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

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 são dois algoritmos de agrupamento especialmente notórios: o *K-means* (CUOMO et al., 2019) e o *Agrupamento Hierárquico*. Implementações e estudos realizados sobre estes foram analisados, e implementações paralelas em GPU tiveram seu desempenho comparado com as seriais 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 significa-

tivo.

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. 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.

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 (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.

[Inserir aqui parágrafo(s) sobre GPUs, o foco do trabalho]

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.

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.

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*) teve sua origem na virada do século, quando a NVIDIA percebeu o potencial das GPUs para além do processamento gráfico e começou a explorar sua capacidade para tarefas de propósito geral. A história do CUDA remonta ao início dos anos 2000, quando a NVIDIA lançou suas primeiras GPUs com arquiteturas capazes de executar operações paralelas de forma eficiente.

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. 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 melhorias significativas 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 e modelagem financeira avançada. Na indústria de entretenimento, o CUDA é fundamental para a renderização de gráficos em filmes, jogos e animações em 3D.

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

2.4 Algoritmo 1: K-Means

[Escrever uma sub-seção sobre o K-Means]

- Um dos algoritmos mais tradicionais, explicar a história dele
- Explicar como ele é usado

2.5 Algoritmo 2: Hierarchical Clustering

[Escrever uma sub-seção sobre o Hierarchical Clustering]

3 Levantamento do Estado da Arte

[Escrever um capítulo sobre o estado da arte de algoritmos de agrupamento de dados]

4 Metodologia de Desenvolvimento e Pesquisa

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.

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*. Por ser um algoritmo mais antigo, já foi muito estudado anteriormente, tanto em versões seriais quanto paralelas, com grande presença na bibliografia da área. Assim, a implementação aqui realizada foi realizada 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 numa 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.

5 Experimentos e Datasets

[Escrever capítulo sobre os experimentos realisados e conjuntos de dados utilizados nestes]

6 Resultados

7 Conclusões e Trabalhos Futuros

8 Desenvolvimento

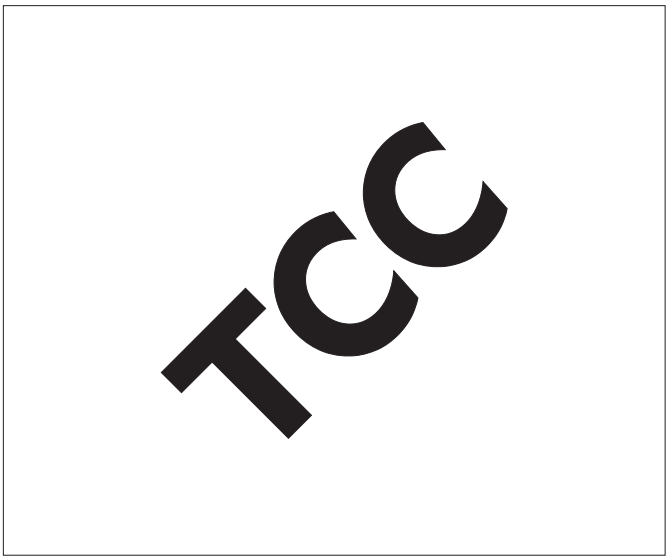

Figura 1 – Imagem de exemplo.

9 Conclusão

E daí?

Referências

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

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 00457906. 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.

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

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.

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.

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.