UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Vinícius Henrique Almeida Praxedes

Paralelização do Algoritmo K-means em GPUs NVIDIA Utilizando a Biblioteca Numba

Uberlândia, Brasil

2024, Abril

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Vinícius Henrique Almeida Praxedes

Paralelização do Algoritmo K-means em GPUs NVIDIA Utilizando a Biblioteca Numba

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Daniel Duarte Abdala

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2024, Abril

Vinícius Henrique Almeida Praxedes

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 01 de novembro de 2016:

Daniel Duarte Abdala
Orientador

Júlia Tannús de Souza
Convidado 1

Anderson Rodrigues dos Santos
Convidado 2

Uberlândia, Brasil
2024, Abril

Resumo

O K-means é um algoritmo notório de agrupamento de dados que vem sendo usado extensivamente desde sua concepção no meio do século XX. Utilizando-o como exemplo de implementação, essa pesquisa visou implementar uma versão paralelizada, usando o processamento vetorial de uma GPU NVIDIA, para obter um ganho de performance significativo, sem afetar a precisão dos resultados. Buscou-se realizar essa implementação utilizando ferramentas de programação de mais alto-nível de abstração, especificamente a linguagem Python e as bibliotecas *Numba*, *Numpy* e *Pandas*, explorando a possibilidade de se atingir grandes aumentos em velocidade de execução sem abrir mão da facilidade de desenvolvimento proporcionada pelo Python, se comparada com a complexidade de realizar uma implementação equivalente em C++ utilizando a API CUDA diretamente. Foi constatada maior velocidade de execução em C++ se comparado ao Python em testes com implementações paralelas em GPU de soma de vetores unidimensionais com mais de 400 milhões de elementos, C++ sendo cerca de 22,63x mais rápido que Python. Não obstante, ao se testar implementações seriais e paralelas em Python do algoritmo k-means foram obtidos ganhos de performance dentro da magnitude esperada ao se comparar com resultados de outros trabalhos da literatura com objetivos semelhantes de otimização do k-means. Esses experimentos foram realizados utilizando cinco conjuntos de dados com o número de instâncias variando entre 3.810 e 13.932.632 e o número de *features* entre 3 a 50. Um aumento médio de peformance entre 9x a 42x foi atingido com sucesso sem prejudicar mensuravelmente a qualidade dos resultados, ao se comparar uma implementação serial em Python usando as bibliotecas *Pandas* e *Numpy* e uma paralela em Python usando, além dessas, a biblioteca *Numba* para rodar funções em uma GPU NVIDIA. Além disso foi observado que o ganho de performance tende a aumentar para conjuntos de dados maiores.

Palavras-chave: K-means, agrupamento de dados, GPGPU, CUDA, Numba, computação paralela.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CPU	*Central Processing Unit* — Unidade de Processamento Central. O principal e mais importante processador num computador. CPUs modernas possuem capacidade razoável de processamento paralelo, com dezenas de núcleos;
GPU	*Graphics Processing Unit* — Unidade de Processamento Gráfico. Um coprocessador especializado para operações vetoriais, comumente usado para operações da computação gráfica, como renderização de imagens. GPUs modernas possuem capacidade altíssima de processamento paralelo, com centenas a milhares de núcleos;
VRAM	*Video Random Access Memory* — Memória de Vídeo de Acesso Randômico. Um componente das GPUs que equivale à RAM das CPUs. Uma memória volátil de alta velocidade, usada para armazenamento de dados necessários às operações gráficas realizadas pela GPU.

Sumário

1 Introdução

A busca pelo menor tempo de execução é uma diretriz ubíqua na computação. Desde os primórdios da área busca-se algoritmos e procedimentos que, dados os mesmos parâmetros de entrada, executem a mesma tarefa na menor parcela de tempo possível. Outros recursos como espaço de memória utilizado, eficiência energética ou uso da rede em muitos cenários são mais importantes que o tempo de execução, mas ainda assim ela continua sendo um dos mais estudados parâmetros para categorização e avaliação de algoritmos e procedimentos na computação. De fato o tempo de execução — em ciclos, ou passos, de processamento — é a métrica utilizada na análise de uma das maiores incógnitas da computação, o problema P *versus* NP (COOK, 1971)(LEVIN, 1973), que diz respeito à complexidade de tempo de um algoritmo e da conferência da validez do resultado gerado por este, basicamente questionando se todos os problemas cujas soluções podem ser conferidas em tempo polinomial (problemas da classe NP) também podem ter soluções encontradas em tempo polinomial (problemas da classe P).

Um grande avanço na quantidade de poder de processamento dos computadores e, portanto, diminuição do tempo de execução de algoritmos, foi a criação dos processadores multinúcleo, permitindo a paralelização de processos. A habilidade de poder executar duas ou mais ações simultaneamente possibilitou muitos ganhos palpáveis na velocidade de execução de algoritmos e procedimentos, porém introduziu uma necessidade de mudança na forma de se pensar em resoluções de problemas computacionalmente: paralelizar um algoritmo serial (não-paralelo) não é uma tarefa trivial, e requer cuidados especiais com concorrência no acesso a recursos da máquina, interdependência de dados e cálculos, sincronização, entre outros dilemas.

Um dos componentes que mais utilizam da paralelização num computador moderno são as GPUs — unidades de processamento gráfico, ou placas de vídeo — que são basicamente processadores especializados em operações vetoriais, altamente paralelizadas, usualmente utilizadas para computação gráfica, e com sua própria memória dedicada, a VRAM. Enquanto processadores de uso geral, CPUs, costumam ter no máximo dezenas de núcleos para processamento paralelo, comparativamente, as GPUs possuem dezenas de milhares de núcleos para operações vetoriais.

No entanto, cada vez mais está sendo descoberto e aproveitado o potencial de uso das GPUs em atividades não apenas voltadas para renderização, interfaces e outras operações gráficas, mas sim para a computação de propósito geral. Diversos algoritmos modernos e antigos beneficiam-se imensamente do poder de alta paralelização proporcionado pelas GPUs, e com ferramentas como a biblioteca e linguagem CUDA criada pela

NVIDIA, está cada vez mais fácil implementar o uso de placas de vídeo em conjunto com processadores convencionais nos mais variados algoritmos.

Nem todo algoritmo pode ser paralelizado, no entanto. Existem procedimentos e algoritmos que são inerentemente seriais (também chamados de sequenciais), como o cálculo do n-ésimo número da sequência de *Fibonacci*, que requer que dois números prévios da sequência tenham sido calculados para obtermos o atual — salvo, é claro, alguma descoberta teórica matemática do comportamento da sequência que nos permitisse uma nova maneira de calcular o n-ésimo elemento sem essa necessidade.

É importante entender também que nenhum algoritmo é paralelizável por completo. Sempre existirão partes de algoritmos que necessariametne devem ser executadas serialmente para seu funcionamento correto. Há um limite teórico de ganho máximo que pode ser obtido ao se paralelizar um algoritmo qualquer. Esse limite é definido pela Lei de Amdahl (RODGERS, 1985): $\frac{1}{1-p}$, onde p é a razão entre tempo de execução gasto rodando código paralelizável e tempo de execução gasto no total.

A paralelização de uma classe de algoritmos em particular é o foco desta pesquisa: os algoritmos de agrupamento de dados, também chamados de clusterização de dados, ou de *clustering*. Tais algoritmos, de forma sucinta, agrupam objetos de maneira que objetos no mesmo grupo, ou *cluster*, sejam mais parecidos entre si, de acordo com alguma métrica, do que com objetos de outros grupos. A análise de clusters é essencial em diversas áreas da computação e estatística, como mineração de dados, aprendizado de máquina, compressão de dados, entre outras.

A hipótese principal deste trabalho é a de que algoritmos de clustering, em geral, são altamente paralelizáveis e apresentam um ganho considerável de desempenho (menor tempo de execução) quando implementados para utilizar o poder de paralelismo vetorial de placas de vídeo NVIDIA, através da plataforma CUDA (NVIDIA Corporation, 2018). Mais que isso, através de uma análise sistemática de estudos prévios e implementações de tais algoritmos em CUDA, visa-se generalizar o processo de paralelização destes. Isto é, identificar quais partes são necessariamente seriais, quais são paralelizáveis, e que sequência de passos gerais deve ser seguida para se conseguir paralelizar com sucesso um algoritmo de clustering qualquer e obter ganhos significativos de desempenho.

O foco principal dessa pesquisa é o notório algoritmo de agrupamento *k-means* (MACQUEEN, 1967) (LLOYD, 1982) (CUOMO et al., 2019). Implementações e estudos realizados sobre ele foram analisados, e uma implementação paralela em GPU teve seu desempenho comparado com a serial em CPU.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho tem como objetivo principal testar a validez de sua hipótese (discutida mais à fundo na seção 1.2) de que algoritmos de clusterização são intrinsecamente paralelizáveis, e que o ganho de velocidade ao serem paralelizados é altamente significativo.

Além disso, deseja-se compilar aqui um bom conhecimento introdutório de como paralelizar esses algoritmos em geral, analisando principalmente o algoritmo aqui estudado a fundo, o k-means, e usando este aprendizado para criar um passo-a-passo mais geral de como realizar tal modificação de código em um algoritmo de agrupamento qualquer.

1.1.2 Objetivos Específicos

Para atingir o objetivo geral, é necessário completar diversos objetivos menores, ou *milestones*, antes, criando um caminho de pesquisa que foi seguido — não necessariamente na ordem apresentada. São estes:

- Pesquisar extensamente a bibliografia da área, realizando assim um levantamento do estado da arte de algoritmos paralelos de agrupamento;
- Estudar implementações já realizadas do k-means, a fim de adquirir conhecimento de como a paralelização em CUDA deve ser realizada;
- Quantificar o ganho de desempenho das implementações paralelas, realizando diversos experimentos de *speed-up*, usando diversos datasets de tamanhos e dimensionalidades variadas;
- Comparar o código serial (sem paralelização) com o código paralelo do k-means, extraindo assim um conhecimento mais generalizado de como paralelizar um algoritmo de clusterização genérico.

1.2 Hipótese

A hipótese que esta pesquisa procura testar é a de que algoritmos de clusterização em geral são inerentemente vetoriais e, consequentemente, se beneficiariam significativamente de arquiteturas de processamento vetoriais, como uma unidade de processamento gráfico, ou GPU.

Um problema ser vetorial diz respeito ao escopo de tipos de dados relevantes ao problema. Grande parte dos problemas da computação são escalares, o que significa que

eles lidam com dados unitários, como por exemplo *integers* ou *floats*, um de cada vez. Já um problema vetorial lida com dados que são conjuntos uni ou multidimensionais, chamados vetores, que são formados por vários itens unitários de dados agrupados.

Um algoritmo que tente resolver um problema vetorial terá desempenho maior quando executado num processador vetorial, isto é, um processador que possui um conjunto de instruções capaz de manipular vetores. Apesar de um algoritmo de um problema vetorial ainda puder ser implementado e executado com sucesso num processador escalar, o desempenho será menor pois os dados vetoriais do problema terão que ter seus elementos processados um a um pelo processador, já que ele não trabalha com vetores propriamente ditos em seu conjunto de instruções.

Grande parte do ganho de desempenho supracitado vem do paralelismo proporcionado pelos processadores vetoriais, como GPUs, ao manipular conjuntos maiores de dados de uma só vez, e em vários núcleos simultaneamente. A natureza vetorial da GPU permite economizar traduções de endereço de memória e operações de obtenção (*fetch*) e decodificação (*decode*) de instruções, se comparado com o processamento escalar de uma CPU, pelo fato de se necessitar, na GPU, um número muito menor de instruções e endereços de memória quando os dados estão agrupados em vetores, que podem ser manipulados e usados em operações como se fossem, cada um, apenas um item de dados.

Este trabalho, então, visa demonstrar que algoritmos de agrupamento de dados, em geral, são intrinsecamente vetoriais. Isto é, qualquer algoritmo de clustering concebível será de natureza vetorial, pois estes analisam dados e tentam agrupá-los de acordo com algum grau de semelhança entre eles, análise esta que pode ser feita usando conjuntos dos itens de dados (vetores), ao invés de individualmente, mesmo que o *dataset* inicial possua apenas dados de natureza escalar. Logo, qualquer algoritmo de agrupamento teria uma parcela do seu código que seria paralelizável e, assim, ganhariam desempenho significativo com uma execução numa GPU. Mais que isso, a parcela de tempo de execução do algoritmo gasta rodando código paralelo cresceria de acordo com o tamanho do conjunto de dados sendo analisado, garantindo ganhos ainda maiores.

1.3 Justificativa

A pesquisa feita aqui pode ser de grande utilidade para a área da computação e ciência de dados, além de impulsionar a implementação de mais algoritmos paralelos de agrupamento de dados.

Com a compilação de conhecimento realizada aqui a intenção é facilitar pesquisas posteriores na área de paralelização de algoritmos de agrupamento e motivar com os experimentos de ganho de desempenho novas implementações paralelas de outros algoritmos desta classe, ilustrando o quão importante é o uso de processadores vetoriais como GPUs

para tornar o uso de algumas destas abordagens de agrupamento realmente práticas.

Além disso, a apresentação nesse estudo de um procedimento mais geral para paralelizar qualquer algoritmo de agrupamento será de grande utilidade para qualquer desenvolvedor ou pesquisador que desejar implementar uma versão acelerada em GPU de um algoritmo do tipo, mesmo este sendo totalmente novo. No mínimo, a pesquisa servirá de ponto de partida para o entendimento e aprendizado de como realizar tal modificação no código do algoritmo, e renderá uma implementação real que serve de base para estudos e otimizações, até se obter eventualmente uma implementação digna para uso prático.

1.4 Estrutura da Monografia

O restante da monografia está organizada da seguinte maneira: o **Capítulo 2** apresenta a fundamentação teórica do trabalho, abordando os tópicos de agrupamento de dados, programação vetorial, a API CUDA da NVIDIA e a biblioteca Python *Numba*. Nesse capítulo é explorado e resumido o conhecimento teórico necessário para o entendimento das implementações e experimentos realizados nessa pesquisa.

O **Capítulo 3** contém um panorama da bibliografia da área de agrupamento de dados, principalmente no que diz respeito a implementações paralelas de algoritmos desse tipo. A função do capítulo é apresentar o que há de melhor e mais avançado dentre as implementações já realizadas por diversos pesquisadores e desenvolvedores, para que seja possível criar uma expectativa realista dos níveis de ganho de desempenho que pode-se esperar da implementação paralela do k-means realizada nesse trabalho.

No **Capítulo 4**, é apresentada toda a metodologia utilizada para realização da pesquisa, desde a lógica e procedimento de identificação de partes de código com alto potencial para paralelização, até as minúscias das implementações do k-means versão serial e versão paralela, incluindo os códigos em Python utilizados.

Já no **Capítulo 5**, são apresentados os resultados e metodologias dos experimentos realizados, além dos conjuntos de dados utilizados para os testes e quaisquer tratamentos realizados nesses datasets para processamento.

Finalmente, as conclusões finais são apresentadas no **Capítulo 6**, além de incluir considerações relevantes para trabalhos futuros.

2 Fundamentação Teórica

Para compreender a pesquisa científica aqui realizada, é necessário primeiro entender o que são algoritmos de agrupamento, tanto de maneira geral quanto específica, explorando os fundamentos e funcionamento principalmente do k-means, o algoritmo aqui estudado. Veremos que a complexidade de tempo desses algoritmos de agrupamento tendem a ser inconvenientemente altas ($O(n \cdot \log n)$, $O(n^2)$ ou até $O(n^3)$ sendo complexidades comuns) e por isso qualquer ganho de velocidade significativo obtido será de imensa relevância para a usabilidade prática do algoritmo.

Também é imprescindível explorar o funcionamento dos processadores vetoriais — sendo as *Unidades de Processamento Gráfico* (GPUs) o principal exemplo destes e exatamente no qual essa pesquisa irá focar — e entender por que usá-los para paralelizar algoritmos de agrupamento proporcionará, em tese, um ganho de velocidade expressivo na execução destes, abrandando o peso de suas complexidades de tempo. Além de tudo isso, será apresentada brevemente a arquitetura utilizada para paralelizar os algoritmos estudados: a plataforma e modelo de programação CUDA, da NVIDIA, que permitirá extrair o poder de paralelização das placas de vídeo NVIDIA além de bibliotecas como a *Numba*, que permite a programação vetorial facilitada na linguagem Python.

2.1 Agrupamento de Dados

O agrupamento de dados, também chamado de clusterização de dados (data clustering, em inglês), é a tarefa de agrupar um conjunto de elementos de modo que cada elemento de um grupo se "pareça" mais com outros elementos do grupo (*cluster*) que pertence do que com elementos dos grupos que não pertencem, dado algum significado bem definido de semelhança entre os dados. É um processo muito comum e virtualmente imprescindível nas áreas de mineração de dados, análise estatística, análise de imagem, aprendizado de máquina, reconhecimento de padrões, e muitas outras.

O significado de um cluster não pode ser bem definido e vai depender do conjunto de dados a ser analisado e a forma que os resultados obtidos serão utilizados — de fato, esse é o principal motivo pelo qual tantos algoritmos diferentes de agrupamento existem (ESTIVILL-CASTRO, 2002). O fator comum na maioria das definições propostas é que um cluster é um conjunto de *datapoints* — pontos, ou objetos de dados ou até mesmo instâncias. Esses objetos são representados num espaço geométrico com o número de dimensões iguais ao número de variáveis necessárias para descrever cada datapoint, e os algoritmos de agrupamento tentam criar grupos nesse espaço que agrupem os objetos de uma maneira significativa, ou útil, para o estudo sendo feito e a definição de "grupo"

sendo utilizada.

Diversos modelos de grupo podem ser usados para definir o que é um grupo. Dentre os modelos mais comuns, destacam-se: **a) modelos de centroide**, onde cada grupo possui um centro e cada datapoint pertencerá ao grupo com centro mais próximo dele, dada uma definição de distância no espaço geométrico dos dados; **b) modelos de densidade**, que definem grupos como regiões densas e conexas no espaço, contrastando com regiões menos densas que separam os grupos; **c) modelos de conectividade**, que constroem grupos a partir de conexões de datapoints definidas por um limiar de distância; **d) modelos de distribuição**, que utilizam de distribuições estatísticas, como a distribuição normal ou exponencial, para modelar o agrupamento dos datapoints; entre dezenas de outros modelos. Entender o modelo de grupo utilizado é essencial para compreender um algoritmo de agrupamento e as diferenças entre a multitude destes.

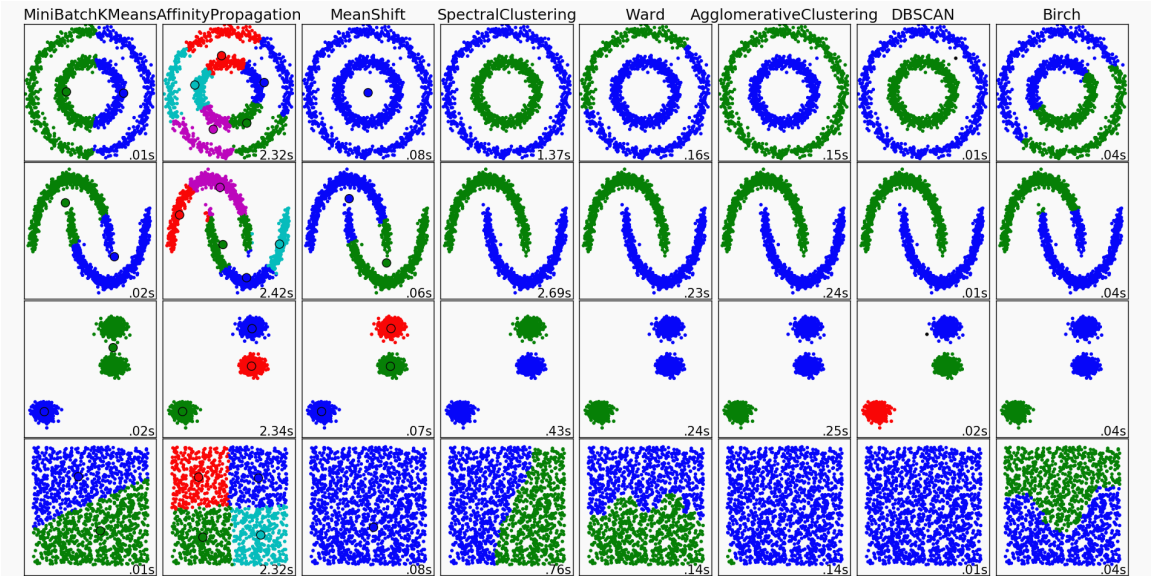
O resultado, ou saída, de um algoritmo de agrupamento é comumente um rotulamento dos datapoints passados na entrada, o que indicará a divisão em grupos feita por ele. Classificações podem ser feitas quanto à natureza do agrupamento obtido pelos algoritmos: ***hard clustering***, onde cada objeto pertence ou não a um grupo; ***fuzzy clustering***, onde cada objeto pertence uma certa porcentagem a cada grupo, o que pode representar, por exemplo, a chance do objeto pertencer àquele grupo, ou até o grau de semelhança do datapoint comparado aos outros datapoints de cada grupo (YANG, 1993). E subclassificações ainda mais granulares podem ser definidas, como: **clusterização de particionamento estrito**, onde cada objeto pertence a exatamente um cluster; **clusterização de particionamento estrito com *outliers***, onde objetos pertencem a exatamente um cluster, ou nenhum cluster, assim sendo considerados *outliers*, entre outras classificações.

Os algoritmos de agrupamento são essenciais na análise de dados, permitindo a identificação de padrões e estruturas em conjuntos de dados não rotulados. Eles pertencem à categoria de **algoritmos de aprendizado não supervisionado** e são amplamente utilizados em diversas áreas da computação, matemática, economia, estatística, dentre outras. A paralelização desses algoritmos visa melhorar a eficiência computacional, permitindo o processamento mais rápido de grandes volumes de dados.

Existem diversos algoritmos de agrupamento, e todos possuem vantagens e desvantagens quanto à eficiência de seus resultados para diferentes tipos de conjuntos de dados, além de diferentes complexidades computacionais e tempos de execução.

Essa diversidade é exemplificada na Figura 1 (SciKit-Learn Developers, 2024b), retirada da documentação da biblioteca **SciKit Learn** para a linguagem Python, que oferece diversas ferramentas e algoritmos essenciais para área de ciência de dados e aprendizado de máquina. Na imagem, são exibidas as saídas de oito algoritmos de agrupamento de dados, ao processarem quatro conjuntos de dados compostos por objetos com duas *fe-*

Figura 1 – Saída de algoritmos de agrupamento de dados ao processarem conjuntos de dados 2D variados



atures (variáveis). Essas saídas são representadas graficamente distribuindo os pontos de dados em dois eixos, cada um representando uma variável que descreve o ponto. A cor dos pontos representa a qual grupo cada objeto de dado pertence. Há também o tempo de execução, em segundos, no canto inferior direito de cada saída.

O algoritmo *Mini-Batch K-Means*, por exemplo, não foi capaz de dividir adequadamente o primeiro conjunto de dados, com dois grupos no formato de círculos circunscritos, enquanto o *DBSCAN* e *Agglomerative Clustering* conseguiram diferenciar os dois grupos sem problemas. O mesmo cenário ocorreu com o segundo conjunto de dados, com datapoints distribuídos em segmentos de elipses. É possível notar também que os algoritmos *Affinity Propagation* e *Spectral Clustering* possuem tempos de execução significativamente maiores que os outros seis algoritmos.

2.2 K-means

O algoritmo **K-means**, também chamado de **K-médias** em português, é notório por sua aplicação em tarefas de agrupamento de dados, e possui raízes históricas que remontam à metade do século XX (BOCK, 2007). A ideia fundamental por trás do k-means, conhecida como quantização vetorial, foi introduzida inicialmente em um contexto de processamento de sinal. Esse algoritmo visa particionar, iterativamente, N observações em K grupos, de maneira que cada observação pertença ao agrupamento com o "centro" mais próximo, resultando na minimização das variações dentro dos agrupamentos (variação *intra-grupo*) e na maximização das variações entre observações em grupos diferenets (variação *inter-grupo*).

A aplicação original desse conceito foi documentada por Hugo Steinhaus em 1957 (STEINHAUS, 1957), mas o algoritmo padrão como é conhecido foi proposto por Stuart Lloyd enquanto trabalhava na Bell Labs em 1957 e publicado formalmente somente em 1982 (LLOYD, 1982), solidificando a base para o moderno entendimento e aplicação do k-means em análise multivariada e outras áreas de processamento de dados.

O termo "k-means" em si só foi ser cunhado em 1967 por James MacQueen (MACQUEEN, 1967), sendo conhecido também como **Algoritmo de Lloyd**, ou até de **Lloyd–Forgy**, pois sua ideia chegou também a ser publicada antes, em 1965, por Edward W. Forgy (FORGY, 1965), nomes que ainda são usados atualmente, embora não tão populares quanto o termo k-means.

O desenvolvimento do algoritmo k-means não parou com essas contribuições iniciais. Ao longo dos anos, várias melhorias e variantes foram propostas, como em 2007 com o **K-means++** (ARTHUR; VASSILVITSKII, 2007), que aprimora significativamente a eficiência e a precisão do algoritmo ao escolher centros iniciais para os agrupamentos de maneira mais cuidadosa. Além disso, a adaptação do k-means para lidar com fluxos de dados e a introdução de variações como **K-Medoids**, **K-Medians**, **K-Center** e **K-Box** ampliaram sua aplicabilidade e desempenho em uma gama ainda maior de contextos de dados.

A importância histórica e a evolução contínua do k-means mostram seu valor duradouro na análise de dados e machine learning. Desde sua concepção, o algoritmo não só inspirou avanços na teoria matemática, mas também provou ser uma ferramenta indispensável na era moderna da computação, especialmente em aplicações de agrupamento de dados em larga escala.

O k-means foi o escolhido como foco desse trabalho por ser, até na modernidade, um algoritmo extremamente utilizado na prática, mesmo com suas limitações, além de ter uma implementação inicialmente simples e de possuir uma estrutura amigável para a paralelização.

A implementação de versões paralelas de outros algoritmos de agrupamento de dados envolveria um processo de identificação de partes de código paralelizável diferente para cada algoritmo, mesmo que semelhante ao realizado aqui para o k-means — processo apresentado no capítulo 4.1. Algumas dificuldades peculiares também poderiam se apresentar para cada algoritmo a ser paralelizado, como operações de redução, onde a entrada é vetorial e a saída é escalar, que são mais difíceis de se paralelizar do que operações onde ambas a entrada e saída são vetoriais. Um exemplo de operação como essa existe no próprio algoritmo k-means, e é comentada mais a fundo no capítulo 4.2.

2.2.1 Algoritmo e Limitações

Um dos motivos da fama do k-means é a simplicidade de se entender sua lógica e funcionamento. Diferentemente de muitos outros algoritmos de agrupamento de dados, ele pode ser descrito com um pseudocódigo extremamente sucinto, como o Algoritmo 1 exibido abaixo, que descreve a implementação, como originalmente proposta, do processo do k-means.

Algoritmo 1 – Implementação padrão em pseudocódigo do k-means

```
Entrada
    P = {P1, P2, ..., Pn}, um conjunto de N objetos de dados (pontos em um
        espaço D–dimensional);
    K, o número de agrupamentos desejado;
    I_max, o número de iterações máximas do algoritmo
Saída
    Um conjunto de K agrupamentos, onde cada um dos N objetos em P está
        associado a exatamente um conjunto.

Passos
1. Escolha arbitrariamente K pontos em P para servirem de centroides
   iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
    2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais
        próximo, calculado pela distância euclidiana;
    2.2 Calcule novos centroides para cada agrupamento através da média
        das coordenadas de todos os pontos do grupo.

Critérios de convergência
    Não há mudança entre os centroides da iteração atual e o da anterior;
    OU...
    O número de iterações realizadas ultrapassa um máximo I_max.
```

Não há garantia de que o processo convirja para um máximo global, apenas para um máximo local que, dependendo dos centróides iniciais escolhidos, pode ser arbitrariamente pior que o máximo global.

A complexidade de tempo do k-means é de $O(NKDI)$ (XU; WUNSCH, 2005), onde I é o número de iterações necessárias para convergir em um resultado, estando na faixa de $[1, I_{max}]$ e dependendo dos centroides iniciais escolhidos.

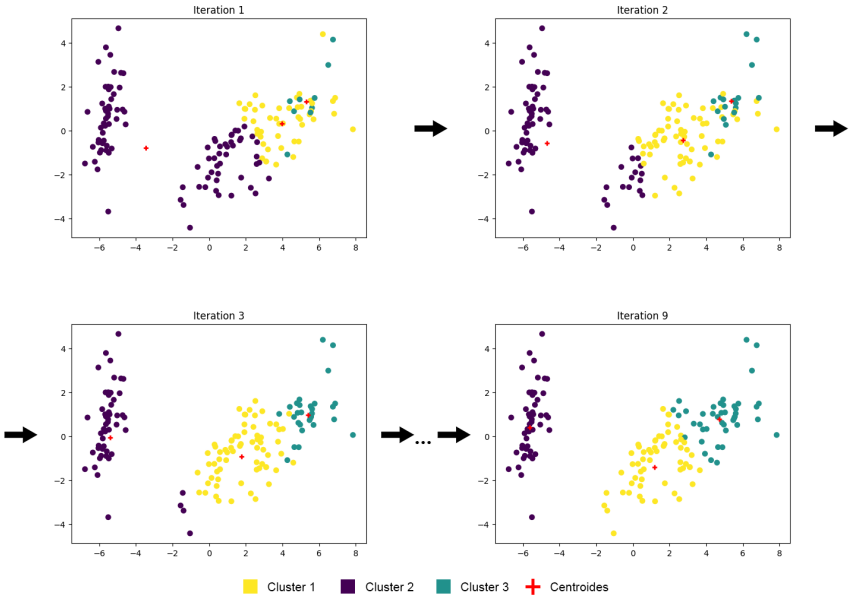
Na Figura 2 mais abaixo há um exemplo de uma execução do k-means, utilizando o dataset Iris, explicado em mais detalhes no Capítulo 5.3.

Tanto as primeiras três quanto a última iteração são exibidas graficamente, com cada um dos 150 datapoints ($N = 150$) tendo sua cor definida por qual dos três grupos ($K = 3$) ele pertence. É importante notar que, como há quatro variáveis em cada instância

($D = 4$) nesse dataset, os eixos X e Y exibidos nos gráfico representam, cada um, duas dimensões, ao invés de apenas uma. O processo de redução de dimensionalidade para exibição gráfica é importante para possibilitar algum tipo de visualização de espaços de mais de três dimensões. Esse processo foi feito apenas na renderização, e não na entrada de dados.

É possível perceber o "movimento" dos centroides ao longo das iterações, representados aqui por uma pequena cruz vermelha, até convergirem para o máximo local, atingido nessa execução em nove iterações ($I = 9$). Nota-se também que, para esse dataset, o algoritmo consegue realizar uma boa separação de um dos grupos em relação ao resto (mais à esquerda no gráfico), mas não os outros dois grupos (mais à direita), que não possuem uma separação tão grande no espaço das variáveis. Esse é um resultado esperado nesse dataset, pois duas das classificações das instâncias não são linearmente diferenciáveis a partir das informações disponíveis nos dados.

Figura 2 – Demonstração do k-means rodando no dataset Iris



É importante entender também as limitações do algoritmo k-means. Como explicitado anteriormente, não há garantias de que o processo encontre o melhor agrupamento possível para o conjunto de dados, apenas o melhor possível a partir dos centroides iniciais.

O algoritmo também é sensível a *outliers*, isto é, pontos de dados que se diferem muito da distribuição geral dos dados. A presença destes pode causar grandes deformações dos clusters — basicamente "puxando" os centroides para mais perto dos *outliers*.

Um dos argumentos necessários na chamada do algoritmo é considerado também uma limitação: o parâmetro K. Como é necessário que o k-means seja informado da quantidade de agrupamentos a serem feitos, isso implica que já se saiba algo a respeito da natureza ou geometria do conjunto de dados a ser agrupado, o que nem sempre é possível,

ainda mais para datasets de alta dimensionalidade, onde uma visualização útil para se interpretar os dados se torna extremamente difícil.

Mesmo com todas essas limitações, o k-means continua um algoritmo ubíquo em diversas áreas da computação, como discutido a seguir, e segue sendo amplamente estudado e aprimorado na literatura.

2.2.2 Aplicações Reais

O k-means, reconhecido por sua simplicidade e eficácia, é empregado em uma variedade de aplicações práticas em diversos campos. Sua capacidade de efetivamente segmentar dados em grupos com características similares o torna uma ferramenta versátil para diversos usos, desde análises de mercado até sistemas de recomendação.

Uma das aplicações mais comuns do k-means é a segmentação de clientes. Nesse contexto, empresas de diversos setores, como bancos, telecomunicações, e-commerce e publicidade, utilizam o k-means para agrupar clientes com base em comportamentos de compra, preferências ou demografia, permitindo o desenvolvimento de estratégias de marketing direcionadas e personalizadas. Esta aplicação não apenas ajuda as empresas a entender melhor seus clientes, mas também a otimizar suas ofertas e serviços de acordo com as necessidades de cada segmento.

Um exemplo concreto dessa aplicação é a realizada em um estudo de caso de 2018 (KARA; FIRAT, 2018), que utilizou o k-means para avaliar não clientes, mas sim realizar a análise de risco de fornecedores comerciais.

Neste estudo, o método k-means foi aplicado para **avaliar e segmentar fornecedores** de uma empresa de maquinário pesado com base em seus perfis de risco. A metodologia incluiu a utilização do Best-Worst Method (BWM) para determinar os pesos de 17 critérios de risco, seguido pela aplicação de Análise de Fatores para redução de dimensionalidade dos dados de risco, resultando em quatro fatores principais. Utilizando esses fatores, o k-means foi empregado para dividir os fornecedores em três clusters, representando diferentes níveis de exposição ao risco, o que permitiu uma análise detalhada das características de risco de cada grupo.

Os resultados do estudo demonstraram que a abordagem baseada em k-means é eficaz para a categorização dos fornecedores em grupos homogêneos quanto aos riscos enfrentados, fornecendo insights significativos para a gestão e o avaliação de fornecedores. Este agrupamento facilitou decisões estratégicas na gestão de riscos, destacando a importância e utilidade de métodos analíticos avançados em ambientes empresariais competitivos. O estudo sugere que a aplicação de técnicas de agrupamento de dados, como o k-means, pode melhorar significativamente as práticas de compra e a gestão de riscos em setores industriais.

Outra aplicação notável do k-means é na organização e classificação de documentos. Por exemplo, em grandes conjuntos de dados de texto, como coleções de artigos ou relatórios de pesquisa, o k-means pode agrupar documentos com temas ou conteúdos semelhantes, facilitando a gestão e a recuperação de informações. Este uso é particularmente valioso em campos como pesquisa acadêmica, onde a capacidade de navegar rapidamente por vastas quantidades de literatura é crucial.

Um exemplo específico dessa aplicação é explorado na documentação da biblioteca Python **Scikit Learn** (SciKit-Learn Developers, 2024a). Nesse exemplo de implementação, é explorado o **agrupamento de documentos de texto** utilizando o método k-means com a biblioteca scikit-learn, focando em agrupar textos por tópicos de maneira eficaz. Ele emprega duas variantes do algoritmo k-means implementadas na biblioteca, o *KMeans* e *MiniBatchKMeans*, e utiliza a Análise Semântica Latente (LSA) para a redução de dimensionalidade e a identificação de padrões "escondidos" nos dados. Além disso, são utilizados dois métodos de vetorização de texto: *TfidfVectorizer* e *HashingVectorizer*, que ajudam na transformação do texto em uma representação numérica que pode ser usada pelo algoritmo de agrupamento.

Para a análise, o dataset *20 newsgroups* é utilizado, selecionando quatro tópicos para manter o custo computacional acessível e melhorar a clareza do problema de agrupamento, removendo metadados desnecessários como cabeçalhos e citações. A eficácia do agrupamento é avaliada através de várias métricas, incluindo homogeneidade, completude, medida-V, índice Rand ajustado e coeficiente de silhueta, destacando que a aplicação da técnica de LSA antes do agrupamento melhora significativamente tanto a estabilidade quanto a qualidade dos clusters. Esses resultados ilustram o potencial dessas técnicas para categorizar e entender grandes conjuntos de dados de texto, proporcionando insights valiosos sobre a organização dos tópicos dentro dos documentos.

O k-means também tem aplicações significativas no processamento de imagens, como na segmentação de imagens, onde o algoritmo agrupa pixels semelhantes para identificar e separar diferentes objetos ou regiões dentro de uma imagem. Esta técnica é amplamente utilizada em tarefas como reconhecimento de padrões, visão computacional e até mesmo na melhoria de algoritmos de compressão de imagem, tornando-a uma ferramenta fundamental na era digital.

2.3 Programação Vetorial

A programação vetorial é uma abordagem computacional que visa aproveitar ao máximo o potencial de processamento de unidades de processamento que suportam operações vetoriais. Essa abordagem permite realizar operações em conjuntos de dados (vetores) de uma só vez, em vez de processar individualmente cada elemento do vetor. Isso resulta

em um aumento significativo no desempenho computacional, especialmente em algoritmos que manipulam grandes volumes de dados.

2.3.1 História

A história da programação vetorial é intrinsecamente ligada ao avanço da computação e à necessidade de lidar com conjuntos massivos de dados de maneira eficiente. O conceito de processamento vetorial remonta ao desenvolvimento dos primeiros supercomputadores e ao surgimento das primeiras GPUs, com destaque para a evolução da arquitetura das GPUs NVIDIA.

A ideia por trás da programação vetorial é aproveitar ao máximo o poder de processamento dos processadores, executando uma mesma instrução em múltiplos conjuntos de dados simultaneamente. Isso é especialmente útil em tarefas que envolvem operações repetitivas sobre grandes vetores ou matrizes de dados, como aquelas encontradas em algoritmos de processamento de imagem, simulações físicas e, mais recentemente, em algoritmos de agrupamento de dados e treinamento de DNNs — *Deep Neural Networks*.

Ao longo do tempo, a programação vetorial evoluiu significativamente, impulsionada pelo avanço das arquiteturas de processadores e pela demanda por computação paralela cada vez mais poderosa. Um dos marcos importantes nessa evolução foi a introdução do tipo de processamento SIMD (Single Instruction, Multiple Data) nos supercomputadores na década de 1970. Essa abordagem permitiu que uma única instrução fosse executada em múltiplos dados simultaneamente, proporcionando um aumento significativo no desempenho computacional para uma ampla gama de aplicações.

Figura 3 – Cubo antes de rotações

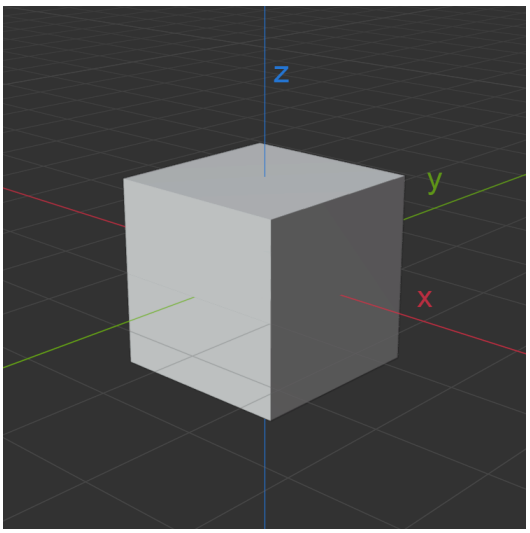
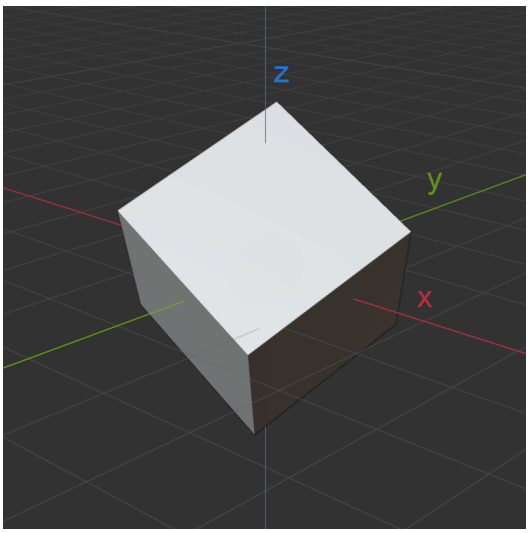


Figura 4 – Cubo após rotações



Com o surgimento das GPUs, desenvolvidas com o propósito inicial de acelerar a renderização gráfica em jogos e aplicações de multimídia, surgiu uma nova oportunidade para a programação vetorial. As GPUs são compostas por centenas ou até milhares

de núcleos de processamento, o que as torna altamente paralelizáveis e adequadas para executar operações vetoriais em larga escala. Isso possibilitou a utilização das GPUs não apenas para gráficos, mas também para tarefas de computação de propósito geral (área chamada também de *GPGPU*), incluindo processamento de grandes conjuntos de dados e algoritmos de aprendizado de máquina.

Um exemplo prático de operações gráficas comumente realizadas por GPUs é exemplificado nas figuras 3 e 4, que exibem, respectivamente, um cubo no espaço tridimensional antes e depois de sofrer duas rotações — uma de 60 graus no sentido horário no eixo x e outra de 30 graus no sentido anti-horário no eixo y, seguindo a convenção da *Regra de Fleming*, popularmente conhecida como "*regra da mão direita*".

Essa operação de rotação pode parecer simples visualmente, mas internamente requer duas multiplicações de matrizes com valores são obtidos através das operações trigonométricas de seno e cosseno. Essas multiplicações de matrizes, exibidas nas equações 2.1 e 2.2, precisam ser realizadas para cada vértice de todos os polígonos visíveis na cena a ser renderizada pela placa de vídeo.

$$P * R_x(-60°) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-60) & -\sin(-60) \\ 0 & \sin(-60) & \cos(-60) \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.866 \\ 0 & -0.866 & 0.5 \end{bmatrix} \tag{2.1}$$

Equação 2.1: Operação para rotacionar pontos 60 graus no sentido horário no eixo x

$$P * R_y(30°) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} \cos(30) & 0 & \sin(30) \\ 0 & 1 & 0 \\ -\sin(30) & 0 & \cos(30) \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot \begin{bmatrix} 0.866 & 0 & 0.5 \\ 0 & 1 & 0 \\ -0.5 & 0 & 0.866 \end{bmatrix} \tag{2.2}$$

Equação 2.2: Operação para rotacionar pontos 30 graus no sentido anti-horário no eixo y

Em uma cena simples, como a da figura 4, haveria apenas oito vértices para serem recalculados, mas é evidente como o número de vértices e operações a serem realizadas aumenta drasticamente em uma cena como a de um jogo de computador moderno, com dezenas a centenas de milhares de faces triangulares formando objetos e personagens na tela. Considerando que em jogos é comum serem exigidas taxas de atualização da tela entre 30 a 60 vezes por segundo para garantir a responsividade dos controles e a imersão do jogador, o recálculo da miríade de vértices que formam cada imagem deve ser realizado muito rapidamente, na faixa de 33 à 16 milissegundos por atualização (*frame*). Como quase sempre existem diversos outros efeitos gráficos a serem aplicados à cada imagem, além da rotação de vértices, esse tempo de execução deve ser mantido ainda mais baixo, para acomodar outras etapas do *pipeline* de vídeo.

Em suma, a programação vetorial desempenha um papel fundamental no avanço da computação paralela e no desenvolvimento de algoritmos eficientes para lidar com conjuntos massivos de dados. Com a contínua evolução da arquitetura de processadores e o aumento da demanda por computação paralela, é esperado que a programação vetorial continue a desempenhar um papel crucial no desenvolvimento de soluções computacionais rápidas e escaláveis para uma variedade de aplicações, como processamento de sinais, computação gráfica, simulações físicas, aprendizado de máquina e muito mais. Ela permite acelerar algoritmos complexos, reduzindo o tempo de execução e aumentando a eficiência computacional.

2.3.2 Processadores Vetoriais

É importante entender que nem todo processador oferece suporte para operações vetoriais, ou as oferecem com níveis de paralelismo inferiores a outros tipos de processadores mais especializados. Essas operações são essenciais para o desenvolvimento de algoritmos eficientes em uma variedade de aplicações computacionais. Examina-se aqui a arquitetura e as capacidades de diversos tipos de processadores, destacando sua importância na aceleração de operações paralelas e no aumento da eficiência computacional.

Os **processadores** com suporte de processamento paralelo **SIMD** desempenham um papel crucial na execução de operações vetoriais, permitindo a aplicação de uma única instrução em múltiplos conjuntos de dados simultaneamente. Esse tipo de processamento paralelo foi o foco de extensões como as SSE (*Streaming SIMD Extensions*), que permitiram um grande aumento de performance na execução de aplicações como processamento de sinal digital e processamento gráfico. Essas extensões são encontradas na gigantesca maioria das CPUs x86 atuais, a família de arquiteturas de processadores mais usada até hoje em computadores pessoais.

Os **processadores VLIW** (*Very Long Instruction Word*) são definidos pela sua capacidade de executar múltiplas operações em paralelo por meio de instruções muito longas. Destaca-se sua presença em sistemas embarcados e a eficiência proporcionada pela execução simultânea de operações vetoriais, especialmente em aplicações de processamento de sinal e comunicações digitais. Arquiteturas deste tipo já foram amplamente utilizadas em GPUs, porém houve uma mudança para arquiteturas RISC (*Reduced Instruction Set Computer*), mais simples, para acelerar também a execução de tarefas não-gráficas.

As **Unidades de Processamento Gráfico** (*Graphical Processing Units*), ou GPUs, são processadores especializados em operações úteis para aplicações gráficas, como cálculos geométricos para renderização 3D, mapeamento de texturas, rotação ou translação de vértices, aplicação de *shaders*, aceleração de decodificação de vídeo, entre muitas outras. Tais operações na grande maioria dos casos envolvem vetores ou matrizes sendo

manipuladas, com cálculos sendo aplicados a todos seus elementos (um exemplo concreto de operação gráfica realizada pela GPU é exibido no Capítulo 2.3.1). Logo, para lidar eficientemente com essas operações, esses processadores possuem uma unidade de memória dedicada (VRAM, *Video RAM*) e empregam milhares de núcleos para processamento paralelo.

As GPUs, sendo processadores intrinsecamente vetoriais, são o foco deste trabalho. Com o advento de ferramentas como a arquitetura CUDA para GPUs produzidas pela NVIDIA, se tornou cada vez mais fácil utilizar esses processadores para aplicações gerais, não apenas gráficas, como é o caso estudado aqui, de se utilizar tais processadores para acelerar algoritmos de agrupamento de dados.

Há também os **processadores DSP** (*Digital Signal Processors*), caracterizados pela sua eficiência na execução de operações vetoriais em tempo real, com foco em aplicações de processamento de sinais digitais. Possuem alta capacidade de lidar com operações complexas de forma rápida e precisa, contribuindo para o desenvolvimento de sistemas de comunicação e multimídia.

Este segmento aborda uma variedade de processadores especializados em diferentes domínios, como processamento de imagens, áudio e vídeo. Destaca-se sua arquitetura — muitas vezes utilizando um conjunto de instruções VLIW — otimizada para operações específicas do domínio e a incorporação de operações vetoriais para melhorar o desempenho em aplicações especializadas.

As **TPUs** (*Tensor Processing Units*) são unidades de processamento especializadas desenvolvidas pela Google para otimizar operações relacionadas a tensores — abstrações geométricas que podem ser representadas como vetores multidimensionais, usadas largamente em algoritmos de aprendizado de máquina e servindo de base para a biblioteca **TensorFlow**, também desenvolvida pela Google. Esses processadores Possuem uma arquitetura voltada para operações matriciais e vetoriais, e contribuem para acelerar o treinamento e a inferência de modelos de inteligência artificial.

O estudo dos processadores que suportam operações vetoriais revela a diversidade de arquiteturas e tecnologias disponíveis para acelerar o processamento paralelo em uma variedade de domínios. Esses processadores desempenham um papel crucial no desenvolvimento de algoritmos eficientes e na melhoria do desempenho computacional em aplicações exigentes. O contínuo avanço dessas tecnologias promete impulsionar ainda mais a inovação na computação e expandir os limites do que é possível realizar com eficiência computacional.

2.3.3 Mudança do Paradigma Serial para o Vetorial

A mudança de paradigma da programação serial para a programação vetorial representa uma verdadeira revolução na eficiência computacional. Antes da adoção generalizada da programação vetorial, os algoritmos eram projetados para serem executados de maneira sequencial, o que limitava significativamente o desempenho e a capacidade de lidar com conjuntos de dados massivos e independentes. Com a programação vetorial, no entanto, os desenvolvedores podem realizar operações em grandes conjuntos de dados de forma paralela, aproveitando ao máximo o poder de processamento disponível (HENNESSY; PATTERSON, 2011).

Um exemplo clássico da mudança de paradigma da programação serial para a programação vetorial pode ser observado na computação gráfica. Antes da adoção da programação vetorial, o processo de renderização de imagens em 3D exigia a aplicação de algoritmos sequenciais para calcular cada pixel individualmente. Com a introdução da programação vetorial através do CUDA, por exemplo, os desenvolvedores podem aproveitar a capacidade das GPUs para realizar cálculos em paralelo, acelerando significativamente o processo de renderização e permitindo a criação de gráficos mais realistas e complexos em tempo real.

Outro exemplo impactante da mudança é encontrado no campo do aprendizado de máquina. Antes da adoção da programação vetorial, os algoritmos de aprendizado de máquina muitas vezes enfrentavam limitações de desempenho devido à necessidade de processar grandes conjuntos de dados de forma sequencial. Com a programação vetorial e o uso de frameworks como TensorFlow e PyTorch, os desenvolvedores podem aproveitar o paralelismo das GPUs para treinar modelos complexos em um tempo significativamente menor, abrindo novas possibilidades para aplicações de inteligência artificial em tempo real e análise de big data.

Embora a programação vetorial ofereça inúmeras vantagens em termos de eficiência computacional e desempenho, ela também apresenta desafios significativos. A otimização de algoritmos para aproveitar ao máximo o paralelismo disponível e lidar com questões de sincronização e acesso concorrente aos recursos do sistema tornou-se uma prioridade para os desenvolvedores. No entanto, esses desafios também representam oportunidades de inovação e avanço na área de computação paralela, incentivando o desenvolvimento de técnicas e ferramentas cada vez mais sofisticadas para maximizar o potencial da programação vetorial.

No contexto deste trabalho, serão abordadas técnicas avançadas de agrupamento de dados, incluindo o algoritmo *k-means*. A aplicação dessas técnicas em um ambiente de programação vetorial, como o CUDA, promete explorar todo o potencial de processamento paralelo das GPUs NVIDIA para acelerar significativamente a análise e o agrupamento de

grandes conjuntos de dados. Ao incorporar essas técnicas em um contexto de programação vetorial, busca-se não apenas demonstrar a eficácia das abordagens de agrupamento de dados, mas também destacar o papel crucial da programação vetorial no desenvolvimento de soluções computacionais eficientes e escaláveis para problemas complexos de análise de dados.

2.4 NVIDIA CUDA

O CUDA (*Compute Unified Device Architecture*) é uma API que permite a utilização de uma placa de vídeo com chiptset da NVIDIA para fins de computação de uso geral (*GPGPU*), permitindo o acesso ao conjunto de instruções da placa e a utilização de seus diversos núcleos de processamento para a computação paralela. Foi projetada para trabalhar com linguagens de programação como *Fortran*, *C* e *C++* e possibilita, de dentro da sintaxe delas, o acesso à memória dedicada da placa (VRAM), memória cache, o gerenciamento dos núcleos e *threads* — seu formato e quantidade —, o escalonamento de tarefas para a CPU e GPU, bem como a transferência de dados de um para o outro.

Embora a computação de propósito geral com GPUs fosse possível antes do lançamento de APIs como CUDA, usando outras APIs mais antigas como *OpenGL* ou *DirectX*, o desenvolvimento era bem dificultado, necessitando a conversão de instruções de código serial e escalar para instruções de código paralelo e vetorial, basicamente obrigando desenvolvedores a "traduzir" as aplicações em análogos gráficos para serem processados como texturas ou shaders na GPU, para depois ter os resultados convertidos de volta para um formato de dados menos abstrato. O CUDA facilitou imensamente esse processo permitindo o uso do poder da GPU sem a necessidade de utilização de técnicas e APIs específicas para operações gráficas.

2.4.1 História

Na virada do século, quando percebeu que desenvolvedores viam potencial nas GPUs para além do processamento gráfico, a NVIDIA começou a explorar sua capacidade para tarefas de propósito geral. Com o aumento da demanda por poder computacional e a necessidade de lidar com conjuntos massivos de dados em aplicações não relacionadas a gráficos, surgiu a ideia de utilizar as GPUs para computação paralela. Assim, em 2006, a NVIDIA lançou o CUDA como parte da arquitetura *Tesla*, usada primeiro na série *G80* de GPUs, marcando o início de uma nova era na computação paralela.

O lançamento do CUDA permitiu que os desenvolvedores aproveitassem o poder de processamento massivo das GPUs para uma ampla gama de aplicações computacionais (SANDERS; KANDROT, 2010). Ao fornecer uma plataforma de programação acessível e eficiente, o CUDA abriu as portas para a aceleração de algoritmos complexos em áreas

como aprendizado de máquina, simulação científica, processamento de imagens e muito mais.

Desde então, o CUDA tem passado por várias iterações e atualizações, incorporando novas tecnologias e recursos para tornar a programação em GPUs mais acessível e eficiente. Uma das principais inovações foi a introdução da arquitetura Fermi em 2010, que trouxe melhorias significativas na eficiência energética e na capacidade de processamento das GPUs NVIDIA. Com a Fermi, o CUDA ganhou suporte para novos recursos, como cálculos de precisão dupla de ponto flutuante, o que o tornou ainda mais adequado para aplicações científicas e de computação de alta precisão.

Além disso, o lançamento da arquitetura Kepler em 2012 marcou outro marco importante para o CUDA. Trouxe consideráveis melhorias na eficiência de computação e na capacidade de execução de instruções paralelas, permitindo o desenvolvimento de algoritmos mais complexos e a execução de tarefas de computação intensiva com maior eficiência.

Ao longo dos anos, o ecossistema em torno do CUDA cresceu significativamente, com uma vasta gama de ferramentas, bibliotecas e frameworks disponíveis para desenvolvedores. O lançamento do CUDA Toolkit proporcionou aos desenvolvedores um conjunto abrangente de ferramentas para desenvolver, otimizar e depurar aplicativos CUDA. Além disso, bibliotecas como cuDNN (*CUDA Deep Neural Network Library*) e cuBLAS (*CUDA Basic Linear Algebra Subprograms*) tornaram-se fundamentais para o desenvolvimento de aplicativos de aprendizado de máquina e processamento de dados em larga escala.

Outro aspecto importante do ecossistema CUDA é a comunidade de desenvolvedores, que continua a crescer e contribuir com uma variedade de projetos e recursos. Plataformas como o NVIDIA Developer Forums e eventos como a Conferência de Desenvolvedores NVIDIA (*NVIDIA GPU Technology Conference*) desempenham um papel crucial na promoção da colaboração e na troca de conhecimentos entre os desenvolvedores CUDA em todo o mundo.

O CUDA encontrou aplicação em uma ampla variedade de setores, incluindo ciências, engenharia, medicina, finanças e entretenimento. Empresas e instituições de pesquisa em todo o mundo têm utilizado o CUDA para acelerar suas pesquisas e desenvolver soluções inovadoras para problemas complexos.

Por exemplo, na área da medicina, o CUDA é usado para acelerar simulações de dinâmica molecular e processamento de imagens médicas. No setor financeiro, ele é utilizado para análise de dados em tempo real, modelagem financeira avançada, além de possibilitar diversas implementações na área das criptomoedas e *blockchains*. Na indústria de entretenimento, o CUDA é fundamental para a renderização de gráficos em filmes, jogos e animações em 3D.

Esses exemplos ilustram o impacto significativo que o CUDA teve em uma variedade de domínios, demonstrando seu papel como uma plataforma essencial para a computação paralela e o desenvolvimento de soluções de alto desempenho em todo o mundo.

2.4.2 Exemplo de Implementação CUDA

Para exemplificar bem o processo de paralelização de um algoritmo utilizando CUDA, é apresentada aqui uma implementação simples, na linguagem C++, de um programa que soma dois vetores unidimensionais com mais de 250 milhões de elementos cada, do tipo ponto flutuante de precisão simples (*float*). Esse algoritmo então é paralelizado utilizando a API CUDA para rodá-lo em uma GPU.

Note que os dois grandes vetores inicializados ocupam cerca de 2 GB de memória no total. É importante que a máquina onde o algoritmo é executado possua essa quantidade de memória RAM disponível, assim como de VRAM na GPU. Mais detalhes sobre a máquina utilizada para testes no Capítulo 5.1.

As implementações, assim como suas explicações contidas nesse capítulo, foram adaptadas de tutoriais disponibilizados no site *NVIDIA Developer* (Mark Harris, 2017).

Os códigos de programas CUDA são salvos como arquivos com a extensão *.cu*, e compilados utilizando a ferramenta *nvcc*, disponível através do CUDA Toolkit (NVIDIA Corporation, 2024a). Existe também uma ferramenta que permite a administração, benchmark e monitoramento facilitados de programas CUDA, o *Nsight Systems* disponível em (NVIDIA Corporation, 2024b). Usando seu comando *nsys nvprof* para rodar um executável CUDA, é possível realizar testes automatizados de velocidade.

O Algoritmo 2 abaixo contém o programa supracitado, que soma os dois vetores, salvando no lugar dos valores do segundo vetor a soma dos valores respectivos de ambos.

Algoritmo 2 – Implementação serial de soma de vetores em C++

```
#include <iostream>
#include <math.h>

// Função que adiciona os elementos de dois vetores
void add(long n, float *x, float *y){
  for (long i = 0; i < n; i++)
    y[i] += x[i];
}

int main(void){
  long N = long(1<<28) + long(1<<27); // 402.653.184 elementos

  float *x = new float[N];
```

```
14   float *y = new float[N];
15
16   // Inicializar vetores no host
17   for (long i = 0; i < N; i++) {
18     x[i] = 3.77f; y[i] = 3.23f;
19   }
20
21   // Rodar na CPU
22   add(N, x, y);
23
24   // Checar se há erros (todos os valores devem ser 7.0)
25   float maxError = 0.0f;
26   for (long i = 0; i < N; i++)
27     maxError = fmax(maxError, fabs(y[i] - 7.0f));
28   std::cout << "Max error: " << maxError << "\n";
29
30   // Liberar memória
31   delete [] x;
32   delete [] y;
33
34   return 0;
35 }
```

O código é bem auto-explicativo com comentários incluídos em diversas linhas. Um trecho notável é o contido nas linhas 25–28, que confere o resultado da soma para encontrar o maior erro — definido pelo valor absoluto da subtração entre o valor encontrado no vetor e o valor esperado. Assim é possível constatar qualquer imprecisão que possa ser introduzida ao se modificar o algoritmo.

Rodando o programa e medindo o tempo de execução da função *add*, é encontrado que o tempo médio entre cem execuções consecutivas é de cerca de **462 milissegundos** e o erro máximo é zero.

Para paralelizar o algoritmo e rodá-lo na GPU, é preciso fazer algumas modificações no código. O Algoritmo 3 abaixo apresenta o programa inteiro, em sua versão vetorial na GPU. Essa implementação, no entanto, é consideravelmente ingênua por motivos explicitados a seguir.

Algoritmo 3 – Implementação vetorial (ingênua) de soma de vetores usando CUDA

```
1 #include <iostream>
2 #include <math.h>
3
4 __global__
5 void add(long n, float *x, float *y) {
6   int index = threadIdx.x;
7   int stride = blockDim.x;
```

```
8
9   for (long i = index; i < n; i += stride)
10    y[i] += x[i];
11 }
12
13 int main(void){
14   long N = long(1<<28) + long(1<<27); // 402.653.184 elementos
15
16   float *x, *y;
17   cudaMallocManaged(&x, N*sizeof(float));
18   cudaMallocManaged(&y, N*sizeof(float));
19
20   for (long i = 0; i < N; i++) {
21     x[i] = 3.77f; y[i] = 3.23f;
22   }
23
24   add<<<1, 1024>>>(N, x, y);
25   cudaDeviceSynchronize();
26
27   float maxError = 0.0f;
28   for (long i = 0; i < N; i++)
29     maxError = fmax(maxError, fabs(y[i] - 7.0f));
30   std::cout << "Max error: " << maxError << "\n";
31
32   cudaFree(x);
33   cudaFree(y);
34
35   return 0;
36 }
```

As mudanças em relação ao algoritmo 2 começam na linha 4, onde acima da função *add* é declarado um *kernel* CUDA usando a palavra reservada *___global___*. Ao se inserir esse especificador imediatamente antes da declaração de uma função, ela é marcada como função que pode rodar na placa de vídeo, sendo chamada de *kernel* na documentação.

Nas linhas 5–11, há outra modifcação na função *add* em si. Agora que ela será executada na GPU, é necessário que ela esteja preparada para "dividir" o trabalho igualmente entre as várias instâncias que serão subidas para execução. Cada *thread* da GPU rodará uma cópia da função, paralelamente.

Essa divisão de processamento é realizada usando palavras reservadas providas pela API do CUDA no C++. Elas permitem que o *kernel* saiba sua posição na divisão de blocos e *threads* de processamento que existe na GPU. A variável ***threadIdx.x*** contém o índice, dentro do seu bloco, da *thread* onde está rodando a instância do *kernel*, enquanto a variável ***blockDim.x*** contém a quantidade de *threads* por bloco.

Os valores dessas variáveis são salvos nas variáveis *index* e *stride* (linhas 6 e 7) e passam a ser utilizadas dentro do laço de repetição (linhas 9 e 10). O contador (variável i) é inicializado com o valor do índice da *thread* atual. O laço continua sendo repetido até i chegar ao valor $n-1$, como na versão serial do algoritmo, mas desta vez ele é incrementado com o valor de *stride*, o número de threads por bloco, a cada iteração. Na prática, isso significa que cada instância da função calcula apenas as somas dos elementos dos vetores com índices de valor $index + a * stride$, onde a varia na faixa $[0, (N \div stride) - 1]$.

Ou seja, se $N = 64$ e houver 16 *threads* por bloco, com apenas um bloco no total, a *thread* de índice 0 somaria os valores *x[i] + y[i]* para $i = \{0, 16, 32, 48\}$, a *thread* de índice 1 para $i = \{1, 17, 33, 49\}$, e assim por diante, até a última *thread*, a de índice 15, trabalhando com $i = \{15, 31, 47, 63\}$. O processamento, então, seria efetivamente dividido em 16 partes iguais, cada uma realizada por uma *thread*.

Nas linhas 17 e 18 pode-se perceber que a alocação de memória também muda em relação à versão serial do algoritmo. Ao invés de ser usada a sintaxe *new float[N]*, é necessário usar uma função da API CUDA, a *cudaMallocManaged*. Ela recebe no primeiro argumento o endereço do ponteiro para a estrutura de dados que será alocada, e no segundo argumento a quantidade de memória a ser alocada.

Na linha 24 também há uma mudança crucial na maneira em que é chamada a função que realiza a adição dos vetores. É necessário adicionar a sintaxe *<<<a, b>>>* entre o nome da função e o parênteses que contém seus argumentos. Essa sintaxe informa ao compilador CUDA que a função deve ser executada na GPU, dividindo o processamento em a blocos, com b *threads* cada.

No caso desta implementação, foram utilizadas 1.024 threads por bloco e apenas um bloco. Como em GPUs NVIDIA modernas existem dezenas de SMs (*Streaming Processors*) que, cada um, conseguem rodar centenas a milhares de *threads* simultaneamente, conclui-se que não está sendo utilizado aqui nada próximo do poder total de processamento do hardware. Este é o motivo pelo qual o Algoritmo 3 é considerado uma implementação ingênua.

Na linha 25 há uma chamada à função *cudaDeviceSynchronize*. Isso é necessário pois a chamada de um *kernel* CUDA não bloqueia a execução do código serial na CPU. É preciso esperar que as *threads* da GPU todas terminem de executar para ler com segurança os vetores que foram manipulados, para evitar problemas de concorrência e garantir a conclusão da lógica do algoritmo. Logo, se torna imprescindível chamar essa função neste ponto do código.

As últimas modificações feitas são nas linhas 32 e 33, onde a memória alocada para os vetores é liberada. Assim como a alocação, o processo agora é feito de maneira diferente: chamando a função *cudaFree*, que administra a memória acessível pela CPU e

GPU.

Finalmente, rodando o programa e medindo novamente o tempo de execução como feito no algoritmo serial, o tempo médio de execução dessa versão é de cerca de **151 milissegundos**. O erro máximo também é zero. Houve uma melhora de velocidade de execução (*speed-up*) de cerca de 3,05 vezes, em relação à versão serial rodando em CPU.

Todavia, essa performance pode ser melhorada ainda mais utilizando mais poder de processamento da placa de vídeo, através da divisão do processamento em mais blocos e threads. Para isso, poucas modificações devem ser feitas ao Algoritmo 3. Os trechos de código modificado seguem abaixo.

Algoritmo 4 – Implementação vetorial de soma de vetores usando CUDA — Trecho 1

```
int blockSize = 1024;
int numBlocks = ceil(N / blockSize);

add<<<numBlocks, blockSize>>>(N, x, y);
```

A modificação acima é feita na chamada da função *add* (linha 24 no Algoritmo 3). O tamanho do bloco é definido como 1.024 threads, como antes, mas o valor é salvo na variável *blockSize*, para ser utilizada na definição do número de blocos. Esse número é definido dividindo N pela quantidade de blocos por *thread*, arredondando para cima (com a função *ceil*) para lidar com o caso de N não ser divisível por *blockSize*. Assim, é garantido que haverão, no mínimo, N *threads* no total. Nesse caso específico ($N = 2^{28}+2^{27}$), haverão exatamente 393.216 *blocos*, cada um com 1.024 *threads*.

No entanto, como há mais de um bloco agora, é necessário que seja modificado também o *kernel add*. Isso pois é preciso lidar com a aritmética de divisão de processamento entre os blocos, além de threads, desta vez.

Algoritmo 5 – Implementação vetorial de soma de vetores usando CUDA — Trecho 2

```
__global__
void add(long n, float *x, float *y){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (long i = index; i < n; i += stride)
    y[i] += x[i];
}
```

O trecho acima (que corresponde às linhas 4–11 do Algoritmo 3), descreve a nova definição do *kernel*. Aqui, é usada uma nova variável disponibilizada pela API CUDA, a *blockIdx.x*, que contém o índice do bloco onde se encontra a *thread* atual. Usando essa variável e as outras duas já conhecidas, é possível dividir ainda mais o processamento da soma dos vetores.

A variável *index* é definida dessa vez pela expressão idiomática CUDA *blockIdx.x * blockDim.x + threadIdx.x*, enquanto a variável *stride* agora é definida por *blockDim.x * gridDim.x*. A última variável dessa expressão é mais uma fornecida pela API, e descreve a quantidade de blocos totais, chamada de *grid*. Essa nova aritmética de identificação do bloco e *thread* onde a função sendo executada se encontra, permite uma divisão de processamento ainda maior que anteriormente.

Para clarificar, num exemplo mais simples onde $N = 128$ e há 2 blocos com 32 *threads* cada, a *thread* de índice 0 do bloco de índice 0 somaria os valores *x[i] + y[i]* para $i = \{0, 64\}$, a *thread* de índice 1 do mesmo bloco para $i = \{1, 65\}$, e assim por diante, até a última *thread* do bloco 0, a de índice 31, trabalhando com $i = \{31, 95\}$. No bloco de índice 1, o mesmo ocorre, mas agora sua *thread* de índice 0 processa os elementos onde $i = \{32, 96\}$, a *thread* 1 trabalha com $i = \{33, 97\}$, até chegar na sua última *thread*, a de índice 31, que processa $i = \{63, 127\}$. O processamento, então, seria efetivamente dividido em 64 partes iguais, cada uma realizada por uma *thread*.

Rodando o algoritmo e medindo o tempo cada execução, obtem-se o tempo médio de cerca de **17 milissegundos**. O erro máximo continua em zero. Houve agora um *speed-up* de cerca de 8,88 vezes em relação à versão vetorial anterior (Algoritmo 3) e de cerca de 27,17 vezes em relação à versão serial rodando em CPU (Algoritmo 2).

Tabela 1 – Comparação de performance da soma de vetores (com $2^{28} + 2^{27}$ elementos) em C++. *Speed-up* calculado em relação à versão serial (CPU)

Implementação	**Tempo de Exec. (ms)**	**Speed-up**
CPU (Algoritmo 2)	462	—
GPU (ingênua) (Algoritmo 3)	151	3,05x
GPU (Algoritmo 4 e 5)	17	27,17x

Como é possível perceber, as implementações em CUDA requerem um conhecimento em uma das linguagens a que dá suporte nativo, como C, C++ e Fortran, além de uma manipulação mais direta da memória da CPU e GPU. Seu uso também prescinde de um gerenciamento avançado da aritmética usada para gerenciar execuções paralelas, gerando um código que pode ser confuso e de difícil manutenção.

Tais dificuldades impulsionaram uma das maiores motivações desta pesquisa: a busca por soluções que simplificassem ainda mais o uso do poder de paralelização das GPUs NVIDIA, o que é explorado na seção seguinte.

2.5 Biblioteca Numba

Como visto no capítulo anterior, a utilização da biblioteca CUDA requer conhecimentos nas linguagens onde ela foi implementada (C, C++, Fortran), além de técnicas

de programação paralela que podem envolver uma aritmética complexa, principalmente para problemas com alta dimensionalidade, como é o caso de muitos datasets utilizados na área de *big data*, *data mining* e ciência de dados.

Como uma tentativa de unir o melhor dos dois mundos — a velocidade de uma linguagem compilada para rodar em diversas CPUs e GPUs, utilizando o poder do paralelismo, e o desenvolvimento rápido e fácil em uma linguagem mais "amigável" e alto nível — a desenvolvedora da distribuição Anaconda criou a biblioteca e compilador *open-source* ***Numba*** (Anaconda, Inc., 2024), trazendo para a linguagem Python a possibilidade de ganhos de performance da área de *GPGPU*. Usando um mínimo de sintaxe nova, é possível acelerar de maneira quase automática códigos Python, rodando tanto em CPUs quanto em GPUs.

2.5.1 Exemplo de Implementação Numba

Seguindo o foco desse trabalho, a biblioteca foi utilizada para tornar mais performáticas as partes mais custosas computacionalmente do algoritmo k-means, através da vetorização de funções anteriormente seriais.

Para exemplificar como foi feito o processo, um algoritmo simples, equivalente ao usado para demonstrar o uso da API CUDA em C++ no Capítulo 2.4.2, foi implementado, primeiro serialmente, e depois paralelamente, utilizando a GPU.

O Algoritmo 6 abaixo executa a adição de dois vetores, retornando o resultado num terceiro vetor novo. Note que foram importadas, em ambas versões do algoritmo, as bibliotecas *Numba* e *Numpy*, importada com o nome *np*. Os comandos de **`import`** foram todos omitidos por brevidade. Foram utilizados vetores e funções do *Numpy* pois são mais performáticos do que seus equivalentes em Python puro.

Algoritmo 6 – Implementação serial de adição de vetores 2D em Python

```
1  def addArrayCPU(a, b):
2      return a + b
3
4  def checkMaxErr(c):
5      # Checando erro máximo (todos elementos devem ser 42.0):
6      minElem = c.min()
7      maxElem = c.max()
8      maxErr = 0.0
9
10     maxErr = max(maxErr, abs(42.0 - minElem))
11     maxErr = max(maxErr, abs(42.0 - maxElem))
12     print(f'Max error: {maxErr}')
13
14 N = 2**28 + 2**27 # 402.653.184 elementos
15
```

```
16 # Inicializando vetores
17 a = np.full(N, 27.2, np.float32)
18 b = np.full(N, 14.8, np.float32)
19
20 # Realizando adição
21 c = addArrayCPU(a, b)
22
23 checkMaxErr(c)
```

O código acima é auto-explicativo e está comentado para melhor compreensão. Se assemelha bastante com o Algoritmo 2 exibido no Capítulo 2.4.2. Os vetores continuam com o mesmo tamanho de $N = 402.653.184$. A única diferença prática é que aqui o resultado é retornado em um novo vetor (variável c), ao invés de haver uma sobrescrita dos valores do segundo vetor da soma (varíavel b).

Rodando o código e realizando a medição da velocidade de execução da função *addArrayCPU*, encontra-se o tempo médio (de cem execuções) de cerca de **1.624,79 milissegundos**. O erro máximo (calculado pela função *checkMaxErr* na última linha) é de zero.

Adaptar esse algoritmo para rodar paralelamente na GPU é extremamente simples usando a biblioteca Numba. O trecho de código abaixo mostra a única modificação a ser feita ao chamar a função que soma os vetores.

Algoritmo 7 – Implementação vetorial de adição de vetores em Python — trecho 1

```
# Inicializando vetor de retorno
c = np.zeros(N, np.float32)

# Realizando adição
addArrayGPU(a, b, c)
```

A função, chamada dessa vez de *addArrayGPU*, agora precisa de um vetor de retorno, inicializado aqui logo antes de sua chamada. Isso é necessário pois o recurso do Numba utilizado na implementação da função não permite o retorno direto de uma variável.

Esse recurso do Numba é o decorador ***guvectorize***, inserido antes da declaração da função *addArrayGPU*. O *guvectorize* (abreviação de ***Generalized Universal Functions***, ou **Funções Generalizadas Universais**) informa ao compilador que a função deve ser compilada para rodar na placa de vídeo, e não na CPU como o resto do código. No trecho de código abaixo temos a declaração da função *addArrayGPU* que faz uso desse decorador.

Algoritmo 8 – Implementação vetorial de adição de vetores em Python — trecho 2

```
@numba.guvectorize(
```

```
    ['void(float32[:],float32[:],float32[:])'],
    '(),()->()', nopython=True, target='cuda'
)
def addArrayGPU(a:np.float32, b:np.float32, c:np.float32):
    c[0] = a[0] + b[0]
```

Com apenas essa modificação a mais o código está pronto para ser executado paralelamente pela GPU. Note que, mesmo a função *addArrayGPU* lidando apenas com valores escalares, eles devem ser acessados como o primeiro elemento de listas (*a[0]* ao invés de apenas *a*). Isso é um requerimento do uso da funcionalidade *guvectorize* do *Numba*.

O funcionamento da função não é exatamente intuitivo. Como é possível ver na única linha do corpo da função, ela trabalha com apenas um elemento de cada vetor. Contudo, diferente de como era feito em C++ com a API CUDA diretamente, não é necessário que seja feita nenhuma aritmética interna para diferenciar as execuções da milhares de instâncias da função.

De fato, tudo o que o corpo da função faz é somar os valores do elemento *a* com o *b* e armazenar o resultado no *c*. Mas quando a função é chamada, o que é passado de argumento não são os elementos dos vetores, mas sim os vetores inteiros. O que parece ser um erro na verdade é exatamente como a implementação do paralelismo funciona ao se utilizar tal recurso do Numba.

Essa abstração só é possível pois a biblioteca Numba se encarrega de traduzir esse código para seu equivalente em CUDA, gerenciando sozinha toda a aritmética necessária para garantir que realmente cada instância da função acesse apenas sua porção definida dos dados dos vetores inteiros passados como argumentos, e o processamento seja de fato dividido entre milhares de *threads*. Além disso, a biblioteca e seu compilador se encarregam de automatizar qualquer alocação e transferência de estrutura de dados entre CPU e GPU ou vice-versa.

Tudo que é necessário para tal funcionamento extremamente automatizado de paralelização é, primeiro, implementar a função vetorial em Python como uma que recebe e processa apenas um elemento da estrutura de dados maior que de fato está sendo processada, como foi feito na última linha do Algoritmo 8, e, segundo, informar ao decorador *guvectorize* alguns argumentos a respeito dos dados que a função irá receber e também sobre seu modo de compilação e execução.

São passados quatro argumentos para o decorador. O primeiro é uma lista de um elemento do tipo *string* que define os tipos de variáveis que cada instância da função vai receber, em sua *thread*. Nesse caso, como há três argumentos em *addArrayGPU*, são definidos três tipos dentro dos parênteses após a palavra *void*. Todos os tipos são *float32[:]*, o que indica que são escalares ou vetores unidimensionas contendo números de ponto flutuante (*float*) de precisão simples. Nesse caso, os elementos serão escalares. Se fossem

vetores bidimensionais, a sintaxe seria *float32[:,:]*; se tridimensionais, *float32[:,:,:]* e assim por diante.

O segundo argumento do decorador é uma *string* que denota a dimensionalidade dos argumentos que cada instância da função vai receber e processar, em sua *thread*. A sintaxe aqui usada, *(),()->()* indica que há dois valores escalares de entrada e um de saída. Para vetores unidimensionais de tamanho n, a sintaxe seria *(n),(n)->(n)*; para bidimensionais de tamanho $n \cdot d$, seria *(n,d),(n,d)->(n,d)* e assim por diante. Importante notar que a dimensionalidade pode diferir entre os argumentos.

Note que ambos o primeiro e segundo argumento do decorador definem o formato de dado que cada instância da função recebe, e não o formato da estrutura de dados inteira a ser processada. Como a função foi definida para receber um valor escalar, quando ela é de fato chamada e um vetor de tamanho n é passado, o compilador do *Numba* infere que devem ser criadas n instâncias da função, e cada uma delas processará apenas um elemento diferente, de valor escalar, paralelizando automaticamente o processo.

O terceiro argumento, de nome *nopython*, deve ser definido como *True* para garantir que o código gerado seja todo executado na GPU. Sem essa definição, recursos não implementados pela biblioteca que fossem utilizados dentro da função iriam causar uma execução na CPU — um modo chamado de *object mode*.

O quarto e último argumento, de nome *target*, recebe uma *string* que define onde a função irá rodar. Nessa pesquisa, sempre foi utilizado o valor *'cuda'* aqui, para que todas as funções paralelizadas rodem na GPU NVIDIA. Outras opções existem, como *cpu* para execução *single-thread* na CPU ou *paralell* para execução *multi-thread* na CPU, mas elas fogem do escopo desse trabalho.

Rodando o algoritmo e medindo, como anteriormente, a velocidade de execução da função *addArrayGPU*, se constata o tempo médio de aproximadamente de **384,86 milissegundos**. O erro máximo também é zero, como esperado. Foi obtido um *speed-up* de cerca de **4,22 vezes** em relação à versão serial implementada usando apenas funções do *Numpy* (que rodava, em média, em 1624.79 ms).

Fica evidente a facilidade muito maior em se implementar o poder do paralelismo proporcionado pela API CUDA ao se usar a linguagem Python em conjunto com a biblioteca Numba.

É possível adentrar na área de GPGPU sem abrir mão de abstrações de mais alto-nível proporcionadas pela linguagem, além de usufruir, simultaneamente, da imensa gama de bibliotecas de ciência de dados implementadas em Python, como *Numpy* e *Pandas*, que foram também amplamente utilizadas nas implementações do algoritmo k-means, exploradas no Capítulo 4.2.

É importante notar também que ainda há uma penalidade considerável no poten-

cial de ganho de performance, se comparando implementações usando CUDA em C++ e as usando Numba em Python. O ganho de velocidade foi de 27,17 vezes em C++, enquanto em Python foi de apenas 4,22 vezes. Ainda assim, o *speed-up* obtido nas execuções do k-means acelerado usando o Numba foram de magnitudes ainda maiores que o apresentado neste exemplo simples, como explicitado no capítulo 5.4.

Não se pode deixar de destacar, no entanto, que a versão em C++ da soma de vetores rodando na GPU (Algoritmos 4 e 5) tem um tempo de execução cerca de 22,63 vezes mais rápido que a versão Python rodando em GPU (Algoritmos 7 e 8) — 17 ms contra 384,86 ms, respectivamente.

3 Levantamento do Estado da Arte

No contexto histórico, o aumento exponencial no volume de dados gerados e armazenados digitalmente tornou-se uma realidade desde as últimas décadas do século XX. Com o advento da internet, mídias sociais, dispositivos inteligentes e sensores, a quantidade de dados disponíveis cresceu exponencialmente. Esses dados não estruturados, em sua maioria, requerem técnicas avançadas para extrair informações valiosas e úteis (HAN; KAMBER; PEI, 2012).

Nesse cenário, os algoritmos de agrupamento emergem como uma ferramenta essencial para entender a estrutura subjacente dos dados, identificar padrões, segmentar clientes, recomendar produtos e até mesmo na medicina para classificar pacientes com base em características semelhantes. No entanto, à medida que os conjuntos de dados crescem em escala e complexidade, a eficiência computacional torna-se uma preocupação crítica.

A necessidade de processamento mais rápido de grandes conjuntos de dados é evidente. Os algoritmos de agrupamento, especialmente quando aplicados a conjuntos de dados volumosos, podem se tornar computacionalmente intensivos e demandar uma quantidade considerável de tempo de execução. Isso não apenas limita a capacidade de análise em tempo hábil, mas também impõe restrições sobre a escalabilidade das soluções de análise de dados.

Portanto, surge a necessidade de paralelizar esses algoritmos, aproveitando o poder computacional de sistemas distribuídos, clusters de computadores ou arquiteturas de hardware com múltiplos núcleos. A paralelização não apenas acelera o processo de agrupamento, mas também permite lidar com conjuntos de dados cada vez maiores, garantindo que as análises permaneçam viáveis e eficientes em um cenário de big data.

As pesquisas aqui resumidas buscam explorar a história da paralelização de algoritmos de agrupamento, destacando os avanços significativos nesta área e sua importância contínua na era da análise de enormes volumes de dados. Este segmento do trabalho serve como uma ponte entre o passado e o presente, ilustrando não apenas estudos mais antigos que formaram uma base teórica e prática na áreal, mas também como a introdução das (GPUs) transformou fundamentalmente este campo. Ao revisitar os marcos históricos, é possível delinear o caminho de inovações e aprimoramentos que permitiram a jornada até a era atual, onde a paralelização possibilita o processamento de conjuntos de dados de magnitude anteriormente inimaginável com eficiência e rapidez sem precedentes.

Como evidenciado nas seções seguintes, a avaliação dos estudos recentes sobre a paralelização de algoritmos de agrupamento, especialmente os que utilizam a plataforma

CUDA para implementá-los em GPUs NVIDIA, revela avanços significativos tanto na eficiência quanto na utilidade prática dos processos de agrupamento. Estas melhorias são notáveis em aplicações que variam desde o processamento de imagens até a física de alta energia.

Os avanços na utilização de processadores vetoriais para paralelização têm demonstrado que é possível obter reduções significativas nos tempos de processamento, mantendo, ou até mesmo melhorando, a qualidade dos resultados destes tipos de algoritmos.

No entanto, apesar dos avanços significativos, ainda existem lacunas importantes a serem preenchidas. Uma das principais lacunas é a falta de algoritmos paralelos que sejam eficientes para diferentes tipos e tamanhos de conjuntos de dados. Outra área que merece atenção é a escalabilidade dos algoritmos paralelizados. À medida que os conjuntos de dados continuam a crescer em tamanho e complexidade, torna-se crucial que os algoritmos de agrupamento possam escalar eficientemente para atender a essas demandas crescentes.

Direções futuras na pesquisa podem incluir o desenvolvimento de algoritmos paralelos que sejam mais adaptáveis a diferentes tipos e tamanhos de dados, além da integração de técnicas de aprendizado de máquina para melhorar a precisão e eficiência dos processos de agrupamento. Além disso, pode ser útil explorar mais a fundo o potencial das novas arquiteturas de GPU e outras plataformas de computação paralela, como as TPUs e FPGAs (vide o capítulo 2.3.2), para avançar ainda mais na paralelização destes algoritmos.

A exploração de técnicas híbridas, que combinem métodos de agrupamento clássicos com novas abordagens baseadas em aprendizado profundo (*Deep Learning*), também pode oferecer caminhos promissores para melhorar tanto a velocidade quanto a qualidade dos algoritmos. Finalmente, há uma necessidade contínua de pesquisa que aborde questões de eficiência energética e sustentabilidade ambiental no contexto da computação de alto desempenho aplicada ao agrupamento de dados.

3.1 Primeiras Implementações Paralelas

A ideia de acelerar a execução de algoritmos de agrupamento de dados utilizando a computação paralela não é nova. Desde o advento das pesquisas envolvendo tais processos e do aumento cada vez mais rápido de volume de dados disponíveis para análise, vem-se fazendo evidente a necessidade de otimizar ao máximo esses algoritmos.

Um dos estudos mais antigos encontrados que tenta paralelizar um algoritmo desse tipo é o "*Parallel K-means Clustering Algorithm on NOWs*" (KANTABUTRA; COUCH, 2000).

Nesse artigo, é apresentada uma abordagem para melhorar a eficiência do **algoritmo k-means** através da paralelização. O objetivo principal é reduzir a complexidade

temporal do algoritmo k-means serial aplicando teorias de computação paralela, alcançando uma melhoria de um fator de $O(K/2)$, onde K é o número de clusters desejados. Além disso, visa permitir que o algoritmo seja executado em uma memória coletiva maior, composta por várias máquinas, superando a limitação de memória de uma apenas uma, o que permite escalar o tamanho do problema em até $O(K)$ vezes o tamanho que poderia ser processado em uma única máquina.

A pesquisa foi realizada utilizando uma rede de estações de trabalho homogêneas com uma rede Ethernet e a comunicação entre processos foi feita por meio da Interface de Passagem de Mensagens (MPI). O método proposto usa uma abordagem mestre-escravo (*master-slave*) onde um processo mestre distribui subconjuntos de dados para processos escravos, que calculam médias locais e participam de um processo iterativo de realocação de pontos de dados para minimizar uma função de erro quadrático. A análise de complexidade de tempo e espaço foi detalhadamente realizada, destacando a eficiência do algoritmo paralelizado em comparação com sua versão serial.

Os resultados experimentais mostraram que, para conjuntos de dados grandes (mais de 700.000 datapoints), a versão paralela do algoritmo conseguiu obter uma melhoria significativa na velocidade de execução, evidenciada por um "speed-up" de até O(K/2) conforme o tamanho do problema aumenta. Isto foi particularmente notável quando o tamanho do problema excedeu a capacidade de memória de uma única máquina, um cenário onde a versão serial do algoritmo não pôde sequer ser executada.

No fim, a pesquisa demonstra que a paralelização do algoritmo k-means em uma rede de estações de trabalho oferece uma melhoria significativa tanto em termos de complexidade temporal quanto de escalabilidade do tamanho do problema, alcançando uma eficiência de 50% na redução do tempo de complexidade. Esta eficiência é considerada alta para a época e dada a natureza global do algoritmo k-means e o uso de um sistema baseado em passagem de mensagens Ethernet. O trabalho sugere que futuras pesquisas poderiam focar na redução do tempo de comunicação ou na adaptação do algoritmo para funcionar com um número flexível de máquinas, visando melhorias ainda maiores na eficiência e escalabilidade.

Outro estudo antigo que mostra o início das tentativas de paralelizção de algoritmos de agrupamento é o "*Parallel K-Means Algorithm on Distributed Memory Multiprocessors*" (JOSHI, 2003). Aqui, é explorada uma **implementação paralela do k-means**, visando acelerar o agrupamento de grandes conjuntos de dados em um cluster de *workstations* da Sun Microsystems. A pesquisa objetiva explorar o paralelismo de dados inerente ao k-means, utilizando o modelo de passagem de mensagens para dividir o conjunto de dados entre processos, buscando reduzir o tempo total de computação.

O algoritmo implementado no estudo divide o conjunto de dados entre os processos de um sistema de memória distribuída, onde cada processo é responsável por uma

parte dos dados. Esta abordagem visa calcular a associação datapoint-cluster para cada partição de forma mais rápida. A comunicação entre os processos, embora necessária para a recomputação dos centróides e a avaliação da qualidade dos clusters, representa um custo significativo, especialmente para conjuntos de dados menores, onde o tempo de comunicação domina em relação ao tempo de processamento dos dados. A pesquisa destaca a importância de minimizar essa comunicação para melhorar a eficiência do algoritmo paralelo.

A implementação utiliza o modelo de Programa Único Múltiplos Dados (SPMD) e a Interface de Passagem de Mensagens (MPI) para a comunicação entre processadores. Inicialmente, o processo raiz calcula os centróides iniciais e os transmite a todos os outros processos. Cada processo então computa distâncias, atribui pontos ao centróide mais próximo, e calcula erros quadráticos médios locais. Essas operações são repetidas até a convergência, com processos recomputando centróides e avaliando a qualidade global do agrupamento.

Embora a paralelização tenha mostrado benefícios, como a distribuição eficiente do processamento e a possibilidade de lidar com conjuntos de dados maiores, o ganho de velocidade ótimo não foi alcançado devido ao custo sequencial associado à escolha dos centróides iniciais e à influência de outliers na formação dos clusters. A pesquisa sugere que futuros trabalhos podem explorar a paralelização em multiprocessadores de memória compartilhada, abordar questões de valores ausentes e outliers, e testar variantes probabilísticas do k-means ou o algoritmo K-modes para conjuntos de dados categóricos de grande escala.

Esse estudo demonstra o potencial e os desafios da paralelização do k-means para o agrupamento de dados em grandes escalas, tendo aberto caminho para pesquisas futuras voltadas à otimização da eficiência e da capacidade de processamento desses algoritmos em ambientes de computação de alta performance, para a época.

Ambos dos últimos dois estudos comentados predatam implementações em GPU de algoritmos de agrupamento de dados, demonstrando tentativas de paralelização com outras técnicas além da área de GPGPU, usando técnica como a computação distribuída, processamento paralelo SIMD, entre outras.

Para explorar as primeiras implementações de algoritmos de agrupamento que utilizaram especificamente placas de vídeo para paralelização é preciso avançar ao menos um ano na linha do tempo.

O estudo mais antigo encontrado a utilizar essa técnica é o "*GPU Acceleration of Iterative Clustering*" (HALL; HART, 2004). Esse estudo apresenta uma abordagem inovadora para **acelerar algoritmos iterativos de agrupamento**, como o **k-means** e a **Análise de Componentes Principais Agrupada (CPCA)**, utilizando o poder

computacional das unidades de processamento gráfico. A pesquisa destaca o potencial das GPUs não apenas para gráficos e renderização, mas também para acelerar algoritmos amplamente utilizados em campos como visão computacional, processamento de sinais, compressão de dados e geometria computacional. O foco é na utilização da arquitetura de streaming de alta performance das GPUs para executar a parte mais custosa computacionalmente desses algoritmos, resultando em acelerações significativas.

O método proposto utiliza a arquitetura das GPUs para realizar as avaliações métricas, que são a parte mais intensiva em termos de cálculos dos algoritmos de agrupamento. Os autores desenvolveram uma versão hierárquica do algoritmo k-means, projetada para aumentar o paralelismo SIMD, resultando em um desempenho melhorado em comparação com abordagens hierárquicas tradicionais do k-means, que podem não aproveitar totalmente o hardware das GPUs. A GPU lida com a tarefa de particonamento dos dados, enquanto a CPU é responsável por atualizar os modelos de cada cluster, otimizando o uso dos recursos computacionais disponíveis.

Nos resultados apresentados, a aceleração obtida varia entre 1,5 a 3 vezes em comparação com a execução apenas na CPU, dependendo do tamanho do conjunto de dados. Este aumento de desempenho é particularmente relevante para datasets grandes e métricas complexas, onde a aceleração permite soluções mais rápidas e viáveis para problemas que antes eram proibitivamente lentos para resolver. O estudo também discute as limitações atuais e potenciais futuras melhorias, incluindo a implementação de etapas de atualização do modelo diretamente nas GPUs, o que poderia reduzir ainda mais o tempo de computação.

No fim, o trabalho demonstra o potencial das GPUs para acelerar significativamente algoritmos de agrupamento iterativos, abrindo caminho para aplicações interativas e em tempo real que antes não eram possíveis. As futuras direções de pesquisa sugeridas incluem a exploração de métodos para minimizar a comunicação entre a GPU e a CPU e expandir a aplicabilidade dessa abordagem para métricas de agrupamento que não podem ser avaliadas independentemente para cada ponto. Este estudo não apenas destaca a utilidade das GPUs além da área gráfica, mas também promove a investigação contínua para explorar plenamente suas capacidades para acelerar uma gama ainda mais ampla de algoritmos computacionais.

3.2 K-means e Variantes

Partindo para estudos mais recentes, representando o estado da arte da área, e focando mais no k-means e suas diversas variantes, é de destaque o estudo "*Parallelization of Partitioning Around Medoids* [...]" (PRAHARA; ISMI; AZHARI, 2020), onde os autores propuseram uma implementação paralela do **algoritmo de agrupamento K-**

Medoids, especificamente da sua versão conhecida como **PAM (*Partitioning Around Medoids*)**, que é utilizada para dividir conjuntos de dados em clusters de forma que minimizem as distâncias internas. Esta versão paralela foi desenvolvida para ser executada em Unidades de Processamento Gráfico (GPUs) utilizando a arquitetura CUDA da NVIDIA.

O principal desafio do K-Medoids reside em seu alto custo em tempo de execução e em uso de espaço de memória, especialmente para grandes conjuntos de dados, o que pode tornar sua aplicação inviável em contextos práticos. Para superar esses empecilhos, os autores optaram por uma abordagem paralela, implementada em CUDA, e que não necessita do pré-cálculo de uma tabela completa de distâncias, algo quase onipresente em implementações anteriores do K-Medoids, reduzindo assim o consumo de memória e acelerando muito o processo de execução.

Os resultados foram promissores, demonstrando que a versão paralelizada em GPU do algoritmo PAM conseguiu uma melhoria significativa de desempenho em comparação com as implementações tradicionais em CPU e até mesmo com implementações em Matlab — ambas estas utilizam a tabela de distâncias pré-calculada, custosa em uso memória. Especificamente, o estudo relatou um aumento de velocidade de 11 a 15 vezes em relação à implementação em CPU, e de 2 a 3 vezes em relação ao Matlab, para grandes volumes de dados.

Este avanço indica que o algoritmo K-Medoids, adaptado para uso altamente paralelizado em GPUs, torna-se uma alternativa mais viável para o agrupamento de grandes conjuntos de dados, oferecendo melhorias tanto em termos de tempo de execução quanto na capacidade de lidar com muitos pontos de dados sem exigir quantidades excessivas de memória. Portanto, a pesquisa contribui significativamente para a área de mineração de dados e aprendizado de máquina, abrindo novas possibilidades para o uso eficiente do K-Medoids em aplicações práticas de big data.

3.3 Algoritmos Hierárquicos

O **Agrupamento Aglomerativo Paralelo** é uma técnica fundamental no campo da mineração de dados e aprendizado de máquina, especialmente quando lidamos com grandes conjuntos de dados. Tradicionalmente, os **algoritmos de agrupamento aglomerativo** (**HAC**, em inglês), conhecidos por sua abordagem hierárquica, eram limitados pela capacidade computacional e de memória das máquinas. Com a evolução da computação paralela, surgiu a necessidade de adaptar estes algoritmos para ambientes onde múltiplos processos podem ser executados simultaneamente, melhorando significativamente a eficiência e a escalabilidade do agrupamento de grandes quantidades de instâncias.

Antes do desenvolvimento do algoritmo k-means, um dos métodos de agrupamento mais populares, havia um forte interesse no agrupamento aglomerativo devido à sua capa-

cidade de revelar a estrutura hierárquica dos dados. No entanto, sua aplicação era bastante limitada devido ao alto custo computacional e à demanda por grandes quantidades de memória. A paralelização do agrupamento aglomerativo surgiu como uma solução para essas limitações, permitindo o processamento de dados em grande escala de uma maneira mais viável.

Um avanço significativo no Agrupamento Aglomerativo Paralelo foi realizado através do desenvolvimento do **framework ParChain**, discutido no artigo "*ParChain: A Framework for Parallel Hierarchical* [...]" (YU et al., 2021). O ParChain propõe uma estrutura para projetar algoritmos paralelos de agrupamento hierárquico aglomerativo que utilizam memória linear, em contraste com a memória quadrática requerida pelos algoritmos paralelos anteriores. Baseado na paralelização do algoritmo de cadeias de vizinhos mais próximos, o ParChain permite que múltiplos clusters sejam mesclados em cada rodada, melhorando a eficiência e a escalabilidade do processo de agrupamento.

O estudo demonstrou que implementações altamente otimizadas do ParChain, utilizando 48 núcleos com *hyper-threading* bidirecional, alcançaram uma aceleração significativa em comparação com os algoritmos paralelos HAC de última geração. Mais especificamente, observou-se uma aceleração entre 5,8–110,1 vezes no tempo de execução, além de uma redução de até 237,3 vezes no espaço necessário. Assim, o framework foi capaz de escalonar para tamanhos de conjuntos de dados com dezenas de milhões de pontos — um feito que os algoritmos existentes não conseguiam alcançar.

A introdução do HAC paralelo, e particularmente do framework ParChain, marcou um ponto de virada na análise de dados em grande escala, permitindo a exploração de estruturas de dados complexas de maneira mais eficiente e profunda. Este desenvolvimento não apenas superou as limitações dos métodos de agrupamento anteriores, mas também abriu novas avenidas para pesquisas futuras, incluindo a otimização de outros critérios de ligação (entre pontos de dados e grupos) e a aplicação em diferentes domínios de dados.

3.4 Outras Pesquisas Relevantes

O estudo "*CLUE: A Fast Parallel Clustering Algorithm for* [...]" (ROVERE et al., 2020) expõe um **novo algoritmo de agrupamento** chamado **CLUstering of Energy (CLUE)**, destinado a otimizar o processo de agrupamento em calorímetros de alta granularidade utilizados em física de alta energia. O algoritmo foi projetado para ser totalmente paralelizável e eficiente, lidando com um grande número de "hits" ou detecções de depósitos de energia, que podem variar em número a cada detecção numa faixa entre milhares a milhões, dependendo da granularidade e do número de partículas que entram no detector.

O CLUE utiliza uma abordagem baseada em densidade para o agrupamento, calculando duas variáveis-chave para cada ponto: a densidade local e a separação. Utiliza

também um índice espacial de grade fixa para acelerar a consulta de vizinhos, atribuindo todos os pontos de dados a quadrantes de uma malha, tornando o processo de busca por vizinhos mais rápido e escalável. Além disso, o algoritmo pode efetivamente identificar e agrupar formatos de clusters não-esféricos e rejeitar ruídos, adaptando-se às necessidades específicas da análise de dados em calorímetros.

A implementação do CLUE em GPUs mostrou ser significativamente mais rápida do que as implementações em CPU de thread único, alcançando um aumento de velocidade de 48 a 112 vezes, dependendo do número de pontos processados. Esse desempenho é crucial para a reconstrução de eventos em física de partículas, onde o tempo de processamento é limitado e grandes volumes de dados precisam ser analisados rapidamente.

O estudo confirmou que o algoritmo CLUE é altamente escalável, mantendo um desempenho linear em relação ao número de pontos de entrada, o que é ideal para o tratamento de dados provenientes de calorímetros de alta granularidade, como os previstos para o CMS no LHC de alta luminosidade.

Este desenvolvimento representa um avanço significativo na análise de dados em física de alta energia, permitindo um processamento de dados mais rápido e eficiente, o que é essencial para explorar o potencial completo de futuros experimentos de física de partículas.

Outro estudo muito relevante é o "*Evaluation of Clustering Algorithms on GPU-Based Edge Computing Platforms*" (CECILIA et al., 2020), que analisou a viabilidade de executar algoritmos de agrupamento de dados, computacionalmente exigentes, em **plataformas de computação de borda** (***Edge Computing***, uma abordagem que permite computação distribuída de baixo custo computacional nas bordas de uma rede, o mais próximo possível do cliente) equipadas com GPUs de baixo consumo. Foram testados três algoritmos de agrupamento diferentes: **K-means**, **Fuzzy Minimals (FM)** e **Fuzzy C-means (FCM)**, em dois contextos: **computação de alto desempenho (HPC)** e computação de borda.

Os resultados mostraram que, ao usar as GPUs em plataformas de borda como a NVIDIA AGX Xavier, foi possível obter uma aceleração significativa em comparação com as versões sequenciais dos algoritmos rodando nas próprias plataformas de borda. Especificamente, observou-se um aumento de velocidade de até 11 vezes para os códigos GPU em relação às versões sequenciais. Além disso, comparando as plataformas de computação de borda com as plataformas HPC, houve economias de energia de até 150% ao usar a computação de borda em vez da versão HPC.

Portanto, este estudo concluiu que as plataformas de computação de borda equipadas com GPUs de baixo consumo oferecem uma alternativa viável e muito mais energeticamente eficiente para a execução de algoritmos de agrupamento de dados pesados.

Isso abre novas possibilidades para aplicativos de IoT avançados, onde a análise de dados pode ser realizada mais perto da fonte de dados, reduzindo a latência e o consumo de energia associados à transmissão de grandes volumes de dados para a nuvem ou outros centros de dados remotos.

Além disto, é destacado no estudo que essas melhoras de desempenho possivelmente possibilitarão a análise de dados considerados como *dark data*: enormes volumes de dados gerados diariamente por dispositivos IoT que costumavam nunca ser de fato analisados. Essa interpretação "inédita" dos dados iria possibilitar a criação de aplicações mais inteligentes numa nova geração de dispositivos IoT, beneficiando a sociedade.

4 Metodologia de Desenvolvimento e Pesquisa

A pesquisa realizada neste trabalho consistiu de estudos e análises de trabalhos prévios, desenvolvimento de uma versão paralelizada do algoritmo de clustering k-means e experimentos sobre essa implementação. Pode-se dividir tal metodologia em um conjunto de etapas que foram realizadas.

A **primeira etapa** consistiu em uma extensa pesquisa bibliográfica. O intuito é levantar o estado da arte na área de algoritmos de agrupamento acelerados com técnicas de paralelismo, como implementações em GPU. O foco foi entender quais algoritmos já foram implementados paralelamente com sucesso, e como foram feitas tais implementações, além dos ganhos em desempenho destas. Essa etapa permitiu agregar conhecimento sobre como utilizar ferramentas como a biblioteca CUDA para acelerar algoritmos de agrupamento, além de mostrar uma prévia da magnitude de ganho de desempenho esperado de uma paralelização média desse tipo de algoritmo.

A **segunda etapa** consistiu na implementação de duas versões, uma serial e uma paralela, de um dos mais antigos e conhecidos algoritmos de agrupamento de dados: o **k-means**. A função dessa etapa da pesquisa foi aprender como programar, na prática, um algoritmo de agrupamento e, depois, como paralelizá-lo utilizando a biblioteca Python *Numba* — que utiliza a plataforma CUDA, internamente. Por ser um algoritmo mais antigo, o k-means já foi muito estudado anteriormente, tanto em versões seriais quanto paralelas, com grande presença na bibliografia da área. Assim, a implementação aqui foi facilitada pelo grande acúmulo de conhecimento bibliográfico.

A **terceira etapa** consistiu na busca de um procedimento geral para paralelizar um algoritmo de agrupamento genérico. Ou seja, o foco foi encontrar um passo-a-passo de identificação de possíveis modificações no código de um algoritmo serial que, ao fim, pudesse transformá-lo numa versão acelerada, usando CUDA, ainda mantendo sua corretude e proporcionando algum ganho significativo de desempenho.

A **quarta etapa**, por fim, consistiu em diversos experimentos de ganho de velocidade, ou *speed-up*, do k-means, que teve aqui sua versão acelerada em GPU implementada e apresentada. Os resultados desses experimentos proporcionaram uma boa visão da magnitude do ganho de desempenho ao paralelizar algoritmos de agrupamento usando CUDA, além de outros conhecimentos, como saber se há um teto ou chão para tais ganhos, como o *speed-up* aumenta ou diminui com o crescimento do número de datapoints ou variáveis no conjunto de dados a ser analisado, e também como outros parâmetros importantes que

não sejam velocidade são afetados, como a precisão dos resultados gerados pelo k-means, pois não há serventia em ganhar grande velocidade enquanto se perde o propósito de se agrupar dados em primeiro lugar, que é a capacidade de encontrar informações reais úteis em grandes datasets.

4.1 Identificando o Potencial de Paralelismo

Antes de se iniciar qualquer implementação que busque ganhar desempenho com o paralelismo, é crucial realizar uma análise para encontrar quais partes de um processo podem ser paralelizadas e quais são necessariamente seriais — isto é, precisam ser realizadas sequencialmente, sem processamentos simultâneos para aceleração.

Evidentemente, nem todo processo pode ser paralelizado. Há requisitos importantes para que uma tarefa possa ser executada de maneira paralela. Primeiro, é necessário que a tarefa possa ser subdivida em diversas outras tarefas menores. Segundo, é necessário que essas tarefas sejam independentes entre si, isto é, que a execução de uma não dependa diretamente do resultado de outra.

Um exemplo de tarefa facilmente paralelizável, como exibido nos Capítulos 2.4.2 e 2.5.1, é a soma de dois vetores. Para dois vetores de tamanho N, é possível calcular a soma deles paralelamente, calculando a soma de cada par de elementos, um de cada vetor, simultaneamente. Isso é possível pois, para dois vetores x e y, a soma dos elementos $x[i]$ e $y[i]$ não depende da soma dos elementos $x[i+1]$ e $y[i+1]$, para qualquer i.

Já um exemplo de uma tarefa inerentemente serial seria o hashing consecutivo de um dado, uma operação onde é calculado o hash de uma mensagem N vezes, cada vez aplicando a operação de hashing novamente sobre o valor gerado pela operação anterior. Para calcular a terceira operação de hash do dado, é estritamente necessário que o hash da segunda operação tenha sido calculado, que por sua vez necessita que a primeira operação de hashinh tenha sido concluída. A paralelização num cenário desses é impossível, pois o resultado de uma sub-tarefa depende de outra, impedindo que sejam executadas simultaneamente.

A identificação de trechos altamente paralelizáveis do algotimo k-means é simples. Abaixo, há um excerto do pseudocódigo que descreve seu processo, exibido por completo no Capítulo 2.2 (Algoritmo 1).

Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
   2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;

2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

É simples perceber que os trechos com maiores potenciais de paralelização são o passo 2.1 e o 2.2.

No passo 2.1, temos o cálculo de todas as distâncias euclidianas entre cada um dos N elementos do dataset e os K centroides de cada grupo. Ou seja, um mesmo cálculo é realizado um total de $N \cdot K$ vezes a cada iteração do algoritmo. Cada um dos cálculos das distâncias independe dos outros, então essa grande operação pode ser altamente paralelizada.

No passo 2.2, o cálculo dos novos centroides é realizado. Essa operação requer a redução de um vetor bidimensional de dimensões $N \cdot D$ em um vetor unidimensional de tamanho D. Essa redução é feita com a soma de todas as coordenadas dos datapoints mais próximos do centroide atualmente sendo recalculado e depois a divisão do valor final pelo número de datapoints que foram somados, realizando efetivamente uma média das coordenadas dos pontos de um grupo, para encontrar o novo centroide do grupo. Novamente, é uma operação de $(N_c - 1) \cdot D$ somas, onde N_c é o número de pontos de dados no grupo cujo centroide está sendo recalculado, com D divisões no final para encontrar a média. Por ser uma operação de redução, onde a saída tem dimensionalidade menor que a entrada, há um grau de dificuldade maior na implementação de uma versão paralela, mas é perfeitamente possível, com subdivisões iterativas dos vetores.

Na verdade, até o passo 1 no pseudocódigo acima é paralelizável, já que a geração dos centroides aleatórios pode ser feita simultaneamente para os K centróides. Porém, como K quase sempre é muito menor que N nos datasets, essa implementação costuma não valer a pena.

Uma parte do k-means que nunca poderia ser paralelizada são as operações entre cada itearação. Isto é, o passo 2.1 da teceira iteração nunca pode ser executado ao mesmo tempo que o mesmo passo 2.1 da segunda iteração do algoritmo. Isso pois a relação datapoint–centroide é passível de mudanças a cada iteração, e o cálculo das distâncias dos centroides numa iteração i depende do cálculo das distâncias da iteração $i - 1$, para qualquer $i > 1$.

4.2 Implementação do K-means

Como detalhado no capítulo 2.2, o **k-means** é um dos mais importantes e amplamente utilizados algoritmos de agrupamento, mesmo com suas diversas limitações. Ele foi selecionado nessa pesquisa como exemplo inicial de paralelização de algoritmo por ser de fácil entendimento e possuir uma implementação relativamente simples.

Foram utilizadas bibliotecas Python renomadas na área de ciência de dados, como *Numpy* e *Pandas*, para facilitar a implementação e garantir uma execução altamente otimizada, visto que essas bibliotecas implementam chamadas em C/C++ para executar suas partes mais computacionalmente custosas, garantindo uma performance superior à chamadas de alto nível (BRESSERT, 2012).

Há duas importações imprescindíveis omitidas nos códigos exibidos neste capítulo, por motivos de brevidade. Uma é a da biblioteca Numpy, importada com o nome *np*; a outra é a biblioteca Pandas, importada com o nome *pd*.

É importante entender que há também um requisito de pré-processamento do conjunto de dados para essas implementações do k-means. Todas as variáveis devem ter seus valores normalizados. Esse processo é necessário para evitar que variáveis com amplitudes de valores maiores influenciem mais significativamente a construção dos clusters do que variáveis com amplitudes menores.

É recomendado, em geral, o método de *normalização min-max* para algoritmos como o k-means, que utilizam a distância euclidiana para a construção de grupos (MILLIGAN; COOPER, 1988). Este processo é explicado detalhadamente no capítulo 5.3.

O **Algoritmo 9** abaixo é a implementação da versão serial, *single thread*, do algoritmo k-means, usada nos experimentos descritos no capítulo 5.

Algoritmo 9 – Implementação serial do k-means

```
1  def kMeansCPU(dataset:pd.DataFrame, k=3, maxIter=100):
2      centroids = pd.concat([(dataset.apply(lambda x:
           float(x.sample().iloc[0]))) for _ in range(k)], axis=1)
3      centroids_OLD = pd.DataFrame()
4
5      datasetLogs = np.log(dataset)
6
7      iteration = 1
8
9      while iteration <= maxIter and not centroids_OLD.equals(centroids):
10         distances = centroids.apply(lambda x: np.sqrt(((dataset - x) **
               2).sum(axis=1)))
11         closestCent = distances.idxmin(axis=1)
12         del distances
13
14         centroids_OLD = centroids
15         centroids = datasetLogs.groupby(closestCent).apply(lambda x:
               np.exp(x.mean())).T
16
17         iteration += 1
18
19     return closestCent
```

A função *kMeansCPU* declarada na linha 1 é uma implementação "concreta" do pseudocódigo apresentado no capítulo 2.2. Ela recebe o dataset com os dados a serem agrupados através de um dataframe do Pandas (variável *dataset*), além dos outros dois argumentos essenciais do k-means: o número de clusters k e o número de iterações máximas *maxIter*.

A primeira instrução a ser executada no algoritmo é a da linha 2, que inicializa os centroides usando valores aleatoriamente selecionados de dentro do conjunto de dados. Note que os centroides gerados não correspondem necessariamente a algum datapoint específico do dataset. Valores de qualquer datapoint podem ser selecionados e misturados para formar um centroide inicial. Isto é, pode existir um centroide inicial $C1 = (1.3, 2.3, 4.4, 3.7)$ mesmo não existindo nenhum datapoint com estes exatos valores no dataset. Em tal cenário, o valor da primeira variável poderia ter sido selecionado de um datapoint $d1 = (1.3, y1, z1, w1)$, enquanto o valor da terceira variável poderia ter originado de um datapoint $d2 = (x2, y2, 4.4, w2)$.

Após a inicialização dos centroides, são inicializadas outras variáveis importantes. Na linha 3 uma variável para armazenar os centroides da iteração anterior à atual é inicializado (*centroids_OLD*), como um dataframe vazio. Na linha 7 é inicializada uma variável de controle (*iteration*) para armazenar o índice da iteração, iniciando em 1.

Além disso, na linha 5 temos um pré-cálculo do logarítimo natural ($\ln x$) de todos os datapoints do dataset. Esses valores são usados mais adiante para facilitar o cálculo da média das variáveis de todos os datapoints de um certo cluster, isto é, na etapa de cálculo dos novos centroides a cada iteração.

Entramos então num laço de repetição (*while loop*) na linha 9, onde o agrupamento dos dados é de fato realizado, iterativamente. Todo o restante do algoritmo é realizado dentro deste loop, exceto o retorno do resultado de classificação no final. A condição de parada é testada aqui, sendo ela (1) a iteração atual, ainda a ser realizada, ser maior que o número de iterações máximas (*maxIter*) ou (2) os centroides recém calculados serem exatamente iguais aos centroides calculados na iteração anterior — o que significa que o agrupamento atual é o melhor que o algoritmo pode atingir nessa execução, ou seja, um máximo local. Como explicado no capítulo 2.2, o k-means não garante que os agrupamentos convirjam para um máximo global, apenas um máximo local.

Dentro do loop, a primeira instrução a ser executada é a de cálculo das distâncias. Isso é feito em uma só linha de código, a linha 10, utilizando largamente o poder de brevidade de código proporcionado pela biblioteca Pandas. O cálculo é realizado utilizando a distância euclidiana, aplicando a todos os centroides uma função lambda que, dado um centroide, calcula o quadrado das diferenças de todos os datapoints do dataset para este centroide. Ainda nessa função lambda, é calculada a raiz quadrada de cada resultado da operação anterior (usando a função *sqrt* da biblioteca Numpy) e, enfim, são somados os

valores obtidos para cada variável de cada datapoint, gerando um valor final de distância para cada datapoint. No final, o resultado desse cálculo é um dataframe Pandas de dimensão $n \cdot k$, que é armazenado na variável *distances*. Assim, a linha i desse dataframe possui k valores $(dC1, dC2, \ldots, dCk)$, onde dCj é a distância do datapoint i para o centroide de índice j na lista de centroides.

Em seguida, na linha 11, é feita a associação de todos os datapoints com seus centroides mais próximos, utilizando o método *idxmin* do dataframe do Pandas, que aqui retorna, para cada datapoint, o índice da coluna com o menor valor. O resultado é armazenado na variável *closestCent*, e é um dataframe de dimensão $n \cdot 1$. Nesse dataframe, a linha i possui apenas um valor (j), onde j é o índice do centroide Cj mais próximo ao ponto i.

Na linha 12 um passo opcional de exclusão explícita da variável *distances* é realizado. Isso apenas informa ao coletor de lixo do Python que a variável pode ser descartada assim que for conveniente. Como a variável não é mais usada dentro do laço de repetição, a variável pode ser deletada sem problemas, liberando mais espaço em memória.

Na linha 14, os centroides usados para os cálculos de distância na iteração atual são salvos na variável *centroids_OLD*. Essa ação é imprescindível, pois é sempre necessário comparar os centroides gerados entre a iteração atual e a anterior a cada repetição do loop.

Na linha 15 finalmente é feito o cálculo dos novos centroides, após armazenar os centroides anteriores na variável *centroids_OLD* na linha 14. Novamente, assim como a operação da linha 10, temos uma cadeia de operações do Pandas. Utilizando o método *groupby* do objeto dataframe, agrupamos os logarítimos do dataset de acordo com a lista de centroides mais próximos. Isso gera k sub-conjuntos de *datasetLogs*, cada um deles contendo apenas os logarítimos dos datapoints mais próximos a um certo centroide. Para cada um destes sub-conjuntos, é aplicada uma função lambda que, dado um sub-conjunto, calcula a média geométrica — que, como discutido em mais detalhes no capítulo 2.2, é o único método correto para encontrar a média de valores normalizados (FLEMING; WALLACE, 1986) — de todos os logarítimos dos datapoints do sub-conjunto. Essa cadeia de operações resulta em um dataframe de dimensão $d \cdot k$, onde d é o número de variáveis que descrevem cada datapoint do dataset (*features*). Esse dataframe é então transposto, acessando a propriedade T do dataframe, resultando em um novo dataframe final de dimensão $k \cdot d$, seguindo o mesmo formato usado na geração dos centroides iniciais e garantindo uma comparação correta entre os centroides de iterações consecutivas.

É importante notar que a operação supracitada do cálculo da média geométrica é realizada aqui utilizando uma operação alternativa (Equação 4.2), mas equivalente, à sua definição comum (Equação 4.1).

$$\sqrt[n]{\prod_{i=1}^{n} a_i} \tag{4.1}$$

Equação 4.1: Definição da média geométrica

$$\exp\left(\frac{1}{n} \cdot \sum_{i=1}^{n} \ln a_i\right) \tag{4.2}$$

Equação 4.2: Uma definição alternativa equivalente da média geométrica

Essa escolha foi feita pois é computacionalmente muito mais eficiente utilizar o exponencial (e^x) da média aritmética das somas dos logaritmos naturais dos n elementos ao invés de utilizar a raiz n-ésima da multiplicação dos n elementos. De fato, para datasets grandes com milhões de instâncias por cluster, seria inviável multiplicar tantos números e depois calcular a raíz n-ésima destes. Uma operação desse tipo necessitaria de quantidades exorbitantes de memória para armazenar o gigantesco resultado da multiplicação dessa miríade de termos. Utilizando a operação equivalente, é possível efetivamente "trocar" essa multiplicação por uma adição, uma operação muito menos custosa em espaço de memória, tornando o cálculo inteiro viável. Além disso, como foram pré-calculados os logarítimos naturais de todos datapoints anteriormente, na linha 5, não é necessário realizar essa operação a cada iteração, economizando mais poder computacional.

Por fim, na linha 17 do algoritmo 9, finalizamos o loop com sua última instrução, o acréscimo do valor 1 na variável de controle *iteration*, para manter a contagem de iterações correta.

Saindo do laço de repetição, o algoritmo k-means é finalizado com uma última instrução, a de retorno do resultado, armazenado em *closestCent*, na linha 19. Como explicado anteriormente, esse resultado é um dataframe do Pandas onde cada datapoint i possui apenas um valor (j), o índice do centroide Cj mais próximo a este datapoint, o que define a qual grupo ele pertence.

Analisando o código explicado acima, é fácil identificar os trechos onde são realizadas as operações mais custosas e com enorme potencial para paralelização. Para isso, basta encontrar as partes onde uma mesma operação é realizada uma quantidade imensa de vezes a cada execução e cujas operações são independentes entre si. Isso significa que a paralelização na GPU é, em tese, plausível e possivelmente acarretaria num ganho de performance. Esse processo é explicado em mais detalhes no Capítulo 4.1.

Há três trechos de código onde tal potencial de ganho de velocidade é imenso. O primeiro é onde calculamos os logarítimos naturais das coordenadas de todos os datapoints, na linha 5, replicada também àbaixo.

```
datasetLogs = np.log(dataset)
```

Essa linha de código é executada apenas uma vez no algoritmo. Porém, o cálculo realizado por ela é feito em todos os elementos do dataset. Logo, é uma operação de complexidade de tempo $O(n)$, podendo ser bem custosa para datasets com um alto número de instâncias. Como o cálculo é uma simples função matemática aplicada à cada datapoint, sem nenhum outro valor de entrada para computar o resultado, conclui-se então que cada operação é independente das outras.

O trecho abaixo é o segundo que é promissor para ganhos com a paralelização. É o da linha 10 do k-means serial, onde são calculadas as distâncias.

```
distances = centroids.apply(lambda x: np.sqrt(((dataset - x) **
    2).sum(axis=1)))
```

Aqui é feita uma operação de cálculo de distância euclidiana entre todas as n instâncias do dataset e todos os k centroides. Essa operação é realizada dentro de um laço de repetição que será executado i vezes, onde i é o número de iterações necessárias para convergir nos centroides finais. Logo, é uma operação de complexidade $O(nki)$, ainda mais custosa que a anterior. Novamente, o cálculo não depende de nenhuma informação a não ser as coordenadas de um datapoint e um centroide em particular, logo são independentes entre si e podem ser paralelizadas.

O terceiro e último trecho custoso e paralelizável abaixo é o da linha 11, onde são encontrados e salvos os índices dos centroides mais próximos, usados no cálculo dos novos centroides a cada iteração.

```
closestCent = distances.idxmin(axis=1)
```

A operação realizada aqui é uma busca feita no vetor de centroides, de tamanho k, para cada datapoint do dataset. Assim, como a linha 10 explicada anteriormente, essa também está dentro do laço de repetição das iterações do k-means. Logo, a operação inteira também possui complexidade $O(nki)$. Uma simples busca pelo menor valor no vetor de centroides é menos custosa computacionalmente que o cálculo das distâncias euclidianas, porém ainda assim há um bom potencial em se paralelizar essa busca na GPU, visto que n cresce muito ao se trabalhar com big data e ciência de dados.

Há também um quarto trecho de código que poderia ser paralelizado para obter mais performance, o da linha 15 do k-means versão serial, onde é feito o cálculo dos novos centróides através da média das coordenadas dos datapoints presentes em cada cluster.

No entanto a dificuldade de implementação de uma versão vetorial paralela dessa parte do código é bem maior que as demais. O motivo de tal complexidade é o fato da operação de cálculo da média das coordenadas de um vetor bidimensional, como é o caso da gigantesca maioria dos datasets, ser uma operação de redução. Isto é, a operação tem como entrada um vetor de tamanho $n \cdot d$ e retorna vetor de tamanho d — a média de todos

elementos para cada variável $\{var_1, var_2, \ldots, var_d\}$, onde d é o número de dimensões, ou *features* do dataset.

Esse tipo de operação de redução de vetores com mais de uma dimensão infelizmente não possui uma implementação simples correspondente na biblioteca *Numba* e implementá-la de maneira performática envolveria manipulação mais minuciosa da memória, threads e blocos da GPU, o que acabou fora do escopo deste trabalho. Porém, como exibido no capítulo 5.4, mesmo sem essa implementação, grandes ganhos de velocidade ainda puderam ser obtidos com sucesso.

Tendo em vista os supracitados trechos de código mais computacionalmente custosos e com grande potencial para paralelização, foi implementada outra versão do k-means, utilizando o alto poder de processamento vetorial da GPU.

O **Algoritmo 10** abaixo é a implementação da versão paralela, *multithread*, do algoritmo k-means, usada nos experimentos (vide capítulo 5). Note que aqui há mais três novas importações omitidas: os módulos integrados *math* e *random*, além da biblioteca *numba*.

Algoritmo 10 – Implementação paralela do k-means (função principal)

```
def kMeansGPU(dataset:pd.DataFrame, k=3, maxIter=100):
    n = len(dataset)
    d = len(dataset.iloc[0])

    randomDPIdxs = random.sample(range(n), k)
    centroids__np = np.zeros((k, d))
    for centroidIdx in range(k):
        randomDP = dataset.iloc[randomDPIdxs[centroidIdx]]
        for dimIdx in range(d): centroids__np[centroidIdx][dimIdx] =
            randomDP.iloc[dimIdx]

    centroids_OLD = pd.DataFrame()

    centroids_OLD__np = centroids_OLD.T.to_numpy()
    dataset__np = dataset.to_numpy()
    del dataset

    datasetLogs = np.zeros((n, d))
    calcLogs(dataset__np, datasetLogs)

    iteration = 1

    while iteration <= maxIter and not np.array_equal(centroids_OLD__np
        ,centroids__np):
        distances = np.zeros((n, k))
        calcDistances(centroids__np, dataset__np, distances)
```

```
25
26          closestCent = np.zeros(n, np.int64)
27          calcClosestCentroids(distances, closestCent)
28          del distances
29
30          centroids_OLD__np = centroids__np.copy()
31
32          meansByClosestCent = np.zeros((k, d))
33
34          for centroidIdx in range(k):
35              meansByClosestCent[centroidIdx] = datasetLogs[closestCent[:,]
                    == centroidIdx].mean(axis=0)
36              centroids__np[centroidIdx] =
                    np.exp(meansByClosestCent[centroidIdx])
37
38          del meansByClosestCent
39          iteration += 1
40
41      return closestCent
```

A função *kMeansGPU* acima é a função principal dessa implementação do k-means, que por sua vez chama outras três onde o processamento mais intenso é realizado paralelamente pela GPU. Essas outras funções são discutidas mais adiante (algoritmos 11, 12 e 13). A estrutura do código não muda muito em relação ao Algoritmo 9, então apenas as partes mais modificadas serão explicadas aqui.

Como aprofundado no capítulo 2.5, as funções implementadas com a biblioteca *Numba* para execução em CUDA possuem diversas limitações. A mais relevante aqui é que não é possível manipular objetos dataframe do Pandas e nem chamar seus métodos, ou sequer chamar funções dessa biblioteca. Por isso, é estritamente necessário que sejam utilizadas funções e objetos equivalentes do *Numpy* dentro das funções vetoriais que rodam em GPU.

Por este motivo, são feitas as declarações de duas variáveis importantes: n na linha 2, armazenando a quantidade de instâncias do dataset, e d na linha 3, armazenando a quantidade de variáveis que compõe cada datapoint. Ambas informações são inferidas a partir do dataframe passado à função. Além disso, são feitas também as conversões de objetos dataframe para arrays do *Numpy* nas linhas 13–15.

O processo de seleção arbitrária dos centroides iniciais também foi modificada em relação ao Algoritmo 9. Ao invés de usar uma cadeia de funções e métodos do *Pandas*, é usada a biblioteca integrada *Random* e um laço de repetição simples para escolher k datapoints aleatórios do dataset, que servirão de centroides iniciais. Esse processo é iniciado na linha 5, com o uso da função *random.sample* para selecionar k números inteiros aleatórios, mas diferentes entre si, de dentro da faixa $[0, n-1]$. Tais números são armazenados

num vetor de tamanho k chamado *randomDPIdxs*, e são usados como índices para acessar diretamente os datapoints aleatórios. Na linha 6, usando a função *np.zeros*, é inicializado o vetor de centroides, *centroids__np*, na forma de um array do *Numpy* de tamanho $k \cdot d$. Todos os valores são inicializados como zero. Então, no laço de repetição das linhas 7–9, é realizado o acesso e armazenamento dos datapoints aleatórios na variável *centroids__np*, usando o método *iloc* do *Pandas* para acesso direto e performático a estes elementos no dataset.

Note que a natureza dos centroides iniciais gerados por esse método se difere bastante da natureza dos gerados pelo método usado no Algoritmo 9. Lá, os centroides são uma mistura de valores retirados de vários datapoints, enquanto aqui os centroides são, cada um, um datapoint aleatório apenas. Essa diferença, no entanto, não tem efeito estatisticamente significativo na precisão dos resultados gerados pelas implementações, como evidenciado no Capítulo 5.5.

Na linha 17 é inicializado o primeiro objeto que será passado a uma função vetorial que será executada na GPU, o array de floats 64 bits (tipo de dado padrão do *Numpy*) *datasetLogs*, de dimensão $n \cdot d$. Note que é necessário sempre inicializar um objeto como este para servir de retorno das funções vetoriais que rodam na GPU, pois elas não podem retornar diretamente um resultado.

Na linha 18 é chamada a primeira função paralelizada, a *calcLogs* (Algoritmo 11). Esse é o passo de cálculo dos logarítimos naturais de todos os datapoints, equivalente à operação realizada na linha 5 do Algoritmo 9.

Dentro do laço de repetição *while*, onde os agrupamentos do k-means são calculados iterativamente, a primeira operação a ser feita é novamente a inicialização de um array multidimensional, *distances*, de tamanho $n \cdot k$, na linha 23, que é então passado na chamada da segunda função paralelizada, a *calcDistances*, na linha 24. Como o nome indica, essa função vetorial realiza, na GPU, o cálculo das distâncias entre todos os datapoints e os k centroides. Corresponde às excuções da linha 10 do Algoritmo 9.

Em seguida, nas linhas 26–28, é realizada a inicialização de mais um array, desta vez unidimensional, de tamanho n, chamado *closestCent* e a chamada da terceira função paralelizada, a *calcClosestCentroids*, que vai encontrar o centroide mais próximo de cada datapoint do dataset. Essa operação é equivalente às realizadas na linha 11 do Algoritmo 9. Note que ao inicializar a variável de retorno, foi necessário passar um segundo argumento para a função *np.zeros*, indicando que o array irá armazenar inteiros de 64 bits. Isso é necessário quando não se deseja usar floats de 64 bits no array. Como *closestCent* irá armazenar apenas os índices dos centroides mais próximos, faz todo sentido utilizar inteiros aqui.

Deste ponto em diante, não é feita nenhuma outra operação vetorial na GPU no

algoritmo. Isso pois o que resta são partes que não são particularmente exigentes em poder de processamento.

Na linha 30, a cópia dos centroides atuais para a variável *centroids_OLD___np* é realizada, permitindo comparar centroides calculados entre duas iterações consecutivas do k-means. Isto é realizado com o método *copy* dos arrays do *Numpy*.

A inicialização de mais um array é feita na linha 32, dessa vez *meansByClosestCent*, de dimensão $k \cdot d$, que irá armazenar, para cada centroide, as médias das coordenadas de todos os datapoints mais próximos dele — ou dos logarítimos naturais destes, mais precisamente.

Na linha 34 é iniciado um outro laço de repetição que realiza, para cada centroide, o cálculo do novo centroide respectivo que será usado na próxima iteração. Tal operação é realizada de maneira serial.

Primeiro, na linha 35, é criada uma "máscara" com a sintaxe *closestCent [:,] == centroidIdx*. Essa sintaxe gera um array booleano do *Numpy* de dimensão igual à da variável à esquerda do símbolo de igual, cujos valores são *True* se, e somente se, a comparação lógica do valor respectivo desse vetor for verdadeira. Isto é, se *closestCent* for o vetor *[0, 1, 0, 2, 1]* e o centroide atualmente sendo recalculado for o de índice 1, o vetor booleano gerado pela expressão será o *[False, True, False, False, True]*. Na prática, isso gera valores verdadeiros apenas para os índices correspondentes aos datapoints mais próximos do centroide atualmente sendo recalculado.

Aplicar essa "máscara" à variável *datasetLogs* gera um vetor apenas com os logarítimos dos datapoints pertencentes ao centroide atual. Esse processo de filtragem se assemelha ao uso do método **`filter`** disponível para objetos como listas em Python. Esse laço de repetição inteiro corresponde às operações da linha 15 no Algoritmo 9.

Ainda na linha 35 é finalmente realizado o cálculo da média dos logarítimos naturais dos datapoints mais próximos ao centroide atual, usando a função *mean* do *Numpy*. O valor é salvo na variável *meansByClosestCent*, no índice correspondente ao centroide atualmente sendo analisado.

Então, na linha 36 é realizada a operação final que gera o novo centroide, o exponencial (e^x), operação inversa ao logarítimo natural ($\ln x$), do valor calculado na linha 35. Com isso, é concluído o cálculo dos novos centroides e o laço de repetição interno é fechado.

As linhas 38 e 39 incluem apenas a marcação da variável *meansByClosestCent* para exclusão (como veio sendo feito até então com qualquer estrutura de dado que tenha se tornado desnecessária após certo ponto do código) e o acréscimo em um da variável de controle *iteration*. O laço de repetição das iterações do k-means é então fechado e, na última linha, a 41, é devolvido o resultado do agrupamento, na variável *closestCent*.

Os Algoritmos 11, 12 e 13 abaixo mostram as funções vetoriais que rodam na GPU NVIDIA, onde os cálculos mais pesados são realizados. Como discutido mais a fundo no capítulo 2.5, funções como essas, implementadas com a biblioteca *Numba* para rodar em CUDA, trabalham com um elemento de cada vez, sendo chamadas milhares de vezes e rodando paralelamente, cada chamada em um núcleo da GPU (*CUDA core*), fazendo o cálculo apenas de seu respectivo elemento ou sub-conjunto da estrutura de dados sendo processada, atingindo altíssimos níveis de paralelização. Estas funções também não podem retornar valores diretamente, necessitando ao invés de uma variável de retorno, passada como argumento. Essa variável foi definida aqui sempre como o último argumento da função.

Algoritmo 11 – Implementação paralela do cálculo dos logarítmos naturais dos datapoints

```
1 @numba.guvectorize(
2     ['void(float64[:],float64[:])'],
3     '(d)->(d)', nopython=True, target='cuda'
4 )
5 def calcLogs(rowDataset:list[np.float64], rowResults:list[np.float64]):
6     for dimIdx, dimValue in enumerate(rowDataset): rowResults[dimIdx] =
          math.log(dimValue)
```

A função *calcLogs* acima é chamada apenas uma vez em cada execução do Algoritmo 10 (na linha 14) para calcular o logarítimo de todos os datapoints do conjunto de dados.

Ela recebe um datapoint apenas (*rowDataset*) e, na linha 6, calcula o logarítimo natural ($\ln x$) de cada variável que compõe o datapoint, salvando o resultado no array de retorno (*rowResults*). A operação é realizada com a função *log* da biblioteca integrada *math*, já que a função correspondente na biblioteca *Numpy* não é suportada dentro de funções *Numba* rodando em CUDA. Nas linhas 2 e 3 são declaradas as especificações obrigatórias de tipo e dimensionalidade dos argumentos da função. Ambos são vetores unidimensionais de floats de 64 bits e de tamanho d, letra que representa a quantidade de dimensões dos datapoints.

Algoritmo 12 – Implementação paralela do cálculo de distâncias

```
1 @numba.guvectorize(
2     ['void(float64[:,:], float64[:], float64[:])'],
3     '(k,d),(d)->(k)', nopython=True, target='cuda'
4 )
5 def calcDistances(centroids:list[list[np.float64]],
      rowDataset:list[np.float64], rowResults:list[np.float64]):
6     d = len(rowDataset)
7
8     for centroidIndex, centroid in enumerate(centroids):
9         distance = 0.0
```

```
10          for dim in range(d): distance += (rowDataset[dim] - centroid[dim])
                ** 2
11          distance = distance ** (1/2)
12
13          rowResults[centroidIndex] = distance
```

A função *calcDistances* acima é chamada a cada iteração do Algoritmo 10 (na linha 20) para calcular as distâncias euclidianas entre todos os datapoints do dataset e cada um dos k centroides.

Ela recebe a lista de todos os centroides (variável *centroids*) e apenas um datapoint do dataset (variável *rowDataset*). A primeira instrução executada na função, na linha 6, é a inferência da dimensionalidade do datapoint, que é salva na variável *d*. Após isso, na linha 8, é iniciado um laço de repetição que, para cada centroide, calcula a distância dele para o datapoint sendo processado.

Esse cálculo é feito usando a fórmula generalizada da distância euclidiana, somando o quadrado das diferenças entre as coordenadas do centroide e as do datapoint e depois calculando a raiz-quadrada do valor final. Esse cálculo é feito nas linhas 9–11. Na última linha do laço de repetição, a linha 13, a distância é salva no vetor de retorno *rowResults*, no índice correspondente ao centroide atual.

Nas linhas 2 e 3 são declarados o tipo e dimensionalidade dos argumentos da função. *centroids* é um vetor multidimensional $k \cdot d$, *rowDataset* é um vetor unidimensional de tamanho d e, por fim, a variável de retorno *rowResults* é um vetor unidimensional de tamanho k. Todas as variáveis são vetores de floats de 64 bits.

Algoritmo 13 – Implementação paralela do cálculo dos centroides mais próximos

```
1  @numba.guvectorize(
2      ['void(float64[:],int64[:])'],
3      '(k)->()', nopython=True, target='cuda'
4  )
5  def calcClosestCentroids(rowDistances:list[np.float64],
       closestCent:np.int64):
6      minDistance = rowDistances[0]
7      minDistanceIndex = 0
8
9      for index, distance in enumerate(rowDistances):
10         if distance < minDistance:
11             minDistance = distance
12             minDistanceIndex = index
13
14     closestCent[0] = minDistanceIndex
```

Por fim, há a função acima, *calcClosestCentroids*, a última função vetorial chamada

pela *kMeansGPU*. A chamada é realizada a cada iteração do Algoritmo 10 (na linha 23) para encontrar o centroide mais próximo de cada datapoint do dataset.

É a mais simples das funções vetoriais usadas nessa implementação. Ela recebe (na variável *rowDistances*) apenas uma linha do vetor multidimensional retornado pelo Algoritmo 12, que contém as distâncias entre um datapoint específico e os k centroides.

Na linha 6 e 7 são inicializadas duas variáveis importantes. A primeira, *minDistance*, irá guardar a menor distância encontrada e é inicializada com o primeiro valor do vetor de distâncias; a segunda variável, *minDistanceIndex* irá guardar, como o nome indica, o índice da menor distância encontrada no vetor de distâncias, e é inicializada com o valor zero, em concordância com o centroide armazenado em *minDistance*.

Então, nas linhas 9–12 é realizada a busca pela menor distância, de maneira simplória. Um laço de repetição percorre todos os k centroides e compara a distância dele ao datapoint com a menor distância encontrada até então. Se ela for estritamente menor, a menor distância salva em *minDistance* é atualizada para corresponder ao centroide atual, assim como o índice em *minDistanceIndex*.

Ao final desse laço de repetição, teremos o índice do centroide mais próximo ao datapoint sendo analisado salvo na variável *minDistanceIndex*. Assim, ela é retornada através da variável de retorno, *closestCent*, na linha 14. Note que essa variável é acessada como se fosse um vetor, mesmo sendo na verdade escalar, com o valor salvo no índice zero. Isso é um detalhe de implementação das funções vetoriais implementadas com Numba, como explicado no capítulo 2.5.

Nas linhas 1–4 temos, como sempre, a definição de dimensionalidade e tipo das variáveis recebidas pela função. *rowDistances* é um vetor unidimensionald e tamanho k, composto por floats de 64 bits. Já a variável de retorno, *closestCent*, é um valor escalar, um inteiro de 64 bits.

5 Datasets e Experimentos

Experimentos foram realizados para averiguar os ganhos de velocidade de execução da implementação paralela em relação à implementação serial do k-means, além de testes de precisão para checar se houveram mudanças quanto à qualidade do resultado do k-means ao se paralelizá-lo.

Neste capítulo são explicados os procedimentos e metodologia de todos os experimentos, além de descrições a respeito dos conjutos de dados utilizados nestes. No capítulo seguinte (Capítulo 5.4) são exibidos e discutidos os resultados dos experimentos.

5.1 Ambiente de Execução

A máquina utilizada para realização de todos os experimentos foi um computador desktop de última geração rodando o sistema operacional **Linux** (com a distro *Endeavour OS*, baseada em *Arch*). O kernel utilizado é o *6.8.4-zen1-1-zen*, com o driver NVIDIA versão 550.67 e CUDA versão 12.4.

Portando um processador **Ryzen 7 7700X** de 8 núcleos e 16 threads, com velocidade de clock máxima de 5,4 GHz. A placa de vídeo NVIDIA utilizada, essencial para as implementações em CUDA, foi a **GeForce RTX 3070**, versão Lite Hash Rate, produzida pela Galax, com 8 GB de VRAM GDDR6, 5.888 CUDA Cores e um clock máximo de 1.730 MHz.

A máquina estava equipada também com **16 GB de memória RAM DDR5**, rodando a 5.200 MHz. Todos os componentes estando instalados numa placa mãe **MAG MSI B650M Mortar Wi-Fi**.

5.2 Testes de Desempenho e Precisão

Os testes de desempenho e precisão foram realizados usando notebooks Jupyter para fácil visualização e manipulação dos dados e resultados. Nesses notebooks, foi realizado o carregamento dos datasets, além de quaisquer pré-tratamentos de dados necessários antes de utilizá-los como entrada nas execuções do k-means.

As execuções do k-means foram realizadas também dentro de notebooks Jupyter, usando os datasets carregados anteriormente no mesmo. Cada versão do algoritmo (CPU vs. GPU) foi executado diversas vezes no mesmo dataset para que diferenças entre execuções também pudesse ser levada em consideração nos resultados — o *k-means*, especialmente, está suscetível a mudanças significativas na eficiência de cada uma de suas

execuções, pelo fato dos centroides iniciais serem selecionados aleatoriamente na implementação aqui testada. Uma execução pode ter centroides iniciais mais próximos dos centroides finais, executando mais rapidamente, enquanto outra pode ter centroides iniciais muito distantes dos centroides finais, demorando mais para atingir a condição de parada.

Cada execução teve seu tempo medido utilizando a função *perf_counter* do pacote integrado *time* do Python. Tal função permite obter um período de tempo extremamente preciso, até a resolução de nanosegundos (10^{-9} segundos), ao se subtrair o resultado de duas de suas chamadas feitas em linhas de código diferentes.

Os tempos de cada execução foram sendo exibidos ao longo dos testes, para exportação posterior dos resultados, e salvos em um acumulador, para que fosse possível calcular médias ao final de cada rodada de execuções. Os tempos mínimos e máximos de cada rodada de execuções também foi salvo para comparações.

Assim, cada dataset teve uma rodada de testes de tempo de execução feitas, uma vez usando o k-means versão serial, rodando em CPU, e outra com o k-means versão paralela, rodando em GPU. Os resultados foram salvos e são apresentados no Capítulo 5.4.

Além dos testes de ganho de velocidade, foram realizados testes de corretude, para checar por qualquer mudança na precisão dos resultados gerados pelo k-means. Esses testes funcionam através da conferência da classificação dos resultados, checados contra uma fonte de verdade existente em cada dataset.

Ao final de cada execução do k-means, na rodada de testes de corretude, era executada uma função de contagem de acertos de classificação sobre o resultado gerado. Como o nome indica, essa função checava a classificação esperada, contida no dataset, em relação à classificação resultante da execução do k-means, contando o número de acertos de classificação. A precisão do resultado era contabilizada como a razão $H \div N$, onde H é o número de acertos totais, em todas as classes, e N o número de instâncias no dataset. As porcentagens de acertos foram então comparadas entre as execuções do k-means CPU e GPU, assim como as velocidades de execução.

Todos os notebooks Jupyter com os testes, os dados não-processados dos resultados, além de muito mais material complementar, estão disponíveis no repositório do GitHub em: https://github.com/vinivosh/ufu-tcc2.

5.3 Datasets Utilizados

Foram escolhidos seis conjuntos de dados para serem utilizados nos experimentos realizados nesse trabalho. O principal critério de escolha foi o tamanho de cada dataset

(número de instâncias (N), variáveis (D) e classses (K)), além de algumas preferências como não existirem valores ausentes nos dados e nem outliers, ambos empecilhos que necessitariam de mais pré-tratamento antes de poder utilizar tais datasets no k-means.

Os datasets foram escolhidos também para representar várias ordens de magnitude de tamanhos e dimensionalidades, para que os experimentos pudessem mostrar como os ganhos de velocidade se comportam com o crescimento do volume de dados, tendo sido testados nos experimentos em ordem crescente de tamanho ($N \cdot D \cdot K$).

Uma espécie de dataset "zero" foi selecionado també para realizar as primeiras versões das implementações do k-means, tanto serial quanto paralelo. Era necessário um dataset extremamente pequeno para realizar testes simples de corretude da implementação. Esse dataset não foi usado nos experimentos finais de *speed-up* e precisão, mas foi essencial para ajudar numa implementação bem sucedida desde o início. Esse é o dataset **Iris** (FISHER, 1988).

Tabela 2 – Datasets escolhidos

Dataset	N	D	K	N*D*K	Modificação?
Rice	3.810	7	2	53.340	Não
HTRU2	17.898	8	2	286.368	Não
MiniBooNE	129.596	50	2	12.959.600	Remoção de outliers
WESAD	4.588.552	8	3	110.125.248	Sub-conjunto
HHAR	13.932.632	3	7	292.585.272	Sub-conjunto

Na Tabela 2 acima, se encontra a relação de todos os cinco datasets utilizados nos experimentos, assim como seus tamanhos e especificações sobre modificações feitas aos dados (pré-processamento).

Todos datasets foram obtidos do repositório de Machine Learning da **Universidade da Califórnia em Irvine (UCI)**. São eles o Rice (CINAR; KOKLU, 2019), HTRU2 (LYON, 2017), MiniBooNE (ROE, 2010), WESAD (LAERHOVEN, 2021) e HHAR (LYON, 2017).

Os dois maiores datasets, WESAD e HHAR, não foram utilizados por completo, mas sim sub-conjuntos dos dados disponíveis. O tamanho N exibido na Tabela 2 expressa o tamanho do sub-conjunto utilizado, e não dos datasets inteiros como disponibilizados no repositório da UCI. O mesmo se aplica para o dataset MiniBooNE, onde uma remoção de outliers foi realizada antes do processamento para evitar resultados espúrios de agrupamento.

Houve também um passo de pré-processamento em comum para todos os datasets utilizados aqui, o de normalização dos dados. Esse processo é necessário para que nenhuma das variáveis que descrevem os dados de cada dataset pudesse contribuir mais que outras dependendo de sua faixa de valores.

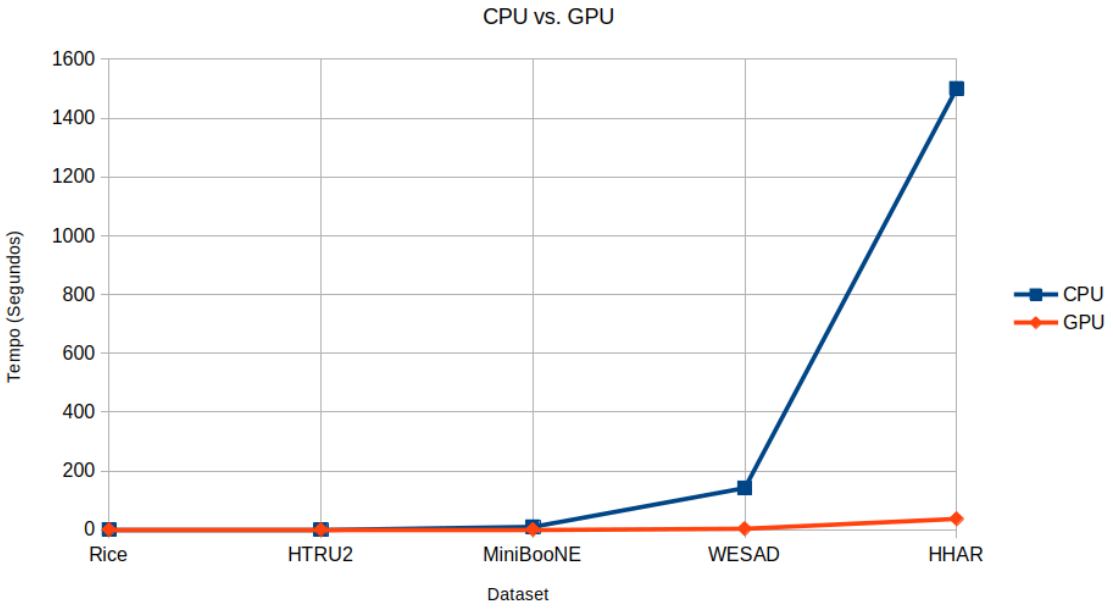
Se a normalização não fosse feita, uma variável que possuísse valores na faixa $[1.1, 2.3]$, por exemplo, iria contribuir muito menos para os cálculos de distância do k-means do que uma outra variável que possuísse valores na faixa $[4.1, 95.3]$, o que poderia gerar classificações muito piores.

O processo feito aqui em cada dataset foi o de **normalização min-max**, o processo de normalização mais recomendado (MILLIGAN; COOPER, 1988) para aplicações onde a distância euclidiana é importante, como é o caso do k-means.

5.4 Ganhos de Velocidade

Utilizando os cinco datasets detalhados anteriormente, foram realizadas diversas rodadas de execuções em ambas versões dos algoritmos aqui analisados. Seus tempos de execução e precisão na classificação foram comparados.

Figura 5 – Tempo médio de execução do k-means



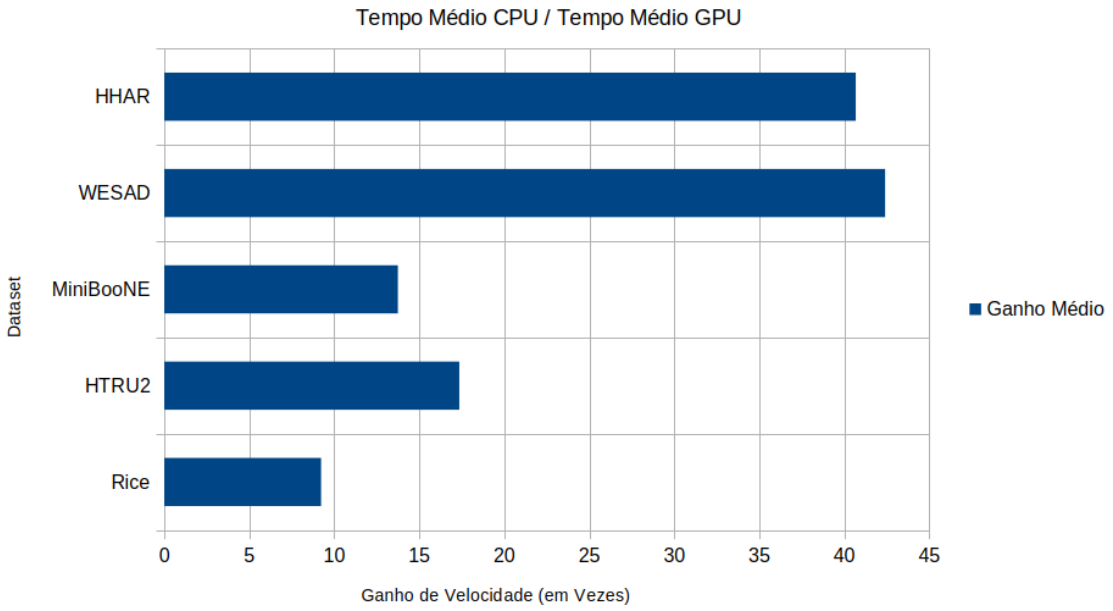
Na Figura 5 e na Tabela 3 é visível como o tempo de execução cresce muito mais rapidamente com o aumento dos tamanhos dos datasets nas execuções do k-means versão CPU em relação às execuções da versão paralela rodando em GPU. Enquanto no maior dataset, HHAR, com mais de 13 milhões de instâncias, o tempo de execução do k-means CPU beira algo probitivo, na casa dos 23 minutos para uma execução média, a versão paralela do k-means ainda não chega a demorar mais de 40 segundos para ser executada.

Os fatores de *speed-up* são mais claros ainda no gráfico de barras da Figura 6 abaixo. O padrão esperado foi de fato experimentalmente encontrado. O ganho de velocidade fica cada vez melhor com datasets cada vez maiores.

Tabela 3 – Velocidade Média de Execução do K-means

Dataset	Execuções	Tempo Médio CPU (S)	Tempo Médio GPU (S)
Rice	100	0.0800	0.0087
HTRU2	100	0.3327	0.0192
MiniBooNE	100	9.6021	0.6988
WESAD	20	141.7909	3.3438
HHAR	10	1500.6155	36.8901

Figura 6 – Ganho médio de velocidade do k-means



Esse resultado corrobora a análise de que, para datasets menores, o esforço de se paralelizar um algoritmo de agrupamento vale muito menos a pena, pois o custo da parte inerentemente serial do algoritmo pode acabar dominando o custo total das execuções, fazendo com que sua melhora máxima (os ganhos nas partes paralelizáveis) se torne menos significativa. Isso remete também à **Lei de Amdahl** (RODGERS, 1985), que postula sobre ganhos máximos possíveis com a paralelização.

A magnitude dos ganhos de velocidade encontrados nos experimentos também condiz com a magnitude encontrada nos trabalhos estado da arte que, como exibidos no Capítulo 3, estavam na faixa de 2 a 80 vezes de ganho de velocidade no caso específico do k-means.

5.5 Precisão

Além dos resultados positivos de ganhos de velocidade, foram obtidos resultados igualmente positivos quanto à precisão dos resultados gerados pelo k-means versão paralela, em relação ao k-means versão serial.

Manter a precisão média num nível extremamente parecido entre as duas implementações era de suma importância para a hipótese desse trabalho. Isso pois não haveria sentido em acelerar drasticamente um algoritmo de agrupamento de dados se esse processo piorasse a precisão dos resultados, indo contra o propósito da utilidade prática do algoritmo.

Tabela 4 – Comparação de Precisão

Dataset	**Execuções**	**P. Média (CPU)**	**P. Média (GPU)**	**Melhora**
Rice	100	91.0433%	91.3821%	+0.372180
HTRU2	100	91.7588%	91.7589%	+0.000061
MiniBooNE	50	50.8039%	50.8288%	+0.048967
WESAD	5	63.5576%	65.0151%	+2.293257
HHAR	2	28.3104%	28.3097%	-0.002485

A Tabela 4 acima mostra detalhadamente a precisão de cada versão do k-means implementada, além da melhora de precisão obtida ao se subtrair a precisão da versão GPU pela a da versão CPU.

Como é possível perceber, em todos os casos menos o último houve uma melhora de precisão. Todas as diferenças de precisão foram de magnitude bem pequena, de menos de 0,37 unidades de diferença nas porcentagens em todos os casos, exceto no databse WESAD, onde houve uma diferença de 2,29 unidades na porcentagem, porém melhorando a precisão, e não piorando. A única piora de precisão foi no maior dataaset, o HHAR, e foi de apenas 0,002485 unidades na porcentagem de acerto.

Diferenças de acertos de classificação tão pequenas como essa indicam que a paralelização do algoritmo não afetou significativamente a precisão do k-means, um resultado positivo que reforça a viabilidade desse tipo de implementação. Não só o ganho de velocidade pode ser grande, como mostrado na capítulo anterior, como a precisão do algoritmo não é prejudicada por isto.

As poucas variações de corretude são tão pequenas que podem ser explicadas pelas diferenças entre os centroides selecionados inicialmente no algoritmo, ao invés de alguma diferença causada pela paralelização em si.

É importante notar também que o valor absoluto da precisão do k-means caiu bastante nos três maiores datasets (MiniBooNE, WESAD e HHAR), saindo da casa dos 90% para a faixa dos 28 à 65%. Isso ocorre pelo motivo da natureza dos dados desses datasets. Todos eles incluem dados temporais (*time series data*), que não costumam ser bem classificados por algoritmos como esse de qualquer maneira. Eles ainda foram usados para a pesquisa simplesmente pela grande dimensionalidade deles.

6 Conclusões e Trabalhos Futuros

Após as implementações e os resultados obtidos dos diversos experimentos feitos em cima delas, pode-se inferir que o objetivo desse estudo foi alcançado com sucesso. Não apenas foram obtidos ganhos de velocidade de mesma magnitude que as observadas no estado da arte da bibliografia a respeito da otimização do k-means através do paralelismo, mas também foi possível alcançar tudo isso utilizando ferramentas de mais alto-nível de abstração: a linguagem Python juntamente com a biblioteca *Numba*.

Como exemplificado nos Capítulos 2.4 e 2.5, implementações com ferramentas como essa facilitam muito o desenvolvimento de algoritmos paralelos a serem executados na GPU, aumentando o poder de toda a comunidade científica, principalmente em áreas como aprendizado de máquina e a ciência de dados, onde o uso da linguagem Python e as diversas ferramentas desenvolvidas para ela já é bem estabelecido.

O acesso facilitado ao poder do parelelismo das GPUs NVIDIA proporcionado pela bibliteca e compilador *Numba* se torna, assim, mais uma grande adição ao arsenal de ferramentas do cientista de dados e pesquisador, além de ser uma ótima porta de entrada para o aprendizado de técnicas de computação paralela em geral.

Também foi constatado que a paralelização do k-means como implementada nessa pesquisa não causa pioras mensuráveis na qualidade dos resultados devolvidos pelo processo, apresentando assim apenas vantagens para o uso das GPUs na aceleração desse notório e onipresente algoritmo de agrupamento de dados.

Além disso, foi possível extrair grande conhecimento das implementações e experimentos realizados na pesquisa e sintetizadas nessa monografia. Acredita-se que o trabalho vá ser de grande utilidade para pesquisadores futuros, como uma introdução às implementações paralelas usando ferramentas de programação de alto-nível, auxiliando no desenvolvimento de versões paralelizadas de novas variações de algoritmos de agrupamento de dados, ou até mesmo de algoritmos completamente novos que possam vir a existir no futuro.

Ainda no assunto do futuro, há diversas recomendações a serem feitas para expandir o trabalho dessa pesquisa futuramente. Como comentado no capítulo 4.2 (após o Algoritmo 9), o trecho específico do k-means que recalcula os centroides a cada iteração, através da média dos pontos de dados pertencentes a cada grupo, não pode ser paralelizado utilizando o decorador *guvectorize* da biblioteca *Numba*, como foi feito com os outros trechos paralelizáveis do algoritmo.

Uma implementação paralela bem sucedida desse trecho, quer requerirá ferramen-

tas mais avançadas do *Numba*, acarretaria em ganhos de velocidade de execução ainda maiores que os atingidos nessa pesquisa.

Além disso, há uma gama de outras ferramentas implementadas na biblioteca que podem servir para impulsionar ainda mais a velocidade do k-means, ou qualquer outro algoritmo de agrupamento de dados, como gerenciamento mais explícito da memória da GPU, controle direto da geometria de blocos e threads CUDA, entra outras. Todas essas são avenidas de melhorias que outras pesquisas futuras podem explorar em implementações mais avançadas.

No fim, o trabalho aqui pode ser expandido, também, com estudos e implementações de outros algoritmos notórios de agrupamento de dados, como abordagens hierárquicas ou baseadas em densidade, utilizando de ferramentas de alto nível como as usadas aqui para unir o melhor da performance com a facilidade de implementação, e sintaxes mais concisas e elegantes.

Referências

Anaconda, Inc. **Numba: A High Performance Python Compiler**. 2024. <https://numba.pydata.org>. [Online; acessado 15 de abril de 2024]. Citado na página 35.

ARTHUR, D.; VASSILVITSKII, S. K-means++: The advantages of careful seeding. In: . [s.n.], 2007. v. 8, p. 1027–1035. Disponível em: <https://theory.stanford.edu/~sergei/papers/kMeansPP-soda.pdf>. Citado na página 17.

BOCK, H.-H. Clustering methods: A history of k-means algorithms. In: _____. **Selected Contributions in Data Analysis and Classification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 161–172. ISBN 978-3-540-73560-1. Disponível em: <https://doi.org/10.1007/978-3-540-73560-1_15>. Citado na página 16.

BRESSERT, E. **SciPy and NumPy: An Overview for Developers**. Sebastopol, CA: O'Reilly Media, 2012. ISBN 9781449305468. Disponível em: <https://books.google.com.br/books?hl=en&lr=&id=c-xzkDMDev0C&oi=fnd&pg=PR2&dq=numpy+faster&ots=Z8PMvUodzc&sig=OS-ctvRZnfUbtG1wUvHtF3qZFso&redir_esc=y#v=onepage&q=fast&f=false>. Citado na página 52.

CECILIA, J. M.; CANO, J.-C.; MORALES-GARCÍA, J.; LLANES, A.; IMBERNÓN, B. Evaluation of clustering algorithms on GPU-based edge computing platforms. **Sensors (Basel)**, MDPI AG, v. 20, p. 6335, nov 2020. Disponível em: <https://www.mdpi.com/1424-8220/20/21/6335>. Citado na página 47.

CINAR, I.; KOKLU, M. **Rice** *CammeoandOsmancik*. 2019. UCI Machine Learning Repository. Disponível em: <https://doi.org/10.24432/C5MW4Z>. Citado na página 66.

COOK, S. A. The complexity of theorem-proving procedures. In: **Proceedings of the Third Annual ACM Symposium on Theory of Computing**. New York, NY, USA: Association for Computing Machinery, 1971. (STOC '71), p. 151–158. ISBN 9781450374644. Disponível em: <https://doi.org/10.1145/800157.805047>. Citado na página 9.

CUOMO, S.; De Angelis, V.; FARINA, G.; MARCELLINO, L.; TORALDO, G. A GPU-accelerated parallel K-means algorithm. **Computers and Electrical Engineering**, v. 75, p. 262–274, 2019. ISSN 0045-7906. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0045790617327994>. Citado na página 10.

ESTIVILL-CASTRO, V. Why so many clustering algorithms. **ACM SIGKDD Explorations Newsletter**, ACM, v. 4, n. 1, p. 65–75, jun 2002. ISSN 19310145. Disponível em: <http://portal.acm.org/citation.cfm?doid=568574.568575>. Citado na página 14.

FISHER, R. A. **Iris**. 1988. UCI Machine Learning Repository. Disponível em: <https://doi.org/10.24432/C56C76>. Citado na página 66.

FLEMING, P. J.; WALLACE, J. J. How not to lie with statistics: the correct way to summarize benchmark results. **Commun. ACM**, Association for Computing Machinery (ACM), v. 29, n. 3, p. 218–221, mar 1986. Disponível em: <https://doi.org/10.1145/5666.5673>. Citado na página 54.

FORGY, E. W. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. **Biometrics**, v. 21, p. 768–769, 1965. Disponível em: <https://api.semanticscholar.org/CorpusID:118110564>. Citado na página 17.

HALL, J. D.; HART, J. C. Gpu acceleration of iterative clustering. jan 2004. Disponível em: <https://www.researchgate.net/profile/John-Hart-17/publication/250176198_GPU_Acceleration_of_Iterative_Clustering/links/5447f8160cf2d62c30529d02/GPU-Acceleration-of-Iterative-Clustering.pdf>. Citado na página 43.

HAN, J.; KAMBER, M.; PEI, J. **Data Mining: Concepts and Techniques**. Elsevier, 2012. xxiii p. ISBN 9780123814791. Disponível em: <https://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-P-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf>. Citado na página 40.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 5. ed. Oxford, England: Morgan Kaufmann, 2011. 288–315 p. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 012383872X. Disponível em: <https://dl.acm.org/doi/pdf/10.5555/1999263>. Citado na página 26.

JOSHI, M. N. Parallel k-means algorithm on distributed memory multiprocessors. In: . [s.n.], 2003. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5598bf67cef4f2727da0b0b94ba085349c7c45e7>. Citado na página 42.

KANTABUTRA, S.; COUCH, A. Parallel k-means clustering algorithm on nows. **NOCTEC Technical Journal**, v. 1, 01 2000. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6197ce32309824420aac79f975e028e440c2be96>. Citado na página 41.

KARA, M. E.; FIRAT, S. U. O. Supplier risk assessment based on best-worst method and k-means clustering: A case study. **Sustainability**, v. 10, n. 4, 2018. ISSN 2071-1050. Disponível em: <https://www.mdpi.com/2071-1050/10/4/1066>. Citado na página 20.

LAERHOVEN, K. V. WESAD: Multimodal Dataset for Wearable Stress and Affect Detection. **F. Wolling**, F. Wolling, mar. 2021. Disponível em: <https://www.eti.uni-siegen.de/ubicomp/home/datasets/icmi18/index.html.en?lang=en>. Citado na página 66.

LEVIN, L. A. **Universal Sequential Search Problems**. 1973. 265–266 p. [Online; acessado 29 de abril de 2024]. Disponível em: <https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=914&option_lang=eng>. Citado na página 9.

LLOYD, S. P. Least squares quantization in pcm. **IEEE Transactions on Information Theory**, v. 28, n. 2, p. 129–137, mar 1982. ISSN 1557-9654. Disponível em: <https://doi.org/10.1109%2FTIT.1982.1056489>. Citado 2 vezes nas páginas 10 e 17.

LYON, R. **HTRU2**. 2017. UCI Machine Learning Repository. Disponível em: <https://doi.org/10.24432/C5DK6R>. Citado na página 66.

MACQUEEN, J. **Some methods for classification and analysis of multivariate observations**. 1967. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967). Disponível em: <https://projecteuclid.org/ebooks/berkeley-symposium-on-mathematical-statistics-and-probability/Some-methods-for-classification-and-analysis-of-multivariate-observations/chapter/Some-methods-for-classification-and-analysis-of-multivariate-observations/bsmsp/1200512992>. Citado 2 vezes nas páginas 10 e 17.

Mark Harris. **An Even Easier Introduction to CUDA**. 2017. <https://developer.nvidia.com/blog/even-easier-introduction-cuda>. [Online; acessado 15 de abril de 2024]. Citado na página 29.

MILLIGAN, G. W.; COOPER, M. C. A study of standardization of variables in cluster analysis. **Journal of Classification**, v. 5, n. 2, p. 181–204, Sep 1988. ISSN 1432-1343. Disponível em: <https://doi.org/10.1007/BF01897163>. Citado 2 vezes nas páginas 52 e 67.

NVIDIA Corporation. **CUDA Zone**. 2018. Disponível em: <https://developer.nvidia.com/cuda-zone>. Citado na página 10.

_____. **CUDA Toolkit**. 2024. <https://developer.nvidia.com/cuda-toolkit>. Online; acessado 15 de Abril de 2024. Citado na página 29.

_____. **NVIDIA Nsight Systems**. 2024. <https://developer.nvidia.com/nsight-systems>. Online; acessado 15 de abril de 2024. Citado na página 29.

PRAHARA, A.; ISMI, D.; AZHARI, A. Parallelization of partitioning around medoids (pam) in k-medoids clustering on gpu. **Knowledge Engineering and Data Science**, v. 3, p. 40–49, 08 2020. Disponível em: <https://www.researchgate.net/publication/343722407_Parallelization_of_Partitioning_Around_Medoids_PAM_in_K-Medoids_Clustering_on_GPU>. Citado na página 44.

RODGERS, D. P. Improvements in Multiprocessor System Design. **Conference Proceedings - Annual Symposium on Computer Architecture**, ACM, New York, NY, USA, v. 13, n. 3, p. 225–231, 1985. ISSN 01497111. Disponível em: <http://doi.acm.org/10.1145/327070.327215>. Citado 2 vezes nas páginas 10 e 68.

ROE, B. **MiniBooNE particle identification**. 2010. UCI Machine Learning Repository. Disponível em: <https://doi.org/10.24432/C5QC87>. Citado na página 66.

ROVERE, M.; CHEN, Z.; PILATO, A. D.; PANTALEO, F.; SEEZ, C. CLUE: A fast parallel clustering algorithm for high granularity calorimeters in high-energy physics. **Frontiers in Big Data**, Frontiers Media SA, v. 3, nov 2020. Disponível em: <https://www.frontiersin.org/articles/10.3389/fdata.2020.591315/full>. Citado na página 46.

SANDERS, J.; KANDROT, E. **CUDA by Example**. Boston, MA: Addison-Wesley Educational, 2010. 6–11 p. ISBN 978-0-13-138768-3. Disponível em: <https://edoras.sdsu.edu/~mthomas/docs/cuda/cuda_by_example.book.pdf>. Citado na página 27.

SciKit-Learn Developers. **Clustering text documents using k-means**. 2024. <https://scikit-learn.org/stable/auto_examples/text/plot_document_clustering.html>. [Online; acessado 17 de abril de 2024]. Citado na página 21.

_____. **Comparing different clustering algorithms on toy datasets**. 2024. <https://scikit-learn.org/0.18/auto_examples/cluster/plot_cluster_comparison.html>. [Online; acessado 29 de abril de 2024]. Citado na página 15.

STEINHAUS, H. Sur la division des corps matériels en parties. **Bulletin de l'Académie Polonaise des Sciences, Classe 3**, v. 4, p. 801–804, 1957. ISSN 0001-4095. Disponível em: <https://zbmath.org/?format=complete&q=an:0079.16403>. Citado na página 17.

XU, R.; WUNSCH, D. Survey of clustering algorithms. **IEEE Transactions on Neural Networks**, v. 16, n. 3, p. 645–678, 2005. Disponível em: <https://axon.cs.byu.edu/Dan/678/papers/Cluster/Xu.pdf>. Citado na página 18.

YANG, M. S. A survey of fuzzy clustering. **Mathematical and Computer Modelling**, Pergamon, v. 18, n. 11, p. 1–16, dec 1993. ISSN 08957177. Disponível em: <https://www.sciencedirect.com/science/article/pii/089571779390202A>. Citado na página 15.

YU, S.; WANG, Y.; GU, Y.; DHULIPALA, L.; SHUN, J. Parchain: A framework for parallel hierarchical agglomerative clustering using nearest-neighbor chain. **CoRR**, abs/2106.04727, jun 2021. Disponível em: <https://arxiv.org/abs/2106.04727>. Citado na página 46.