UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Vinícius Henrique Almeida Praxedes

Paralelização de Algoritmos Notórios de Agrupamento de Dados em GPUs NVIDIA

Uberlândia, Brasil

2023, Novembro

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Vinícius Henrique Almeida Praxedes

Paralelização de Algoritmos Notórios de Agrupamento de Dados em GPUs NVIDIA

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

2023, Novembro

Vinícius Henrique Almeida Praxedes

Paralelização de Algoritmos Notórios de Agrupamento de Dados em GPUs NVIDIA

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

Professor

Professor

Uberlândia, Brasil
2023, Novembro

Resumo

TODO: Escrever o resumo após terminar a monografia. Segundo a ABNT (2003, 3.1-3.2), o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: Até, cinco, palavras-chave, separadas, por, vírgulas.

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 de Gráficos. 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
CUDA	[inserir informação]

Sumário

1 Introdução

A busca pelo menor tempo de execução é uma diretriz ubíqua na computação. Desde os primórdios da área buscamos algoritmos e procedimentos que, dados os mesmos parâmetros de entrada, executem a mesma tarefa na menor quantidade 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.

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, GPUs possuem dezenas, 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.

E é a paralelização de uma classe de algoritmos em particular que 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 os 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 linguagem 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 de pesquisa é o notório algoritmo de agrupamento *K-means* (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 vasto conhecimento de como paralelizar esses algoritmos em geral, analisando principalmente os dois aqui estudados a fundo (K-Means e Agrupamento Hierárquico) e usando este aprendizado para criar um passo-a-passo genérico 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 dos dois algoritmos aqui estudados, a fim de adquirir conhecimento de como a paralelização em CUDA deve ser realizada;
- Paralelizar um algoritmo de clustering "novo", isto é, nunca antes paralelizado e exibido em trabalho científico, a fim de solidificar o conhecimento e prática de programação em CUDA. Foi escolhido o algoritmo de Agrupamento Hierárquico para tal;
- Quantificar o ganho de desempenho das implementações paralelas, realizando diversos experimentos de *speedup*, usando diversos datasets de tamanhos e dimensionalidades variadas;
- Comparar o código serial (sem paralelização) com o código paralelo dos dois algoritmos analisados, extraindo assim um conhecimento 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 unidimensionais, 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 genérico para paralelizar qualquer algoritmo de agrupamento será de extrema 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.

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 dos dois algoritmos pesquisados. Veremos que a complexidade de tempo desses algoritmos 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: **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; **modelos de densidade**, que definem grupos como regiões densas e conexas no espaço, contrastando com regiões menos densas que separam os grupos; **modelos de conectividade**, que constroem grupos a partir de conexões de datapoints definidas por um limiar de distância; **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.

2.2 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.

2.2.1 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.

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 (FLYNN, 1972). 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.

Com o surgimento das GPUs, inicialmente desenvolvidas para 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 emblemático da aplicação da programação vetorial em GPUs NVIDIA é o algoritmo de agrupamento de dados conhecido como k-means. O k-means é amplamente utilizado em análise de dados e mineração de dados para agrupar pontos de dados em clusters com base em características semelhantes. A paralelização deste algoritmo em GPUs NVIDIA pode resultar em um significativo aumento de desempenho, permitindo o processamento rápido de grandes conjuntos de dados (Shane, 2012).

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.

2.2.2 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. 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.

2.2.3 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. 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* e o *Hierarchical Clustering*. 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.

2.3 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.

2.3.1 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 processa-

mento 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.

2.3.2 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 1 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 1 – Implementação serial de soma de vetores em C++

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

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

int main(void){
  int N = 1<<28; // 268.435.456 elementos

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

  // Inicializar vetores no host
  for (int 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 (int 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 perceber 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 **308 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 2 abaixo apresenta o programa inteiro, em sua versão vetorial na GPU. Essa implementação, no entanto, é consideravelmente ingênua por motivos explicitados a seguir.

Algoritmo 2 – Implementação vetorial (ingênua) de soma de vetores usando CUDA

```
1  #include <iostream>
2  #include <math.h>
3
4  __global__
5  void add(int n, float *x, float *y) {
6    int index = threadIdx.x;
7    int stride = blockDim.x;
8
```

```
9   for (int i = index; i < n; i += stride)
10    y[i] += x[i];
11 }
12
13 int main(void){
14   int N = 1<<28; // 268.435.456 elementos
15
16   float *x, *y;
17   cudaMallocManaged(&x, N*sizeof(float));
18   cudaMallocManaged(&y, N*sizeof(float));
19
20   for (int 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 (int 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 1 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 argmento 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»>* enter o nome da função e o parêntesis 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 2 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 **93 milissegundos**. O erro máximo também é zero. Houve uma melhora de velocidade de execução (*speed-up*) de cerca de 3,31 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 2. Os trechos de código modificado seguem abaixo.

```
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 2). 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 ($N = 2^{28}$), haverão exatamente 262.144 *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.

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

O trecho acima (que corresponde às linhas 4–11 do Algoritmo 2), 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 **11 milissegundos**. O erro máximo continua em zero. Houve agora um *speed-up* de cerca de 8,45 vezes em relação à versão vetorial anterior (Algoritmo 2) e de exatamente 28 vezes em relação à versão serial rodando em CPU (Algoritmo 1).

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.

2.4 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.

2.4.1 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, extremamente semelhante ao usado para demonstrar o uso da API CUDA em C++ no Capítulo 2.3.2, foi implementado, primeiro serialmente, e depois paralelamente, utilizando a GPU.

O Algoritmo 3 abaixo executa a adição de dois vetores, retornando o resultado num terceiro vetor novo. No entanto, os vetores aqui, diferentemente dos exemplos em C++ anteriores, são bidimensionais. Note que foram usadas, 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 3 – Implementação serial de adição de vetores 2D em Python

```
def addArrayCPU(a, b):
    return a + b

def checkMaxErr(c):
    # Checando erro máximo (todos elementos devem ser 42.0):
    minRow = c.min(axis=0)
    maxRow = c.max(axis=0)
    maxErr = 0.0
    for dIdx in range(D):
        maxErr = max(maxErr, abs(42.0 - minRow[dIdx]))
        maxErr = max(maxErr, abs(42.0 - maxRow[dIdx]))
    print(f'Max error: {maxErr}')

D = 2**2
N = int(2**28 * 1.5) // D # N * D = 402.653.184 elementos

# Inicializando vetores
a = np.full((N, D), 27.2, np.float32)
b = np.full((N, D), 14.8, np.float32)
```

```
20
21 # Realizando adição
22 c = addArrayCPU(a, b)
23
24 checkMaxErr(c)
```

O código acima é auto-explicativo e está comentado para melhor compreensão. Se assemelha bastante com o Algoritmo 1 exibido no Capítulo 2.3.2. As diferenças práticas são o tamanho dos vetores. Aqui, eles possuem a dimensões $N \cdot D$, onde $N = 100.663.296$ e $D = 4$, totalizando 402.653.184 elementos. Isso são exatamente 50% a mais elementos do que havia no exemplo em C++.

Rodando o código e realizando a medição da velocidade de execução da função *addArrayCPU*, encontra-se o tempo médio em 15 execuções seguidas de cerca de 1769,42 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.

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

```
@numba.guvectorize(
    ['void(float32[:],float32[:],float32[:])'],
    '(d),(d)->(d)', nopython=True, target='cuda'
)
def addArrayGPU(a, b, c):
```

```
d = len(a)
for dIdx in range(d):
    c[dIdx] = a[dIdx] + b[dIdx]
```

Com apenas essa modificação a mais o código está pronto para ser executado paralelamente pela GPU.

Porém, o funcionamento da função não é tão intuitivo. Como é possível ver nas três linhas do corpo da função, ela trabalha com apenas um elemento do vetor — uma linha, se imaginando o vetor bidimensional inteiro como uma tabela. 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 iterar sobre as D variáveis de um elemento dos vetores "externos" (com N elementos), somando os valores do elemento a ao b e armazenando 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 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 apenas um elemento da estrutura de dados maior que de fato é que está sendo processada, como foi feito aqui, 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 aqui 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 vetores unidimensionas contendo números de ponto flutuante (*float*) de precisão simples.

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, *(d),(d)->(d)* indica que há dois vetores de entrada, ambos de tamanho d, e um de saída, também de tamanho d.

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 dado inteira a ser processada. Como a função foi definida para receber um vetor unidimensional de tamanho d, quando ela é de fato chamada e um vetor bidimensional de tamanho $n \cdot d$ é passado, o compilador do Numba infere que devem ser criadas n instâncias da função, e cada uma delas processará apenas um vetor de tamanho d diferente, paralelizando automaticamente o processo.

O terceiro argumento, de nome *nopython*, deve ser definida 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 398,35 milissegundos. O erro máximo também é zero, como esperado. Foi obtido um *speed-up* de cerca de 4,44 vezes em relação à versão serial implementada usando apenas funções do *Numpy*.

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.

2.5 K-Means

TODO: Escrever uma sub-seção sobre o K-Means.

- **Um dos algoritmos mais tradicionais, explicar a história dele. Citar os primeiros trabalhos introduzindo o algoritmo;**
- **Explicar brevemente seu funcionamento (talvez com um pseudo-código bem alto nível? Diagramas??);**
- **Explicar como ele é usado (aplicações reais).**

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.

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 demons-

trado 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 2.2.2), 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

TODO: Parágrafo(s) introdutório(s) básicos aqui.

TODO: Citar aqui os trabalhos mais antigos que encontrar! Citar, necessariamente:

- **Um dos trabalhos mais antigos que tentaram paralelizar um algoritmo de agrupamento de dados (seja por CPU, GPU, TPU, etc. até paralelização em nível de instrução da CPU tá valendo). Imagino que seria no máximo de 2005 esse trabalho.**

  Candidato: "Parallel K-means Clustering Algorithm on NOWs", de 2000. Link.

  Candidato: "Parallel K - Means Algorithm on Distributed Memory Multiprocessors", de 2003. Link.

- **Um dos trabalhos mais antigos que tenha parelelizado especificamente em GPU um algoritmo de agrupamento de dados. Esse muito provavelmente**

vai ser de 2007 a 2011, com o lançamento e melhoramento do CUDA da NVIDIA

Candidato: "GPU Acceleration of Iterative Clustering", de 2004. Link.

- **Um dos primeiros trabalhos que tentaram paralelizar especificamente o k-means, seja por CPU, GPU, etc. Se esse for, ao mesmo tempo, algum dos anteriores, tá ótimo também**

  Candidato: Vide segundo item.

3.2 K-Means e Variantes

No estudo "*Parallelization of Partitioning Around Medoids* [...]" (PRAHARA; ISMI; AZHARI, 2020), 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 capacidade 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

TODO: Escrever um parágrafo introdutorio básico aqui?

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 **plata-**

formas 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

TODO: Adicionar em algum lugar desse capítulo uma figura, um diagrama, explicativo sobre a metodologia usada para parelelizar um algoritmo de agrupamento genérico. Vide arquivo "..\Notes_230922_171133.pdf".

A pesquisa realizada neste trabalho consistiu de estudos e análises de trabalhos prévios, desenvolvimento de versões paralelizadas de dois algoritmos de clustering e experimentos sobre essas implementações. Pode-se dividir tal metodologia em um conjunto de etapas que foram realizadas.

TODO: Re-escrever grande parte dessa descrição de etapas do trabalho para condizer com a pesquisa feita somente a respeito do K-Means...

A **primeira etapa** consistiu em uma extensa pesquisa bibliográfica. O intuito é levantar o estado da arte na área de algoritmos de agrupamento acelerados em GPU usando a linguagem e biblioteca CUDA da NVIDIA. O foco foi entender quais algoritmos já foram implementados com sucesso em CUDA, e como foram feitas tais implementações, além dos ganhos em desempenho destas. Essa etapa permitiu agregar conhecimento sobre como utilizar 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 realizada foi feita puramente para fins de aprendizado. Os ganhos de velocidade da versão paralela foi então comparada também com os ganhos obtidos nos trabalhos analisados na primeira etapa. Isso serviu como uma validação da corretude da implementação feita.

A **terceira etapa** consistiu na implementação de uma versão paralela um pouco mais "inédita" de algum algoritmo de agrupamento, ou seja, um algoritmo cuja implementação paralela foi raramente estudada à fundo em pesquisas. O algoritmo escolhido para essa etapa foi o de **Agrupamento Hierárquico**. Usando o aprendizado adquirido nas etapas anteriores, este algoritmo foi paralelizado em Python, utilizando novamente

a biblioteca *Numba*, e seus resultados comparados com a versão serial (rodando somente em uma thread, na CPU) para garantir corretude.

A **quarta 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 modificações no código de um algoritmo serial que, ao fim, o transforme numa versão acelerada usando CUDA desse mesmo algoritmo, ainda mantendo sua corretude e proporcionando algum ganho significativo de desempenho.

A **quinta etapa**, por fim, consistiu em diversos experimentos de ganho de velocidade, ou *speedup*, dos dois algoritmos que tiveram aqui sua versões aceleradas em GPU implementadas e apresentadas. 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 *speedup* 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 o uso de memória — afinal, a VRAM das GPUs são comumente mais limitadas em capacidade do que a RAM utilizada pelas CPUs.

4.1 Análise de Potencial de Paralelização

TODO: Escrever uma seção aqui explicando bem como é feita a identificação das partes paralelizáveis de um algoritmo, especialmente um de clustering. Incluir:

- **Parágrafos introdutórios;**
- **Exemplo de identificação num algoritmo toy (não-clustering);**
- **Exemplo de identificação no pseudocódigo do K-Means;**
- **Exemplo de identificação no pseudocódigo do Hierarquichal Clustering.**

4.2 Implementação do K-Means

Como detalhado no capítulo 2.5, 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 4** abaixo é a implementação da versão serial, *single thread*, do algoritmo k-means, usada nos experimentos descritos no capítulo 5.

Algoritmo 4 – 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 2.5. 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 2.5, 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 2.5, 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, mas equivalente, à sua definição:

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

Sendo a operação equivalente:

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

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 4, 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 6.1, 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 5** 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 5 – Implementação paralela do K-Means (Função Principal)

```
1  def kMeansGPU(dataset:pd.DataFrame, k=3, maxIter=100):
2      n = len(dataset)
3      d = len(dataset.iloc[0])
4
5      randomDPIdxs = random.sample(range(n), k)
6      centroids__np = np.zeros((k, d))
7      for centroidIdx in range(k):
8          randomDP = dataset.iloc[randomDPIdxs[centroidIdx]]
9          for dimIdx in range(d): centroids__np[centroidIdx][dimIdx] =
               randomDP.iloc[dimIdx]
10
11     centroids_OLD = pd.DataFrame()
12
13     centroids_OLD__np = centroids_OLD.T.to_numpy()
14     dataset__np = dataset.to_numpy()
15     del dataset
16
17     datasetLogs = np.zeros((n, d))
18     calcLogs(dataset__np, datasetLogs)
19
20     iteration = 1
21
22     while iteration <= maxIter and not
```

```
        np.array_equal(centroids_OLD__np ,centroids__np):
23          distances = np.zeros((n, k))
24          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 6, 7 e 8). A estrutura do código não muda muito em relação ao Algoritmo 4, então apenas as partes mais modificadas serão explicadas aqui.

Como aprofundado no capítulo 2.4, 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 4. 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 4. 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 6.2.

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 6). Esse é o passo de cálculo dos logarítimos naturais de todos os datapoints, equivalente à operação realizada na linha 5 do Algoritmo 4.

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 4.

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 4. 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 4.

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 6, 7 e 8 abaixo mostram as funções vetoriais que rodam na GPU NVIDIA, onde os cálculos mais pesados são realizados. Como discutido mais a fundo no capítulo 2.4, 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 6 – 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 5 (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 7 – 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 5 (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 8 – 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 5 (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 7, 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 2.4.

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 Experimentos e Datasets

Nesse trabalho, 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 corretude para checar se houveram mudanças quanto à precisão 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 6) são exibidos e discutidos os resultados dos experimentos.

5.1 Ambiente de Execução

TODO: Escrever sobre minha máquina onde realizei os testes.

5.2 Testes de Desempenho e Precisão

TODO: Escrever brevemente sobre como foram feitos os testes de desempenho e precisão (a.k.a. corretude). Não imagino que eu vá ter que mostrar código aqui... certo?

5.3 Datasets Utilizados

TODO: Escrever uma intro a respeito do processo de escolha dos datasets. Incluir aqui uma boa explicação do passo de pré-processamento comum feito para todos os datasets (normalização min-max), que incluirá com certeza essa citação: (MILLIGAN; COOPER, 1988).

TODO: Escrever sobre o dataset 1, Rice.

TODO: Escrever sobre o dataset 2, HTRU2.

TODO: Escrever sobre o dataset 3, MiniBooNE.

TODO: Escrever sobre o dataset 4, WESAD.

TODO: Escrever sobre o dataset 5, HHAR.

6 Resultados

Utilizando os cinco datasets detalhados no capítulo 5, foram realizadas diversas rodadas de execuções em ambas versões dos algoritmos aqui analisados.

Cada versão do algoritmo 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.

6.1 K-Means

TODO: Descrever melhor os resultados das execuções do k-means CPU e k-means GPU.

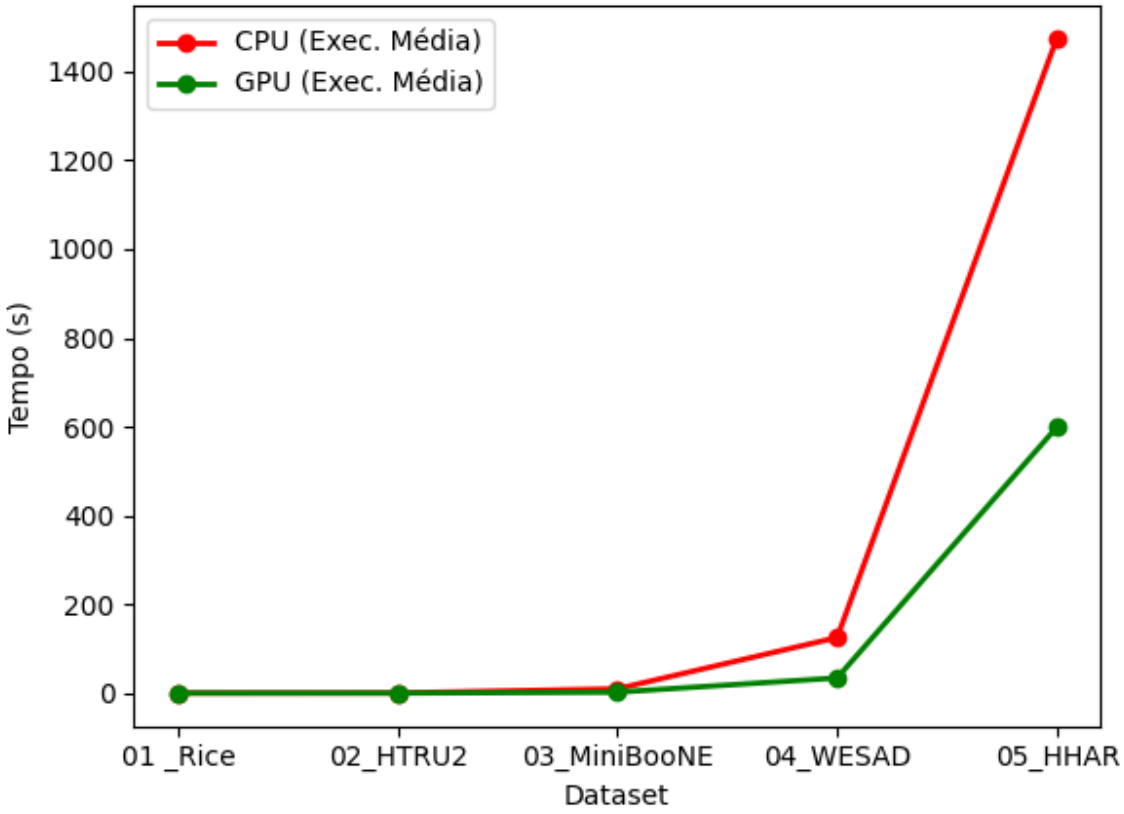
TODO: Atualizar a figura, usando os novos resultados e um gráfico de barras dessa vez.

TODO: Adicionar as informações em forma de tabela também.

6.2 Precisão

TODO: Escrever uma seção apresentando e discutindo os resultados dos testes de precisão e corretude rodados com cada dataset e implementação do K-Means.

Figura 1 – Tempos de execução média do K-Means



7 Conclusões e Trabalhos Futuros

TODO: Escrever capítulo com conclusões do trabalho + sugestões do que prosseguir para trabalhos futuros.

Referências

Anaconda, Inc. **Numba: A High Performance Python Compiler**. 2024. <https://numba.pydata.org>. [Online; acessado 15 de abril de 2024]. Citado na página 28.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6028**: Resumo - apresentação. Rio de Janeiro, 2003. 2 p. Citado na página 3.

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 41.

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 37.

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.

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 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 33.

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 19.

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 22.

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 41 e 53.

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 22.

_____. **NVIDIA Nsight Systems**. 2024. <https://developer.nvidia.com/nsight-systems>. Online; acessado 15 de abril de 2024. Citado na página 22.

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 35.

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 na página 10.

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 37.

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 20.

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 36.

Parte I

Anexos

ANEXO A – Eu sempre quis aprender latim

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANEXO B – Coisas que eu não fiz mas que achei interessante o suficiente para colocar aqui

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

ANEXO C – Fusce facilisis lacinia dui

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.