PPR

Programación Paralela.

Práctica 1: Práctica de Programación de GPUs.

Implementación de algoritmos paralelos de datos en GPU usando CUDA.

Autora: Cristina María Crespo Arco
Correo: cmcrespo@correo.ugr.es

Profesor: José Miguel Mantas Ruiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Curso 2021 - 2022

Índice

1. Información del ordenador que ha realizado las pruebas.

- Sistema Operativo: Ubuntu 22.04
- Procesador: Intel(R) Core(TM) i5-7200U CPU @ 2.50GH
- Memoria RAM: DDR4 con capacidad de 32GB y velocidad de 2133 MT/s
- Tarjeta gráfica: nVidia GM108M [GeForce 920MX] Intel HD Graphics 620
- Versión CUDA: cuda-12.1
- Compilador: nvcc
- Compilación de cada programa:
  - nvcc -I./includes -O3 -m64 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_70,code=compute_70 floyd.cu Graph.cc -o floyd
  - nvcc -I./includes -O3 -m64 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_70,code=compute_70 floyd_mod.cu Graph.cc -o floyd_mod

2. Implementación en CUDA del Algoritmo de Floyd

2.1. Ejercicio 1:

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. Para realizar este ejercicio, se recomienda tomar como referencia, además de la implementación CUDA del algoritmos de Floyd que se os proporciona, la implementación de la suma de matrices cuadradas ya que, en este ejemplo, también se usa una grid bidimensional de bloques bidimensionales.

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 tamaños y diferentes tamaños de bloque CUDA

2.1.1. Modificación en el código.

Para realizar la implementación de este algoritmo he partido del algoritmo *floyd* proporcionado por el profesor y el algoritmo de suma de matrices. Para modificar el algoritmo de *floyd* unidimensional, y obtener un algoritmo bidimensional.

- floyd.cu

```
...
__global__ void floyd_kernel(int * M, const int nverts, const int k) {
  int ij = threadIdx.x + blockDim.x * blockIdx.x;
  int i= ij / nverts;
  int j= ij - i * nverts;
  if (i < nverts && j < nverts) {
    int Mij = M[ij];
    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;
    }
  }
}

int main (int argc, char *argv[]) {
...
  //*****************************************
  // GPU phase ORIGINAL
  //*****************************************

  time=clock();
```

```
  err = cudaMemcpy(d_In_M, A, size, cudaMemcpyHostToDevice);
  if (err != cudaSuccess) {
    cout << "ERROR CUDA MEM. COPY" << endl;
  }

  // Main Loop
  for(int k = 0; k < niters; k++) {
    //printf("CUDA kernel launch \n");
    int threadsPerBlock = blocksize;
    int blocksPerGrid = (nverts2 + threadsPerBlock_original - 1)
                            / threadsPerBlock;
    // Kernel Launch
    floyd_kernel<<<blocksPerGrid,threadsPerBlock>>>(d_In_M, nverts, k);
    err = cudaGetLastError();

    if (err != cudaSuccess) {
      fprintf(stderr, "Failed to launch kernel! ERROR= %d\n",err);
      exit(EXIT_FAILURE);
    }
  }
  err =cudaMemcpy(c_Out_M, d_In_M, size, cudaMemcpyDeviceToHost);
  if (err != cudaSuccess) {
    cout << "ERROR CUDA MEM. COPY" << endl;
  }

  Tgpu=(clock()-time)/CLOCKS_PER_SEC;

  cout << "Time spent on GPU = " << Tgpu << endl << endl;
...
}
```

- floyd_mod.cu

```
__global__ void floyd_kernel(int * M, const int nverts, const int k) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;  // Compute row index
  int i = blockIdx.y * blockDim.y + threadIdx.y;  // Compute column index

  int ij= i * nverts + j; // Compute global 1D index

  if (i < nverts && j < nverts) {
    int Mij = M[ij];
    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;
    }
  }
}
```

```
int main (int argc, char *argv[]) {
...
  //*********************************************
  // GPU phase MODIFICADO
  //*********************************************

  time=clock();

  err = cudaMemcpy(d_In_M, A, size, cudaMemcpyHostToDevice);
  if (err != cudaSuccess) {
    cout << "ERROR CUDA MEM. COPY" << endl;
  }

  // Main Loop
  for(int k = 0; k < niters; k++) {
    //printf("CUDA kernel launch \n");
    dim3 threadsPerBlock_modificado (blocksize_, blocksize_);
    dim3 blocksPerGrid_modificado(ceil((float)(nverts)
              /threadsPerBlock_modificado.x), ceil ((float)
              (nverts)/threadsPerBlock_modificado.y) );
    // Kernel Launch
    floyd_kernel<<<blocksPerGrid_modificado,
               threadsPerBlock_modificado>>>(d_In_M, nverts, k);
    err = cudaGetLastError();

    if (err != cudaSuccess) {
      fprintf(stderr, "Failed to launch kernel! ERROR= %d\n",err);
      exit(EXIT_FAILURE);
    }
  }
  err =cudaMemcpy(c_Out_M, d_In_M, size, cudaMemcpyDeviceToHost);
  if (err != cudaSuccess) {
    cout << "ERROR CUDA MEM. COPY" << endl;
  }

  Tgpu_modificado=(clock()-time)/CLOCKS_PER_SEC;

  cout << "Time spent on GPU modified version= " << Tgpu_modificado
       << endl << endl;
...
}
```

2.1.2. Tabla comparativa.

- Bsize = 64:

	TCPU(sec)	TGPU_orig	SGPU_orig	TGPU_mod	SGPU_mod
N = 400	0.191503	0.030517	6.27529	0.031694	6.04225
N = 1000	2.65976	0.603154	4.40975	1.09407	2.43107
N = 1400	7.41122	1.64142	4.51514	3.45898	2.14261
N = 2000	21.4984	4.76521	4.51154	8.22065	2.61517

  - Gráfica:

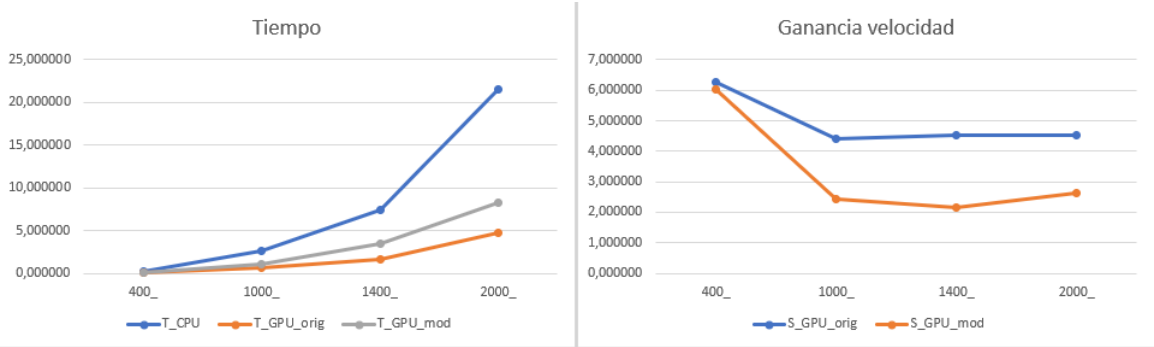

Figura 1: Gráfica 8x8.

- Bsize = 256:

	TCPU(sec)	TGPU_orig	SGPU_orig	TGPU_mod	SGPU_mod
N = 400	0.169656	0.030582	5.54758	0.021359	7.94307
N = 1000	2.66056	0.602336	4.41707	0.620597	4.2871
N = 1400	7.38411	1.63955	4.50375	1.7207	4.29134
N = 2000	21.4706	4.76271	4.50807	4.83932	4.4367

  - Gráfica:

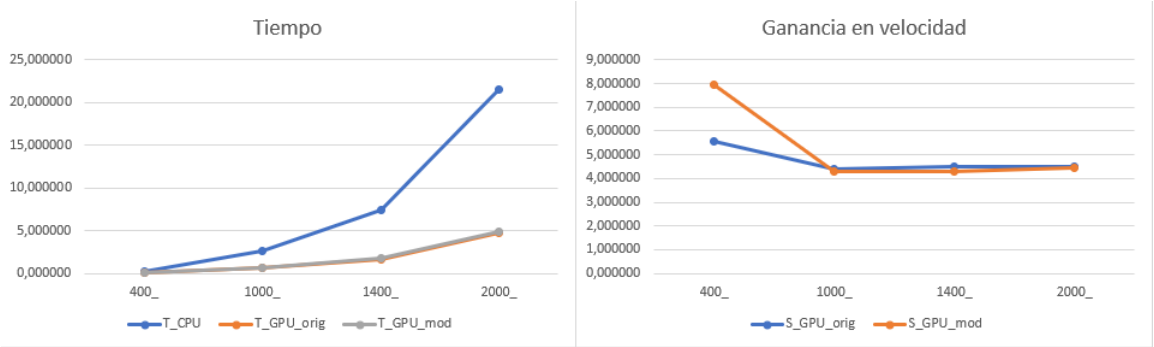

Figura 2: Gráfica 16x16.

- Bsize = 1024:

	TCPU(sec)	TGPU_orig	SGPU_orig	TGPU_mod	SGPU_mod
N = 400	0.169904	0.031824	5.33886	0.023241	7.31053
N = 1000	2.66404	0.619768	4.29844	0.658723	4.04424
N = 1400	7.43444	1.68476	4.41276	1.84124	4.03773
N = 2000	21.5584	4.90346	4.39657	5.06706	4.25462

  - Gráfica:

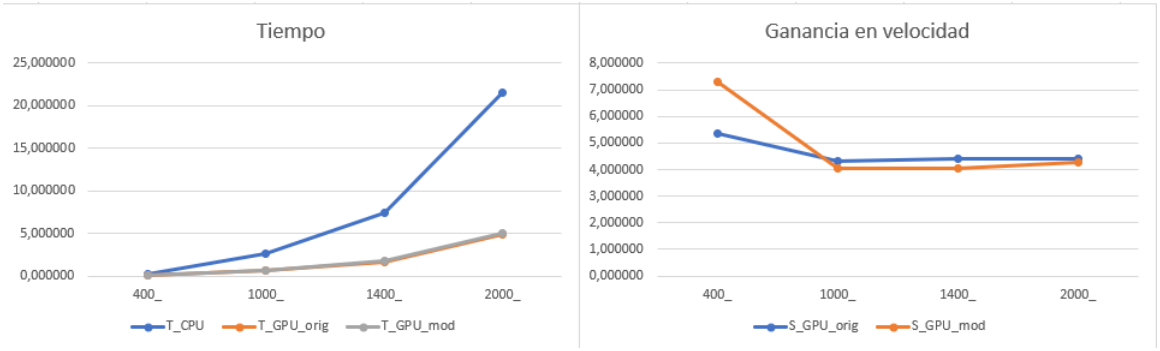

Figura 3: Gráfica 32x32

- Gráfica comparativa ganancia de velocidad:

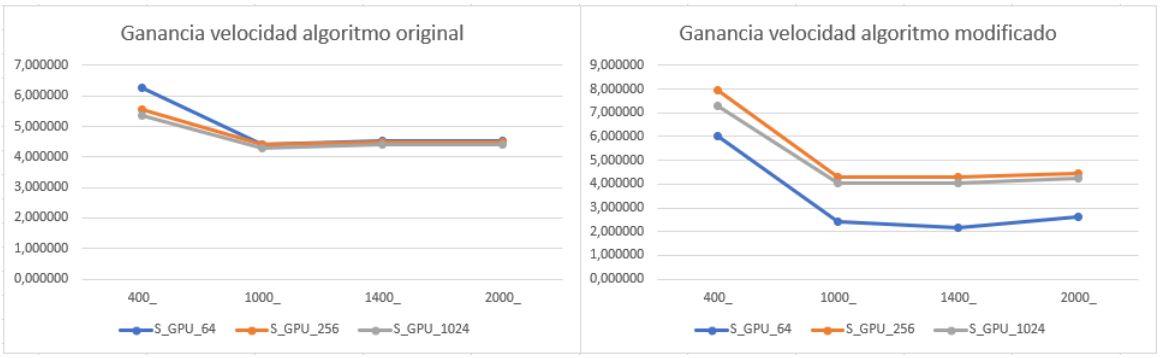

Figura 4: Gráfica ganancia velocidad

2.2. Ejercicio 2:

Extender la implementación en CUDA C desarrollada para que se calcule también la media aritmética de todos los caminos más cortos encontrados. Usar para ello un kernel CUDA de reducción aplicado al vector que almacena los valores de la matriz resultado. Se pueden usar los kernels de reducción disponibles en las CUDA Samples: `https://github.com/nvidia/cuda-samples` o usar el kernel de reducción que se muestra en las diapositivas y en la carpeta "Ejemplo_suma_vectores_reduccion" de los códigos CUDA que se os proporciona.

También se recomienda, al realizar la suma definitiva en el host, usar una variable destino de tipo long int para evitar desbordamientos.

2.2.1. Modificación en el código.

Para realizar la implementación de este algoritmo he partido del algoritmo para calcular el mínimo que hice para el ejercicio de clase.

He modificado el algoritmo para que realice una suma en lugar de calcular el mínimo. Para ello he aplicado un algoritmo de reducción, y usando un vector de memoria compartida.

Una vez finalizada la suma de todas las hebras de cada bloque se sincronizan las hebras y se copia la información del vector de memoria compartida en el vector de salida.

Una vez ha finalizado la ejecución del kernel, para calcular la media se realiza una división en el host del valor obtenido tras el algoritmo de reducción y el número total de elementos del vector *nverts2*.

- floyd_mod_red.cu

```
//***** KERNEL REDUCCION *****
__global__ void floyd_reduccion(int *M, long int *suma,
                                const int nverts2) {
extern __shared__ float sdata[];

int tid = threadIdx.x;
int i = blockIdx.x * blockDim.x + threadIdx.x;
sdata[tid] = ((i < nverts2) ? M[i] : 0.0f);
__syncthreads();

for(int s = blockDim.x/2; s > 0; s >>= 1){
   if (tid < s)
     sdata[tid] += sdata[tid+s];
   __syncthreads();
}
if (tid == 0)
    suma[blockIdx.x] = sdata[0];
}
```

```
int main(int argc, char *argv[]){
  ...
  //***** REDUCCION GPU *****
  dim3 threadsPerBlock_reduccion(blocksize, 1);
  dim3 numBlocks_reduccion(ceil ((float)(nverts2) /
                           threadsPerBlock_reduccion.x), 1);

  // Vector suma en CPU
  long int *suma_GPU;
  suma_GPU = (long int*) malloc(numBlocks_reduccion.x*sizeof(long int));

  // Minimum vector  to be computed on GPU
  long int *suma_GPU_d;
  cudaMalloc((void **) &suma_GPU_d, sizeof(float)*numBlocks_reduccion.x);

  int smemSize_reduccion = threadsPerBlock_reduccion.x*sizeof(float);

  time = clock();
  // Llamada al kernel de reduccion
  floyd_reduccion<<<numBlocks_reduccion, threadsPerBlock_reduccion,
                 smemSize_reduccion>>>(d_In_M, suma_GPU_d, nverts2);

  Tgpu_reduccion = (clock() - time) / CLOCKS_PER_SEC;

  /* Copy data from device memory to host memory */
  cudaMemcpy(suma_GPU, suma_GPU_d, numBlocks_reduccion.x*sizeof(float),
              cudaMemcpyDeviceToHost);

  // Perform final reduction in CPU
  long int sumaGPU = 0;
  float mediaGPU;
  for (int i = 0; i < numBlocks_reduccion.x; i++)
    sumaGPU += suma_GPU[i];
  mediaGPU = sumaGPU / nverts2;
  ...
}
```

2.2.2. Tabla comparativa.

- Bsize = 64:

	TGPU(sec)	TCPU	SGPU = TCPU/TGPU
N = 400	0.000068	0.000038	0.558824
N = 1000	0.000015	0.000319	21.2667
N = 1400	0.000016	0.00064	40
N = 2000	0.000019	0.001312	69.0526

  - Gráfica:

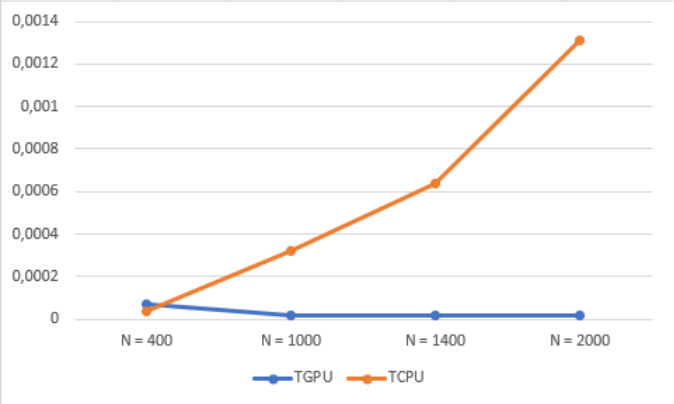

Figura 5: Gráfica 16.

- Bsize = 256:

	TGPU(sec)	TCPU	SGPU = TCPU/TGPU
N = 400	0.00001	0.000053	5.3
N = 1000	0.00002	0.000334	16.7
N = 1400	0.000018	0.000659	36.6111
N = 2000	0.000018	0.00131	72.7778

  - Gráfica:

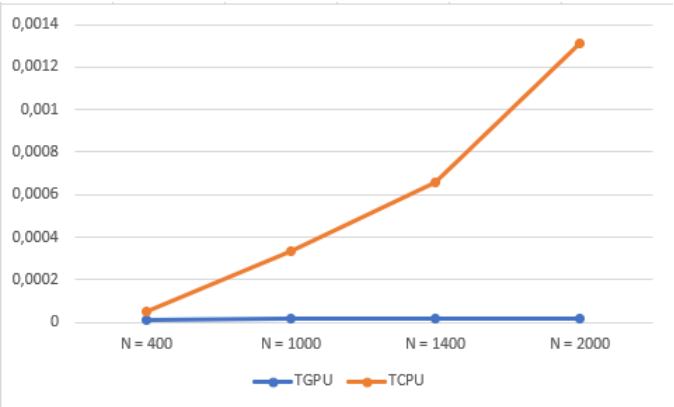

Figura 6: Gráfica 256.

- Bsize = 1024:

	TGPU(sec)	TCPU	SGPU = TCPU/TGPU
N = 400	0.000009	0.000037	4.11111
N = 1000	0.000014	0.000322	23
N = 1400	0.000017	0.00065	38.2353
N = 2000	0.000017	0.002135	125.588

  - Gráfica:

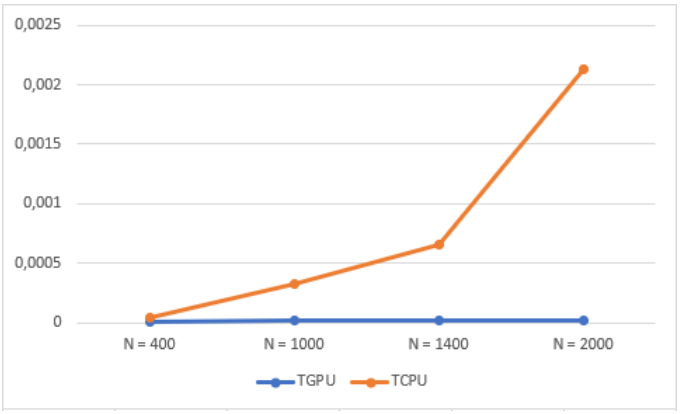

Figura 7: Gráfica 1024

- Gráfica comparativa ganancia de velocidad:

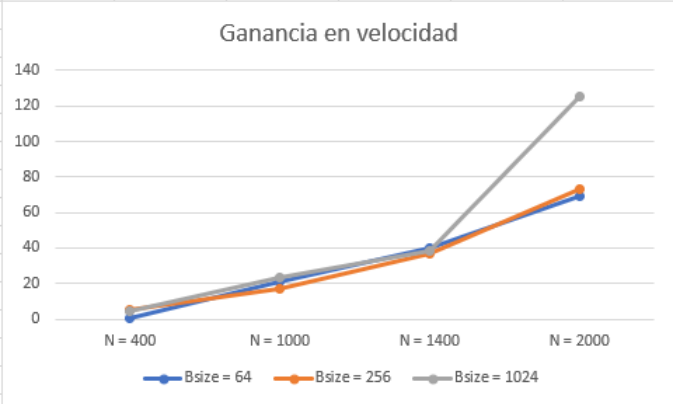

Figura 8: Gráfica ganancia velocidad

3. Implementación CUDA de una operación vectorial

3.1. Ejercicio 1:

Se deberán 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 un vector en memoria compartida ya que es necesario para realizar reducciones a nivel de bloque de forma eficiente en CUDA.

3.1.1. Implementación del código.

- Cálculo del vector C:
  - Implementación en CPU:

```
int main(){
  time = clock();

  for (int i = 0; i < N; i++) {
    C_CPU[i] = 0.0;
    int bloqueActual = ceil(i/M);
    for (int j = bloqueActual * M; j < ((bloqueActual * M) + M); j++) {
      C_CPU[i] += (i*B[j] - A[j]*A[j]) / ((i+2) * (max(A[j], B[j])));
    }
  }

  Tcpu_ej1 = (clock() - time) / CLOCKS_PER_SEC;
}
```

  - Implementación en GPU: Para calcular el valor del vector de C, he usado una hebra CUDA para calcular cada elemento del vector. Por tanto, cada hebra realizaba un bucle *for* que se encarga de recorrer los elementos de cada bloque de hebras al que pertenecía la hebra, y sumar el valor calculado al que ya había almacenado en el vector tenía el elemento del vector.

    Antes de realizar la copia del vector del host en memoria compartida he inicializado el vector C a 0.0 para asegurar que la suma calculada en CUDA no tomaba valores incorrectos.

```
//********* KERNEL PRIMER PUNTO *********
__global__ void kernel_calcularC(float *A, float *B, float *C, const int N) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  for (int j = (blockIdx.x * blockDim.x); j < ((blockIdx.x * blockDim.x)
             + blockDim.x); j++) {
    C[i] += (i*B[j] - A[j]*A[j]) / ((i+2) * (max(A[j], B[j])));
  }
}
```

```
int main(){
  for (int i = 0; i < N; i++)
    C[i] = 0.0;

  /* Copiar los datos de los vectores en el host a los
      vectores en memoria compatida */
  cudaMemcpy(C_d, C, sizeof(float)*N, cudaMemcpyHostToDevice);

  dim3 threadsPerBlock(M, 1);
  dim3 numBlocks(NBlocks, 1);

  int smemSize = threadsPerBlock.x*sizeof(float);

  Tgpu_total = clock();
  time = clock();

  /*Llamada al kernel*/
  kernel_calcularC<<<numBlocks, threadsPerBlock>>>(A_d, B_d, C_d, N);
  cudaDeviceSynchronize();

  Tgpu_ej1 = (clock() - time) / CLOCKS_PER_SEC;

  /* Copiar los datos de la memoria compartida a la memeoria del host */
  cudaMemcpy(C, C_d, N*sizeof(float),cudaMemcpyDeviceToHost);
}
```

- Implementación en GPU con variables en memoria compartida: Para la implementación del algoritmo para calcular el vector C mediante el uso de variables en memoria compartida he usado un vector en memoria principal donde se va a almacenar el resultado de la suma.

  Al principio, se inicializa el vector entero a 0.0 y se sincronizan las hebras. A continuación, cada hebra realiza la suma en el vector de memoria compartida y se vuelven a sincronizar las hebras. Por último, se almacena la información del vector en memoria compartida *sdata* en el vector de salida *C*.

```
//***** KERNEL CON VARIABLES DE MEMORIA COMPARTIDA *****
__global__ void kernel_calcularC_mc(float *A, float *B, float *C,
                                    const int N) {
  extern __shared__ float sdata[];
  int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[i] = 0.0f;

  __syncthreads();

  for (int j = (blockIdx.x * blockDim.x); j < ((blockIdx.x * blockDim.x)
              + blockDim.x); j++) {
    sdata[i] += (i*B[j] - A[j]*A[j]) / ((i+2) * (max(A[j], B[j])));
```

```
    __syncthreads();
  }

  C[i] = sdata[i];
}

int main(){
  dim3 threadsPerBlock(M, 1);
  dim3 numBlocks(NBlocks, 1);

  int smemSize = threadsPerBlock.x*sizeof(float);

  /* VARIABLES MEMORIA COMPARTIDA */
  Tgpu_total_mc = clock();
  time = clock();

  /*Llamada al kernel*/
  kernel_calcularC_mc<<<numBlocks, threadsPerBlock, smemSize>>>(A_d, B_d,
                                                                C_mc_d, N);
  cudaDeviceSynchronize();

  Tgpu_ej1_mc = (clock() - time) / CLOCKS_PER_SEC;

  /* Copiar los datos de la memoria compartida a la memeoria del host */
  cudaMemcpy(C_mc, C_mc_d, N*sizeof(float),cudaMemcpyDeviceToHost);
}
```

- Calcular el máximo del vector C:
  - Implementación en CPU:

```
int main(){
  time = clock();

  float maximo_CPU = -100000.0f;
  for (int i = 0; i < N; i++) {
    maximo_CPU = max(maximo_CPU,C_CPU[i]);
  }

  Tcpu_ej2 = (clock() - time) / CLOCKS_PER_SEC;
}
```

- Implementación en GPU: Para realizar la implementación de este algoritmo he partido del algoritmo que realizamos en el ejercicio de clase para calcular el mínimo de un vector. Lo he modificado para que en lugar de calcular el mínimo calcule el valor máximo mediante un algoritmo de reducción, usando un vector de memoria compartida.

```
//***** KERNEL SEGUNDO PUNTO *****
__global__ void kernel_calcularMaximo(float * V_in, float * V_out,
                                      const int N) {
  extern __shared__ float sdata[];

  int tid = threadIdx.x;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int i_seg =  gridDim.x * blockDim.x + i;

  float segundo_valor;

  sdata[tid] = ((i < N) ? V_in[i] : -100000000.0f);
  segundo_valor = ((i_seg < N) ? V_in[i] : -100000000.0f);

  sdata[tid] = max(sdata[tid], segundo_valor);
  __syncthreads();

  for(int s = blockDim.x/2; s > 0; s >>= 1){
    if (tid < s)
      sdata[tid]=max(sdata[tid],sdata[tid+s]);
    __syncthreads();
  }

  if (tid == 0)
    V_out[blockIdx.x] = sdata[0];
}

int main(){
  dim3 threadsPerBlock(M, 1);
  dim3 numBlocks(NBlocks, 1);

  int smemSize = threadsPerBlock.x*sizeof(float);

  time = clock();

  /*Llamada al kernel*/
  kernel_calcularMaximo<<<numBlocks, threadsPerBlock, smemSize>>>(C_d,
                                                                  vmax_d, N);

  Tgpu_ej2 = (clock() - time) / CLOCKS_PER_SEC;

  /* Copiar los datos de la memoria compartida a la memeoria del host */
```

```
  cudaMemcpy(vmax, vmax_d, numBlocks.x*sizeof(float),cudaMemcpyDeviceToHost);

  float maximo_GPU = -10000000.0f;
  for (int i=0; i<numBlocks.x; i++){
    maximo_GPU = max(maximo_GPU,vmax[i]);
  }
}
```

- Cálculo del vector D:
  - Implementación en CPU:

```
int main(){
  time = clock();

  int indice;
  for (int k = 0; k < NBlocks; k++) {
    D_CPU[k] = 0.0;
    for (int j = 0; j < M; j++) {
      indice = k * M + j;
      D_CPU[k] +=  C_CPU[indice];
    }
    D_CPU[k] /= M;
  }

  Tcpu_ej3 = (clock() - time) / CLOCKS_PER_SEC;
}
```

  - Implementación en GPU: Para realizar la implementación de este algoritmo he partido del algoritmo de suma de un vector que realice de la relación de ejercicios de CUDA.

    Para calcular la media he añadido la división al final del kernel, una vez se han sincronizado las hebras, del valor obtenido en cada bloque entre el número de hebras del bloque. Al igual que en el apartado anterior he aplicado un algoritmo de reducción, y usando un vector de memoria compartida.

```
//***** KERNEL TERCER PUNTO *****
__global__ void kernel_calcularMediaBloque(float * V_in, float * V_out,
                                           const int N) {
  extern __shared__ float sdata[];
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[tid] = ((i < N) ? V_in[i] : 0.0f);

  __syncthreads();

  for (int s=blockDim.x/2; s>0; s>>=1) {
```

```
    if (tid < s)
      sdata[tid] += sdata[tid + s];

    __syncthreads();
  }

  if (tid == 0)
    V_out[blockIdx.x] = sdata[0] / blockDim.x;
}

int main(){
  dim3 threadsPerBlock(M, 1);
  dim3 numBlocks(NBlocks, 1);

  int smemSize = threadsPerBlock.x*sizeof(float);

  time = clock();

  /*Llamada al kernel*/
  kernel_calcularMediaBloque<<<numBlocks, threadsPerBlock, smemSize>>>(C_d,
                                                                       D_d, N);

  Tgpu_ej3 = (clock() - time) / CLOCKS_PER_SEC;

  /* Copiar los datos de la memoria compartida a la memeoria del host */
  cudaMemcpy(D, D_d, numBlocks.x*sizeof(float),cudaMemcpyDeviceToHost);

  Tgpu_total = (clock() - Tgpu_total) / CLOCKS_PER_SEC;
}
```

3.2. Ejercicio 2:

Se deberán comparar los resultados obtenidos, tanto en tiempo de ejecución como en la precisión de los valores obtenidos con respecto al código CPU (disponible en el archivo transformacion.cc) solo para el cálculo del vector C. Para ello, se utilizarán valores muy altos de N (N $<$ 2000000), probando los tres tamaños de bloque indicados al inicio de este ejercicio y usando las versiones con y sin memoria compartida para el cálculo del vector C. En los tiempos tomados para el cálculo del vector C en GPU, solo se medirá el tiempo tomado en la ejecución del kernel de cálculo de C, por lo que habrá que invocar la función "cudaDeviceSynchronize()" justo después de invocar el kernel correspondiente y antes de tomar el instante de tiempo final.

3.2.1. Tabla comparativa.

- M = 64:
  - Tiempos del cálculo de C:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	0.400916	0.019675	0.000547
N = 4096000	0.823214	0.039379	0.000542
N = 20480000	4.02772	0.1964	0.000208
N = 64000000	12.9685	0.607559	0.000227

    - Gráfica:

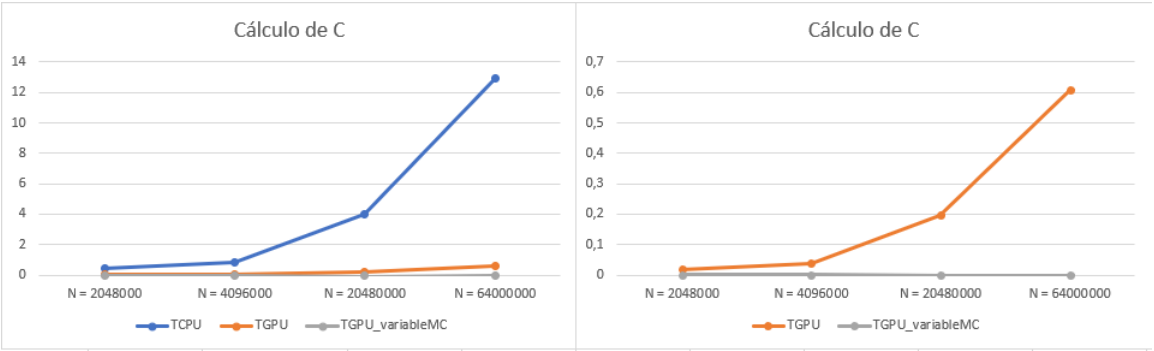

Figura 9: Gráfica M=64, cálculo de C.

- Tiempo total:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	0.402457	0.029196	0.00056
N = 4096000	0.826262	0.058251	0.000551
N = 20480000	4.04289	0.288174	0.00022
N = 64000000	13.0157	0.89321	0.000242

  - Gráfica:

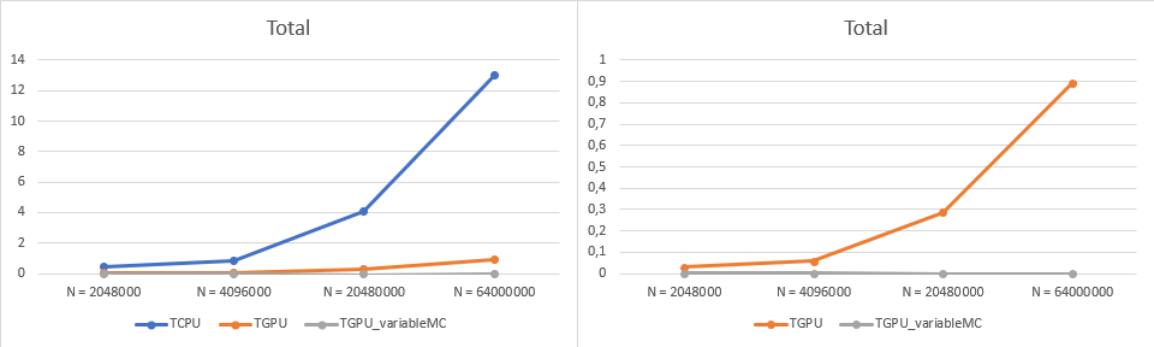

Figura 10: Gráfica M=64, tiempo total.

- M = 128:

  - Tiempos del cálculo de C:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	0.784609	0.037587	0.000529
N = 4096000	1.59865	0.075169	0.000549
N = 20480000	7.89135	0.365303	0.000219
N = 64000000	24.5876	1.15495	0.000551

  - Gráfica:

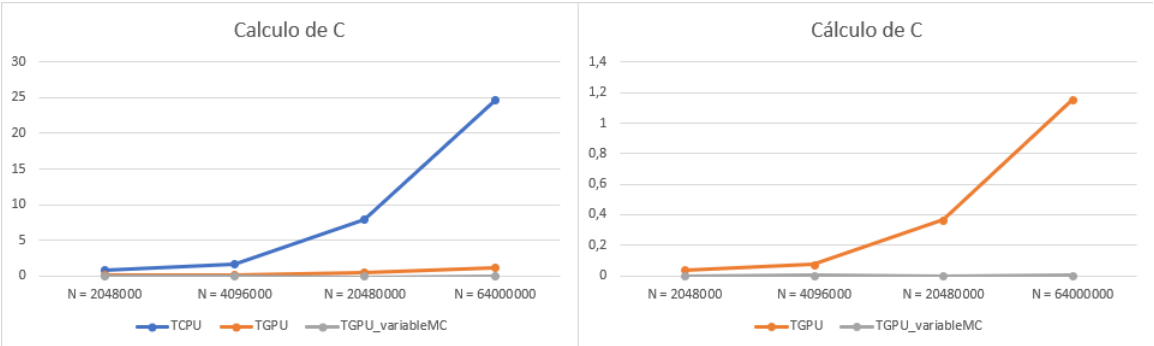

Figura 11: Gráfica M=128, cálculo de C.

- Tiempo total:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	0.786491	0.046954	0.000536
N = 4096000	1.60241	0.093775	0.00056
N = 20480000	7.91026	0.456185	0.000228
N = 64000000	24.6462	1.43849	0.000562

  - Gráfica:

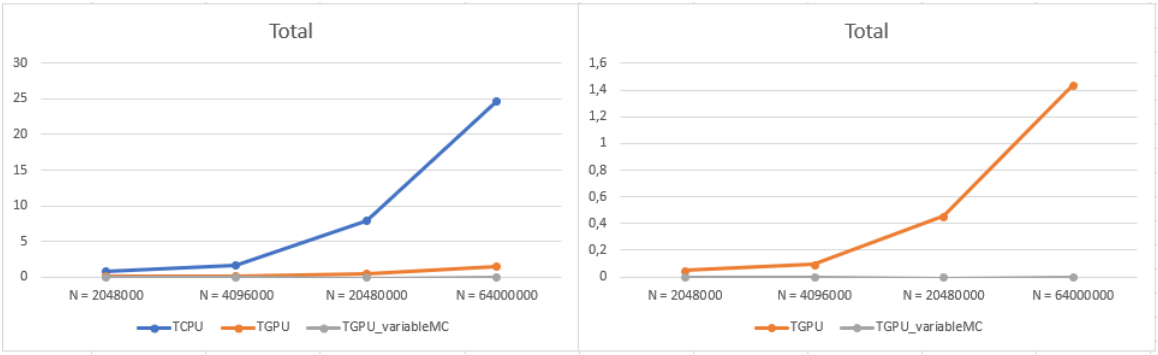

Figura 12: Gráfica M=128, tiempo total.

- M = 256:

  - Tiempos del cálculo de C:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	1.58258	0.072594	0.000539
N = 4096000	3.11323	0.145168	0.000551
N = 20480000	15.5998	0.709811	0.000538
N = 64000000	48.6513	2.22198	0.000563

  - Gráfica:

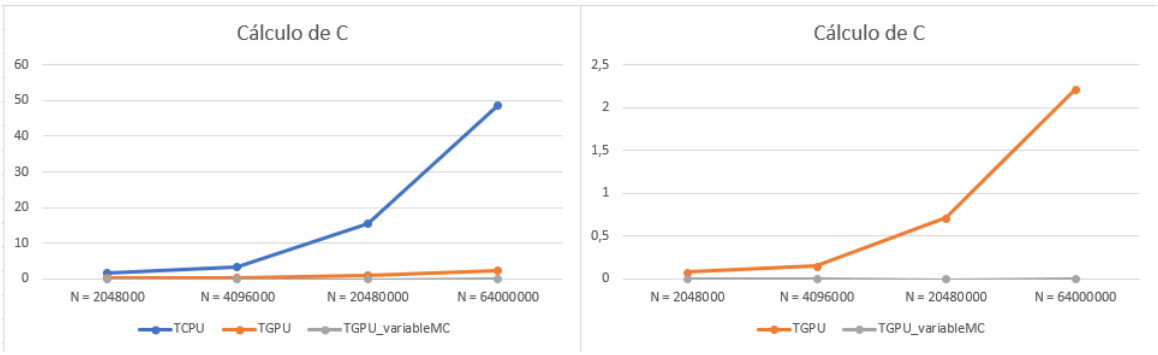

Figura 13: Gráfica M=256, cálculo de C.

- Tiempo total:

	TCPU	TGPU	TGPU_variableMC
N = 2048000	1.58492	0.082356	0.000547
N = 4096000	3.11786	0.164208	0.000561
N = 20480000	15.6229	0.804521	0.000548
N = 64000000	48.7236	2.51758	0.000575

  - Gráfica:

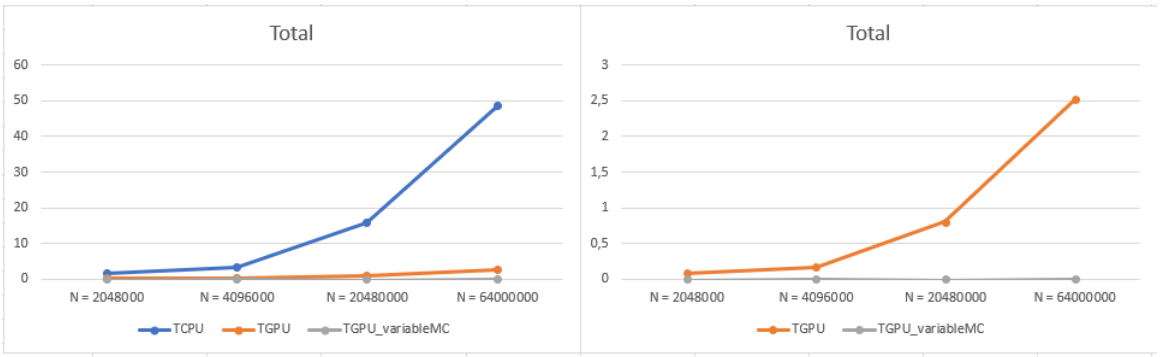

Figura 14: Gráfica M=256, tiempo total.