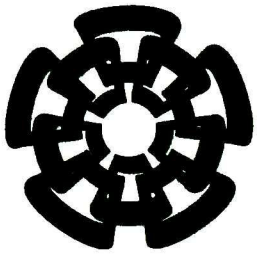


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

Adaptación de la metodología de paralelización de bucles en el modelo del politopo para la implementación de algoritmos en GPUs

Tesis que presenta:
Daniel Robles Valdez

para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Director de Tesis
Dr. Deni Librado Torres Román

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Diciembre de 2013.

Adaptación de la metodología de paralelización de bucles en el modelo del politopo para la implementación de algoritmos en GPUs

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Daniel Robles Valdez
Ingeniero en Electrónica
Instituto Tecnológico de Sonora 2006-2011

Becario de Conacyt, expediente no. 263584

Director de Tesis
Dr. Deni Librado Torres Román

CINVESTAV del IPN Unidad Guadalajara, Diciembre de 2013.

Adaptación de la metodología de paralelización de bucles en el modelo del politopo para la implementación de algoritmos en GPUs

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Daniel Robles Valdez
Ingeniero en Electrónica
Instituto Tecnológico de Sonora 2006-2011

Becario de Conacyt, expediente no. 263584

Director de Tesis
Dr. Deni Librado Torres Román

CINVESTAV del IPN Unidad Guadalajara, Diciembre de 2013.

Resumen

Esta tesis presenta una introducción al lenguaje de programación CUDA C para la implementación de algoritmos en GPU's. Explica como es la interacción entre el CPU y el GPU, la asignación de tareas en los procesadores y manejar los diferentes tipos de memoria de un GPU.

Se presenta como modelar bucles anidados empleando el modelo del politopo, el criterio de legalidad de una transformación unimodular y como construir la matriz de transformación a partir de las dependencias de datos.

Se analizan e implementan tres algoritmos: suma de vectores, matriz transpuesta y descomposición QR. Se comparan los resultados obtenidos de las versiones secuenciales con las versiones paralelas en CUDA C y se analiza el gasto de memoria de los algoritmos para diferentes tamaños de matrices o vectores.

Con las adaptaciones a la metodología de paralelización de bucles anidados, se obtuvo una versión de la descomposición QR, en donde se reduce la memoria requerida y los tiempos de ejecución resultantes.

Con la arquitectura de CUDA se reduce significativamente el tiempo de ejecución para algoritmos paralelizados que procesan un alto volumen de datos.

Abstract

This thesis presents an introduction to the CUDA C programming language for implementing algorithms on GPU 's. Explain how is the interaction between the CPU with the GPU, the allocation of tasks to processors and handle different types of memory in a GPU.

Furthermore, a strategy for modelling nested loops, as well as, a criterion of legality of a unimodular transformation, and how to build a transformation matrix from data depencences are presented.

We analyzed and implemented three algorithms: vector sum, transposed matrix and QR decomposition. The results of sequential and parallel CUDA C versions are compared and memory usage of algorithms for different size matrices or vectors are analyzed.

With adjustments to the methodology of parallelization nested loops, a version of the QR decomposition, where reduced memory requirements, and resulting execution times was obtained.

With this CUDA architecture, a significant time reduction is obtained for parallelized algorithms which process high volume of data .

Agradecimientos

- A Dios por sus cuidados y múltiples bendiciones.
- A mis padres por apoyarme siempre en cada etapa de mi vida.
- A mi novia Raquel Esperón por acompañarme y por su cariño.
- A mi asesor Dr. Deni Librado Torres Román por su apoyo y dirección.
- Al Dr. Manuel E. Guzmán Rentería por compartir sus ideas, conocimientos y consejos.
- A mis amigos y compañeros de generación: Laura, Alberto, Joaquín, Benjamín y Gustavo, por la buena convivencia y su apoyo.
- A David Castro y a Ángel Jovany por compartir sus ideas y conocimientos.
- Al Consejo Nacional de Ciencia y Tecnología por el apoyo económico.

Contenido

Capítulo 1

Introducción

1.1. Introducción

Estamos en la era dorada del computo GPU (graphics processing unit). Desde la introducción de CUDA (Compute Unified Device Architecture) en 2007, más de 100 millones de computadoras con GPUs compatibles con CUDA han sido construidos y utilizados alrededor del mundo. Como resultado, muchos investigadores tienen ahora que reformular sus modelos computacionales e invertir en software para crear aplicaciones de alto rendimiento basados en esta tecnología [1].

Es sabido, que muchas aplicaciones en ciencia e ingeniería, en donde se procesa un gran volumen de datos, pasan una parte considerable de su tiempo de ejecución en el procesamiento de bucles. Por tal motivo, es pertinente desarrollar técnicas en donde se modifique el código para que se ejecute, en paralelo, la mayor cantidad de iteraciones de los bucles.

1.2. Motivación

Actualmente en el área de procesamiento digital de señales, así como en otras áreas de la ingeniería, como por ejemplo: procesamiento de audio, formación de imágenes médicas y modelado estadístico, entre otras, se están empleando en gran medida las GPU's. Por tal motivo, existe una gran interés en mejorar el nivel de desempeño (uso de memoria, uso de procesadores y tiempo de ejecución) de diferentes algoritmos [1].

En lo general, en el área de procesamiento de señales, audio e imágenes, se están implementando algoritmos que realizan el cálculo de la FFT (Fast Fourier Transform) a larga escala, el reconocimiento automático del habla y en la reconstrucción de imágenes tomográficas en 3D, sólo por mencionar algunas.

En lo particular, en CINVESTAV (Centro de Investigación y de Estudios Avanzados) Unidad Guadalajara departamento de telecomunicaciones, se han hecho varios esfuerzos

que han producido trabajos de investigación [2], [3] y [4]. Los cuales aplican una metodología para la paralelización de algoritmos usando el modelo del politopo. Por lo tanto, se cuenta con una metodología de paralelización de algoritmos; la cual, se encuentra orientada hacia una implementación física en términos de un arreglo sistólico de procesadores.

Lo cual deja de manifiesto que el área del computo paralelo de alto desempeño no ha sido explorado, abriéndose con ello una amplia variedad de posibles soluciones usando herramientas orientadas a software. Con las cuales, se disponga de códigos paralelos en sentido estricto o matemático para su ejecución sobre una GPU, empleando las herramientas de CUDA C, convirtiéndose esta tarea en un gran reto.

1.3. Objetivos

Adaptar la metodología de paralelización de bucles, empleado en el modelo del politopo, con la finalidad de lograr la implementación de algoritmos sobre GPU's utilizando el lenguaje CUDA C.

Los objetivos particulares son los siguientes:

- Describir aspectos teóricos y prácticos para el manejo de GPU's como un coprocesador paralelo.
- Describir aspectos teóricos y prácticos en la utilización del lenguaje CUDA C como herramienta para el manejo de las GPU's.
- Introducir bases teóricas para la transformación de algoritmos que poseen bucles anidados.
- Modelar un algoritmo secuencial que realiza la descomposición QR transformandolo a su versión paralela, a través de matrices de transformación unimodulares.
- Proponer alternativas de estructuración del código para el buen manejo de los procesadores de una GPU.
- Establecer buenas prácticas, que permitan el uso eficiente de los recursos (memoria global y procesadores) de una GPU.

1.4. Perspectiva de los capítulos

En el capítulo 2 se presenta una introducción al lenguaje de programación CUDA C para la implementación de algoritmos en GPU's. Se explica como es la interacción entre el CPU (Central Processing Unit) con el GPU, la asignación de tareas en los procesadores y manejar los diferentes tipos de memoria de un GPU. Además, se presenta una sección con buenas prácticas de programación en CUDA.

El capítulo 3 inicia con los antecedentes matemáticos para modelar bucles anidados en el modelo del politopo. Presenta el criterio de legalidad de una transformación unimodular y como construir la matriz de transformación a partir de las dependencias de datos. Finalmente, aborda los tipos de paralelismo interno y externo.

En el capítulo 4 se analizan e implementan en CUDA C tres algoritmos: suma de vectores, matriz transpuesta y descomposición QR. Se comparan los resultados obtenidos de las versiones secuenciales con las versiones paralelas en CUDA y se analiza el gasto de memoria de los algoritmos para diferentes tamaños de matrices o vectores.

El capitulo 5 presenta las conclusiones y el trabajo futuro.

Capítulo 2

CUDA y GPU

2.1. Introducción a CUDA

2.1.1. Modelo de programa en CUDA

Un programa en CUDA consiste en la ejecución de código fuente en el ***host*** (CPU con su memoria) y en el ***device*** (GPU con su memoria). El *device* es manejado por funciones especiales de CUDA C que son escritas en el código del ***host***, por lo que el *device* se puede ver como un esclavo del *host*.

El compilador **nvcc** (NVIDIA C compiler) identifica y separa el código que se requiere ejecutar en el *device* del código del *host*. El *código host* es compilado y es ejecutado por un proceso ordinario en el CPU, este puede ser escrito en lenguaje estándar C/C++. El *código device* puede escribirse usando C extendido con identificadores para etiquetar funciones que se ejecutarán en el GPU, llamadas funciones ***kernel***, y sus estructuras de datos asociadas[5]. Los identificadores para diferenciar las funciones del *host* y del *device* se muestran en la tabla 2.1. Si no se coloca un calificador a una función, el compilador la asociará a una función que se ejecutará en el *host* [6].

La secuencia básica de un programa en CUDA es la siguiente:

1. Generar los datos en el *host*.
2. Asignar memoria en el *device*.
3. Enviar los datos del *host* hacia el *device*.
4. Ejecutar la función *kernel* para el procesamiento de los datos.
5. Recolectar los datos procesados, realizando una copia del *device* al *host*.
6. Liberar toda la memoria utilizada en el *host* y en el *device*.

Tabla 2.1: Identificadores de funciones.

Calificador	Ejecutado en:	Puede ser llamado desde:
device	Device	Device
global	Device	Host
host	Host	Host

Uno de los retos en la programación en CUDA es repartir las tareas entre el *device* y el *host*, de tal manera que nuestro programa completo se ejecute en la menor cantidad de tiempo. Para determinar que partes de código deben ser ejecutados por el *host* o por el *device*, es necesario examinar la dependencia de datos, que en ocasiones, resulta algo complicado. **Paralelismo** se refiere a la propiedad de un programa, mediante el cual, muchas operaciones aritméticas se pueden realizar de manera segura y de manera simultánea en las estructuras de datos.

Por lo que el código secuencial que exhibe poco o nada de paralelismo debe ser ejecutado en el *host*, mientras que el código que exhibe el mayor paralelismo debe ser ejecutado en el *device*.

2.1.2. Jerarquía de hilos de ejecución

Para hablar de como CUDA y su modelo de programación puede explotar el paralelismo existente en un programa, es necesario primero establecer la terminología básica que se va a estar empleando en el desarrollo de esta sección y en las siguientes.

Llamaremos **lanzamiento de *kernel***, a la ejecución de una función en el *device* con el calificador _global_. Al lanzar una función *kernel*, automáticamente se crean un conjunto de hilos de ejecución que llamaremos **threads**. En cada uno de los *threads* que se creen, se va a tener una copia del cuerpo de la función *kernel*, es decir, se van a ejecutar las mismas instrucciones, la diferencia es que los datos utilizados en el procesamiento podrán ser diferentes.

Cada *thread* tiene su identificador asociado llamado **threadIdx**, el cual puede ser de 1, 2 o hasta 3 dimensiones. Mismo que puede ser útil para referenciar diferentes localidades de memoria del *device*. *ThreadIdx* es una variable tipo uint3 que contiene tres campos de tipo entero no signado, los cuales determinan su posición en las 3 dimensiones.

Los *threads* son el nivel inferior en la jerarquía, seguidos por el nivel de *blocks*. Un **block** es un contenedor de *threads*, que al igual que los *threads*, puede ser de 1, 2 o hasta 3 dimensiones. Su identificador **blockIdx** tambien es una variable tipo uint3 que contiene tres campos de tipo entero sin signo, los cuales determinan la posición del*block* en las 3 dimensiones del *blocks*.

El nivel superior en la jerarquía es el *grid*. Un **grid** es el conjunto de todos los *blocks* que a su vez contienen *threads* en un lanzamiento de *kernel*. Al lanzar una función *kernel*

se creará un *blocks*, para dimensionar los *blocks* y los *threads*, se utilizan las variables tipo dim3 **gridDim** y **blockDim** respectivamente (las variables dim 3 son un vector con 3 campos enteros, los campos sin definir por defecto se pondrán en 1) [6] y [7].

Un lanzamiento de *kernel* se declara:

```
__global__ kernel<<< gridDim, blockDim >>>(parámetros);
```

Donde <<< *gridDim*, *blockDim* >>> dimensiona todo el conjunto de *threads* en el *blocks*. Todos *threads* tendrán una cópia de los parámetros, con la cual, podrán efectuar cálculos especificados en el cuerpo de la función.

La figura 2.1 muestra de manera gráfica los conceptos establecidos anteriormente. En la cual, se muestra de manera general el dimensionamiento del *blocks* en sus tres dimensiones de direccionamiento para los *blocks* y para *threads*.

Las variables gridDim y blockDim son constantes que pueden ser utilizadas en el *kernel* por todos los *threads*. Las variables *blockIdx* y *threadIdx* toman rangos de valores:

```
blockIdx.x = {0, 1, ..., gridDim.x − 1}
blockIdx.y = {0, 1, ..., gridDim.y − 1}
blockIdx.z = {0, 1, ..., gridDim.z − 1}

threadIdx.x = {0, 1, ..., blockDim.x − 1}
threadIdx.y = {0, 1, ..., blockDim.y − 1}
threadIdx.z = {0, 1, ..., blockDim.z − 1}
```

2.1.3. Ejemplo de programa en CUDA: Suma de dos vectores

Para aplicar los conceptos establecidos hasta el momento, se desarrollará un programa ejemplo, el cual tiene como objetivo ejecutar en paralelo la suma de dos arreglos *host_a* y *host_b* (con 5 elementos cada uno, por simplicidad); guardando el resultado en el arreglo *host_c*.

En el desarrollo de este ejemplo seguiremos la secuencia básica de un programa en CUDA descrita anteriormente. El primer paso es generar los datos en el *host*, los cuales se quiere procesar.

```
// Generación de datos en el host:
const int arraySize = 5;
const int host_a[arraySize] = { 1, 2, 3, 4, 5 };
const int host_b[arraySize] = { 10, 20, 30, 40, 50};
int host_c[arraySize] = { 0, 0, 0, 0, 0 };
```

Para hacer uso de la memoria del *device* es necesario reservarla, esto se realiza con la función de CUDA *cudaMalloc*, la cual asigna espacio en la memoria DRAM (Dynamic

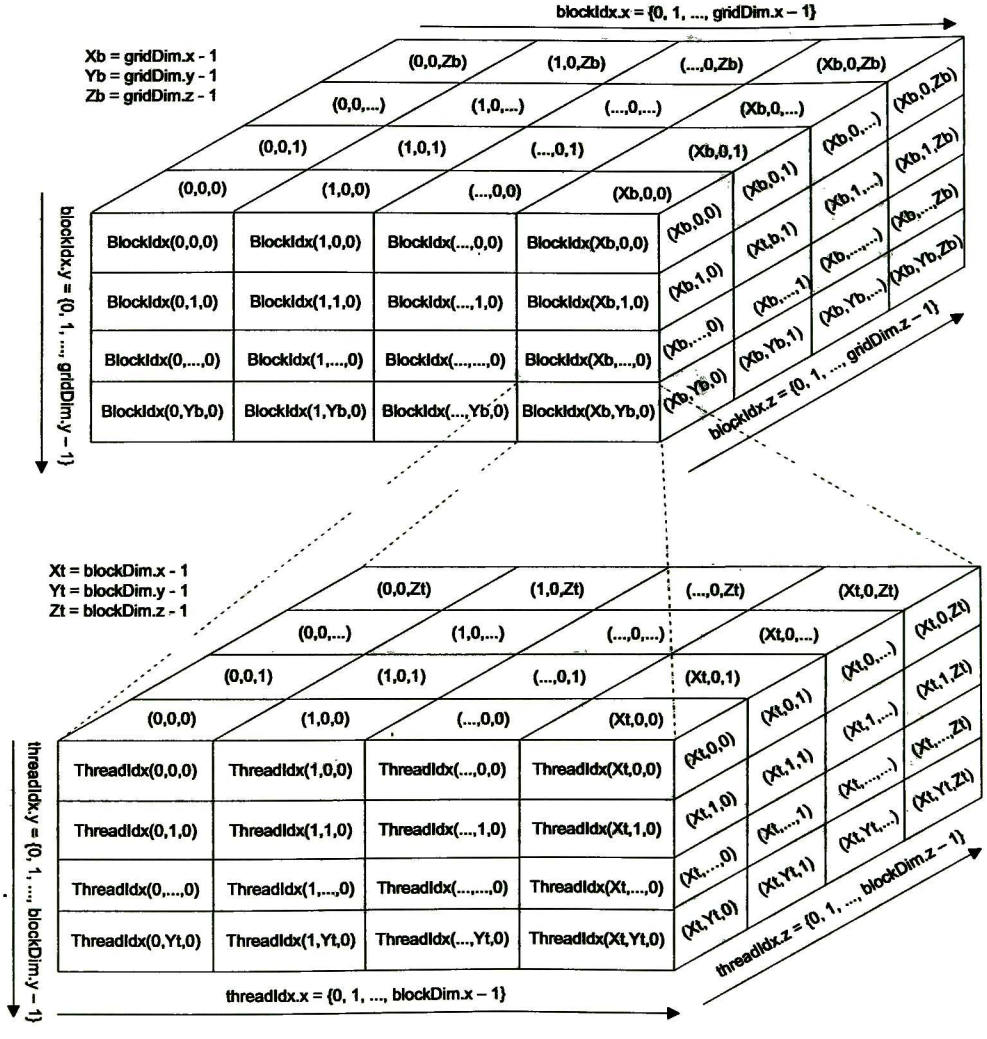


Figura 2.1: Dimensionamiento de un *grid* CUDA, con el lanzamiento de *kernel*: _global_ *kernel*<<< *gridDim, blockDim* >>> (*void*);

Random-Access Memory) del *device*. El segundo parámetro *size*sizeof(int)* corresponde al número de bytes a reservar.

```
// Asignación de memoria en el device:
cudaMalloc((void**)&dev_c, size * sizeof(int));
cudaMalloc((void**)&dev_a, size * sizeof(int));
cudaMalloc((void**)&dev_b, size * sizeof(int));
```

Una vez reservado la memoria en el *device* y generado los datos a procesar, debemos transferir los datos del *host* hacia el *device*, esto se realiza mediante la función *cudaMemcpy*.

```
// Envío de los datos de entrada hacia el device:
cudaMemcpy(dev_a, host_a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, size * sizeof(int), cudaMemcpyHostToDevice);
```

El programa requiere realizar 5 sumas, como no existe dependencia de datos entre los arreglos de entrada, se pueden realizar en paralelo. Para esto se necesitan 5 *threads*, los cuales fueron organizados en un bloque que contiene los 5 *threads*.

```
// Dimensionamiento del grid para la función kernel
dim3 blockDim;
// Dimensionamiento de blocks
gridDim.x = 1; gridDim.y = 1; gridDim.z = 1; // Un bloque
// Dimensionamiento de Threads, 5 hilos en el eje x por bloque
dim3 blockDim;
blockDim.x = size; blockDim.y = 1; blockDim.z = 1;
// Lanzamiento del kernel con un threadspara cada elemento del arreglo
addKernel<<<gridDim, blockDim>>>(dev_c, dev_a, dev_b);
```

El cuerpo del *kernel* consta de la suma de los arreglos *dev_a* y *dev_b*, guardando el resultado en el arreglo *dev_c*.

```
// Función kernel, ejecutada en el device
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

Las variables *dimGrid* y *dimBlock*, son las que dimensionan los *threads* en el *blocks*. La figura 2.2 muestra de manera más clara como impactan estas variables en el *blocks*, en la cual, se ilustra el *blocks* generado por el *kernel* del programa suma de dos vectores. La variable *i* que se encuentra en el cuerpo del *kernel*, se crea en los 5 *threads*, toma valores diferentes en cada *threads*debido a que *threadIdx*.x varia de cero a 4 (blockDim.x - 1 = 5 - 1). Por lo que *i* indexará localidades diferentes en los arreglos *dev_a*, *dev_b* y *dev_c*.

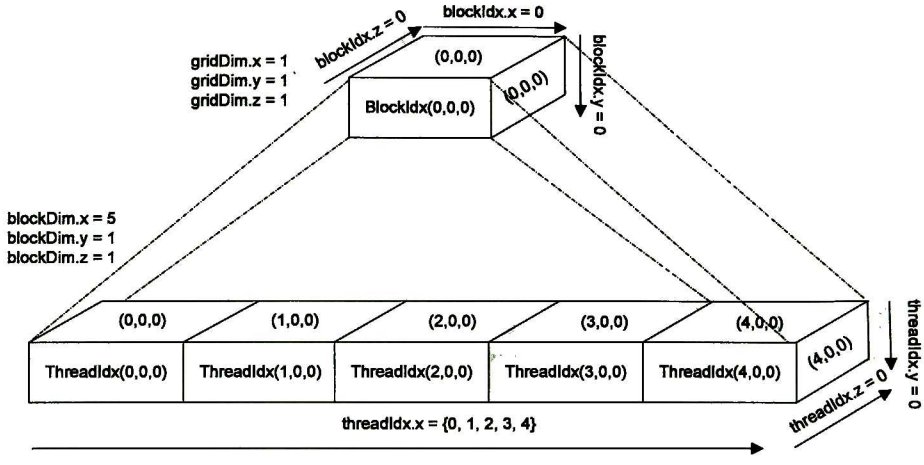


Figura 2.2: *Grid* generado por el *kernel* suma de dos vectores

Además, se agregó la función *cudaDeviceSynchronize()* para que el *host* espere a que el *device* termine de ejecutar el *kernel*, para que el arreglo *dev_c* de salida contenga los elementos calculados por el *kernel*.

```
// Esperar que el kernel termine de ejecutarse totalmente.
cudaDeviceSynchronize();
```

En este punto ya se calculó la suma, por lo que es tiempo de transferir los resultados del *device* al *host* y liberar la memoria utilizada en el *device*.

```
// Copia del arreglo procesado hacia el host.
cudaMemcpy(host_c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
//Liberación de la memoria del device
cudaFree(dev_c); cudaFree(dev_a); cudaFree(dev_b);
```

El código completo de este ejemplo se encuentra en el apendice ??. El resultado del programa debe ser *host_c* = {*11, 22, 33, 44, 55*}. Este programa se puede utilizar para comprobar la correcta instalación de los drivers del GPU, así como del software de CUDA.

2.1.4. Medición de tiempos de ejecución en el GPU

La medición de tiempos en CUDA se realiza mediante "eventos" Un *evento* es una estampa de tiempo del GPU que se registra en un punto, en el tiempo, especificado por el usuario. Utiliza un reloj propio del GPU, independiente del reloj del CPU. El tiempo resultante mediante eventos está en el orden de los mili-segundos. Es posible crear más de un evento, además de poder colocar un evento dentro de otro; esto último es útil cuando se requiere medir tiempos parciales [8], [6]. Para indicar la sintaxis del código se utilizará el color azul para diferenciarla del texto común, para mayor detalle revisar *CUDA API Reference Manual* [9].

La secuencia básica para el manejo de eventos es la siguiente:

1. Crear las variables, del tipo cudaEvent, de inicio y paro.
   cudaEvent_t start, stop;

2. Crear el evento de inicio y paro.
   cudaEventCreate(&start); cudaEventCreate(&stop);

3. Guardar la estampa de tiempo inicial.
   cudaEventRecord(start , 0);

4. Escribir la parte de código del *device* que se desea medir. Alguna tarea en el GPU: copias de memoria, ejecución de *kernel*.

5. Guardar la estampa de tiempo final. El segundo parámetro corresponde al *stream* en el que se está ejecutando éste código (por defecto se trabaja con el *stream* cero).
   cudaEventRecord(stop, 0);

6. Esperar a que el GPU termine su ejecución hasta la estampa de paro.
   cudaEventSynchronize(stop);

7. Calcular la diferencia entre la estampa final y la inicial. El tiempo se guarda en la variable *elapsedTime*, en milisegundos.
   float elapsedTime;
   cudaEventElapsedTime(&elapsedTime, start, stop);

8. Destruir los eventos de inicio y paro.
   cudaEventDestroy(stop); cudaEventDestroy(start);

Para visualizar todas las operaciones realizadas durante todo el programa, ver el tiempo y los recursos empleados; *Visual Profiler* de NVIDIA puede ser útil, el cual se proporciona de manera gratuita en el *CUDA Toolkit*. Además, la herramienta de depuración *Nsight* junto con *Visual Studio* o *Eclipse*, también es capaz de realizar un análisis de tiempos. Para más detalles ver el capítulo 3 de [10] y el capítulo 4 de [11].

2.2. Arquitectura GPU

2.2.1. Clasificación de sistemas de cómputo

El sistema de clasificación de Flynn divide a todo el mundo de computo en cuatro grupos: SISD (*Single Instruction Single Data*), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) y MIMD (Multiple Instruction Multiple Data). El sistema SISD son computadoras von Neumann con un sólo procesador, mientras que los sistemas SIMD son computadoras *pipeline*.

Las computadoras MIMD trabajan de manera asíncrona. Cada procesador ejecuta su propio programa con un control individual, por lo que multiples *threads* de control son ejecutados simultáneamente. Por otra parte, en las computadoras SIMD, los elementos de procesamiento están hechos de unidades aritméticas y lógicas (ALUs), memoria local y una unidad de comunicación para tener acceso a la red de interconexión. Debido a que un solo decodificador de instrucciones está presente, la ejecución de un programa SIMD es siempre síncrona.

Los arreglos sistólicos son una combinación de los sistemas SIMD MIMD y *pipeline*. Se maneja por un reloj principal, contiene multiples elementos de procesamiento interconectados. Se alimenta constantemente desde el exterior con datos hacia los procesadores, los cuales mediante su red de conexión pueden pasar resultados entre procesadores [12].

2.2.2. Características de los GPU

Los GPU son especiales para cómputo intensivo, altamente paralelo. Está diseñado para dedicar más transistores para el procesamiento de datos que a la captura de datos en caché y al control de flujo del programa (ver figura 2.3), esto último se debe a que el mismo programa es ejecutado en muchos *threads* en paralelo. Tienen su propia memoria (DRAM), memorias compartidas y registros, las cuales establecen su propia jerarquía de memoria.

Las arquitecturas de los GPUs se agrupan por su ***compute capability***, la cual se representa por "*y.x*", donde "*y*" es el número de revisión mayor y "*x*" es el número de revisión menor (por ejemplo, *compute capability* 2.1). *Devices* con el mismo número de revisión mayor tienen la misma arquitectura de núcleos. El número de revisión mayor es 3 para *devices* basados en la arquitectura *Kepler*, 2 para *devices* basados en la arquitectura *Fermi*, y 1 para *devices* basados en la arquitectura *Tesla*. El número de revisión menor corresponde a una mejora de la arquitectura en el incremento de los núcleos, además de la posibilidad de incluir nuevas características. [6]

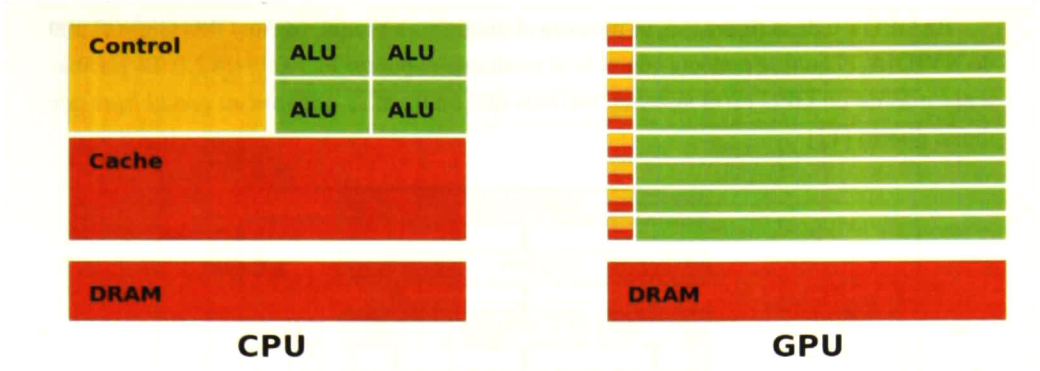


Figura 2.3: Diferencias CPU y GPU.

2.2.3. Multiprocesadores

La arquitectura GPU de NVIDIA está construida por un arreglo escalable de **Streaming Multiprocessors** (SMs). Un SM es un arreglo de núcleos llamados **Streaming Processor** (SP), los cuales realizan las operaciones aritméticas y lógicas de los *kernels*. Cuando en un programa CUDA, el *host* realiza un lanzamiento de *kernel*, los *blocks* del *blocks* son enumerados y distribuidos hacia los SMs disponibles para la ejecución del código de los *threads*. Los *threads* contenidos por un bloque se ejecutan al mismo tiempo en un multiprocesador. Cuando la ejecución de un bloque termina, nuevos bloques son distribuidos en los multiprocesadores vacantes.

Un SM está diseñado para ejecutar cientos de *threads* de manera concurrente. Para manejar un gran número de *threads*, emplea una arquitectura única llamada **SIMT** (*Single-Instruction, Multiple-Thread*). En la cual el SM crea, maneja, agenda y ejecuta los *threads* en grupos de 32 *threads* paralelos, llamados **warps**. Los *threads* que componen el *blocks* empiezan juntos pero tienen su propio contador de instrucciones y sus registros, lo que les permite ejecutarse independientemente. La arquitectura SIMT habilita a los programadores escribir código paralelo a nivel de thread, los cuales pueden ser independientes o trabajar de manera coordinada [6].

En la figura 2.4, se muestra el diagrama a bloques de un SM, como se puede observar, un SM cuenta con uno o más bloques **SPU** (Special-Purpose Unit), los cuales realizan instrucciones especiales de alta velocidad, operaciones tales como : seno, coseno y exponenciales. Además, cuenta con un bloque de registros que trabaja a la misma velocidad que los núcleos SP, por lo tanto no tiene tiempo de espera en ésta memoria. También cuenta con un bloque de memoria compartida, la cual es accesible sólo para un SM, puede ser usada como caché manejándola desde el código del *kernel*.

Cada SM tiene un bus separado para los espacios de memoria *texture*, constante y global. La memoria *texture* y constante son de solo lectura y se encuentran sobre la memoria global. La memoria global es suministrada via **GDDR** (Graphic Double Data Rate) cuyo bus puede ser de hasta 512 bits, dando un ancho de banda de 5 a 10 veces más que el encontrado en CPUs, arriba de 190 GB/s con el hardware Fermi.

En GPU # 0 de la figura 2.5, se muestra el diagrama a bloques de un GPU G80/GT200 de NVIDIA, el cual se conecta con el *host* mediante el puerto PCI express 2.0. Es posible tener más de un GPU en el sistema, mismos que también se comunican con el *host* por dicho puerto [11].

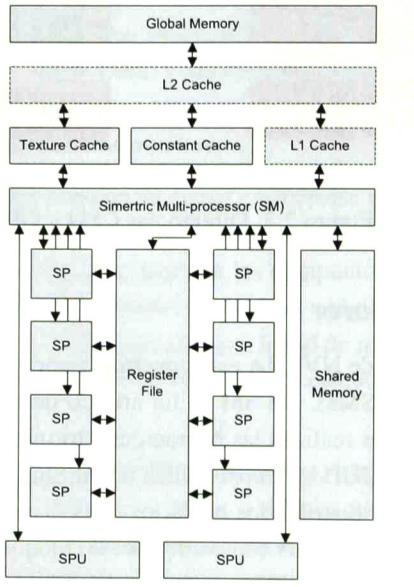


Figura 2.4: Diagrama de un streaming processor (SP)

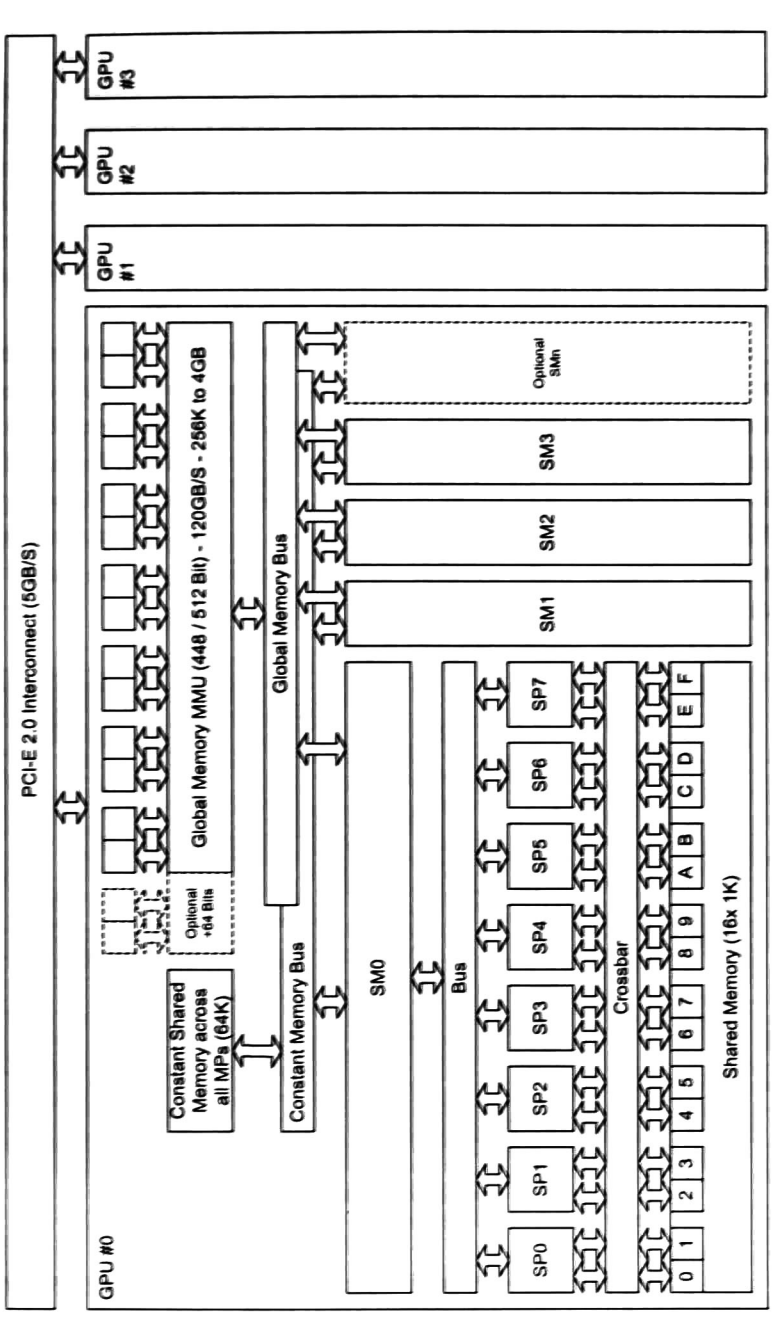


Figura 2.5: Diagrama a bloques de un GPU (G80/GT200).

2.2.4. Jerarquía de memoria

En CUDA, la memoria del *host* y la del *device* son entidades separadas. Por lo que es necesario hacer llegar los datos del CPU hacia el GPU, para que sean procesador por el GPU y posteriormente entregar los resultados hacia el CPU. El *device* cuenta con una memoria DRAM, en la cual, como se muestra en la figura 2.6, se encuentran las memorias: global, local, constante (*constant*) y *texture*. En el GPU se encuentran los SM, los cuales tienen su propia memoria compartida (*shared*) y registros, además de memorias cache para el manejo de las memorias constante y *texture*.

La jerarquía de memoria se establece al observar el ámbito de las variables en los diferentes tipos de memoria, además de observar quien puede escribir y leer en ellas. Para visualizar esto, nos basaremos en la figura 2.7, en la cual con flechas con una dirección se indica que son de solo lectura, mientras que con una flecha bidireccional se indica que puede ser de lectura y escritura. Como vemos, el CPU solo puede leer y escribir en la memoria DRAM (*global, constant y texture*). Los *threads* pueden leer de la memoria compartida propia del *block* que lo contiene, tienen su propio bloque de registros y memoria local; además, tienen acceso a la DRAM.

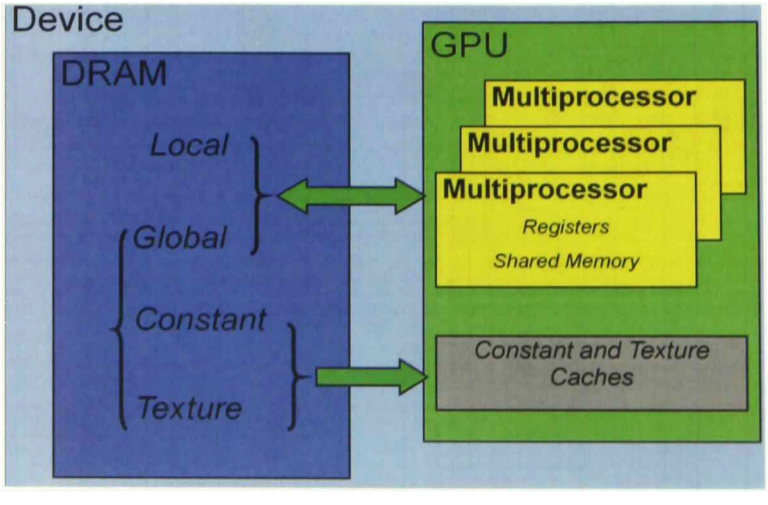


Figura 2.6: Localización de memorias del *device*

La tabla 2.2 muestra como diferenciar y declarar las variables en los diferentes espacios de memoria; así como el ámbito y el tiempo de vida que tendrán. Por ejemplo utilizando el calificador de variable __constant__, se reservará espacio en la memoria constante, la cual se encuentra en la DRAM, todos los *threads* de todos los bloques podrán leer de ella, mientras no se libere dicha memoria seguirá reservada en el *device* durante toda la aplicación. Entendemos como aplicación a la ejecución de todo el *código host* y todo el *código device*, pudiéndose realizar más de un lanzamiento de *kernel* durante su tiempo de ejecución.

Tabla 2.2: Identificadores de variables.

Declaración de variable	Memoria que utiliza	Ámbito	Tiempo de vida
Variables diferentes a los arreglos	Registro	*Thread*	*Kernel*
Arreglos	Local	*Thread*	*Kernel*
__shared__	Compartida	*Block*	*Kernel*
__device__	Global	*Grid*	Aplicación
__constant__	Constante	*Grid*	Aplicación

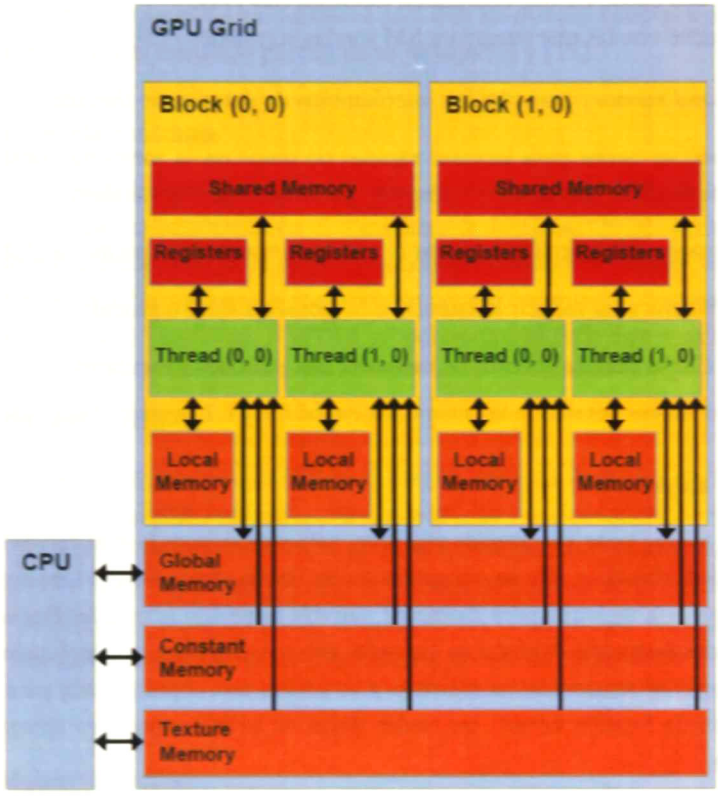


Figura 2.7: Ámbito de las memorias del *device*

El ancho de banda y la latencia son dos conceptos importantes para describir las características de las memorias. **Ancho de banda**, es la cantidad de datos que se pueden leer o almacenar en la DRAM en un periodo dado de tiempo. **Latencia**, es el tiempo que toma en realizar la petición de búsqueda (*fetch request*) de alguna localidad de memoria [11]. A continuación se presentarán los diferentes tipos de memorias que se encuentran en el *device*, puntualizando sus características más representativas.

2.2.4.1. Memorias Caché

Una ***caché*** es un banco de memoria de alta velocidad que se encuentra físicamente cercano al núcleo de procesamiento. Son caras en términos de silicio, lo cual se traduce en grandes chips, baja producción y procesadores más caros. La máxima velocidad de una caché es inversamente proporcional al tamaño de la caché. La caché L1 es la más rápida, pero está limitada en tamaño generalmente alrededor de 16K, 32K o 64K. Por lo general es asignada a un solo procesador. La caché L2 es más lenta, pero más larga, típicamente de 256K a 512K.

La arquitectura *Fermi* fue la primera implementación GPU, en incorporar una caché L1 por SM y una caché L2 que comparten todos los SM [11].

Las caché con las que cuenta un SM son las siguientes [13]:

- *Shared memory* para rápidos intercambios de datos entre *threads*.
- *Constant cache* para lecturas rápidas de datos en la memoria constante, con la capacidad de que un dato se lea por los *threads* que lo soliciten.
- *Texture cache* para incrementar el ancho de banda de la memoria global.
- *L1 Cache* para reducir la latencia a la memoria local o global.
- *L2 Cache* acelera accesos de memoria con patrones irregulares.

Para mayor detalle de las memorias caché L1 y L2, revisar el capítulo 5 de [10].

2.2.4.2. Registros

Cada *thread* tiene sus propios registros, esta memoria es la más rápida de todas por su localidad, debido a que se encuentra en los multiprocesadores. Una función *kernel* típicamente usa registros para mantener variables que son accesadas frecuentemente y son privadas para cada *thread*. Las variables (no arreglos) que son declaradas en el cuerpo del *kernel* se almacenan en registros y se genera una copia privada para cada *thread* que ejecuta la función *kernel*, las cuales dejan de existir cuando su *thread* termina su ejecución.

2.2.4.3. Memoria local

Accesos de memoria local ocurren solo para variables automáticas. Una variable automática es aquella que se declara en el *código device* sin ninguno de los calificadores __device__, __shared__, o __constant__ . Generalmente, las variables automáticas residen en registros, excepto los casos siguientes:

Arreglos que podrían consumir demasiado espacio en la memoria de registros.

Cualquier variable que el compilador decide pasar a la memoria local, cuando el *kernel* utiliza más registros que los disponibles por SM.

2.2.4.4. Memoria compartida

La memoria compartida (*shared*) es una caché L1 controlada por el usuario. Se crea una versión privada por cada*block*, cada *thread* en el *block* puede tener acceso a dicha memoria durante la ejecución del *kernel*, mientras que, *thread* de otros *blocks* no pueden hacerlo. Una vez que el *kernel* termina su ejecución, el contenido de las varibles *shared* dejan de existir.

Los accesos a la memoria compartida es sumamente rápido y altamente en paralelo. Los programadores de CUDA a menudo usan memoria compartida para mantener una porción de los datos de la memoria global que es altamente usada en la fase de ejecución del *kernel*. Además de que es un medio eficaz para que los *threads* cooperen al compartir los datos de entrada y el resultado parcial de su trabajo [5] y [11].

2.2.4.5. Memoria constante

Memoria de solo lectura para el *device* que es escrita por el *host*. Se utiliza para almacenar datos que no cambiarán en el transcurso de la ejecución del *kernel*. Las variables *constant* se almacenan en la memoria global pero utilizan la caché *constant* para su acceso eficiente. Con patrones apropiados de acceso, se pueden conseguir accesos rápidos y en paralelo. El hardware NVIDIA proporciona 64KB de ésta memoria.

2.2.4.6. Memoria Global

El *host* puede transferir datos de ella, es el tipo de memoria de mayor capacidad. Los accesos a la memoria global son lentos, sin embargo, las variables globales son visibles para todos los *threads* de todos los *kernels*. Su contenido se mantiene durante toda la aplicación. Por lo tanto, variables globales se pueden utilizar como un medio para que los *threads* colaboren (compartan datos) a través de los bloques. Las variables globales se utilizan frecuentemente para pasar información de un lanzamiento de *kernel* a otro [5].

2.2.4.7. Memoria Texture

Está montada sobre la memoria global, con mecanismos internos para su acceso en una, dos o hasta tres dimensiones. Es memoria de solo lectura para el *device*, es escrita por el *host*. Para el manejo de la memoria utiliza arreglos CUDA (***cuda array***), los cuales tienen características para su manejo en 2 y 3 dimensiones, solo pueden ser leídos por los *kernel* a travez de funciones *texture* y debe ser ligado a una referencia *texture*.

A continuación se presentará la secuencia básica para el manejo de Texture en 3 dimensiones. En azul se presenta fracciones del código ejemplo del apéndice A.2.
En el cual se lee un arreglo de 3 dimensiones, se le suma uno y el resultado se copia en un arreglo lineal. Con esto se puede observar el manejo de un arreglo de 3 dimensiones y como guardarlo en un arreglo lineal.

1. Declaración de las referencias *texture* de 3 dimensiones, esto se debe realizar fuera de la función principal.
   texture<float, cudaTextureType3D, cudaReadModeElementType> textReference;

2. Asignación de memoria en el *device* mediante *CUDA arrays*, estableciendo el tipo de dato del arreglo mediante *cudaCreateChannelDesc*.
   cudaChannelFormatDesc channelDesc;
   channelDesc = cudaCreateChannelDesc<float>();

3. Declaración de los *CUDA array*.
   cudaArray *cudaArray

4. Dimensionamiento del CUDA array, estableciendo su volumen.
   volumeSize = make_cudaExtent(size, size, size);
   cudaMalloc3DArray(&cudaArray, &channel, volumeSize);

5. Preparación de los parámetros de copia en tres dimensiones. Estableciendo el tipo y tamaño de la copia. *make_cudaPitchedPtr* cambia el arreglo lineal *host_CubeMatrix* en un arreglo tridimensional.
   copyparms.extent = volumeSize;
   copyparms.dstArray = cudaArray; // Arreglo destino.
   copyparms.kind = cudaMemcpyHostToDevice;
   copyparms.srcPtr = make_cudaPitchedPtr((void*)host_CubeMatrix, ...
   sizeof(float)*size, size, size); // Arreglo fuente.

6. Envió de los datos hacia el *device*.
   cudaMemcpy3D(&copyparms);

7. Ligar el *CUDA array* con la referencia *texture*.
   cudaBindTextureToArray(textReference, cudaArray, channel);

8. Ejecutar la función *kernel*.
   kernel<<< dimGrid, dimBlock >>>(*device*_CubeMatrix, size);

9. Copiar el arreglo procesado cuOutputArray hacia el *host*.
   cudaMemcpy(host_CubeMatrix, *device*_CubeMatrix, ...
   sizeof(float)*size*size*size, cudaMemcpyDeviceToHost);

10. Desenlazar la referencia *texture* para liberar el recurso.
    cudaUnbindTexture(textReference);

11. Liberar los *CUDA array* del *device*.
    cudaFreeArray(cudaArray);

2.2.4.8. Memoria Surface

La memoria ***surface*** es similar a *texture*, con la diferencia que tiene funciones de lectura y de escritura. En azul se presenta fracciones del código ejemplo del apéndice A.3. En el cual se lee un arreglo de 2 dimensiones, se le suma uno y el resultado se copia en un arreglo de 2 dimensiones. Con esto se puede observar el manejo de un arreglo de 2 dimensiones, esto es útil cuando se requiere procesar algoritmos que utilicen arreglos matriciales.

La secuencia básica para el manejo de Surface en 2 dimensiones es la siguiente.

1. Declaración de las referencias *surface* de 2 dimensiones, esto se debe realizar fuera de la función principal.
   surface<void, cudaSurfaceType2D> inputSurfRef;
   surface<void, cudaSurfaceType2D> outputSurfRef;

2. Asignación de memoria en el *device* mediante *CUDA arrays*, estableciendo el tipo de dato del arreglo mediante *cudaCreateChannelDesc*.
   cudaChannelFormatDesc channelDesc;
   channelDesc = cudaCreateChannelDesc<float>();

3. Declaración de los *CUDA array*.
   cudaArray* cuInputArray; cudaArray* cuOutputArray;

4. Dimensionamiento del CUDA array, estableciendo su volumen.
   cudaMallocArray(&cuInputArray, &channelDesc, ...
   width, height, cudaArraySurfaceLoadStore);
   cudaMallocArray(&cuOutputArray, &channelDesc, ...
   width, height, cudaArraySurfaceLoadStore);

5. Envió de los datos hacia el *device*.
   cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size, ...
   cudaMemcpyHostToDevice);

6. Ligar el *CUDA array* con la referencias *surface*.
   cudaBindSurfaceToArray(inputSurfRef, cuInputArray);
   cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);

7. Ejecutar la función *kernel*.
   copyKernel<<< dimGrid, dimBlock >>>(width, height);

8. Copiar el arreglo procesado cuOutputArray hacia el *host*.
   cudaMemcpyFromArray(h_data_out, cuOutputArray, 0, 0, size, ...
   cudaMemcpyDeviceToHost);

9. Liberar los *CUDA array* del *device*.
   cudaFreeArray(cuInputArray);
   cudaFreeArray(cuOutputArray);

2.3. Buenas prácticas de programación

2.3.1. Utilización de los multiprocesadores

Una de las claves para buen desempeño es mantener ocupados los multiprocesadores en el *device* lo más posible. Es importante diseñar la aplicación para que utilicen los *blocks* y *threads* de tal manera que maximice la utilización de los recursos de hardware del *device* (multiprocesadores, memoria compartida y registros). Las instrucciones en los *threads* se ejecutan secuencialmente, y como resultado, ejecutar otros *warps* cuando un *warp* está en pausa es la única manera de ocultar las latencias y mantener el hardware ocupado.

Es importante una métrica para la determinación de cuan efectivamente el hardware se mantiene ocupado. Esta métrica es *occupancy*. **Occupancy** es la razón del número de *warps* activas por multiprocesador (*warps1*) y el número máximo de las *warps* activas posibles (*warps2*). (El último número está determinado por el hardware del GPU que se esté usando). Otra manera de ver *occupancy* es el porcentaje de la capacidad del hardware para procesar *warps* que están activamente en uso [8].

$$occupancy = \frac{warps1}{warps2}$$

NVIDIA proporciona una herramienta en excel que calcula la *occupancy*, el cual facilita a los desarrolladores afinar el optimo balance de recursos del GPU y para probar diferentes escenarios más facilmente. Esta herramienta llamada *"CUDA Occupancy Calculator"* se encuentra en el subdirectorio *tools* de *CUDA Toolkit installation*. Los recursos que impactan sobre *occupancy* son el dimensionamiento de *blocks* y *threads*, los registros en los *threads* y la memoria compartida por los *blocks*. Para ilustrar como impactan estos recursos en la *occupancy*, se utilizará la herramienta antes mencionada.

Las características y las especificaciones técnicas del *device* se presentan de manera detallada en el apéndice F.1 de *"NVIDIA CUDA C Programming Guide"* [6] y están agrupadas según su *compute capability*. A continuación se presentará un ejemplo para el cálculo de la *occupancy*, para esto, tomaremos como base un *device* de *compute capability* 2.0, cuyas características son las siguientes:

- Número máximo de *threads* por multiprocesador es 1536.
- Número de *threads* por *blocks* es 32.
- Número máximo de *warps* activas por multiprocesador es 48.
- Número máximo de *blocks* por multiprocesador es 8.

- Número máximo de *threads* en un *block* es 1024.
- Numero máximo de registros de 32 bits por multiprocesador es 32000.
- La cantidad máxima de memoria compartida por multiprocesador es 49182 bytes.

Para tener una *occupancy* del 100 %, se deben de activar las 48 *warps*, lo cual implica que se utilizarán los 1536 *threads* disponibles por multiprocesador. Para que esto pase, el producto del número de *blocks* por multiprocesador y el número de *threads* por *block* debe ser 1536. Si elegimos utilizar los 8 bloques por multiprocesador, a cada *block* le corresponderá 192 *threads*, mientras que, si elegimos tener solo un *block*, máximo se podrá utilizan los 1024 *threads*. Esto último acarreará tener una *occupancy* del 67 %, debido a que con los 1024 *threads* se utilizarán solo 32 *warps* de las 48 disponibles por multiprocesador (32 *warps* utilizadas / 48 *warps* disponibles * 100 = 67 %). La figura 2.8 muestra el impacto de variar el número de *threads* por *block* en la activación de las *warps*. Para una *occupancy* del 100 % se debe elegir el número de *threads* por *block* de manera que se utilicen las 48 *warps* disponibles.

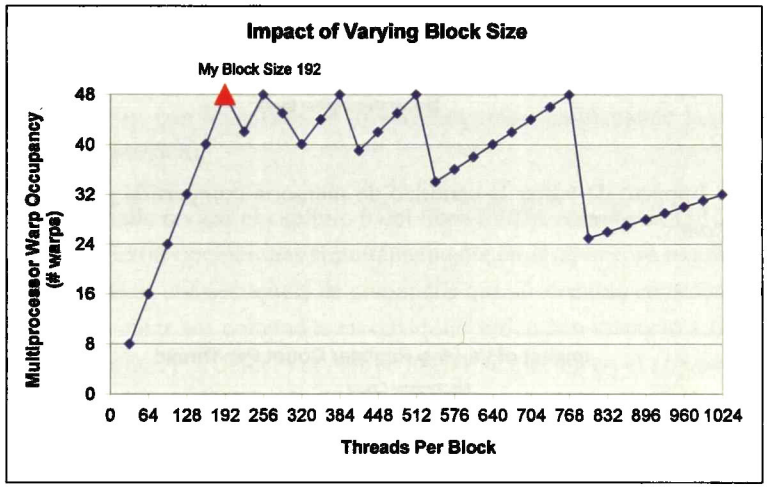


Figura 2.8: Impacto de variar el tamaño de los *blocks* en la *occupancy*.

Otro recurso que impacta en la *occupancy* son los registros. El almacenamiento en registros permite a los *threads* mantener variables locales cercanas para su rápido acceso. El número máximo de registros por multiprocesador es de 32000, los cuales se reparten en los *threads* que maneje dicho multiprocesador. Por ejemplo, si se utilizan los 1536 *threads*, entonces le corresponden 20 registros máximo por *thread*, para tener una *occupancy* del 100 %. La figura 2.10 muestra el impacto de variar el número de registros en la utilización de las *warps*. Entre más se sobrepase el límite de 20 registros, menos *warps* podrán utilizarse para el computo del *kernel*, lo cual impactará notablemente en la *occupancy*.

La memoria compartida también puede ser una restricción en la *occupancy*. El volumen total de memoria compartida (*shared*) se reparte en el número de *blocks* que se hayan creado por multiprocesador. En nuestro ejemplo, se tiene con 49152 bytes y se estableció que se utilizarían 192 *threads* por *block*, lo cual permite tener hasta 8 *blocks* en el multiprocesador, por consiguiente, le corresponde hasta 6144 bytes a cada *block*. En la figura 2.9 se muestra como afecta el sobrepasar el límite de 6144 bytes para una *occupancy* del 100 %.

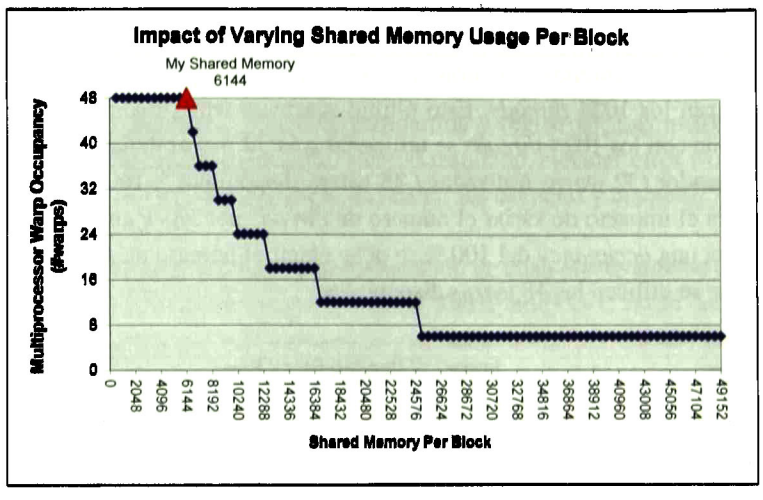


Figura 2.9: Impacto de variar la cantidad de memoria compartida por *block* (en bytes) en la *occupancy*.

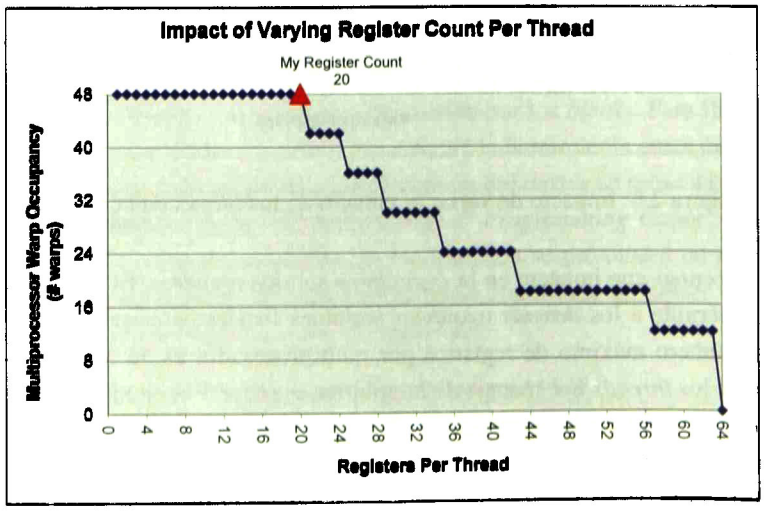


Figura 2.10: Impacto de variar el número de registros por *thread* en la *occupancy*.

2.3.2. Concurrencia *host/device*

Además de la concurrencia a nivel de *threads*, CUDA también cuenta con algunas herramientas de hardware y software que permiten realizar operaciones al mismo tiempo en el *host* y en el *device*. Dichas operaciones son las siguientes:

- **Concurrencia CPU/GPU:** Debido a que el CPU y el GPU son dispositivos separados, pueden operar independiente el uno del otro. Las funciones *kernel* se pueden ejecutar a la par de alguna función en el *host*, si se desea que el *host* espere a que el *device* termine de ejecutar alguna función se puede utilizar la barrera de sincronía *cudaDeviceSynchronize()* . *El programa debe ser particionado de manera que el código con poco o ningún paralelismo de datos, se ejecute en el host, mientras que el código con alto paralelismo de datos se ejecute en el device.*

- **Concurrencia Memcpy/kernel:** Algunos GPUs tienen la capacidad de tener más de un mecanismo de copia, por lo que las transferencias de datos entre el *host* y el *device* pueden ser realizadas mientras los SMs están procesando *kernels*. *Para verificar esto, se requiere verificar la propiedad del device llamada asyncEngineCount, la cual es 1 para devices que lo soportan.*

- **Concurrencia Multi-GPU:** Es posible conectar varios GPU mediante el puerto *PCI-express*, con los cuales se incrementa considerablemente la capacidad de computo en paralelo.

Para mayor detalle revisar el capítulo 6 del libro *CUDA Handbook* [13].

Para ejecutar ciertas operaciones simultáneamente en el *device*, se requiere el uso de *streams*. Un **stream** es una secuencia de comandos que se ejecutan en orden. Diferentes *streams* pueden ejecutar sus comandos en cualquier orden con respecto a otro *stream* o de manera concurrente. Por defecto el *código device* se ejecuta en el *stream* cero, por lo tanto, si no se especifica el parámetro correspondiente al *stream* en un lanzamiento de *kernel* o en una transferencia de memoria, se trabajará dentro del *stream* cero.

Al igual que los eventos, los *streams* se pueden sincronizar. Con *cudaDeviceSynchronize*, el *device* espera a que todos los comandos predecesores en todos los *streams* han terminado. *CudaStreamSynchronize* toma como parámetro un determinado *stream* y espera a que todos los comandos predecesores a dicho *stream* terminen. Con *cudaStreamWaitEvent()*, un determinado stream espera a que un evento, dado como parámetro, termine antes de continuar con su ejecución de comandos. Para verificar que un stream ha terminado su ejecución, se emplea la instrucción *cudaStreamQuery()* [6].

2.3.3. Optimizaciones de memoria

Las optimizaciones de memoria son de vital importancia para mejorar rendimiento. Su objetivo es el de maximizar el uso del hardware para obtener el mayor el ancho de banda posible durante las transferencias de datos.

- Es importante minimizar las transferencias de datos entre el *host* y el *device*, incluso si eso significa ejecutar más *kernels* en el GPU que no demuestran aceleración comparándolo con ejecutarlo en el CPU.
- Largas transferencias de memoria tiene mejor rendimiento que realizar numerosas transferencias pequeñas, esto debido al encabezado asociado a cada transferencia.
- Para incrementar la utilización del *device* y reducir tiempos, es recomendable realizar las transferencias de memoria mientras se está ejecutando funciones *kernel*.
- Es recomendable utilizar memoria compartida para evitar transferencias redundantes de la memoria global. La secuencia típica del manejo de esta memoria es la siguiente:
  - Cargar los datos de la memoria global a la memoria compartida.
  - Sincronizar todos los *threads* del *block* para que cada *threads* pueda leer el dato correcto que pudo ser cargado por algún otro *thread* del *block*.
  - Procesar los datos de la memoria compartida.
  - Sincronizar de nuevo, si es necesario, para asegurar que la memoria compartida ha sido actualizada con los resultados.
  - Escribir el resultado en la memoria global. Si ya no se requiere mantener dichos datos en el *device*, se procede a transferir los resultados de la memoria global al *host* [6].

- *La caché texture está optimizada con localidad espacial en dos dimensiones, por lo tanto, threads del mismo blocks que leen direcciones texture o surface que se encuentran cercanas tendrán mejor desempeño.* Leer la memoria del *device* a través de una referencia *texture* o *surface* puede ser una alternativa ventajosa para leer memoria global, los beneficios son los siguientes:
  - Mayor ancho de banda se puede obtener si los accesos corresponden a localidades de memoria cercanas, aun cuando las lecturas no siguen los patrones de acceso que se deben respetar para buen desempeño (ver las secciones 5.3.2.1 y 5.3.2.4 de *CUDA Programming Guide* [6]).
  - Cálculos para el direccionamiento de la memoria se realizan fuera del *kernel* por unidades dedicadas.
  - Datos de entrada de 8 y 16 bit pueden ser convertidos opcionalmente a valores de punto flotante de 32 bits en el rango de [0.0, 1.0] o [-1.0, 1.0] [6].
- Asignar y liberar memoria del *device* mediante *cudaMalloc()* y *cudaFree()* son operaciones costosas, por lo tanto, se debe buscar reutilizar la memoria del *device* lo más posible para reducir su impacto en el desempleño total de la aplicación [8].

2.4. Resumen del capítulo

El capítulo presenta una base introductoria a los GPU y a la programación en CUDA C. En donde el GPU trabaja en conjunto con el CPU, los cuales se comunican mediante el puerto *PCI express*. El lenguaje de programación que se utiliza es CUDA C, el cual está basado en el lenguaje tradicional C, al cual se le agregaron funciones especiales para el manejo de la memoria y de los procesadores del GPU. Se presenta una secuencia básica de un programa en CUDA, además de un ejemplo completo, el cual puede ser utilizado para comprobar la correcta instalación de los controladores del GPU.

Uno de los temas cruciales en la programación en CUDA es el manejo y dimensionamiento de los *threads* para maximizar la utilización de los SM, por lo cual se dedica una sección en donde se presenta como dimensionar todo el conjunto de *threads* para el procesamiento paralelo de datos.

El correcto manejo de la memoria es vital para obtener buenos resultados en los cálculos y para reducir tiempos de ejecución. Una sección se dedica al manejo de la memoria, en donde se muestran las características de los diferentes tipos de memoria y como transferir datos entre el CPU y el GPU. CUDA C tiene funciones especiales para manejar la memoria DRAM del GPU en 2 y 3 dimensiones mediante *texture* y *surface*, se presenta la secuencia básica para utilizar *texture* en 3 dimensiones y otra secuencia para manejar *surface* de 2 dimensiones.

Al final del capítulo se presenta una sección en donde se puntualizan buenas prácticas de programación en CUDA, como la utilización de los multiprocesadores, la concurrencia entre el CPU y el GPU, y el manejo de memoria.

Capítulo 3

Transformación de algoritmos con bucles anidados

3.1. Antecedentes matemáticos

3.1.1. Vectores y matrices

Los vectores enteros en $\mathbb{Z}^n$ poseen n elementos, se manejarán por defecto como vectores fila con elementos enteros, por ejemplo $\mathbf{i} = (1, 2, \cdots, n)$. El vector nulo, $\mathbf{0}$, tiene todos sus n elementos igual a cero.

A continuación enumeraremos algunas relaciones de orden entre los vectores $\mathbf{a}$ y $\mathbf{b}$ de n elementos:

- $\mathbf{a}$ y $\mathbf{b}$ son iguales, escrito $\mathbf{a} = \mathbf{b}$, si $a_i = b_i, 1 \leq i \leq n$.
- $\mathbf{a}$ es menor que $\mathbf{b}$, escrito $\mathbf{a} < \mathbf{b}$, si $a_i < b_i, 1 \leq i \leq n$.
- $\mathbf{a} \leq \mathbf{b}$, si $a_i \leq b_i, 1 \leq i \leq n$, que no es lo mismo que $\mathbf{a} < \mathbf{b} \vee \mathbf{a} = \mathbf{b}$.
- $\mathbf{a}$ es lexicográficamente menor que $\mathbf{b}$, escrito $\mathbf{a} < \mathbf{b}$, si existe un j, $1 \leq j \leq n$, tal que $\mathbf{a} <_j \mathbf{b}$.

  $\mathbf{a}$ es lexicográficamente menor que $\mathbf{b}$ a nivel j, escrito $\mathbf{a} <_j \mathbf{b}$, si $a_i = b_i, 1 \leq i < j \leq n$ y $a_j < b_j$.

  $\mathbf{a} \leq \mathbf{b}$, si $\mathbf{a} < \mathbf{b} \vee \mathbf{a} = \mathbf{b}$.

Un vector lexicográficamente positivo es un vector cuyo primer elemento (de izquierda a derecha) diferente de cero, es positivo, por ejemplo $(0, 2, -11, 0)$. De manera similar, un vector lexicográficamente negativo es un vector cuyo primer elemento (de izquierda a derecha) diferente de cero, es negativo, por ejemplo $(0, -2, 11, 0)$.

El vector signo es el signo de sus elementos. Por lo tanto, si **d** = (2,-3,0), entonces **sig(d)** = $(sig(2), sig(-3), sig(0)) = (1, -1, 0)$.

Una matriz real **A** de *n x m* tiene *n* filas y *m* columnas de números de números reales

$$\mathbf{A}^{mxn} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

El vector $\mathbf{a}_t$ contiene los elementos de la fila *t* de la matriz $\mathbf{A}^{mxn}$, por ejemplo, $\mathbf{a}_2 = (a_{21}, a_{22}, \cdots, a_{2m})$. El vector $\mathbf{a}^r$ contiene los elementos de la columna *r* de la matriz $\mathbf{A}^{mxn}$, por ejemplo, $\mathbf{a}^1 = (a_{11}, a_{21}, \cdots, a_{n1})$.

3.1.2. Reducción de matrices a la forma escalonada

Reducir una matriz de *m x n* a la forma escalonada (echelon form) significa encontrar una matriz unimodular **U** de *m x n* y una matriz escalonada **S**, tal que **UA** = **S**. Esta reducción siempre es posible; las matrices **U** y **S** no son únicas. A continuación se proporciona un método para encontrar tales matrices.

Algoritmo 3.1 Reducción escalonada de una matriz

Entrada: Matriz **A** entera de *m x n*.
Salida: Matriz unimodular **U** y la matriz escalonada **S** de *m x n*, tal que **UA** = **S**

```
U ← I_m, S ← A, i_0 ← 0
for j = 1, n, 1 do
    if existe al menos un elemento s_ij no cero con i_0 < i ≤ m then
        i_0 ← i_0 + 1
        for i = m, i_0 + 1, −1 do
            while s_ij ≠ 0 do
                σ ← sign(s_(i−1)j * s_ij)
                τ ← ⌊|s_(i−1)j|/|s_ij|⌋
                Multiplicar στ por la fila i y restarlo de la fila (i − 1) en (U; S)
                Intercambiar las filas i e (i − 1) en (U; S)
            end while
        end for
    end if
end for
```

En algunas aplicaciones, para una matriz **A** dada, se necesita encontrar una matriz unimodular **V** y una matriz escalonada **S**, tal que **A** = **VS**. **V** podría ser la inversa de la matriz unimodular **U** del algoritmo 3.1. Sin embargo, si **U** no se necesita, el algoritmo 3.1 puede ser modificado para producir directamente a **V** y **S** tal que **A** = **VS**. [14]

Algoritmo 3.2 Reducción escalonada de una matriz (modificada)

Entrada: Matriz **A** entera de m x n.

Salida: Matriz unimodular **V** y la matriz escalonada **S** de m x n, tal que **A** = **VS**

```
V ← I_m, S ← A, i_0 ← 0
for j = 1, n, 1 do
    if existe al menos un elemento s_ij no cero con i_0 < i ≤ m then
        i_0 ← i_0 + 1
        for i = m, i_0 + 1, −1 do
            while s_ij ≠ 0 do
                σ ← sign(s_(i−1)j * s_ij)
                τ ← ⌊|s_(i−1)j|/|s_ij|⌋
                Multiplicar στ por la fila i y restarlo de la fila (i − 1) en S
                Multiplicar στ por la columna i − 1 y sumarlo a la columna i en V
                Intercambiar las filas i e (i − 1) en S
                Intercambiar las columnas i e (i − 1) en V
            end while
        end for
    end if
end for
```

3.1.3. Solución de sistemas de desigualdades lineales

Una desigualdad lineal con n incógnitas puede expresarse:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq c \tag{3.1}$$

donde $a_1, a_2, \cdots, a_n$ son escalares. De manera similar, un sistema de ecuaciones de desigualdades puede expresarse en su forma matricial:

$$\mathbf{Ax} \leq \mathbf{c} \tag{3.2}$$

El problema es encontrar el vector **x** tal que la desigualdad 3.2 se cumpla. En el capítulo 2 de la tesis [4] se aborda este tema y se presenta el algoritmo de eliminación de Fourier - Motzkin, además de dos métodos de simplificación de sistemas de desigualdades: algoritmo de simplificación Ad - Hoc y el algoritmo de simplificación Exacta.

3.2. Modelado de bucles anidados

Un **bucle anidado** o un nido de bucles es un conjunto de bucles en el que un bucle contiene uno o más bucles del tipo *for*. Un **nido de bucles perfecto** es un bucle anidado en el que no existen sentencias entre un bucle y otro, es decir, todas las sentencias se encuentran dentro del bucle más interno. Escribiremos $\mathbf{L} = (L_1, L_2, \cdots, L_m)$ para representar en forma vectorial el nido de m bucles. Cada bucle tiene una **variable índice**

asociada, que en conjunto forman el **vector de iteración** $\mathbf{x} = (x_1, x_2, \cdots, x_m)$. Los límites inferiores y superiores de una variable índice x_r, correspondiente al bucle L_r, son p_r y q_r, respectivamente, donde $1 \leq r \leq m$. Los pasos de las variables índice son unitarios. El cuerpo del nido de bucles es $H(x_1, x_1, \cdots, x_m)$ o $H(\mathbf{x})$, el cual es una secuencia de sentencias de asignación. A continuación se presenta de manera general un nido con m de bucles **L**:

```
L1 :  for x1 = p1, q1, 1 do
L2 :      for x2 = p2, q2, 1 do
 ⋮            ⋮
Lm :          for xm = pm, qm, 1 do
                  H(x1, x1, ··· , xm)
              end for
          end for
      end for
```

Un **espacio de iteración** se asocia a un nido de bucles **L**, el cual contiene un punto, en el espacio m dimensional, por cada iteración. Dicho **punto de iteración** se representa por $\mathbf{i} = (i_1, i_2, \cdots, i_m)$, donde las valores $i_1, i_2, \cdots, i_m$ corresponden a los valores que toman $x_1, x_2, \cdots, x_m$, respectivamente, por ejemplo: $\mathbf{i} = (p_1, q_2, \cdots, p_m)$.

El espacio de iteración se puede representar mediante el conjunto de dos desigualdades llamadas las *restricciones* en **x**:

$$\left.\begin{array}{rcl} \mathbf{p} & \leq & \mathbf{xP} \\ \mathbf{xQ} & \leq & \mathbf{q} \end{array}\right\} \tag{3.3}$$

Donde los vectores **p** y **q** (con m elementos) son respectivamente los vectores de límite inferior y superior; **P** y **Q** son las matrices de m x m de límite inferior y superior. La ecuación 3.3 se puede escribir ahora como un politopo. Un **politopo** en $\mathbb{R}^m$ es un conjunto acotado de la forma $\{\mathbf{x} \in \mathbb{R}^m : \mathbf{xA} \leq \mathbf{c}\}$ para alguna matriz real **A** y un vector real **c**. Para modelar el espacio de iteración para nuestro nido de bucles original, se escribirá en forma de politopo y se llamará **politopo fuente** . El politopo fuente se representará como $\mathbf{xA} \leq \mathbf{b}$, donde **A** se compone de la concatenación de filas de las matrices $\mathbf{Q}_0 = -1 * \mathbf{Q}$ y **P**; de manera similar **b** se compone de la concatenación de filas los vectores $\mathbf{p}_0 = -1 * \mathbf{p}$ y **q**. [15]

3.3. Dependencia de datos

La dependencia de los datos en un nido de bucles es lo que restringe las iteraciones que se pueder ejecutar en paralelo. De allí la importancia de hacer un correcto análisis de

dependencia entre las iteraciones en bucles. Una iteración $H(\mathbf{j})$ depende de una iteración diferente $H(\mathbf{i})$, si las siguientes condiciones se cumplen:

1. **$H(\mathbf{i})$ es ejecutada antes que $H(\mathbf{j})$ en L. esto es, $\mathbf{i} \prec \mathbf{j}$.**

2. **Existe una localidad de memoria a la que se le hace una referencia (de lectura o escritura) por ambas iteraciones, y al menos una de las dos referencias es de escritura.**

3. **Que una localidad de memoria no es escrita por ninguna iteración $H(\mathbf{k})$ que viene entre $H(\mathbf{i})$ y $H(\mathbf{j})$, tal que $\mathbf{i} \prec \mathbf{k} \prec \mathbf{j}$.**

Los tipos de dependencia de datos son los siguientes:

- **De flujo: ocurre cuando una variable es asignada o definida en una sentencia y es usada en una sentencia posterior. También se le conoce como *read after write* o dependencia verdadera.**

- **Anti-dependencia: ocurre cuando una variable es usada en una sentencia y es reasignada por una sentencia posterior. También se le conoce como *write after read*.**

- **De salida: ocurre cuando una variable es asignada en una sentencia y se modifica su valor en una sentencia posterior.**

- **De entrada: ocurre cuando una variable es usada por una sentencia y posteriormente se vuelve a utilizar en otra sentencia.**

Las dependencias de salida y anti-dependencia surgen por el reuso o reasignación de variables, también son llamadas dependencias falsas. La dependencia de flujo es llamada dependencia *verdadera* ya que es inherente en el cálculo y no puede ser eliminada por el uso de variables temporales. [16]

Si una iteración $H(\mathbf{j})$ depende de una iteración $H(\mathbf{i})$, entonces la diferencia $\mathbf{d} = \mathbf{j} - \mathbf{i}$ entres dos puntos de iteración se le llama vector distancia para el nido de bucles L. Ya que por definición $\mathbf{i} \prec \mathbf{j}$, tenemos $\mathbf{d} \succ \mathbf{0}$, esto es, un vector distancia es siempre positivo (lexicográficamente).

N es el número de los distintos vectores distancia de L, y D es el conjunto de estos vectores. La matriz distancia para L es una matriz de $N \times m$ cuyas filas se componen de los vectores en D, en cualquier orden. La matriz distancia también se le conoce como matriz de dependencia y se denota por $\mathcal{D}$.

Si d_ℓ es el primer elemento no cero de un vector $\mathbf{d} = (d_1, d_2, \cdots, d_m)$ a partir de la izquierda, entonces es el elemento líder de $\mathbf{d}$ y ℓ es el nivel de $\mathbf{d}$. El nivel de un vector positivo $\mathbf{d}$ es ℓ si y solo si $\mathbf{d} \prec_\ell \mathbf{0}$. Un nivel de dependencia para L es el nivel del vector distancia para L. Si ℓ es un nivel de dependencia, entonces decimos que existe

una dependencia en **L** a nivel ℓ, o que el bucle L_ℓ *lleva* la dependencia. Existen m niveles de dependencia posibles: $1, 2, \cdots, m$.

El **grafo de dependencia** de **L** es un grafo dirigido cuyos vértices son los puntos de iteración de **L**; y existe una arista dirigida desde un vértice u hacia un vértice v si, y solo si, la iteración correspondiente a v depende de la iteración correspondiente a u. Para dibujar el grafo de dependencia, colocamos todos los puntos del espacio de iteración en un plano m-dimensional (m es el número de bucles anidados) y después, dibujamos una flecha desde el punto de iteración **i** hacia el punto de iteración **j** cuando la iteración *H(j)* dependa de la iteración *H(i)*.

3.4. Transformaciones unimodulares

3.4.1. Matrices unimodulares

Una matriz unimodular tiene tres propiedades importantes. Primero, *es cuadrada*, lo cual significa que mapea un espacio de iteración n-dimensional a un espacio de iteración n-dimensional. Segundo, tiene todos sus *componentes enteros*, por lo que mapea vectores enteros a vectores enteros. Tercero, el *valor absoluto de su determinante es uno*, por lo que el volumen del espacio de iteración se conserva. Debido a estas propiedades, el producto de dos matrices unimodulares es unimodular y su inversa también es unimodular.

Una transformación compuesta puede ser sintetizada a partir del producto de una secuencia de transformaciones elementales. [16][17]

3.4.2. Legalidad de una transformación unimodular

Decimos que una transformación es legal si el programa transformado mantiene las relaciones de dependencia, el programa transformado debe de arrojar el mismo resultado que el programa original. El objetivo de la transformación es cambiar el orden de ejecución del programa manteniendo la dependencia de los datos original para de extraer el paralelismo en dicho programa.

Teorema 3.1. *La transformación unimodular de un bucle anidado* $\mathbf{L}$ *definida por una matriz* $\mathbf{U}$ *(unimodular) es válida si y solo si* $\mathbf{dU} \succ \mathbf{0}$ *para cada vector distancia* $\mathbf{d}$ *de* $\mathbf{L}$ *[18, Theorem 3.1].*

Si un bucle en el programa transformado no lleva la dependencia, éste puede ser ejecutado en paralelo.

[18]

3.4.3. Procedimiento

Esta sección tiene como objetivo plantear el procedimiento para transformar un nido de bucles anidados (perfecto) mediante matrices de transformación unimodulares, además de brindar un ejemplo para ilustrarlo.

1. Encontrar todos los vectores distancia **d**.
2. Seleccionar la matriz de transformación **U**.
3. Verificar la legalidad de la transformación, $\mathbf{dU} > \mathbf{0}$ para cada vector distancia **d**.
4. Calcular el nuevo vector de iteración $\mathbf{y} = (y_1, y_2, \cdots, y_m)$, $\mathbf{x} = \mathbf{y}(\mathbf{U}^{-1})^t$.
5. Expresar los límites del espacio de iteración original en forma de politopo, $\mathbf{xA} \leq \mathbf{b}$
6. Calcular el nuevo espacio de iteración, $\mathbf{y}(\mathbf{U}^{-1})^t\mathbf{A} \leq \mathbf{b}$.
7. Resolver el sistema de desigualdades aplicando el método de eliminación de *Fourier-Motzkin* [4], para encontrar los nuevos límites inferiores y superiores de **y**.
8. Reemplazar **x** por **y** en el cuerpo del nido de bucles (ahora **H(y)**) y en los nuevos límites de los bucles *for*.

Llamaremos **politopo destino** al espacio de iteración del nido de bucles resultante después de realizar una transformación.

Ejemplo 3.1

Transformar el siguiente nido de bucles con la matriz unimodular $\mathbf{U} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$:

```
for x1 = 2, N, 1 do
    for x2 = 2, N, 1 do
        A(x1, x2) = A(x1 − 1, x2) + A(x1, x2 − 1)
    end for
end for
```

Para resolver este ejemplo, seguiremos el procedimiento antes mencionado:

1. Encontrar todos los vectores distancia **d**.
   Podemos observar que existen dos dependencias en el cuerpo de los bucles. El primer vector distancia $\mathbf{d}_1$ se extrae de $A(x_1, x_2)$ y $A(x_1-1, x_2)$; $\mathbf{d}_1 = (x_1-(x_1-1), x_2-x_2) = (1,0)$. El segundo vector distancia $\mathbf{d}_2$ se extrae de $A(x_1, x_2)$ y $A(x_1, x_2-1)$; $\mathbf{d}_2 = (x_1 - x_1, x_2 - (x_2 - 1)) = (0,1)$.
   La matriz distancia es $\mathcal{D} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. En la figura 4.8 se representa la dependencias $\mathbf{d}_1$ y $\mathbf{d}_2$ en azul y rojo respectivamente.

2. Seleccionar la matriz de transformación **U**.
   La matriz unimodular que se utilizará es $\mathbf{U} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

3. Verificar la legalidad de la transformación, $\mathbf{dU} \succ \mathbf{0}$ para cada vector distancia **d**.

   $$\mathbf{d}_1\mathbf{U} = (1,0)\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = (1,1) \succ \mathbf{0}$$

   $$\mathbf{d}_2\mathbf{U} = (0,1)\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = (0,1) \succ \mathbf{0}$$

   Como $\mathbf{d}_1\mathbf{U}$ y $\mathbf{d}_2\mathbf{U}$ son lexicográficamente positivos la transformación es legal.

4. Calcular el nuevo vector de iteración $\mathbf{y} = (y_1, y_2, \cdots, y_m)$, mediante $\mathbf{x} = \mathbf{y}\mathbf{U}^{-1}$
   Para esto necesitamos calcular $\mathbf{U}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$,
   ahora $\mathbf{x} = \mathbf{y}\mathbf{U}^{-1} = (x_1, x_2) = (y_1, y_2)\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = (y_1, -y_1 + y_2)$.

5. Expresar los límites del espacio de iteración original en forma de politopo, $\mathbf{xA} \leq \mathbf{b}$.

   El espacio de iteración original consiste en todos los vectores enteros (x_1, x_2) tal que:

   $$\left.\begin{array}{rcl} 2 & \leq & x_1 \leq N \\ 2 & \leq & x_2 \leq N \end{array}\right\} \tag{3.4}$$

   De las desigualdades en 4.4 se obtienen los límites inferiores del vector de iteración $\mathbf{x} = (x_1, x_2)$ y pueden escribirse como

   $$\left.\begin{array}{rcl} 2 & \leq & x_1 \\ 2 & \leq & x_2 \end{array}\right\}$$

o en forma matricial como

$$(2,2) \leq (x_1, x_2)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.5}$$

De manera similar, de las desigualdades en 4.4 dan los límites inferiores del vector de iteración $\mathbf{x} = (x_1, x_2)$ y pueden escribirse como

$$\left.\begin{aligned} x_1 &\leq N \\ x_2 &\leq N \end{aligned}\right\}$$

o en forma matricial como

$$(x_1, x_2)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \leq (N, N) \tag{3.6}$$

De las ecuaciones 3.5 podemos extraer $\mathbf{p} = (2,2)$ y $\mathbf{P} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$; de 3.6 podemos extraer $\mathbf{q} = (N, N)$ y $\mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

Ahora ya estamos en condiciones de obtener el politopo fuente $\mathbf{xA} \leq \mathbf{b}$.

$$(x_1, x_2)\begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \leq (-2, -2, N, N) \tag{3.7}$$

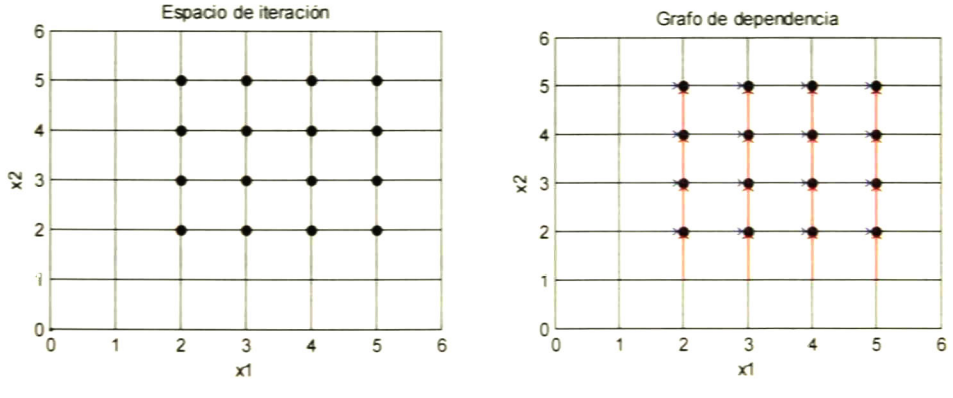


Figura 3.1: Espacio de iteración y grafo de dependencia del politopo fuente.

6. Calcular el nuevo espacio de iteración, $\mathbf{yU}^{-1}\mathbf{A} \leq \mathbf{b}$.

dado que $\mathbf{A} = \begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$, $\mathbf{b} = (-2, -2, N, N)$ y $\mathbf{U}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$

$\mathbf{yU}^{-1}\mathbf{A} \leq \mathbf{b}$ es equivalente a:

$$(y_1, y_2)\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \leq (-2, -2, N, N)$$

$$(y_1, y_2)\begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & -1 & 0 & 1 \end{pmatrix} \leq (-2, -2, N, N)$$

y finalmente

$$(-y_1, y_1 - y_2, y_1, -y_1 + y_2) \leq (-2, -2, N, N) \tag{3.8}$$

7. Resolver el sistema de desigualdades aplicando el método de eliminación de *Fourier-Motzkin* [4], para encontrar los nuevos límites inferiores y superiores de **y**.

   De 3.8 se extraen las siguientes desigualdades:

$$\begin{array}{rrrl} - & y_1 & & \leq -2 \\ & y_1 & -y_2 & \leq -2 \\ & y_1 & & \leq N \\ - & y_1 & +y_2 & \leq N \end{array}$$

   Resolviendo para y_1

$$max(2, y_2 - N) \leq y_1 \leq min(N, y_2 - 2)$$

   Resolviendo para y_2

$$4 \leq y_2 \leq 2N$$

8. Reemplazar **x** por **y** en el cuerpo del nido de bucles (ahora **H(y)**) y en los nuevos límites de los bucles *For*.

   El programa destino resultante es el siguiente:

for $y_2 = 4, 2N, 1$ **do**
  for $y_1 = max(2, y_2 - N), min(N, y_2 - 2), 1$ **do**
    $x_1 = y_1$;
    $x_2 = -y_1 + y_2$;
    $A(x_1, x_2) = A(x_1 - 1, x_2) + A(x_1, x_2 - 1)$;
  end for
end for

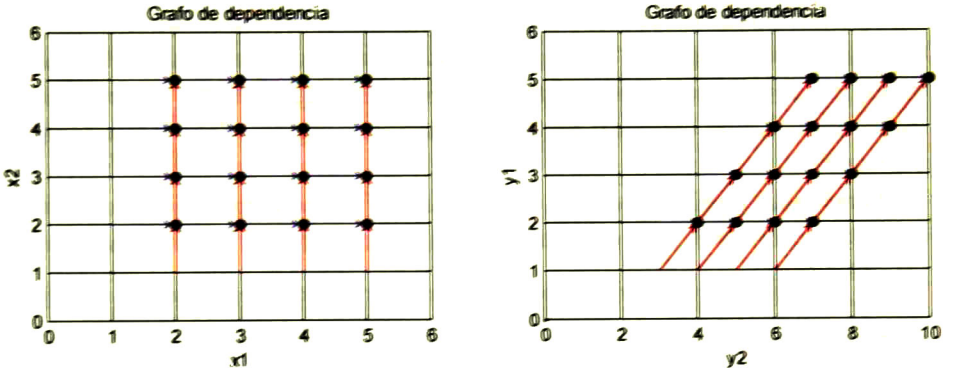


Figura 3.2: Grafo de dependencia del politopo fuente (izquierda) y del politopo destino (derecha).

Como el bucle externo lleva la dependencia, debe ser ejecutado de manera secuencial; mientras que, el bucle interno puede ser ejecutado en paralelo.

3.4.4. Transformaciones elementales

3.4.4.1. Permutación

La matriz de permutación parte de la matriz identidad y después se realizan intercambios entre sus filas. Por ejemplo la matriz $\mathbf{U} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$, en donde se intercambian las filas 1 y 3. Si se tiene un vector distancia $\mathbf{d}$ = (2,1,-1), esta transformación no será legal, ya que $\mathbf{dU}$ = (-1,1,2) es lexicográficamente negativo.

3.4.4.2. Inversión

La inversión del bucle k-ésimo es representada por la matriz identidad, pero con el k-ésimo elemento en la diagonal igual a menos uno, en vez de uno. Por ejemplo, en un nido con 3 bucles si hacemos inversión en el segundo bucle, la matriz de transformación unimodular resulta: $\mathbf{U} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

3.4.4.3. Skewing

Su matriz unimodular general cuando se tiene un espacio de iteración bidimensional es $\mathbf{U} = \begin{pmatrix} 1 & f \\ 0 & 1 \end{pmatrix}$. Donde f representa el valor de *skew*. Esta transformación siempre es legal ya que no se modifica la dependencia a nivel uno; si se tiene un vector distancia $\mathbf{d} = (d_1, d_2)$, después de aplicar "skewing" el vector distancia se convertirá en $\mathbf{dU} = (d_1, f * d_1 + d_2)$. d_1 permanece sin cambio, por lo que no corre riesgo de tomar valores negativos, lo que ocasionaría el vector $\mathbf{dU}$ sea lexicográficamente negativo y que la transformación sea ilegal.

Para brindar un ejemplo de esta transformación, retomaremos el ejemplo desarrollado en la sección 3.4.3, en la que tomaremos el programa fuente y aplicaremos la matriz $\mathbf{U} = \begin{pmatrix} 1 & f \\ 0 & 1 \end{pmatrix}$. Después de aplicar todo el procedimiento de transformación el programa destino es el siguiente:

for $y_2 = 2 + 2f, N + fN, 1$ **do**
  for $y_1 = max(2, \frac{y_2 - N}{f}), min(N, \frac{y_2 - 2}{f}), 1$ **do**
    $x_1 = y_1;$
    $x_2 = -fy_1 + y_2;$
    $A(x_1, x_2) = A(x_1 - 1, x_2) + A(x_1, x_2 - 1);$
  end for
end for

La figura 3.2 muestra el grafo de dependencia original y el transformado con un valor "*skew*" f de 1; en lado izquierdo de la figura 3.3 se muestra el grafo de dependencia para f igual a 2 y 3. Como se puede observar en estas figuras, el mejor valor de f es uno, ya que requiere menos iteraciones en el índice t, además se pueden realizar mas puntos de iteración en paralelo, en el eje p. Con un valor f mayor de 3 ya se perdería la posibilidad de ejecutar puntos de iteración en paralelo, ejecutándose un punto en cada tiempo, incluso se podría tener iteraciones vacías, por lo que se debe seleccionar el valor f tal que se reduzca el número de iteraciones en el bucle externo y se puedan ejecutar la mayor cantidad posible de puntos de iteración en paralelo.

3.4.4.4. Desplazamiento

Esta transformación traslada el grafo de dependencia sin modificar su volumen, forma y sus dependencias. Esto es útil cuando se requiere que los bucles empiecen a iterar desde algún valor definido. Para ilustrar este efecto, seguiremos con el resultado obtenido después de aplicar "*skewing*". En rojo se marca el cambio para hacer que índice t empiece a iterar con valor de uno, mientras, que de manera similar, se marca en verde para el índice p. En la figura 3.3 se muestra el efecto del cambio antes mencionado en los grafos de dependencia para diferentes valores de f.

for $t = 2f + 2 - (2f + 1), N + fN - (2f + 1), 1$ **do**
  for $p = max(2 - 1, \frac{t-N+(2f+1)}{f} - 1), min(N - 1, \frac{t-2+(2f+1)}{f} - 1), 1$ **do**
    $x_1 = p + 1;$
    $x_2 = -f(p + 1) + t + (2f + 1);$
    $A(x_1, x_2) = A(x_1 - 1, x_2) + A(x_1, x_2 - 1);$
  end for
end for

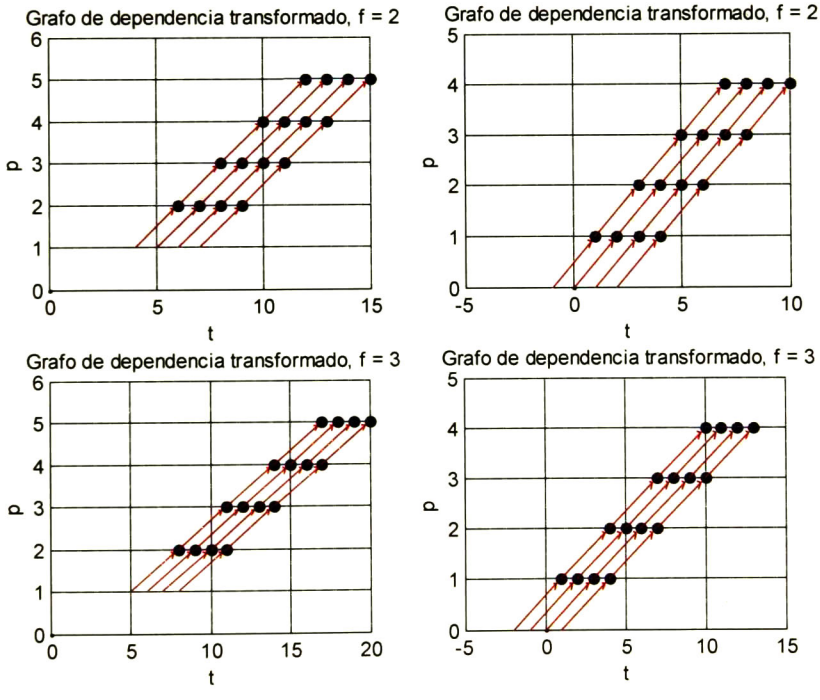


Figura 3.3: Desplazamiento de los grafos de dependencia para que el primer punto de iteración sea (t,p) = (1,1).

3.5. Tipos de paralelismo

3.5.1. Introducción

Paralelismo interno y **paralelismo externo** se refiere a la posibilidad de ejecutar de manera paralela los bucles mas externos o internos, respectivamente, en un nido de bucles. En la figura 3.4 se muestra un ejemplo en donde se tienen dos bucles y uno de ellos se puede ejecutar en paralelo, indicado por un *forall*, el índice *t* representa un determinado tiempo y el índice *p* representa un determinado número de elementos de procesamiento.

Como podemos observar en la parte izquierda, tenemos que el bucle externo es un *for* por lo que las iteraciones t son secuenciales, mientras que, el bucle interno es un *forall* por lo que las iteraciones en p se pueden ejecutar de manera paralela. En cada *t* se pueden ejecutar los 5 puntos de iteración que se encuentran encerrados en el recuadro azul (bajo el supuesto que no existen dependencias en *p*). Por tanto, si se tuvieran 5 elementos de procesamiento, se podrían ejecutar los 5 puntos de iteración en la dirección de *p* en cada tiempo. Los 5 elementos de procesamiento deben esperar a que todos terminen para que la siguiente iteración de *t* pueda comenzar, de allí la razón por la que a este tipo de paralelismo también se le conoce como *paralelismo síncrono*, ó también *„paralelismo vertical.*

Ahora describiremos el paralelismo externo, de igual manera si se tienen 5 elementos de procesamiento, cada uno de ellos puede ejecutar el bucle interno (de manera secuencial) para las 5 iteraciones en *t*, ya que no se necesita sincronía entre los elementos de procesamiento, este tipo de paralelismo también se le conoce como *paralelismo asíncrono*, o también, *paralelismo horizontal.*

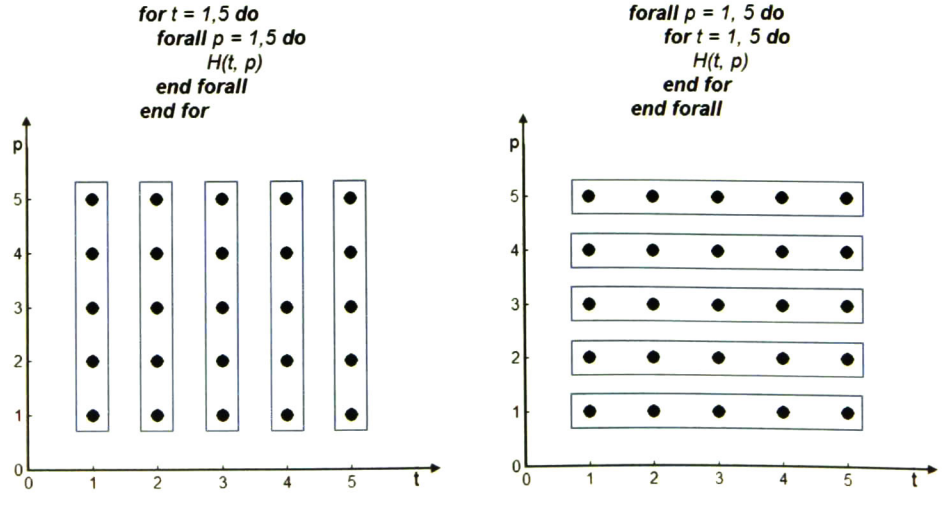


Figura 3.4: Tipos de paralelismo. Izquierda: paralelismo interno, derecha: paralelismo externo.

3.5.2. Paralelización de bucles internos

Tiene el objetivo de encontrar una matriz unimodular U, tal que $\mathbf{dU} >_1 \mathbf{0}$ ($\mathbf{d} \in D$).

Teorema 3.2. *Dado cualquier nido L de m bucles. Existe una transformación unimodular válida $L \mapsto L_U$ tal que, los bucles $L_{U_1}, L_{U_2}, \cdots, L_{U_m}$ de L_U se pueden ejecutar en paralelo. [18, Theorem 3.7]*

Para lograr este tipo de paralelismo se presenta otra alternativa para encontrar una matriz unimodular a partir del conjunto de vectores distancia en vez de la composición de matrices de transformación elementales. Primero aplicamos el algoritmo 3.3 al conjunto de vectores distancia en *D*.

Algoritmo 3.3 Vector de calendarización

Entrada: Conjunto D finito no vacío de vectores positivos $\mathbf{d} = (d_1, d_2, \cdots, d_m)$
Salida: Vector $\mathbf{u} = (u_1, u_2, \cdots, u_m)$ con elementos no negativos, tal que el mayor común divisor de los elementos de **u** es igual a 1 y $d_1u_1 + d_2u_2 + \cdots + d_mu_m \geq 1$

```
for r = 1, m, 1 do
    D_r ← {d ∈ D : d >_r 0}
end for
for r = m, 1, -1 do
    if D_r = ∅ then
        u_r ← 0
    else
        u_r ← ⌈máx_{d∈D_r}{(1 − d_{r+1}u_{r+1} − d_{r+2}u_{r+2} − ··· − d_m u_m)/d_r}⌉^+
    end if
end for
```

Una vez obtenido el vector de calendarización **u**, el objetivo ahora es encontrar una matriz unimodular U de *m x m* tal que:

- La primera columna de U sea $(u_1, u_1, \cdots, u_m)$.
- La k-ésima fila de U sea $(1, 0, 0, \cdots, 0)$, donde k corresponde al elemento u_k, el cual es el primer elemento no cero en la secuencia: $u_m, u_{m-1}, \cdots, u_1$ (nótese que $u_k = 1$).
- La matriz obtenida eliminando la columna 1 y la fila k de U es la matriz identidad I de dimensión *(m-1) x (m-1)*.

Para analizar un ejemplo de este método se recomienda revisar la página 1073 de [19].

3.5.3. Paralelización de bucles externos

En esta sección se estudiará la existencia de una transformación válida $\mathbf{L} \mapsto \mathbf{L_U}$ tal que uno o más bucles externos de $\mathbf{L_U}$ se pueden ejecutar en paralelo. La matriz distancia de $\mathbf{L_U}$ es $\mathcal{D}_\mathbf{U}$ ($\mathcal{D}_\mathbf{U} = \mathcal{D}\mathbf{U}$). La primera columna de $\mathcal{D}_\mathbf{U}$ no puede tener un elemento negativo, ya que todas las filas de esta matriz debe ser positiva (lexicográficamente). Si esta columna tiene un elemento positivo, entonces el bucle más externo de L_{U_1} de $\mathbf{L_U}$ lleva la dependencia, y por lo tanto no puede ejecutarse en paralelo. L_{U_1} puede ejecutarse en paralelo si y solo si la primera columna de $\mathcal{D}_\mathbf{U}$ es igual al vector cero.

El problema es encontrar una matriz unimodular **U**, tal que, todas las filas del producto $\mathcal{D}\mathbf{U}$ sean positivas y una o más columnas del lado izquierdo de $\mathcal{D}\mathbf{U}$ sean vectores ceros. Un ejemplo de estas matrices es la siguiente:

$$\begin{pmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 10 & 1 \\ 0 & 0 & 5 & 1 \\ 0 & 0 & 6 & 2 \end{pmatrix}$$

El rango de una matriz es igual al número de sus columnas linealmente independientes (que es el mismo número de sus filas linealmente independientes). La matriz $\mathcal{D}_\mathbf{U}$ tiene m columnas, y por lo tanto su rango no es mayor que m. El rango es exactamente m si todas las columnas son linealmente independientes. Tener al menos una columna cero implica que se tiene estrictamente un rango menor que m.

Teorema 3.3. *Considerando un nido L de m bucles. Sea $\mathcal{D}$ la matriz de dependencia de L y ρ el rango de $\mathcal{D}$. Si $\rho < m$, entonces existe una transformación válida $\mathbf{L} \mapsto \mathbf{L_U}$ tal que, los $(m-\rho)$ bucles externos de $\mathbf{L_U}$ se pueden ejecutar en paralelo. [18, Theorem 3.8]*

No se puede obtener más de $(m-\rho)$ bucles externos paralelos por una transformación unimodular. A continuación se presenta la secuencia de pasos para que a partir de la matriz $\mathcal{D}$, de $N \times m$, se obtenga la matriz transformación, tal que $(m-\rho)$ bucles externos puedan ejecutarse en paralelo:

1. Calcular la transpuesta $\mathcal{D}'$ de $\mathcal{D}$ (de tamaño $m \times N$).
2. Por el algoritmo 3.1, encontrar una matriz unimodular **V** de $m \times m$ y una matriz escalonada **S** de $m \times N$ tal que $\mathbf{V}\mathcal{D}' = \mathbf{S}$.
3. Calcular ρ y n, donde ρ es el rango de **S** y n es $m - \rho$.
4. Por el algoritmo 3.3, encontrar un vector **u** de m elementos, tal que $\mathbf{du} > 0$ para cada fila **d** de $\mathcal{D}$.
5. Construir una matriz **A** entera de $m \times (n + 1)$ definiendo sus columnas $\mathbf{a}^k$ como sigue:

- $\mathbf{a}^k \leftarrow \mathbf{v}_{\rho+k}$ para $k = \{1, \cdots, n\}$, donde $\mathbf{v}_{\rho+k}$ es el (ρ+k)-ésima fila de **V**.
- $\mathbf{a}^{n+1} \leftarrow \mathbf{u}$

Entonces tenemos $\mathbf{da}^1 = \mathbf{da}^2 = \cdots = \mathbf{da}^n = 0$ y $\mathbf{da}^{n+1} > 0$, para cada fila **d** de $\mathcal{D}$.

6. Por el algoritmo 3.2, encontrar una matriz unimodular **U** de m x m y una matriz escalonada $\mathbf{T} = (t_{rk})$ de m x $(n + 1)$ tal que $\mathbf{A} = \mathbf{UT}$.
   Si $t_{(n+1)(n+1)} < 0$, entonces multiplicar por -1 la fila $(n + 1)$ de **T** y la columna $(n + 1)$ de **U**.
   Así tenemos la matriz **U** tiene todas las propiedades deseadas. **U** satisface $\mathbf{du}^1 = \mathbf{du}^2 = \cdots = \mathbf{du}^n = 0$ y $\mathbf{du}^{n+1} > 0$, para cada fila **d** de $\mathcal{D}$.

Para analizar un ejemplo de este método revisar la página 1074 de [19].

3.6. Resumen del capítulo

El capítulo presenta los fundamentos básicos para extraer el paralelismo de bucles anidados mediante transformaciones unimodulares. Al inicio se presentan conceptos matemáticos necesarios en este capítulo y en el cuatro. Se establecen relaciones de orden para vectores y reducciones escalonadas de una matriz con elementos enteros. Posteriormente se expone como modelar matemáticamente un nido de bucles con las dependencias de datos que contenga.

Las transformaciones de bucles parten de extraer las dependencias existentes entre las iteraciones, una vez recolectado el conjunto de dependencias, se procede a buscar una matriz de transformación unimodular que cambie el orden de ejecución de las iteraciones, respetando las dependencias de los datos.

Se presentan transformaciones unimodulares elementales, con las que, realizando el producto de ellas se puede obtener una matriz de transformación que cambie el orden de ejecución en la forma deseada sin necesidad de realizar dos o más transformaciones sucesivas.

Finalmente se presenta el método del hiperplano para obtener paralelismo de bucles interno, el cual, además de las transformaciones elementales, ayudará a obtener una matriz unimodular a partir de la matriz de dependencia de datos, tal que, *(m-1)* bucles puedan ejecutarse en paralelo (de un nido de m bucles). Además, se presenta un método para obtener paralelismo de bucles externos, en donde, no es necesaria una sincronía entre los procesadores que ejecutan las iteraciones, a diferencia del paralelismo interno en donde si se requiere. Para elegir el tipo de paralelismo, se debe analizar las características del hardware en donde se ejecutarán los algoritmos.

Capítulo 4

Implementación de algoritmos en CUDA

4.1. Suma de vectores

La suma de dos vectores se puede expresar como $c_i = a_i + b_i$, para $0 \leq i < N$, donde N es número de elementos de los vectores. La misma se puede realizar con un solo bucle *for*, donde en cada iteración se va obteniendo c_i, como sigue:

for $i = 0, N-1, 1$ **do**
    $c(i) = a(i) + b(i)$
end for

Como se puede observar, los arreglos a y b son de sólo lectura, mientras que c es de sólo escritura. Se lee o escribe una localidad de memoria diferente de a , b y c en cada iteración de i, por lo que no tienen dependencia de datos entre las iteraciones al no cumplirse las condiciones mencionadas en la sección 3.3 de dependencia de datos. Debido a que no existen dependencias, todas las iteraciones del bucle pueden ser ejecutadas en paralelo, sin necesidad de realizar alguna transformación unimodular. A continuación presentaremos una versión secuencial que se ejecuta en el *device* por un *thread* que posteriormente se comparará a la versión paralela que se ejecuta en N *threads*. Se mostrará sólo el dimensionamiento del *grid* y la implementación de *kernel* para las dos versiones de este algoritmo, el código completo se proporciona en el CD de la tesis.

4.1.1. Versión secuencial

El dimensionamiento del *grid* para la versión secuencial en CUDA del algoritmo suma de vectores es el siguiente:

```
// Dimensionamiento del grid para la función kernel
// Dimensionamiento de blocks
dim3 gridDim(1, 1, 1); // Un block
// Dimensionamiento de Threads
dim3 blockDim(1, 1, 1); // 1 Thread por block
// Lanzamiento de kernel
VectorAddKernel<<<gridDim, blockDim>>>(dev_a , dev_b, dev_c, dataSize);
```

Como la idea es lanzar solo un *thread* configuramos el *grid* para que tenga un *block* con un *thread*. Lanzamos el *kernel VectorAddKernel* con los parámetros: *dev_a, dev_b* y *dev_c* que corresponden a los arreglos *a, b* y *c*, respectivamente, *dataSize* corresponde al valor de *N*. Los arreglos *dev_a, dev_b* y *dev_c* se crearon mediante la instrucción *cudaMalloc* por lo tanto se encuentran almacenados en la memoria global del *device*.

El código del kernel es el siguiente:

```
// Suma de dos vectores en CUDA 1 thread
__global__ void VectorAddKernel(float *a, float *b, float *c, int dataSize)
{
    for (int tid = 0; tid < dataSize; tid++){
        c[tid] = a[tid] + b[tid];
    }
}
```

Esta versión no aprovecha las capacidades de cómputo paralelo del GPU, se realizó para tener como referencia el tiempo de duración de la ejecución del algoritmo de manera secuencial en un *thread* para después sacar la relación con la versión paralela, la cual se ejecuta en *N threads*.

4.1.2. Versión Paralela

El primer reto en implementar un algoritmo con bucles anidados en CUDA es dimensionar el *grid* para que cada iteración *for* se ejecute en un *thread* aprovechándose lo más posible los multiprocesadores de nuestros GPUs. Como buscamos aprovechar al máximo la cantidad de *threads* disponibles por multiprocesador, el dimensionamiento del *grid* no es tan intuitivo como la versión anterior. Para esto debemos tomar en cuenta las características de nuestro GPU, las cuales se encuentran el la tabla 4.1, en donde el número máximo de *threads* por *block* es de 1024. Sin embargo, se decidió no utilizar ese número ya que 512 *threads* quedarían sin utilizarse en los MP, debido a que el número máximo de *threads* por MP es de 1536. Entonces, se decidió que el número máximo de *threads* por *block* fuera de 768, para que se utilizaran los 1536 *threads* por MP, al

poderse ejecutar dos *blocks* de 768 *threads* en cada MP y así tener una *occupancy* del 100 %.

Para determinar la cantidad máxima de *blocks* en el *grid* ya fue directamente los 35535 por cada dimensión (x, y, z). La idea del siguiente dimensionamiento es utilizar las 3 dimensiones de los *blocks* para incrementar las posibilidades de acceso lineal a los arreglos *a*, *b* y *c* de la memoria global:

```
// Dimensionamiento del grid para la función kernel
dim3 gridDim(1, 1, 1); dim3 blockDim(1, 1, 1);
int maxNumThreads = 768;
dim3 maxNumBlocks(35535, 35535, 35535);
if(dataSize <= maxNumThreads)
{
    blockDim.x = dataSize;
}else{
    blockDim.x = maxNumThreads;
    if( ceil( dataSize / blockDim.x ) <= maxNumBlocks.x )
    {
        gridDim.x = ceil( dataSize / blockDim.x );
    }else {
        gridDim.x = maxNumBlocks.x;
        if( ceil( dataSize / (blockDim.x * gridDim.x)) <= maxNumBlocks.y )
        {
            gridDim.y = ceil( dataSize / (blockDim.x * gridDim.x) );
        }else {
            gridDim.y = maxNumBlocks.y;
            if( ceil( dataSize / (blockDim.x * gridDim.x * ···
            blockDim.y * gridDim.y) ) <= maxNumBlocks.z )
            {
                gridDim.z = ceil( dataSize / (blockDim.x * gridDim.x * ···
                blockDim.y * gridDim.y) );
            }else {
                cout<<endl<< " El número de datos excede la ···
                capacidad del grid"<<endl;
            }
        }
    }
}
// Lanzamiento de kernel
VectorAddKernel<<<gridDim, blockDim>>>(dev_a , dev_b, dev_c, dataSize);
```

El número de *blocks* y de *threads* que se emplearán en el *grid* se calculan en base a *dataSize*. Si sólo se utilizara un *block*, entonces el tamaño máximo de *dataSize* que se podría calcular sería 768, al combinar el número de *threads* utilizando hasta los 35535 *blocks* en el eje *'x'*, se podía manejar un *dataSize* de hasta 27,290,880 (blockDim.x * maxNumThreads = 35535 * 768 = $27 * 10^6$). Si además utilizamos los 35535 *blocks* disponibles en el eje *'y'* el tamaño de *dataSize* que se podría manejar sería de 969,781,420,800 (blockDim.y * blockDim.x * maxNumThreads = 35535*35535*768 = $969 * 10^9$), con lo cual ya estaríamos trabajando con memorias de centenas de gigabytes. El tamaño máximo de *dataSize* con esta manera de configurar el *grid* se alcanza al utilizar las tres dimensiones de los *block*, con lo cual *dataSize* se incrementa hasta $34{,}446 * 10^{15}$ (blockDim.z * blockDim.y * blockDim.x * maxNumThreads = 35535 * 35535 * 35535 * 768), con lo cual, se supera por mucho los 5.3 GB de memoria global de nuestro GPU.

El *kernel* para la suma de vectores es el siguiente:

```
// Suma de dos vectores en CUDA en paralelo
__global__ void VectorAddKernel(float *a, float *b, float *c, int dataSize)
{
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    int z = blockDim.z * blockIdx.z + threadIdx.z;
    int tid = x + y * blockDim.x * gridDim.x + ···
    z * blockDim.x * gridDim.x * blockDim.y * gridDim.y;
    if ( tid < dataSize ){
        c[tid] = a[tid] + b[tid];
    }
}
```

En el *kernel* la varible *'tid'* es la que permitirá el acceso a las diferentes localidades de memoria de los arreglos *a*, *b* y *c*. Es recomendable colocar una condicional para verificar que sólo se realice el cálculo en nuestro *kernel* por los *threads* que sean necesarios dependiendo nuestro tamaño de problema, para evitar que los *threads* sobrantes quieran utilizar memoria que no se ha reservado; i.e en este caso, que se realice la suma cuando *'tid'* sea menor que *dataSize*.

4.1.3. Análisis de resultados

Se realizó también una versión secuencial en lenguaje *C*, la cual se comparará con las versiones secuencial CUDA un thread y CUDA paralelo. Se observará el comportamiento de tiempo que presentan las versiones, se medirá con el *speedup* la relación en tiempo de la implementación secuencial y la paralela, y finalmente, se analizará el costo en memoria de las versiones.

Tabla 4.1: Características *device*: TESLA C2075.

Característica	Valor
Compute capability	2.0
Número de Multiprocesadores	14
Número de CUDA Cores por MP	32 a 1.15 GHz
Número total de CUDA Cores	448
Número de *threads* por *warp*	32
Máximo número de *threads* por MP	1536
Número máximo de *blocks* por MP	8
Máximo número de *threads* por *block*	1024
Tamaño máximo para cada dimensión (x,y,z) de un *block*	1024, 1024, 64
Tamaño máximo para cada dimensión (x,y,z) de un *grid*	65535, 65535, 65535
Cantidad total de memoria global	5376 MBytes
Número total de registros disponibles por *block*	32768
Cantidad total de memoria compartida por *block*	49152 bytes

Tabla 4.2: Características *host*.

Característica	Valor
Procesador	Intel(R) Core(TM) i7-2600K
Número de Cores	8 a 3.4 GHz
Memoria RAM	12 GBytes

El factor de ***speedup*** (S_p) de un cómputo paralelo usando p procesadores se define por la razón:

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

donde, T_1 es el tiempo obtenido del cómputo en un procesador, y T_p es el tiempo obtenido del mismo cómputo en p procesadores. En otras palabras, S_p es la relación entre el tiempo de procesamiento secuencial y el tiempo de procesamiento paralelo. Normalmente, el algoritmo secuencial es el mejor algoritmo conocido para un determinado problema computacional. El *speedup* muestra la ganancia de velocidad de un cómputo paralelo; cuanto mayor sea S_p, es mejor.

$$1 \leq S_p \leq p \tag{4.2}$$

El factor de *speedup* es normalmente menor que el número de procesadores debido al tiempo perdido en sincronización, tiempo de comunicación y los encabezados (*overheads*) requeridos por un cálculo paralelo [20].

La figura 4.1 muestra el comportamiento de tiempo (en segundos) para las tres versiones antes mencionadas. Se tomaron 140 muestras, cada una con el promedio de eje-

cutar 500 iteraciones para cada tamaño muestral. Los tamaños de los arreglos van desde 1536 hasta 1536*14*10, con incrementos de 1536, esto se determinó así para observar el comportamiento al utilizar los 1536 *threads* por los 14 MP de nuestro GPU y sobrepasar 10 veces la cantidad de threads máximos que se pueden ejecutar en paralelo en el GPU (1536*14 = 21504 threads). Se utilizó una escala logarítmica para poder observar la tendencia de las tres versiones, ya que el tiempo de la versión CUDA 1 *thread* está muy por encima de las otras dos. Observamos que la versión CUDA 1 *thread* presenta un comportamiento lineal y creciente al incrementar el tamaño de los arreglos *a*, *b* y *c*.

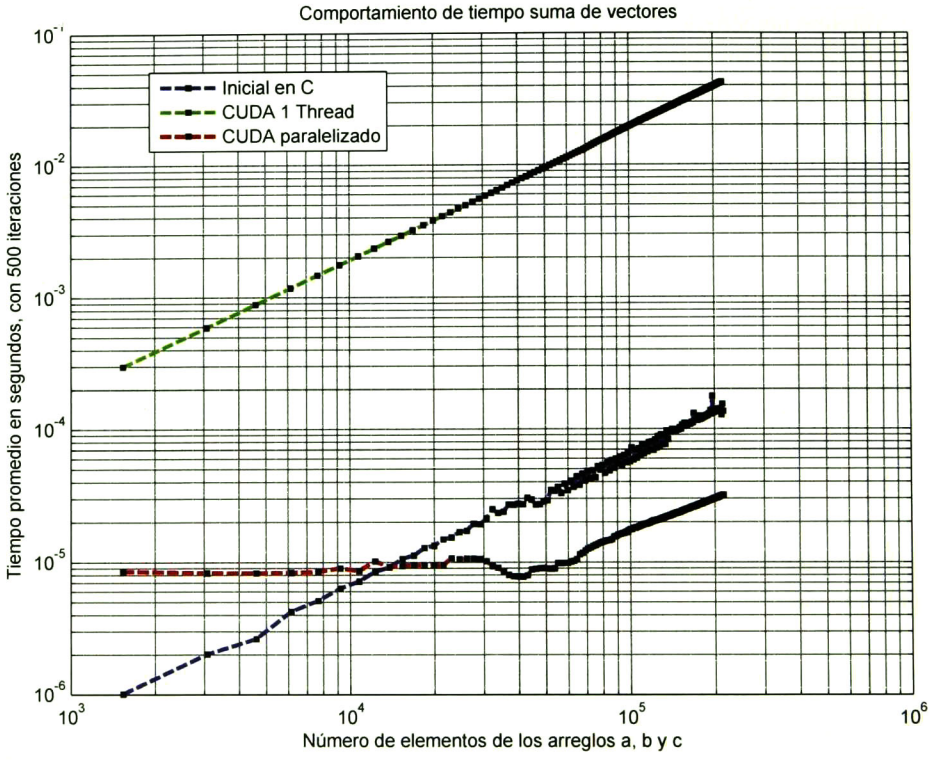


Figura 4.1: Comportamiento de tiempo del algoritmo suma de vectores.

La versión secuencial en *C*, presenta un crecimiento oscilatorio para tamaños mayores de los arreglos, esto es esperado ya que la ejecución del *thread* en el CPU que realiza el cálculo está bajo el control del sistema operativo, en donde se ejecutan múltiples tareas y se reparte el tiempo del procesdor del CPU entre ellas. Debido a lo anterior, se cambió la prioridad a "alta" al proceso donde ejecutamos nuestros códigos y cerramos la mayor cantidad de programas posibles.

La versión paralela en CUDA presenta prácticamente la misma duración de tiempo hasta 20,000, lo cual se esperaba, ya que el GPU puede ejecutar (teóricamente) hasta 21,504 *threads* de manera concurrente. Una vez que se sobrepasa este límite, ya se

verá afectado el tiempo de ejecución por la rapidez del calendarizador de los *warps*. A partir de los 60,000 (aproximadamente) ya presenta un comportamiento creciente casi lineal.

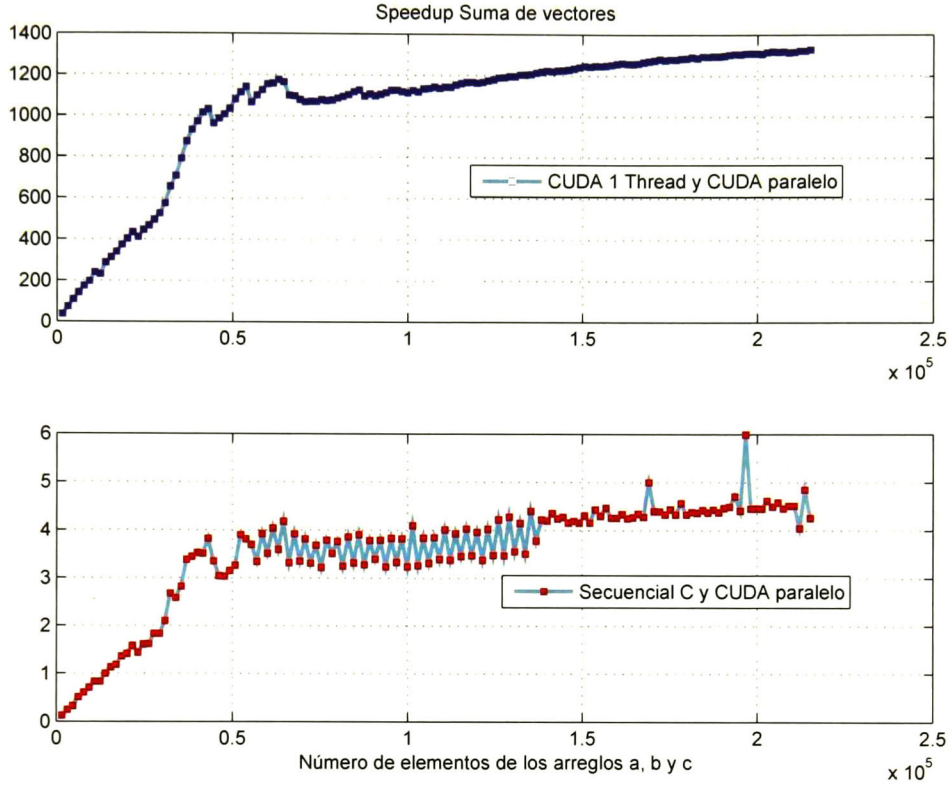


Figura 4.2: *Speedup* de suma de vectores.

Observamos que la versión secuencial en *C* supera a la versión paralela en CUDA hasta llegar a los 13824 (de tamaño de matriz), para tamaños superiores la versión paralela en CUDA es más rápida. Esto concuerda con los resultados obtenidos al comparar las versiones secuenciales en *C* y en CUDA, en donde la versión en CUDA es más lenta que la versión en C. Como se puede ver en las tablas 4.1 y 4.2, los *cores* del CPU son más rápidos que los del GPU. Sin embargo, al utilizar un gran número de *threads* en paralelo para realizar los cálculos, se pueden obtener mejoras en tiempo significativas a pesar de que los cores del CPU sean más rápidos.

La figura 4.2 muestra en la parte superior, el *speedup* entre la versión secuencial en CUDA y la versión paralela en CUDA, se puede observar un incremento mayor en el *speedup* para tamaños inferiores a 60,000 ya que se utilizando cada vez más los *threads* que puede ejecutar nuestro GPU, para tamaños superiores el incremento es menor ya que todos los *threads* disponibles en el GPU (21504) están realizando más de una operación de suma. En la parte inferior muestra el *speedup* entre la versión secuencial en *C* y la

versión paralela en CUDA, un *speedup* menor a uno, indica que la versión secuencial en *C* es más rápida, de lo contrario, la versión paralela será S_p veces más rápida que la secuencial en *C*. De nuevo notamos que se presentan irregularidades u oscilaciones en el *speedup* lo cual se ocasiona por el comportamiento irregular de la ejecución en el *host* por los factores antes mencionados.

También se sacó el comportamiento de tiempo (en segundos, tiempo promedio en 100 iteraciones) para tamaños de los arreglos, en el orden de millones, desde un millón hasta diez millones, el cual se muestra en la figura 4.3. Se conservó la tendencia que se tenía en la gráfica anterior (figura 4.2) en donde la versión más lenta era CUDA 1 *thread*, seguida por la versión secuencial en C y la más rápida es la paralela en CUDA. En la parte superior se tiene una escala logarítmica, mientras que en la parte inferior la escala es lineal. De manera similar, se obtuvo el *speedup* (figura 4.4) para estos tamaños de arreglos. Se observa que el *speedup* es cercano entre los diferentes tamaños de los arreglos, con lo cual podemos afirmar que estamos cerca, en la práctica, del *speedup* máximo.

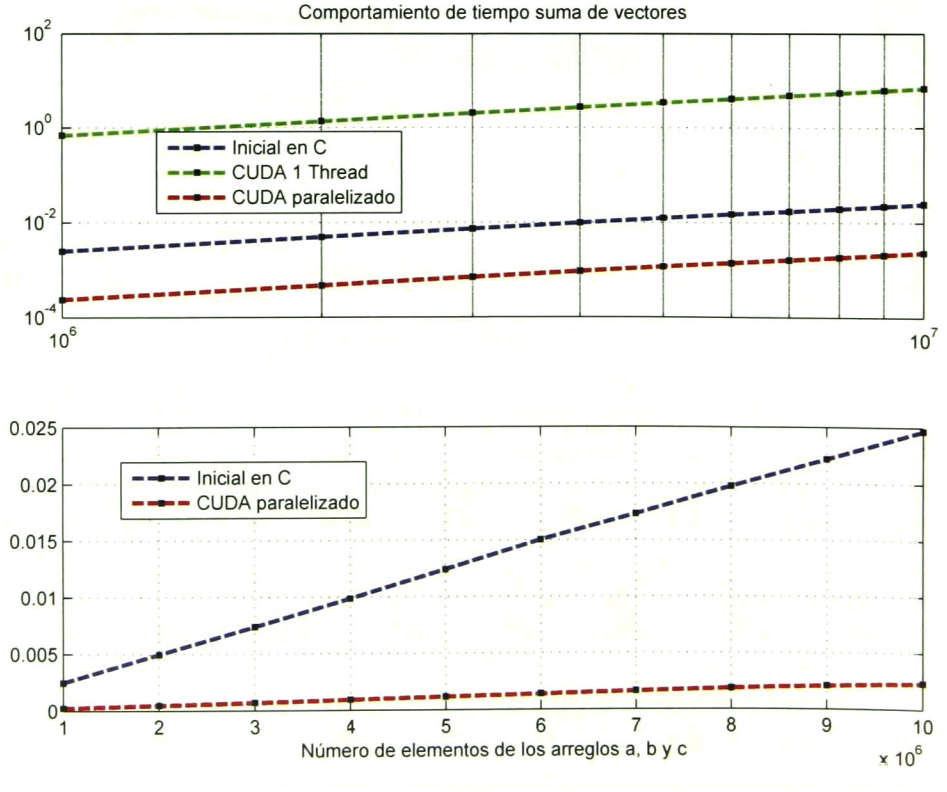


Figura 4.3: Comportamiento de tiempo (en segundos) del algoritmo suma de vectores para millones de elementos.

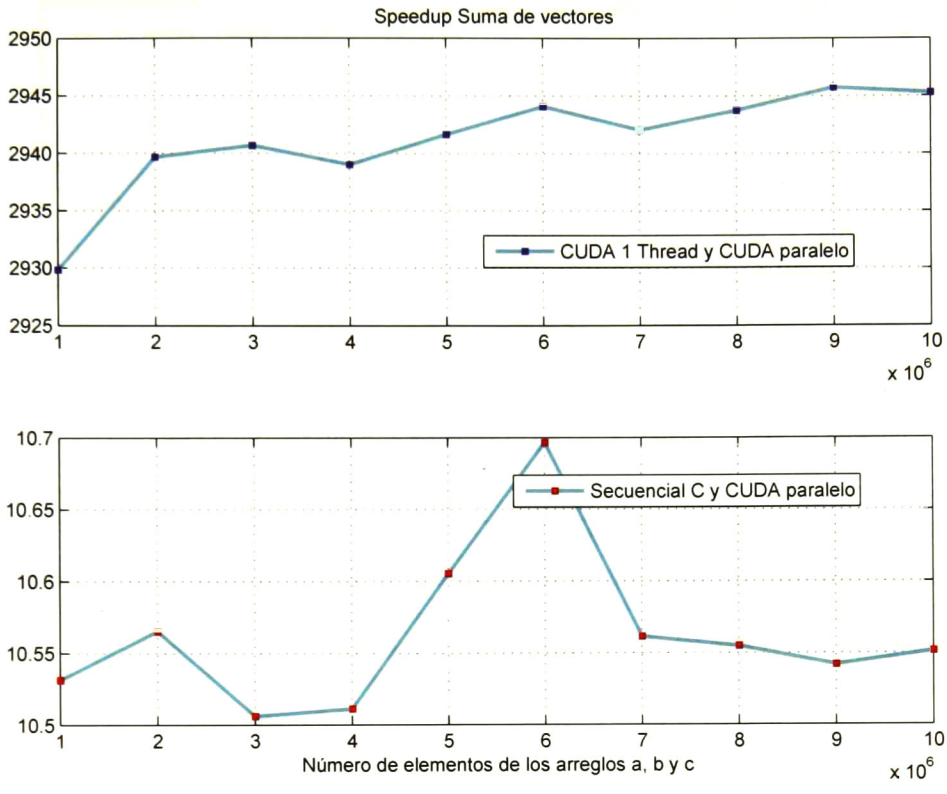


Figura 4.4: *Speedup* de Suma de vectores para millones de elementos.

A continuación presentaremos el gasto de memoria de las versiones. Las tres versiones necesitan generar los arreglos *a*, *b* y *c*, por lo tanto, el gasto de memoria se puede escribir como 3**dataSize**4 bytes, donde dataSize es el número de elementos de los arreglos, se multiplica por 4, ya que es el número de bytes necesarios por una variable de punto flotante. Las versiones de CUDA necesitan también reservar ese espacio en memoria: 3**dataSize**4 bytes. Se puede evitar utilizar el vector *c*, guardando el resultado en el vector *a* o *b*, con lo cual la memoria necesaria en el *device* se reduce a 2**dataSize**4 bytes.

4.2. Transpuesta de una matriz

4.2.1. Transpuesta de una matriz

La definición de la transpuesta de una matriz es la siguiente: Sea $\mathbf{A} = (a_{ij})$ una matriz de $m \times n$. Entonces la transpuesta de **A**, que se escribe $\mathbf{A}^t$, es la matriz de $n \times m$ obtenida al intercambiar los renglones por las columnas de **A**. De manera breve, se puede escribir $\mathbf{A}^t = (a_{ji})$. Simplemente se coloca el renglón i de **A** como la columna i de $\mathbf{A}^t$ y la columna j de **A** como renglón j de $\mathbf{A}^t$.

La transpuesta de una matriz se puede calcular de manera computacional mediante dos bucles *for*, el bucle externo barre las filas, mientras que el bucle interno barre las columnas de nuestra matriz **A**, como lo describe el siguiente algoritmo:

Algoritmo 4.1 Matriz transpuesta

Entrada: Matriz **A** de $m \times n$, donde m es el número de filas y n es el número de columnas.

Salida: Matriz **B** de $n \times m$, tal que, $\mathbf{B} = \mathbf{A}^t$.

```
for i = 0, M - 1, 1 do
    for j = 0, N - 1, 1 do
        B(j, i) = A(i, j)
    end for
end for
```

La transpuesta de la matriz **A** se guarda en la matriz **B**. Como se puede observar, cada localidad de la matriz **A** es de sólo lectura, mientras que cada localidad de la matriz *B* es de sólo escritura, por lo que no existe dependencia de datos en este algoritmo por lo que ambos bucles pueden ser ejecutados en paralelo. A continuación presentaremos diferentes implementaciones en CUDA para ejemplificar como se puede lograr el paralelismo interno y externo. Además, se presentará la versión paralela en la que ambos bucles se ejecutan con un solo *kernel*, sin necesidad de utilizar un bucle, y al final de esta sección se realizará una comparativa entre las versiones.

4.2.2. Versión secuencial

El dimensionamiento del *grid* para la versión secuencial en CUDA del algoritmo matriz transpuesta es el siguiente:

```
// Dimensionamiento del grid para la función kernel
// Dimensionamiento de blocks
dim3 gridDim; gridDim.x = 1; gridDim.y = 1; gridDim.z = 1; // Un block
// Dimensionamiento de Threads, 1 Thread por block
dim3 blockDim; blockDim.x = 1; blockDim.y = 1; blockDim.z = 1;
// Lanzamiento de kernel
MatrixTransposeKernel<<<gridDim, blockDim>>>(M , N);
```

Como la idea es lanzar solo un *thread*, configuramos el *grid* para que tenga un *block* con un *thread*. Lanzamos el *kernel MatrixTransposeKernel* con los parámetros *rows* y *columns*, los cuales indican la cantidad de filas y columnas de **A**, respectivamente.

El código del *kernel* es el siguiente:

```
// MatrizTransposeCUDA 1 thread
__global__ void MatrixTransposeKernel(int rows, int columns)
{
    float data = 0;
    for (int i = 0; i < rows; i++){
        for (int j = 0; j < columns; j++){
            // B(j , i) = A(i , j)
            surf2Dread(&data, matrixA_surfRef, i*4, j);
            surf2Dwrite(data, matrixB_surfRef, j*4, i);
        }
    }
}
```

Las matrices **A** y **B** se ligaron a las *surfice* de dos dimensiones matrixA_sufRef y matrixB_sufRef, respectivamente. En la variable *data* se escribe el valor del elemento a_{ij} de **A** que posteriormente se va a guardar en el elemento b_{ij} de **B**.

4.2.3. Versión con paralelismo externo

Como se vio en la sección 3.5, el paralelismo externo se caracteriza por que los bucles más externos se ejecuten en paralelo. Nuestro algoritmo base es el 4.1, en donde el bucle externo recorre las filas de **A**, este bucle es el que vamos a ejecutar en paralelo en esta versión. Para esto, debemos crear la cantidad de *threads* y *blocks* necesarios para ejecutar las M filas en paralelo, esto se realiza como sigue:

```
// Dimensionamiento del grid para la función kernel
dim3 gridDim; gridDim.x = 1; gridDim.z = 1;
dim3 blockDim; blockDim.x = 1; blockDim.z = 1;
if (M < 768){
    blockDim.y = M; gridDim.y = 1;
}else{
    blockDim.y = 768; gridDim.y = ceil(M / blockDim.y);
}
// Lanzamiento de kernel
MatrixTransposeKernel<<<gridDim, blockDim>>>(M , N);
```

Por las razones mencionadas en la sección 4.1.2 (versión paralela de suma de vectores), se fijará el número máximo de *threads* por *block* en 768. Si el número de filas sobrepasa dicho número, se empleará más de un *block*. Para calcular el número de *blocks* se divide el número de filas de la matriz *A* entre los 768 *threads* y al resultado aplicamos la operación techo (*ceil*). En este caso utilizamos el eje 'y' de los *blocks*. Ahora solo resta lanzar el *kernel MatrixTransposekernel*, cuyo código es el siguiente:

```
// MatrizTransposeCUDA paralelismo de índice externo con M threads,
//filas en paralelo
__global__ void MatrixTransposeKernel(int rows, int columns)
{
    float data = 0;
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    if (i < rows){
        for (int j = 0; j < columns; j++){
            // B(j , i) = A(i , j)
            surf2Dread(&data, matrixA_surfRef, i*4, j);
            surf2Dwrite(data, matrixB_surfRef, j*4, i);
        }
    }
}
```

Como se realizó la operación *ceil* para calcular el número de *blocks*, es posible que el número de *threads* del *grid* supere al número de filas de **A**, por lo tanto, primero verificamos que el identificador de los *threads* (en este casi *i*) sea menor al número de filas para que los *threads* sobrantes no realicen alguna operación que pueda alterar el resultado deseado. Como podemos observar y/o deducir, en cada *thread* se ejecuta un bucle *for* para barrer las columnas y así poder localizar cada elemento de **A** y **B**.

A continuación presentaremos otra versión de paralelismo externo, ahora calculando las columnas en paralelo. Para esto calculamos el número de *threads* y *blocks* en función del número de columnas de **A**, de manera similar que en la versión anterior:

```
// Dimensionamiento del grid para la función kernel
dim3 gridDim; gridDim.y = 1; gridDim.z = 1;
dim3 blockDim; blockDim.y = 1; blockDim.z = 1;
if (N < 768){
    blockDim.x = N; gridDim.x = 1;
}else{
    blockDim.x = 768; gridDim.x = ceil(N / blockDim.x);
}
// Lanzamiento de kernel
MatrixTransposeKernel<<<gridDim, blockDim>>>(M , N);
```

Debido a que el bucle asociado a las columnas es el bucle interno, primero realizamos una permutación de bucle al algoritmo 4.1 para que ahora sea el bucle externo el que itere las columnas. Como no existe dependencia de datos en el algoritmo 4.1 es posible intercambiar los bucles sin realizar todo el proceso de una transformación unimodular. El código del *kernel* es el siguiente:

```
// MatrizTransposeCUDA paralelismo de índice externo con N threads,
//columnas en paralelo
_global_ void MatrixTransposeKernel(int rows, int columns)
{
    float data = 0;
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    if (j < columns){
        for (int i = 0; i < rows; j++){
            // B(j , i) = A(i , j)
            surf2Dread(&data, matrixA_surfRef, i*4, j);
            surf2Dwrite(data, matrixB_surfRef, j*4, i);
        }
    }
}
```

Ahora las columnas se calculan en paralelo, cada *thread* se asocia a una columna de **A** y ejecuta un ciclo *for* para barrer las filas y así tener acceso a cada elemento de **A**.

4.2.4. Versión con paralelismo interno

Partiendo como en las versiones paralelas anteriores, se parte del algoritmo 4.1, ahora deseamos ejecutar en paralelo el bucle interno el cual está asociado a las columnas de la matriz **A**. Calculamos el número de *blocks* y *threads* en base al número de columnas, de igual forma que en la primera versión de paralelismo externo. La gran diferencia es que se necesitará lanzar más de un *kernel*, uno para cada columna de **A**:

```
// Dimensionamiento del grid para la función kernel
dim3 gridDim; gridDim.x = 1; gridDim.z = 1;
dim3 blockDim; blockDim.x = 1; blockDim.z = 1;
if (M < 768){
    blockDim.y = M; gridDim.y = 1;
}else{
    blockDim.y = 768; gridDim.y = ceil(M / blockDim.y);
}
// Lanzamiento de kernel
for (int j = 0; j < columns; j++){
    MatrixTransposeKernel<<<gridDim, blockDim>>>(M , j);
}
```

El *kernel* necesita el valor de '*j*' y el número de filas de **A**. El código del kernel es el siguiente:

```
// MatrizTransposeCUDA paralelismo de índice interno con M threads,
//filas en paralelo
__global__ void MatrixTransposeKernel(int rows, int j)
{
    float data = 0;
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    if (i < rows){
        // B(j , i) = A(i , j)
        surf2Dread(&data, matrixA_surfRef, i*4, j);
        surf2Dwrite(data, matrixB_surfRef, j*4, i);
    }
}
```

Los *threads* se asocian a las filas de **A** por lo cual se tiene acceso en paralelo a todos los elementos de la columna '*j*'. Como podemos notar, el bucle *for* se ejecuta fuera del *kernel*.

4.2.5. Versión con paralelismo en 2 dimensiones

Nuestro objetivo en esta sección es utilizar la capacidad de *CUDA* para poder manejar *threads* en dos dimensiones. Para esto utilizamos los *blocks* y *threads* en el eje 'y' y 'x' para manejar las filas y las columnas de **A**, respectivamente. Primero definimos el número de *threads* por cada *block* que vamos a utilizar, el cual será el producto de *blockDim.x*, *blockDim.y* y *blockDim.z*. Para tener una *occupancy* del 100 % el número de *threads* por *block* debe ser uno de los valores siguientes: 192, 256, 384, 512, 768; los cuales se extrajeron de la figura 2.8. Elegimos 256, ya que es el único valor que tiene raíz entera, así, podemos fijar *blockDim.x* y *blockDim.y* en 16 y *blockDim.z* en uno. El dimensionamiento del *grid* es el siguiente:

```
// Dimensionamiento del grid para la función kernel
dim3 gridDim; gridDim.z = 1;
dim3 blockDim; blockDim.z = 1;
if (M < 16){      blockDim.y = M; gridDim.y = 1;
}else{
    blockDim.y = 16; gridDim.y = ceil(M / blockDim.y);
}
if (N < 16){      blockDim.x = N; gridDim.x = 1;
}else{
    blockDim.x = 16; gridDim.x = ceil(N / blockDim.x);
}
// Lanzamiento de kernel
MatrixTransposeKernel<<<gridDim, blockDim>>>(M , N);
```

Se requiere sólo lanzar un *kernel* para hacer el cálculo de toda la matriz **B**, cuyo código es el siguiente:

```
// MatrizTransposeCUDA paralelismo de índice externo con M x N threads,
// filas y columnas en paralelo
__global__ void MatrixTransposeKernel(int rows, int columns)
{
    float data = 0;
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < rows && j < columns){
        surf2Dread(&data, matrixA_surfRef, i*4, j);
        surf2Dwrite(data, matrixB_surfRef, j*4, i);
    }
}
```

Verificamos que los *threads* en los ejes 'y' y 'x' se encuentren dentro de las dimensiones de **A**. No es necesario para esta versión el utilizar ciclos *for* ya que ambos bucles se pueden ejecutar en paralelo. Además, las coordenadas para localizar los elementos de **A** se generan con las fórmulas de i y j que utilizan el identificador de *thread* y las dimensiones del *grid*.

4.2.6. Análisis de resultados

La figura 4.5 muestra el resultado de las 6 versiones implementadas, en donde se muestra el tiempo promedio al ejecutarse 100 veces cada algoritmo para cada tamaño de matriz (matrices cuadradas). La versión secuencial CUDA 1 *thread* es generalmente la más lenta, para tamaños de matrices de 2, 4 y 8 la versión CUDA paralelismo interno es la más lenta, debido a que se hicieron 2, 4 y 8 lanzamientos de kernel, respectivamente. Cada lanzamiento de kernel tiene su costo en tiempo, al crearse *grids* cada vez, en asignar y liberar recursos de procesamiento (CUDA *cores*). La versión de paralelismo interno no está del todo despreciable, ya que para matrices de 1024 logra superar al tiempo obtenido por la versión secuencial en *C*.

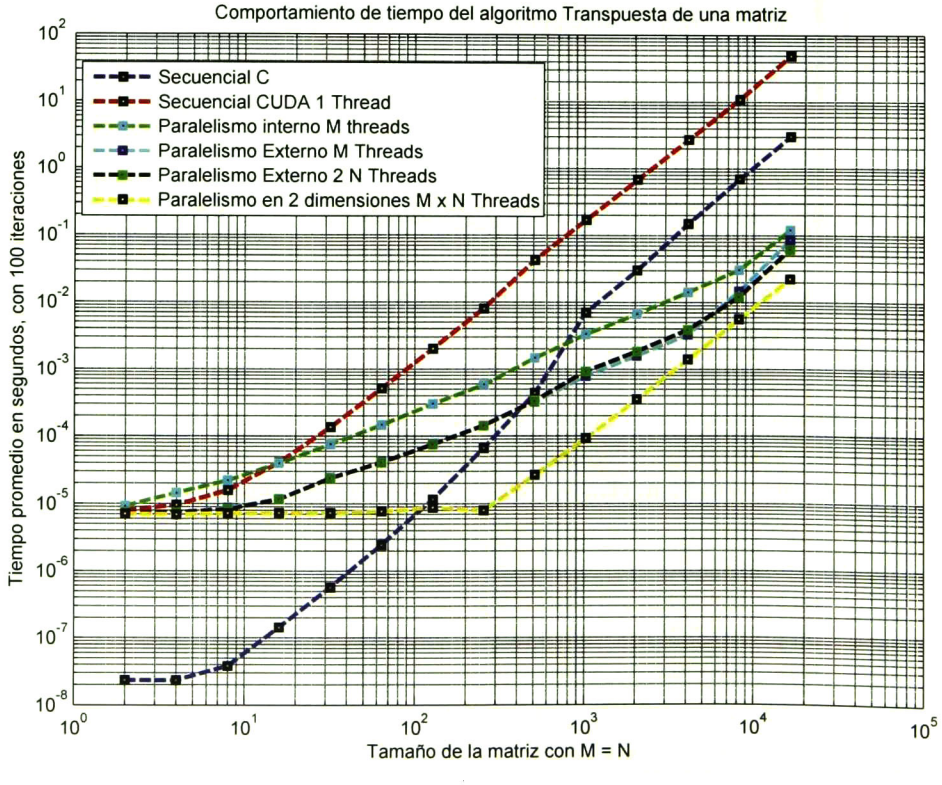


Figura 4.5: Comportamiento de tiempo del algoritmo Transpuesta de una Matriz.

Tabla 4.3: Resultados (en segundos) de matriz transpuesta de $m \times m$.

tamaño	CUDA secuencial	CUDA paralelismo interno	CUDA paralelismo externo	CUDA paralelismo 2D
2	7.8691×10^{-6}	9.2175×10^{-6}	7.3708×10^{-6}	7.0774×10^{-6}
4	9.5606×10^{-6}	1.4477×10^{-5}	7.4271×10^{-6}	6.9552×10^{-6}
8	1.5829×10^{-5}	2.2225×10^{-5}	8.6617×10^{-6}	7.2092×10^{-6}
16	4.0528×10^{-5}	3.9765×10^{-5}	1.1723×10^{-5}	7.3007×10^{-6}
32	1.3621×10^{-4}	7.6218×10^{-5}	2.4026×10^{-5}	7.2787×10^{-6}
64	5.1180×10^{-4}	1.4772×10^{-4}	4.1092×10^{-5}	7.7151×10^{-6}
128	2.0033×10^{-3}	3.0305×10^{-4}	7.7076×10^{-5}	8.7951×10^{-6}
256	8.0118×10^{-3}	5.8856×10^{-4}	1.4421×10^{-4}	7.9948×10^{-6}
512	4.2780×10^{-2}	1.4745×10^{-3}	3.2753×10^{-4}	2.6688×10^{-5}
1024	1.7087×10^{-1}	3.3319×10^{-3}	7.9461×10^{-4}	9.2394×10^{-5}
2048	6.8379×10^{-1}	6.7262×10^{-3}	1.5934×10^{-3}	3.5381×10^{-4}
4096	2.7343	14.2883×10^{-3}	3.3127×10^{-3}	1.4022×10^{-3}
8192	10.9443	30.9368×10^{-3}	14.7046×10^{-3}	5.6040×10^{-3}
16384	49.1541	121.325×10^{-3}	86.4963×10^{-3}	22.4468×10^{-3}

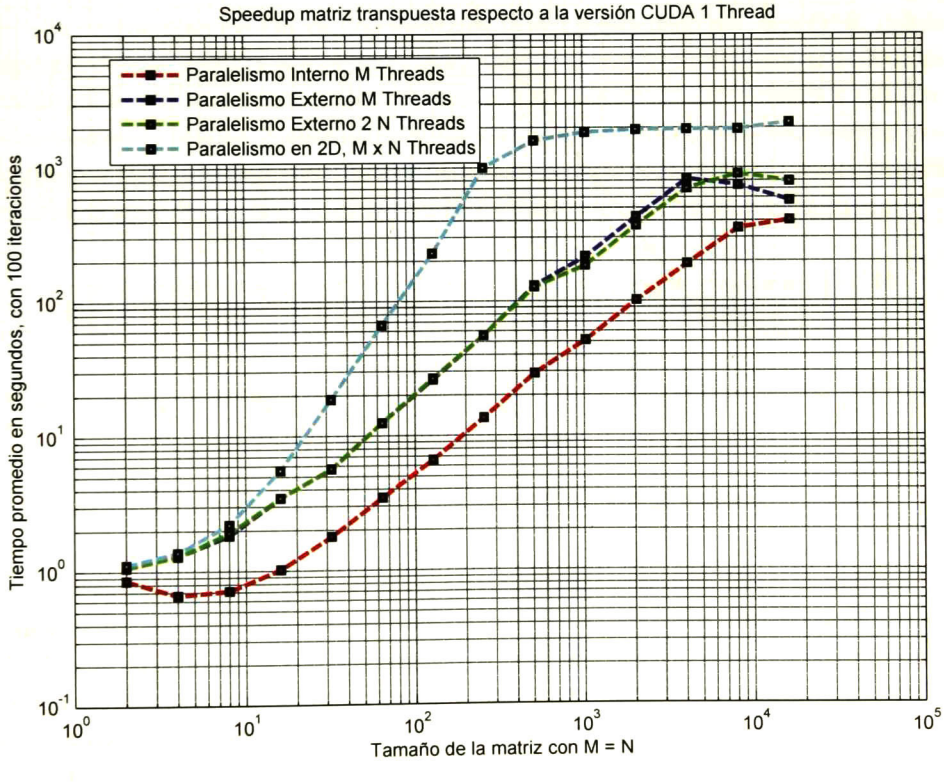


Figura 4.6: *Speedup* de Matriz Transpuesta respecto a la versión CUDA 1 *Thread*.

Las versiones de paralelismo externo, se ejecutan en menor tiempo que la versión de paralelismo interno, lo cual demuestra que tiene mejores resultados utilizar el paralelismo externo en CUDA, debido a que se lanzan menos *kernel* y no se necesita tanta interacción del *host* con el *device* para la ejecución de los mismos.

La mejor versión implementada es la paralela 2D, a partir de tamaños de matriz de 128 ya presenta menores tiempos de ejecución que la versión secuencial en C. Para tamaños de matriz de 2 a 256 se obtiene prácticamente el mismo tiempo, ya que los CUDA *cores* no se utilizan completamente hasta tamaños de 128, ya para tamaños de 256 (256*256 = 65536 *threads* necesarios) se superan los 21504 threads máximos que pueden ser ejecutados en el GPU de manera concurrente.

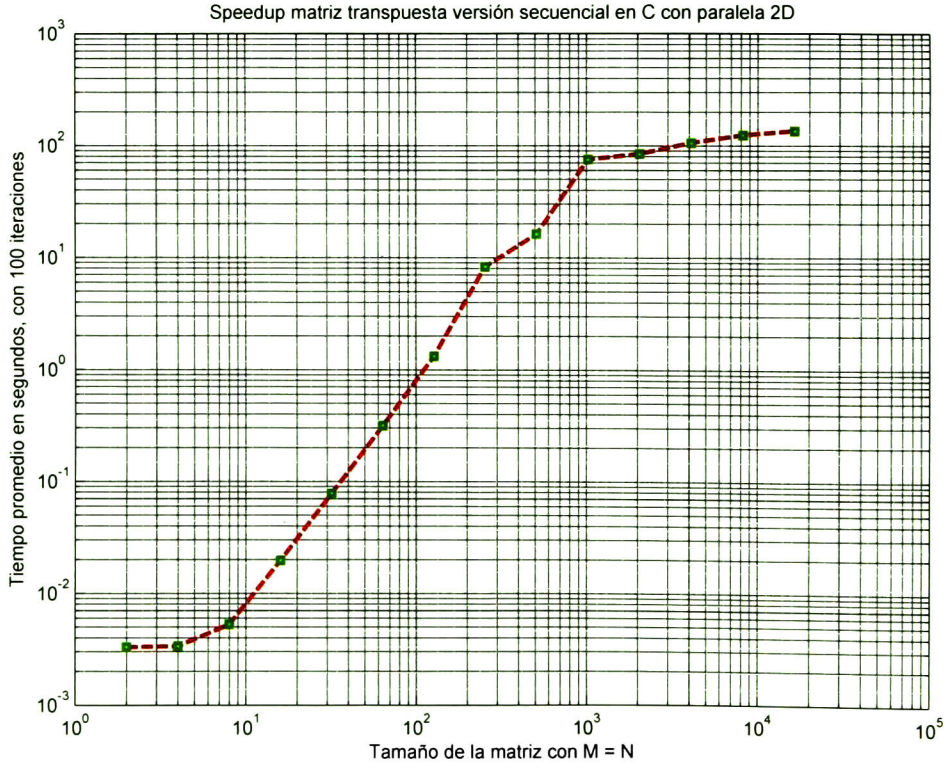


Figura 4.7: *Speedup* de Matriz Transpuesta respecto a la versión CUDA 1 *Thread*.

Como se puede observar en las figuras 4.5 y 4.6, las dos versiones de paralelismo externo tienen prácticamente el mismo comportamiento y el mismo *speedup*. Para tamaños mayores de 512 ya se dejan de traslapar. Al ejecutarse la misma cantidad de *threads* (debido a que manejamos matrices cuadradas) y que en cada *thread* sólo se realizan accesos a memoria vía *surfaces* el factor que pudo influir en el resultado es la manera en como las funciones de lectura y escritura de *surface* tienen implementado los accesos a memoria en dos dimensiones, lo cual ya está fuera de nuestro control, sin embargo la diferencia

de tiempo no es grande y a veces despreciable.

En la figura 4.7 se muestra el *speedup* de la versión secuencial en *C* con la versión paralela 2D. Como se puede observar, para matrices menores de 128 presenta mejores resultados la versión secuencial en *C* al obtenerse factores de *speedup* menores a uno. A partir de 128 la versión paralela supera a la secuencial, lográndose un *speedup* mayores de 100, lo cual nos dice que la versión paralela es 100 veces más rápida que la versión secuencial en *C*. Como se muestra en la figura 4.6 se alcanza un *speedup* mayor a 1000 entre las versiones de CUDA secuencial y paralela 2D. Para ver los valores de manera exacta de las versiones de CUDA, se proporciona la tabla 4.3, además en el CD de la tesis se presentan los códigos de las versiones y códigos para reproducir las gráficas mostradas.

A continuación analizaremos el gasto de memoria de las versiones. Todas las versiones requieren almacenar en el *host* las dos matrices, *A* y *B*, lo cual tiene un costo de 2*4*filas*columnas. Las versiones de CUDA necesitan guardar las matrices *A* y *B* en el *device*, por ejemplo para matrices cuadradas de 16384, se necesita reservar 2*4*16384*16384 = 2147483648 bytes (2.147 GB).

4.3. Descomposición QR

La descomposición QR de una matriz **A** de *m x n* está dado por **A** = **Q***R, donde $\mathbf{Q} \in \mathbb{R}^{mxm}$ es ortogonal y $\mathbf{R} \in \mathbb{R}^{mxn}$ es triangular superior, además **R** puede ser calculada mediante $\mathbf{R} = \mathbf{Q}^t * \mathbf{A}$. Los métodos principales para realizar la descomposición QR son: Gram-Schmidt (MGS), Householder y las rotaciones de Givens. En este estudio nos enfocaremos al método de rotaciones de Givens, en donde cada rotación va a insertar un cero en la matriz **A** hasta que sea igual a **R**. La matriz de rotación **G** tiene la forma:

$$\mathbf{G}_{(i,k)} = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & i & & k & & \end{matrix}$$

Donde los parámetros *c* (i.e coseno) y *s* (i.e seno) pueden ser calculados como sigue

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}} \tag{4.3}$$

A su vez, x_i y x_k son elementos de una columna de **A** en donde queremos insertar un cero, el cual se inserta en x_k una vez aplicada la rotación, (i.e después de multiplicar

$\mathbf{G}_{(i,k)} * \mathbf{A}$). La matriz **Q** se forma por el producto de las matrices de rotación, $\mathbf{Q} = \mathbf{G}_1 * \mathbf{G}_2 * \cdots * \mathbf{G}_t$, donde t es el número de ceros que se introducen o generan en **A** [21].

Ejemplo de la descomposición QR de la matriz **A** de 4 filas por 4 columnas:

$$\mathbf{A} = \begin{pmatrix} 12 & -51 & 4 & 1 \\ 6 & 167 & -68 & 1 \\ -4 & 24 & -41 & 1 \\ 7 & 8 & 9 & 1 \end{pmatrix} = \mathbf{Q} * \mathbf{R} =$$

$$\begin{pmatrix} 0{,}7667 & -0{,}3894 & -0{,}4485 & 0{,}2440 \\ 0{,}3833 & 0{,}9052 & -0{,}0324 & 0{,}1805 \\ -0{,}2556 & 0{,}1698 & -0{,}8339 & -0{,}4588 \\ 0{,}4472 & -0{,}0144 & 0{,}3201 & -0{,}8351 \end{pmatrix} * \begin{pmatrix} 15{,}6225 & 22{,}3607 & -8{,}4971 & 1{,}3416 \\ 0 & 175{,}0143 & -70{,}1771 & 0{,}6742 \\ 0 & 0 & 37{,}4829 & -0{,}9947 \\ 0 & 0 & 0 & -0{,}8695 \end{pmatrix}$$

4.3.1. Versión original

Partimos del algoritmo 4.2, en donde el índice 'i' se relaciona con las filas y los índices 'k' y 'j' con columnas de **A**.

Algoritmo 4.2 Algoritmo original descomposición QR

Entrada: Matriz **A** de m x n y $m \geq n$.

Salida: Sobrescribe la matriz **A** con **R**, tal que, $\mathbf{R} = \mathbf{Q}^t * \mathbf{A}$.

```
for k = 0, N - 1, 1 do
    for i = M - 1, k + 1, -1 do
        r = sqrt(a[-i - 1][k]^2 + a[-i][k]^2)
        c = a[-i - 1][k] / r
        s = a[-i][k] / r
        for j = k, N - 1, 1 do
            temp = a[-i - 1][j]
            a[-i - 1][j] = c * temp + s * a[-i][j]
            a[-i][j] = -s * temp + c * a[-i][j]
        end for
    end for
end for
```

4.3.2. Transformación del algoritmo

Para la transformación del algoritmo 4.2 se analizaron las técnicas utilizadas en la tesis [2] en donde, el objetivo final (de las transformaciones) era implementar la descomposición QR en un arreglo sistólico de procesadores . En nuestro caso, queremos transformar el algoritmo de tal manera que podamos tener una versión paralela que se

ejecute en un GPU, para esto tomaremos las partes de la metodología que sean compatibles y podamos aprovechar en un GPU.

La primera técnica que vamos a emplear es la de **hundimiento de código** , la cual consiste en conseguir un nido de bucles perfecto. Un nido de bucles perfecto (como se definió en la sección 3.2) contiene todas las sentencias dentro del bucle más interno. Además se añadió una condicional para que evitar divisiones entre cero cuando r sea igual a cero, en el cálculo de c y s. El resultado de estas modificaciones se ven plasmadas en el algoritmo 4.3.

Algoritmo 4.3 Hundimiento de código

Entrada: Matriz **A**.
Salida: Matriz **R** de la descomposición QR de **A**.

```
for k = 0, N - 1, 1 do
    for i = M - 1, k + 1, -1 do
        for j = k, N - 1, 1 do
            if j == k then
                r = sqrt(a[i - 1][j]^2 + a[i][j]^2)
                if j < N - 1 then
                    if r == 0 then
                        c = 0
                        s = 0
                    else
                        c = a[i - 1][j] / r
                        s = a[i][j] / r
                    end if
                    a[i - 1][j] = r
                    a[i][j] = 0
                end if
            else
                temp = a[-i - 1][j]
                a[i - 1][j] = c * temp + s * a[i][j]
                a[i][j] = -s * temp + c * a[i][j]
            end if
        end for
    end for
end for
```

La siguiente técnica que se empleó fue la de **normalización** La normalización consiste en hacer que todos los bucles tengan incrementos positivos [2]. Esto es necesario para habilitar la trasformación del espacio de iteración mediante una matriz unimodular, como se verá mas adelante. El algoritmo 4.4 ya contiene esta modificación.

Algoritmo 4.4 Normalización

Entrada: Matriz **A**.
Salida: Matriz **R** de la descomposición QR de **A**.

```
for k = 0, N - 1, 1 do
    for i = - M + 1, -k - 1, 1 do
        for j = k, N - 1, 1 do
            if j == k then
                r = sqrt(a[-i - 1][j]^2 + a[-i][j]^2)
                if j < N - 1 then
                    if r == 0 then
                        c = 0
                        s = 0
                    else
                        c = a[-i - 1][j] / r
                        s = a[-i][j] / r
                    end if
                    a[-i - 1][j] = r
                    a[-i][j] = 0
                end if
            else
                temp = a[-i - 1][j]
                a[-i - 1][j] = c * temp + s * a[-i][j]
                a[-i][j] = -s * temp + c * a[-i][j]
            end if
        end for
    end for
end for
```

Hasta este punto se han venido realizando las mimas modificaciones en el código, la primera diferencia aparece al aplicar la técnica de **concordancia de índices** . La cual consiste en mapear cada una de las asignaciones presentes en el código, al espacio de índices $x = [i, j, k]$ determinado por el número de bucles presentes en el programa [2]. Para lograr la concordancia de índices, en la tesis anterior, se modificó el programa para que todas las variables fueran accesadas por los tres índices (i, j, k). Al realizar esto las variables r, c, s y tl, que eran variables temporales, ahora se convierten en arreglos en 3 dimensiones, de manera similar con la matriz **A**, que pasó de ser un arreglo bidimensional a un arreglo en 3 dimensiones. El efecto emplear ésta técnica de esta manera, es el de aumentar de manera considerable el consumo de memoria de nuestro algoritmo. Una vez obtenida la concordancia de índices, se aplicaron las técnicas de asignación única y eliminación global. **Asignación única** consiste en que cada variable se le asigna valor una sola vez, durante toda la ejecución del programa. La **eliminación**

de comunicación global evita la difusión global de datos y la sustituye por múltiples comunicaciones punto a punto. El resultado de aplicar estas 3 técnicas se refleja en el algoritmo 4.5 el cual se obtuvo en la tesis [2].

Algoritmo 4.5 Politopo fuente 3D

Entrada: Matriz **A**.
Salida: Matriz **R** de la descomposición QR de **A**.

for k = 0, N - 1, 1 **do**
for i = -M + 1,-k - 1, 1 **do**
for j = k, N - 1, 1 **do**
if j == k **then**
if i == -M + 1 **then**
$r[-i][j][k] = a[-i][j][k]$
end if
$r[-i-1][j][k] = \sqrt{a[-i-1][j][k]^2 + r[-i][j][k]^2}$
if r[-i-1][j][k] == 0 **then**
$c[-i][j+1][k] = 0$
$s[-i][j+1][k] = 0$
else
$c[-i][j+1][k] = \dfrac{a[-i-1][j][k]}{r[-i-1][j][k]}$
$s[-i][j+1][k] = \dfrac{r[-i][j][k]}{r[-i-1][j][k]}$
end if
if i == -k - 1 **then**
$a[-i-1][j][k+1] = r[-i-1][j][k]$
end if
else
if i == -M - 1 **then**
$t1[-i][j][k] = a[-i][j][k]$
end if
$t1[-i-1][j][k] = c[-i][j][k] * a[-i-1][j][k] + s[-i][j][k] * t1[-i][j][k]$
$a[-i][j][k+1] = -s[-i][j][k] * a[-i-1][j][k] + c[-i][j] * t1[-i][j][k]$
if j == -k - 1 **then**
$a[-i-1][j][k+1] = t1[-i-1][j][k]$
$c[-i][j+1][k] = c[-i][j][k]$
$s[-i][j+1][k] = s[-i][j][k]$
end if
end if
end for
end for
end for

Analizando el código y las tres técnicas mencionadas anteriormente, se observó que las variables c y s eran las necesarias para calcular las rotaciones, y que si deseábamos realizar varias rotaciones en paralelo se debía almacenar (para cada rotación) el valor de c y de s en arreglos bidimensionales para eliminar la comunicación global. Las variables r sólo se utiliza para calcular c y s por lo que puede ser guardada en una variable temporal. De manera similar, la variable $t1$ se utiliza para guardar temporalmente para guardar un elemento de **A**, por lo que puede ser almacenada en un registro. El algoritmo 4.6 muestra muestra como queda nuestro politopo fuente.

Algoritmo 4.6 Politopo fuente

Entrada: Matriz **A**.
Salida: Matriz **R** de la descomposición QR de **A**.

```
for k = 0, N - 1, 1 do
    for i = -M + 1, -k - 1, 1 do
        for j = k, N - 1, 1 do
            if j == k then
                r = sqrt(a[-i - 1][j]^2 + a[-i][j]^2)
                if j < N - 1 then
                    if r == 0 then
                        c[-i - 1][j] = 0
                        s[-i - 1][j] = 0
                    else
                        c[-i - 1][j] = a[-i - 1][j] / r
                        s[-i - 1][j] = a[-i][j] / r
                    end if
                    a[-i - 1][j] = r
                    a[-i][j] = 0
                end if
            else
                temp = a[-i - 1][j]
                a[-i - 1][j] = c[-i - 1][j - 1] * temp + s[-i - 1][j - 1] * a[-i][j]
                a[-i][j] = -s[-i - 1][j - 1] * temp + c[-i - 1][j - 1] * a[-i][j]
                if j < N - 1 then
                    c[-i - 1][j] = c[-i - 1][j - 1]
                    s[-i - 1][j] = s[-i - 1][j - 1]
                end if
            end if
        end for
    end for
end for
```

La técnica de asignación única se descartó completamente ya que implica que cada resultado de una operación se guarde en memoria, en el caso de arreglos de procesadores sistólicos se puede evitar almacenar en memoria cada resultado preliminar, pasando los resultados intermedios mediante la red de interconexión de los procesadores. En un GPU no es posible transferir datos de un procesador a otro para poder tener acceso a datos comunes es necesario guardarlos en memoria global del GPU (ver figura 2.7).

4.3.3. Versión paralela

El paso siguiente es aplicar la transformación unimodular al programa politopo destino. Para eso, seguiremos el procedimiento establecido en la sección 3.4.3 para modificar el algoritmo 4.6:

1. Encontrar todos los vectores distancia **d**.
   Podemos observar que existen tres dependencias en el cuerpo de los bucles. El primer vector distancia $\mathbf{d}_1$ se extrae de $a(-i-1, j)$ y $A(-i, j)$; $\mathbf{d}_1 = (-i-1-(-i), j-j) = (-1, 0)$. El segundo vector distancia $\mathbf{d}_2$ se extrae de $a(-i, j)$ y $A(-i-1, j)$; $\mathbf{d}_2 = (-i-(-i-1), j-j) = (1, 0)$. El tercer vector distancia $\mathbf{d}_3$ se extrae de $c(-i-1, j)$ y $c(-i-1, j-1)$; $\mathbf{d}_2 = (-i-1-(-i-1), j-(j-1)) = (0, 1)$.
   La matriz distancia es $\mathcal{D} = \begin{pmatrix} -1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$. Se agregó unos a la columna correspondiente a k, debido a que existe una dependencia de flujo en las matrices **A**, **C** y S, al definir el valor de una de sus columnas (en la iteración k) y utilizar el mismo valor en una iteración posterior (k+1). En la figura 4.8 se representa el politopo fuente y el politopo destino.

2. Seleccionar la matriz de transformación **U**.
   La matriz unimodular que se utilizará es $\mathbf{U} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}$, la cual se obtiene de aplicar el método descrito en la sección 3.5.2.

3. Verificar la legalidad de la transformación, $\mathbf{dU} \succ \mathbf{0}$ para cada vector distancia **d**.

$$\mathbf{d}_1\mathbf{U} = (-1, 0, 1)\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} = (1, 0, 1) \succ \mathbf{0}$$

$$\mathbf{d}_2\mathbf{U} = (1, 0, 1)\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} = (3, 0, 1) \succ \mathbf{0}$$

$$\mathbf{d}_3\mathbf{U} = (0,1,1)\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 2 & 0 & 1\end{pmatrix} = (3,1,1) > \mathbf{0}$$

Como $\mathbf{d}_1\mathbf{U}$, $\mathbf{d}_2\mathbf{U}$ y $\mathbf{d}_2\mathbf{U}$ son lexicográficamente positivos la transformación es legal.

4. Calcular el nuevo vector de iteración $\mathbf{y} = (t, p_2, p_2)$, mediante $\mathbf{x} = \mathbf{y}\mathbf{U}^{-1}$

   Para esto necesitamos calcular $\mathbf{U}^{-1} = \begin{pmatrix}1 & 0 & 0\\ -1 & 1 & 0\\ -2 & 0 & 1\end{pmatrix}$,

   ahora $\mathbf{x} = \mathbf{y}\mathbf{U}^{-1} = (i,j,k) = (t, p_1, p_2)\begin{pmatrix}1 & 0 & 0\\ -1 & 1 & 0\\ -2 & 0 & 1\end{pmatrix} = (t - p1 - 2p_2, p_1, p_2)$.

5. Expresar los límites del espacio de iteración original en forma de politopo, $\mathbf{x}\mathbf{A} \leq \mathbf{b}$.

   El espacio de iteración original consiste en todos los vectores enteros (i,j,k) tal que:

$$\left.\begin{array}{lcccl} -M+1 & \leq & i & \leq & -K-1\\ k & \leq & j & \leq & N-1\\ 0 & \leq & k & \leq & N-1 \end{array}\right\} \tag{4.4}$$

   El politopo fuente $\mathbf{x}\mathbf{A} \leq \mathbf{b}$ es:

$$(i,j,k)\begin{pmatrix}-1 & 0 & 0 & 1 & 0 & 0\\ 0 & -1 & 0 & 0 & 1 & 0\\ 0 & 1 & -1 & 1 & 0 & 1\end{pmatrix} \leq (M-1,0,0,-1,N-1,N-1) \tag{4.5}$$

6. Calcular el nuevo espacio de iteración, $\mathbf{y}\mathbf{U}^{-1}\mathbf{A} \leq \mathbf{b}$.

   dado que $\mathbf{A} = \begin{pmatrix}-1 & 0 & 0 & 1 & 0 & 0\\ 0 & -1 & 0 & 0 & 1 & 0\\ 0 & 1 & -1 & 1 & 0 & 1\end{pmatrix}$, $\mathbf{b} = (M-1,0,0,-1,N-1,N-1)$ y $\mathbf{U}^{-1} = \begin{pmatrix}1 & 0 & 0\\ -1 & 1 & 0\\ -2 & 0 & 1\end{pmatrix}$

   $\mathbf{y}\mathbf{U}^{-1}\mathbf{A} \leq \mathbf{b}$ es equivalente a:

$$(t, p_1, p_2)\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 1 & 0 & 1 \end{pmatrix} \leq (M-1, 0, 0, -1, N-1, N-1)$$

$$(t, p_1, p_2)\begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & -1 & 1 & 0 \\ 2 & 1 & -1 & -1 & 0 & 1 \end{pmatrix} \leq (M-1, 0, 0, -1, N-1, N-1)$$

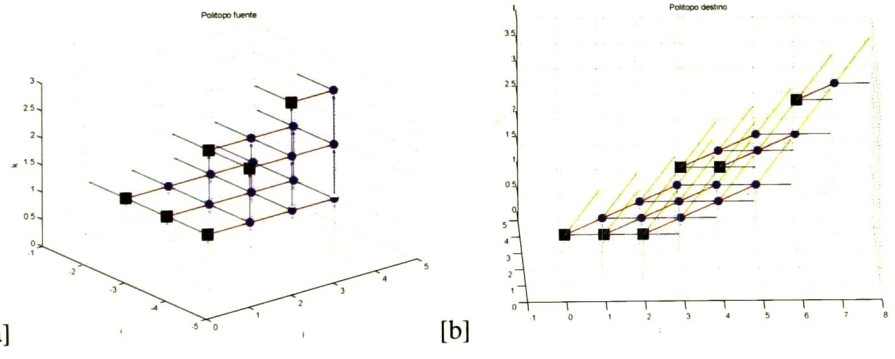
7. Resolver el sistema de desigualdades aplicando el método de eliminación de *Fourier-Motzkin* [4], para encontrar los nuevos límites inferiores y superiores de **y**. Los nuevos límites para **y** después de hacer que t empiece en cero y eliminar los límites redundante, son:

$$0 \leq t \leq M + 2 * N - 2$$

$$\lceil \text{máx}(0, \frac{t - M + 2}{2}) \rceil \leq p1 \leq \lfloor \text{mín}(t + M - 1, N - 1) \rfloor$$

$$\text{máx}(0, t - p1 - M + 2) \leq p2 \leq \lfloor \text{mín}(p1, \frac{t - p1}{2}) \rfloor$$

8. Reemplazar **x** por **y** en el cuerpo del nido de bucles (ahora **H(y)**) y en los nuevos límites de los bucles *For*. El programa destino resultante es el algoritmo 4.7.



[a] [b]

Figura 4.8: Grafo de dependencia del politopo fuente (izquierda) y del politopo destino (derecha).

Algoritmo 4.7 Politopo destino 2D

Entrada: Matriz A.

Salida: Matriz R de la descomposición QR de A.

```
for t = 0, M + 2*N - 2, 1 do
    for p1 = ⌈máx(0, (t − M + 2)/2)⌉, ⌊mín(t + M − 1, N − 1)⌋, 1 do
        for p2 = máx(0, t − p1 − M + 2), ⌊mín(p1, (t − p1)/2)⌋, 1 do
            i = -( -( t + ( 1 - M ) ) + p1 + 2*p2 )
            j = p1
            k = p2
            if j == k then
                r = √(a[−i − 1][j]² + a[−i][j]²)
                if j < N - 1 then
                    if r == 0 then
                        c[−i − 1][j] = 0
                        s[−i − 1][j] = 0
                    else
                        c[−i − 1][j] = a[−i − 1][j] / r
                        s[−i − 1][j] = a[−i][j] / r
                    end if
                    a[−i − 1][j] = r
                    a[−i][j] = 0
                end if
            else
                temp = a[−i − 1][j]
                a[−i − 1][j] = c[−i − 1][j − 1] * temp + s[−i − 1][j − 1] * a[−i][j]
                a[−i][j] = −s[−i − 1][j − 1] * temp + c[−i − 1][j − 1] * a[−i][j]
                if j < N - 1 then
                    c[−i − 1][j] = c[−i − 1][j − 1]
                    s[−i − 1][j] = s[−i − 1][j − 1]
                end if
            end if
        end for
    end for
end for
```

El politopo destino obtenido en [2] es el siguiente:

Algoritmo 4.8 Politopo destino 3D

Entrada: Matriz A.

Salida: Matriz R de la descomposición QR de A.

for t = 0, M + 2*N - 2, 1 **do**
for p1 = $\lceil$máx(0, $\frac{t-M+2}{2}$)$\rceil$, $\lfloor$mín($t+M-1, N-1$)$\rfloor$, 1 **do**
for p2 = máx(0, $t-p1-M+2$), $\lfloor$mín($p1, \frac{t-p1}{2}$)$\rfloor$, 1 **do**
i = -(-(t + (1 - M)) + p1 + 2*p2)
j = p1
k = p2
if j == k **then**
if i == -M + 1 **then**
$r[-i][j][k] = a[-i][j][k]$
end if
$r[-i-1][j][k] = \sqrt{a[-i-1][j][k]^2 + r[-i][j][k]^2}$
if r[-i-1][j][k] == 0 **then**
$c[-i][j+1][k] = 0$
$s[-i][j+1][k] = 0$
else
$c[-i][j+1][k] = \frac{a[-i-1][j][k]}{r[-i-1][j][k]}$
$s[-i][j+1][k] = \frac{r[-i][j][k]}{r[-i-1][j][k]}$
end if
if i == -k - 1 **then**
$a[-i-1][j][k+1] = r[-i-1][j][k]$
end if
else
if i == -M - 1 **then**
$t1[-i][j][k] = a[-i][j][k]$
end if
$t1[-i-1][j][k] = c[-i][j][k] * a[-i-1][j][k] + s[-i][j][k] * t1[-i][j][k]$
$a[-i][j][k+1] = -s[-i][j][k] * a[-i-1][j][k] + c[-i][j] * t1[-i][j][k]$
if j == -k - 1 **then**
$a[-i-1][j][k+1] = t1[-i-1][j][k]$
$c[-i][j+1][k] = c[-i][j][k]$
$s[-i][j+1][k] = s[-i][j][k]$
end if
end if
end for
end for
end for

4.3.4. Implementación en CUDA

El primer problema a resolver es como vamos a mapear los índices p_1 y p_2 (son los que se pueden ejecutar en paralelo) en los *threads* del GPU. Observando el politopo destino desde los ejes p_1 y p_2, nos percatamos que se forma un trapecio con los puntos de iteración. En CUDA es posible manejar *threads* en dos dimensiones pero el espacio que pueden cubrir es rectangular. Si cubrimos el espacio de iteración con *threads* de *3 x 4* (para el ejemplo mostrado en la figura 4.9) se 3 *threads* no realizarían algún cálculo.

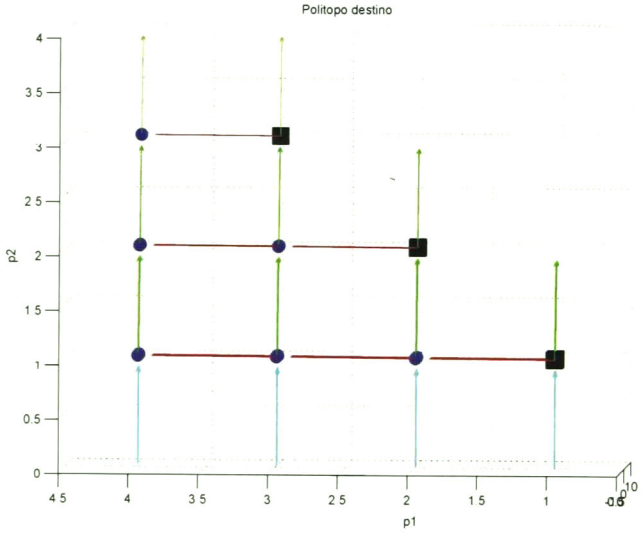


Figura 4.9: Politopo destino visto desde los ejes p_1 y p_2.

Para no utilizar una cantidad de *threads* mayor a la necesaria, se guardó las coordenadas de p_1 y p_2 para cada punto de iteración del algoritmo. Antes de lanzar el *kernel*, es necesario guardar todas las coordenadas y pasárselas al *device*. En cada iteración de t se lanza un *kernel*, debido a que es diferente la cantidad de puntos de iteración que se ejecutan en cada iteración de t, es necesario dimensionar el *grid* cada vez.

Al realizar lo anterior se implementaron en CUDA los algoritmos 4.8 y 4.7. Más adelante presentaremos los resultados obtenidos con estas versiones CUDA paralelo 3D y CUDA paralelo 2D versión 1.

Otra alternativa que se propone es la de dejar fijo la cantidad de *threads* para que cubra el espacio de iteración para p_1 y p_2. Para p_1 es necesario lanzar M *threads* y para p_2 es necesario lanzar $N - 1$ *threads*. La ventaja de esta versión es la de ahorrarse la memoria de guardar las coordenadas de p_1 y p_2. Otra ventaja es que no requiere cambiar el tamaño del *grid* para cada lanzamiento de *kernel*. En el código del *kernel*, cada *thread* verifica si debe hacer operaciones, evitando que se realicen cálculos indeseables. Esta versión la llamaremos CUDA paralelo 2D versión 2.

Los códigos completos se proporcionan en el CD de la tesis, además se encuentran los archivos que contienen los tiempos de ejecución para cada versión.

4.3.5. Análisis de resultados

De manera similar a los algoritmos anteriores, se presenta en la figura 4.10 el comportamiento de tiempo de las 6 versiones desarrolladas. Se utiliza una escala logarítmica para poder visualizar los tiempos de las versiones y así poder compararlas. Las 6 versiones desarrolladas son las siguientes:

- Original en *C*.
- Original en CUDA 1 *thread*.
- Politopo destino en CUDA 1 *thread*.
- CUDA paralelo 3D.
- CUDA paralelo 2D versión 1.
- CUDA paralelo 2D versión 2.

Se tomaron 13 muestras de tiempo promedio (con 100 iteraciones) con matrices cuadradas de 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 y 16384. Se presentará el comportamiento de tiempo, el *speedup* y el gasto de memoria para cada versión.

Primero analizaremos los resultados de las versiones secuenciales en CUDA (original 1 *thread* y politopo destino 1 *thread* 2D), los cuales se pueden observar en la tabla 4.5 y en la figura 4.10. Para tamaños de matriz de 2, 4 y 8 (matrices cuadradas), compiten e incluso arrojan menores tiempos de ejecución que las versiones paralelas. A partir de tamaños de 16, comienzan a ser mucho mayores sus tiempos que los del resto de las versiones. La versión original se ejecuta en menor tiempo que la versión politopo destino, esto se debe a que se agregaron mas instrucciones al código original para permitir su ejecución paralela, además de incrementar el uso de la memoria global al almacenar, en matrices, las variables *c* y *s* de las rotaciones de Givens.

La primera versión paralela que se realizó fue la paralela 3D, en donde las transformaciones al algoritmo original fueron descritas en la tesis [2]. Observamos que esta versión es la más lenta de las versiones paralelas y puede procesar matrices de hasta 512. Cabe aclarar que no es el límite exacto, puede ejecutar matrices un poco mayores, pero para tamaños de 1024 por 1024 era insuficiente la cantidad de memoria global del *device* (5.3 GB) para guardar las matrices necesarias para el cálculo. Además, esta versión no presentó un tiempo menor a la versión secuencial en *C* por lo que descartamos por completo el uso de esta versión para alguna aplicación donde se requiera el cálculo de la descomposición QR.

Si comparamos los resultados obtenidos por las versiones paralelas en 2D, observamos que la versión 1 se ejecuta en menos tiempo que la versión 2, tabla 4.4. Sin embargo, la versión 2 puede procesar matrices de hasta 16384 por 16384, lo cual no lo puede hacer la versión 1 debido a que se agota la memoria del *device*, al necesitar almacenar las coordenadas de todos los puntos de iteración. Ambas versiones tienen menores tiempos que la versión secuencial en *C* para tamaños de 256 en adelante. con lo cual, si se utilizan en lugar de la versión secuencial en *C*, se puede reducir considerablemente el tiempo de ejecución, en el orden de los milisegundos, segundos y hasta minutos, dependiendo del tamaño de la matriz (ver tabla 4.4).

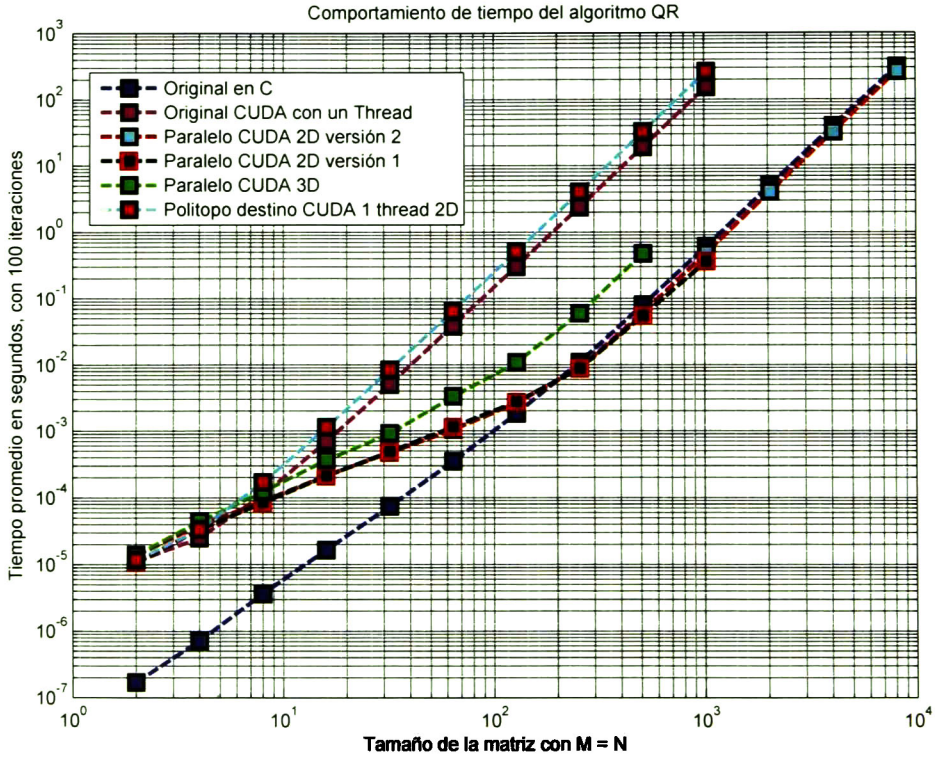


Figura 4.10: Comportamiento de tiempo del algoritmo QR.

En la figura 4.11 se muestra el *speedup* de las tres versiones paralelas en CUDA con respecto a la versión original CUDA 1 *thread*. También se ve reflejado en el *speedup* que la versión paralela 3D es la menos buena de las tres, ya un *speedup* menor no es deseable, alcanzando un S_p de 40 máximo, mientras que las versiones 2D alcanzaron un S_p de 300, para matrices de 512.

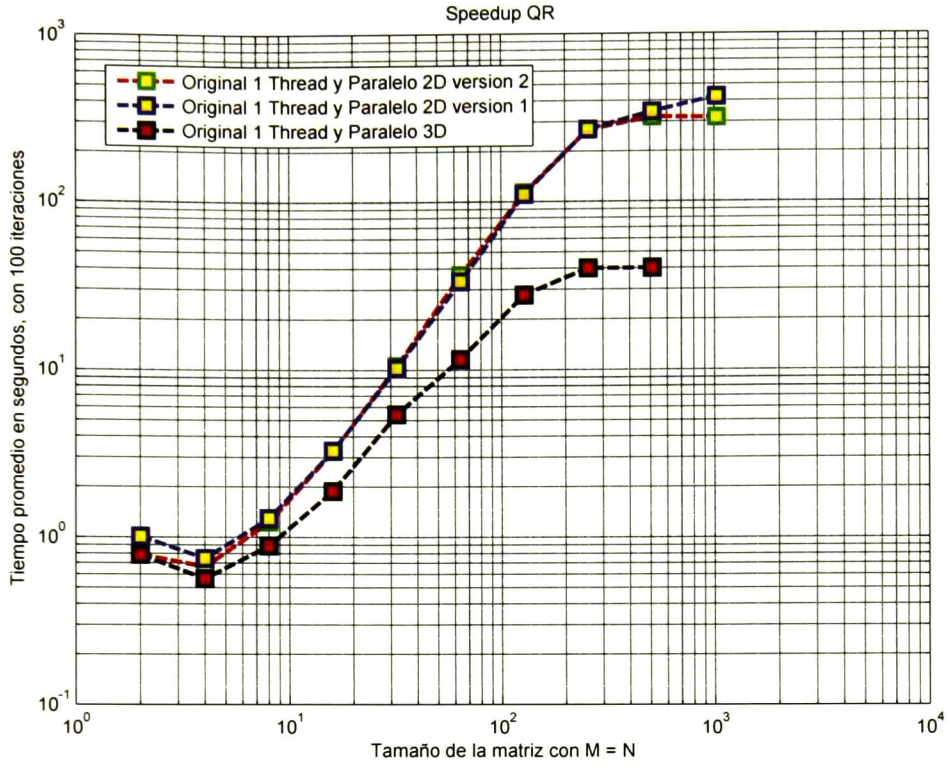


Figura 4.11: *Speedup* del algoritmo QR respecto a la versión inicial CUDA 1 Thread

El gasto de memoria (en bytes) de las versiones (ordenadas de menor a mayor) es el siguiente:

- Original en *C*: $4 * M * N$, almacenamiento de **A**.
- Original en CUDA 1 *thread*: $4 * M * N$, almacenamiento de **A**.
- Politopo destino en CUDA 1 *thread*: $4*(M*N+2(M-1)(N-1))$, almacenamiento de **A** (de *M x N*) , **C** y **S** (de *M-1 x N-1*) .
- CUDA paralelo 2D versión 2: $4 * M * N + 4 * 2(M-1)(N-1)$, almacenamiento de **A** (de *M x N*), **C**, **S** (de *M-1 x N-1*).
- CUDA paralelo 2D versión 1: $4 * M * N + 4 * 2(M-1)(N-1) + (4 * 2 * \sum_{1}^{k}(N-k+1)(M-k))$, almacenamiento de **A** (de *M x N*), **C**, **S** (de *M-1 x N-1*) y las los arreglos de p_1 y p_2 para guardar las coordenadas, (p_1, p_2), de cada punto de iteración.
- CUDA paralelo 3D: $4*5*M*N*N+(4*2*\sum_{1}^{k}(N-k+1)(M-k))$, almacenamiento de **A**, **C**, **S**, **R**, **T1** (de *MxNxN*) y las los arreglos de p_1 y p_2 para guardar las coordenadas de cada punto de iteración.

Tabla 4.4: Resultados (en segundos) del algoritmo QR de *m x m*.

Tamaño	Original en *C*	CUDA paralelo 3D	CUDA paralelo 2D	CUDA paralelo 2D modificado
2	1.6959×10^{-7}	1.3849×10^{-5}	1.0779×10^{-5}	1.4118×10^{-5}
4	7.1053×10^{-7}	4.4032×10^{-5}	3.3363×10^{-5}	3.7221×10^{-5}
8	3.6286×10^{-6}	1.2307×10^{-4}	8.4365×10^{-5}	8.9248×10^{-5}
16	1.6538×10^{-5}	3.6829×10^{-4}	2.1125×10^{-4}	2.1445×10^{-4}
32	7.4164×10^{-5}	9.1738×10^{-4}	4.8875×10^{-4}	4.7753×10^{-4}
64	3.5035×10^{-4}	3.3072×10^{-3}	1.1419×10^{-3}	1.0519×10^{-3}
128	1.8341×10^{-3}	1.0798×10^{-2}	2.7250×10^{-3}	2.6380×10^{-3}
256	1.0927×10^{-2}	5.9477×10^{-2}	8.8129×10^{-3}	8.6856×10^{-3}
512	8.0249×10^{-2}	4.7495×10^{-1}	5.5524×10^{-2}	5.7673×10^{-2}
1024	6.1041×10^{-1}		3.6387×10^{-1}	4.6675×10^{-1}
2048	5.0741 x			3.8823
4096	39.6876			31.270
8192	308.7717			254.560
16384	2474.19			2082.62

En las versiones originales (original **C** y original CUDA 1 *thread*) sólo es necesario almacenar la matriz **A**, después de la transformación del código para las versiones en 2D, se incrementó la memoria necesaria al guardar las variables **c** y **s** que antes se guardaban en registros temporales. En la versión CUDA paralelo 2D versión 1, se requiere guardar las coordenadas (p_1, p_2), lo que ocasiona un incremento en memoria considerable en comparación con la versión 2 CUDA paralelo 2D, donde no se quiere guardar los arreglos p_1 y p_2 (ver tabla 4.6). Debido a que sólo se pudieron procesar matrices de hasta 1024 por el agotamiento de memoria en el *device*, se realizó la versión 2 CUDA paralelo 2D. En donde, se pueden procesar matrices de hasta 16384 y el consumo de memoria es de 3.2 GB de los 5.3 GB disponibles en el *device*. La versión CUDA paralelo 3D es la que tiene un gasto de memoria mayor, debido a que requiere almacenar 5 arreglos tridimensionales (de $M \times N \times N$) y las coordenadas (p_1, p_2).

Tabla 4.5: Resultados (en segundos) del algoritmo QR en CUDA 1 *thread*.

Tamaño	Original CUDA	Politopo destino CUDA
2	1.0869×10^{-5}	1.1718×10^{-5}
4	2.4898×10^{-5}	3.3462×10^{-5}
8	1.0905×10^{-4}	1.7003×10^{-4}
16	6.9291×10^{-4}	1.1324×10^{-3}
32	4.9775×10^{-3}	8.3797×10^{-3}
64	3.8257×10^{-2}	6.4577×10^{-2}
128	3.0037×10^{-1}	5.0748×10^{-1}
256	2.3789	4.0232
512	19.0429	32.7627
1024	153.2867	262.7216

Tabla 4.6: Consumo de memoria del *device* (en bytes) del algoritmo QR de *m x m*.

Tamaño	Original CUDA 1 *thread*	CUDA paralelo 3D	CUDA paralelo 2D versión 1	CUDA paralelo 2D versión 2
2	16	176	40	24
4	64	1,440	296	136
8	256	11,584	1,992	648
16	1,024	92,800	13,704	2,824
32	4,096	742,656	99,080	11,784
64	16,384	5,941,760	747,016	48,136
128	65,536	47,535,104	5,786,620	194,568
256	262,144	380,282,880	45,520,904	782,344
512	1,048,576	3,042,267,136	361,050,120	3,137,544
1024	4,194,304	24,338,145,280	2,875,887,608	12,566,536
2048	16,777,216	194,705,178,624	22,956,785,672	50,298,888
4096	67,108,864	1,557,641,461,760	183,453,188,104	211,261,064
8192	268,435,456	12,461,131,759,616	1,466,820,657,160	805,175,304
16384	1,073,741,824	99,689,054,208,000	11,731,344,949,256	3,220,963,336

4.4. Resumen del capítulo

En la primera parte del capítulo se desarrolla e implementa en CUDA el algoritmo suma de vectores. Se parte de la definición matemática, se analiza la dependencia de datos entre los vectores y se realiza una versión paralela en CUDA. El mayor aporte en esta sección es el de presentar una solución cuando se requiere manejar una cantidad muy grande de *threads*, se utilizan las 3 dimensiones de *blocks* para incrementar el rango de acceso lineal a la memoria del *device*, permitiendo procesar arreglos de $3{,}4461x10^{16}$ que para almacenarlos se requería $1{,}3784x10^{16}$ GB.

En la segunda parte se desarrolla e implementa el algoritmo de matriz transpuesta en CUDA. En donde se presentan diferentes soluciones utilizando paralelismo externo, interno y paralelismo en 2D. En donde nos podemos dar cuenta que es preferible utilizar el paralelismo externo en CUDA, para evitar el tiempo necesario para lanzar muchas funciones *kernel*.

Finalmente, se desarrolla el algoritmo de descomposición QR. En donde se utilizan los conceptos descritos en el capítulo 3 para la transformación de algoritmos con bucles anidados mediante transformaciones unimodulares. Se modifica la transformación que se fue realizada tesis [2], con lo cual se obtiene una mejora considerable al incrementar el tamaño de matriz (cuadrada) que se puede procesar de 512 a 16384. Tomando en cuenta el consumo, se redujo el costo en memoria, de necesitar 3 GB se redujo a 3 MB para matrices de 512. Si comparamos el consumo en memoria para matrices de 16384, nuestra versión necesita 3.2 GB, mientras que la versión en [2] requiere 99, 689 GB lo cual hace imposible su ejecución en los GPU actuales.

Para comparar el desempeño de las diferentes versiones de los algoritmos, se obtuvo el tiempo promedio de ejecución, el *speedup* y el gasto de memoria. Con el tiempo de ejecución nos podemos dar cuenta que tanto tiempo nos ahorramos al realizar el cómputo de algoritmos en el GPU, al compararlo con el tiempo de ejecución en el CPU. Con el *speedup* obtenemos cuantas veces es más rápido nuestro algoritmo paralelo frente a su versión secuencial y con el gasto de memoria podemos deducir el tamaño máximo del problema que podemos resolver en un determinado GPU.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo, se presentan aspectos teóricos para el manejo de GPU's, explicando sus características, manejo de procesadores, y los diferentes tipos de memoria.

Se realizó la implementación en CUDA C del algoritmo que realiza la suma de vectores. Proporcionándose una alternativa para mapear las tres dimensiones de los *blocks* y de los *threads* a una dimensión, logrando un mayor rango de acceso lineal en los arreglos. Esto es útil cuando nuestro problema procesa un alto volumen de datos y requiere manejar numerosos *threads* en una dimensión.

Además, se implementó el algoritmo de matriz transpuesta en CUDA C. Al comparar los resultados obtenidos de las diferentes versiones paralelas de este algoritmo, se comprobó que en los GPU's es mejor procurar implementar un paralelismo externo en lugar de un paralelismo interno. Debido a que se emplea un tiempo en dimensionar y crear el *grid* cada vez que se lanza una función *kernel*, se encontró, que es mejor lanzar la menor cantidad de *kernels* que el algoritmo permita, utilizando la mayor cantidad de *threads* que el algoritmo requiera.

Con las adaptaciones a la metodología de paralelización de bucles anidados, se obtuvo una versión de la descomposición QR, en donde se redujo la memoria requerida, y los tiempos de ejecución resultantes.

Finalmente, se observó que para problemas donde el volumen de datos es pequeño, es posible que el CPU ejecute más rápido un algoritmo que un GPU (aunque se ejecute una versión paralela). Existe un punto de cruce en donde el GPU supera los tiempos de ejecución del CPU, dicho punto de cruce será en general diferente para cada algoritmo. Por lo tanto, el uso de un GPU representa grandes reducciones en el tiempo de ejecución de nuestros algoritmos, a medida que crece el volumen de datos a procesar y cuando se realiza una buena paralelización.

5.2. Trabajo futuro

De esta investigación se propone el siguiente trabajo futuro:

Es necesario profundizar más en el estudio del paralelismo externo ya que presentó mejor rendimiento en el GPU que el paralelismo interno.

- Analizar y adaptar las técnicas de embaldosado (*loop tilling* o *loop blocking*) para su utilización en CUDA C, esto es útil cuando el algoritmo requiera más recursos de memoria y/o procesamiento que con los que cuenta físicamente el GPU empleado.
- Estudiar y analizar los conceptos de localidad temporal y espacial de los datos, para lograr un uso más eficiente de las memorias caché de los GPU e introducirlos a la transformación de los algoritmos.
- Estudiar las arquitecturas más novedosas de los GPU's, las cuales poseen multiprocesadores mejorados, permiten realizar paralelismo dinámico, permiten realizar un mejor uso de los recursos de la tarjeta GPU (Hyper-Q) y utilizar el nuevo juego de instrucciones.
- Estudiar las técnicas para manejar los múltiples cores de los CPU's actuales; por medio de threads, lo cual permitirá manejar un GPU o más (formar un *cluster* de GPU's) de manera concurrente.

Bibliografía

[1] W. Hwu W. Mei. *GPU Computing Gems, Jade Edition.* CINVESTAV GDL, 2012.

[2] R. Gómez. *Transformación del algoritmo matricial que efectúa la descomposición QR de su forma secuencial a su forma paralela.* CINVESTAV GDL, Dec 2011.

[3] J. Flores. *Arquitectura con paralelismo para el cálculo de valores singulares de una matriz.* CINVESTAV GDL, Dec 2009.

[4] M. Martínez. *Operaciones polinomiales con técnicas de paralelización en arreglos de procesadores.* CINVESTAV GDL, Oct 2012.

[5] W. Hwu D. Kirk. *Programming Massively Parallel Processors, A Hands-on Approach.* Elsevier Inc., 2011.

[6] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide 4.2.* 2012.

[7] E Kandrot. J. Sanders. *CUDA by Example, An Introduction to General-Purpose GPU Programming.* Elsevier Inc.-Morgan Kaufmann, 2011.

[8] NVIDIA Corporation. *CUDA C Best Practices Guide, Design Guide.* 2012.

[9] NVIDIA Corporation. *CUDA API Reference Manual 5.0.* 2012.

[10] R. Farber. *CUDA , Application Design and Development.* Elsevier Inc.-Morgan Kaufmann, 2011.

[11] S. Cook. *CUDA Programming, A Developer's Guide to Parallel Computing with GPUs.* Elsevier Inc.-Morgan Kaufmann, 2013.

[12] T. Braunl. *Parallel Programming, an Introduction.* Prentice Hall, 1993.

[13] N. Wilt. *CUDA Handbook, A Comprehensive Guide to GPU Programming.* Addison-Wesley, 2013.

[14] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1994.

[15] C. Lengauer. Loop parallelization in the polytope model. *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416, 1993.

[16] M. Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley, 1996.

[17] M. Lam M. Wolf. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on (Volume:2 , Issue: 4)*, 2, Oct 1991.

[18] U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer, jun 1994.

[19] D. Padua. *Encyclopedia of Parallel Computing.* Springer, Dec 2011.

[20] D. Moldovan. *Parallel Processing, From Applicaions to Systems.* Morgan Kaufmann, 1993.

[21] C. Van Loan G. Golub. *Matrix Computations.* The Johns Hopkins University Press, 1996.

Apéndice A

Programas CUDA C

A.1. Programa Ejemplo, suma de vectores de 5 elementos

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__host__ void addWithCuda(int *c, const int *a, const int *b, size_t size);

// Función kernel, ejecutada en el device
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i   threadIdx.x;
    c[i]   a[i]   b[i];
}

// Función principal
int main()
{
    // Generación de datos en el host
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Función auxiliar del host para sumar los vectores en paralelo.
    addWithCuda(c, a, b, arraySize);

    // Impresión de resultados
    printf("{1,2,3,4,5} + {10,20,30,40,50}   {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    return 0;
}

// Función auxiliar del host para sumar los vectores en paralelo.
__host__ void addWithCuda(int *c, const int *a, const int *b, size_t size)
{
    // Arreglos de memoria del device
    int *dev_a = 0;         int *dev_b   0;    int *dev_c = 0;

    // Asignación de memoria en el device
    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size   sizeof(int));
    cudaMalloc((void**)&dev_b, size   sizeof(int));

    // Envío de los datos de entrada hacia el device.
    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    // Dimensionamiento del grid para la función kernel
    dim3 gridDim;
    // Dimensionamiento de blocks
    gridDim.x   1; gridDim.y = 1; gridDim.z   1;  // Un bloque
    dim3 blockDim;
    // Dimensionamiento de Threads
    blockDim.x   size; blockDim.y = 1; blockDim.z   1; // 5 hilos en el eje x por bloque

    // Lanzamiento del kernel con un thread para cada elemento del arreglo.
    addKernel<<<gridDim, blockDim>>>(dev_c, dev_a, dev_b);
```

```
    // Esperar que el kernel termine de ejecutarse totalmente
    cudaDeviceSynchronize();

    // Copia del arreglo procesado hacia el host.
    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

    //Liberación de la memoria del device
    cudaFree(dev_c);    cudaFree(dev_a);    cudaFree(dev_b);

    return;
}
```

A.2. Ejemplo de manejo de memoria Texture 3D

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#include <math.h>

using namespace std;

//declare texture reference
texture<float,cudaTextureType3D,cudaReadModeElementType> textReference;

// kernel function
__global__ void kernel(float *device_CubeMatrix, int size)
{
	int xIndex;
	int yIndex;
	int zIndex;

	//calculate each thread global index
	xIndex = threadIdx.x + blockIdx.x * blockDim.x;
	yIndex = threadIdx.y + blockIdx.y * blockDim.y;
	zIndex = threadIdx.z + blockIdx.z   blockDim.z;

	device_CubeMatrix[zIndex *size *size + yIndex *size + xIndex]
	tex3D(textReference,xIndex,yIndex,zIndex) + 1;

	return;
}

int main()
{
	float *host_CubeMatrix;
	float *device_CubeMatrix;

	cudaArray *cudaArray;
	cudaExtent volumeSize;
	cudaChannelFormatDesc channel;
	cudaChannelFormatDesc channel2;

	cudaMemcpy3DParms copyparms={0};

	int size   2;

	//allocate host and device memory
	host_CubeMatrix = (float*)malloc(sizeof(float)*size*size*size);
	cudaMalloc((void**)&device_CubeMatrix,sizeof(float)*size*size*size);

	//initialize host matrix before usage
	for(int loop=0; loop<size*size*size;loop++)
		host_CubeMatrix[loop]   (float)rand()/(float)(RAND_MAX-1);

	cout<<"matriz h : "<<endl;
	for(int i   0;i<size*size*size;i++)
	{
		cout<<host_CubeMatrix[i]<<endl;
	}

	//set cuda array volume size
```

```
volumeSize   make_cudaExtent(size,size,size);

//create channel to describe data type
channel = cudaCreateChannelDesc<float>();
channel2 = cudaCreateChannelDesc<float>();

//allocate device memory for cuda array
cudaMalloc3DArray(&cudaArray,&channel,volumeSize);

//set cuda array copy parameters
copyparms.extent = volumeSize;
copyparms.dstArray = cudaArray;
copyparms.kind   cudaMemcpyHostToDevice;

// 3D copy from host_CubeMatrix to cudaArray
copyparms.srcPtr = …
make_cudaPitchedPtr((void*)host_CubeMatrix,sizeof(float)*size,size,size);
cudaMemcpy3D(&copyparms);

//set texture filter mode property
//use cudaFilterModePoint of cudaFilterModeLinear
textReference.filterMode   cudaFilterModePoint;

//set texture address mode property
//use cudaAddressModeClamp or cudaAddressModeWrap for integer coordinates
textReference.addressMode[0]   cudaAddressModeClamp;
textReference.addressMode[1]   cudaAddressModeClamp;
textReference.addressMode[2]   cudaAddressModeClamp;
//bind texture reference with cuda array
cudaBindTextureToArray(textReference,cudaArray, channel);

// preparing kernel launch
dim3 blockDim; dim3 gridDim;
blockDim.x   size;   blockDim.y = size;   blockDim.z = size;
gridDim.x    1;     gridDim.y  = 1;     gridDim.z    1;

//execute device kernel
kernel<<< gridDim   blockDim >>>( device_CubeMatrix, size );

//unbind texture reference to free resource
cudaUnbindTexture(textReference);

//copy result matrix from device to host memory
cudaMemcpy(host_CubeMatrix, device_CubeMatrix, …
sizeof(float)*size*size*size, cudaMemcpyDeviceToHost);

cout<<"matriz d : "<<endl;
for(int i   0; i<size*size*size; i++)
{
       cout<<host_CubeMatrix[i]<<endl;
}

//free host and device memory
free(host_CubeMatrix);
cudaFree(device_CubeMatrix);
cudaFreeArray(cudaArray);

system("pause");
```

A.3. Ejemplo de manejo de memoria Surface 2D

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#include <math.h>


using namespace std;

// 2D surfaces
surface<void, cudaSurfaceType2D> inputSurfRef;
surface<void, cudaSurfaceType2D> outputSurfRef;

// kernel:  copy and increment by one
__global__ void copyKernel(int width, int height)
{
	// Calculate surface coordinates
	unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
	unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
	if (x < width && y < height)
	{
		float data;
		// Read from input surface
		surf2Dread(&data, inputSurfRef, x   4, y);
		// Write to output surface
		data   data + 1;
		surf2Dwrite(data, outputSurfRef, x * 4, y);
	}
}


// Host code
int main()
{
	int width   3;
	int height = 3;
	int size = sizeof(float)*width*height;

	//allocate host and device memory
	float *h_data;
	float *h_data_out;
	h_data     = (float*)malloc(sizeof(float)*height*width);
	h_data_out = (float*)malloc(sizeof(float)*height*width);

	//initialize host matrix before usage
	for(int loop=0; loop<width*height;loop++)
		h_data[loop] = (float)rand()/(float)(RAND_MAX-1);

	cout<<"datos entrada : "<<endl<<endl;
	for(int i   0;i<width*height;i++)
	{
		cout<<h_data[i]<<endl;
	}

	// Allocate CUDA arrays in device memory
	cudaChannelFormatDesc channelDesc;
	channelDesc = cudaCreateChannelDesc<float>();
	cudaArray* cuInputArray;     cudaArray* cuOutputArray;
```

```
        cudaMallocArray(&cuInputArray, &channelDesc, width, height, …
        cudaArraySurfaceLoadStore);
        cudaMallocArray(&cuOutputArray, &channelDesc, width, height, …
        cudaArraySurfaceLoadStore);

        // Copy to device memory some data located at address h_data in host memory
        cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);

        // Bind the arrays to the surface references
        cudaBindSurfaceToArray(inputSurfRef, cuInputArray);
        cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);

        // Invoke kernel
        dim3 dimBlock(3, 3, 1);
        dim3 dimGrid(1,1,1);
        copyKernel<<<dimGrid, dimBlock>>>(width, height);

        // Copy to host memory some data located at address outputSurfRef in device memory
        cudaMemcpyFromArray(h_data_out,cuOutputArray,0,0   size, cudaMemcpyDeviceToHost);

        // Display
        cout<<endl<<"datos de salida : "<<endl<<endl;
        for(int i   0;i<width*height;i++)
        {
              cout<<h_data_out[i]<<endl;
        }
        // Free device memory
        free(h_data);
        cudaFreeArray(cuInputArray);
        cudaFreeArray(cuOutputArray);

        system("pause");
        return 0;
}
```

Glosario

Término	Descripción	Página
Ancho de banda	Cantidad de datos que se pueden leer o almacenar en la DRAM en un periodo dado de tiempo.	17
Asignación única	Transformación para que a cada variable, en el nido de bucles, se le asigne valor una sola vez.	68
Block	Conjunto de *threads*; puede estar organizado unidimiensional, bidimensional o tridimensionalmente.	6
BlockDim	Variable con 3 campos para dimensionar el número de *threads* en las tres dimensiones del *block* (blockDim.x, blockDim.y, blockDim.z).	7
BlockIdx	Identificador tridimensional de los *blocks* que permite su localización en el grid.	6
Caché	Banco de memoria de alta velocidad que se encuentra físicamente cercano al núcleo de procesamiento.	18
Compute capability	Agrupación de las arquitecturas CUDA por las características de los núcleos SP, consta de dos números, el de mayor y menor revisión.	12
Concordancia de índices	Transformación en la que se mapea cada una de las asignaciones presentes en el código al espacio de índices correspondiente al nido de bucles.	68
CUDA Array	Arreglos optimizados para su lectura en una, dos o tres dimensiones, ligados a una referencia *texture* o *surface*.	19
Device	Unidad de procesamiento gráfico (GPU) y su memoria.	5
Eliminación de comunicación global	Transformación que evita la difusión global de datos y la sustituye por múltiples comunicaciones punto a punto.	69
Espacio de iteración	Contiene un punto en el espacio m dimensional, por cada iteración de un nido de m bucles.	32

Término	Descripción	Página
Evento	Estampa de tiempo del GPU que se registra en un punto, en el tiempo, especificado por el usuario.	11
Grafo de dependencia	Grafo dirigido que plasma con flechas, en el espacio de iteración y las dependencias de datos.	34
Grid	Conjunto de *blocks*; puede estar organizado unidimiensional, bidimensional o tridimensionalmente. Se crea en un lanzamiento de kernel	6
GridDim	Variable con 3 campos para dimensionar el número de *blocks* en las tres dimensiones del grid (gridDim.x, gridDim.y, gridDim.z).	7
Host	Unidad central de procesamiento (CPU) y su memoria.	5
Hundimiento de código	Transformación en la que se busca tener un nido de bucles perfecto.	67
Kernel	Función que se ejecuta por el GPU.	5
Lanzamiento de Kernel	Ejecución de una función en el device con el calificador __global__.	6
Latencia	Cantidad de tiempo que toma en realizar la petición de búsqueda (*fetch request*) de alguna localidad de memoria.	17
Nido de bucles	Conjunto de bucles en el que un bucle contiene uno o más bucles del tipo *for*.	31
Nido de bucles perfecto	Bucle anidado en el que no existen sentencias entre un bucle y otro.	31
Normalización	Transformación para que todos los bucles tengan incrementos positivos.	67
Nvcc	Compilador de C de NVIDIA.	5
Paralelismo	Propiedad de un programa, mediante el cual, muchas operaciones aritméticas se pueden realizar de manera segura y de manera simultánea en las estructuras de datos.	6
Paralelismo externo	Tipo de paralelismo en el que los más bucles externos se ejecutan en paralelo.	42
Paralelismo interno	Tipo de paralelismo en el que los más bucles internos se ejecutan en paralelo.	42
Politopo	Conjunto acotado de la forma $\{\mathbf{x} \in \mathbb{R}^m : \mathbf{xA} \leq \mathbf{c}\}$ para alguna matriz real $\mathbf{A}$ y un vector real $\mathbf{c}$.	32
Politopo fuente	Politopo asociado al programa original.	32
Politopo destino	Politopo asociado al programa transformado.	35
Speedup	Relación del tiempo de ejecución secuencial en un procesador, con el tiempo de ejecución paralela con p procesadores.	51

Término	Descripción	Página
Stream	Secuencia de comandos que se ejecutan en orden.	25
Streaming Multiprocessor (MP)	Conjunto de SP's (Streaming Processor), encargado de la ejecución de los warps.	13
Streaming Processor (SP)	Núcleo capaz de realizar operaciones aritméticas y lógicas en paralelo.	13
ThreadIdx	Identificador tridimensional de los *threads* que permite su localización en los *blocks*.	6
Threads	Instancia del kernel que se ejecuta en los SP's.	6
Warp	Grupo de 32 Threads. Es la unidad mínima de creación, administración, planificación y ejecución en un SM (Streaming Multiprocessor).	13

Acrónimos

ALU	Arithmetic Logic Unit
DRAM	Dynamic Random-Access memory
GDDR	Graphic Double Data Rate
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
NVCC	NVIDIA C Compiler
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
SP	Streaming Processor
SPU	Special-Purpose Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit

Lista de figuras

Lista de tablas

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Adaptación de la metodología de paralelización de bucles en el modelo del politopo para la implementación de algoritmos en GPUs

del (la) C.

Daniel ROBLES VALDEZ

el día 16 de Diciembre de 2013.

Dr. Yuriy Shkvarko Sosnoff
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara

Dr. Deni Librado Torres Román
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara

Dr. Ramón Parra Michel
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara