Universidade Federal de São Carlos

Programa de Pós-Graduação em Ciência da Computação

CCO-129-7 Introdução à Computação de Alto Desempenho (2021/2)

Exercício Programa - EP 6

OpenHPC UFSCar

Alcides Mignoso e Silva 760479

Submission Date : 18/04/2022

1 Atividade

O objetivo desta atividade é paralelizar a execução do algoritmo de Warshall para que seja executado em GPUs através de CUDA, visando otimizações utilizando memória compartilhada da GPU. O código base utilizado foi o existente em `https://github.com/HPCSys-Lab/HPC-101/blob/main/examples/warshall/appWarshall.c` e este relatório visa discutir valores de tempo de execução, speedup e escalabilidade das execuções.

As execuções foram realizadas utilizando matrizes com o número de linhas e colunas igual a 1024, 2048, 2071 e 4096, em um computador equipado com uma GPU RTX 2060 SUPER utilizando CUDA na versão 11.6, e uma CPU Ryzen 7 3700X.

Em relação a paralelização utilizando a memória compartilhada dos blocos da GPU, a estratégia utilizada foi fazer com que cada thread copiasse uma porção da matriz que seria acessada (na verdade se torna uma cópia de uma porção do vetor tendo em vista que somente uma linha é acessada por cada bloco) composta pela quantidade de $1/numero\_threads * numero\_elementos$ elementos. Outras otimizações também foram aplicadas evitando acessos desnecessários - como pode ser visto na linha 6 do código referenciado, evitando que um mesmo acesso seja feito várias vezes dentro do loop de cada thread.

O melhor comportamento para a GPU apareceu com blocos de 16x16 threads, sendo o aumento do número de threads (em relação a versão que não utiliza memória compartilhada) justificável em decorrência do melhor aproveitamento da memória compartilhada.

```
1  __global__ void warshall_gpu(int *m, int num_threads) {
2    int k = blockIdx.x * blockDim.x + threadIdx.x;
3    int i = blockIdx.y * blockDim.y + threadIdx.y;
4
5    __shared__ int m_column_shared[NUM_ELEMENTS];
6    int common_m_i_k = getAt(m, i, k) == 1;
7
8    if (k >= NUM_ELEMENTS || i >= NUM_ELEMENTS)
9      return;
10   for (int j = threadIdx.x * (NUM_ELEMENTS / num_threads);
11        j < (threadIdx.x + 1) *
12             (NUM_ELEMENTS / num_threads) && j < NUM_ELEMENTS;
13        j++) {
14     m_column_shared[j] = getAt(m, k, j);
15   }
16   __syncthreads();
17
18   if(!common_m_i_k) return;
19
20   for (int j = 0; j < NUM_ELEMENTS; j++) {
21     if (m_column_shared[j] == 1) {
22       setAt(m, i, j, 1);
23     }
24   }
25 }
```

2 Resultados

2.1 Tabelas

A tabela abaixo trás os tempos de execução do programa (em segundos) na versão sequencial e na paralelizada.

Dim. matriz	T(sequential)	T(cuda)	T(cuda opt)	Speedup	Speedup opt
1024	4,2127	0,0247	0,0085	170,66614	495,6144706
2048	26,2407	0,1462	0,0579	179,542671	453,2072539
3072	87,6586	0,5060	0,1534	173,2430581	571,4378227
4096	218,1629	1,1702	0,3669	186,4375653	594,6113382

2.2 Discussão

Como pode-se perceber através do gráfico de speedup e da tabela de tempos de execução, o desempenho na GPU utilizando memória compartilhada superou fortemente o desempenho sem a utilização da mesma, constatando a eficácia da utilização deste tipo de memória para prover acessos mais rápidos aos elementos necessários.

Em resumo, a versão final faz com que as threads na GPU tenham valores de duas dimensões fixados, copia a parte a ser acessada da matriz de elementos para um vetor na memória compartilhada e itera sobre este vetor para dentro da terceira dimensão - equivalente ao terceiro laço de repetição no código sequencial. Como não existem muitas estruturas condicionais dentro do kernel, nem dependência entre dados, e as operações são as mesmas para qualquer posição da matriz e dependem de uma grande quantidade de acessos a memória, a paralelização utilizando cuda e com memória compartilhada se mostra muito eficiente.