Ejericicio: Manejo de matrices y medición de tiempos de ejecución.

Subtemas: Reservar espacios de memoria bidimensionales en la GPU, copiar información en forma de matrices hacia y desde la GPU, medir tiempos de ejecución

Manejo de memoria bidimensional

- cudaMallocPitch: reserva un espacio de memoria de dos dimensiones
- cudaMemcpy2D: copia una matriz de la CPU a la GPU ó de la GPU a la CPU

  Cuando se llama a una matriz dentro de una función de la GPU los que se tiene es un apuntador a un arreglo que contiene arreglos, por lo tanto para poder manejar los datos de forma individual se necesita crear otro apuntador al renglón que se quiera manejar:

  float *apuntador_renglon = (float *) ((char *)apuntador_matriz + número_renglon * tamaño_espacios

Medir tiempos de desempeño

- cudaEvent_t: es un tipo dato de CUDA que sirve como marca temporal para realizar las mediciones de desempeño
- cudaEventCreate(&evento): Prepara la marca para usarse
- cudaEventRecord(evento): Fija la marca temporal
- cudaEventSynchronize(evento): Detiene la CPU hasta que el evento termine
- cudaEventElapsedTime(&tiempo, evento1, evento2): mide el tiempo entre los dos eventos y guarda el tiempo en ms en la variable tiempo que es de tipo float
- cudaEventDestroy(evento): libera la marca temporal del evento espesificado.

Código ejemplo

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

using namespace std;

__global__ void MultiplicarMatricesSecuencial(float *matriz1_GPU, float *matriz2_GPU, float *matriz3_GPU, int TDM, size_t pitch){    //Este modulo usa 1 solo thread
    for(int i =0; i < TDM; i++){
        for(int j = 0; j < TDM; j++){
            float *elementos_matriz1 = (float *) ((char*)matriz1_GPU + j * pitch);                  //Obtenemos el j-esimo renglon de la matriz
            float *elementos_matriz3 = (float *) ((char*)matriz3_GPU + j * pitch);
            elementos_matriz3[i] = 0;
            for(int x = 0; x < TDM; x++){
                float *elementos_matriz2 = (float *) ((char*)matriz2_GPU + (x) * pitch);
                elementos_matriz3[i] += elementos_matriz1[x] * elementos_matriz2[i];                //Sumamos en la i-esima laumna del renglon de la matriz que trabajamos
                free(elementos_matriz2);

            }
        }
    }
}
```

Código ejemplo

```
__global__ void MultiplicarMatricesOn(float *matriz1_GPU, float *matriz2_GPU, float *matriz3_GPU, int TDM,size_t pitch){  //Este modulo usa n^2 threads
    const unsigned int idx = threadIdx.x + (blockDim.x *  blockIdx.x);

    const unsigned int j = idx / TDM;
    const unsigned int i = idx % TDM;

    float *elementos_matriz1 = (float *) ((char*)matriz1_GPU + j * pitch);
    float *elementos_matriz3 = (float *) ((char*)matriz3_GPU + j * pitch);
    elementos_matriz3[i] = 0;
    for(int x = 0; x < TDM; x ++){
        float *elementos_matriz2 = (float *) ((char*)matriz2_GPU + x * pitch);
        elementos_matriz3[i] += elementos_matriz1[x] * elementos_matriz2[i];
        free(elementos_matriz2);
    }
}
```

Códgio ejemplo

```
int main(){
    int TDM = 50;

    int TDM2 = 1;
    unsigned int NDH = pow(TDM2,2);                                    // Número de hilos
    unsigned int numero_bloques =  ceil( (float) NDH / (float) TDM2);          // Tamño de la matriz (cuadrada)TDM );
    unsigned int hilos_bloque = ceil( (float) NDH / (float) numero_bloques);    // Tamño de la matriz (cuadrada)ero_bloques );

    float matriz1_host[TDM][TDM];                                      //Creamos arrglos bidimensionales en la CPU
    float matriz2_host[TDM][TDM];

    for(int i = 0; i < TDM; i++){
        for(int j = 0; j < TDM; j++){
            matriz1_host[i][j] = (int)(i + j);                         //Llenamos la matriz que creamos antes con valores
            matriz2_host[i][j] = (int)(i + j);
        }
    }
```

Código ejemplo

```
/* ********** Muestra las matrices que se van a multiplicar
cout << "Matrices a multiplicar \nMatriz 1" << endl;
for(int i = 0; i < TDM; i++){
    for(int j = 0; j < TDM; j++){
        cout << *(*(matriz1_host + i) + j) << "\t";
    }
    cout << "\n";
}
cout << "\nMatriz 2" << endl;
for(int i = 0; i < TDM; i++){
    for(int j = 0; j < TDM; j++){
        cout << *(*(matriz2_host + i) + j) << '\t';
    }
    cout << "\n";
}
*/
```

Código ejemplo

```
size_t pitch;                                                   //  Esta variable contiene el tamaño de los espacios
                                                                //interiores de la matriz
   float *matriz1_GPU; cudaMallocPitch(&matriz1_GPU, &pitch, TDM * sizeof(float), TDM );        //  Hacemos las reservaciones en memoria de las
   float *matriz2_GPU; cudaMallocPitch(&matriz2_GPU, &pitch, TDM * sizeof(float), TDM );        //matrices que vamos a necesitar
   float *matriz3_GPU; cudaMallocPitch(&matriz3_GPU, &pitch, TDM * sizeof(float), TDM );

   cudaMemcpy2D(matriz1_GPU, pitch, matriz1_host, TDM * sizeof(float), TDM * sizeof(float), TDM, cudaMemcpyHostToDevice); //Copiamos los valores de la matriz en
   cudaMemcpy2D(matriz2_GPU, pitch, matriz2_host, TDM * sizeof(float), TDM * sizeof(float), TDM, cudaMemcpyHostToDevice); //en la CPU a la GPU

   cudaEvent_t inicio, alto;                                    //Variables para el control de los eventos
   float tiempo_computo;                                        // Variable para almacenar el tiempo transcurrido (ms)

   for(TDM2 = 1; TDM2 <= TDM; TDM2++){

      NDH = pow(TDM2,2);                                        // Número de hilos que se lanzarán
      numero_bloques =  ceil( (float) NDH / (float) TDM2);
      hilos_bloque = ceil( (float) NDH / (float) numero_bloques);
      tiempo_computo = 0;                                       //Esta variable contendrá el tiempo en ms que demora el evento
      cudaEventCreate(&inicio); cudaEventCreate(&alto);         //Creamos los eventos
      cudaEventRecord(inicio);                                  //Creamos una marca temporal, una especia de bandera
      MultiplicarMatricesOn<<<numero_bloques, hilos_bloque>>>(matriz1_GPU, matriz2_GPU, matriz3_GPU, TDM2,  pitch);
      cudaEventRecord(alto);                                    // Creamos una marca temporal, otra bandera
      cudaEventSynchronize(alto);                               // Bloquea la CPU para evitar que se continué con el programa hasta que se completen los eventos
      cudaEventElapsedTime(&tiempo_computo, inicio, alto);      //Calcula el tiempo entre los eventos
      cudaEventDestroy(inicio); cudaEventDestroy(alto);         // Se liberan los espacios  de los eventos para poder medir de nuevo más tarde
}
```

Código ejemplo

```
        cout << "Tiempo de computo en n^2 threads para una matriz de "<< TDM2 << ": "<<tiempo_computo << "ms"<<endl;

        cudaEventCreate(&inicio); cudaEventCreate(&alto);
        cudaEventRecord(inicio);
        MultiplicarMatricesSecuencial<<<1, 1>>>(matriz1_GPU, matriz2_GPU, matriz3_GPU, TDM2, pitch);
        cudaEventRecord(alto);
        cudaEventSynchronize(alto);
        cudaEventElapsedTime(&tiempo_computo, inicio, alto);
        cudaEventDestroy(inicio); cudaEventDestroy(alto);

        cout << "Tiempo de computo en secuencia para una matriz de "<< TDM2 << ": "<< tiempo_computo << "ms\n" << endl;
    }
```

Código ejemplo

```
/* ******************** Muestra la matriz de salda de CUDA
 float matriz_salida[TDM][TDM];
 cudaMemcpy2D(matriz_salida, TDM * sizeof(float), matriz3_GPU, pitch, TDM * sizeof(float), TDM, cudaMemcpyDeviceToHost);
  //Copiamos los datos de la matriz de la GPU a la CPU

 cout.precision(3);

 cout << "\nMatriz Multiplicada" << endl;
 for(int i = 0; i < TDM; i++){
     for(int j = 0; j < TDM; j++){
         cout << matriz_salida[i][j] << "\t";
     }
     cout << "\n";
 }
 //free(matriz1_host); free(matriz2_host);*/
 cudaFree(matriz1_GPU); cudaFree(matriz2_GPU), cudaFree(matriz3_GPU);

 return 0;
```