Análise empírica de implementação paralela do Algoritmo K-Means usando CUDA

Luiz Gustavo Hafemann[1] and Fabiano Silva[1]

[1]Departamento de informática, Universidade Federal Do Paraná

***Resumo*—Esse artigo mostra uma análise empírica de uma implementação paralela do Algoritmo K-Means usando CUDA. Uma versão serial e duas versões paralelas foram implementadas - sendo uma versão paralela com o uso de memória compartilhada, e outra apenas utilizando memória global. Testes de desempenho mostram uma performance 10 vezes superior da implementação paralela, comparada com uma versão otimizada da implementação sequencial, e 18 vezes superior a implementação do algoritmo K-Means na biblioteca SciPy.**

***Keywords*-kmeans, CUDA, paralelismo**

I. Introdução

Em aprendizagem de máquina e reconhecimento de padrões, K-Means é um algoritmo de aprendizagem não-supervisionada utilizado para *clustering* (agrupamento). O objetivo do algoritmo é particionar um conjunto de n observações em k agrupamentos, definindo-se k centroides, onde cada observação pertence ao grupo do centroide mais próximo. Embora tal problema seja classificado como NP-Difícil, um algoritmo tratável, conhecido como algoritmo de Lloyd é amplamente usado com bons resultados práticos, apesar de não garantir um resultado ótimo. Nesse artigo, referências ao algoritmo K-Means utilizam o algoritmo de Lloyd descrito abaixo.

O algoritmo, descrito a seguir, é computacionalmente caro para uma quantidade grande de dados e um número grande de centroides. Esse fato motiva a busca de soluções paralelas para o algoritmo, de forma a reduzir seu tempo de execução. Outro fato que motiva a busca de uma solução paralela é que há pouca dependência de dados no problema, e com isso vários cálculos podem ser executados simultaneamente.

A plataforma de computação paralela CUDA permite a criação de programas para serem executados de forma paralela em Unidades de processamento Gráfico (GPUs). Tal plataforma possui uma grande quantidade de processadores (da ordem de centenas a poucos milhares) e rápido acesso a memória. Tais processadores são menos capazes do que processadores (CPUs) modernos, mas apresentam ganhos na quantidade de processadores por placa. Dadas essas características, a plataforma CUDA mostra-se uma boa candidata para atingir bons resultados no processamento paralelo do algoritmo K-Means. Essa hipótese é explorada nesse artigo, onde uma versão serial e duas versões paralelas do algoritmo foram implementadas[1].

II. A plataforma CUDA

A plataforma de desenvolvimento CUDA foi criada pela NVIDA em 2006, e tem por objetivo disponibilizar o uso de unidades de processamento gráfico (placas de vídeo) como uma plataforma de processamento paralelo de propósito geral. Essa plataforma busca tomar proveito da maior capacidade teórica de processamento de placas de vídeo, comparadas com processadores convencionais (CPUs). As unidades de processamento gráfico possuem unidades de processamento (*cores*) muito mais simples e menos capazes do que processadores convencionais, mas sua simplicidade permite incluir uma grande quantidade de processadores em uma mesma placa (acima de 1500 *cores*, em placas modernas), proporcionando uma capacidade total de processamento teoricamente superior[1].

Na plataforma CUDA, programas destinados a rodar na unidade de processamento gráfico são chamados *Kernels*. Embora a linguagem C seja usada para a criação de tais *Kernels*, o modelo para criar esses programas é diferente do que normalmente se utiliza para programas seriais. No caso da plataforma CUDA, é comum criar-se um programa para iniciar uma grande quantidade de *threads* (da ordem de milhares, ou até milhões), onde o *kernel* é desenvolvido para ser executado em cada uma dessas *threads*. Essa quantidade de *threads* costuma ser superior ao número de processadores da placa gráfica, e são agrupados em blocos. Esses blocos são escalonados para execução pela plataforma CUDA, de acordo com as capacidades de processamento da placa de vídeo disponível (ver Figura 1). Um exemplo: para se transformar uma imagem colorida em imagem preto e branco, pode-se criar um kernel que opere sobre apenas um pixel; dividir a imagem em blocos de *threads* (uma para cada pixel), e chamar o kernel para ser executado nesses blocos.

Outro ponto importante da plataforma CUDA é o acesso à memória. A memória é definida de forma hierárquica: Memória local (visível pela *thread*), Memória compartilhada (visível pelas *threads* de um bloco) e memória global (visível por todas as *threads*) (ver Figura 2). O acesso à memória é mais rápido na memória local e na memória compartilhada com o bloco. Dessa forma, programas que façam operações que utilizem os mesmos dados várias vezes podem ter seu

[1]Código disponível em https://github.com/luizgh/kmeans_cuda

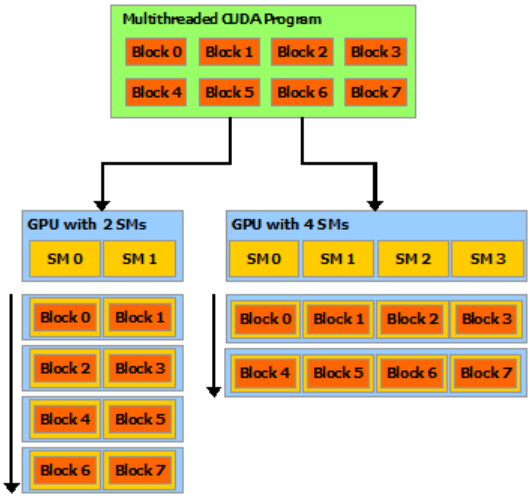

Figura 1: Modelo de execução de um programa CUDA em diferentes placas de vídeo [1]

desempenho melhorado com o uso da memória compartilhada, da seguinte forma: 1) Copiar os dados da memória global para a memória compartilhada; 2) Sincronizar as *threads*, de forma que as mesmas esperem até que todos os dados sejam copiados; 3) No processamento, utilizar as variáveis da memória compartilhada ao invés da memória global.

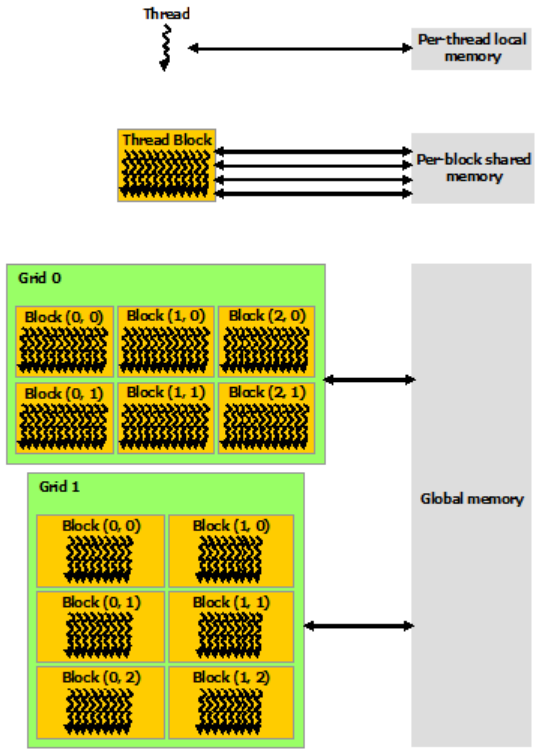

Figura 2: Hierarquia de memória na plataforma CUDA [1]

III. Algoritmo K-Means

O algoritmo K-means é comumente utilizado para agrupamento de dados. Dado uma base de dados (chamados de amostras), e um inteiro K, o algoritmo busca agrupar as amostras em K grupos, definindo K centroides, ondem cada amostra pertence ao centroide mais próximo.

O algoritmo [2, p. 149] é subdividido em três partes: Inicialização, Atribuição e Atualização:

Inicialização: Iniciar K centroides $m^{(k)}$ a partir de amostras aleatórias

Atribuição: Cada amostra $x^{(n)}$ é assinalada para o centroide mais próximo. Denotamos por b_n^k um valor booleano (i.e. $b_n^k \in \{0,1\}$ que indica se a amostra n pertence ao centroide k.

$$b_n^k = \begin{cases} 1 & \text{se } d(m^{(k)}, x^{(n)}) = \min_j d(m^{(j)}, x^{(n)}) \\ 0 & \text{caso contrário} \end{cases} \quad (1)$$

Onde $d(x, y)$ calcula a distância entre dois pontos (por exemplo, a distância euclidiana).

Atualização: A posição dos centroides é atualizada para o ponto central das amostras assinaladas a cada centroide:

$$m^{(k)} = \frac{\sum_n b_n^k x^{(n)}}{\sum_n b_n^k} \quad (2)$$

IV. Implementação Sequencial

A implementação sequencial foi feita em C++. A estrutura de dados utilizada para as bases de dados (amostras) é um vetor bidimensional de tipo *float*, de tamanho $Dim * N$, onde Dim é o número de dimensões de cada amostra, e N é o número de amostras. Visando minimizar o *overhead* no acesso aos dados, essa estrutura foi implementada como um vetor unidimensional, onde a posição $[x, y]$ é acessada como $[x*Dim+y]$. A mesma estrutura de dados foi utilizada para armazenar a posição dos centroides, com tamanho $Dim*K$, onde K é o número de centroides.

A. Otimizações

O programa foi implementado seguindo o algoritmo descrito na seção anterior, com duas otimizações:

1) Na fase de atribuição, mantem-se em memória somente um vetor, representando qual centroide está associado a cada amostra, ao invés de manter uma matrix que associa um valor booleano para cada par $(amostra, centroide)$
2) Utiliza-se o mesmo loop para a fase de atribuição, e para parte da fase de atualização: Após selecionado a qual centroide pertence uma amostra, a amostra é imediatamente somada a um vetor que será usado para atualizar as posições dos centróides. Para tanto, a implementação mantém dois vetores auxiliares: $Posicao$ (de tamanho $K * Dim$): guarda a soma das amostras associadas a cada centroide, e $Contagem$

(de tamanho K): guarda o número de amostras associadas a cada centroide.

B. Algoritmo

Utilizando as otimizações acima, obtemos o seguinte algoritmo, descrito abaixo em pseudo-código:

Algorithm 1 Kmeans Serial

Require: x (N amostras), $k \geq 1$, dim
Ensure: $m =$ posição dos k centroides
  Inicializar centroides m a partir de amostras aleatórias
  while Posição dos centroides mudou na última iteração **do**
    $Zera\_Vetores$ {Posicao e Contagem}
    for each $amostra \in$ x **do**
      $c \leftarrow Centroide\_Mais\_Próximo(amostra)$
      $Contagem[c] \leftarrow Contagem[c] + 1$
      for $i := 1$ to dim **do**
        $Posicao[c, i] \leftarrow Posicao[c, i] + amostra[i]$
      end for
    end for
    for each $cent \in$ m **do**
      for $i := 1$ to dim **do**
        $m[cent, i] \leftarrow Posicao[cent, i]/Contagem[cent]$
      end for
    end for
  end while

V. Implementação Paralela

Para a implementação paralela do algoritmo, primeiro definimos o modelo de dados que será utilizado.

A fim de melhor aproveitar a capacidade da GPU, fazemos uma mudança no algoritmo para reduzir a dependência dos dados. A mudança ocorre na fase da atribuição, quando selecionamos a qual centroide pertence cada amostra. Definimos a matrix $Distancia$, de tamanho $K * N$, para guardar a distancia entre o centroide k e a amostra n. Dessa forma, podemos dividir o trabalho de forma que cada *thread* trabalhe para calcular a distancia de um centroide para uma amostra, conforme ilustrado na figura 3

A. Implementação utilizando memória global

O algorimo paralelo segue uma estratégia similar ao processamento sequencial. Para cada funcionalidade (e.g. a fase de atualização), foi criado um *kernel* separado, em primeiro lugar porque algumas funções iteram sobre os dados (amostras) enquanto outras iteram sobre os centroides, e em segundo lugar para facilitar a análise de performance (identificar quais passos consomem mais recursos) - ver algoritmo 2.

Os *kernels* implementados foram os seguintes:

1) Zera_Vetores: Esse *kernel* zera os vetores $Posicao$ e $Contagem$. Uma thread é lançada para cada centroide, e zera as entradas pertencentes a este centroide.

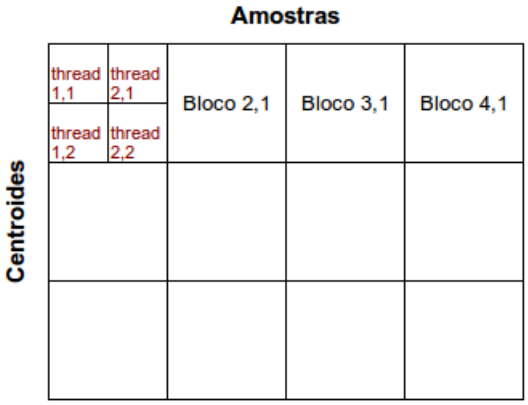

Figura 3: Divisão do trabalho em *threads* e blocos. A *thread* (1,2) calcula a distancia da amostra 1 para o centroide 2. Nesse exemplo, os blocos são de tamanho 2x2, mas na implementação esse tamanho é definido em tempo de execução.

Algorithm 2 Kmeans Paralelo

Require: x (N amostras), $k \geq 1$
Ensure: $m =$ posição dos k centroides
  Inicializar centroides m a partir de amostras aleatórias
  while Posição dos centroides mudou na última iteração **do**
    Executa kernel $Zera\_Vetores$
    Executa kernel $Calcula\_Distancias$
    Executa kernel $Soma\_Posicoes\_centroides$
    Executa kernel $Calcula\_Posicoes\_centroides$
  end while

2) Calcula_Distancias: Esse é o *kernel* que utiliza mais recursos computacionais, e calcula parte da fase de Atribuição. O objetivo é calcular as distâncias entre cada par (centroide, amostra), isto é, popular a matriz $Distancia$. O *kernel* no entanto é relativamente simples: calcula a distância entre um centroide e uma amostra, utilizando a definição de distância euclidiana. Uma thread é lançada para cada par (centroide, amostra)

3) Soma_Posicoes_Centroides: Esse é o *kernel* principal da solução, que implementa o equivalente ao primeiro *loop* da implementação sequencial. Seu objetivo é iterar sobre os exemplos n e:

- Encontrar o cluster mais próximo, usando a matrix $Distancia$: $c \leftarrow \underset{k}{\operatorname{argmin}}\, Distancia(x^{(n)}, m^{(k)})$
- Calcular o vetor $Posicao$
- Calcular o vetor $Contagem$

Para esse *kernel*, uma thread é lançada para cada amostra.

4) Calcula_Posicoes_Centroides: Esse *kernel* implementa o equivalente ao segundo *loop* da implementação sequencial. Seu objetivo é calcular as posições atualizadas

dos centroides, dados os vetores $Posicao$ e $Contagem$. Uma thread é lançada para cada centroide.

B. Implementação utilizando memória compartilhada

Após testes iniciais com a implementação paralela utilizando memória global, nota-se se que a maior parte do processamento (entre 60% e 90%) concentra-se no kernel $Calcula\_Distancias$. Dessa forma, foi implementada uma segunda versão paralela, visando explorar o uso de memória compartilhada por bloco para melhorar o desempenho do algoritmo. A implementação dos outros *kernels* não foi modificada, sendo que o *kernel* Calcula_Distancias foi alterado conforme o algoritmo 3.

Algorithm 3 Calcula_Distancias (memoria compartilhada)

Require: $x, m, indice\_thread\_x, indice\_thread\_y$
{indice_thread_x referencia a amostra, indice_thread_y referencia o centroide}
Ensure: $d =$ distancia entre x e m
$Carrega\_Amostra\_X\_MemCompartilhada$
$Carrega\_Centroide\_Y\_MemCompartilhada$
Sincronizar threads
$d \leftarrow Calcula\_Distancia\_centroide$

VI. Análise empírica

A. Sobre o experimento

1) Base de dados: A base de dados escolhida para os experimentos foi a base Wine Quality [3]. Essa base de dados foi escolhida pelo seu tamanho moderado e por possuir todas as variáveis (dimensões) como números reais, não exigindo nenhum pré-processamento. Essa base de dados possui 6497 amostras, com 11 dimensões (a dimensão de resultado foi desconsiderada, visto que K-Means é um algoritmo de aprendizagem não-supervisionado).

2) Implementações: As implementações testadas foram as descritas acima (sequencial, paralela com memória global e paralela com memória compartilhada) com a adição da implementação sequencial da biblioteca SciPy[4]. SciPy é uma biblioteca *open-source* de computação científica, disponível para a linguagem Python. SciPy possui a parte crítica da sua funcionalidade implementada em C e fortran, e demostra boa performance no uso prático.

3) Plataformas utilizadas (sequencial): Os testes das implementações sequenciais foram realizados utilizando o seguinte hardware:

- Intel(R) Core(TM) i7-975 @ 3.33GHz com 12GB RAM, rodando Debian 7.0
- Intel(R) Core(TM) i7-3770 @ 3.40GHz com 16GB RAM, rodando Ubuntu 13.04

Ambas *CPUs* possuem quatro núcleos com *hyperthreading* habilitado. Nos testes, a segunda CPU (Core-i7 3770) obteve melhores resultados. Os resultados abaixo refletem execuções nessa CPU.

4) Plataformas utilizadas (paralelo): Os testes das implementações paralelas foram realizados utilizando os seguintes placas gráficas:

- GeForce GTX 650 @1.15GHz, com 2GB de memória, 2 multiprocessadores com 192 cores cada (384 cores no total)
- Tesla C2050 @1.06GHz, com 3GB de memória, 14 multiprocessadores com 32 cores cada (448 cores no total).

Nos testes, a segunda GPU (Tesla C2050) obteve melhores resultados. Os resultados abaixo refletem execuções nessa GPU.

5) Metodologia: As implementações foram testadas no hardware acima, sempre com a mesma base de dados, mas com número de centroides diferentes. Para reduzir o efeito de interferências externas, as implementações foram executadas nos sistemas com baixo *load*, e cada implementação foi executada 100 vezes, sendo reportada a média do tempo de execução.

B. Resultados

A figura 4 mostra o tempo de execução de cada uma das implementações consideradas, com diferentes entradas (números de centroides). Nota-se um crescimento aproximadamente linear no tempo de execução dado o número de centroides, e uma diferença considerável entre as implementações paralelas e sequenciais.

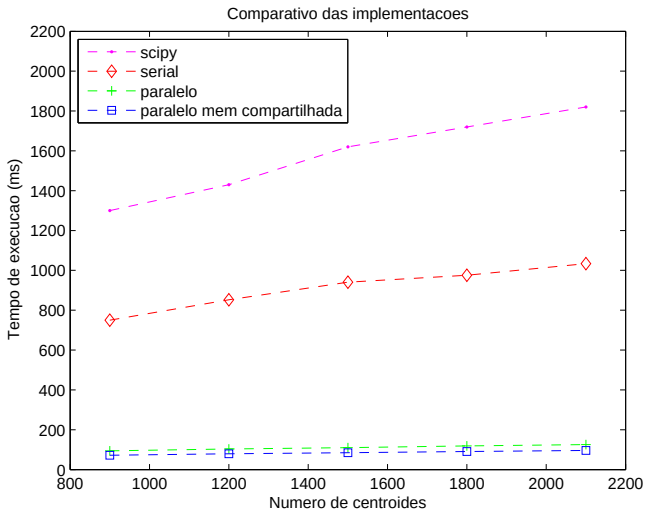

Figura 4: Tempo de execução

A figura 5 mostra um comparativo entre a implementação paralela com memória compartilhada, e as implementações seriais. A figura mostra o **speedup** da implementação paralela, isto é, o quão mais rápido é a implementação paralela, comparada com a versão sequencial. O **speedup** é definido formalmente como $S_p = \frac{T_s}{T_p}$, onde T_s é o tempo de execução

do algoritmo sequencial e T_p é o tempo de execução do algoritmo paralelo.

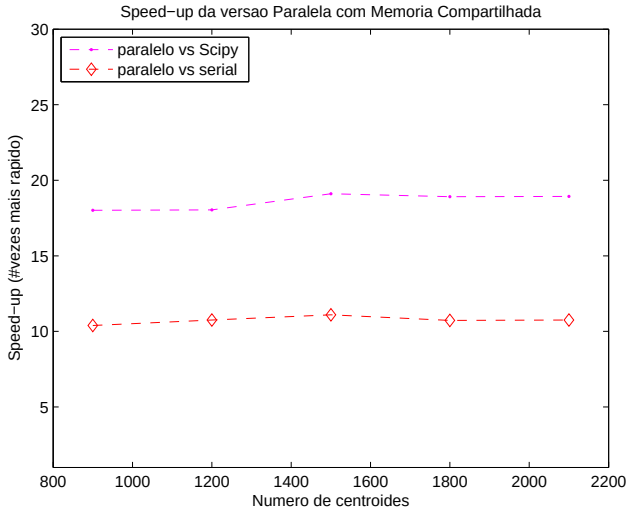

Figura 5: *Speedup* da versão paralela com memória compartilhada

A figura 6 mostra um comparativo entre a implementação paralela com memória compartilhada, e a implementação paralela somente utilizando memória global.

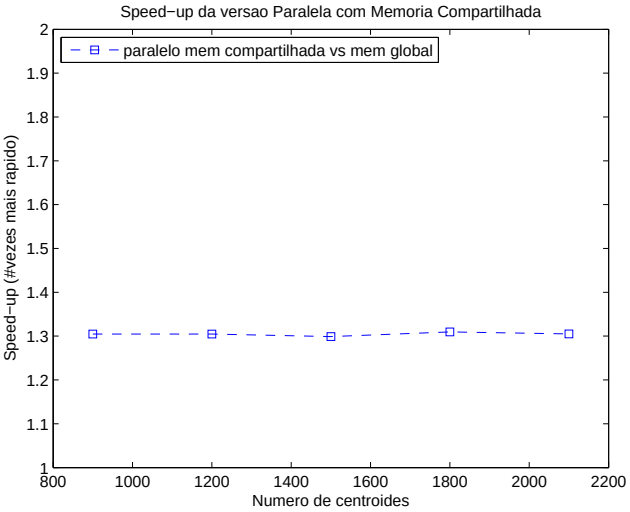

Figura 6: *Speedup* da versão paralela com memória compartilhada

VII. Conclusão e trabalhos futuros

Os resultados dos experimentos indicam uma melhora significativa no tempo de execução do algoritmo ao se utilizar uma solução paralela. Nos testes realizados a versão paralela mostrou-se 10 vezes mais rápida que a versão sequencial, e 18 vezes mais rápida que a implementação da biblioteca SciPy. Os testes também mostraram uma melhora de aproximadamente 30% na performance do algoritmo ao se utilizar memória compartilhada. Nota-se que a implementação paralela descrita nesse artigo utiliza uma grande quantidade de memória, portanto impossibilitando seu uso em grandes bases de dados. No entanto, os algoritmos paralelos descritos acima podem ser adaptados para evitar o uso dessa memória adicional. O autor sugere esse tema como possível trabalho futuro.

Referências

[1] NVIDIA. (2008) Cuda c programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.

[3] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547–553, 2009.

[4] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/