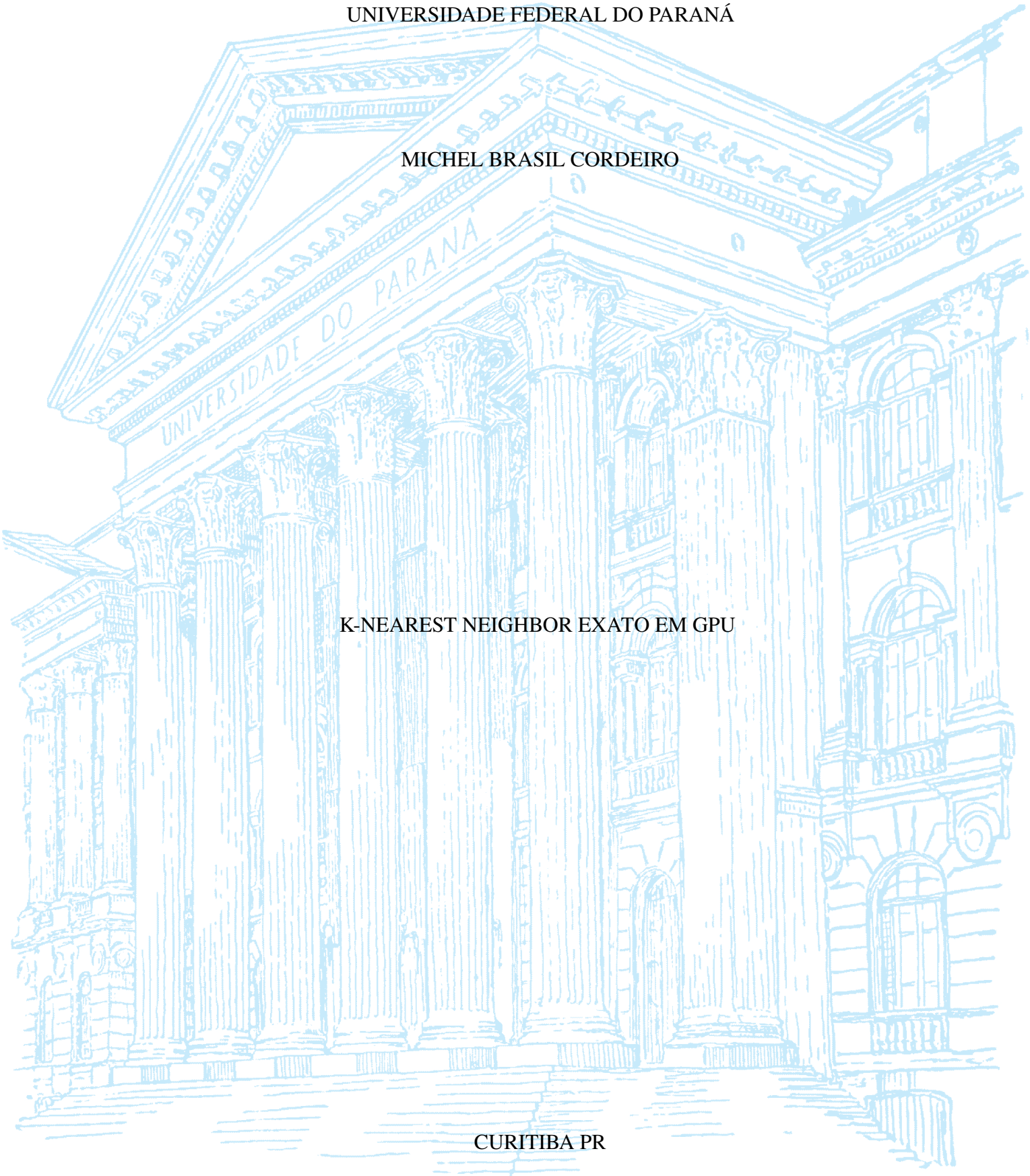
UNIVERSIDADE FEDERAL DO PARANÁ

MICHEL BRASIL CORDEIRO



K-NEAREST NEIGHBOR EXATO EM GPU

CURITIBA PR

2023

MICHEL BRASIL CORDEIRO

K-NEAREST NEIGHBOR EXATO EM GPU

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciências da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Wagner M. Nunan Zola.

CURITIBA PR

2023

AGRADECIMENTOS

Primeiramente, gostaria de expressar minha profunda gratidão à minha mãe, que desde o início me proporcionou amor, incentivo e apoio incondicional. Agradeço pelos valiosos ensinamentos e por sempre me motivar a estudar e perseguir os meus sonhos.

Aos meus irmãos, agradeço pelos momentos de descontração, pela assistência constante e por sempre estarem dispostos a ajudar. Obrigado por serem minha família, minha base e minha fonte de amor e apoio.

Um agradecimento especial ao meu orientador pela ajuda, paciência, confiança depositada em mim e pelas valiosas oportunidades proporcionadas ao longo de todo o processo. O apoio recebido foi fundamental para o meu sucesso acadêmico, e expresso minha sincera gratidão por toda a ajuda que recebi.

RESUMO

Algoritmos de aprendizado de máquina costumam ser computacionalmente caros. Sendo assim, diversas abordagens podem ser utilizadas para acelerar esses algoritmos. Uma estratégia é utilizar recursos de *hardware*, como o paralelismo em CPU ou GPU. Nesse sentido, este trabalho apresenta uma implementação eficiente em GPU do algoritmo de aprendizado de máquina *K-Nearest Neighbor* (KNN). O algoritmo proposto foi comparado com outras implementações paralelas em CPU e GPU, incluindo algoritmos da biblioteca FAISS, amplamente utilizada para busca por similaridade em GPU. Os resultados mostram que a implementação proposta está bem otimizada para conjuntos de dados pequenos, alcançando aceleração de até duas vezes em relação aos outros algoritmos quando o conjunto de consulta possui apenas um elemento.

Palavras-chave: K-Vizinhos Mais Próximos. Computação Paralela em GPU. Aprendizado de Máquina.

ABSTRACT

Machine learning algorithms are often computationally expensive. Therefore, various approaches can be employed to speed up these algorithms. One strategy is to leverage hardware resources, such as parallelism on CPU or GPU. In this context, this work presents an efficient GPU implementation of the K-Nearest Neighbor (KNN) machine learning algorithm. The proposed algorithm was compared with other parallel implementations on CPU and GPU, including algorithms from the widely used FAISS library for GPU-based similarity search. The results demonstrate that the proposed implementation is well-optimized for small datasets, achieving acceleration of up to two times compared to other algorithms when the query set contains only one element.

Keywords: K-Nearest Neighbor. GPU Parallel Computing. Machine Learning.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Algoritmos de aprendizado de máquina vêm se tornando cada vez mais importantes nos últimos anos. Esses algoritmos são utilizados para diversos propósitos, como mineração de dados, processamento de imagens, análise preditiva, entre outros (Mahesh, 2020).

O *K-nearest neighbor* (KNN) é um simples algoritmo de aprendizado de máquina que pode ser usado para resolver diversos tipos de problemas. Seu objetivo consiste em encontrar em um conjunto de referência os K pontos mais próximos para cada ponto em um conjunto de consulta (Cordeiro et al., 2023).

Para alcançar esse objetivo, o algoritmo calcula a distância entre o ponto consultado e cada ponto no conjunto de referência. Sendo assim, a complexidade do KNN para um único ponto de consulta é $O(N \times D)$, no qual N é o tamanho do conjunto de referência e D é a dimensão dos pontos (LaViale, 2023). Isso faz com que o algoritmo KNN se torne computacionalmente custoso para altas dimensões e grandes conjuntos de dados.

Por essa razão, diversas estratégias podem ser necessárias para acelerar o algoritmo. Uma possibilidade é particionar o espaço de busca, reduzindo o número de distâncias a serem calculadas. O problema dessa abordagem é que alguns dos K pontos mais próximos podem não estar na mesma partição, resultando em uma solução aproximada para o KNN (Trabelsi, 2020). Portanto, algoritmos que utilizam essa estratégia são chamados de *Approximate Nearest Neighbor* (ANN). Outra maneira de acelerar o KNN é aproveitando recursos de *hardware*, como o paralelismo em GPU, sem prejudicar a precisão do algoritmo.

1.1 OBJETIVOS

Este trabalho apresenta uma implementação eficiente do KNN utilizando paralelismo em GPU. O algoritmo proposto é otimizado para quando o conjunto de busca possui poucos pontos. Sendo assim, o algoritmo será chamado de *Small Batch Query K-Nearest Neighbor* (SBQ-KNN). Este trabalho também apresenta uma implementação do algoritmo *K-selection*, cuja função é encontrar os K menores valores de um vetor, sendo utilizado como etapa final do SBQ-KNN.

1.2 JUSTIFICATIVA

Existem diversas aplicações em que não é possível realizar a busca por vizinhos mais próximos em grandes lotes, resultando em um tamanho reduzido para o conjunto de consulta. Um exemplo desse cenário é quando os pontos do conjunto de consulta são móveis (Song e Roussopoulos, 2001; Gu et al., 2017). Nesse caso, a pesquisa deve ser realizada no instante que for requisitada, pois os pontos de consulta se modificam com o tempo, impossibilitando que a agregação de consultas seja realizada. Esse problema surge em aplicações de sistemas de informações geográficas e sistemas de posicionamento global, como encontrar os hotéis mais próximos de um viajante, por exemplo.

Uma variação desse problema consiste em realizar consultas em um conjunto de referência com pontos que também são móveis (Iwerks et al., 2003; Güting et al., 2010; Li et al., 2022). Isso implica em consultas sendo realizadas em um conjunto de referência que também se modifica ao longo do tempo, tornando difícil o uso de estruturas de dados complexas que dividem o espaço de busca. Esse problema é comum em aplicativos de transporte urbano,

nos quais o objetivo é encontrar os motoristas mais próximos de um passageiro. Além disso, é possível mencionar problemas relacionados ao uso do algoritmo KNN em fluxos de dados (Liu e Ferhatosmanoğlu, 2003; Yeh et al., 2008). Nessas aplicações, a busca pelos pontos mais próximos é realizada enquanto os dados são coletados, o que impede que ela seja feita em grandes lotes.

Por fim, existem casos em que o KNN é utilizado por algoritmos de inteligência artificial. Um exemplo disso é o algoritmo SLIDE-GPU (Meyer e Nunan Zola, 2023), uma rede neural de classificação extrema que utiliza um algoritmo de ANN para encontrar quais neurônios de uma determinada camada devem ser ativados. Outro exemplo é o algoritmo KNN-Q (Lin et al., 2020), que utiliza o KNN para selecionar os estados mais próximos de um estado no qual o algoritmo é incapaz de operar.

Esses dois últimos exemplos são particularmente interessantes, pois neles o KNN faz consultas isoladas, um ponto por vez. Além disso, o desempenho do KNN impacta diretamente o desempenho de um algoritmo maior. O algoritmo SLIDE-GPU será melhor explicado no Capítulo 3.

1.3 ANÁLISE E RESULTADOS

Para avaliar sua eficiência, o SBQ-KNN foi comparado com outros quatro algoritmos: um algoritmo de KNN paralelo em CPU, proposto em (Rieger e Zola, 2023); um algoritmo de ANN, proposto em (Meyer et al., 2022); e dois algoritmos da biblioteca FAISS (FAISS, 2023), descritos em (Johnson et al., 2019). Esses algoritmos serão explicados no Capítulo 3.

Os resultados da comparação demonstraram que o SBQ-KNN se destaca quando o conjunto de consulta possui apenas um ponto, apresentando tempo de execução menor do que os outros algoritmos em todas as bases de dados testadas. Além disso, o SBQ-KNN alcança uma aceleração de até 2 vezes em relação aos algoritmos da biblioteca FAISS quando o conjunto de consulta possui apenas um elemento. No entanto, à medida que o número de pontos no conjunto de consulta aumenta, o SBQ-KNN não escala tão bem quanto o algoritmo do FAISS, sendo superado nesse cenário.

1.4 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados os fundamentos teóricos que serão utilizados no restante do trabalho. No Capítulo 3 estão os trabalhos relacionados, onde são apresentados os algoritmos que motivaram a criação deste trabalho, bem como os algoritmos que foram utilizados para analisar o desempenho do SBQ-KNN.

O Capítulo 4 descreve os detalhes da implementação do algoritmo proposto neste trabalho. No Capítulo 5 é feita a análise do desempenho do algoritmo implementado, comparando-o com outras implementações do algoritmo de KNN. Por fim, o Capítulo 6 apresenta as conclusões.

2 FUNDAMENTOS TEÓRICOS

Este capítulo aborda os fundamentos teóricos deste trabalho. Na primeira seção, são apresentados alguns conceitos de aprendizado de máquina, incluindo a descrição do algoritmo *K-Nearest Neighbor*. Na segunda seção, são discutidos conceitos de programação paralela em GPU. Esses elementos são essenciais para compreender as estratégias de implementação propostas neste trabalho e para analisar o desempenho dos algoritmos.

2.1 APRENDIZADO DE MÁQUINA

Aprendizado de máquina é um ramo da inteligência artificial que se baseia na ideia de que sistemas podem aprender com os dados, identificar padrões e tomar decisões, sem que a intervenção humana seja necessária (Mahesh, 2020). De acordo com Arthur Samuel, um dos pioneiros no assunto, aprendizado de máquina pode ser definido como o campo de estudo que dá ao computador a habilidade de aprender sem ser explicitamente programado (Mahesh, 2020).

Algoritmos de aprendizado de máquina podem ser divididos em dois tipos: *Lazy learners* e *eager learners* (Keita, 2022). A principal característica dos algoritmos *eager learners* é a capacidade de construírem um modelo durante a fase de treinamento (Keita, 2022). Assim, esses algoritmos utilizam o modelo construído para fazer predições. Dessa forma, algoritmos *eager learners* podem demandar muito tempo durante a fase de treinamento, uma vez que precisam criar e aprimorar um modelo de predições antes de começar a realizar inferências.

Já os algoritmos *lazy learners* não constroem modelos, apenas memorizam os dados de treinamento (Keita, 2022). Portanto, sempre que precisam realizar predições, esses algoritmos fazem uma busca no conjunto de treinamento. Nessa categoria, encontra-se o algoritmo *K-Nearest Neighbor*, que será apresentado a seguir.

2.1.1 *K-Nearest Neighbor*

O *K-Nearest Neighbor* (KNN) é um algoritmo de aprendizado de máquina que recebe dois conjuntos de pontos como entrada: um conjunto P com pontos de referência e um conjunto Q de pontos a serem consultados. O objetivo é encontrar os K pontos do conjunto P que estão mais próximos de cada ponto do conjunto Q.

A saída do algoritmo é uma matriz $|Q| \times K$, na qual cada linha representa os K pontos mais próximos de um ponto do conjunto Q. Para realizar esse objetivo, o algoritmo calcula a distância entre cada ponto Q e cada ponto de P. Diversas definições de distância podem ser consideradas, sendo a distância euclidiana a mais utilizada.

Existem diversas abordagens para acelerar o KNN. Essas abordagens podem utilizar estratégias algorítmicas, como utilizar um algoritmo de *Approximate Nearest Neighbor* (ANN), ou de implementação, aproveitando recursos de *hardware* como o paralelismo em CPU ou GPU (Meyer et al., 2022). O principal benefício de usar GPUs é que elas oferecem uma taxa de processamento e largura de banda de memória muito maior do que as CPUs com faixa de preço e consumo de energia similares (CUDA, 2023). Por esse motivo, muitas aplicações têm aproveitado essas capacidades para implementar algoritmos que executam mais rapidamente na GPU do que na CPU (CUDA, 2023).

Este trabalho utiliza GPU para acelerar o KNN, sem utilizar métodos que prejudicam a precisão do algoritmo. A próxima seção introduzirá conceitos importantes de computação paralela em GPU.

2.2 COMPUTAÇÃO PARALELA EM GPU

Existem diferenças substanciais entre a arquitetura de processadores orientados à maior vazão de operações, como GPUs (*graphics processing units*) e processadores orientados a executar operações com menor latência, denominados CPU (*central processing unit*). CPUs tem como objetivo executar sequências de operações, em fluxos denominados *threads*, o mais rápido possível, podendo executar algumas dezenas de *threads* em paralelo (CUDA, 2023). Por outro lado, o objetivo das GPUs é realizar computação paralela de alta vazão, sendo capaz de executar milhares de *threads* simultaneamente.

Para desenvolver códigos de programação que executem em GPU, é necessário utilizar uma interface de programação, como a *Compute Unified Device Architecture* (CUDA). CUDA é uma plataforma de computação paralela e interface de programação criada pela Nvidia em 2006 (Oh, 2012). Ela estende a linguagem de programação C++, permitindo que os programadores definam funções que serão executadas na GPU (CUDA, 2023). Essas funções são chamadas de *kernels*.

2.2.1 Hierarquia de *Threads*

CUDA organiza as *threads* em uma hierarquia. Primeiramente, as *threads* são organizados em *warps* (Johnson et al., 2019) e, dentro de um *warp*, cada *thread* é chamada de *lane*. Todas as *lanes* compartilham o mesmo contador de instrução, o que significa que todas estão executando a mesma instrução em um determinado momento (Johnson et al., 2019). Quado as *lanes* precisam seguir diferentes fluxos de execução, devido a algum desvio condicional, por exemplo, acontece uma divergência de *warp* (Johnson et al., 2019). Por compartilhar o mesmo contador de instruções, o *warp* inteiro terá que passar pelo fluxo de execução, mas apenas as *lanes* que necessitam executar aquele fluxo estarão ativas, impactando negativamente o desempenho do algoritmo.

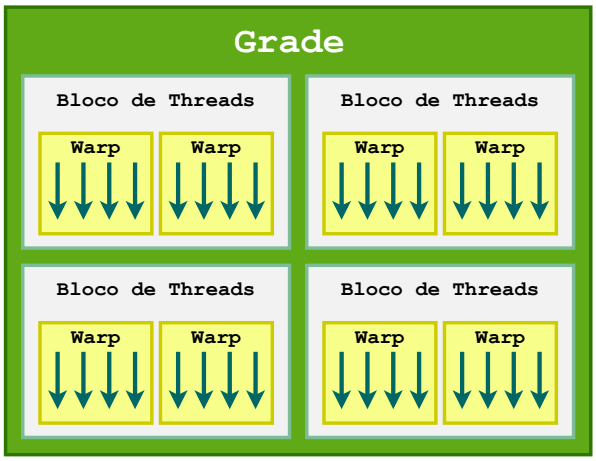


Figura 2.1: Hierarquia de *threads* em CUDA.

Por sua vez, vários *warps* são agrupados em blocos de *threads*. A quantidade de *threads* em um bloco é configurada pelo programador e pode conter até 1024 *threads* (CUDA, 2023) em

arquiteturas atuais. Todas as *threads* de um bloco executam em um mesmo multiprocessador da GPU. Após a execução completa, novos blocos podem ser escalonados, sendo que blocos de diferentes *kernels* podem ser executados simultaneamente. Por fim, os blocos de *threads* são organizados em grades. A Figura 2.1 ilustra como as *threads* são organizadas.

2.2.2 Hierarquia de *Memória*

Assim como as *threads*, a memória da GPU também é organizada em uma hierarquia. A memória global tem maior capacidade de armazenamento e alta vazão, mas apresenta maior latência de acesso na GPU. Ela é compartilhada entre todos os blocos de *threads*, permitindo que qualquer *thread* possa acessá-la.

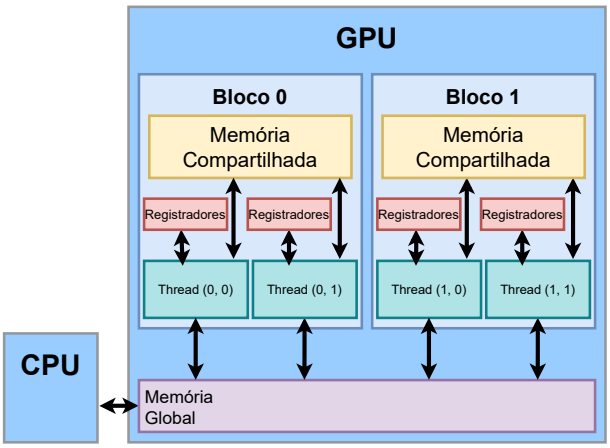


Figura 2.2: Hierarquia de memória na GPU.

Cada multiprocessador (MP) da GPU possui uma memória compartilhada (*shared memory*) visível apenas para as *threads* que fazem parte de um mesmo bloco de *threads* alocado àquele MP. Essa memória possui uma latência muito baixa em relação à memória global, podendo ser comparada com a memória *cache* L1 de uma CPU (Johnson et al., 2019), sendo no entanto controlada por software. Em compensação, a memória compartilhada possui capacidade de armazenamento limitada.

Cada MP possui uma grande quantidade de registradores, tipicamente 65K registradores, que são alocados às *threads* para armazenamento de variáveis locais. CUDA possui funções que permitem às *lanes* compartilhar variáveis entre os registradores, sendo essa a forma mais rápida de trocar informações dentro de um *warp*.

2.2.3 Sincronização

CUDA possui muitas funções de sincronização, cada uma com suas características. Para entender esse trabalho, é preciso conhecer três funções de sincronização. A primeira é a *sync_warp*, que faz a sincronização entre as *lanes* de uma *warp*, sem interferir na execução das *threads* de outras *warps*.

Por mais que todas as *lanes* executam instruções simultaneamente, isso não garante que todas as leituras e escritas aconteçam ao mesmo tempo. Por isso, a sincronização entre *warps* é importante para garantir que todas as *lanes* terminaram de fato a instrução que iniciaram.

A segunda função é a *__syncthreads*, que faz a sincronização entre as *threads* de um bloco. Esse é o tipo de sincronização mais utilizado, pois as *threads* de um bloco costumam

trabalhar juntas para resolver um mesmo problema, e muitas vezes precisam ser sincronizadas para compartilhar informações.

Por fim, é possível sincronizar a GPU com a CPU. Isso é feito pela função *cudaDeviceSynchronize*, que garante que todos os *kernels* lançados pelo CPU terminaram de executar na GPU e que os resultados da computação estão prontos.

2.2.4 Ocupação da GPU

A ocupação da GPU é uma medida de paralelismo de *threads* em um programa CUDA (Baxter, 2013). Os multiprocessadores possuem um limite para a quantidade de *threads* que podem ser executadas simultaneamente. A quantidade de registradores e *shared memory* também é limitada.

Sendo assim, se toda a *shared memory*, ou todos os registradores de um multiprocessador forem alocados para apenas um bloco, esse bloco terá que ser executado isoladamente, mesmo que o multiprocessador possua *threads* disponíveis para executar outros blocos.

Além disso, os multiprocessadores limitam a quantidade máxima de *threads* que um bloco pode alocar. Consequentemente, não é possível utilizar todos recursos do multiprocessador utilizando apenas um bloco. Portanto, para maximizar a ocupação da GPU, é importante alocar adequadamente os recursos dos multiprocessadores entre os blocos de *threads*.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos relacionados. O primeiro artigo apresentado envolve duas implementações do KNN em GPU (Cordeiro et al., 2023). Esses algoritmos serviram como base para a implementação do KNN proposto neste trabalho.

O segundo trabalho (Rieger e Zola, 2023) apresenta uma implementação paralela do algoritmo KNN para cluster de CPUs. Em seguida, é introduzido o artigo *Billion-Scale Similarity Search with GPUs* (Johnson et al., 2019), que descreve dois algoritmos que estão disponíveis na biblioteca FAISS (FAISS, 2023) e são amplamente utilizados para realizar buscas por similaridade em GPU.

Também é apresentado o artigo sobre o ANN-RFSK (Meyer et al., 2022), um algoritmo de busca aproximada que demonstrou potencial para superar o estado da arte em diversos cenários. Por fim, o algoritmo SLIDE-GPU (Meyer e Nunan Zola, 2023), mencionado na introdução, é explicado com mais detalhes.

3.1 KNN EXATO EM GPU

O artigo **KNN Exato em GPU** (Cordeiro et al., 2023) apresenta duas implementações do KNN em GPU. A primeira é chamada de "Um Ponto Por *Kernel*"(PPK). Nessa versão do algoritmo, é lançado um *kernel* para cada ponto no conjunto de consulta. Cada *kernel* é lançado com a exata quantidade de blocos de *threads* necessários para atingir a máxima ocupação da GPU.

Para atingir esse objetivo, o conjunto de referência P é particionado entre os blocos, que encontram os K-vizinhos mais próximos apenas na partição que receberam. Dessa forma, essa versão consegue ser muito eficiente para lidar com pequenos conjuntos de busca. No entanto, após a execução do algoritmo, é necessário juntar os vetores de KNN que cada bloco encontrou para formar o resultado final, adicionando um custo extra ao algoritmo.

A segunda implementação do algoritmo é chamado de "Um Ponto Por Bloco"(PPB). Nessa versão, cada bloco de *threads* é responsável por encontrar os K-vizinhos mais próximos de um ponto do conjunto Q. Dessa forma, é lançado apenas um *kernel* CUDA com o número de blocos igual ao número de pontos no conjunto Q. A principal vantagem dessa versão é que cada bloco realiza a busca completa no conjunto P, tornando desnecessário juntar os resultados depois.

Entretanto, é importante observar que o número de blocos depende da quantidade de pontos no conjunto Q, e se a quantidade for muito pequena, pode não haver blocos suficientes para alcançar a ocupação máxima da GPU.

A conclusão do artigo é que cada uma das versões é otimizada para diferentes quantidades de pontos no conjunto Q. Alguns resultados da comparação podem ser vistos na tabela 3.1. Os tempos estão em milissegundos e os melhores tempos estão destacados em negrito.

3.2 ALGORITMO KNN PARALELO EM *CLUSTER* COM MPI

O artigo **Algoritmo KNN paralelo em *cluster* com MPI** (Rieger e Zola, 2023) apresenta uma implementação paralela do KNN em CPU, utilizando a biblioteca Open MPI. O estudo avalia a escalabilidade do algoritmo, executando-o em vários processadores.

BASE DE DADOS COM 70.000 PONTOS, 784 DIMENSÕES (MNIST)

	versão: Exato PPK					versão: Exato PPB				
	|Q|					|Q|				
K	1	16	32	64	128	1	16	32	64	128
32	**0,39**	**6,14**	12,28	24,54	49,12	10,05	9,24	**9,90**	**12,47**	**24,80**
64	**0,40**	**6,21**	12,41	24,80	49,65	9,80	9,12	**9,85**	**12,34**	**24,75**
128	**0,40**	**6,22**	12,42	24,83	49,64	10,22	9,34	**9,97**	**12,47**	**25,03**

BASE DE DADOS COM 1.275.219 PONTOS, 128 DIMENSÕES (ImageNet)

	versão: Exato PPK					versão: Exato PPB				
	|Q|					|Q|				
K	1	16	32	64	128	1	16	32	64	128
32	**1,13**	**17,93**	**35,87**	71,86	143,90	48,03	47,54	48,86	**64,45**	**122,09**
64	**1,14**	**18,08**	**36,20**	72,63	145,40	48,18	47,66	48,82	**64,71**	**123,30**
128	**1,16**	**18,39**	**36,84**	73,96	148,26	48,10	47,78	49,10	**65,52**	**123,62**

BASE DE DADOS COM 3.000.000 PONTOS, 300 DIMENSÕES (GoogleNews300)

	versão: Exato PPK					versão: Exato PPB				
	|Q|					|Q|				
K	1	16	32	64	128	1	16	32	64	128
32	**6,14**	**98,35**	**196,96**	394,32	789,75	254,85	243,34	259,50	**296,52**	**566,37**
64	**6,15**	**98,43**	**197,05**	394,45	790,03	255,55	243,21	255,41	**293,65**	**562,42**
128	**6,15**	**98,55**	**197,25**	394,94	790,77	256,40	245,07	260,87	**297,93**	**568,66**

BASE DE DADOS COM 300.000 PONTOS, 128 DIMENSÕES

	versão: Exato PPK					versão: Exato PPB				
	|Q|					|Q|				
K	1	16	32	64	128	1	16	32	64	128
32	**0,29**	**4,47**	**8,93**	17,87	35,90	11,48	11,11	11,43	**14,67**	**26,44**
64	**0,30**	**4,64**	**9,27**	18,54	37,24	11,69	11,10	11,39	**14,54**	**26,31**
128	**0,33**	**4,99**	**9,97**	19,98	39,94	11,77	11,36	11,69	**14,93**	**26,86**

Tabela 3.1: Comparação entre a versão PPK e PPB. FONTE:(Cordeiro et al., 2023)

Para computar o KNN, o algoritmo utiliza uma matriz virtual, na qual cada linha representa um ponto do conjunto de consulta e armazena as K menores distâncias calculadas até o momento para aquele ponto. Junto com a matriz, é mantida uma variável com o maior valor entre as K distâncias. Quando uma distância menor é calculada, o algoritmo atualiza a matriz virtual e a variável que armazena a maior distância.

Essa implementação paralela do algoritmo do KNN foi comparada com uma versão sequencial do KNN. Os resultados demonstram que a versão paralela apresenta uma excelente escalabilidade, proporcionando uma aceleração igual e, em alguns casos, até superior (super linear) ao número de processos utilizados.

A Figura 3.1 apresenta alguns dos resultados destacados no artigo. Nessa comparação, foi utilizada uma base com dados gerados aleatoriamente composta por 70 mil pontos de 784 dimensões.

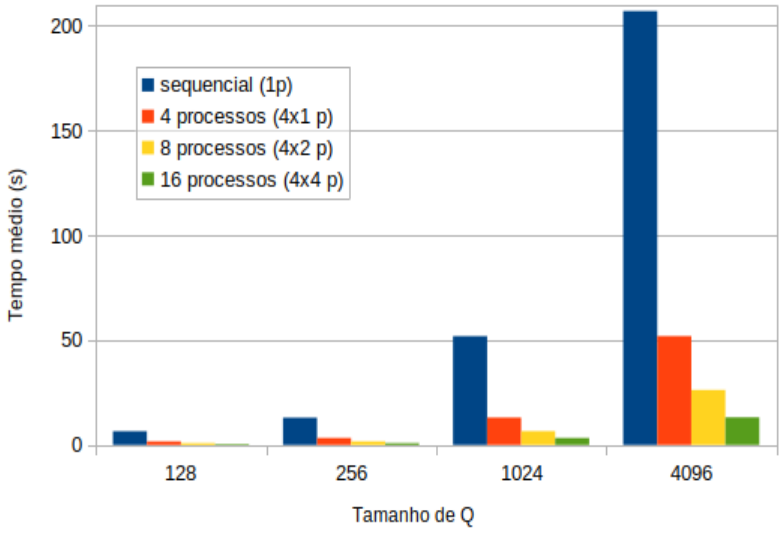


Figura 3.1: Comparação entre a versão sequencial e paralela do KNN em CPU. FONTE: (Rieger e Zola, 2023).

3.3 *BILLION-SCALE SIMILARITY SEARCH WITH GPUS*

O artigo ***Billion-Scale Similarity Search with GPUs*** (Johnson et al., 2019) apresenta dois algoritmos implementados em GPU para resolver o KNN. O primeiro algoritmo é chamado de *index flat*, que computa todas as distâncias dos pontos de Q para os pontos de P e depois utiliza um algoritmo de *K-selection* para encontrar os K vizinhos mais próximos.

Para ser mais exato, o *index flat* não realiza todo o cálculo da distância de uma única vez. Ao invés de calcular $||p_q - q_j||^2$, o algoritmo usa a igualdade $||p_i - q_j||^2 = ||p_i||^2 + ||q_i||^2 - 2 < q_i, p_j >$ e calcula apenas o termo $-2 < q_i, p_j >$, gerando uma matriz de distâncias parciais. Após isso, ele utiliza o *K-selection* modificado que soma $||p_i||^2$ ao elemento da matriz antes de adicioná-lo aos registradores que computam o *K-selection*. O termo $||q_i||^2$ não é levado em consideração.

Segundo os autores do artigo, o algoritmo de *K-selection* implementado por eles é o atual estado da arte para *K-selection* em GPU. Esse algoritmo é eficiente porque consegue manter toda a estrutura de dados em registradores, fazendo com que algoritmos de ordenação e junção de vetores sejam executados muito mais rapidamente.

O segundo algoritmo apresentado nesse artigo é denominado de *IndexIVF*, que é um algoritmo de ANN. Esse algoritmo utiliza uma técnica de particionamento parecida com o *K-means* para reduzir o espaço de busca. Dessa forma, é necessário que o *IndexIVF* passe por um treinamento antes de poder ser utilizado.

Durante esse treinamento, ele particionará o conjunto P, e construirá uma estrutura de dados para realizar buscas de forma eficiente. Antes de treinar, o *IndexIVF* recebe o número de partições como parâmetro. Quanto maior o número de partições, mais rápido será realizar uma busca, porém menor será a precisão do algoritmo.

Essas duas implementações do artigo mostraram ser muito eficientes e se tornaram o estado da arte em busca por similaridade em GPU. Ambas estão implementadas na biblioteca FAISS, que está disponível em seu repositório do GitHub (FAISS, 2023).

3.4 ANN-RSFK: BUSCA GENÉRICA DE SIMILARIDADE EM GPU

No artigo **ANN-RSFK: Busca genérica de similaridade em GPU** (Meyer et al., 2022) é apresentado o algoritmo ANN-RSFK, um algoritmo de busca de vizinhos aproximados que particiona o espaço de busca. Para fazer isso, é utilizado o algoritmo RSFK que cria árvores que dividem a base de dados de referência com a ajuda de hiperplanos.

O ANN-RSFK é então comparado com três implementações em GPU: uma implentação do KNN (FLATL2) e duas implementações de ANN (IVFPQ e IVFFLAT), todas disponíveis na biblioteca FAISS. No resultado da comparação, observou-se que o algoritmo ANN-RSFK possui melhor custo-benefício em termos de tempo e acurácia em comparação com a estratégia IVFPQ.

Para acurácias menores que 50%, o método ANN-RSFK foi o melhor algoritmo. Entretanto, a estratégia IVFFLAT apresentou menor tempo de execução para acurácias mais altas. Os resultados dos experimentos do artigo podem ser vistos na Figura 3.2.

3.5 *TOWARDS A GPU ACCELERATED SELECTIVE SPARSITY MULTILAYER PERCEPTRON ALGORITHM USING K-NEAREST NEIGHBORS SEARCH*

Para lidar com o alto custo computacional associado ao treinamento de redes neurais em problemas de classificação extrema, o artigo ***Towards a GPU accelerated selective sparsity multilayer perceptron algorithm using K-Nearest Neighbors search*** (Meyer e Nunan Zola, 2023) propõe o SLIDE-GPU, um modelo baseado no algoritmo Sub-LInear Deep learning Engine (SLIDE) (Daghaghi et al., 2021).

O SLIDE utiliza um conceito chamado seletividade esparsa, que consiste em selecionar diferentes neurônios da rede neural considerando algum mecanismo de seleção. A versão do SLIDE proposta por (Daghaghi et al., 2021) utiliza a técnica *Locality-Sensitive Hashing* (LSH) para selecionar os neurônios. O LSH utiliza funções de *hash* para criar valores semelhantes para

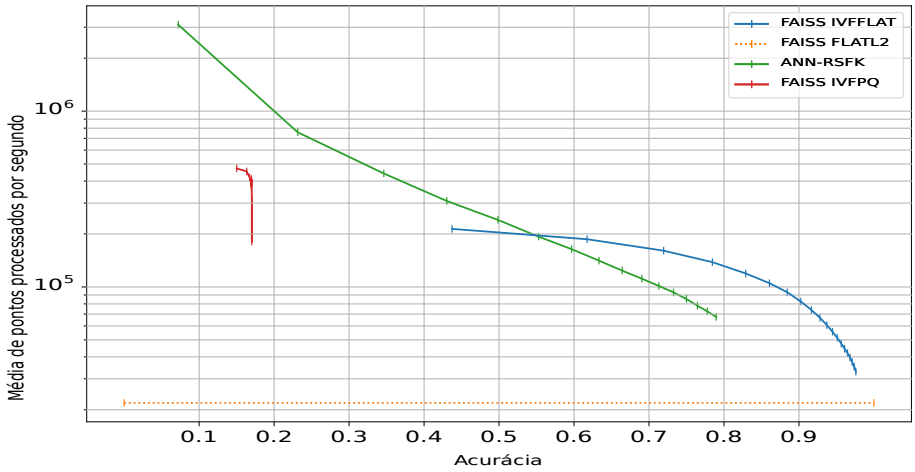


Figura 3.2: Comparação de desempenho entre o algoritmo ANN-RSFK e os algoritmos da biblioteca FAISS. FONTE: (Meyer et al., 2022).

vetores que estão próximos no sistema utilizado. Essa versão do SLIDE é implementada para ser executada em paralelo na CPU e, portanto, é chamada de SLIDE-CPU.

Já o SLIDE-GPU é implementado em GPU e utiliza um algoritmo de ANN como mecanismo de seleção. O SLIDE-GPU utiliza a função IVFFLAT disponível na biblioteca FAISS. Essa função implementa o algoritmo IndexIVF proposto no artigo apresentado na seção 3.3.

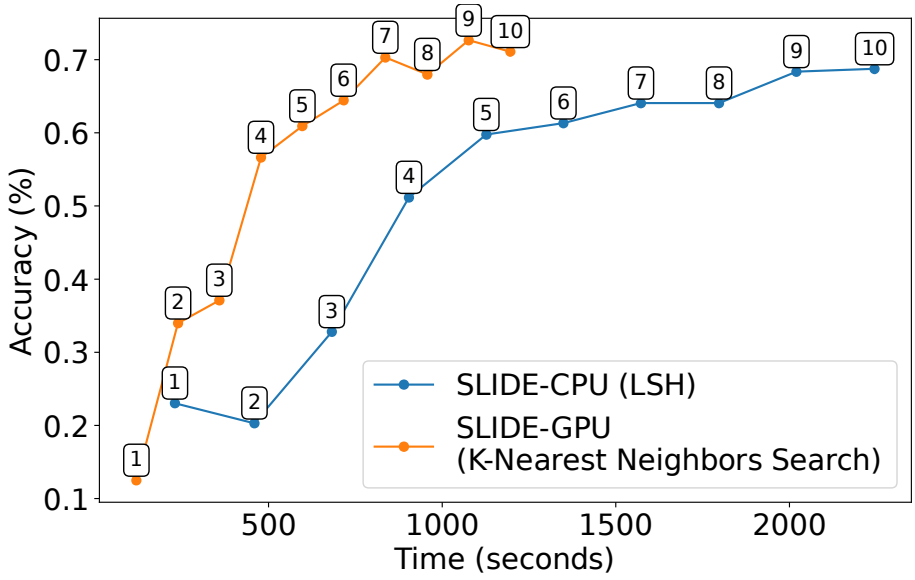


Figura 3.3: Comparação entre o SLIDE-GPU e o SLIDE-CPU. FONTE: (Meyer e Nunan Zola, 2023).

Por fim, essas duas versões são comparadas. O resultado da comparação está na Figura 3.3, na qual é possível notar que o SLIDE-GPU não apenas obtém um menor tempo de execução, mas também uma precisão superior em relação ao SLIDE-CPU.

A Figura 3.4 mostra os tempos de seleção e ativação dos neurônios para as duas versões do SLIDE. É possível notar que, no caso do algoritmo SLIDE-GPU, o tempo gasto na seleção dos neurônios usando o ANN representa uma proporção significativa do tempo total do algoritmo. Isso significa que substituir o ANN por um algoritmo de busca de vizinhos mais eficiente resultaria em um ganho considerável de desempenho para o SLIDE-GPU.

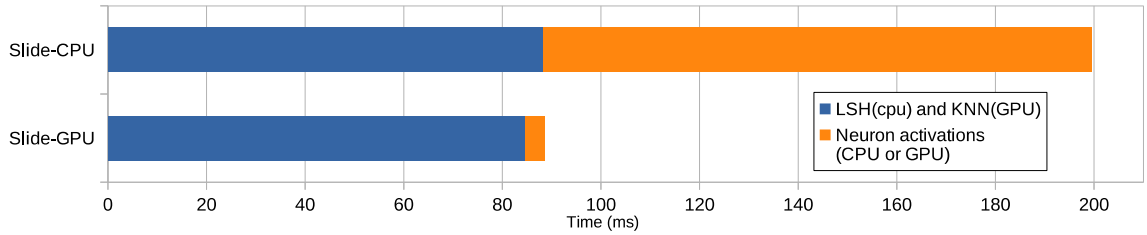


Figura 3.4: Análise comparativa dos tempos de ativação e seleção de neurônios para ambas as versões do SLIDE. FONTE: (Meyer e Nunan Zola, 2023).

4 IMPLEMENTAÇÃO DO SBQ-KNN EM GPU

Este capítulo descreve a implementação do SBQ-KNN e está dividido em três seções: a primeira descreve a função principal do algoritmo; a segunda apresenta o *kernel* que realiza o cálculo do KNN em GPU; por fim, a terceira descreve o algoritmo de *K-selection*, utilizado para agregar resultados parciais do KNN quando necessário.

4.1 FUNÇÃO PRINCIPAL

Algoritmo 1 SBQ-KNN

Input: Q, P, k
Output: $knnExato$

1: $q \leftarrow |Q| \div Nblocks$
2: $knnExato[0 : q * Nblocks - 1] \leftarrow KNNkernel(Q, q, P, 1, k)$
3: $remaining \leftarrow |Q| \mod Nblocks$
4: **if** $remaining > 0$ **then**
5: $\quad q_i \leftarrow |Q| - remaining$
6: $\quad j \leftarrow 0$
7: $\quad$ **while** $q_i < |Q|$ **do**
8: $\quad\quad knnParcial[j] \leftarrow KNNkernel(Q[q_i], 0, P, Nblocks, k)$
9: $\quad\quad j \leftarrow j + 1$
10: $\quad\quad q_i \leftarrow q_i + 1$
11: $\quad$ **end while**
12: $\quad cudaDeviceSynchronize()$
13: $\quad KnnExato[q * Nblocks : |Q| - 1] \leftarrow KSelection(KnnParcial, k)$
14: **end if**
15: **return** $KnnExato$

O algoritmo 1 descreve a função SBQ-KNN, que é a função principal do algoritmo. Ela recebe dois conjuntos como entrada: o conjunto de referência P, o conjunto de consulta Q; e o valor do K. A saída do algoritmo é a matriz $knnExato$, na qual a linha i contém os K-vizinhos mais próximos do ponto $q_i \in Q$.

Na primeira linha, o conjunto Q é dividido igualmente entre os blocos, permitindo que os blocos trabalhem de forma independente. Por isso, o tamanho do conjunto Q é dividido por $Nblocks$, que representa a quantidade de blocos utilizados no *kernel*.

Em seguida, a função $KNNKernel$ é chamada para calcular os K vizinhos mais próximos na GPU. Essa função recebe quatro parâmetros como entrada: O conjunto Q; a quantidade de pontos de Q que cada bloco deve processar; o conjunto P; a quantidade de partições de P; e o valor do K. Nessa primeira etapa não é necessário particionar P e por isso esse parâmetro é fixado em 1.

Como cada bloco calculou os K-vizinhos mais próximos de q pontos, a saída da função $KNNKernel$ é uma matriz $q \times Nblocks$, que será armazenada nas primeiras q linhas da matriz $KnnExato$. A função $KNNKernel$ será detalhada na próxima seção.

Se a quantidade pontos no conjunto Q for divisível pela quantidade de blocos, todos os pontos de Q foram processados. Caso contrário, será necessário computar os pontos remanescentes. Isso acontece nas linhas 5 a 13 do algoritmo.

Como a quantidade de pontos remanescentes é menor que a quantidade de blocos, é necessário balancear o trabalho, particionando o conjunto P entre os blocos. Para realizar essa divisão, basta passar $Nblocks$ como o último parâmetro para a função $KNNKernel$. Desta vez, todos os blocos do *kernel* devem calcular o mesmo ponto de Q, e para garantir isso, o segundo parâmetro da função é fixado em 0.

Para que a função KNNKernel funcione corretamente, é necessário passar a porção do conjunto Q cujos pontos ainda não foram processados. Por esse motivo, na linha 8, o parâmetro passado é $Q[q_i]$, em vez de somente Q.

Ao particionar o conjunto P, cada bloco de *threads* encontrará os K vizinhos mais próximos apenas na partição que lhe foi entregue, gerando um KNN parcial. Para encontrar a solução final, será necessário utilizar um outro algoritmo que junte as K menores distâncias do KNN parcial. Para isso, a função $KSelection$ é chamada na linha 13. Essa função recebe uma matriz como entrada e retorna os K menores elementos para cada linha da matriz.

Antes de chamar a função $KSelection$, é necessário realizar a sincronização entre a GPU e a CPU. Isso é feito utilizando a função $cudaDeviceSynchronize$, na linha 12, que garante que todos os *kernels* tenham concluído a execução. Esse procedimento é importante para garantir que os vetores contendo o KNN parcial estejam prontos para serem passados à função $KSelection$. Por fim, os resultados são armazenados nas últimas linhas da matriz $KnnExato$.

4.2 IMPLEMENTAÇÃO DO KNN EM GPU

A função que computa o KNN em GPU é chamado de $KNNKernel$ e está representado no algoritmo 2. Como já foi mencionado, essa função recebe cinco parâmetros como entrada: O primeiro parâmetro é conjunto Q; o segundo é chamado de nq e representa a quantidade de pontos de Q que cada bloco deve calcular; o terceiro parâmetro é o conjunto P; o quarto é o np, que representa a quantidade de partições em que P foi dividido; e, por último, a função recebe o valor do K. A saída do algoritmo é uma matriz armazenada na memória global que contém os K vizinhos mais próximos que cada bloco encontrou dentro da partição que recebeu.

Inicialmente, é necessário delimitar o espaço de busca em que cada bloco irá trabalhar. Isso é feito nas 15 primeiras linhas do algoritmo. Se nq for diferente de 0, significa que cada bloco trabalhará com diferentes pontos do conjunto Q. Caso contrário, todos os blocos vão trabalhar com o primeiro ponto. É importante mencionar que o $KNNKernel$ não recebe o conjunto Q inteiro, mas apenas os pontos que serão processados.

Cada bloco de um *kernel* possui um índice que fica armazenado em uma variável embutida no CUDA chamada de $blockIdx$. Nas linhas 2 e 3, essa variável é utilizada para encontrar quais pontos do conjunto Q cada bloco irá processar.

Após isso, o algoritmo verifica se há a necessidade de particionar o conjunto P. Se o parâmetro np for igual à 1, O conjunto P não será particionado. Caso contrário, o algoritmo encontra o tamanho de cada partição e a posição da partição no conjunto P.

Para que a execução do KNN seja mais eficiente, cada bloco armazena as K menores distâncias calculadas em um vetor na *shared memory*. Esse vetor é chamado de $sharedKNN$ e precisa ser preenchido antes do início da computação do KNN.

Para isso, as K primeiras distâncias calculadas são armazenadas diretamente no vetor. Em seguida, é utilizada uma função de redução para encontrar o maior elemento do vetor. O resultado da redução é armazenado em uma variável chamada $farthestNeighbor$, que também é armazenada na *shared memory*, e mantém a maior distância do vetor $sharedKNN$. Tudo isso acontece nas linhas 18 à 22.

Algoritmo 2 KNNKernel

Input: Q, nq,P,np,K
Output: $globalKNN$

1: **if** $nq \neq 0$ **then**
2: $Q\_begin \leftarrow blockIdx \times nq$
3: $Q\_end \leftarrow (blockIdx + 1) \times nq$
4: **else**
5: $Q\_begin \leftarrow 0$
6: $Q\_end \leftarrow 1$
7: **end if**
8: **if** $np = 1$ **then**
9: $P\_begin \leftarrow 0$
10: $P\_end \leftarrow |P|$
11: **else**
12: $partition\_size \leftarrow |P| \div np$
13: $P\_begin \leftarrow blockIdx \times partition\_size + warpIdx$
14: $P\_end \leftarrow (blockIdx + 1) \times partition\_size$
15: **end if**
16: **for** $q_i \leftarrow Q\_begin$ **to** Q_end **do**
17: $p_i \leftarrow P\_begin$
18: **for** $i \leftarrow 0$ **to** K **do**
19: $sharedKNN[i] \leftarrow warpDistance(q_i, p_i)$
20: $p_i \leftarrow p_i + WarpsPerBlock$
21: **end for**
22: $farthestNeighbor \leftarrow warpMaxReduction(sharedKNN)$
23: **while** $p_i < P\_end$ **do**
24: $d \leftarrow warpDistance(q_i, p_j)$
25: $sync\_threads()$
26: **if** $d < farthestNeighbor$ **then**
27: $sharedKNN \leftarrow sharedKNN - farthestNeighbor$
28: $sharedKNN \leftarrow sharedKNN \cup p_j$
29: $farthestNeighbor \leftarrow warpMaxReduction(sharedKNN)$
30: **end if**
31: $p_i \leftarrow p_i + WarpsPerBlock$
32: **end while**
33: $globalKNN[q_i] \leftarrow sharedKNN$
34: **end for**
35: **return** $globalKNN$

Tanto o cálculo da distância, quanto a redução para encontrar o vizinho mais distante, são feitos a nível de *warp*. Para calcular as distâncias, cada *lane* soma a diferença entre os valores de cada dimensão dos pontos de Q e P, e então eleva ao quadrado. Em seguida, as *lanes* utilizam os registradores compartilhados para somar as diferenças quadráticas e produzir a distância final.

A redução funciona de maneira semelhante. Cada *lane* busca o elemento mais distante no vetor $sharedKNN$. Quando a *warp* termina de percorrê-lo, as *lanes* compartilham as variáveis e retornam o maior elemento. Manter as *warps* trabalhando em conjunto evita que haja divergências no fluxo de execução e possibilita que haja a troca de informações através de registradores.

Essa é a razão pela qual o ponto p_i é incrementado em $WarpsPerBlock$, que representa a quantidade de *warps* ativas em um bloco. Como as *warps* trabalham em paralelo, são calculadas

$WarpsPerBlock$ distâncias a cada iteração do laço de repetição. Além disso, a variável $warpIdx$, que é o índice da *warp*, é somada ao início da partição na linha 13, assegurando que cada *warp* do bloco processe diferentes pontos de P.

A execução do KNN acontece de fato nas linhas 23 a 32. Sempre que uma distância é calculada, na linha 24, as *threads* sincronizam e verificam se a distância calculada é menor que a distância do vizinho mais distante. Se for menor, a *warp* atualiza o vetor compartilhado e realiza uma redução para encontrar o novo vizinho mais distante.

A sincronização na linha 25 é importante para garantir que a variável $farthestNeighbor$ está atualizada quando a leitura for realizada. Por fim, o vetor $sharedKNN$ é transferido para a matriz $globalKNN$, que será retornada para a função principal. Isso é feito para cada ponto de Q que os blocos receberam.

4.3 *K-SELECTION*

Algoritmo 3 K_Selection

```
   Input: vector, K
   Output: k_selected
 1: k_selected[0 : K − 1] ← vector[0 : K − 1]
 2: largest ← blockMaxReduction(k_selected)
 3: reductionIsNeeded ← false
 4: for i ← K to |vector| do
 5:     e_i ← vector[i]
 6:     if e_i < largest then
 7:         k_selected ← k_selected − largest
 8:         k_selected ← k_selected ∪ e_i
 9:         reductionIsNeeded ← true
10:     end if
11:     sync_threads()
12:     if reductionIsNeeded then
13:         largest ← blockMaxReduction(k_selected)
14:         reductionIsNeeded ← false
15:     end if
16: end for
17: return k_selected
```

O *K-selection* é um algoritmo que tem como objetivo encontrar os K menores elementos de um vetor não ordenado. No contexto deste trabalho, o *K-selection* recebe como entrada os vetores de KNN parcial, quando não foi possível realizar a consulta em todo o conjunto P.

A implementação do *K-selection* pode ser observada no algoritmo 3. Esse algoritmo armazena os K menores valores encontrados em vetor na *shared memory*. Esse vetor é chamado de $k\_selected$. Na primeira linha, o vetor $k\_selected$ é inicializado com os K primeiros valores do vetor de entrada.

A variável $largest$ é utilizada para armazenar o maior elemento do vetor $k\_selected$. Para encontrar esse elemento, é utilizada a função $blockMaxReduction$, que é implementada da seguinte forma: primeiro, todas as *threads* do bloco percorrem o vetor $k\_selected$, mantendo o maior elemento. Então, acontece uma redução à nível de *warp*, na qual as *lanes* trocam suas variáveis utilizando registradores para encontrar o maior valor. As *warps* armazenam os valores encontrados na *shared memory* e, por fim, uma *warp* faz a reduções desses valores e guarda o resultado final na variável $largest$.

A variável $reductionIsNeeded$, que é inicializada com *false* na terceira linha, fica na *shared memory* e é utilizada para informar quando a variável $largest$ está desatualizada. Quando isso acontece, todas as *threads* devem fazer a redução para encontrar o novo maior elemento do vetor $k\_selected$ e atualizar a variável $largest$.

A parte principal do algoritmo está nas linhas 4 a 16. Cada *thread* lê um valor do vetor de entrada e compara seu valor com o valor da variável $largest$. Se for menor, a *thread* atualiza o vetor $k\_selected$ e armazena *true* na variável $reductionIsNeeded$. Em seguida, todas as *threads* verificam se há a necessidade de fazer uma redução. A função $sync\_threads$ é necessária na linha 11 para garantir que a variável $reductionIsNeeded$ estará atualizada quando for feita a leitura. Se for preciso, a redução é feita e a variável $largest$ é atualizada. Então as *threads* voltam a percorrer o vetor de entrada.

Devido à necessidade de sincronizar as *threads* para fazer o *K-selection*, só é possível fazer a seleção de um ponto de consulta por bloco. Portanto, o *kernel* do *K-selection* é lançado com número de blocos igual à quantidade de pontos de Q que receberam uma busca parcial pelo KNN. Isso faz com que seja muito difícil implementar um *K-selection* que alcance o máximo de ocupação da GPU, uma vez que o número de pontos de Q que recebem a busca parcial é sempre menor que o número de blocos para alcançar ocupação máxima da GPU.

5 ANÁLISE DOS ALGORITMOS

Neste capítulo é analisado o desempenho do algoritmo SBQ-KNN, descrito no capítulo anterior. Primeiramente, o SBQ-KNN é comparado com a versão do KNN em *cluster* de CPUs. Em seguida, é feita a comparação com o ANN-RSFK. O objetivo é entender como o desempenho do KNN exato se comporta em relação ao desempenho de um algoritmo de busca por vizinhos aproximados. Por último, o SBQ-KNN é comparado com os dois algoritmos da biblioteca FAISS. Todos os algoritmos que serão comparados com o SBQ-KNN foram apresentados no Capítulo 3.

5.1 METODOLOGIA

Para os algoritmos que executam em GPU, a máquina utilizada para realizar os experimentos possui um processador Intel Xeon Silver 4314 @ 2.40GHz com 16 núcleos, 32GB de RAM e uma GPU NVIDIA A4500 que possui 56 multiprocessadores CUDA. O sistema operacional da máquina é o Linux Ubuntu 20.04.3 LTS e as implementações foram compiladas com a versão 11.7 do CUDA.

Cada multiprocessador pode executar 1536 *threads* e cada bloco pode ter no máximo 1024. Portanto, os blocos utilizados para calcular o KNN devem conter 768 *threads*. Assim, cada multiprocessador da GPU executará 2 blocos, atingindo o limite máximo de execução de cada multiprocessador. Considerando que a GPU possui 56 multiprocessadores, serão lançados 112 blocos por kernel, ocupando toda a GPU.

O número de *threads* para o *K-Selection* varia dependendo da quantidade de blocos. Se a quantidade de blocos lançados for menor que a quantidade de multiprocessadores, que neste caso é 52, cada bloco é lançado com 1024 *threads*.

Se o número de blocos for maior que 52, e ainda forem utilizadas 1024 *threads*, a GPU só executará um bloco por multiprocessador, pois não haverá recursos para executar dois blocos, o que poderia dobrar o tempo de execução do *K-Selection*. Por esse motivo, quando o número de blocos é maior que 52, a quantidade de *threads* por bloco é reduzida para 768.

Para o algoritmo de KNN paralelo em CPU, foi utilizado um *cluster* que possui 8 processadores Intel(R) Xeon(R) E5462 @ 2.80GHz, com 8 núcleos, 32 GB de RAM e o sistema operacional Linux Ubuntu 20.04 LTS. A versão da biblioteca OpenMPI utilizada é 4.0.3. O tempo foi medido no processo principal e apenas para o cálculo do KNN, assim como descrito no artigo (Rieger e Zola, 2023).

Para garantir a consistência dos resultados, cada experimento foi repetido 10 vezes e a média dos tempos totais de cada algoritmo foi reportada. Também foram calculados os intervalos de confiança com nível de confiança de 95% e não foram observadas variações maiores que 1% em relação à média dos tempos de execução.

5.1.1 Base de Dados

Os experimentos foram realizados utilizando bases de dados artificiais geradas aleatoriamente, usando a função *rand* padrão da linguagem C++. Os tamanhos das bases de dados e o número de consultas variam entre os diferentes experimentos.

Para comparar a implementação do KNN em GPU com a implementação do KNN em CPU, foram gerados 700 mil pontos de 128 dimensões. Para a comparação entre o SBQ-KNN e o ANN-RSFK, foi gerada uma base de dados de 300 mil pontos de 128 dimensões.

Por fim, para comparar o SBQ-KNN com os algoritmos da biblioteca FAISS, foram geradas três bases de dados artificiais mas com tamanhos de bases de dados reais. A primeira possui o tamanho da base de dados MNIST (Deng, 2012), com 70.000 pontos de 784 dimensões, a segunda possui o tamanho da ImageNet (Deng et al., 2009), com 1.275.219 pontos de 128 dimensões, e a última possui o tamanho da base de dados GoogleNews300 (Google, 2013), com 3 milhões de pontos de 300 dimensões.

5.2 COMPARAÇÃO ENTRE O SBQ-KNN E O KNN PARALELO EM CLUSTER DE CPU

Para executar o KNN CPU, foram utilizados 64 processos, 8 processos para cada processador do *cluster*. Os resultados estão na Tabela 5.1 e na Figura 5.1. Foram realizados testes com os valores de 512, 1024, 2048, 4096, 8192 para os tamanhos do conjunto Q. O K foi fixado em 2048.

	BASE DE DADOS COM 700 MIL PONTOS, 128 DIMENSÕES				
K	32				
|Q|	512	1024	2048	4096	8192
SBQ-KNN	**0,25**	**0,42**	**0,78**	**1,44**	**2,54**
KNN em CLUSTER	4,07	8,06	15,99	31,99	63,68
Aceleração	**16,3**	**19,2**	**20,5**	**22,2**	**25,1**

Tabela 5.1: Tempos de execução em segundos para o SBQ-KNN e o KNN em CPU.

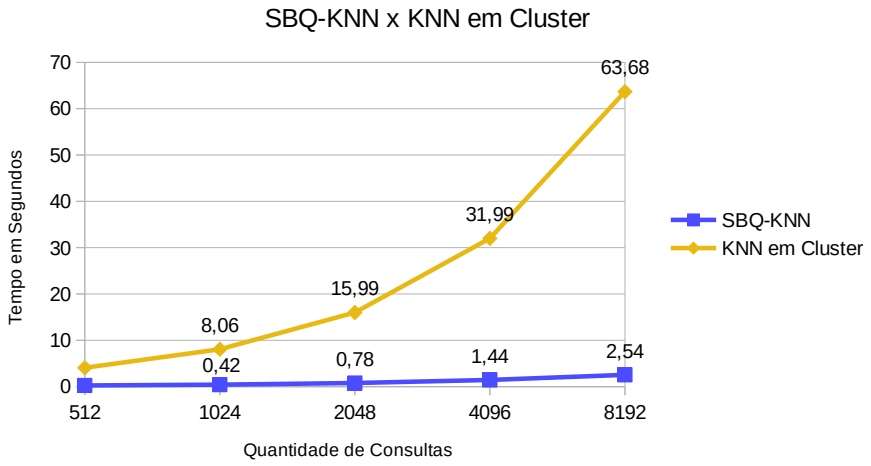


Figura 5.1: Gráfico com a comparação entre o SBQ-KNN e o KNN em CPU.

Com esses resultados é possível notar que, mesmo utilizando 8 processadores, o SBQ-KNN superou o KNN em CPU em 25 vezes para 8192 pontos de consulta. Além disso, é possível notar que o KNN em GPU escala melhor do que o KNN em CPU, pois a aceleração do algoritmo aumenta com o tamanho do conjunto Q.

5.3 COMPARAÇÃO ENTRE O SBQ-KNN E O ANN-RSFK

Os testes foram realizados com $|Q|$ igual a 1, 16, 32, 64 e 128 e o valor do K foi fixado em 32. A quantidade de árvores que o ANN-RSFK utiliza para particionar o espaço de busca

também foi variado. Os testes foram realizados com três quantidades de árvores: 25, 50 e 100. Para cada um desses parâmetros, foi calculada a precisão média do algoritmo.

É importante mencionar que, antes que o ANN-RSFK comece a executar as buscas, é necessário um tempo para criar as árvores de partições, mas esse tempo não foi levado em consideração. Os resultados dos testes estão na Tabela 5.2 e na Figura 5.2.

	BASE DE DADOS 300 MIL PONTOS, 128 DIMENSÕES				
K	32				
|Q|	1	16	32	64	128
SBQ-KNN	**0,39**	**3,28**	**7,47**	**14,74**	**20,95**
ANN-RSFK (Accuracy = 31,75%)	501,80	513,81	514,07	520,59	521,21
ANN-RSFK (Accuracy = 48,92%)	1001,30	1024,04	1034,28	1035,52	1048,58
ANN-RSFK (Accuracy = 61,41%)	1985,59	2067,23	2095,04	2100,60	2081,48
Aceleração	**1286,7**	**156,6**	**68,8**	**35,3**	**24,9**

Tabela 5.2: Tempos de execução em milissegundos para o SBQ-KNN e o ANN-RSFK.

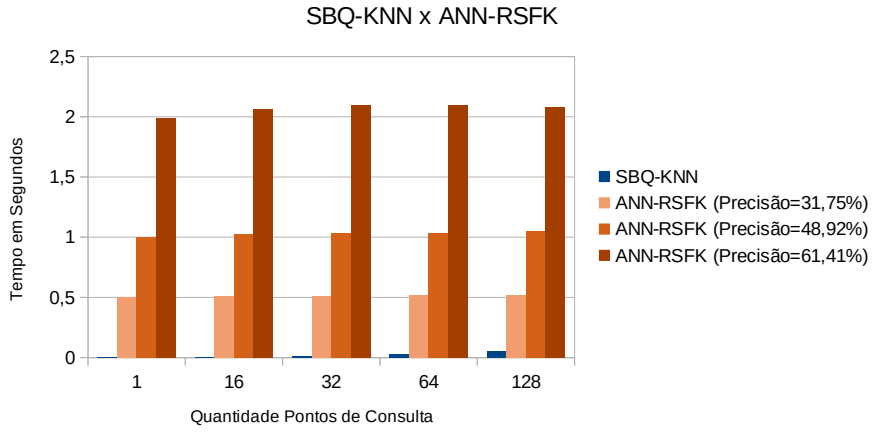


Figura 5.2: Comparação do tempo de execução entre os algoritmos SBQ-KNN e ANN-RSFK.

Primeiramente, é possível notar que o SBQ-KNN apresenta um tempo de execução menor do que o ANN-RSFK, apresentando uma aceleração de mais de 1286 vezes em relação ao melhor tempo do ANN-RSFK para um ponto de consulta. Isso acontece porque o SBQ-KNN é otimizado para realizar muitas consultas em conjuntos de dados muito maiores do que aqueles utilizados nos testes. Dessa forma, a latência de percorrer as árvores que particionam o espaço de busca domina completamente o tempo total do algoritmo. Isso pode ser verificado pelo fato de que o tempo de consulta do ANN-RSFK não aumenta com o $|Q|$, indicando que o algoritmo ainda não está sendo bem aproveitado.

5.4 COMPARAÇÃO ENTRE O SBQ-KNN E OS ALGORITMOS DA BIBLIOTECA FAISS

Para essa análise, os valores de $|Q|$ escolhidos para os experimentos foram 1, 16 e 32. Os valores de K escolhidos foram 32, 64 e 128. As funções da biblioteca FAISS que foram utilizadas são: o FLATL2, que implementa o *index flat*, e o IndexIVFFlat, que implementa

IndexIVF. Ambos algoritmos foram apresentados no Capítulo 3. Como o IndexIVFFlat depende do número de partições, os testes foram executados variando esse parâmetro em 2, 5 e 11. Para cada um desses valores, foi calculada a precisão média de todos os experimentos. Os resultados estão na Tabela 5.3.

	BASE DE DADOS COM 70.000 PONTOS, 784 DIMENSÕES								
K	32			64			128		
|Q|	1	16	32	1	16	32	1	16	32
SBQ-KNN	**0,41**	3,07	6,09	**0,44**	3,13	6,7	**0,51**	3,34	6,84
Index Flat	0,65	**0,69**	**0,71**	0,69	**0,75**	**0,76**	0,72	**0,77**	**0,8**
IndexIVF (Accuracy=18.7 %)	4,4	4,19	4,21	4,03	4,02	4,11	4,07	3,92	4,07
IndexIVF (Accuracy=24.5 %)	7,87	7,52	7,54	7,83	7,79	7,95	7,32	7,02	7,29
IndexIVF (Accuracy=51.6 %)	14,25	13,64	14,15	14,79	14,14	14,42	15,32	14,64	14,68

	BASE DE DADOS COM 1.275.219 PONTOS, 128 DIMENSÕES								
K	32			64			128		
|Q|	1	16	32	1	16	32	1	16	32
SBQ-KNN	**1,22**	13,4	30,03	**1,36**	13,61	30,33	**1,6**	13,85	30,96
Index Flat	5,12	**5,44**	**5,77**	5,39	**5,57**	**5,87**	5,56	**5,73**	**6,06**
IndexIVF (Accuracy=18.7 %)	10,06	9,54	9,56	9,28	8,9	9,24	10,78	10,76	10,97
IndexIVF (Accuracy=24.5 %)	22,03	20,99	21,04	21,20	20,26	20,66	20,37	19,52	20,27
IndexIVF (Accuracy=51.6 %)	33,79	32,23	32,29	32,54	31,10	31,69	31,28	29,96	31,09

	BASE DE DADOS COM 3.000.000 PONTOS, 300 DIMENSÕES								
K	32			64			128		
|Q|	1	16	32	1	16	32	1	16	32
SBQ-KNN	**6,32**	71,96	155,35	**6,51**	72,17	156,74	**6,66**	72,85	156,04
Index Flat	15,84	**16,82**	**17,28**	15,97	**17,04**	**17,49**	16,4	**17,47**	**17,96**
IndexIVF (Accuracy=18.7 %)	69,7	66,27	66,47	67,32	64,00	65,17	64,93	61,73	63,86
IndexIVF (Accuracy=24.5 %)	152,26	145,4	145,86	147,12	140,36	142,96	141,97	135,32	140,05
IndexIVF (Accuracy=51.6 %)	317,68	303,74	304,56	307,30	293,60	298,94	296,92	283,45	293,31

Tabela 5.3: Tempos de execuções em milissegundos para o SBQ-KNN e os algoritmos da biblioteca FAISS.

Primeiramente, é possível notar que o algoritmo *IndexIVF* não apresentou bons resultados, mesmo para níveis baixo de precisão. Isso acontece porque, assim como o ANN-RSFK, o *IndexIVF* é um algoritmo de ANN que depende de grandes volumes de dados para ser eficiente.

Outro resultado interessante é que o algoritmo implementado neste trabalho apresenta excelentes resultados quando existe apenas um único ponto no conjunto Q, com aceleração de mais de duas vezes para duas bases de dados. Esses resultados estão destacados na Figura 5.3.

Com o objetivo de identificar os pontos em que o SBQ-KNN supera o *index flat*, foram realizados testes variando o $|Q|$ de 1 até 4 na base de dados com 70 mil pontos. Os resultados

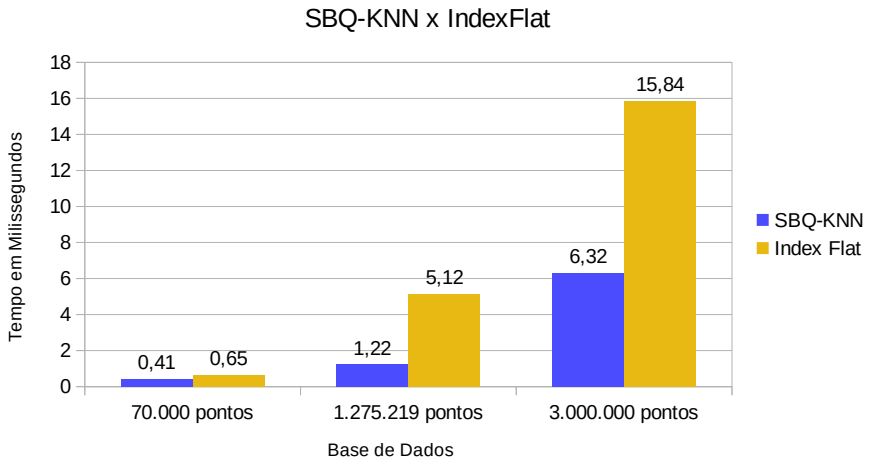


Figura 5.3: Comparação do tempo de execução entre os algoritmos SBQ-KNN e ANN-RSFK.

estão apresentados na Figura 5.4. Verifica-se que o SBQ-KNN supera o *index flat* quando o conjunto Q possui até dois elementos. No entanto, observou-se que o algoritmo proposto não apresentou uma escalabilidade tão forte quanto a do *index flat*.

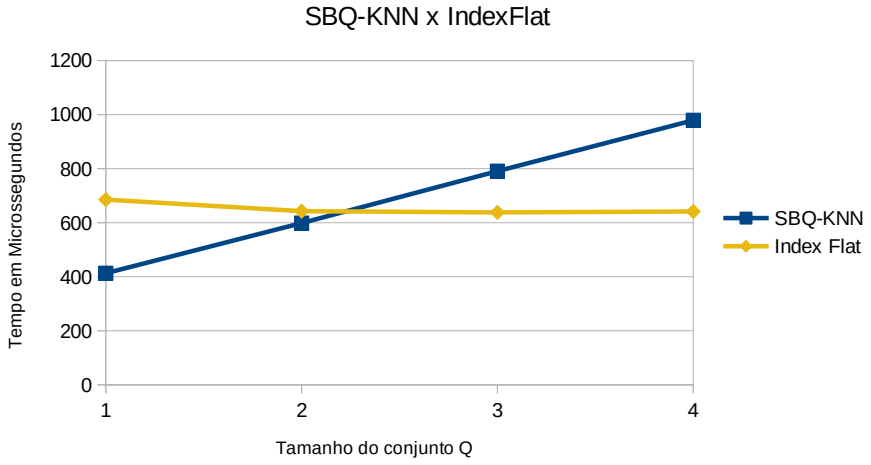


Figura 5.4: Comparação do tempo de execução entre os algoritmos SBQ-KNN e *index-flat*.

6 CONCLUSÃO

Este trabalho apresentou uma proposta de implementação eficiente em GPU do algoritmo de aprendizado de máquina *K-Nearest Neighbor*. Esse algoritmo foi otimizado para poucos pontos de consulta e, por isso, foi chamado de *Small Batch Query K-nearest neighbor* (SBQ-KNN). Junto com o algoritmo, também foi proposta uma implementação para o algoritmo *K-Selection*, que é usado pelo KNN em situações específicas.

Com o propósito de avaliar a eficiência do SBQ-KNN, seu desempenho foi comparado a quatro algoritmos. O primeiro deles consistiu em uma versão paralela do KNN em CPU. Nessa análise, destacou-se que o SBQ-KNN demonstrou uma aceleração de 25 vezes para 8192 pontos de consulta.

A segunda comparação foi realizada com o algoritmo ANN-RSFK, que é implementado em GPU, assim como o SBQ-KNN. Nessa análise, foi possível observar uma aceleração de mais de 1286 vezes em relação ao melhor tempo do ANN-RSFK para um ponto de consulta. Isso ocorre porque os algoritmos de ANN, em geral, utilizam estruturas de dados complexas que adicionam uma grande latência ao tempo total do algoritmo. Portanto, esses algoritmos dependem de um grande volume de dados para se tornarem eficientes.

Por último, o algoritmo proposto é comparado com dois algoritmos da biblioteca FAISS, biblioteca que implementa diversos algoritmos que realizam buscas por similaridade em GPU. O primeiro algoritmo é o *IVFFlat*, um algoritmo de ANN que apresentou resultados similares ao ANN-RSFK, sendo superado para poucos pontos no conjunto de consulta.

O outro algoritmo é o *index flat*, um algoritmo de KNN exato. O SBQ-KNN mostrou estar bem otimizado para poucos pontos no conjunto de consulta, superando o *index flat* em todas as bases de dados testadas para um ponto no conjunto de consulta. Porém, o SBQ-KNN não obteve a mesma escalabilidade que o *index flat*, apresentando tempos de execução maiores para mais pontos no conjunto de consulta.

REFERÊNCIAS

Baxter, S. (2013). Performance. https://moderngpu.github.io/performance.html. Acessado em 14/07/2023.

Cordeiro, M., Meyer, B. e Zola, W. (2023). KNN exato em GPU. Em *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*, páginas 17–20, Porto Alegre, RS, Brasil. SBC.

CUDA (2023). CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Acessado em 07/07/2023.

Daghaghi, S., Meisburger, N., Zhao, M., Wu, Y., Gobriel, S., Tai, C. e Shrivastava, A. (2021). Accelerating SLIDE deep learning on modern CPUs: Vectorization, quantizations, memory optimizations, and more.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. e Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. Em *2009 IEEE Conference on Computer Vision and Pattern Recognition*, páginas 248–255.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142.

FAISS (2023). Faiss repository. https://github.com/facebookresearch/faiss. Acessado em 07/07/2023.

Google (2013). Google Code Archive. https://code.google.com/archive/p/word2vec/. Acessado em 15/08/2023.

Gu, Y., Liu, G., Qi, J., Xu, H., Yu, G. e Zhang, R. (2017). The moving K diversified nearest neighbor query. Em *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, páginas 31–32.

Güting, R., Behr, T. e Xu, J. (2010). Efficient k-nearest neighbor search on moving object trajectories. *The Vldb Journal - VLDB*, 19.

Iwerks, G. S., Samet, H. e Smith, K. (2003). Continuous k-nearest neighbor queries for continuously moving points with updates. Em *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, página 512–523. VLDB Endowment.

Johnson, J., Douze, M. e Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Keita, Z. (2022). Classification in machine learning: An introduction. https://www.datacamp.com/blog/classification-machine-learning. Acessado em 07/07/2023.

LaViale, T. (2023). Deep Dive on KNN: Understanding and Implementing the K-Nearest Neighbors Algorithm. https://arize.com/blog-course/knn-algorithm-k-nearest-neighbor/#:~:text=The%20time%20complexity%20of%20the,other%20point%20in%20the%20dataset. Acessado em 14/07/2023.

Li, J., Ni, C., He, D., Li, L., Xia, X. e Zhou, X. (2022). Efficient KNN query for moving objects on time-dependent road networks. *The VLDB Journal*, 32(3):575–594.

Lin, H., Shen, Z., Zhou, H., Liu, X., Zhang, L., Xiao, G. e Cheng, Z. (2020). KNN-Q learning algorithm of bitrate adaptation for video streaming over HTTP. Em *2020 Information Communication Technologies Conference (ICTC)*, páginas 302–306.

Liu, X. e Ferhatosmanoğlu, H. (2003). Efficient k-NN search on streaming data series. Em Hadzilacos, T., Manolopoulos, Y., Roddick, J. e Theodoridis, Y., editores, *Advances in Spatial and Temporal Databases*, páginas 83–101, Berlin, Heidelberg. Springer Berlin Heidelberg.

Mahesh, B. (2020). Machine learning algorithms-a review. *International Journal of Science and Research (IJSR)*, 9(1):381–386.

Meyer, B., Pozo, A. e Zola, W. (2022). ANN-RSFK: Busca genérica de similaridade em GPU. Em *Anais da XXII Escola Regional de Alto Desempenho da Região Sul*, páginas 89–90, Porto Alegre, RS, Brasil. SBC.

Meyer, B. H. e Nunan Zola, W. M. (2023). Towards a GPU accelerated selective sparsity multilayer perceptron algorithm using K-nearest neighbors search. Em *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ICPP Workshops '22, New York, NY, USA. Association for Computing Machinery.

Oh, F. (2012). What is CUDA? `https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/`. Acessado em 07/07/2023.

Rieger, H. e Zola, W. (2023). Algoritmo KNN paralelo em cluster com MPI. Em *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*, páginas 21–24, Porto Alegre, RS, Brasil. SBC.

Song, Z. e Roussopoulos, N. (2001). K-nearest neighbor search for moving query point. Em Jensen, C. S., Schneider, M., Seeger, B. e Tsotras, V. J., editores, *Advances in Spatial and Temporal Databases*, páginas 79–96, Berlin, Heidelberg. Springer Berlin Heidelberg.

Trabelsi, E. (2020). Comprehensive guide to approximate nearest neighbors algorithms. `https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6`. Acessado em 07/07/2023.

Yeh, M.-Y., Wu, K.-L., Yu, P. e Chen, M.-S. (2008). LeeWave: Levelwise distribution of wavelet coefficients for processing kNN queries over distributed streams. *PVLDB*, 1.