KNN paralelo em GPU para grandes volumes de dados com agregação de consultas

Michel B. Cordeiro[1], **Wagner M. Nunan Zola** [1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

michel.brasil.c@gmail.com, wagner@inf.ufpr.br

Abstract. *Machine learning algorithms often come with a high computational cost. Therefore, several approaches can be employed to accelerate these algorithms. One of the strategies involves the utilization of graphics processing units (GPU). In this scenario, this article presents an efficient GPU implementation of the K-Nearest Neighbor (KNN) for exact queries. The proposed algorithm was compared with algorithms available in the widely used FAISS library for GPU-based similarity search. Experiments have demonstrated that our new exact KNN algorithm outperforms FAISS for large datasets when there is only one point in the search set. The new kernel also presents better results with the aggregation of queries, being a good alternative implementation for applications that can perform queries in parallel for small batches, where it obtained an acceleration of up to 4.76 times compared to the exact algorithm from the FAISS library, and up to 10.46 times in relation to the approximate algorithm.*

Resumo. *Algoritmos de aprendizado de máquina geralmente apresentam um alto custo computacional. Várias abordagens podem ser empregadas para acelerar esses algoritmos. Uma das estratégias envolve a utilização de unidades de processamento gráfico (GPU). Nesse cenário, este artigo apresenta uma implementação eficiente para processamento do algoritmo exato para consultas K-Nearest Neighbor (KNN) em GPU. O algoritmo proposto foi comparado com algoritmos disponíveis na biblioteca FAISS amplamente utilizada para busca de similaridade baseada em GPU. Experimentos demonstraram que nosso novo algoritmo para KNN exato supera o FAISS para grandes conjuntos de dados quando há apenas um ponto no conjunto de pesquisa. O novo kernel também apresenta melhores resultados com a agregação de consultas, sendo uma boa alternativa para uso em aplicações que podem realizar consultas paralelas em pequenos lotes, onde obteve aceleração de até 4.76 vezes em relação ao algoritmo exato da biblioteca FAISS, até 10.46 vezes em relação ao algoritmo aproximado.*

1. Introdução

Algoritmos de aprendizado de máquina tem se tornado cada vez mais importantes nos últimos anos. Esses algoritmos têm sido utilizado para diversos propósitos, como mineração de dados, processamento de imagens, análise preditiva, entre outros [Mahesh 2020]. O *K-nearest neighbor* (KNN), ou K-vizinhos mais próximos, é um simples algoritmo de aprendizado de máquina que pode ser usado para resolver problemas

de regressão e classificação [Mahesh 2020]. Seu objetivo consiste em encontrar, em um conjunto de referência, os K pontos mais próximos de cada ponto em um conjunto de consulta [Cordeiro et al. 2023]. Dessa forma, o KNN recebe dois conjuntos de pontos como entrada: um conjunto P com pontos de referência e um conjunto Q de pontos a serem consultados. A saída do algoritmo é uma matriz $|Q| \times K$, na qual cada linha da matriz representa os K pontos mais próximos de um ponto do conjunto Q.

A complexidade do KNN para um único ponto de consulta é $O(N \times D)$ [Ukey et al. 2023], no qual N é o tamanho do conjunto de referência e D é o número de dimensões dos pontos. Isso ocorre porque, para encontrar os pontos mais próximos, é necessário calcular a distância entre o ponto consultado e cada outro ponto no conjunto de referência. À medida que o número de dimensões e o tamanho dos conjuntos aumentam, aumenta o custo computacional do KNN. Por isso, diversas estratégias podem ser necessárias para acelerar o algoritmo. Uma delas é particionar o espaço de busca, reduzindo assim o número de distâncias a serem calculadas. O problema dessa abordagem é a possibilidade de alguns dos K pontos mais próximos do ponto consultado não estarem na mesma partição. Portanto, algoritmos que utilizam dessa estratégia são chamados de busca de vizinhos aproximados, ou *Approximate Nearest Neighbor search* (ANN), pois apresentam uma solução aproximada para o KNN [Meyer et al. 2022]. Outra maneira de acelerar o KNN é aproveitar recursos de *hardware*, como o paralelismo em GPU.

Existem cenários em que o KNN é utilizado em aplicações de inteligência artificial, em que várias consultas podem ser feitas em paralelo de maneira agregada. Quando o conjunto de consultas é muito grande, por exemplo, do mesmo tamanho do conjunto de dados, o custo computacional fica muito alto, demandando o uso de algoritmos aproximados [Meyer et al. 2021]. No entanto, algoritmos aproximados incorrem em custos adicionais na criação de estruturas de indexação de dados para aceleração de consultas. Por outro lado, algumas aplicações podem fazer buscas agregadas em pequenos conjuntos de pesquisa [Meyer and Nunan Zola 2023] com menor custo por consulta, neste caso evitando o tempo adicional de criação de estruturas para aceleração das buscas. Nesse contexto, este trabalho apresenta uma implementação do algoritmo KNN exato utilizando paralelismo em GPU. O novo *kernel* para KNN exato aqui apresentado obteve bons resultados com a agregação de consultas, sendo uma boa alternativa para uso em aplicações que podem realizar consultas paralelas em pequenos lotes.

O restante do artigo está organizado da seguinte forma: na Seção 2 estão os trabalhos relacionados, onde são apresentados os algoritmos que motivaram a criação deste trabalho, bem como os algoritmos que foram utilizados para análise comparativa do KNN proposto. A Seção 3 descreve os detalhes da implementação do algoritmo. Na Seção 4 é apresentada a metodologia dos experimentos. Na Seção 5 é feita a análise dos resultados e, por fim, a Seção 6 apresenta as conclusões.

2. Trabalhos Relacionados

Existem diversas aplicações para o algoritmo KNN em que não é possível realizar buscas em grandes lotes, resultando em um tamanho reduzido para o conjunto de consulta. Um exemplo dessas aplicações é quando os pontos do conjunto de consulta são móveis [Song and Roussopoulos 2001, Gu et al. 2017]. Nesse cenário, o tamanho do conjunto de consulta é limitado pela quantidade de objetos em movimento. Esse problema surge em

aplicações de sistemas de informações geográficas e sistemas de posicionamento global, como encontrar os hotéis mais próximos de um viajante ou os postos de gasolina mais próximos de um motorista, por exemplo.

Uma variação desse problema consiste em realizar consultas em um conjunto de referência com pontos que também sofrem alterações ao longo do tempo [Iwerks et al. 2003, Güting et al. 2010, Li et al. 2022]. Isso implica que cada consulta será executada em um conjunto de referência diferente, 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. Também é possível mencionar problemas relacionados ao uso do algoritmo KNN em fluxos de dados [Liu and 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.

Além disso, existem casos em que o KNN é utilizado por algoritmos de inteligência artificial. Um exemplo disso é o algoritmo SLIDE-GPU [Meyer and Nunan Zola 2023], uma rede neural de classificação extrema que utiliza um algoritmo de ANN para encontrar quais neurônios de uma determinada camada serão ativados. Nessa aplicação, as consultas KNN foram feitas uma a uma, mas poderiam ser agregadas em pequenos lotes, com melhoria de desempenho. Outro exemplo é o algoritmo KNN-Q [Lin et al. 2020], que emprega o KNN para selecionar os estados mais próximos quando o algoritmo de inteligência artificial encontra um estado no qual 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. Os trabalhos citados nesta seção exemplificam cenários nos quais a busca em grandes lotes não é viável. Outra situação possível está relacionada a aplicações voltadas para a *data streaming*, em que as instâncias são processadas uma a uma, ou quando é possível agregar várias consultas sem introduzir grandes latências, sendo inviável agregar grandes lotes de consultas nesses casos. Isso evidencia possíveis aplicações para as implementações propostas neste trabalho.

Existem diversos trabalhos que utilizam GPUs para acelerar o KNN. O artigo ”KNN Exato em GPU” [Cordeiro et al. 2023] apresenta duas implementações do KNN em GPU utilizando a linguagem CUDA (NVIDIA). A primeira é chamada de ”Um Ponto Por *Kernel*” (PPK). Essa versão utiliza todos os recursos da GPU para computar um ponto do conjunto de busca por vez, lançando um *kernel* CUDA para cada ponto. A segunda implementação do algoritmo utiliza um bloco de *threads* para calcular cada ponto e, por isso, é chamado de ”Um Ponto Por Bloco”(PPB). Nessa versão, é lançado apenas um *kernel* CUDA, cujo número de blocos é igual a quantidade de pontos no conjunto de consulta. Esses dois algoritmos serviram de base para a implementação do KNN proposto neste trabalho.

Uma implementação do KNN em GPU apresentada em [Velentzas et al. 2020] realiza o particionamento dos dados construindo uma estrutura de indexação baseada em *bins*. Essa abordagem reduz o espaço de busca e a quantidade de distâncias calculadas, sendo mais apropriada quando os dados estão uniformemente distribuídos nas regiões de particionamento. No entanto, é importante observar que esse algoritmo possui um tempo adicional de construção e ordenação dos dados que devem ser considerados, como nos

trabalhos de KNN aproximados.

Dois algoritmos implementados em GPU para resolver o KNN são apresentados em [Johnson et al. 2019]. O primeiro algoritmo, chamado *index flat*, 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. Segundo os autores, o algoritmo de *K-selection* apresentado 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, denominado *IndexIVF*, é um algoritmo aproximado (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 ser utilizado. Sendo assim, o *IndexIVF* recebe o número de partições como parâmetro. Quanto maior for o número de partições, mais rápido será realizada a busca, porém menor será a precisão do algoritmo. Esses dois algoritmos descritos no artigo estão disponíveis na biblioteca FAISS e são amplamente utilizados para realizar buscas por similaridade em GPU.

3. Implementação

Os algoritmos foram implementados em C++ utilizando CUDA. A estratégia utilizada neste trabalho foi dividir o conjunto de busca entre os *warps* do bloco de *threads*. Por esse motivo, o algoritmo é chamado de "Um Ponto Por Warp" (PPW). O *kernel* PPW foi codificado no modelo de *threads* persistentes [Gupta et al. 2012]. A função principal do *kernel* PPW está apresentada no Algoritmo 1. Ela é executada na CPU e seu propósito é chamar as funções *kernel* que farão a execução do KNN na GPU. Essa função recebe dois conjuntos como entrada: o conjunto de referência P, o conjunto de consulta Q e o valor do K, e sua saída é a matriz $knnExato$, onde a linha i contém os K-vizinhos mais próximos do ponto $q_i \in Q$.

Algorithm 1 Função Principal

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

1: $knnParcial \leftarrow PPWkernel << nblocos, nthreads >> (Q, P, k)$
2: $cudaDeviceSynchronize()$
3: $knnExato \leftarrow Kselection << |Q|, nthreads >> (knnParcial, k)$
4: $cudaDeviceSynchronize()$
5: **return** $knnExato$

O algoritmo $PPWkernel$ realiza o cálculo das distâncias na GPU. Para evitar a necessidade de sincronizar *threads* de blocos diferentes, cada bloco realiza a consulta apenas em uma partição do conjunto P. O problema dessa abordagem é que cada bloco encontrará os K-vizinhos mais próximos apenas na partição de pontos que recebeu, gerando uma matriz com o resultado parcial da busca. Cada linha dessa matriz representa um elemento de Q e contém os K vizinhos mais próximos encontrados por cada bloco de threads. Portanto, os K primeiros elementos da linha consistem nos K vizinhos encontrados pelo primeiro bloco, os K elementos seguintes foram encontrados pelo segundo bloco e assim por diante. Dessa forma, para gerar a matriz com o resultado exato, será necessário utilizar um algoritmo que seleciona os K pontos mais próximos em cada linha da matriz. Isso é feito pelo algoritmo $Kselection$. Para sincronizar as duas funções,

é utilizada a função $cudaDeviceSynchronize()$, que garante que todas as funções *kernel* lançadas anteriormente na GPU terminaram de executar. A saída do $Kselection$ é a matriz $KnnExato$, com o resultado final do KNN.

O $PPWkernel$ está representado no Algoritmo 2. Ele recebe como entrada os conjuntos P e Q e o valor de K. Os conjuntos são armazenados em matrizes em que as linhas representam pontos, e as colunas representam os valores para cada dimensão do ponto. A saída do algoritmo é uma matriz armazenada na memória global que contém os K vizinhos mais próximos encontrados por cada bloco dentro da partição que recebeu.

Algorithm 2 PPWkernel

Input: Q,P,K **Output:** $globalKNN$

1: $partition\_size \leftarrow |P| \div nblocos$
2: $P\_begin \leftarrow blockIdx \times partition\_size$
3: $P\_end \leftarrow P\_begin + partition\_size$
4: $q \leftarrow warpIdx \bmod |Q|$
5: $stride \leftarrow nwarps \div |Q|$
6: $p_i \leftarrow P\_begin + \lfloor warpIdx \div |Q| \rfloor$
7: **for** $i \leftarrow 0$ **to** K **do**
8: $\quad sharedKNN[q][i] \leftarrow warpDistance(q_i, p_i)$
9: $\quad p_i \leftarrow p_i + stride$
10: **end for**
11: $farthestNeighbor[q] \leftarrow warpMaxReduction(sharedKNN[q])$
12: **while** $p_i < P\_end$ **do**
13: $\quad d \leftarrow warpDistance(q_i, p_j)$
14: $\quad$ **if** $d < farthestNeighbor[q]$ **then**
15: $\quad\quad sharedKNN[q] \leftarrow sharedKNN[q] - farthestNeighbor[q]$
16: $\quad\quad sharedKNN[q] \leftarrow sharedKNN[q] \cup p_j$
17: $\quad\quad farthestNeighbor[q] \leftarrow warpMaxReduction(sharedKNN[q])$
18: $\quad$ **end if**
19: $\quad p_i \leftarrow p_i + stride$
20: **end while**
21: $globalKNN[q][blockIdx * k : (blockIdx + 1) * k] \leftarrow sharedKNN[q]$
22: **return** $globalKNN$

Cada bloco de *threads* possui um índice que é armazenado na variável $blockIdx$. Essa variável é utilizada para delimitar a partição do conjunto P em que cada bloco executará a busca. Isso é feito nas três primeiras linhas do algoritmo. Para encontrar qual ponto de Q cada *warp* irá computar, calcula-se o resto da divisão do seu índice pela quantidade de pontos no conjunto Q, utilizando o operador mod na linha 4. Dessa forma, se a quantidade de pontos em Q for menor que a quantidade de *warps*, existirá mais de um *warp* computando o mesmo ponto de Q. Portanto, é preciso calcular número de pontos que serão incrementados a cada distância calculada, o que é feito na linha 5. Para que dois *warps* que processam o mesmo ponto q não calculem distâncias para os mesmos pontos de P, é necessário incrementar o início da partição em $\lfloor warpIdx \div |Q| \rfloor$.

Com o objetivo de melhorar a eficiência do algoritmo, cada bloco armazena as K menores distâncias calculadas em uma matriz na memória compartilhada. Isso implica que haverá queda no desempenho se o valor de K for grande o suficiente para a matriz

não caber na memória compartilhada pois, nesse cenário, a matriz será armazenada na memória global da GPU. No entanto, é importante ressaltar que o algoritmo funciona para qualquer valor de $K > 0$. Essa matriz é chamada de $sharedKNN$ e é inicializada com as K primeiras distâncias calculadas. Em seguida, é utilizada uma função de redução para encontrar o maior elemento de cada linha. O resultado da redução é armazenado na variável $farthestNeighbor$, que também é armazenada na memória compartilhada. Sendo assim, existe uma variável $farthestNeighbor$ para cada ponto de Q.

A execução do KNN acontece de fato nas linhas 12 a 20. Sempre que uma distância é calculada, ela é comparada com o valor de $farthestNeighbor$. Se for menor, o *warp* atualiza a linha da matriz e realiza uma redução para encontrar a nova maior distância. Por fim, a matriz $sharedKNN$ é copiada para a matriz $globalKNN$, que será retornada para a função principal.

Tanto o cálculo da distância (função $warpDistance$), quanto a redução (função $warpMaxReduction$) para encontrar o vizinho mais distante, são realizados cooperativamente por *threads* de cada *warp*. Assim, podemos dizer que o $PPWkernel$ tem um enfoque centrado em warps (*warp centric*) [Meyer et al. 2021].

A função $warpDistance$, descrita no Algoritmo 3, recebe dois pontos como entrada e produz como saída a distância entre esses pontos. Para calcular essa distância, cada *lane* percorre as dimensões dos pontos, somando a diferença ao quadrado dos valores encontrados, conforme indicado nas linhas 1 a 7. Em seguida, as *lanes* utilizam a função $\_\_shfl\_xor\_sync()$ para somar as distâncias parciais e produzir a distância final. Ao usar a função $\_\_shfl\_xor\_sync()$, é possível compartilhar dados por meio de registradores, sendo mais eficiente do que utilizar a memória compartilhada.

Algorithm 3 warpDistance

Input: q_i, p_i
Output: $dist$

1: $dist \leftarrow 0$
2: $j \leftarrow laneIdx$
3: **while** $j < D$ **do**
4: $\quad diff \leftarrow q_i[j] - p_i[j]$
5: $\quad dist \leftarrow dist + diff * diff$
6: $\quad j \leftarrow j + warpSize$
7: **end while**
8: $j \leftarrow 1$
9: **while** $j < warpSize$ **do**
10: $\quad sh\_dist \leftarrow \_\_shfl\_xor\_sync(dist, j)$
11: $\quad dist \leftarrow dist + sh\_dist$
12: $\quad j \leftarrow j * 2$
13: **end while**
14: **return** $dist$

Algorithm 4 warpMaxReduction

Input: vec
Output: max

1: $max \leftarrow 0$
2: $j \leftarrow laneIdx$
3: **while** $j < K$ **do**
4: $\quad max \leftarrow Max(max, vec[j])$
5: $\quad j \leftarrow j + warpSize$
6: **end while**
7: $j \leftarrow 1$
8: **while** $j < warpSize$ **do**
9: $\quad sh\_max \leftarrow \_\_shfl\_xor\_sync(max, j)$
10: $\quad max \leftarrow Max(max, sh\_max)$
11: $\quad j \leftarrow j * 2$
12: **end while**
13: **return** max

A redução está descrita no Algoritmo 4 e funciona de forma semelhante. Esse algoritmo recebe um vetor como entrada e retorna o maior elemento contido nele. No contexto deste trabalho, a função recebe uma linha da matriz de distâncias $sharedKNN$ e retorna a maior distância. Para isso, cada *lane* percorre o vetor, mantendo o maior elemento encontrado. Ao terminar de percorrer o vetor, a função $\_\_shfl\_xor\_sync()$ é utilizada

para compartilhar as variáveis e retornar o maior elemento. Vale destacar que cada *lane* realiza $\lceil K \div warpSize \rceil$ comparações no primeiro laço do algoritmo e $\lceil \ln(warpSize) \rceil$ no segundo laço, fazendo com que o algoritmo seja muito eficiente para valores pequenos de K. Manter os *warps* trabalhando juntos impede que haja divergências no fluxo de execução, além de permitir que sempre troquem informações através de registradores.

4. Metodologia

A máquina utilizada para realizar os experimentos possui um processador Intel Xeon Silver 4314 @ 2.40GHz com 16 núcleos (32 *hyperthreads*), 32GB de RAM e uma GPU NVIDIA A4500 que possui 56 multiprocessadores CUDA. O sistema operacional utilizado foi o Linux Ubuntu 20.04.3 LTS e as implementações foram compiladas com a versão 11.7 da biblioteca CUDA.

Na arquitetura dessa GPU, cada multiprocessador (MP) pode executar 1536 *threads*, e cada bloco pode ter no máximo 1024 *threads*. Portanto, os blocos utilizados para calcular o KNN devem conter 768 *threads*. Assim, como o kernel KNN utiliza apenas 42 registradores por *thread*, cada MP da GPU executará 2 blocos de *threads*, atingindo a máxima ocupação de cada multiprocessador nessa configuração, pois a arquitetura da GPU utilizada disponibiliza 65536 registradores por MP. Considerando que a GPU possui 56 multiprocessadores, serão lançados 112 blocos por *kernel*, ocupando toda a GPU.

Os experimentos foram realizados utilizando bases de dados artificiais geradas aleatoriamente, usando a função *rand* padrão da linguagem C++. Os seus tamanhos foram baseados em bases de dados reais, conforme a Tabela 1. Dessa forma, foram gerados bases de dados com os tamanhos do MNIST [Deng 2012], que possui 70 mil pontos de 784 dimensões, ImageNet [Deng et al. 2009], com 1.275.219 pontos de 128 dimensões, e GoogleNews300 [Google 2013], que possui 3 milhões de pontos de 300 dimensões.

Tabela 1. Bases de dados utilizados

Nome	Número de pontos	Número de dimensões
MNIST	70000	784
ImageNet	1275219	128
GoogleNews300	3000000	300

Cada experimento foi repetido 30 vezes, durante os quais a vazão média de consultas por segundo foi medida. Além disso, foram calculados os intervalos de confiança, com nível de confiança de 95%. No entanto, não foram observadas intervalos maiores que 2% em relação à média e, por isso, esses valores não foram reportados. A análise do Kernel PPW foi realizada em quatro etapas. Na primeira etapa (Figura 1), o PPW foi comparado com o Index Flat, variando o valor de K de 10 a 120. Nessa etapa, foi considerada a base de dados com as dimensões do ImageNet e apenas um ponto no conjunto de consulta. Na segunda etapa (Tabelas 2 e 3), o PPW foi comparado com algoritmos de KNN exatos: PPK, PPB e Index Flat, sendo esse último da biblioteca FAISS. Essa etapa considerou as três bases de dados. Os valores de K escolhidos foram 64 e 128. Na terceira etapa (Tabela 4), o PPW foi comparado com o algoritmo de ANN *IndexIVF*, também da biblioteca FAISS, considerado apenas a base de dados GoogleNews300 e o valor do K igual à 128. Como o *IndexIVF* depende do número de partições, os testes

foram executados variando esse parâmetro em 2, 5 e 11. Também foram calculadas as precisões médias e os intervalos com 95% de confiança para cada um desses parâmetros. A precisão foi calculada comparando a saída do IndexFlat com as saídas dos algoritmos de KNN exatos. As saídas dos algoritmos exatos também foram comparadas entre si, e nenhuma discrepância foi encontrada, demonstrando a corretude dos resultados. Por fim, na quarta etapa (Figura 2), o PPW é analisado em relação ao modelo *roofline*, que ilustra os limites superiores de desempenho que um programa pode alcançar com base nos recursos de hardware disponíveis.

5. Resultados e discussão

Ao analisar os resultados da primeira etapa de experimentos (Figura 1), é possível verificar que K possui pouca influência no tempo de execução dos algoritmos, sendo o algoritmo de *K-selection* o principal fator responsável pelo aumento no tempo de execução do KNN PPW. Isso sugere que ainda há espaço para otimização desse algoritmo. A vazão de consultas (# Consultas/s) é apresentada na segunda etapa de experimentos (Tabelas 2 e 3). Inicialmente, observa-se que o PPW supera os demais algoritmos em todos os conjuntos de dados quando o conjunto Q possui menos de 5 elementos. Além disso, o PPW atingiu aceleração de até 4,76 vezes para valor de K igual à 64 na base de dados ImageNet. Também é possível destacar que o PPK apresenta desempenho superior ao Index Flat (FAISS) para conjuntos de dados com poucos pontos de consulta. No entanto, esse algoritmo não escala tão bem quanto o PPW, demonstrando que o PPW é melhor que o PPK em agregar os pontos do conjunto de consulta. Contudo, o PPW não apresenta a mesma escalabilidade que o *Index Flat*, sendo superado à medida que o conjunto Q aumenta.

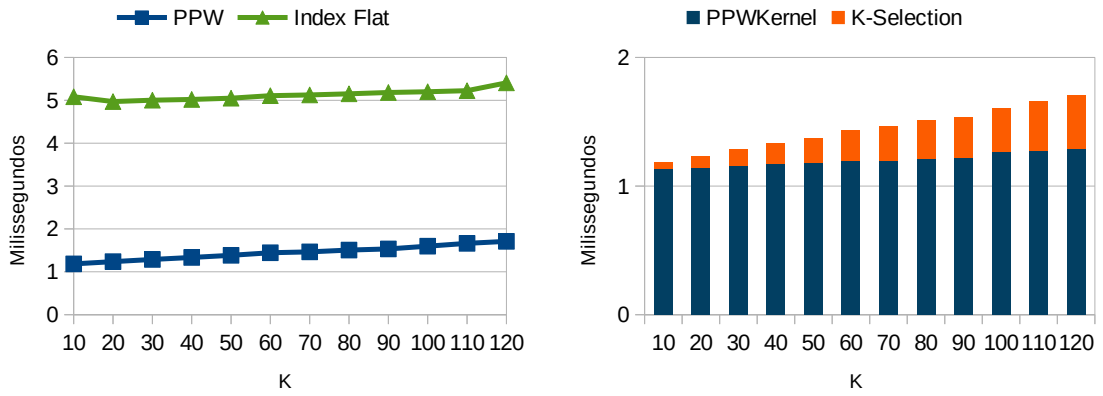

Figura 1. Experimentos variando-se o valor de K. No gráfico à esquerda observamos que o tamanho de K tem pouca influência no tempo de execução para os algoritmos PPW e Index Flat. À direita, foram detalhados os tempos de execução para os dois kernels integrantes do algoritmo PPW.

Os resultados da terceira etapa (comparação com algoritmos aproximados) estão na Tabela 4. É possível notar que o PPW apresentou maior vazão em todos os testes. Isso acontece porque o *IndexIVF* (FAISS) é otimizado para realizar consultas em conjuntos de dados muito maiores do que aqueles utilizados nos testes. Consequentemente, a latência associada à manipulação das estruturas de dados que particionam o espaço de busca domina completamente o tempo total do algoritmo. Nesse cenário o *kernel* PPW apresentou aceleração de até 10,46 vezes em relação ao *IndexIVF*.

	BASE DE DADOS COM 70.000 PONTOS, 784 DIMENSÕES (MNIST)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**1.724,14**	**3.076,92**	**4.477,61**	**5.970,15**	**6.849,32**	**6.976,74**	**7.368,42**	7.339,45	7.563,03	6.944,44	7.382,55	7.643,31
PPK	1.538,46	1.941,75	2.112,68	2.197,80	2.242,15	2.272,73	2.317,88	2.339,18	2.362,20	2.380,95	2.391,30	2.390,44
PPB	100,60	203,67	308,96	412,80	518,13	623,05	729,17	839,45	945,38	1.053,74	1.161,56	1.271,19
IndexFlat	990,10	1.960,78	2.970,30	3.960,40	4.807,69	5.769,23	6.730,77	**7.692,31**	**8.653,85**	**9.615,38**	**10.576,92**	**11.538,46**
Aceleração (PPW/IndexFlat)	**1,74**	**1,57**	**1,51**	**1,51**	**1,42**	**1,21**	**1,09**	**0,95**	**0,87**	**0,72**	**0,70**	**0,66**

	BASE DE DADOS COM 1.275.219 PONTOS, 128 DIMENSÕES (ImageNet)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**769,23**	**1.379,31**	**1.724,14**	**2.000,00**	**1.945,53**	**1.935,48**	**1.933,70**	**1.797,75**	**1.978,02**	**1.893,94**	**1.933,22**	**1.846,15**
PPK	657,89	727,27	746,27	760,46	766,87	772,20	776,05	777,45	781,25	781,86	781,81	783,29
PPB	20,42	41,14	62,12	82,75	103,37	123,86	144,30	165,19	185,61	206,31	226,94	247,42
IndexFlat	161,55	323,62	487,80	626,96	781,25	934,58	1.085,27	1.234,57	1.386,75	1.536,10	1.684,53	1.834,86
Aceleração (PPW/IndexFlat)	**4,76**	**4,26**	**3,53**	**3,19**	**2,49**	**2,07**	**1,78**	**1,46**	**1,43**	**1,23**	**1,15**	**1,01**

	BASE DE DADOS COM 3.000.000 PONTOS, 300 DIMENSÕES (GoogleNews300)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**170,65**	**307,69**	**406,50**	**424,63**	**434,78**	**417,25**	**435,05**	**410,89**	**418,22**	366,57	384,88	397,88
PPK	148,59	151,63	152,75	153,37	153,89	153,93	154,12	154,29	154,45	154,46	154,32	154,40
PPB	3,85	7,77	11,77	15,71	19,69	23,68	27,68	31,86	35,87	39,80	43,81	47,86
IndexFlat	46,36	92,59	139,21	176,83	220,85	264,78	308,51	352,11	395,60	**438,79**	**481,82**	**524,93**
Aceleração (PPW/IndexFlat)	**3,68**	**3,32**	**2,92**	**2,40**	**1,97**	**1,58**	**1,41**	**1,17**	**1,06**	**0,84**	**0,80**	**0,76**

Tabela 2. Resultados de vazão de consultas (# Consultas/s) dos experimentos para $K = 64$. Na última linha está a aceleração do PPW em relação ao *IndexFlat*. Células em verde claro indicam melhor desempenho do PPW.

Na Figura 2 estão os gráficos de *roofline* para o conjunto de dados ImageNet. O gráfico *roofline* é utilizado para avaliar o desempenho de algoritmos, considerando tanto a capacidade de processamento quanto a largura de banda de memória disponível. No eixo X está a intensidade aritmética, que representa a relação entre as operações de cálculo (FLOP/s) realizadas pelo código e a quantidade de dados buscados da memória (Bytes). O eixo Y representa a taxa de execução de operações por segundo (Tera FLOP/s). As linhas indicam os limites teóricos de desempenho do sistema, considerando a capacidade máxima de operações do processador e a largura de banda da memória. O círculo preenchido em azul representa o desempenho do algoritmo em relação a esses parâmetros.

À esquerda, está o gráfico de *roofline* para um ponto no conjunto de consulta. Nesse cenário, o PPW apresentou intensidade aritmética de 1,06 FLOP/byte e performance de 0,53 TFLOP/s, significando que está próximo do limite teórico para aquela intensidade aritmética, que é de 0,64 TFLOP/s. Isso significa que o desempenho do algoritmo está sendo limitado pela busca de dados em memória e devido à baixa intensidade aritmética. Isso não acontece quando o valor de $|Q|$ é igual à 8, no gráfico da direita, onde o PPW apresenta intensidade aritmética de 8,12 FLOP/byte e vazão de 0,81 TFLOP/s, onde o limite teórico é de 4,92 TFLOP/s, significando que o processador ainda tem capacidade de processamento não totalmente utilizada.

6. Conclusões

O *K-nearest neighbor* (KNN) é um algoritmo simples de aprendizado de máquina utilizado em diversas aplicações. Seu objetivo é encontrar em um conjunto de referência os

	BASE DE DADOS COM 70.000 PONTOS, 784 DIMENSÕES (MNIST)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**1.234,57**	**2.222,22**	**3.296,70**	**4.444,44**	**4.950,50**	5.405,41	5.833,33	6.015,04	6.250,00	5.882,35	6.321,84	6.557,38
PPK	1.075,27	1.538,46	1.764,71	1.904,76	1.984,13	2.047,78	2.095,81	2.133,33	2.158,27	2.178,65	2.191,24	2.209,94
PPB	98,23	198,61	301,81	403,23	505,56	608,52	712,11	819,67	924,02	1.028,81	1.135,19	1.240,95
IndexFlat	970,87	1.923,08	2.912,62	3.883,50	4.716,98	**5.660,38**	**6.603,77**	**7.547,17**	**8.490,57**	**9.345,79**	**10.280,37**	**11.214,95**
Aceleração (PPW/IndexFlat)	**1,27**	**1,16**	**1,13**	**1,14**	**1,05**	**0,95**	**0,88**	**0,80**	**0,74**	**0,63**	**0,61**	**0,58**

	BASE DE DADOS COM 1.275.219 PONTOS, 128 DIMENSÕES (ImageNet)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**621,12**	**1.123,60**	**1.395,35**	**1.702,13**	**1.718,21**	**1.729,11**	**1.767,68**	**1.673,64**	**1.807,23**	**1.766,78**	**1.797,39**	1.744,19
PPK	529,10	623,05	652,17	670,02	681,20	684,93	688,98	695,65	696,59	698,81	703,77	705,05
PPB	20,29	40,86	61,63	82,08	102,54	122,82	143,09	163,77	184,09	204,62	224,99	245,30
IndexFlat	156,74	314,47	473,93	609,76	759,88	909,09	1.057,40	1.204,82	1.351,35	1.499,25	1.644,25	**1.785,71**
Aceleração (PPW/IndexFlat)	**3,96**	**3,57**	**2,94**	**2,79**	**2,26**	**1,90**	**1,67**	**1,39**	**1,34**	**1,18**	**1,09**	**0,98**

	BASE DE DADOS COM 3.000.000 PONTOS, 300 DIMENSÕES (GoogleNews300)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**162,87**	**293,26**	**373,60**	**393,31**	**406,83**	**404,86**	**413,96**	**402,21**	**408,35**	360,10	380,36	393,18
PPK	141,24	146,84	148,29	149,70	150,38	150,53	150,73	151,06	151,24	151,15	151,31	151,65
PPB	3,83	7,73	11,72	15,64	19,59	23,56	27,55	31,71	35,70	39,74	43,78	47,83
IndexFlat	45,48	90,74	136,36	173,16	216,36	259,29	301,98	344,83	387,26	**429,74**	**472,10**	**514,14**
Aceleração (PPW/IndexFlat)	**3,58**	**3,23**	**2,74**	**2,27**	**1,88**	**1,56**	**1,37**	**1,17**	**1,05**	**0,84**	**0,81**	**0,76**

Tabela 3. Resultados de vazão de consultas (# Consultas/s) em experimentos para $K = 128$. Na última linha está a aceleração do PPW em relação ao *IndexFlat*. Células em verde claro indicam melhor desempenho do PPW.

	Vazão (# consultas/s) BASE DE DADOS COM 3.000.000 PONTOS, 300 DIMENSÕES (GoogleNews300)											
|Q|	1	2	3	4	5	6	7	8	9	10	11	12
PPW	**162,87**	**293,26**	**373,60**	**393,31**	**406,83**	**404,86**	**413,96**	**402,21**	**408,35**	**360,10**	**380,36**	**393,18**
IndexIVF (42,6%±0,5%)	4,34	8,68	13,25	17,62	21,98	26,40	30,82	35,74	40,17	44,56	49,02	53,52
IndexIVF (24,5%±0,4%)	7,08	14,17	21,62	28,77	35,88	43,08	50,29	58,34	65,57	72,82	80,19	87,56
IndexIVF (18,8%±0,2%)	15,57	31,17	47,57	63,29	78,93	94,79	110,65	128,31	144,23	160,21	175,94	191,94
Aceleração (PPW/IndexIVF)	**10,46**	**9,41**	**7,85**	**6,21**	**5,15**	**4,27**	**3,74**	**3,13**	**2,83**	**2,25**	**2,16**	**2,05**

Tabela 4. Resultados de vazão de consultas (# Consultas/s) dos experimentos que comparam o desempenho do PPW com o IndexIVF. As médias de precisão e os intervalos de confiança estão indicados entre parênteses. A aceleração é calculada em relação ao parâmetro que demonstrou a maior taxa de consultas por segundo. Resultados de aceleração com fundo verde claro indicam onde o *kernel* PPK obteve maior aceleração.

K pontos mais próximos de cada ponto em um conjunto de consulta. Existem diversas estratégias que podem ser utilizadas para acelerar o KNN. Este artigo apresentou um algoritmo em GPU do KNN otimizado para pequenos lotes de consulta. O algoritmo proposto foi chamado de ”Um Ponto Por Warp” (PPW) devido a sua estratégia de dividir o cálculo do KNN entre os *warps*. O PPW foi então comparado com os algoritmos PPK e PPB, propostos em [Cordeiro et al. 2023], e dois algoritmos disponíveis na biblioteca FAISS: o *Index Flat* e o *IndexIVF*.

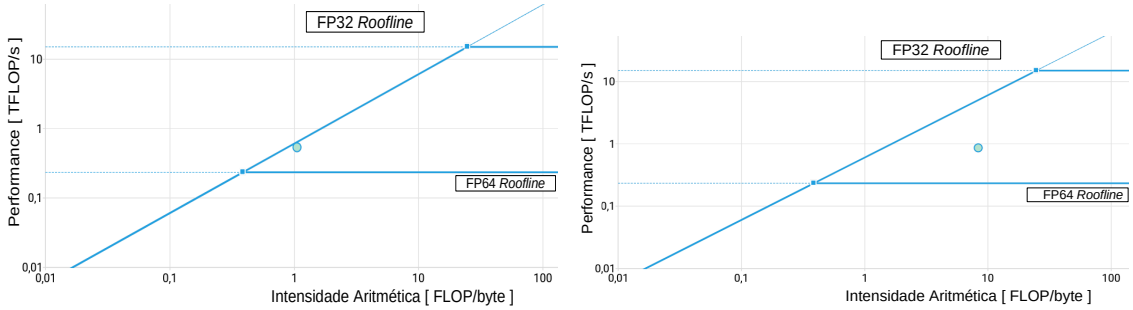

Figura 2. Gráfico de *roofline* para o algoritmo PPW. À esquerda está o gráfico para um ponto no conjunto de consulta e à direita está o gráfico para uma consulta agregada de 8 pontos ($|Q| = 8$).

Durante a análise foi possível verificar que o PPW superou os outros algoritmos em todas as bases de dados com conjuntos de consulta com menos que 5 elementos, obtendo aceleração de até 4.76 vezes em relação ao algoritmo exato da biblioteca FAISS, e até 10.46 vezes em relação ao algoritmo aproximado. Em comparação com o *IndexIVF*, o PPW apresentou melhor desempenho em todos os testes realizados. Isso ocorreu porque o *IndexIVF* é otimizado para grandes conjuntos de consulta, onde é eficiente consumir tempo de processamento extra na construção das estruturas de indexação para aceleração de buscas. Em relação ao *IndexFlat*, o PPW demonstrou estar bem otimizado para consultas em pequenos lotes, no entanto, esse algoritmo não escala tão bem quanto o *IndexFlat*, sendo superado à medida que o conjunto Q aumenta.

O algoritmo desenvolvido neste artigo será disponibilizado em `https://github.com/MichelBC/KNN-GPU`.

Agradecimentos

Este trabalho foi parcialmente suportado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processo 407644/2021-0.

Referências

Cordeiro, M., Meyer, B., and Zola, W. (2023). KNN exato em GPU. In *Anais da XXIII Escola Regional de Alto Desempenho da Região Sul*, Porto Alegre, RS, Brasil. SBC.

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

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., and Zhang, R. (2017). The moving K diversified nearest neighbor query. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 31–32.

Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A study of persistent threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*.

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

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

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

Li, J., Ni, C., He, D., Li, L., Xia, X., and 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., and Cheng, Z. (2020). KNN-Q learning algorithm of bitrate adaptation for video streaming over HTTP. In *2020 Information Communication Technologies Conference (ICTC)*, pages 302–306.

Liu, X. and Ferhatosmanoğlu, H. (2003). Efficient k-NN search on streaming data series. In Hadzilacos, T., Manolopoulos, Y., Roddick, J., and Theodoridis, Y., editors, *Advances in Spatial and Temporal Databases*, pages 83–101. 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., and Nunan Zola, W. M. (2021). Warp-centric k-nearest neighbor graphs construction on GPU. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, New York, NY, USA. Pub. ACM.

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

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

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

Ukey, N., Yang, Z., Li, B., Zhang, G., Hu, Y., and Zhang, W. (2023). Survey on exact knn queries over high-dimensional data space. *Sensors*, 23(2).

Velentzas, P., Vassilakopoulos, M., and Corral, A. (2020). A partitioning gpu-based algorithm for processing the k nearest-neighbor query. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, New York, NY, USA.

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