Práctica 1. Implementación de algoritmos paralelos de datos en GPU usando CUDA

Jose Luis Gallego Peña

1. Implementación en CUDA del Algoritmo de Floyd

Para su ejecución, recolección de resultados y representación gráfica se usa el script **ejecuciones.sh** que borra anteriores resultados, compila el programa y lo ejecuta varias veces para distintos tamaños de bloque (8x8, 16x16, 32x32), y dentro de cada tamaño de bloque se analizan diferentes tamaños del problema (400, 1000, 1400 y 2000). Por último llama al script **graficas.gp**, que coge los datos de los archivos de resultados y los representa en dos gráficas usando la utilidad **gnuplot**: una para los tiempos y otra para las ganancias, ambas exportadas a un archivo PNG. Todos los archivos de datos y las gráficas se encuentran en la carpeta **resultados**.

1.1. Modificación grid bidimensional

Se ha modificado la implementación CUDA del algoritmo de Floyd para que las hebras se organicen como un grid bidimensional de bloques cuadrados de hebras bidimensionales.

El kernel calcula los índices de cada hebra para filas y columnas, a partir de ellos calcula el índice global de todas las hebras y bloques y finalmente a partir de ello calcula los nuevos índices con los cuales acceder a la matriz. Se comprueba que puedan operar en la matriz y entonces se realizan los cálculos del algoritmo para la iteración k.

```
// Kernel para resolver el algoritmo de Floyd con bloques bidimensionales
__global__ void floyd_kernel_2d(int * M, const int nverts, const int k) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;  // Índice de filas de hebra
    int i = blockIdx.y * blockDim.y + threadIdx.y;  // Índice de columnas de
hebra

    int ij = i * nverts + j;    // Índice global del elemento en la matriz
    // Para poder acceder a los valores dentro de la matriz por fila y columna
    int i2 = ij / nverts;       // Fila correspondiente al índice global
    int j2 = ij - i2 * nverts;  // Columna correspondiente al índice global

    if (i2 < nverts && j2 < nverts) {
        int Mij = M[i2 * nverts + j2];

        // Evitar los 0 de la matriz (evitar el mismo vertice)
        if (i != j && i != k && j != k) {
            int Mikj = M[i2 * nverts + k] + M[k * nverts + j2];
            Mij = (Mij > Mikj) ? Mikj : Mij;
            M[ij] = Mij;
        }
    }
}
```

La llamada a la función se realiza en un bucle for para las k iteraciones. El número de hebras por bloque se saca del argumento blocksize del programa, multiplicándolo por si mismo puesto que es bidimensional. En cuanto al número de bloques por grid se hace de la misma manera que en el ejemplo de matrices en cuda, teniendo en cuenta para cada dimensión la longitud de esta.

```
for (int k = 0 ; k < niters ; k++) {
        // Tamaño del bloque (número de hebras por bloque)
        dim3 threadsPerBlock(blocksize, blocksize);
        // Tamaño del grid (número de bloques por grid)
        int tamx = ceil((float) (nverts) / threadsPerBlock.x);
        int tamy = ceil((float) (nverts) / threadsPerBlock.y);
        dim3 blocksPerGrid(tamx, tamy);

        floyd_kernel_2d<<< blocksPerGrid, threadsPerBlock >>>(d_In_M, nverts, k);
        err = cudaGetLastError();

        if (err != cudaSuccess) {
            fprintf(stderr, "Failed to launch kernel 2! ERROR= %d\n",err);
            exit(EXIT_FAILURE);
        }
    }
```

1.2. Cálculo de la longitud del mayor camino

Se ha extendido la implementación del programa añadiendo un nuevo kernel que calcula la longitud del camino de mayor longitud dentro de los caminos más cortos encontrados anteriormente por haber aplicado Floyd. El kernel es de reducción y usa hebras unidimensionales, en el que cada bloque analiza una parte de la matriz y almacena los resultados en memoria compartida, para finalmente tener un vector en memoria compartida cuyo tamaño es el número de bloques y procesarlo en CPU para realizar la última reducción y sacar el máximo.

```
// Kernel para calcular la longitud del camino de mayor longitud dentro de los
caminos
// más cortos encontrados usando anteriormente el algoritmo de Floyd
// Se realiza mediante reducción
__global__ void mayor_longitud_reduce(int * M, int * max, const int nverts) {
    extern __shared__ int sdata[];  // Datos en memoria compartida

    int tid = threadIdx.x;
    int ij = threadIdx.x + blockDim.x * blockIdx.x; // Índice global de hebra
    const int i = ij / nverts;
    const int j = ij - i * nverts;

    // Comprobar que la hebra del bloque puede operar en la matriz y obtener su
distancia
    // Si no, se le pone un valor demasiado bajo para que no pueda ser el máximo
    sdata[tid] = (i < nverts && j < nverts && M[i * nverts + j] != INF) ? M[i *
nverts + j] : -1000000;
    // Esperar a que acaben todas las hebras para terminar de llenar la memoria
compartida
    __syncthreads();

    // Hacer la reducción en memoria compartida
    // Se hace una división por dos más eficiente mediante un desplazamiento de
bits
```

```
    for (int s = blockDim.x / 2 ; s > 0 ; s >>= 1) {
        // Comprobar que la hebra actual está dentro de la mitad que está
operando
        if (tid < s) {
            // Calcular máximo en la posición que le corresponde a la hebra
actual
            if (sdata[tid] < sdata[tid + s]) {
                sdata[tid] = sdata[tid + s];
            }
        }

        // Esperar a que acaben todas las hebras para poder decidir el máximo del
bloque
        __syncthreads();
    }

    // La primera hebra se ocupa de escribir resultado de este bloque en la
memoria local
    if (tid == 0) {
        max[blockIdx.x] = sdata[0];
    }
}
```

El tamaño de la memoria compartida se calcula antes de llamar el kernel y se le pasa como un parámetro de ejecución de CUDA. Se crea además un vector en CPU en el cual se copian los contenidos calculados por el kernel para realizar la reducción final en CPU.

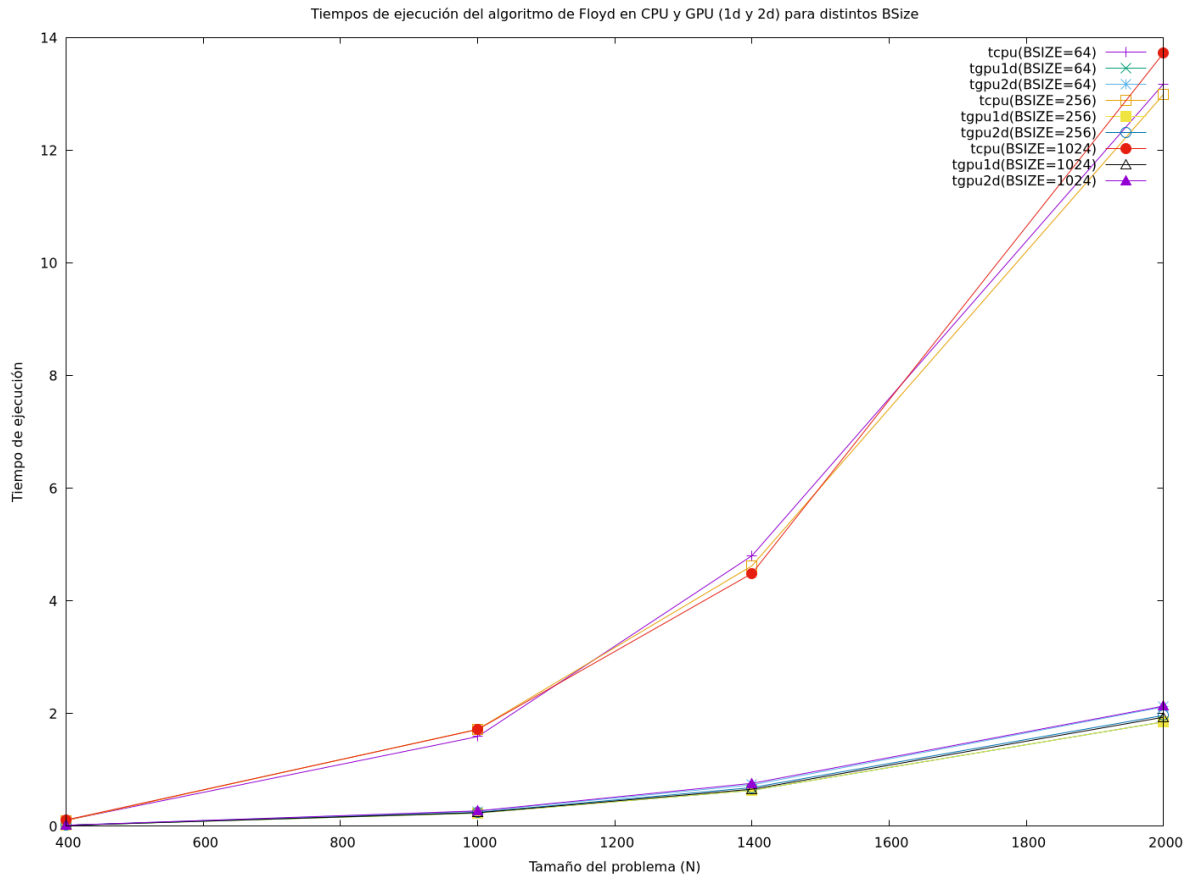
```
// Tamaño del bloque (número de hebras por bloque)
int threadsPerBlock = blocksize * blocksize;
// Tamaño del grid (número de bloques por grid)
int blocksPerGrid = (nverts2 + threadsPerBlock - 1) / threadsPerBlock;
// Tamaño de la memoria compartida
int smemSize = threadsPerBlock * sizeof(int);
// Longitud del camino de mayor longitud
int * d_max;
cudaMalloc ((void **) &d_max, sizeof(int)*blocksPerGrid);

// Calcular la longitud del camino de mayor longitud
mayor_longitud_reduce<<< blocksPerGrid, threadsPerBlock, smemSize >>>(d_In_M,
d_max, nverts);

// Terminar de computar el reduce en CPU
// Se calcula el máximo de los encontrados en los distintos bloques
int * h_max = (int*) malloc(blocksPerGrid*sizeof(int));
cudaMemcpy(h_max, d_max, blocksPerGrid*sizeof(int), cudaMemcpyDeviceToHost);
int longitudMaxima = -1000000;
for (int i = 0 ; i < blocksPerGrid ; i++) {
    longitudMaxima = (longitudMaxima > h_max[i]) ? longitudMaxima : h_max[i];
}
```

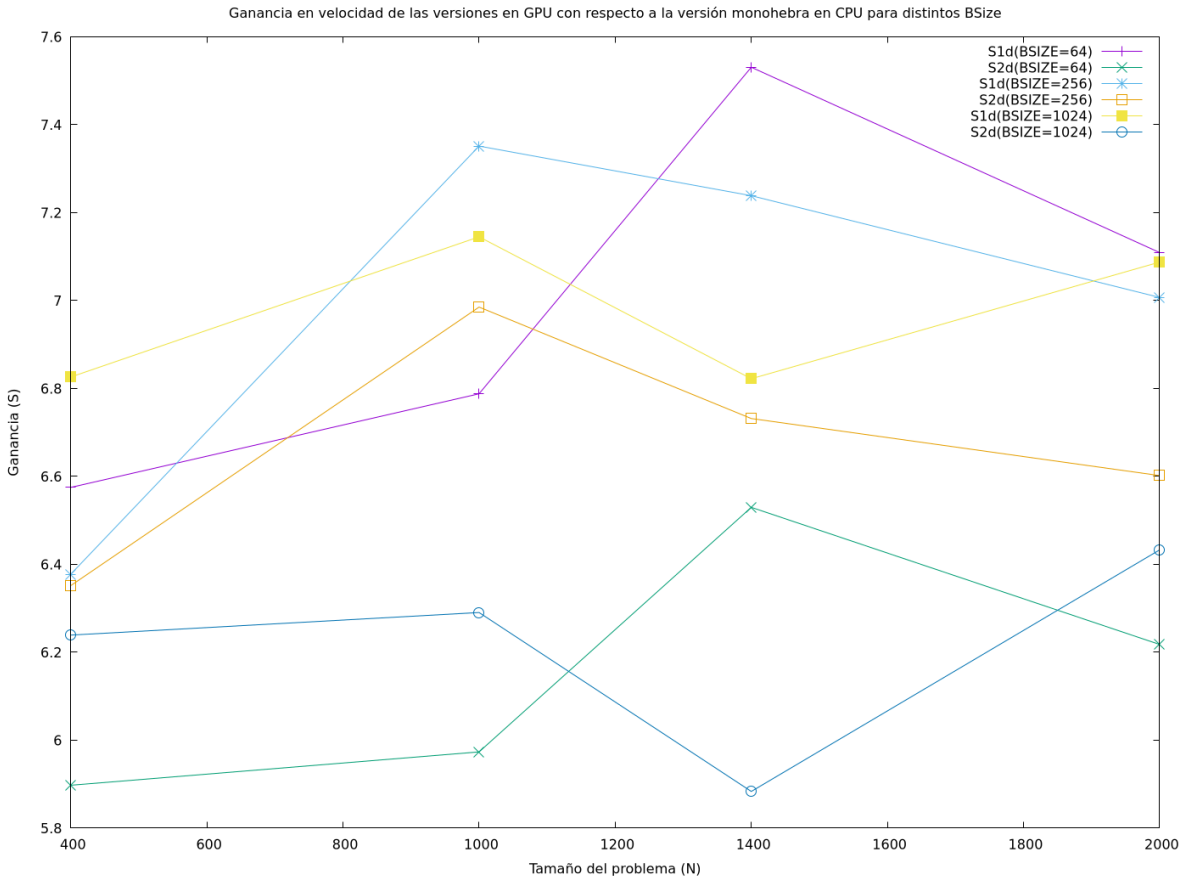
1.3. Resultados experimentales obtenidos

En la carpeta **resultados** se tienen tres ficheros con el nombre **tiempos_.dat**, siendo cada uno de los tamaños de bloque estudiados para multiplicar por si mismo y tener bloque bidimensional (8, 16 y 32). En ellos, la primera columna representa el tamaño del problema N (400, 1000, 1400 y 2000), y las siguientes columnas representan respectivamente el tiempo en CPU, el tiempo en GPU para bloques unidimensionales, el tiempo en GPU para bloques bidimensionales, la ganancia con respecto a la solución unidimensional y la ganancia con respecto a la solución unidimensional. Vemos a continuación por ejemplo la tabla para blocksize = 8.

Tamaño del problema (N)	TCPU	TGPU_1D	TGPU_2D	S_1D	S_2D
400	0.107546	0.0163569	0.0182359	6.57495	5.89749
1000	1.59137	0.23444	0.266412	6.78797	5.97335
1400	4.80378	0.637903	0.735697	7.53059	6.52957
2000	13.1779	1.85365	2.11941	7.10918	6.21772

En la misma carpeta también se tienen dos gráficas: **grafica_tiempos.png** con la representación de los tiempos de ejecución distintos con respecto al tamaño del problema para todos los blocksize estudiados, y **grafica_ganancia.png** con las ganancias resultado de los anteriores datos.





Como se puede ver en los resultados obtenidos, los tiempos en CPU llegan a ser el doble de lentos que en GPU, ya que la GPU aprovecha el paralelismo para realizar los cálculos de forma más eficiente y rápida. Sin embargo dentro de los propios tiempos en GPU vemos muy pequeña diferencia entre todas las versiones, siendo lo único destacable que para tamaños de bloque mayores, el algoritmo con bloques bidimensionales tarda más.

En cuanto a las ganancias podemos observar que cuanto menor es el tamaño de bloque, mayor es la ganancia con respecto a CPU, y que en concreto como ya hemos visto antes la versión unidimensional es más eficiente. Tenemos un caso especial, y es que con el tamaño de bloque más grande (1024) en bloques unidimensionales tenemos mejores valores que para bloques más pequeños. Sin embargo en todos los casos existe una mejora con respecto al algoritmo en CPU de más de 5 veces y llegando hasta casi 8 con respecto al original, lo cual expresa el beneficio que conlleva resolver el problema en paralelo.

2. Implementación en CUDA de una operación vectorial

Se implementa una operación vectorial sobre un vector de entrada A, con N+4 valores generados aleatoriamente, obteniendo un vector resultado B con N valores. Además, se calcula el máximo de entre los valores del vector.

Al igual que en el ejercicio anterior, para su ejecución, recolección de resultados y representación gráfica se usa el script **ejecuciones.sh** que borra anteriores resultados, compila el programa y lo ejecuta varias veces para distintos tamaños de bloque unidimensional (256, 512, 1024) para un tamaño del problema fijo que se puede cambiar dentro del script (N = 200000). Por último llama al script **graficas.gp**, que coge los datos del archivo de resultados y los representa en dos gráficas, una para los tiempos y otra para las ganancias, ambas exportadas a un archivo PNG. Todos los archivos de datos y las gráficas se encuentran en la carpeta **resultados**.

2.1. Cálculo del vector B con variable en memoria compartida

Se implementa un kernel en el cual se crea una variable en memoria compartida donde se vuelcan los contenidos del vector A, y para calcular el vector B se leen los datos de memoria compartida. Cada hebra se ocupa de una posición del vector B, empezando en la hebra/posición 2 puesto que el algoritmo exige que cada valor se calcule con dos posiciones hacia detrás, por tanto no se podría empezar ni en 0 ni en 1 porque no habría valores antes y se saldría del vector.

Tanto en esta versión como en la siguiente se llama con los mismos parámetros de tamaño de bloque (especificado al llamar al programa) y bloques por grid (calculado como en el ejercicio anterior). En cuanto al tamaño de la memoria compartida, es del mismo tamaño que el vector B, es decir, N.

```
// Kernel que realiza el cálculo del vector B con memoria compartida
__global__ void vectorial_shared(float * A, float * B, const int n) {
    extern __shared__ float sdata[];    // Datos en memoria compartida
    int i = threadIdx.x + blockDim.x * blockIdx.x;  // Índice global de hebra
    float Ai, Aim1, Aim2, Aip1, Aip2;

    if (i >= 2 && i < n + 2) {
        // Almacenar A en memoria compartida
        sdata[i] = A[i];

        // Esperar a que todas las hebras terminen de llenar la memoria
compartida
        __syncthreads();

        // Hacer la operación usando la memoria compartida
        // Cada hebra se ocupa de una posición del vector B
        Aim2 = sdata[i - 2];
        Aim1 = sdata[i - 1];
        Ai = sdata[i];
        Aip1 = sdata[i + 1];
        Aip2 = sdata[i + 2];
        B[i] = (pow(Aim2, 2) + 2.0 * pow(Aim1, 2) + pow(Ai, 2)
        - 3.0 * pow(Aip1, 2) + 5.0 * pow(Aip2, 2)) / 24.0;
    }
}
```

2.2. Cálculo del vector B con variable en memoria global

Se implementa un kernel en el cual se opera igual que en el anterior pero sin volcar los datos en memoria compartida, simplemente usando los vectores ya especificados al kernel que están en memoria global. La asignación de hebras a posiciones en B es la misma que en el anterior.

```
// Kernel que realiza el cálculo del vector B en memoria global
__global__ void vectorial_global(float * A, float * B, const int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;  // Índice global de hebra
    float Ai, Aim1, Aim2, Aip1, Aip2;

    if (i >= 2 && i < n + 2) {
        // Hacer la operación usando la memoria global
        // Cada hebra se ocupa de una posición del vector B
        Aim2 = B[i - 2];
```

```
        Aim1 = B[i - 1];
        Ai = B[i];
        Aip1 = B[i + 1];
        Aip2 = B[i + 2];
        B[i] = (pow(Aim2, 5) + 2.0 * pow(Aim1, 5) + pow(Ai, 5)
        - 3.0 * pow(Aip1, 5) + 5.0 * pow(Aip2, 5)) / 24.0;
    }
}
```

2.3. Cálculo del valor máximo de todos los valores almacenados en B

Se implementa un kernel de reducción para buscar el máximo en el vector, del mismo estilo que el visto en el ejercicio anterior. A cada bloque se le asigna una parte del vector en el que aplican la reducción en memoria compartida, donde cada hebra va almacenando en su posición asignada del vector el mayor que encuentra.

```
// Kernel que calcula el valor máximo de entre los valores de un vector usando
reduce
__global__ void maximo_reduce(float * B, float * max, const int n) {
    extern __shared__ float sdata[];    // Datos en memoria compartida

    int tid = threadIdx.x;
    int i = threadIdx.x + blockDim.x * blockIdx.x;  // Índice global de hebra

    // Almacenar vector en memoria compartida
    // Comprobar que la hebra del bloque puede operar en el vector
    // Si no, se le pone un valor demasiado bajo para que no pueda ser el máximo
    sdata[tid] = (i < n + 2) ? B[i] : -1000000.0f;
    // Esperar a que acaben todas las hebras para terminar de llenar la memoria
compartida
    __syncthreads();

    // Hacer la reducción en memoria compartida
    // Se hace una división por dos más eficiente mediante un desplazamiento de
bits
    for (int s = blockDim.x / 2 ; s > 0 ; s >>= 1) {
        // Comprobar que la hebra actual está dentro de la mitad que está
operando
        if (tid < s) {
            // Calcular máximo en la posición que le corresponde a la hebra
actual
            if (sdata[tid] < sdata[tid + s]) {
                sdata[tid] = sdata[tid + s];
            }
        }

        // Esperar a que acaben todas las hebras para poder decidir el máximo del
bloque
        __syncthreads();
    }

    // La primera hebra se ocupa de escribir resultado de este bloque en la
memoria local
    if (tid == 0) {
        max[blockIdx.x] = sdata[0];
```

```
    }
}
```

Finalmente se realiza la última reducción en CPU.

```
// Calcular el valor máximo de todos los valores almacenados en B
float * d_max;
cudaMalloc ((void **) &d_max, sizeof(float)*blocksPerGrid);
maximo_reduce<<< blocksPerGrid, threadsPerBlock, smemSize >>>(B, d_max, n);
float * h_max = (float*) malloc(blocksPerGrid*sizeof(float));
cudaMemcpy(h_max, d_max, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost);

// Terminar reduce en CPU
mx = maximo_reduce_cpu(h_max, blocksPerGrid);
```

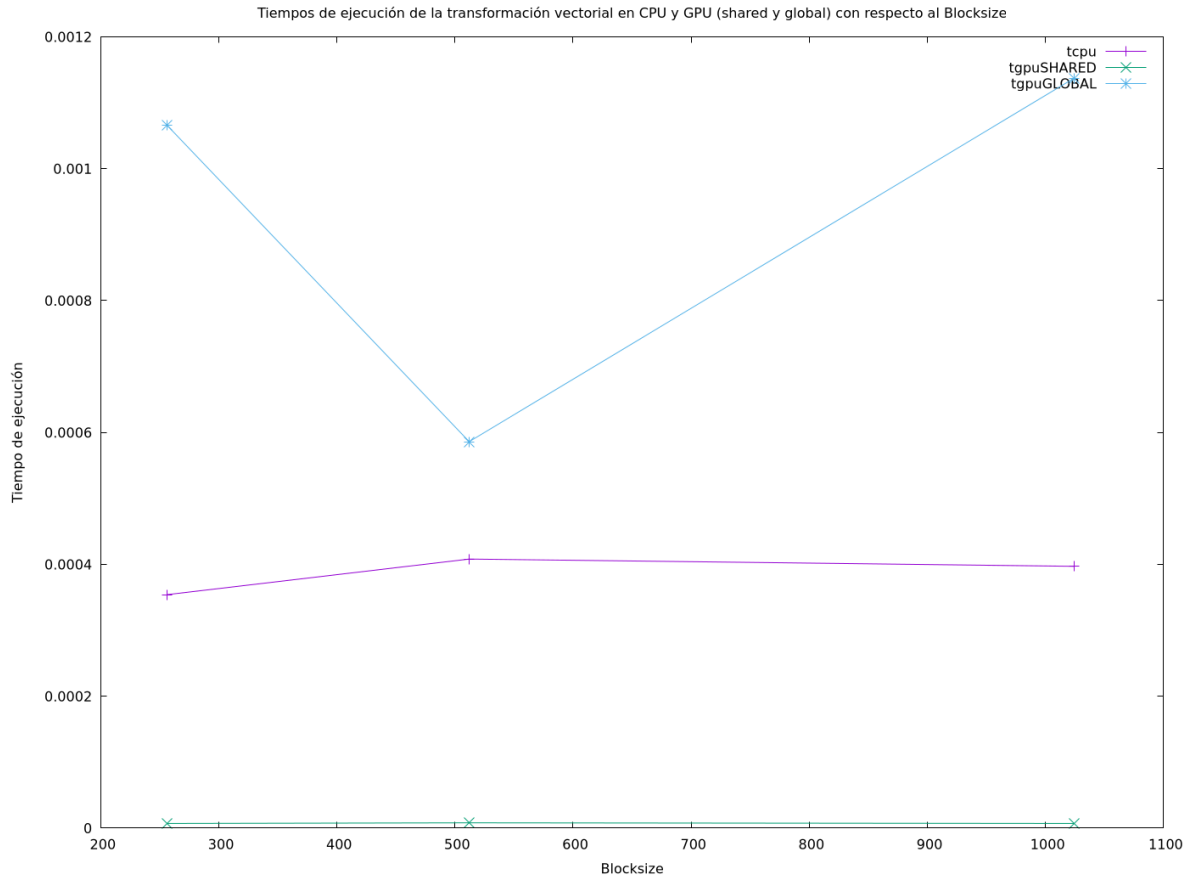
```
// Función para terminar de hacer el máximo con reduce en CPU
float maximo_reduce_cpu(float * h_max, int blocksPerGrid) {
    // Terminar de computar el reduce en CPU
    // Se calcula el máximo de los encontrados en los distintos bloques
    float mx = -1000000.0f;
    for (int i = 0 ; i < blocksPerGrid ; i++) {
        mx = (mx > h_max[i]) ? mx : h_max[i];
    }

    return mx;
}
```

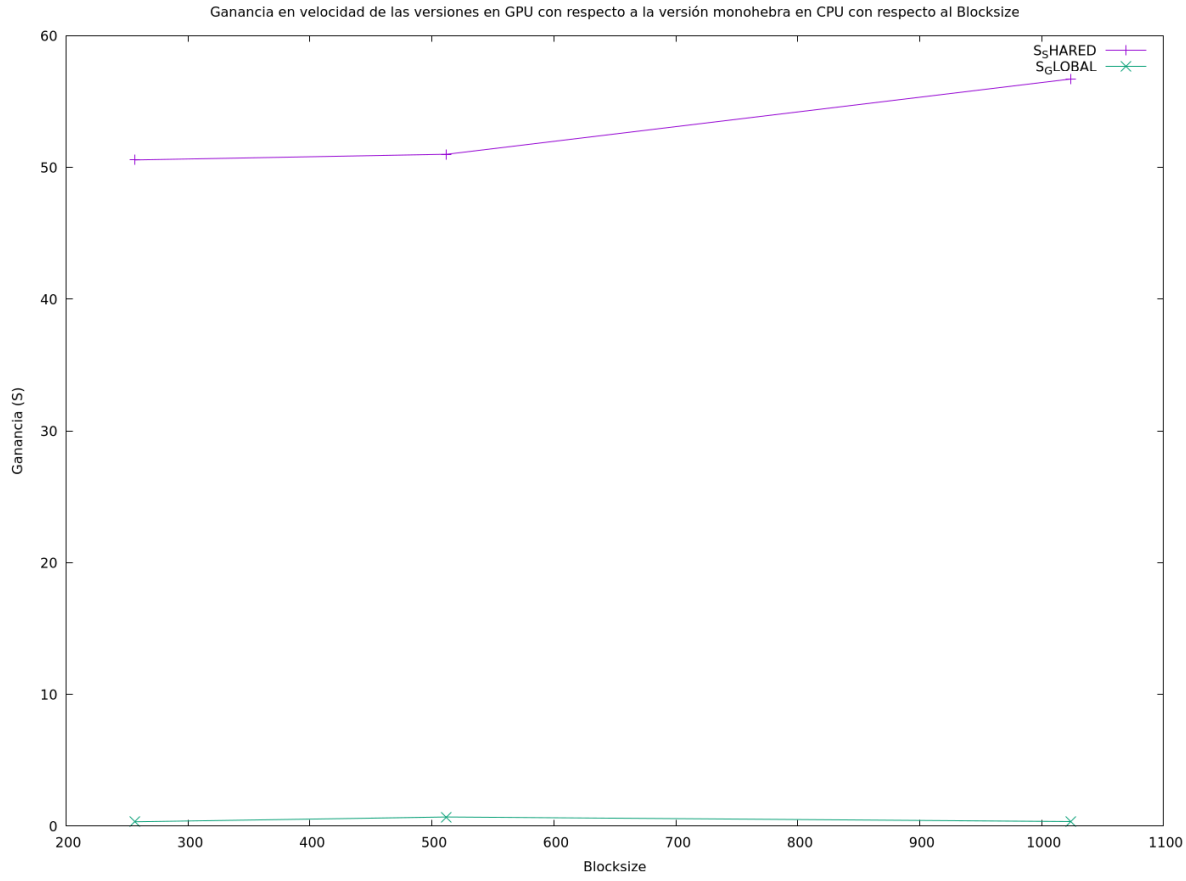
2.4. Resultados experimentales obtenidos

En la carpeta **resultados** se tiene un fichero con el nombre **tiempos.dat** en el que la primera columna representa el tamaño de bloque (256, 512, 1024), y las siguientes columnas representan respectivamente el tiempo en CPU, el tiempo en GPU en memoria compartida, el tiempo en GPU en memoria global, la ganancia con respecto a la solución en memoria compartida y la ganancia con respecto a la solución en memoria global. Vemos a continuación la tabla con los datos.

Blocksize	TCPU	TGPU_SHARED	TGPU_GLOBAL	S_SHARED	S_GLOBAL
256	0.0003540	0.0000070	0.0010660	50.5714286	0.3320826
512	0.0004080	0.0000080	0.0005860	51.0000000	0.6962457
1024	0.0003970	0.0000070	0.0011370	56.7142857	0.3491645

En la misma carpeta también se tienen dos gráficas: **grafica_tiempos.png** con la representación de los tiempos de ejecución distintos con respecto al blocksize, y **grafica_ganancia.png** con las ganancias resultado de los anteriores datos.





Podemos observar que el tiempo de ejecución en PCPU, como es lógico, se mantiene siempre constante porque no le afecta el blocksize. Además, el tiempo de ejecución en CPU es mejor que el tiempo en GPU global, que además varía mucho según el blocksize. Por otra parte, el tiempo en GPU shared es el mejor de los tres puesto que el acceso a memoria compartida es más rápido, y además se mantiene prácticamente constante da igual el blocksize que se escoja.

La ganancia con respecto al tiempo en memoria global sale muy próxima a 0 puesto que los tiempos en CPU son mejores, por tanto no se obtiene prácticamente ninguna mejora en el paralelismo con respecto a la CPU. Sin embargo la ganancia en memoria compartida aumenta muy considerablemente, teniendo un speedup de 50 veces mayor con respecto al tiempo en CPU, lo cual ejemplifica perfectamente la ventaja de la rapidez que tiene la memoria compartida.