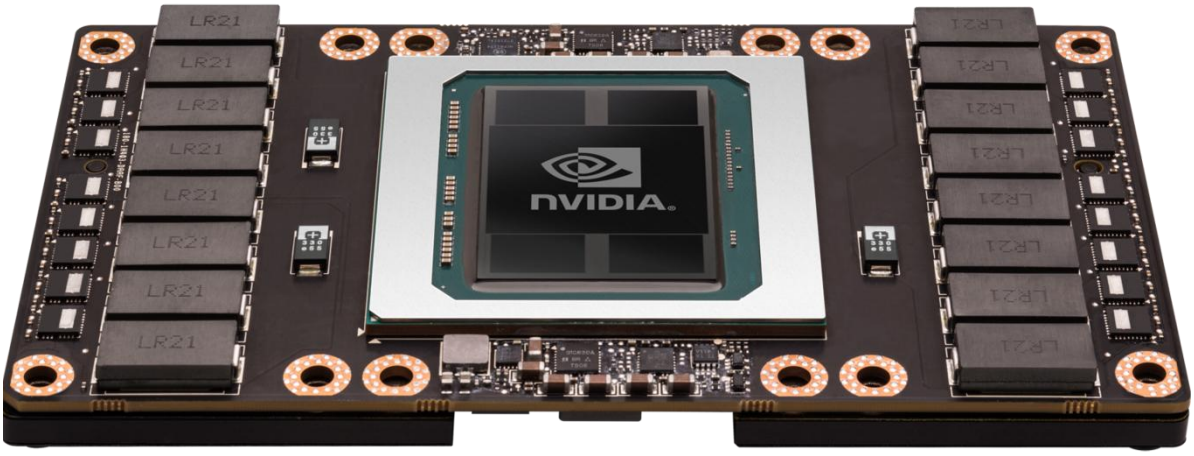
F-IV. Computhon - Calculo Normales



Álvaro Jover Álvarez

Héctor Ruiz Montesinos

Alejandro Pascual Gómez

Jordi Amorós Moreno

Índice

Introducción

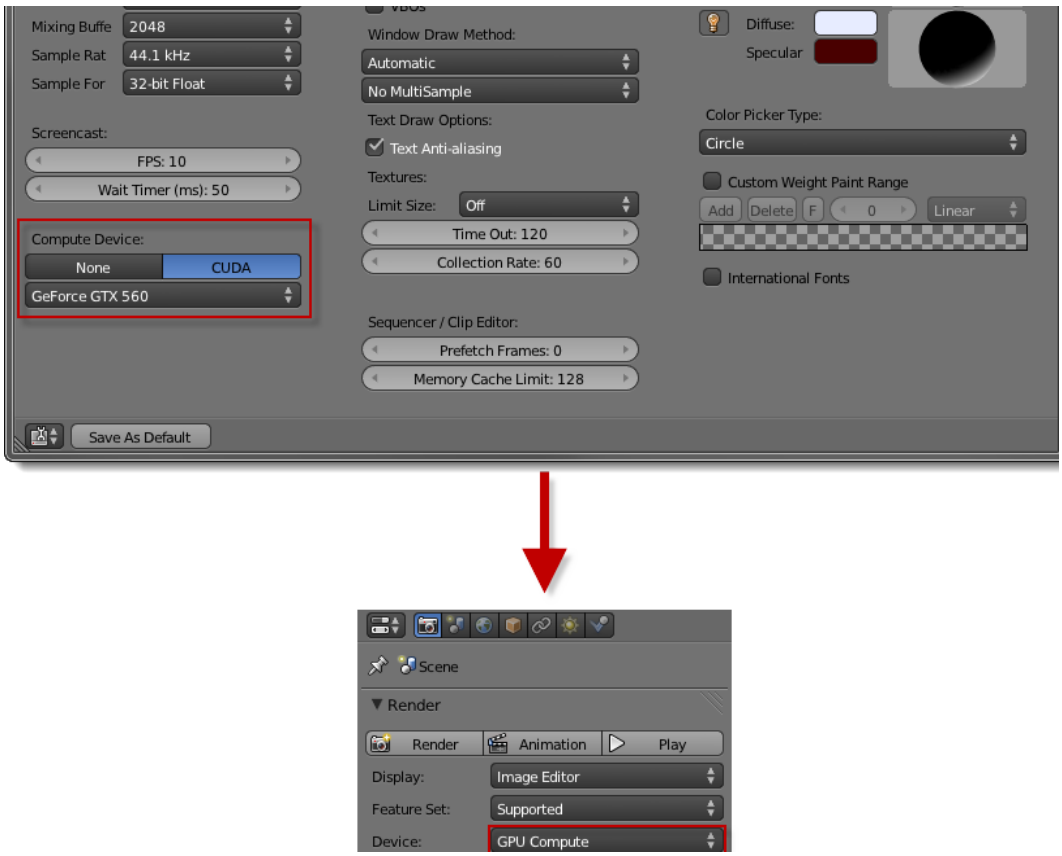
En ésta fase de la práctica se implementara en **CUDA** la obtención de los vectores normales correspondientes a un conjunto amplio de puntos 3D en función de los **vecinos** más cercanos.

El **cálculo de normales** es un aspecto muy importante en el campo de los **gráficos** por **computador**, usándose en ámbitos de **modelaje**, entre otras muchas cosas. Un ejemplo práctico del uso de estos seria la **retopología**: Un modelador 3D crea una escultura con un millón de polígonos, lo cual **no es eficiente** para el renderizado en tiempo real, por lo tanto se hace el cálculo de las normales de ese objeto de **alto poligonaje** y se crea una versión de **bajo poligonaje** de ese objeto y se **aplican** las **normales** del otro obteniendo así el **mismo resultado**, un objeto con un millón de polígonos lucirá igual que uno con veinte mil.

CUDA

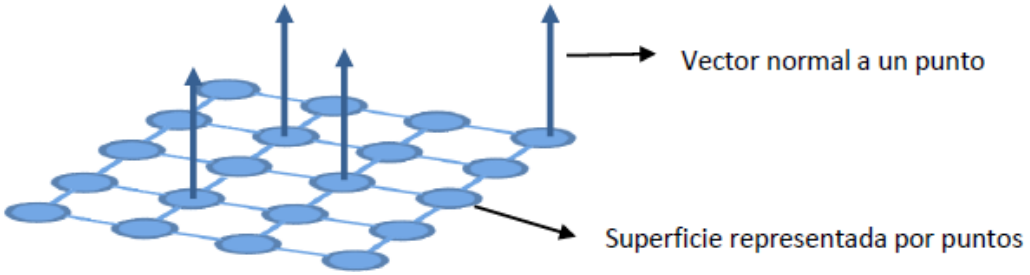
CUDA, Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo), es una arquitectura de **cálculo paralelo** de **NVIDIA** que hace provecho de la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un **incremento** extraordinario del **rendimiento** del sistema.

Hoy en día podemos ver CUDA en diversos programas de edición gráfica y modelaje 3D, como **Blender**:



Calculo de normales

Se considera que los puntos de la superficie están organizados en una estructura de **malla** que permite determinar la **vecindad** de un **punto dado**.



Para calcular la **normal** de un **punto**, se calcula la media de las **normales** de las **rectas** que forma el punto con cada uno de sus vecinos. El **algoritmo** podría ser el siguiente (extraído de la práctica):

```
// P es el punto del que tu quieres calcular la normal
para toda P de los puntos de la superficie
    // En vectorMedio almacenaremos la suma de las normales con las vecindades
    vectorMedio(P)=(0,0,0);
    // Para cada vecino de P al cual llamamos Q
    para todo punto Q de la vecindad de P
        // Se calcula la recta de P a Q
        recta=CalcularRecta(P,Q);
        // Calculamos la normal de esa recta, y se anyade al vector medio
        vectorMedio=vectorMedio+CalculaNormal(recta);
    fin para
    // Una vez finalizado ese bucle se divide el vector medio entre el numero de vecinos que habia
    // Obteniendo asi la normal
    vectorMedio(P)=vectorMedio(P)/numPuntosVecindad;
fin para
```

- **Implementación en C++**

Junto al **enunciado** del ejercicio, hay adjunto un fichero en el cual esta implementado el algoritmo en **C++** el cual vamos a comentar brevemente, al igual que el uso de las **funciones** que hace:

```
for (int u = 0; u<S.UPoints; u++)           // Recorrido de todos los puntos de la superficie
{
    for (int v = 0; v<S.VPoints; v++)
    {
        normal.x = 0;
        normal.y = 0;
        normal.z = 0;
        numDir = 0;
        for (int nv = 0; nv < 8; nv++)  // Para los puntos de la vecindad
        {
            vV = v + vecindadV[nv];//U: coordenada X
            vU = u + vecindadU[nv];//V: coordenada Y

            //Comprueba que el punto no se sale de los limites de la matriz
            //S.Buffer: es la malla, con todos los puntos
            //S.VPoints y S.UPoints son el tamanyo de la malla (VxU)
```

```
        //Calcula la coordenada correspondiente con la distancia que hay
        //entre el punto actual (v,u) y el vencino (vV, uU)((este va cambiando para
        //calcular todos los vecinos))
        if (vV >= 0 && vU >= 0 && vV<S.VPoints && vU<S.UPoints)
        {
            direct1.x = S.Buffer[v][u].x - S.Buffer[vV][vU].x;
            direct1.y = S.Buffer[v][u].y - S.Buffer[vV][vU].y;
            direct1.z = S.Buffer[v][u].z - S.Buffer[vV][vU].z;
            oKdir1 = 1;
        }
        //Si se sale de los limites, las coordenadas valen 0
        //para no alterar los datos
        else
        {
            direct1.x = 0.0;
            direct1.y = 0.0;
            direct1.z = 0.0;
            oKdir1 = 0;
        }

        //Ahora cogemos un segundo punto
        vV = v + vecindadV[nv + 1];
        vU = v + vecindadU[nv + 1];

        //Igual que con el anterior
        if (vV >= 0 && vU >= 0 && vV<S.VPoints && vU<S.UPoints)
        {
            direct2.x = S.Buffer[v][u].x - S.Buffer[vV][vU].x;
            direct2.y = S.Buffer[v][u].y - S.Buffer[vV][vU].y;
            direct2.z = S.Buffer[v][u].z - S.Buffer[vV][vU].z;
            oKdir2 = 1;
        }
        else
        {
            direct2.x = 0.0;
            direct2.y = 0.0;
            direct2.z = 0.0;
            oKdir2 = 0;
        }

        //Si han existido los dos puntos, entonces calculamos la normal
        //de la recta que une los dos puntos

        //Se ahorra el calculo de un tercer punto para no calcular la normal
        //de un plano (es mas costoso en cuanto a calculos)
        if (oKdir1 == 1 && oKdir2 == 1)
        {
            normal.x += direct1.y*direct2.z - direct1.z*direct2.y;
            normal.y += direct1.x*direct2.z - direct1.z*direct2.x;
            normal.z += direct1.x*direct2.y - direct1.y*direct2.x;
            numDir++;
        }
    }
```

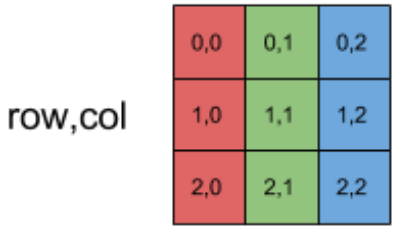
```
            //Vectores con los puntos del fichero, almacenados en variable global
            //en el archivo .h estan, tanto los de CPU como los de GPU

            //Aqui hay que almacenar los resultados
            NormalUCPU[cont] = normal.x / (float)numDir;
            NormalVCPU[cont] = normal.y / (float)numDir;
            NormalWCPU[cont] = normal.z / (float)numDir;
            cont++;
        }
    }

    return OKCALC;                                  // Simulación CORRECTA
```

- **Implementación en CUDA**

El algoritmo de **C++** ya venía resuelto junto al enunciado, sin embargo, nuestra tarea consistía en pasar ese código **C++** a **CUDA**, primeramente empezamos trazando unos planteamientos iniciales:

```
//Problema para computar el algoritmo teniendo la malla 3D aplanada en un vector de 1 Dimension

        /* ------------> S.UPoints (u)
        | 0  3  6  9
        | 1  4  7  10
        | 2  5  8  11
        v
    S.VPoints (v)

    Esto pasado a h_Buffer (unidimensional) queda:

    ------------------------> S.UPoints * S.Vpoints (id)
    0 1 2 3 4 5 6 7 8 9 10 11

    Obtendremos los dos indices 'v' y 'u' a partir del indice unidimensional:

        v = id % S.VPoints
        u = id / S.VPoints

    */
```

Básicamente este planteamiento consiste en aplanar el array bidimensional en un array de una sola dimensión para trabajar en un formato denominado **Column Major**.



Vamos a iniciar el **algoritmo** reservando memoria en la **GPU** para todos los datos que vamos a necesitar una vez hagamos la invocación al **kernel**. Haremos uso de **cudaMalloc** para la reserva de **memoria** y de **cudaMemcpy** para la copia de datos. En este caso haremos sizeof (int)*9 porque es un vector de enteros de **tamaño** 9, las posiciones de memoria pasadas a **cudaMalloc** será donde se realice esta reserva.

- **cudaMalloc(dirección memoria, tamaño de la reserva en bytes)**

A **cudaMemcpy** le pasaremos los dos punteros, origen y destino, siendo origen **vecindadU** (el de la **CPU**) y el destino **d_vU**, que sería nuestro puntero. También le pasamos el tamaño, el mismo que el anterior, y el tipo de copia, que en esta caso es **Host to Device**, de la **CPU a la GPU**.

- **cudaMemcpy(device, host, tamaño en bytes, tipo de copia)**

```
float *d_NormalVGPU;
float *d_NormalUGPU;
float *d_NormalWGPU;

//Allocate on device memory for 3D Surface and the 3 normal vectors (result)
cudaMalloc(&d_Buffer, U*V * sizeof(TPoint3D));
cudaMalloc(&d_NormalVGPU, sizeof(float)*U*V);
cudaMalloc(&d_NormalUGPU, sizeof(float)*U*V);
cudaMalloc(&d_NormalWGPU, sizeof(float)*U*V);

//Copy to device 3D Surface
cudaMemcpy(d_Buffer, h_Buffer, sizeof(TPoint3D)* U*V, cudaMemcpyHostToDevice);

time = getTime();

getNormal<<< U*V/512 + 1, 512 >>>(d_Buffer, d_NormalUGPU, d_NormalVGPU, d_NormalWGPU, U, V);

end_time = getTime();
noTransfer = (end_time - time);

cudaMemcpy(NormalVGPU, d_NormalVGPU, U*V * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(NormalUGPU, d_NormalUGPU, U*V * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(NormalWGPU, d_NormalWGPU, U*V * sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(d_Buffer);
cudaFree(d_NormalVGPU);
cudaFree(d_NormalUGPU);
cudaFree(d_NormalWGPU);

return OKCALC;
```

En getNormal haremos la invocación al **kernel** en el cual sucede lo siguiente:

- Hemos establecido como parámetros (U*V/512) + 1 para el **numero de bloques** (siendo U y V la altura y la anchura de la malla respectivamente) y 512 para el numero de hilos por cada bloque.
- Pasamos también los **punteros** almacenados en la memoria de la GPU (d_)
- Además también el **tamaño** de la **malla** con U y V

- **Implementación del kernel**

El siguiente fragmento de código comentado muestra como hemos logrado implementar el **algoritmo** en sí:

```
__global__ void getNormal(TPoint3D *d_Buffer, float *d_NormalUGPU, float *d_NormalVGPU, float *d_NormalWGPU, int U, int V) {

    // Calculo de una id unica para cada thread
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    // Mientras la id esté dentro de los limites de nuestra matriz...
    if (id < U*V) {
        // Inicializamos las vecindades y otros parametros que usaremos mas adelante
        int d_vU[9]={-1,0,1,1,1,0,-1,-1,-1};
        int d_vV[9]={-1,-1,-1,0,1,1,1,0,-1};

        int vecindad, oKdir1, oKdir2, numDir = 0, v, u, vV, vU;

        TPoint3D normal, direct1, direct2;
        normal.x = 0;
        normal.y = 0;
        normal.z = 0;
```

```
        // Para cada punto de la malla calculamos las rectas con sus vecindades y sus normales asociadas
        for (int nv = 0; nv < 8 ; nv ++) {

            // Ya que estamos utilizando un "column major" vamos a calcular sus indices a partir de la id y el
            // tamaño de la columna
            v = id % V;      //get row index
            u = id / V;      //get column index


            // Calculo del punto de la vecindad (row, column)
            vV=v+d_vV[nv];
            vU=u+d_vU[nv];

            // Calculo del id correspondiente asociado a los indices de la vecindad
            vecindad = vU * V + vV;

            // Si el punto se encuentra dentro de los limites de nuestra matriz...
            if (vV >= 0 && vU >=0 && vV<V && vU<U) {
                // Calculamos la recta que une los dos puntos
                direct1.x = d_Buffer[id].x - d_Buffer[vecindad].x;
                direct1.y = d_Buffer[id].y - d_Buffer[vecindad].y;
                direct1.z = d_Buffer[id].z - d_Buffer[vecindad].z;
                oKdir1=1;
            }else
            {
                direct1.x=0.0;
                direct1.y=0.0;
                direct1.z=0.0;
                oKdir1=0;
            }

            vV=v+d_vV[nv+1];
            vU=v+d_vU[nv+1];

            vecindad = vU * V + vV;
```

```
            // Hacemos lo mismo para la siguiente vecindad
            if (vV >= 0 && vU >=0 && vV<V && vU<U) {
                direct2.x = d_Buffer[id].x - d_Buffer[vecindad].x;
                direct2.y = d_Buffer[id].y - d_Buffer[vecindad].y;
                direct2.z = d_Buffer[id].z - d_Buffer[vecindad].z;
                oKdir2=1;
            }else
            {
                direct2.x=0.0;
                direct2.y=0.0;
                direct2.z=0.0;
                oKdir2=0;
            }

            // Si las dos vecindades existen y se han calculado sus rectas, calculamos su normal asociada a estas dos
            if (oKdir1 == 1 && oKdir2 == 1) {
                normal.x += direct1.y * direct2.z - direct1.z * direct2.y;
                normal.y += direct1.x * direct2.z - direct1.z * direct2.x;
                normal.z += direct1.x * direct2.y - direct1.y * direct2.x;
                numDir++;
            }
        }

        // Finalmente añadimos al vector resultado la media de todas las normales del punto
        d_NormalUGPU[id] = normal.x/(float)numDir;
        d_NormalVGPU[id] = normal.y/(float)numDir;
        d_NormalWGPU[id] = normal.z/(float)numDir;
    }
}
```

Una vez acabado el algoritmo, copiamos la memoria de la GPU al HOST y liberamos toda la memoria de la GPU (device), empleando cudaFree, su uso:

- cudaFree(puntero)

Evaluación resultados

Ejemplo de ejecución:

```
C:\Users\byflo\onedrive\documents\visual studio 2012\Projects\calculoNormales\Debug\calculoNormales.exe
Cßlculo de las normales de la superficie...
Cßlculo correcto!
Tiempo ejecuci¾n GPU : 0.196103s
Tiempo de ejecuci¾n en la CPU : 0.002870s
Se ha conseguido un factor de aceleraci¾n 0.014636x utilizando CUDA
Se ha conseguido un factor de aceleraci¾n 56.934175x utilizando CUDA (sin tener en cuenta transferencia de datos)
```

Basándonos en **10 ejecuciones** con el **mismo número** de **elementos** para sacar una media traza común en **CPU** y **GPU**, hemos logrado los siguientes resultados:

Numero Ejecución	Tiempo CPU	Tiempo GPU	Factor de aceleración CUDA sin transferencia de datos
1	**0.002870**	**0.196103**	**56.934175x**
2	**0.003007**	**0.160913**	**69.664390x**
3	**0.003132**	**0.154619**	**69.644303x**
4	**0.002971**	**0.132280**	**67.869002x**
5	**0.003047**	**0.141754**	**66.854299x**
6	**0.002946**	**0.139669**	**106.086876x**
7	**0.002916**	**0.134057**	**66.627621x**
8	**0.003287**	**0.143711**	**57.305253x**
9	**0.003093**	**0.140303**	**73.185796x**
10	**0.003158**	**0.130867**	**75.818836x**

A continuación mostraremos las **características** de la **CPU** y la **GPU** usada:

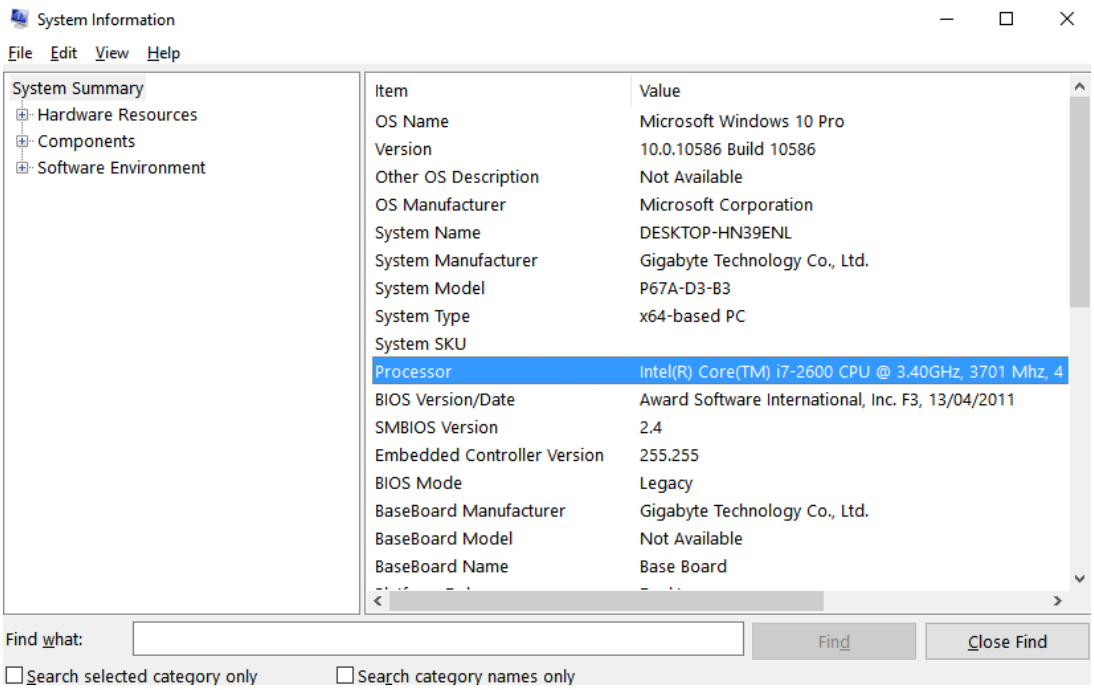
- Para mostrar la GPU utilizaremos este fragmento de código

```
int device;
cudaGetDevice(&device);

struct cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, device);

printf("\nDevice properties: \n");

printf("  Device name: %s\n", prop.name);
printf("  Memory Clock Rate (KHz): %d\n",
       prop.memoryClockRate);
printf("  Memory Bus Width (bits): %d\n",
       prop.memoryBusWidth);
printf("  Peak Memory Bandwidth (GB/s): %f\n",
       2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
printf("  Clock Rate (KHz): %i\n", prop.clockRate);
printf("  Total Global Memory (MB): %i\n\n", prop.totalGlobalMem/1048576);
```

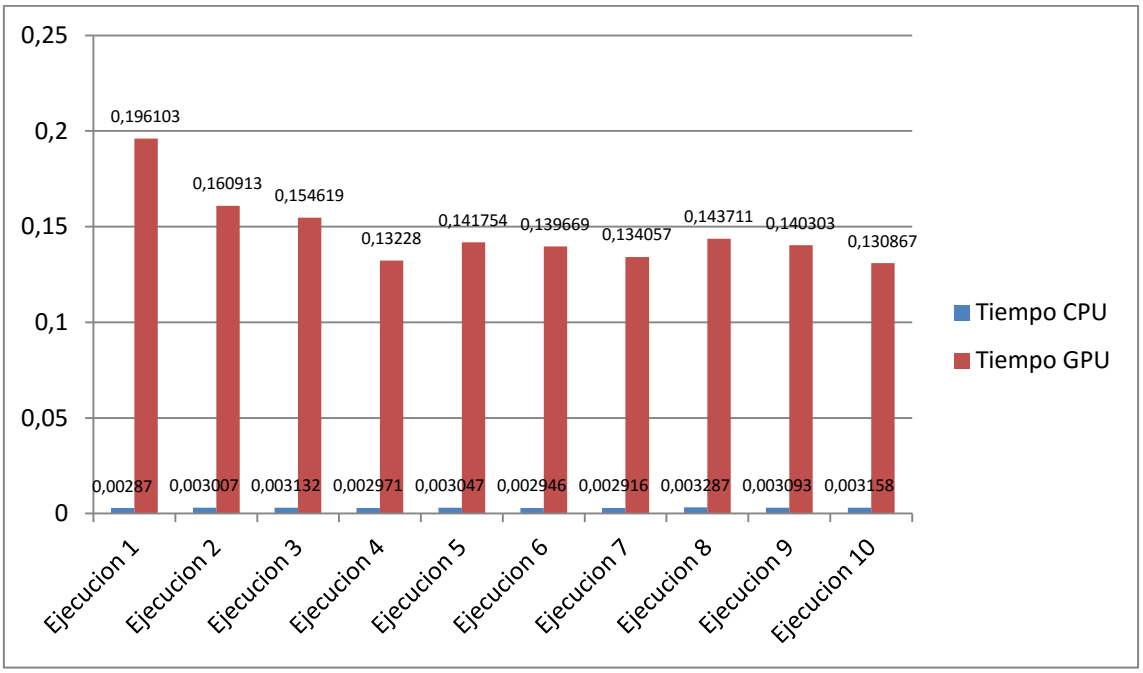
El cual nos dará la siguiente salida

```
Device properties:
  Device name: GeForce GTX 960
  Memory Clock Rate (KHz): 3505000
  Memory Bus Width (bits): 128
  Peak Memory Bandwidth (GB/s): 112.160000
  Clock Rate (KHz): 1253000
  Total Global Memory (MB): 2048
```

- Para la CPU simplemente tenemos que mirar las **especificaciones** del sistema



Aquí la **grafica** basada en los resultados de las **ejecuciones anteriores**:



Ante estos resultados y teniendo en cuenta que **ignorando transferencias de datos** la aceleración de **GPU** es **muy elevada**, de **media +60** (ver tabla de arriba), podemos deducir que la mayor parte del tiempo de cálculo del algoritmo se pierde en la transferencia de los datos de la **CPU** a la **GPU**.

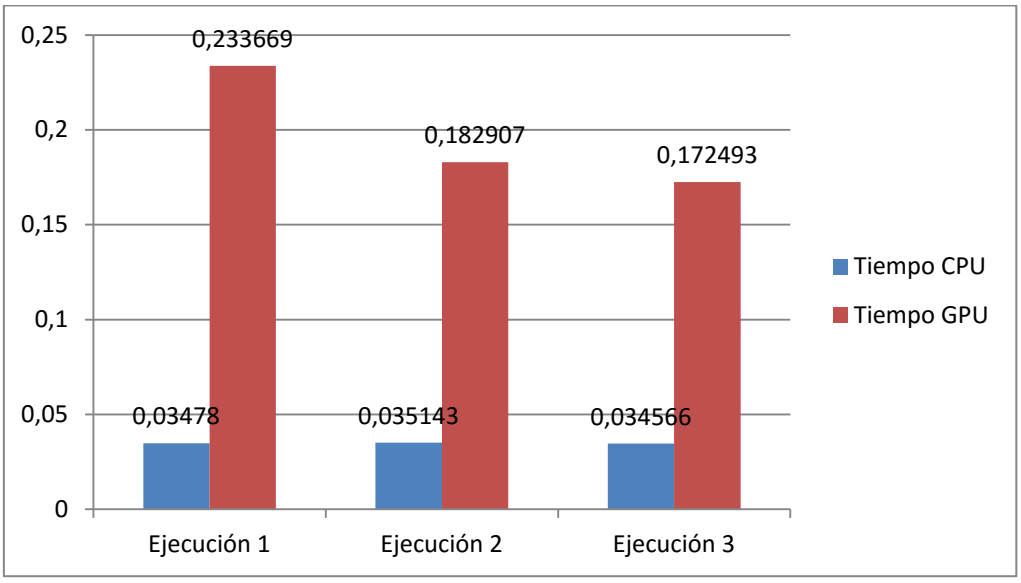
Asumimos también que para tallas de problema mucho mayores, el tiempo de cálculo en la CPU crecerá de manera exponencial mientras que los tiempos de transferencia de datos lo harán de forma lineal. Deducimos por tanto que cuanto más crezca la talla del problema, mejores aceleraciones obtendremos de manera global.

Tamaño de la malla 400 x 460

Numero Ejecución	Tiempo CPU	Tiempo GPU	Factor de aceleración CUDA sin transferencia de datos
1	**0.034780**	**0.233669**	**556.608682x**
2	**0.035143**	**0.182907**	**657.740122x**
3	**0.034566**	**0.172493**	**673.588316x**

El factor de aceleración medio teniendo en cuenta data transfer es de aprox. 0,2x, sigue siendo mucho más lento en la GPU, pero podemos comprobar que con una talla más grande hemos obtenido mejores resultados.

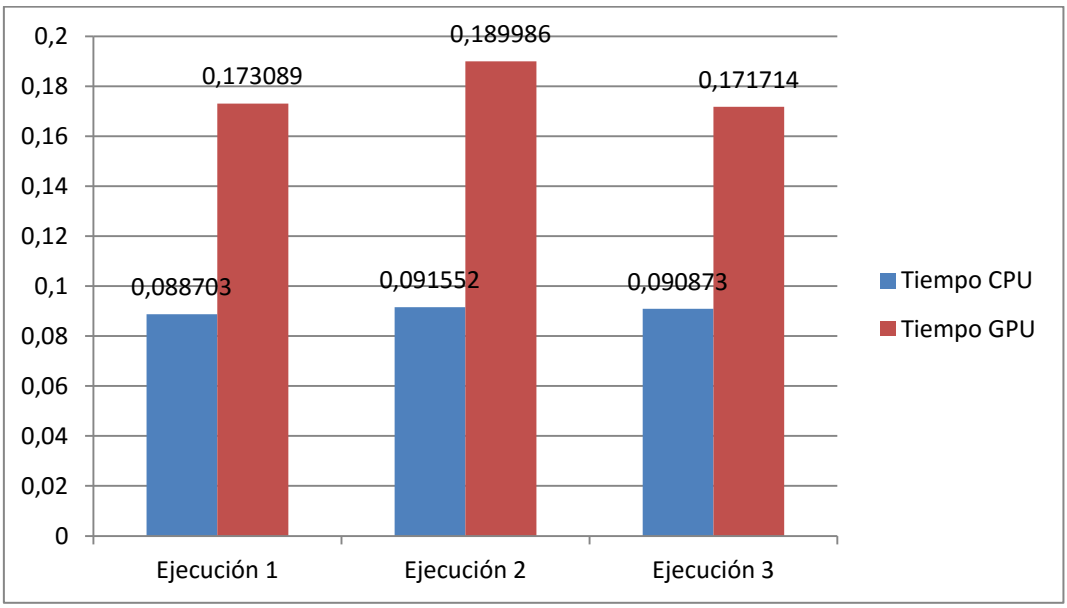
- **Grafica malla 400 x 460**



Tamaño de la malla 610 x 730

Numero Ejecución	Tiempo CPU	Tiempo GPU	Factor de aceleración CUDA sin transferencia de datos
1	**0.088703**	**0.173089**	**1447.566931x**
2	**0.091552**	**0.189986**	**2120.936754x**
3	**0.090873**	**0.171714**	**1440.407102x**

- **Grafica malla 610 x 730**

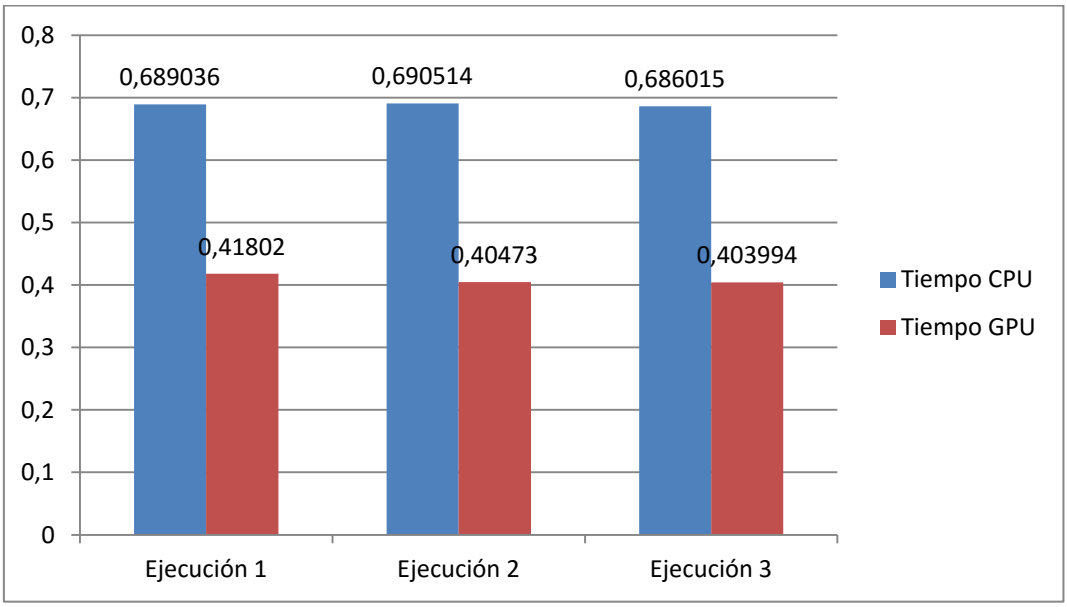


A golpe de vista podemos observar que el factor de aceleración medio teniendo en cuenta transferencias de datos de CPU es aprox. de 0,5x.

Tamaño de la malla 1700 x 1900

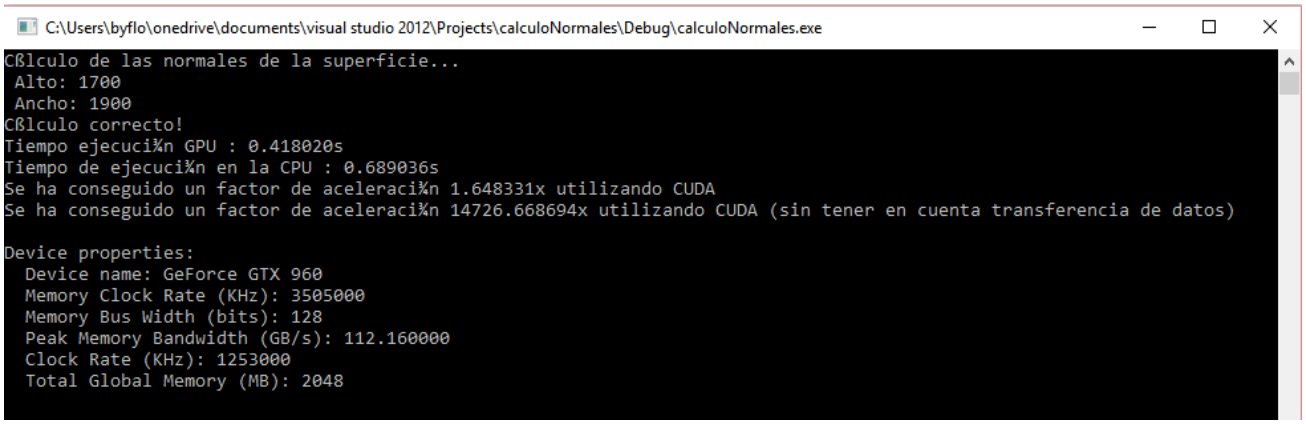
Numero Ejecución	Tiempo CPU	Tiempo GPU	Factor de aceleración CUDA sin transferencia de datos
1	**0.689036**	**0.418020**	**14726.668694x**
2	**0.690514**	**0.404730**	**8258.238531x**
3	**0.686015**	**0.403994**	**10770.740658x**

- **Grafica malla 1700 x 1900**



Como conclusión general y como habíamos deducido anteriormente, para tallas más elevadas, la GPU a pesar de tener un tiempo elevado de transferencia de datos consigue mejores tiempos gracias al multithreading.

Ejecución del algoritmo sobre una malla de 1700 x 1900



Para crear estas mallas de mayor tamaño hemos creado un pequeño programa que genera aleatoriamente puntos 3D para la malla.

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

float RandomFloat(float a, float b) {
    float random = ((float) rand()) / (float) RAND_MAX;
    float diff = b - a;
    float r = random * diff;
    return a + r;
}

int main() {

    int U, V;

    cin >> U;
    cin.get();

    cin >> V;
    cin.get();

    ofstream fout("test.for");

    fout << "Ancho=" << U << endl;
    fout << "Alto=" << V<< endl;

    for (unsigned i = 1; i <= U*V; i++) {
        fout << setprecision(3) << fixed << "    " << RandomFloat(0, 30 + i) << "    "
            << RandomFloat(0, 20 + i) << "   " << RandomFloat(0, 10 + i) << endl;
    }

    fout.close();

    return 0;
}
```

Referencias

- https://devtalk.nvidia.com/
- http://stackoverflow.com/
- https://bluewaters.ncsa.illinois.edu/documents/10157/63094/NCSA02a-window-example-cpp-08-14-13.pdf
- Pdf's de prácticas
- http://www.nvidia.es/object/cuda-parallel-computing-es.html
- https://es.wikipedia.org/wiki/CUDA
- http://www.tomshardware.com/
- http://www.pcworld.com/
- http://dmi.uib.es/~josemaria/files/CUDA-v36-ESP-para-UIB.pdf