Universidade Federal de São Carlos

Programa de Pós-Graduação em Ciência da Computação

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

Projeto Final

OpenHPC UFSCar

Alcides Mignoso e Silva 760479

Submission Date : 02/05/2022

1 Atividade

O objetivo desta atividade é paralelizar a simulação da propagação de uma onda acústica em um domínio 2D. O código base utilizado foi o existente em `https://github.com/HPCSys-Lab/HPC-101/blob/main/examples/acoustic-wave/wave_seq.c` e este relatório visa discutir valores de tempo de execução, speedup e performance das execuções.

As execuções foram realizadas utilizando matrizes com o número de linhas e colunas igual a 1000, 2000, 3000, 5000 e 7000, em um computador equipado com uma GPU RTX 2060 SUPER utilizando CUDA na versão 11.6, e uma CPU Ryzen 7 3700X.

A paralelização foi feita utilizando CUDA, em duas implementações diferentes, e adicionalmente OpenMP, para fins de comparação. Foram feitas 5 execuções para cada instância do problema e os valores apresentados no relatório dizem respeito a média desses 5 valores.

Em relação a paralelização com OpenMP, bastou adicionar a diretiva "pragma omp parallel for collapse(2)" nos laços de repetição de processamento para que a divisão entre threads acontecesse. Um ponto importante é que não pode-se paralelizar o loop mais exterior (o de iterações), dado que existe uma dependência entre qualquer iteração X e X+1 para o problema.

Abaixo segue o trecho com a paralelização utilizando OpenMP.

```
// wavefield modeling
for(int n = 0; n < iterations; n++) {
    #pragma omp parallel for collapse(2)
    for(int i = HALF_LENGTH; i < rows - HALF_LENGTH; i++) {
        for(int j = HALF_LENGTH; j < cols - HALF_LENGTH; j++) {
            // index of the current point in the grid
            int current = i * cols + j;

            //neighbors in the horizontal direction
            float value =
                (prev_base[current + 1] - 2.0 *
                prev_base[current] + prev_base[current - 1])
                    / dxSquared;

            //neighbors in the vertical direction
            value +=
                (prev_base[current + cols] - 2.0 *
                prev_base[current] + prev_base[current - cols])
                    / dySquared;

            value *= dtSquared * vel_base[current];

            next_base[current] = 2.0 * prev_base[current]
                - next_base[current] + value;
        }
    }
```

O problema da dependência entre as iterações também ocorreu para a paralelização com CUDA, fazendo com que iterações diferentes não pudessem rodar

simultâneamente. Dessa forma, a primeira solução encontrada foi rodar uma iteração por vez no kernel da GPU, resultando no seguinte código:

```
__global__ void _acoustic_wave(float *prev_base, float *next_base,
                               int iterations, int rows, int cols,
                               int num_threads, int stencil_radius,
                               float dxSquared, float dySquared,
                               float dtSquared, float wave_velocity) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i >= rows - stencil_radius || j <= stencil_radius ||
      i <= stencil_radius || j >= cols - stencil_radius)
    return;

  int current = i * cols + j;

  // neighbors in the horizontal direction +
  // neighbors in the vertical direction
  // * dtSquared * wave_velocity * wave_velocity
  next_base[current] =
      ((((prev_base[current + 1] - 2.0 * prev_base[current] +
          prev_base[current - 1]) /
         dxSquared) +
        ((prev_base[current + cols] - 2.0 * prev_base[current] +
          prev_base[current - cols]) /
         dySquared)) *
       (dtSquared * wave_velocity * wave_velocity)) +
      ((2.0 * prev_base[current]) - next_base[current]);
}

void applyWave(char *argv[]) {
  ... // pulando para a parte que interessa
  for (int i = 0; i < iterations; i++) {
    _acoustic_wave<<<numBlocks, threadsPerBlock>>>(
        prev_base, next_base, iterations, rows, cols,
        thread_limit_per_block_sqrt, stencil_radius,
        d_x_sqr, d_y_sqr, d_t_sqr,
        wave_velocity);
    // swap das matrizes
    tools::swapf(&next_base, &prev_base);
  }
  ...
}
```

Ainda utilizando CUDA, algumas otimizações puderam ser aplicadas visando evitar alguns acessos a memória global. O trecho abaixo é referente a segunda implementação em CUDA, e também a que melhor performou.

```
__global__ void _acoustic_wave(float *prev_base, float *next_base,
                               int iterations, int rows, int cols,
                               int num_threads, int stencil_radius,
                               float dxSquared, float dySquared,
                               float dtSquared,
                               float wave_velocity_pow2) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i >= rows - stencil_radius || j <= stencil_radius ||
      i <= stencil_radius || j >= cols - stencil_radius)
    return;

  int current = i * cols + j;

  int prev_base_current = 2.0 * prev_base[current];

  // neighbors in the horizontal direction +
  // neighbors in the vertical direction
  // * dtSquared * wave_velocity * wave_velocity
  next_base[current] =
      ((((prev_base[current + 1] - prev_base_current
                  + prev_base[current - 1]) /
         dxSquared) +
        ((prev_base[current + cols] - prev_base_current +
          prev_base[current - cols]) /
         dySquared)) *
       (dtSquared * wave_velocity_pow2)) +
      ((prev_base_current)-next_base[current]);
}

void applyWave(char *argv[]) {
  ... // pulando para a parte que interessa
  for (int i = 0; i < iterations; i++) {
    _acoustic_wave<<<numBlocks, threadsPerBlock>>>(
        prev_base, next_base, iterations, rows, cols,
        thread_limit_per_block_sqrt, stencil_radius,
        d_x_sqr, d_y_sqr, d_t_sqr,
        wave_velocity*wave_velocity);
    // swap das matrizes
    tools::swapf(&next_base, &prev_base);
  }
  ...
}
```

Em ambas as execuções com CUDA, utilizou-se o número de threads por bloco sendo $X * X$, *com* $X \in 32, 16, 8, 4$ (X para a dimensão x e X para a dimensão y) e o número de blocos sendo o número de linhas dividido por X (dimensão x) multiplicado pelo número de colunas divido por X (dimensão y), ambos somados a 1 por questões de consistência. Como a imagem era 2D, não foi necessário

utilizar a terceira dimensão (dimensão z).

Para todas as execuções (as de CUDA, sequencial e OpenMP) utilizou-se 50_000 como o número de timesteps, resultando em 7071 iterações até o fim da execução.

2 Resultados

2.1 Tabelas

A tabela abaixo trás os tempos de execução do programa (em segundos) na primeira versão paralelizada utilizando GPU (cuda) para diferentes número de threads por bloco na GPU e diferentes dimensões da imagem de input.

Dimensão	GPU 32x32	GPU 16x16	GPU 8x8	GPU 4x4
1000x1000	2,2575	1,9725	1,9505	3,8843
2000x2000	8,7361	8,2561	8,2604	16,5394
3000x3000	20,9632	19,6959	19,6438	39,2412
5000x5000	63,3662	59,6253	59,5995	119,1445
7000x7000	133,8334	127,1207	127,5289	254,1898
10000x10000	299,8932	286,4739	287,9618	572,5802

A tabela abaixo trás os tempos de execução do programa (em segundos) na segunda versão (otimizada) paralelizada utilizando GPU (cuda) para diferentes número de threads por bloco na GPU e diferentes dimensões da imagem de input.

Dimensão	GPU opt 32x32	GPU opt 16x16	GPU opt 8x8	GPU opt 4x4
1000x1000	1,1973	0,7758	0,7475	1,1202
2000x2000	5,1997	2,6235	2,6225	3,814
3000x3000	9,6349	6,1577	5,5019	9,8469
5000x5000	27,5344	17,0886	15,3273	29,1733
7000x7000	54,0454	33,7878	29,9317	56,8166
10000x10000	131,1551	72,3378	74,8411	137,9531

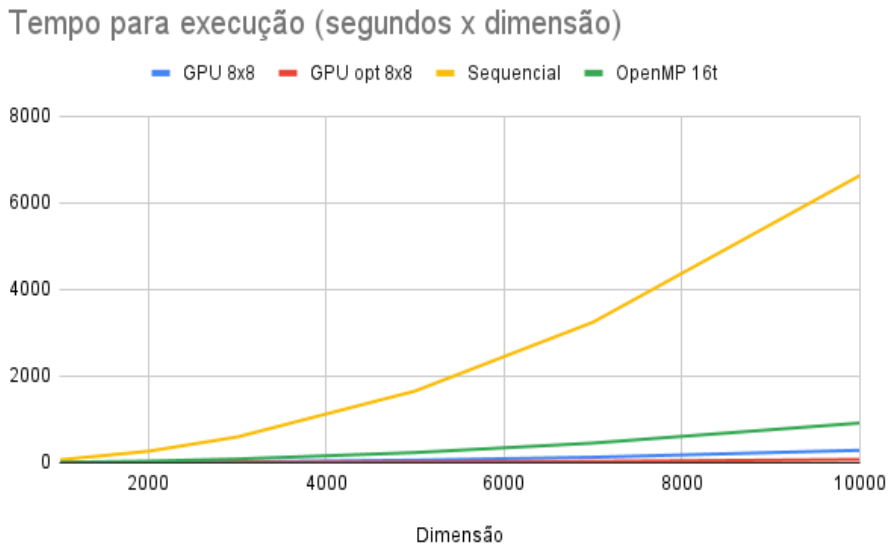
A tabela abaixo trás os tempos de execução do programa (em segundos) na versão sequencial e na paralelizada utilizando OpenMP para diferentes dimensões da imagem de input. O número de threads com OpenMP foi 16, dado que foi o que apresentou o melhor resultado.

Dimensão	Sequencial	OpenMP 16t
1000x1000	67,4985	9,5165
2000x2000	266,0032	37,0011
3000x3000	595,9611	86,1543
5000x5000	1656,9103	235,5845
7000x7000	3246,0915	454,3027
10000x10000	6630,8976	917,0153

A tabela abaixo trás os valores de "speedup" comparando a melhor execução em GPU de ambas as implementações com a execução sequencial do problema.

Dimensão	Speedup	Speedup opt
1000x1000	34,60574212	90,29899666
2000x2000	32,20221781	101,4311535
3000x3000	30,33838158	108,3191443
5000x5000	27,80074162	108,1019031
7000x7000	25,45377165	108,4499544
10000x10000	23,02700428	88,59968119

O gráfico abaixo trás uma comparação entre os tempos de execução (em segundos, na vertical) utilizando GPU (versão otimizada "opt" e a não otimizada), OpenMP e sem paralelização para diferentes número de linhas e colunas da imagem de input (na horizontal).



2.2 Discussão

Como pode-se perceber através do gráfico e das tabelas de tempos de execução, o desempenho de ambas as implementações utilizando GPU (CUDA) foi muito bom quando comparado com a versão sequencial e até mesmo com a versão paralelizada utilizando threads (OpenMP). Os valores utilizando a GPU chegaram a ser uma ordem de grandeza menor que os valores com a versão sequêncial.

Pode-se perceber que a melhor execução com GPU, para ambas as implementações, foi utilizando blocos com o número de threads sendo 8x8, mas que ficou muito próximo da execução com 16x16 threads. Com mais ou menos threads do que esses valores (32x32 e 4x4) o número de threads se torna ou muito alto ou muito baixo, causando overhead de comunicação e distribuição de tarefas (no caso de muito alto) ou falta de aproveitamento da GPU (no caso de muito baixo). O mesmo comportamento aparece utilizando OpenMP considerando threads da CPU.

Ainda em relação a utilização da GPU/CUDA, o desempenho poderia ter sido ainda melhor caso todas as iterações pudessem rodar em uma única chamada do kernel cuda, mas como isso não é possível devido a dependência entre os dados, ocorre uma pequena perda de desempenho. Em relação a comparação entre a versão não otimizada e a versão otimizada, ficou claro no gráfico e tabelas que a versão otimizada, por reduzir o número de acessos a memória global trazendo alguns valores que são acessados mais de uma vez para a memória local da thread, melhorou bastante o tempo de execução das tarefas – evidenciando claramente a importância de utilizar a memória corretamente nas implementações.

Uma abordagem ainda melhor, visando um ambiente com mais de um nó disponível na rede, seria dividir a matriz da imagem inicial em pequenos "chunks" e espalhar eles entre a rede (possível utilização de OpenMPI) para que cada nó processe a sua parte (seja utilizando CPU ou GPU) e retorne-a para ser unida pelo nó principal.