UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

Informe de Memoria de Título

Implementación paralela de algoritmo de agrupación jerárquica en GPU

Ronald Alfonso Valenzuela Fica

Concepción, Julio de 2010

Resumen

La clasificación a través del análisis de datos es frecuentemente utilizada para la inferencia de modelos. Sin embargo, los algoritmos de agrupación jerárquica acumulativa son poco utilizados debido a su gran costo computacional. Estos tienen la ventaja de agregar menor distorsión a los resultados, pues no requieren información que restrinja la cantidad o la forma en que se establecen los grupos.

Un algoritmo de agrupación jerárquica en particular es hoy utilizado en la inferencia de modelos de tractos neuronales. Grandes cantidades de datos son procesados y el algoritmo se aplica en numerosas ocasiones, esto lo transforma en un factor limitante para la investigación de los modelos.

En este trabajo se desarrolla una implementación paralela del algoritmo utilizado en la investigación señalada. Para esto se utiliza CUDA, una plataforma diseñada para utilizar un procesador gráfico en el cómputo algoritmos de propósito general.

Los resultados que se han conseguido muestran aceleraciones de hasta 50 veces por sobre la implementación original. Se programó en C una versión secuencial del algoritmo, ésta que consigue acelerar más de 4 veces al algoritmo inicial. La implementación paralela, desarrollada en CUDA, consigue un tiempo de ejecución 13 veces mayor que la versión secuencial.

A mi madre y en memoria de mi padre...

Les agradezco todo lo que soy

Agradecimientos

Gracias a Pamela Guevara por su paciencia y disponibilidad para explicar los detalles de su investigación. A mi Profesor Miguel Figueroa por su apoyo y confianza en mi trabajo. A mis amigos y compañeros que dedicaron algunos momentos a discutir mi trabajo. A todos con los que compartí este largo trayecto en búsqueda del futuro profesional. A mi familia: María Eugenia, Rodrigo, Alexandra y a Constanza; gracias por su confianza y soporte.

Índice

Índice de figuras

Índice de cuadros

Lista de acrónimos

PTX Parallel Thread Execution

SIMT Single Instruction Multiple Threads

GPU Graphical Processing Unit

TPC Texture Processing Cluster

DWMRI Diffusion Weighted Magnetic Resonance Imaging

DTI Diffusion Tensor Imaging

HARDI High Angular Resolution Diffusion Imaging

FPGA Field Programmable Gate Array

1. Introducción

1.1. Introducción general

Los algoritmos de agrupación jerárquica [1] tienen un gran potencial en la tarea de clasificación de información. Su desarrollo riguroso comienza a mediados del siglo XX, en el área de la taxonomía, donde se utilizaron para establecer relaciones entre familias de animales en base a características comunes [1]. La tarea de éstos consiste en obtener, a partir de un conjunto arbitrario de datos, un dendrograma que represente los niveles en que estos datos pueden unirse entre sí para conformar grupos de similitud. Al ser un ordenamiento de carácter no asistido, no requieren información adicional sobre como realizar la agrupación. Sin embargo, éstos suelen ser poco utilizados debido a sus altos requerimientos de cómputo y almacenamiento, p.e. $O(N^3)$ y $O(N^2)$ respectivamente para los métodos de matriz almacenada y tomando en cuenta que son aplicados sobre una gran cantidad de datos[2].

Este tipo de algoritmos tienen su mayor aplicación en el campo de la investigación. En particular, este trabajo se centra, como se explica en el Capítulo 2, en un estudio biomédico sobre las estructuras que componen la materia blanca cerebral a partir de imágenes obtenidas por "Diffusion Weighted Magnetic Resonance Imaging (DWMRI)". La utilización del algoritmo escogido representa un desafío en términos de costo computacional debido a la gran cantidad de datos a procesar y a las características del mismo. Por otro lado, estas mismas características evidencian un potencial de cálculo altamente paralelo.

El reciente desarrollo en el área de procesadores gráficos ha permitido la utilización de estos recursos de procesamiento masivamente paralelo en el ámbito de la investigación científica y análisis de datos. Esto sumado a la ventaja económica que presentan estos dispositivos y a su fuerte introducción en el mercado de la computación de propósito general los hacen candidatos ideales como plataforma para la aceleración de algoritmos como el de agrupación jerárquica.

Los grandes fabricantes de procesadores gráficos han desarrollado tecnologías que permiten sacar provecho de estos productos a través de lenguajes computacionales de alto nivel, lo que facilita la migración

[1] Entiéndase para este trabajo como traducción de hierarchical clustering

desde la computación secuencial tradicional hacia el paradigma de la programación paralela.

En este trabajo se presenta el desarrollo de una aplicación que permite acelerar el proceso de agrupación jerárquica, utilizando para esto un dispositivo de video compatible con la arquitectura CUDA de NVIDIA, que provee una interfaz de programación que extiende ANSI C, permitiendo desarrollar rutinas para esta plataforma.

Como resultado, se han conseguido dos implementaciones del algoritmo en estudio. La primera, una versión secuencial escrita en ANSI C, que ha superado las expectativas, logrando una aceleración de 4 veces sobre la original. La segunda, satisface plenamente los objetivos al obtener una aceleración cercana a 50 veces sobre la implementación original y de 13 sobre la versión en C. Le implementación paralela muestra mayor variación en los resultados, debido a características de la implementación que son analizadas en el Capítulo 5, sin embargo, cumple plenamente con los requerimientos.

1.2. Objetivos

1.2.1. Objetivo general

Conseguir una aceleración en el tiempo de cómputo de un algoritmo de agrupación jerárquica a través de una implementación de éste sobre un procesador gráfico.

1.2.2. Objetivos específicos

Obtener una implementación secuencial eficiente y determinar a partir de éste, requerimientos o condiciones particulares del algoritmo, en el contexto del caso de prueba seleccionado.
Definir una arquitectura apropiada para desarrollar la implementación paralela sobre un Graphical Processing Unit (GPU).
Desarrollar, validar y evaluar la implementación paralela desarrollada sobre la arquitectura seleccionada.

1.3. Alcances y limitaciones

Se implementará un algoritmo de agrupación jerárquica que considera promedios entre distancias medias. Este algoritmo queda definido por el marco de aplicación. La implementación se realizará en lenguaje CUDA sobre una plataforma NVIDIA.

1.4. Temario

El Capítulo 2, presenta el marco de aplicación que se le ha dado a este trabajo, éste introduce y explica la investigación citada en [3].
En el Capítulo 3 se desarrolla la teoría detrás de los algoritmos de agrupación jerárquica y luego se propone un algoritmo secuencial. Se describe el propósito del algoritmo y luego se presentan algunos métodos para caracterizar las distancias de los grupos formados. Finalmente, se describe el algoritmo propuesto a través de pseudo lenguaje y un diagrama de flujo.
Durante el Capítulo 4 se realiza una evaluación del algoritmo secuencial presentado. Se introduce el set de prueba. Se describe la metodología utilizada para validar el algoritmo. Se analizan los requerimientos

de precisión numérica. Finalmente, se realiza un perfil de tiempo para la implementación lograda.

El Capítulo 5 introduce la computación paralela sobre procesadores gráficos y se presentan algunas arquitecturas y plataformas que la habilitan. Se determina CUDA como la plataforma de trabajo y se realiza una descripción de la arquitectura, del modelo de ejecución y de algunos detalles relevantes. Finalmente se presenta un apartado sobre consideraciones numéricas asociadas al sistema de precisión, arquitectura y algoritmo en cuestión.

En el Capítulo 6 se desarrolla el algoritmo paralelo diseñado para GPU. Se describe el trabajo orientado a kernels. Se propone un diagrama de flujo compatible con este modelo. Se desarrolla cada uno de los procedimientos paralelizables en forma de kernels.

El Capítulo 7 consiste en la validación y contraste de los resultados obtenidos del algoritmo sobre GPU. Se muestra una validación de los resultados en una forma similar al Capítulo 4. Luego se presenta un perfil de ejecución del algoritmo, así como otros datos estadísticos relevantes. Para finalizar, se muestra la aceleración lograda por la implementación sobre el algoritmo original. Para terminar, el Capítulo 8 presenta las conclusiones de este trabajo.

2. Marco de aplicación

La agrupación jerárquica define algoritmos que proponen resolver una gran diversidad de problemas de clasificación, ver [4]. Este trabajo en particular, centra su atención en la investigación titulada *"Inference of a HARDI fiber bundle atlas using a two level clustering strategy"* [3]. Tanto el estudio y la validación de la solución que este trabajo propone, se ha realizado en el espacio de muestras de esta aplicación. La siguiente sección describe aquella investigación.

2.1. Una plataforma para la inferencia de modelos jerárquicos de materia blanca

La utilización de tecnología "Diffusion Weighted Magnetic Resonance Imaging (DWMRI)" permite la obtención de imágenes biomédicas de alta resolución. Estas reflejan el desplazamiento de moléculas de agua en los tejidos para distintas direcciones del espacio. Modelos de reconstrucción como Diffusion Tensor Imaging (DTI) permiten obtener vectores tridimensionales que indican la orientación preferencial de las moléculas de agua. Estos modelos permiten obtener en forma no invasiva información sobre la micro estructura de los tejidos vivos. En el caso del cerebro, éstos permiten también estimar la orientación de los fascículos de fibras.
En particular, las imágenes de DWMRI de alta resolución angular (High Angular Resolution Diffusion Imaging (HARDI)), junto con modelos de reconstrucción de mayor orden, presentan una mejor descripción de la difusión en cada voxel. La técnica, llamada tractografía, permite reconstruir los tractos o fibras neuronales en función de la dirección preferencial de difusión en cada voxel. Un ejemplo de esta reconstrucción se observa en la Figura 1. Estas fibras son los axones de las neuronas, que interconectan partes del sistema nervioso y van desde un sector de la corteza cerebral a otro. Los tractos, en su conjunto, constituyen la materia blanca del cerebro.

En la investigación citada [3], se propone establecer un modelo que agrupe conjuntos de fibras neuronales de acuerdo a patrones encontrados primero en el estudio de individuos aislados y luego reclasificados según su correlación con la ocurrencia colectiva, es decir, entre sujetos.
Este estudio comienza a partir de un conjunto de fibras previamente obtenidos. El conjunto es prime-

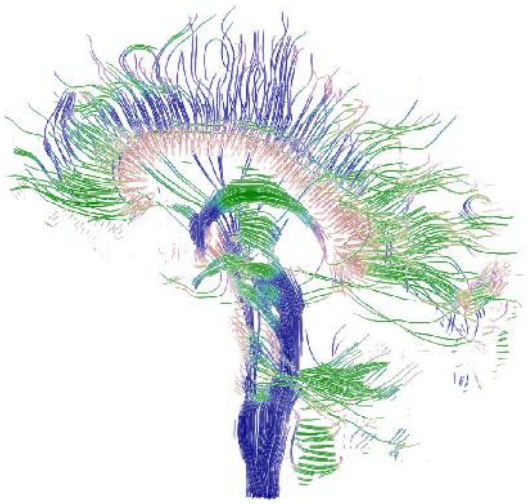

Figura 1: Reconstrucción de tractos neuronales. Disponible a través de Wikipedia, cortesía de Gordon Kindlmann del "Scientific Computing and Imaging Institute", University of Utah.

ramente particionado en cinco grupos que serán analizados por separado. La división corresponde a: hemisferio izquierdo, derecho, zona interhemisférica, cerebelo y tractos descartados. Los subconjuntos son luego separados de acuerdo a la longitud de los tractos, los cuales también se estudiarán por separado. Con estas divisiones establecidas, comienza el proceso de agrupación. Lo primero es la obtención de parcelas que dividen espacialmente el cerebro. Este agrupamiento utiliza el algoritmo k-means [5], que propone maximizar la distancia de los centroides de cada cluster a partir de una métrica establecida. Una vez que estos grupos son establecidos, se les asigna una medida de afinidad entre parcelas, que corresponde al número de fibras que atraviesan desde una parcela A a una B. Estos valores pueden ser ordenados en forma matricial o de grafo, con la información para cada par. Luego, estos datos, son utilizados como entrada de un algoritmo de agrupación jerárquica acumulativa. Esto se muestra en la Figura 2, en la cual observamos de izquierda a derecha primero fibras y parcelas que las incluyen, luego una matriz asociada a cada parcela con una afinidad establecida entre los pares y finalmente en colores una agrupación de estas parcelas según la conectividad de fibras entre ellas.

La aplicación del algoritmo de agrupación jerárquica busca obtener grupos de parcelas que son

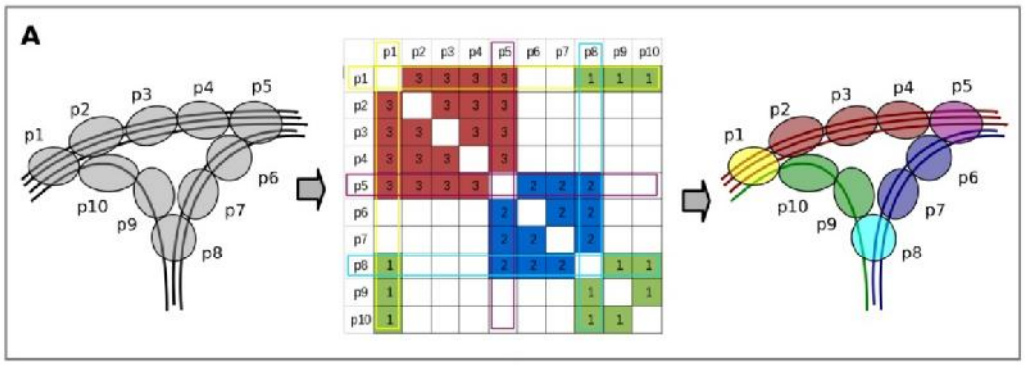

Figura 2: Proceso de agrupación acumulativa. Cortesía de Pamela Guevara, CEA, Neurospin. Fragmento de imagen a ser enviada a Neuroimage.

atravesados por un conjunto particular de fibras. Las agrupaciones de parcelas obtenidas sirven luego para extraer los grupos de fibras buscados, logrando así una segmentación de cada set. Por razones que se explican en el capítulo 3, este procedimiento requiere de una gran capacidad de cómputo. La reducción del gran tiempo de cálculo que requiere esta etapa es uno de los objetivos de esta memoria de título. Es posible observar de la Figura 3 la relación que existe entre la parcelación original, árbol de clusters y las fibras obtenidas.

El estudio continúa. Para cada grupo de fibras identificado se realiza una segmentación. Se identifican grupos delgados y homogéneos de fibras basados en los extremos. Cuando éstos se han obtenido para toda la materia blanca, se calcula un tracto para cada grupo delgado, basado en su centroide. Estos grupos son nuevamente clasificados por un algoritmo de agrupación jerárquica.

Para la obtención de un modelo final, se realiza una validación entre sujetos, que consiste en la reaplicación de un algoritmo de agrupación jerárquica considerando las fibras calculadas con los centroides de los grupos de tractos para cada sujeto.

El análisis completo para un conjunto de datos, al que se le aplican todas las etapas mencionadas, toma aproximadamente ocho horas. Del total, al rededor de un 30 % corresponde al algoritmo de agrupación jerárquica acumulativa. Es decir, dos horas y veinticuatro minutos de cada ejecución son atribuidas a dicho procedimiento. Esta es la motivación del desarrollo de esta memoria.

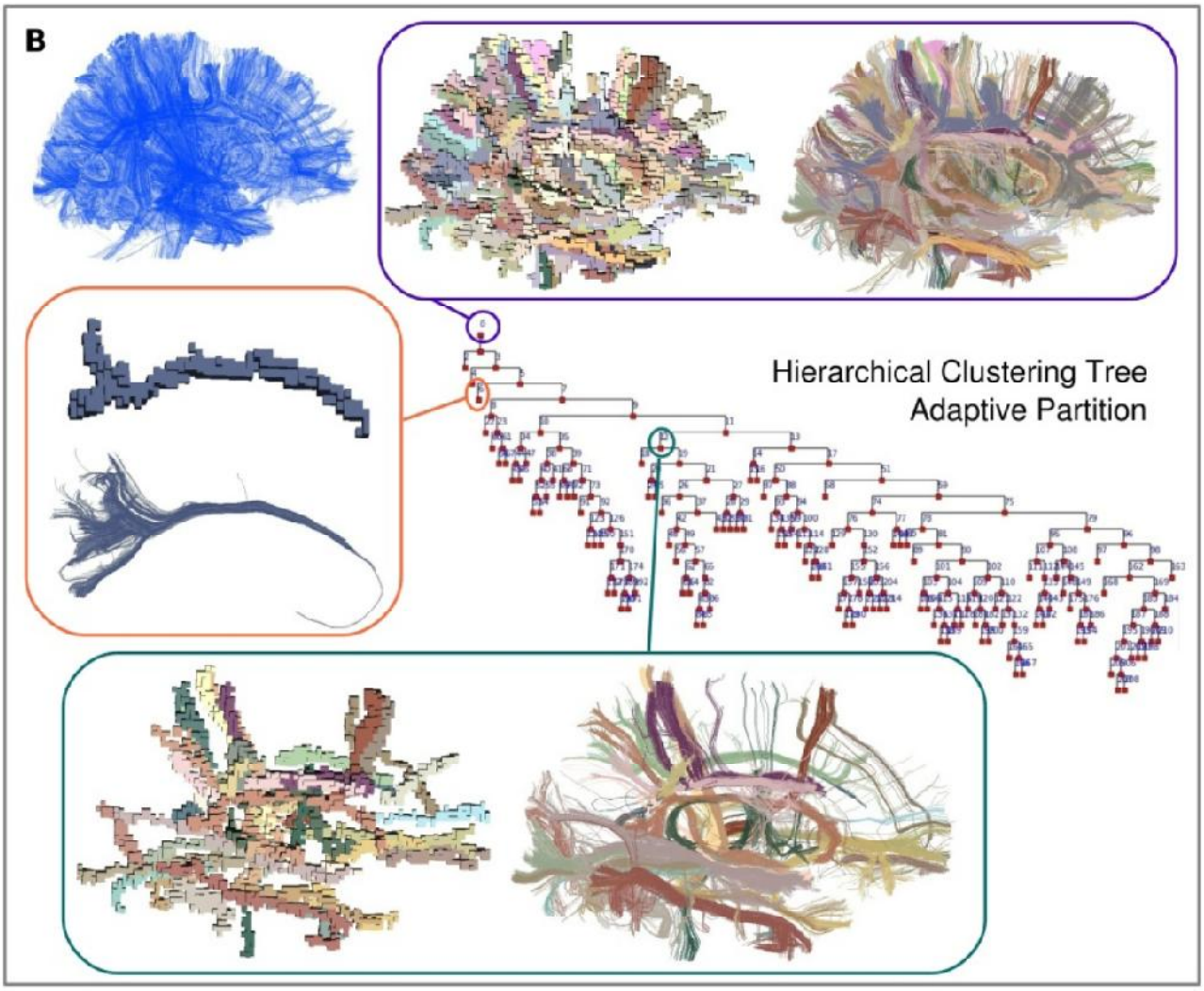

Figura 3: Resultados de la agrupación y visualización de los clusters. Cortesía de Pamela Guevara, CEA, Neurospin. Fragmento de imagen a ser enviada a Neuroimage.

3. Desarrollo secuencial del algoritmo

3.1. Teoría

A continuación se presentan los fundamentos teóricos necesarios para la implementación y evaluación del algoritmo de agrupación jerárquica.

3.1.1. Agrupación jerárquica acumulativa

Estos algoritmos de clasificación se caracterizan por comenzar con un conjunto de N elementos, para el cual se conoce la distancia entre cada par, según una medida arbitraria. A partir de esta matriz de datos se procede a agrupar progresivamente los elementos que se encuentran más cercanos entre sí. El algoritmo termina cuando se forma el último grupo, momento para el cual se han formado N-1 nuevos clusters.
Como resultado se ha obtenido un vector de padres que registra la pertenencia de cada elemento o grupo a otro y un vector de alturas que indica las distancias a las que los grupos se han formado. Esta información es suficiente para formar un dendrograma (ver Figura 4), estructura que permite analizar la jerarquía obtenida de los resultados.

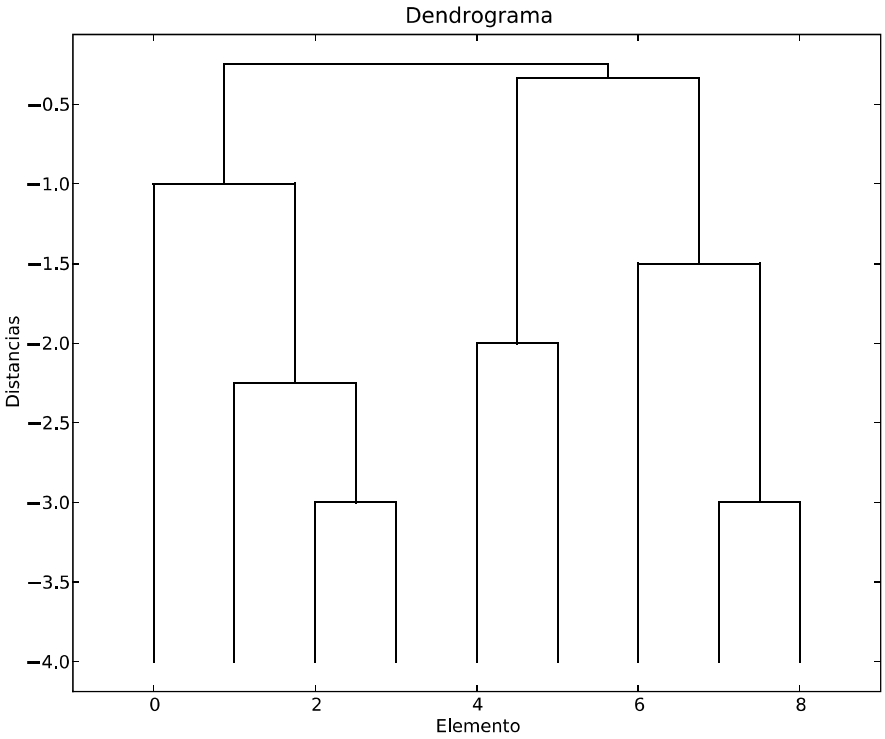

Figura 4: Dendrograma de ejemplo.

3.1.2. Cálculo de distancias sobre el nuevo cluster

El procedimiento explicado en la sección anterior, puede resumirse como sigue:

```
Entrada: Datos (N, M)
Salida: Dendrograma
        1. for i = 1 a N - 1
                2. Encontrar el par de elementos mas cercano
                3. Combinar este par en un nuevo Cluster
                4. Calcular las distancias involucradas con
                   este nuevo grupo.
        5. return dendrogram
```

Sin embargo, el paso 5 debe examinarse con más detalle. Para calcular las distancias de los elementos a los nuevos grupos, las alternativas más comunes son:

Single-linkage La distancia entre el nuevo cluster y otro elemento corresponde a la mínima entre las distancias de los elementos que pertenecen a este y otro.

$$d(x,y) = min\{d(x,y) : x \in A, y \in B\} \tag{1}$$

Complete-linkage La distancia entre el nuevo cluster y otro elemento corresponde a la máxima entre las distancias de los elementos que pertenecen a este y otro.

$$d(x,y) = max\{d(x,y) : x \in A, y \in B\} \tag{2}$$

Average-linkage La distancia entre el nuevo cluster y otro elemento corresponde a la media de todas las distancias de un grupo con las de otro.

$$d(x,y) = \frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} d(x,y) \tag{3}$$

En los dos primeros métodos, el algoritmo escoge una distancia arbitrariamente, eligiendo perder el aporte del resto de los elementos. La ventaja del método *Average linkage* sobre los otros, es su mayor capacidad para conservar las características de los elementos que son agrupados. Esto permite que los grupos se distribuyan de mejor manera, maximizando las propiedades que comparten. Por otro lado, su principal desventaja es que resulta particularmente costoso en términos computacionales y su cálculo resulta poco práctico cuando el número de elementos se incrementa. Es por esto que se escoge *Average linkage* como foco de este trabajo, que busca la aceleración del cálculo mediante el procesamiento paralelo.

3.2. Implementación secuencial

Para evaluar el desempeño de una implementación secuencial del algoritmo, este fue desarrollado en lenguaje ANSI C, basado en una implementación original escrita en Python.
El algoritmo de la sección anterior puede detallarse de la siguiente manera:

```
Entrada: Datos (N, M)
Salida: Dendrograma
 1. for i = 1 a N - nC
      2. Emparejamiento: Encontrar el par de elementos de mayor afinidad.
      3. Supresión: Eliminar el par encontrado.
      4. Ponderación: Aplicar un peso a las distancias de los elementos
         originales según su ocurrencia.
      5. Para cada lado, realizar la suma acumulativa
         de las distancias ponderadas y eliminar entradas duplicadas.
         5.1. Recolección: Se obtienen los índices del nuevo cluster.
         5.2. Integración: Se obtienen los pares de esas posiciones.
         5.3. Fusión: Se fusionan las distancias involucradas.
 6. return dendrograma
```

En el esquema anterior se detalla el procedimiento. Es aplicado sobre una matriz que representa las M relaciones conocidas entre N elementos. De aquí que el número de grupos que pueden formarse por agrupación de pares sea N-1. Sin embargo, ciertos componentes fueron descartados, dado que no presentan similitud significativa con sus compañeros, estos son los componentes inconexos denotados por nC. El procedimiento utilizado para calcular nC a partir de un grafo de entrada se presenta en el Apéndice B.

En el siguiente diagrama (Figura 5) se resume los pasos señalados por el esquema, además presenta las principales estructuras necesarias para manejar la información. Los dos vectores *edge* representan cada uno de los N elementos, mientras que el vector *weights* de largo M, representa las distancias entre los elementos de los vectores *edge*. En los llamados *height* y *father*, se almacenan los resultados, los valores necesarios para reconstruir un dendograma. El primero almacena las distancias existentes entre los elementos fusionados. El segundo, el índice del cluster formado a partir de la fusión de un elemento. Pop corresponde a un vector auxiliar utilizado para calcular los pesos en la etapa de ponderación.

El algoritmo explicado se utiliza en 4 versiones secuenciales del programa, en cada una se utiliza una precisión diferente, es decir, los números se almacenan y procesan usando distintas representaciones digitales, esto es: precisión simple y doble de punto flotante precisión simple entera, además de una versión especial de enteros de 18 bits. Esto, como se explica en la sección siguiente, permite evaluar los requerimientos de la aplicación en cuanto a la precisión numérica y será determinante para definir la plataforma.

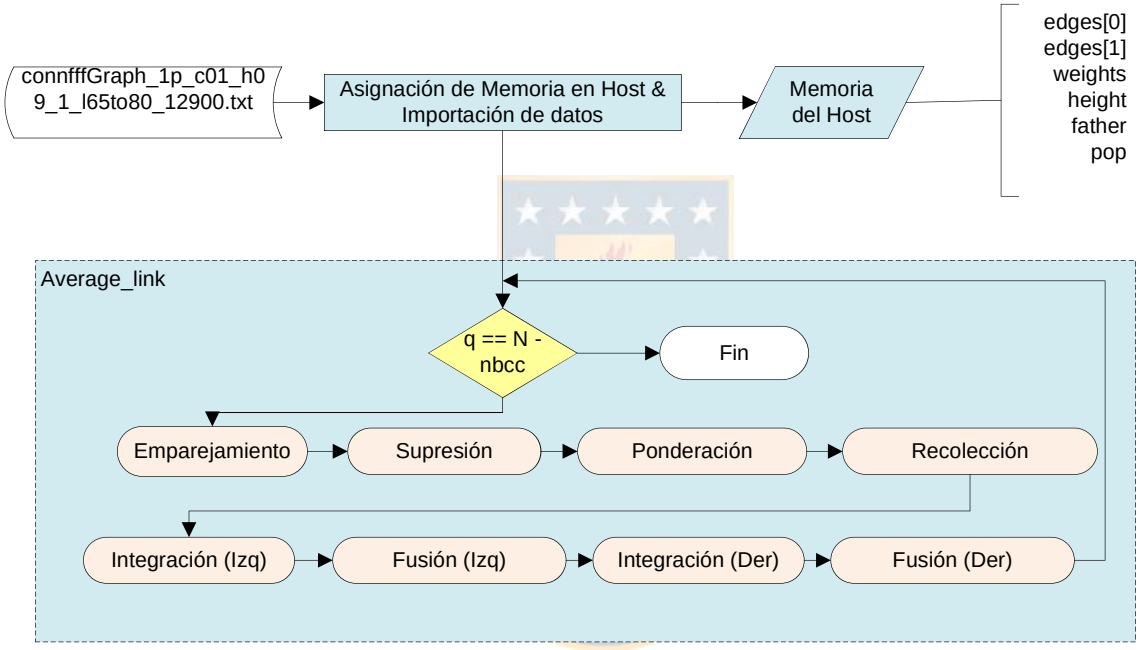

Figura 5: Diagrama de flujo para implementiación en C.

4. Evaluación del algoritmo secuencial

4.1. Datos de prueba

Para realizar la evaluación de esta implementación, se trabajó con un conjunto de datos obtenidos por el estudio señalado en [3]. Sobre éstos se ejecutó una versión original del algoritmo, el cual está escrito en Python. Los resultados obtenidos con esta versión son conocidos y fueron validados en el experimento citado.

Estos datos corresponden a valores de afinidad, los que representan el grado de conectividad entre dos parcelas o conjuntos de voxels en un modelo tridimensional del cerebro. La conectividad se calcula en base a la cantidad de fibras que atraviesan ambas parcelas, como se explica en el Capítulo 2.

4.2. Validación de los resultados

Para validar la implementación se ha considerado como salida del algoritmo dos vectores, los necesarios para reconstruir un dendrograma. El primero, es un vector con las afinidades obtenidas al momento de realizar cada fusión, es decir, a la creación de cada cluster. Esto corresponde a la distancia de unión para dos lineas en un dendrograma. El segundo es un vector que identifica las parcelas padres de cada par.

4.2.1. Evaluación de la precisión numérica requerida

Como parte del proceso de validación las pruebas se realizaron con distinto nivel de precisión numérica, como forma de medir el impacto que podría presentar una arquitectura diferente. De esta manera, el algoritmo implementado, se realizó en 4 versiones: punto flotante de doble precisión en 64 bits, punto flotante de precisión simple en 32 bits, punto fijo en 32 bits y una emulación de punto fijo con enteros de 18 bits. De estas precisiones numéricas, se han escogido las tres primeras por ser las que se encuentran frecuentemente implementadas en la mayoría de las arquitecturas de cómputo general y la última por que corresponde a la resolución nativa de los multiplicadores más típicos incrustados en “Field Programmable Gate Array (FPGA)”, una plataforma reconfigurable para implementar circuitos digitales.

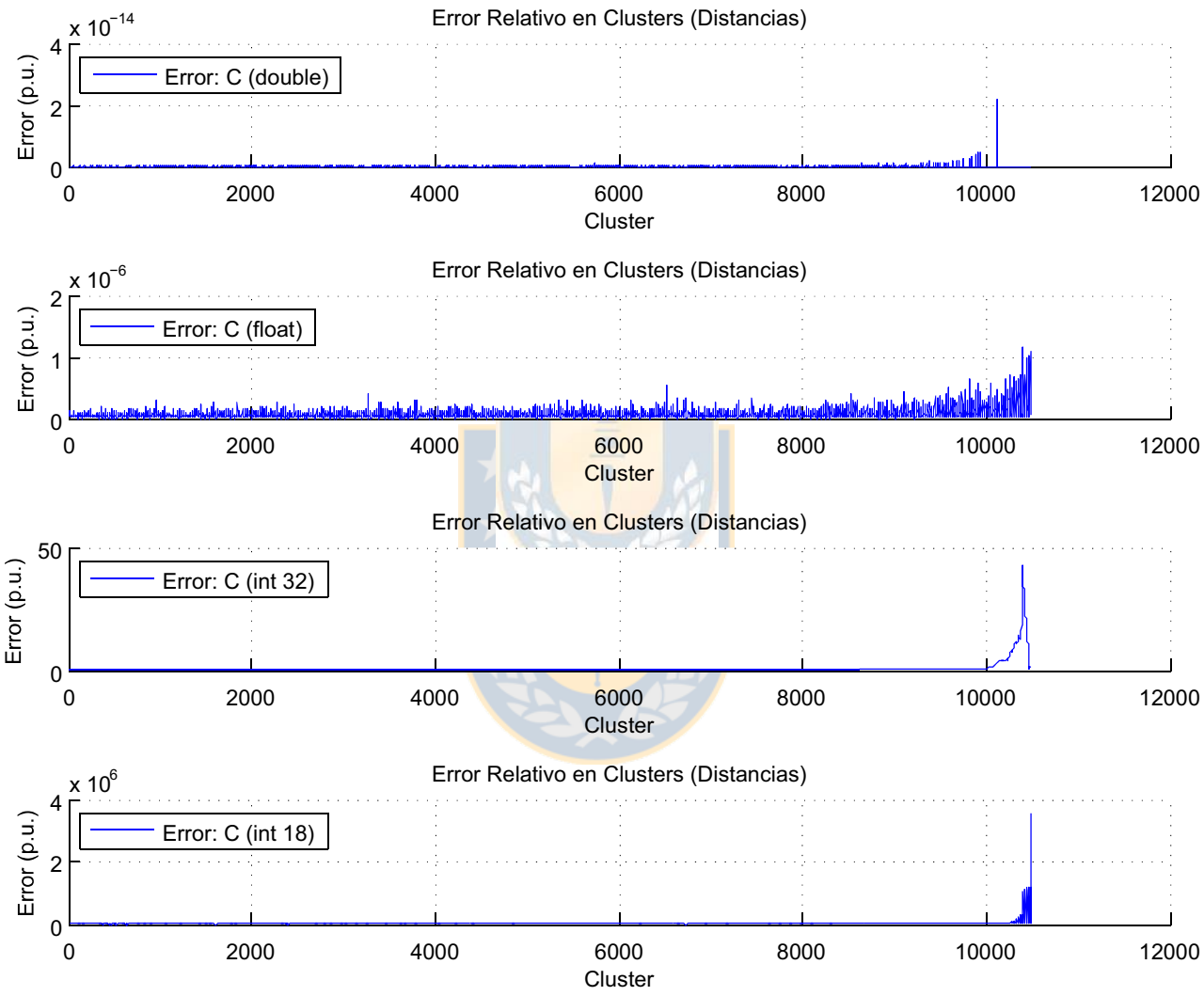

Figura 6: Error relativo calculado entre el vector de distancias de referencia y el obtenido con cada precisión.

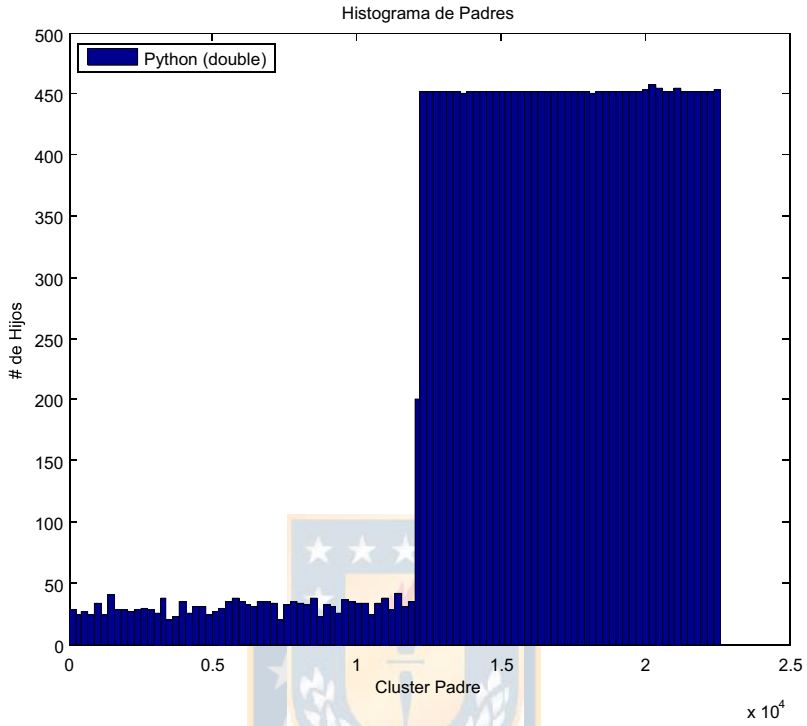

Figura 7: Histograma construido a partir del vector de padres (algoritmo original) .

La Figura 6 presenta los vectores de error calculados a partir de los resultados como:

$$e^k_{rel} = \frac{|heights^k_{ref} - heights^k_p|}{heights^k_{ref}} \tag{4}$$

Donde e^k_{rel} corresponde al error relativo de una muestra k, calculado ente el vector de distancias de referencia, proveniente de la implementación en Python, $heights_{ref}$ y $heights_p$, correspondiente a una precisión p con la nueva implementación.

De la imagen se observa que entorno a la formación de los últimos clusters el error en las distancias se incrementa. Para las aritméticas de punto flotante el error es despreciable, es decir, sus valores no se traducen en un cambio en la forma en que son generados los clusters. Esto no sucede con las arquitecturas de punto fijo en las que el error relativo resulta varios órdenes de magnitud más grande. Estos valores indican que los clusters se generan en distancias diferentes a la implementación original, reflejando indi-

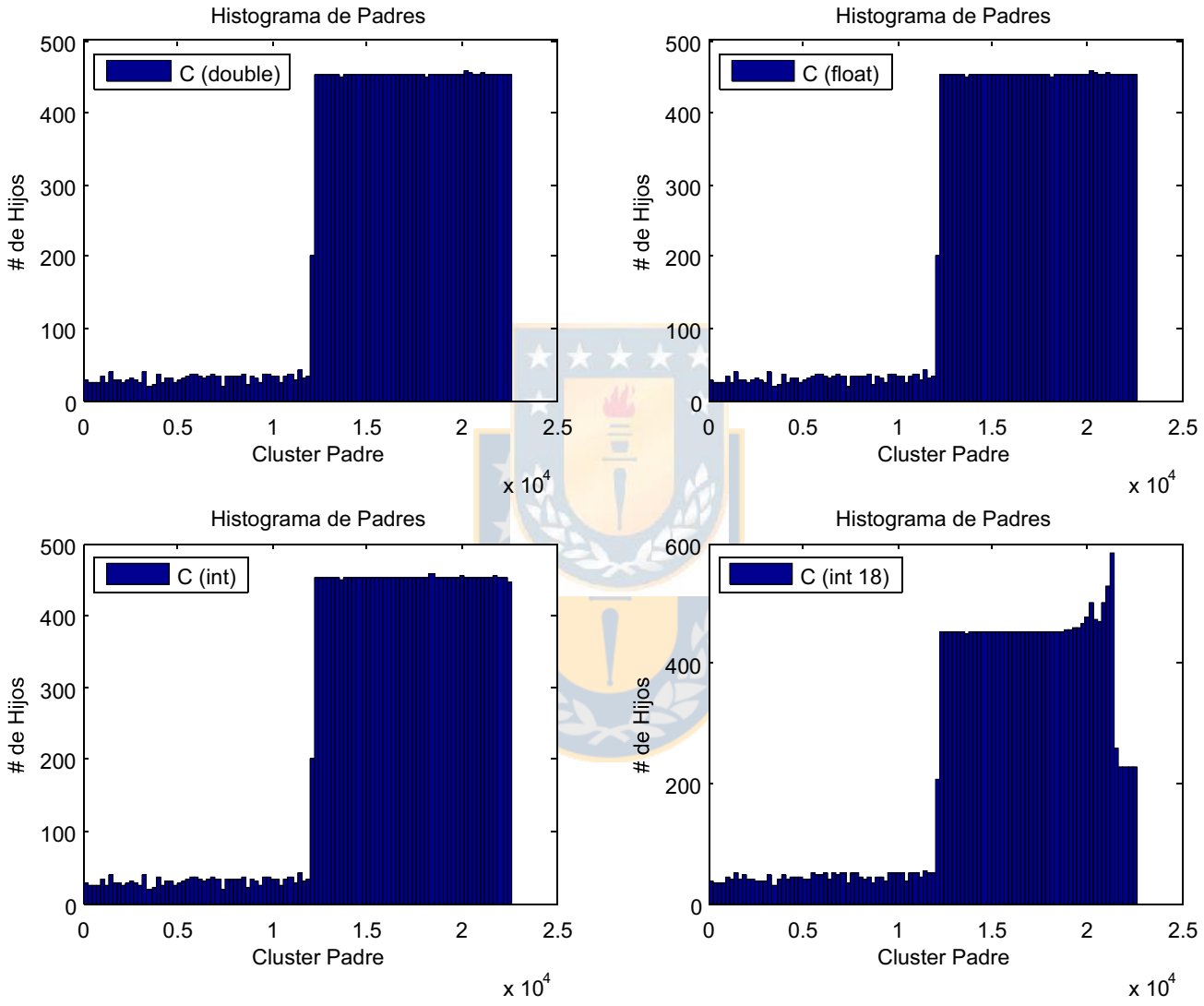

Figura 8: Histograma construido a partir de vector de padres (versiones propias).

rectamente variación en los elementos que los componen. En consecuencia se descarta la implementación en una plataforma que opere con aritmética de punto fijo.
La evaluación que puede realizarse sobre el vector de padres, no es significativa si se compara uno a uno, tampoco es relevante la diferencia entre sus elementos ya que estos sólo representan etiquetas dentro del sistema. De esta forma se decide analizar la distribución de sus elementos. Para esto se revisa el vector de padres de forma gráfica en las Figuras 7 y 8, que presentan histogramas construidos a partir de la cuantización del vector en 100 grupos. De esta forma se puede tener una idea de como varía la distribución de los padres al avanzar la formación de clusters. En la Figura 7 se observa este histograma para la versión original del algoritmo, mientras que en la Figura 8 se observa el contraste con las versiones implementadas. Revisando esta última, de izquierda a derecha, arriba se tiene precisión doble, la forma aparece idéntica a la referencia. Luego se muestra precisión simple, donde se ve una distribución similar. Abajo, el caso siguiente es precisión de 32 bits en aritmética de punto fijo, acá la distribución presenta pocas variaciones cerca de los clusters finales. Esto significa, que existirán algunas diferencias en las ramas superiores del dendrograma, pero se conserva una estructura original. Finalmente en 18 bits de punto fijo la diferencia es mayor y más prolongada, indicando que la estructura del dendrograma en la parte superior se distorsiona completamente.
Con estos antecedentes se descarta la implementación en aritmética de punto fijo al no ser recomendable para esta aplicación, ya que resulta en una alteración de los clusters resultantes.

4.3. Perfil de ejecución

Se realizó un perfil de tiempo sobre la implementación secuencial, esto permite clasificar algunos segmentos del código según su costo relativo dentro de la ejecución lineal. Para el análisis, el código fue dividido en segmentos, a cada uno de estos se le ha asociado un verbo según el pseudo código de la sección 3.2. Los resultados del perfil se muestran en el Cuadro 1, con los que se construye el gráfico de la Figura 9. Las mediciones corresponden a la implementación en C con precisión doble de punto flotante, el tiempo de referencia original total se ha medido para la implementación Python en doble precisión de punto flotante.

Figura 9: Utilización de la CPU por segmento.

Las estadísticas del Cuadro 1 muestran el perfil obtenido para la implementación secuencial. Además se tiene la contrastación del tiempo total de ejecución entre la nueva implementación y la original. Debido a que esta última no fue intervenida, no se tiene un perfil más detallado. Se observa que es durante la etapa de Ponderación donde se invierte la mayor parte del tiempo, esto puede atribuirse a las operaciones de multiplicación y división involucradas. Por otro lado las operaciones que requieren recorrer la matriz completa sin este tipo de operaciones como Emparejamiento, Supresión y Recolección presentan tiempos similares. Los pasos comentados y en particular el de ponderación presentan una oportunidad fuerte de aceleración debido a que la poca dependencia de datos en el flujo de cada paso simplifica la ejecución paralela.

Segmento Análogo	Tiempo (s)	Tiempo Original(s)
Emparejamiento	43.89	–
Supresión	64.60	–
Ponderación	118.84	–
Recolección	71.83	–
Integración	0.10	–
Fusión	0.15	–
Total	299.42	1274.37

Cuadro 1: Resumen de perfil de ejecución para implementación secuencial

4.4. Conclusiones

De los resultados se concluye que para esta aplicación especifica, no es conveniente una implementación que utilice aritmética de punto fijo. Por otro lado, resulta irrelevante, al utilizar aritmética de punto flotante, hacerlo con precisión doble o simple. Esto debido a que, para ambas, el error relativo de las distancias esta permanentemente por debajo de $1{,}5x10^{-4}$, lo que indica que no se altera en absoluto la formación de los clusters. Esto se refleja también en la distribución de los padres.

El perfil muestra que hay cuatro zonas del código donde se espera tener un impacto significativo, debido a que los largos tiempos de cómputo se deben, en mayor medida, a la cantidad de datos procesados y no a la complejidad algorítmica. Además, se sabe que la dependencia entre los cálculos de cada elemento es baja por lo que los procesos son altamente paralelizables.

5. Estrategias de implementación paralela

5.1. Arquitecturas de procesamiento paralelo sobre GPU

5.1.1. Computación en GPU

Las unidades de procesamiento gráfico (Graphical Processing Unit (GPU)) se originaron como piezas de hardware especificas capaz de realizar eficientemente operaciones únicas rígidas para cálculos tridimensionales y de representación gráfica. A la fecha, éstas han evolucionado para convertirse en procesadores programables con sólidas interfaces de alto nivel[6].

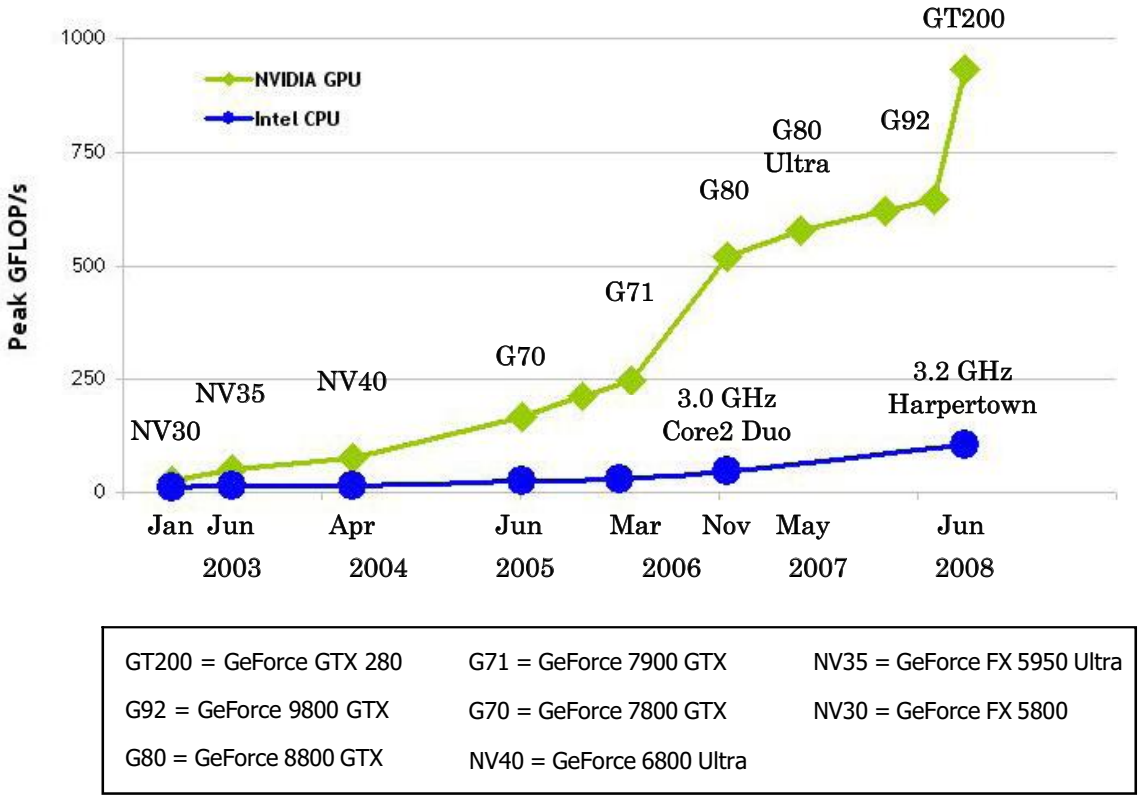

Figura 10: Comparación de rendimiento GPU/CPU en las últimas generaciones [7].

La Figura 10 compara la evolución que han tenido GPU y CPUs desde Enero de 2003 hasta Junio de 2008 en términos cantidad máxima de instrucciones de punto flotante por segundo. Se observa que en los últimos años hay una gran distanciación entre las capacidades de los GPU respecto a los CPU. Las GPU se caracterizan por poseer un reducido conjunto de instrucciones, lo que se traduce en una reducción del área utilizada por cada core. Sin embargo, los requerimientos de cómputo de la computación gráfica están asociados a utilizar estas instrucciones sobre gran cantidad de datos. Esto ha tenido como consecuencia que los GPU evolucionen en torno a la escalabilidad y puedan integrar centenares de pequeños cores en un sólo chip, explicando los resultados del gráfico.
Según la discusión del párrafo anterior las GPU se presentan como una óptima plataforma de cómputo. Sin embargo, para obtener provecho de esta situación ser requiere que el algoritmo a implementar permita la ejecución masivamente paralela de operaciones sobre una gran cantidad de datos. Esto se observó en el algoritmo de agrupación jerárquica que se implementó.

5.1.2. Lenguajes modernos para computación sobre GPU

OpenCL

Es un estándar industrial abierto para programar un grupo heterogéneo de CPUs, GPUs u otras dispositivos computacionales digitales. Establece una plataforma de software para programación paralela, incluyendo un lenguaje, API, bibliotecas y sistemas para asistir el desarrollo.[8]
OpenCL aparece como un intento de unificar y estandarizar el trabajo que comenzaron empresas como NVIDIA, AMD y Microsoft, entre otros, para que programadores puedan desarrollar código eficiente de forma transversal a cualquier plataforma.
Si bien el futuro de OpenCL es prometedor, a la fecha de este trabajo el soporte adoptado por la industria en el mercado no tiene la madurez suficiente para considerarlo una alternativa.

ATI Stream SDK

Para el desarrollo de aplicaciones GPU, el SDK de ATI proporciona Brook+ un compilador optimizado para lenguaje Brook, una variante de ANSI C. Además dispone de un conjunto de bibliotecas específicas

para funciones matemáticas, genéricas y de procesamiento de video optimizadas para sus arquitecturas. Recientemente se ha incorporado soporte OpenCL para esta plataforma.

NVIDIA CUDA

La arquitectura CUDA fue lanzada en Noviembre de 2006, e introduce un nuevo modelo de programación paralela. NVIDIA provee un ambiente de desarrollo basado en C, incluyendo nvcc, un compilador de C capaz de reconocer directivas específicas de CUDA. Sin embargo, el ambiente de programación ha sido pensado para soportar distintas interfaces para el desarrollo de aplicaciones paralelas como son: OpenCL, DirectCompute o FORTRAN [7].

Al estar basado en un lenguaje estándar como ANSI C, CUDA provee una muy rápida curva de aprendizaje. Por otro lado, la fuerte entrada en el mercado de parte de NVIDIA favorece el escalamiento y la portabilidad de una aplicación para esta plataforma, haciendo de esta una alternativa económica para facilitar tareas de análisis científico.

El desarrollo de un lenguaje uniforme, transversal a las arquitecturas provistas por esta compañía, presenta una ventaja sustancial permitiendo reutilización y escalamiento del software. Nuevas funcionalidades son agregadas con el desarrollo de nuevas generaciones de hardware, sin embargo, existe una política de retrocompatibilidad que garantiza lo anterior. Con cerca de cuatro años de desarrollo, CUDA ha generado la formación de una gran comunidad virtual, que sin duda favorece el aprendizaje y fomenta la discusión en torno a los desarrollos.

5.2. CUDA: Detalles de la arquitectura

5.2.1. Modelo de ejecución

CUDA utiliza un modelo de ejecución basado en Parallel Thread Execution (PTX), una arquitectura virtual para ejecución paralela de hebras, la cual provee una capa de abstracción entre el desarrollo de aplicaciones y las arquitecturas específicas de cada generación.[9] En este modelo se introduce el concepto de Single Instruction Multiple Threads (SIMT), el cual se sustenta en la base de que el hardware se distribuye en uno o más multiprocesadores, conteniendo cada uno un conjunto de múltiples núcleos ca-

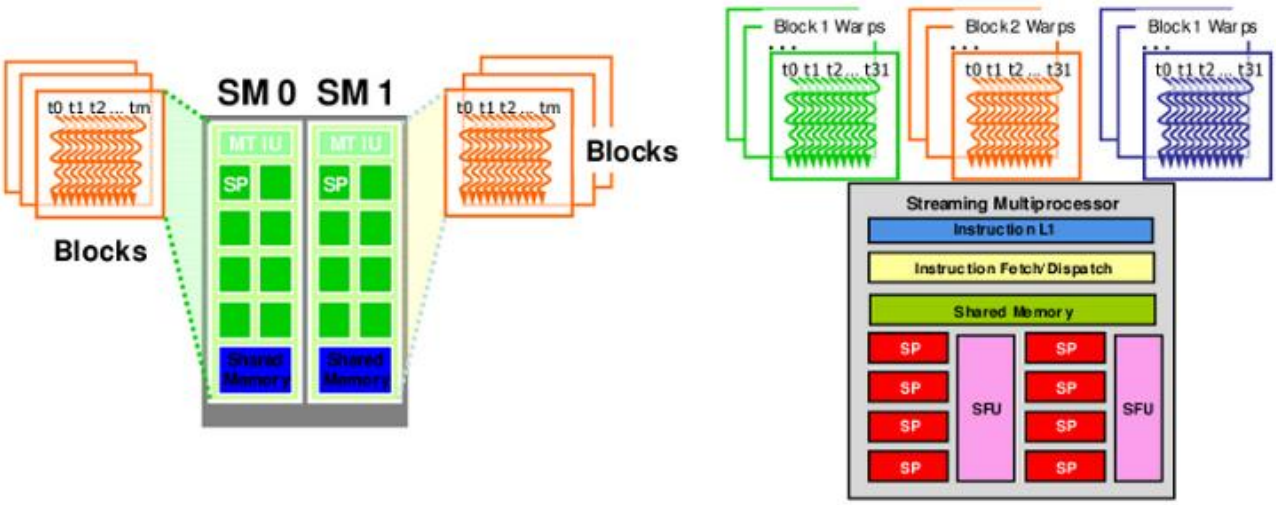

Figura 11: Asignación de hebras en la serie de GPU GeForce 8. Original en "GPU Computing"[6].

paces de ejecutar la instrucción de una hebra cada uno, manteniendo una propia dirección de instrucción y estado de los registros.

A cada multiprocesador se le asigna un bloque constituido por un conjunto de hebras, como se muestra en la Figura 11. Para ser ejecutadas, las hebras son agrupadas en warps[2], un grupo que comienza su ejecución de forma conjunta pero que el flujo de alguno de sus elementos puede verse alterado en el proceso. Sólo una instrucción es ejecutada a la vez para un conjunto de warps activos. En el caso de que existan warps divergentes, éstos son ejecutados en serie con cada uno de los flujos posibles, deshabilitando los warps que no pertenecen al flujo. Este comportamiento impacta incrementando el tiempo de ejecución, debido a que coarta el flujo concurrente [10].

5.2.2. Modelo de hardware

Una instrucción SIMT es capaz de controlar múltiples unidades de procesamiento. Para soportar estas instrucciones el hardware referencial se constituye de:

- Un conjunto de registros de 32 bits por procesador.

[2]warp: grupos de hebras que será asignada a los núcleos de un multiprocesador por la unidad SIMT para su ejecución

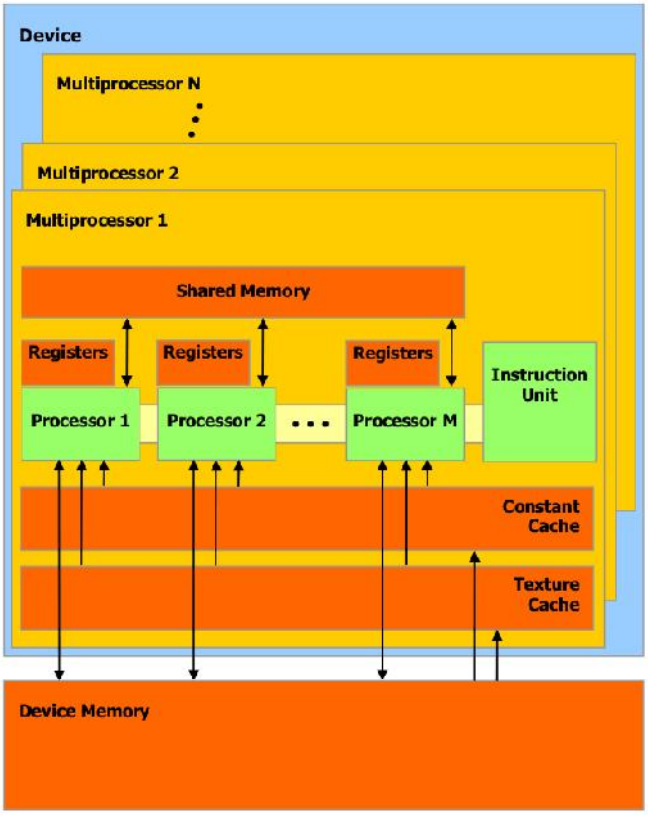

Figura 12: Modelo, conjunto de multiprocesadores con memoria compartida. Tomado de "NVIDIA Compute PTX"[9].

- Un cache paralelo compartido por todos los núcleos del multiprocesador. Constituye un espacio de memoria compartida para las hebras.
- Una memoria cache de sólo lectura compartida por los núcleos escalares, designada como espacio constante de memoria.
- Un cache de sólo lectura para texturas, compartida para los núcleos, que puede ser accedida mediante una unidad especial que implementa distintos medios de direccionamiento y filtrado.

Se puede observar este modelo en la Figura 12. Además las hebras pueden acceder a un espacio de memoria global del dispositivo para lectura o escritura que no utiliza cache.

5.2.3. Sincronización

Como forma de controlar el flujo paralelo de las hebras y evitar carreras críticas, CUDA proporciona la función **__syncthreads()**, que permite sincronizar su ejecución. Un aspecto importante sobre la ejecución de las hebras es que cada instrucción equivalente, perteneciente a un mismo warp, se ejecuta de forma simultánea. Por ende, entre hebras de un mismo warp no se producen carreras críticas.
Además, desde las arquitecturas con capacidad 1.1, NVDIA ha introducido un pequeño subconjunto de instrucciones atómicas [7]. Estas instrucciones permiten realizar operaciones aritméticas que evitan condiciones de carreras críticas y en algunos casos evitan la necesidad de sincronización entre hebras.

5.2.4. Memoria

Un aspecto a tener en cuenta cuando se realizan accesos a la memoria global del dispositivo desde un kernel, es el concepto de acceso ordenado (*coalescing*). Es objetivo de éste es minimizar el número de transacciones al acceder a un bloque de datos desde la memoria global debido a la alta latencia que tiene accederla.
Para esto, es necesario que los accesos a memoria global se hagan a direcciones alineadas a 4, 8 o 16 bytes y que las hebras accedan a los datos de forma secuencial para cada grupo de 16 hebras, es decir medio warp. Es decir, si se lee un dato de 4 bytes en cada hebra, la hebra 0 debe acceder a un puntero de memoria alineado a 4 bytes, mientras que la hebra 1 debe acceder a las 4 siguientes a partir de esta dirección (ver Figura 13). Cumplir con esta condición implica que la transferencia de datos se hará de forma concurrente logrando en una sola transacción la copia de 64 bytes. De lo contrario, esta copia de 64 bytes se realizará mediante 16 transacciones secuenciales.

5.2.5. Sumario

A continuación se destacan los aspectos más relevantes del estudio anterior:

- Los Warps activos se ejecutan de forma sincronizada.
- Warps divergentes son ejecutados en serie.

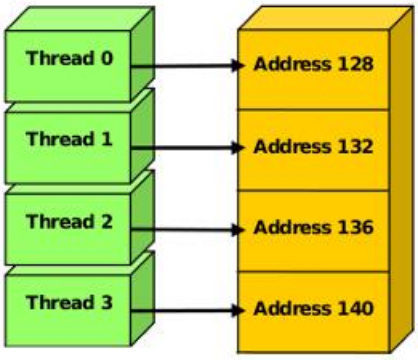

Figura 13: Acceso ordenado a la memoria global.

- Cada núcleo cuenta con una memoria compartida para las hebras que ejecuta y además tiene acceso a una memoria global.
- Además de instrucciones específicas de sincronización, las arquitecturas con capacidad 1.1 cuentan con algunas instrucciones atómicas.
- Para minimizar la cantidad de accesos a la memoria global es importante que los accesos sean ordenados de la misma forma en que son accedidos por las hebras.

5.3. Consideraciones numéricas

5.3.1. Estándar IEEE para representación y aritmética de punto flotante

El estándar IEEE 754-1987 [11] define una representación de punto flotante como una cadena de dígitos caracterizado por tres componentes: un signo, un exponente con signo y una mantisa. Esto puede observarse en la Figura 14. Esta cadena puede o no tener significado numérico y de tenerlo es calculado como el producto con signo entre la mantisa y la base numérica escogida elevada a la potencia del exponente.

Por ejemplo, para una base binaria:

$$valor = (-1)^S \cdot M \cdot 2^E \qquad 1{,}0 \leq M < 2{,}0 \tag{5}$$

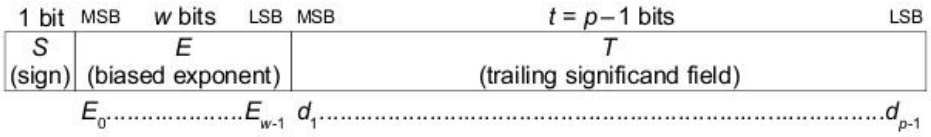

Figura 14: Formato IEEE para representación de punto flotante

Notación normalizada y codificación en exceso exponente

La forma de codificar un número mediante esta representación utiliza el concepto de normalización de la mantisa y codificación en exceso del exponente.

En el primero, la idea es codificar una mantisa capaz de representar un número de forma única. El método utilizado consiste en almacenar la parte fraccionaria de un número entre uno y dos que al multiplicarse por la potencia elevada al exponente pueda reconstruir el valor original. Por ejemplo, el valor $0{,}75_D$, puede representarse por $1{,}1_B \cdot 2^{-1}$ equivalente a 0,11 o bien $\frac{1}{2} + \frac{1}{4}$, pero con la restricción de que la mantisa M debe estar entre 1 y 2. Así es posible anotar solo un 10...n para referirse a 1,10...n, de esta forma tenemos almacenados en n bits un valor de n+1.

La codificación en exceso del exponente por su parte consiste en una forma de ubicar de forma consecutiva los valores posibles para los exponentes desde los más negativos a los positivos. Para conseguir esto, al exponente real se le agrega un valor $A = 2^n - 1$, donde n corresponde al número de bits utilizados para representar el exponente.

5.3.2. Denormalización y trucado abrupto a cero

Como consecuencia de los anterior, la notación en punto flotante no incluye al número cero. Además a medida que los números son más cercanos a éste, el error dado al mayor número cercano representable se hace más grande. Para lidiar con este problema, el estándar IEEE define el método de denormalización. Éste consiste en utilizar el exponente cero de la notación en exceso, para alterar la interpretación de los valores de la mantisa. Con este método, cuando el exponente es 0, la mantisa pasa de 1.xx...n a 0.xx...n.

Una alternativa a este método, más fácil de implementar, es la de el truncado abrupto a cero. Con este método, todo numero de exponente 0 es directamente truncado al valor 0.

5.3.3. Impacto del algoritmo sobre la precisión de los resultados

Cuando una operación aritmética genera un resultado que no puede ser exactamente representado por la notación, esto genera un error. Es común que esto suceda con la notación de punto flotante cuando se realiza el desplazamiento requerido para igualar los exponentes, proceso requerido por ejemplo, antes de una operación de suma. Cuando las diferencias entre los números son significativas, puede suceder que la cantidad de bits necesarias para mantener el aporte de cada operando no sea suficiente.
Esta nueva consideración acarrea un problema que debe ser considerado al pasar de una implementación secuencial a una paralela. El error acarreado es dependiente del flujo del programa ya si se opera sobre grupos similares de números se reducirá la magnitud del error acumulado. Como consecuencia, entre dos implementaciones paralelas o entre una paralela y otra secuencial, se tendrá una diferencia del error total acumulado para una secuencia de operaciones asociativas.

5.3.4. Diferencias entre el Std IEEE 754 y la implementación adoptada por CUDA

En general CUDA ha intentado adoptar casi por completo el estándar IEEE para la representación y aritmética de números en punto flotante, sin embargo, en algunos aspectos clave se han alejado de éste. Los siguientes aspectos son los relevantes para este trabajo:

- En precisión simple de punto flotante, el método de denormalización no está soportado y números con esta notación son convertidos a cero.
- La división y raíz cuadrada son implementadas de forma no estándar a través de los recíprocos.
- Para la multiplicación y suma sólo se soportan los métodos de redondear hacia el próximo par y redondear hacia el cero.
- La instrucción fusionada de multiplicación y suma (FMAD). Ésta acelera la serie de operaciones, pero trunca el resultado intermedio de la multiplicación, agregando error al resultado respecto al que se obtendría con las dos operaciones en serie que cumplan el estándar.

Todas estas limitaciones en el trato numérico serán eliminadas en la próxima generación de GPUs de NVIDIA llamada Fermi [12]. En esta, la aritmética cumple el estándar IEEE 754-2008.

5.3.5. Conclusiones

Del estudio anterior se concluye la existencia de un error intrínseco en la adaptación del algoritmo. Este error está asociado al proceso de paralelización, a la composición del algoritmo y a la elección de la arquitectura. De esta forma, la factibilidad de la implementación estará sujeta a los requerimientos de precisión impuestos por la información a ser analizada.

6. Desarrollo paralelo del algoritmo en lenguaje CUDA

6.1. Desarrollo basado en kernels

CUDA establece como elementos de ejecución los kernels, programas breves destinados a ser ejecutados de forma masivamente paralela. De esta forma, para adaptar la versión secuencial, se divide el código de la implementación en C en segmentos de código breves y con alto potencial de ejecución paralela. Para cada una, se desarrolla un kernel y una función C asociada que permite la ejecución de éste. La Figura 15 muestra un diagrama de flujo con la estructura del algoritmo.

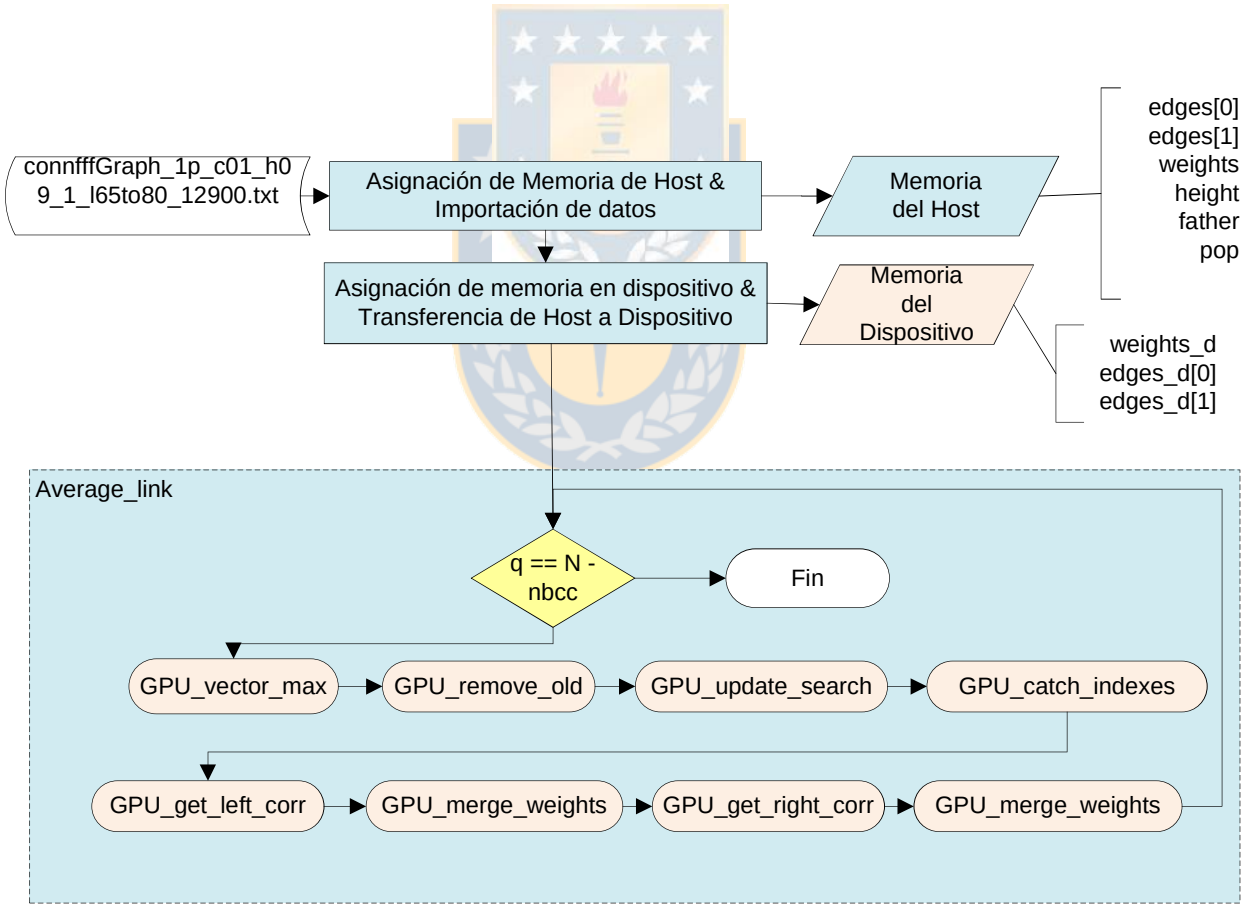

Figura 15: Diagrama de flujo para implementación CUDA.

A continuación se detalla el proceso asociado a cada uno de los kernels desarrollados.

Emparejamiento y Supresión (vector_max_reduction_kernel_evol) A través de reducción paralela se realiza una búsqueda del mayor elemento dentro de la matriz de afinidad.

Ponderación (update_new_clusters_kernel) Actualiza las afinidades de los nuevos clusters ponderando las entradas de la matriz de afinidad por la población del nodo respectivo.

Recolección (search_edges_write_weight) Busca dentro de los vectores de nodos aquellos elementos que están siendo fusionados y reemplaza las etiquetas con las del nuevo cluster.

Recolección (catch_index) Identifica los nodos cuyas afinidades deben ser actualizadas ya que que representan a un nuevo cluster. La nueva afinidad corresponde a la suma de las de los elementos que corresponden a nodos iguales.

Integración (get_corr) Obtiene los nodos identificados por la etapa de Recoleción. El vector obtenido debe ser ordenado de forma ascendente.

Fusión (merge_weights) Efectúa la suma de las afinidades que fueron ponderadas y que ahora corresponden a las distancias al nuevo cluster.

6.1.1. Búsqueda de máxima afinidad

Esta etapa del algoritmo es implementada por el kernel **vector_max_reduction_kernel_evol**, envuelto por la función GPU_vector_max. Para conseguir el objetivo se utilizó una técnica denominada reducción paralela. Esta técnica permite realizar una operación utilizando como operando, virtualmente, todos los elementos de un arreglo. Este método permite mantener la utilización de los recursos paralelos al máximo. El procedimiento consiste en realizar una copia de la matriz a la memoria compartida. Para esto cada hebra está encargada de copiar un dato. De forma simultánea a esta transferencia se realiza la primera comparación entre cada elemento cargado y el que se encuentra a una distancia igual al número total de hebras. Esto permite mantener sólo valores máximos y reducir el tamaño de la búsqueda siguiente a la mitad. El siguiente paso es realizar una nueva comparación pero con la mitad de los nuevos datos, lo que

se repite hasta quedar con un solo elemento.

En la Figura 16 se puede observar como en cuatro iteraciones se obtiene el máximo de un vector de diez y seis elementos. Con esto se logra reducir una operación que teóricamente tiene un costo de $O(N)$ a una de sólo $O(\log_2 N)$.

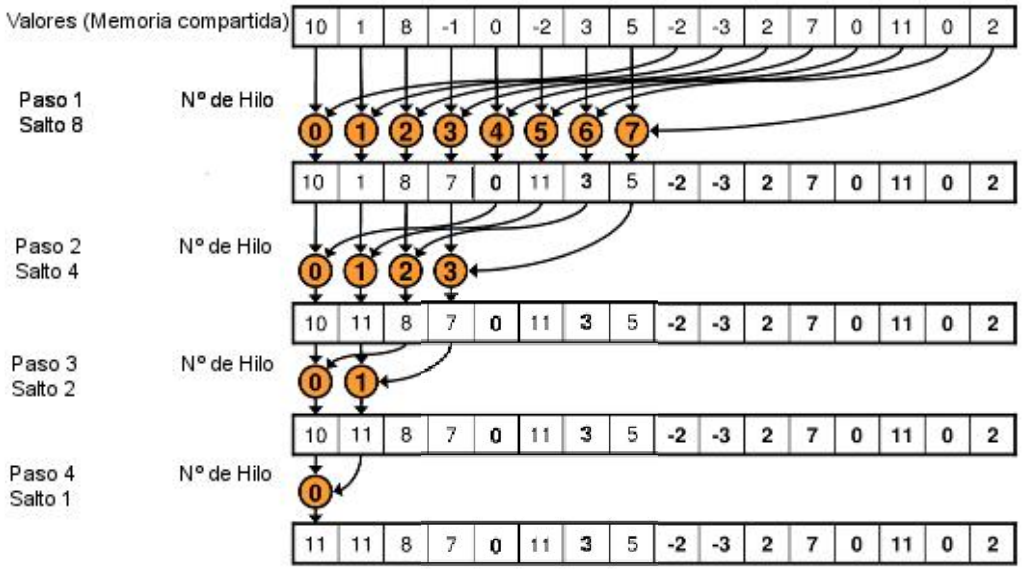

Figura 16: Reducción implementada para la búsqueda del máximo.

6.1.2. Remover ocurrencias del máximo

Una vez identificados, los vértices de mayor afinidad deben fusionarse. Para esto, el primer paso es eliminar todas las entradas que involucran este par. Esto se realiza por el kernel **search_edges_write_weight** en la función GPU_remove_old. El procedimiento adoptado para esto, es particionar los vectores de vértices y pesos y repartir el trabajo entre el número de hebras total con que se decide operar. Con esto se reduce el número de operaciones desde $O(N)$ a $O(N/t)$, donde t es el número de hebras que se ejecutan de forma simultánea.

Es también importante señalar que el acceso a la memoria se realiza de forma alineada y en orden para cada hebra, lo que acelerara las transferencias de memoria necesarias.

6.1.3. Ponderación de las distancias y actualizacion de pesos

Para calcular la afinidad entre el cluster generado por la fusión hacia el resto de los elementos, es necesario buscar en la matriz por cada entrada de cualquiera de los vertices involucrados en la fusión. Al encontrarse, sus pesos son ponderados por el número de elementos agrupados por cada uno, que es mantenido y actualizado en otro vector para cada iteración. Esto se realiza por el kernel **update_new_clusters_kernel** dentro de la función GPU_update_search. El método de particionamiento es igual al de la sección anterior, esperando una optimización similar.

6.1.4. Fusión y actualización final de los pesos

Finalmente, es necesario realizar la sumatoria de todas las distancias que fueron ponderados para poder actualizar la matriz de afinidad. Esto se realiza en dos pasos. Primero, se elabora un listado de los vértices que se deben incorporar en la sumatoria, esto es realizado por una combinación de los kernels catch_index y get_corr en las funciones GPU_catch_index y GPU_get_corr respectivamente. Estas funciones, realizan su trabajo en un esquema similar a los kernels anteriores.

En el segundo paso sin embargo, se debe ser más cauteloso, ya que al actualizar la matriz de afinidad es posible encontrar situaciones de carreras críticas.

Esto se ilustra en el siguiente ejemplo:

índice	Vertice
0	v_1
1	v_1
2	v_1
3	v_3
4	v_2
5	v_2

Cuadro 2: Matriz con los índices que deben ser actualizados por vértice.

El Cuadro 2 muestra una lista con índices de los elementos que deben ser actualizados, las entradas asociadas a estos en el vector de afinidad deben reemplazarse con la del nuevo cluster. La segunda columna del Cuadro 3 representa un vector con las afinidades para 6 pares de elementos.

índice	$K=0$	$K=1$	$K=2$	...	K=5
0	a	$-inf$	$-inf$	...	$-inf$
1	b	$a+b$	$-inf$	...	$-inf$
2	c	c	$a+b+c$	...	$a+b+c$
3	d	d	d	...	d
4	e	e	e	...	$-inf$
5	f	f	e	...	$e+f$

Cuadro 3: Ejemplo de actualización de la matriz de pesos. Implementación secuencial.

En el Cuadro 3 se aprecia la evolución que tendrá la matriz durante una implementación secuencial. En k pasos se logra el resultado final. Se infiere de acá que una implementación paralela directa producirá resultados incorrectos al realizar una suma simultánea entre pares de elementos solicitados, produciendo una carrera crítica. La solución propuesta consiste en realizar la suma de forma secuencial para cada índice asociado a un vector determinado. El Cuadro 4 muestra la situación con esta implementación. A diferencia de la situación anterior, en la que el número de pasos dependía del largo de la lista de actualización, ahora éste depende de la cantidad máxima de índices asociados a un vértice determinado.

Indice	$K=0$	$K=1$	$K=2$
0	a	$-inf$	$-inf$
1	b	$a+b$	$a+b$
2	c	c	$a+b+c$
3	d	d	d
4	e	$-inf$	$-inf$
5	f	$e+f$	e

Cuadro 4: Ejemplo de actualización de la matriz de pesos. Implementación propuesta.

Es importante mencionar que este procedimiento se sostiene en el hecho de que los indices son ordenados durante cada iteración. El algoritmo que efectúa el ordenamiento, por simplicidad, se ejecuta sobre la CPU. El sobre cálculo introducido como consecuencia, se reflejará principalmente en un incremento en los tiempos medidos para de transferencias entre la memoria del CPU y la del GPU.

7. Validación y contrastación de resultados

7.1. Validación de los resultados

De igual forma y con el mismo set de datos especificado en la sección 4.2 los resultados fueron validados mediante la evaluación de los vectores de afinidades y de padres.

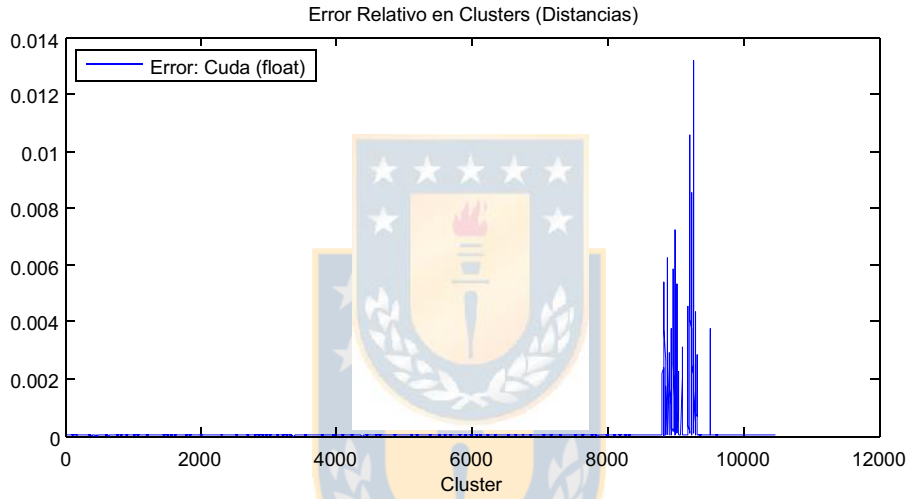

Figura 17: Error relativo calculado entre el vector de distancias de referencia y el obtenido con la implementiación CUDA.

En la Figura 17 se puede ver el error entre el vector de distancias de referencia y el obtenido mediante la implementación en CUDA. El error se resalta en los últimos clusters. Se observa que el máximo error relativo no supera el 1.5 %.
Por otro lado, en la Figura 18 se aprecia la distribución del vector de padres para la implementación en CUDA y el de la versión original. De este se observa que ambos tienen una composición similar.

Además de esta revisión visual de la distribución, se calculó la tasa de similitud respecto al vector de padres original. De los 22604 clusters finales, alrededor de un 10 % de éstos pertenece a un padre distinto de la implementación original, es decir el cluster generado al mezclarse con otro tiene un nombre

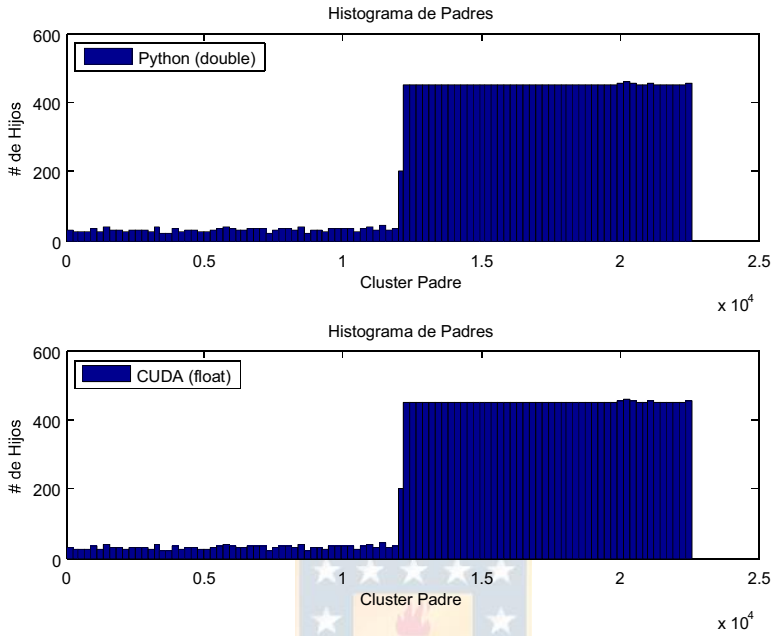

Figura 18: Histograma construido a partir del vector de padres para implementación en CUDA.

diferente debido a que la mezcla ocurrió en otra secuencia. De éstos sólo un 3.7 % poseen un elemento padre que difiera en más de 10 clusters de diferencia, lo que es consecuente en términos de la similitud de la distribución. Para entender esta conclusión, debe considerarse que debido al nuevo flujo concurrente, se espera tener variaciones dado que el error acumulado (comentado en el Capítulo 5) se propaga por rutas distintas. Sin embargo, esta diferencia no es significativa para la aplicación en cuestión debido a que la estructura del árbol (determinada por la distribución de los clusters) se conserva. Para la aplicación esto se traduce en que los grupos principales de fibras se mantienen.

7.2. Perfil de ejecución

Es posible realizar un perfil de ejecución para la implementación en CUDA desarrollada gracias a la utilización de registros especiales provistos por la arquitectura, que son utilizados como contadores. Este análisis ha sido tomado en cuenta a través del desarrollo del algoritmo y ha sido relevante para optimizar y converger a la implementación final.

Los contadores pueden dividirse en dos grupos, los que son incrementados durante cada warp y los que son incrementados por cada acceso de memoria de un multiprocesador perteneciente a un Texture Processing Cluster (TPC). Es importante destacar que los contadores en cualquier caso son sólo representativos de un subgrupo de multiprocesadores en la GPU. [13]

En la Figura 19 se presenta el porcentaje de utilización de la GPU. Esta información, al igual que los contadores, es obtenida a través de la herramienta de perfil provista por NVIDIA (cudaprof). Se observa de la Figura que el mayor porcentaje tiempo utilizado por la GPU es atribuible al kernel *vector_max_reduction_kernel_evol*, seguido por *catch_indexes* y finalmente con una cifra igualmente importante, las transferencias de memoria entre la utilizada por el procesador y la del dispositivo gráfico.

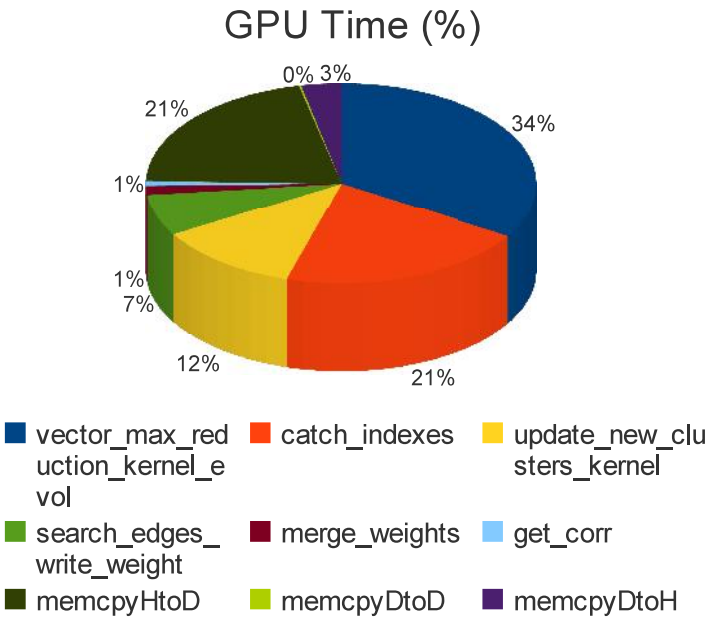

Figura 19: Utilización de la GPU por kernel.

A continuación se describen los contadores que describen estadísticas asociadas a cada warp:

branch Número de saltos tomado por las hebras ejecutando el kernel.

divergent branch Número de saltos dentro de un warp que siguió un camino de ejecución diferente del resto.

instructions Número de instrucciones ejecutadas.

warp serialize Número de warps que debieron ser ejecutados en secuencia por un conflicto de acceso a los bancos de memoria, sea local compartida o global.

sm cta launched Número de bloques que se lanzaron en un multiprocesador.

Cuadro 5: Resumen de contadores por warp

Method	sm cta	branch	div. branch	instr.	warp serialize
vector_max_reduction_kernel_evol	138,91	3749,21	554,63	20143,2	19253,4
catch_indexes	344,44	12760,4	361,59	36668,1	0
update_new_clusters_kernel	344,43	19658,3	382,16	42356,3	0
search_edges_write_weight	344,5	11400,6	364,66	28362,3	0
merge_weights	0,28	9,5	2,31	69,69	0
get_corr	0,27	4,31	0,27	22,85	0

Observando el Cuadro 5, se ve que la carga mayor le corresponde a los 4 primeros kernels, que son además los que lanzan el mayor número de hebras, ya que operan sobre la matriz completa. Sin embargo, en el primer kernel se aprecia una diferencia notable, una gran cantidad de los warps fueron ejecutados de forma serial por la divergencia en el flujo de ejecución. Esto es esperable debido a que la estructura de datos con que trabaja es ligeramente más compleja que el vector con que trabajan los otros kernels.

Los contadores relacionados con cada TPC son:

gld uncoalesced Número de operaciones de carga desde memoria global en que se accede de forma no ordenada.

gld coalesced Número de operaciones de carga desde memoria global en que se accede de forma ordenada.

gst uncoalesced Número de operaciones de escritura a la memoria global en que se accede de forma no ordenada.

gsd coalesced Número de operaciones de escritura a la memoria global en que se accede de forma ordenada.

cta launched Número de bloques de hebras en un TPC.

Cuadro 6: Resumen de contadores por TPC

Method	gld uncoal.	gld coal.	gst uncoal.	gst coal.	cta
vector_max_reduction_kernel_evol	53315,6	0,17	1093,98	34,45	277,8
catch_indexes	0	5510,99	123,18	9,91	688,89
update_new_clusters_kernel	119,41	11059,9	249,54	347,91	688,57
search_edges_write_weight	0	5552,45	0,26	1,03	689
merge_weights	318,88	9,29	232,54	0,16	0,62
get_corr	73,5	4,62	0	34,59	0,62

Las estadísticas del Cuadro 6 permiten justificar la tesis anterior para el peor desempeño del kernel *vector_max_reduction_kernel_evol*. En éste se observa una gran cantidad de accesos desordenados a memoria (uncoalesced), atribuibles a la estructura que maneja los datos. Esta limitación era conocida a la hora de diseñar dicho kernel pero la decisión de implementación se tomó en base a un trade off entre la cantidad de instrucciones y el orden de accesos de memoria.

7.3. Resultados (aceleración)

El hardware utilizado para realizar las pruebas está caracterizado en el cuadro 7. Además de las características que impactan el rendimiento es importante destacar la versión de la arquitectura CUDA. La GPU utilizada es compatible con la arquitectura 1.1. Esto, entre otras cosas, permite la utilización de un pequeño conjunto de funciones atómicas.

Cuadro 7: Características del Hardware utilizado en las pruebas.

Tipo	Modelo del Chip	Frecuencia	Cores	Memoria	Arquitectura CUDA
CPU	AMD Athlon X2 5200+	2.6 Ghz	2	2GB 800Mhz DDR2	N/A
GPU	NVIDIA GeForce 9600GT	675 MHz	64	512 1.8Ghz DDR3	1.1

El Cuadro 8 muestra los tiempos de referencia obtenidos para la ejecución del algoritmo sobre el set de prueba. Estos datos son tomados desde el timer de sistema, a partir de la función *clock()*. De esta tabla se obtiene que la implementación en C logra una aceleración [3] de 4.34 respecto a la versión original, mientras que la versión CUDA tiene una aceleración de 51.44 sobre la misma y 11.85 respecto a la versión en C.

Cuadro 8: Resumen de tiempos de ejecución.

Tiempos de Ejecución (s)		
double/Python	C/float	CUDA/float
1274,37	293,5	24.77

Con el Cuadro 9 se puede contrastar la aceleración lograda para cada parte del algoritmo. Se ve que la etapa de emparejamiento resulta la más demorosa en la GPU. Esto se explica con la estructura de datos que requiere, ya que resulta en un acceso más lento a los datos e incrementa la complejidad. Por otro lado, se puede destacar la aceleración en la etapa de ponderación que es la más larga debido a que además de recorrer la matriz completa realiza las operaciones mas lentas como multiplicaciones.

Es importante mencionar que los tiempos tabulados para el algoritmo en GPU corresponden al tiempo de ejecución de las funciones escritas en C que se utilizan para ejecutar los kernels, es decir sus envoltorios o wrappers.

[3] Aceleración calculada como el tiempo original sobre el tiempo de la versión mejorada. ej: $S = \frac{T_c}{T_{cuda}}$

Segmento Análogo	Tiempo CPU (s)	Tiempo GPU(s)
Emparejamiento	42.92	12,76
Supresión	63.17	1,84
Ponderación	116.21	2,5
Recolección	70.24	4,21
Integración 7	0.1	0,11
Fusión	0.15	0,61
Total	292.8	22.03

Cuadro 9: Comparación de tiempos parciales entre CPU y GPU.

7.4. Escalamiento

Esta sección realiza una estimación de la aceleración máxima que puede conseguirse con la implementación actual ejecutada sobre un procesador más avanzado.

El tiempo neto utilizado en los kernels se ha medido como 19.75 segundos, que equivale al 77.6 % del tiempo total consumido.

Utilizando la ley de Amdahl, es posible determinar el máximo teórico que se puede lograr a través de mejorar la parte ejecutada en la tarjeta gráfica. Sea f la fracción que se puede mejorar y $S_{parcial}$ la aceleración sobre la misma, Amdahl plantea:

$$S_{total}(S_{parcial}) = \frac{1}{(1-f) + \frac{f}{S_{parcial}}} \tag{6}$$

Para obtener el máximo teórico se calcula el limite en que $S_{parcial} \to \infty$, lo que resulta en $S_{total} = 4{,}4710$. Esto nos deja con un tiempo de ejecución de tan sólo 5 segundos. Sin embargo es evidente que la aceleración parcial se debe modelar como una función no lineal ya que depende de una serie de factores como la cantidad de núcleos disponibles, la velocidad de los procesadores, la capacidad de los buses, sincronización y dependencia en los datos. De esta manera $S_{parcial}$ podría saturarse mucho antes de que se alcance el máximo teórico.

Para estimar el impacto del incremento de núcleos se realizaron pruebas adicionales sobre una tarjeta GeForce 9800GTX. Esta GPU cuenta 256 núcleos. El tiempo total de ejecución medido fue de 19.85 se-

gundos. Es decir, al cuadriplicar el número de núcleos, sólo se obtiene una mejora del 25 %.

A partir de estos resultados se puede construir una estimación del escalamiento al incrementar la disponibilidad de núcleos. Se asume una dependencia lineal entre la aceleración parcial y el número de núcleos. Con el modelo obtenido se construye la Figura 20. En esta se destaca con un cuadrado el resultado del experimento y se proyecta la aceleración total en función del número de núcleos. Se observa por ejemplo, que para una arquitectura de 512 núcleos se puede esperar una aceleración cercana a 1.5 sobre la implementación actual.

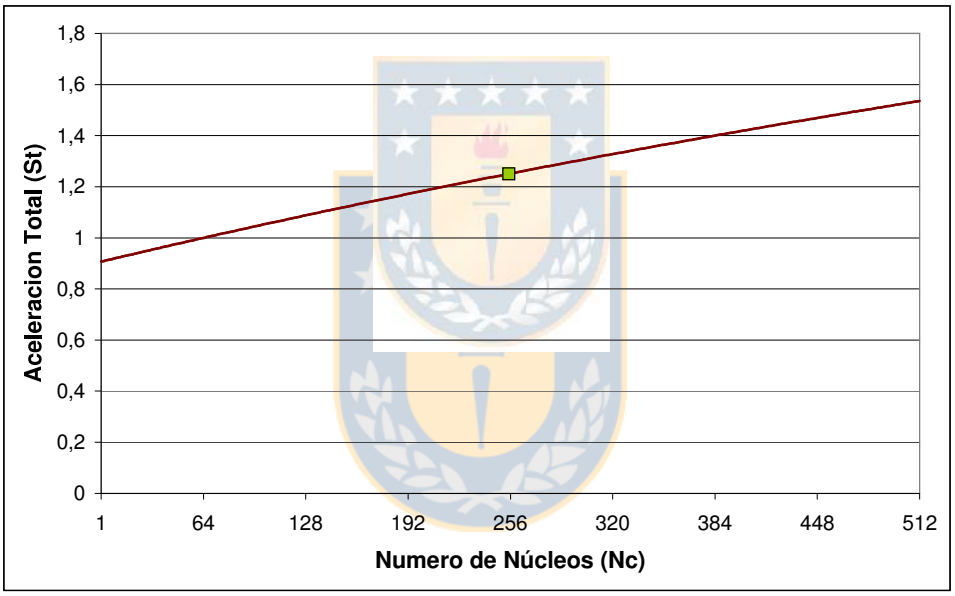

Figura 20: Diagrama de flujo para implementiación CUDA.

Otra observación es que el 22.4 % es código se ejecuta en la CPU, parte de este es potencialmente paralelizable y puede ser implementado sobre la GPU. Este corresponde a un ordenamiento del vector recibido en la etapa de recolección, un algoritmo eficiente de ordenamiento diseñado para ejecutarse en paralelo podría permitir reducir aun más el flujo secuencial y reducir las transferencias de memoria.

8. Conclusiones y discusión

8.1. Conclusiones

Este trabajo ha cumplido su objetivo principal. Luego del estudio y evaluación del algoritmo secuencial se ha determinado con éxito una plataforma capaz de explotar el potencial de ejecución paralela del algoritmo. Se ha demostrado que las características de la aplicación permiten obtener una buena implementación en GPU, logrando una exitosa aceleración que supera 50 veces del tiempo original de ejecución. Además, se ha conseguido una implementación secuencial escrita en ANSI C que supera la original con una aceleración de más de 4 veces, mejora que se explica sólo en el hecho de obtener un mejor control del algoritmo gracias a que la codificación se desarrolla a un nivel mas bajo que en el planteamiento inicial.
Se ha calculado una estimación del máximo teórico para la aceleración que puede lograrse con la implementación en CUDA, se obtuvo que la aceleración potencial de mejorar la tecnología no supera 4.47 veces sobre la versión actual. También se elaboró una estimación de la aceleración total que se puede conseguir al variar la cantidad de núcleos disponibles.
Al ser un lenguaje de alto nivel, basado en C++, CUDA permite una corta curva de aprendizaje, lo que facilitó el desarrollo. Además gracias al bajo costo del hardware requerido, fue posible realizar este desarrollo con recursos independientes.
Otras desventajas aparecen en el hecho de que CUDA utiliza un modelo propietario de NVIDIA y a pesar de que se conoce un modelo práctico, la arquitectura real es desconocida y puede variar en cada GPU. Esto finalmente impacta en la capacidad de modelar el comportamiento del algoritmo. Tampoco es posible obtener estadísticas detalladas para todos los núcleos de forma directa, a pesar de que existen los registros detallados en el Capítulo 7, éstos sólo analizan uno de los procesadores.

8.2. Trabajo futuro

La aplicación de la implementación secuencial, escrita en C, en el experimento que se explica en el Capítulo 2, es bastante directa. Actualmente se tienen noticias de una aplicación exitosa y los resultados en

aceleración se correlacionan con los obtenidos. Sin embargo para la versión en CUDA, la implementación es menos directa, ya que requiere de una inversión en hardware. De esta forma, es aún necesario definir una plataforma específica y justificar económicamente la inversión, para luego adaptar la implementación en CUDA con el estudio original.

En cuanto a optimizar el algoritmo más allá, es posible incursionar en algoritmos de búsqueda que sean propicios para correr en GPU, ya que debido que en la etapa de Integración esto lograría reducir las transferencias entre la memoria de la CPU y la de la GPU. Se espera además que al mejorar la tecnología se reduzcan las limitaciones que radican en la notación y manipulación numérica digital, que se explicaron en el Capítulo 5.

Referencias

[1] R. R. Sokal and C. D. Michener, "A statistical method for evaluating systematic relationships," *University of Kansas Scientific Bulletin*, vol. 38, pp. 1409–1438, 1958.

[2] M. Dash, K. L. Tan, and H. Liu, "Efficient yet accurate clustering," *Data Mining, IEEE International Conference on*, vol. 0, p. 99, 2001.

[3] P. Guevara, Y. Cointepas, D. Riviére, C. Poupon, B. Thirion, and J.-F. Mangin, "Inference of a HARDI fiber bundle atlas using a two-level clustering strategy," Para revisión en MICCAI 2010.

[4] A. K. Jain, M. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, no. 3, pp. 264–323, September.

[5] J. MacQueen, "Some methods for classification and analysis of multivariate observations," *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, 1:281-297.*

[6] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[7] NVIDIA, "NVIDIA CUDA: Programming guide ver 2.3.1," *NVIDIA Corporation, Santa Clara, California, August 2009*.

[8] K. O. W. Group, "The OpenCL specification," *December 2008, Available Online: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf*.

[9] NVIDIA, "NVIDIA compute PTX: Parallel thread execution, ISA version 1.4," *NVIDIA Corporation, Santa Clara, California, March 2009*.

[10] D. B. Kirk and W. W. Hwu, *"Programming Massively Parallel Processors: A Hands-on Approach"*. Morgan Kaufmann, 2010.

[11] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 2008.

[12] NVIDIA, “NVIDIA Next Generation CUDA Compute Architecture: Fermi,” *Whitepaper*, pp. 13 –14, 2009.

[13] NVIDIA, “NVIDIA CUDA: Visual profiler version 2.3.1,” *NVIDIA Corporation, Santa Clara, California, August 2009*.

A. Contenido digital

El objetivo principal de este anexo es facilitar la re utilización del código desarrollado. Para esto, se presentan una descripción de los archivos involucrados y la estructura en que se encuentran, también se describe el prototipo de la función principal y se muestra un ejemplo de archivo entrada que puede ser procesada por esta implementación.

A.1. Archivos del proyecto

De forma anexa a este informe, los todos archivos del proyecto se han recopilado en la siguiente manera:


```
./
├── doc
│   ├── src
│   │   └── informe_src.tar.gz
│   └── pdf
│       └── informe.pdf
└── code
    ├── hac_C
    │   └── hac_float.c
    └── hac_cuda
        ├── hac_float.cu
        ├── kernels_and_wrappers.cu
        └── kernels_and_wrappers.h
```

A continuación se presenta el detalle de los archivos mencionados:

informe_src.tar.gz Contiene todo el código fuente para generar el presente documento.

informe.pdf Este documento.

hac_float.c Código fuente auto contenido que implementa el algoritmo en lenguaje C a través de la función *Average_link*.

hac_float.cu Código fuente de la implementación CUDA, es análogo al anterior, este no incluye los kernels.

kernels_and_wrappers.cu Código de los kernels y funciones diseñadas para ejecutarlos.

kernels_and_wrappers.h Cabeceras para el pre procesamiento de los kernels y sus wrappers.

A.2. Prototipos

Tanto en la versión escrita en C como en la escrita en CUDA, el algoritmo se implementa a través de una función principal llamada *Average_link*.

```
// Prototipo de la version C
int Average_link(
        // Direccion de un vector de dos punteros a vectores de elementos
                int **edges,
        // Direccion de un vector conteniendo las afinidades para cada par
                float *weights,
        // Direccion de un vector que recibe las distancias de clustering
                float *height,
        // Direccion de un vector que recibe los padres de cada cluster
                int *father,
        // Direccion de un vector vacio utilizado para ponderacion
                int *pop,
        // Numero de componentes no conexas
                int nbcc,
        // Numero de pares
                int nV,
        // Numero de elementos
                int nE)
```

La función análoga en la implementación cuda, corresponde a:

```
// Prototipo de la version CUDA
int Average_link(
        // Direccion de un vector de dos punteros a vectores de elementos
                int **edges,
```

```
// Analogo en memoria del dispositivo
        int **edges_d,
// Direccion de un vector de estructuras con afinidades para cada par
        struct smax *weights,
// Analogo en memoria del dispositivo
        struct smax *weights_d,
// Direccion de un vector que recibe las distancias de clustering
        float *height,
// Direccion de un vector que recibe los padres de cada cluster
        int *father,
// Direccion de un vector vacio utilizado para ponderacion
        int *pop,
// Numero de componentes no conexas
        int nbcc,
// Numero de pares
        int nV,
// Numero de elementos
        int nE)
```

Se puede observar que los prototipos de ambas versiones son bastante similares con la excepeción de que para cuda se agruegan punteros a vectores dentro de la memoria del dispositivo. Además para el vector de afinidades, se utiliza una estructura especial que almacena un par (valor,índice) debido detalles de la implementación.

A.3. Datos de entrada

Para la realización de las pruebas, se ha establecido un formato de texto plano como elemento de entrada. Siendo bastante simple, la primera línea presenta dos columnas, la primera corresponde a la cantidad de elementos a clasificar, mientras que la segunda a la cantidad de pares disponibles con las respectivas afinidades entre ellas. Después de esta línea se presentan los datos en tres columnas, mostrando el número asociado al primer elemento, al segundo y la afinidad respectivamente. Esto se muestra en el ejemplo siguiente:

```
12119 705352
```

```
2 1 0.816667
1 2 0.816667
7 6 1.10606
...
```

B. Cálculo de componentes no conexas

El código presentado a continuación permite calcular las componentes no conectadas dentro de un set a analizar.

```
int nbcc_calc(int nV, int **edges) {
        int maxv = 2*nV;
        int *cc;
        int su,sv;
        int n1,ne;
        int k=0;
        int remain = nV;
        for (n1 = 0; n1 < nV; n1++)
                cc[n1] = -1;
        while (remain > 0) {
                n1 = 0;
                while (cc[n1]>-1) n1++;
                cc[n1] = k;
                su = 0;
                sv = 1;
                while (sv > su){
                        su = sv;
                        for(ne=0; ne<nE; ne++){
                                if(cc[edges[0][ne]]==k)
                                        cc[edges[1][ne]] = k;
                                if(cc[edges[1][ne]]==k)
                                        cc[edges[0][ne]] = k;
                        }
                        sv = 0;
                        for(n1=0; n1<nV; n1++)
                                sv += (cc[n1]==k);
```

```
                }
                remain = remain-su;
                k++;
        }
        nbcc = k;
        return nbcc;
}
```