Computo Paralelo

Tarea 4

Rubén Pérez Palacios Lic. Computación Matemática
Profesor: Dr. Francisco Javier Hernández López

9 de marzo de 2022

Reportes

Se explica la solución de los ejercicios, así como la implementación en secuencial, paralelo en cpu y en gpu.

Para todas las soluciones se uso la siguiente librería para el manejo de imagenes:

Para la paralelización en cpu se uso la siguiente librería e instrucción:

```
        #include <omp.h>
        omp_set_num_threads(8);//cantidad de hilos a paralelizar
```

Para la paralelización en gpu se uso la siguiente librería e instrucción:

```
        #include <cuda_runtime.h>
        #include <device_launch_parameters.h>
        cudaSetDevice(0);
        cudaMemcpy(/*...*/);//copiar memoria entre la gpu y algo mas
        cudaMalloc(/*...*/);//alocar memoria en la gpu
        cudaDeviceSynchronize();//esperar a que la gpu termine de ejecutar
↪   kernels.
        cudaFree(/*...*/);//desalojar memoria en la gpu
```

Ejercicio 1

Descripción

Dado un vector de números reales V de tamaño N, programar lo siguiente

1. $S_1[i] = V[i] + V[i+1]$ para $i = 0, \cdots, N-2$, con S_1 otro vector de tamaño $N-1$.
2. $S_2[i] = \frac{V[i+1]+V[i-1]}{2}$ para $i = 1, \cdots, N-2$, con S_2 otro vector de tamaño $N-2$.

Solución

Se recorrio cada posición de S_1, S_2 para calcularlos con ciclos for

```
    //S_1
    for (int i = 0; i < N - 1; i++)
    v_1[i] = v[i] + v[i + 1];
    //S_2
    for (int i = 1; i < N - 1; i++)
    v_1[i] = (v[i + 1] + v[i - 1])/2;
```

Paralelización

1. CPU:

   Se uso *omp parallel for* con la directiva *default(shared)* ya que las variables de iteración fueron declaradas localmente y todas puedan acceder a los vectores origen y al resultado.

   ```
       //S_1
       #pragma omp parallel for default(shared)
   for (int i = 0; i < N - 1; i++)
       v_1[i] = v[i] + v[i + 1];
       //S_2
       #pragma omp parallel for default(shared)
   for (int i = 1; i < N - 1; i++)
       v_1[i] = (v[i + 1] + v[i - 1])/2;
   ```

2. GPU:

   Para ello se hizo la alocación y copias de memoria para los vectores:

   ```
       cudaMalloc((void**)&v_device, N * sizeof(long double));
       cudaMalloc((void**)&v_1_device, (N-1) * sizeof(long double));
       cudaMemcpy(v_device, v, N * sizeof(long double),
   ↪  cudaMemcpyHostToDevice);
   ```

Después se uso una función kernel para resolver el problema, de tipo *__global__*, con número de bloques y tamaño de malla

```
        int threads = 512, grid = divUp(N, threads);
```

la cual es

```
        //S_1
        __global__ void Solve(long double *v, long double *v_1, int N) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                if (idx < N) {
                        v_1[idx] = v[idx] + v[idx + 1];
                }
        }
        //S_2
        __global__ void Solve(long double *v, long double *v_1, int N) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                if (idx < N) {
                        v_1[idx] = (v[idx + 1] + v[idx - 1])/2;
                }
        }
```

Ejercicio 2

Descripción

Dadas dos matrices A y B de tamaño $N \times M$ con valores enteros positivos, programar lo siguiente:

1. $C_1[i][j] = A[i][j] + B[N-1-i][M-1-j]$ para $i = 0, \cdots, N-1$ y $j = 0, \cdots, M-1$ con C_1 otra matriz de tamaño $N \times M$.
2. $C_2[i][j] = (\alpha) * A[i][j] + (1-\alpha) * B[N-1-i][M-1-j]$ para $i = 0, \cdots, N-1$ y $j = 0, \cdots, M-1$, con α un número real constante entre $[0, 1]$ y C_1 otra matriz de tamaño $N \times M$.

Solución

Se recorrio cada posición de C_1, C_2 para calcularlos con 2 ciclos for anidados

```
        //C_1
        for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            C[i*M+j] = A[i*M+j] + B[(N - 1 - i)*M+(M - 1 - j)];
        //C_2
        for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            C[i*M+j] = (alpha)*A[i*M+j] + (1 - alpha) * B[(N - 1 - i)*M+(M - 1 -
↪  j)];
```

Paralelización

1. CPU:

   Para la paralelización primero se uso $omp\ parallel\ for\ collapse(2)$, con las directivas $private(index, index1)$ porque cada iteración tiene su propio indice y $default(shared)$ ya que las variables de iteración fueron declaradas localmente y todas puedan accerder a las matrices por multiplicar y al resultado, así los primeros dos for anidados se paralelizaran.

   ```
           //C_1
           #pragma omp parallel for collapse(2) default(shared)
       for (int i = 0; i < N; i++)
           for (int j = 0; j < M; j++)
               C[i*M+j] = A[i*M+j] + B[(N - 1 - i)*M+(M - 1 - j)];
           //C_2
           #pragma omp parallel for collapse(2) default(shared)
       for (int i = 0; i < N; i++)
           for (int j = 0; j < M; j++)
               C[i*M+j] = (alpha)*A[i*M+j] + (1 - alpha) * B[(N - 1 - i)*M+(M - 1
   ↪  - j)];
   ```

2. GPU:

   Para ello se hizo la alocación y copias de memoria para las matrices:

```
        cudaMalloc((void**)&A_device, N * M * sizeof(double));
        cudaMalloc((void**)&B_device, N * M * sizeof(double));
        cudaMalloc((void**)&C_device, N * M * sizeof(double));
        cudaMemcpy(A_device, A, N * M * sizeof(double),
↪   cudaMemcpyHostToDevice);
        cudaMemcpy(B_device, B, N * M * sizeof(double),
↪   cudaMemcpyHostToDevice);
```

   Después se uso una función kernel para resolver el problema, de tipo *__global__*, con número de bloques y tamaño de malla

```
        dim3 threads(16, 16, 1), grid(divUp(M, 16), divUp(N, 16), 1);
```

   la cual es

```
        //C_1
        __global__ void Solve(double *A, double *B, double *C, int N, int M) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                int idy = blockIdx.y * blockDim.y + threadIdx.y;
                int index = (idy)*M + (idx);
                int index1 = (N - 1 - idy) * M + (M - 1 - idx);
                if (index < N*M) {
                        C[index] = A[index] + B[index1];
                }
        }
        //C_2
        __global__ void Solve(double *A, double *B, double *C, double alpha,
↪   int N, int M) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                int idy = blockIdx.y * blockDim.y + threadIdx.y;
                int index = (idy)*M + (idx);
                int index1 = (N - 1 - idy) * M + (M - 1 - idx);
                if (index < N*M) {
                        C[index] = (alpha)*A[index] + (1-alpha)*B[index1];
                }
        }
```

Ejercicio 3

Descripción

Dada una matriz A de tamaño $N \times K$ y una matriz B de tamaño $K \times M$ con valores en punto flotante de 64 bits (double), programar la multiplicación de las matrices A y B:

1. Usando OpenMP
2. Usando CUDA con memoria global (GM)
3. Usando CUDA con memoria compartida (SM)

Solución

Se recorrio cada posición de C para calcularla con 2 ciclos for anidados

```
    for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        C[i * M + j] = 0.0;
        for (int k = 0; k < K; k++)
            C[i * M + j] += A[i * K + k] * B[j * K + k];
    }
}
```

Paralelización

1. CPU:

   Para la paralelización primero se uso $omp\,parallel\,for\,collapse(2)$, con las directivas $private(index, index1)$ porque cada iteración tiene su propio indice y $default(shared)$ ya que las variables de iteración fueron declaradas localmente y todas puedan accerder a las matrices por multiplicar y al resultado, así los primeros dos for anidados se paralelizaran.

```
    #pragma omp parallel for collapse(2) default(shared)
 for (int i = 0; i < N; i++)
 {
    for (int j = 0; j < M; j++)
    {
        C[i * M + j] = 0.0;
        for (int k = 0; k < K; k++)
            C[i * M + j] += A[i * K + k] * B[j * K + k];
    }
 }
```

2. GPU:

   Para ello se hizo la alocación y copias de memoria para las matrices:

```
        cudaMalloc((void**)&A_device, N * K * sizeof(double));
        cudaMalloc((void**)&B_device, M * K * sizeof(double));
        cudaMalloc((void**)&C_device, N * M * sizeof(double));
        cudaMemcpy(A_device, A, N * K * sizeof(double),
↪   cudaMemcpyHostToDevice);
        cudaMemcpy(B_device, B, M * K * sizeof(double),
↪   cudaMemcpyHostToDevice);
```

   Después se uso una función kernel para resolver el problema, de tipo *__global__*, con número de bloques y tamaño de malla

```
        dim3 threads(16, 16, 1), grid(divUp(M, 16), divUp(N, 16), 1);
```

   la cual es

```
        __global__ void mat_prod(double *A, double *B, double *C, int N, int
↪   K, int M)
        {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                int idy = blockIdx.y * blockDim.y + threadIdx.y;
                int index = (idy)*M + (idx);
                C[index] = 0.0;
                for (int k = 0; k < K; k++)
                        C[index] += A[idy * K + k] * B[idx * K + k];
        }
```