Paralelização e Otimização em GPU do Problema do Par de Pontos Mais Próximos

Eduardo Ferreira Baroni[1], **Gabriel Medina Braga**[1], **Nahri Moreano**[1]

[1]Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

eduardobaroni97@gmail.com, medina_cdz@hotmail.com, nahri@facom.ufms.br

Resumo. *O Problema do Par de Pontos Mais Próximos consiste em encontrar, em um conjunto de pontos, o par mais próximo. As contribuições desse trabalho são: (a) a proposta de uma nova abordagem, baseada no particionamento espacial, para solução do problema; (b) o desenvolvimento de uma solução sequencial, usando a nova abordagem, que obtém desempenho melhor que soluções baseadas na abordagem tradicional de divisão e conquista; (c) o desenvolvimento de uma solução paralela em GPU, usando a nova abordagem, que atinge speedups de até 79,8 e 142,8 (para densidades de pontos baixa e alta, respectivamente), em relação à melhor solução sequencial; e (d) a proposta de otimizações para melhorar o desempenho das soluções desenvolvidas.*

Abstract. *The Closest Pair Problem consists in finding the closest pair of points in a set of points. The contributions of this work are: (a) the proposal of a new approach, based on spatial partitioning, for the solution of the problem; (b) the development of a sequential solution, based on the new approach, which achieves better performance than solutions based on the traditional divide and conquer approach; (c) the development of a parallel solution for execution in GPU, also based on the new approach, which reaches speedups up to 79.8 and 142.8 (for low and high density of points, respectively), with respect to the best sequential solution; and (d) the proposal of optimizations to improve the performance of the developed solutions.*

1. Introdução

O Problema do Par de Pontos Mais Próximos (*Closest Pair Problem* – CPP) consiste em, dado um conjunto N de pontos de dimensão D, encontrar o par mais próximo. Esse problema, da área de Geometria Computacional, pode ser aplicado para resolver problemas de diferentes campos [Rajasekaran and Pathak 2014]. Nesse trabalho é abordado o CPP para 2 dimensões (caso planar), usando a distância euclidiana.

Uma solução sequencial ótima para o CPP é baseada na técnica de divisão e conquista [Shamos 1975, Cormen et al. 2014]. Soluções paralelas para o CPP, baseadas em modelos teóricos de computação, são abordadas em [Atallah and Goodrich 1985, Blelloch and Maggs 2004, Wang et al. 2005, Yang 1999]. Uma implementação usando parallel STL e baseada na divisão e conquista obteve *speedup* de quase 3 em uma máquina com 24 núcleos [Sgier 2016]. A paralelização baseada na divisão e conquista e usando OpenMP atingiu *speedups* de até 3,1 em um processador *multicore* com 4 núcleos [Nakano and Carmo 2017].

A abordagem da divisão e conquista não favorece a exploração de paralelismo na solução do CPP, especialmente para plataformas *many-core*, pois o grau de paralelismo da solução reduz a cada nível da conquista. Este trabalho propõe a abordagem de particionamento espacial para resolução do CPP, que é usada para o desenvolvimento de uma solução sequencial e uma solução paralela adequada para execução em GPU. Também são apresentadas otimizações para melhorar o desempenho das soluções.

Este texto possui cinco seções. A solução do CPP usando o particionamento espacial é apresentada na Seção 2. Na Seção 3 são propostas a paralelização e otimizações para a solução do CPP. Os resultados obtidos em uma avaliação de desempenho são analisados na Seção 4. Por fim, a Seção 5 conclui a discussão e sugere trabalhos futuros.

2. Algoritmo para o CPP Baseado no Particionamento Espacial

A abordagem de particionamento espacial divide o espaço de pontos em regiões, ao longo do eixo X, de forma que cada região tenha a mesma quantidade de pontos (Figura 1). Com base nessa abordagem, foram desenvolvidas uma solução sequencial e uma solução paralela para execução em GPU. O Algoritmo 1 descreve o algoritmo sequencial que encontra o par de pontos mais próximos, tal que *deltaMinimo* terá a distância entre eles. Por simplicidade, o algoritmo omite a obtenção do par de pontos mais próximos, que é realizada atualizando esse resultado toda vez que *deltaMinimo* é atualizado.

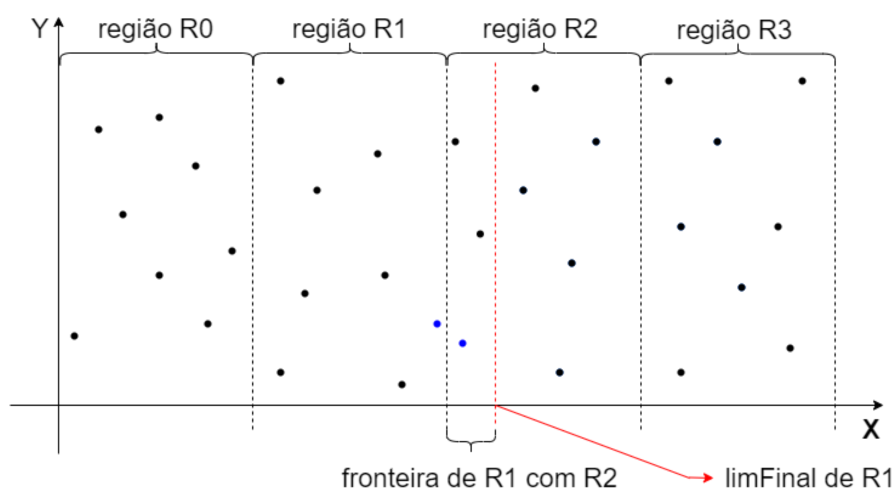

Figura 1. Abordagem de particionamento espacial para resolver o CPP

O número de regiões *nRegioes* no qual o espaço de pontos será dividido é determinado com cada região tendo *ptsRegiao* pontos, exceto possivelmente a última região. No passo 1, os pontos são ordenados pela coordenada X. A ordenação dos pontos e o particionamento espacial do espaço de pontos, segundo essa ordenação em X, permitem calcular a distância apenas entre pontos de uma mesma região, além de investigar pontos da fronteira entre duas regiões. No passo 2, calcula-se a estimativa *deltaInicial* para a distância mínima do conjunto de pontos, computando a distância entre cada ponto e o seguinte (na ordem em X) e obtendo o mínimo. Essa estimativa é um limite superior para a distância mínima do conjunto de pontos (isto é, $deltaMinimo \leq deltaInicial$) e é usada para definir o comprimento em X da fronteira entre uma região e a seguinte.

Algoritmo 1: Solução sequencial baseada no particionamento espacial

Entradas: Coordenadas X e Y dos N pontos e número de pontos por região *ptsRegiao*
Saída: Distância entre par de pontos mais próximos *deltaMinimo*

```
nRegioes ← N ÷ ptsRegiao                           // Calcula número de regiões
se N % ptsRegiao ≠ 0 então nRegioes ← nRegioes +1
Ordena(N, X, Y)    // Passo 1:  Ordena pontos segundo coordenada X
deltaInicial ← ∞   // Passo 2:  Calcula estimativa de distância mín
para j ← 0 até N − 2 faça
    a ← DistânciaEuclidiana(X[j], Y[j], X[j + 1], Y[j + 1])
    se a < deltaInicial então deltaInicial ← a
deltaMinimo ← deltaInicial         // Passo 3:  Encontra par mais próximo
para i ← 0 até nRegioes −2 faça     // Para cada região, exceto a última
10  limFinal = X[(i + 1)× ptsRegiao −1]+ deltaInicial
    para j ← i× ptsRegiao até (i + 1)× ptsRegiao −1 faça
12      para k ← j + 1 até k < N e X[k] ≤ limFinal faça
            a ← DistânciaEuclidiana(X[j], Y[j], X[k], Y[k])
14          se a < deltaMinimo então deltaMinimo ← a
para j ← i× ptsRegiao até N − 2 faça                  // Para última região
16  para k ← j + 1 até N − 1 faça
        a ← DistânciaEuclidiana(X[j], Y[j], X[k], Y[k])
        se a < deltaMinimo então deltaMinimo ← a
```

O passo 3 calcula, para cada região i, a distância entre cada ponto j da região e todos os pontos k seguintes (na ordem X) daquela região, e todos os pontos k da fronteira entre as regiões i e $i + 1$. A última região não possui fronteira com uma região seguinte. Para cada região i, exceto a última, calcula-se, na linha 10 do Algoritmo 1, o limite *limFinal* da fronteira entre i e $i + 1$ como a soma da coordenada X do último ponto da região i com o *deltaInicial*, como mostrado na Figura 1. Assim trata-se a situação em que os dois pontos mais próximos encontram-se em regiões distintas. Ao final do passo 3, *deltaMinimo* terá a distância entre o par de pontos mais próximos.

3. Paralelização

A solução do CPP baseada na abordagem de particionamento espacial apresenta alto grau de paralelismo e é adequada para processadores de muitos núcleos. Cada passo do Algoritmo 1 pode ser paralelizado. A solução paralela foi desenvolvida para execução em GPU, usando o modelo CUDA de programação paralela.

No passo 1, a ordenação é realizada através da função *sort_by_key* da biblioteca Thrust [Bell and Hoberock 2012] de algoritmos paralelos para GPU. No passo 2, a paralelização se dá com cada *thread* calculando a distância entre um ponto i e o ponto $i+1$ seguinte. O número de blocos é calculado usando o atributo *MaxThreadsPerBlock* da GPU. Em seguida, uma redução é realizada, através da função *min_element* da biblioteca Thrust, para obter o mínimo das distâncias calculadas pelas *threads*, que é a estimativa *deltaInicial*. No passo 3, cada bloco fica responsável por uma região i e cada *thread* do bloco, por um ponto j dentro da região i. Dessa forma, explora-se paralelismo entre

a computação de diferentes regiões e entre pontos de uma mesma região. Cada *thread* calcula a distância entre o ponto i e os pontos seguintes daquela região e da fronteira com a região seguinte. Em seguida, uma nova redução é realizada, para obter o mínimo das distâncias calculadas pelas *threads*, que é a solução *deltaMinimo* para o CPP.

Algumas otimizações são propostas para as soluções sequencial e paralela com o objetivo de melhorar seus desempenhos. A otimização 1 explora a ordenação dos pontos em X para reduzir a quantidade de cálculos de distância. Nos laços das linhas 12 e 16 do Algoritmo 1, calcula-se a distância entre o ponto j da região e todos os pontos k seguintes (na ordem X) da região e da fronteira com a região seguinte. Se a distância no eixo X entre j e k for maior ou igual ao *deltaMinimo*, então a distância entre j e os pontos seguintes a k também será e o laço pode ser interrompido. A otimização 2 reduz dinamicamente a fronteira entre regiões para diminuir a quantidade de cálculos de distância. Na linha 10 do Algoritmo 1, calcula-se o limite *limFinal* da fronteira entre as regiões i e $i + 1$ usando *deltaInicial* e esse limite é usado no controle do laço da linha 12. À medida que pares de pontos mais próximos são encontrados e *deltaMinimo* diminui, é possível reduzir a fronteira entre as regiões, fazendo com que o laço termine antes. Para isso, a linha 10 do Algoritmo 1 é modificada para calcular *limFinal* usando *deltaMinimo*, e toda vez que *deltaMinimo* é atualizado, dentro da condição na linha 14, *limFinal* é recalculado.

4. Resultados e Discussão

Experimentos foram realizados para avaliar o desempenho das soluções sequencial e paralela propostas e das otimizações desenvolvidas. As soluções sequencial e paralela básicas (*Sequencial V0* e *Paralelo V0*) são baseadas no particionamento espacial (Seções 2 e 3). A partir dessas soluções básicas, as soluções *Sequencial V1* e *Paralelo V1* foram obtidas com a aplicação da otimização 1. Nas soluções *Sequencial V2* e *Paralelo V2* foram aplicadas as otimizações 1 e 2 (Seção 3). A solução *Sequencial DC* é uma implementação sequencial, apresentada em [Nakano and Carmo 2017], baseada na divisão e conquista e com a aplicação de várias otimizações (entre elas, a eliminação da recursão).

A plataforma de execução foi um computador com processador Intel Xeon CPU E3-1270 v6, com frequência do *clock* 3,8GHz e 62,4GB de memória RAM, e uma GPU NVIDIA TITAN V, com 5.120 núcleos e 12GB de memória RAM. Foram geradas aleatoriamente entradas de 200M, 400M e 600M de pontos, uniformemente distribuídos no plano, com densidades baixa DB = 1/100M e alta DA = 1/1M. Densidade é a razão entre número de pontos e área do espaço de pontos. Para cada programa foram realizadas 5 execuções e os tempos de execução tiveram variação máxima de 1,4%. A medição de tempo dos programas paralelos inclui a execução no processador *host*, na GPU e a transferência de dados entre ambos. Em todas as execuções, adotou-se *ptsRegiao* = 32.

A Figura 2(a) mostra o tempo de execução das soluções sequenciais. A otimização 1 proporciona um grande ganho de desempenho, pois interrompe os laços mais internos do passo 3 do Algoritmo 1. A otimização 2 tem um impacto menor, por ser aplicada em conjunto com a anterior. Assim, a solução Sequencial V2 reduz apenas um pouco mais o tempo de execução, em relação à Sequencial V1. A solução Sequencial DC possui o pior tempo de execução, pois mesmo otimizada para reduzir cálculos de distância, realiza a execução completa da árvore binária de controle da divisão e conquista. A melhor solução sequencial (V2) é 1,87 vezes mais rápida que a Sequencial DC para 600M pontos.

A Figura 2(b) mostra o tempo de execução das soluções paralelas. A solução Paralelo V0 básica já apresenta um tempo de execução muito baixo. A otimização 1 proporciona um pequeno ganho de desempenho, tornando a solução Paralelo V1 a melhor solução paralela, enquanto a otimização 2 não trouxe ganho significativo.

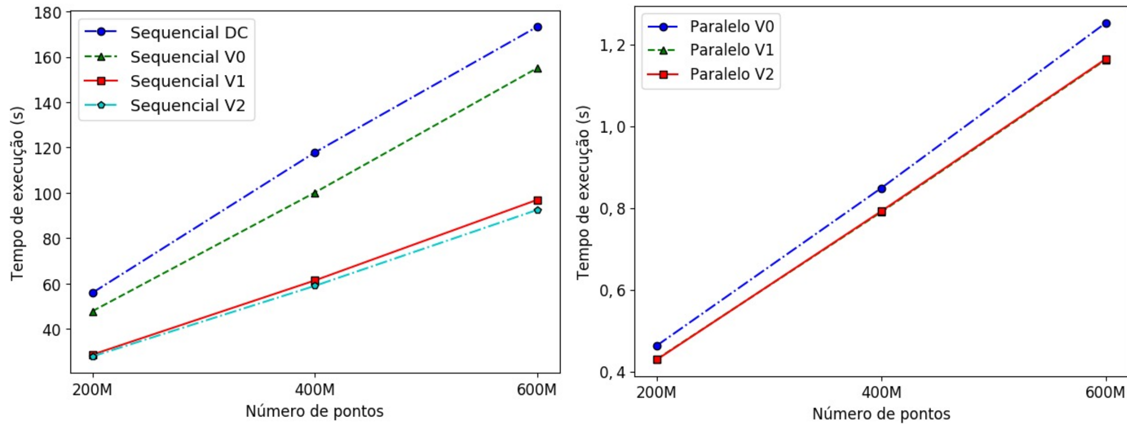

Figura 2. Tempo de execução das soluções (a) sequenciais e (b) paralelas, para densidade DB = 1/100M

A Figura 3 mostra o tempo de execução das melhores soluções sequencial (V2) e paralela (V1) e o *speedup* desta em relação à primeira, variando a densidade. A solução paralela é 79,8 e 142,8 vezes mais rápida que a sequencial, para densidades baixa e alta, respectivamente, para 600M pontos. O *speedup* aumenta à medida que o tamanho da entrada aumenta, para ambas densidades, ou seja, o tempo de execução da solução paralela cresce de maneira mais lenta que o da sequencial. O *speedup* para densidade alta é bem maior, pois ela piora o desempenho da solução sequencial muito mais que o da paralela.

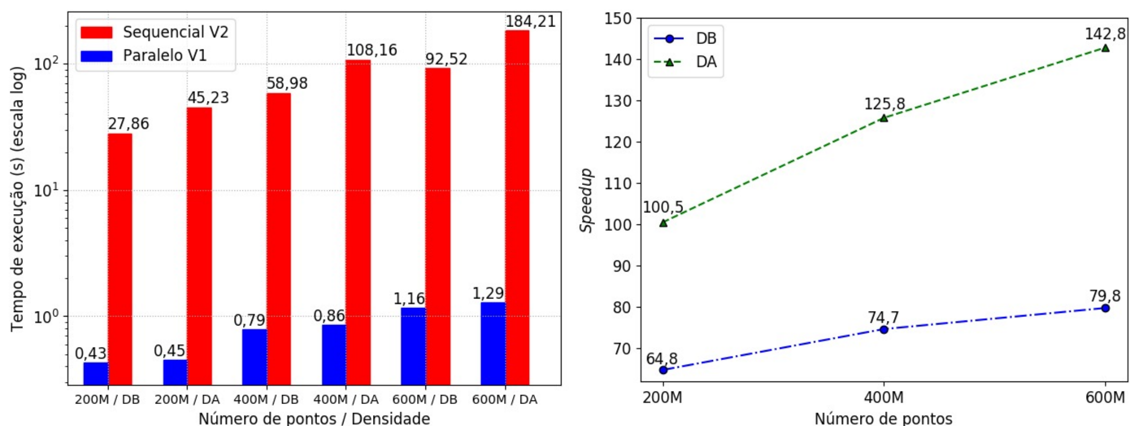

Figura 3. (a) Tempo de execução (escala log) das melhores soluções sequencial e paralela, e (b) *Speedup* da melhor paralela em relação à melhor sequencial, para densidades DB = 1/100M e DA = 1/1M

5. Conclusões

Esse trabalho propôs a abordagem de particionamento espacial para resolver o CPP, utilizando as estratégias de ordenar os pontos segundo uma das coordenadas, calcular de forma rápida uma estimativa inicial da distância mínima e particionar o espaço de pontos em regiões que podem ser tratadas separadamente. Tais estratégias permitiram o desenvolvimento de uma solução sequencial com bom desempenho e de uma solução paralela, com alto grau de paralelismo e adequada para processadores *many-core*.

A solução sequencial apresentada, combinada com as otimizações propostas, obteve tempo de execução 1,87 vezes menor que uma solução otimizada baseada no algoritmo ótimo da divisão e conquista. As otimizações desenvolvidas mostraram-se bastante eficazes para reduzir o tempo de execução da solução sequencial, proporcionando um ganho de desempenho de 1,68 vezes em relação à mesma solução não otimizada. A solução paralela apresentou um ótimo desempenho, alcançando *speedups* de 79,8 e 142,8 (para densidades de pontos baixa e alta, respectivamente), em relação à melhor solução sequencial, para entradas com 600M pontos. Além disso, o desempenho da solução paralela é pouco afetado pela densidade de pontos da entrada, que interfere significativamente no desempenho da solução sequencial.

Como trabalho futuro, soluções para o caso 3D do CPP podem ser desenvolvidas estendendo de forma simples as soluções para o caso planar baseadas no particionamento espacial, o que não acontece com a solução baseada na divisão e conquista.

Referências

Atallah, M. and Goodrich, M. (1985). Efficient parallel solutions to some geometric problems. Technical Report 85-504, Purdue University.

Bell, N. and Hoberock, J. (2012). *Thrust: A Productivity-Oriented Library for CUDA*, chapter 26 – GPU Computing Gems. Elsevier.

Blelloch, G. and Maggs, B. (2004). *Parallel Algorithms*, chapter 10 – Computer Science Handbook. Chapman & Hall/CRC, 2 edition.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2014). *Introduction to Algorithms*. MIT Press.

Nakano, J. and Carmo, L. (2017). Otimizações e paralelização do problema do par de pontos mais próximos. Technical report, Universidade Federal de Mato Grosso do Sul.

Rajasekaran, S. and Pathak, S. (2014). Efficient algorithms for the closest pair problem and applications. `https://arxiv.org/abs/1407.5609`. Acessado em fevereiro 2020.

Sgier, S. (2016). Prototype implementation and evaluation on algorithmic motifs in scientific computing. Master's thesis, Swiss Federal Institute of Technology Zurich.

Shamos, M. (1975). Geometric complexity. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing*, pages 224–233.

Wang, Y., Horng, S., and Wu, C. (2005). Efficient algorithms for the all nearest neighbor and closest pair problems on the linear array with a reconfigurable pipelined bus system. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):193–206.

Yang, C. (1999). Computational geometry on the broadcast communication model. *Journal of Information Science and Engineering*, 16(3):383–395.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001, e do Programa de Educação Tutorial (PET).