Aceleração da Busca em Largura em Grafos de Larga Escala utilizando CUDA

Erick Rocha Amorim[1], **Rodrigo Lopes de Aquino**[1], **Nahri Moreano**[1,2]

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

[2]Professora responsável

{erochaamorim,rodrigolaquino}@gmail.com, nahri@facom.ufms.br

1. Introdução

A *Compute Unified Device Architecture* (CUDA) [NVIDIA Corporation], arquitetura desenvolvida pela NVIDIA, conta com uma plataforma adequada para que os programadores possam explorar o alto poder computacional das GPUs (*Graphic Processing Units*) para computação de propósito geral. Neste trabalho, apresentaremos experimentos sobre a aceleração da busca em largura sobre grafos de larga escala utilizando a plataforma CUDA.

2. Desenvolvimento

2.1. Algoritmo Paralelo de Busca em Largura

O algoritmo paralelo de busca em largura proposto por [Harish and Narayanan 2007] foi usado como ponto de partida para o desenvolvimento das soluções paralelas implementadas neste trabalho. Este algoritmo é baseado no modelo de programação CUDA é composto por dois *kernels*, o *kernel* principal, apresentado no Algoritmo 1, e um *kernel* de sincronização, apresentado no Algoritmo 2.

Algoritmo 1: *Kernel* principal da busca em largura em GPU

```
tid ← getThreadId
if F_a[tid] then
    F_a[tid] ← falso
    foreach vizinho nid de tid do
        if X_a[nid] = falso then
            C_a[nid] ←
            C_a[tid] + 1
            F_ua[nid] ←
            verdadeiro
        end
    end
end
```

Algoritmo 2: *Kernel* de atualização da busca em largura em GPU

```
tid ← getThreadId
if F_ua[tid] then
    F_a[tid] ← verdadeiro
    X_a[tid] ←
    verdadeiro
    F_ua[tid] ← falso
    terminar ← falso
end
```

O *kernel* principal utiliza os vetores F_a e X_a, que representam a fronteira da busca e os vértices que já foram visitados, respectivamente, para coordenar os vértices que serão visitados na iteração atual. Ao final da execução, o vetor de custos C_a conterá a distância até a raiz de cada um dos vértices.

O vetor de atualização da fronteira, F_{ua}, guarda os vértices que deverão ser visitados na próxima iteração. Essa atualização é controlada pelo *kernel* de sincronização, que atualiza o vetor F_a usando o vetor F_{ua}. Este segundo *kernel* é necessário pois não há, em CUDA, forma de sincronizar todas as *threads* através de blocos distintos.

2.2. Soluções para Busca em Largura em GPU

A partir do algoritmo paralelo de busca em largura proposto por [Harish and Narayanan 2007], foram desenvolvidas quatro soluções em CUDA. A primeira solução é uma implementação elaborada diretamente a partir dos Algoritmos 1 e 2, onde todas as estruturas de dados estão armazenadas na memória global da GPU.

A segunda solução é baseada na primeira e utiliza duas variáveis locais a mais no *kernel* principal, uma para armazenar o custo do vértice e outra para armazenar o índice do vértice vizinho corrente, economizando acessos à memória global. O *kernel* de atualização não sofre alterações, permanecendo idêntico ao da primeira solução.

A terceira solução em GPU usa como ponto de partida a segunda solução, porém armazena na memória compartilhada, uma estrutura de dados para as arestas do grafo, e precisa copiá-las da memória global para a compartilhada. Para isso, o *kernel* principal é bastante alterado. O *kernel* de atualização permanece o mesmo da primeira solução. Apesar das alterações no *kernel* principal melhorarem o acesso à memória global, o baixo reuso de dados e a carga extra de trabalho causaram redução do desempenho.

A quarta solução em GPU usa como ponto de partida a segunda solução, e explora a otimização do acesso à memória global. Para isso, no *kernel* principal a *thread* não visita os vértices vizinhos, apenas os marca para visita, e em um novo *kernel*, cada *thread* realiza a visita do próprio vértice.

2.3. Resultados

Um computador *desktop* com processador Intel Core i5 650 e 2GB de memória RAM foi utilizado para executar a implementação sequencial. Este computador também foi utilizada como *host* para a GPU, conectada a ele através de um barramento PCI-Express. A GPU utilizada é uma NVIDIA GeForce GT 430, com 1GB de memória global, contendo um SM formado por 48 SPs (CUDA *cores*). A Figura 1 compara o tempo de execução das soluções em GPU com aquele obtido pela implementação sequencial.

Mesmo com uma baixa eficiência no acesso à memória global e uma ocupação relativamente baixa da GPU, a primeira solução obteve um *speedup* de cerca de 2,1 em relação à solução sequencial.

A segunda solução obteve desempenho quase idêntico à primeira, alcançando um *speedup* inexpressivo, de em média 1,01, em relação à primeira solução. Era de se esperar que o ganho desempenho fosse mais significativo, já que supostamente a nova solução realiza menos acessos à memória. Entretanto, a GPU utilizada possui capacidade de computação 2.1 e conta com uma cache L2. Na primeira solução em tempo de compilação, foram alocados registradores para aqueles dois valores, ou a cache foi utilizada em tempo de execução, devido à reutilização dos valores. Mesmo se as otimizações automáticas não tivessem, a frequência de reuso desses dados é reduzida, logo o *speedup* da segunda solução em relação à primeira ainda seria limitado.

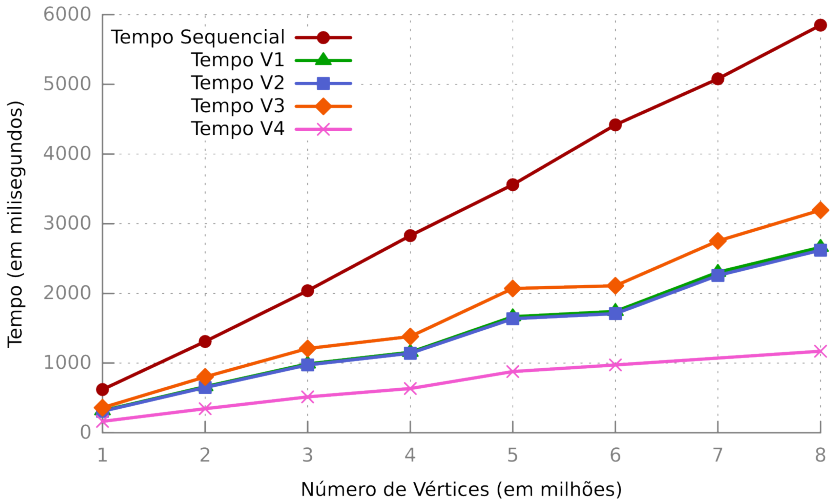

Figura 1. Comparação dos tempos de execução das soluções

Apesar do aumento da ocupação e do melhor aproveitamento da largura de banda, a terceira solução obteve um resultado inferior à segunda, em termos de tempo de execução. Isso se deve ao fato de que a cada nível da busca, um número limitado de vértices realmente utiliza suas arestas, e estes vértices estão distribuídos de forma aleatória entre os blocos. Por mais que a memória compartilhada tenha sido usada de forma eficiente, muitas arestas foram transferidas sem necessidade, constituindo uma carga de trabalho improdutiva. Mesmo assim, a terceira solução ainda é mais rápida que a solução sequencial da busca em largura.

As modificações melhoraram ligeiramente o desempenho do *kernel* principal na quarta solução, em relação à segunda solução. Além disso, o novo *kernel* possui alta eficiência de leitura. Com essa otimização, a quarta solução em GPU obteve um *speedup* médio de cerca de 4,15 em relação à solução sequencial, o melhor desempenho dentre todas as soluções apresentadas neste trabalho.

3. Considerações Finais

A partir da solução inicial, baseada no algoritmo proposto por [Harish and Narayanan 2007], experimentos foram realizados, buscando melhorar o desempenho a cada nova solução desenvolvida. Por fim, a melhor solução paralela alcançou um *speedup* de 4,15 em relação à solução sequencial, resultado aparentemente modesto, mas muito satisfatório dada capacidade da GPU utilizada.

Referências

[Harish and Narayanan 2007] 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*, pages 197–208.

[NVIDIA Corporation] NVIDIA Corporation. CUDA Parallel Computing Plataform. http://www.nvidia.com/object/cuda_home_new.html. Acessado em maio de 2014.