Santi Estebanell, 229858
Dorian Erazo Orozco, 219395

Introducción a la Programación Paralela y Distribuida: LAB 4

Image Processing (CUDA)

Exercise 1:

En este ejercicio utilizaremos CUDA para ejecutar las funciones del anterior lab. En CUDA, la GPU o device se encarga de ejecutar los kernels que le indicamos, que serán nuestras funciones para cada imágen. Para ello hay que declarar las funciones como "__global__" y eliminar los bucles, que serán sustituidos por la condición if, que dependerá del número de hilo y bloque en el que se ejecute. Haremos lo siguiente para cada una de las funciones:

```
/*invert*/
__global__ void invert(int* image, int* image_invert, int nx, int ny){

   int indx = threadIdx.x + blockIdx.x * blockDim.x;
   int indy = threadIdx.y + blockIdx.y * blockDim.y;

   if(indx >= 0 && indx <= nx - 1){
      if(indy >= 0 && indy <= ny - 1){
         image_invert[pixel(indx,indy,nx)] = 255-image[pixel(indx,indy,nx)];
      }
   }
}
```

A continuación, en nuestro main vamos a declarar el tamaño de nuestro Grid y de cada uno de los bloques que lo conforman.

```
/* Saving blocks */
dim3 dimBlock(B,B,1);
int dimgx = (nx+B-1)/B;
int dimgy = (ny+B-1)/B;
dim3 dimGrid(dimgx, dimgy, 1);
```

El siguiente paso será reservar memoria en nuestra máquina y también en la GPU, para ello usaremos las funciones malloc() y cudaMalloc(). Reservamos espacio para las imágenes y para nuestro grid.

```
/* Allocate CPU and GPU pointers */
int*   image = (int *) malloc(sizeof(int)*nx*ny);
int*   image_invert  = (int *) malloc(sizeof(int)*nx*ny);
int*   image_smooth  = (int *) malloc(sizeof(int)*nx*ny);
int*   image_detect  = (int *) malloc(sizeof(int)*nx*ny);
int*   image_enhance = (int *) malloc(sizeof(int)*nx*ny);

cudaMalloc((void **)&dev_image, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_invert, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_smooth, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_detect, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_enhance, sizeof(int)*nx*ny);

cudaMalloc((void **) &dev_blockMax,dimgx*dimgy*sizeof(double));
```

Para ejecutar los kernels en nuestra GPU, primero hay que pasarle la información de la imágen original que vamos a modificar en nuestro device:

```
/* copy mem for image */
cudaMemcpy(dev_image, image, sizeof(int)*nx*ny, cudaMemcpyHostToDevice);
```

Santi Estebanell, 229858
Dorian Erazo Orozco, 219395

Una vez tenemos todo preparado, tan solo nos queda ejecutar los kernels en la GPU, los lanzaremos de la siguiente manera:

```
invert<<<dimGrid, dimBlock>>>(dev_image, dev_image_invert, nx, ny);
smooth<<<dimGrid,dimBlock>>>(dev_image, dev_image_smooth, nx, ny);
detect<<<dimGrid,dimBlock>>>(dev_image, dev_image_detect, nx, ny);
enhance<<<dimGrid,dimBlock>>>(dev_image, dev_image_enhance, nx, ny);
```

En cuanto se hayan ejecutado, nos quedará recuperar las imágenes modificadas en la gpu, las tenemos que pasar a nuestra máquina (host).

```
/* memory copy from device to host */
cudaMemcpy(image_invert, dev_image_invert, nx*ny*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(image_smooth, dev_image_smooth, nx*ny*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(image_detect, dev_image_detect, nx*ny*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(image_enhance, dev_image_enhance, nx*ny*sizeof(int), cudaMemcpyDeviceToHost);
```

Una vez recuperadas, guardamos las imágenes en nuestro fichero. Tan sólo nos queda liberar el espacio que hemos reservado en nuestra máquina y en la GPU:

```
/* Deallocate CPU and GPU pointers*/
free(image);
free(image_invert);
free(image_smooth);
free(image_detect);
free(image_enhance);

cudaFree(dev_image);
cudaFree(dev_image_invert);
cudaFree(dev_image_detect);
cudaFree(dev_image_smooth);
cudaFree(dev_image_enhance);
cudaFree(dev_blockMax);
```

Para medir los tiempos obtenidos en la ejecución de los kernels, nos hemos valido de la función “cudaEventRecord()”, la cual utiliza “eventos” para marcar los puntos entre los que queremos medir el tiempo.

Santi Estebanell, 229858
Dorian Erazo Orozco, 219395

Exercise 2:

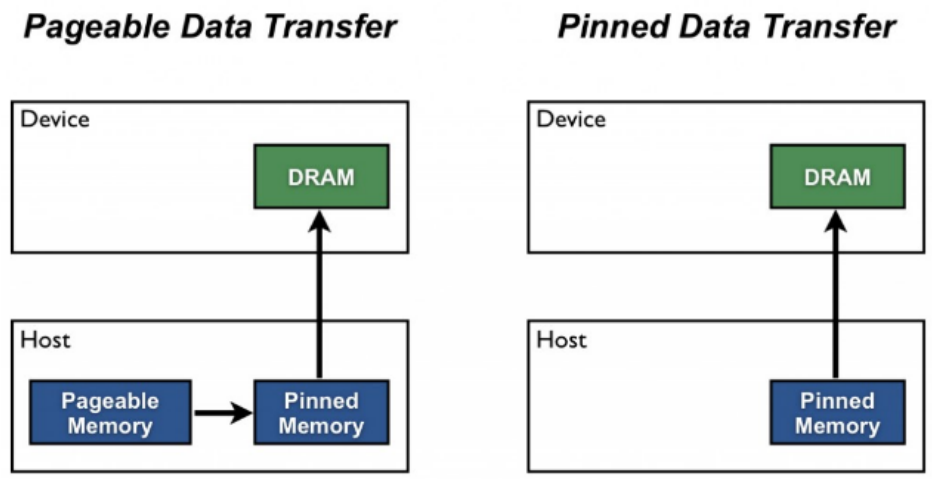
Para resolver el problema 2 volvemos a usar la imagen que nos dan, “lleo.txt”. Vamos a reciclar las funciones modificadas en el problema 1. Por lo tanto esa parte será exactamente igual.

Todo lo que iremos cambiado será en el main program.
Lo primero que añadimos al código es lo propio que nos pide el problema 2, los streams. Creamos cuatro streams, uno para cada función y su memcopy.

Cabe decir que los streams los creamos con el objetivo de ejecutar concurrentemente la iteración de cada tipo de imagen.

```
179     // cuda stream
180     cudaStream_t stream1, stream2, stream3, stream4;
181     cudaStreamCreate(&stream1);
182     cudaStreamCreate(&stream2);
183     cudaStreamCreate(&stream3);
184     cudaStreamCreate(&stream4);
```

Lo siguiente es hacer las reservas de memoria tanto como el *Host (nuestra máquina)* como en *Device (GPU)*. Para reservar la memoria en el Host lo podemos hacer de dos maneras, con un *malloc* de C o con pinned memory (*cudaMallocHost()*). Nosotros lo hemos escrito de las dos maneras, lo único que cambia es que pinned memory debería de ser más rápido.



La pinned memory la usamos para la preparación de las transferencias entre device y host. Si usamos pinned nos ahorramos la parte de “pageable memory”.

En el código nos queda de la siguiente manera:

Santi Estebanell, 229858
Dorian Erazo Orozco, 219395

```
193     /* Allocate CPU and GPU pointers */
194     //Para pinned memory
195     int*   image;
196     cudaMallocHost((void **) &image, nx*ny*sizeof(int));
197
198     int*   image_invert;
199     cudaMallocHost((void **) &image_invert, nx*ny*sizeof(int));
200
201     int*   image_smooth;
202     cudaMallocHost((void **) &image_smooth, sizeof(int)*nx*ny);
203
204     int*   image_detect;
205     cudaMallocHost((void **) &image_detect, sizeof(int)*nx*ny);
206
207     int*   image_enhance;
208     cudaMallocHost((void **) &image_enhance, sizeof(int)*nx*ny);
209
210     /*
211     sin pinned memory
212     int*   image =(int *) malloc(sizeof(int)*nx*ny);
213     int*   image_invert  = (int *) malloc(sizeof(int)*nx*ny);
214     int*   image_smooth  = (int *) malloc(sizeof(int)*nx*ny);
215     int*   image_detect = (int *) malloc(sizeof(int)*nx*ny);
216     int*   image_enhance = (int *) malloc(sizeof(int)*nx*ny);
217     */
```

También debemos de reservar la memoria para cada imagen en nuestra GPU igual que en el problema 1 (*cudaMalloc*). A parte debemos de pasar nuestra imagen base del Host al Device (*cudaMemcpy*) para poder tratarla en el destino.

```
cudaMalloc((void **)&dev_image, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_invert, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_smooth, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_detect, sizeof(int)*nx*ny);
cudaMalloc((void **)&dev_image_enhance, sizeof(int)*nx*ny);

/* Read image and save in array imgage */
readimg(filename,nx,ny,image);
cudaMemcpy(dev_image, image, nx*ny*sizeof(int), cudaMemcpyHostToDevice);
```

Ahora viene la parte en la que llamamos a los kernels y las funciones se ejecutan de manera concurrente. Es decir, usamos los streams que hemos creado. Para invocar un kernel en este caso la sintaxis es diferente, por ejemplo:

invert<<<dimGrid,dimBlock, 0, stream1>>>(dev_image, dev_image_invert, nx, ny);

Aquí tenemos la invocación del kernel. invert<<<bloques, threads, smem (0), número de stream>>>.
A diferencia del problema uno es que aquí usamos un smem 0 y asignamos un stream a cada función.

```
231     cudaEventRecord(start);
232     invert<<<dimGrid,dimBlock, 0, stream1>>>(dev_image, dev_image_invert, nx, ny);
233
234     smooth<<<dimGrid, dimBlock, 0, stream2>>>(dev_image, dev_image_smooth, nx, ny);
235
236     detect<<<dimGrid, dimBlock, 0, stream3>>>(dev_image, dev_image_detect, nx, ny);
237
238     enhance<<<dimGrid, dimBlock, 0, stream4>>>(dev_image, dev_image_enhance, nx, ny);
239     cudaEventRecord(stop);
```

Santi Estebanell, 229858
Dorian Erazo Orozco, 219395

Para finalizar el procesamiento de imágenes llamamos a nuestro memcopy, pero esta vez de manera asíncrona respecto al host, por lo cual la llamada puede regresar antes de que se complete la copia.
Aparte de eso sincronizamos cada stream, que se encarga de esperar a que cada tarea del stream se complete.

```
241     cudaMemcpyAsync(image_invert, dev_image_invert, nx*ny*sizeof(int), cudaMemcpyDeviceToHost, stream1);
242     cudaStreamSynchronize(stream1);
243
244     cudaMemcpyAsync(image_smooth, dev_image_smooth, sizeof(int)*nx*ny, cudaMemcpyDeviceToHost, stream2);
245     cudaStreamSynchronize(stream2);
246
247     cudaMemcpyAsync(image_detect, dev_image_detect, sizeof(int)*nx*ny, cudaMemcpyDeviceToHost, stream3);
248     cudaStreamSynchronize(stream3);
249
250     cudaMemcpyAsync(image_enhance, dev_image_enhance, sizeof(int)*nx*ny, cudaMemcpyDeviceToHost, stream4);
251     cudaStreamSynchronize(stream4);
252
253     cudaEventSynchronize(stop);
```

Finalmente liberamos la memoria utilizada y destruimos los streams.

```
270     /* Deallocate CPU and GPU pointers*/
271     free(image);
272     cudaFreeHost(image_invert);
273     cudaFreeHost(image_smooth);
274     cudaFreeHost(image_detect);
275     cudaFreeHost(image_enhance);
276
277     cudaFree(dev_image);
278     cudaFree(dev_image_invert);
279     cudaFree(dev_image_detect);
280     cudaFree(dev_image_smooth);
281     cudaFree(dev_image_enhance);
282
283     cudaStreamDestroy(stream1);
284     cudaStreamDestroy(stream2);
285     cudaStreamDestroy(stream3);
286     cudaStreamDestroy(stream4);
287 }
```

El tiempo total con respecto al problema uno no mejora mucho, en clase se comentó que era por la tecnología de las GPU que usamos, aún así hemos conseguido recortar unas décimas (depende de cuando lo ejecutemos).

Con Pinned Memmory y comprobación de las imágenes:

```
Total time transformations: 0.940032
[u162359@ip-10-49-0-22 2_image_streams]$ sh scripts/compare.sh
```

Sin Pinned Memmory y comprobación de las imágenes:

```
lleo.txt 1920 1214
Total time transformations: 1.055232
[u162359@ip-10-49-0-22 2_image_streams]$ sh scripts/compare.sh
```