Práctica evaluable Computadores Avanzados

Curso 2022/23 – Computadores Avanzados

Universidad de Castilla-La Mancha

Escuela Superior de Informática

Alberto Vázquez Martínez

Alberto.Vazquez1@alu.uclm.es

Contenido

1. Ejercicio 1

En mi caso, para realizar las diferentes mediciones que se piden en el ejercicio, he utilizado eventos de CUDA.

	Tiempo		
Tiempo total de la sesión	0.352576 ms		
Kernel	0.013760 ms		
cudaMallocDA	0.103968 ms		
cudaMallocDB	0.004576 ms		
cudaMallocDC	0.003968 ms		
	Latencia	Tiempo dedicado	Ancho de banda (efectivo)
cudaMemcpy DA ← HA	0.017664 ms	0.017664 ms	0.226449 GB/s
cudaMemcpy DB ← HB	0.005920 ms	0.005920 ms	0.675676 GB/s
cudaMemcpy HC ← DC	0.010688 ms	0.010688 ms	0.374252 GB/s

2. Ejercicio 2

Para comprobar si hay errores en las llamadas de *cudaMalloc()*, cudaM*emcpy()* y la llamada al *kernel*, utilizamos funciones de la API de CUDA. Por cada una de las llamadas a la función *cudaMalloc()* defino un código que comprueba si se ha producido un error asíncrono, ya que la llamada a *cudaMalloc()* y es asíncrona. Para comprobar errores que son asíncronos, hay que utilizar la función c*udaDeviceSynchronize()* para parar la ejecución en ese punto del código.

```
cudaMalloc((void**)&DA, size);

errAsync = cudaDeviceSynchronize();
if (errAsync != cudaSuccess)
  printf("Async malloc error: %s\n", cudaGetErrorString(errAsync));
```

Igualmente para la función *memcpy()*, pero en este caso la llamada es síncrona.

```
cudaMemcpy(DA, HA, size, cudaMemcpyHostToDevice);

errSync = cudaGetLastError();
if (errSync != cudaSuccess)
  printf("Sync memcpy error: %s\n", cudaGetErrorString(errSync));
```

En el caso de la llamada al *kernel*, compruebo si se producen tanto errores síncronos como asíncronos.

```
errSync = cudaGetLastError();
errAsync = cudaDeviceSynchronize();

if (errSync != cudaSuccess)
  printf("Sync kernel error: %s\n", cudaGetErrorString(errSync));
if (errAsync != cudaSuccess)
  printf("Async kernel error: %s\n", cudaGetErrorString(errAsync));
```

En cada una de las comprobaciones utilizo la función *cudaGetErrorString()* para obtener información acerca del error.

Si cambio el número N a 600, en mi caso, no produce ningún error. Para que me produzca un error, el número N debe ser superior a 1024, ya que, mi número máximo de hilos por bloque es de 1024.

```
→  S03 ./query_properties
Device Number: 0
Device name: NVIDIA GeForce GTX 950M
Memory Clock Rate (KHz): 2505000
Memory Bus Width (bits): 128
Peak Memory Bandwidth (GB/s): 80.160000
Max thread per block: 1024
```

Por tanto, si por ejemplo, pongo un N igual a 1200, me produce un error síncrono en la función *memcpy()* que copia los datos del *device* al *host*.

```
→  Practica ./suma-vectores1b
Sync memcpy error: invalid configuration argument
error en componente 1
```

Después se imprime que existe un error en el resultado y de la suma y finaliza la ejecución.

3. Ejercicio 3

Para especificar que queremos ejecutar el *kernel* con un bloque y una hebra, lo indicamos de la siguiente manera:

```
VecAdd <<<1, 1>>>(DA, DB, DC);  // N hilos ejecutan el kernel en paralelo
```

El primer argumento (1), especifica el número de *thread blocks* y el segundo argumento (1), especifica el número de hilos en cada *thread block.* Pero, si compilo y ejecuto el programa, salta un error en los resultados obtenidos.

```
→  Practica ./suma-vectores2
error en componente 1
```

El problema es que esta función ejecuta una única operación, por tanto, la primera operación es correcta, pero las siguientes operaciones fallan.

```
→  Practica ./suma-vectores2
0 + 0 = 0   correcto
-1 + 3 = 0   incorrecto
error en componente 1
```

Entonces esta llamada funciona como una función secuencial, ya que solo se realiza la primera llamada. Por tanto, hay que ejecutar el *kernel* en un bucle y pasarle el índice correspondiente en cada iteración.

```
for (i=0; i<N; i++) {
  VecAdd <<<1, 1>>>(DA, DB, DC, i);
}
```

```
__global__ void VecAdd(int* DA, int* DB, int* DC, int i)
{
  DC[i] = DA[i] + DB[i];
}
```

4. Ejercicio 4

Para especificar que queremos ejecutar N bloques con una hebra por bloque lo hacemos de la siguiente manera:

```
VecAdd <<<N, 1>>>(DA, DB, DC);  // N hilos ejecutan el kernel en paralelo
```

Sin embargo, nos da error el *kernel* ya que estamos ejecutando N bloques, por lo que hay que calcular el indice del array para cada bloque. En este caso tenemos que modificar el *kernel* de la siguiente manera. Además de eliminar el bucle del ejercicio anterior.

```
__global__ void VecAdd(int* DA, int* DB, int* DC)
{
  int i = blockIdx.x*blockDim.x;
  DC[i] = DA[i] + DB[i];
}
```

Utilizamos un índice global para acceder a los elementos de los vectores utilizando las variables *blockDim*, *blockIdx*, No es necesario añadir el *threadIdx.x* ya que solo estamos ejecutando un hilo por bloque, por tanto, siempre será el primer hilo (0). Tampoco sería necesario añadir el *blockDim.x* ya que tenemos una única dimensión.

5. Ejercicio 5

Los resultados que he obtenido midiendo los tiempos de ejecución del *kernel* de los tres programas son los siguientes:

	Suma-vectores1b.cu	Suma-vectores2.cu	Suma-vectores3.cu
Tiempo empleado (ms)	0.017536	1.761280	0.017408

Como se puede comprobar el *kernel* de suma-vectores2 es el que más tarda ya que es una ejecución completamente secuencial, no hay paralelismo, por tanto, es lógico que sea el que más tiempo emplea. Los otros dos *kernels* son similares ya que ejecutar N bloques con un único hilo es igual a ejecutar único bloque de N hilos

6. Ejercicio 6

No podemos usar el programa suma-vectores3 ya que tenemos un grid unidimensional, y el número máximo de bloques que podemos usar es 65535, que es mayor a 100000, por tanto, es necesario modificarlo.

Voy a ejecutar N/1024 bloques para así reducir el número de bloques y aumentar el número de hilos por bloque para así poder ejecutar esta operación.

```
blockSize = 1024;
gridSize = (int)ceil((float)N/blockSize);
cudaEventRecord(start);

VecAdd <<<gridSize, blockSize>>>(DA, DB, DC);
```

Además, ahora hay que añadir en el código del *kernel*, en el cálculo del índice, el índice del hilo del bloque y además comprobar que no es mayor al tamaño del vector.

```
__global__ void VecAdd(int* DA, int* DB, int* DC)
{
  int i = blockIdx.x*blockDim.x+threadIdx.x;

  if (i < N) {
    DC[i] = DA[i] + DB[i];
  }
}
```

7. Ejercicio 7

Para poder realizar la suma de dos vectores con tamaño mayor a 65535*512 componentes he modificado el código del *kernel* para que se procesen más de un elemento dentro del *kernel* usando un bucle *for*, para que así cada hilo ejecute una parte del vector.

```
__global__ void VecAdd(int* DA, int* DB, int* DC)
{
  int id = ((blockIdx.x * blockDim.x) + threadIdx.x) * tb;

  for (int i = 0; i < tb; i++) {
    if (id < N) {
      DC[id + i] = DA[id + i] + DB[id + i];
      }
  }
}
```

Además, defino los vectores del host como punteros para que así no salte un error a la hora de reservar espacio de memoria.

```
int main()
{ cudaFree(0);
  int *DA, *DB, *DC;
  int *HA = new int[N], *HB = new int[N], *HC = new int[N];
  int i; int size = N*sizeof(int);
```

Por último, defino el grid y los hilos por bloque que ejecutarán el *kernel*. Para ello defino dos estructuras de tipo *dim3*, una para el número de hilos que se ejecutarán por bloque, el cual lo obtengo usando la función *cudaGetDeviceProperties()* para obtener el máximo número de hilos que se pueden ejecutar por bloque en el dispositivo y calculo el tamaño del *grid* teniendo en cuenta el número máximo de hilos, el tamaño del bloque (512) y el tamaño de los vectores (N=65535*512).

```
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, 0);
int maxThreads = devProp.maxThreadsPerBlock;
int numBlocks = (N / (maxThreads * tb)) + 1;

dim3 dimGrid(numBlocks);
dim3 dimBlock(maxThreads);

VecAdd <<<dimGrid, dimBlock>>>(DA, DB, DC);
```

8. Ejercicio 8

Para reservar memoria *pinned* en los tres vectores *host*, primero defino los vectores como punteros y después usando la función *cudaMallocHost()* se reserva la memoria.

```
int *HA, *HB, *HC, *DA, *DB, *DC;
int i, dg; int size = N*sizeof(int);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaError_t testerr;
testerr = cudaMallocHost((void**)&HA, size);
if (testerr!= cudaSuccess) {
printf("Error en cudaMalloc HA: %s\n", cudaGetErrorString(testerr));
exit(0);
}
testerr = cudaMallocHost((void**)&HB, size);
if (testerr!= cudaSuccess) {
printf("Error en cudaMalloc HB: %s\n", cudaGetErrorString(testerr));
exit(0);
}
testerr = cudaMallocHost((void**)&HC, size);
if (testerr!= cudaSuccess) {
printf("Error en cudaMalloc HC: %s\n", cudaGetErrorString(testerr));
exit(0);
}
```

Para medir los tiempos he utilizado los eventos que proporciona la API de CUDA.

Tiempos cudaMalloc	DA	DB	DC
suma-vectores	0.106720 ms	0.004896 ms	0.077856 ms
suma-vect-pinned	0.010240 ms	0.522048 ms	0.005888 ms

9. Ejercicio 9

a) No hay coalescencia en las escrituras en memoria global, ya que, los accesos que se hacen en el puntero *dev_ b* tienen un *stride* de 43 elementos o 172 bytes, sin embargo, las lecturas en el puntero *dev_ a* si son coalescentes en memoria ya que los hilos leen de forma consecutiva posiciones contiguas de memoria.

b) Para que haya coalescencia es necesario utilizar memoria compartida para evitar los grandes recorridos a través de la memoria global.

```
__global__ void trspta1coalesc(int *dev_a, int *dev_b, int filas, int cols)
{
  __shared__ int tile[H][K];

  int x_index = blockIdx.x * blockDim.x;
  int y_index = blockIdx.y * blockDim.y;

  int ix = x_index + threadIdx.x;
  int iy = y_index + threadIdx.y;

  if ((ix<cols)&&(iy<filas)) {
    tile[threadIdx.y][threadIdx.x] = dev_a[ix+cols*iy];
  }

  __syncthreads();

  int index_out = x_index * filas + y_index;

  if (((x_index+threadIdx.y)<cols) && ((y_index+threadIdx.x)<filas)) {
    dev_b[index_out + threadIdx.y * filas + threadIdx.x] = tile[threadIdx.x][threadIdx.y];
  }
}
```

10. Ejercicio 10

En el código anterior, es necesario utilizar la directiva *__syncthreads()* ya que los datos que escriben los hilos en el puntero *dev_ b* son diferentes a los datos que se leen en el puntero *dev_ a*, por tanto es necesario utilizar una barrera de sincronización por bloques.

```
__global__ void trspta2(int *dev_a, int *dev_b, int filas, int cols)
{
  __shared__ int tile[H][K];

  int x_index = blockIdx.x * blockDim.x;
  int y_index = blockIdx.y * blockDim.y;

  int ix = x_index + threadIdx.x;
  int iy = y_index + threadIdx.y;

  if ((ix<cols)&&(iy<filas)) {
    tile[threadIdx.y][threadIdx.x] = dev_a[ix+cols*iy];
  }

  __syncthreads();

  int index_out = x_index * filas + y_index;

  if (((x_index+threadIdx.y)<cols) && ((y_index+threadIdx.x)<filas)) {
    dev_b[index_out + threadIdx.y * filas + threadIdx.x] = tile[threadIdx.x][threadIdx.y];
  }
}
```

11. Ejercicio 11

Todos los elementos de la matriz se asignan al mismo banco de memoria compartida, por tanto, esto da lugar a colisiones en bancos de memoria a la hora de realizar las lecturas de las columnas. Para solucionar esto, simplemente hacemos el *tile* de memoria compartida un poco más ancho.

```
__global__ void trspta3(int *dev_a, int *dev_b, int filas, int cols)
{
  __shared__ int tile[H][K+1];

  int x_index = blockIdx.x * blockDim.x;
  int y_index = blockIdx.y * blockDim.y;

  int ix = x_index + threadIdx.x;
  int iy = y_index + threadIdx.y;

  if ((ix<cols)&&(iy<filas)) {
    tile[threadIdx.y][threadIdx.x] = dev_a[ix+cols*iy];
  }

  __syncthreads();

  int index_out = x_index * filas + y_index;

  if (((x_index+threadIdx.y)<cols) && ((y_index+threadIdx.x)<filas)) {
    dev_b[index_out + threadIdx.y * filas + threadIdx.x] = tile[threadIdx.x][threadIdx.y];
  }
}
```

12. Ejercicio 12

Para comparar los tiempos obtenidos por los tres programas que calculan la traspuesta de una matriz, utilizaré eventos que proporciona la API de CUDA. Si medimos los tiempos y los anchos de banda obtenidos para calcular la traspuesta de una matriz de 25 filas por 43 columnas los resultados son prácticamente similares unos a otros. En este caso estoy ejecutando 100 veces cada kernel para tener unos resultados más fiables que con los de una única ejecución.

100 ejecuciones	Traspuesta1.cu	Traspuesta2.cu	Traspuesta3.cu
Tiempo (ms)	0.352864	0.348864	0.356864
Ancho de banda (GB/s)	2.44	2.47	2.41

Esto se debe a que el tamaño de la matriz es bastante reducido y solo se realiza una única ejecución del *kernel*, por lo que, para comprobar realmente la mejora, voy a aumentar el tamaño de la matriz a 250 filas y 430 columnas y además voy a ejecutar 100 veces cada *kernel*. Estos son los resultados obtenidos:

100 ejecuciones	Traspuesta1.cu	Traspuesta2.cu	Traspuesta3.cu
Tiempo (ms)	3.265152	2.556224	2.551360
Ancho de banda (GB/s)	26.34	33.64	33.71

Ahora sí que podemos ver una mejora con respecto al primer programa que no aprovecha la técnica de coalescencia en memoria en su totalidad.