07, 5.91), (8.75, 1.90), (11.35, 7.08)$
		Bloco 2: $(2.03, 5.44), (9.36, 3.17), (13.16, 7.24)$
		Bloco 3: $(3.04, 4.47), (9.85, 4.74), (15.09, 7.78)$
4	16	Bloco 0: $(1.06, 4.56), (8.07, 1.08), (9.94, 6.59), (14.93, 8.92)$
		Bloco 1: $(1.07, 5.91), (8.75, 1.90), (11.35, 7.08)$
		Bloco 2: $(2.03, 5.44), (9.36, 3.17), (13.16, 7.24)$
		Bloco 3: $(3.04, 4.47), (9.85, 4.74), (15.09, 7.78)$

Tabela 5.3: Protótipos encontrados em função do número de blocos

Em execuções paralelas os protótipos semelhantes são aglutinados usando agrupamento hierárquico, conforme ilustrado na Figura 5.3. O dendograma gerado a partir da execução com dois blocos de *threads* é mostrado na Figura 5.3(a). Nessa figura é fácil verificar que os protótipos encontrados representam dois grupos bem definidos, separados por retângulos tracejados. Deste modo, a solução final é a média dos protótipos contidos em cada grupo, isto é, $P_1 = (2.28, 4.79)$ e $P_2 = (10.96, 4.79)$. Similarmente, os protótipos encontrados pela execução paralela com quatro blocos são mostrados na Figura 5.3(b) e a média dos protótipos é $P_1 = (1.80, 5.09)$ e $P_2 = (11.16, 5.38)$.

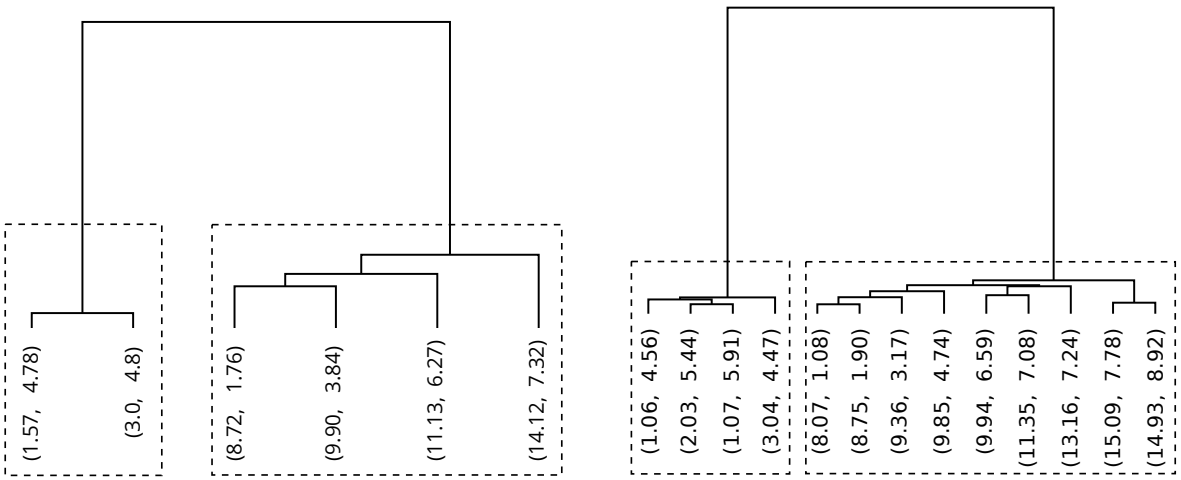


Figura 5.3: Dendogramas gerados a partir das execuções paralelas

Um resumo dos resultados encontrados é mostrado na Tabela 5.4. Os protótipos mostrados representam a solução final do algoritmo, portanto, após o agrupamento hierárquico das execuções paralelas em GPU.

Execução	**Blocos**	**Protótipos Finais**
Sequencial	-	$(2.12, 4.96), (11.43, 5.42)$
PFMGPU	1	$(2.12, 4.96), (11.43, 5.42)$
PFMGPU	2	$(2.28, 4.79), (10.96, 4.79)$
PFMGPU	4	$(1.80, 5.09), (11.16, 5.38)$

Tabela 5.4: Resumo dos resultados no primeiro teste

5.4.4 Execuções do PFMMPI

Na implementação em CUDA cada bloco de *threads* processa uma partição de objetos. Já na versão desenvolvida em MPI, cada partição é atribuída a um processo, com um processo por máquina. Assim, enquanto os protótipos são identificados pelos blocos de *threads* na versão em CUDA, na implementação em MPI esses resultados são encontrados por processos.

Os protótipos encontrados por cada processo são mostrados na Tabela 5.5. Os protótipos encontrados na execução com um processo são iguais aos resultados obtidos pela execução sequencial. Já nas execuções com dois e quatro processos, os protótipos encontrados são semelhantes aos obtidos, respectivamente, às execuções paralelas em GPU com dois e quatro blocos de *threads*.

Execução	**Processos**	**Protótipos encontrados por cada processo**
PFMMPI	1	Processo 0: $(2.12, 4.96), (11.43, 5.42)$
PFMMPI	2	Processo 0: $(1.57, 4.78), (11.13, 6.27), (8.72, 1.76)$
		Processo 1: $(9.90, 3.84), (14.12, 7.32), (3.00, 4.80)$
PFMMPI	4	Processo 0: $(1.06, 4.56), (8.07, 1.08), (9.94, 6.59), (14.93, 8.92)$
		Processo 1: $(1.07, 5.91), (8.75, 1.90), (11.35, 7.08)$
		Processo 2: $(2.03, 5.44), (9.36, 3.17), (13.16, 7.24)$
		Processo 3: $(3.04, 4.47), (9.85, 4.74), (15.09, 7.78)$

Tabela 5.5: Resultados obtidos no primeiro teste

A implementação em MPI também realiza o agrupamento hierárquico dos protótipos. Como os resultados encontrados são iguais aos obtidos pela execução em GPU, os dendogramas gerados também são semelhantes aos mostrados na Figura 5.3. O resultado final do algoritmo é mostrado na Tabela 5.6. Vale destacar que a saída final do algoritmo é igual à execução em GPU, garantindo que as duas implementações são equivalentes.

Execução	**Processos**	**Protótipos Finais**
Sequencial	-	$(2.12, 4.96), (11.43, 5.42)$
PFMMPI	1	$(2.12, 4.96), (11.43, 5.42)$
PFMMPI	2	$(2.28, 4.79), (10.96, 4.79)$
PFMMPI	4	$(1.80, 5.09), (11.16, 5.38)$

Tabela 5.6: Resultados obtidos no primeiro teste

5.5 Teste de acurácia e desempenho em dados sintéticos

Em todos os testes é dada a devida importância na classificação dos objetos. A partir deste teste, porém, também é feita a análise de desempenho do algoritmo. Esse binômio, precisão e desempenho, exerce um papel fundamental neste trabalho. Logo, neste teste, 10 mil pontos foram gerados aleatoriamente, de modo que representem três objetos bem definidos, conforme mostrado na Figura 5.4.

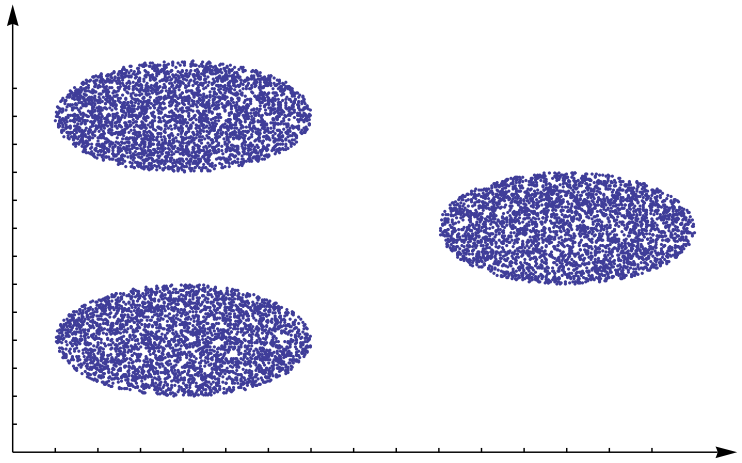


Figura 5.4: Objetos usados no segundo teste

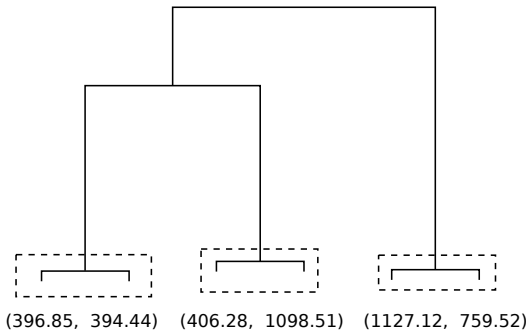
5.5.1 Protótipos

Os protótipos encontrados neste teste são mostrados na Tabela 5.7. As execuções paralelas em MPI e CUDA encontraram os mesmos protótipos. Por esse motivo, não há distinção entre execuções usando MPI ou CUDA. O número de partições é igual ao número de processos em MPI ou de blocos de *threads* em GPU. Assim como no exemplo anterior, os protótipos encontrados pela execução sequencial e a execução paralela com uma partição são exatamente iguais, pois não há particionamento dos dados.

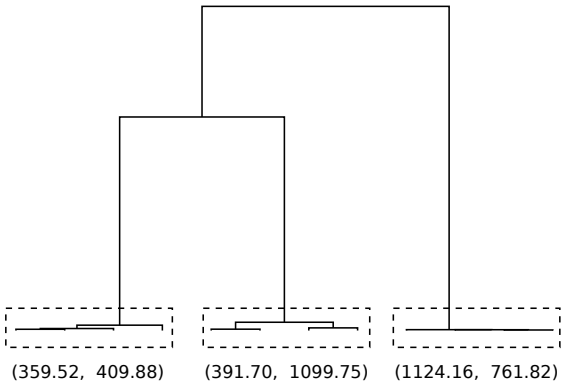
Execução	Partições	Protótipos Finais
Sequencial	1	$(407.84, 395.64), (393.62, 1084.59), (1125.10, 749.52)$
Paralela	1	$(407.84, 395.64), (393.62, 1084.59), (1125.10, 749.52)$
Paralela	2	$(396.85, 394.44), (406.28, 1098.51), (1127.12, 759.52)$
Paralela	4	$(359.52, 409.88), (391.70, 1099.75), (1124.16, 761.82)$
Paralela	8	$(361.11, 403.26), (350.18, 1105.10), (1111.99, 749.87)$

Tabela 5.7: Protótipos encontrados no segundo teste

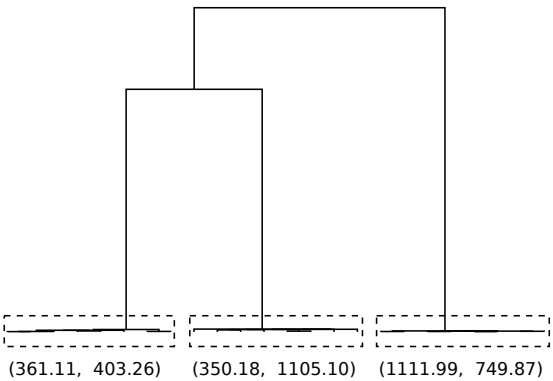
Os protótipos mostrados na Tabela 5.7 são a solução final do algoritmo, portanto, eles foram obtidos após o agrupamento hierárquico dos protótipos encontrados em cada partição. Os dendogramas obtidos a partir das execuções com duas, quatro e oito partições são mostrados, respectivamente, nas Figuras 5.5(a), 5.5(b) e 5.5(c). Percebe-se que em cada dendograma há uma clara divisão em três grupos, separados por retângulos tracejados. A média dos protótipos contidos em cada agrupamento, indicada nas figuras, representa os protótipos finais encontrados pelo algoritmo.



(a)



(b)



(c)

Figura 5.5: Dendogramas gerados no segundo teste

5.5.2 Tempo de Execução

Para garantir a precisão na coleta do tempo consumido, diversas execuções devem ser realizadas e o tempo final é o tempo médio dessas execuções. Neste teste, cinco execuções de cada implementação foram realizadas. A variância dessas execuções foi pequena, o que elimina possíveis interferências no resultado. Além disso, o tempo de comunicação é coletado tanto para o PFMGPU quanto para o PFMMPI, com o objetivo de medir o impacto no tempo total de execução.

Os resultados são mostrados na Tabela 5.8. Na execução em GPU há um bloco de *threads* por partição, com 960 *threads* em cada bloco. Já na implementação em MPI é utilizado um processo por partição, com até oito processos em máquinas distintas. Neste teste, as execuções paralelas obtiveram desempenhos expressivos, com destaque à execução do PFMGPU com oito partições, que foi aproximadamente 100 vezes mais rápida que a execução sequencial ou 29 vezes a sua execução com apenas uma partição.

Na implementação em MPI o maior custo de comunicação ocorre na transferência das partições para as máquinas, pois o custo de transferir as respostas encontradas é pequeno. Além disso, considerando que o arquivo de entrada tem apenas 83Kb, o custo de comunicação teve pouca influência sobre o tempo total de execução do programa. É interessante notar, contudo, que o custo de comunicação aumentou proporcionalmente ao número de processos, apesar de transferirem menos dados por processo. Isso ocorre pois existe uma parcela fixa no custo da comunicação, e esse valor aumenta em razão do número de processos. Já nas execuções em GPU, o custo de comunicação refere-se às transferências entre o *host* e o *device*. Considerando que a mesma quantidade de dados é transferida, independentemente do número de blocos, o tempo gasto é aproximadamente o mesmo. Por fim, o tempo de transferência é muito baixo em relação ao tempo total de execução.

Execução	**Partições**	**Tempo Médio**	**Variância**	**Tempo Comunicação**
Sequencial	1	170.596s	0.00602	–
PFMGPU	1	50.330s	0.00005	2.7124×10^{-5}s
PFMGPU	2	10.282s	0.00002	2.7156×10^{-5}s
PFMGPU	4	3.582s	0.00001	2.7125×10^{-5}s
PFMGPU	8	1.706s	0.00003	2.7156×10^{-5}s
PFMMPI	1	170.664s	0.00020	–
PFMMPI	2	47.395s	0.00092	1.7630×10^{-2}s
PFMMPI	4	11.341s	0.00044	4.4240×10^{-2}s
PFMMPI	8	3.020s	0.00009	1.0290×10^{-1}s

Tabela 5.8: Resultados obtidos no segundo teste

5.6 Teste sobre dados reais – menor volume

Neste teste são utilizados dados reais, extraídos do *U.S. Board on Geographic Names* (65). As coordenadas geográficas mostradas na Figura 5.6 representam a localização de 20237 escolas em quatro regiões bem definidas dos EUA, portanto, o dobro de objetos em relação ao teste anterior.

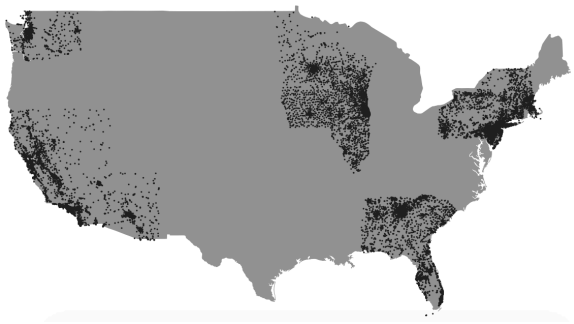


Figura 5.6: Objetos usados no terceiro teste

5.6.1 Protótipos

Os protótipos encontrados nas execuções com uma, duas, quatro e oito partições são destacados, respectivamente, nas Figuras 5.7(a), 5.7(b), 5.7(c) e 5.7(d).

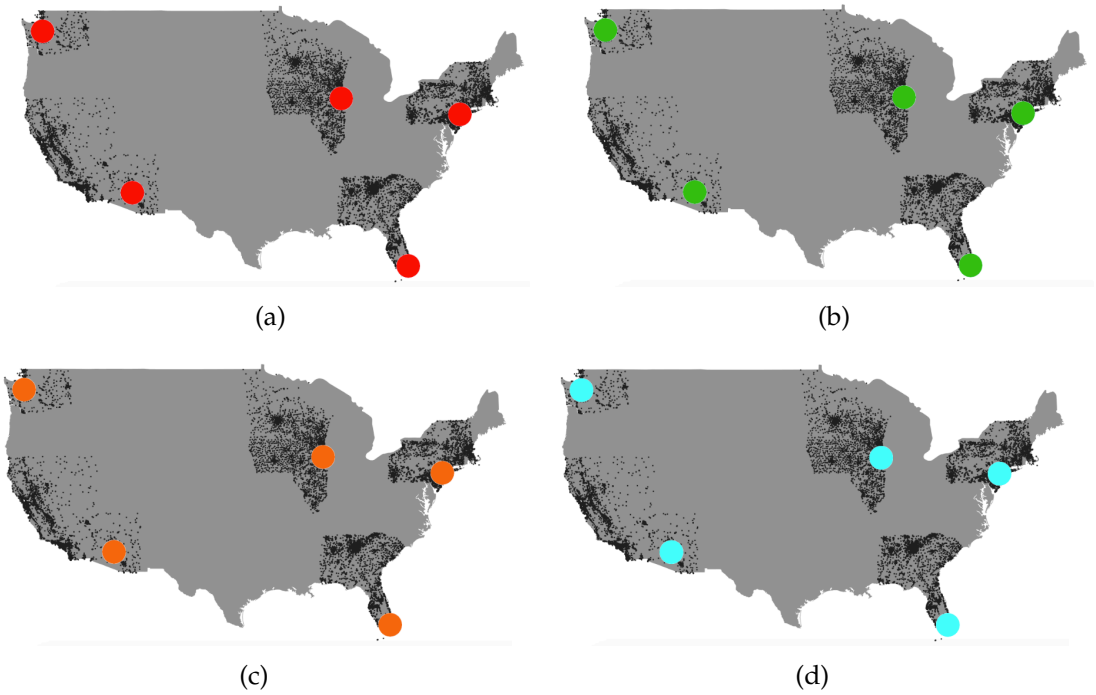


Figura 5.7: Protótipos encontrados no terceiro teste

Os protótipos mostrados na Figura 5.7, na escala utilizada, podem levar a conclusão que foram encontrados os mesmos resultados em todas as execuções, o que não é verdade. Há uma variação nos protótipos encontrados, devido aos diferentes objetos contidos em cada partição, mas em razão da escala temos a percepção que são iguais. Além disso, a uniformidade entre as partições aumenta a proximidade entre protótipos encontrados por diferentes processos.

Como esperado, a execução sequencial e as paralelas com apenas uma partição encontraram protótipos idênticos. Além disso, as execuções paralelas em MPI e CUDA encontraram os mesmos protótipos em execuções equivalentes, isto é, com o mesmo número de partições. Por esse motivo não há distinção entre processos encontrados por execuções em MPI ou CUDA.

Os dendogramas obtidos a partir das execuções com duas, quatro e oito partições são mostrados, respectivamente, nas Figuras 5.8(a), 5.8(b) e 5.8(c). Nessas figuras, os protótipos foram obtidos a partir do corte destacado com pontos tracejados. Percebe-se que o corte dos dendogramas nas posições indicadas produz cinco agrupamentos, que correspondem aos protótipos encontrados.

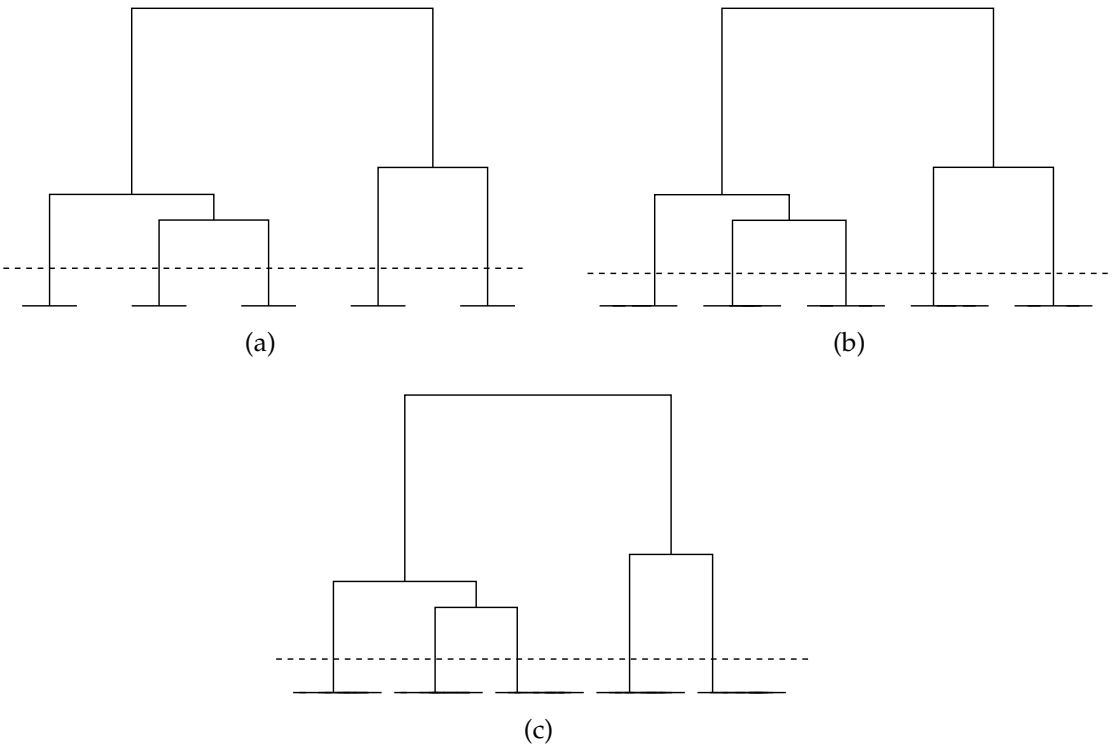


Figura 5.8: Protótipos encontrados no terceiro teste

5.6.2 Tempo de Execução

Os resultados do terceiro teste são mostrados na Tabela 5.9. Assim como no teste anterior, o tempo médio foi calculado a partir de cinco execuções. A precisão dos tempos coletados pode ser assegurada por sua baixa variância. Neste teste também foram utilizados entre um e oito processos na implementação em MPI, com um processo por partição. De modo semelhante, na implementação em CUDA, foram utilizados entre um e oito blocos com 960 *threads* cada, com uma partição por bloco.

Apesar de processar o dobro de objetos em relação ao teste anterior, os tempos de execução foram menores. Isso mostra que o tempo de execução não é necessariamente proporcional ao número de objetos, mas também deve-se considerar a densidade dos conjuntos e o número de agrupamentos.

Em relação ao desempenho das execuções, as implementações paralelas neste teste também alcançaram resultados consideráveis, sem prejuízo à qualidade dos resultados obtidos, conforme mostrado anteriormente. As execuções em GPU alcançaram menores tempos de execução quando comparados às execuções equivalentes em MPI, principalmente nas execuções com oito partições. Neste caso, a execução em GPU foi 52 vezes mais rápida que a versão sequencial e 21 vezes em comparação com sua execução com apenas uma partição. Enquanto isso, o PFMMPI com oito partições foi 43 vezes mais rápido que a execução sequencial.

O tempo de comunicação nas execuções em MPI foi pequeno. Embora, esse tempo represente uma parcela significativa na execução com oito partições, o tempo total gasto ainda compensa sua execução. Sobre tempo de comunicação do PFMGPU, houve um crescimento linear em relação ao teste anterior, em razão do aumento de dados. Além disso, conforme foi constatado no teste anterior, o tempo de comunicação não é influenciado pelo número de blocos.

Execução	**Partições**	**Tempo Médio**	**Variância**	**Tempo Comunicação**
Sequencial	1	52.683s	0.000088	–
PFMGPU	1	20.596s	0.000632	5.1855×10^{-5}s
PFMGPU	2	5.468s	0.000132	5.1880×10^{-5}s
PFMGPU	4	1.512s	0.000027	5.1906×10^{-5}s
PFMGPU	8	0.973s	0.000088	5.2066×10^{-5}s
PFMMPI	1	52.720s	0.000144	–
PFMMPI	2	13.739s	0.000161	3.9351×10^{-2}s
PFMMPI	4	3.667s	0.000054	1.2138×10^{-1}s
PFMMPI	8	1.203s	0.000102	2.4945×10^{-1}s

Tabela 5.9: Resultados obtidos no terceiro teste

5.7 Teste sobre dados reais – base completa

Neste teste, também são utilizados dados extraídos do *U.S. Board on Geographic Names* (65). Agora, as coordenadas geográficas de 40646 escolas dos EUA, mostradas na Figura 5.9, serão avaliadas, sem distinções bem definidas entre os agrupamentos.

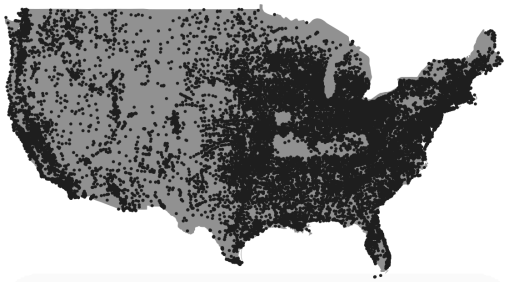
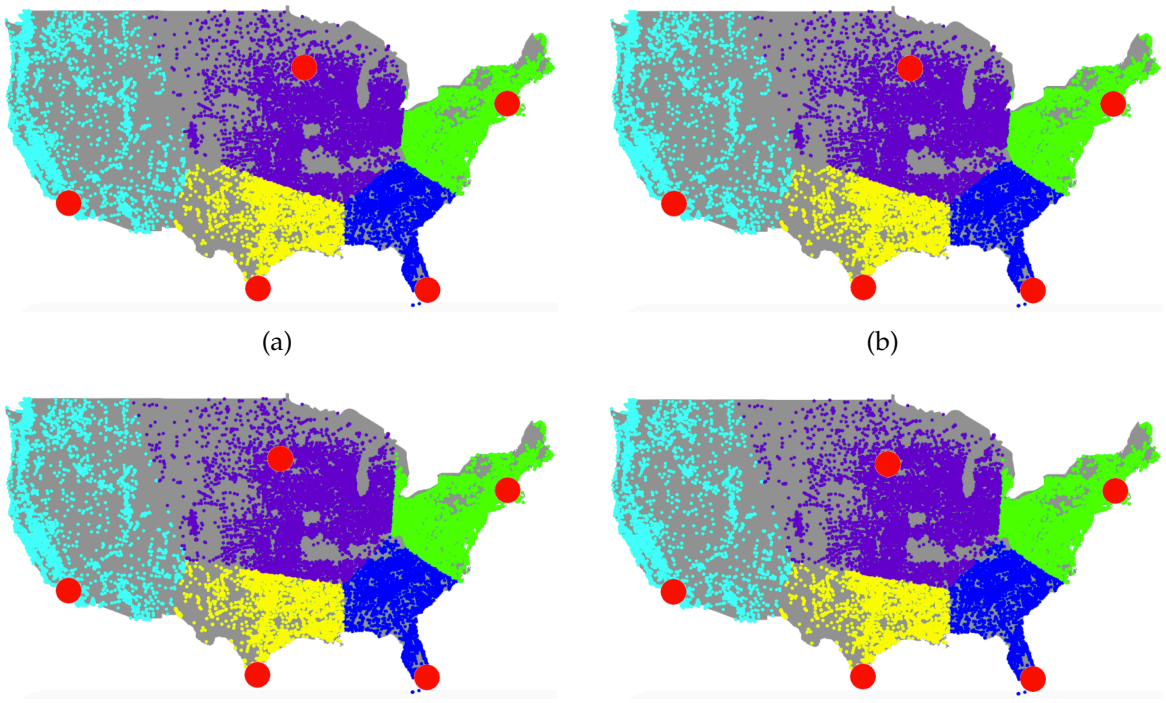


Figura 5.9: Objetos usados no quarto teste

5.7.1 Protótipos

Os protótipos encontrados nas execuções com uma, duas, quatro e oito partições são destacados em vermelho, respectivamente, nas Figuras 5.10(a), 5.10(b), 5.10(c) e 5.10(d). Os agrupamentos encontrados são mostrados em cores distintas.



(a) (b)

(c) (d)

Figura 5.10: Protótipos encontrados no quarto teste

É importante destacar que os agrupamentos da Figura 5.10 foram obtidos transformando o particionamento do tipo *fuzzy* em *hard*, isto é, cada objeto é atribuído exclusivamente a um conjunto, de acordo com seu grau de pertinência. Na prática, o *Fuzzy Minimals* oferece uma transição suave entre um agrupamento e outro, ou seja, não existe uma mudança abrupta entre os agrupamentos.

Os valores numéricos dos protótipos são mostrados na Tabela 5.10. Tanto a execução sequencial, quanto as execuções paralelas com apenas uma partição encontraram os mesmos protótipos, conforme esperado. As execuções paralelas em MPI e CUDA encontraram protótipos semelhantes em todas as execuções. Devido à escala dos mapas, os protótipos encontrados em cada execução parecem ser idênticos, porém há uma variação no seu resultado, em razão da distribuição dos objetos entre as partições. Como o grande volume de dados utilizado aumenta a homogeneidade das partições, a variação nesses resultados é menor que no caso anterior.

Execução	**Partições**	**Protótipos Finais**
Sequencial	1	$(26.03236, -80.25505), (26.24011, -98.08960), (33.97310, -118.07499)$ $(42.17509, -71.88173), (44.92812, -93.23445)$
Paralela	1	$(26.03236, -80.25505), (26.24011, -98.08960), (33.97310, -118.07499)$ $(42.17509, -71.88173), (44.92812, -93.23445)$
Paralela	2	$(26.02220, -80.25686), (26.24081, -98.09069), (33.97266, -118.07321)$ $(42.23272, -71.76836), (44.92741, -93.23358)$
Paralela	4	$(26.01952, -80.25754), (26.24076, -98.08957), (33.97288, -118.07297)$ $(42.28238, -71.66564), (44.70104, -95.73210)$
Paralela	8	$(26.02024, -80.25769), (26.24147, -98.08879), (33.97271, -118.07298)$ $(42.32400, -71.60247), (44.42456, -95.55855)$

Tabela 5.10: Protótipos encontrados no quarto teste

Os dendogramas obtidos a partir das execuções com duas, quatro e oito partições são mostrados, respectivamente, nas Figuras 5.11(a), 5.11(b) e 5.11(c). Note que há uma divisão em cinco grupos em cada dendograma na altura tracejada, e os protótipos finais são obtidos a partir da média desses grupos. Apesar da semelhança entre os dendodramas, à medida que mais partições são adicionadas, há um aumento do número de protótipos encontrados em cada partição, diminuindo a distância entre os grupos na base do dendograma.

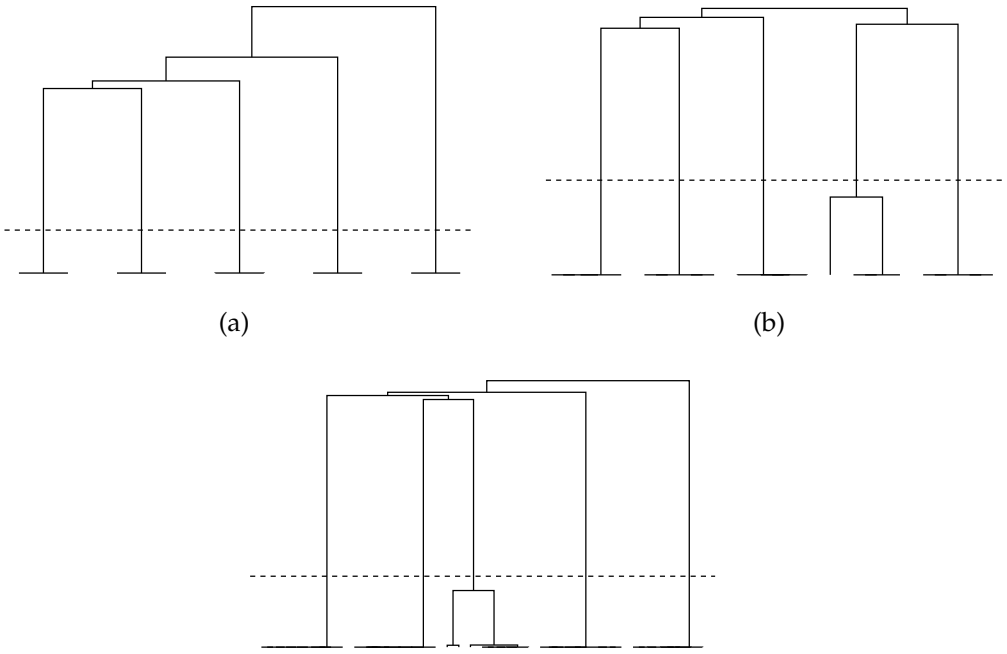


Figura 5.11: Protótipos encontrados no quarto teste

5.7.2 Acurácia

Conforme mostrado nos testes deste capítulo, o particionamento do dados provoca uma variação nos protótipos encontrados, pois diferentes conjuntos de dados estão sendo analisados. Normalmente, em agrupamentos bem definidos essa variação não têm impacto na atribuição dos objetos aos grupos. Porém, quando não há uma clara distinção entre os agrupamentos, uma variação mínima nos protótipos encontrados pode alterar os agrupamentos obtidos. Por esse motivo é importante que as partições tenham representantes significativos do conjunto de dados inteiro. Além disso, essa variação é reduzida à medida que aumenta a quantidade de dados analisados.

A acurácia dos agrupamentos formados em relação à execução sequencial é mostrada na Tabela 5.11. Como as execuções paralelas em MPI e CUDA encontraram exatamente os mesmos protótipos, não há distinção entre a acurácia dessas execuções. Os agrupamentos formados das execuções paralelas com uma partição são exatamente iguais aos obtidos na execução sequencial, por esse motivo a acurácia é de 100%. Ao acrescentar partições, há uma variação no conjunto de dados analisado, influenciando a acurácia do resultado final. Neste teste, a variação máxima foi em torno de 3% em relação à execução sequencial, o que garante a precisão do método na formação de agrupamentos.

Execução	Partições	Acurácia
Sequencial	1	100%
Paralela	1	100%
Paralela	2	99.79%
Paralela	4	96.67%
Paralela	8	96.87%

Tabela 5.11: Acurácia dos agrupamentos em relação à execução sequencial

5.7.3 Tempo de Execução

Os resultados do quarto teste são mostrados na Tabela 5.12. Assim como os testes anteriores, o tempo médio foi calculado a partir de cinco execuções. Também como nos testes anteriores, houve uma baixa variância, assegurando precisão nos tempos coletados. A metodologia de um a oito processos na implementação em MPI, bem como a utilização de um a oito blocos com 960 *threads* em GPU também foi empregada.

As implementações paralelas neste teste também obtiveram desempenho significativo. Com destaque às execuções em GPU, que alcançaram menores tempos de execução quando comparados às execuções semelhantes em MPI. Até quatro partições, o tempo médio da execução em GPU é aproximadamente a metade do tempo da implementação em MPI. Essa diferença diminui na execução com oito partições, pois a GPU utilizada possui apenas quatro *Streaming Multiprocessors* (SMs).

Neste experimento os ganhos das execuções em GPU foram próximos aos obtidos no teste anterior. Na execução com oito blocos, o PFMGPU foi 52 vezes mais rápido que a execução sequencial e 28 vezes melhor quando comparado a sua execução com apenas uma partição. Já o desempenho do PFMMPI para oito processos foi melhor, sendo 57 vezes menor que a execução sequencial.

O tempo de comunicação gasto nas execuções em MPI foi pequeno em relação ao tempo total de execução, devido ao tamanho do arquivo de entrada com apenas 494Kb. Esse custo aumenta em função do número de processos, devido aos custos de transmissão de dados explicados anteriormente. Em relação ao tempo de transmissão do PFMGPU, conforme verificou-se no teste anterior, houve um crescimento proporcional ao número de objetos manipulados, com baixa variação entre execuções com diferentes quantidades de blocos.

Execução	Partições	Tempo Médio	Variância	Tempo Comunicação
Sequencial	1	248.284s	0.010087	–
PFMGPU	1	133.945s	0.000231	1.0101×10^{-4}s
PFMGPU	2	40.166s	0.000657	1.0104×10^{-4}s
PFMGPU	4	8.963s	0.000035	1.0095×10^{-4}s
PFMGPU	8	4.724s	0.000433	1.0109×10^{-4}s
PFMMPI	1	248.447s	0.000621	–
PFMMPI	2	62.510s	0.000044	7.2839×10^{-2}s
PFMMPI	4	14.974s	0.000135	2.2059×10^{-1}s
PFMMPI	8	4.323s	0.000070	4.9745×10^{-1}s

Tabela 5.12: Resultados obtidos no quarto teste

5.8 Teste usando OpenMP

Como processadores modernos possuem a capacidade de processamento paralelo por sua estrutura *multi-core* é interessante investigar também a aceleração obtida em sua implementação em memória compartilhada, usando, por exemplo, o OpenMP. Por meio desse padrão, o programador consegue paralelizar laços usando diretivas de compilação, sem a necessidade de criar *threads* explicitamente. Além disso, essa solução também fornece funções e variáveis ambiente para lidar com outras questões comuns em programação paralela, como barreiras, variáveis compartilhadas e instruções atômicas.

Desse modo, desenvolveu-se uma solução em OpenMP para o algoritmo de *Fuzzy Minimals*. Essa solução também segue os princípios adotados nas outras implementações paralelas deste trabalho, como particionamento dos dados, cálculo do fator R e classificação hierárquica dos protótipos. É importante destacar que, ao contrário das outras soluções paralelas apresentadas, a paralelização em OpenMP oferece uma baixa invasão no código sequencial, devido ao uso de suas diretivas, facilitando o mapeamento para este ambiente.

A solução desenvolvida em OpenMP, chamada PFMmp, é mostrada no Algoritmo 12. Inicialmente, é definido o número de partições, que também será usado para especificar a quantidade de *threads*, ou seja, há uma *thread* por partição. Em seguida, os objetos são lidos, distribuídos uniformemente entre as partições e o fator R é calculado. Na linha 7 é utilizada a diretiva fornecida pelo OpenMP, definindo o número de *threads* que irão chamar a função *Fuzzy Minimals*. Essa função é exatamente a mesma implementação sequencial, ou seja, a paralelização ocorre apenas na chamada ao seu código. Por fim, após o processamento das *threads*, é feita a classificação hierárquica dos protótipos.

Algoritmo 12 Programa principal da implementação PFMmp

```
1: procedimento MAIN
2:     p = número de partições
3:     Leia n objetos e armazene em X
4:     Distribua objetos de X entre as p partições
5:     Calcule o Fator R e armazene em r
6:     size = n/p
7:     omp parallel num_threads(p)
8:         id = identificador da thread
9:         V[id] = FuzzyMinimals(X[size * id], r)
10:    Classifique hierarquicamente os protótipos em V
11: fim procedimento
```

Para avaliar o desempenho desta implementação foi utilizado o mesmo conjunto de dados discutido na Seção 5.7. Conforme esperado, os protótipos encontrados são exatamente os mesmos mostrados anteriormente naquela seção, na Tabela 5.10. Já os tempos de execução são mostrados na Tabela 5.13. Apesar do processamento local, sem o custo de comunicação entre ambientes distintos, todas as execuções em OpenMP tiveram tempos de execução superiores às execuções em CUDA e MPI. Nos demais casos de teste, apresentados nas seções anteriores, esse padrão de desempenho se repetiu, não sendo aqui incluídos.

Execução	Partições	Tempo Médio	Variância
PFMmp	1	249.037s	0.000655
PFMmp	2	65.819s	0.000157
PFMmp	4	16.477s	0.000297
PFMmp	8	7.492s	0.000451

Tabela 5.13: Resultados obtidos no quarto teste

5.9 Considerações Finais

Neste capítulo foram apresentados os testes e resultados com o PFMGPU. Primeiramente, foi apresentada a validação da paralelização do método, garantindo que tanto a versão sequencial quanto as implementações paralelas retornassem resultados esperados. Em seguida, os demais testes avaliaram o desempenho das implementações sob quantidades, densidades e números de agrupamentos distintos. As implementações paralelas apresentaram desempenhos consideráveis, com destaque à implementação PFMGPU, que, em geral, obteve melhores resultados em relação às implementações em MPI e OpenMP.

Capítulo 6

Conclusões

Atualmente estamos presenciando uma revolução na coleta e no processamento de grandes volumes de dados conhecidos como *big data*. Nessa nova era, a informação é o ativo mais valioso que as organizações possuem, podendo transformar dados brutos, que olhados de maneira isolada não agregam valor, em um conhecimento robusto sobre o comportamento dos indivíduos. Esses dados são frequentemente utilizados com objetivos comerciais, porém também podem ser explorados pelo poder público, por exemplo, na identificação e controle de epidemias, oferecendo melhores serviços de acordo com a necessidade de cada região, reconhecendo tendências, etc.

Foi visto ao longo deste trabalho que a análise dessas informações não é uma tarefa trivial. Por esse motivo diversos algoritmos e muitas melhorias foram propostas nos últimos anos. Algumas dessas soluções são eficientes em contextos específicos, como, por exemplo, determinadas distribuições de conjuntos de dados, mas não apresentam resultados satisfatórios em outras situações. Outras alternativas são mais flexíveis, porém seu custo computacional inviabiliza sua operação. Portanto, o desafio começa já na escolha da alternativa que melhor se ajusta ao seu problema.

Nesse aspecto, o algoritmo *Fuzzy Minimals* apresenta diversas vantagens. Com ele os objetos não pertencem exclusivamente a um único agrupamento, embora isso também seja possível atribuindo o objeto ao grupo com maior pertinência. Essa característica permite realizar uma transição entre os diferentes grupos, essencial em conjuntos de dados que não sejam bem definidos. Além disso, não é preciso estipular o número de agrupamentos contidos no conjunto de dados e os grupos não precisam ter tamanhos ou formatos específicos. Esses fatores, além de facilitar a execução do algoritmo para qualquer conjunto de dados, também reduzem seu tempo de execução em relação a outras alternativas.

Em razão dos crescentes volumes de dados processados, qualquer solução viável de agrupamento de dados passa, obrigatoriamente, pela sua paralelização em uma arquitetura alvo. Evidentemente arquiteturas distintas exigem implementações distintas, porém não basta mapear o código de um ambiente para outro. Para extrair todo o potencial de uma determinada plataforma é preciso conhecer esse ambiente e projetar o código de acordo as características de sua arquitetura.

Dentre as plataformas disponíveis identificamos a GPU como um equipamento de alto desempenho e baixo custo, com resultados expressivos em problemas que demandam um alto grau de paralelismo. Porém, alcançar esses níveis de desempenho não é uma tarefa simples. Para que isso ocorra, além do problema ser adequado às características da GPU, também é preciso conhecer seus níveis de memória e utilizá-los de forma equilibrada, além de compreender a organização das *threads* para manter os multiprocessadores ocupados e dominar suas bibliotecas para utilizar funções otimizadas da plataforma.

Diversos experimentos foram realizados neste trabalho. Foram utilizados desde conjuntos de dados pequenos, para demonstrar o funcionamento e a precisão do método, até conjuntos maiores com dados produzidos artificialmente ou extraídos a partir de informações reais, para avaliar seu desempenho. Mostramos que as soluções paralelas desenvolvidas alcançaram um alto desempenho, sem prejudicar a precisão dos resultados. Além disso, constata-se que o tempo de execução não é exclusivamente proporcional ao tamanho do conjunto de dados, mas também deve-se levar em consideração outros fatores, como sua densidade e o número de grupos.

É importante destacar o melhor desempenho das execuções em GPU em relação à solução em MPI, produzindo tempos de execução até 100 vezes mais rápidos que a execução sequencial. Nas execuções com oito partições os desempenhos das implementações paralelas se aproximaram ou foram semelhantes, pois a GPU utilizada possui apenas quatro SMs. Portanto, neste caso o aumento de partições usa mais recursos que o disponível na placa. Assim, constatamos que a GPU proporciona um alto desempenho com custos menores, tanto em relação à aquisição dos equipamentos quanto a sua operação (manutenção, consumo de energia, etc).

Finalmente, como principal contribuição deste projeto, desenvolveu-se uma solução inédita em GPU do algoritmo *Fuzzy Minimals*. Esta proposta é importante em um cenário com demandas crescentes por classificação de dados, que exigem um alto desempenho. Esta solução mostrou que é possível atingir desempenhos expressivos sem ter a disposição uma infraestrutura mais complexa e custosa. Além disso, a solução desenvolvida atingiu um desempenho significativamente melhor que o alcançado em (20), que obteve ganhos de até 10 vezes em relação ao sequencial nele avaliado.

6.1 Trabalhos Futuros

Como consequência deste trabalho é possível apontar algumas direções como trabalhos futuros:

- **Solução usando Intel Xeon Phi**
  Com o objetivo de competir com as GPUs no segmento de computação de alto desempenho a Intel lançou o coprocessador Intel Xeon Phi. Essa também é uma plataforma massivamente paralela, mas sua arquitetura e organização são diferentes das CPUs e GPUs. Existe um certo debate entre os fabricantes e também entre os desenvolvedores sobre qual é a melhor solução. Portanto, como não existe nenhuma avaliação de desempenho sobre o *Fuzzy Minimals* usando esse equipamento, essa seria uma pesquisa interessante.

- **Soluções híbridas**
  Neste trabalho foram avaliados os desempenhos das plataformas isoladamente, portanto, enquanto um equipamento realizava todo o processamento, o outro permaneceu ocioso. Para evitar que isso ocorra, seria possível implementar uma solução híbrida para explorar toda a infraestrutura disponível no ambiente. Isso seria feito, por exemplo, utilizando CPU e GPU, por meio de bibliotecas como OpenMP, MPI e CUDA. Seria necessário avaliar as circunstâncias que compensam a execução em cada plataforma, como distribuir o processamento para a CPU quando o problema exige mais recursos que a GPU dispõe.

- **Solução multi-GPU**
  No trabalho desenvolvido foi utilizado apenas uma GPU, mas também seria interessante avaliar o desempenho em múltiplas GPUs. Isso poderia ser feito, por exemplo, instalando mais de uma placa na mesma máquina e lançando um *kernel* para cada um desses dispositivos. Isso também poderia ser ampliado para máquinas distintas, nesse caso seria preciso utilizar a biblioteca MPI para a troca de mensagens entre esses equipamentos.

- **Conjuntos de dados**
  O principal objetivo neste trabalho foi desenvolver uma solução de alto desempenho para o problema de agrupamento de dados, sem estipular um contexto específico para o conjunto de dados analisado. Conforme verificou-se ao longo deste texto, existe uma infinidade de aplicações de agrupamento de dados, portanto, seria pertinente aplicar esta solução em outros contextos.

6.2 Publicações

Uma versão preliminar deste trabalho foi publicada na VII Escola Regional de Alto Desempenho de São Paulo, sob o título "Agrupamento de dados em GPU".

Os resultados aqui apresentados foram submetidos no *The Journal of Supercomputing*, sob o título "*Parallel fuzzy minimals on GPU*", estando ainda em avaliação no momento da escrita deste texto.

Referências Bibliográficas

1 CUDA C Programming Guide. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>. Acesso em: 08/06/2016.

2 JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: a review. *ACM computing surveys (CSUR)*, Acm, v. 31, n. 3, p. 264–323, 1999.

3 CISCO VNI Forecast and Methodology, 2015-2020. Disponível em: <http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html>. Acesso em: 02/02/2017.

4 ZHAO, L.; PAN, S. J.; YANG, Q. A unified framework of active transfer learning for cross-system recommendation. *Artificial Intelligence*, Elsevier, 2016.

5 MAULIK, U.; SARKAR, A. Gene microarray data analysis using parallel point symmetry-based clustering. *Bioinformatics: Computational Techniques and Engineering*, p. 293, 2010.

6 PAN, L.-m. et al. Vertical co-current two-phase flow regime identification using fuzzy c-means clustering algorithm and relieff attribute weighting technique. *International Journal of Heat and Mass Transfer*, Elsevier, v. 95, p. 393–404, 2016.

7 YENILMEZ, F.; GIRGINER, N. Comparison of indicators of womens labour between turkey and eu member states by employing multidimensional scaling analysis and clustering analysis. *Applied Economics*, Taylor & Francis, v. 48, n. 13, p. 1229–1239, 2016.

8 YU, X. et al. Profiling and relationship of water-soluble sugar and protein compositions in soybean seeds. *Food chemistry*, Elsevier, v. 196, p. 776–782, 2016.

9 BHATIA, V.; RANI, R. A parallel fuzzy clustering algorithm for large graphs using pregel. *Expert Systems with Applications*, Elsevier, v. 78, p. 135–144, 2017.

10 MACQUEEN, J. et al. Some methods for classification and analysis of multivariate observations. In: OAKLAND, CA, USA. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. [S.l.], 1967. v. 1, n. 14, p. 281–297.

11 RAO, S. T.; PRASAD, E.; VENKATESWARLU, N. A scalable k-means clustering algorithm on multi-core architecture. In: IEEE. *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*. [S.l.], 2009. p. 1–9.

12 LI, Y. et al. Speeding up k-means algorithm by gpus. In: IEEE. *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. [S.l.], 2010. p. 115–122.

13 JAROŠ, M. et al. Implementation of k-means segmentation algorithm on intel xeon phi and gpu: Application in medical imaging. *Advances in Engineering Software*, Elsevier, v. 103, p. 21–28, 2017.

14 BEZDEK, J. C. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981. ISBN 0306406713.

15 DUNN, J. C. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. Taylor & Francis, 1973.

16 FLORES-SINTAS, A.; CADENAS, J.; MARTIN, F. A local geometrical properties application to fuzzy clustering. *Fuzzy Sets and Systems*, Elsevier, v. 100, n. 1, p. 245–256, 1998.

17 FLORES-SINTAS, A.; CADENAS, J. M.; MARTIN, F. Detecting homogeneous groups in clustering using the euclidean distance. *Fuzzy Sets and Systems*, Elsevier, v. 120, n. 2, p. 213–225, 2001.

18 SOTO, J.; FLORES-SINTAS, A.; PALAREA-ALBALADEJO, J. Improving probabilities in a fuzzy clustering partition. *Fuzzy Sets and Systems*, Elsevier, v. 159, n. 4, p. 406–421, 2008.

19 HAN, J.; PEI, J.; KAMBER, M. *Data mining: concepts and techniques*. [S.l.]: Elsevier, 2011.

20 TIMÓN, I. et al. Parallel implementation of fuzzy minimals clustering algorithm. *Expert Systems with Applications*, Elsevier, v. 48, p. 35–41, 2016.

21 FRIEDRICHS, M. S. et al. Accelerating molecular dynamic simulation on graphics processing units. *Journal of computational chemistry*, Wiley Online Library, v. 30, n. 6, p. 864–872, 2009.

22 SANDERS, J.; KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0131387685, 9780131387683.

23 NVIDIA GeForce 256. Disponível em: <http://www.nvidia.com/page/geforce256.html>. Acesso em: 16/02/2016.

24 OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. In: WILEY ONLINE LIBRARY. *Computer graphics forum*. [S.l.], 2007. v. 26, n. 1, p. 80–113.

25 OWENS, J. D. et al. Gpu computing. *Proceedings of the IEEE*, IEEE, v. 96, n. 5, p. 879–899, 2008.

26 LANGDON, W. B.; BANZHAF, W. A simd interpreter for genetic programming on gpu graphics cards. In: *Genetic Programming*. [S.l.]: Springer, 2008. p. 73–85.

27 MICHALAKES, J.; VACHHARAJANI, M. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, World Scientific, v. 18, n. 04, p. 531–548, 2008.

28 MANAVSKI, S. A. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In: IEEE. *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*. [S.l.], 2007. p. 65–68.

29 CAMARGO, R. Y. D.; ROZANTE, L.; SONG, S. W. A multi-gpu algorithm for large-scale neuronal networks. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 23, n. 6, p. 556–572, 2011.

30 BUCK, I. et al. Brook for gpus: stream computing on graphics hardware. In: ACM. *ACM Transactions on Graphics (TOG)*. [S.l.], 2004. v. 23, n. 3, p. 777–786.

31 MCCOOL, M. et al. Shader algebra. In: ACM. *ACM Transactions on Graphics (TOG)*. [S.l.], 2004. v. 23, n. 3, p. 787–795.

32 TAYLOR, G. *Energy efficient circuit design and the future of power delivery*. Disponível em: <http://cseweb.ucsd.edu/classes/wi10/cse241a/slides/Energy.pdf>. Acesso em: 16/02/2016.

33 NVIDIA GeForce GTX TITAN X. Disponível em: <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications>. Acesso em: 09/05/2016.

34 JAIN, A. K. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, Elsevier, v. 31, n. 8, p. 651–666, 2010.

35 CLATWORTHY, J. et al. The use and reporting of cluster analysis in health psychology: A review. *British journal of health psychology*, Wiley Online Library, v. 10, n. 3, p. 329–358, 2005.

36 CRAIG, N.; ALDENDERFER, M.; MOYES, H. Multivariate visualization and analysis of photomapped artifact scatters. *Journal of Archaeological Science*, Elsevier, v. 33, n. 11, p. 1617–1627, 2006.

37 ZADEH, L. A. Fuzzy sets. *Information and control*, Elsevier, v. 8, n. 3, p. 338–353, 1965.

38 JOHNSON, S. C. Hierarchical clustering schemes. *Psychometrika*, Springer, v. 32, n. 3, p. 241–254, 1967.

39 JAIN, A. K.; DUBES, R. C. *Algorithms for clustering data*. [S.l.]: Prentice-Hall, Inc., 1988.

40 ZAHN, C. T. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on computers*, IEEE, v. 100, n. 1, p. 68–86, 1971.

41 DEMPSTER, A. P.; LAIRD, N. M.; RUBIN, D. B. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, JSTOR, p. 1–38, 1977.

42 MITCHELL, T. M. et al. *Machine learning. WCB*. [S.l.]: McGraw-Hill Education, 1997.

43 DRINEAS, P. et al. Clustering large graphs via the singular value decomposition. *Machine learning*, Springer, v. 56, n. 1-3, p. 9–33, 2004.

44 MAO, J.; JAIN, A. K. A self-organizing network for hyperellipsoidal clustering (hec). *Neural Networks, IEEE Transactions on*, IEEE, v. 7, n. 1, p. 16–29, 1996.

45 LI, X.; FANG, Z. Parallel clustering algorithms. *Parallel Computing*, Elsevier, v. 11, n. 3, p. 275–290, 1989.

46 DHILLON, I. S.; MODHA, D. S. A data-clustering algorithm on distributed memory multiprocessors. In: *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*. London, UK, UK: Springer-Verlag, 2000. p. 245–260. ISBN 3-540-67194-3. Disponível em: <http://dl.acm.org/citation.cfm?id=648035.744385>.

47 JUDD, D.; MCKINLEY, P. K.; JAIN, A. K. Large-scale parallel data clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, v. 20, n. 8, p. 871–876, 1998.

48 KANTABUTRA, S.; COUCH, A. L. Parallel k-means clustering algorithm on nows. *NECTEC Technical journal*, v. 1, n. 6, p. 243–247, 2000.

49 FORMAN, G.; ZHANG, B. Distributed data clustering can be efficient and exact. *ACM SIGKDD explorations newsletter*, ACM, v. 2, n. 2, p. 34–38, 2000.

50 ZHANG, Y.-P. et al. Parallel implementation of clarans using pvm. In: IEEE. *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*. [S.l.], 2004. v. 3, p. 1646–1649.

51 RAO, S. N. T.; PRASAD, E. V.; VENKATESWARLU, N. B. A scalable k-means clustering algorithm on multi-core architecture. In: *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proc. of Intl Conf. on*. [S.l.: s.n.], 2009. p. 1–9.

52 LI, Y. et al. Speeding up k-means algorithm by gpus. In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. [S.l.: s.n.], 2010. p. 115–122.

53 LV, Z. et al. Parallel k-means clustering of remote sensing images based on mapreduce. In: *Proc of the 2010 Intl Conf on Web Information Systems and Mining, WISM*. [S.l.]: Springer, 2010. p. 162–170.

54 KWOK, T. et al. Parallel fuzzy c-means clustering for large data sets. *Euro-Par 2002 parallel processing*, Springer, p. 27–58, 2002.

55 ALMAZROOIE, M.; VADIVELOO, M.; ABDULLAH, R. Gpu-based fuzzy c-means clustering algorithm for image segmentation. *arXiv preprint arXiv:1601.00072*, 2016.

56 TRIPATHY, B.; MITTAL, D.; HUDEDAGADDI, D. P. Hadoop with intuitionistic fuzzy c-means for clustering in big data. In: SPRINGER. *Proceedings of the International Congress on Information and Communication Technology*. [S.l.], 2016. p. 599–610.

57 RASMUSSEN, E. M.; WILLETT, P. Efficiency of hierarchic agglomerative clustering using the icl distributed array processor. *Journal of Documentation*, MCB UP Ltd, v. 45, n. 1, p. 1–24, 1989.

58 OLSON, C. F. Parallel algorithms for hierarchical clustering. *Parallel computing*, Elsevier, v. 21, n. 8, p. 1313–1325, 1995.

59 GARG, A. et al. Pbirch: a scalable parallel clustering algorithm for incremental data. In: IEEE. *Database Engineering and Applications Symposium, 2006. IDEAS'06. 10th International*. [S.l.], 2006. p. 315–316.

60 IBM – What Is Big Data: Bring Big Data to the Enterprise. Disponível em: <http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>. Acesso em: 03/02/2016.

61 KWOK, T. et al. Parallel fuzzy c-means clustering for large data sets. In: *Euro-Par 2002 Parallel Processing*. [S.l.]: Springer, 2002. p. 365–374.

62 DROSINOS, N.; KOZIRIS, N. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In: IEEE. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. [S.l.], 2004. p. 15.

63 BUSTAMAM, A.; BURRAGE, K.; HAMILTON, N. A. Fast parallel markov clustering in bioinformatics using massively parallel computing on gpu with cuda and ellpack-r sparse format. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, IEEE Computer Society Press, v. 9, n. 3, p. 679–692, 2012.

64 RUGGIERO, M. A. G.; LOPES, V. L. d. R. *Cálculo numérico: aspectos teóricos e computacionais*. [S.l.]: Makron Books do Brasil, 1997.

65 UNITED States Board on Geographic Names. Disponível em: <https://geonames.usgs.gov/domestic/index.html>. Acesso em: 19/01/2017.