Universidad De Granada

Grado ingeniería informática (2017 – 2018)

Programación Paralela

Implementación de algoritmos paralelos de datos en GPU usando CUDA

Trabajo realizado por Antonio Miguel Morillo Chica.

---

1. Implementación del Algoritmo de Floyd

Disponemos del Algoritmo de Floyd para el cálculo de todos los caminos más cortos en un grafo etiquetado. El número de vértices N puede ser cualquiera (no cumple ninguna condición de divisibilidad). Esta implementación usa una grid unidimensional de bloques de hebras CUDA unidimensionales (con forma alargada). Se describe en la misma un kernel que se lanza una vez para cada iteración k=0, ..., N−1. Cada hebra es responsable de actualizar un elemento de la matriz resultado parcial A(k+1) y ejecutará el siguiente algoritmo:

Kernel Actualizacion A(i,j) en iteración k

$$A_{ij}^{k+i} = min\{A_{ij}^{k}, A_{ik}^{k} + A_{kj}^{k}\}$$

end;

En esta versión, las hebras CUDA se organizan como una Grid unidimensional de bloques de hebras unidimensionales. Teniendo en cuenta que cada hebra se encarga de actualizar un elemento de la matriz A en la iteración k, tendríamos que tener al menos N×N hebras. Los tamaños de bloque de hebras CUDA usuales son: 64, 128, 256, 512 ó 1024.

1.1. Ejercicios propuestos.

. Modificar la implementación CUDA del algoritmo de Floyd para que las *hebras CUDA se organicen como una grid bidimensional de bloques cuadrados de hebras bidimensionales. Los ejemplos de tamaños de bloque usuales son: 8 × 8, 16 × 16 y 32 × 32. Realizar también medidas de tiempo de ejecución sobre los algoritmos implementados. El programa plantilla que se ofrece incluye la toma de los tiempos de ejecución del algoritmo en CPU y en GPU. Deberán realizarse las siguientes medidas para problemas de diferentes tamaño y diferentes tamaño de bloque CUDA*

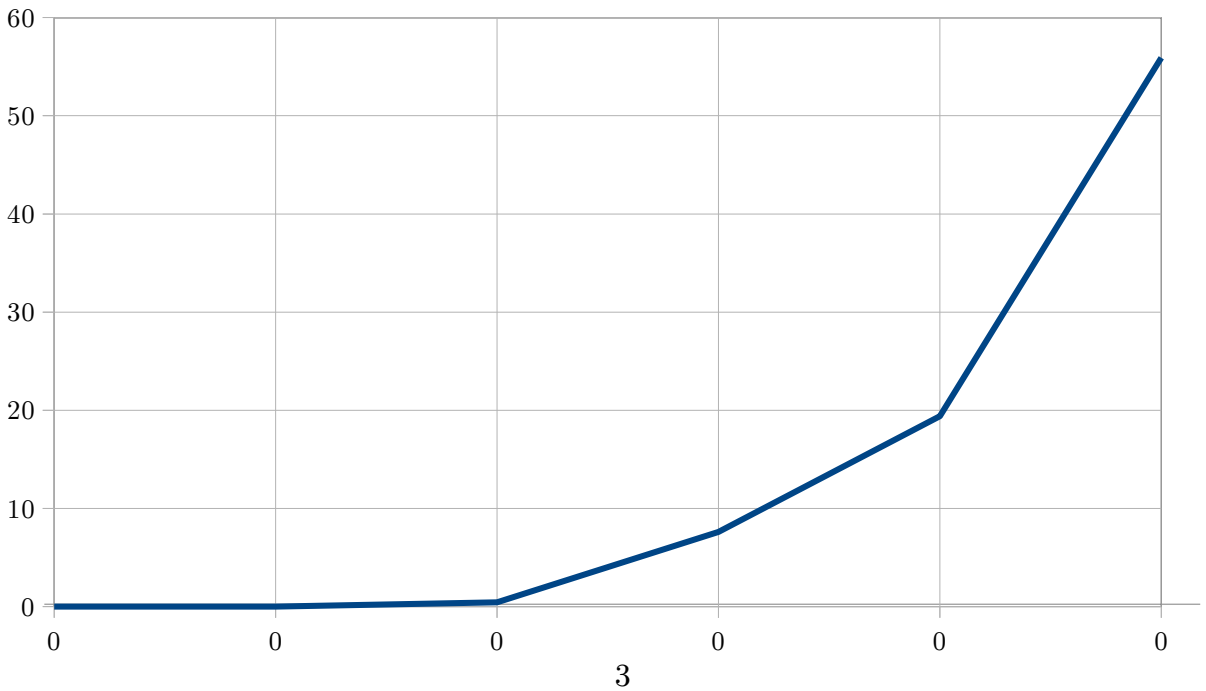
1.1.1. Algoritmo de Floyd Secuencial

Para la obtención del tiempo secuencial el código implementado es el siguiente:

```
int main (int argc, char **argv){
  { ... }
  double t1=clock();
    for(int k = 0; k < nverts; k++)
      for(int i = 0; i < nverts; i++)
        for(int j = 0; j < nverts; j++)
          if (i != j && i != k && j != k) {
            int vikj = min(G.arista(i,k) + G.arista(k,j), G.arista(i,j));
            G.inserta_arista(i,j,vikj);
          }
  double t2 = clock();
  double tSecuencial = (t2-t1) / CLOCKS_PER_SEC;
  { ... }
}
```

La tabla de tiempos para la ejecución secuencial ha sido la siguiente, también aporto una tabla comparativa de los tiempos en relación a los diferentes grafos. La ganancia no está expresada ya que la ejecución solo se hace con una hebra y no tenemos que compararla como haremos acontinuación entre la CPU y GPU con diferentes tamaños y de blockes.

	input4	input64	input400	input1000	input1400	Input2000
Tiempo	0,000002	0,002672	0,427492	7,6	19,39	55,9



1.1.2. Algoritmo de Floyd 1D.

El kernel aportado en la práctica es para un grid unidimensional, aunque se exponga el código en la memoria no ha sido modificado. El kernel sería el siguiente:

```
int ij = threadIdx.x + blockDim.x * blockIdx.x;
if (ij < nverts * nverts) {
   int Mij = M[ij];
   int i= ij / nverts;
   int j= ij - i * nverts;
   if (i != j && i != k && j != k) {
        int Mikj = M[i * nverts + k] + M[k * nverts + j];
   Mij = (Mij > Mikj) ? Mikj : Mij;
   M[ij] = Mij;
   }
}
```

El código de ejecución para ello no lo aporto ya que es muy largo y sería redundante. Los resultados obtenidos en relación entre la GPU y la CPU con los diferentes entrada de datos ha sido:

Bsize = 16	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,000027	0,000199	0,001108	0,002728	0,550778	2,9727
GANANCIA	0,07	13,43	385,82	2784,12	35,2	21,17

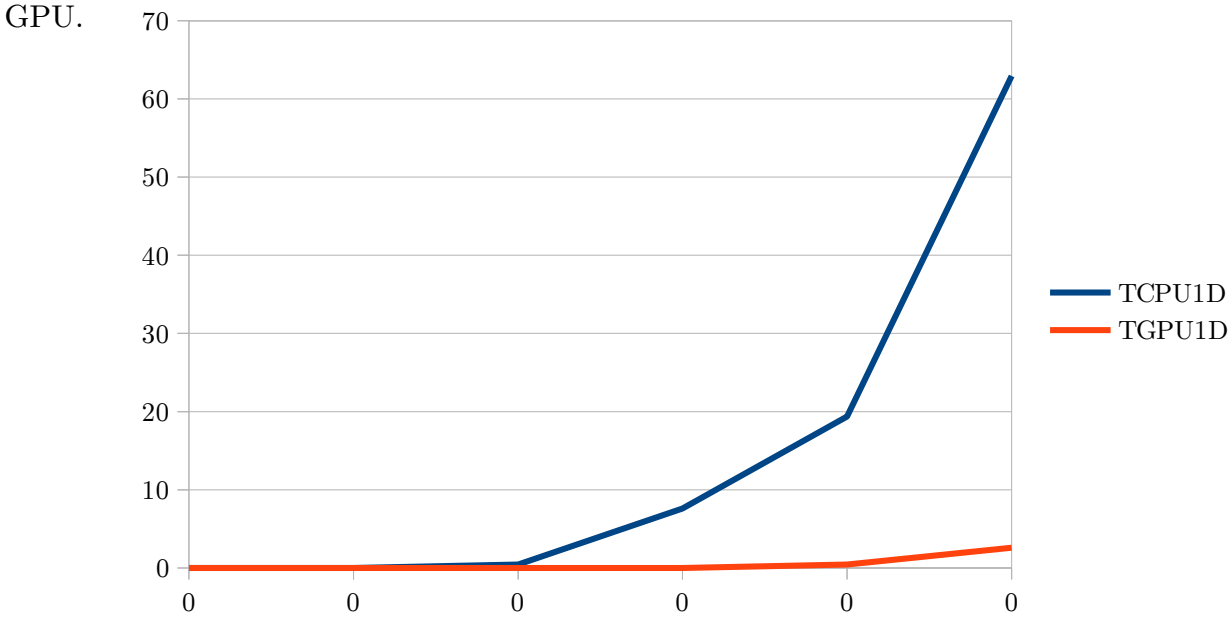
Bsize = 32	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,000026	0,00022	0,001179	0,002774	0,453468	2,71401
GANANCIA	0,08	12,15	362,59	2737,96	42,76	23,19

Bsize = 64	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,000026	0,000216	0,001205	0,002707	0,452166	2,57208
GANANCIA	0,08	12,37	354,77	2805,72	42,88	24,47

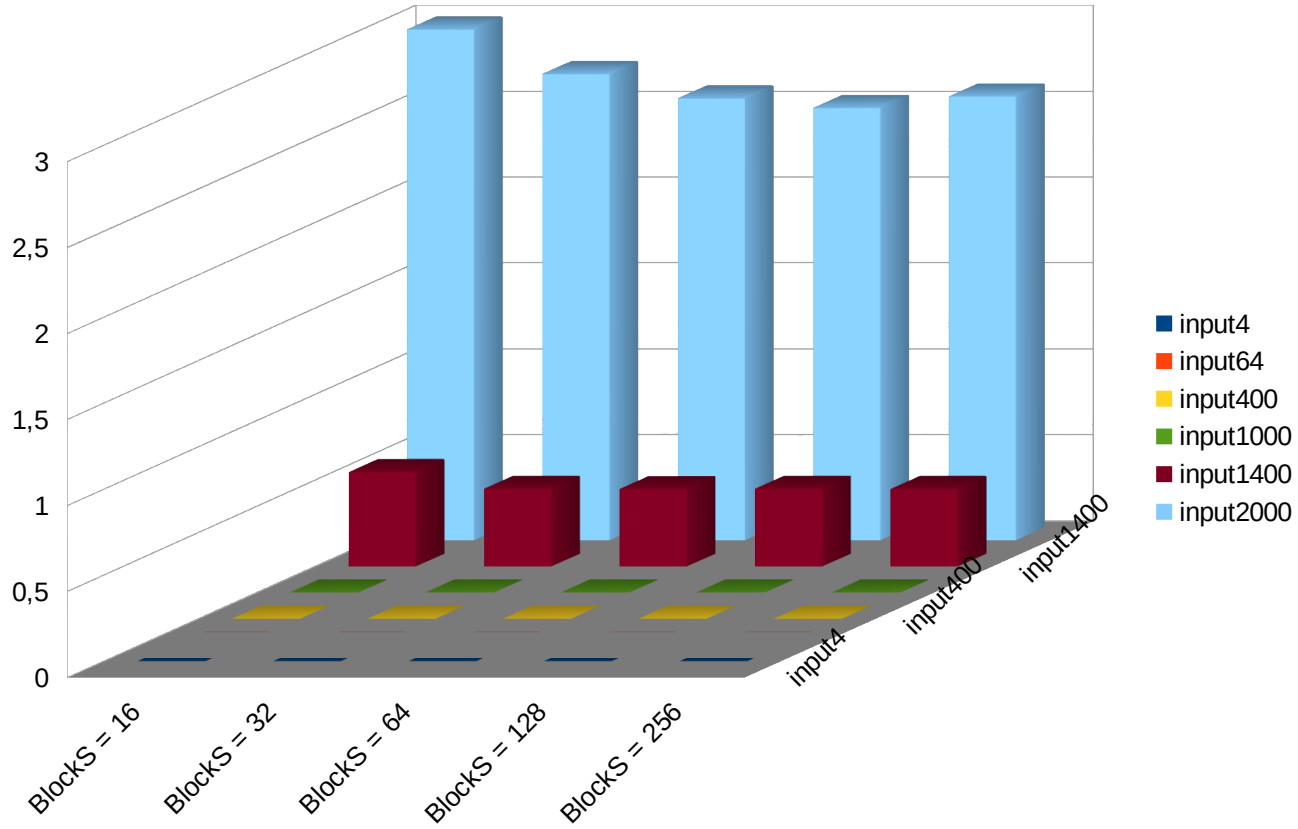
Bsize = 128	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,000026	0,000202	0,001396	0,002755	0,453536	2,51607
GANANCIA	0,08	13,23	306,23	2756,84	42,75	25,01

Bsize = 256	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,000026	0,000216	0,001072	0,002751	0,452365	2,58237
GANANCIA	0,08	12,37	398,78	2760,85	42,86	24,37

Como podemos ver los resultados con diferentes tamaños de bloques a partir de 64 bloques no mejora aunque creo que es por la cantidad de hebras de mi GPU.



	input4	input64	input400	input1000	input1400	input2000
BlockS = 16	0,000027	0,000199	0,001108	0,002728	0,550778	2,9727
BlockS = 32	0,000026	0,00022	0,001179	0,002774	0,453468	2,71401
BlockS = 64	0,000026	0,000216	0,001205	0,002707	0,452166	2,57208
BlockS = 128	0,000026	0,000202	0,001396	0,002755	0,453536	2,51607
BlockS = 256	0,000026	0,000216	0,001072	0,002751	0,452365	2,58237



Los tiempos obtenidos son practicamente igueales, para los distintos blockSize aunque mejor dicho lo que ocurre es que la mejora máxima existe con 64 de tamaño, despues no mejora.

1.1.3. Algoritmo de Floy 2D.

El kernel implementado para un grid bidmensional para la ejecución en la GPU es el siguiente:

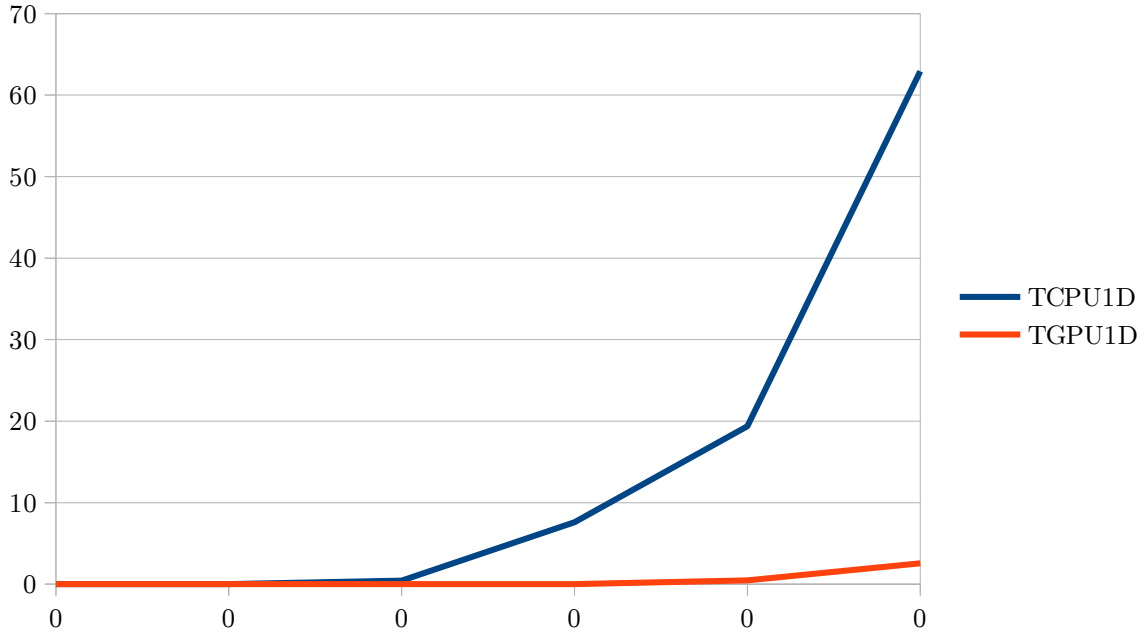
```
__global__ void floyd_kernel_2D(int * M, const int nverts, const int k) {
  int i = blockIdx.y * blockDim.y + threadIdx.y,  // índice i en la matriz
      j = blockIdx.x * blockDim.x + threadIdx.x,  // índice j en la matriz
      ij;

  if (i < nverts && j < nverts) {
    if (i != j && i != k && j != k) {
      ij = i * nverts + j;
      M[ij] = min(M[i * nverts + k] + M[k * nverts + j], M[ij]);
    }
  }
}
```

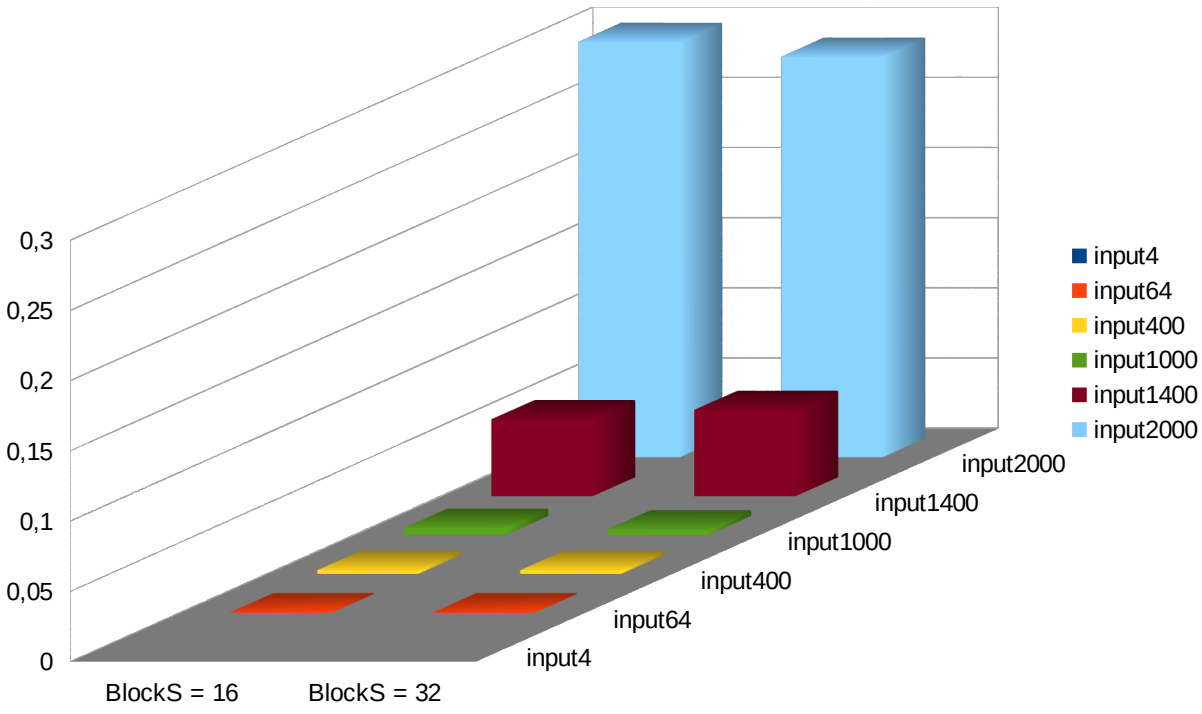
Los tiempos para esta implementación solo los he podido sacar con tamaños de bloque pequeños porque sino poducían un error en el kenel.

Bsize = 16	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,0000065	0,000416	0,002857	0,005323	0,054997	0,296118
GANANCIA	0,31	6,42	149,63	1426,84	352,56	212,51

Bsize = 32	input4	input64	input400	input1000	input1400	input2000
$TCPU_{1D}$	0,000002	0,002672	0,427492	7,6	19,39	62,9274
$TGPU_{1D}$	0,0000048	0,000323	0,002832	0,004194	0,061603	0,285521
GANANCIA	0,42	8,27	150,95	1810,94	314,76	220,39



	input4	input64	input400	input1000	input1400	input2000
BlockS = 16	0,0000065	0,000416	0,002857	0,005323	0,054997	0,296118
BlockS = 32	0,0000048	0,000323	0,002832	0,004194	0,061603	0,285521



1.2. Floyd para el camino de mayor longitud.

Extender la implementación en CUDA C desarrollada para que se calcule también la longitud del camino de mayor longitud dentro de los caminos más cortos encontrados. Usar para ello un kernel CUDA de reducción aplicado al vector que almacena los valores de la matriz resultado.

Para este apartado unicamente he implementado un pequeño código secuencial que se ejecut en la clase grafo que encuentra de todos los A_{ij} el menor y el mayor camino entre dos puntos.

```
void Graph::obtenMasCorto() {
  int i_f = 0, j_f = 1;
  int vij_corto = A[i_f * vertices + j_f], i,j;

  for(i=0;i<vertices;i++)
    for(j=0;j<vertices;j++)
      if (A[i*vertices+j] < A[i_f*vertices+j_f] && A[i*vertices+j] != 0){
        i_f = i; j_f = j;
      }

  cout << " ->  Más corto: ["<<i_f <<","<< j_f <<"]" << " cuya distancia es: " <<
          A[i_f*vertices+j_f] << endl;
}

void Graph::obtenMasLargo() {
  int i_f = 0, j_f = 0;
  int vij_largo = A[i_f * vertices + j_f], i,j;

  for(i=0;i<vertices;i++)
    for(j=0;j<vertices;j++)
      if ((A[i*vertices+j] > A[i_f*vertices+j_f]) &&
          (A[i*vertices+j] != 1000000) && (A[i*vertices+j] != INF) ){
        i_f = i; j_f = j;
      }

  cout << " ->  Más Largo: ["<<i_f <<","<< j_f <<"]" << " cuya distancia es: " <<
           A[i_f*vertices+j_f] << endl;
}
```

2. Implementación CUDA de una operación vectorial.

Se pretende realizar una serie de cálculos utilizando dos vectores de entrada A y B. Estos vectores podrán ser de longitud variable N (A[i], B[i] i = 0, .., N − 1 y contendrán números en coma flotante. Los vectores A y B se encuentran (solo a nivel conceptual) divididos en bloques de elementos contiguos de igual tamaño. El tamaño de bloque puede ser de 64, 128 ó 256 elementos (N = k×M, donde M ∈ {64, 128, 256} y k ∈ N). Con estos datos de entrada se calcula:

- *Un vector resultado C con N celdas tal que el valor C[i], i = 0, .., N − 1,*
- *La suma de los valores de C pertenecientes a cada bloque, almacenando el resultado en un vector D con tantas posiciones como bloques se hayan definido*
- *El valor máximo mx de todos los valores almacenados en C, independientemente de la división en bloques.*

2.1. Operación con y sin variables compartidas.

Realizar dos implementaciones CUDA C: una que realiza el cálculo del vector C (primera fase de cálculo) utilizando variables en memoria compartida y otra que no utilice este tipo de almacenamiento para esta fase de cálculo. Para el resto de fases se recomienda usar memoria compartida.

Para el apartado con sin memoria compartida el kernel es:

```
__global__ void transKernel(float * d_phi, float * d_phi_new, float cu, int n){
  int i=threadIdx.x+blockDim.x*blockIdx.x+1;

  if (i<n+2)
    d_phi_new[i] = 0.5*((d_phi[i+1]+d_phi[i-1]) - cu*(d_phi[i+1]-d_phi[i-1]));

  if (i==1) d_phi_new[0]=d_phi_new[1];
  if (i==n+1) d_phi_new[n+2]=d_phi_new[n+1];
}
```

Para cuando usamos memoria compartida:

```
__global__ void transKernel2(float *d_phi,float *d_phi_new,float cu,const int n){

  int li=threadIdx.x+1; //local index in shared memory vector
```

```
  int gi= blockDim.x*blockIdx.x+threadIdx.x+1; // global memory index
  int lstart=0; // start index in the block (left value)
  int lend=BLOCKSIZE+1; // end index in the block (right value)
  __shared__ float s_phi[BLOCKSIZE + 2]; //shared mem. vector
  float result;

  // a) Load internal points of the tile in shared memory
  if (gi<n+2) s_phi[li] = d_phi[gi];

  // b) Load the halo points of the tile in shared memory
  if (threadIdx.x == 0) // First Thread (in the current block)
    s_phi[lstart]=d_phi[gi-1];
  if (threadIdx.x == BLOCKSIZE-1) // Last Thread
    if (gi>=n+1) // Last Block
      s_phi[(n+2)%BLOCKSIZE]=d_phi[n+2];
    else
      s_phi[lend]=d_phi[gi+1];

  __syncthreads(); // Barrier Synchronization

  if (gi<n+2){
    // Lax-Friedrichs Update
    result=0.5*((s_phi[li+1]+s_phi[li-1])-cu*(s_phi[li+1]-s_phi[li-1]));
    d_phi_new[gi]=result;
  }
  // Impose Boundary Conditions
  if (gi==1) d_phi_new[0]=d_phi_new[1];
  if (gi==n+1) d_phi_new[n+2]=d_phi_new[n+1];

}
```

Aunque no he conseguido hacer que compile a la perfección y no entiendo el porqué.

2.2. Comparación de resutados.

Comparar los resultados obtenidos, tanto en tiempo de ejecución como en los valores obtenidos con respecto al código disponible para CPU. Para ello, se utilizarán valores altos para N ($N > 20000$), probando los tres tamaños de bloque indicados al inicio de este ejercicio y las versiones con y soin memoria compartida para el cálculo del vector C