Simulação e Análise de Modelos de Difusão de Contaminantes em Água usando CUDA

Gabriel Amaral[1], **Lucas Kuroki**[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP)
São José dos Campos – SP – Brazil

kenzo.kuroki@unifesp.br

Resumo. *Este trabalho visa modelar a difusão de concentração em uma grade bidimensional. Foram desenvolvidas implementações de threads paralelas, utilizando o CUDA e o notebook Collab. Os resultados mostram melhorias consideráveis em relação ao sequencial e ao openMp.*

1. Introdução

Para a modelagem de uma difusão de contaminantes em água, foram desenvolvidos códigos no formato sequencial e paralelizado através do OpenMP. Visando utilizar GPUs para calcular a equação de difusão, foi implementado o modelo da difusão em CUDA no ambiente Collab.

1.1. Metodologia

Foi necessário prepara-lo instalando ferramentas do CUDA para sua utilização, os comandos podem ser vistos abaixo.

```
// Verificar a presen a de uma GPU
!nvidia-smi

// Instalar o compilador CUDA
!apt-get install -q nvidia-cuda-toolkit g++ freeglut3-dev libx11
   -dev libxmu-dev libxi-dev libglu1-mesa-dev
```

Listing 1. Comandos para preparar o ambiente

No código que implementa o modelo com CUDA, foram definidos os mesmos parametros de grade das versões anteriores. Em seguida temos a função da difusão que usa uma grade de threads, com cada thread atualizando uma célula da matriz(linha 12 e 13). Uma função auxiliar atomicAddDouble, que realiza somas atômicas, e um kernel computeDifference, que faz os cálculos e armazena em na global dif, nele, cada thread é responsável por calcular a diferença absoluta entre as matrizes(linha 47) e o vetor é usado para acumular as diferenças localmente dentro de cada bloco, realizando um reduction atraves da thread 0, que acumula os valores da global dif.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 2000      // Tamanho da grade
#define T 1000      // numero de iteracoes no tempo
```

```
#define D 0.1       // Coeficiente de difusao
#define DELTA_T 0.01
#define DELTA_X 1.0

__global__ void diff_eq(double *C, double *C_new) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int index = i * N + j;

    if (i > 0 && i < N-1 && j > 0 && j < N-1) {
        C_new[index] = C[index] + D * DELTA_T * (
            (C[index + N] + C[index - N] + C[index + 1] + C[
               index - 1] - 4 * C[index]) / (DELTA_X * DELTA_X)
        );
    }
}

static __device__ double atomicAddDouble(double* address, double
    val) {
    unsigned long long int* address_as_ull = (unsigned long long
        int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
           __double_as_longlong(val + __longlong_as_double(
           assumed)));
    } while (assumed != old);

    return __longlong_as_double(old);
}


__global__ void compute_difference(double *C, double *C_new,
   double *dif) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int index = i * N + j;

    __shared__ double dif_per_block[256];
    int t_idx = threadIdx.x + threadIdx.y * blockDim.x;

    if (i < N && j < N) {
        dif_per_block[t_idx] = fabs(C_new[index] - C[index]);
    } else {
        dif_per_block[t_idx] = 0.0;
```

```
50      }
51
52      __syncthreads();
53
54      for (int s = blockDim.x * blockDim.y / 2; s > 0; s >>= 1) {
55          if (t_idx < s) {
56              dif_per_block[t_idx] += dif_per_block[t_idx + s];
57          }
58          __syncthreads();
59      }
60
61      if (t_idx == 0) {
62          atomicAddDouble(dif, dif_per_block[0]);
63      }
64  }
```

Listing 2. código usando CUDA, parte 1

Na função principal, iniciamos as matrizes, a memória e alocamos memória na GPU, por meio do cudaMalloc. Após reservar espaço fizemos uma cópia do host para a GPU nas linhas 20 e 21, em seguida configuramos a quantidade de grids e blocos, no caso, 16*16= 256 threads e o grid se ajusta conforme a variável global N.

No bloco de iteracoes no tempo, atualizamos a nova matriz de concentração, e a cada 100 iterações zeramos o ddiff, chamamos um novo calculo por meio da função computeDifference, pegamos a diferença acumulada para o host e exibimos a diferença média entre iterações. E no final, colocamos a matriz atualizada no host e imprimimos no terminal a concentração no centro.

```
1   int main() {
2       // Aloca  o de memoria no host
3       double *C = (double *)malloc(N * N * sizeof(double));
4       double *C_new = (double *)malloc(N * N * sizeof(double));
5       // Inicializa  o da memoria
6       for (int i = 0; i < N; i++) {
7           for (int j = 0; j < N; j++) {
8               C[i * N + j] = 0.0;
9               C_new[i * N + j] = 0.0;
10          }
11      }
12      C[(N/2) * N + (N/2)] = 1.0; // Inicializar concentra  o
            alta no centro
13      // Aloca  o de memoria na GPU
14      double *d_C, *d_C_new, *d_dif;
15      cudaMalloc((void **)&d_C, N * N * sizeof(double));
16      cudaMalloc((void **)&d_C_new, N * N * sizeof(double));
17      cudaMalloc((void **)&d_dif, sizeof(double));
18
19      // Copiar dados do host para a GPU
20      cudaMemcpy(d_C, C, N * N * sizeof(double),
            cudaMemcpyHostToDevice);
```

```
21      cudaMemcpy(d_C_new, C_new, N * N * sizeof(double),
            cudaMemcpyHostToDevice);
22
23      // Dimens es do bloco e da grade
24      dim3 threadsPerBlock(16, 16);
25      dim3 blocksPerGrid((N + threadsPerBlock.x - 1) /
            threadsPerBlock.x, (N + threadsPerBlock.y - 1) /
            threadsPerBlock.y);
26
27      // Iteracoes no tempo
28      for (int t = 0; t < T; t++) {
29          diff_eq<<<blocksPerGrid, threadsPerBlock>>>(d_C, d_C_new
                );
30          if ((t % 100) == 0) {
31              cudaMemset(d_dif, 0, sizeof(double));
32              compute_difference<<<blocksPerGrid, threadsPerBlock
                    >>>(d_C, d_C_new, d_dif);
33              double difmedio;
34              cudaMemcpy(&difmedio, d_dif, sizeof(double),
                    cudaMemcpyDeviceToHost);
35              difmedio /= ((N-2)*(N-2));
36              printf("Itera  o %d - Diferen a  m dia=%g\n", t,
                    difmedio);
37          }
38
39          // Trocar os ponteiros
40          double *temp = d_C;
41          d_C = d_C_new;
42          d_C_new = temp;
43      }
44
45      // Copiar resultado final da GPU para o host
46      cudaMemcpy(C, d_C, N * N * sizeof(double),
            cudaMemcpyDeviceToHost);
47
48      // Exibir resultado final no centro da grade
49      printf("Concentra  o final no centro: %f\n", C[(N/2) * N +
            (N/2)]);
50
51      // Liberar memoria
52      cudaFree(d_C);
53      cudaFree(d_C_new);
54      cudaFree(d_dif);
55      free(C);
56      free(C_new);
57      return 0;
58  }
```

Listing 3. código usando CUDA, parte 2

2. Resultados e Discussões

Para esta implementação em CUDA, utilizando a ferramenta time nos comandos de execução e obtivemos os seguintes valores de tempo:

```
nvcc -o diffusion diffusion.cu -lm
time ./diffusion
!nvcc -o diffusion diffusion.cu -lm
!time ./diffusion
Itera  o 0 - Diferen a m dia=2.00401e-09
Itera  o 100 - Diferen a m dia=1.23248e-09
Itera  o 200 - Diferen a m dia=7.81794e-10
Itera  o 300 - Diferen a m dia=5.11528e-10
Itera  o 400 - Diferen a m dia=4.21632e-10
Itera  o 500 - Diferen a m dia=3.62223e-10
Itera  o 600 - Diferen a m dia=3.05976e-10
Itera  o 700 - Diferen a m dia=2.57135e-10
Itera  o 800 - Diferen a m dia=2.21174e-10
Itera  o 900 - Diferen a m dia=2.00244e-10
Concentra  o final no centro: 0.095045

real    0m1.536s
user    0m1.073s
sys 0m0.297s
```

Listing 4. Comandos e resultados da execução do código

Com o tempo real foram calculados os speedups das implementações antigas e dessa com CUDA, e comparadas na figura abaixo.

Numero de threads	Tempo (s)	Speedup
Serial	67,13	1
OpenMp(8 threads)	10,611	6,326
CUDA	1,536	43,704

Figure 1. Medidas de desempenho usando implementação sequencial, openMp e CUDA

Podemos observar que utilizando o OpenMp com 8 threads, obtivemos um speedup próximo de 6, já utillizando o CUDA, o ganho foi muito maior, com speedup de 43, isso revela a capacidade que GPUs possuem em realizar tarefas atômicas, como de redução e calculo de matrizes, com muita velocidade, sendo assim, uma ferramenta muito poderosa na área de pesquisa que exigem calculos simples mas em grande escala.