Paralelização da Indexação de Dados no OPTICS via GPU*

**Danilo Melo , Sávyo Toledo , Guilherme Andrade ,
Fernando Mourão , Leonardo Rocha**

DCOMP/UFSJ - São João del-Rei, MG , Brasil

{daniloamaral,savyomachado,gandrade,fhmourao,lcrocha}@ufsj.edu.br

Abstract. *Organizing the large volume of data existing in several domains has become one of the biggest problems in Computer Science. Among the different strategies propose to deal efficiently and effectively with this purpose, we highlight those related to density-based clustering strategies, such as DBSCAN and OPTICS. These strategies allow defining clusters of arbitrary shape, dealing with the presence of data noise robustly. In this work, we present a new approach to make OPTICS feasible based on data indexing strategy using graphs, reducing considerably its complexity. Exploring some parallelization opportunities using GPUs, we show in our experiments that our proposal can be over* $200x$ *faster than its sequential version using CPU.*

Resumo. *Organizar o grande volume de dados que vem sendo gerado recentemente tornou-se um dos grandes problemas na computação. Dentre as estratégias que se propõem a lidar com esse problema, destacam-se os de agrupamento baseado em densidade, tais como DBSCAN e OPTICS. Essas técnicas permitem agrupamento de dados em formato arbitrário lidando com ruídos de forma robusta. Nesse trabalho apresentamos uma nova abordagem do OPTICS baseada em uma estratégia de indexação de dados por meio de grafos, reduzindo consideravelmente a complexidade do algoritmo. Explorando algumas oportunidades de paralelização utilizando GPUs, mostramos que a nossa proposta é até* $200x$ *mais rápida do que sua versão sequencial usando CPU.*

1. Introdução

Observamos recentemente um aumento significativo no volume de dados que vem sendo gerado. Organizar esses dados e encontrar informações úteis para os usuários tornou-se um dos maiores desafios em Ciência da Computação. A todo o momento surgem novos modelos e algoritmos que sejam capazes de lidar com esses dados de forma eficiente (tempo de resposta e uso adequado dos recursos computacionais) e eficaz (qualidade da resposta). Dentre esses trabalhos, destacam-se os relacionados a agrupamento, que consiste em organizar grandes conjuntos de objetos em diferentes grupos (i.e., *clusters*) de acordo com alguma métrica de similaridade determinada [Jain et al. 1999].

Embora existam muitos algoritmos de agrupamento propostos na leitura, a efetividade da maioria deles é bastante limitada em função da existência de parâmetros de entrada muito sensíveis. Pequenas mudanças nesses parâmetros leva a resultados bastante distintos [Aggarwal and Reddy 2013]. Diante desses desafios, estratégias de agrupamento

*Esse trabalho foi parcialmente financiado por CNPq, CAPES, FINEP, Fapemig, e INWEB.

baseadas em densidade [Ester et al. 1996] têm se apresentado como boas alternativas, uma vez que são capazes de formar grupos em formato arbitrário, além de lidar com a presença de ruídos nos dados de forma robusta. A principal ideia por trás dessas propostas é que *clusters* são áreas com alta densidade (muitos objetos) separadas por áreas de baixa densidade (poucos objetos). Uma área é considerada densa se existirem mais objetos que $MinPts$ com uma distância entre eles menor que ϵ'. No entanto, escolher os valores adequados para $MinPts$ e, especialmente, ϵ' ainda não é trivial. Em [Ankerst et al. 1999], é apresentado o OPTICS, um algoritmo capaz de lidar com esses problemas. Ao invés de definir diretamente os *clusters*, o OPTICS produz uma ordenação dos objetos (pontos) de acordo com uma estrutura denominada *reachability*. Iniciando com um valor alto de ϵ, essa estrutura é capaz de representar diferentes densidades dentro do conjunto de dados, sendo que os *clusters* podem ser extraídos utilizando diferentes valores de $\epsilon' \leq \epsilon$.

Como o OPTICS é baseado em distância, em que é necessário calcular a distância entre todos pares de objetos, ele apresenta problemas de eficiência, principalmente quando aplicado em cenários grandes e complexos. Observamos na literatura algumas propostas que visam tornar a utilização desse tipo de algoritmo computacionalmente viável, seja por meio de indexação de dados [Christen 2012], seja utilizando paralelização em diferentes unidades de processamento [Patwary et al. 2013]. Com relação a estratégias de paralelização, propostas que utilizam placas aceleradoras gráficas (GPUs) vêm recebendo merecido destaque, uma vez que são capazes de proporcionar níveis mais elevados de paralelismo, associados a baixos consumos de energia [Andrade et al. 2013].

Nesse trabalho, apresentamos uma nova abordagem para tornar o OPTICS computacionalmente viável baseada em uma estratégia de indexação de dados paralelizada em GPU. Similarmente ao proposto em [Patwary et al. 2013], nossa abordagem representa os dados originais como um grafo $G(V, E)$, onde V são os objetos e E as arestas que ligam objetos que estejam a uma distância menor que ϵ. Para representar esse grafo, adaptamos a estrutura de dados compacta METIS [Karypis and Kumar 1998] utilizando três vetores: Va representa os vértices (o índice representa o vértice e cada posição armazena seu grau e a posição no vetor Ea onde se inicia sua lista de adjacência); Ea_n armazena os vértices na lista de adjacência; e Ea_d armazena o peso da aresta (distância). A construção dessa estrutura é feita completamente em paralelo em GPU e, no final, a lista de adjacência de cada vértice também é ordenada em paralelo. A partir dessa estrutura, a execução do OPTICS se torna muito rápida, uma vez que as operações originalmente mais caras são reduzidas para $O(1)$. Sendo assim, a complexidade do OPTICS passa a ser dominada pelo tempo de construção de uma *heap*, cujo custo é $O(E * logV)$. Em nossos experimentos de avaliação, demonstramos que essa abordagem resulta em um algoritmo extremamente eficiente, alcançando *Speedup* superiores a $200\times$ utilizando apenas uma GPU.

A implementação do algoritmo e execuções de experimentos foram realizadas pelo aluno Danilo Melo, sob a orientação do professor Leonardo Rocha. As análises dos resultados foram feitas com a colaboração do professor Fernando Mourão. Esse trabalho também contou com a colaboração do aluno Sávyo Toledo na implementação da versão sequencial do algoritmo, bem como do aluno de pós-graduação Guilherme Andrade na geração e validação dos conjuntos de dados utilizados nos experimentos de avaliação.

2. Trabalhos Relacionados

Agrupamento de dados consiste, basicamente, em organizar dados em grupos semanticamente consistentes, baseado em alguma métrica de similaridade previamente definida [Jain et al. 1999]. Vários são os desafios relacionados ao uso dessas técnicas, tais como configurar corretamente os parâmetros de entrada, escolher adequadamente uma métrica de similaridade, bem como trabalhar com grandes volumes de dados. Encontramos na literatura uma grande variedade de algoritmos que visam lidar com esses desafios [Aggarwal and Reddy 2013], que vão desde técnicas simples, porém amplamente utilizadas, como o k-means [Jain et al. 1999], até técnicas mais elaboradas e focadas em contextos específicos, como agrupamento de subespaços [Agrawal et al. 1998]. Um conjunto de técnicas de agrupamento que vem recebendo grande atenção é aquele relacionado com o agrupamento baseado em densidade [Ester et al. 1996, Ankerst et al. 1999]. Tais técnicas são distinguidas pela facilidade de execução e pela aplicabilidade em contextos diferentes. Além disso, essas técnicas não exigem que número de *clusters* seja definido como parâmetro de entrada, como é feito por diversas técnicas.

O algoritmo de agrupamento baseado em densidade mais referenciado na literatura é o DBSCAN [Ester et al. 1996]. Seu funcionamento é baseado no cálculo de proximidade entre cada par de objetos, a qual é definida de acordo com a métrica de similaridade adotada. Os objetos são agrupados a partir de um raio mínimo de proximidade, definido como parâmetro de entrada. O principal problema associado ao DBSCAN é que os *clusters* podem apresentar densidades diferentes, sendo necessário executar o DBSCAN para diferentes raios mínimos de proximidade. O principal algoritmo proposto para resolver este problema é o OPTICS [Ankerst et al. 1999], que ao contrário da maioria das técnicas mencionadas, não produz diretamente *clusters*. Baseado em um valor elevado do raio mínimo de proximidade (ϵ), o OPTICS define uma ordenação dos objetos da coleção que pode representar diferentes *clusters*. A partir deste conjunto de dados ordenado, é possível extrair *clusters* usando diferentes valores de $\epsilon' \leq \epsilon$.

Apesar de encontrar clusters de densidades diferentes com apenas uma execução, o OPTICS ainda é uma proposta baseada no cálculo das distâncias entre todos os pares de objetos de uma coleção, assim como o DBSCAN, o que é computacionalmente muito caro. Uma das estratégias mais utilizadas para melhorar o desempenho destes algoritmos é a indexação dos dados [Christen 2012, Karypis and Kumar 1998]. Outra estratégia adotada para tornar esses algoritmos capazes de serem aplicados em cenários de grandes volume de dados é a paralelização dos mesmos utilizando múltiplas unidades de processamento [Patwary et al. 2013] e, mais recentemente, utilizando placas aceleradoras gráficas (GPU) [Andrade et al. 2013]. Nesse trabalho apresentamos uma nova e eficiente abordagem do OPTICS baseada na indexação de dados paralelizada em GPU.

3. Indexação de Dados em GPU para o OPTICS

3.1. OPTICS: Uma visão geral

Como já mencionado, o OPTICS não produz diretamente os *clusters*, mas uma ordenação dos objetos que pode representar diferentes *clusters*. A partir de um raio de proximidade mínimo ϵ, essa ordenação encapsula as informações de todos os *clusters* que podem ser gerados utilizando-se valores de $\epsilon' \leq \epsilon$. O OPTICS é baseado em três conceitos

principais:ϵ-*neighborhood*, *core-distance* e *reachability-distance*, conforme definido abaixo:

Definição 3.1. (ϵ-*neighborhood*): A vizinhança de um objeto p é o conjunto de objetos s em que $distance(p, s) \leq \epsilon$:

$$N_\epsilon(p) = \{s | distance(p, s) \leq \epsilon\}$$

Definição 3.2. (*core-distance de p*): Tomando $MinPts$ como um número natural e *MinPts-distance(p)* como a distância de p para seu $MinPts$ vizinho mais próximo, a *core-distance de p* é definida abaixo:

$$coreDist_{\epsilon,MinPts}(p) = \begin{cases} Undefined, & \text{se } |N_\epsilon(p)| < MinPts, \\ \textit{MinPts-distance(p)}, & \text{caso contrário.} \end{cases}$$

Definição 3.3. (*reachability-distance*): A *reachability-distance* de um objeto p para um objeto o é definido:

$$reachDist_{\epsilon,MinPts}(p, o) = \begin{cases} Undefined, & \text{se } N_\epsilon(o) < MinPts, \\ \textit{max}(coreDist(o), distance(o, p)), & \text{caso contrário.} \end{cases}$$

Intuitivamente, a *core-distance* corresponde à menor distância ϵ' entre p e um objeto em ϵ-*neighborhood* , ou $Undefined$ se p não é *core*. A *reachability-distance* de um objeto p para outro objeto o é a menor distância tal que p é alcançável a partir de o se o é *core*. Todos os objetos começam com a reachability-distance como $Undefined$.

Algorithm 1 Algoritmo OPTICS

```
function OPTICS(Dataset, ε, MinPts)
    for all ((p ∈ Datase) and (p.processed ≠ True)) do
        N = neighbors(p, ε)
        p.processed = True
        output p
        p.setCoreDist(p, ε, MinPts)
        if p.coreDist ≠ UNDEFINED then
            Seeds = empty priority queue
            Update(N, p,Seeds)
            for all q ∈ Seeds do
                N' = neighbors(q, ε)
                q.processed = True
                output q
                q.setCoreDist(q, ε, MinPts)
                if q.coreDist ≠ UNDEFINED then
                    Update(N', q)
                end if
            end for
        end if
    end for
end function

function UPDATE(N, p, Seeds)
    cdist = p.coreDist
    for all ((o ∈ N) and (o.processed ≠ True)) do
        newrdist = max(cdist, distance(p, o))
        if o.reachDist = UNDEFINED then
            // o is not in Seeds
            o.reachDist = newrdist
            insert(o, Seeds)
        else
            if newrdist < o.reachDist then
                o.reachDist = newrdist
                // move-up on queue
                decrease(o, Seeds)
            end if
        end if
    end for
end function
```

Conforme ilustrado no Algoritmo 1, o OPTICS mantém uma fila de prioridade chamada $Seeds$ para expandir a ordenação dos objetos. A prioridade é definida de acordo com a *reachability-distance* dos objetos. Primeiro, ele seleciona aleatoriamente um objeto p que ainda não foi processado e determina sua *core-distance*. Se p é um objeto *core* ($coreDist_{\epsilon,MinPts}(p) \neq Undefined$), para cada vizinho $q \in N_\epsilon(p)$ é calculada uma nova *reachability-distance*. Se q não está em $Seeds$, ele é inserido no mesmo de acordo com sua *reachability-distance*. Se q já está em $Seeds$ e sua *reachability-distance* atual é maior do que recentemente calculada, q é movido para cima na fila de prioridades $Seeds$. Todo este processo é repetido até que $Seeds$ esteja vazia e não existam mais objetos a serem processados. Com base na saída do algoritmo OPTICS, podemos extrair *clusters* de densidades distintas $\epsilon' \leq \epsilon$ percorrendo a ordenação gerada, associando os objetos aos *clusters* de acordo com a *reachability-distance* e a *core-distance* dos objetos. Conforme

podemos observar, o tempo de execução do OPTICS é dominado pelo tempo de se obter os ϵ-*neighborhood* de cada objeto da coleção, cujo custo é $O(V^2)$. Propomos uma adaptação da estrutura METIS [Karypis and Kumar 1998] para contornar esse problema.

3.2. Implementação Sequencial

Em nossa proposta, representamos os dados como um grafo $G(V, E)$, onde V representa objetos de uma dada coleção e E as arestas que ligam os objetos que estejam dentro do raio mínimo (distância menor do que ϵ). Trata-se de um grafo ponderando em que os pesos representam a distância entre dois objetos. Esta distância pode ser calculada por métricas de similaridade (e.g., Similaridade de Cosseno) ou distância (e.g., Distância Euclidiana), de acordo com cada cenário. Como nosso objetivo principal é a paralelização do algoritmo usando GPUs, onde existe uma grande limitação da memória disponível, escolhemos representar o grafo usando uma lista de adjacência compacta, adaptando a estrutura METIS [Karypis and Kumar 1998]. Para isso, usamos três vetores: Va que representa os vértices; Ea_n que armazena a lista de adjacência de cada vértice; e Ea_d que armazena o peso das arestas. Em Va o índice i representa o vértice e cada posição do vetor guarda dois valores: o número total de vértices adjacentes de i (grau); e a posição no vetor Ea onde se inicia sua lista de adjacência.

Ilustramos esta estrutura de dados por meio da Figura 1. Por exemplo, para encontrar a lista de adjacência do objeto com identificação 0, primeiro devemos obter os valores armazenados na posição 0 de Va. O primeiro valor representa o grau do vértice (neste caso 2) e a segunda o índice em que a sua lista de adjacência começa em Ea_n (índice 0). Em seguida, podemos obter a lista de adjacência de 0 visitando duas posições em Ea_n a partir da posição 0. Também é possível obter a distância entre os objetos acessando Ea_d. Neste caso, obtemos os objetos 2 e 3, que têm distância 0.5 e 1.75 para o objeto 0, respectivamente. A complexidade de espaço desta estrutura de dados é $O(V + E)$.

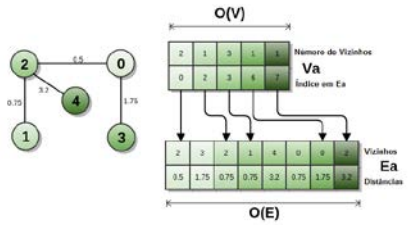


Figura 1. Estrutura de dados utilizada para representar o grafo.

No Algoritmo 2, ilustramos o processo de construção do grafo que irá representar o conjunto de dados. Como podemos observar, o algoritmo recebe como parâmetros de entrada o ($dataset$) (conjunto de dados), ϵ e $MinPts$, retornando o grafo resultante utilizando a estrutura acima descrita. Para cada objeto, primeiro devemos calcular a distância para os demais objetos da coleção. Se o valor calculado é inferior ao parâmetro de entrada ϵ, uma aresta é criada entre esses objetos, e o peso da aresta é armazenado em Ea_d. Após este primeiro passo, devemos ordenar pela distância a lista de adjacência de cada vértice. No caso da execução sequencial, adotamos o algoritmo QuickSort, que tem uma complexidade de $O(nlogn)$. Essa etapa de ordenação é muito importante, uma vez que reduz o custo de se encontrar a *core distance* a $O(1)$ (i.e., determinar o $MinPts^{esimo}$ vizinho).

Depois de indexar os dados, o passo seguinte é aplicar o OPTICS, conforme apresentado no Algoritmo 1. Neste caso, as funções para determinar o total de vizinhos de um objeto p, bem como a sua *core-distance*, são $O(1)$. Em nosso caso, adotamos uma estrutura *heap* para representar a fila de prioridade $Seeds$. Portanto, a complexidade de tempo do OPTICS passa a ser dominada pelo processo de construção e manutenção da *heap* dada por $O(E*logV)$. Por outro lado, o processo de indexação tem uma complexidade de tempo de $O(V^2)$, uma vez que, no pior dos casos, será necessário uma comparação entre cada par de objetos. Embora existam técnicas mais elaboradas de indexação que visam reduzir essa complexidade, nossa escolha foi baseada nas oportunidades de paralelização que podem ser exploradas usando GPUs, conforme veremos na próxima seção.

Algorithm 2 Algoritmo para a Construção do Grafo

```
function MAKEGRAPH(MinPts, ε, dataset, Graph)
    for all p ∈ dataSet do
        for all q ∈ dataset do
            if proximity(p, q) ≤ ε then
                InsertEdge(p, q, Graph)
            end if
        end for
        dataset = dataset - p
        SortAdjacentList(p)
    end for
end function
```

3.3. Implementação paralela

Conforme discutido na seção anterior, o OPTICS é dividido em duas fases principais: (1) construção do grafo e (2) execução propriamente dita do OPTICS. Nesse trabalho nos concentramos na paralelização da primeira etapa, por três razões principais. Em primeiro lugar, a complexidade de construção do grafo é maior que a execução do OPTICS quando o mesmo utiliza nossa estrutura. Em segundo lugar, essa etapa de construção possui muitas oportunidades de paralelização. Por fim, o processo de paralelização do OPTICS usando a estrutura de grafo proposta é limitado à paralelização da estrutura *heap*, a qual possui muitas seções críticas. Como veremos na seção 4, o processamento do OPTICS em si corresponde a menos de 1% do tempo total de execução.

Como descrito na Seção 3.2, o processo de construção do grafo envolve o preenchimento dos vetores Va e Ea (Ea_n e Ea_d) com complexidade de espaço de $O(V)$ e $O(E)$, e está dividido em quatro passos básicos: (1) cálculo do primeiro valor de Va; (2) cálculo do segundo valor de Va; (3) construção de Ea; e, finalmente (4) ordenação das listas de adjacências de Ea_n pela distância armazenada em Ea_d. Todas essas etapas foram paralelizadas usando GPUs, conforme descrito abaixo e resumido na Figura 2.

- **Cálculo do grau dos vértices:** Para cada vértice, calculamos o número total de vértices adjacentes. Nesta etapa, podemos usar os vários núcleos da GPU para processar múltiplos vértices em paralelo. Nossa estratégia paralela usando GPU atribui uma *thread* (núcleo) para cada vértice, isto é, cada entrada do vetor Va. Cada núcleo da GPU, então, irá contar quantos vértices adjacentes têm o ponto sob a sua responsabilidade, preenchendo o primeiro valor no vetor Va. Como podemos ver, não há nenhuma dependência ou comunicação entre as tarefas paralelas, ou seja, trata-se de um problema embaraçosamente paralelo. Assim, a complexidade computacional pode ser reduzida de $O(V^2)$ para $O(V)$.

- **Cálculo do índice:** O segundo valor em Va corresponde ao índice inicial em Ea contendo a lista de adjacência de um vértice particular. O cálculo desse valor depende do índice inicial da lista de adjacência e do grau do vértice anterior. Por exemplo, o índice inicial para o vértice 0 é 0, uma vez que ele é o primeiro vértice. Para o vértice 1, o índice inicial é dado pela soma entre o índice do vértice anterior (0) com seu respectivo grau, já calculado no passo anterior (2). Conforme podemos perceber, temos uma dependência de dados onde o próximo vértice depende do cálculo dos vértices anteriores. Este é um problema que pode ser eficientemente resolvido em paralelo por meio de um método de **exclusive_scan** [Blelloch 1990]. Dessa forma, utilizamos a biblioteca *thrust*, distribuída como parte do CUDA SDK, que fornece, entre outros algoritmos, uma aplicação **exclusive_scan** optimizada adequada para o nosso problema.
- **Montagem das listas de adjacência:** Estando o vetor Va completamente preenchido, ou seja, para cada vértice conhecemos seu grau e o índice inicial de sua lista de adjacência, calculados nos dois passos anteriores, podemos agora efetivamente montar a lista de adjacência compacta, representada por Ea. Seguindo a lógica do primeiro passo, atribuímos uma *thread* de GPU para cada vértice. Cada uma dessas *threads* estará responsável por preencher em Ea toda a lista de adjacência de um vértice em particular. A lista de adjacência de cada vértice inicia-se no índice referenciado no segundo valor do índice de Va e possui um *offset* igual ao grau de cada vértice armazenado no segundo valor de Va.
- **Ordenação das listas de adjacências:** Tendo o vetor Ea_n e Ea_d completamente preenchidos, podemos agora simplesmente ordenar cada lista adjacente apresentada em Ea. Seguindo a lógica da terceira etapa, atribuímos uma *thread* de GPU para cada vértice. Cada uma delas ordenará a lista de adjacência do vértice atribuído a ela. Ao invés de usar o Quicksort, adotamos o SelectionSort, uma vez que a complexidade deste algoritmo é sempre $O(n^2)$, no pior caso, melhor caso e caso médio, evitando assim um desbalanceamento de carga de trabalho entre as *threads*, o que é normalmente muito crítico no caso de paralelização em GPUs.

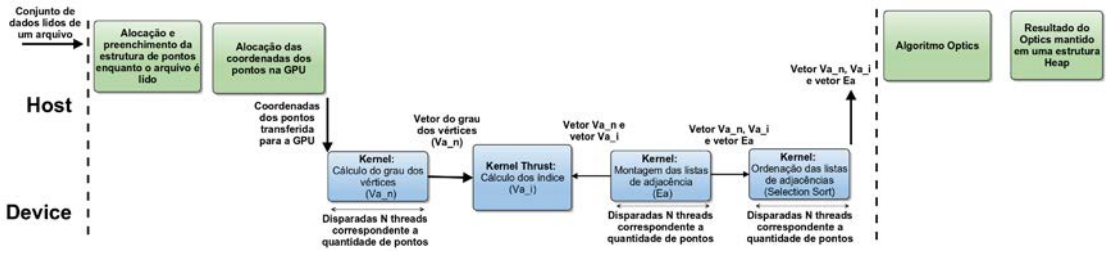


Figura 2. Ilustração do cálculo e transferência dos dados.

4. Avaliação Experimental

4.1. Amostra de dados

Para avaliar a estratégia proposta, geramos várias coleções de dados de diferentes tamanhos, variando entre 5.000 e 700.000 o número total de objetos de entrada em um espaço bidimensional (dois atributos). Para cada coleção, extraímos os tempos de execução referentes à construção do grafo, processo de ordenação das listas de adjacências,

processamento do OPTICS em si, bem como o tempo total gasto pelo algoritmo, tanto para a versão serial implementada em CPU quanto para a versão em GPU proposta nesse trabalho. A partir destes dados, avaliamos e comparamos o Speedup alcançado pela nossa abordagem utilizando GPU em cada parte do algoritmo, bem como na aplicação como um todo. Em todos os nossos testes usamos um conjunto de 20 *clusters Gaussianos* gerados aleatoriamente. Além disso, os parâmetros de entrada são fixos para todos os testes sendo $MinPts = 4$ e $\epsilon = 0.05$. A escolha de toda esta configuração das coleções de teste baseou-se no trabalho apresentado em [Böhm et al. 2009], em que os autores propõem e avaliam uma versão paralela do DBSCAN.

Todas as implementações foram feitas em C e C para CUDA (nVidia), e todos os experimentos foram realizadas utilizando uma máquina 4.1.13 GNU/Linux, com 32 GB de memória e um processador Intel Core processador i7-4930K de 3.40 GHz. Utilizamos também uma GPU Tesla K40c, com 2880 CUDA núcleos organizados em 15 multiprocessadores, com 12 GB de memória. A métrica de distância adotada foi a Distância Euclidiana. Nas subseções seguintes, primeiramente, apresentamos uma análise do perfil de execução do algoritmo a fim de verificar quais são as fases computacionalmente mais custosas para, assim, justificar as escolhas de paralelização descritas. Em seguida, apresentamos os resultados obtidos no processo de construção do grafo, e finalmente no algoritmo como um todo, apresentando o Speedup e os ganhos alcançados.

4.2. Execução do Profiling

A fim de avaliar quais etapas do algoritmo são computacionalmente mais custosas, medimos o tempo de execução para a construção do grafo, para o processo de ordenação das listas de adjacências, para o processamento do OPTICS, além do tempo total gasto pelo algoritmo em CPU. A Figura 3 mostra o tempos de execução de cada etapa para todos os conjuntos de dados testados. Observamos que o tempo de construção do grafo domina o tempo de execução do algoritmo para todos os conjuntos de dados testados, sendo 99,97% do tempo total para uma entrada de 700.000 objetos. Este resultado confirma que a paralelização da construção do grafo foi uma boa escolha. A etapa de ordenação também é uma operação importante, pois reduz a complexidade de algumas operações do OPTICS em si para $O(1)$. Apesar de termos paralelizado este passo, o mesmo representa uma fração pequena do tempo total. Assim, nossas análises foram concentradas no Speedup alcançado na construção do grafo e no tempo de execução total do algoritmo.

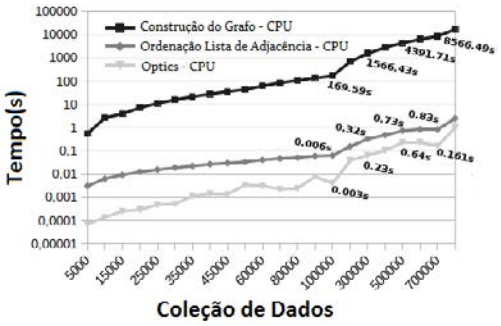
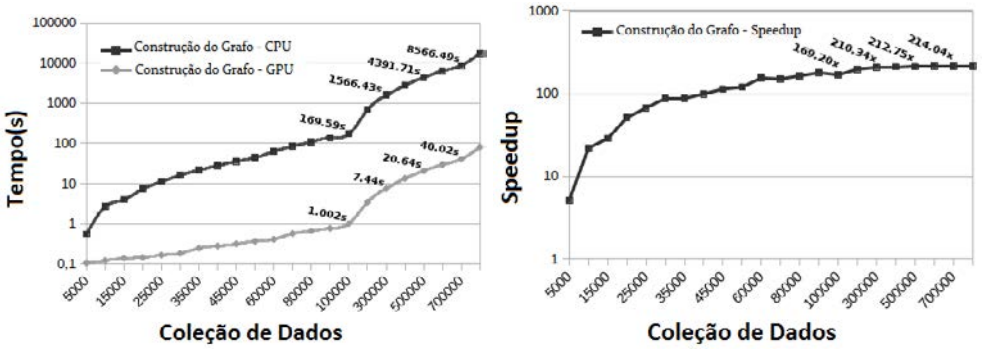


Figura 3. Tempo de execução de cada etapa na CPU

4.3. Avaliação da construção do grafo

Conforme descrito anteriormente, a construção de grafo apresenta a maior complexidade de tempo ($O(V^2)$), exigindo, assim, a maior parte do tempo de processamento de todo o

algoritmo. Na Figura 4, apresentamos os resultados obtidos com o paralelização em GPU deste passo. Como pode ser visto, existe uma redução significativa no tempo de execução desta etapa. O Speedup aumenta significativamente até $N = 100,000$. Para os valores de N superior a 100.000 o crescimento é menor, estabilizando por volta de $N = 600,000$ com um Speedup de $214\times$. Esta estabilização pode ser explicada pelo fato do aumento do tamanho de entrada representar também um aumentado no número de blocos de *threads* da GPU. Assim, a sobrecarga gerada para escalonar essas *threads* torna-se um fator limitante, estagnando o Speedup a partir desse ponto. De qualquer maneira, um Speedup de $214\times$ usando apenas uma GPU pode ser considerado um resultado contundente.

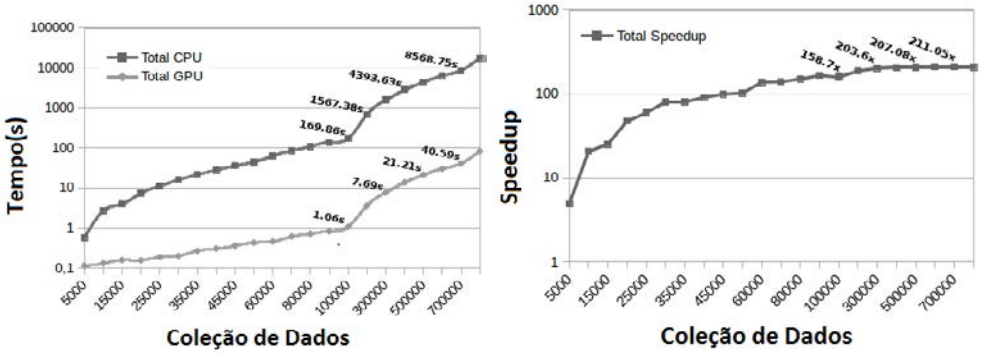


(a) Runtime (b) Speedup

Figura 4. Tempo de execução e Speedup da Construção do Grafo

4.4. Avaliação do tempo total

Na Figura 5 apresentamos os resultados relacionados ao tempo total de execução. Avaliando o tempo de execução alcançado por nosso algoritmo, podemos ver que o Speedup máximo atingido foi de $211\times$, diminuindo o tempo de execução de $8.568,75s$ na CPU para $40,59s$ na GPU, com $700,000$ pontos. Dessa forma, mesmo com uma implementação multicore deste algoritmo que apresentasse um Speedup linear, seriam necessários mais de 200 CPUs para atingir o mesmo resultado alcançado pela implementação em GPU. Se tomarmos em consideração o custo, os nossos resultados são ainda mais impressionantes. Enquanto uma GPU, como o que foi utilizada nos experimentos, custa cerca de US$2.000.00, a obtenção de uma configuração com 200 núcleos de CPU custaria um valor consideravelmente superior.



(a) Runtime (b) Speedup

Figura 5. Tempo de execução e Speedup Total

5. Conclusão e Trabalhos Futuros

Nesse trabalho, apresentamos uma nova abordagem para tornar o OPTICS computacionalmente viável baseada em uma estratégia de indexação de dados paralelizada em GPU. Representamos coleções de dados como grafo $G(V, E)$, onde V representa os objetos e E as arestas que ligam os objetos que têm distâncias entre si menores do que um raio ϵ. Construímos uma estrutura de dados adaptada da METIS [Karypis and Kumar 1998], adotando uma estratégia de paralelização para GPUs. Utilizando essa representação, a complexidade do OPTICS é reduzida para $O(E * logV)$. A fim de avaliarmos empiricamente nossa proposta, desenvolvemos vários conjuntos de testes onde o número de objetos a serem agrupados foi variado entre 5.000 e 700.000 objetos num espaço bidimensional. Em seguida, medimos o tempo de execução do OPTICS, tanto para uma implementação serial em CPU quanto para nossa abordagem em GPU. Nessa avaliação, mostramos que nossa proposta é até $200\times$ mais rápida do que sua versão sequencial usando CPU, usando apenas uma única GPU. Como trabalho futuro, temos proposta de novas estratégias de paralelização do OPTICS em si utilizando GPUs. Pretendemos também estender essas estratégias para usar várias GPUs, bem como avaliar todas as propostas sobre cenários de dados reais, onde o volume é muito grande e os dados são representados por várias dimensões.

Referências

Aggarwal, C. C. and Reddy, C. K. (2013). *Data Clustering: Algorithms and Applications*. Chapman & Hall/CRC, 1st edition.

Agrawal, R., Gehrke, J., Gunopulos, D., and Raghavan, P. (1998). Automatic subspace clustering of high dimensional data for data mining applications. In *ACM SIGMOD*.

Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R., and Rocha, L. (2013). G-dbscan: A gpu accelerated algorithm for density-based clustering. ICCS.

Ankerst, M., Breunig, M. M., Kriegel, H.-P., and Sander, J. (1999). Optics: Ordering points to identify the clustering structure. In *ACM SIGMOD*.

Blelloch, G. E. (1990). Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Böhm, C., Noll, R., Plant, C., and Wackersreuther, B. (2009). Density-based clustering using graphics processors. In *ACM CIKM*.

Christen, P. (2012). A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*.

Ester, M., peter Kriegel, H., S, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323.

Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392.

Patwary, M. A., Palsetia, D., Agrawal, A., Liao, W.-k., Manne, F., and Choudhary, A. (2013). Scalable parallel optics data clustering using graph algorithmic techniques. In *Proc. of ACM SC*.