Implementação Paralela de Múltiplos K-Means em GPU

Walter Bueno[1], Olavo Silva[1], José A. Nacif, Ricardo Ferreira[1]

[1]email: ricardo@ufv.br, Universidade Federal de Viçosa, Brazil

Resumo. *O algoritmo K-means possui intensidade aritmética $O(3K)$ e seu desempenho é limitado pela memória para valores pequenos de k. Implementações paralelas utilizam valores altos de k para obter um melhor desempenho. Entretanto, a maioria dos problemas práticos busca valores baixos de k, ou seja, poucos grupos. Outro desafio é encontrar o melhor valor de k. Neste trabalho, propomos uma implementação paralela eficiente em GPU que explora múltiplos valores de k simultaneamente, utilizando adequadamente as arquiteturas de GPU para maximizar o desempenho. Comparada com a implementação da Nvidia Rapids CuML, nossa implementação mostrou ganhos de até 140 vezes em aceleração, para valores baixos de k, onde múltiplas execuções simultâneas de diferentes k são realizadas. O K-means também pode ser usado para redução de dimensionalidade. Apresentamos uma implementação com múltiplas chamadas do K-means para buscar quais atributos são mais adequados para a redução de dimensionalidade. Mostramos um exemplo de redução de um conjunto de dados com 18 atributos numéricos para uma codificação de 3 bits com uma pequena perda de acurácia, ou seja, uma redução de 96 vezes e aceleração de 790 vezes.*

1. Introdução

O algoritmo K-means é amplamente utilizado [Kanungo et al. 2002] para agrupar dados em *clusters*, redução de dimensionalidade e compressão de dados. O problema de encontrar o melhor agrupamento com a soma das distâncias euclidianas é NP-difícil [Aloise et al. 2009], e heurísticas iterativas são utilizadas. Versões paralelas em GPU podem oferecer alto desempenho, mas em geral usam valores altos de k, o que não faz sentido para a maioria das aplicações onde buscamos poucos grupos para classificar os dados. As versões em GPU utilizam valores altos porque a intensidade aritmética da implementação é $O(3k)$: quanto maior o k, maior será o desempenho. Entretanto, este cenário não é realista e vai contra o perfil de uso do algoritmo. Além disso, o desafio é descobrir qual é o melhor valor de k.

Este trabalho busca o melhor valor de k e, ao mesmo tempo, utiliza a GPU de maneira eficiente ao calcular simultaneamente vários k, aumentando assim a intensidade aritmética e explorando o potencial da GPU. A implementação também faz uso eficiente dos registradores, utilizando a abordagem de mais trabalho por thread [Volkov 2010], calculando vários pontos em cada thread e fazendo uma redução inicial local no nível da thread, depois no nível do grupo, utilizando e reutilizando a memória compartilhada, deixando apenas a redução final na memória global. As etapas de classificação e ajuste dos centroides são realizadas na GPU sem intervenção da CPU. Resultados experimentais mostraram ganhos médios de aceleração de 20 a 70 vezes em comparação com a biblioteca cuML da Nvidia [Nvidia 2024], de 3 ordens de grandeza em comparação com a biblioteca Scikit-learn [Pedregosa et al. 2011] e de 2-3x comparada com uma implementação eficiente em GPU [He et al. 2022].

Além disso, este trabalho apresenta o uso do K-means para redução de dimensionalidade. Escolhemos um conjunto de dados com 5 milhões de amostras e 18 atributos numéricos. O K-means foi configurado para selecionar aleatoriamente três variáveis e fazer seu agrupamento em 8 categorias, reduzindo assim para uma representação de 3 bits. Se considerarmos atributos de 16 bits, temos uma redução de 96x. Exploramos 10 mil soluções e mostramos que existem soluções com acurácia de 76,4% (apenas 1,5% pior que o conjunto completo).

Este trabalho está organizado da seguinte forma: a seção 2 apresenta os pontos principais do K-means e da GPU. A seção 3 introduz a abordagem proposta para múltiplos valores de k e a seção 4 introduz a implementação para redução de dimensionalidade. A seção 5 apresenta os experimentos realizados. Finalmente, as seções 6 e 7 discutem os trabalhos relacionados e resumem as principais conclusões deste trabalho.

2. Fundamentos

Esta seção busca apresentar de maneira objetiva os principais pontos para implementação eficiente do K-means e o uso efetivo das GPUs.

2.1. Desmistificando a intensidade aritmética do K-means

A primeira limitação do K-means de desempenho é a operação de redução, que mesmo que explore o paralelismo da árvore de redução sempre terá no máximo a intensidade aritmética igual a 1, ou seja, será um problema limitado pelo desempenho da memória. A prova é bem simples como ilustra a Figura 1(a). Seja f o número de folhas da árvore de redução, que são as operações de memória, f sempre será maior que o número de nós internos de operação $(f-1)$, no limite a redução tem intensidade aritmética 1. No exemplo temos 8 folhas (memória) e 7 somas, resultando em uma intensidade de $\frac{7}{8} = 0,875$.

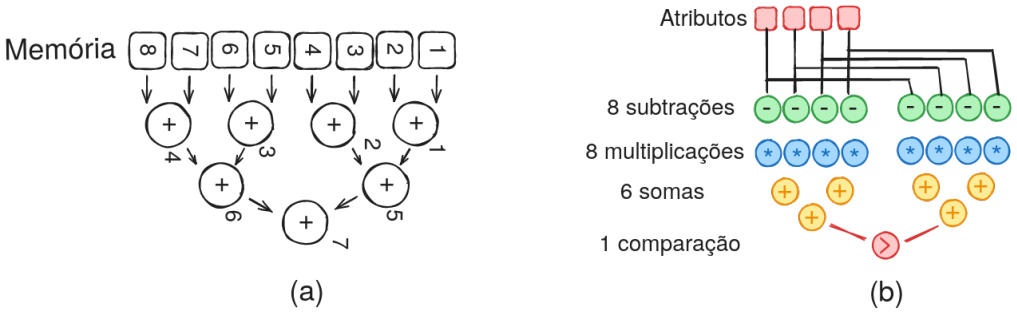

Figura 1. (a) Intensidade da Redução; (b) Exemplo de K-means com K=2 e N=4.

O algoritmo K-means realiza para cada atributo uma subtração, uma multiplicação para ter o quadrado no cálculo da distância euclidiana e depois faz a redução das distâncias para cada centroide, ou seja, três operações. Para calcular as distâncias, precisamos ler os pontos com seus atributos na memória. Como temos K centroides, podemos aproximar a intensidade aritmética para o valor de $3k$, pois será $\frac{3k \cdot pontos}{pontos} = 3k$. A Figura 1(b) ilustra um exemplo com 4 atributos ($n = 4$) e dois centroides, onde podemos contar 4 leituras e 23 operações: 8 subtrações, 8 multiplicações, 6 adições e uma comparação, ou seja, $\frac{23}{4} = 5,75$ que pode ser aproximado para $3k = 3 \cdot 2 = 6$ de intensidade aritmética. Ou seja, quanto maior o valor de k, maior será o desempenho do K-means, que pode deixar de ser limitado por memória e passar a ser limitado pela computação.

2.2. Desmistificando a aceleração de K-means em GPU

O primeiro ponto é a dependência do desempenho à intensidade aritmética (número de operações de cálculo por operação de memória). Suponha uma GPU com 10 mil núcleos a 2 GHz, que resulta em uma taxa de pico de desempenho de $10.000 \times 2G$ que é igual a 20 Tera Flops por segundo. Se cada operação precisa de dois floats de 32 bits, ou seja, 8 bytes, precisamos de uma taxa de entrega mínima de 160 Tera Bytes por segundo. Supondo uma vazão de memória de 1 Tera Byte por segundo, para atingir o pico, precisamos de uma intensidade aritmética igual ou maior que 160. Ou seja, como a intensidade é $3K$ nas implementações de K-means, se k for alto, teremos um bom desempenho, porém existe o desafio de obter desempenho para valores baixos de k.

O segundo ponto é o uso da memória mais rápida da GPU, os registradores. O código deve priorizar o uso de registradores. Em 2010, a técnica proposta por Volkov [Volkov 2010] mostrou que a multiplicação de matrizes pode ser otimizada calculando mais pontos por threads, onde o uso de registradores para 1, 2, 4, 8 e 36 pontos por threads foi de 21, 28, 41, 63 e 63, respectivamente. Ou seja, não dobrou quando dobramos o número de pontos. Em compensação, o desempenho aumentou de 242, 341, 427, 485 e 838 Gflops para 1, 2, 4, 8 e 36 pontos por threads, respectivamente. O desempenho máximo é bem próximo da taxa de pico da GPU na época [Volkov 2010]. Este trabalho modificou a estratégia de projeto da Nvidia que passou a priorizar ter mais registradores por threads. As últimas gerações podem ter até 255 registradores por thread. Na nossa implementação, adotamos a estratégia de calcular vários pontos por thread. Para validar se a GPU está sendo usada de forma efetiva, usaremos a métrica de Gops/s em comparação com a taxa de pico de Gops/s, além do tempo de execução.

O terceiro ponto é fazer a redução de forma eficiente. A redução envolve comunicação e concorrência, iremos usar os registradores ao calcular mais pontos por thread e fazer uma primeira redução local. Depois buscamos mitigar o uso da memória compartilhada para redução dentro do bloco e da memória global para redução final.

3. Multik - Uma Implementação Eficiente em GPU

As implementações de K-means têm o valor de K como parâmetro principal. Para descobrir qual o melhor K, o usuário faz várias chamadas. Como o objetivo é descobrir o valor de K, a proposta deste trabalho é calcular múltiplos K simultaneamente. A vantagem desta abordagem é aumentar a intensidade aritmética, reusando o dado de entrada e fazendo uso eficiente da GPU. A Figura 2 mostra a estrutura de implementação para cálculo com k=2, 3 e 4 simultaneamente. A intensidade será $3{\cdot}2{+}3{\cdot}3{+}3{\cdot}4 = 3{\cdot}(2{+}3{+}4) = 27$. Ou seja, mesmo para baixos valores de k é possível aumentar a intensidade e o desempenho da GPU.

A maioria dos trabalhos executa o K-means em dois passos ou apenas avalia o primeiro passo. O primeiro passo é a classificação, que irá percorrer todos os pontos e determinar para cada ponto qual é o centroide. A Figura 2 mostra apenas a fase de classificação. O segundo passo irá re-centralizar o centroide, observando o valor médio em todas as direções para cada grupo. Neste caso, as threads têm que se comunicar para uma redução geral de todos os pontos. Poucos trabalhos fazem os dois passos simultaneamente [Lutz et al. 2018], evitando a transferência de dados GPU e CPU, quando parte da redução é realizada na CPU.

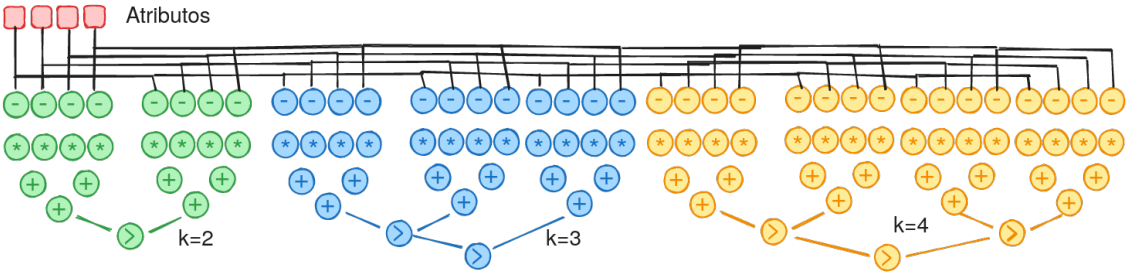

Figura 2. Diagrama para N=2 para cálculo simultâneo com $k = 2, 3$ e 4.

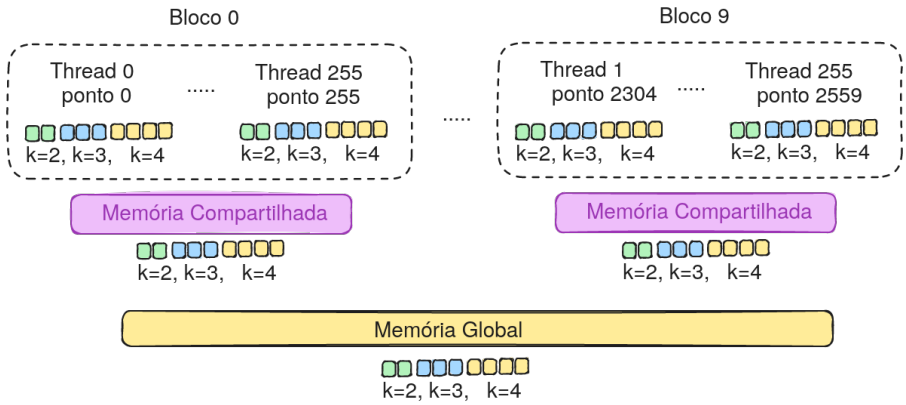

Figura 3. Redução de todos os pontos para re-centralizar os centroides.

O segundo passo para recentralização dos centroides é ilustrado na Figura 3, onde consideramos um ponto por thread. Neste exemplo, cada bloco de threads terá que reduzir os pontos de cada centroide, neste exemplo K=2, 3 e 4. Este passo pode ser feito usando a memória compartilhada. No final, temos que reduzir todos os pontos de todos os blocos utilizando a memória global ou a CPU.

A primeira contribuição deste trabalho é o uso de múltiplos K ao mesmo tempo como ilustrado na Figura 2. Podemos observar que esta abordagem aumenta a intensidade aritmética. A segunda contribuição é a redução no segundo passo do algoritmo ilustrado na Figura 3. Como a redução tem grande impacto no tempo de execução [He et al. 2022], usamos múltiplos pontos como proposto em [Volkov 2010] para otimizar a multiplicação de matrizes. A vantagem no caso do K-means, é que a maior parte da redução ficará concentrada na thread usando apenas registradores. Apenas a parte final da redução usará a memória compartilhada e, posteriormente, a memória global. O desafio nesta técnica é observar o uso de registradores e evitar a ocorrência de *spill*, ou seja, a thread fazer uso da cache quando o número de variáveis vivas é maior que o número dos registradores.

Por exemplo, suponha que temos 1 milhão de pontos. Alocando um ponto por thread, podemos ter mil blocos com mil threads. Para reduzir os centroides teremos que fazer pelo menos 1 milhão de acessos à memória compartilhada (uma por thread) e mil acessos à memória global para cada ponto e cada atributo. Agora se temos 100 pontos para cada thread, teremos apenas 10 blocos, reduzindo para 10 mil acessos à memória compartilhada e 10 acessos à memória global por ponto por atributo, ou seja, uma redução de um fator 100 no acesso aos recursos compartilhados.

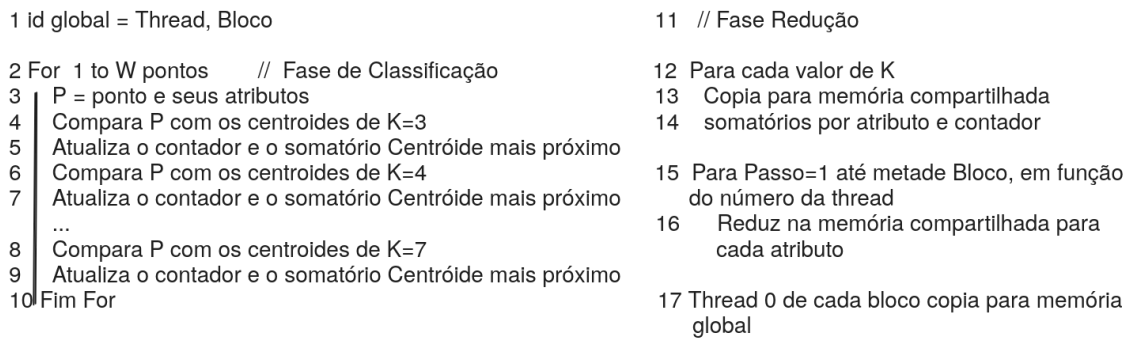

Figura 4. Pseudo-código da Implementação Multi-K com redução integrada.

A Figura 4 mostra o pseudo-código da implementação do algoritmo com múltiplos valores de K. O primeiro parâmetro é o número mínimo e máximo dos valores de K. A etapa de classificação considera os diversos valores de K para cada ponto simultaneamente. No exemplo da Figura 4, avaliamos K = 3, 4, 5, 6 e 7 nas linhas 4 a 8 do pseudo-código. Um gerador de código CUDA foi desenvolvido que automatizar o processo. O gerador é de código aberto e está disponível publicamente no repositório [Bueno 2024]. O conjunto de pontos que é classificado em cada thread será sumarizado em um contador por centroide e um acumulador para cada dimensão do centroide. Assim, o processo de redução para re-centralizar os novos centroides já começa a ser realizado localmente, em paralelo, em cada thread durante a fase de classificação (linhas 2-10).

Para finalizar a redução, primeiro temos que somar os acumuladores de cada thread por dimensão para um dos centroides dentro do bloco. Para fazer este passo, usamos a memória compartilhada. Primeiro, cada thread copia seus dados para memória compartilhada (linhas 12-14). Depois fazemos uma redução padrão usando a memória compartilhada para cada dimensão [Cheng et al. 2014]. Finalmente, os resultados dos acumuladores estarão sumarizados na memória compartilhada e a Thread 0 de cada bloco que irá copiar para memória global, onde será feita a redução final e re-centralização dos centroides.

Para possibilitar a redução com vários valores de K simultaneamente implementamos uma política de partição, pois o número de variáveis pode exceder a capacidade da memória compartilhada. Por exemplo, para K=3 com 8 atributos, teremos 9 variáveis por centroide (um contador e 8 acumuladores), que irá ocupar $3 \times 9 \times 4$ bytes = 108 bytes por thread. Se temos 256 threads por bloco precisamos de 27.648 bytes. Para o exemplo com K variando de 3 à 7, serão necessário 225 KBytes. Suponha uma memória compartilhada de 48 KBytes. Neste caso o processo é realizado em 5 passos, onde o gerador de código se ocupa de fazer o particionamento.

4. Redução de Dimensionalidade

O K-means pode ser eficiente para redução de dimensionalidade. Uma abordagem é selecionar um sub-conjunto de atributos aleatoriamente e depois agrupá-los com o K-means [Boutsidis et al. 2014]. Neste trabalho, implementamos em CUDA uma ferramenta de exploração dos atributos usando o K-means. Primeiro um conjunto de r atributos é selecionado. Suponha $r = 3$ para um conjunto de dados com 18 atributos $a_1, a_2, \ldots, a_{18}$.

Um conjunto aleatório de três atributos poderia ser (a_3, a_5, a_{12}) ou (a_8, a_{10}, a_{18}). O algoritmo K-means é executado com i iterações para k centroides: $k_0, k_1, \ldots k_7$. Suponha $k = 8$. Finalmente, o conjunto de dados é reescrito, onde todas as amostras que pertencem ao centroide k_0 serão codificadas com 0, as que pertencem ao centroide k_1 com 1, etc.

Para ter uma implementação eficiente da exploração na GPU, a abordagem foi priorizar o reuso da leitura do conjunto de dados. Como precisamos executar milhares de cópias do K-means aplicadas sobre atributos distintos, optamos por dar mais trabalho a cada thread. Cada thread irá selecionar um subconjunto de 3 atributos e executar localmente todo o código do K-means, incluindo a leitura dos pontos, classificação e ajuste dos centroides localmente. Assim, todas as threads irão ler os dados ao mesmo tempo, fazendo reuso da memória.

5. Resultados Experimentais

5.1. Trabalho por Thread

Como já demonstrado para o problema de multiplicação de matrizes por [Volkov 2010], aumentar a carga de trabalho de cada thread e usar um número menor de threads pode melhorar o desempenho. Considere a GPU T4 da Nvidia com 40 multiprocessadores com 64 cores cada, totalizando 2.560 cores. Escolhemos a GPU T4 pois ela está disponível publicamente através do Google Colab.

A Tabela 1 ilustra um experimento com um conjunto de dados sintético com 32 milhões de amostras com 8 atributos de 32 bits [Bueno 2024], ou seja, 1 GByte da dados. Executamos a abordagem proposta para avaliar os valores de K variando de 3 a 7. Fixamos o bloco em 256 threads para maximizar os uso de registradores. Como cada multiprocessador tem 64K registradores, se usarmos um bloco com 256 threads, teremos $\frac{64k}{256} = \frac{2^{16}}{2^8} = 2^8 = 256$ registradores por thread. Apresentamos o tempo médio de 10 execuções.

Tabela 1. Relação entre o tempo de execução e a carga de trabalho por thread.

N	faixa de K	Grade	Pontos	Tempo	Gops/s
8	3...7	1	131.072	2.080,60ms	47,69
8	3...7	40	3.277	134,49ms	737,74
8	3...7	51	2.571	165,46ms	599,65
8	3...7	80	1.639	142,47ms	696,42
8	3...7	101	1.298	132,83ms	746,96
8	3...7	120	1.093	140,83ms	704,53
8	3...7	151	869	121,77ms	814,80
8	3...7	200	656	119,67ms	829,10
8	3...7	301	436	123,95ms	800,47
8	3...7	400	328	120,70ms	822,03
8	3...7	3.197	141	180,19ms	550,63
8	3...7	11.916	11	251,46ms	394,57
8	3...7	131.072	1	1035,70ms	95,80

A coluna Grade mostra quantos blocos de thread foram disparados e a coluna Pontos mostra quantos pontos cada thread irá avaliar. Por exemplo, se a Grade=1, ou seja, se usamos 1 bloco de 256 threads, cada thread irá avaliar $\frac{32M}{256} = 128K = 131.072$ pontos.

Apesar da alta carga de trabalho por thread, o desempenho é o pior da tabela com 47,69 Gops/s. Isto ocorre pois gera uma subutilização da GPU com apenas 1 dos 40 multiprocessadores ocupados. Quando aumentamos o tamanho da Grade para 40 blocos, todos os multiprocessadores serão utilizados, que aumenta o desempenho em 15,5x, subindo para 737,74 Gops/s. Observe que se utilizarmos 51 blocos, teremos uma sub-utilização com o desempenho de 599,65 Gops/s, pois primeiro executam 40 multiprocessadores e depois apenas 11, ficando 29 multiprocessadores ociosos. Portanto é recomendado um múltiplo do número de processadores. O gerador é parametrizado e pode ser ajustar para cada GPU em função do número de multiprocessadores. O melhor desempenho é alcançado com 200 blocos e 656 pontos por thread, em destaque na tabela. Podemos observar que usar um ponto por thread não é vantajoso, pois gera uma degradação de desempenho pois a redução tem um peso importante no tempo de execução, que é amortizado com a redução local dentro das threads com muitos pontos (300 ou mais).

Apesar da classificação ser bem eficiente na GPU, a redução sempre irá limitar o desempenho. A estratégia de muitos pontos por thread com redução local integrada amortiza o problema, mas não é possível usar o desempenho de pico da GPU no passo de redução, devido a limitações de paralelismo desta operação.

5.2. Múltiplos K Simultâneos

Como a maior parte das aplicações do K-means busca agrupamentos com valores baixos de K, a intensidade aritmética será baixa e mesmo o passo de classificação será limitado pela memória. Para avaliar o desempenho do Multi-K, escolhemos diferentes cenários para buscar o melhor K avaliando várias faixas de valores simultaneamente. A Tabela 2 mostra o número de atributos, a faixa de valores de K e o desempenho do Multi-k em comparação com a implementação da Nvidia CuML [Nvidia 2024] e a biblioteca padrão sciki-learn [Pedregosa et al. 2011] e a versão otimizada para GPU proposta em [He et al. 2022].

Tabela 2. Tempo de Execução do MultiK em comparação com CumL [Nvidia 2024], scikit-learn [Pedregosa et al. 2011] e [He et al. 2022].

		MultiK		Aceleração do MultiK				Implementação [He et al. 2022]	
	Faixa		Tempo	CuML		Sklearn			
N	de K	Gops/s	ms	Vários	Único	Vários	Único	Vários	Único
4	3...5	667,05	35,84	140x	61x	1.812x	1.330x	8,77x	3,43x
4	3...7	753,42	65,33	130x	34x	1.718x	1.352x	8,28x	1,94x
4	3...12	546,86	265,72	70x	8x	1.129x	1.052x	4,07x	0,89x
8	3...5	633,50	76,21	69x	31x	908x	651x	5,50x	2,41x
8	3...7	695,49	142,66	64x	16x	923x	847x	5,41x	1,28x
8	3...12	762,63	383,13	50x	6x	884x	996x	4,12x	0,85x
12	3...5	701,32	103,60	57x	27x	834x	670x	5,21x	2,08x
12	3...7	699,21	213,41	50x	13x	718x	657x	4,42x	1,13x
12	3...12	630,75	696,10	31x	4x	639x	664x	2,78x	0,57x
	Média	676,69	220,22	73x	22x	1.062x	913x	4,10x	1,029x

Todos os exemplos foram executados no Google Colab com a GPU T4. Os tempos foram medidos com nvprof e para o cuML. Só consideramos o tempo gasto pelo método *fit* que faz a classificação e a redução, medidos com os eventos da biblioteca

CuPy. O conjunto de dados sintético tem 32 milhões de amostras [Bueno 2024]. Foram disparados 80 blocos com 1.639 pontos para cada thread. Para valores baixos de k, o desempenho do MultiK pode ser até 140x mais rápido que a implementação cuML da Nvidia [Nvidia 2024] considerando a execução simultânea de múltiplos k. Em média, Multik é 73x mais rápido. Comparado com a implementação proposta em [He et al. 2022] que também usa a memória compartilhada na redução, o MultiK é 4,1x mais rápido, em média.

Como calculamos múltiplos k simultaneamente, evitamos várias chamadas do K-means e aumentamos a intensidade aritmética. Podemos emular um múltiplo de k, considerando que o cuML faria o cálculo de um somatório dos valores de K, ou seja, $\sum_{K_{min}}^{K_{max}} K$. Por exemplo, para k na faixa $3 \ldots 5$ teremos $\sum_{3}^{5} = 3+4+5 = 12$. Desta forma, teremos uma intensidade aritmética maior, que seria o equivalente do ponto de vista de custo computacional à calcular múltiplos valores de k. Este valor é apresentado na coluna **Único**. Apesar do cuML e a implementação [He et al. 2022] não terem a opção único, acreditamos que poderiam ser adaptadas. Considerando uma estimativa simples de desempenho, usando o valor único, a versão Multi-K ainda seria 22 vezes mais rápida que o cuML. Avaliamos também uma implementação Multi-K utilizando PyCUDA [Klöckner et al. 2012] e não houve perda de desempenho com a interface em Python para integração, semelhante a proposta do cuML. Com relação a implementação proposta em [He et al. 2022], o Multi-K teria um desempenho equivalente, porém se considerarmos os valores baixos de k, o Multi-K é de 2 a 3 vezes mais rápido.

Finalmente, para verificar se a GPU está sendo usado com um desempenho satisfatório, calculamos o desempenho Gops/s considerando o tempo de execução para calcular $3 \times K \times N \times M$, onde N e M são o número de atributos e número de amostras, respectivamente. Neste experimento usamos $M = 32$ milhões e vários valores de N, como ilustrado na tabela. Com relação a biblioteca scikit-learn, o ganho de desempenho é de três ordens de magnitude.

5.3. MultiK-K em diferentes GPUs

Tabela 3. Tempo de Execução do MultiK em diferentes GPU e Custo na Nuvem.

		GPU T4		GPU A100			Comparação
N	Faixa	Gops/s	ms	Gops/s	ms	Relativo	Acel
4	3...5	627,46	7,62ms	3.541,67	1,35ms	0,94	5,64
8	3...5	610,77	15,81ms	2.917,30	3,31ms	0,80	4,78
16	3...5	476,81	40,7ms	3.245,19	5,98ms	1,13	6,81
32	3...5	269,42	144,41ms	1.783,05	21,82ms	1,10	6,62
4	3...7	712,80	13,81ms	2.845,01	3,46ms	0,67	3,99
8	3...7	690,46	28,74ms	3.627,74	5,47ms	0,88	5,25
16	3...7	428,15	93,06ms	3.279,32	12,15ms	1,28	7,66
32	3...7	267,49	298,49ms	1.820,84	43,85ms	1,13	6,81
media	NaN	510,42	80,33ms	2.882,52	12,17ms	1,10	5,94

Para avaliarmos a escalabilidade e relação custo desempenho na nuvem da implementação proposta, fizemos uma avaliação do desempenho na GPU T4 com 2.560 núcleos e na GPU A100 com 6.912 núcleos, 40 MB de cache L2 (6 vezes mais que a T4) e desempenho de pico de 19,2 Tera flops/s de 32 bits. Em termos de custo na nuvem,

atualmente a A100 é 6 vezes mais cara. A Tabela 3 mostra os desempenho em Gops/s e o tempo de execução das duas GPUs. A coluna **Acel** mostra a aceleração da A100 em relação a T4. A coluna **relativo** mostra o desempenho divido pelo fator 6 de custo, ou seja, se o valor for menor que 1, a T4 é mais vantajosa, caso contrário, a melhor opção é a GPU A100. Podemos observar que ganho médio é de um fator 5,94, bem próximo de 6. Ou seja, a opção pela A100 aumenta o desempenho com um aumento proporcional de custo. Além disso, a partir de N maior que 16, temos que a A100 é mais vantajosa que a T4, dado que apresenta uma maior quantidade de núcleos e capacidade adicional de transmissão de memória.

5.4. Redução de Dimensionalidade

Para avaliar a exploração dos atributos com redução de dimensionalidade descrita na Seção 4, escolhemos o conjunto de dados Susy do repositório UCI [Whiteson 2014] com 5 milhões de amostras e 18 atributos. Para fazer a exploração de quais atributos são os melhores para redução, optamos por executar 10 mil cópias do K-means, onde cada cópia é alocada em uma thread. Cada thread irá escolher três atributos aleatórios e executar 5 iterações do K-means. Como o conjunto de dados tem 5 milhões de atributos, cada thread irá executar $3 \cdot K \cdot N \cdot 5$ milhões de operações, onde $k = 8$ centroides e $n = 3$ atributos.

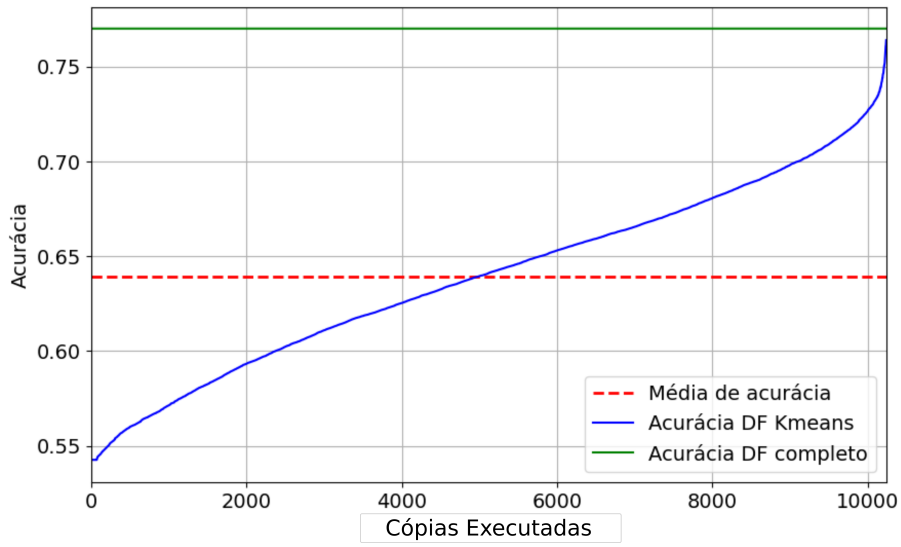

Figura 5. Acurácia dos 10 mil agrupamentos avaliados para Susy.

A Figura 5 mostra a acurácia das 10 mil reduções de dimensionalidades com três atributos diferentes. Para calcular a acurácia, cada ensaio de três atributos teve sua tabela re-codificada e executamos uma classificação com árvores aleatórias da biblioteca Scikitlearn. A linha horizontal na parte superior mostra a acurácia de referência com 18 atributos de 32 bits com acurácia de 77,9%. Podemos observar que a acurácia dos 10 mil ensaios variou de 54,24% a 76,39%. Ou seja, a exploração do espaço de soluções para este conjunto de dados mostrou ser possível encontrar um subconjunto de três atributos re-codificados com uma acurácia próxima do conjunto completo, reduzindo de 576 bits ($18 \cdot 32$ bits) para apenas três bits, ou seja, um fator de 192 vezes de redução.

A Tabela 4 mostra os 5 melhores resultados de seleção e os cinco piores. Podemos observar que os atributos 1, 7 e 18 foram selecionados por três ensaios dos 10 mil. A

Tabela 4. Os melhores e piores agrupamentos da exploração de 10 mil combinações de 3 atributos para conjunto de dados Susy com 18 atributos e 5 milhões de amostras.

Os melhores agrupamentos de 3 atributos					
Atributos	1, 7, 4	16, 1, 7	18, 1, 7	1, 18, 7	7, 18, 1
Acurácia	76,11%	76,21%	76,25%	76,30%	76,39%
Os cinco piores agrupamentos de 3 atributos					
Atributos	6, 8, 17	3, 13, 14	18, 6, 17	17, 13, 3	3, 17, 6
Acurácia	54,24%	54,26%	54,26%	54,26%	54,26%

pequena oscilação na acurácia é devido ao processo de treinamento que tem um fator aleatório na construção dos modelos com árvores. Podemos observar que os atributos 1 e 7 aparecem em todas as cinco melhores. Já para as piores, apesar do atributo 18 aparecer em uma delas, não podemos afirmar nada, pois a baixa acurácia pode ser devido a seleção dos outros dois atributos do subconjunto.

A implementação mostrou um desempenho médio de 1,12 Tera Flops/s na GPU T4, executando todas as cópias do K-means em apenas 16 segundos. Fizemos um segundo teste com 20 mil cópias e o tempo de execução foi proporcional, em torno de 33 segundos. Diferente da execução com múltiplos K simultâneos, onde podemos emular o custo de execução simulando o valor do somatório dos k nas outras implementações, no cenário da exploração, temos que executar 10 mil cópias das outras implementações. Para o conjunto de dados Susy, a execução de 10 mil cópias em cuML consome 12.978 segundos ou 3 horas e 33 minutos, ou seja, é 790 vezes mais lenta que nossa implementação.

6. Trabalhos Relacionados

As primeiras propostas de K-means em GPU apresentam baixo desempenho por usarem operações com a memória global ou operações atômicas para redução. Por exemplo, para os valores de 2 a 6 *clusters*, o desempenho é de 3,5 a 9 Gops/s apenas [Baydoun et al. 2016]. Mesmo com valores altos de K, como K=100, o desempenho é inferior a 3 Gops/s [Bhimani et al. 2015].

A implementação proposta em [Li et al. 2023] faz uso de valores altos de K, com K=256, para obter desempenho em comparação com a implementação Rapids CuML [Nvidia 2024] da Nvidia, com ganho de até 2 vezes. Porém, o conjunto de dados é 4 vezes menor que os avaliados no nosso trabalho.

A atualização do centroide domina o tempo de execução. A implementação proposta em [Penha et al. 2018] faz uso das operações atômicas que degrada o desempenho da redução, obtendo 76,8 Gops/s para K=16 e 8 atributos no conjunto de dados Census com 2 milhões de elementos em uma GPU 1080. A implementação proposta em [He et al. 2022] faz uso da memória compartilhada para reduzir o impacto das operações atômicas. Para K=4 e N=4, a implementação [He et al. 2022] atinge o desempenho de 67,3 a 171,2 GOps/s em uma GPU Nvidia 2080. A proposta de calcular simultaneamente a classificação do ponto e a atualização do centroide na GPU foi introduzida em [Lutz et al. 2018]. Este é o trabalho mais próximo da nossa proposta, porém sua implementação foi realizada em OpenCL usando a cache ao invés de explicitamente usar os registradores para maximizar os cálculos usando a memória local das threads. Para

K=4 e N=4, a implementação [Lutz et al. 2018] realiza 536 GOps/s em uma GPU Nvidia 1080. Porém não realiza o cálculo de múltiplos valores de k, calcula apenas 1 valor de k por vez, nem otimiza a redução calculando múltiplos pontos por thread.

Implementações em CPU fazem uso de uma busca mais inteligente, evitando comparar todos os centroides com todos os elementos do conjunto de dados. Um acelerador com um método exato chamado "Ball K-means" foi proposto em [Xia et al. 2020], que usa uma esfera para descrever cada agrupamento, focando na redução do cálculo da distância entre o ponto e o centroide. O "Ball K-means" pode encontrar exatamente os agrupamentos vizinhos para cada grupo, resultando em cálculos de distância apenas entre um ponto e os centroides dos grupos vizinhos, em vez de todos os centroides. Além disso, cada grupo pode ser dividido em "área estável" e "área ativa", sendo que esta última é subdividida em áreas "anulares" exatas. Apesar da redução do espaço, o desempenho relativo para um valor de K=30 é, em média, de apenas 35 Gops/s. Outra alternativa é o uso de aceleradores de FPGA na nuvem [da Silva Alves et al. 2023, Bragança et al. 2021], mas também tem desempenho para valores altos de k.

7. Conclusões

Este trabalho apresenta um gerador de código para o algoritmo K-means em GPU. O desafio é ter eficiência e descobrir o melhor valor de K, considerando que a maior parte das aplicações busca um número reduzido de agrupamentos. A solução é calcular simultaneamente vários valores de K, aumentando a intensidade aritmética, mesmo para baixos valores de K. Comparando com a biblioteca Rapids CuML da Nvidia e implementações recentes [He et al. 2022], a implementação proposta faz com redução local com registradores, gerando um ganho significativo de desempenho. Trabalhos futuros irão investigar o uso de múltiplas GPUs com múltiplas threads e múltiplos pontos para conjunto de dados com dezenas de Gigabytes, uma vez que a solução é particionada e escalável. Outras técnicas de redução também podem ser exploradas [Bueno et al. 2024]. Além disso, mostramos que o K-means pode ser interessante na redução de dimensionalidade, onde trabalhos futuros buscarão a integração com outros modelos de aprendizado de máquina.

Agradecimentos

Apoio financeiro da FAPEMIG APQ-01577-22, PIBIC FAPEMIG, CNPq e UFV. Este trabalho também foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

Aloise, D., Deshpande, A., Hansen, P., and Popat, P. (2009). Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75:245–248.

Baydoun, M., Dawi, M., and Ghaziri, H. (2016). Enhanced parallel implementation of the k-means clustering algorithm. In *2016 3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pages 7–11. IEEE.

Bhimani, J., Leeser, M., and Mi, N. (2015). Accelerating k-means clustering with parallel implementations and gpu computing. In *IEEE HPEC*.

Boutsidis, C., Zouzias, A., Mahoney, M. W., and Drineas, P. (2014). Randomized dimensionality reduction for k-means clustering. *IEEE Trans on Information Theory*.

Bragança, L., Canesche, M., Penha, J., Carvalho, W., Comarela, G., Nacif, J. A. M., and Ferreira, R. (2021). An open source custom k-means generator for aws cloud fpga accelerators. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*.

Bueno, W. (2024). Gerador de código multik para k-means. `https://github.com/arduinoufv/multiKmeans`.

Bueno, W., da Silva, O., Nacif, J., and Ferreira, R. (2024). Redução de dimensionalidade para Árvores aleatórias. In *Workshop de Iniciação Científica - Simpósio em Sistemas Computacionais de Alto Desempenho*.

Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA c programming*. John Wiley & Sons.

da Silva Alves, M., Silva, L. B., Penha, J., Ferreira, R., and Nacif, J. A. M. (2023). Kcgra–uma arquitetura reconfigurável de domínio específico para k-means. In *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*. SBC.

He, G., Vialle, S., and Baboulin, M. (2022). Parallel and accurate k-means algorithm on cpu-gpu architectures for spectral clustering. *Concurrency and Computation: Practice and Experience*, 34(14):e6621.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. (2002). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. (2012). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174.

Li, M., Frank, E., and Pfahringer, B. (2023). Large scale k-means clustering using gpus. *Data Mining and Knowledge Discovery*, 37(1):67–109.

Lutz, C., Breß, S., Rabl, T., Zeuch, S., and Markl, V. (2018). Efficient and scalable k-means on gpus. *Datenbank-Spektrum*, 18:157–169.

Nvidia (2024). cuml gpu-accelerated machine learning. `https://docs.rapids.ai/api/cuml/stable/`.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

Penha, J. C., Bragança, L., Coelho, K., Canesche, M., Silva, J., Comarela, G., Nacif, J. A. M., and Ferreira, R. (2018). A gpu/fpga-based k-means clustering using a parameterized code generator. In *High Performance Computing Systems (WSCAD)*. IEEE.

Volkov, V. (2010). Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA.

Whiteson, D. (2014). SUSY. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C54606.

Xia, S., Peng, D., Meng, D., Zhang, C., Wang, G., Giem, E., Wei, W., and Chen, Z. (2020). Ball k k-means: Fast adaptive clustering with no bounds. *IEEE transactions on pattern analysis and machine intelligence*, 44(1):87–99.