Programación Paralela

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

UNIVERSIDAD DE GRANADA

Realizado por: Daniel Díaz Pareja
Fecha: 07/04/2017

Índice

1. Implementación en CUDA del algoritmo de floyd.

Se dispone de una implementación en CUDA 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). Una implementación inicial usa un 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^{k+1}_{i,j} = min{A^k_{i,j}, A^k_{i,k} + A^k_{k,j}}
  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, 256 ó 1024.

1.1 Implementación con un grid bidimensional de bloques bidimensionales cuadrados.

En este ejercicio se ha modificado 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.

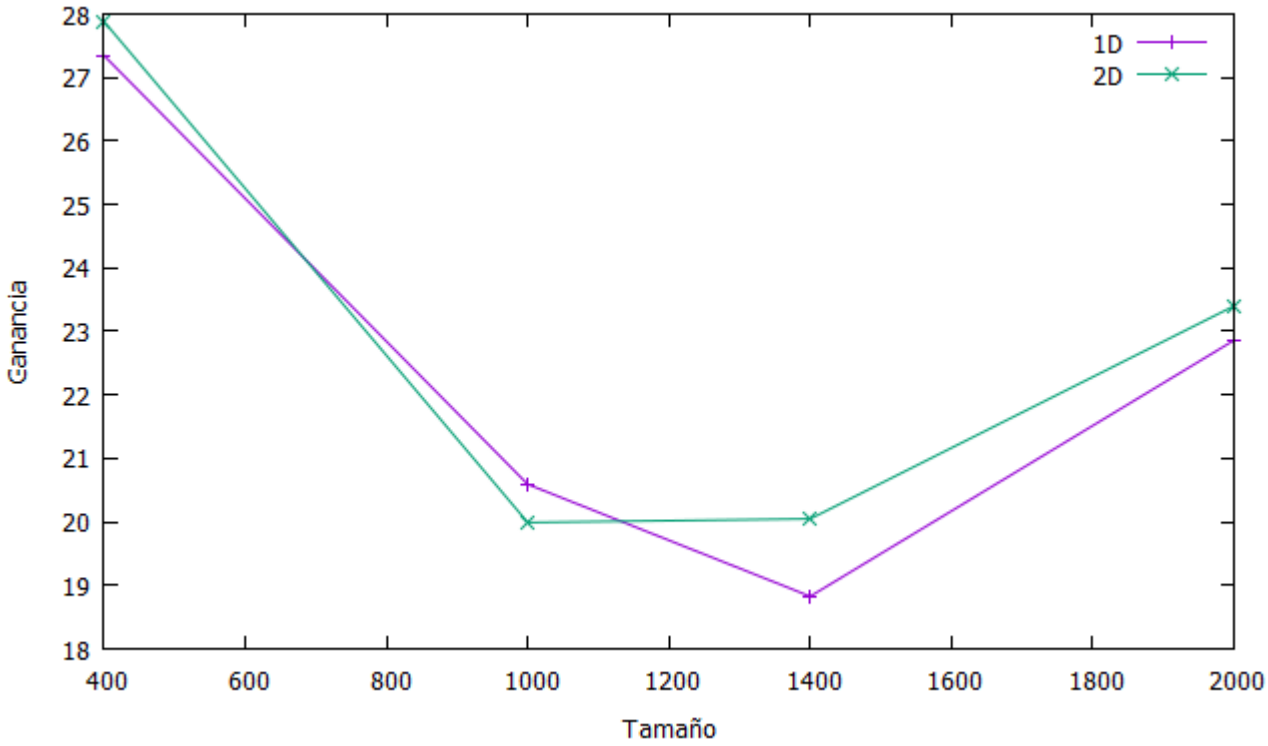
1.1.1 Medidas de tiempo y ganancia de velocidad.

Se han realizado medidas de tiempo de ejecución (en segundos), sobre los algoritmos implementados, para problemas de diferente tamaño y diferente tamaño de bloque CUDA, y se ha representado la ganancia de las versiones en GPU con respecto a la versión monohebra en CPU: $SGPU_{1D}$ y $SGPU_{2D}$. También se ha realizado una gráfica que ilustra como varía la ganancia en velocidad al aumentar el tamaño de problema para cada valor de tamaño de bloque.

- Para tamaño de bloque 64 (8 x 8):

	TCPU	$TGPU_{1D}$	$SGPU_{1D}$	$TGPU_{2D}$	$SGPU_{2D}$
n = 400	0.193113	0.00706196	27.3455	0.00692606	27.8821
n = 1000	1.78718	0.0867929	20.5913	0.089401	19.9906
n = 1400	4.34185	0.230632	18.8259	0.216584	20.0469
n = 2000	13.9267	0.609221	22.8598	0.595141	23.4007

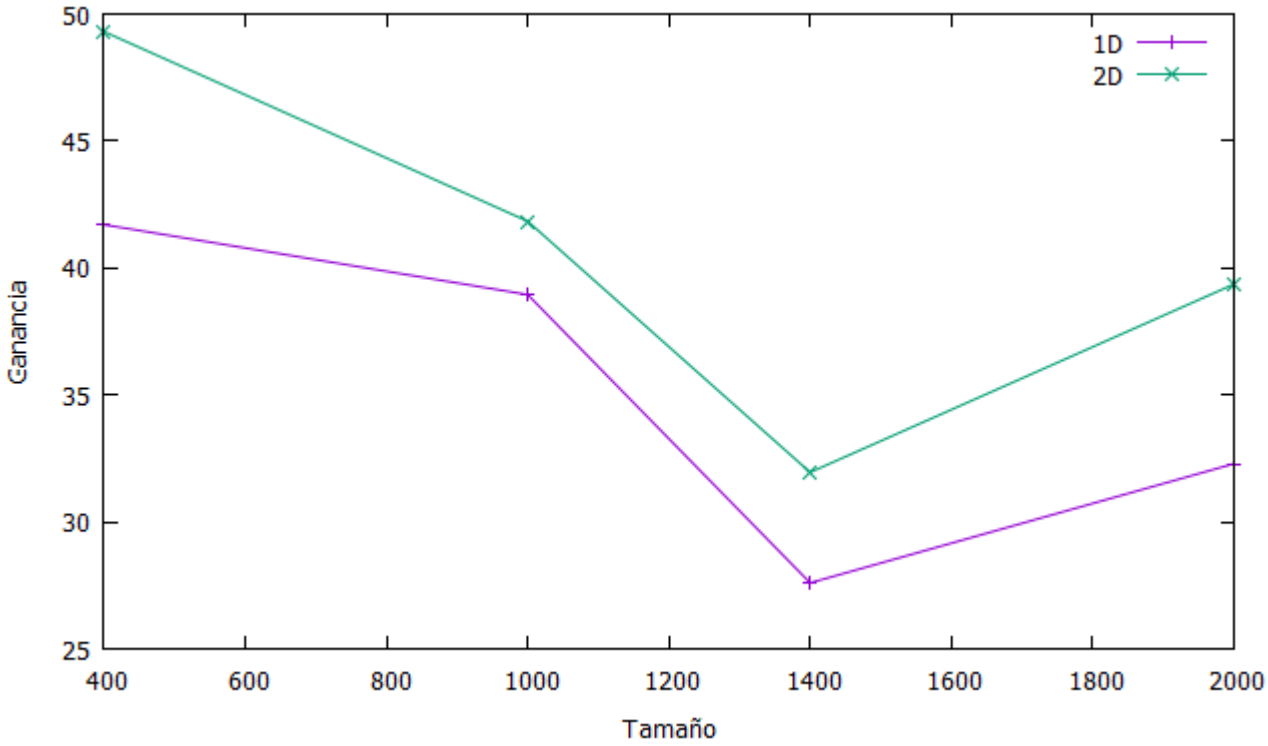
Speedup:



- Para tamaño de bloque 256 (16 x 16):

	TCPU	$TGPU_{1D}$	$SGPU_{1D}$	$TGPU_{2D}$	$SGPU_{2D}$
n = 400	0.245977	0.005898	41.7051	0.00498891	49.3048
n = 1000	2.34447	0.060194	38.9485	0.056036	41.8386
n = 1400	4.41541	0.159985	27.5989	0.138184	31.9531
n = 2000	13.9821	0.432987	32.2922	0.355221	39.3618

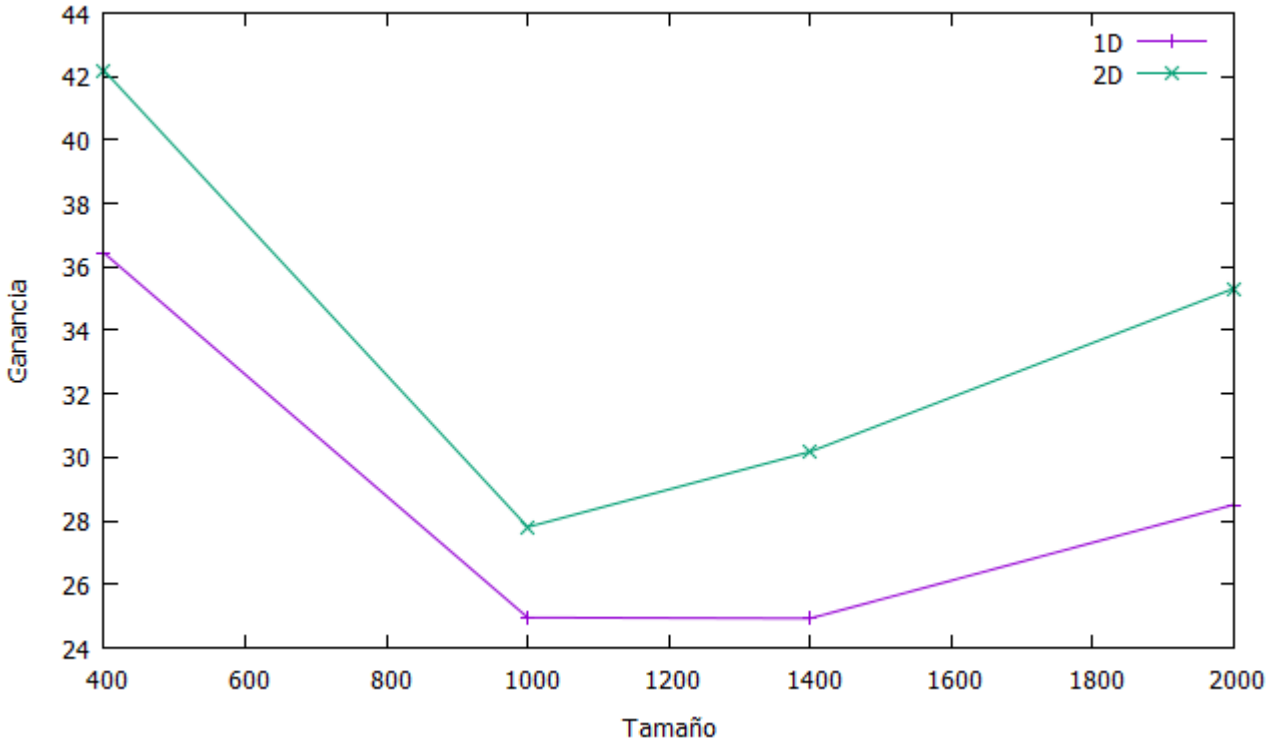
Speedup:



- Para tamaño de bloque 1024 (32 x 32):

	TCPU	$TGPU_{1D}$	$SGPU_{1D}$	$TGPU_{2D}$	$SGPU_{2D}$
n = 400	0.246216	0.00675702	36.4386	0.00583601	42.1891
n = 1000	1.79575	0.0719638	24.9535	0.064585	27.8044
n = 1400	4.46215	0.178995	24.9289	0.147847	30.1808
n = 2000	13.9363	0.488828	28.5096	0.394556	35.3215

Speedup:



1.2 Cálculo del camino de mayor longitud dentro de los caminos encontrados (Reducción en CUDA C).

En este apartado simplemente pasamos a comentar lo más destacado del kernel de reducción que se detalla a continuación:

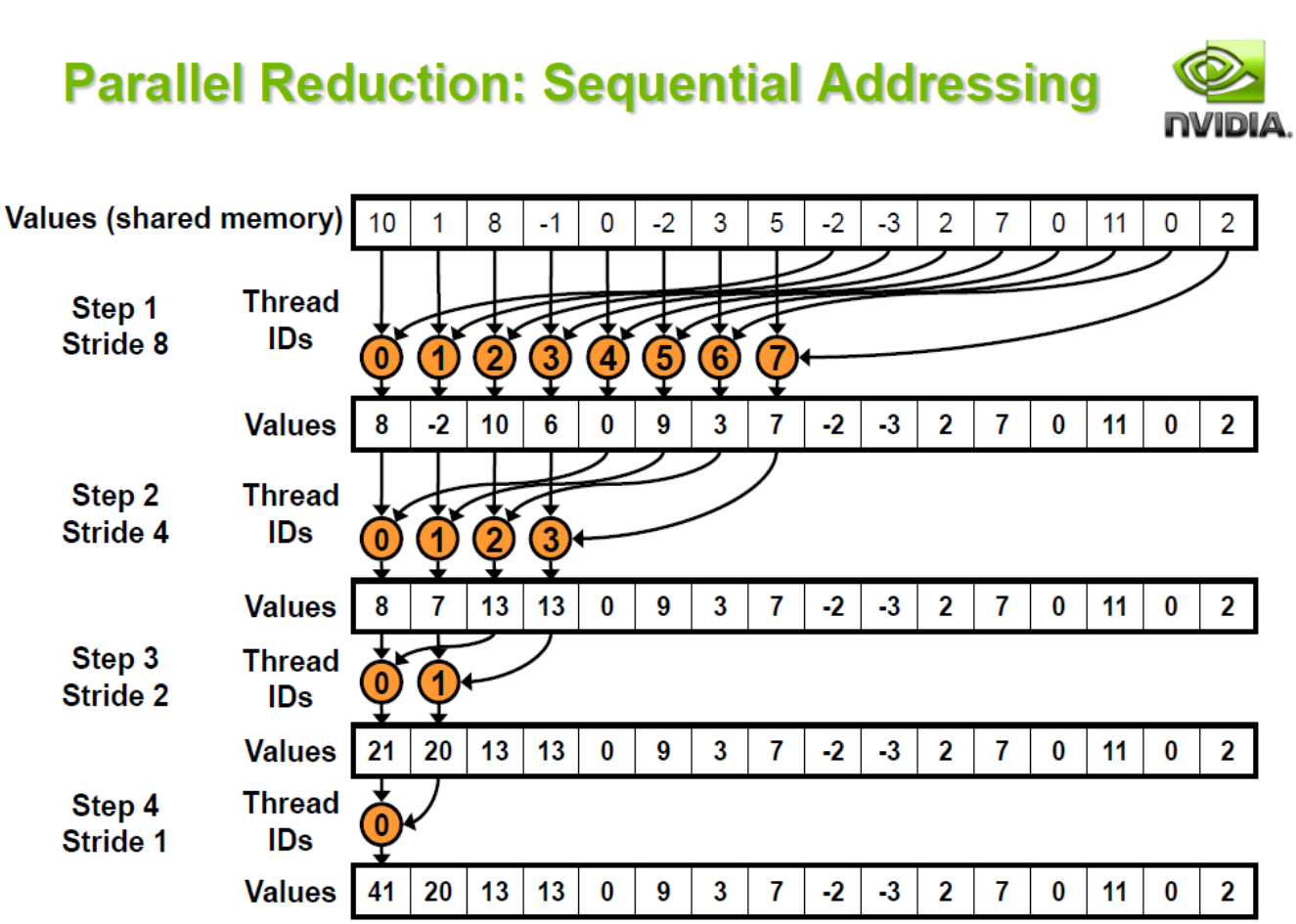
```
1  __global__ void floyd_reduction(int * M, int * M_out, int elems)
2  {
3    extern __shared__ int sdata[];
4    int ij = blockIdx.x * blockDim.x + threadIdx.x;
5    if (ij < elems){
6
7      // Copiamos a memoria compartida
8      int tid = threadIdx.x;
9      sdata[tid] = M[ij];
10     __syncthreads(); // Esperamos a que todas terminen de copiar
11
12     // fase de reducción
13     int n; // primer valor a comparar
14     int m; // segundo valor a comparar
15     for (int s = blockDim.x/2; s>0; s >>= 1) {
16         if (tid < s) {
17           n = sdata[tid];
18           m = sdata[tid+s];
19           sdata[tid] = (n > m) ? n : m;
20         }
21         __syncthreads(); // Para que la siguiente iteración
22                          //no empiece a calcular si alguna
23                          // hebra no ha terminado su reducción
24     }
25   }
26
27   // escribimos la reduccion final del bloque
28   if (threadIdx.x == 0) M_out[blockIdx.x] = sdata[0];
29
30 }
```

Respecto a los parámetros del kernel:

- **int * M** es la matriz de entrada con los caminos de Floyd ya calculados en el apartado anterior.
- **int * M_out** es un vector resultante con la reducción que cada bloque hará de un "trozo" de M.
- **int * elems** es el número de elementos de M, que debe ser nverts * nverts.

Como cosas a destacar, se ha utilizado memoria compartida (como se puede ver en el vector declarado en la línea 3), para intentar reducir los accesos a memoria global de device, de manera que cada hebra de un bloque traerá un dato de M a dicho vector compartido.

En cuanto a la fase de reducción, la detallamos de una manera gráfica a continuación:



Es decir, en cada paso del bucle for (línea 15), un número "s" de hebras harán la reducción con el valor correspondiente a su ID dentro del bloque y el desplazado "s" posiciones respecto a dicho valor. Finalmente, es la hebra 0 la que coloca la reducción final obtenida del bloque en el vector M_out (línea 28).

A continuación se muestra también la llamada al kernel:

```
1   /************************ GPU Reduction Phase ********************/
2   t1 = cpuSecond();
3
4   // Copia de host a device
5   err = cudaMemcpy(d_In_M_reduction, h_Out_M_2D, size, cudaMemcpyHostToDevice);
6   if (err != cudaSuccess) cout << "ERROR COPIA REDUCTION" << endl;
7
8   int smemSize = blocksizeReduction*sizeof(int);
9   floyd_reduction<<<nBlocksReduction, blocksizeReduction,
```

```
10 smemSize>>>( d_In_M_reduction, d_Out_M_reduction, nverts*nverts);
11   err = cudaGetLastError();
12
13   if (err != cudaSuccess){
14      fprintf(stderr, "Failed to launch reduction kernel!\n");
15      exit(EXIT_FAILURE);
16   }
17
18   // Copia de device a host del resultado de la reducción
19   cudaMemcpy(h_Out_M_reduction, d_Out_M_reduction, nBlocksReduction*sizeof(int),
20         cudaMemcpyDeviceToHost);
21   cudaDeviceSynchronize();
22
23   int mayor_gpu = h_Out_M_reduction[0];
24   for (int i = 1; i < nBlocksReduction; i++){
25         int val = h_Out_M_reduction[i];
26         mayor_gpu = (val > mayor_gpu) ? val : mayor_gpu;
27   }
28   double Tgpu_red = cpuSecond() - t1;
29   cout << "Tiempo gastado reducción GPU = " << Tgpu_red <<
30         ". Resultado: " << mayor_gpu << endl;
```

El kernel utiliza un grid unidimensional de bloques unidimensionales. La variable **nBlocksReduction** (el número de bloques a lanzar) se calcula a partir de las hebras de cada bloque de la siguiente manera en un punto anterior del código:

```
int nBlocksReduction = ceil(float(nverts2)/blocksizeReduction);
```

Donde blocksizeReduction es el tamaño de cada bloque (se ha establecido en 256) y nverts2 es el número de elementos de la matriz M.
Es decir, repartimos los elementos de esta matriz de manera que cada bloque hará la reducción de “blocksizeReduction” elementos (256).

Finalmente, en el bucle for de la línea 24, se calcula de forma secuencial el mayor elemento de la reducción obtenida por cada bloque.

1.3 Conclusiones.

Respecto a la utilización de un grid bidimensional de bloques bidimensionales contra un grid unidimensional de bloques unidimensionales, podemos ver que se obtiene una ligera mejora utilizando los primeros, obteniendo los mejores tiempos para tamaño de bloque 256 (16 x 16). Obviamente, ambos son muy superiores al tiempo del algoritmo secuencial.

En cuanto al cálculo del mayor camino dentro de los caminos encontramos, podemos destacar que, comparando los tiempos de ejecución del cálculo del camino más largo en GPU y con un algoritmo secuencial, es más rápido el segundo, ya que, aunque en GPU se hace en paralelo, pagamos demasiado coste de sobrecarga por llevar datos de memoria del host a device para las pocas operaciones que hay que hacer con los datos (simplemente una operación "mayor que"), además del tiempo que se pierde al tener que sincronizar las hebras.

2. Implementación CUDA de una operación vectorial.

Se pretende realizar una serie de cálculos utilizando dos vectores de entrada A y B. Estos vectorespodrán ser de longitud variable N 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.

Con estos datos de entrada se calcula:

1. Un vector resultado C con N celdas tal que el valor C[i], i = 0, .. , N-1, se calcula de la siguiente forma:

$$C[i] = \sum_{j \in M_i} (A[j] \cdot i + B[j]), \quad \text{Si} \quad ceil\,(A[j] \cdot i)) \quad \text{es par.}$$

$$C[i] = \sum_{j \in M_i} (A[j] \cdot i - B[j]), \quad \text{Si} \quad ceil\,(A[j] \cdot i) \quad \text{es impar.}$$

   donde M_i es el conjunto de M índices (M es el tamaño de bloque) al que pertenezca la posición i.

2. 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:

$$D[j] = \sum_{i \in B_j} C[j], \qquad j = 1, ..., k$$

   donde B_j es el conjunto de M índices del j-ésimo bloque y k es el número de bloques considerado.

3. El valor máximo mx de todos los valores almacenados en C, independientemente de la división en bloques.

$$mx = \max\{C[i], \quad i = 0...N-1\}.$$

2.1 Implementación en CUDA C.

Se han realizado dos implementaciones CUDA C: una que realiza el cálculo del vector C utilizando variables en memoria compartida y otra que utiliza memoria global para esta fase de cálculo. Para el resto de fases se ha utilizado memoria compartida.

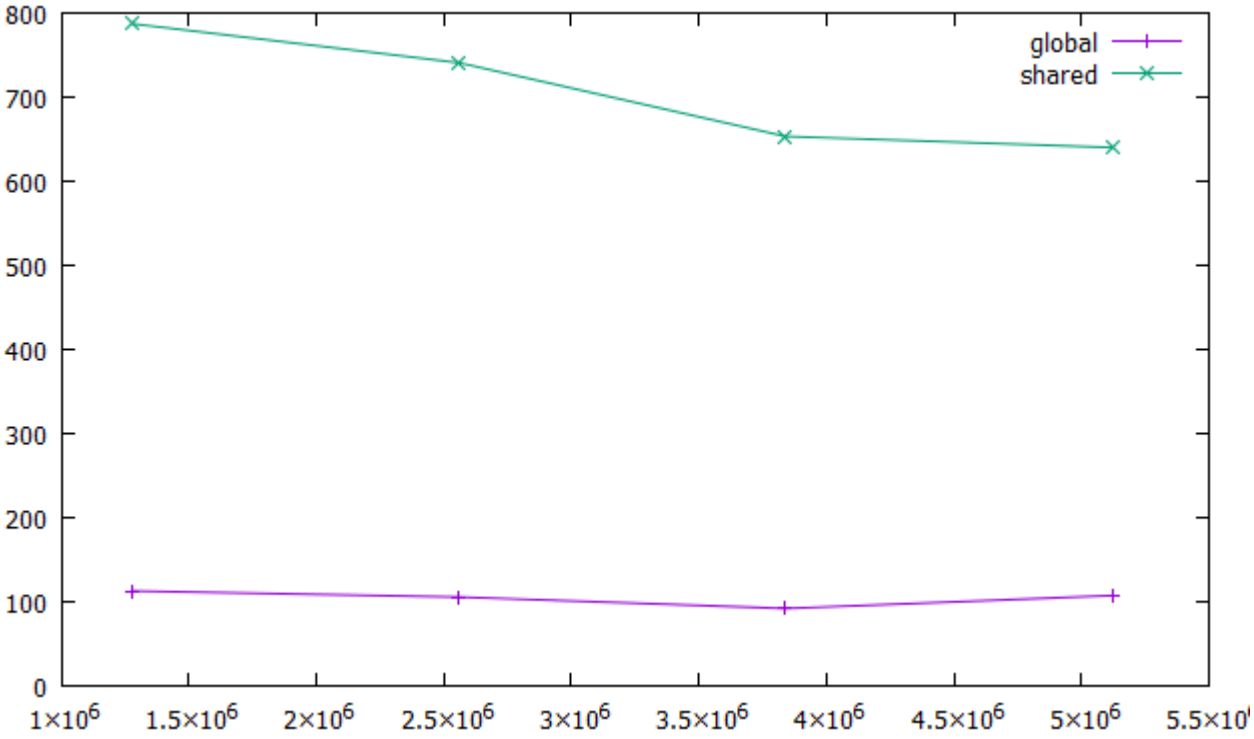
2.2 Resultados.

Se ha utilizado tamaños de vector > 20000 para tres tamaños de bloque: 64, 128 y 256. En las gráficas de Speedup, el eje Y es la ganancia y el eje X el tamaño de vector utilizado.

- Para tamaño de bloque 64:

	TCPU	$TGPU_{GLOBAL}$	$SGPU_{GLOBAL}$	$TGPU_{SHARED}$	$SGPU_{SHARED}$
n = 1280000	1.04998	0.00928211	113,12	0.00133395	787,12
n = 2560000	1.94344	0.018369	105,8	0.00262308	740,9
n = 3840000	2.53859	0.02754	92,18	0.00388503	653,43
n = 5120000	3.28937	0.030575	107,58	0.00513792	640,21

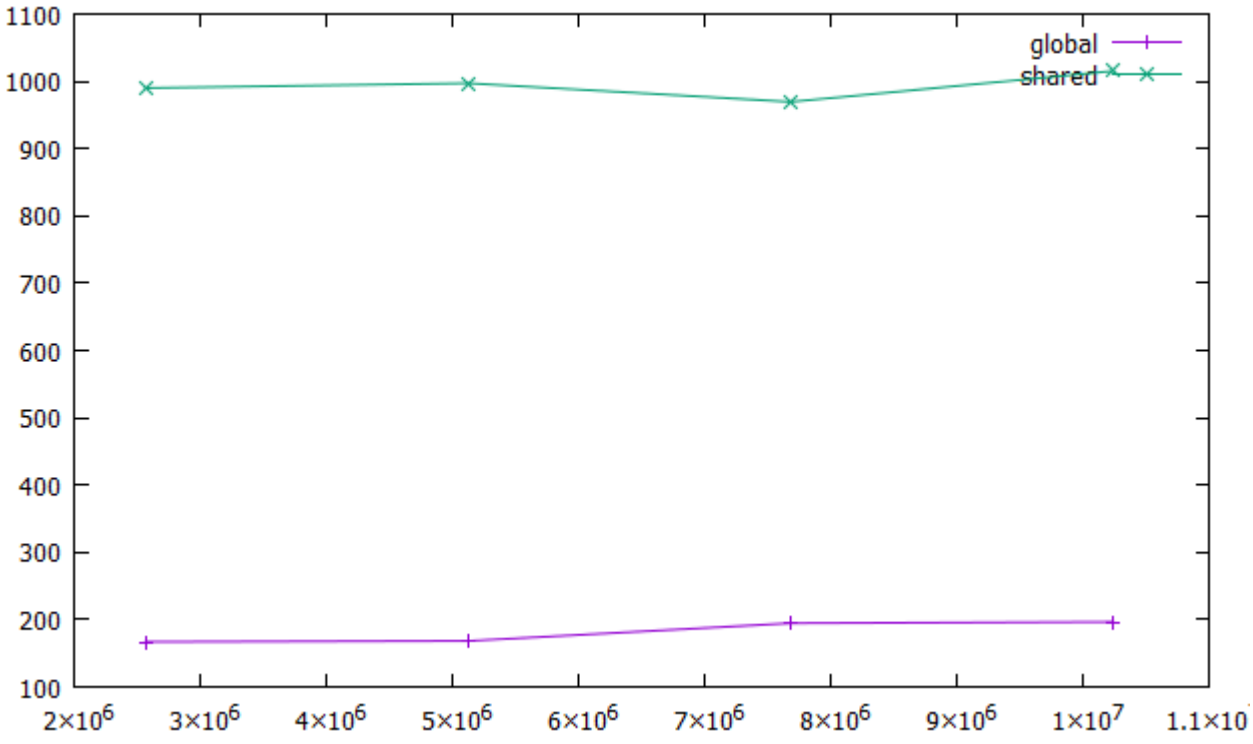
Speedup:



- Para tamaño de bloque 128:

	TCPU	$TGPU_{GLOBAL}$	$SGPU_{GLOBAL}$	$TGPU_{SHARED}$	$SGPU_{SHARED}$
n = 2560000	3.12285	0.0186799	167,18	0.00315404	990,11
n = 5120000	6.25894	0.037133	168,56	0.00627685	997,15
n = 7680000	9.07605	0.0465741	194,87	0.00936103	969,56
n = 10240000	12,15	0.061698	196,9	0.011965	1015.3

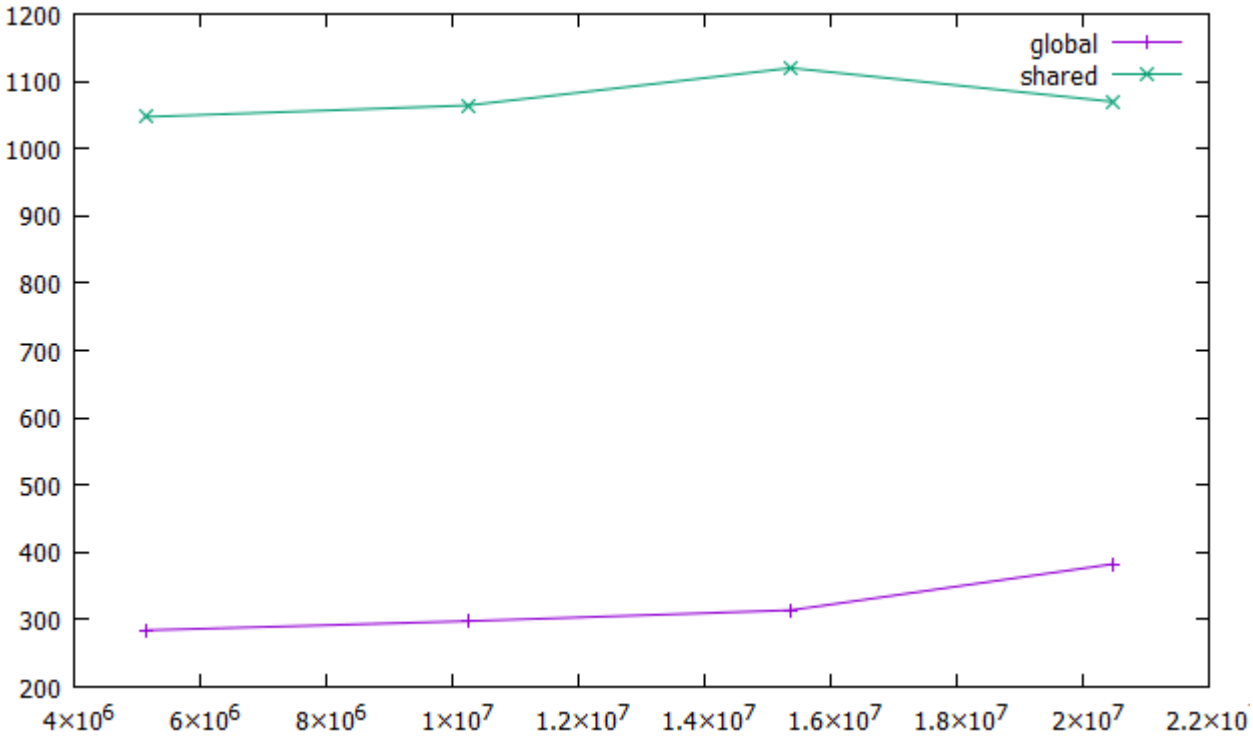
Speedup:



- Para tamaño de bloque 256:

	TCPU	$TGPU_{GLOBAL}$	$SGPU_{GLOBAL}$	$TGPU_{SHARED}$	$SGPU_{SHARED}$
n = 5120000	11,46	0.040308	284,3	0.0109429	1047,23
n = 10240000	23.1881	0.0778022	298,04	0.0217879	1064.27
n = 15360000	36.1017	0.114899	314,2	0.032238	1119.85
n = 20480000	45.9569	0.120045	382,83	0.0429649	1069.64

Speedup:



2.3 Conclusiones.

Como vemos en las distintas gráficas, el tiempo utilizando memoria compartida es muy superior al tiempo utilizando memoria global. Era de esperar, ya que se realizan muchas operaciones aritméticas con los datos de los vectores, por lo que tener un tiempo de acceso corto a los mismos, utilizando memoria compartida, mejora los tiempos de ejecución notablemente.

3. Plataforma de cómputo utilizada y herramientas de compilación.

Se ha utilizado una tarjeta gráfica Nvidia Titan X para realizar las pruebas. Como opciones de compilación se ha utilizado -O3 como nivel de optimización y -m64 para compilar para arquitecturas de 32 y 64 bits.

4. Bibliografía.

1. Apuntes aportados en clase por el profesor José Miguel Mantas Ruiz y notas tomadas por mi.

2. Transparencias de Nvidia sobre como realizar una reducción en CUDA: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf