SISTEMAS COMPLEJOS EN MÁQUINAS PARALELAS

TRABAJO PRÁCTICO 4

PROGRAMACIÓN EN PARALELO EMPLEANDO EL LENGUAJE CUDA (NVIDIA)

Estudiante

FREDY ANDRÉS MERCADO NAVARRO
DNI: 94.872.342
Maestría en Simulación Numérica y Control
Docentes: Guillermo Marshall, Esteban Mocskos.
Cuatrimestre: I-2012
17 de Octubre

Universidad de Buenos Aires
Ciudad Autónoma de Buenos Aires
Argentina
2012

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1. INTRODUCCIÓN

El presente trabajo busca aprovechar la capacidad de cómputo de una tarjeta gráfica Nvidia habilitada para realizar cálculos numéricos. El lenguaje de programación desarrollado por Nvidia para este propósito recibe el nombre de CUDA. Este lenguaje de programación será aplicado para calcular la Norma Frobenius de una matriz de números reales.

El cálculo de la Norma Frobenius consiste en hallar la raíz cuadrada de la suma de los cuadrados de todos los términos de una matriz. Dado que aquellos números que son cero no aportan al cálculo, se decidió emplear como archivo de entrada un archivo de texto que contiene todos los términos de la matriz que son diferentes de cero. El formato que posee el archivo de entrada corresponde al del Matrix Market (.mtx) [1]. Este formato suministra tres columnas. La primera corresponde a una fila, la segunda a una columna y la tercera al valor de esa posición de la matriz.

En términos generales, los dos programas desarrollados leen la matriz fila por fila del archivo .mtx, elevan esos términos al cuadrado, los suman, y por último obtienen la raíz cuadrada de dicha suma.

2. DEFINICIÓN DEL PROBLEMA

El trabajo práctico consiste en encontrar la norma Frobenius de una matriz. Dada una matriz A de m x n, la norma Frobenius se puede definir como:

$$\left||A|\right|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$$

Se deberá realizar el programa secuencial en Matlab o C para usar como comprobación de resultado. Luego se deberá implementar el cálculo de la norma utilizando el lenguaje CUDA.

Se deberá diseñar el algoritmo de manera tal que se pueda utilizar variando las distribuciones de bloques e hilos en la placa. En cada caso, deberá buscarse utilizar adecuadamente los recursos de la placa, optimizando el acceso a memoria.

Se deberá presentar un informe y el código generado (en formato electrónico, puede ser vía mail). El informe debe contener la descripción del o los programas entregados y de la arquitectura elegida para implementar el paralelismo, dificultades encontradas, detalles de uso y conclusiones. Se deberán mostrar las curvas de escalabilidad y se deberá analizar el impacto de la forma de acceso a memoria elegido mediante la variación de la distribución de hilos y bloques.

3. DISEÑO DEL ALGORITMO SERIAL

Como parte del trabajo práctico se debió diseñar un programa que calculara en forma secuencial. Esto con el objeto de utilizar sus resultados y compararlos numéricamente con los resultados de la Norma Frobenius calculada por el programa paralelo escrito en lenguaje CUDA.

3.1 Descripción general del programa

El algoritmo serial lee un archivo en formato .mtx en el cual está almacenada la información de la matriz a la cual se le calculará la Norma Frobenius. La columna de los valores de la matriz es almacenada en un arreglo unidimensional, el cual es la base para los cálculos del ciclo for en el cual se suman los cuadrados de cada posición del arreglo. Por último se calcula la raíz cuadrada de esa suma. Dado que la medición de tiempos debe ser lo más precisa posible, el ciclo for se repite 1000 veces y el tiempo medido con la función MPI_Wtime es dividido sobre 1000 para hallar un valor de cómputo promedio. Este valor será empleado para evaluar la escalabilidad del código una vez se midan los tiempos con el programa paralelo.

3.2 Código desarrollado

A continuación se presenta el código serial desarrollado para calcular la Norma Frobenius de una matriz de entrada en el formato del Matrix Market:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<mpi.h>

#define REP 1000

int main(int argc, char **argv) {

  int a, b, i, j, NZA;
  double valor, suma, norma, *vector;
  double tinicial[5], tfinal[5], tiempo[5];

  sscanf(argv[2], "%d", &NZA);

  vector = calloc(NZA, sizeof(double));

  MPI_Init(&argc, &argv);

  //LECTURA DE DATOS
        tinicial[0] = MPI_Wtime();
  FILE *file;
  file = fopen(argv[1], "r");

  for (i=0; i<NZA; i++) {
    fscanf(file, "%d %d %lf\n", &a, &b, &valor);
    vector[i] = valor;
```

```
  }

  fclose(file);
        tfinal[0] = MPI_Wtime();

  //CALCULOS
  printf("\nCalculando...\n");

        tinicial[1] = MPI_Wtime();
  for (j=0; j<REP; j++) {
    suma = 0.0;
    for (i=0; i<NZA; i++)
      suma += vector[i] * vector[i];

    norma = sqrt(suma);
  }
        tfinal[1] = MPI_Wtime();

  tiempo[0] = tfinal[0] - tinicial[0];
  tiempo[1] = (tfinal[1] - tinicial[1])/REP;

  printf("Suma de cuadrados = %.15f\n", suma);
  printf("Norma Frobenius = %.15f\n\n", norma);

  printf("Tiempo lectura [0] = %f s\n", tiempo[0]);
  printf("Tiempo cálculos [1] = %f s\n", tiempo[1]);

  MPI_Finalize();

  free(vector);

  return 0;
}
```

3.3 Compilación y ejecución

El programa se estuvo compilando con la siguiente línea de comandos bajo el sistema operativo Linux-Ubuntu:

- Compilar: mpicc -o tp4Serial5.out tp4Serial5.c -lm -Wall

Las pruebas se estuvieron ejecutando con la siguiente línea de comandos:

- Correr: mpirun -np 1 ./tp4Serial5.out matrizA9.mtx NZA

Ejemplo: mpirun –np 1 -/tp4Serial5.out matrizA9.mtx 863353

Es necesario clarificar que MPI es empleado sólo para la medición de tiempos.

3.4 Prueba del algoritmo secuencial

Al ejecutar el programa como lo muestra el ejemplo en el numeral anterior, los resultados obtenidos fueron:

- Suma de cuadrados = 0.005434614921759.
- Norma Frobenius = 0.07371984076054.
- Tiempo cálculos = 0.002893 segundos.

Tiempo cálculos es un tiempo que se tomó entre el inicio del ciclo for para la suma de cuadrados y el cálculo de la raíz cuadrada de dicha suma. Este tiempo no tiene en cuenta los tiempos de lectura de la matriz A.

Para la prueba del algoritmo secuencial tanto el archivo de entrada como el número de posiciones diferentes de cero (Non-zeros) son ingresados a través de línea de comandos en la terminal.

4. DISEÑO DEL ALGORITMO PARALELO CON LENGUAJE CUDA

4.1 Descripción general del programa

El código generado calcula la Norma Frobenius de una matriz de cualquier dimensión, cuyo archivo de entrada contenga todos los valores de la matriz que son diferentes de cero. El programa lee un archivo con extensión .mtx y lo guarda en el arreglo de nombre "valor". Una vez este arreglo es copiado al dispositivo, los hilos de cada bloque computan el cuadrado de cada posición y lo suman. Esta suma es función también de la cantidad de hilos por bloque, de la Id de cada bloque y de la Id o posición de cada hilo dentro de su bloque. La suma de cada hilo es almacenada en un arreglo que está compartido entre todos los hilos de un bloque (él único arreglo compartido que contiene el programa) de nombre "cache". El paso siguiente consiste en reducir todos los valores del arreglo caché a la posición cero. Una vez todos los hilos de todos los bloques de la grid van terminando el cálculo, la suma resultante de cada bloque es asignada a una posición del arreglo "c".

El paso siguiente consiste en copiar el arreglo c al Host y allí reducir nuevamente todas las posiciones de c a una sola, para luego calcular su raíz cuadrada y terminar con el cálculo de la Norma. Este paso se desarrolló siguiendo la recomendación de la referencia [4], la cual comenta que una máquina masivamente paralela, como una GPU, tiende a malgastar sus recursos al realizar los últimos pasos de una reducción.

La función imprimetxt2 no está habilitada y sólo se empleó durante el proceso de evaluación del algoritmo. La función leeA se encarga de asignar todos los valores del archivo matrizA9.mtx al arreglo valor. La función SumaCuadrados es la función que se ejecuta en el dispositivo, y depende del número de hilos por bloque y del número de bloques por grid.

4.2 Código desarrollado

Al igual que el código serial, el código paralelo posee como entrada una matriz en el formato del Matrix Market. El código diseñado se observa a continuación:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<cuda.h>

const int NZA = 863353;
const int blocksPerGrid = 1; //max: 65535.
const int threadsPerBlock = 128; //max: 1024.
        //Necesariamente Potencia de 2.
        //Opcional múltiplo de 32 (1 warp).
        //Para dividir/2 durante suma.

void leeA(char *filename, double *valor, int NZA);
void imprimetxt2(char *, double *, int);
__global__ void SumaCuadrados(double *valor, double *c);

int main(void) {
```

```
  double *valor, c, *partial_c;
  double *dev_valor, *dev_partial_c;
  int i;

  cudaEvent_t start, stop;
  cudaEvent_t start_cpy1, stop_cpy1;
  cudaEvent_t start_SumaCuadrados, stop_SumaCuadrados;
  cudaEvent_t start_cpy2, stop_cpy2;

  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventCreate(&start_cpy1);
  cudaEventCreate(&stop_cpy1);
  cudaEventCreate(&start_SumaCuadrados);
  cudaEventCreate(&stop_SumaCuadrados);
  cudaEventCreate(&start_cpy2);
  cudaEventCreate(&stop_cpy2);

  //Asignamos memoria a CPU
  valor = (double*)calloc(NZA, sizeof(double));
  partial_c = (double*)calloc(blocksPerGrid, sizeof(double));

  //Asignamos memoria a GPU
  cudaMalloc((void**)&dev_valor, NZA*sizeof(double));
  cudaMalloc((void**)&dev_partial_c,
blocksPerGrid*sizeof(double));

  //Llenamos memoria en host con datos
  leeA("matrizA9.mtx", valor, NZA);

  //Imprimimos vector valor (sólo para revisión)
//imprimetxt2("./salida_p/valor_p.txt", valor, NZA);

  cudaEventRecord(start, 0);

  cudaEventRecord(start_cpy1, 0);
  //Copiamos el arreglo valor a la GPU
  cudaMemcpy(dev_valor, valor, NZA*sizeof(double),
cudaMemcpyHostToDevice);
  cudaEventRecord(stop_cpy1, 0);
  cudaEventSynchronize(stop_cpy1);

  cudaEventRecord(start_SumaCuadrados, 0);
  SumaCuadrados<<<blocksPerGrid,threadsPerBlock>>>(dev_valor,
dev_partial_c);
  cudaEventRecord(stop_SumaCuadrados, 0);
  cudaEventSynchronize(stop_SumaCuadrados);

  cudaEventRecord(start_cpy2, 0);
  //Copiamos el arreglo 'c' de la GPU a la CPU
  cudaMemcpy(partial_c, dev_partial_c,
blocksPerGrid*sizeof(double), cudaMemcpyDeviceToHost);
  cudaEventRecord(stop_cpy2, 0);
  cudaEventSynchronize(stop_cpy2);

  //Suma final en CPU
```

```
  c = 0.0;
  for (i=0; i<blocksPerGrid; i++)
    c += partial_c[i];

  //Norma Frobenius
  double norma;
  norma = sqrt(c);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  float elapsedTime, elapsedTime_cpy1, elapsedTime_cpy2;
  float elapsedTime_SumaCuadrados;
  cudaEventElapsedTime(&elapsedTime, start, stop);
  cudaEventElapsedTime(&elapsedTime_cpy1, start_cpy1,
stop_cpy1);
  cudaEventElapsedTime(&elapsedTime_SumaCuadrados,
start_SumaCuadrados, stop_SumaCuadrados);
  cudaEventElapsedTime(&elapsedTime_cpy2, start_cpy2,
stop_cpy2);

  printf("blocksPerGrid = %d\n", blocksPerGrid);
  printf("threadsPerBlock = %d\n", threadsPerBlock);
  printf("Suma de cuadrados = %.15f\n", c);
  printf("Norma Frobenius = %.15f\n", norma);
  printf("Tiempo cpy1 = %f s\n", elapsedTime_cpy1/1000);
  printf("Tiempo func = %f s\n",
elapsedTime_SumaCuadrados/1000);
  printf("Tiempo cpy2 = %f s\n", elapsedTime_cpy2/1000);
  printf("Tiempo de procesamiento = %f s\n", elapsedTime/1000);

  //imprimetxt2("./salida_p/partial_c.txt", partial_c,
blocksPerGrid);

  cudaEventDestroy(start);
  cudaEventDestroy(stop);
  cudaEventDestroy(start_cpy1);
  cudaEventDestroy(stop_cpy1);
  cudaEventDestroy(start_SumaCuadrados);
  cudaEventDestroy(stop_SumaCuadrados);
  cudaEventDestroy(start_cpy2);
  cudaEventDestroy(stop_cpy2);

  //Libera memoria en GPU
  cudaFree(dev_valor);
  cudaFree(dev_partial_c);

  //Libera memoria en CPU
  free(valor);
  free(partial_c);
}

  //FUNCIONES

void leeA(char *filename, double *valor, int NZA) {
```

```
  int i, m, n;
  double val;
  FILE *file;

  file = fopen(filename, "r");

  for (i=0; i<NZA; i++)
    valor[i] = 0.0;

  for (i=0; i<NZA; i++) {
    fscanf(file, "%d %d %lf\n", &m, &n, &val);
    valor[i] = val;
  }

  fclose(file);
}

void imprimetxt2(char *filename, double *vec1, int mfilas) {
  int i;
  FILE *file;
  file = fopen(filename, "w");
  for (i=0; i<mfilas; i++)
    fprintf(file, "%d %.15e\n", i+1, vec1[i]);
  fclose(file);
}

__global__ void SumaCuadrados(double *valor, double *c) {

  __shared__ double cache[threadsPerBlock];

  int tid = threadIdx.x + blockIdx.x * blockDim.x;

    //CÁLCULOS DE CADA THREAD
  double temp = 0.0;
  while (tid < NZA) {
    temp += valor[tid] * valor[tid];
    tid += blockDim.x * gridDim.x;
  }

  //Llenamos vector cache con valor de cada THREAD
  cache[threadIdx.x] = temp;

  //Sincronizamos threads en este bloque
  __syncthreads();

    //CÁLCULOS DE SUMA DE THREADS
  int i = blockDim.x/2;
  while (i != 0) {
    if (threadIdx.x < i)
    cache[threadIdx.x] += cache[threadIdx.x + i];
    __syncthreads();
    i /= 2;
  }

  if (threadIdx.x == 0)
    c[blockIdx.x] = cache[0];
```

```
}
```

4.1 Compilación y ejecución

El programa paralelo se compiló con la siguiente línea de comandos bajo el sistema operativo Linux-Ubuntu:

- Compilar: nvcc -o tp4Paralelo3.out tp4Paralelo3.cu -lm –arch=sm_20.

Las pruebas se ejecutaron con la siguiente línea de comandos:

- Correr: existen dos opciones. Su elección depende de la máquina en la cual se ejecuta. Si la máquina está preparada para correr directamente programas con lenguaje CUDA, basta con ejecutar en la línea de comandos:

  ./tp4Paralelo3.out.

  Para las pruebas desarrolladas en este trabajo práctico se empleó una máquina DELL XPS L502X, la cual posee una tecnología que acopla las tarjetas Intel y Nvidia, llamada Optimus. Las dificultades para ejecutar CUDA en dicha máquina fueron superadas instalando el paquete Bumblebee y cargándolo con controladores Bumblebee-nvidia.

  Lo anterior permitió ejecutar el programa anteponiendo el comando optirun a la orden de ejecución del programa. De esta manera, la ejecución obedeció a los comandos:

  optirun ./tp4Paralelo3.out.

4.2 Prueba del algoritmo paralelo

Para las pruebas con el programa CUDA, el archivo de entrada, el número de Non-zeros de la matriz A, el número de hilos por bloque y el número de bloques por grid, son todos ingresados editando el contenido del archivo tp4Paralelo3.cu. Las cuatro variables se encuentran, tres entre las líneas 28 y 30, y una en la línea 67 de tp4Paralelo3.cu.

5. PRUEBAS Y ANÁLISIS DE RESULTADOS

La matriz de entrada para el programa secuencial y el paralelo se llama matrizA9.mtx, y es una matriz que posee 863353 números diferentes de cero. No nos interesan las dimensiones de la matriz ni sus propiedades, dado que el cálculo de la norma no depende de ello. Lo único que se tuvo en cuenta fue seleccionar una matriz de números reales. Dicha matriz fue seleccionada de la Colección de matrices dispersas de la Universidad de Florida [3].

5.1 Sectores de código para toma de tiempos

A continuación se ilustran los sectores de código a los cuales se realizó medición de tiempos. En la sección 4.2, donde se aprecia el código completo, se pueden identificar las líneas de código delimitadas por cada orden cudaEventRecord():

5.1.1 Tiempo copia Host to Device

Es el tiempo transcurrido durante la copia del arreglo unidimensional de valores de la matriz, entre Host y la memoria de la tarjeta gráfica. Esta medida de tiempo está delimitada por las órdenes:

```
cudaEventRecord(start_cpy1,0) y
cudaEventRecord(stop_cpy1,0).
```

5.1.2 Tiempo función CPU

Comprende el tiempo transcurrido durante los cálculos de la última reducción y la raíz cuadrada de la misma. Son los únicos cálculos realizados por el Host. La medida está delimitada por:

```
cudaEventRecord(start_SumaCuadrados,0) y
cudaEventRecord(stop_SumaCuadrados,0).
```

5.1.3 Tiempo copia Device to Host

Es el tiempo transcurrido durante la copia del arreglo c del Device al Host. La medida de tiempo está delimitada por:

```
cudaEventRecord(start_cpy2,0) y
cudaEventRecord(stop_cpy2,0).
```

5.1.4 Tiempo total

Es el tiempo que transcurre desde el envío inicial de la información al Device y el último cálculo del Host, que corresponde a la raíz cuadrada, último cómputo para hallar la norma Frobenius de la matriz. Está delimitado por las órdenes:

```
cudaEventRecord(start,0) y
cudaEventRecord(stop,0).
```

5.1.5 Tiempo función CPU

Es el resultado de restar a Tiempo total, Tiempo de copia Host to Device, Tiempo función GPU y Tiempo copia Device to Host.

5.1.6 Tiempo total sin MemCpy

Es el tiempo resultante de sumar Tiempo función GPU y Tiempo función CPU.

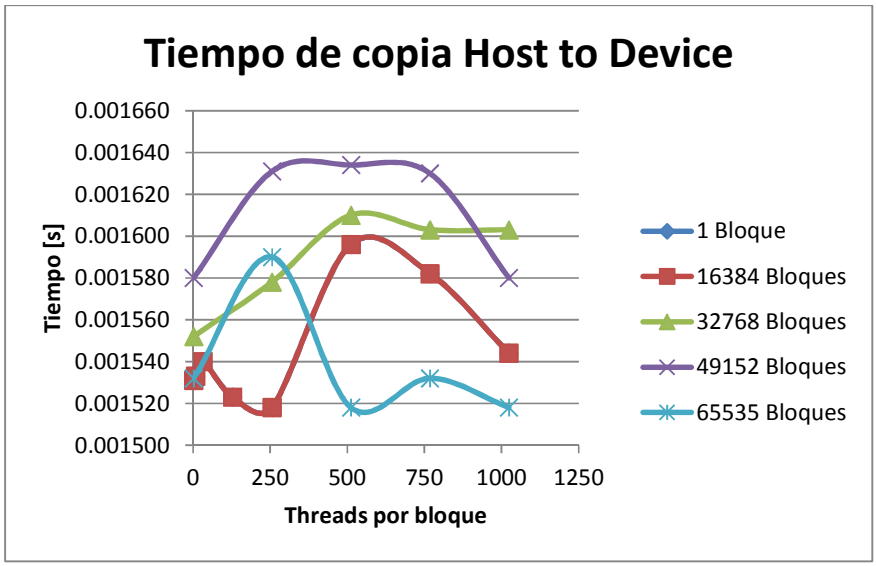
5.2 Resultados de las mediciones de tiempo

Los resultados numéricos de los tiempos tomados para evaluar el desempeño del programa están consignados en la tabla del anexo A2. Todos los tiempos se calcularon ejecutando el programa CUDA repetidamente, manteniendo constantes la matriz de entrada (matrizA9.mtx) y la cantidad de posiciones de la matriz que son diferentes de cero (NZA o Non-Zeros de matriz A). El número de hilos por bloque y de bloques por grid fueron variados de tal forma que cubrieran todo el rango de valores aceptables por la tarjeta gráfica GeForce GT 525M. Estas propiedades se pueden hallar en el anexo A1.

El número de bloques por grid está entre 1 y 65535. El número de hilos por bloque está entre 1 y 1024.

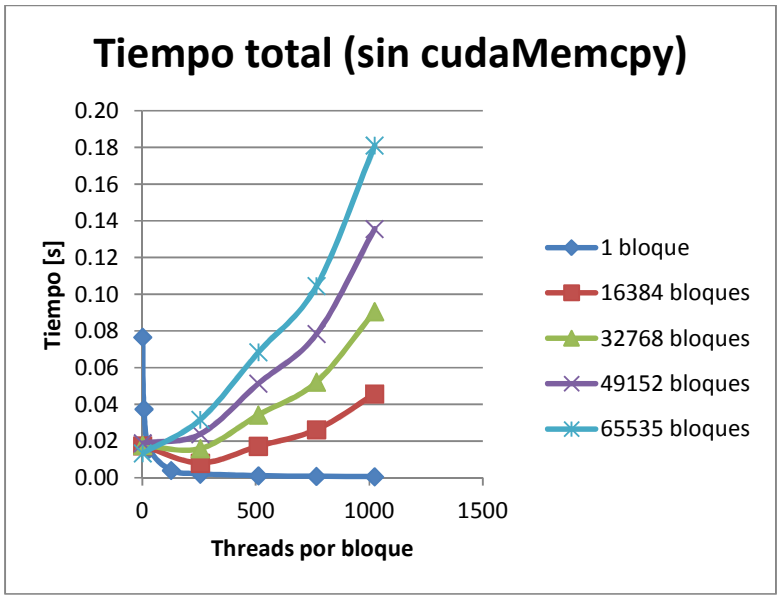
A continuación se presentan los resultados de las mediciones de los tiempos de ejecución del algoritmo paralelo para los diferentes sectores de código descritos en la sección anterior.

Ilustración 1. Tiempo de copia Host to Device.



Haciendo referencia a la Ilustración 1, tenemos que el proceso de copia Host to Device siempre copió la misma cantidad de datos durante todas las pruebas, es decir, un arreglo de 863535 datos. A partir de la figura solo podemos concluir que el tiempo de copia Host to Device no está relacionado con el número de bloques por grid o de hilos por bloque, sólo de la cantidad de datos a copiar (NZA), dado que no se evidencia ninguna tendencia que muestre una relación entre el tiempo de copia Host to Device y lo bloques por grid, ni entre dicha copia y la cantidad de hilos por bloque. La máxima diferencia entre estas mediciones es de 2e-5 segundos.

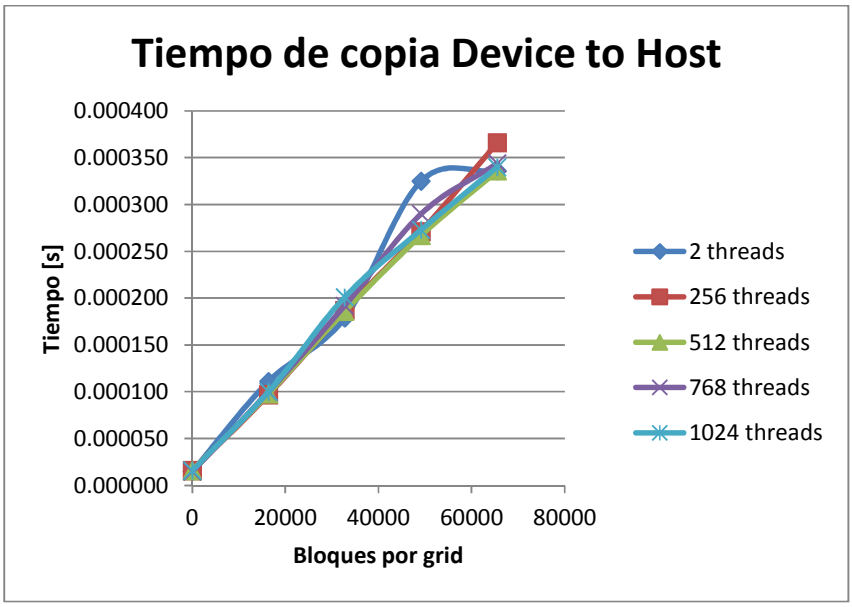
Ilustración 2. Tiempo total sin incluir los tiempos de copia de datos



Restando la influencia de los tiempos de copia, en la Ilustración 2 se observa que el incremento del número de hilos afecta drásticamente el tiempo de cálculo. Para este problema en específico, mientras menos bloques se empleen, mejor es la respuesta del código al aumento del número de hilos.

En la Ilustración 2 no se grafican las tendencias para un número de bloques comprendido entre 1 y 16384, ya que éstas son similares a la curva de tiempos para 1 bloque. Nótese la diferencia apreciable del tiempo de cómputo para 1 bloque. Esta tendencia en la reducción de tiempos se observará más claramente al analizar el Speed Up del código.

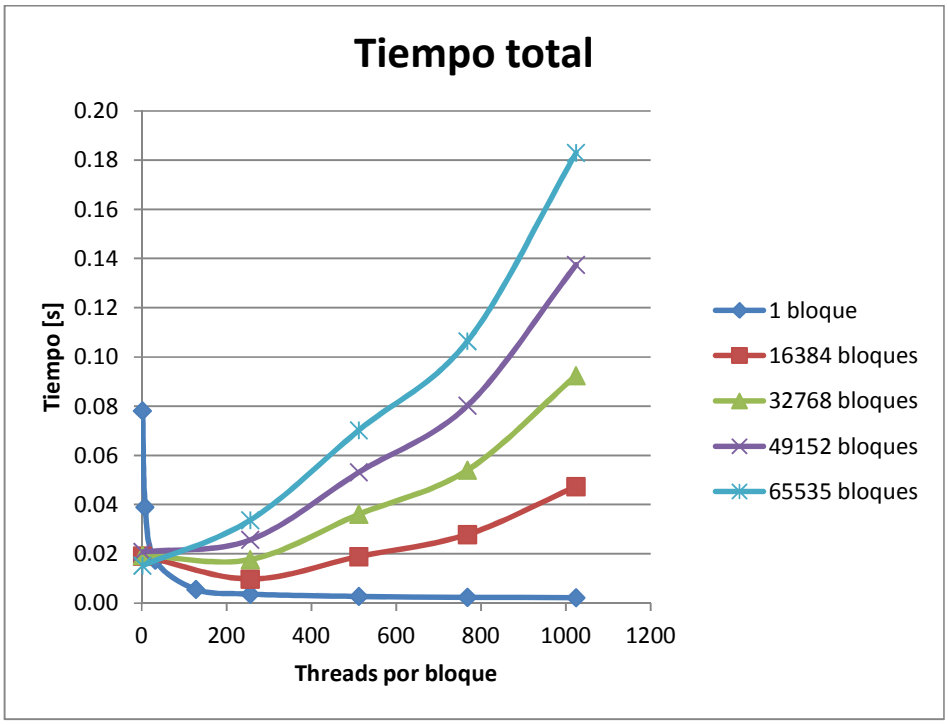
Ilustración 3. Tiempo de copia Device to Host.



La cantidad de datos a copiar entre el Device y el Host es directamente proporcional al número de bloques por grid, como se aprecia en la Ilustración 3. Mientras más bloques, más posiciones tendrá el arreglo c.

La tendencia lineal que observamos en la Ilustración 3 expone que los tiempos de copia Device to Host no dependen del número de threads, solo del número de bloques por grid (lo cual es obvio). El incremento del tiempo de copia Device to Host se incrementa en forma lineal con el incremento del número de bloques por grid.

Ilustración 4. Tiempo total de cómputo.



La Ilustración 4 es muy similar a la Ilustración 2. El tiempo de copia Host to Device es aproximadamente constante y muy corto. El tiempo de copia Device to Host aumenta linealmente con los bloques. Estas dos linealidades son las responsables de que las tendencias de tiempos de cálculo no cambien.

5.3 Análisis de Speed Up del código

El Speed up de un programa paralelo es [2]:

$$S = \frac{T_{serial}}{T_{paralelo}}$$

El valor de T_{serial} permanece constante para todos los Speed Up calculados. Su valor es 0.002893 segundos, que es el promedio de 1000 iteraciones del mismo ciclo secuencial. Por otro lado, $T_{paralelo}$ fue elegido como el tiempo total sin tener en cuenta los tiempos de copia Host a Device ni Device a Host. La razón por la cual se realizó esto obedece a que, para el cálculo específico de matrizA9.mtx, el tiempo de cálculo serial está dentro del mismo orden de

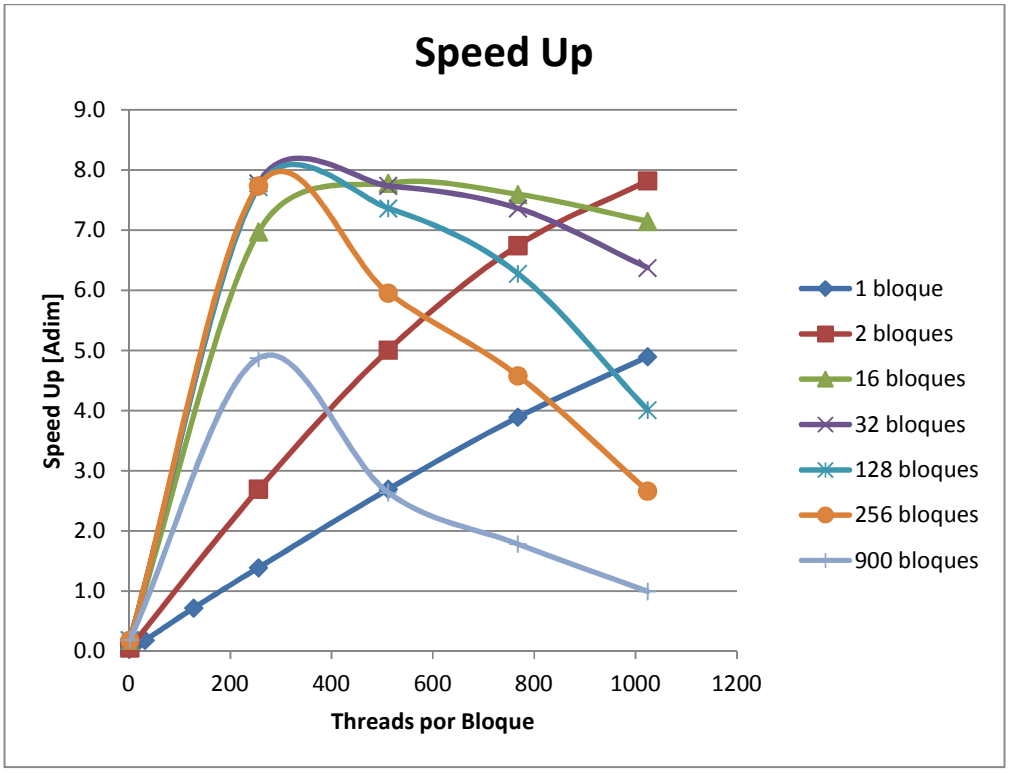
magnitud que los tiempos de copiado Host a Device y viceversa. Fue entonces necesario calcular el Speed Up despreciando estos tiempos de copia, ya que considerarlos implica obtener números que no reflejan el aumento en la velocidad de cómputo que otorga la tarjeta gráfica.

Para grandes cantidades de bloques el Speed Up del algoritmo paralelo, aplicado a este problema específico, es bastante pobre. Se encontró que por encima de aproximadamente 900 bloques los cálculos del Speed Up estaban todos por debajo de 1; umbral por debajo del cual no tiene sentido utilizar un programa paralelo para los cálculos. Estos datos numéricos no fueron graficados, sólo aquellos que otorgan Speed Up mayores que 1.

El Speed Up máximo alcanzado fue 7.82, calculado con 2 bloques y 1024 hilos por bloque. A pesar de esto, un Speed Up muy similar pudo haber sido alcanzado con 32 a 256 bloques y 256 hilos por bloque. Esto indica que, para este problema en particular, tenemos un algoritmo que escala empleando pocos bloques y muchos hilos, o bien varios bloques y pocos hilos, pero no incrementando ambos.

Las tendencias se observan en la Ilustración 5.

Ilustración 5. Speed Up del algoritmo paralelo (CUDA)



6. DIFICULTADES ENCONTRADAS

- Por comodidad, hubiera sido preferible ingresar los parámetros de entrada a través de la línea de comandos para la ejecución del programa paralelo. Esto no se realizó por la imposibilidad de leer los datos del arreglo argv[] por fuera de la función main().

- A pesar de que el computador DELL XPS L502X posee tarjeta Nvidia GeForce GT 525M habilitada para CUDA, no fue fácil encontrar controladores para la tarjeta bajo el sistema Linux-Ubuntu, debido a que la presencia de la tecnología Optimus no permitía el reconocimiento de la placa Nvidia. A pesar de disponer de dos placas TESLA en el Departamento de Cómputo de la facultad, se prefirió experimentar con la placa de un computador portátil para ganar experiencia en el manejo de problemas relacionados con la habilitación del dispositivo.

- A pesar de trabajar con doble precisión, habilitándola a través del comando -arch=sm_20 en la línea de comandos, no fue posible obtener un resultado numérico de la norma Frobenius igual al hallado con el algoritmo serial. El cálculo de la norma serial es igual a 0.07371984076054, mientras que los cálculos con el algoritmo paralelo resultaron en normas con los últimos cuatro o cinco dígitos decimales diferentes.

- Una dificultad radica en la medición de tiempos para el programa serial. Esto se solucionó empleando MPI para la medición de tiempos. Se desconoce si existe alguna contraindicación que desvirtúe el uso de MPI sólo para medir tiempos.

- Nunca se empleó un programa manejador de versiones de los programas y de sus modificaciones. Esta herramienta no se aprendió a manejar y permanece pendiente.

7. CONCLUSIONES Y OBSERVACIONES

- Los tiempos de comunicación disminuyen significativamente el desempeño del programa CUDA para cálculos numéricos muy pequeños y en donde la cantidad de datos es grande. Para problemas de esta clase conviene reducir el número de bloques por grid y aumentar el número de hilos por bloque.

- El desempeño de un programa escrito en CUDA depende del Latency Hiding. Mientras más ocupada se encuentre la tarjeta gráfica en todo momento, mejor justifica el tiempo perdido en pasaje de información, tanto de Host a Device como de Device a Host.

- Los tiempos de copia de información Device a Host varían en forma lineal con el aumento del número de bloques por grid.

- El Speed Up máximo logrado con el algoritmo escrito en CUDA para este problema específico es de 7.82, lo cual implica que se incrementó la velocidad de cómputo casi 8 veces empleando una GPU.

- El Speed Up obtenido es una variable que también depende del problema específico a resolver.

- Para problemas de cómputo intensivos no sería necesario despreciar los tiempos de copia Host a Device ni Device a Host, ya que estos tendrían un orden de magnitud diferente y serían despreciables en comparación con el tiempo empleado para los cálculos numéricos en el Device.

8. REFERENCIAS BIBLIOGRÁFICAS

[1] Formato de almacenamiento de matrices del Matrix Market: http://math.nist.gov/MatrixMarket/mmio-c.html. Ultima fecha de consulta: 17 de octubre de 2012.

[2] PACHECO, Peter S. An introduction to parallel programming. Morgan Kaufmann. Elsevier Inc. 2011.

[3] Página de Internet: Colección de matrices dispersas de la Universidad de Florida. UF Sparse Matrix Collection: http://www.cise.ufl.edu/research/sparse/matrices/list_by_type.html. Ultima fecha de consulta: 17 de octubre de 2012.

[4] SANDERS, Jason. Kandrot, Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. NVIDIA. Addison-Wesley. Página 81.

ANEXO

A1. ESPECIFICACIONES TÉCNICAS DE LA PLACA NVIDIA GeForce GT 525M

```
Device Count = 1

 --- Información general del dispositivo ---
Nombre: GeForce GT 525M
Compute capability: 2.1
Clock rate: 1200000
Device copy overlap:   Habilitado

 --- Memoria ---
Total global mem:  1073414144
Total constant mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512

 --- Multi Processor information ---
Multiprocessor count:  2
Shared mem per MP:  49152
Registers per MP:  32768
Threads in Warp:  32
Max threads per block:  1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (65535, 65535, 65535)
```

A2. TABLA DE MEDICIONES DE TIEMPO PARA EL CÁLCULO DEL DESEMPEÑO DEL PROGRAMA

En la siguiente tabla se consignan los tiempos de cómputo paralelo durante el cálculo de la Norma Frobenius de matrizA9.mtx

	VARIABLE	TIEMPO DE CÓMPUTO [segundos]				
	Bloques por grid	-	-	**1**		
	Hilos por bloque	-	-	**2**	**8**	**32**
1	Tiempo copia Host to Device	-	-	0.001531	0.001533	0.001540
2	Tiempo función GPU	-	-	0.076556	0.037396	0.015832
3	Tiempo copia Device to Host	-	-	0.000015	0.000016	0.000017
4	Tiempo función CPU	-	-	0.000024	0.000023	0.000024
5	Tiempo total sin MemCpy	-	-	0.076580	0.037419	0.015856
6	Tiempo total	-	-	0.078126	0.038968	0.017413
7	Speed Up (adimensional)	-	-	0.037777	0.077314	0.182455
	Bloques por grid	**1**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001523	0.001518	0.001596	0.001582	0.001544
2	Tiempo función GPU	0.004009	0.002059	0.001051	0.000723	0.000567
3	Tiempo copia Device to Host	0.000016	0.000016	0.000016	0.000015	0.000016
4	Tiempo función CPU	0.000022	0.000026	0.000023	0.000021	0.000024
5	Tiempo total sin MemCpy	0.004031	0.002085	0.001074	0.000744	0.000591
6	Tiempo total	0.005570	0.003619	0.002686	0.002341	0.002151
7	Speed Up (adimensional)	0.717688	1.387530	2.693669	3.888441	4.895093
	Bloques por grid	**2**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001591	0.001536	0.001509	0.001522	0.001494
2	Tiempo función GPU	0.050464	0.001051	0.000555	0.000406	0.000348
3	Tiempo copia Device to Host	0.000017	0.000015	0.000017	0.000016	0.000016
4	Tiempo función CPU	0.000023	0.000023	0.000023	0.000023	0.000022
5	Tiempo total sin MemCpy	0.050487	0.001074	0.000578	0.000429	0.000370
6	Tiempo total	0.052095	0.002625	0.002104	0.001967	0.00188
7	Speed Up (adimensional)	0.057302	2.693669	5.005190	6.743590	7.818919
	Bloques por grid	**4**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001528	0.001546	0.001503	0.001544	0.001548
2	Tiempo función GPU	0.03742	0.000555	0.000341	0.000354	0.000348
3	Tiempo copia Device to Host	0.000016	0.000016	0.000017	0.000015	0.000015
4	Tiempo función CPU	0.000026	0.000030	0.000025	0.000021	0.000022
5	Tiempo total sin MemCpy	0.037446	0.000585	0.000366	0.000375	0.000370
6	Tiempo total	0.03899	0.002147	0.001886	0.001934	0.001933
7	Speed Up (adimensional)	0.077258	4.945299	7.904372	7.714667	7.818919
	Bloques por grid	**16**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**

1	Tiempo copia Host to Device	0.001535	0.001597	0.001532	0.001574	0.001531
2	Tiempo función GPU	0.015838	0.000394	0.00035	0.000358	0.000383
3	Tiempo copia Device to Host	0.000016	0.000014	0.000015	0.000015	0.000016
4	Tiempo función CPU	0.000024	0.000021	0.000022	0.000023	0.000022
5	Tiempo total sin MemCpy	0.015862	0.000415	0.000372	0.000381	0.000405
6	Tiempo total	0.017413	0.002026	0.001919	0.00197	0.001952
7	Speed Up (adimensional)	0.182386	6.971084	7.776882	7.593176	7.143210
	Bloques por grid	**32**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001515	0.001529	0.001531	0.001505	0.001502
2	Tiempo función GPU	0.015842	0.00035	0.000353	0.00037	0.000431
3	Tiempo copia Device to Host	0.000016	0.000015	0.000015	0.000018	0.000018
4	Tiempo función CPU	0.000023	0.000022	0.000021	0.000023	0.000023
5	Tiempo total sin MemCpy	0.015865	0.000372	0.000374	0.000393	0.000454
6	Tiempo total	0.017396	0.001916	0.00192	0.001916	0.001974
7	Speed Up (adimensional)	0.182351	7.776882	7.735294	7.361323	6.372247
	Bloques por grid	**128**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001529	0.001528	0.001551	0.001532	0.001543
2	Tiempo función GPU	0.015807	0.000352	0.000372	0.000437	0.000699
3	Tiempo copia Device to Host	0.000016	0.000016	0.000016	0.000015	0.000016
4	Tiempo función CPU	0.000026	0.000023	0.000021	0.000024	0.000023
5	Tiempo total sin MemCpy	0.015833	0.000375	0.000393	0.000461	0.000722
6	Tiempo total	0.017378	0.001919	0.00196	0.002008	0.002281
7	Speed Up (adimensional)	0.182720	7.714667	7.361323	6.275488	4.006925
	Bloques por grid	**256**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001541	0.001537	0.00153	0.001526	0.001552
2	Tiempo función GPU	0.015813	0.000351	0.000463	0.00061	0.001064
3	Tiempo copia Device to Host	0.000017	0.000016	0.000016	0.000016	0.000015
4	Tiempo función CPU	0.000024	0.000023	0.000023	0.000022	0.000023
5	Tiempo total sin MemCpy	0.015837	0.000374	0.000486	0.000632	0.001087
6	Tiempo total	0.017395	0.001927	0.002032	0.002174	0.002654
7	Speed Up (adimensional)	0.182673	7.735294	5.952675	4.577532	2.661454
	Bloques por grid	**900**				
	Hilos por bloque	**128**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001573	0.001516	0.001547	0.00158	0.001518
2	Tiempo función GPU	0.015297	0.000568	0.001074	0.001598	0.002891
3	Tiempo copia Device to Host	0.000019	0.000017	0.000018	0.000018	0.000018
4	Tiempo función CPU	0.000032	0.000026	0.000024	0.000026	0.000025
5	Tiempo total sin MemCpy	0.015329	0.000594	0.001098	0.001624	0.002916
6	Tiempo total	0.016921	0.002127	0.002663	0.003222	0.004452
7	Speed Up (adimensional)	0.188727	4.870370	2.634791	1.781404	0.992112
	Bloques por grid	**16384**				
	Hilos por bloque	**2**	**256**	**512**	**768**	**1024**

1	Tiempo copia Host to Device	0.001564	0.001615	0.001592	0.001559	0.001579
2	Tiempo función GPU	0.017346	0.007935	0.017125	0.026108	0.045605
3	Tiempo copia Device to Host	0.000111	0.000097	0.000098	0.000100	0.000100
4	Tiempo función CPU	0.000096	0.000077	0.000078	0.000079	0.000094
5	Tiempo total sin MemCpy	0.017442	0.008012	0.017203	0.026187	0.045699
6	Tiempo total	0.019117	0.009724	0.018893	0.027846	0.047378
7	Speed Up (adimensional)	0.165864	0.361083	0.168168	0.110475	0.063306
	Bloques por grid	**32768**				
	Hilos por bloque	**2**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001552	0.001578	0.001610	0.001603	0.001603
2	Tiempo función GPU	0.017500	0.015774	0.034143	0.052071	0.090470
3	Tiempo copia Device to Host	0.000179	0.000187	0.000186	0.000192	0.000201
4	Tiempo función CPU	0.000132	0.000148	0.000160	0.000153	0.000165
5	Tiempo total sin MemCpy	0.017632	0.015922	0.034303	0.052224	0.090635
6	Tiempo total	0.019363	0.017687	0.036099	0.054019	0.092439
7	Speed Up (adimensional)	0.164077	0.181698	0.084337	0.055396	0.031919
	Bloques por grid	**49152**				
	Hilos por bloque	**2**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001580	0.001631	0.001634	0.001630	0.001580
2	Tiempo función GPU	0.018613	0.023620	0.051109	0.078124	0.135436
3	Tiempo copia Device to Host	0.000325	0.000271	0.000267	0.000290	0.000273
4	Tiempo función CPU	0.000247	0.000195	0.000219	0.000220	0.000204
5	Tiempo total sin MemCpy	0.018860	0.023815	0.051328	0.078344	0.135640
6	Tiempo total	0.020765	0.025717	0.053229	0.080264	0.137493
7	Speed Up (adimensional)	0.153393	0.121478	0.056363	0.036927	0.021329
	Bloques por grid	**65535**				
	Hilos por bloque	**2**	**256**	**512**	**768**	**1024**
1	Tiempo copia Host to Device	0.001532	0.001590	0.001518	0.001532	0.001518
2	Tiempo función GPU	0.013070	0.031454	0.068164	0.104036	0.180867
3	Tiempo copia Device to Host	0.000336	0.000366	0.000336	0.000344	0.000340
4	Tiempo función CPU	0.000242	0.000265	0.000244	0.000524	0.000246
5	Tiempo total sin MemCpy	0.013312	0.031719	0.068408	0.104560	0.181113
6	Tiempo total	0.015180	0.033675	0.070262	0.106436	0.182971
7	Speed Up (adimensional)	0.217323	0.091207	0.042290	0.027668	0.015973