Flaviu Vasile Buturca

Programación paralela en CUDA

TRABAJO FINAL DE CARRERA

Dirigido por Carles Aliagas Castell

Grado de Ingeniería Informática

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2016

UNIVERSIDAD ROVIRA I VIRGILI

Resumen

Escuela Técnica Superior de Ingeniería

Trabajo Final de Carrera

Programación paralela en CUDA

por Flaviu Vasile Buturca

En la actualidad, las unidades de procesamiento gráfico (GPU) tienen un alto rendimiento en cuanto a operaciones computacionales en paralelo. Por ello, en este proyecto se desarrolla y estudia el comportamiento de varios algoritmos programados para la GPU.

Los algoritmos son utilizados con frecuencia en diferentes ámbitos, por lo que es importante estudiar diferentes alternativas para mejor el rendimiento de éstos, siendo la utilización de la GPU una de ellas.

El estudio se realiza sobre un total de 6 algoritmos que presentan un alto nivel de paralelización, haciéndolos buenos candidatos para su ejecución en la GPU. También se comentan detalles y mejoras en rendimiento de diferentes optimizaciones realizadas a lo largo del estudio.

Agradecimientos

Quiero agradecer a mi director del proyecto Carles Aliagas Castell por la idea de realizar el proyecto sobre éste tema, siendo un campo de la informática que no conocía mucho. También quiero agradecerle por resolverme todas las dudas que han surgido durante la realización de este trabajo.

También quiero dar las gracias a Xavier Conesa por ayudarme en pasar los esquemas que se utilizan en las explicaciones, de papel a formato digital.

Índice

Índice de figuras

Capítulo 1

Introducción

El proyecto tiene como objetivo general el estudio y desarrollo de varios algoritmo populares programados para aprovechar el gran nivel de paralelización que se ofrece en las unidades de procesamiento gráfico (GPU) actuales. Se ha utilizado la plataforma CUDA ofrecida por NVIDIA para llevar a cabo el proyecto.

El estudio se realiza sobre un total de 6 algoritmos diferentes que presentan un alto nivel de paralelización, haciéndolos buenos candidatos para su ejecución en la GPU. Los algoritmos son:

- Quicksort
- Mergesort
- K-means
- N-Body
- Floyd-Warshall
- Monte Carlo

Además de la implementación de los algoritmos, este proyecto también se centra en la importancia de tener en cuenta todas las características de una GPU, documentando todos los cambios realizados relacionados al funcionamiento de éstas para mejorar el rendimiento de cada uno de los algoritmos.

Algunas de las implementaciones de los algoritmos realizados necesitan GPUs con una versión del compute capability mínima de 3.5 ya que se aprovechan algunas de las funcionalidades que se han introducido a partir de tal versión. La versión del compute

capability describe las características y funciones que una cierta GPU ofrece al programador.

Las pruebas se realizaron en la máquina Roquer proporcionada por la universidad, con las siguientes características:

- Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz
- NVIDIA GeForce GTX 980
- Cuda 7.0

A continuación se hará una introducción a la arquitectura de las GPU y a la plataforma CUDA utilizada para la programación de éstas. Después se explicará, uno por uno, la implementación y los resultados de los algoritmos anteriormente mencionados.

Capítulo 2

Unidad de procesamiento gráfico (GPU)

2.1. Arquitectura

La CPU tiene un propósito más general, diseñada para el procesamiento en serie, ya que se compone de unos pocos núcleos muy complejos que realizan cálculos de forma secuencial.

La GPU, al contrario a la CPU, está optimizada para trabajar con grandes cantidades de datos sobre los cuales se realizan las mismas operaciones de forma concurrente. Dispone de una gran cantidad de núcleos sencillos que se pueden ejecutar en paralelo para resolver problemas de una gran variedad de ámbitos como la inteligencia artificial o la minería de datos.

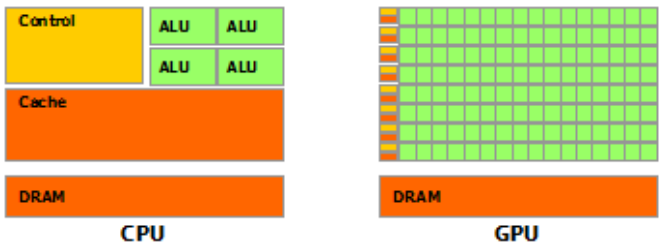


FIGURA 2.1: Comparación entre la arquitectura de una CPU y una GPU.

La ejecución de un programa altamente paralelizable en la GPU puede aumentar enormemente su rendimiento en comparación con la CPU. Éstos programas normalmente realizan las mismas operaciones sobre una cantidad de datos muy elevada sin muchas dependencias entre ellos.

Para llevar a cabo este paralelismo, la arquitectura de una GPU se compone de dos componentes principales, la memoria global y de una serie de unidades de procesamiento denominadas streaming multiprocessors (SM) o multiprocesadores.

La memoria global es el punto de comunicación de datos entre la CPU y la GPU, donde la CPU se encarga de realizar todas trasferencia de datos entre la memoria utilizada por ella misma y la memoria global de la GPU para que puede procesas los datos. La GPU sobre la que se ha realizado los estudios dispone de 4 GB de memoria global.

Aparte de la memoria global, también se dispone de otros tipos de memoria, entre los cuales se encuentra los diferentes niveles de caches o la memoria compartida. Éstas memorias son mucho más rápidas que la memoria global y, en el caso de la memoria compartida, está en manos del programador para aprovechar de las ventajas que proporciona, ya que se proporciona el control total sobre ella, siendo un punto muy importante a la hora implementar un algoritmo.

El segundo componente principal son los multiprocesadores, éstos pueden venir en cantidades variables (16 en el caso de la GPU utilizada), y son los encargados de realizar toda la computación. En cada uno de ellos contienen sus propios registros, unidades de control, cachés y es donde reside la memoria compartida, de ahí su gran rendimiento.

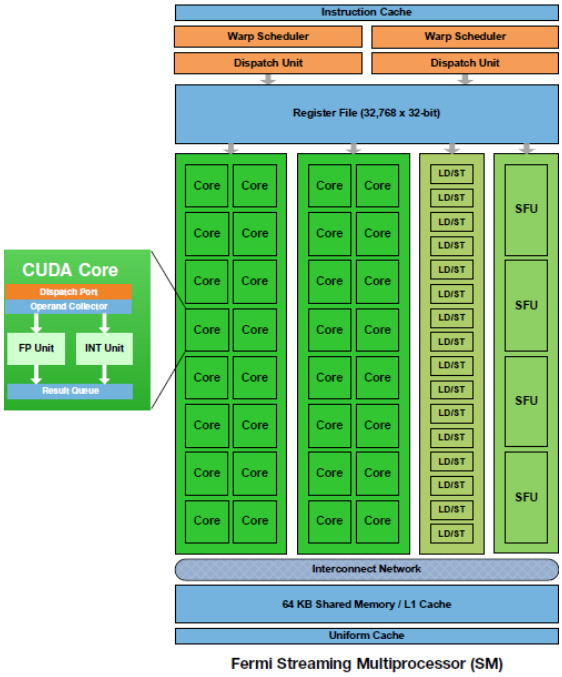


FIGURA 2.2: Arquitectura de un multiprocesador Fermi.

Las instrucciones se ejecutan en los llamados CUDA cores (núcles), residentes en los multiprocesadores. La cantidad de éstos núcleos varía dependiendo de la generación de la arquitectura.

2.2. CUDA

CUDA es una plataforma de cálculo paralelo creada por NVIDIA que proporciona una cuantas extensiones de C, C++ y Fortran que permiten aprovechar la potencia de una GPU. Ésto hace más fácil la utilización de los recursos que oferece una GPU a los desarrolladores.

2.2.1. Modelo de programación

En ésta sección se introduce los conceptos básicos detrás del modelo de programación.

2.2.1.1. Kernels

Los kernels son las funciones de una aplicación, implementadas por el desarrollador, que se ejecutan en la GPU. Cada kernel es ejecutado por una multitud de hilos en paralelo, por lo que cada hilo ejecuta el mismo código, pero se pueden tomar divergencias.

A cada hilo se le asigna un identificador diferente para que se puedan diferenciar dentro de un mismo kernel.

La CPU es la encargada de comunicar a la GPU que kernels debe ejecutar. Éste proceso es asíncrono, por lo que, una vez inicializado un kernel en la GPU, la CPU puede seguir realizando operación concurrentemente, al menos que se especifique una barrera.

El lanzamiento asíncrono de kernels abre la posibilidad de ejecutar más de uno. El orden de ejecución depende de stream que se la ha asignado a cada uno. Los kernels dentro de un mismo stream se ejecutan en serie, en cambio, si se encuentran en streams distintos, se ejecutarán de forma concurrente siempre y cuando sea posible.

Las GPUs con una versión del compute capability mayor a 3.5 disponen de una funcionalidad llamada dynamic parallelism, que ofrece la posibilidad de ejecutar un kernel dentro de otro.

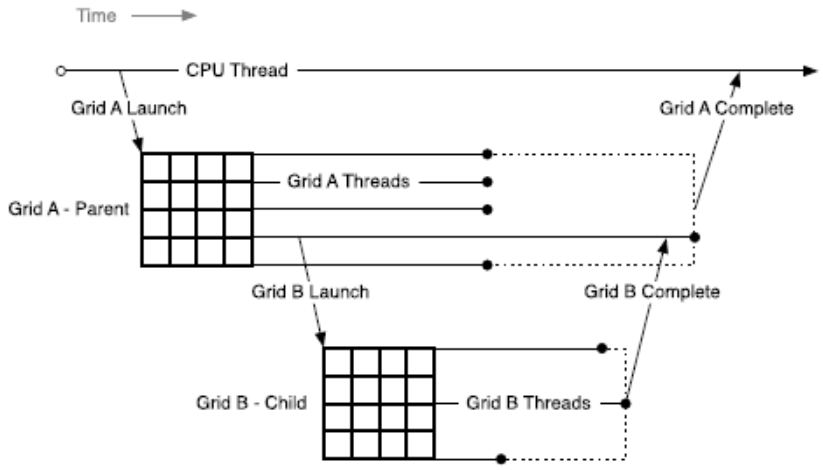


FIGURA 2.3: La ejecución de un kernel dentro de otro (dynamic parallelism).

2.2.1.2. Jerarquía de los hilos

Los hilos se agrupan en bloques, que a su vez, se agrupan en mallas (*grids*). A cada kernel se le asigna una malla, y es lo que indica el grado de paralelización con el que éste kernel se debe ejecutar.

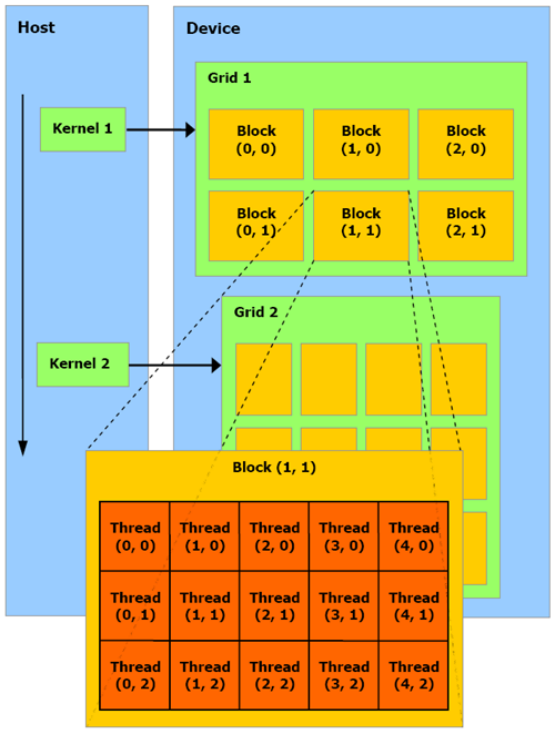


FIGURA 2.4: Asignación de grid a kernels y la jerarquía de éstos.

Las agrupaciones de hilos en bloques se pueden realizar en una, dos o tres dimensiones. Los bloques se agrupan en mallas. Ésto facilita la programación de kernels, ya que hace

posible el acceso de datos estructurados en una lista, matriz o volumen de forma más natural.

Hay un número límite de hilos por bloque, ya que, todos los hilos del bloque se ejecuta en un multiplocesador, compartiendo los recursos disponibles. También hay un número límite de bloques por multiprocesador que viene dado por:

- El número de registros utilizados por los hilos
- La memoria compartida total asignada a cada bloque
- El número total de hilos que se pueden ejecutar
- Un límite máximo de bloques

La memoria compartida puede ser utilizada para la comunicación entre los hilos de un bloque.

Durante la ejecución, cada bloque es agrupado en un conjunto de 32 hilos denominados warps, que comparten el mismo contador de programa, por lo que ejecutan la misma instrucción. Ésto conlleva ciertas limitaciones que hay que tener en cuenta en cuanto a la programación de un kernel, ya que, de no ser así, el rendimiento puede empeorar.

Una de las limitaciones viene dada por el acceso a memoria global, ya que las peticiones se realizan a nivel de warp, accediendo 32 elementos coalescentes. Si los datos no se encuentran seguidos en memoria, se deben realizar varias transferencias, empeorando el rendimiento.

Otra limitación viene dada en el caso de una divergencia, que ocurre cuanto hilos de dentro de un mismo warp toman un camino distinto. El problema se que un warp comparte el contador del programa, por lo que, todos los hilos deben seguir el mismo camino, teniendo que desactivar los hilos que tienen un camino distinto.

Capítulo 3

Quicksort

3.1. Introducción

La ordenación de elementos se utiliza en una gran variedad de aplicaciones, tanto pequeñas como grande, siendo una tarea muy importante y puede resultar crítica en los casos en que se opera sobre un elevado número de elementos. Por ello, se han desarrollado una alta variedad de algoritmos diferentes para resolver el problema de una forma rápida y eficiente, siendo Quicksort uno de ellos.

Quicksort es un algoritmo eficiente de ordenación basado en la técnica de divide y vencerás, creado por Tony Hoare en 1959 y publicado en 1961[1]. La idea básica de este tipo de algoritmos consiste en ir dividiendo la lista de elementos en sublistas más pequeñas y operando sobre éstas para reducir la complejidad.

Se utilizará el algoritmo de ordenación bitónica (Bitonic sort) cuando las sublistas resultantes sean menores a un determinado límite para mejorar el rendimiento final, tal y como Hoare sugiere en su documento.

La ordenación bitónica es un algoritmo paralelo diseñado por Ken Barcher[2], muy eficiente en cuanto a pequeñas cantidades de elementos y gran número de procesadores.

Éste algoritmo consta de dos partes. La primera se encarga de transformar la lista en una secuencia bitónica y la segunda se encarga de obtener la lista final ordenada a partir de ésta secuencia bitónica. El proceso está descrito en más detalle en la sección 3.2.3.

3.2. Diseño

En el algoritmo se realizan los siguientes pasos:

1. Elegir un elemento de la lista al que se llama pivote.

2. Dividir la lista en dos sublistas con todos los elementos más pequeños que el pivote en una sublista y todos los elementos más grandes que el pivote en una segunda sublista. Una vez reordenados todos los elementos se coloca el pivote entre las dos sublistas resultantes.

3. Se repite el procedimiento de forma recursiva sobre cada una de las sublistas resultantes que tengan una longitud mayor que uno.

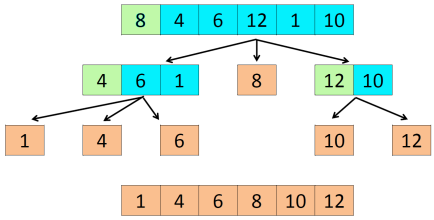


FIGURA 3.1: Simple ejemplo del funcionamiento del Quicksort

Tal y como se ha mencionado, el algoritmo se realiza de forma recursiva, pero en la GPU, este proceso esta limitado a una cierta profundidad de recursividad, por lo que el algoritmo se dividirá en diferentes fases explicadas más adelante. La implementación recursiva se realiza solo para listas de tamaño reducido y una implementación iterativa para listas de tamaño mayor.

La necesidad de crear múltiplos bloques para tener una alta ocupación de la GPU y mejorar el rendimiento en casos de listas grandes requiere de sincronización entre éstos bloques, resultando en una carga adicional en casos de tener un único bloque realizando la ordenación. De forma parecida, en las iteraciones finales, cuando el número de elementos a ordenar es más pequeño, se necesita de muchos movimientos de pequeñas cantidades de éstos en memoria, degradando el rendimiento.

Debido a los casos mencionados anteriormente, se divide el algoritmo en 3 fases distintas dependiendo del número de elementos a procesar. Las primeras 2 fases son implementaciones del Quicksort muy parecidas, se diferencían en el hecho de que en la primera fase se realiza de forma iterativa y necesita sincronización entre bloques y la segunda fase se realiza de forma recursiva y prescinde de tal sincronización. Finalmente, la tercera fase es realizada por algoritmo de ordenación bitónica explicada en la sección 3.2.3.

La utilización de una u otra fase depende de la cantidad de segmentos en que se divide una lista. Un segmento es la cantidad de elementos que un bloque puede procesar, limitado por la cantidad de memoria compartida que éste puede utilizar. De este modo, si una lista dispone de 2 o más segmentos, se utilizará la fase 1 del algoritmo, ya que se

deberá utilizar más de un bloque, obligando la sincronización entre éstos. Si se dispone de un solo segmento, se procesará la segunda fase del algoritmo, y finalmente, si la lista es menor que un determinado límite (véase sección 3.2.3), se utilizará el algoritmo de la ordenación bitónica.

De forma iterativa se asigna una de las 3 fases a cada una de las sublistas resultantes de la iteración anterior dependiendo de los criterios mencionados, hasta que todos los elementos de la lista queden ordenados y el algoritmo acaba.

La sincronización necesaria en la fase 1 se realizará en la memoria global de la GPU utilizando las funciones atómicas que se facilitan el lenguaje CUDA.

En cada iteración se elige un pivote nuevo para cada una de las sublistas resultantes de la iteración anterior. El pivote elegido siempre es el primer elemento de dicha sublista, pero diferentes métodos se pueden utilizar también.

A continuación se explicará con más detalle cada una de las fases por separado.

3.2.1. Fase 1

Inicialmente, la lista se divide en m segmentos de igual longitud ((A) figura 3.2) y a cada bloque se le asigna un segmento ((B) figura 3.2).

Una vez cada bloque sabe que segmento tiene que procesar, los hilos del bloque cuentan la cantidad de números menores/mayores que el pivote que encuentran en tal segmento ((C) figura 3.2). Esto es necesario ya que cada hilo necesita saber la cantidad de elementos que el hilo con identificador menor que él encontró para saber en que posición de las sublistas resultantes se tiene que almacenar cada elemento encontrado.

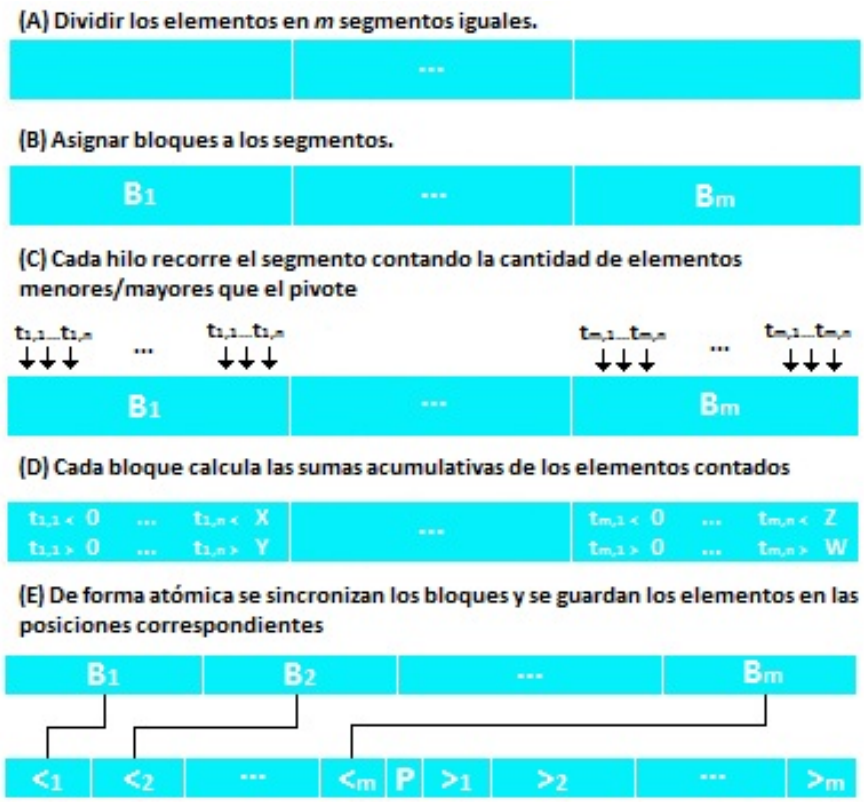


FIGURA 3.2: Pasos realizados en la fase 1 del Quicksort

Pero simplemente contando estos valores no bastan, ya que cada hilo tiene su valor, sin saber los valores de los demás, para ello se realiza una suma acumulativa entre todos los hilos del bloque ((D) figura 3.2). Calculando la suma acumulativa entre los hilos de un bloque se puede saber cuantos elementos se van a procesar por hilos con identificadores menores, pudiendo calcular la posición de la sublista en la que dicho hilo puede almacenar los elementos que procesa.

El algoritmo de la suma acumulativa esta basado en el presentado por Hillis y Steele en 1985[3]. De implementa y estudia dos formas distintas de dicho algoritmo, uno utilizando memoria compartida y otro utilizando las funciones de warps existentes en CUDA. En el apartado de implementación se explica con más detalle éstas dos implementaciones.

Una nueva sincronización se debe realizar, esta vez entre bloques, ya que cada bloque debe saber en que posición guardar sus valores ((E) figura 3.2). La comunicación entre bloques es difícil, ya que no se puede saber el orden en el que éstos se van a ejecutar, para ello, se puede realizar de dos modos. En el primero se separa el kernel en dos kernels distintos, cuando un kernel acaba, se sabe que cada bloque acabó su trabajo anterior y se puede ejecutar un segundo kernel en el que los nuevos bloques están sincronizados, pero esta implementación es poco eficiente. El segundo, y el implementado, trata de utilizar

las primitivas atómicas que el lenguaje CUDA ofrece. Para ello, cada bloque actualiza un contador atómico con la cantidad de elementos menores/mayores que encuentra, guardando su valor antiguo. Este valor antiguo representa el desplazamiento que debe utilizar dicho bloque. También queda preparado el contador para el siguiente bloque.

Por ejemplo, asumiremos que se tienen dos bloques que tienen que escribir 100 y 150 elementos respectivamente y el bloque con 100 elementos es el primero en actualizar los contadores. Al actualizar el contador, el valor inicial es 0 y se incrementa en 100. Ahora el primer bloque sabe que su desplazamiento es 0 y escribe 100 elementos. Cuando el siguiente bloque actualiza el contador, este devuelve el valor 100, por lo que sabe que debe escribir sus valores iniciando en la posición 100.

Finalmente, cuando el bloque sabe su desplazamiento inicial y cada hilo sabe su desplazamiento en el bloque, se vuelve a recorrer el segmento, pero esta vez se pueden mover los valores a sus respectivas posiciones. Debido a que ciertos bloques pueden escribir en posiciones de la lista que no se han procesado aun, es necesario utilizar un búfer doble.

3.2.2. Fase 2

Esta segunda fase es prácticamente igual a la primera fase, con las diferencias de que se ejecuta únicamente sobre un segmento, necesitando un solo bloque para su procesamiento, eliminando la parte de sincronización. De este modo, una vez calculado la suma acumulativa, se puede escribir directamente los elemento en las nuevas sublistas.

Una vez creadas las dos nuevas sublistas, en vez de devolver el control a la CPU para iniciar la siguiente iteración sobre éstas, el propio kernel lanza 2 nuevos kernels para tal fin, gracias a la funcionalidad de dynamic parallelism disponible en las nuevas GPUs. La recursión sigue hasta que uno de los siguientes criterios se cumple:

1. Si una de las sublistas resultantes tiene una longitud igual o inferior especificada por el límite de la ordenación bitónica, se realiza la ordenación con tal algoritmo.
2. Si las llamadas recursivas llegan a la profundidad máxima y la sublista tiene una longitud superior a la de la ordenación bitónica, una nueva implementación más lenta de la ordenación bitónica, pero funcional con listas de cualquier tamaño, es utilizada.

3.2.3. Fase 3

Ir dividiendo la lista en sublistas muy pequeñas puede degradar el rendimiento del algoritmo ya que se necesita muchos movimientos de pequeñas cantidades de datos en

memoria. Debido a ello, si la longitud de la lista es igual o menor de un determinado límite se aplica un algoritmo diferente, el de la ordenación bitónica.

Este caso se puede dar en dos ocasiones distintas, si en la fase 1 se crea una sublista igual o más pequeña que el límite establecido o en las llamadas recursivas de la fase 2, explicado en la sección 3.2.2.

Antes de la explicación de la ordenación bitónica, se debe entender qué es una secuencia bitónica. Una secuencia $a = (a_1, a_2, ..., a_n)$ de longitud n es bitónica, si y solo si se da una de las siguientes condiciones:

- $a = (a_1 \leq a_2 \leq ... \leq a_k \geq ... \geq a_n)$, donde k es $1 < k < n$
- $a = (a_1 \geq a_2 \geq ... \geq a_k \leq ... \leq a_n)$, donde k es $1 < k < n$
- Cualquier de los dos puntos anteriores con la lista rotada

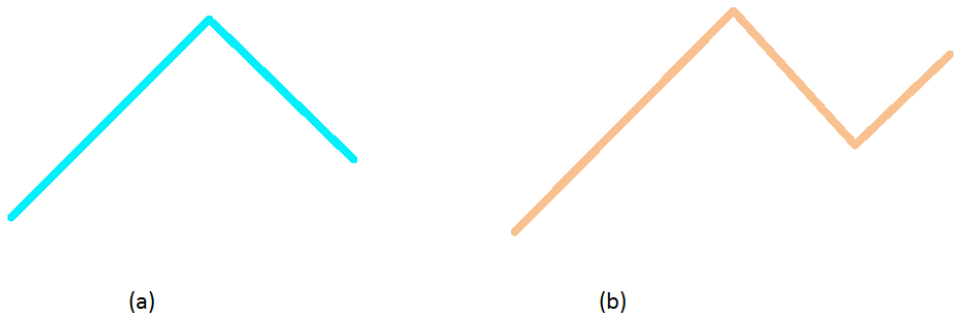


FIGURA 3.3: (a) Una secuencia bitónica. (b) Una secuencia no bitónica.

Si, a partir de una secuencia bitónica se cambian los valores de a_k y $a_{k+n/2}$ en el caso de que $a_k < a_{k+n/2}$, para $1 < k < n/2$, se obtiene dos secuencias bitónicas (véase figura 3.4).

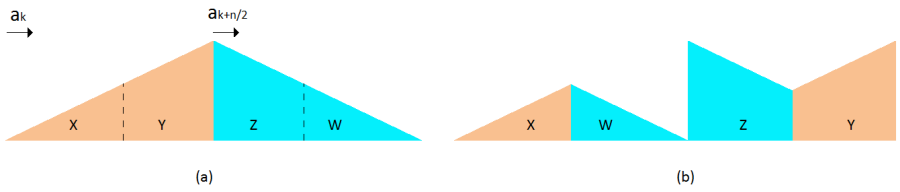


FIGURA 3.4: (a) Una secuencia bitónica. (b) La misma secuencia bitónica dividida en dos nuevas secuencias bitónica.

Éste proceso se repite para éstas nuevas secuencias bitónicas hasta que se tenga una lista ordenada.

Nótese que antes de ordenar, primero se debe transformar la lista inicial en una secuencia bitónica. Para ello, si se tienen dos listas ordenadas, tal que $a_1 \leq a_2 \leq ... \leq a_k$ y $b_1 \leq b_2 \leq ... \leq b_k$, se puede crear una secuencia bitónica si la segunda es girada, obteniendo $a_1, a_2, ..., a_n, b_n, ..., b_2, b_1$. Si las dos listas tienen un único elemento cada uno, este proceso crearía una lista ordenada.

Es posible realizar el paso anterior y crear dos secuencias bitónicas a la vez con un simple cambio a lo mencionado sobre la división de secuencias bitónicas. El cambio consiste en hacer las comparaciones del siguiente modo: $a_k < a_{n-k}$, para $1 < k < n/2$. Se puede observar el resultado en la figura 3.5. Se referenciará a este proceso como combinación (o caja marrón más adelante).

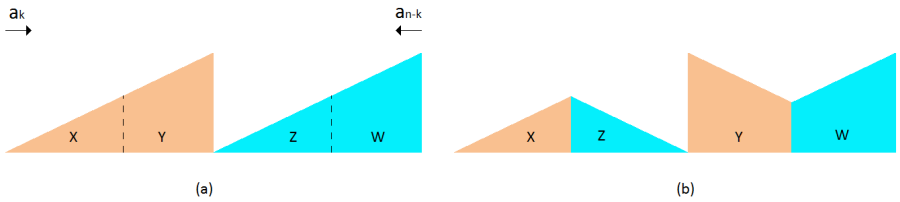


FIGURA 3.5: (a) Dos listas ordenadas. (b) La misma secuencia bitónica dividida en dos nuevas secuencias bitónica.

Por lo que, el algoritmo de la ordenación bitónica consiste en, primero dividir la lista inicial en sublistas de 2 elementos y combinarlos, acto seguido, se combinan éstas listas creando 2 secuencias bitónicas de 2 elementos cada una. Ahora se puede aplicar la recursividad anterior para ordenar éstas listas, creando listas ordenadas de 4 elementos. Si se repite el proceso de combinar las listas ordenadas en secuencias bitónicas y ordenar éstas secuencias se obtendrá una lista final ordenada.

En la siguiente figura se puede observar éste proceso, donde las cajas marrones representan la combinación de las listas ordenadas y las cajas rojas la ordenación de éstas.

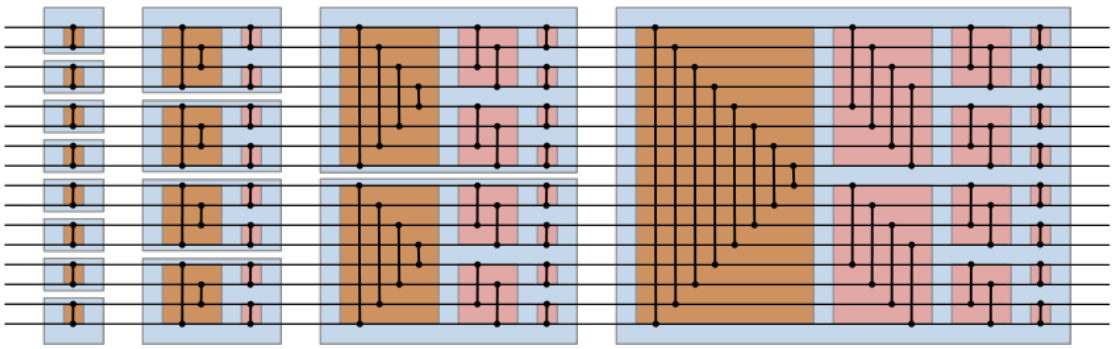


FIGURA 3.6: Comparaciones que se realizan en una ordenación bitónica sobre una lista de 16 elementos. Las cajas marrones representan la combinación de las listas ordenadas y las cajas rojas la división de una secuencia bitónica en 2 secuencias bitónicas más pequeñas.

3.3. Implementación

En esta sección se les proporciona nombres a las fases para facilitar la explicación y ayudar con el entendimiento. Empezando por la fase 1 hasta la fase 3 se renombrarán como ordenación bitónica, Small Quicksort y Big Quicksort respectivamente.

Cada bloque que se creará tendrá 1024 hilos, ya que de esta forma aumenta el rendimiento por varias razones:

- El algoritmo de ordenación bitónica puede realizar el ordenamiento de una cantidad mayor de elementos
- En las dos fases de Quicksort se mejorará la utilización de la memoria compartida, debido a que se le puede asignar una cantidad mayor a cada bloque sin llegar al límite de memoria compartida concurrente por multiprocesador.
- En el multiprocesador de la GPU que se utiliza (y todos que tengan un compute capability mayor que 3.0), puede ejecutar de forma concurrente 2048 hilos, por lo que, se podrían ejecutar 2 bloques de 1024 hilo de forma concurrente en cada multiprocesador.

El procesador es el encargado tanto de reservar memoria, inicializar elementos y transferir los datos necesarios entre éste y la GPU como de la gestión de todos los búferes dobles y la iteración del algoritmo.

A continuación se explica en detalle la implementación de cada una de las fases junto con sus optimizaciones empezando por Bitonic sort y seguido de Small Quicksort y Big Quicksort.

3.3.1. Ordenación bitónica

La implementación de la ordenación bitónica se puede realizar utilizando dos bucles anidados. Volviendo a la figura 3.6, el bucle exterior representa las cajas azules y el bucle interior las cajas rosas. Las cajas marrones se realizan al inicio de cada nueva iteración del bucle exterior.

Los bucles pueden ser realizados dentro del kernel siempre y cuando la lista que se tiene que ordenar tenga una longitud igual o inferior al doble del tamaño de un bloque, ya que de este modo, un único bloque puede realizar toda la ordenación. El límite que se mencionaba en secciones anteriores viene dado por ésto.

También se presenta una segunda implementación en la que los bucles se realizan fuera del kernel para poder ordenar listas con mayor tamaño, utilizada en caso de que se llegue a la máxima profundidad de la recursividad de la fase 2. Para ello se crea dos kernels nuevos, uno para las cajas marrones y otro para las cajas rosas.

La paralelización viene dada por las comprobaciones que se realizan en las cajas marrones y rosas. Cada hilo se encarga de realizar la comprobación y el cambio de elementos (si es necesario) entre dos elementos, en cada una de las cajas.

3.3.1.1. Memoria compartida

Para el caso de tener una lista que puede ser ordenada por un único bloque utilizando el primer kernel descrito en la sección anterior, se puede utilizar memoria compartida para reducir los accesos a memoria global. Inicialmente se transfiere toda la lista a memoria compartida, después se realiza la ordenación bitónica, y finalmente, se transfiere a memoria global de nuevo. De este modo, se reduce la cantidad de accesos a memoria global, aumentando el rendimiento.

No es eficiente utilizar memoria compartida en el caso en el que se tenga que utilizar más de un bloque para la ordenación, ya que, en cada lanzamiento del kernel se realiza una única lectura y escritura a memoria global.

3.3.1.2. Realizar la primera comprobación fuera del bucle

La primera comprobación (primeras cajas marrones de la figura 3.6) se realiza fuera de los bucles, por lo que se puede facilitar el cálculo de los índices que se tienen que comprobar y cambiar, resultando en una pequeña mejora en rendimiento.

3.3.2. Small Quicksort

La implementación es muy parecida a la descrita en la fase 1 del apartado de diseño, lo que se utilizará la figura 3.2 para facilitar la explicación.

Al inicio del algoritmo, cada hilo del bloque se encarga de contar la cantidad de elementos menores/mayores que el pivote se le ha asignado ((C) figura 3.2). Una vez acabado, se realiza la suma acumulativa de los resultados conseguidos entre los hilos ((D) figura 3.2). Para ellos se necesita de memoria compartida para que todos los hilos puedan comunicarse entre ellos, también se necesita dos búferes dobles para que el resultado de un hilo no sobreescriba unos valores que otro hilo no ha leído. Son dos búferes ya que el mismo procesamiento se aplica tanto para los valores menores que el pivote como para los mayores. Considerando que los bloques son de 1024 hilos y un elemento es de 4 bytes, se necesita un total de *2 * 2 * (1024 * 4 B) = 16384 B = 16 KB* de memoria compartida por cada bloque.

Ahora que se sabe el desplazamiento de cada hilo, se puede empezar a reordenar los elementos en dos sublistas, pero debido a que se trabaja en paralelo y un hilo puede sobreescribir un elemento de otro, se debe utilizar un búfer doble de la lista.

Finalmente, el hilo con identificador 0 se encarga de mover el pivote en medio de las dos sublistas y de realizar las llamadas recursivas al mismo kernel o al algoritmo de ordenación bitónica dependiendo de los criterios mencionados en la sección 3.2.2.

Los siguientes subapartados describen optimizaciones realizadas sobre el algoritmo junto con sus resultados.

3.3.2.1. Uso de streams

Hasta el momento no hay paralelización entre sublistas, siempre que se realiza una llamada recursiva sobre una de las dos listas, esta llamada debe acabar para que se inicie la segunda ya que todos los kernels se declaran en el stream por defecto, resultando en una ejecución en serie perdiendo gran parte de paralelización. El resultado es un recorrido en anchura de las ramas que la recursividad crea, ejecutando un solo nodo a la vez, cuando claramente se pueden paralelizar.

Creando y asignando nuevos streams a cada nodo desbloquea tal limitación, ejecutando en paralelo todos los nodos a lo ancho de un nivel.

3.3.2.2. Instrucciones *shuffle*

Las instrucciones *shuffle* dan la posibilidad de intercambiar datos entre diferentes hilos de un mismo warp sin la utilización de memoria compartida o barreras. Existen 4 variantes: __shfl_up, __shfl_down, __shfl_xor, __shfl, pero solo utilizaremos la primera variante.

```
int __shfl_up(int var, unsigned int delta);
```

FIGURA 3.7: Cabecera de la instrucción __shfl_up

La instrucción copia un registro de un hilo del mismo warp con identificador más pequeño. Se necesitan dos parámetros, el valor que se desea mandar y un offset dentro del warp (figura 3.7).

El __shfl_up se puede utilizar para calcular la suma acumulativa utilizando un bucle formando el patrón descrito en la siguiente figura, donde, cada punto de un mismo nivel representan los warps y las flechas, la copia de los datos de un hilo al siguiente.

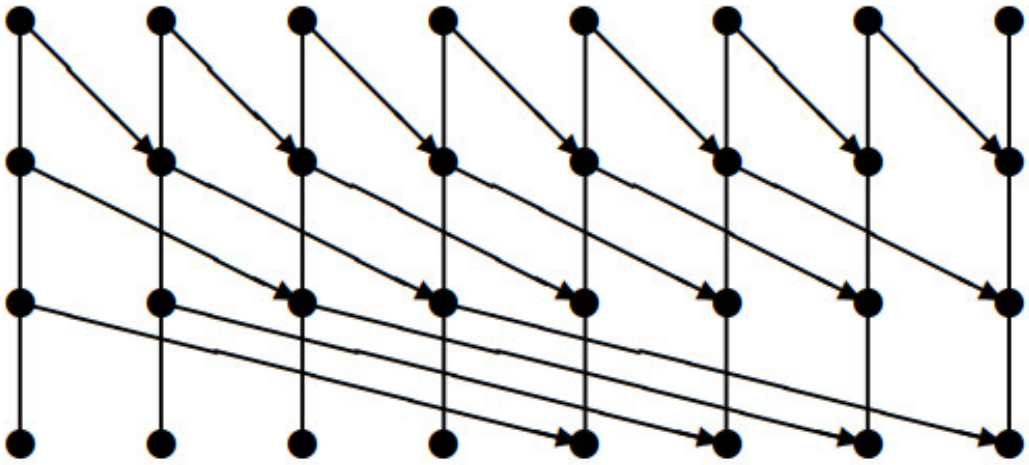


FIGURA 3.8: Ilustración de una suma acumulativa

Las ventajas de la utilización de instrucciones a nivel de warp en el caso de la suma acumulativa son la poca cantidad de barreras de sincronización entre hilos que se necesitan, la utilización de registros en vez de memoria compartida para compartir los valores dentro de un mismo warp y, por ello, la reducción de memoria compartida necesaria, pudiendo utilizarla para otros fines como se describe en los apartados siguiente.

También tiene desventaja, su implementación es más compleja, resultando en un código más difícil de entender, ya que, primero se debe realizar una suma acumulativa dentro de

cada warp, el último hilo de cada warp debe almacenar su valor en memoria compartida (por lo que no se prescinde del todo de memoria compartida), el primer warp debe leer los elementos de la memoria, realizar una nueva suma acumulativa sobre éstas, guardar sus resultados, y que cada hilo de los otros warps sume el valor correspondiente a su warp a la suma acumulativa inicial.

Debido a la complejidad añadida al algoritmo, su rendimiento es peor que las implementaciones anteriores (figura 3.11), pero es necesario utilizar ésta técnica para poder liberar memoria compartida que se aprovechará, tal y como se explica en los apartados siguientes.

3.3.2.3. Cargar el segmento a memoria compartida

La lectura del segmento se realiza 2 veces, una al contar la cantidad de elementos mayores/menores que el pivote y otra cuando realmente se realiza el movimiento de datos. Cargando el segmento en memoria compartida se puede reducir la cantidad de lectoras de memoria global, aumentando el rendimiento.

Gracias a las instrucciones utilizadas en la sección 3.3.2.2, se dispone de una cantidad de memoria compartida mayor, pudiendo cargar un segmento de mayor tamaño.

Cargando la lista a memoria compartida viene con otra ventaja, y es el hecho de que se puede eliminar el búfer doble de la lista. Se tiene que tener en cuenta que realmente el búfer doble si se necesita en el caso del Big Quicksort, y realmente no se elimina, pero sí que se reduce complejidad de ésta fase del algoritmo.

3.3.2.4. Guardar las sublistas en memoria compartida

Los accesos seguidos a memoria global mejora el rendimiento, y si se observa, las lecturas ya se realizan de este modo, pero no las escrituras. Para ello, en vez de escribir directamente a memoria global, primero se escriben a memoria compartida, que no tiene presente ésta limitación, y finalmente, de forma seguida se copia a memoria global.

El tamaño del segmento viene dado por la longitud máxima que se puede almacenar en memoria compartida teniendo en cuenta que también se reserva dicha memoria para la suma acumulativa y una copia de las sublistas antes de ser copiadas a memoria global.

3.3.3. Big Quicksort

La implementación de esta fase es muy parecida a la anterior, de hecho realiza el mismo funcionamiento pero con la diferencia de que en ésta fase se utiliza más de un bloque para la creación de las sublistas. Las dos fases comparten gran parte de código, siendo la única diferencia la necesidad de sincronizar los múltiplos bloques y la eliminación de las llamadas recursivas, que se realizan de forma iterativa en la CPU.

La sincronización se realiza utilizando unos contadores atómicos en la memoria global de la GPU, tal y como se especifica en la sección 3.2.1. Debido a que puede haber más de una instancia del kernel en ejecución de forma concurrente, se debe reservar una lista. La CPU será la encargada de gestionar tal lista, asignando un solo valor de ella a cada ejecución del kernel.

Una unidad de la lista consta de 3 enteros, dos para los contadores de los desplazamientos menores/mayores que el pivote y un contador para la cantidad de bloques que han acabado. Los contadores de los desplazamientos son necesarios para que cada bloque sepa donde debe escribir sus elementos y el contador de bloque se utiliza para saber cual es el último bloque en acabar, que es el encargado de mover el pivote entre las dos sublistas resultantes, reiniciar los contadores y guardar los tamaños de las nuevas sublistas.

Los tamaños de las sublistas se guardan en un lista de igual tamaño que la lista de los contadores atómicos residente en memoria global también. Esta lista consta de 3 enteros, el inicio de la primera sublista, la posición del pivote y el fin de la segunda sublista y es utilizada para comunicar a la CPU de los tamaños de las sublistas resultantes. a la que, la CPU, en cada iteración, carga los valores y, a partir de los tamaños de las sublistas resultantes, inicia una de las 3 fases comentadas continuando con el algoritmo.

El tamaño de las 2 listas de control comentadas anteriormente es determinado a partir del tamaño de la lista a ordenar considerando el peor de los casos, cuando esta dividida en sublistas de tamaños iguales a los segmentos + 1. Esto es debido a que si los las sublistas fueran de tamaños menores, se ejecutaría una de las otras fases que no necesitan este tipo de listas de control.

A nivel de kernel no se presentan más optimizaciones, ya que ésta fase se aprovecha de todas las fases anteriores (sección 3.3.2), por lo que los siguientes subapartados se centrarán más en la optimización de la iteración en la CPU.

3.3.3.1. Uso de streams

Igual que en el apartado de Small Quicksort, la utilización de streams en cada uno de los lanzamientos de kernels puede aumentar la concurrencia de éstos, mejorando el rendimiento.

Se crea una cantidad máxima de streams al inicio del algoritmo, y se van asignando de forma circular a cada uno de los kernels.

3.3.3.2. Atrasar la creación de streams

Hay que tener en cuenta que la creación de streams tiene una sobrecarga adicional para el procesador, y crear una cantidad de éstos antes del algoritmo de ordenación, con la posibilidad de que no se utilicen todos, atrasa la ejecución de la fase iterativa.

Por ello, se creara un stream siempre que sea necesario, lanzando el primer kernel lo antes posible. También hay que tener en cuenta que la ejecución de un kernel es asíncrona, por lo que, mientras la GPU realiza los calculos ordenados, la CPU está libre para hacer otras tareas, como puede ser la creación de nuevos streams. De este modo se solapa la creación de nuevos streams con la ejecución de los kernels en la GPU.

A cada iteración se aprovechan los streams ya creados, previniendo la creación de nuevos si no es necesario.

3.3.3.3. Iteración en la GPU

A cada iteración del algoritmo realizado en la CPU, se debe desplazar la lista de particiones de la memoria de la GPU a la memoria RAM para que la CPU pueda tener acceso a ella. Esto puede ser lento, y puede atrasar el algoritmo.

Aprovechando el dynamic parallelism (ejecución de nuevos kernels desde la GPU y no CPU), la iteración se puede realizar en la GPU.

Como se podrá observar en los resultados, esto no mejora el tiempo de ejecución. Esto es debido a la complejidad del bucle, ya que un hilo de la GPU es significativamente más lento que uno de la CPU, y también a los lentos accesos a memoria global.

3.4. Resultados

Primero se comenta los resultados de la primera fase, a continuación la primera fase junto con la segunda, después el resultado final de las tres fases juntas y finalmente se compara la implementación en serie con la mejor implementación para la GPU.

Las fases se comentan separadas para observar los resultados de las optimizaciones.

También hay que mencionar que en las secciones de la ordenación bitónica y el Small Quicksort, la forma de obtener los tiempos es diferente a las demás, ya que se cronometra únicamente la ejecución del kernel. Ésta desición se ha tomado por varias razones:

1. En éstas fases se trabajan con listas muy reducidas, teniendo el tiempo de ejecución muy reducido de por sí, dificultando la observación de la mejora.
2. Todo el trabajo realizado por la CPU es igual en las dos fases. Teniendo en cuenta la asignación de memoria y las transferencias, entre otras cosas, pueden ocultar las mejoras de los cambios que se realizan.
3. Todas las optimizaciones se realizan a nivel del kernel que se cronometra, sin variar lo demás.

En el caso de las optimizaciones para el Big Quicksort, sí que se tiene en cuenta la ejecución de todo el algoritmo, al igual que en el caso de la implementación en serie.

3.4.1. Ordenación bitónica

En esta sección se compara los resultados de la implementación de un único kernel del algoritmo. La implementación del doble kernel se compara en la siguiente sección junto a los resultados del Small Quicksort.

Todas los experimentos mostrados a continuación fueron realizados sobre una lista de 2048 elementos, el máximo posible para este kernel.

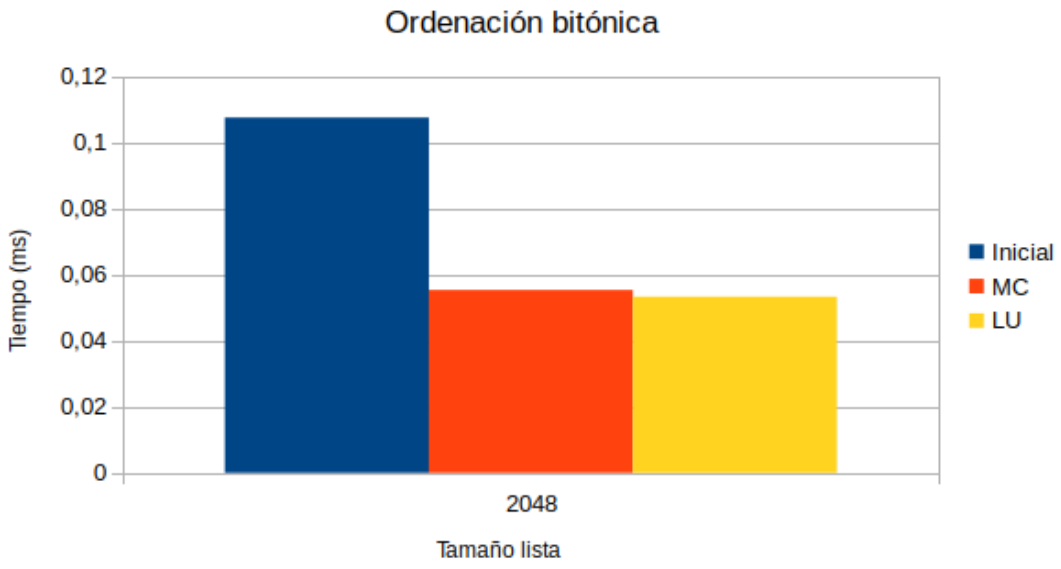


FIGURA 3.9: Resultados del experimento sobre cada una de las optimizaciones de la ordenación bitónica utilizando listas de 2048 elementos.

La utilización de memoria compartida muestra unos resultados muy buenos, en comparación con no utilizarla, con un speedup de aproximadamente 2. Ésto es debido a que el algoritmo realiza muchos accesos a memoria, si éstos se realizan a la memoria compartida en vez de la global, el rendimiento mejora considerablemente.

Para la última implementación, los resultados apenas se aprecian, pero teniendo en cuenta que ésta fase se ejecuta una gran cantidad de veces, cada mejora cuenta.

En la siguiente figura se compara la implementación realizada con un kernel con la realizada con dos. Como era de esperar, hay una gran diferencia en cuanto al tiempo de ejecución, con un speedup de aproximadamente 10 en favor de la implementación de un kernel.

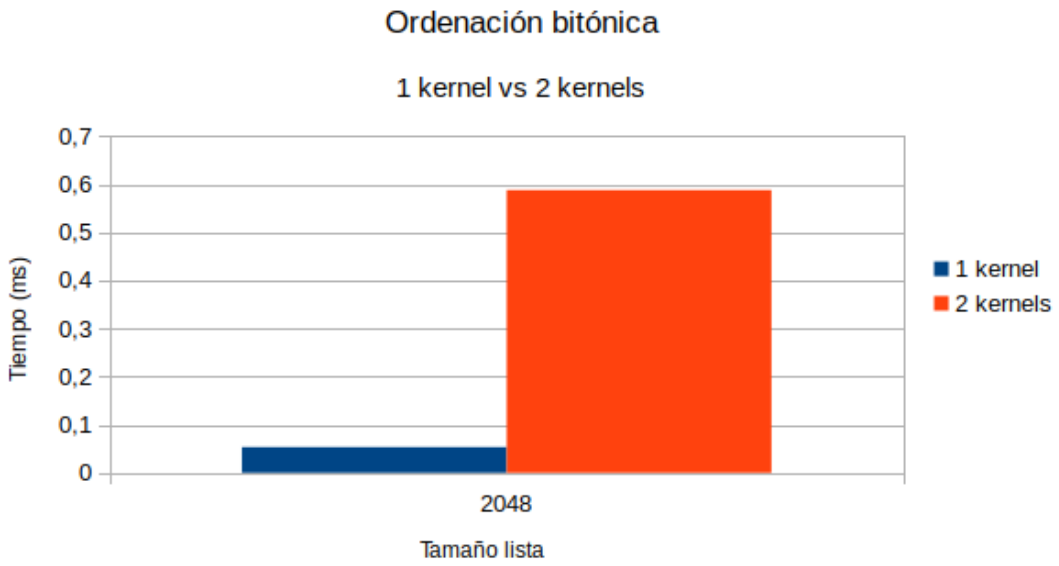


FIGURA 3.10: Diferencia en cuanto al tiempo de ejecución de la implementación del bitonic sort en un kernel y en dos kernels.

Obviamente, esto es debido a la gran cantidad de inicializaciones de kernels que se ejecutan.

3.4.2. Small Quicksort

Debido a la limitación de la longitud de la lista comentada en la sección 3.3.2, los experimentos se han realizado con la máxima longitud del segmento posible (en éste caso de 6080).

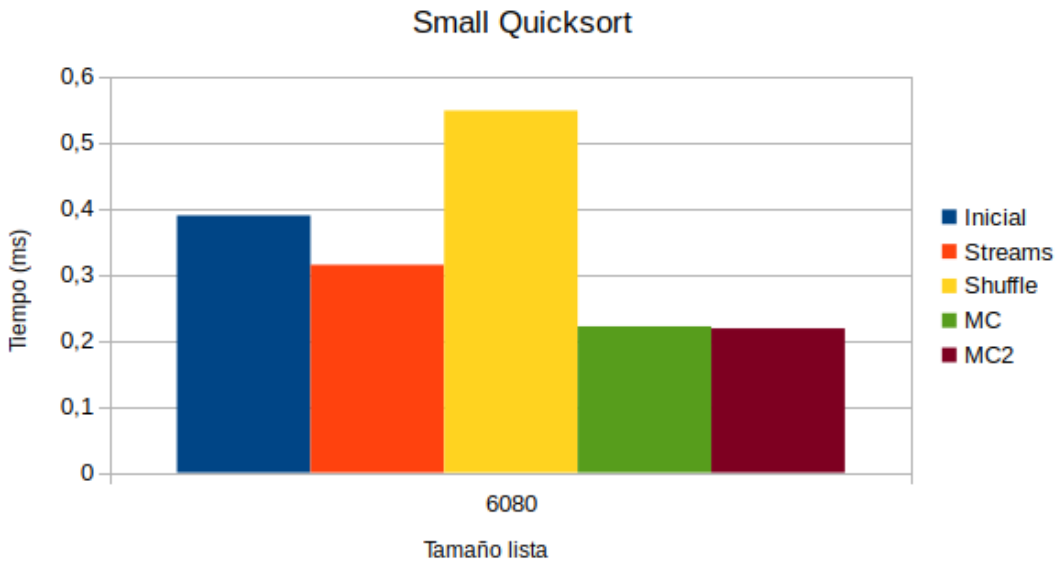


FIGURA 3.11: Resultados del experimento sobre cada una de las optimizaciones del Small Quicksort.

Como se puede observar en la figura anterior, la mayor diferencia en rendimiento se da en el caso de la introducción de los streams, con un speedups de 1,26. En el caso de la introducción de las instrucciones shuffle, el rendimiento es muy pobre tal y como se comentó en la sección 3.3.2.2, pero ha sido necesario para poder aprovechar al máximo la memoria compartida para las siguiente optimizaciones. El speedup final se encuentra a un valor muy aproximado a 1.77.

Podemos observar, al igual que en el caso de la ordenación bitónica, que aunque parezca que los speedups no son muy grandes, este algoritmo se ejecuta una cantidad de veces muy elevada con listas grandes, además de ser las bases de la siguiente fase, Big Quicksort.

3.4.3. Big Quicksort

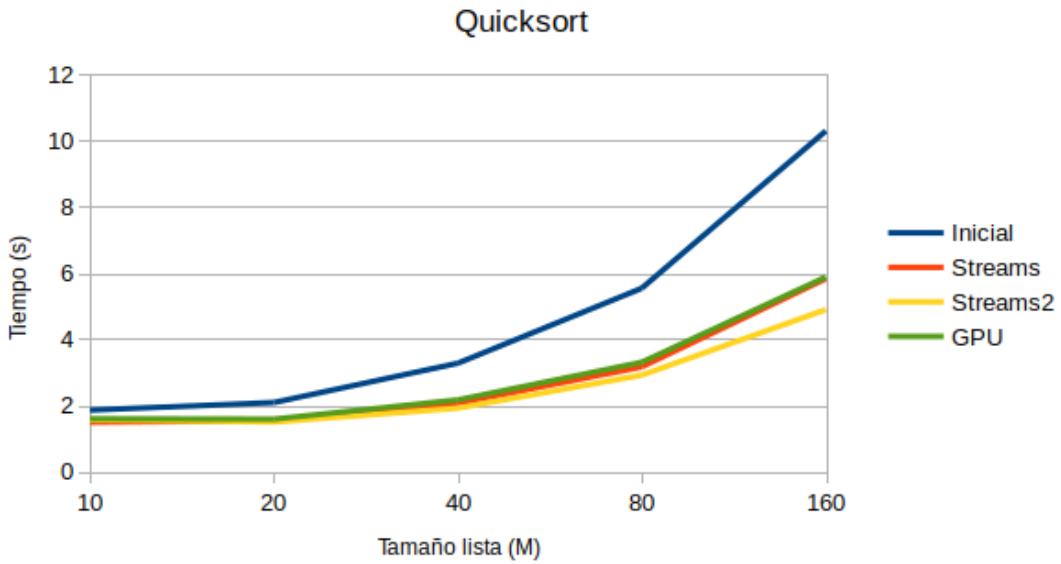


FIGURA 3.12: Diferencia de tiempos entre optimizaciones del algoritmo Quicksort.

Tal y como se ha observado en la sección anterior, el mayor aumento en rendimiento viene dado por el uso de streams, ya que, al utilizarlos, los kernels lanzados por la CPU pueden ser ejecutados de forma concurrente en la GPU, siempre y cuando se pueda. El speedup conseguido con los streams es de 1,76 en el caso de listas de tamaño de 160 millones, comparando la implementación inicial y la segunda.

También se puede observar una ligera mejora entre entre la segunda y tercera implementación.

La implementación de todo el algoritmo en la GPU tiene un rendimiento algo menor, ésto es debido a que el algoritmo es complejo, y un hilo de la GPU es más lento que uno de la CPU, por lo que compensa realizar las inicializaciones de otros kernels en la CPU si éste proceso es costoso.

Teniendo en cuenta la implementación inicial y la tercera, el speedup es de 2,1 en el caso de una lista de 160 millones de elementos, y según la tendencia, aumentaría más si se dispondría de suficiente memoria en la GPU.

3.4.4. CPU vs GPU

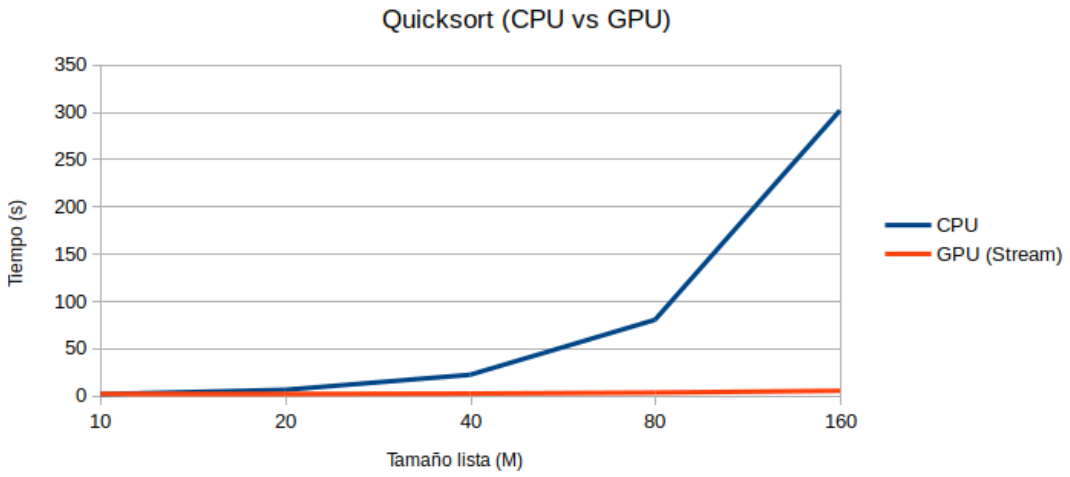


FIGURA 3.13: Diferencia de tiempos entre la ejecución del algoritmo en la CPU y en la GPU.

Se puede observar que la implementación del algoritmo en la GPU tiene un speedup muy elevado en comparación con la implementación en serie y tiene a aumentar a medida que la lista crece. El speedup es del 27.5 en el caso de una lista de 80 millones de elementos y de 51.21 en el caso de una lista de 160 milliones de elementos, casi el doble.

El speedup aumenta considerablemente a medida que se utiliza una lista más grande, aprovechando el paralelismo proporcionado por la GPU, pero éste valor tiene un límite de alrededor de 300 millones de elementos (en el caso de la GPU utilizada), por falta de memoria en la GPU.

Capítulo 4

Merge sort

4.1. Introducción

El Merge sort es un algoritmo de ordenación basado en la técnica de divide y vencerás y funcionamiento parecido al del Quicksort. Fue inventado por John von Neumann en 1945[4].

La idea básica consta de dos pasos:

1. Dividir la lista no ordenada en múltiples sublistas de longitud 1.
2. Las sublistas resultantes se van mezclando creando nuevas sublistas ordenadas. El proceso se repite hasta obtener una lista ordenada.

Existen dos formas de dividir la lista inicial en sublistas. Una consiste en realizar llamadas recursivas dividiendo la lista en dos sublistas por la mitad en cada nivel hasta obtener sublistas de longitud 1 (top-down), realizando la mezcla a medida que se sube por la recursividad creada (figura 4.1). La segunda, consiste en tratar la lista inicial como sublistas de longitud 1 y, iterativamente (bottom-up), mezclarlas hasta obtener la lista ordenada.

Debido a limitaciones en el uso de la recursividad en la GPU, el primer método no es apto para la implementación, y se implementará el segundo método.

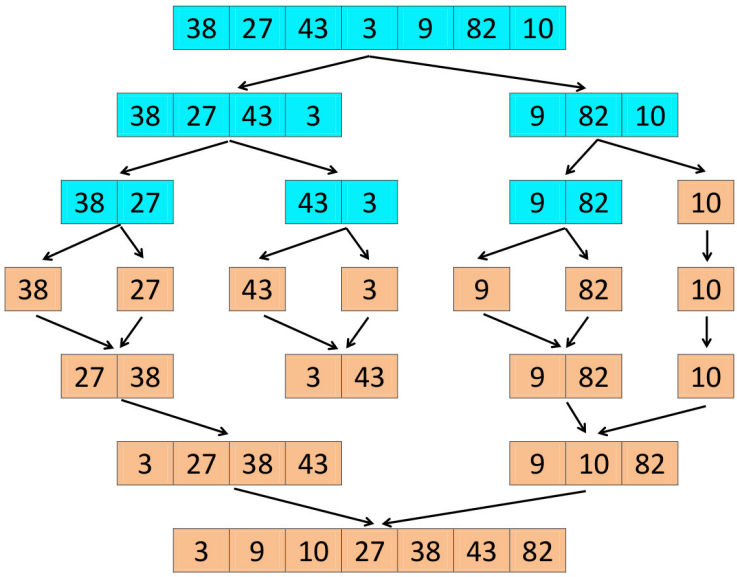


FIGURA 4.1: Ejemplo de ordenación por mezcla recursiva (top-down)

Al igual que en el algoritmo Quicksort, es ineficiente utilizar este algoritmo para mezclar sublistas muy pequeñas. En la implementación en serie se suele utilizar el ordenamiento por inserción, pero debido a la dificultad e ineficiencia de su implementación en serie, se utilizará el ordenamiento por selección (Selection sort).

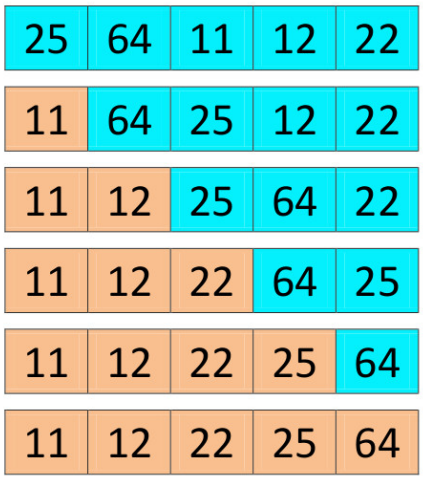


FIGURA 4.2: Ejemplo de ordenación por selección

El algoritmo divide la lista en dos partes, una sublista de elementos ordenados inicialmente de longitud 0 y otra sublista de elementos no ordenados. Se busca el mínimo (o máximo dependiendo del resultado deseado) elemento de la sublista de elementos no ordenados, se intercambia con el elemento de la primera posición de la misma sublista (añadiéndolo al final de la sublista de elementos ordenados) y se mueve los límites de

entre estas dos sublista. Se repite el procedimiento hasta que todos los elementos estén en la sublista de ordenados.

La elección de un algoritmo diferente al de la ordenación bitónica es para estudiar el funcionamiento de múltiplos algoritmos en la GPU.

4.2. Diseño

La implementación recursiva (top-down) no tiene una implementación directa en la GPU, ya que las llamadas recursivas tienen una cierta profundidad, fallando el lanzamiento de todos los nuevos kernels a partir de dicha profundidad, imposibilitando la implementación del algoritmo cuando se desea ordenar listas de gran tamaño.

La implementación iterativa (bottom-up) es la que se estudiará, ya que se elimina la limitación anterior por no disponer de recursividad.

El algoritmo se realizará en dos fases distintas:

1. Inicialmente se empleará el algoritmo de ordenación por selección para ordenar sublistas de longitud m.
2. Mezclar las sublistas ordenadas en sublistas ordenadas de tamaños mayores de forma iterativa hasta obtener una sola lista ordenada.

FIGURA 4.3: Las dos fases del algoritmo

La forma de mezclar dos sublistas ordenadas en paralelo es una tarea mucho más complicada que en serie, pero se han diseñado dos algoritmos distintos para realizar esta mezcla. Uno de ellos consiste en buscar, de forma dicotómica, la posición en la que un valor de una sublista se situaría en la segunda sublista resultando en una nueva sublista ordenada. El segundo está basado en la búsqueda de las intersecciones de las diagonales del algoritmo *Merge Path*[5].

4.2.1. Búsqueda dicotómica en las sublistas

Se considera dos listas ordenadas A y B con longitudes que pueden variar entre ellas. Se escoge un elemento de la lista A, llamado x y posición i, y de forma dicotómica, se busca dos elementos, tal que $y < x < z$, donde y y z son elementos consecutivos de la lista B con posiciones j y k respectivamente. La posición final de dicho elemento x viene dada de la suma de su posición y la posición del elemento y $(i + j)$. Esta operación se repite para todos los elementos de A. Los elementos de B se mezclan de igual forma con los elementos de A, invirtiendo los roles de las listas.

Este algoritmo es altamente paralelizable ya que ningún elemento de las listas depende de otro, y todos de pueden ordenar en paralelo.

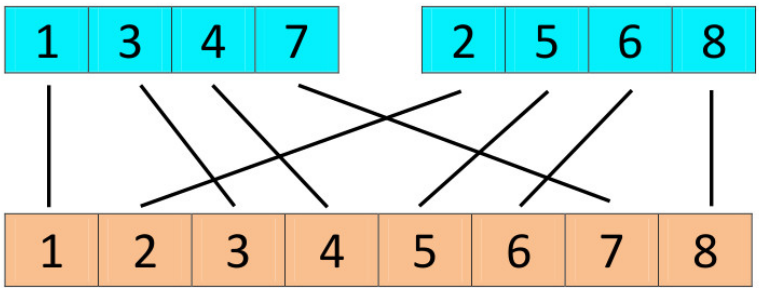
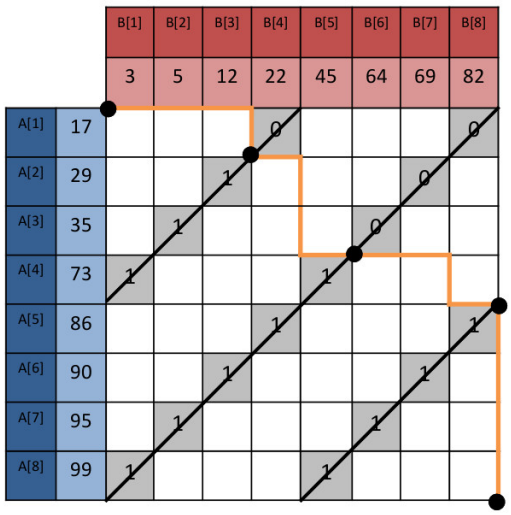


FIGURA 4.4: Ejemplo de mezclar dos sublistas con búsquedas dicotómicas.

4.2.2. Merge Path

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		3	5	12	22	45	64	69	82
A[1]	17	1	1	1	0	0	0	0	0
A[2]	29	1	1	1	1	0	0	0	0
A[3]	35	1	1	1	1	0	0	0	0
A[4]	73	1	1	1	1	1	1	1	0
A[5]	86	1	1	1	1	1	1	1	1
A[6]	90	1	1	1	1	1	1	1	1
A[7]	95	1	1	1	1	1	1	1	1
A[8]	99	1	1	1	1	1	1	1	1

(a)



(b)

FIGURA 4.5: (a) Se muestra la matriz de mezcla (Merge Matrix) al combinar dos listas ordenadas, la frontera entre los zeros y unos indican el resultado del *Merge Path*. (b) Las intersecciones de las diagonales y el *Merge Path*.

Igual que en el sección 4.2.1, se considera dos listas ordenadas, A y B, con longitudes que pueden variar. La matriz de mezcla (Merge Matrix) ((a) figura 4.5) consiste en crear una matriz en la que, la dimensión de sus filas sea igual a la longitud de la lista A y la dimensión de sus columnas igual a la longitud de la lista B.

Al mezclar las listas, se va construyendo el *Merge Path*. Refiriéndonos a la figura 4.5 (a), empezando por el rincón superior izquierdo y considerando i como índice de la lista A y j como índice de la lista B, si $A[i] > B[j]$, se mueve una posición a la derecha y se incrementa la posición de j, en caso contrario se mueve una posición hacia abajo y se incrementa la posición de i. Este proceso se repite hasta que se llega al rincón inferior derecho. La suma de los índices i y j indican la posición de la nueva lista, en la que se debe guardar el menor elemento resultante de la comparación de $A[i]$ y $B[j]$.

4.3. Implementación

Como se ha descrito en el apartado de diseño, el algoritmo consta de 2 fases. Estas dos fases no son parecidas a las del Quicksort ya que la segunda fase depende de la primera, una vez acabada la primera, se inicia la segunda.

En fase inicial se divide la lista en sublistas de longitud n y se realiza un ordenamiento por selección sobre cada una de ellas. El valor n viene determinado por el doble de la dimensión del bloque utilizado, debido a que es el valor óptimo utilizado por la reducción que se realiza (cada hilo realiza una comparación), explicada a continuación.

Cada bloque es encargado de ordenar una sublista, por lo que no hay comunicación entre ellos, aumentando la paralelización que se puede obtener. El funcionamiento es muy parecido al original, con el simple cambio de que la búsqueda del valor más pequeño se realiza en paralelo gracias al algoritmo de reducción. Una vez encontrado el valor mínimo, un hilo se encarga de hacer el intercambio de éste elemento en su posición correspondiente.

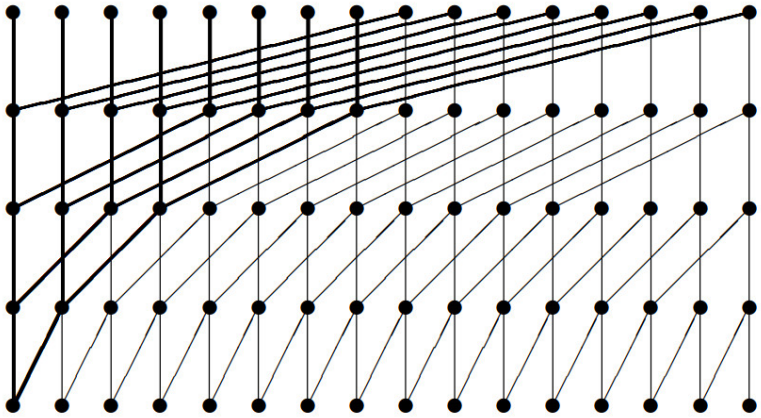


FIGURA 4.6: Ilustración de los pasos del algoritmo de reducción.

Como se puede observar en la figura 4.6, a cada paso nuevo al realizar la reducción, se necesita sobreescribir los elementos anteriores, perdiéndolos. Una posible solución inicial sería hacer una copia de la lista y aplicar el algoritmo sobre ella, pero de esta manera sí se encuentra el elemento mínimo, pero se pierda su posición de la lista original, impidiendo que se pudiera hacer el intercambio. Para ello, en vez de realizar una copia, se crea una lista de índices sobre la que se realiza la reducción y los elementos se acceden mediante éstos índices.

Teniendo en cuenta la división de la lista de un bloque en dos sublistas mencionadas en el apartado de diseño (una lista ordenada y otra no ordenada), el algoritmo final realiza los siguientes pasos de forma iterativa:

1. En memoria compartida, se guardan los números de 1 a n, donde n es la longitud de la sublista asignada al bloque actual, siendo éstos las posiciones de los elementos de la sublista a ordenar.

2. Se busca el mínimo elemento de la sublista no ordenada utilizando el algoritmo de reducción, las posiciones de la memoria compartida y los elementos de la lista.

3. Un hilo se encarga de intercambiar las posiciones del elemento mínimo resultante con el primer elemento de la sublista no ordenada, incrementar la longitud de la sublista ordenada y reducirla de la lista no ordenada

4. Repetir los pasos anteriores hasta que no quede ningún elemento a ordenar

La mezcla de las sublistas resultantes se realiza de forma iterativa, empezando por sublistas de tamaño inicial n y doblando el tamaño hasta tener una sola lista ordenada (fase 2, figura 4.3). Inicialmente, se implementa el algoritmo de búsqueda dicotómica y en la sección 4.3.4 se explica la implementación del algoritmo basado en *Merge Path*.

La búsqueda dicotómica se realiza de igual que en la sección 4.2.1, en la que cada hilo del bloque se encarga de buscar la posición correspondiente de un sólo elemento. Esto calcula sólo los posiciones de una lista, por lo que se tiene que realizar dos veces, calculando las posiciones de la segunda lista también.

A continuación se explicará cada una de las optimizaciones realizadas y la comparación de rendimiento entre éstas. Todas las optimizaciones se refieren al algoritmo de ordenación por selección o a la forma en la que la CPU maneja los datos, ya que la implementación de los algoritmos de mezcla son relativamente cortos sin mucho margen de optimización.

4.3.1. Memoria compartida para la lista

La lista, residente en memoria global, se accede múltiples veces y de forma no seguida, ya sea para hacer las comparaciones para buscar el mínimo como para intercambiar este mínimo con su posición correspondiente. La lentitud con la que opera esta memoria, reduce el rendimiento global del algoritmo.

Copiando inicialmente la lista a memoria compartida, realizando las ordenaciones necesarias en dicha memoria y, finalmente, copiando el resultado a memoria global, reduce la cantidad de lecturas y escrituras que se realizan sobre ésta a sólo una lectura y una escritura. Además, al trabajar en la memoria compartida, ya no se debe preocupar de accesos no seguidos, ya que ésta no presenta tal limitación.

Todo lo anterior reduce los altos retardos que se tiene al acceder a memoria global.

4.3.2. Aumentando el tamaño de la lista

Teniendo en cuenta la división de la lista en una sublista de elementos ordenados y otra de elementos no ordenados, tal y como se menciona en la sección 4.1, siempre que se añade un elemento a la lista de ordenados, el algoritmo de reducción debe funcionar sobre una lista de elementos no ordenados cada vez más pequeña. Por ello, a cada iteración, se debe hacer una comprobación para que los hilos no accedan una dirección de memoria no válida. En la figura 4.7 se puede apreciar lo esto, siendo las líneas discontinuas las que acceden fuera de la lista.

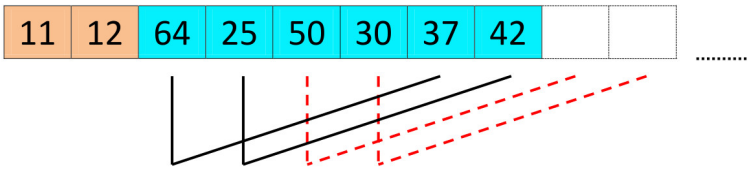


FIGURA 4.7: Los últimos hilos accedería fuera de la lista si no se realiza la comprobación.

Para reducir la cantidad de comprobaciones que un mismo hilo realiza (ya que este tipo de instrucciones reduce el rendimiento), se dobla la lista en memoria compartida y se rellena las posiciones mayores que el tamaño de la lista con el número más alto que se puede representar. De este modo, se puede prescindir de la comprobación mencionada anteriormente ya que ahora los hilos que anteriormente no cumplían con ella, sí la cumplen, pero el resultado de la reducción es el mismo, ya que la mínimo de cualquier número con el máximo número representable posible, nunca resultara en este máximo número (figura 4.8).

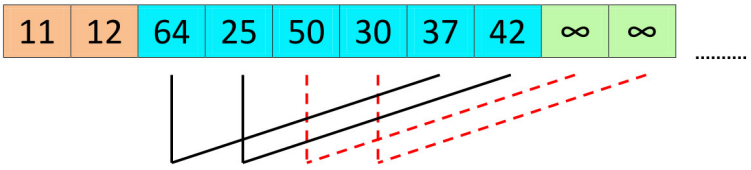


FIGURA 4.8: Al alargar la lista con valores máximos se puede eliminar la comprobación de fin de lista.

4.3.3. Uso de streams

Actualmente, la mezcla de las sublistas resultantes del ordenamiento por selección se realizan en serie a nivel de bloques, pero realmente todas las mezclas que se realizan en una iteración se pueden realizar en paralelo. Debido a que no se realizan en paralelo, no se aprovecha todo el paralelismo que la GPU ofrece, especialmente cuando se deben mezclar sublistas de tamaños reducidos.

Si se crean y asignan streams a los kernels que realizan la mezcla a cada iteración, éstos se ejecutarán de forma concurrente en la GPU, aumentando la ocupación de ésta y mejorando el rendimiento. Por lo que se crearan streams para cada kernel.

4.3.4. Merge Path

En esta sección se implementará el algoritmo basado en el *Merge Path*, sustituyendo el basado en la búsqueda dicotómica realizada en las dos sublistas.

Se considera A y B las dos listas que se tienen que mezclar y i y j los índices de estas listas respectivamente. Para realizar éste proceso en paralelo cada hilo busca la intersección de una diagonal d con el *Merge Path*, tal que $i + j = d$ ((b) figura 4.5). La intersección indica la posición final en la nueva lista, siendo ésta igual a d, y los dos elementos candidatos para tal posición situados en los índices i y j de las listas A y B respectivamente.

La propiedad $i + j = d$ simplifica la implementación, ya que el cálculo de los índices de las dos listas se reduce a una búsqueda dicotómica sobre la lista A, y el índice de la lista B se obtiene a partir de éste, siendo $j = d$ - i. El código final se muestra a continuación:

```
int lower_bound = max(0, idx - arrayB_size);
int upper_bound = min(idx, arrayA_size);

while (lower_bound < upper_bound)
{
        int mid_point = (lower_bound + upper_bound) >> 1;

        if (array_A[mid_point] > array_B[idx - mid_point - 1])
                upper_bound = mid_point;
        else
                lower_bound = mid_point + 1;
}

int a_idx = lower_bound;
int b_idx = idx - lower_bound;
```

donde la variable *idx* representa la diagonal d y *a_idx* y *b_idx* representan los índices i y j respectivamente.

4.3.5. Solapando la copia de datos a la GPU con el ordenamiento por selección

Existen dos tipos de transferencia de datos entre memorias, una síncrona en la que la CPU se encarga de transferir los datos y otra asíncrona, en la que la transferencia es realizada por una DMA. El proceso actual se realiza de forma síncrona, en la que primero se copia todo a la memoria de la GPU y se inician los cálculos necesarios.

Teniendo en cuanta que cada bloque realiza la ordenación sobre una pequeña porción de la lista, se pueden solapar la transferencia de datos con esta ordenación. Para ello, en un bucle, y siempre en un stream distinto, se realiza una transferencia de forma asíncrona de un segmento de la lista y se realiza el ordenamiento pos selección sobre ella. El procesador no tiene que esperar a que acaben estas dos operaciones, ya que

son asíncronas, y puede, en la siguiente iteración, encolar la siguiente transferencia y ordenación, en un stream distinto hasta acabar con toda la lista.

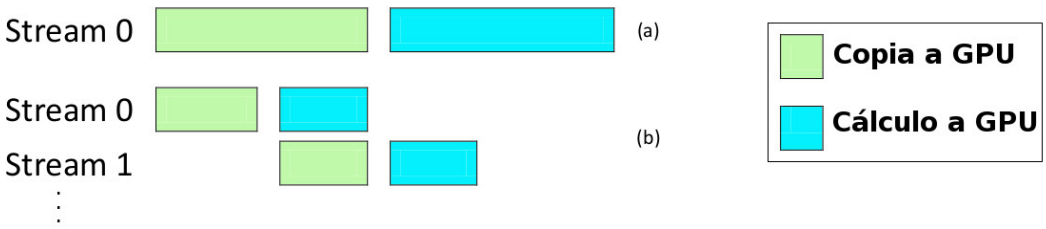


FIGURA 4.9: (a) No existe solapación, los cálculos empieza una vez que se haya copiado la lista entera. (b) Se solapa transferencia de datos con cálculo.

La GPU ejecutará la ordenación de un segmento siempre que se acabe con la transferencia de tal segmento. El resultado se puede observar en la figura 4.9, donde en (a) se indica la implementación inicial y en (b) la solapación de la transferencia de datos y cálculo.

4.4. Resultados

4.4.1. Merge sort

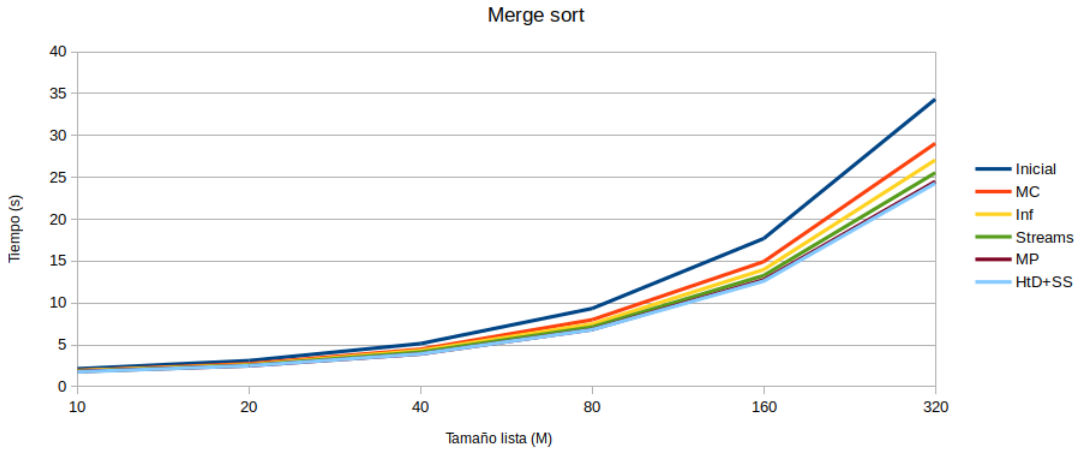


FIGURA 4.10: Comparación entre cada una de las optimizaciones implementadas.

El uso de memoria compartida es el que mayor speedup proporciona, ya que se realiza una cantidad muy elevada de accesos a la lista, llevando la lista a memoria compartida elimina casi todos estos accesos de memoria global (explicado en la sección 4.3.1). El speedup conseguido es de aproximadamente 1.2 con listas de un tamaño de 320 millones de elementos. Todas las demás aumentan ligeramente el rendimiento.

La concurrencia de la transferencia de datos a la GPU y la realización del ordenamiento por selección ha proporcionado un speedup muy bajo comparado con lo esperado. Ésto es debido a que el algoritmo de la ordenamiento por selección no es un algoritmo diseñado para la ejecución paralela, por lo que tiene un coste elevado en comparación con otros. Si se mira a los resultado que proporciona el NVIDIA Visual Profiler, se observa que el tiempo de cálculo que se realiza sobre un segmento transferido a GPU es aproximadamente 100 veces mayor, por lo que se ha realizado muy poco trabajo concurrente entre la CPU y la GPU.

El speedup final conseguido entre la implementación inicial y la final es de 1.4.

4.4.2. GPU vs CPU

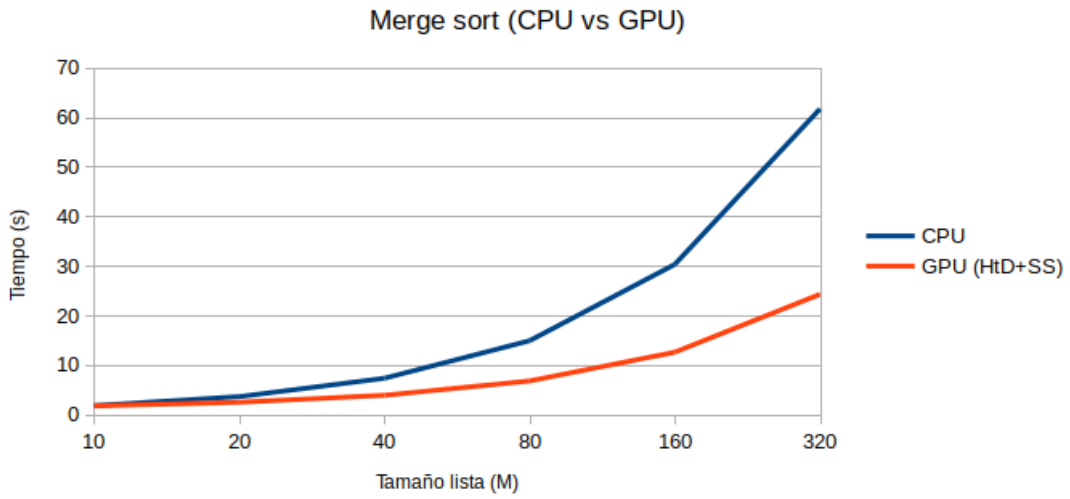


FIGURA 4.11: Comparación entre la versión en serie y en paralelo del algoritmo Mergesort.

En el grafo anterior se puede observar que realmente el algoritmo en la GPU gana en frente de su equivalente en serie, pero los resultados no son tan bueno como en el Quicksort (figura 3.13). Ésto es debido en parte a la lentitud del ordenamiento por selección. En cambio, se debe mencionar que éste algoritmo utiliza menos memoria que el Quicksort, ya que no necesita todas las listas que Quicksort utiliza para la comunicación entre los bloques de la GPU, y entre la GPU y CPU.

El speedup conseguido con una lista de 320 millones elementos es de 2.54.

Capítulo 5

K-means

5.1. Introducción

K-means es un algoritmo heurístico de agrupamiento, utilizado en la minería de datos, segmentación de mercados, astronomía, entre otros. Tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo más cercano a la media.

Fue propuesto por primera vez por Stuart Lloyd en 1957, pero no fue publicado hasta 1982[6].

Al ser un algoritmo heurístico, puede que no siempre tenga el mismo resultado, ya que esto depende enormemente en la selección de las posiciones iniciales de los centroides. Por ello se debe ir con precaución en la elección de éstas posiciones, o repetir el proceso hasta que se dé un resultado más óptimo.

5.2. Diseño

Dado un conjunto de n observaciones y un conjunto de k centroides generados aleatoriamente, el algoritmo repite los siguientes pasos hasta que converge:

1. A cada observación se le asigna el centroide más cercano a ésta.
2. Se calcula la media de cada uno de los puntos de las observaciones con el mismo centroide asignado en el apartado anterior. Éste nuevo punto representa la nueva coordenada de tal centroide.

Éstos pasos se repiten hasta que las nuevas coordenadas de los centroides sean iguales a las anteriores (el algoritmo converge). Finalmente, todas las observaciones de tienen asignado el mismo centroide pertenecer al mismo grupo.

Se puede observar el resultado gráficamente utilizando el script en Octave proporcionado. En el script se debe introducir el nombre del archivo de salida del algoritmo en la variable *filename*. El archivo de salida se obtiene utilizando la opción -o del programa.

5.3. Implementación

La distribución de memoria en la GPU que se utiliza inicialmente está compuesta de dos listas de longitud n para las observaciones, una para las coordenadas X y otra para las coordenadas Y, otras dos de longitud k para las coordenadas de los centroides y una última auxiliar de longitud n que se utiliza para guardar el índice del centroide más cercano de cada una de las observaciones.

El algoritmo consta de dos kernels, el primero se encarga de buscar cual es el centroide más cercano a cada una de las observaciones y el segundo calcula las nuevas posiciones de éstos centroides a partir de los resultados obtenidos en el primer kernel. Estos dos kernels se ejecutan repetidamente hasta que hasta que se llega a una convergencia que se comprueba en la CPU.

En el primer kernel, cada hilo se encarga de buscar el centroide más cercano de una única observación. Para ello, se calcula la distancia euclidiana de la observación a cada uno de los centroides y se guarda la mínima distancia en la lista auxiliar anteriormente mencionada para que el próximo kernel pueda recibir los resultados. En la distancia euclidiana se prescinde de la raíz cuadrada ya que es un cómputo muy costoso para la GPU y realmente no es necesario ya que se puede saber igualmente cual es la distancia más corta.

La cantidad de bloques que se lanzan para es segundo kernel es igual a la cantidad de centroides que se especifican (poca paralelización, véase sección 5.3.3). Cada bloque se encarga de recorrer la lista auxiliar con los índices de los centroides a los que pertenece cada observación en búsqueda de las observaciones que tienen el identificador del centroide más cercano igual al identificador del bloque. A medida que un hilo de los bloques encuentra una observación que cumple lo anterior, suma en sus registros las coordenadas y aumenta un contador que se utilizará para calcular la nueva posición del centroide.

Una vez recorrida toda la lista, cada hilo guarda los valores encontrados en memoria compartida y pasan a realizar una reducción tanto de las coordenadas como de los

contadores. Finalmente, un único hilo calcula la media de las nuevas coordenadas a partir del resultado obtenido de la reducción.

La CPU se encarga de comprobar si el algoritmo llegó a una convergencia, en caso contrario, vuelve a repetir todo el proceso de la ejecución de los dos kernels tal y como se especificó en la sección 5.2. También se especifica una cantidad máxima de iteraciones para prevenir el caso en el que el algoritmo no converge.

A continuación se explicará la evolución del algoritmo y el impacto que cada cambio tiene sobre su rendimiento.

5.3.1. Uso de memoria compartida

Como siempre, reducir la cantidad de accesos a la memoria global copiando los datos a la memoria compartida siempre que sea posible aumenta el rendimiento del algoritmo.

Si analizamos los dos kernels, el único en el que sus hilos comparten datos es en el caso del primer kernel, en el que todos los hilos de un bloque acceden a las mismas coordenadas de los centroides.

Se cambia el kernel para que los hilos del bloque se encarguen de cargar estas coordenadas a memoria compartida, reduciendo la cantidad de lecturas a memoria global.

5.3.2. Combinar las coordenadas en un vector

Los hilos son capaces de realizar peticiones de 4, 8 o 16 bytes de datos de memoria global con una sola instrucción[7]. Para realizar este tipo de peticiones, el lenguaje CUDA ofrece primitivos especiales para tal finalidad. Estos primitivos pueden ser tanto enteros como *int*, *int2* o *int4* cada uno de 4, 8 o 16 bytes respectivamente, como números de coma flotantes de doble precisión, *double* o *double2* para 8 o 16 bytes respectivamente. También se proporcionan otras primitivas como *int3* o *double4*, pero éstas no se realizan con una única instrucción.

Por ahora, los kernels utilizan listas separadas para cada una de las coordenadas, tanto para las observaciones como para los centroides. Se pueden combinar estas listas utilizando utilizando enteros dobles (*int2*), de los cuales, el primer elemento es la coordenada X y el segundo es la coordenada Y.

Este cambio reduce las instrucciones necesaria siempre que se quiera acceder a coordenadas a la mitad.

5.3.3. Mayor paralelización al calcular las nuevas medias de los centroides

Si tenemos en cuenta la implementación del segundo kernel explicada en la sección 5.3, se puede observar que si se tiene una cantidad pequeña de centroides, se lanzan pocos bloques, reduciendo la ocupación de la GPU. Normalmente no se utiliza un gran número de centroides, pero sí un gran número de observaciones, lo que conlleva a la creación de pocos bloques, pero que realizan mucha computación, siendo el punto más importante de paralelizar.

Para conseguir una mayor paralelización, se divide el kernel en dos nuevos kernels.

Para facilitar la explicación del funcionamiento del primer kernel, nos centraremos en los calculos que se realizan para un único centroide.

Anteriormente un único bloque se encargaba de realizar las sumas de todas las coordenadas del grupo de centroide que coincidía con su identificador. Con el nuevo primer kernel, primero se divide el trabajo en segmentos de tamaño m, donde m es el número máximo de bloques concurrentes que se pueden ejecutar en la GPU con el tamaño éstos prefijado. Ahora, cada bloque tiene como objetivo sumar todas las coordenadas, contar todas las ocurrencias de éste grupo, hacer la reducción de los resultado anteriores y guardar el valor en memoria global.

Es necesario utilizar un grid de doble dimensión, una dimensión utilizada para saber que segmento debe calcular cada bloque y una segunda dimensión para identificar sobre qué centroide debe realizar tales cálculos.

Finalmente, el segundo kernel realiza la suma y la reducción final, con un único hilo de cada bloque calculando las nuevas coordenadas de los centroides.

Esta optimización aumenta la paralelización enormemente, especialmente en casos con muchas observaciones y pocos centroides.

5.3.4. Iteración en la GPU

También se ha implementado el bucle principal del algoritmo en la GPU para estudiar su rendimiento, ya que de este modo no se tiene que transferir de la GPU las nuevas coordenadas para comprobar si el algoritmo llegó a una convergencia y se puede comprobar directamente en la GPU por un hilo.

5.4. Resultados

5.4.1. K-means

Debido a la dificultad de encontrar una lista más grande que converja en la misma cantidad de iteraciones que una lista menor (o que simplemente converja, ya que se puede dar el caso en el que no lo hace), los algoritmos se han ejecutado repetidas veces para tener el resultado final. Nótese que el problema es en la comparación de la misma implementación a medida que el tamaño de la lista crece y no entre optimizaciones.

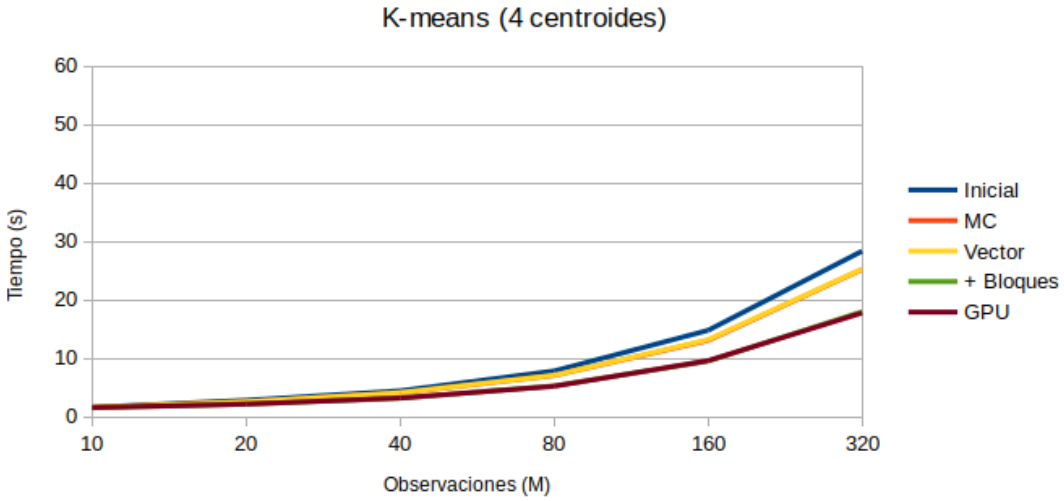


FIGURA 5.1: Comparación entre cada una de las optimizaciones implementadas con 4 centroides.

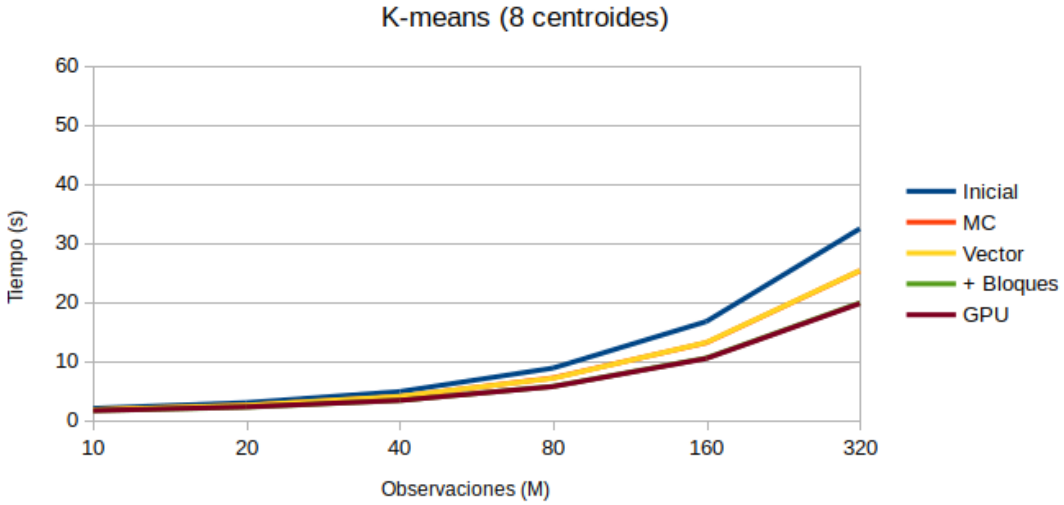


FIGURA 5.2: Comparación entre cada una de las optimizaciones implementadas con 8 centroides.

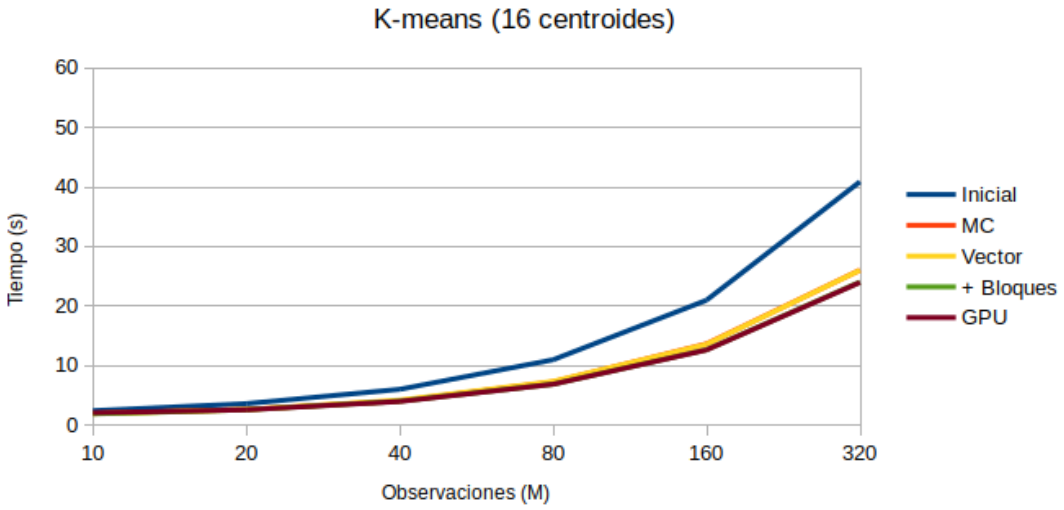


FIGURA 5.3: Comparación entre cada una de las optimizaciones implementadas con 16 centroides.

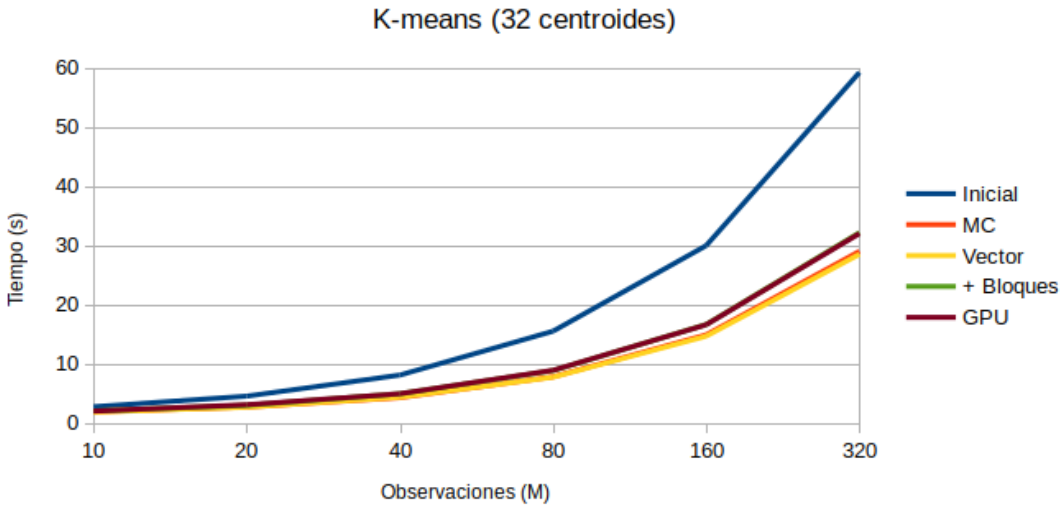


FIGURA 5.4: Comparación entre cada una de las optimizaciones implementadas con 32 centroides.

La utilización de primitivas vectoriales y la ejecución del algoritmo completamente en la GPU (3 y 5 respectivamente) presentan unas mejoras casi inapreciables en el rendimiento, incluso en el segundo caso, se dan ocasiones en las que tiene un rendimiento ligeramente inferior a la implementación anterior.

Utilizar memoria compartida para guardar las coordenadas de los centroides (2) presenta una gran mejora en el rendimiento que aumenta a medida que el número de centroides aumentan, con un speedup de 1.13, 1.28, 1.57 y 2 para 4, 8, 16 y 32 centroides, respectivamente y 320 millones de observaciones. Esto es debido a que se realizan n X k

comparaciones (por lo que, accesos a memoria) en cada iteración, donde n es el número de observaciones y k es el número de centroides.

Lo contrario pasa con la cuarta implementación, a medida que se aumenta el número de centroides, el speedup disminuye, llegando incluso a ser peor que otras implementaciones con 32 centroides. Esto es debido a que, sin éste cambio, si se tiene un número muy pequeño de centroides la ocupación también es baja, ya que se crearían un número muy bajo de bloques, pero con el cambio, se crean más bloques aumentando la ocupación, pero con el coste de que se debe realizar una comunicación final entre bloques que tienen asignado el mismo centroide. La comunicación compensa en casos con pocos centroides, pero si se tienen más, las implementaciones anteriores también crean más bloques, aumentando la ocupación, y sin la necesidad de comunicación entre bloques, supera en rendimiento a ésta implementación.

Menciono también que en la GPU que se han realizado los experimentos y con el tamaño de bloque elegido, la máxima cantidad de bloques concurrentes que se pueden ejecutar es de 32, ahí el buen rendimiento que se tiene en las implementaciones anteriores.

Los speedups que se consiguen son del 1.41, 1.28, 1.08 y 0.89 con 4, 8, 16 y 32 centroides respectivamente.

5.4.2. CPU vs GPU

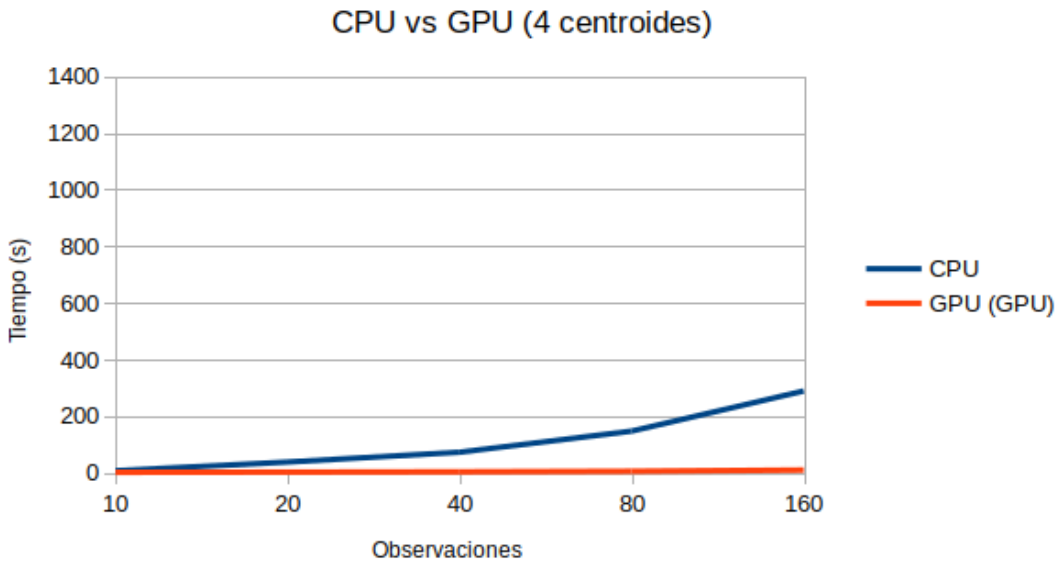


FIGURA 5.5: Comparación entre la implementación en CPU y GPU con 4 centroides.

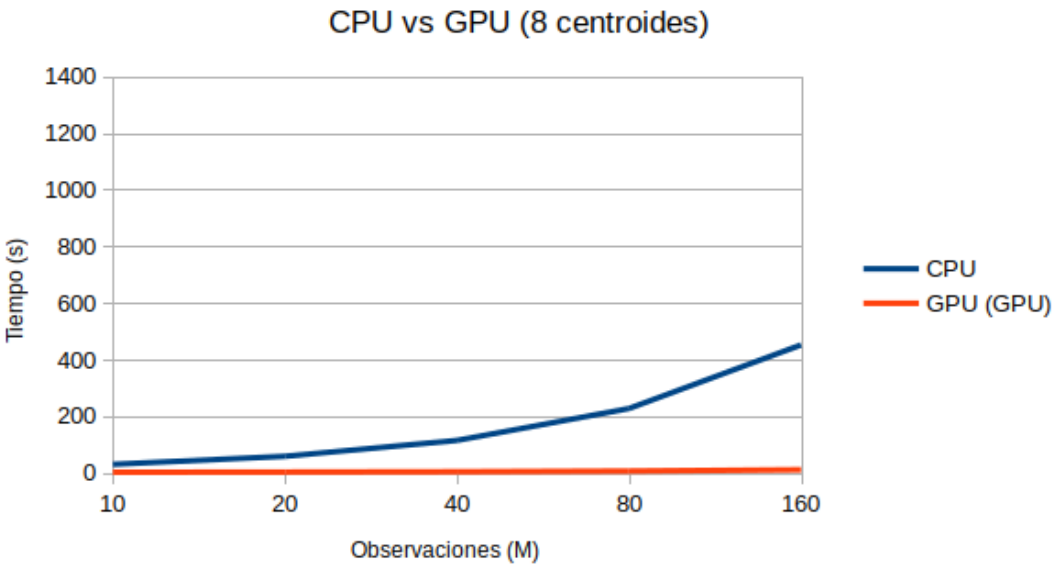


FIGURA 5.6: Comparación entre la implementación en CPU y GPU con 8 centroides.

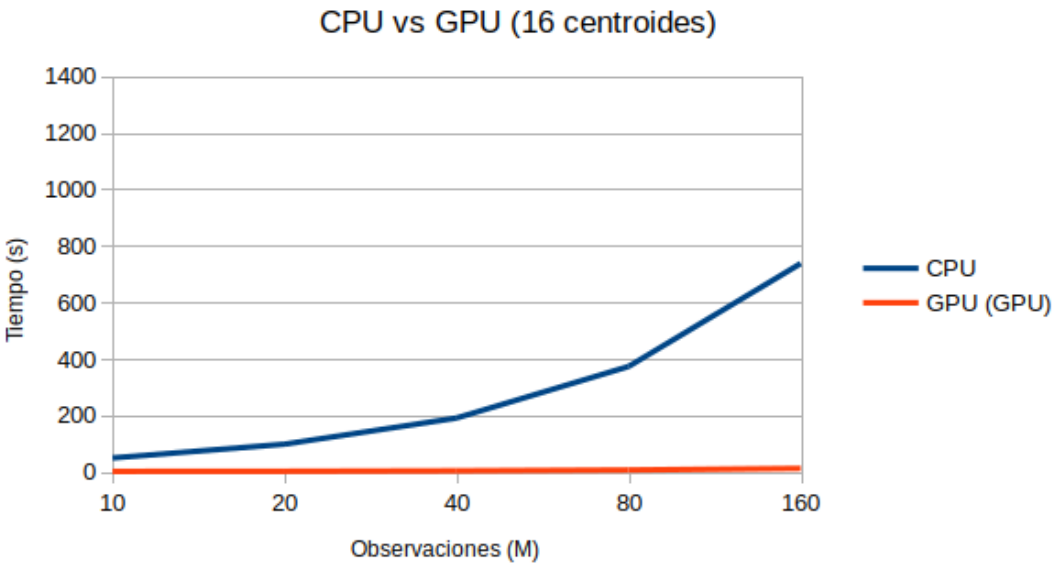


FIGURA 5.7: Comparación entre la implementación en CPU y GPU con 16 centroides.

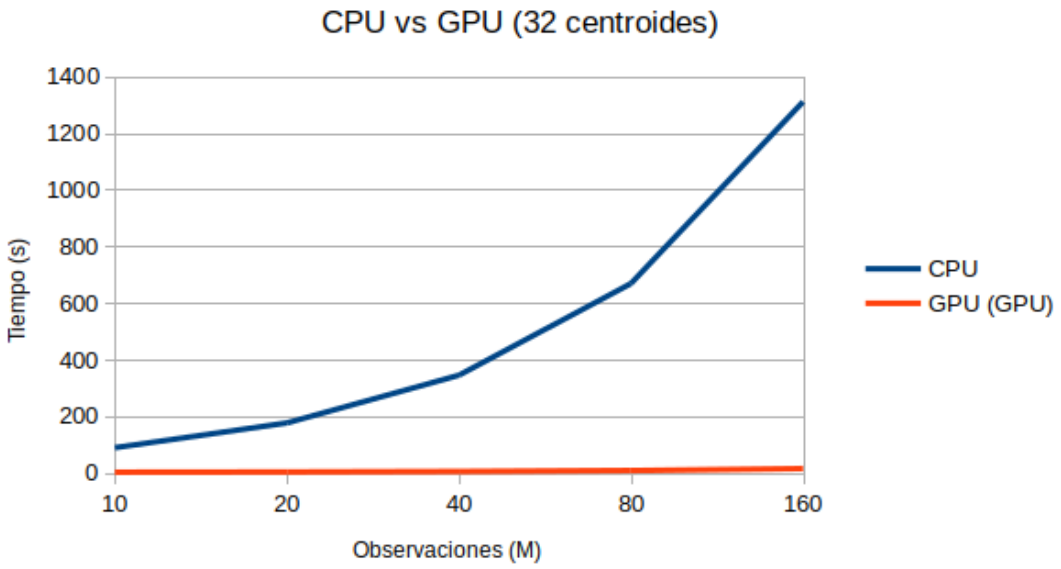


FIGURA 5.8: Comparación entre la implementación en CPU y GPU con 32 centroides.

Como se puede observar en las gráficas anteriores, el algoritmo en la GPU tiene un rendimiento muy alto comparado en el de la CPU. Los speedups van de 30, 43.4, 59 y 89.6 con 4, 8, 16 y 32 centroides respectivamente y 160 millones de observaciones. El speedup tiende a crecer a medida que se añaden más observaciones o más centroide.

Capítulo 6

N-Body + Leapfrog

6.1. Introducción

El algoritmo de N-Body aproxima la evolución de un sistema de cuerpos en el que todos los cuerpos interactúan con todos los demás utilizando algún tipo de fuerza. Estos cuerpos pueden variar enormemente en escala, desde átomos hasta cuerpos celestiales. Las aplicaciones son numerosas en áreas como la astrofísica, dinámica molecular, simulación de fluidos, entre otros.

La implementación que se realizará en este capítulo es llamada *all-pairs*, un método que calcula las fuerzas que actúan entre todos los cuerpos del sistema. El coste computacional es muy alto ($O(N^2)$), por lo que no se suele utilizar sólo, sino que en combinación con otros métodos. Su utilización generalmente se utiliza para calcular las fuerzas de cuerpos cercanos de un mismo grupo, entre cuerpos más lejanos se utilizan algoritmos más rápidos como el de Barnes-Hut[8].

El algoritmo N-Body únicamente calcula las fuerzas que se actúan, para realizar una simulación en el tiempo se utilizará el algoritmo Leapfrog. Leapfrog es un algoritmo de segundo orden muy popular si se desea una precisión modesta, y es utilizado debido a su menor coste en cuanto a computación. Si se desea mayor precisión se debería utilizar un algoritmo de mayor orden.

Estos dos algoritmos se ejecutarán en un bucle hasta que se desea parar la simulación.

6.2. Diseño

En este documento se calcula la aceleración de cada cuerpo a partir de la fuerza de atracción que realizan los demás cuerpos del sistema sobre éstos, con el fin de simular una atracción gravitacional.

Dado N cuerpos con la posición inicial x_i y una velocidad v_i para $1 \leq i \leq N$, la aceleración a_i sobre el cuerpo i realizada por el cuerpo j viene dada por la siguiente ecuación:

$$\vec{a_i} \approx \sum_{1 \leq j \leq N} \frac{m_j \vec{r_{ij}}}{\sqrt{(\|\vec{r_{ij}}\|^2 + \varepsilon^2)^3}} \tag{6.1}$$

donde m_i y m_j son las masas de los cuerpos i y j respectivamente, $\vec{r_{ij}} = x_j - x_i$ es el vector del cuerpo i al cuerpo j y ε^2 es un factor de suavizado utilizado en simulaciones de astrofísica para prevenir la gran aceleración que se realiza a medida que los cuerpos se acercan entre ellos.

Más información sobre la obtención de la ecuación 6.1 se puede encontrar en el libro online mencionado en la bibliografía[9].

Tal y como se menciona en la sección 6.1, se utiliza el algoritmo Leapfrog para la simulación en el tiempo de los cuerpos.

Dado $\vec{r_i}$ como posición inicial, $\vec{v_i}$ velocidad inicial, $\vec{a_i}$ aceleración inicial y dt como la constante de los intervalos de tiempo, la ecuación del algoritmo Leapfrog viene dada por:

$$\vec{r_{i+1}} = \vec{r_i} + \vec{v_i} dt + \vec{a_i} (dt)^2 / 2 \tag{6.2}$$

$$\vec{v_{i+1}} = \vec{v_i} + (\vec{a_i} + \vec{a_{i+1}}) dt / 2 \tag{6.3}$$

Para más información sobre la ecuación y el algoritmo véase el papel de Peter Young[10].

También se ofrece un script en Octave con el que se pueden ver los resultados del algoritmo utilizando el archivo proporcionado por éste. Se tiene que tener en cuenta que para que el algoritmo genere el archivo, se debe utilizar la opción -o.

6.3. Implemención

Inicialmente se explicará la implementación del algoritmo N-Body sin incluir la simulación en el tiempo utilizando Leapfrog. La razón de ello es la fácil explicación de la implementación del algoritmo paso a paso y la mejor visualización de las mejoras en rendimiento a medida que se realizan cambios.

6.3.1. N-Body

El algoritmo utiliza 3 listas, la primera para las coordenadas de las posiciones, la segunda para las aceleraciones y una última para las masas. Para las primeras 2 listas se necesita almacenar vectores de 3 elementos, por lo que se utiliza los vectores de coma flotante que el lenguaje CUDA proporciona al programador para facilitar la programación en este tipo de situaciones (*float3*). En caso contrario, se deberían utilizar un total de 7 listas.

La implementación del algoritmo es muy simple, cada hilo se encarga de calcular la aceleración final de un sólo cuerpo utilizando la ecuación descrita en 6.1, tal y como se muestra en la siguiente figura.

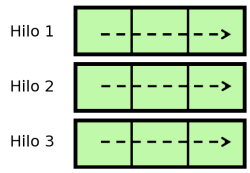


FIGURA 6.1: Cada fila representa el trabajo realizado por cada hilo, calculando secuencialmente la aceleración de un cuerpo.

Nótese que cada fila representa el acceso de múltiplos hilos a la misma lista de posiciones y no a una matriz, por la cual se puede confundir en la figura.

6.3.1.1. Mejorar la organización de las listas

La utilización de los vectores *float3* pueden facilitar la programación, pero engañan en cuanto a su rendimiento. Puede que parezca que los accesos a memoria se hagan de forma seguida, pero no es así, ya que un hilo sólo puede realizar peticiones de 4, 8 o 16 bytes, pero no de 12, por lo que se traduciría a 3 peticiones de 4 bytes. Ésto crea un acceso a memoria global con un desplazamiento de 3 posiciones entre hilos, tal y como se muestra en la figura 6.2.

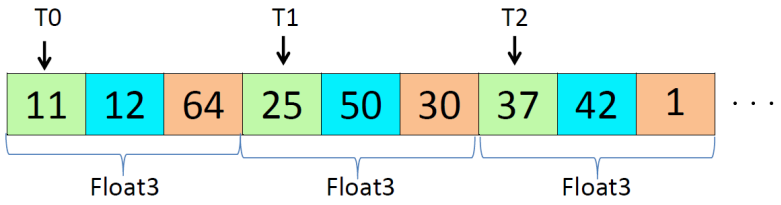


FIGURA 6.2: Accesos de los hilos a memoria en caso de utilizar *float3*.

Para mejorar el acceso, se combinan las lista de las posiciones con la lista de masas, creando listas de vectores *float4*. Éstos vectores son de 16 bytes, por lo que se compilan a una única instrucción y mejoran el rendimiento de transferencia de datos entre los hilos y la memoria global.

6.3.1.2. Memoria compartida

Cada hilo recorre toda la lista de posiciones, desde el principio al final, con la finalidad de calcular la aceleración final (véase figura 6.1). Esto requiere de muchos accesos a la misma dirección de memoria situada en la memoria global por parte de todos los ellos.

Si inicialmente, todos los hilos copian la lista a memoria compartida, se reduciría la cantidad de lecturas a memoria global de un mismo elemento al total de los bloques, en vez del total de los hilos. Desafortunadamente, la memoria compartida es limitada y no se pueden cargar listas muy grandes. Para solucionar éste problema, se divide la lista en segmentos más pequeños que sí se pueden copiar.

Inicialmente, todos los hilos de un bloque se encargan de copiar el primer segmento, de realizar los cálculos necesarios sobre éste y de forma iterativa se repite el proceso hasta finalizar con toda la lista. Se puede observar el procedimiento en la siguiente figura.

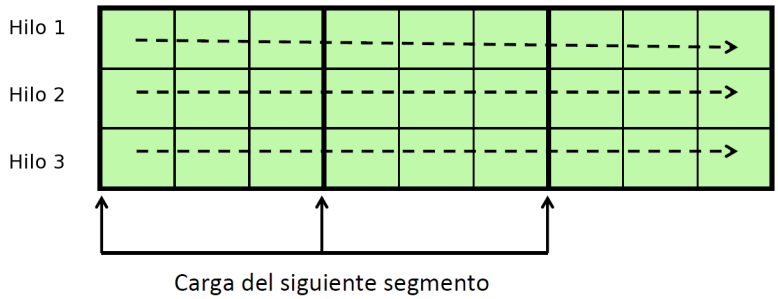


FIGURA 6.3: Un hilo realizando los cálculos sobre múltiplos segmentos. El bloque entero se encarga de cargar un segmento a memoria compartida en los puntos indicados pos las flechas.

6.3.1.3. Estructuras de listas (SoA)

En la sección 6.3.1.1 se explica como la utilización de las correctas primitivas vectoriales que ofrece el lenguaje CUDA puede mejorar el rendimiento y el acceso seguido a memoria compartida.

Pero las coordenadas de las aceleraciones siguen siendo listas de *float3*, traducidas a 3 instrucciones, empeorando el acceso seguido a memoria global. La utilización de *float4* sin aprovechar último elemento alargaría la lista un 25 % de lo actual, saturando el ancho de banda.

Ésta organización de datos, listas de estructuras (AoS), empeoran el rendimiento y da una falsa impresión de que los hilos acceden a direcciones seguidas de memoria, cuando no es el caso.

Para solucionar este problema, se crea una estructura de listas (SoA) para las aceleraciones. Esta estructura tendrá 3 listas teniendo todas las coordenadas de una misma dimensión en direcciones seguidas de memoria, mejorando el acceso a éstas por los hilos.

6.3.1.4. Aceleración hardware de operaciones aritméticas

Si utilizamos el NVIDIA Visual Profiler, podemos observar que el kernel realiza una gran cantidad de cálculo comparado con otras operaciones como el acceso a memoria o el control de flujo, véase figura 6.4. Optimizando este punto puede mejorar el rendimiento de forma considerable, pero no se puede cambiar la ecuación que se utiliza, 6.1.

También se puede observar que la utilización es del 65 % aproximadamente. Esto puede ser debido a dependencias de datos en los cálculos, ya que operaciones aritméticas como la división o la raíz cuadrada que utilizamos en nuestra ecuación requieren de muchos ciclos.

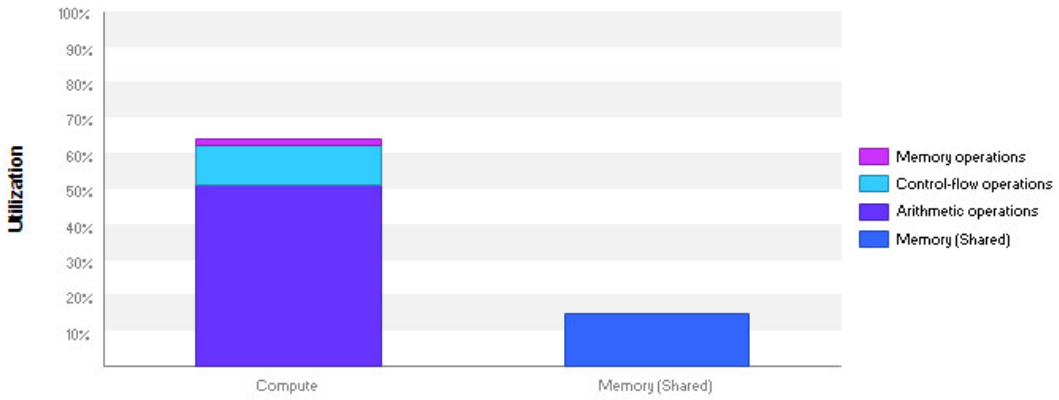


FIGURA 6.4: Gráfica de las operaciones realizadas en el multiprocesador sin la función *rsqrtf)*. La cantidad de operaciones aritméticas es mucho más alta que las demás tipos de operaciones.

Las GPUs de NVIDIA ofrecen funciones que se aprovechan de la aceleración por hardware para calcular ciertas operaciones aritméticas a cambio de una pequeña pérdida de precisión. Una de éstas funciones es la nombrada *rsqrtf*, que recibe un sólo parámetro n y devuelve el resultado de la ecuación 6.4. El compilador se encarga de traducir ésta función en instrucciones hardware que se aprovechan de tal aceleración hardware.

$$\frac{1}{\sqrt{n}} \tag{6.4}$$

Si se vuelve a la ecuación 6.1, se observa que utilizando la función *rsqrtf* se puede calcular la inversa de la raíz cuadrada por hardware, tal y como se muestra a continuación.

$$\vec{a_i} \approx \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\sqrt{(\|\vec{r_{ij}}\|^2 + \varepsilon^2)^3}} \tag{6.1}$$

$$\vec{a_i} \approx \sum_{1 \leq j \leq N} m_j \vec{r}_{ij} \frac{1}{\sqrt{(\|\vec{r_{ij}}\|^2 + \varepsilon^2)^3}} \tag{6.5}$$

$$\vec{a_i} \approx \sum_{1 \leq j \leq N} m_j \vec{r}_{ij} \quad rsqrtf(\quad (\|\vec{r_{ij}}\|^2 + \varepsilon^2)^3 \quad) \tag{6.6}$$

Utilizando ésta función se reduce enormemente el tiempo que los hilos pasan haciendo cálculos, ya que, dos de las operaciones aritméticas que más ciclos utilizan en la GPU (división y raíz cuadrada) se calculan en un menor tiempo. El resultado se puede observar en la figura 6.5, que muestra una utilización mucho más alta que la anterior implementación (figura 6.4) (casi un 90 %) debido al menor tiempo que se espera para una dependencia.

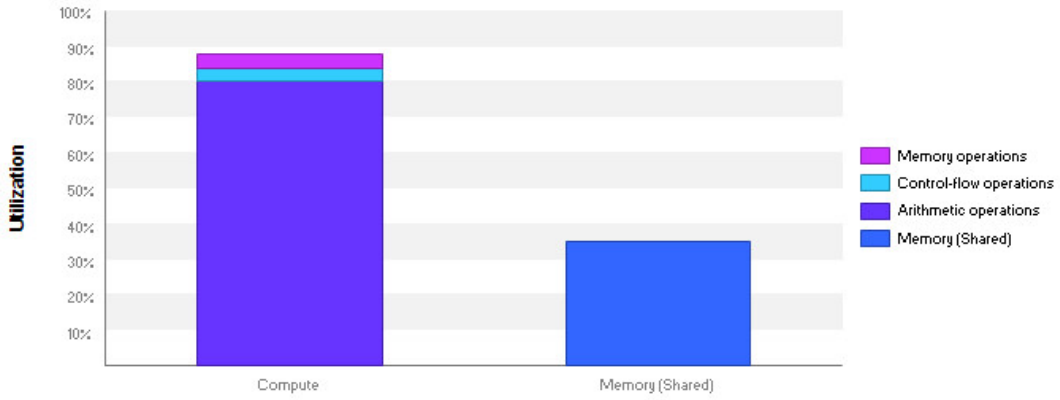


FIGURA 6.5: Gráfica de las operaciones realizadas en el multiprocesador utilizando la función *rsqrtf)*. La ocupación es muy alta.

6.3.1.5. Parámetros de compilación

En la sección anterior se explicó como la función *rsqrtf* utiliza la aceleración hardware para mejorar el rendimiento de los cálculos. El hardware puede realizar los cálculos que definen ésta función en una sola instrucción, pero si el parámetro de entrada que se utiliza es muy cercano a 0 (números denormales) o 0 su resultado es indeterminado. Por ello, el compilador genera unas instrucciones que comprueban si el parámetro presenta este problema, previniendo de que esto ocurra.

Pero en nuestro caso, si nos fijamos a la ecuación de la aceleración 6.1, se puede observar que nunca se dará tal caso especial, ya que se utiliza un factor de suavizado ε, pudiendo prescindir de las instrucciones de comprobación. Se le puede indicar al compilador que no genere tal instrucciones con el parámetro *-ftz=true*.

En la sección de resultados se puede observar la mejora en rendimiento.

6.3.2. Leapfrog

La paralelización es particularmente simple, cada hilo de un bloque se encarga de realizar las operaciones necesarias presentadas en las ecuaciones 6.2 y 6.3 de un cuerpo.

Una directa implementación del algoritmo tal y como se menciona en la sección de diseño (6.2) consiste en:

1. Calcular las aceleraciones iniciales ($\vec{a_0}$) de todos los cuerpos, ecuación 6.1.
2. Actualizar las posiciones, ecuación 6.2.

3. Utilizando las nuevas posiciones ($r_{i+1}^{\rightarrow}$), calcular $a_{i+1}^{\rightarrow}$, ecuación 6.1.

4. Actualizar las velocidades ($v_{i+1}^{\rightarrow}$), ecuación 6.3.

5. Volver al paso 2 hasta que se especifique el fin del algoritmo.

El algoritmo realiza k iteraciones, donde k es un parámetro de entrada del programa.

Si volvemos a la ecuación 6.3, se puede observar que se necesitarían 2 listas para almacenar las actualizaciones. Para prevenir la comunicación y gestión extra de tener dos listas, se realiza una serie de cambios.

Se define

$$v_{i+1}^{\rightarrow} = v_{i+1/2}^{\rightarrow} + a_{i+1}^{\rightarrow} dt/2, \tag{6.7}$$

donde

$$v_{i+1/2}^{\rightarrow} = \vec{v_i} + \vec{a_i} dt/2 \tag{6.8}$$

Ahora podemos cambiar $r_{i+1}^{\rightarrow}$ como

$$r_{i+1}^{\rightarrow} = \vec{r_i} + v_{i+1/2}^{\rightarrow} dt \tag{6.9}$$

Demostraciones:

$$\begin{aligned} r_{i+1}^{\rightarrow} &= \vec{r_i} + v_{i+1/2}^{\rightarrow} dt \\ &= \vec{r_i} + (\vec{v_i} + \vec{a_i} dt/2) dt \\ &= \vec{r_i} + \vec{v_i} dt + \vec{a_i} (dt)^2/2 \end{aligned} \tag{6.2}$$

$$\begin{aligned} v_{i+1}^{\rightarrow} &= v_{i+1/2}^{\rightarrow} + a_{i+1}^{\rightarrow} dt/2 \\ &= \vec{v_i} + \vec{a_i} dt/2 + a_{i+1}^{\rightarrow} dt/2 \\ &= \vec{v_i} + (\vec{a_i} + a_{i+1}^{\rightarrow}) dt/2 \end{aligned} \tag{6.3}$$

Con lo anteriormente mencionado, los pasos de la implementación del algoritmo son:

1. Calcular las aceleraciones iniciales ($\vec{a_0}$) de todos los cuerpos, ecuación 6.1.

2. Calcular $v_{i+1/2}^{\rightarrow}$, ecuación 6.8.

3. Actualizar las nuevas posiciones ($r_{i+1}^{\rightarrow}$) utilizando la nueva ecuación 6.9.
4. Utilizando las nuevas posiciones, calcular $a_{i+1}^{\rightarrow}$, ecuación 6.1.
5. Actualizar las velocidades ($v_{i+1}^{\rightarrow}$), ecuación 6.7.
6. Volver al paso 2 hasta que se especifique el fin del algoritmo.

Para el calculo de las actualizaciones se utiliza el kernel ya implementado en la sección anterior. Se implementan dos nuevos kernels para el cálculo de las posiciones y velocidades (se puede utilizar el mismo kernel para $v_{i+1/2}^{\rightarrow}$ y $v_{i+1}^{\rightarrow}$).

El bucle principal se realiza en la CPU, ya que al final de cada iteración se da la opción de guardar las posiciones en un fichero utilizado para ver los resultados utilizando el script de Octave, cosa que la GPU no puede realizar.

6.4. Resultados

6.4.1. N-Body

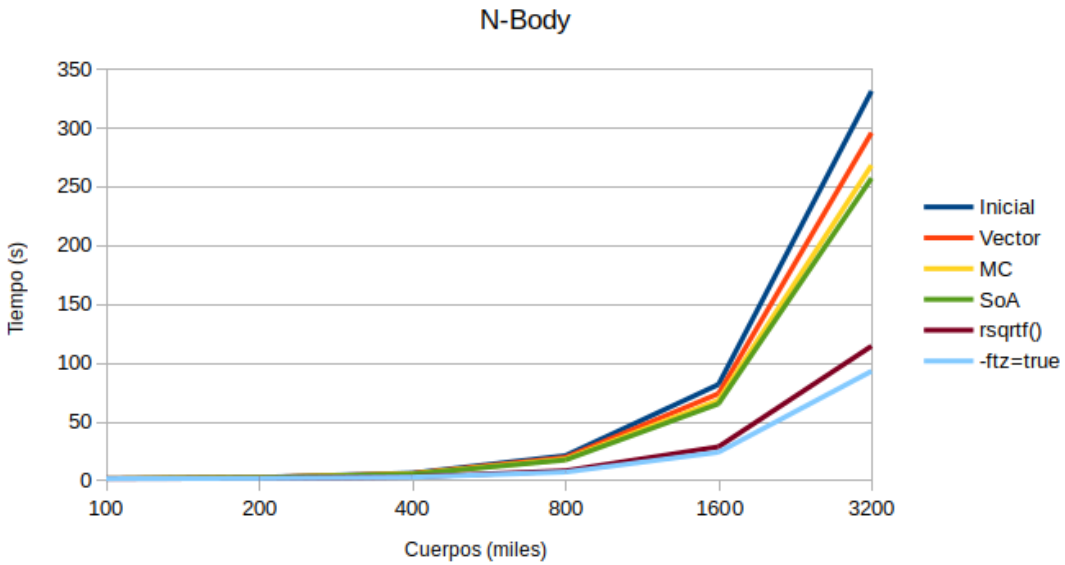


FIGURA 6.6: Diferencia de tiempos entre las optimizaciones del algoritmo N-Body.

En el caso de aprovechar las primitivas vectoriales, el algoritmo presenta un speedup de 1.12 frente a la implementación inicial, para el cómputo de 3.2 millones de cuerpos. Con la idea de ir añadiendo segmentos a la memoria compartida para reducir el acceso a memoria global se ha aumentado el rendimiento en un 1.1 de speedup. La reordenación

de la forma en la que se almacenan en memoria los cuerpos tiene como resultado un speedup de 1.04. La diferencia más grande se da en el caso de utilizar de la función *rsqrtf*, con un speedup de 2.25 frente a la implementación anterior. Finalmente, en el caso de prescindir de las comprobaciones innecesarias generadas por el compilador, el speedup conseguido es del 1.23.

El speedup mencionado en el caso de la utilización de la función *rsqrtf* es debido a que el algoritmo realiza una gran cantidad de divisiones y raíces cuadradas, realizar ese cálculo utilizando la función *rsqrtf* reduce enormemente el tiempo dedicado para la computación de éstas (véase 6.3.1.4).

También se consigue un speedup importante con un simple parámetro para el compilador.

6.4.2. N-Body + Leapfrog

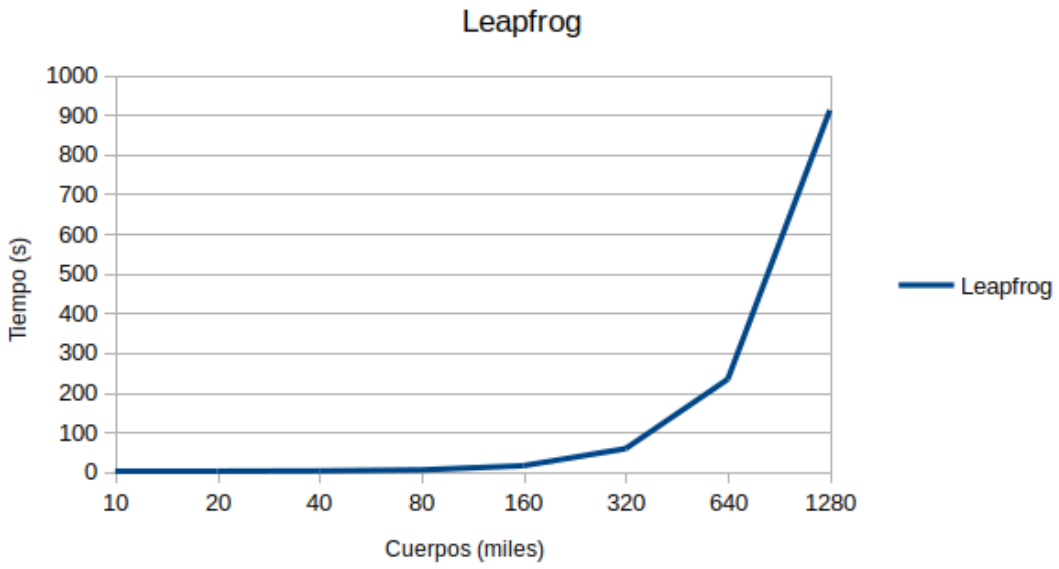


FIGURA 6.7: Diferencia de tiempos entre las optimizaciones del algoritmo N-Body + Leapfrog con 100 iteraciones.

Los resultados son positivos considerando que se han realizado 100 iteraciones, lo que implica la ejecución de 100 veces de los dos algoritmos.

6.4.3. CPU vs GPU

La CPU tiene un resultados muy pobres en comparación con los resultados de la GPU. Comparando las implementaciones con 160 y 320 mil cuerpos, la que ofrece un speedup

de 365 y 723 respectivamente, una enorme diferencia. Las diferencias entre los resultados vienen dados por el gran nivel de paralelización que ofrece el algoritmo N-Body.

No se presenta la comparación del algoritmo Leapfrog debido a las grandes diferencias de tiempos.

Capítulo 7

Floyd-Warshall

7.1. Introducción

Floyd-Warshall es un algoritmo utilizado para encontrar las distancias mínimas entre todos los pares de vértices de un grafo dirigido y ponderado. Con una pequeña modificación del algoritmo, que se tiene en cuenta en la implementación, es posible obtener el camino mínimo entre todos los pares de vértices también.

Éste algoritmo es reconocido actualmente por la publicación de Robert Floyd en 1962[11], pero es igual a publicaciones anteriores realizadas por Bernard Roy en 1959[12] y Stephen Warshall en 1962[13].

En comparación con el algoritmo de Dijkstra, Floyd-Warshall es una mejor elección sólo en casos en los que se dispongan de un grafo denso, donde $|E|$ es mayor que $|V|^2$, ya que se tiene una complejidad computacional de $O(|V|^3)$ y Dijkstra de $O(|V| \cdot |E| \, log \, |V|)$.

7.2. Diseño

Se utiliza una matriz D^0 como parámetro de entrada. Si existe una arista entre dos vértices i y j, la matriz contiene el peso de éstos en las coordenadas (i,j) y en caso contrario, contiene un infinito positivo. La diagonal de ésta contiene el valor cero.

La idea principal del algoritmo consiste en coger un vértice intermedio k y actualizar la distancia entre todos demás pares restantes de vértices (i,j) que pasan por éste si esta distancia es menor. Este proceso se repite n veces $(1 < k < n)$, donde n es la cantidad de vértices, para cada uno de éstos vértices.

La siguiente fórmula describe el proceso anteriormente descrito:

$$D_{ij}^{k} = min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{n-1}) \tag{7.1}$$

Para el resultado de los caminos mínimos se utiliza otra matriz P que inicialmente contiene infinito positivo en las coordenadas (i,j) si no hay aristas que conecte los vértices i y j o j si la hay. A cada iteración n (vértice intermedio) se actualiza el valor de las coordenadas (i,j) con el valor de la la coordenada (k,j) solo si $D_{ij}^{n-1} > D_{ik}^{n-1} + D_{kj}^{n-1}$, tal y como se describe en la siguiente ecuación:

$$P_{ij}^{k} = \begin{cases} P_{kj}^{k-1}, & D_{ij}^{k-1} > D_{ik}^{k-1} + D_{kj}^{n-1} \\ P_{ij}^{k-1}, & en \quad caso \quad contrario \end{cases} \tag{7.2}$$

7.3. Implementación

La implementación en serie se puede realizar con 3 bucles anidados que, cada uno de ellos recorren todos los vértices, el primero para k, seguido para el índice i y el último para el índicej. Para la implementación en paralelo se debe estudiar qué bucles se pueden realizar en paralelo.

No es posible paralelizar el bucle exterior ya que todas las iteraciones dependen de las anteriores, pero sí es posible paralelizar los otros dos bucles. Por lo que, el bucle exterior lo debe realizar la CPU, creando un nuevo kernel para cada k, $1 < k < n$ y los demás se realizan en la GPU.

La inicialización de un kernel tiene una sobrecarga adicional, ya que la CPU debe comunicar a la GPU de todos los parámetros necesarios. Pero teniendo en cuenta que ésta inicialización se realiza de forma asíncrona y que todos los kernels de un mismo stream se ejecutan en serie, la CPU va encolando todas las inicializaciones en el mismo stream, dejando que el planificador de la GPU se encargue de su ejecución en serie. De éste modo, mientras la GPU realiza los cálculos de los primeros kernels, la CPU va encolando más, escondiendo la sobrecarga de la creación de éstos.

En la GPU, cada hilo realiza el cálculo de una sola coordenada, que viene calculada a partir de su propio identificador y el del bloque. Para facilitar el cálculo de la GPU, se crean bloques en 2 dimensiones de tamaño (n, n / b), donde n es la cantidad de vértices y b es la cantidad de hilos por bloque. Por lo que, las coordenadas i y j de cada hilo vienen dadas por el identificador de la primera dimensión del bloque y por el cálculo $blockIdx.y \cdot blockDim.x + threadId$ respectivamente, donde $blockIdx.y$ es el identificador

de la segunda dimensión del bloque, *blockDim.x* es la cantidad total de bloques en la primera dimensión (n) y *threadId* es el identificador del hilo.

7.3.1. Reducir la cantidad de bloques

A medida que la cantidad de vértices crece, la cantidad de bloques que se crean crece también, superando la cantidad máxima que la GPU puede ejecutar de forma concurrente. El planificador de bloques de la GPU se encarga de la ejecución de éstos, pero una gran cantidad de ellos puede reducir el rendimineto debido a la gran cantidad de cambios de contexto. Por ello, en vez de asignar una coordenada a un hilo, se asigna una fila entera a un bloque, tal y como se muestra en la figura 7.1.

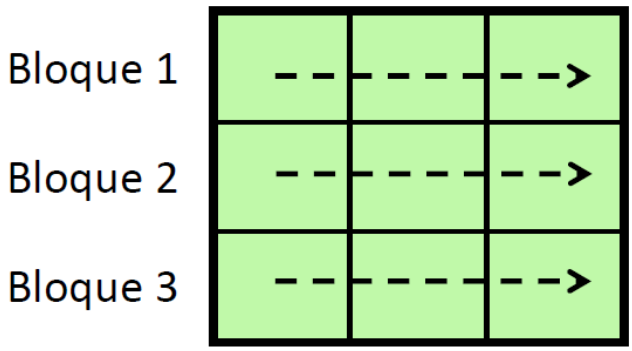


FIGURA 7.1: Cada bloque se encarga de calcular una única fila.

De éste modo reduciremos la cantidad de bloques que se crea asignando más trabajo a cada hilo. También simplifica el cálculo de las coordenadas por parte de los hilos, ya que ésta vez solo se necesitan n bloques en una única dimensión.

7.3.2. Reducir acceso a memoria global

Si volvemos a la ecuación 7.1, se puede observar que a cada iteración k, los elementos de las filas y columnas k se acceden repetidamente, n veces cada uno para ser exactos. Éstos accesos se realizan a memoria global, por lo que disminuyen el rendimiento, y con la implementación actual, un único valor que se puede aprovechar por bloque, el de la fila.

Lo ideal sería que cada hilo realizara todos los cálculos de una columna o fila (preferiblemente de una columna por los accesos consecutivos a memoria), de éste modo, cada uno

de ellos podría guardar el valor correspondiente con su columna y iteración, reduciendo el acceso a memoria global. Pero es difícil llegar a una ocupación alta de la GPU, y en algunos casos, como el nuestro, imposible.

Ésto es debido a que se necesita de una cantidad de vértices muy altas para que cada uno de los hilos tenga asignado cálculo. En nuestro caso, se dispone de 16 multiprocesadores de 2048 hilos cada uno, un total de 32768 hilos. Se necesitaría de la misma cantidad de vértices para tener todos los hilos ocupados. Si se tiene en cuenta que se necesita 2 matrices de nXn y que los elementos son enteros de 4 bytes, la cantidad total de memoria que necesitamos para almacenar éstas matrices es de 8GB, y en nuestro caso solo se dispone de 4 GB, por lo que hace imposible llegar a una ocupación mayor del 50 % con tal implementación.

Para aprovechar la idea, lo que sí es más viable, es asignar más de una fila a cada bloque en vez de una como en el caso anterior, y se dividen las columnas en segmentos. De éste modo, cada hilo de un bloque carga el elemento correspondiente a él del primer segmento y realiza los cálculos de todas las filas. El proceso se repite has que no queden más segmentos y se puede observar en la figura 7.2.

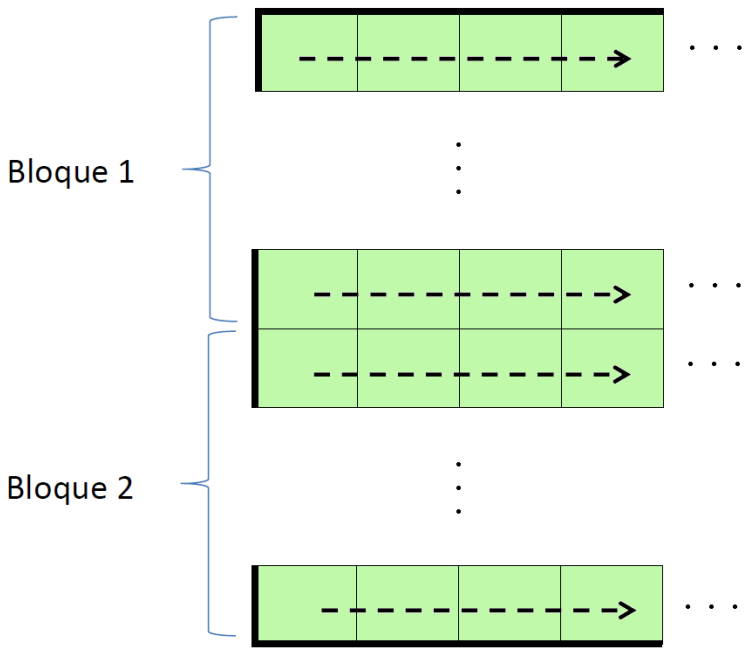


FIGURA 7.2: Cada bloque se encarga de calcular múltiplas filas.

7.4. Resultados

7.4.1. Floyd-Warshall

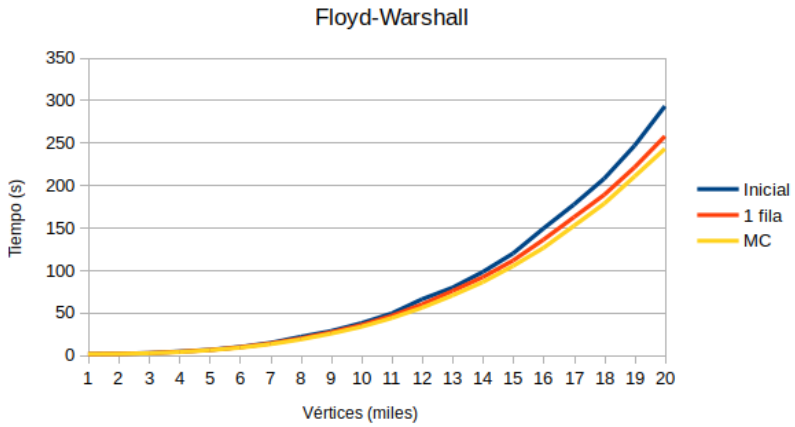


FIGURA 7.3: Comparación entre cada una de las optimizaciones implementadas del algoritmo Floyd-Warshall.

El algoritmo tiene una complejidad computacional de $O(N^3)$, por lo que, añadir más vértices aumenta el tiempo de ejecución del algoritmo consideradamente. Por ello, no se han realizado pruebas con mayor número de vértices, pero se puede observar la tendencia del tiempo que tarda en la gráfica.

El speedup obtenido al comparar la implementación inicial respecto a la segunda (reducir la cantidad de bloques que se crean 7.3.1) es del 1.14 con una cantidad de 20 mil cuerpos.

En el caso del uso de memoria compartida no se ha obtenido un speedup considerable (un 1.06 frente a la implementación anterior), ya que el aprovechamiento de los datos de ésta memoria en reducido.

7.4.2. CPU vs GPU

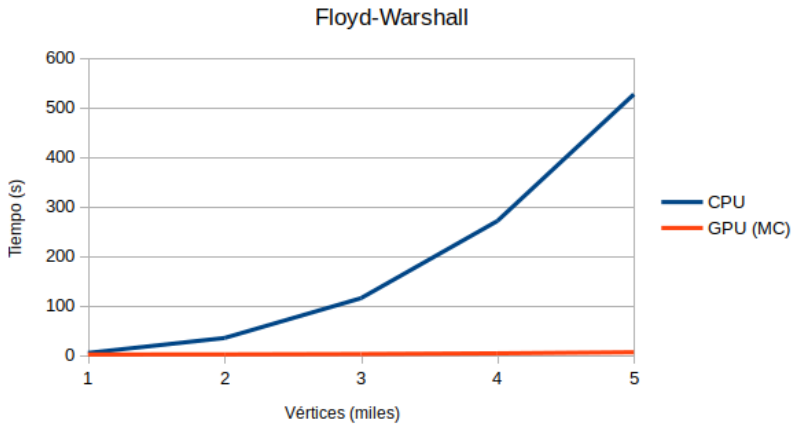


FIGURA 7.4: Comparación entre la implementación en CPU y GPU del algoritmo Floyd-Warshall.

Floyd-Warshall es un algoritmo muy costoso, pero muy paralelizable, y esto se puede observar en el gráfico anterior. Los speedups son del 3.86, 26, 55.13, 78.37 y 92.94 para una cantidad de vértices de 1, 2, 3, 4 y 5 mil respectivamente. A medida que se añaden más vértices, el speedup aumenta.

Capítulo 8

Monte Carlo

8.1. Introducción

Monte Carlo es un algoritmo... El nombre proviene del casino de Monte Carlo por ser "la capital del juego al azar".

Las probabilidades de los diferentes números obtenidos al tirar dos dados en un simple ejemplo y popular. Consiste en tirar los dados una elevada cantidad de veces y contar las ocurrencias de cada valor de las sumas de los dados. Si finalmente se cuentan los resultados, se podría observar que los valores 2 y 12 son los valores que menor ocurrencia, ya que se puede representar con una única combinación, (1,1) y (6,6) respectivamente. El valor 7 tendrá la mayor ocurrencia, ya que puede ser representado por muchas más combinaciones.

En la implementación se utilizará éste algoritmo para aproximar el valor de π.

8.2. Diseño

Tal y como se ha descrito en la sección de introducción, se utilizará para aproximar el valor de π.

Supongamos que tenemos un círculo de radio R dentro de un cuadrado y tocando los 4 bordes, cada borde con una longitud de *2R*. Si, de forma aleatoria, se dibujan puntos en éste cuadrado, se podría aproximar π contando los puntos que se han dibujado dentro del círculo y los de fuera.

Para el cálculo, primero tenemos que tener en cuenta que el área del círculo es $A_{círculo} = \pi R^2$ y el área del cuadrado es $A_{cuadrado} = (2R)^2 = 4R^2$. Si calculamos el ratio entre las dos áreas obtenemos $\pi/4$.

$$ratio = \frac{\pi R^2}{4R^2} = \pi/4 \tag{8.1}$$

Finalmente podemos estimar π como:

$$\pi = 4 * R_{círculo}/R_{cuadrado} \tag{8.2}$$

donde $R_{círculo}$ es la cantidad de números dentro del círculo y $R_{cuadrado}$ es la cantidad de número en el cuadrado.

El algoritmo recibe como parámetro de entrada la cantidad de números aleatorios que se desean crear.

8.3. Implementación

Inicialmente, la CPU calcula la máxima cantidad de bloques que se pueden ejecutar de forma concurrente en la GPU y, a partir del parámetro de entrada que el usuario proporciona al programa y la cantidad de hilos totales, calcula cuántos números aleatorios debe generar cada hilo, distribuyendo el trabajo de forma igual a cada uno de éstos.

La generación de número aleatorios se realizan en la GPU gracias a las librerías cuRAND. Para que los hilos no generen los mismos números aleatorios, ya que la semilla es generada por la CPU y posteriormente enviada a la GPU, a partir del identificador de hilo y el identificador del bloque, se calcula un identificador único para cada hilo que se utiliza junto con la semilla proveniente de la CPU para generar secuencias diferentes en cada hilo.

Una vez proporcionada la semilla a la librería cuRAND, cada hilo genera de forma aleatoria coordenadas que estén dentro del cuadrado especificado en la sección 8.2 y comprueba si éstas coordenadas se encuentran dentro del círculo también, si es así, se aumenta un contador. Este proceso se repite la cantidad de veces especificada por la CPU. Se considera que las coordenadas están dentro del círculo si:

$$X^2 + Y^2 < R^2, \tag{8.3}$$

donde X y Y son las coordenadas generadas y R es el radio del círculo.

Una vez acabada la generación de los números aleatorios y contado todos los puntos de dentro del círculo, se realiza una reducción por bloque en la que se van sumando los contadores de cada uno de los hilos de un bloque. Cada bloque guarda el resultado en memoria global.

Finalmente, la CPU recoge éstos resultados, realiza la última reducción, y calcula la aproximación de π a partir de la ecuación 8.3.

Se ha implementado dos formas distintas de hacer la reducción de la GPU, una utilizando memoria compartida y otra utilizando las funciones *suffle* mencionadas en la sección 3.3.2.2.

También se ha implementado una versión que funciona con números mayores a 2^{32} ya que, para generar 2^{32} números aleatorios el algoritmo tarda alrededor de 1 segundo, del cuál, la mayoría son inicializaciones.

8.4. Resultados

8.4.1. MonteCarlo

Los resultados con números menores a 2^{32} oscilan entre 0.9 y 1 centésima de segundo en las tres implementaciones, sin aparente diferencia de rendimiento entre éstas.

Ésto es debido a que la ejecución del kernel es muy rápida en generar una cantidad menor a 2^{32} números aleatorios, y la inicialización del algoritmo y la comunicación entre la CPU y la GPU es mucho más lenta.

A continuación se muestra el rendimiento del algoritmo a partir de 10 mil millones de números aleatorios generados.

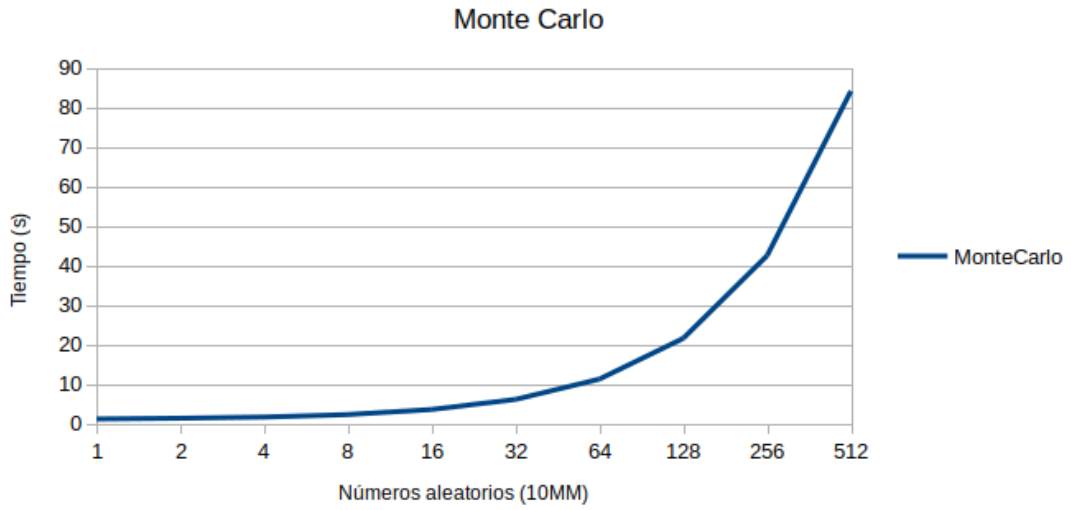


FIGURA 8.1: Rendimiento del algoritmo MonteCarlo en la GPU.

Se puede observar que a medida que la cantidad de números aleatorios generados dobla, el tiempo del algoritmo también dobla, pero esto es solo apreciable a partir de una gran cantidad de éstos números, cuando el algoritmo supera en tiempo la inicialización de éste.

8.4.2. CPU vs GPU

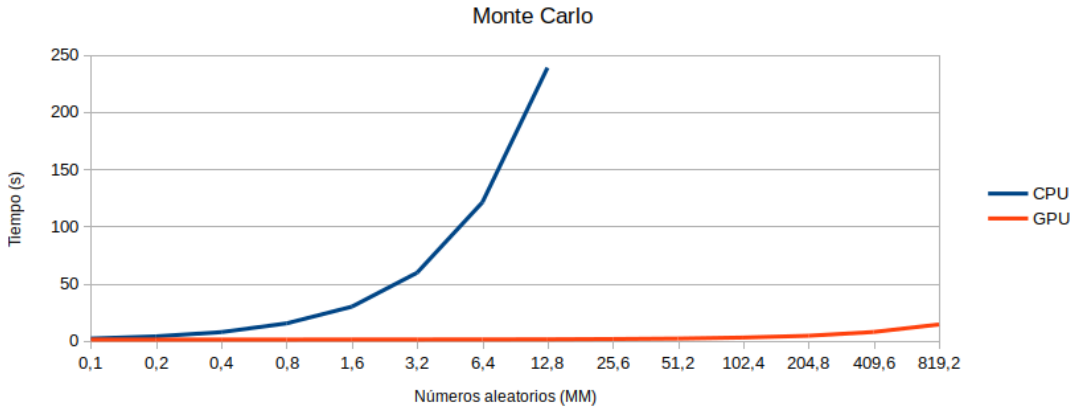


FIGURA 8.2: Comparación de rendimiento del algoritmo MonteCarlo entre la CPU y GPU.

La diferencia entre el tiempo de ejecución comparando la CPU y GPU es abismal, y esto es normal, ya que las GPUs disponen de un número muy elevado de hilos (32768 en la que se utilizó), todos dedicados a la generación de números aleatorios en comparación con uno solo de la CPU.

Para hacerse la idea, la CPU tarda 238.62 segundos en generar y calcular π con 6.4 mil millones de número. La GPU tarda 1.17 segundos (speedup de 203.9), de los cuales, la mayor parte del tiempo es la inicialización del kernel y la transferencia de los resultados de la GPU a la CPU.

Capítulo 9

Conclusiones

Normalmente las GPUs se utilizan para computación gráfica, pero también son una alternativa muy buena para programas con alto nivel de paralelización. Su uso en ámbitos como la ciencia, simulación o inteligencia artificial resulta muy atractivo.

Si bien el hardware actual tiene un gran potencial para la computación de general, es necesario desarrollar un programa que permita explotarlo adecuadamente, y esto conlleva un estilo de programación completamente diferente al habitual.

La plataforma CUDA, junto con su gran cantidad de librerías, facilita el desarrollo de éstos programas y, herramientas como NVIDIA Visual Profiler, ayudan a estudiar su comportamiento para entender y a mejorar su rendimiento.

En este documento se ha estudiado y implementado varios algoritmos populares para la GPU con unos resultados muy buenos en comparación con la versión secuencial ejecutada en la CPU. También se ha demostrado la importancia de conocer el hardware, primero para implementar de forma correcta el algorimo y segundo, realizar las optimizaciones necesarias para aumentar el rendimiento lo máximo posible.

Apéndice A

Kernels Quicksort

```
__device__ void compareAndSwapIfNecessary(int *in_array, int *out_array,
    uint my_idx, uint compare_idx, const int array_size)
{
        int my_value, compare_value;

        if (compare_idx < array_size)
        {
                my_value = in_array[my_idx];
                compare_value = in_array[compare_idx];

                if (my_value > compare_value)
                {
                        out_array[my_idx] = compare_value;
                        out_array[compare_idx] = my_value;
                }
        }
}

__global__ void bitonicSortInitialStep(int *g_in_array, int *g_out_array,
     uint size, const int array_size)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        uint my_idx = 2 * idx - (idx & ((size >> 1) - 1));
        uint compare_idx = my_idx ^ (size - 1);

        compareAndSwapIfNecessary(g_in_array, g_out_array, my_idx,
    compare_idx, array_size);
}

__global__ void bitonicSortLastStep(int *g_in_array, int *g_out_array,
    uint stride, const int array_size)
```

```
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        uint my_idx = idx + (idx & ~(stride - 1));
        uint compare_idx = my_idx + stride;

        compareAndSwapIfNecessary(g_in_array, g_out_array, my_idx,
    compare_idx, array_size);
}

__device__ void bigBitonicSortStream(int *g_in_array, int *g_out_array,
    const int array_size, cudaStream_t stream)
{
        const dim3 gridSize((hibit(array_size - 1) + blockDim.x - 1) /
    blockDim.x, 1, 1);

        for (uint size = 2; size < array_size << 1; size <<= 1)
        {
                bitonicSortInitialStep<<<gridSize, blockDim.x, 0, stream
    >>>(g_in_array, g_out_array, size, array_size);

                for (uint stride = size >> 2; stride > 0; stride >>= 1)
                {
                        bitonicSortLastStep<<<gridSize, blockDim.x, 0,
    stream>>>(g_in_array, g_out_array, stride, array_size);
                }
        }
}

__device__ void bigBitonicSort(int *g_in_array, int *g_out_array, const
    int array_size)
{
        const dim3 gridSize((hibit(array_size - 1) + blockDim.x - 1) /
    blockDim.x, 1, 1);

        for (uint size = 2; size < array_size << 1; size <<= 1)
        {
                bitonicSortInitialStep<<<gridSize, blockDim.x>>>(
    g_in_array, g_out_array, size, array_size);

                for (uint stride = size >> 2; stride > 0; stride >>= 1)
                {
                        bitonicSortLastStep<<<gridSize, blockDim.x>>>(
    g_in_array, g_out_array, stride, array_size);
                }
        }
}
```

```
__global__ void bitonicSort(int *g_in_array, int *g_out_array, const int
    array_size)
{
        extern __shared__ int s_array[];
        int my_idx, compare_idx;
        int tid = threadIdx.x;

        if (tid < array_size)
                s_array[tid] = g_in_array[tid];
        if (tid + blockDim.x < array_size)
                s_array[tid + blockDim.x] = g_in_array[tid + blockDim.x];
        __syncthreads();

        my_idx = 2 * tid;
        compare_idx = my_idx + 1;

        compareAndSwapIfNecessary(s_array, s_array, my_idx, compare_idx,
    array_size);

        for (int i = 4; i < array_size << 1; i <<= 1)
        {
                my_idx = 2 * tid - (tid & ((i >> 1) - 1));
                compare_idx = my_idx ^ (i - 1);

                compareAndSwapIfNecessary(s_array, s_array, my_idx,
    compare_idx, array_size);
                __syncthreads();

                for (int j = i >> 2; j > 0; j >>= 1)
                {
                        my_idx = tid + (tid & ~(j - 1));
                        compare_idx = my_idx + j;

                        compareAndSwapIfNecessary(s_array, s_array,
    my_idx, compare_idx, array_size);
                        __syncthreads();
                }
        }

        if (tid < array_size)
                g_out_array[tid] = s_array[tid];
        if (tid + blockDim.x < array_size)
                g_out_array[tid + blockDim.x] = s_array[tid + blockDim.x
    ];
}

__device__ int loadValues(int *in_array, int *out_array, const int
    array_size)
```

```
{
        const int tid = threadIdx.x;

        for (int i = tid; i < array_size; i += blockDim.x)
                out_array[i] = in_array[i];

        __syncthreads();

        return out_array[0];
}

__device__ int2 countPivot(int *array, const int array_size, const int
    pivot)
{
        const int tid = threadIdx.x;
        int ltPivot = 0, gtPivot = 0;

        for (int i = tid; i < array_size; i += blockDim.x)
        {
                if (array[i] < pivot)
                        ltPivot++;
                else
                        gtPivot++;
        }

        return make_int2(ltPivot, gtPivot);
}

__device__ int2 warpPrefixSum(int2 data, int lane_id)
{
        for (int i = 1; i < warpSize; i <<= 1)
        {
                int value_x = __shfl_up(data.x, i);
                int value_y = __shfl_up(data.y, i);

                if (lane_id >= i)
                {
                        data.x += value_x;
                        data.y += value_y;
                }
        }

        return data;
}

__device__ int2 prefixSum(int *s_ltPivot_scan, int *s_gtPivot_scan, int2
    pivot_count)
{
```

```
int tid = threadIdx.x;

int lane_id = tid & (warpSize - 1);
int warp_id = tid / warpSize;
int2 pivot_scan;

pivot_scan = warpPrefixSum(pivot_count, lane_id);

if ((tid & (warpSize - 1)) == (warpSize - 1))
{
        s_ltPivot_scan[warp_id] = pivot_scan.x;
        s_gtPivot_scan[warp_id] = pivot_scan.y;
}

__syncthreads();

if (warp_id == 0)
{
        int2 value_scan;

        value_scan.x = s_ltPivot_scan[lane_id];
        value_scan.y = s_gtPivot_scan[lane_id];

        value_scan = warpPrefixSum(value_scan, lane_id);

        s_ltPivot_scan[lane_id] = value_scan.x;
        s_gtPivot_scan[lane_id] = value_scan.y;
}

__syncthreads();

if (warp_id > 0)
{
        pivot_scan.x += s_ltPivot_scan[warp_id - 1];
        pivot_scan.y += s_gtPivot_scan[warp_id - 1];
}

pivot_scan.x = __shfl_up(pivot_scan.x, 1);
pivot_scan.y = __shfl_up(pivot_scan.y, 1);

if (lane_id == 0)
{
        if (warp_id > 0)
        {
                pivot_scan.x = s_ltPivot_scan[warp_id - 1];
                pivot_scan.y = s_gtPivot_scan[warp_id - 1];
        }
        else
```

```
                {
                        pivot_scan.x = 0;
                        pivot_scan.y = 0;
                }
        }

        return pivot_scan;
}

__device__ void reorderValues(int *in_array, int *out_array, const int
    array_size, const int pivot, int lt_offset, int gt_offset)
{
        const int tid = threadIdx.x;
        int ltPivot = 0, gtPivot = 0;

        //The pivot is added in between both lt/gt arrays at the 1st
    iteration by the 1st thread
        for (int i = tid; i < array_size; i += blockDim.x)
        {
                int value = in_array[i];
                if (value < pivot)
                {
                        out_array[lt_offset + ltPivot] = value;
                        ltPivot++;
                }
                else
                {
                        out_array[gt_offset + gtPivot] = value;
                        gtPivot++;
                }
        }

        __syncthreads();
}

__global__ void perSegmentQuicksort(int *g_in_array, int *g_out_array,
    const int array_size, const int depth)
{
        const int tid = threadIdx.x;

        extern __shared__ int s_mem[];
        int *s_ltPivot_scan = s_mem;
        int *s_gtPivot_scan = s_ltPivot_scan + warpSize;
        int *s_in_array = s_gtPivot_scan + warpSize;
        int *s_out_array = s_in_array + array_size;

        int2 pivot_count, pivot_scan;
        uint total_lt_values, total_gt_values;
```

```
    const int pivot = loadValues(g_in_array, s_in_array, array_size);

    pivot_count = countPivot(s_in_array, array_size, pivot);

    pivot_scan = prefixSum(s_ltPivot_scan, s_gtPivot_scan,
pivot_count);

    total_lt_values = s_ltPivot_scan[blockDim.x / warpSize - 1];
    total_gt_values = s_gtPivot_scan[blockDim.x / warpSize - 1];

    reorderValues(s_in_array, s_out_array, array_size, pivot,
pivot_scan.x, pivot_scan.y + total_lt_values);

    for (int i = tid; i < array_size; i += blockDim.x)
            g_out_array[i] = s_out_array[i];

    __syncthreads();

    if (tid == 0)
    {
            cudaStream_t lstream, rstream;
            cudaStreamCreateWithFlags(&lstream, cudaStreamNonBlocking
);
            cudaStreamCreateWithFlags(&rstream, cudaStreamNonBlocking
);

            //Left side (lower than the pivot)
            if (total_lt_values > 1)
            {
                    if (total_lt_values <= BITONIC_THRESHOLD)
                            bitonicSort<<<1, blockDim.x,
total_lt_values * sizeof(int), lstream>>>(g_out_array, g_out_array,
total_lt_values);
                    else if (depth == MAX_DEPTH - 1)
                            bigBitonicSortStream(g_out_array,
g_out_array, total_lt_values, lstream);
                    else
                            perSegmentQuicksort<<<1, blockDim.x, (
warpSize * 2 + total_lt_values * 2) * sizeof(int), lstream>>>(
g_out_array, g_out_array, total_lt_values, depth + 1);
            }

            //Right side (higher than the pivot)
            g_out_array += 1 + total_lt_values;
            total_gt_values--; // Subtracting the pivot
            if (total_gt_values > 1)
            {
```

```
                        if (total_gt_values <= BITONIC_THRESHOLD)
                                bitonicSort<<<1, blockDim.x,
    total_gt_values * sizeof(int), rstream>>>(g_out_array, g_out_array,
    total_gt_values);
                        else if (depth == MAX_DEPTH - 1)
                                bigBitonicSortStream(g_out_array,
    g_out_array, total_gt_values, rstream);
                        else
                                perSegmentQuicksort<<<1, blockDim.x, (
    warpSize * 2 + total_gt_values * 2) * sizeof(int), rstream>>>(
    g_out_array, g_out_array, total_gt_values, depth + 1);
                }

                cudaStreamDestroy(lstream);
                cudaStreamDestroy(rstream);
        }
}

__global__ void bigQuickSort(int *g_in_array, int *g_out_array, const int
     initial_pos, const int final_pos, const int segment_max_size,
    partition_t *g_partition, counters_t *g_counters)
{
        const int tid = threadIdx.x;
        const int array_size = final_pos - initial_pos;

        int start = initial_pos + segment_max_size * blockIdx.x;
        int end = start + segment_max_size;
        if (end > final_pos)
                end = final_pos;
        if (blockIdx.x == 0) //Not counting the pivot
                start++;

        const int my_segment_size = end - start;

        __shared__ int s_counters[2];
        extern __shared__ int s_mem[];
        int *s_ltPivot_scan = s_mem;
        int *s_gtPivot_scan = s_ltPivot_scan + warpSize;
        int *s_in_array = s_gtPivot_scan + warpSize;
        int *s_out_array = s_in_array + my_segment_size;

        int2 pivot_count, pivot_scan;
        uint total_lt_values, total_gt_values;
        int blocks;

        loadValues(g_in_array + start, s_in_array, my_segment_size);

        const int pivot = g_in_array[initial_pos];
```

```
    pivot_count = countPivot(s_in_array, my_segment_size, pivot);

    pivot_scan = prefixSum(s_ltPivot_scan, s_gtPivot_scan,
pivot_count);

    total_lt_values = s_ltPivot_scan[blockDim.x / warpSize - 1];
    total_gt_values = s_gtPivot_scan[blockDim.x / warpSize - 1];

    reorderValues(s_in_array, s_out_array, my_segment_size, pivot,
pivot_scan.x, pivot_scan.y + total_lt_values);

    if (tid == 0)
    {
            s_counters[0] = atomicAdd(&(g_counters->lt_next_position)
, total_lt_values);
            s_counters[1] = atomicAdd(&(g_counters->gt_next_position)
, total_gt_values);
    }

    __syncthreads();

    for (int i = tid; i < total_lt_values; i += blockDim.x)
    {
            g_out_array[i + s_counters[0] + initial_pos] =
s_out_array[i];
    }

    for (int i = tid; i < total_gt_values; i += blockDim.x)
    {
            g_out_array[array_size - s_counters[1] - total_gt_values
+ i + initial_pos] = s_out_array[i + total_lt_values];
    }

    if (tid == 0)
    {
            blocks = atomicAdd(&(g_counters->blocks), 1);

            if (blocks == gridDim.x - 1)
            {
                    int pivot_position = g_counters->lt_next_position
 + initial_pos;
                    g_out_array[pivot_position] = pivot;
                    g_in_array[pivot_position] = pivot;

                    g_counters->lt_next_position = 0;
                    g_counters->gt_next_position = 0;
                    g_counters->blocks = 0;
```

```
                        g_partition->start = initial_pos;
                        g_partition->pivot_position = pivot_position;
                        g_partition->end = final_pos;
                }
        }
}
```

Apéndice B

Kernels Mergesort

```
__global__ void selectionSort(int *g_array, const int array_size)
{
        extern __shared__ int s_mem[];
        int *s_array = s_mem;
        int *s_array_idx = s_array + blockDim.x * 4;

        const int global_idx = blockIdx.x * blockDim.x * 2;
        const int tid = threadIdx.x;
        const int bdim = blockDim.x;

        int chunk_size = bdim * 2;
        if (global_idx + chunk_size > array_size)
                chunk_size = array_size - global_idx;

        s_array[tid] = (global_idx + tid < array_size) ? g_array[
    global_idx + tid] : INT_MAX;
        s_array[tid + bdim] = (global_idx + tid + bdim < array_size) ?
    g_array[global_idx + tid + bdim] : INT_MAX;
        s_array[tid + bdim + bdim]  = INT_MAX;
        s_array[tid + bdim + bdim + bdim]  = INT_MAX;

        for (int j = 0; j < chunk_size - 1; j++)
        {
                s_array_idx[tid] = tid;
                s_array_idx[tid + bdim] = tid + bdim;

                __syncthreads();

                for (int i = bdim; i > 0; i >>= 1)
                {

                        if (tid < i)
```

```
                        {
                                int left_value = s_array[s_array_idx[tid]
    + j];
                                int right_value = s_array[s_array_idx[tid
    + i] + j];

                                if (left_value > right_value)
                                        s_array_idx[tid] = s_array_idx[
   tid + i];
                        }

                        __syncthreads();
                }

                if (tid == 0)
                {
                        int swap = s_array[j];
                        s_array[j] = s_array[s_array_idx[0] + j];
                        s_array[s_array_idx[0] + j] = swap;
                }
        }

        __syncthreads();

        for (int i = tid; i < chunk_size; i += bdim)
                g_array[global_idx + i] = s_array[i];
}

__global__ void mergePath(int *array_A, int *array_B, int *array_C, const
    int arrayA_size, const int arrayB_size)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        int a_idx, b_idx;

        if (idx >= arrayA_size + arrayB_size)
                return;

        int lower_bound = max(0, idx - arrayB_size);
        int upper_bound = min(idx, arrayA_size);

        while (lower_bound < upper_bound)
        {
                int mid_point = (lower_bound + upper_bound) >> 1;

                if (array_A[mid_point] > array_B[idx - mid_point - 1])
                        upper_bound = mid_point;
                else
                        lower_bound = mid_point + 1;
```

```
        }

        a_idx = lower_bound;
        b_idx = idx - lower_bound;

        if (a_idx == arrayA_size)
                array_C[idx] = array_B[b_idx];
        else if (b_idx == arrayB_size)
                array_C[idx] = array_A[a_idx];
        else
        {
                if (array_A[a_idx] > array_B[b_idx])
                        array_C[idx] = array_B[b_idx];
                else
                        array_C[idx] = array_A[a_idx];
        }
}

int *mergeArrays(int *d_in_array, int *d_out_array, const int
    initial_size, const int array_size)
{
        cudaStream_t stream;
        int *swap;

        for (int i = initial_size; i < array_size; i <<= 1)
        {
                for (int j = 0; j < array_size; j += i << 1)
                {
                        int chunk_size_left = array_size - j;
                        if (chunk_size_left > i)
                        {
                                checkErrors( cudaStreamCreateWithFlags(&
    stream, cudaStreamNonBlocking) );

                                dim3 blockSize(BLOCK_SIZE, 1, 1);
                                if (chunk_size_left >= i * 2)
                                {
                                        dim3 gridSize((i * 2 + BLOCK_SIZE
     - 1)/BLOCK_SIZE, 1, 1);
                                        mergePath<<<gridSize, blockSize,
    0, stream>>>(d_in_array + j, d_in_array + j + i, d_out_array + j, i, i
    );
                                        checkErrors( cudaGetLastError() )
    ;
                                }
                                else
                                {
```

```
                                        dim3 gridSize((chunk_size_left +
    BLOCK_SIZE - 1)/BLOCK_SIZE, 1, 1);
                                        mergePath<<<gridSize, blockSize,
    0, stream>>>(d_in_array + j, d_in_array + j + i, d_out_array + j, i,
    chunk_size_left - i);
                                        checkErrors( cudaGetLastError() )
    ;
                                }
                                checkErrors( cudaStreamDestroy(stream) );
                        }
                        else
                                checkErrors( cudaMemcpyAsync(d_out_array
    + j, d_in_array + j, chunk_size_left * sizeof(int),
    cudaMemcpyDeviceToDevice) );
                }

                swap = d_in_array;
                d_in_array = d_out_array;
                d_out_array = swap;

                checkErrors( cudaDeviceSynchronize() );
        }

        return d_in_array; //It's reversed because of the last swap
}
```

Apéndice C

Kernels K-means

```
__global__ void calcClosestDistances(int2 *g_observationsPos, int
    observations, int2 *g_kPos, int k_size, int *g_closestK)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        extern __shared__ int2 s_kPos[];

        if (threadIdx.x < k_size)
                s_kPos[threadIdx.x] = g_kPos[threadIdx.x];

        __syncthreads();

        if (idx >= observations)
                return;

        int2 pos = g_observationsPos[idx];
        int2 kPos = s_kPos[0];

        int tempX = pos.x - kPos.x;
        int tempY = pos.y - kPos.y;

        int min_dist = tempX * tempX + tempY * tempY;
        int min_idx = 0;

        int dist;

        for (int i = 1; i < k_size; i++)
        {
                kPos = s_kPos[i];

                tempX = pos.x - kPos.x;
                tempY = pos.y - kPos.y;
```

```
                dist = tempX * tempX + tempY * tempY;
                if (dist < min_dist)
                {
                        min_dist = dist;
                        min_idx = i;
                }
        }

        g_closestK[idx] = min_idx;
}

__global__ void firstStageReduce(int2 *g_observationsPos, int *g_closestK
    , int2 *g_sum, int *g_obs, const int observations, const int
    observations_per_block)
{
        extern __shared__ int s_mem[];
        int2 *s_sum = (int2 *)s_mem;
        int *s_obs = (int *)(&s_sum[blockDim.x]);

        int tid = threadIdx.x;
        int centroid = blockIdx.y;

        int2 sum = {0, 0};
        int amountObservations = 0;

        int starting_position = observations_per_block * blockIdx.x;
        int last_position = starting_position + observations_per_block;
        if (last_position > observations)
                last_position = observations;

        for (int i = starting_position + tid; i < last_position; i +=
    blockDim.x)
        {
                if (g_closestK[i] == centroid)
                {
                        int2 actual_observationPos = g_observationsPos[i
    ];
                        sum.x += actual_observationPos.x;
                        sum.y += actual_observationPos.y;
                        amountObservations++;
                }
        }

        s_sum[tid] = sum;
        s_obs[tid] = amountObservations;

        __syncthreads();
```

```
        for (int i = blockDim.x >> 1; i > 0; i >>= 1)
        {
                if (tid < i)
                {
                        // s_sum[tid] = s_sum[tid] + s_sum[tid + i];
                        int2 tempA = s_sum[tid];
                        int2 tempB = s_sum[tid + i];
                        int2 tempSum;

                        tempSum.x = tempA.x + tempB.x;
                        tempSum.y = tempA.y + tempB.y;

                        s_sum[tid] = tempSum;
                        s_obs[tid] = s_obs[tid] + s_obs[tid + i];
                }

                __syncthreads();
        }

        if (tid == 0)
        {
                int pos = blockIdx.y * gridDim.x + blockIdx.x;
                g_sum[pos] = s_sum[0];
                g_obs[pos] = s_obs[0];
        }
}

__global__ void reduceAndUpdateCentroids(int2 *g_sum, int *g_obs, int2 *
    g_kPos, const int array_size)
{
        extern __shared__ int s_mem[];
        int2 *s_sum = (int2 *)s_mem;
        int *s_obs = (int *)(&s_sum[blockDim.x]);

        int tid = threadIdx.x;

        int2 sum = {0, 0};
        int amountObservations = 0;

        int starting_pos = array_size * blockIdx.x;
        for (int i = tid; i < array_size; i += blockDim.x)
        {
                int2 actual_sum = g_sum[i + starting_pos];
                sum.x += actual_sum.x;
                sum.y += actual_sum.y;
                amountObservations += g_obs[i + starting_pos];
        }
```

```
        s_sum[tid] = sum;
        s_obs[tid] = amountObservations;

        __syncthreads();

        for (int i = blockDim.x >> 1; i > 0; i >>= 1)
        {
                if (tid < i)
                {
                        // s_sum[tid] = s_sum[tid] + s_sum[tid + i];
                        int2 tempA = s_sum[tid];
                        int2 tempB = s_sum[tid + i];
                        int2 tempSum;

                        tempSum.x = tempA.x + tempB.x;
                        tempSum.y = tempA.y + tempB.y;

                        s_sum[tid] = tempSum;
                        s_obs[tid] = s_obs[tid] + s_obs[tid + i];
                }

                __syncthreads();
        }

        if (tid == 0)
        {
                amountObservations = s_obs[0];

                if (amountObservations != 0)
                {
                        int2 temp = s_sum[0];

                        temp.x /= amountObservations;
                        temp.y /= amountObservations;

                        g_kPos[blockIdx.x] =  temp;
                }
        }
}

__global__ void kMeans(int2 *g_observationsPos,
                       int2 *g_kPos,
                       int *g_closestK,
                       int2 *g_sum,
                       int *g_obs,
                       const int observations,
                       const int k_size,
                       const int grid_dist_size,
```

```
                        const int grid_reduce_size,
                        const int block_size,
                        const int reduce_observations_per_block)
{
        extern __shared__ int2 s_pos_mem[];
        int2 *s_kPos = s_pos_mem;
        int2 *s_new_kPos = s_kPos + k_size;

        bool convergence;
        int loop = 0;
        int tid = threadIdx.x;

        s_kPos[tid] = g_kPos[tid];

        __syncthreads();

        // Grid/Block sizes
        const dim3 gridSizeDistances(grid_dist_size, 1, 1);
        const dim3 gridSizeReduce(grid_reduce_size, k_size, 1);
        const dim3 blockSize(block_size, 1, 1);

        do
        {
                if (tid == 0)
                {
                        calcClosestDistances<<<gridSizeDistances,
    blockSize, k_size * sizeof(int2)>>>(g_observationsPos, observations,
    g_kPos, k_size, g_closestK);
                        firstStageReduce<<<gridSizeReduce, blockSize,
    blockSize.x  * (sizeof(int2) + sizeof(int))>>>(g_observationsPos,
    g_closestK, g_sum, g_obs, observations, reduce_observations_per_block)
    ;
                        reduceAndUpdateCentroids<<<k_size, blockSize,
    blockSize.x  * (sizeof(int2) + sizeof(int))>>>(g_sum, g_obs, g_kPos,
    gridSizeReduce.x);
                        cudaDeviceSynchronize();
                }

                __syncthreads();

                s_new_kPos[tid] = g_kPos[tid];

                __syncthreads();

                convergence = checkConvergenceInt2(s_kPos, s_new_kPos,
    k_size);

                int2 *swap = s_kPos;
```

```
                s_kPos = s_new_kPos;
                s_new_kPos = swap;

                loop++;
        }while (!convergence && loop < MAX_LOOPS);
}
```

Apéndice D

Kernels N-Body + Leapfrog

D.1. N-Body

```
__global__ void nBody(float4 *g_positions, PointsArray g_accelerations,
    const int num_bodies, const int tile_size)
{
        extern __shared__ float4 s_positions[];

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

        float3 acceleration = {0.0f, 0.0f, 0.0f};
        float3 r;

        float4 temp_pos;
        float4 my_position;

        if (idx < num_bodies)
                my_position = g_positions[idx];

        for (int tile_pos = 0; tile_pos < num_bodies; tile_pos +=
    tile_size)
        {
                for (int i = tid; i < tile_size && (i + tile_pos) <
    num_bodies; i += blockDim.x)
                        s_positions[i] = g_positions[i + tile_pos];

                __syncthreads();

                if (idx < num_bodies)
                {
```

```
                        for (int i = 0; i < tile_size && (i + tile_pos) <
    num_bodies; i++)
                        {
                                temp_pos = s_positions[i];
                                r.x = my_position.x - temp_pos.x;
                                r.y = my_position.y - temp_pos.y;
                                r.z = my_position.z - temp_pos.z;

                                float temp = r.x * r.x + r.y * r.y + r.z
   * r.z + EPS2;
                                temp = rsqrtf(temp * temp * temp);
                                temp = temp * temp_pos.w;

                                acceleration.x += r.x * temp;
                                acceleration.y += r.y * temp;
                                acceleration.z += r.z * temp;
                        }
                }

                __syncthreads();
        }

        if (idx < num_bodies)
        {
                g_accelerations.x[idx] = acceleration.x;
                g_accelerations.y[idx] = acceleration.y;
                g_accelerations.z[idx] = acceleration.z;
        }
}
```

D.2. Leapfrog

```
__global__ void leapFrogHalfVelocities(PointsArray g_velocities,
    PointsArray g_accelerations, const int dt, const int num_bodies)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        if (idx < num_bodies)
        {
                float3 velocity = {g_velocities.x[idx], g_velocities.y[
   idx], g_velocities.z[idx]};
                float3 acceleration = {g_accelerations.x[idx],
   g_accelerations.y[idx], g_accelerations.z[idx]};

                velocity.x += 0.5f * acceleration.x * dt;
                velocity.y += 0.5f * acceleration.y * dt;
                velocity.z += 0.5f * acceleration.z * dt;
```

```
                g_velocities.x[idx] = velocity.x;
                g_velocities.y[idx] = velocity.y;
                g_velocities.z[idx] = velocity.z;
        }
}

__global__ void leapFrogPositions(float4 *g_positions, PointsArray
    g_velocities, const int dt, const int num_bodies)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        if (idx < num_bodies)
        {
                float4 position = g_positions[idx];
                float3 velocity = {g_velocities.x[idx], g_velocities.y[
    idx], g_velocities.z[idx]};

                position.x -= velocity.x * dt;
                position.y -= velocity.y * dt;
                position.z -= velocity.z * dt;

                g_positions[idx] = position;
        }
}
```

Apéndice E

Kernels Floyd-Warshall

```
__global__ void floydWarshall(int *g_dist_table, int *g_seq_table, const
    int vertices, const int rows_per_block, const int iteration)
{
        int tid = threadIdx.x;

        int first_row = blockIdx.x * rows_per_block;
        int last_row = first_row + rows_per_block;
        if (last_row > vertices)
                last_row = vertices;

        for (int col = tid; col < vertices; col += blockDim.x)
        {
                int horizontal_pos, horizontal_value;

                horizontal_pos = iteration * vertices + col;
                horizontal_value = g_dist_table[horizontal_pos];

                for (int row = first_row; row < last_row; row++)
                {
                        int my_pos, vertical_pos;

                        my_pos = row * vertices + col;
                        vertical_pos = row * vertices + iteration;

                        if (g_dist_table[my_pos] > g_dist_table[
    vertical_pos] + horizontal_value)
                        {
                                g_dist_table[my_pos] = g_dist_table[
    vertical_pos] + horizontal_value;
                                g_seq_table[my_pos] = g_seq_table[
    vertical_pos];
                        }
```

```
                    }
            }
}
```

Apéndice F

Kernels Monte Carlo

```
__global__ void monteCarlo(unsigned int seed, curandState_t *states,
    unsigned long *g_block_results, unsigned long loops, unsigned long
    total_numbers)
{
        extern __shared__ unsigned long s_sum[];
        long idx = blockIdx.x * blockDim.x + threadIdx.x;
        long total_tid = gridDim.x * blockDim.x;
        int tid = threadIdx.x;
        curandState_t local_state = states[idx];
        unsigned long in_circle = 0;
        float x, y;

        if (total_tid * (loops-1) + tid >= total_numbers)
                loops--;

        curand_init(seed, idx, 0, &local_state);
        for (unsigned long i = 0; i < loops; i++)
        {
                x = curand_uniform(&local_state);
                y = curand_uniform(&local_state);

                if (x*x + y*y < 1)
                        in_circle++;
        }

        s_sum[tid] = in_circle;

        __syncthreads();

        for (int i = blockDim.x >> 1; i > 0; i >>= 1)
        {
                if (tid < i)
```

```
                        s_sum[tid] = s_sum[tid] + s_sum[tid + i];

                __syncthreads();
        }

        if (tid == 0)
                g_block_results[blockIdx.x] = s_sum[0];

}
```

Bibliografía

[1] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4 (7), 1961. doi: 10.1145/366622.366642. URL https://dx.doi.org/10.1145%2F366622.366644.

[2] K. Batcher. Sorting networks and their applications. *In Proceedings of the AFIPS Spring Joint Computing Conference*, 32:307–314, 1968. doi: 10.1145/1468075.1468121. URL http://dx.doi.org/10.1145/1468075.1468121.

[3] W. Daniel y Steele Jr. Hillis. Guy l. data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. URL http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=7903.

[4] Donald Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2 edition, 1998. ISBN 0-201-89685-0.

[5] Saher Odeh y Yitzhak Birk Oded Green. Merge path - a visually intuitive approach to parallel merging. Technical report, Georgia Institute of Technology y Technion - Israel Institute of Technology, 2012.

[6] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

[7] Cuda pro tip: Increase performance with vectorized memory access, . URL https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/.

[8] Josh Barnes y Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, Dec 1986. doi: 10.1038/324446a0. URL http://dx.doi.org/10.1038/324446a0.

[9] Hubert Nguyen. *Gpu Gems 3*, chapter 31. Fast N-Body Simulation with CUDA. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428. URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html.

[10] Peter Young. Physics 115/242. the leapfrog method and other "symplectic" algorithms for i ntegrating newton's laws of motion, 2014. URL http://young.physics.ucsc.edu/115/leapfrog.pdf.

[11] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962. ISSN 0001-0782. doi: 10.1145/367766.368168. URL http://doi.acm.org/10.1145/367766.368168.

[12] Bernard Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[13] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. ISSN 0004-5411. doi: 10.1145/321105.321107. URL http://doi.acm.org/10.1145/321105.321107.

[14] John Mellor-Crummy. Bitonic sort. URL https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158.

[15] Sorting networks. URL https://mitpress.mit.edu/sites/default/files/Chapter%2027.pdf.

[16] Exploring n = 2 with a leapfrog algorithm. URL http://www.artcompsci.org/vol_1/v1_web/node33.html.

[17] Monte carlo estimation for pi. URL http://polymer.bu.edu/java/java/montepi/MontePi.html.

[18] Floyd-warshall algorithm. URL http://www.programming-algorithms.net/article/45708/Floyd-Warshall-algorithm.

[19] Parallel prefix sum (scan) with cuda. URL http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf.

[20] Optimizing parallel reduction in cuda. URL https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

[21] Cuda c programming guide, . URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[22] Cuda c best practices guide. URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[23] Nvidia visual profiler, . URL https://developer.nvidia.com/nvidia-visual-profiler.

[24] How to query device properties and handle errors in cuda c/c++. URL https://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/.