relat_chen

November 5, 2019

1 Projeto 3 - Supercomputação

2 TSP GPU

2.0.1 Lucas Chen Alba

2.1 Descrição do problema

Este projeto consiste em realizar simulações do conhecido problema do caixeiro-viajante (https://en.wikipedia.org/wiki/Travelling_salesman_problem) e analisar o ganho de velocidade com a utilização de técnicas de computação paralela e otimizações. A simulação consiste em encontrar o caminho ótimo ou sub-ótimo para um dado número de nós na rede do caxeiro. A técnica usada é a de montar sequências aleatórias e selecionar a melhor dentre elas

2.2 Medições de tempo

Para as medições do tempo gasto nas simulações foi utilizada a biblioteca *chrono* para CPU e a biblioteca nativa do CUDA para GPU.

2.2.1 Tamanhos das entradas

Foram testadas as entradas com os seguintes número de nós:
100, 200 nós para a simulação em CPU e GPU

2.2.2 Número de execuções

Para cada tamanho de entrada, foram executadas as simulações 5 vezes para maior confiança nos resultados.

A simulação utiliza-se de duas configurações de *flags* de compilação distintas:
-O3 (para GPU)
-O3 -fopenmp (para CPU)

2.3 Organização geral do código

2.3.1 Local Search (CPU)

Para a otimização discreta "Local Search" foram implementadas duas novas funções:
void local_search(std::vector<std::vector> points, double &best_cost, std::vector<std::vector> &best_sol): Esta função irá receber um caminho aleatório e irá

otimizá-lo até chegar em uma solução sub-ótima, significando um mínimo local. Ela faz isso a partir da checagem de "cruzamentos" entre dois pares de pontos quaisquer do caminho, ela então realiza um "swap" dos pontos, assim, quando todos os cruzamentos forem resolvidos, este será o mínimo local.

bool check_intersec(std::vector p1, std::vector p2, std::vector q1, std::vector q2): Esta função recebe uma sequência de 4 pontos e checa se há alguma interseção entre os segmentos de reta formados pelos dois pares de pontos.

Na função *main()* serão gerados 10000 caminhos aleatórios, caminhos estes que serão passados para chamadas da função local_search(), este número define a quantidade de caminhos iniciais aleatórios seráo otimizados pelo Local Search.

2.3.2 Random (GPU, CUDA)

Este código realiza a divisão de tarefas em "kernels". Inicialmente são pré-calculadas as distâncias entre os nós da rede. Essas distâncias são então guardadas em um vetor **device** (memória global da GPU).

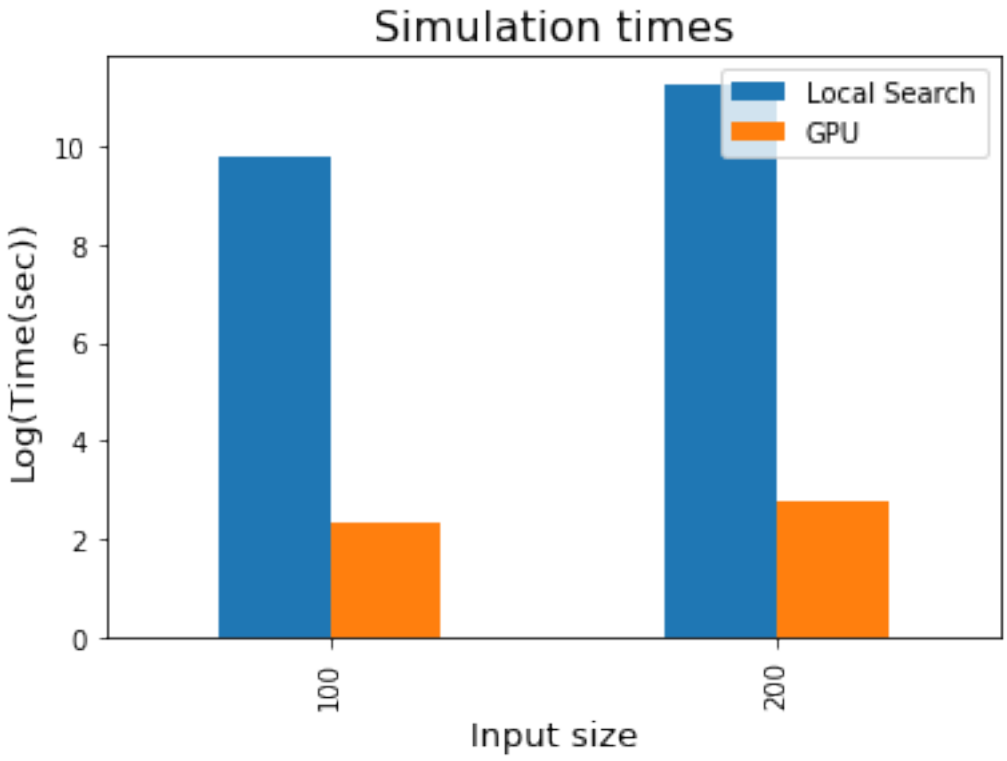
Assim, podemos criar um kernel para cada caminho aleatório. Foram criados 10000 kernels, cada um realizando então um "shuffle" nos caminhos, guardando a respectiva sequência e a distância (calculada a partir do vetor de distâncias) total em um outro vetor **device**. Assim, conseguimos utilizar a função da biblioteca thrust (thrust::min_element) para localizar o índice do menos caminho.

```
In [3]: import pandas as pd
        import matplotlib.pyplot as plt
        import math
        import statistics

In [13]: # Execution times
         cpu100_times = [17.8287, 17.8200, 17.8235, 17.8299, 17.8265]
         cpu200_times = [69.9345, 86.0268, 87.1931, 76.3408, 70.3024]


         cpu100_mean = statistics.mean(cpu100_times)
         cpu200_mean = statistics.mean(cpu200_times)
         print("CPU mean time (input 100): ", cpu100_mean)
         print("CPU mean time (input 200): ", cpu200_mean)

         cpu100_mean = math.log(1000*(statistics.mean(cpu100_times)))
         cpu200_mean = math.log(1000*(statistics.mean(cpu200_times)))

         gpu100_times = [0.0106, 0.0106, 0.0106, 0.0106, 0.0106]
         gpu200_times = [0.0159, 0.0160, 0.0159, 0.0159, 0.0160]


         gpu100_mean = statistics.mean(gpu100_times)
         gpu200_mean = statistics.mean(gpu200_times)
         print("GPU mean time (input 100): ", gpu100_mean)
         print("GPU mean time (input 200): ", gpu100_mean)
```

```
        gpu100_mean = math.log(1000*(statistics.mean(gpu100_times)))
        gpu200_mean = math.log(1000*(statistics.mean(gpu200_times)))


        # Mean times
        groups = [[cpu100_mean, gpu100_mean], [cpu200_mean, gpu200_mean]]
        group_labels = ['100', '200']

        # Convert data to pandas DataFrame.
        df = pd.DataFrame(groups, index=group_labels).T

        # Plot
        pd.concat(
            [df.loc[0].rename('Local Search'),
             df.loc[1].rename('GPU')],
            axis=1).plot.bar()
        plt.xlabel('Input size', fontsize=13)
        plt.ylabel('Log(Time(sec))', fontsize=13)
        plt.title('Simulation times', fontsize=16)
```

```
CPU mean time (input 100):  17.82572
CPU mean time (input 200):  77.95952
GPU mean time (input 100):  0.0106
GPU mean time (input 200):  0.0106
```

```
Out[13]: Text(0.5, 1.0, 'Simulation times')
```



2.4 Resultados e análises

Percebemos um alto ganho no desempenho quando comparamos a versão CPU com a GPU. Isso se deve ao fato da GPU possuir muitos kernels que realizam as terefas simultaneamente, enquante na CPU (Local Search) cada task realiza também as tarefas em paralelo, porém é necessário um bloco de código inserido em uma região crítica (#pragma omp critical), para evitar escritas simultâneas na memória.

```
In [ ]:
```