KNN exato em GPU

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

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

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

Resumo. *Este artigo apresenta duas implementações do KNN em GPU. Cada implementação é otimizada para diferentes quantidades de pontos no conjunto de consultas. Essas versões são comparadas com algoritmo RSFK, que é uma aproximação do KNN. As implementações mostraram estar bem otimizado quando a quantidade de pontos é pequena, alcançando uma aceleração de até 80 vezes em relação ao RSFK, também em GPU, para consultas de 128 pontos.*

1. Introdução

Algoritmos de aprendizado de máquina são essenciais para resolver diversos tipos de problemas. Um desses algoritmos é o *K-Nearest Neighbor* (KNN), ou K-Vizinhos Mais Próximos, em português. O objetivo do KNN é encontrar os K pontos de um conjunto de referência que estão mais próximos de cada ponto de um conjunto de consulta. Por ser um algoritmo computacionalmente caro, diversas estratégias foram desenvolvidas para tornar o KNN mais eficiente. Este trabalho apresenta duas implementações do KNN em GPU. Essas implementações são otimizadas para pequenas quantidades de pontos no conjunto de busca. Por fim, as versões são comparadas com a implementação em GPU do RSFK [Meyer et al. 2021][Meyer et al. 2022], algoritmo que encontra um resultado aproximado para o KNN. O presente trabalho apresenta uma versão exata para o KNN, baseado em técnicas desenvolvidas para o RSFK. A nova versão exata é eficiente para aplicação a conjuntos de busca pequenos ou quando se busca melhor acurácia de resultados.

2. Fundamentos teóricos

O KNN recebe dois conjuntos de pontos de como entrada: Um conjunto P com pontos de referência e um conjunto Q com os pontos de consulta. Vamos considerar N como sendo o tamanho do conjunto P e $|Q|$ sendo o tamanho do conjunto Q. A saída do KNN é uma matriz $|Q| \times K$ no qual cada linha da matriz representa os K pontos do conjunto P que estão mais próximos de cada ponto do conjunto Q. O algoritmo faz isso calculando a distância entre cada ponto de Q para cada ponto de P. Esses pontos possuem D dimensões e diferentes definições de distâncias podem ser consideradas. Esse algoritmo possui a complexidade computacional $O(|Q| \times N \times D)$. A aceleração do KNN pode ser feita por estratégias algorítmicas ou de implementação aproveitando recursos de hardware como o paralelismo em GPU [Johnson et al. 2019]. Um algoritmo que visa acelerar KNN é o *Random Sample Forest KNN* (RSFK), que particiona o espaço de busca e o coloca em uma estrutura de árvore. Ao fazer isso, o algoritmo se torna muito mais eficiente, pois para encontrar os K pontos mais próximos de um ponto q, basta calcular a distância entre q e os pontos que estão na mesma partição que o ponto q. Porém, existe a possibilidade de um dos K vizinhos do ponto q não estarem na sua partição. Para minimizar esse

problema, o algoritmo pode utilizar de algumas estratégias, como construir mais árvores com partições, o que aumenta a precisão mas diminui a eficiência do algoritmo. Por fim, é importante ressaltar que para ter um ganho real no desempenho, o RSFK precisa que o número de pontos nos conjuntos Q e P sejam grandes o suficientes para compensar o sobrecusto de criar as árvores de partições. É importante mencionar que o RSFK pode ser dividido em duas principais etapas. A primeira etapa consiste em construir as estruturas necessárias para realizar a busca, e a segunda etapa representa a busca em si já utilizando as estruturas computadas. Esse tipo de divisão é uma estratégia comumente utilizada em algoritmos que implementam algoritmos que resolvem o KNN [Johnson et al. 2019].

3. Implementação

Os algoritmos foram implementados em C++ utilizando CUDA (NVIDIA). Duas estratégias de paralelização do KNN foram consideradas. A primeira consiste em utilizar todos os recursos da GPU para encontrar os K-vizinhos mais próximos de um ponto do conjunto Q de cada vez. Ou seja, um Kernel CUDA é lançado para cada ponto no conjunto Q. Por esse motivo, essa versão será chamada de "Um Ponto Por Kernel (PPK)". A segunda estratégia consiste em utilizar um bloco de threads para cada ponto no conjunto Q. Dessa forma, lançaremos um Kernel com $|Q|$ blocos de threads e cada bloco será responsável por encontrar o KNN de um ponto de Q. Essa versão será chamada de "Um Ponto Por Bloco (PPB)". A Figura 1 ilustra o funcionamento das duas versões do KNN.

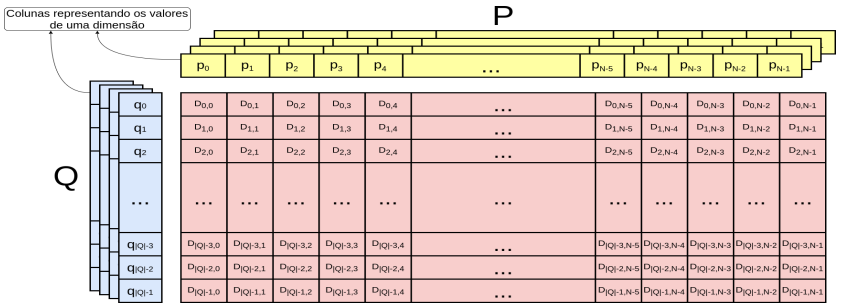

Figura 1. Funcionamento do algoritmo KNN. O conjunto P está na parte superior (amarelo) e o conjunto Q (azul) está na parte esquerda. No centro está a matriz de distâncias na qual cada elemento Di, j representa a distância entre o ponto qi e o ponto pi. Na versão PPK, cada linha da matriz será calculada por um kernel, enquanto que, na versão PPB, cada linha será calculada por um bloco de threads.

O cálculo de distância é feita a nível de warp, no qual cada warp calcula uma distância por vez. Dessa forma, as threads da warp somam as diferenças existentes em cada dimensão e depois fazem uma redução dos valores obtidos. Esse método foi baseado na versão do RSFK em [Meyer et al. 2022]. Esse artigo afirma que utilizar warps ao invés de threads para calcular as distâncias possui as seguintes vantagens: facilita o acesso coalescido à memória; mitiga o número de threads ociosas se o número de distâncias a serem calculadas for menor que o número de threads; a comunicação por registradores dentro das warps é mais rápida que outras formas de comunicação, como as comunicações feitas pela memória compartilhada ou global, por exemplo.

4. Metodologia

Os experimentos foram realizados em um processador Intel Xeon Silver 4314 CPU @ 2.40GHz com 16 núcleos, 32GB de RAM, sistema operacional Linux Ubuntu 20.04.3

LTS e com GPU NVIDIA A4500 que possui 56 multiprocessadores (MP) CUDA. Para se obter o máximo de threads ativas por MP devemos lançar os kernels CUDA com 2 blocos de 768 threads por MP. As implementações paralelas foram compiladas com a versão 11.7 de CUDA. Foram medidos os tempos apenas da execução do kernel, ou seja, os tempos de alocação de memória e transferência de dados não foram considerados. Com isso, pretende-se obter apenas o tempo necessário para calcular o KNN. Os testes foram divididos em duas partes. A primeira parte tem como objetivo comparar as duas versões do KNN exato. Para isso, foi utilizada uma base de dados artificial com os tamanhos das bases de dados reais: MNIST (70.000 pontos de 784 dimensões), ImageNet (1.275.219 pontos de 128 dimensões) e GoogleNews (3 milhões de pontos de 300 dimensões). Cada experimento foi realizado 25 vezes e reportado a média dos tempos. Os valores de $|Q|$ escolhidos para os experimentos foram 1, 16, 32 e 128, para K igual à 32,64 e 128. Na segunda parte, as implementações do KNN exato foram comparadas com o algoritmo RSFK [Meyer et al. 2021]. Para esse algoritmo, além do tempo de calcular o KNN, foram medidos também os tempos de criação das árvores de partições. Para realizar a comparação, foi escolhido uma base de dados artificial de 300.000 pontos de 128 dimensões como conjunto P. Os valores de 1, 16, 32, 64 e 128 para $|Q|$, e o K foi fixado em 128. Para o RSFK, também foram feitos testes variando entre 25, 50 e 100 árvores de partições construídas.

5. Resultados e discussão

Os resultados da primeira parte podem ser observados na Tabela 1. Podemos observar que a versão PPK possui desempenho superior ao PPB para poucos pontos no conjunto Q. Isso pode ser explicado pelo fato de que o PPB subutiliza os recursos da GPU quando $|Q|$ é menor que o número de blocos de threads necessários para preencher a GPU. Também é possível observar que o tempo de execução do PPK é linear em $|Q|$, o que já era esperado. Outro fato interessante é que o tempo do cálculo do KNN na versão PPB dobra quando $|Q|$ é igual à 128. Isso acontece porque a GPU utilizada nos testes permite no máximo 112 blocos de 768 threads, impedindo que todos os pontos sejam calculados simultaneamente. Além disso, é possível notar que o K possui pouco impacto no tempo do algoritmo. Isso acontece porque o vetor com os K pontos mais próximos é mantido inteiro na shared memory, o que significa que o custo para acessá-lo e atualizá-lo é muito pequeno. Por esse motivo, enquanto o vetor dos KNN caber na shared memory, o valor de K não terá muito impacto no desempenho. Por fim, 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.

Os resultados da segunda parte dos experimentos estão na Figura 2. O tempo de criação das árvores de partições domina completamente o tempo de execução do algoritmo. Isso acontece porque o tamanho dos conjuntos P e Q não são grandes o suficiente para compensar a criação das árvores. Mesmo se considerarmos apenas o tempo do cálculo do KNN, podemos observar que as implementações descritas neste artigo apresentam grandes ganhos de desempenho, mostrando ser muito eficiente para encontrar o KNN exato quando o conjunto Q é pequeno. Na Figura também é possível observar a acurácia obtida pela versão aproximada (RSFK), que atinge no melhor dos casos aproximadamente 61% de acurácia. Em algumas aplicações, a acurácia exigida pode ser próxima ou igual a 100%, o que demandaria o uso de mais árvores no RSFK, e consequentemente aumentaria o seu tempo de execução. É importante observar que, as técnicas propostas neste artigo

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 1. Resultado da primeira parte dos Experimentos. Tempo em milisegundos gasto para calcular o KNN em cada uma das bases de dados. Em negrito estão os tempos em que cada versão apresentou o melhor desempenho.

permitiram uma aceleração entre 19 e 80 vezes na etapa de busca em relação ao RSFK.

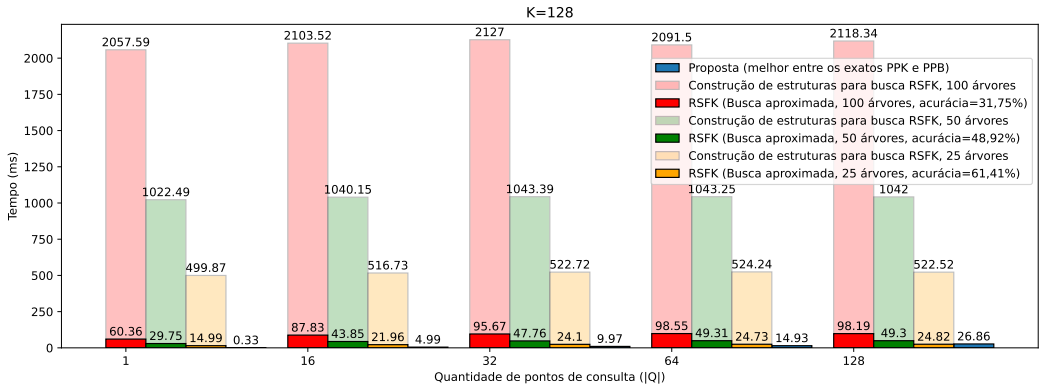

Figura 2. Comparação de diferentes algoritmos para calcular 128 pontos de consulta em uma base de 300000 pontos. Para cada valor de $|Q|$, foram medidos os tempos do RSFK (em GPU) para diferentes quantidades de árvores construídas.

6. Conclusões

Este trabalho apresentou duas implementações do KNN em GPU, uma na qual utiliza-se todos os recursos da GPU para encontrar o KNN de um ponto e outra que utiliza apenas um bloco por ponto. Foi possível notar que cada versão se sobressai para diferentes tamanhos de conjuntos de consulta. Por último, as versões mostraram ter o desempenho superior ao algoritmo de KNN aproximado RSFK (também em GPU), mostrando estarem muito otimizadas para quando existem poucos pontos no conjunto de consultas.

Referências

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

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.