

# Práctica 7. Programación de GPUs. Kernel

Métodos Numéricos para la Computación

Grado en Ingeniería Informática. Mención Computación

Escuela de Ingeniería Informática

Universidad de Las Palmas de Gran Canaria



# Contenidos

- Utilizar Visual Studio para construir aplicaciones que utilizan CUDA.
- Aprender a identificar el número y las características de las GPUs compatibles.
- Desarrollar aplicaciones sencillas de matrices, con uso de Kernels.
- Utilizar NSIGHT, si es posible, y el Occupancy Calculator.
- Utilizar múltiples hilos OpenMP para gestionar streams de la GPU, incrementando las prestaciones de las aplicaciones numéricas.



# Tarea 1. miGPU

- Aplicación que nos permita conocer las características de la GPU instalada. Es una modificación del ejemplo DeviceQuery de NVIDIA.
- Utilizaremos el plugin de CUDA dentro de Visual Studio para desarrollar aplicaciones. Esto facilitará enormemente nuestro trabajo, por ejemplo en el uso transparente del compilador nvcc.
- Visual Studio crea un programa ejemplo que realiza suma de vectores. Borraremos todos los contenidos para adaptarnos a nuestro proyecto.
- La función `printDeviceProperties()` se le suministra al alumno en el Campus Virtual. Insertarla en el proyecto.
- Más información sobre propiedades de la GPU en:  
[http://developer.download.nvidia.com/compute/cuda/4\\_1/rel/toolkit/docs/online/group\\_CUDART\\_DEVICE\\_g5aa4f47938af8276f08074d09b7d520c.html](http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group_CUDART_DEVICE_g5aa4f47938af8276f08074d09b7d520c.html)



# Tarea 1. Pasos

1. Crear un proyecto denominado miGPU del tipo CUDA
2. Borrar los contenidos de kernel.cu.
3. Renombrar el fichero kernel.cu como DeviceProperties.cpp. Este proyecto no utiliza Kernels y el compilador NVCC no necesita separar el código para la GPU. Todo el código se ejecuta en la CPU mediante el uso de la librería Runtime.
4. Añadir un elemento nuevo, que se denominará printDeviceProperties.cpp en el que copiaremos el contenido de un fichero suministrado en el Campus Virtual.
5. Generar la solución, en modo Release.
6. Ejecutar
7. Interpretar los resultados de la GPU.



# Código principal

```
miGPU (Ám)
1 #include <stdio>
2 #include "cuda_runtime.h"
3
4 extern void printDeviceProperties(cudaDeviceProp devProp);
5
6 int main(int argc, char *argv[]){
7
8     // cuenta el numero de dispositivos
9     int devCount;
10    cudaGetDeviceCount(&devCount);
11    printf("Buscando dispositivos CUDA ...\n");
12    printf("Hay %d dispositivos CUDA\n",devCount);
13
14    // imprime las características de cada uno
15    for (int i = 0; i < devCount; i++){
16        printf("\nDispositivo CUDA #%d\n",i);
17        cudaDeviceProp devProp;
18        cudaGetDeviceProperties(&devProp, i);
19        printDeviceProperties(devProp);
20    }
21
22    return 0;
23 }
```

Obtiene el número de GPUs

Obtiene las características de la GPU i

Función auxiliar suministrada



```
C:\Windows\system32\cmd.exe
Buscando dispositivos CUDA ...
Hay 1 dispositivos CUDA

Dispositivo CUDA #0
Name: GeForce GT 540M
Compute capability: 2.1
Total global memory: 2147483648
Total shared memory per block: 49152
Total registers per block: 32768
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024

Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64

Maximum dimension 0 of grid: 65535
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535

Clock rate: 1344000
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 2
Kernel execution timeout: Yes
Presione una tecla para continuar . . .
```

Comentario de los datos de la GPU:

El nombre de la placa es GT 570M, luego es un portátil (M de Mobility)

La *compute capability* es la 2.1, luego soporta aritmética de doble precisión.

Posee 2Gb de memoria en la GPU

El Warp es de 32 hilos que ejecutan la misma instrucción en modo SIMD

El número de hilos por bloques es de 1024, aunque luego admite diversas configuraciones en x,y,z

El número de multiprocesadores es bajo: 2 y lentos 1.34 Ghz. No se espera grandes potencias de cálculo.



```
C:\Windows\system32\cmd.exe
Buscando dispositivos CUDA ...
Hay 1 dispositivos CUDA

Dispositivo CUDA #0
Name: GeForce GT 530
Compute capability: 2.1
Total global memory: 2147483648
Total shared memory per block: 49152
Total registers per block: 32768
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024

Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64

Maximum dimension 0 of grid: 65535
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535

Clock rate: 1400000
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 2
Kernel execution timeout: Yes
Presione una tecla para continuar . . . _
```

Otro ejemplo: Los equipos del Laboratorio 1-2, con una placa GT530 que es muy similar en características y prestaciones al caso anterior. El reloj es ligeramente más rápido (1.4 Ghz)



# Tarea 2. Suma de matrices

- Desarrollaremos un programa que sume ponderadamente dos matrices en la GPU usando un Kernel. Es una operación demasiado simple para que la GPU resulte competitiva.
- Realizará la siguiente operación:  $C(i, j) = \alpha A(i, j) + \beta B(i, j)$
- Se utilizarán matrices de grandes dimensiones, pero acorde a la memoria de GPU disponible. Por ejemplo 3 matrices de 6Kx6K. La eficiencia se facilita si las dimensiones de las matrices son múltiplos de 2.
- Mediremos tiempos para comparar con la CPU, incluyendo tiempos de transferencia de los datos/resultado y de ejecución del Kernel.
- Las expectativas de prestaciones son escasas dada la simplicidad de las operaciones y el sobre-coste de las transferencias.





# Tarea 2. Pasos

1. Crear un proyecto del tipo CUDA denominado sumaGPU.
2. Añadiremos nuevos elementos como serán las copias de los ficheros eTimer.h y eTimer.cpp para facilitar el cronometraje.
3. Aprovecharemos la declaración de la función del Kernel e incluiremos la totalidad del código que se explica en las siguientes hojas.
4. Ejecutar el programa.
5. Interpretar los resultados.
6. Descargarse la hoja de cálculo Occupancy Calculator e introducir los datos. (compute capability: 2.1, hilos por bloque:  $32 \times 16 = 512$ , registros 12, memoria compartida: 0)  
Probar 512 hilos/bloque y luego 1024 hilos/bloque.



# Cabeceras y Kernel

```
sumaGPUlab (Ámbito global)
1 #include <stdio>
2 #include <random>
3 #include <string.h>
4
5 #include "cuda_runtime.h"
6 #include "device_launch_parameters.h"
7
8 // incluir o comentar la línea según se desee cronometrar
9 #define CRONO
10
11 #if defined CRONO
12 #include "eTimer.h" // utilidad para medir tiempos
13 #endif
14
15 // define el tamaño de la matriz a 6K
16 #define N 6*1024
17
18 __global__ void sumaKernel( double *c, const double *a, const double *b,
19                             const double alpha, const double beta){
20
21     // localiza las coordenadas absolutas en base al bloque y al hilo
22     int x = blockIdx.x*blockDim.x + threadIdx.x;
23     int y = blockIdx.y*blockDim.y + threadIdx.y;
24     c[y*N+x] = alpha*a[y*N+x] + beta*b[y*N+x];
25 }
26
```

El código incorpora el cronometraje  
opcional mediante directivas de preproceso.  
Si se desean se puede eliminar

Código del Kernel



# Programa principal(1)

```
27
28 int main(int argc, char *argv[])
29 {
30     cudaError_t cudaStatus;
31
32     double *A, *B, *C;
33     double alpha = 0.7;
34     double beta = 0.6;
35     std::default_random_engine generador;
36     std::normal_distribution<double> distribucion(0.0, 1.0);
37
38     // reservamos espacio en la memoria central para A,B y C
39     // version de Microsoft para malloc alineado
40     A = (double*)_aligned_malloc(N*N*sizeof(double), 64);
41     B = (double*)_aligned_malloc(N*N*sizeof(double), 64);
42     C = (double*)_aligned_malloc(N*N*sizeof(double), 64);
43
44     // rellenamos aleatoriamente las matrices A y B
45     for (int i = 0; i < N; i++){
46         for (int j = 0; j < N; j++){
47             A[i*N + j] = distribucion(generador);
48             B[i*N + j] = distribucion(generador);
49         }
50     }
51 }
```

Reserva de espacio en memoria central usando la versión de Microsoft de malloc() con alineamiento



# Programa principal(2)

```
51
52     cudaStatus = cudaSetDevice(0);
53 #if defined CRONO
54     eTimer *Tcpu = new eTimer();
55     eTimer *THtD = new eTimer();
56     eTimer *Tkernel = new eTimer();
57     eTimer *TDtH = new eTimer();
58
59     Tcpu->start();
60 #endif
61     // sumamos en la CPU
62     for (int i = 0; i < N; i++)
63         for (int j = 0; j < N; j++)
64             C[i*N + j] = alpha*A[i*N + j] + beta*B[i*N + j];
65 #if defined CRONO
66     Tcpu->stop();
67     Tcpu->report("CPU");
68 #endif
69
70     // imprimimos unos casos de prueba
71     for (int i = 0; i < 5; i++) printf("%lf ", C[i]);
72     printf("\n%lf\n", C[N*N-1]);
73     // para evitar un posterior falso test
74     memset(C, 0, N*N*sizeof(double));
75     for (int i = 0; i < 5; i++) printf("%lf ", C[i]);
76     printf("\n%lf\n", C[N*N - 1]);
77
78
```

Se crean los cronómetros

Cronometraje del cálculo en CPU

Casos de prueba: Se imprime los 5 primeros resultados y el último

Para asegurar futuros falsos test, borramos todos los resultado y nos aseguramos



# Programa principal(3)

```
77
78
79 // La parte de la GPU -----
80
81 // alamacen en la memoria de la GPU para A,B,C
82 double *dev_A, *dev_B, *dev_C;
83
84 // Reserva espacio para C
85 cudaStatus = cudaMalloc((void**)&dev_C, N*N* sizeof(double));
86 // Reserva espacio para B
87 cudaStatus = cudaMalloc((void**)&dev_B, N*N* sizeof(double));
88 // Reserva espacio para A
89 cudaStatus = cudaMalloc((void**)&dev_A, N*N* sizeof(double));
90
91 // inicio de proceso en GPU
92 #if defined CRONO
93     THtD->start();
94 #endif
95 // Copia la matriz A desde CPU a GPU
96 cudaStatus = cudaMemcpy(dev_A, A, N*N*sizeof(double), cudaMemcpyHostToDevice);
97 // Copia la matriz B desde CPU a GPU
98 cudaStatus = cudaMemcpy(dev_B, B, N*N*sizeof(double), cudaMemcpyHostToDevice);
99 #if defined CRONO
100     THtD->stop();
101     THtD->report("HostToDevice");
102
103     double AnchoBanda = 2*N*N*sizeof(double) / THtD->get();
104     printf("\nAncho de Banda (promedio): % 1f GBs\n",AnchoBanda*1.0e-9);
105
106     Tkernel->start();
107 #endif
```

Reserva de espacio en la GPU

Transferencia cronometrada de datos a la GPU

Cálculo del ancho de banda promedio



# Programa principal(4)

```
105
106 Tkernel->start();
107 #endif
108 // dimensiona el Grid de bloques y el bloque de hilos
109 dim3 Grid, Block;
110 Block.x = 32;
111 Block.y = 16;
112 Grid.x = N/Block.x;
113 Grid.y = N/Block.y;
114
115 // lanza el Kernel
116 sumaKernel <<< Grid, Block >>>(dev_C, dev_A, dev_B,alpha,beta);
117
118 // comprueba error en el lanzamiento del Kernel
119 cudaStatus = cudaGetLastError();
120 if (cudaStatus != cudaSuccess) {
121     fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
122     exit(1);
123 }
124 // espera hasta que finalice y si se han producido errores de ejecucion
125 cudaStatus = cudaDeviceSynchronize();
126 #if defined CRONO
127 Tkernel->stop();
128 Tkernel->report("Kernel");
129
130 TDtH->start();
131 #endif
132 // copia el resultado de C desde la GPU hasta la CPU
133 cudaStatus = cudaMemcpy(C, dev_C, N*N*sizeof(double), cudaMemcpyDeviceToHost);
134 #if defined CRONO
135 TDtH->stop();
136 TDtH->report("DeviceToHost");
137 #endif
138
```

Dimensionar el grid y los bloques  
Cada bloque de 32x16 =512 hilos

Recupera los resultados



# Programa principal(5)

Imprime los casos de prueba calculados por la GPU

```
138 // imprimimos unos casos de prueba
139 for (int i = 0; i < 5; i++) printf("%lf ", C[i]);
140 printf("\n%lf\n", C[N*N - 1]);
141
142
143 #if defined CRONO
144     delete Tcpu;
145     delete THtD;
146     delete Tkernel;
147     delete TDtH;
148 #endif
149 // Resetea la GPU para que Visual Studio recupere datos de traceado
150 cudaStatus = cudaDeviceReset();
151
152 return 0;
153 }
```

Destruye los cronómetros

Resetea el dispositivo.



Tiempo en la CPU

Casos de prueba en la CPU

Efectivamente borramos

Tiempo de transferencia  
de A y B a la GPU

Ancho de banda  
promedio

Tiempo de lanzamiento  
y ejecución del Kernel

Tiempo de recogida de resultados

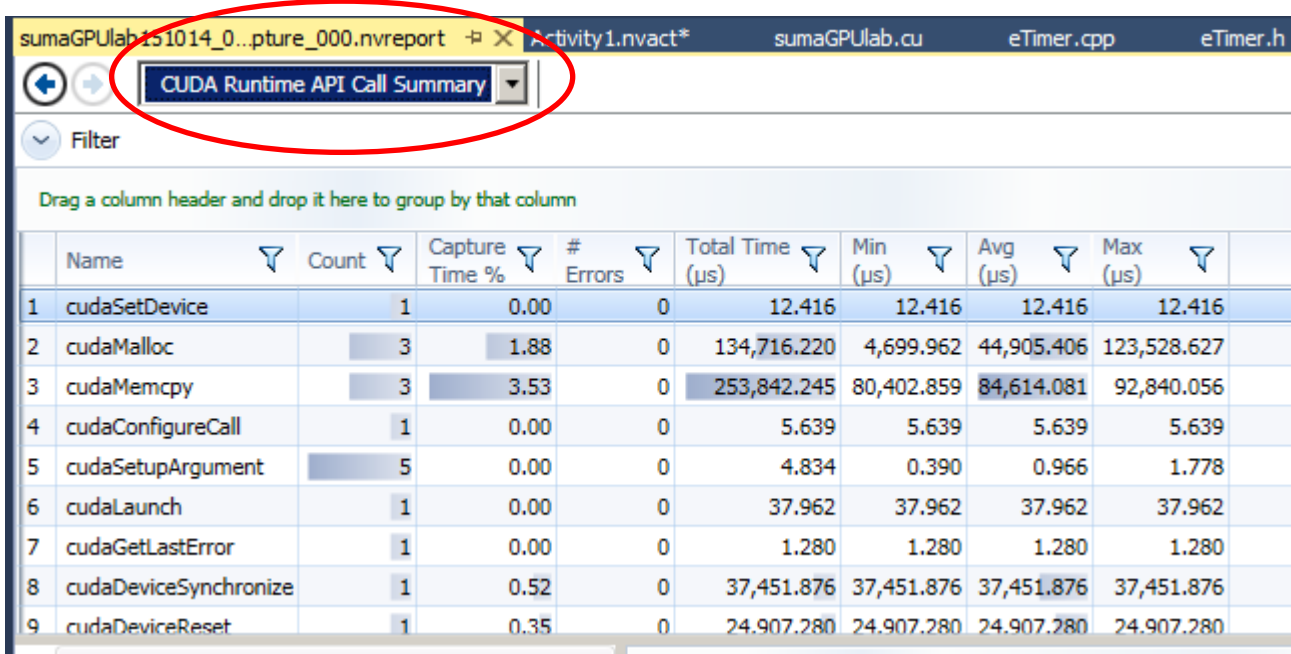
Casos de prueba en la GPU, exactamente igual,

Conclusión: lo que es estrictamente el calculo (Kernel) es bastante más rápido, tarda la tercera parte, pero si sumamos los tiempos de transferencia de datos y de resultados, entonces las prestaciones son inferiores. No se deben utilizar GPUs de forma generalizada.



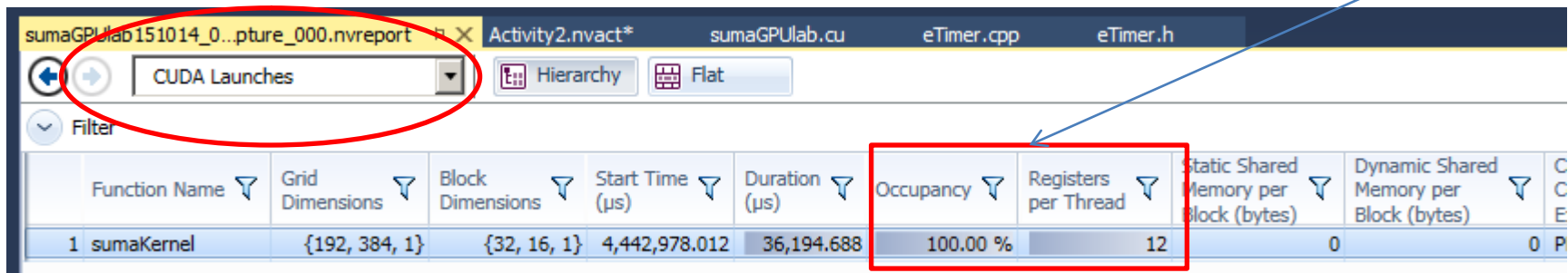
# Análisis NSIGHT

NSIGHT es una herramienta de NVIDIA integrada en Visual Studio que permite trazar la ejecución de programas CUDA. Necesita acceder físicamente a la GPU por ello se debe ejecutar en modo administrador. En los laboratorios no podemos hacerlo en la actualidad.



	Name	Count	Capture Time %	# Errors	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	cudaSetDevice	1	0.00	0	12.416	12.416	12.416	12.416
2	cudaMalloc	3	1.88	0	134,716.220	4,699.962	44,905.406	123,528.627
3	cudaMemcpy	3	3.53	0	253,842.245	80,402.859	84,614.081	92,840.056
4	cudaConfigureCall	1	0.00	0	5.639	5.639	5.639	5.639
5	cudaSetupArgument	5	0.00	0	4.834	0.390	0.966	1.778
6	cudaLaunch	1	0.00	0	37.962	37.962	37.962	37.962
7	cudaGetLastError	1	0.00	0	1.280	1.280	1.280	1.280
8	cudaDeviceSynchronize	1	0.52	0	37,451.876	37,451.876	37,451.876	37,451.876
9	cudaDeviceReset	1	0.35	0	24,907.280	24,907.280	24,907.280	24,907.280

12 registros usados, 100% Ocupancy

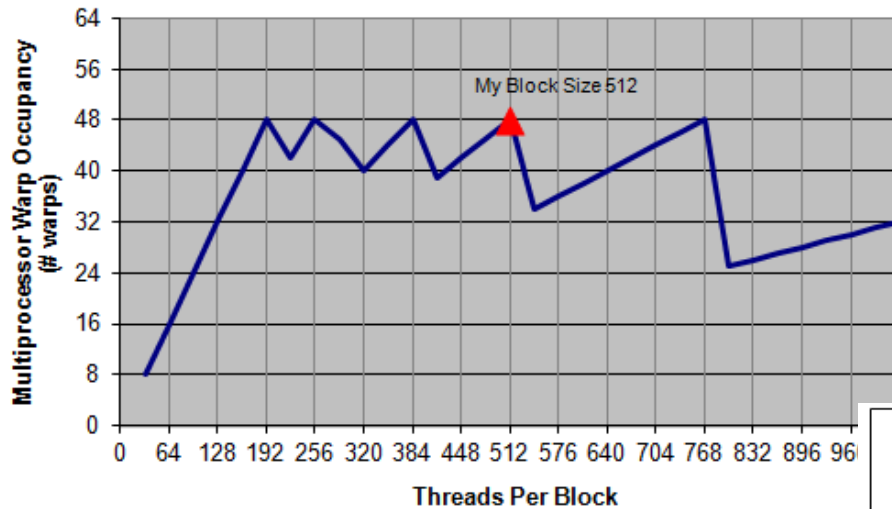


	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)	Dynamic Shared Memory per Block (bytes)	CCE
1	sumaKernel	{192, 384, 1}	{32, 16, 1}	4,442,978.012	36,194.688	100.00 %	12	0	0	P

Los datos de cronometraje proporcionados por NSIGHT están en la línea de los obtenidos en el programa por la clase eTimer

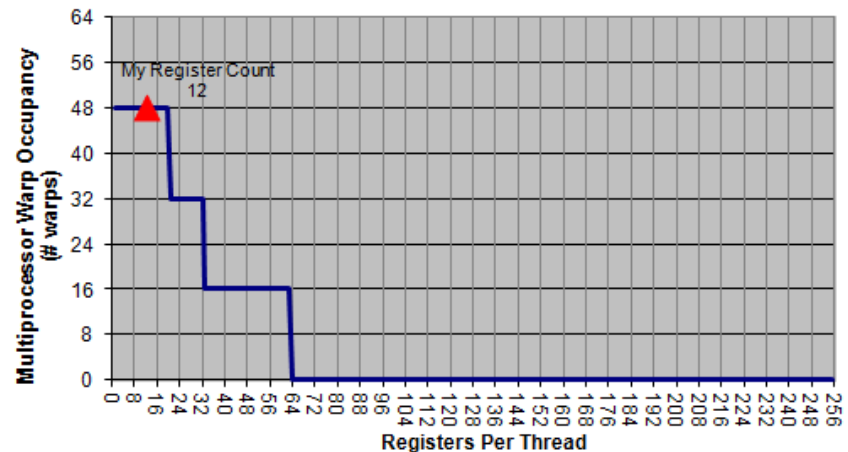
# Ocupancy Calculator

Impact of Varying Block Size



	A	B
4	Just follow steps 1, 2, and 3 below! (or click here for help)	
5		
6	1.) Select Compute Capability (click):	2,1
7	1.b) Select Shared Memory Size Config (bytes)	
8		
9		
10	2.) Enter your resource usage:	
11	Threads Per Block	512
12	Registers Per Thread	12
13	Shared Memory Per Block (bytes)	0
14		
15	(Don't edit anything below this line)	

Impact of Varying Register Count Per Thread



Hoja Excel disponible en la carpeta /tools

Buena Ocupancy y número de registros correcta. Un aumento de las variables internas del hilo podría producir infrautilización

# Tarea 3. Pinned memory

- Practicar con un ejemplo suministrado por el profesor con las diferencias entre memoria convencional o paginada y **pinned memory** (podemos traducirlo por memoria clavada/congelada/reservada).
- Probar con matrices de 4Kx4K y si la cantidad de memoria lo permite, con 6Kx6K



# Tarea 3. Pasos

1. Crear un proyecto CUDA
2. Eliminar el contenido de kernel.cu y copia el del fichero suministrado pinnedMem.cu. EL proceso también se puede realizar eliminando del proyecto el fichero kernel.cu y copiando a la carpeta del proyecto el fichero pinnedMem.cu. Posteriormente, incluir nuevos elementos existentes.
3. Copiar también la clase eTimer con los ficheros eTimer.h y eTimer.cpp
4. Ejecutar el programa con al versión de distribución que refiere al uso de memoria pinned. Si no se puede ejecutar, entonces reducir las dimensiones de las matrices, que por defecto es 4Kx4K.
5. Anotar los datos de tiempo en una tabla.
6. Editar el fichero y descomentar la línea de #undef. Ahora se usara memoria paginada.
7. Ejecutar y anotar los datos.
8. En la memoria incluir la tabla y comentar los resultados obtenidos.



```

C:\Windows\system32\cmd.exe

Pinned Memory

<CPU> Time <Min,Aver,Max>: 38.696 msec. 38.696 msec. 38.696 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

<HostToDevice> Time <Min,Aver,Max>: 41.608 msec. 41.608 msec. 41.608 msec.

<Kernel> Time <Min,Aver,Max>: 16.229 msec. 16.229 msec. 16.229 msec.

<DeviceToHost> Time <Min,Aver,Max>: 20.251 msec. 20.251 msec. 20.251 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

Presione una tecla para continuar . . . _

```

## Ejemplo con Pinned memory

	Name	Count	Capture Time %	# Errors	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	cudaSetDevice	1	0.00	0	40.670	40.670	40.670	40.670
2	cudaMallocHost	3	5.80	0	261,113.785	34,615.240	87,037.928	177,763.368
3	cudaMalloc	3	0.16	0	7,178.291	2,161.925	2,392.763	2,625.451
4	cudaMemcpy	3	1.38	0	62,223.109	20,319.772	20,741.036	20,960.975
5	cudaConfigureCall	1	0.00	0	11.594	11.594	11.594	11.594
6	cudaSetupArgument	3	0.00	0	4.090	0.359	1.363	2.125
7	cudaLaunch	1	0.00	0	40.037	40.037	40.037	40.037
8	cudaDeviceSynchronize	1	0.42	0	18,907.396	18,907.396	18,907.396	18,907.396
9	cudaFree	3	0.12	1	5,603.359	2.578	1,867.786	2,832.443
10	cudaDeviceReset	1	1.25	0	56,186.879	56,186.879	56,186.879	56,186.879
11	cudaFreeHost	3	0.97	3	43,733.865	1.150	14,577.955	43,730.555

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	miKernel	{128, 256, 1}	{32, 16, 1}	5,788,863.968	16,147.136	100.00 %	8	0



```

C:\Windows\system32\cmd.exe

Paged Memory

<CPU> Time <Min,Aver,Max>: 76.452 msec. 76.452 msec. 76.452 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

<HostToDevice> Time <Min,Aver,Max>: 76.332 msec. 76.332 msec. 76.332 msec.

<Kernel> Time <Min,Aver,Max>: 16.624 msec. 16.624 msec. 16.624 msec.

<DeviceToHost> Time <Min,Aver,Max>: 37.143 msec. 37.143 msec. 37.143 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

Presione una tecla para continuar . . .

```

Ejemplo con Paged memory

	Name	Count	Capture Time %	# Errors	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	cudaSetDevice	1	0.00	0	30.565	30.565	30.565	30.565
2	cudaMalloc	3	18.29	0	592,412.804	1,003.507	197,470.934	589,987.423
3	cudaMemcpy	3	3.42	0	110,884.696	36,734.900	36,961.565	37,131.186
4	cudaConfigureCall	1	0.00	0	11.976	11.976	11.976	11.976
5	cudaSetupArgument	3	0.00	0	3.893	0.434	1.297	2.001
6	cudaLaunch	1	0.00	0	45.824	45.824	45.824	45.824
7	cudaDeviceSynchronize	1	0.55	0	17,942.176	17,942.176	17,942.176	17,942.176
8	cudaFree	3	0.18	1	5,724.330	3.692	1,908.110	2,868.239
9	cudaDeviceReset	1	0.34	0	10,975.177	10,975.177	10,975.177	10,975.177

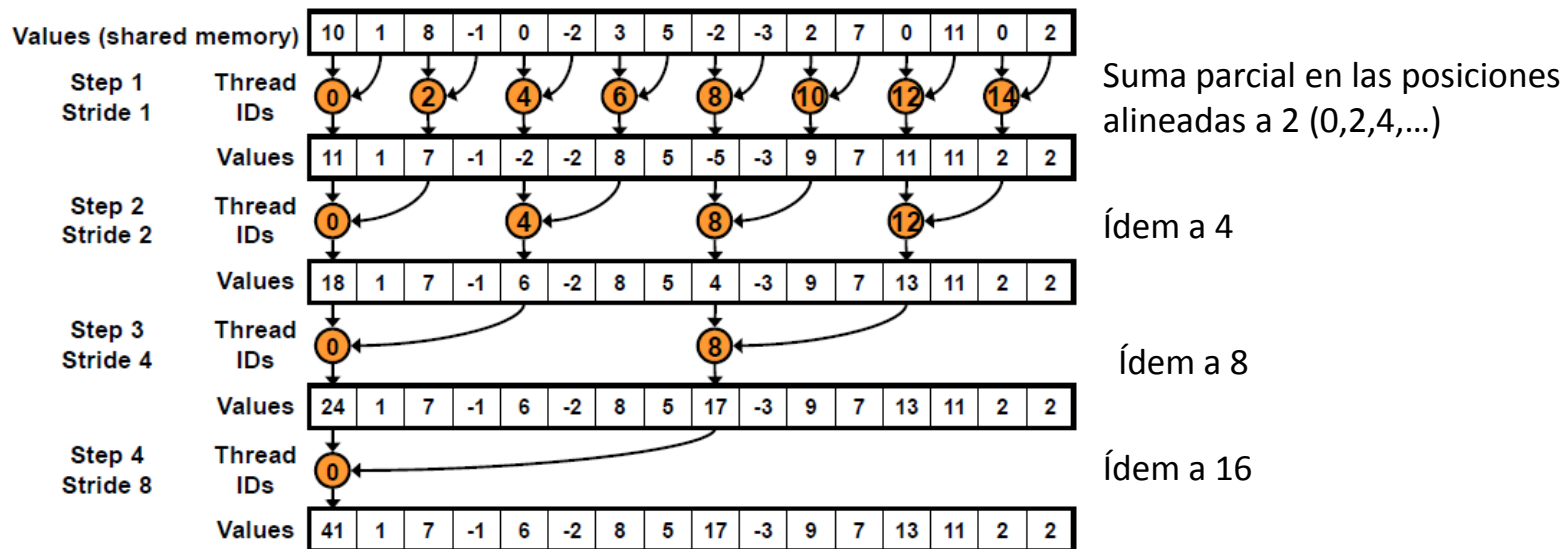


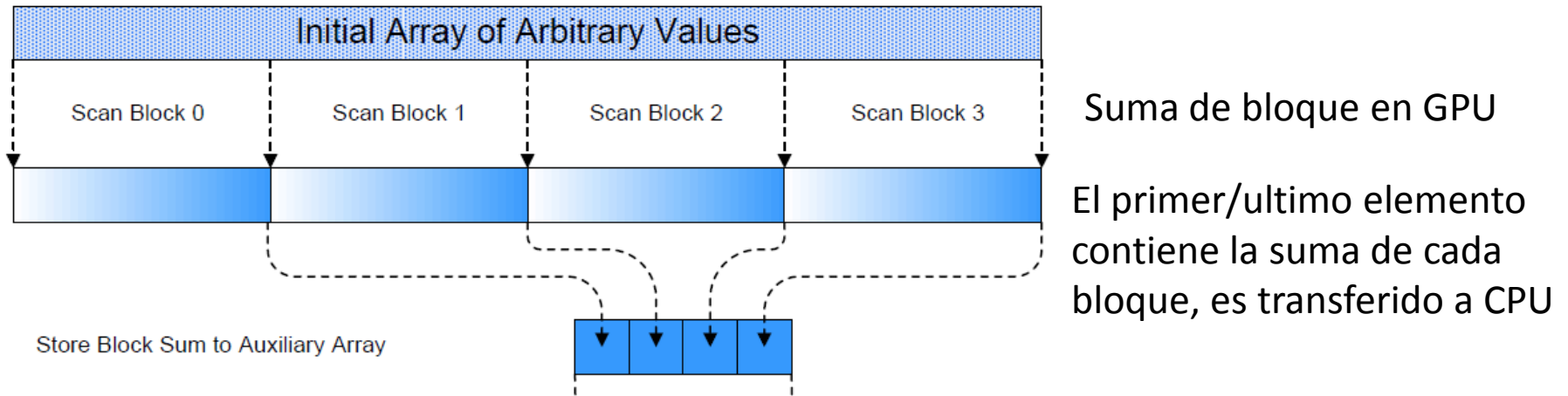
# (Opcional)Tarea 4. Reducción Paralela

Las operaciones de reducción son operaciones repetitivas, tales como la suma u otras, que se repiten sobre grandes vectores de datos. Por ejemplo sumar todos los datos de un vector. Estas operaciones pueden paralelizarse para reducir los costes computacionales.

La reducción de sumas es un algoritmo clásico de aplicación de GPU. Una exposición de muchas variantes, en orden de eficiencia creciente se expone en:

[https://docs.nvidia.com/cuda/samples/6\\_Advanced/reduction/doc/reduction.pdf](https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf)





Se suman secuencialmente en CPU las sumas de las sumas de bloques

La realización de sumas es más eficiente en la CPU si el número es bajo. Utilizaremos la GPU para sumar los números de bloques 1D (por ejemplo de 512 datos), pero grandes cantidades de bloques. El resultado de las sumas a su vez debería sumarse jerárquicamente, sin embargo, la forma más eficiente es transferir las mismas a la CPU y sumarla allí secuencialmente.





El código presentado introduce algunas variantes, uso de memoria compartida estática, suma de punto flotante en doble precisión y cambio en el orden de los parámetros.

```
/*
Ejemplo de suma masiva de números mediante reducción.
El algoritmo y sus mejoras se encuentra en:
https://docs.nvidia.com/cuda/samples/6\_Advanced/reduction/doc/reduction.pdf

Juan Méndez para MNC, juan.mendez@ulpgc.es
*/

#include <stdio>
#include <random>
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#pragma once
#ifdef __INTELLISENSE__ // __syncthreads() no esta incluido en el analizador visual
void __syncthreads();
#endif

#include "eTimer.h"

#define N 16*1024*1024
#define BLOCKSIZE 512
```

2<sup>24</sup>



## Programa principal

El alumno cambiará el lugar del cronómetro, para incluir solamente el Kernel.

```
delete Tcpu;
delete Tgpu;
status = cudaFree(dev_A);
status = cudaFree(dev_sum);
status = cudaFreeHost(host_A);
status = cudaFreeHost(host_sum);
status = cudaDeviceReset();
```

```
return 0;
```

```
int main(int argc, char *argv[])
{
    double *host_A, *dev_A;
    double suma, *host_sum, *dev_sum;
    int sizeA = N*sizeof(double);
    int nblock = N / BLOCKSIZE;
    int sizeSum = nblock*sizeof(double);

    std::default_random_engine generador;
    std::normal_distribution<double> distribucion(0.0, 1.0); // la suma debe ser casi 0

    eTimer *Tcpu = new eTimer();
    eTimer *Tgpu = new eTimer();

    cudaError_t status;
    status = cudaGetDevice(0);

    status = cudaMallocHost((void **)&host_A, sizeA); // rCPU y GPU
    status = cudaMalloc((void **)&dev_A, sizeA);
    status = cudaMallocHost((void **)&host_sum, sizeSum);
    status = cudaMalloc((void **)&dev_sum, sizeSum);

    for (int x = 0; x < N; x++) host_A[x] = distribucion(generador); // inicialización de datos

    Tcpu->start(); // suma secuencial en CPU
    suma = 0.0;
    for (int x = 0; x < N; x++) suma += host_A[x];
    Tcpu->stop();
    Tcpu->report("CPU");
    printf("Suma en CPU: %lf\n", suma);

    Tgpu->start(); // suma en GPU
    status = cudaMemcpy(dev_A, host_A, sizeA, cudaMemcpyHostToDevice); // transferencia de datos
    dim3 Block(BLOCKSIZE, 1, 1);
    dim3 Grid;
    Grid.x = nblock;
    Grid.y = Grid.z = 1;
    mireduccion1<<<Grid, Block>>>(dev_sum, dev_A, N); // Kernel
    status = cudaDeviceSynchronize();
    status = cudaMemcpy(host_sum, dev_sum, sizeSum, cudaMemcpyDeviceToHost); // recogida de sumas parciales
    suma = 0.0;
    for (int x = 0; x < nblock; x++) suma += host_sum[x]; // suma de las sumas
    Tgpu->stop();
    Tgpu->report("GPU");
    printf("Suma en GPU: %lf\n", suma);
}
```



```
C:\Windows\system32\cmd.exe

<CPU> Time <Min,Aver,Max>:  19.590 msec.   19.590 msec.   19.590 msec.
Suma en CPU: -581.039898

<GPU> Time <Min,Aver,Max>:  50.537 msec.   50.537 msec.   50.537 msec.
Suma en GPU: -581.039898
Presione una tecla para continuar . . . _
```

## Reduccion 1

```
// adaptación desde la referencia, pagina 9 con memoria compartida estatica
__global__ void mireduccion1(double *odata, const double *idata, const int n){

    __shared__ double buffer[BLOCKSIZE]; // memoria compartida estatica

    unsigned int thread = threadIdx.x;
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;

    buffer[thread] = idata[x];
    __syncthreads();

    for (unsigned int p = 1; p < BLOCKSIZE; p *= 2){
        if (thread % (2*p) == 0){
            buffer[thread] += buffer[thread + p];
        }
        __syncthreads();
    }

    if (thread==0) odata[blockIdx.x] = buffer[0];
}
```

Pero, el programa en GPU es más lento que en CPU!!!.

La referencia citada propone multitud de líneas de mejora. Pero la referencia se centra únicamente en el tiempo del Kernel, mientras que aquí hemos medido el tiempo total.



# ¿Qué hemos aprendido?

1. Programar la CPU+GPU para ejecutar funciones de la librería Runtime.
2. Ejecutar programas Kernel sencillos.
3. Todos los programas son mezclas de código que se ejecuta en la GPU (Kernels) y en la CPU (Librería Runtime).
4. Valorar el impacto de las transferencias de memoria en las prestaciones del conjunto CPU+GPU.
5. La conclusión más importante: La GPU acelera el compute, pero las transferencias degradan la aceleración. El tipo de aplicaciones en el que el resultado puede ser muy favorable es limitado: **No mucha transferencia de datos, gran cantidad de cálculo pero descomponible en trozos muy sencillos.**



# Qué debe entregar el alumno

- Cada alumno entregará en el Campus Virtual una memoria en PDF o Word en la que estará contenida una descripción del trabajo realizado, incluyendo descripción, el listado MATLAB o C de la actividad realizada y la captura de pantalla de las gráficas o imágenes generadas. Para autenticar las imágenes cuando sea posible el alumno incluirá su nombre en cada imagen mediante la función `title()`.
- En principio la tarea quedará abierta para su entrega hasta cierta fecha que se indicará.
- Se puede trabajar en grupo en el Laboratorio, pero la memoria elaborada y entregada será individual.



# Bibliografía

CUDA C Programming Guide, NVIDIA Corporation.

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#abstract>

Cuda Runtime API, NVIDIA Corporation,

<https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors, Morgan Kaufman, 2010.

[http://analog.nik.uni-obuda.hu/ParhuzamosProgramozasuHardver/02\\_GPGPU-Irodalom/02\\_GPGPU-](http://analog.nik.uni-obuda.hu/ParhuzamosProgramozasuHardver/02_GPGPU-Irodalom/02_GPGPU-Irodalom_MagyarBalint/Programming%20Massively%20Parallel%20Processors.pdf)

[Irodalom\\_MagyarBalint/Programming%20Massively%20Parallel%20Processors.pdf](http://analog.nik.uni-obuda.hu/ParhuzamosProgramozasuHardver/02_GPGPU-Irodalom_MagyarBalint/Programming%20Massively%20Parallel%20Processors.pdf)

J. Sanders, E. Kandrot, CUDA by Examples, Addison-Wesley, 2011

**J. Cheng, M. Grossman, T. MacKercher, Profesional CUDA C Programming, Wrox/Wiley, 2014**

