Algoritmo de Dijkstra: Proposta de Paralelização em CUDA

Sebastião V. Guimarães Neto[1], **Hélio Guardia**[1]

[1]Departamento de Computação - Universidade Federal de São Carlos

sneto@estudante.ufscar.br

Resumo. *Este trabalho apresenta um estudo sobre a paralelização do algoritmo de Dijkstra para busca do menor caminho em grafos utilizando CUDA e analisa seu desempenho em comparação com a versão sequencial, comparando os tempos de execução e o speedup obtido.*

1. Introdução

O problema de encontrar o menor caminho entre vértices em um grafo ponderado possui diversas aplicações em áreas como redes de transporte, telecomunicações e computação. Este estudo propõe a paralelização do algoritmo de Dijkstra com o objetivo de melhorar seu desempenho em grafos de grande porte. Inicialmente, descreve-se a versão sequencial do algoritmo, bem como a estrutura dos grafos gerados para os testes. Em seguida, apresentam-se as estratégias de paralelização empregadas com CUDA para execução em GPU. Por fim, os resultados obtidos são discutidos, com ênfase nos tempos de execução e no *speedup*.

2. O Algoritmo de Dijkstra

O algoritmo de Dijkstra, proposto em 1959, é amplamente reconhecido como um dos métodos clássicos para encontrar o menor caminho entre vértices em grafos ponderados [Dijkstra 1959]. Desde sua formulação, diversas abordagens têm sido propostas com o intuito de acelerar sua execução, incluindo variações baseadas em estruturas de dados mais eficientes, como filas de prioridade, e estratégias de paralelização utilizando GPU [Harish and Narayanan 2007].

Na implementação tradicional, utiliza-se uma matriz de adjacência para armazenar os pesos das conexões entre os vértices. A cada iteração, a matriz é percorrida para identificar o vértice com a menor distância entre aqueles ainda não visitados. Essa abordagem resulta em uma complexidade de tempo de $O(n^2)$, onde n representa o número de vértices do grafo.

3. Desenvolvimento do Trabalho

Este trabalho implementa uma versão paralela do algoritmo de Dijkstra com CUDA. Parte da lógica sequencial do programa permanece em CPU, enquanto a GPU acelera etapas que se beneficiam do paralelismo massivo, como o relaxamento das arestas e parte da seleção do nó com menor distância.

Os grafos foram gerados aleatoriamente com controle de nós, arestas e semente, e representados por uma matriz de adjacência linearizada para compatibilidade com CUDA. A função CUDA principal (`relax_edges`) realiza o relaxamento das arestas a partir de

um nó u escolhido na CPU; cada *thread* da GPU verifica se é possível melhorar a distância até o vértice v, utilizando `atomicMin` para garantir consistência em atualizações concorrentes.

Código 1. Trecho do kernel CUDA pelo relaxamento paralelo das arestas

```
__global__ void relax_edges(int *adjMatrix, int *dist, int *visited,
    int u, int n) {
    int v = blockIdx.x * blockDim.x + threadIdx.x;
    if (v < n && !visited[v]) {
        int weight = adjMatrix[u * n + v];
        if (weight!=INF && dist[u]!=INF && dist[u]+weight < dist[v])
            atomicMin(&dist[v], dist[u] + weight);
    }
}
```

Além disso, a função `minDistance`, que originalmente realizava a busca sequencial do vértice com menor distância, foi parcialmente paralelizada. Um kernel CUDA realiza uma redução paralela em blocos para encontrar candidatos locais ao menor valor, que são então analisados pela CPU para determinar o mínimo global. Essa estratégia reduz significativamente o custo dessa operação sem eliminá-lo por completo, já que a decisão final ainda depende da CPU.

Código 2. Trecho do kernel para encontrar o menor valor de `dist[]`

```
__global__ void min_distance_kernel(int *dist, int *visited, int *
    resultIndices, int *resultDists, int n) {
    __shared__ int localMinDist[BLOCK_SIZE];
    __shared__ int localMinIndex[BLOCK_SIZE];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    int myDist = INF;
    int myIndex = -1;
    if (i < n && visited[i] == 0 && dist[i] < INF){   myDist = dist[i];
        myIndex = i;
    }
    localMinDist[tid] = myDist;
    localMinIndex[tid] = myIndex;
    __syncthreads();
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            if (localMinDist[tid + stride] < localMinDist[tid]) {
                localMinDist[tid] = localMinDist[tid + stride];
                localMinIndex[tid] = localMinIndex[tid + stride];}
        }
        __syncthreads();
    }
    if (tid == 0) {
        resultIndices[blockIdx.x] = localMinIndex[0];
        resultDists[blockIdx.x] = localMinDist[0];}}
```

A função `dijkstra_cuda`, executada na CPU, realiza as iterações principais do algoritmo. Em cada passo, ela:

- Executa a versão paralelizada de `minDistance` para encontrar o nó u com menor distância;

- Atualiza o vetor de visitados e envia para a GPU;
- Lança o kernel CUDA `relax_edges` com uma thread por vértice;
- Sincroniza a GPU e copia o vetor de distâncias atualizado.

Apesar da comunicação frequente entre CPU e GPU, a aceleração nas etapas paralelizadas torna a abordagem eficiente em grafos grandes. Trata-se de uma estratégia híbrida, alinhada ao que foi proposto por Harish e Narayanan [Harish and Narayanan 2007] e Martín et al. [Martín et al. 2009], que observaram ganhos ao paralelizar apenas as partes mais custosas do algoritmo.

Em termos de complexidade, a versão paralela mantém o mesmo pior caso da implementação sequencial, $O(n^2)$, devido à necessidade de percorrer todos os vértices e arestas em algumas etapas. No entanto, a divisão do trabalho entre milhares de threads permite acelerar significativamente o tempo de execução real. Por exemplo, o relaxamento das arestas, originalmente sequencial, é realizado em paralelo com complexidade $O(n)$ por iteração, reduzindo o tempo total sem alterar a ordem teórica. A seleção do nó mínimo ainda depende parcialmente da CPU, mas a combinação de paralelismo nas etapas mais custosas gera ganhos substanciais na prática, como evidenciado nos resultados.

4. Resultados e Discussão

Os testes foram realizados no Google Colab, utilizando uma GPU NVIDIA Tesla T4 (CUDA 11.x), com 2.560 núcleos e 16 GB de memória. Por ser uma plataforma gratuita, não há controle preciso sobre a alocação de recursos, o que pode gerar variações nos tempos. Nos experimentos CUDA, foram utilizados blocos com `BLOCK_SIZE = 32` threads e número de blocos calculado como `(n + 31) / 32`, onde `n` representa o número de nós do grafo, assegurando que cada vértice seja processado por uma thread. A constante `INF = INT_MAX` foi adotada para representar distâncias infinitas.

As implementações sequencial (CPU) e paralela (CUDA) foram executadas com o mesmo conjunto de parâmetros, variando-se o número de nós e arestas dos grafos gerados. Os tempos de execução foram medidos em segundos e estão apresentados na Figura 1 e na Tabela 1.

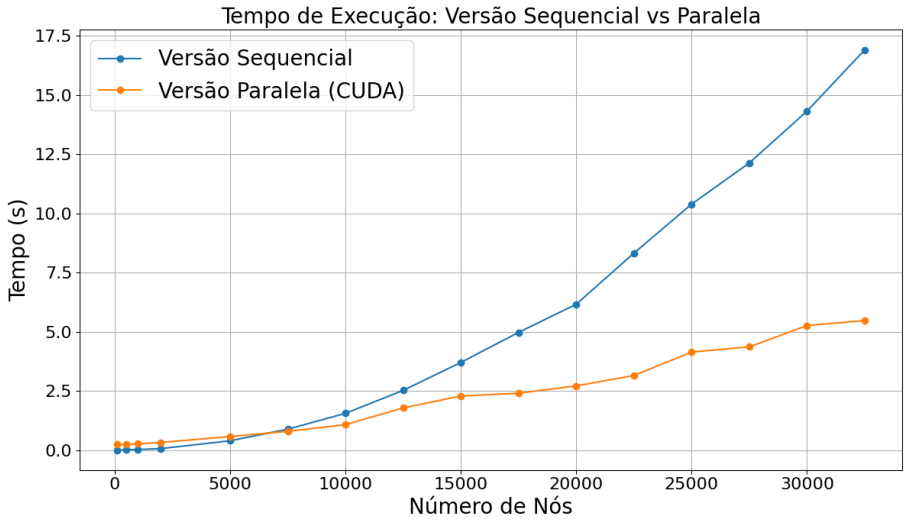

Figura 1. Tempo de execução das versões sequencial e paralela para diferentes tamanhos de grafo.

Tabela 1. Tempos de execução e speedup para diferentes tamanhos de grafo.

Nós	Arestas	Tempo CPU (s)	Tempo GPU (s)	Speedup
100	150	0.00	0.24	0.01
500	750	0.01	0.23	0.02
1000	1500	0.02	0.26	0.06
2000	3000	0.06	0.32	0.19
5000	7500	0.39	0.57	0.68
7500	11250	0.88	0.79	1.11
10000	15000	1.55	1.07	1.45
12500	16250	2.52	1.78	1.42
15000	22500	3.70	2.28	1.62
17500	26250	4.97	2.40	2.07
20000	30000	6.16	2.71	2.27
22500	33750	8.32	3.15	2.65
25000	37500	10.39	4.14	2.51
27500	41250	12.13	4.36	2.78
30000	45000	14.32	5.26	2.72
32500	48750	16.91	5.47	3.09

Em grafos pequenos, a versão sequencial é mais eficiente devido ao overhead das transferências. À medida que o número de nós cresce, a paralela se torna mais vantajosa, especialmente no relaxamento das arestas. A Tabela 1 mostra crescimento no speedup, atingindo 3,09$\times$ nas maiores instâncias, o que evidencia a eficácia da abordagem híbrida. Embora a seleção do nó mínimo ainda dependa da CPU, os ganhos da paralelização parcial compensam essa limitação. A partir de cerca de 7.500 nós, a versão paralela supera consistentemente a sequencial, indicando que os ganhos de paralelismo passam a compensar os custos de comunicação.

Como trabalho futuro, propõe-se eliminar essa dependência com estruturas paralelas na GPU e ampliar os testes para outras arquiteturas e topologias de grafos (estrela, árvore, clique, cíclico), variando o diâmetro e o grau médio, a fim de avaliar o impacto estrutural sobre o desempenho.

Referências

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, pages 197–208. Springer.

Martín, P. J., Asenjo, R., and Gavilanes, A. (2009). Cuda solutions for the sssp problem. In Bader, M., Bézard, M., Botelho, L. C. L., Inês, P. R. M., and ao M. L. M. Palma, J., editors, *High Performance Computing for Computational Science – VECPAR 2008*, volume 5336 of *Lecture Notes in Computer Science*, pages 203–216. Springer.