# PRACTICA ENTRGABLE #3: PROCESAMIENTO DE IMÁGENES

Aaron Rojas Y David Peirotén

```
/**
*
* ARQUITECTURA DE COMPUTADORES
* 2º Grado en Ingenieria Informatica
* Curso: 2022 - 2023
*
* ENTREGA no.3 <Procesamiento de Imagenes>
*
* EQUIPO : TE - C - 25
* MIEMBROS : Aaron Rojas Gutierrez y David Peiroten Herrero
*
*
*
*/

// includes
#include <stdio.h>

#include <stdlib.h>

#include <cuda_runtime.h>

#include "gpu_bitmap.h"

#define ANCHO 20 // Dimension horizontal
#define ALTO 20 // Dimension vertical
// HOST: funcion llamada desde el host y ejecutada en el host
__host__ void propiedades_Device(int deviceID);
__host__ void leerBMP_RGBA(const char * nombre, int * w, int * h, unsigned char
** imagen);
// Funcion que lee un archivo de tipo BMP:
// -> entrada: nombre del archivo
// <- salida : ancho de la imagen en pixeles
// <- salida : alto de la imagen en pixeles
// <- salida : puntero al array de datos de la imagen en formato RGBA

//Funcion Kernel
__global__ void kernel( unsigned char  *host_color, unsigned char *host_gris ){

	int x = threadIdx.x + blockIdx.x * blockDim.x;
	// coordenada vertical de cada hilo
	int y = threadIdx.y + blockIdx.y * blockDim.y;
	// indice global de cada hilo (indice lineal para acceder a la memoria)
	int myID = x + y * blockDim.x * gridDim.x;
	int miPixel = myID * 4;
	int gris = host_color[miPixel + 0] * 0.299 + host_color[miPixel + 1] *
0.587 + host_color[miPixel + 2] * 0.114;

	host_gris[miPixel + 0] = gris; // canal R
	host_gris[miPixel + 1] = gris;// canal G
	host_gris[miPixel + 2] = gris;// canal B
	host_gris[miPixel + 3] = 0; // canal alfa
```

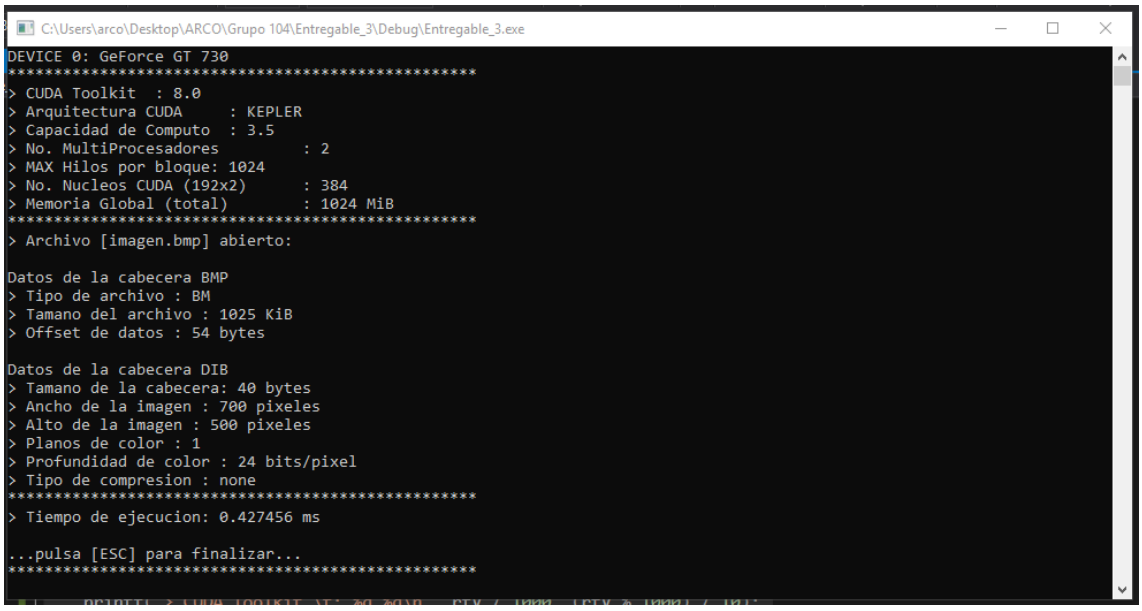
```
}

////////////////////////////////////////////////////////////////////////
// MAIN: rutina principal ejecutada en el host
int main(int argc, char ** argv) {
	// buscando dispositivos
	int deviceCount;
	cudaGetDeviceCount(&deviceCount);
	if (deviceCount == 0)
	{
		printf("!!!!!No se han encontrado dispositivos CUDA!!!!!\n");
		printf("<pulsa [INTRO] para finalizar>");
		getchar();
		return 1;
	}
	else
	{
		printf("Se han encontrado <%d> dispositivos CUDA:\n", deviceCount);
		for (int id = 0; id < deviceCount; id++)
		{
			propiedades_Device(id);
		}
	}

	//Declarar variables y eventos para monitorizar el tiempi
	cudaEvent_t start;
	cudaEvent_t stop;

	//Creacion de eventos
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	////////////////////////////////////////////////////////////////////////
///
	// Leemos el archivo BMP
	unsigned char * host_color, *dev_gris;
	int ancho, alto;
	leerBMP_RGBA("imagen.bmp", &ancho, &alto, &host_color);
	////////////////////////////////////////////////////////////////////////
///
	// Declaracion del bitmap RGBA:
	// Inicializacion de la estructura RenderGPU
	RenderGPU foto(ancho, alto);
	// Tamaño del bitmap en bytes
	size_t img_size = foto.image_size();
	// Asignacion y reserva de la memoria en el host (framebuffer)
	unsigned char * host_bitmap = foto.get_ptr();
	unsigned char *dev_color;
	// Copiamos en el framebuffer los datos leidos para su visualizacion:
	// Reserva en el device
	cudaMalloc((void**)&dev_color, img_size);
	cudaMalloc((void**)&dev_gris, img_size);
	cudaMemcpy(dev_color, host_color, img_size, cudaMemcpyHostToDevice);
	dim3 hilosB(20, 20);

	// Calculamos el numero de bloques necesario (un hilo por cada pixel)
	dim3 Nbloques(ancho /20, alto /20);
	// Salida
	// Visualizacion
	cudaEventRecord(start, 0);
	kernel << <Nbloques, hilosB >> >(dev_color,dev_gris);
	cudaEventRecord(stop, 0);
```

```
	//Sincronizacion de eventos
	cudaEventSynchronize(stop);

	//Calculo de intentos en milisegundos
	float elapsedTime;

	cudaEventElapsedTime(&elapsedTime, start, stop);

	// Visualizacion y salida
	printf("*************************************************\n");
	printf("> Tiempo de ejecucion: %f ms\n", elapsedTime);
	printf("\n...pulsa [ESC] para finalizar...\n");
	printf("*************************************************\n");
	// liberacion de recursos
	cudaEventDestroy(start);
	cudaEventDestroy(stop);
	cudaMemcpy(host_bitmap, dev_gris, img_size, cudaMemcpyDeviceToHost);
	foto.display_and_exit();
	// Fin
	return 0;
}
////////////////////////////////////////////////////////////////////////////
// Funcion que lee un archivo de tipo BMP:
__host__ void leerBMP_RGBA(const char * nombre, int * w, int * h, unsigned char
** imagen) {
	// Lectura del archivo .BMP
	FILE * archivo;

	// Abrimos el archivo en modo solo lectura binario
	if ((archivo = fopen(nombre, "rb")) == NULL) {
		printf("\nERROR ABRIENDO EL ARCHIVO %s...", nombre);
		// salida
		printf("\npulsa [INTRO] para finalizar");
		getchar();
		exit(1);
	}
	printf("> Archivo [%s] abierto:\n", nombre);
	// En Windows, la cabecera tiene un tamaño de 54 bytes:
	// 14 bytes (BMP header) + 40 bytes (DIB header)

	// BMP HEADER
	// Extraemos cada campo y lo almacenamos en una variable del tipo adecuado
	// posicion 0x00 -> Tipo de archivo: "BM" (leemos 2 bytes)
	unsigned char tipo[2];
	fread(tipo, 1, 2, archivo);
	// Comprobamos que es un archivo BMP
	if (tipo[0] != 'B' || tipo[1] != 'M') {
		printf("\nERROR: EL ARCHIVO %s NO ES DE TIPO BMP...", nombre);
		// salida
		printf("\npulsa [INTRO] para finalizar");
		getchar();
		exit(1);
	}
	// posicion 0x02 -> Tamaño del archivo .bmp (leemos 4 bytes)
	unsigned int file_size;
	fread(&file_size, 4, 1, archivo);

	// posicion 0x06 -> Campo reservado (leemos 2 bytes)
	// posicion 0x08 -> Campo reservado (leemos 2 bytes)
	unsigned char buffer[4];
	fread(buffer, 1, 4, archivo);
```

```
	// posicion 0x0A -> Offset a los datos de imagen (leemos 4 bytes)
	unsigned int offset;
	fread(&offset, 4, 1, archivo);

	// imprimimos los datos
	printf(" \nDatos de la cabecera BMP\n");
	printf("> Tipo de archivo : %c%c\n", tipo[0], tipo[1]);
	printf("> Tamano del archivo : %u KiB\n", file_size / 1024);
	printf("> Offset de datos : %u bytes\n", offset);
	// DIB HEADER
	// Extraemos cada campo y lo almacenamos en una variable del tipo adecuado
	// posicion 0x0E -> Tamaño de la cabecera DIB (BITMAPINFOHEADER) (leemos 4
bytes)
	unsigned int header_size;
	fread(&header_size, 4, 1, archivo);

	// posicion 0x12 -> Ancho de la imagen (leemos 4 bytes)
	unsigned int ancho;
	fread(&ancho, 4, 1, archivo);

	// posicion 0x16 -> Alto de la imagen (leemos 4 bytes)
	unsigned int alto;
	fread(&alto, 4, 1, archivo);

	// posicion 0x1A -> Numero de planos de color (leemos 2 bytes)
	unsigned short int planos;
	fread(&planos, 2, 1, archivo);

	// posicion 0x1C -> Profundidad de color (leemos 2 bytes)
	unsigned short int color_depth;
	fread(&color_depth, 2, 1, archivo);

	// posicion 0x1E -> Tipo de compresion (leemos 4 bytes)
	unsigned int compresion;
	fread(&compresion, 4, 1, archivo);

	// imprimimos los datos
	printf(" \nDatos de la cabecera DIB\n");
	printf("> Tamano de la cabecera: %u bytes\n", header_size);
	printf("> Ancho de la imagen : %u pixeles\n", ancho);
	printf("> Alto de la imagen : %u pixeles\n", alto);
	printf("> Planos de color : %u\n", planos);
	printf("> Profundidad de color : %u bits/pixel\n", color_depth);
	printf("> Tipo de compresion : %s\n", (compresion == 0) ? "none" :
"unknown");
	// LEEMOS LOS DATOS DEL ARCHIVO
	// Calculamos espacio para una imagen de tipo RGBA:
	size_t img_size = ancho * alto * 4;
	// Reserva para almacenar los datos del bitmap
	unsigned char * datos = (unsigned char *)malloc(img_size);;
	// Desplazamos el puntero FILE hasta el comienzo de los datos de imagen: 0
+ offset
	fseek(archivo, offset, SEEK_SET);
	// Leemos pixel a pixel, reordenamos (BGR -> RGB) e insertamos canal alfa
	unsigned int pixel_size = color_depth / 8;
	for (unsigned int i = 0; i < ancho * alto; i++) {
		fread(buffer, 1, pixel_size, archivo); // leemos el pixel i
		datos[i * 4 + 0] = buffer[2]; // escribimos canal R
		datos[i * 4 + 1] = buffer[1]; // escribimos canal G
		datos[i * 4 + 2] = buffer[0]; // escribimos canal B
		datos[i * 4 + 3] = buffer[3]; // escribimos canal alfa (si lo hay)
	}
```

```
	// Cerramos el archivo
	fclose(archivo);
	// PARAMETROS DE SALIDA
	// Ancho de la imagen en pixeles
	*w = ancho;
	// Alto de la imagen en pixeles
	*h = alto;
	// Puntero al array de datos RGBA
	*imagen = datos;
	// Salida
	return;
}
__host__ void propiedades_Device(int deviceID)
{
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, deviceID);
	// calculo del numero de cores (SP)
	int cudaCores = 0;
	int SM = deviceProp.multiProcessorCount;
	int major = deviceProp.major;
	int minor = deviceProp.minor;
	const char *archName;
	switch (major)
	{
	case 1:
		//TESLA
		archName = "TESLA";
		cudaCores = 8;
		break;
	case 2:
		//FERMI
		archName = "FERMI";
		if (minor == 0)
			cudaCores = 32;
		else
			cudaCores = 48;
		break;
	case 3:
		//KEPLER
		archName = "KEPLER";
		cudaCores = 192;
		break;
	case 5:
		//MAXWELL
		archName = "MAXWELL";
		cudaCores = 128;
		break;
	case 6:
		//PASCAL
		archName = "PASCAL";
		cudaCores = 64;
		break;
	case 7:
		//VOLTA(7.0) //TURING(7.5)
		cudaCores = 64;
		if (minor == 0)
			archName = "VOLTA";
		else
			archName = "TURING";
		break;
	case 8:
		// AMPERE
```

```
			archName = "AMPERE";
			cudaCores = 64;
			break;
		default:
			//ARQUITECTURA DESCONOCIDA
			archName = "DESCONOCIDA";
		}
		int rtV;
		cudaRuntimeGetVersion(&rtV);
		// presentacion de propiedades
		printf("****************************************************\n");
		printf("DEVICE %d: %s\n", deviceID, deviceProp.name);
		printf("****************************************************\n");
		printf("> CUDA Toolkit \t: %d.%d\n", rtV / 1000, (rtV % 1000) / 10);
		printf("> Arquitectura CUDA \t: %s\n", archName);
		printf("> Capacidad de Computo \t: %d.%d\n", major, minor);
		printf("> No. MultiProcesadores \t: %d\n", SM);
		printf("> MAX Hilos por bloque: %d\n", deviceProp.maxThreadsPerBlock);
		printf("> No. Nucleos CUDA (%dx%d) \t: %d\n", cudaCores, SM,
cudaCores*SM);
		printf("> Memoria Global (total) \t: %u MiB\n",
			deviceProp.totalGlobalMem / (1024 * 1024));
		printf("****************************************************\n");
}
```




Display: Arquitectura de Computadores