ENTREGABLE 1: TEMPORIZACION DE GPU

AARON ROJAS Y DAVID PEIROTEN

```
/*
* ARQUITECTURA DE COMPUTADORES
* 2º Grado en Ingenieria Informatica
* Curso: 2022-2023
*
* ENTREGA no.1 <Temporizacion de GPU>
*
* EQUIPO:TE-C-25
* MIEMBROS: Aaron Rojas Gutierrez y David Peiroten Herrero
*
*
*
*/


//Cargamos las librerias necesarias
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

//Definimos el tamano de las columnas y las filas
#define TAMF 7 // tamaño de la FILA
#define TAMC 25 // tamaño de la COLUMNA

//Prototipos de la funciones
__host__ void propiedades_Device(int deviceID);

// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma_gpu(int *A, int *C)
{
	//Variables con los indices de las columnas y filas
	int IndFila = threadIdx.y;
	int IndColumna = threadIdx.x;

	int filaprueba;
	// Calculamos la suma:
	// C[fila][columna] = A[fila][columna] + B[fila][columna]
	// Para ello convertimos los indices de 'fila' y 'columna' en un indice
lineal
	int myID = IndColumna + IndFila * blockDim.x;
	int myID2 = IndColumna + (IndFila + 6) * blockDim.x;
	int myID3 = IndColumna + (IndFila + 1) * blockDim.x;
	C[myID3] = A[myID];
	//Si la columna es par la cambiamos por 0
	if (IndFila == 7) {
		C[myID] = A[myID];
	}
	else {
		C[myID3] = A[myID];
	}
}
// 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);
	}
}

//Calculamos el numero de hilos lanzados
int n_hilos;

n_hilos = TAMF * TAMC;

cudaDeviceProp deviceProp;
int deviceID;
cudaGetDevice(&deviceID);
cudaGetDeviceProperties(&deviceProp, deviceID);
// salida del programa
time_t fecha;
time(&fecha);

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

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


// declaraciones
int *hst_A, *hst_C;
int *dev_A, *dev_C;
// reserva en el host
hst_A = (int*)malloc(TAMF*TAMC * sizeof(int));
hst_C = (int*)malloc(TAMF*TAMC * sizeof(int));
// reserva en el device
cudaMalloc((void**)&dev_A, TAMF*TAMC * sizeof(int));
cudaMalloc((void**)&dev_C, TAMF*TAMC * sizeof(int));
// incializacion
srand((int)time(NULL));
for (int i = 0; i<TAMF*TAMC; i++)
{
	hst_A[i] = rand() % 9 + 1;
}
// copia de datos
cudaMemcpy(dev_A, hst_A, TAMF*TAMC * sizeof(int), cudaMemcpyHostToDevice);

// dimensiones del kernel
dim3 Nbloques(1);
dim3 hilosB(TAMC, TAMF);
```

```
	//Marcamos el tiempo de inicio
	cudaEventRecord(start, 0);

	// llamada al kernel bidimensional de NxN hilos
	suma_gpu << <Nbloques, hilosB >> >(dev_A, dev_C);
	// recogida de datos
	cudaMemcpy(hst_C, dev_C, TAMF*TAMC * sizeof(int), cudaMemcpyDeviceToHost);
	//Marca de finalizacion

	cudaEventRecord(stop, 0);

	//Sincronizacion de eventos
	cudaEventSynchronize(stop);

	//Calculo de intentos en milisegundos
	float elapsedTime;

	cudaEventElapsedTime(&elapsedTime, start, stop);

	// impresion de resultados
	printf("> Hemos lanzado %d hilos \n",n_hilos);
	printf("> Eje x --> %d \n", TAMC);
	printf("> Eje y --> %d \n", TAMF);
	printf("> Tiempo de ejecucion: %f ms\n", elapsedTime);
	printf("*************************************************\n");
	// liberacion de recursos
	cudaEventDestroy(start);
	cudaEventDestroy(stop);

	// impresion de resultados
	printf("MATRIZ A:\n");
	for (int i = 0; i<TAMF; i++)
	{
		for (int j = 0; j<TAMC; j++)
		{
			printf("%d ", hst_A[j + i*TAMC]);
		}
		printf("\n");
	}
	printf("\n");


	printf("MATRIZ B:\n");
	for (int i = 0; i<TAMF; i++)
	{
		for (int j = 0; j<TAMC; j++)
		{
			printf("%d ", hst_C[j + i*TAMC]);
		}
		printf("\n");
	}
	// salida

	time(&fecha);
	printf("*************************************************\n");
	printf("<pulsa [INTRO] para finalizar>");
	getchar();
	return 0;
}
__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");
}
////////////////////////////////////////////////////////////////////////////
```