Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

3D-GRABCUT: GrabCut en Volúmenes empleando programación paralela en el GPU con CUDA.

Autor: Pablo E. Temoche Gil
Tutor: Esmitt Ramírez J.
Co-tutor: Rhadamés Carmona S.

Caracas, Mayo de 2011

Resumen

La segmentación de imágenes es un campo de amplio estudio dentro de la computación gráfica. El objetivo principal es obtener una región de interés dentro de un espacio más amplio. Una de las técnicas de segmentación de imágenes 2D creadas recientemente que obtiene excelentes resultados es llamada GrabCut [1]. GrabCut es una técnica basada en el algoritmo de GraphCut [2] donde la imagen es representada como un grafo de flujo, donde el objetivo es encontrar el corte mínimo del grafo para separar la imagen en fondo (*background*) y región a segmentar (*foreground*). En este trabajo se implementa la técnica de GrabCut para segmentar volúmenes bajo un esquema de algoritmo paralelo. La implementación utiliza estructuras de datos y algoritmos paralelos que son ejecutados en el GPU para mejorar los tiempos de respuestas. En la literatura, no se encontraron implementaciones previas de GrabCut en el GPU para volúmenes. Las pruebas demuestran excelentes resultados del algoritmo con los volúmenes de prueba empleados.

Palabras claves GrabCut, segmentación de imágenes, datos volumétricos, CUDA, flujo en redes.

Índice general

Introducción

Planteamiento del Problema

La segmentación de volúmenes es una parte importante en el estudio y análisis de las imágenes médicas para el diagnóstico y posterior planificación del tratamiento, pero este proceso suele tomar un tiempo considerable empleando procesadores de propósito generalen ciertas técnicas donde el proceso no resulta sencillo. Con la tecnología actual, el cálculo necesario para la segmentación se puede hacer de manera más rápida y precisa a través del GPU el cual provee procesamiento gráfico especializado con alto desempeño.

Motivación

En el campo de la medicina asistencial encontramos diferentes formatos de imágenes específicas en el aréa. Estos archivos tienen la difícil misión de reprsentar imágenes con una gran resolución, así como permitir el menor tamaño posible para su posterior uso en la telemedicina.

Las fuentes clásicas que generan imágenes digitales en medicina son :

- La Ultrasonografía.
- La Resonancia Magnética Nuclear.
- La Tomografía Axial Computarizada.

Dichas imágenes son de gran utilidad para los diagnósticos médicos, pero para un análisis más completo se necesita aplicarles diferentes métodos de extracción de las regiones de interes dependiendo del caso de estudio, a dichos métodos se les conoce con el nombre de " **segmentación de imágenes** ".

La segmentación de imágenes es el proceso de separar una imagen en al menos dos regiones distintas, de tal manera que cada región es homogénea con respecto a ciertos criterios de similitud pre-definidos.

Los avances en la tecnología permiten actualmentela utilización de procesadores gráficos más avanzados ayudando a reducir las limitaciones que antes se presentaban en campo de la segmentación mediante hardware gráfico. En este trabajo se propone una implementación del algoritmo de GrabCut de una forma paralela para la segmentación de volúmenes utilizando el GPU bajo la arquitectura CUDA. Dicha implementación obtiene buenos resultados en la segmentación de un subvolúmen de interés (*foreground*) con respecto al resto del volúmen (*background*).

Propuesta de solución

Es por ello que se plantea desarrollar una aplicación computacional que permita realizar la segmentación de volúmenes utilizando la técnica GrabCut cambiando su enfoque de 2D hacia uno 3D, la cual es de alto desempeño usando el hardware gráfico paralelo de la GPU.

Objetivos Generales

Desarrollar un prototipo para segmentar volúmenes, utilizando la ayuda de la GPU para acelerar el cálculo del algoritmo.

Objetivos Específicos

- Implementar el algoritmo GrabCut para volúmenes acelerando su procesamiento con la utilización del GPU.
- Paralelizar las estructuras de datos y algoritmos necesarios que puedan ser ejecutados en el la tarjeta gráfica tomando en cuenta las limitaciones de la misma.
- Construir una interfaz intuitiva y amigable para la segmentación de volúmenes utilizando la técnica antes mencionada.
- Evaluar la aplicación desarrollada.

Alcance de este trabajo

- La implementación al ser desarrollada en la arquitectura CUDA solo se puede ejecutar en dispositivos gráficos NVidia.
- Existen limitaciones con el uso de la memoria de la GPU.
- Por ser un prototipo se trabajará con archivos de datos volúmetricos simples.

Este documento presenta en el capítulo 1 el planteamiento del problema original para este trabajo de grado. A continuación, en el capítulo 2, se explican algunos conceptos necesarios empleados en este documento. El capítulo 3 explica detalladamente el enfoque realizado denominado 3D-GrabCut, el cual consiste en la aplicar la técnica de segmentación de GrabCut para imágenes 3D (volúmenes). Luego, en el capítulo 4 se muestran detalles de implementación de 3D-GrabCut como un algoritmo paralelo ejecutado en el GPU bajo la arquitectura CUDA. Los resultados son analizados y expuestos en el capítulo 5 denominado Pruebas y resultados. Finalmente, se dan las conclusiones y trabajos futuros a realizar en nuestra investigación.

Capítulo 1

Marco Teórico

En el presente capítulo, se explicarán brevemente los conceptos asociados a los datos volumétricos, las funciones de transferencia, la segmentación de volúmenes y finalmente algunas técnicas de segmentación enfocándonos principalmente en GrabCut.

1.1. Datos Volúmetricos

Un conjunto de datos volúmetricos, según Hansen y Johnson [3], es un conjunto V de muestras (x, y, z, w), donde w representa alguna propiedad de los datos en una ubicación 3D (x, y, z). Dicho valor puede ser simplemente 0 ó un entero $(i \neq 0)$ perteneciente a un conjunto I, el valor 0 indicaría que dicha posición es vacía y el valor i indicaría la presencia de un objeto O_i. Los datos pueden tomar varios valores; con dichos valores se representan algunas propiedades de los datos, incluyendo por ejemplo: color, densidad, calor o presión, etc. El valor w puede ser un vector representando por ejemplo: velocidad en cada posición, resultados de múltiples modalidades de exploración, como anatómicas (Tomografías Computarizadas-*Computed Tomography* **CT**, Imágenes de Resonancia Magnética-*Magnetic Resonance Imaging* **MRI**) y de imágenes funcionales (Tomografías por emisión de positrones-*Positron Emission Tomography* **PET**, Imágenes de Resonancia Magnética Funcionales-*Funcional Resonance Magnetic Imaging* **fRMI**), o tripletas de color (**RGB**). Los datos del volumen pueden ser variantes con el tiempo, en cuyo caso V se convierte en un conjunto de muestras en 4D (x, y, z, t, w).

1.2. Volume Rendering

Según Levoy [4] el Volume Rendering (en lo sucesivo VR), se define como el proceso mediante el cual datos volumetricos son procesados para mostrar una proyección en un espacio 2D. Se puede decir que el VR es la simulación aproximada de la propagación de la luz a través de un medio semi-transparente representado por el volumen. Las bases que constituyen el proceso de visualización son los modelos físicos de propagación de la luz sobre materiales que tienen índices de color y opacidad variables. Los datos usados en VR provienen de un espacio tridimensional discreto, dividido en **vóxeles**[1].
Por el hecho de ser un volumen de datos, la cantidad de procesamiento necesaria para representar estos datos aumenta en la medida en que las dimensiones del volumen aumenta. El VR se mantiene como área activa en el campo de la visualización científica y continuamente surgen avances para mejorar su velocidad y calidad.

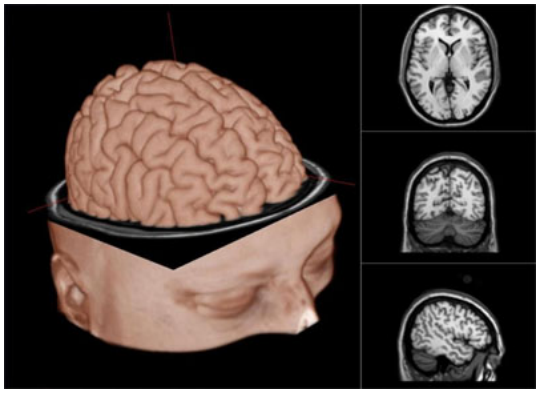


Figura 1.1: Visualización Volumétrica obtenida de una Tomografía computarizada

La visualización de volúmenes es un método de extracción de información significativa de un volumen de datos mediante el uso de gráficos interactivos e imágenes, y se refiere a la representación, modelado, manipulación, y despliegue de dichos volúmenes de datos [5]. Su objetivo es proporcionar mecanismos dentro de los conjuntos de datos volumétricos para la toma de muestras en estructuras voluminosas y complejas. Esto abarca una gama de técnicas de proyección y sombreado en un volumen de datos, así como para la extracción interactiva de información significativa mediante transformaciones, cortes, la segmentación, la translucidez de control, mediciones, etc. Normalmente, el volumen de datos se representa como una cuadrícula regular 3D discreta de elementos

[1]La palabra proviene de la contracción del término en inglés "*volumetric pixel*", es la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, el equivalente del píxel en un objeto 2D.

de volumen y se agrupan en un búfer de volumen. Como alternativa, otras estructuras de datos y formatos son empleados para el almacenamiento y la manipulación de los datos, como por ejemplo la descomposición en octrees [6].

1.2.1. Proceso de Visualización de Volúmenes

El proceso de visualización simula el comportamiento de la luz al atravesar un medio semi-transparente y mientras lo hace, pueden ocurrir varios fenómenos:

- La luz es absorbida por los elementos del volumen.
- La luz se dispersa en los elementos del volumen.
- La luz se emite desde los elementos del volumen.

Aunque la idea general de VR es simular este proceso, no es tan sencillo en la práctica, pues si se tomaran en consideración todos estos factores, la cantidad de cálculo involucrado sería desmesurada. Numéricamente consistiría en evaluar por completo la simulación del proceso físico, estando presente la simulación del medio circundante, reflexión, refracción entre otros. Una reducción del modelo óptico general es suficiente para hacer una visualización de los datos de manera apropiada. En el modelo simplificado de VR solamente se estudia cómo, en el proceso de transporte, la luz es absorbida y emitida por los elementos del volumen. Lo anterior puede ser resumido en la siguiente ecuación tomada de [7]:

$$L(x) = \int_{x}^{x_b} e^{-\int_{x}^{x'} \phi_t(x'')\mathrm{d}x''} . \epsilon(x')\mathrm{d}x \tag{1.1}$$

Donde $L(x)$ es la radiancia calculada en términos de una variable unidimensional de posición llamada x, y se toma en cuenta la interacción de los coeficientes de extinción (opacidades) $e^{-\int_{x}^{x'} \phi_t(x'')\mathrm{d}x''}$ y de los colores $\epsilon(x')\mathrm{d}x$ para cada muestra del volumen. Actualmente no es posible evaluar numéricamente de manera exacta la integral en un computador por el hecho de requerir sumas infinitas. Por lo tanto, una integración numérica se requiere. Según [8] la aproximación numérica para la integral del VR más común, es el cálculo de una suma de Riemann para n segmentos iguales de longitud $d = \frac{D}{n}$. (Donde $D = x_b - x$).

Aproximando los valores en la ecuación anterior queda de la siguiente manera:

$$L(x) = \sum_{i=0}^{n-1} e^{-\sum_{j=0}^{i-1} \phi_j \Delta_x} . \epsilon_i \Delta_x = \sum_{i=0}^{n-1} \epsilon_i \Delta_x . \prod_{j=0}^{i-1} e^{-\phi_j \Delta_x} \tag{1.2}$$

Haciendo algunas reducciones, se obtiene la siguiente ecuación:

$$\begin{aligned} L(x) &= \sum_{i=0}^{n-1} c_i \alpha_i . \prod_{j=0}^{i-1} 1 - \alpha_j \\ &= c_0\alpha_0 + c_1\alpha_1(1-\alpha_0) + c_2\alpha_2(1-\alpha_0)(1-\alpha_1) + ... \\ &\quad ... + c_{n-1}\alpha_{n-1}(1-\alpha_0)...(1-\alpha_{n-2}) \end{aligned} \tag{1.3}$$

En la que c_i representa el valor del color y α_i representa la opacidad para el i-ésimo vóxel de los datos de entrada, tomando n muestras en una dirección de visión dada. La ecuación anterior se conoce como la función de composición de VR.

1.2.2. Función de Transferencia

El volumen puede modelarse como un material semitransparente, en el cual el color y la opacidad de cada vóxel se obtiene a partir de funciones aplicadas a sus respectivos valores escalares. Estas funciones se denominan *Funciones de Transferencia* (FT) y el proceso de aplicar tales funciones a los valores escalares de cada vóxel se denomina *Clasificación*. A continuación, explicaremos dos de las clasificaciones existentes: Pre-Clasificación y Post-Clasificación.

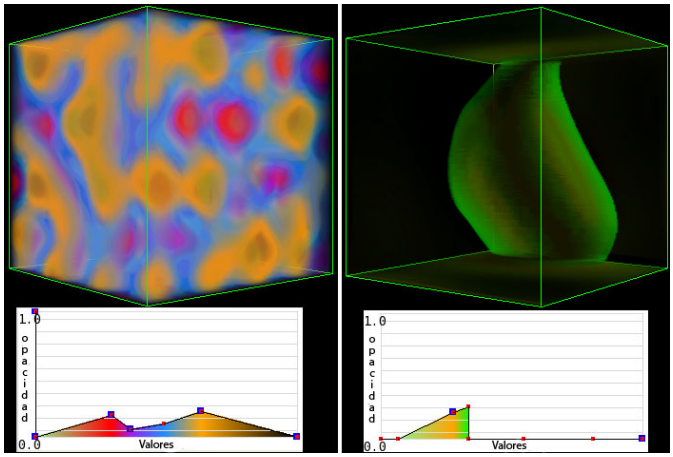


Figura 1.2: Dos funciones de tranferencia distintas aplicadas al mismo conjunto de datos.

Para la construcción del volumen resultante proveniente de las muestras, es necesaria una interpolación entre las mismas para la construcción de vóxeles inexistentes en el volumen a partir de los vóxeles existentes. La diferencia entre estos dos tipos básicos de clasificación es el orden en el que se lleva a cabo los procesos de aplicación de la función de transferencia e interpolación de muestras.

1. **Pre-Clasificación:** Dado el volumen, se toman las muestras y se aplica la función de transferencia y luego la interpolación.
2. **Post-Clasificación:** Primero se interpolan las muestras del volumen para después aplicar la función de transferencia.

La pre-clasificación y post-clasificación generan resultados distintos cuando la interpolación no es conmutativa con la función de transferencia original. Al aplicar pre-clasificación, las imágenes tienden a ser más difusas que utilizando post-clasificación.

1.3. Segmentación de Volúmenes

La segmentación de imágenes (*en este caso de estudio las imágenes tridimensionales o volúmenes*) es un proceso fundamental en diferentes áreas como por ejemplo la medicina, la bioingeniería, en el campo de entretenimiento, entre otros. Aunque se han planteado muchas alternativas para resolver el problema, aún no existe una que pueda cubrir todas las necesidades. La segmentación es un proceso donde la imagen se divide en diferentes regiones para aislar las regiones de interés. La segmentación de imágenes se define como el particionamiento de una imagen en regiones constituyentes que pueden ser o no solapadas, las cuales son homogéneas con respecto a alguna característica tales como intensidad o textura [9]. Si definimos el dominio de un volumen como V, por medio de la segmentación se definirá un conjunto $S_k \subset V$, en cuyo caso la unión de dichos conjuntos formará todo el espacio de dominio de V. Así, los conjuntos S_k deben satisfacer:

$$V = \bigcup_{k=1}^{N} S_k \tag{1.4}$$

donde cada S_k está conectado. Idealmente, un método de segmentación encuentra conjuntos que corresponden a diferentes estructuras o regiones de interés de la imagen. Al eliminar la restricción de que las regiones estén conectadas, determinar los conjuntos S_k es llamado *clasificación de píxel* y a los conjuntos se les denomina *clases*. La clasificación de píxeles frecuentemente es un objetivo deseable en el tratamiento de imágenes,

particularmente cuando se necesita clasificar regiones desconectadas que pertenecen a la misma clase. La determinación del número de clases K en la clasificación de píxeles puede ser un problema complejo, por lo que generalmente se asume conocida, basado en conocimientos previos.

1.4. Técnicas de Segmentación

El número de algoritmos de segmentación encontrados en diferentes fuentes de estudio es muy elevado. Muchos de estos algoritmos son específicos para un problema en particular, teniendo poca significancia para muchos otros problemas. A continuación presentaremos algunas técnicas importantes para la segmentación de imágenes.

1.4.1. Umbralización

Esta es probablemente la más simple de las técnicas de segmentación para volúmenes. En esta técnica, un único valor denominado umbral es utilizado para crear una partición binaria de intensidades. Todos los vóxeles con intensidades superiores al umbral se agrupan en una clase y aquellos con intensidad por debajo del umbral se agrupan en otra clase. El uso de un único valor de umbral determina un volumen binario segmentado.

Esta técnica puede extenderse a la utilización de múltiples umbrales, cuando una región está definida por dos umbrales, un umbral *inferior* y un umbral *superior*. Cada vóxel del volumen de entrada pertenece a una de las regiones basado en su intensidad. Esta técnica se conoce como multiumbralización. En la Figura 1.3 se muestra un histograma perteneciente a un volumen al cual se le va a aplicar una umbralización tomando dos valores para los umbrales T_1 y T_2 los cuales generan tres regiones (1,2,3) como se puede apreciar en dicho histograma.

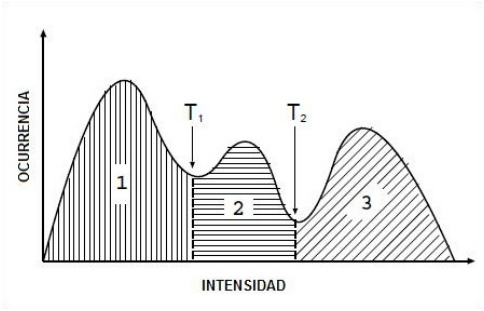


Figura 1.3: Histograma con dos Umbrales T_1 y T_2 los cuales dividen el histograma en tres regiones.

1.4.2. Modelos Deformables

Los fundamentos matemáticos de los modelos deformables conocidos en la literatura como snakes, contornos deformables, contornos activos,etc. representan la confluencia de la geometría, física, y la teoría de la aproximación. La geometría sirve para representar la forma del objeto, la física impone limitaciones a como la forma puede variar en el espacio y tiempo, y la teoría de la aproximación óptima provee las bases formales de los mecanismos para adecuar los modelos a los datos medidos [10]. Los modelos deformables son utilizados para delinear el borde de un objeto de interés en la imagen utilizando curvas o superficies paramétricas cerradas que se deforman bajo la influencia de fuerzas externas e internas. Las principales ventajas de los modelos deformables son su capacidad para generar directamente curvas paramétricas cerradas o superficies de las imágenes y su incorporación de límites suaves que proporciona solidez al ruido y bordes falsos. Una desventaja es que requieren la interacción manual con un modelo inicial y elegir los parámetros adecuados. En la Figura 1.4 se observa el proceso para delinear el borde, primero se selecciona un punto de partida, luego se aplican las fuerzas internas hasta que alcance un estado de equilibrio con las fuerzas externas que son frecuentemente derivadas de la imagen para llevar la curva o superficie hacia la característica de interés deseada.

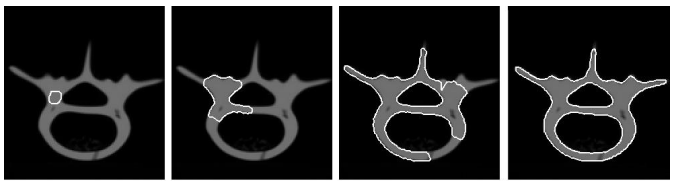


Figura 1.4: Proceso de deformación de una curva empleando modelos deformables.

1.4.3. Crecimiento de Regiones (*Region Growing*)

El objetivo del algoritmo de crecimiento de regiones es determinar zonas uniformes dentro de la imagen, a partir de sus propiedades locales. Es posible efectuar la detección de una o más regiones de interés, de acuerdo a cada aplicación particular. El crecimiento se inicia a partir de la especificación de puntos iniciales para cada región (*semillas*) y luego, mediante un proceso iterativo, se van incorporando a las mismas los vóxeles de la imagen que satisfacen un criterio de conectividad y similitud determinado (como se muestra en la Figura 1.5), hasta que no se encuentren más elementos que cumplan dicho criterio (*Los criterios pueden ser según la intensidad del vóxel, varianza, color, forma, tamaño, etc.*). Una de las desventajas de este algoritmo es la selección de la semilla, la cual debe ser de forma manual. Al mismo tiempo, puede ser sensible al ruido, causando que las regiones extraídas tengan agujeros e inclusive que se desconecten.

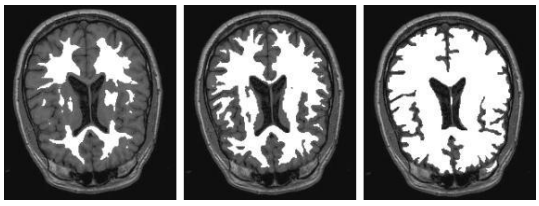


Figura 1.5: Etapas del algoritmo de crecimiento de regiones (corte axial de una MRI).

1.4.4. Transformada de Watershed

La transformación Watershed es una técnica morfológica de segmentación de imágenes de niveles de gris. Es un método de segmentación para imágenes 2D y 3D [11] basado en regiones, que divide todo el dominio de la imagen en conjuntos conexos. El concepto de Watershed procede del campo de la topografía donde en un relieve topográfico, las líneas Watershed son las fronteras de separación entre las cuencas de deyección de ríos y lagos. Además, cada cuenca está asociada a un mínimo local de relieve. La transformación Watershed se puede aplicar a imágenes en escala de grises (multinivel), tomando en cuenta que la intensidad de un punto representa una altura equivalente en un relieve topográfico asociado. El caso tridimensional tiene el mismo significado topológico aunque no se pueda visualizar el relieve 4D equivalente.
Se han propuesto distintos algoritmos eficientes para hallar la transformación Watershed en imágenes digitales multinivel uno de ellos lo presenta Roerdink en [12]. Estos algoritmos asignan una etiqueta especial a los píxeles integrantes de las líneas Watershed, que constituyen la frontera de separación entre las cuencas o regiones Watershed. Adicionalmente también se etiqueta cada una de estas cuencas con un valor identificativo, para facilitar el tratamiento posterior. Con el objeto de separar zonas homogéneas

de la imagen, se opera a partir de una imagen gradiente tal como se muestra en la Figura 1.6. De esta manera se espera que los bordes de alto valor de gradiente se correspondan siempre con líneas Watershed.

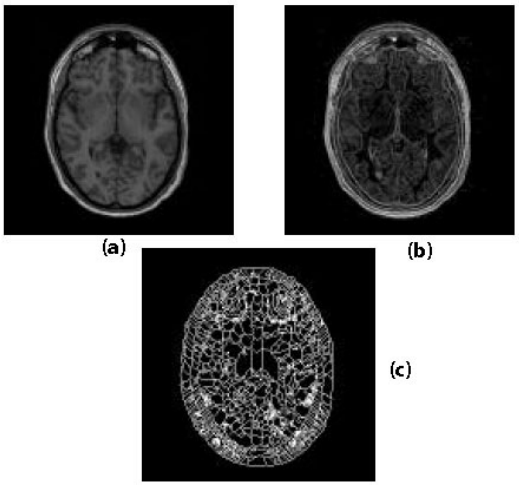


Figura 1.6: (a) MRI cerebral y su (b) imagen gradiente. (c) Aplicando la transformada de Watershed.

Una desventaja de este método es que debido al gran número de mínimos locales que presenta una imagen digital ruidosa, cada uno de ellos asociado a una cuenca Watershed, este operador morfológico produce una gran sobresegmentación en pequeñas regiones cuando se aplica a una imagen sin preprocesar. En una MRI típica puede haber del orden de decenas de miles de mínimos locales. Con un filtrado anisotrópico se puede reducir notablemente la sobresegmentación, a la vez que se respeta la localización de los bordes significativos de la imagen. Aún así, es necesario recurrir a algoritmos de unión de regiones basados en la reducción iterativa de una estructura RAG (*Region Adjacent Graph* - Grafo de Región Adyacente).

1.4.5. Graph Cuts

Muchos de los problemas que surgen en computación gráfica pueden ser expresados en términos de minimización de energía. En los últimos años, los algoritmos de corte mínimo sobre redes de flujo han surgido como una herramienta para una exacta o aproximada minimización de energía [13]. La técnica básica consiste en construir un grafo especializado para la función de energía a ser minimizada de tal manera que el corte mínimo sobre el grafo también minimice la energía. El corte mínimo, a su vez, puede ser computado muy eficientemente por algoritmos de máximo flujo. A continuación explicaremos con más detalle el proceso de segmentación basado en GraphCut, mostrando algunos conceptos asociados al mismo.

Problema de etiquetado de píxeles

El uso de la minimización de energía es para resolver el problema del etiquetado de píxeles, el cual en sí mismo es una generalización de algunos problemas como restauración de imágenes, segmentación, etc. En dicho problema, las variables representan píxeles individuales y los posibles valores para una variable individual (e.g. las intensidades). La idea es que dada como entrada un conjunto de píxeles P y un conjunto de etiquetas L, la meta es encontrar un conjunto de etiquetado $f(P) \longrightarrow L$. La Figura 1.7 ofrece una ilustración del problema.

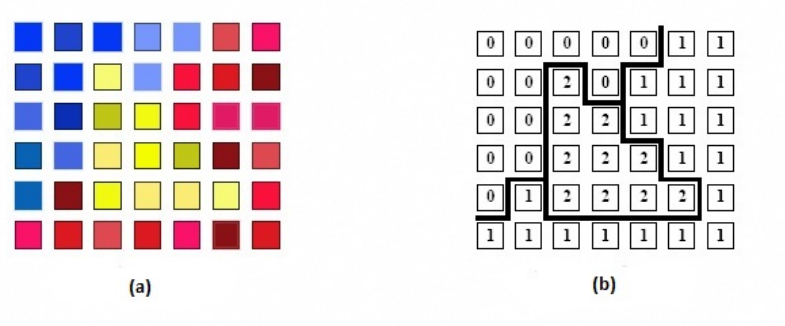


Figura 1.7: Ejemplo de etiquetado de una imagen: en (a) un conjunto de píxeles y en (b) un conjunto de píxeles formados a partir de las intensidades similares de los mismos.

Usualmente, ciertas limitaciones visuales son usadas para limitar dichas etiquetas. En el procesamiento de las imágenes, estas etiquetas se obtienen a partir de varias pruebas con las imágenes, dichas limitaciones son reflejadas al definir el problema de etiquetado en términos de minimización de alguna función de energía. Una forma estándar de la función de energía tomada de [14] es:

$$E(x) = \sum_{p \in P} D_p(f_p) + \sum_{p,q \in N} V_{p,q}(f_p, f_q) \tag{1.5}$$

Donde $N \subset P \times P$ es un sistema de píxeles vecinos, $D_p(f_p)$ es una función derivada de los datos observados que mide el costo de asignar la etiqueta f_p al píxel p. La función $V_{p,q}(f_p, f_q)$ mide el costo de asignar las etiquetas f_p, f_q a los píxeles adyacentes, p, q. Una posible solución al problema de etiquetado mencionado, es emplear los Campos Aleatorios de Markov como se propone en [15].

Campos Aleatorios de Markov (MRF - Markov Random Fields)

Los campos aleatorios de Markov son un modelo generativo usado en procesamiento de imágenes y computación gráfica para resolver problemas de etiquetado. Esta sección introduce la teoría de campos aleatorios de Markov (*MRF*), la cual es comúnmente utilizada para modelar el problema de optimización mostrado anteriormente. Un *MRF* consta de 3 conjuntos; un conjunto S de lugares, un sistema de vecinos N y un conjunto de variables aleatorias F. El sistema de vecinos $N = \{N_i | i \in S\}$ donde cada N_i es un subconjunto de lugares de S los cuales forma el vecindario del lugar i. El campo aleatorio $F = \{F_i | i \in S\}$ consiste de variables aleatorias F_i que toman un valor f_i de un conjunto de etiquetas $L = \{l_1, l_2...\}$. Un conjunto particular de etiquetas, denotado por f es llamado configuración de F. La probabilidad de una configuración particular f,$P(F = f)$ debe satisfacer la propiedad de Markov para que F sea un campo aleatorio de Markov.

$$P(f_i | f_{S-i}) = P(f_i | f_{N-i}), \forall i \in S.$$

Esto significa que el estado de cada variable aleatoria depende solamente del estado de sus vecinos. El hecho que una etiqueta dependa de las etiquetas de sus vecinos nos permite modelar el problema como un *MRF*. Desde una perspectiva probabilística, para estimar la configuración f basándose en los datos observados D que maximiza la función de probabilidad $P(D|f)$. Al emplear el teorema de Bayes, esta función de probabilidad puede ser expresada como una función de energía $E(f)$ y el máximo a posteriori estimado de f puede maximizar dicha función de energía.

Etiquetado de píxeles como un problema de grafos

Greig et al. [16] descubrieron que los algoritmos de corte mínimo pueden ser usados para minimizar ciertas funciones de energía. En base a esto es posible construir un grafo de flujo a partir de una imagen con el objetivo de segmentarla. En dicho grafo se considera el etiquetado de píxeles, utilizando cada píxel de la imagen como un vértice de dicho grafo. Un grafo $G =< V, E >$ es definido como un conjunto de vértices V y un conjunto de aristas dirigidas E las cuales conectan a dichos nodos. En cada arista $\varepsilon \in E$ en el grafo se le asigna un peso no negativo (costo) w_ε. Cada vértice del grafo es conectado con sus 8-vecinos (para el caso de imágenes en 2D) a través de arcos no dirigidos llamados ***N-links***. Así el peso de cada arco corresponde a un costo por discontinuidad entre los píxeles. Adicionalmente se agregan dos vértices especiales llamados la fuente S y el destino T. La fuente representa el objeto a segmentar en la imagen (*foreground*), y el destino representa el fondo de la imagen (*backgroung*). Todos los vértices del grafo están conectados por un arco con la fuente y a través de otro con el destino, dichos arcos

se denominan ***T-links***, el valor de dichos arcos corresponde al costo para etiquetar el vértice como parte de la fuente ó el destino.

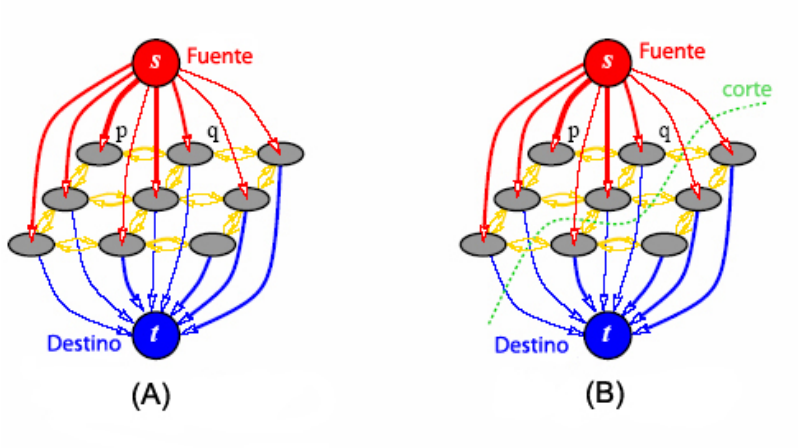


Figura 1.8: Ejemplo del etiquetado de píxeles como un problema de grafos donde se observa al grafo de flujo (A) y al mismo grafo con su corte mínimo (B)

A partir del grafo de flujo, es posible calcular el corte mínimo (*min-cut*) [17], que divide el grafo en dos grafos no conectados como se muestra en la Figura 1.8(B) uno de los cuales contiene al vértice fuente y el otro grafo contiene al vértice destino. El resto de los vértices se encuentran conectados a uno de estos dos grafos. Generalmente, los vértices conectados a la fuente representan el objeto de interés de la imagen, y los vértices conectados al destino representan el fondo de la imagen.

A continuación presentaremos algunas definiciones tomadas de [17], con el objetivo de mostrar de una manera formal los conceptos utilizados para la obtención de un corte mínimo en un grafo.

Red de flujo

Es un grafo dirigido con pesos en sus aristas los cuales son de valor positivo (*los cuales hacen referencia a capacidades*). Una red de flujo-st (*st-flow network*) es definido como una red de flujo que tiene dos vértices identificados, una fuente s y un destino t.

Definiremos al flujo total entrante de un vértice (*la suma de todos los flujos de las aristas dirigidas hacia el vértice*) como *inflow*, al flujo total saliente del vértice (*la suma de todos los flujos sobre las aristas que salen del vértice*) como *outflow*, y a la diferencia entre los dos (*inflow menos outflow*) como el *netflow* del vértice.

st-flow

Un flujo de s hacia t en un *st-flow network* es un conjunto de valores no negativos asociados a cada vértice. Se puede decir que un flujo es factible si satisface la condición que un flujo en una arista no puede ser mayor que la capacidad de la misma y la condición de equilibrio local en la cual el *netflow* de cada vértice es 0 (*excepto para s y t*).

Flujo máximo

Es posible definir un flujo máximo dentro de un *st-flow network* como el flujo-st tal que ningún otro flujo de s hacia t sea mayor.

Corte Mínimo

Un corte C de flujo sobre un grafo con dos terminales es una partición de los nodos del grafo en dos sub-conjuntos disjuntos de S y T donde la fuente s esta en S y el destino t esta en T. Un corte mínimo de una red de flujo, es un corte cuya capacidad es la menor sobre todos los cortes $s-t$ de la red.

Correspondencia entre Flujo Máximo/Mínimo Corte

Uno de los resultados fundamentales en optimización combinatorial puede ser resuelto encontrando el máximo flujo desde la fuente S hacia el destino T [13]. De una manera más simple, se podría ver el máximo flujo como la máxima *"cantidad de agua"* que puede ser enviada desde la fuente hacia el destino interpretando las aristas del grafo como *"tuberías"* con capacidades igual al peso de las aristas. El teorema de Ford - Fulkerson (ver [18]) declara que un flujo máximo desde una fuente S hacia un destino T satura un conjunto de aristas en el grafo dividiendo los nodos en dos conjuntos disjuntos $\{S,T\}$, correspondiéndose con un corte mínimo. En conclusión, se puede decir que los problemas de corte mínimo y el máximo flujo son equivalentes

Teorema 1: El teorema del flujo máximo - mínimo corte. Si f es un flujo en una red de flujo $G=(V,E)$ con una fuente S y un destino T, entonces el valor del máximo flujo es igual a la capacidad de un mínimo corte [18].

Algoritmos de Flujo Máximo (Max-Flow)

Una de las técnicas más conocidas para hallar el flujo máximo en un grafo es el algoritmo de Ford -Fulkerson [18] el cual será descrito a continuación.

Algoritmo de Ford - Fulkerson: Esta técnica permite mejorar cualquier flujo existente previamente exceptuando el máximo. El algoritmo comienza con un flujo de 0, y luego se le aplica el método iterativamente para ir aumentando el flujo, hasta que no se pueda aplicar más el método, en ese momento se considera que se consiguió el flujo máximo. Para la aplicación de dicha técnica es necesario el conocimiento de dos conceptos básicos: grafo residual y *augmenting path*.

Considerando un grafo de flujo, el grafo residual tiene los mismos vértices que el grafo original; y uno o dos arcos por cada arco existente en el original, específicamente, si se tiene un arco X-Y, que va de un nodo X a un nodo Y, y se envía flujo a través de este arco, si la cantidad de flujo enviada es menor a la capacidad, entonces se tiene un arco X-Y con una capacidad igual a la diferencia entre la capacidad y el flujo (esto es llamado capacidad residual). Si el flujo es positivo entonces se crea un arco reverso Y-X con una capacidad igual al flujo enviado en X-Y. Si el arco está saturado, entonces el arco Y-X tendría capacidad igual a la capacidad del arco original X-Y. Un *augmenting path* es un camino de flujo desde la fuente hasta el destino en el grafo residual.

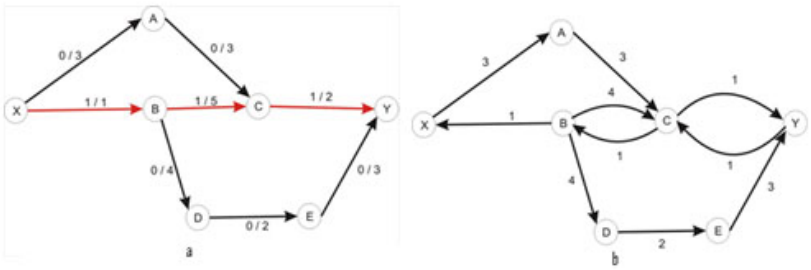


Figura 1.9: Grafo de flujo y su grafo residual

En la Figura 1.9 se observa un ejemplo de grafo residual. En la Figura 1.9(A) se tiene un grafo con flujo de la fuente X al destino Y a través de los arcos X-B-C-Y; y en la Figura 1.9(B) tenemos el grafo residual, donde el arco X-B al quedar saturado se elimina y se crea el arco B-X con igual peso al original, adicionalmente se crea el arco C-B y el arco Y-C con capacidad igual al flujo enviado. Este proceso es considerado una iteración del algoritmo de Ford-Fulkerson, ahora en cada nueva iteración se consigue un nuevo *augmenting path*, hasta que no sea posible conseguir ningún camino de la fuente al destino. Se puede observar en la Figura 1.9, que es posible enviar un flujo de valor 1 a través del camino X-A-C-Y al saturar el arco C-Y. De esta forma se obtiene el grafo

residual como se observa en la Figura 1.10.

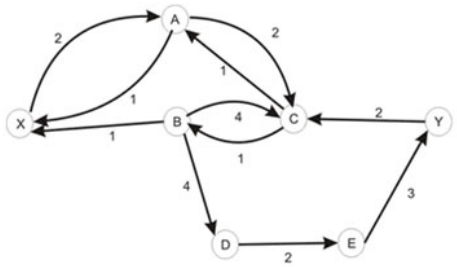


Figura 1.10: Grafo residual de la Figura 14 luego de enviar flujo a través de X-A-C-Y

Siguiendo con el ejemplo, mostrado en la Figura 1.10 es posible aumentar el flujo a través del camino X-A-C-B-D-E en 1, lo que dejaría el grafo residual de la Figura 1.11.

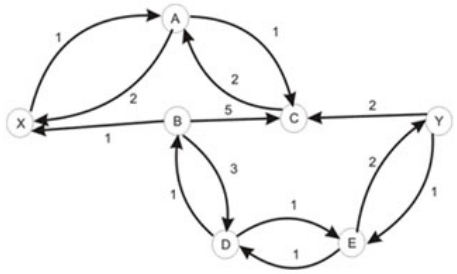


Figura 1.11: Grafo residual luego de mandar flujo a través de X-A-C-B-D-E, de la Figura 1.10

Luego de esta iteración ya no existe ningún *augmenting path* en el grafo residual, por lo que el algoritmo ha finalizado, dejando el grafo de flujo de la Figura 1.12, donde el corte mínimo del grafo se consigue eliminando los arcos saturados X-B y C-Y, y el costo del Max-flow (equivalente al corte mínimo) es la suma del peso de los arcos saturados, que da un valor de 3 en el ejemplo.

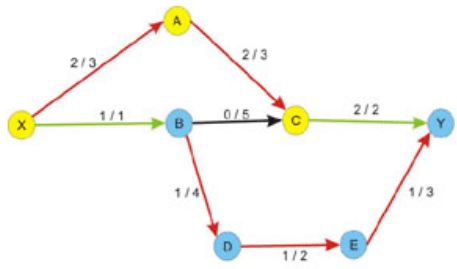


Figura 1.12: Grafo de flujo final

La forma como se consigue el camino de la fuente al destino puede variar la rapidez con la que se ejecuta el algoritmo.

Capítulo 2

GrabCut en el GPU

En este capítulo se presenta el algoritmo GrabCut enfocado en la segmentación de volúmenescreado en este trabajo de grado. Antes de ellos, se presenta una introducción a la arquitectura CUDA y sus componentes, la cual será utilizada para paralelizar el algoritmo mencionado.

2.1. Arquitectura CUDA

Compute Unified Device Architecture (CUDA) creado por NVidia en noviembre de 2006, es una arquitectura de cómputo paralelo de propósito general que permite incrementar el rendimiento de la GPU así como también para procesamiento de propósito general. Con las herramientas y arquitectura de CUDA los desarrolladores están logrando grandes avances en diversos campos como imágenes médicas, exploración de recursos naturales, creación de aplicaciones como reconocimiento de imágenes en tiempo real, así como la reproducción de video en alta definición. Estos avances se logran a través de APIs estándares tales como, OpenCL, DirectX Compute, C/C++, Fortran, Java, Phyton y .NET de Microsoft. En la Figura 2.1 se puede observar un ejemplo de la interacción de CUDA con las aplicaciones.

La GPU es una arquitectura altamente paralelizable que contiene cientos de núcleos (*cores*) en los cuales se pueden ejecutar colectivamente miles de hilos concurrentes, por lo que es ideal aprovechar al máximo este nivel de cómputo simultáneo [19]. NVidia busca aprovechar el paralelismo de la GPU mediante un modelo y un ambiente de programación paralelo que explote el poder de la unidad de procesamiento gráfico para programaciones de propósito general, de manera de obtener la máxima eficacia. Por

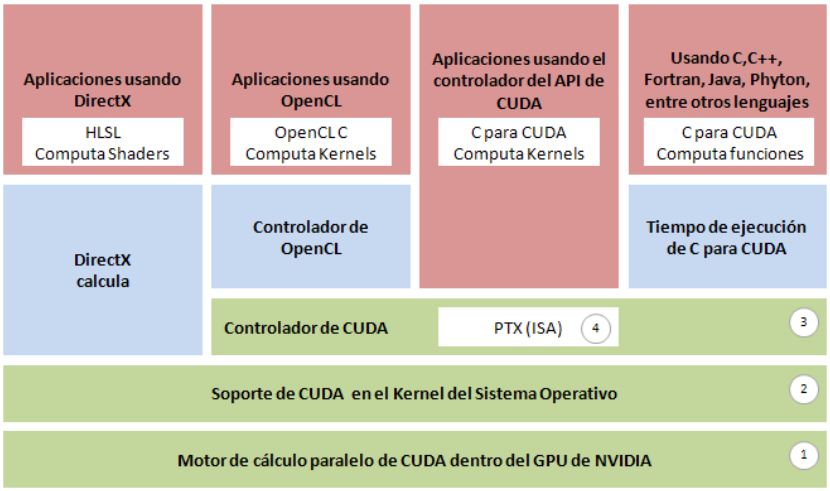


Figura 2.1: CUDA e interacciones con aplicaciones

ello es creada la arquitectura CUDA que permite a los programadores enfocarse en el desarrollo de algoritmos paralelos y no en la implementación de los mecanismos de paralelismo. Este modelo es esencialmente una extensión del lenguaje de programación C, por lo que la implementación de los algoritmos paralelos en CUDA es bastante directa, y requiere de una baja curva de aprendizaje respecto a la sintaxis y semántica del lenguaje por parte de los desarrolladores.

CUDA también está desarrollado para permitir programación heterogénea, en donde las aplicaciones usan el tanto la GPU como la CPU, siendo estos últimos tratados como dispositivos distintos con espacios de memorias privados o propios. Las porciones seriales del programa desarrollado en esta arquitectura se ejecutan en la CPU siempre y cuando no sea multicore, a la que se ha denominado *host*, mientras que las porciones paralelas del programa se ejecutan en la GPU, que ha sido llamado dispositivo (*device*), como núcleos computacionales (*kernels*). En otras palabras, un *kernel* es una función que es llamada por un *host* y es ejecutada por un *device* [19]. Sólo un *kernel* puede ser ejecutado en un *device* a la vez y cuando un *kernel* es ejecutado, son lanzados varios hilos concurrentes (hilos de CUDA) encargados de ejecutar el mismo código presentado en dicho *kernel* en paralelo.

Hay algunas distinciones que deben hacerse entre los hilos de CUDA y los hilos de CPU. Los hilos de CUDA son extremadamente ligeros en términos de creación y cambio de contexto. Miles de hilos de CUDA pueden ser creados en solo algunos ciclos de reloj y como resultado no hay sobrecarga (*overhead*) que tenga que ser amortizada durante la ejecución del *kernel*. Dado que en CUDA el intercambio de hilos es de bajo costo, las penalizaciones de tiempo asociadas a un intercambio de hilo cuando se encuentra en el estado suspendido o bloqueado son mínimas.

Cuando un *kernel* de CUDA es lanzado, se ejecuta como un arreglo de hilos paralelos,

donde cada hilo corre el mismo código, pero en regiones distintas de memoria que son asignadas a cada hilo y sobre las cuales se ejecuta alguna operación (Figura 2.2). Supóngase que se desea sumar dos arreglos A y B de 100 posiciones cada uno, en este caso, cada hilo i suma la posición $A_i + B_i$ y arroja el resultado en un arreglo C_i. En el caso que se acaba de describir, cada hilo se ejecuta independientemente, pero esto no siempre es así, por lo tanto CUDA también permite realizar operaciones de cómputo dependientes. La pieza faltante en el ejemplo anterior son los medios de cooperación entre los hilos [19]. La cooperación de los hilos es valiosa por varias razones:

1. Los hilos pueden compartir resultado para evitar computaciones redundantes, de forma similar que sucede en la memorización que emplea la programación dinámica.

2. Los hilos pueden compartir el acceso a memoria, lo cual puede reducir los requerimientos de ancho de banda.

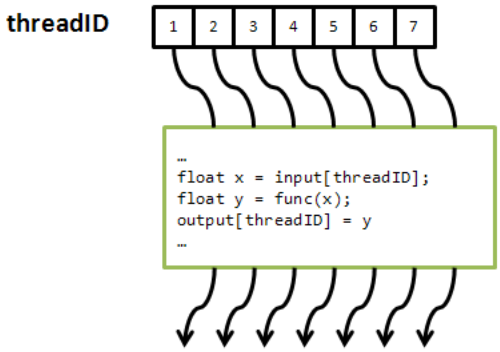


Figura 2.2: Hilos de un kernel [19]

Aún cuando la cooperación es deseable, la colaboración sin límites entre todos los hilos de un *kernel* puede influenciar de forma negativa el rendimiento, siendo a su vez no escalable. Sin embargo, reduciendo la cooperación de los hilos en pequeños grupos, la arquitectura alcanza cooperación y escalabilidad al mismo tiempo. Cuando un *kernel* es lanzado, los hilos son arreglados en una rejilla de bloques de hilos denominado *grid*. Los hilos en el mismo bloque cooperan mediante memoria compartida y sincronización mediante barreras, mientras que hilos en bloques distintos no pueden cooperar. Esta ordenación permite a los programas escalar de forma transparente entre distintos modelos de GPU, de tal forma que CUDA distribuye los bloques de un *kernel* de forma apropiada a los multiprocesadores de la GPU, sin importar el número de bloques o multiprocesadores.

2.1.1. Jerarquía de memoria

Una GPU posee N multiprocesadores según su especificación. Cada multiprocesador del *device* (GPU) contiene generalmente 8 procesadores de hilos, de manera que 8N procesadores de hilos ejecutan los hilos de *kernel*[1]. Cada hilo de un multiprocesador posee un pequeño espacio de memoria denominado registros y cada multiprocesador posee una memoria compartida para habilitar la cooperación entre los hilos. Además de los espacios de memoria de cada multiprocesador dentro del *device*, éste también posee un espacio de memoria (DRAM) el cual se encuentra dividido en memoria local y global [19]. Así cada *kernel* posee diferentes accesos de memoria en diferentes niveles, que serán descritos brevemente a continuación:

- *Por hilo*: Puede acceder a los registros dentro del chip del multiprocesador y a la memoria local no cacheada fuera de dicho chip.
- *Por bloque*: Cada bloque puede acceder rápidamente a la pequeña memoria compartida dentro del chip.
- *Por dispositivo(device)*: Cada *kernel* tiene acceso a la memoria global fuera del chip que no es cacheada. Esta memoria global es consistente en el tiempo y permite operaciones de entrada y salida entre *kernels*.

La CPU y la GPU se comunican a través de la memoria global, la cual puede ser accedida por la CPU para lectura y escritura, tal como se muestra en el diagrama de la Figura 2.3.

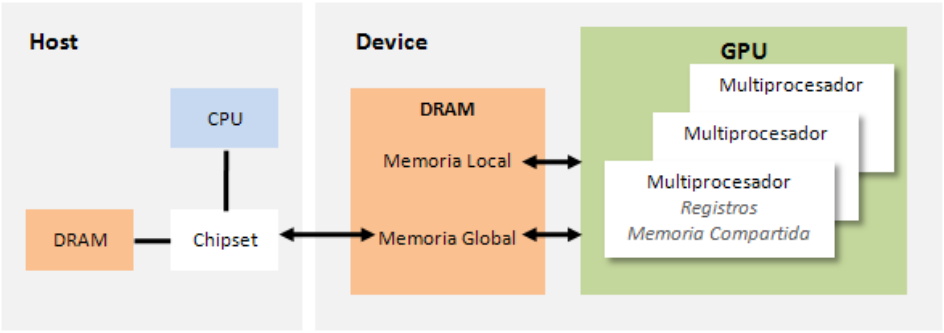


Figura 2.3: Esquema de comunicación entre un Host y un Device a través de la memoria [19]

[1]Al hablar de hilos de kernels en este contexto se refiere a los hilos de *kernel* de CUDA y no del sistema operativo.

2.1.2. Comunicación entre la CPU y la GPU

En la Figura 2.3 se observa como la CPU y la GPU tienen espacios de memoria separados y a su vez se observa que tanto la GPU como la CPU pueden acceder al espacio de memoria global DRAM del *device*, lo que permite a la CPU manejar la memoria del *device* para acceder a datos, crear o liberar espacios de la misma y escribir datos en ella.

La comunicación entre la CPU y la GPU es lo que permite la definición de los kernels desde un programa principal, debido a que los argumentos de la funciones de kernels son copiadas directamente desde el *host* al *device*, siempre y cuando los parámetros sean datos simples, como *int*, *char*, entre otros. Si se desea pasar un arreglo o tipos de datos no elementales, debe emplearse la instrucción de acceso `cudaMemCpy`.

En CUDA se manejan tipos de clasificadores de funciones que especifican si una función es ejecutada o es llamada por el *host* o por el *device*. De la misma forma, las variables declaradas en memoria tienen asociado un tipo de clasificador que especifica el tipo de memoria en la cual se encuentran. Cabe acotar que también existen variables desclasificadas, tales como los escalares y tipos de vectores que son almacenados en los registros, y en caso de no poder ser alojadas en el registro son establecidas en la memoria local del *device*.

La arquitectura de CUDA extiende C al permitir al programador definir funciones en dicho lenguaje denominadas "*kernels*", que cuando son llamadas son ejecutadas N veces en paralelo por N hilos de CUDA diferentes mediante la cooperación y sincronización entre los hilos, en lugar de ejecutarlo una sola vez como regularmente lo hace C. También se conoce el manejo de memoria en este modelo y cómo los *kernels* intervienen en este manejo de memoria.

2.1.3. Ejecución

El *software* de desarrollo de CUDA comprende un conjunto de bibliotecas optimizadas, encontrándose entre las más importantes: math.h, FFT (*Fast Fourier Transform*) y BLAS (*Basic Linear Algebra Subprograms*). Además proporciona un código fuente C que integra la CPU con la GPU, un compilador C de NVidia, un lenguaje intermedio denominado PTX, los drivers de CUDA, un depurador [20] y demás componentes que pueden visualizarse en la Figura 2.4.

CUDA compila una aplicación C/C++ de CUDA enviándola al NVCC (compilador C de NVidia), en donde se generan dos códigos objetos, uno para la CPU y otro para la GPU. El código que va a la GPU es pasado previamente por el lenguaje intermedio PTX independiente de la plataforma, el cual transforma el código al vuelo al momento

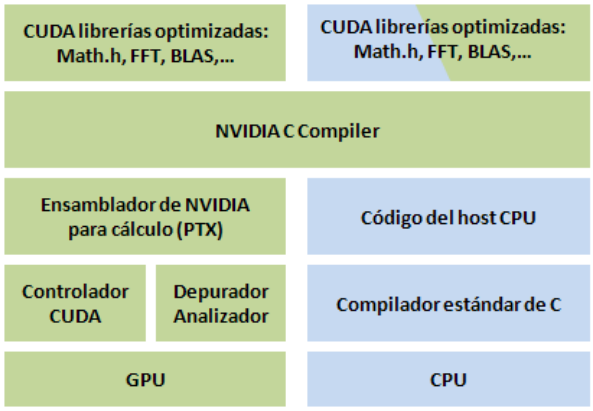


Figura 2.4: *Software* de desarrollo de CUDA[20]

de la ejecución, a un binario que pueda ser ejecutado por la tarjeta gráfica sin importar cual sea esta (Figura 2.5).

El modelo de ejecución de CUDA se resume en los siguientes ítems:

- Los hilos son procesados por un procesador de hilos.
- Los bloques de hilos son ejecutados en multiprocesadores.
- Los bloques de hilos no migran.
- La cantidad bloques de hilos concurrentes que residen en un multiprocesador están limitados por la memoria compartida y los registros de cada hilo.
- Un *kernel* es lanzado como un *grid* de bloques de hilos.
- Solo un *kernel* puede ejecutarse sobre un dispositivo a la vez.

Todos los kernels convocados por el *host* son asíncronos, por lo que el control retorna a la CPU inmediatamente, pero los *kernels* se ejecutan después que todas las llamadas previas a CUDA han sido completadas. En el caso de la sincronización del *device*, los bloques de un *kernel* pueden cooperar entre ellos mediante una memoria compartida y sincronizando su ejecución para coordinar los accesos a memoria.
Un vez presentada la introducción a CUDA, explicaremos GrabCut enfocado a la segmentación de volúmenes.

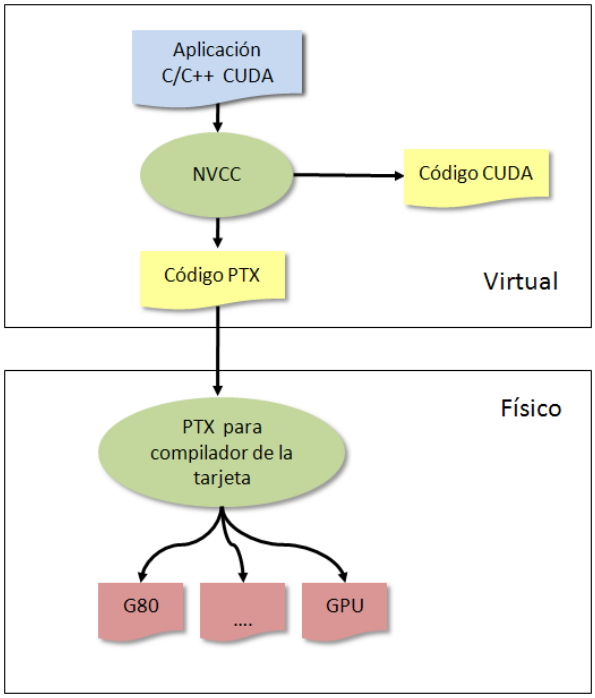


Figura 2.5: Esquema de compilación de CUDA[20]

2.2. GrabCut

Esta técnica fue presentada por Vladimir Kolmogorov et al.[1] en el año 2004 y ha sido modificada en la presente investigación extrapolando su campo de uso, orientándola a la segmentación de volúmenes, acelerando el procesamiento con la utilización de la arquitectura CUDA.

Este método de segmentación requiere poca intervención del usuario. En el trabajo de Kolmogorov se selecciona un recuadro alrededor del objeto a segmentar (para imágenes 2D), en nuestro caso se va a seleccionar un cubo (bounding box) que contenga el objeto a segmentar, luego se realiza la segmentación de la imagen de manera automática.

GrabCut utiliza el mismo enfoque que GraphCut el cual fue explicado en el capítulo anterior, en la cual se definieron muchos términos que también serán utilizados en esta técnica para la segmentación de imágenes. La idea es crear un grafo de flujo de redes a partir del volumen a segmentar donde por cada vóxel del volumen se genera un vértice que lo representa en el grafo, y luego se conecta cada vértice con 6 de sus 26 vecinos a través de arcos dirigidos (***N-link***). Siguiendo este enfoque se tienen dos vértices más, la fuente y el destino del grafo de flujo. La fuente estará conectada con el sub-volumen que representa el objeto a segmentar (*foreground* el cual está conectado al nodo S), y el

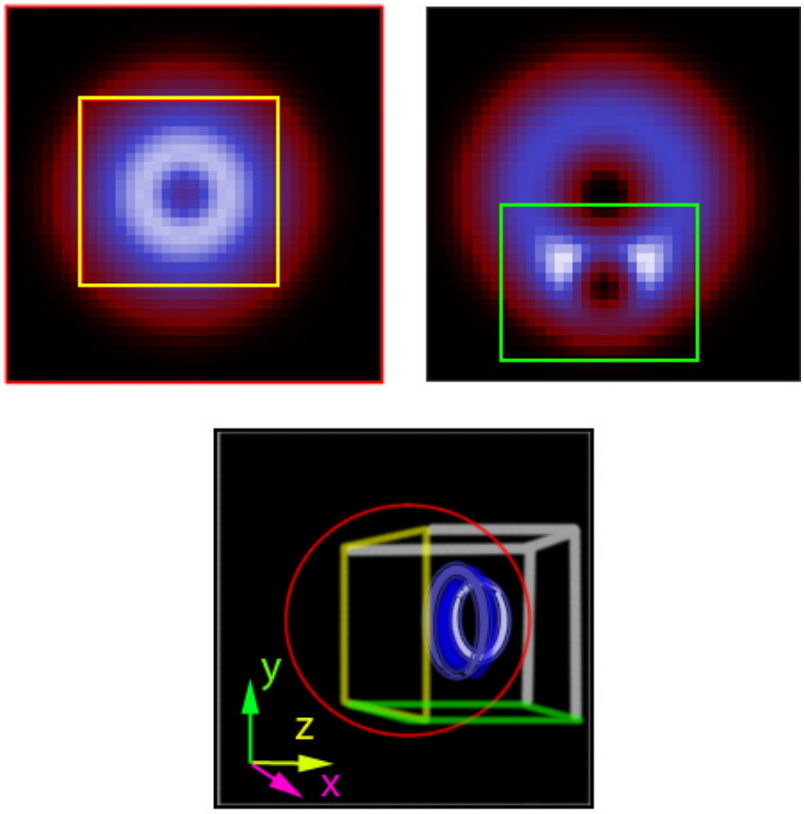


Figura 2.6: Selección del volumen a segmentar.

destino estará conectado con el sub-volumen que representa el fondo (*background* el cual está conectado al nodo T), es decir, aquellos vóxeles que no pertenecen al objeto que se desea segmentar. Cada uno de los vértices se conecta a través de un arco con la fuente y a través de otro con el destino (***T-links***). Para el cálculo del peso de estos arcos se utilizan modelos mixtos gaussianos (*Gaussian Mixture Models* - GMM) uno para el *foreground* y otro para el *background*. Cada GMM está dividido en k componentes Gaussianos (En el trabajo de Kolmogorov [1] $k = 5$).

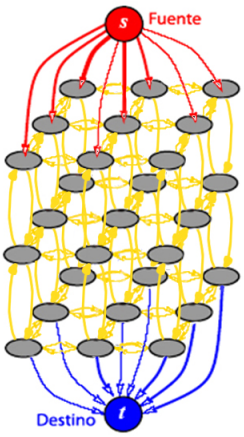


Figura 2.7: Grafo a partir de un volumen

A dicho grafo de flujo se le aplica un algoritmo de corte mínimo (*min-cut*), la técnica de GrabCut utiliza el algoritmo de Ford-Fulkerson [18] para hallar en corte mínimo de un grafo, para nuestro caso de estudio desarrollaremos un método llamado *Push-Relabel* el cual puede ser paralelizado. De esta manera el grafo puede ser separado en dos grafos disjuntos, uno conectado a la fuente que representa al objeto de interés y el otro grafo conectado al destino que representa el sub-volumen de fondo.

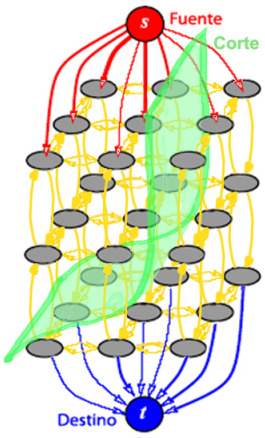


Figura 2.8: Corte del grafo que representa el volumen.

Inicialmente se tiene un arreglo, $f = \{f_1, f_2, .., f_k, .., f_N\}$, donde f_k representa un vóxel del volumen en el espacio de color RGB y N el número de vóxeles del volumen.

Igualmente, se tiene un arreglo $A = \{\alpha_1, \alpha_2, .., \alpha_k, .., \alpha_N\}$ donde cada α_k representa el *matte* del vóxel f_k. El *matte* es un valor de intensidad que, en el caso de GrabCut

puede tomar dos posibles valores, *foregorund* si el vóxel pertenece al objeto y *backgorund* si pertenece al sub-volumen de fondo. Este valor varía a lo largo de la ejecución del algoritmo, y se toma para construir el resultado final. A su vez esta marca indica si el vóxel pertenece al GMM *foreground* o al GMM *background*.

Por cada vóxel se almacena el número del componente gaussiano al que pertenece. Este número en conjunto con el *matte* permite conocer a cual componente de los 10 creados (5 para el GMM *foreground* y 5 para el GMM *background*) pertenece el vóxel.

Finalmente, se utilizan 3 marcas temporales para los vóxeles llamadas *trimap*. Dichas marcas pueden ser de un vóxel perteneciente al sub-volumen de fondo(*background*), vóxel de objeto(*foreground*) o vóxel desconocido (*unknown*), y son colocadas por el usuario, y no varían a través de la ejecución del algoritmo a menos que este las modifique intencionalmente.

Todos los vóxeles marcados como *trimap foreground* siempre van a estar marcados como *matte foreground*. Igualmente los vóxeles seleccionados como *trimap background* siempre están definidos como *matte background*; por lo tanto, los vóxeles que modificarán su *matte* en la ejecución del algoritmo serán los que fueron seleccionados inicialmente como *trimap unknown*.

A continuación se describe paso a paso el algoritmo a ejecutar y que se muestran en la Figura 2.9:

1. El usuario sitúa un cubo(*bounding box*) que contenga el objeto a segmentar, de forma que los vóxeles afuera del cubo son marcados como *trimap background*, y los vóxeles dentro del cubo son seleccionados como *trimap unknown*.

2. El *matte* de los vóxeles es inicializado de la siguiente manera: los vóxeles pertenecientes a *trimap background* son en principio *matte background*, mientras que los vóxeles con valor *trimap unknown* se le asigna el valor de *matte foreground*.

3. Se crean dos GMMs, uno para el *foreground* y otro para el *background*. Cada uno con 5 componentes gaussianos. El GMM *foreground* se inicializa con los vóxeles de valor *matte foreground*, mientras que el el GMM *background* se inicia con los vóxeles de valor *matte background*.

4. Cada vóxel en el GMM *foreground* es asignado al componente donde es más probable que pertenezca, de igual manera cada vóxel del GMM *background* se le asigna el componente que es mas probable que pertenezca.

5. Los GMM son eliminados, y se crean unos nuevos a partir de la información obtenida anteriormente. Cada vóxel es asignado a su GMM respectivo (GMM *foreground* o GMM *background*) y al componente de ese GMM que le fue asignado en el paso 4.

6. Se construye el grafo y se consigue el *min-cut*. En base al resultado obtenido del algoritmo de *min-cut* se modifica el valor *matte* de algunos vóxeles. Aquellos que pertenezcan conectados a la fuente quedan como *matte foreground* y los conectados al destino como *matte background*.

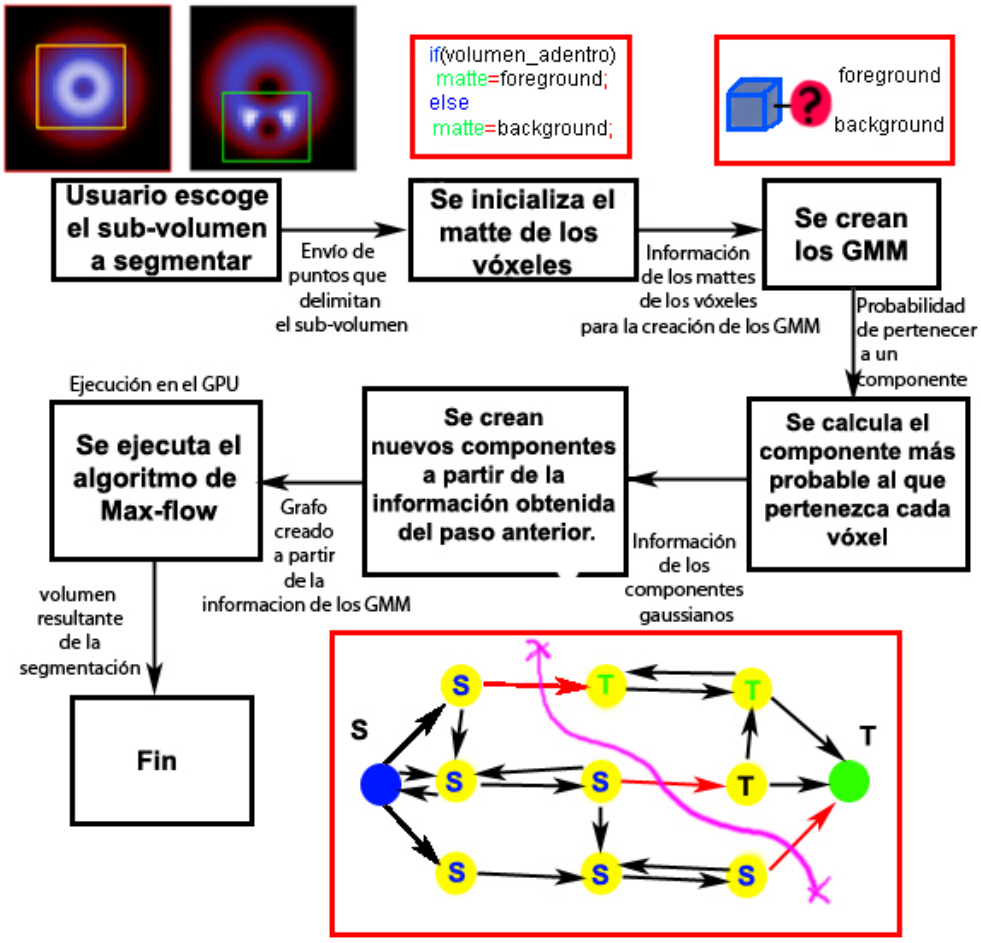


Figura 2.9: Flujo de ejecución del 3D-GrabCut

2.2.1. Calculo de los N-Links

Para el cálculo del N-Link entre un vóxel m y un vóxel n se utiliza la siguiente fórmula extraída de [21].

$$N(m,n) = \frac{50}{dist(m,n)} e^{-B\|m-n\|^2}$$

Donde $dist(m,n)$ representa la distancia entre dos puntos en nuestro caso la distancia siempre es igual a 1. $\|m-n\|$ constituye la distancia euclidiana en el espacio de color, y se calcula de la siguiente manera:

$$\|m-n\| = (R_m - R_n)^2 + (G_m - G_n)^2 + (B_m - B_n)^2$$

Donde R_m,G_m,B_m representan los valores de los canales de color rojo, verde y azul respectivamente, en el vóxel m.

B es una constante que asegura que existan diferentes valores entre cambios de contraste alto y contraste bajo, y se calcula por medio de la fórmula:

$$B = \frac{1}{\frac{2}{P} \times \sum_{m=0}^{P} \sum_{n=0}^{V} \|m-n\|^2}$$

Donde P simboliza el número de vóxeles del volumen, V es el número de vecinos del vóxel. Debido a que se utilizarán solo 6 vecinos de cada vóxel cada uno en dirección de un eje de coordenadas ($\longrightarrow X+, \longleftarrow X-, \uparrow Y+, \downarrow Y-, \swarrow Z+, \nearrow Z-$), V es siempre igual a 6, excepto en los bordes del volumen. El objetivo de la fórmula $N(m,n)$ para el cálculo de los *N-Links* es tener valores grandes para los vóxeles contiguos con colores similares, y valores pequeños para vóxeles con colores muy diferentes (posible borde del objeto a segmentar). Esto a fin de que sea más probable que el algoritmo de *min-cut* realice un corte por los *N-Links* que conectan vóxeles con colores diferentes.

2.2.2. Gaussian Mixture Models

En estadística cuando se grafica un conjunto de valores, siempre se trata de comparar la gráfica generada con alguna distribución conocida pero en muchos casos ésta no es comparable con ninguna distribución, y calcular la función de densidad de probabilidad que genera dicha gráfica (vea [22] para información acerca de la Gaussiana, modelos mixtos y función de densidad de probabilidad). En estos casos se utilizan los modelos mixtos, donde la gráfica inicial es dividida en dos o más componentes, donde cada uno se asemeje a una función de densidad de probabilidad conocida. Luego, la probabilidad se calcula como la suma de las probabilidades de las funciones de densidad de cada uno de los componentes gaussianos.

En el caso de los *Gaussian Mixture Models*, la función de probabilidad inicial es dividida en componentes Gaussianos (vea [23]); concretamente para el algoritmo de

GrabCut se utilizan 5 componentes. Debido a que se está trabajando con volúmenes, los mismos se encuentran en escala de grises para lo cual utilizaremos la información proveniente de la función de transferencia para obtener un color correspondiente para cada vóxel, los cuales por medio de este método poseerán 3 componentes (R, G, B), las gaussianas generadas son multivariadas, es necesario calcular diversos elementos para las mismas, como son la matriz de covarianza, la inversa de la matriz y el determinante.

Para el cálculo y la división de los componentes es necesario calcular los autovalores y autovectores de la matriz de covarianza (vea [22] para información acerca la matriz de covarianza y autovalores y autovectores en esta matriz).

La creación e inicialización de los GMM (paso 3 del algoritmo) y sus componentes se realiza de la siguiente manera. Primero se tiene un sólo componente dentro del GMM donde se agregan todos los vóxeles pertenecientes a ese GMM; por ejemplo, todos los vóxeles con valor *matte foreground* se agregan al GMM *foreground*. Después se calcula la media, el peso del componente, la matriz de covarianza, la inversa de la matriz, el determinante, los autovalores y autovectores del componente.

Para el cálculo de las variables de cada componente se debe tener en cuenta que cada vóxel es un vector de tres componentes donde el primero representa la intensidad de rojo, el segundo componente la intensidad de verde y el tercero la intensidad de azul; lo que se denota de la siguiente manera para un vóxel m:

$$m = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Donde R, G y B representa las intensidades de rojo, verde y azul respectivamente del vóxel m. Por ejemplo un vóxel m con la máxima intensidad de color rojo con 1 byte por intensidad se representa de la siguiente manera:

$$m = \begin{bmatrix} 255 \\ 0 \\ 0 \end{bmatrix}$$

La media de un componente del GMM se calcula sumando los vóxeles agregados al mismo (suma de vectores) y luego dividiendo el resultado entre el número de vóxeles agregados (multiplicación escalar - vector). Por ejemplo, si se agregan 3 vóxeles m, n y o a un componente, con valores de intensidad descritos a continuación:

$$m = \begin{bmatrix} 240 \\ 5 \\ 41 \end{bmatrix}, n = \begin{bmatrix} 100 \\ 65 \\ 47 \end{bmatrix}, o = \begin{bmatrix} 54 \\ 125 \\ 210 \end{bmatrix}$$

El valor de la media es el siguiente:

$$V = \frac{1}{3}\left(\begin{bmatrix} 240 \\ 5 \\ 41 \end{bmatrix} + \begin{bmatrix} 100 \\ 65 \\ 47 \end{bmatrix} + \begin{bmatrix} 54 \\ 125 \\ 210 \end{bmatrix}\right)$$

$$V = \frac{1}{3}\begin{bmatrix} 394 \\ 195 \\ 298 \end{bmatrix} = \begin{bmatrix} 131 \\ 65 \\ 99 \end{bmatrix}$$

La covarianza $Cov(X,Y)$ entre dos variables X,Y es una medida que permite estudiar la relación entre estas dos variables cuantitativas. La matriz de covarianza permite almacenar la covarianza de todas las posibles combinaciones de un conjunto de variables aleatorias. En el caso de GrabCut se tiene 3 variables aleatorias, que serían las intensidades de Rojo(R), Verde(V), y Azul(A) de los vóxeles. La matriz de covarianza quedaría definida de la siguiente manera:

$$\Sigma = \begin{bmatrix} Cov(R,R) & Cov(R,V) & Cov(R,A) \\ Cov(V,R) & Cov(V,V) & Cov(V,A) \\ Cov(A,R) & Cov(A,V) & Cov(A,A) \end{bmatrix}$$

Debido a que $Cov(X,Y) = Cov(Y,X)$, la matriz de covarianza es simétrica. La covarianza de dos variables aleatorias se define de la siguiente manera:

$$Cov(X,Y) = E(XY) - E(X)E(Y)$$

Donde $E(X)$ representa la esperanza de X, que no es más que la media de X:

$$E(X) = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{2.1}$$

Donde N simboliza el número de muestras y X_i el valor de la muestra i.

En el caso de un componente gaussiano para el problema de GrabCut, N es el número de vóxeles agregado al componente. Como se observa, $E(R)$ es equivalente al primer componente del vector (media del componente Rojo), $E(G)$ el valor del segundo (Verde) y $E(B)$ el valor del tercero (Azul).

La $E(X,Y)$ viene dada por la siguiente fórmula:

$$E(X,Y) = \frac{1}{N}\sum_{i=1}^{N} X_i Y_i \tag{2.2}$$

Entonces la fórmula final de la covarianza resulta:

$$Cov(X,Y) = E(X,Y) - E(X)E(Y) = \left(\frac{1}{N}\sum_{i=1}^{N} X_i Y_i\right) - \frac{1}{N}\sum_{i=1}^{N} X_i \frac{1}{N}\sum_{i=1}^{N} Y_i \qquad (2.3)$$

Luego de obtener esta matriz, se obtiene la inversa de ella, el determinante, los autovalores y autovectores.

El siguiente paso en el algoritmo consiste en dividir este primer componente en dos, utilizando los autovectores, escogiendo el punto de división y cuáles píxeles se agregarán al nuevo componente y cuáles se quedarán en el componente original. Para el cálculo del punto P donde se va a dividir el componente se utiliza la siguiente función:

$$P = \langle v(i).\delta(i) \rangle$$

Donde $v(i)$ representa la media del componente i, y $\delta(i)$ simboliza el primer autovector del componente i. Finalmente $\langle . \rangle$ constituye la operación producto punto entre dos vectores.

Luego, por cada vóxel m se calcula el peso del mismo m_p:

$$\text{Donde: } m_p = \langle \delta(i).m \rangle$$

Donde m es un vector de 3 componentes, y estos representan los valores de rojo, verde y azul del vóxel.

Finalmente si $m_p > P$ el vóxel m es agregado al nuevo componente, si es menor, es agregado al componente que se está dividiendo.

Se calculan nuevamente la media, peso y la matriz de covarianza para los componentes. Se escoge el mayor de los autovalores de la matriz de covarianza de cada componente, se elige el componente asociado al mayor autovalor para dividirlo, y se utiliza el autovector asociado para realizar nuevamente la división como se explicó anteriormente. Este proceso es repetido hasta alcanzar el número de componentes deseados.

Para el paso 4 del algoritmo se procede de la siguiente manera. Primero por cada vóxel del volumen, se calcula cuál es el componente de su GMM al que es más probable que pertenezca. Por ejemplo: para un vóxel m con valor *matte foreground* se computa la probabilidad de que pertenezca a cada uno de los 5 componentes del GMM *foreground*, y luego se almacena el número de componentes que corresponde a la mayor probabilidad.

La probabilidad $P(m,i)$ de que un píxel m pertenezca al componente i viene dada por la siguiente fórmula [21]:

$$P(m,i) = \pi(\alpha_m, i)\frac{1}{\sqrt{det\Sigma(\alpha_m,i)}} \times exp\left(\frac{1}{2}[m - v\,(\alpha_m, i)]^t \Sigma\,(\alpha_m, i)^{-1}\,[m - v\,(\alpha_m, i)]\right)$$

Donde α_m Representa el GMM en el cual se está trabajando. En el caso de calcular $P(m,i)$ para un vóxel m con *matte foreground* entonces α_m indica que se está trabajando en el GMM *foreground*, mientras que si se está computando $P(m,i)$ para un vóxel m con *matte background* la variable α_m representa el GMM *background*.

i Representa el número del componente de GMM.

$\pi(\alpha_m, i)$ Equivale al peso del componente i en el GMM α_m. Esto se calcula dividiendo el número de vóxeles agregados al componente i entre el número de vóxeles agregados al GMM α_m.

$\Sigma(\alpha_m, i)$ simboliza la matriz de covarianza del componente i en el GMM α_m, y $det\Sigma(\alpha_m, i)$ es el determinante de esta matriz.

$v(\alpha_m, i)$ es un vector de 3 valores, cada uno representa la media de rojo, verde y azul respectivamente del componente i del GMM α_m.

m es un vector de 3 componentes con los valores R,G,B del píxel.

$m - v(\alpha_m, i)$ es una resta de vectores, lo que da como resultado otro vector de 3 componentes.

$[m - v(\alpha_m, i)]^t$ es el vector traspuesto del vector resultado de la resta.

$\Sigma(\alpha_m, i)^{-1}$ es la inversa de la matriz de covarianza del componente i en el GMM α_m.

A continuación en el paso 5 del algoritmo mostrado en la sección 2.2; se reinicializan los componentes de ambos GMMs (matriz de covarianza, media, etc.) y se asigna cada vóxel al componente de su GMM que obtuvo la mayor probabilidad de pertenecer a él.

2.2.3. Cálculo de los T-Links

Como se mencionó previamente, cada vóxel tiene un T-Link que lo conecta al *foreground* (T_{fore}) y otro que lo conecta al *background* (T_{back}).

Si el usuario ha seleccionado un vóxel m como *trimap foreground*, se asegura que el corte mínimo del grafo no desconecte este vértice del *foreground*, por lo tanto al T_{fore} de ese vértice se le asigna un valor K. Donde K representa el mayor peso posible que pueda existir en el grafo, y a T_{back} se le otorga 0. Igualmente, si el usuario ha elegido un vóxel como *trimap background*, a su valor T_{back} se le asigna K y al valor T_{fore} se le establece 0.

En caso de que el vóxel tenga valor de *trimap unknown* a los valores T_{fore} y T_{back} se le estipula $P_{fore}(m)$ y $P_{back}(m)$ respectivamente, donde $P_{fore}(m)$ es la probabilidad que el vóxel m pertenezca al GMM *foreground* y $P_{back}(m)$ la probabilidad de que éste pertenezca al GMM *background*.

La probabilidad $P(m)$ del vóxel m viene dada por la siguiente fórmula [21]:

$$P(m) = -log \sum_{i=1}^{k} [P(m, i)] \tag{2.4}$$

donde k representa el número de componentes en el GMM, 5 en el caso de GrabCut.

2.2.4. Corte mínimo de grafo (*min-cut*)

Sea un grafo $G = (V, A)$ un conjunto de nodos y un conjunto de arcos que relacionan estos nodos. A partir de este punto se asume que todos los grafos mencionados son ponderados. El corte de un grafo consiste en eliminar arcos del mismo, hasta que se consigan dos grafos disjuntos. El peso del corte de un grafo consiste en la suma de los pesos de los arcos eliminados para conseguir el corte. El corte mínimo de un grafo (*min-cut*) es el corte que tiene el peso mínimo, entre todos aquellos posibles del grafo.

Debido al teorema de *Max-flow/min-cut* que podemos conseguir en [18], el corte mínimo del grafo puede ser conseguido utilizando un algoritmo de *Max-flow*, donde los arcos que resultan saturados (arcos con peso 0) por el *Max-flow* son los arcos que se eliminan para conseguir el corte mínimo.

2.2.5. Flujo en redes (*Max-flow*)

Un grafo de flujo es aquel en el cual un nodo puede enviar flujo a través de un arco entre los dos nodos conectados a él, donde el flujo pasado por el arco no puede exceder a capacidad de éste. Adicionalmente, la cantidad de flujo que recibe un nodo es igual a la cantidad de flujo que sale de él, al menos que sea un nodo fuente, el cual solo envía flujo, o un nodo destino, el cual unicamente recibe flujo. Cuando la cantidad de flujo enviada por un arco es equivalente al peso del mismo, se considera que dicho arco está saturado.

Entre los algoritmos más utilizados están el de Ford-Fulkerson [24] y el de Goldberg y Tarjan[25]. El primero realizado por Ford y Fulkerson y modificado por Edmonds-Karp [26], para resolver el problema de *min-cut/maxflow* sobre grafos calculando repetidamente *augmenting paths* desde la fuente s hacia el destino t en el grafo donde el flujo es

llevado hasta que no se pueda conseguir ningún *augmenting path*. El segundo algoritmo, por Goldberg and Tarjan[25], envía el flujo desde s hacia t sin violar las capacidades de las aristas. Pero en lugar de examinar completamente el grafo residual para encontrar un *augmenting path*, el algoritmo de *Push-Relabel* trabaja a nivel local, observando los vecinos de cada vértice del grafo residual. En el algoritmo *Push-Relabel* existen dos operaciones básicas: La primera envía el exceso flujo de un vértice hacia uno de sus vecinos (*Push*) y la segunda reetiqueta un vértice para que esté en condiciones para mandar su exceso de flujo a un vecino (*Relabel*).

El algoritmo *Push-Relabel* ha sido paralelizado por Anderson y Setubal[27]. Después Bader y Sachdeva desarrollaron una optimización *cache-aware*[2][28]. Alizadeh y Goldberg[29] presentaron una implementación del algoritmo en paralelo en una máquina de conexión paralela masiva CM-2. Otros dos intentos de paralelizar el algoritmo en el GPU también fueron reportados en [30][31]. En el 2008 Vibhav Vineet y P.J. Narayanan implementaron el algoritmo de Graph Cuts en el GPU utilizando CUDA [32]. Siguiendo este último enfoque se desarrollará el algoritmo de *Push-Relabel* utilizando CUDA para acelerar el tiempo de obtención de resultados, que será explicado a continuación.

2.2.6. Push- Relabel

Sea $G = (V, E)$ un grafo y s, t son la fuente y el destino. El algoritmo *Push-Relabel* construye y mantiene un grafo residual todo el tiempo. El grafo residual G_f del grafo G tiene la misma topología, pero consiste de las aristas las cuales pueden admitir más flujo. La capacidad residual $c_f(u, v) = c(u, v) - f(u, v)$ es la medida del flujo adicional que puede ser enviado desde u hacia v despúes de enviar el flujo $f(v, u)$, donde $c(u, v)$ es la capacidad de la arista (u, v). El algoritmo mantiene 2 cantidades: el exceso de flujo $e(v)$ para cada vértice y la altura $h(v)$ para todos los vértices. El exceso de flujo $e(v) \geq 0$ es la diferencia entre el flujo total entrante y el flujo total saliente para un nodo v a través de sus aristas. La altura $h(v)$, es un estimado conservativo de la distancia del vértice v desde el destino t. Inicialmente todos los vértices tienen una altura de 0 a excepción de la fuente s el cual tiene una altura $n = |V|$, el cual es el numero de nodos del grafo.

Como se había comentado anteriormente el algoritmo consta de dos operaciones básicas las cuales serán descritas a continuación.

[2]Un algoritmo *cache-aware* está diseñado para minimizar el movimiento de paginas de memoria dentro y fuera de la memoria caché del procesador. La idea es tratar de evitar lo que se conoce como fallos de caché, que causa el bloqueo del procesador mientras carga los datos desde memoria RAM en la caché del procesador.

1. **Push** Una operación *Push* desde el vértice u hacia el vértice v consiste en enviar una parte del exceso de flujo en u hacia v. Tres condiciones deben cumplirse para poder realizar esta operación:

   - $e(u) > 0$. Tiene que existir un exceso de flujo en u.
   - $c(u,v) - f(u,v) > 0$. Capacidad disponible desde u hacia v.
   - $h(u) > h(v)$. Solo se puede enviar flujo hacia un nodo con menor altura.

   El flujo enviado es igual al $min(e(u), c(u,v) - f(u,v))$.

2. **Relabel** Realizar esta operación consiste en incrementar la altura de un vértice u hasta que sea por lo menos mayor en 1 que algunos de los vértices con capacidad disponible para enviarles flujo. Las condiciones para esta operación son:

   - $e(u) > 0$. Tiene que existir un exceso de flujo en u.
   - $h(u) \leq h(v)$ para todo v tal que $c(u,v) - f(u,v) > 0$. Solo los vecinos con capacidad disponible cuentan para aplicar el *Relabel*.

   El aumento de la altura del vértice u es: $h(u) = 1 + min(h(v) : (u,v) \in E_f)$

Para inicializar el algoritmo se realiza una operación denominada *Preflow(G,s)* en la cual la fuente s envia su flujo el cual es infinito por todas sus aristas hacia los vértices a los cuales esta conectada con capacidad disponible.

Un algoritmo genérico *Push-Relabel* seria:

Algoritmo 1 Algoritmo *Push-Relabel*

```
1: funcion PUSH-RELABEL
2:     Inicializar_Preflow(G, s)
3:     mientras push or relabel hacer      ▷ mientras se pueda aplicar push o relabel
4:         Push()
5:         Relabel()
6:     fin mientras
7: fin funcion
```

2.2.7. 3D-GrabCut para el flujo máximo con Push-Relabel

Para este caso de estudio se creó un nuevo algoritmo paralelo de *Push-Relabel*. *Push* es una operación local de cada nodo donde se envía flujo a sus vecinos reduciendo el exceso propio. Un vértice también puede recibir flujo de sus vecinos, debido a que estas operaciones se hacen de forma paralela esto puede ocasionar ciertas inconsistencias en el momento de la actualización del flujo.

Esto se puede evitar utilizando funciones atómicas [33], las cuales realizan una operación de lectura, escritura o modificación de una palabra de 32 o 64 bits en memoria global o compartida, esta operación garantiza que se realice la misma hasta el final sin la interferencia de otro hilo de ejecución evitando el posible error mencionado anteriormente.

En el algoritmo original de *push-relabel*, en la etapa de *Relabel* local si un nodo no tiene posibilidad de llegar hasta el destino este debería incrementar su altura h hasta que supere la altura de la fuente por 1 y devolverle el exceso de flujo. Conociendo que la altura de la fuente es igual a N el cual es el número de vértices, el algoritmo debería efectuar en el peor de los casos N operaciones *Relabel*. Para nuestro caso de estudio estamos tratando con imágenes volumétricas se utiliza un valor N muy grande (*i.e. en un volumen de* $128 \times 128 \times 128$*, N igual a* 2097152).

En 3D-GrabCut se reetiquetarán los vértices de manera global, en base a la distancia con el nodo destino. Dicha operación global se hace en base a que nodos tienen capacidad disponible con el destino t, dichos nodos tendrán una altura igual a la altura del destino más uno (como la altura del destino es 0 la altura de estos nodos será igual a 1). Así sucesivamente en la siguiente iteración les tocará a los nodos que tengan capacidad disponible con otro nodo que ya esté etiquetado y su nueva altura será igual a la altura del nodo conocido más uno. A medida que transcurren las iteraciones del algoritmo algunos nodos quedarán aislados del destino y su única conexión será con la fuente, esto ocasionará que el *Relabel* global no los tome en cuenta y se quedarán sin etiquetar. Al ocurrir dicha acción y si los nodos tienen un exceso, el mismo nunca será drenado y el algoritmo nunca se detendrá. Para solventar esto se utilizará el mismo enfoque y se etiquetaran los nodos que el *Relabel* no tomó en cuenta en base a la fuente, de esta manera si un nodo esta con capacidad disponible con la fuente s y se encuentra sin etiquetar, su altura será igual a la altura de la fuente más uno(como la altura de la fuente es igual a N la altura de estos nodos será igual a $N+1$), y se seguirá hasta que a todos los nodos se les asigne una altura se puede apreciar un esquema de esta operación en la Figura 2.10.

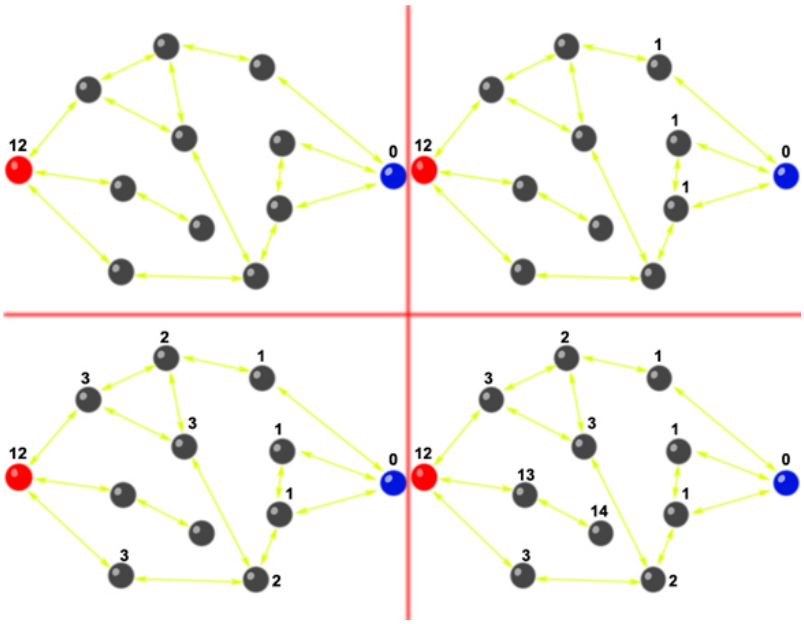


Figura 2.10: Operación relabel

Una desventaja que se encontró en el momento del desarrollo de la técnica fue que a medida que el algoritmo se iba ejecutando existían hilos que por no poseer exceso no ejecutaban ninguna instrucción. Para solventar esto en cada iteración se va creando una cola la cual contiene el indice de los nodos que van a ejecutarse en la siguiente iteración para poder crear los hilos necesarios y no quede ninguno inactivo para utilizar toda la capacidad del dispositivo.

A continuación se presenta el algoritmo pseudo-formal *Push-Relabel*. Las funciones *Pre-flow, Push, Relabel_ Global* y *Crear_ Cola* se mostrarán con más detalle en la sección anexos.

Algoritmo 2 Algoritmo Max-flow

```
1: funcion MAX-FLOW
2:     Relabel_Global()
3:     Preflow()
4:     mientras Exceso hacer                ▷ mientras algún nodo tenga exceso
5:         Push()
6:         Relabel_Global()
7:         Crear_Cola()
8:     fin mientras
9: fin funcion
```

Capítulo 3

Detalles de Implementación

3.1. Recursos de Hardware/Software de implementación:

Para la implementación de la aplicación se decidió utilizar el lenguaje de programación C sobre la arquitectura CUDA, llamado CUDA C [33], para realizar las operaciones que se van a ejecutar de forma paralela en la GPU. Para la interfaz del programa se empleó QT [34] la cual es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario.

Para la visualización de los volúmenes se utilizó *OpenGL*® el cual es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

Una de las limitaciones que podemos encontrar al utilizar CUDA es la utilización de un hardware específico, para poder utilizar esta herramienta es necesario poseer una tarjeta gráfica multinucleo NVidia compatible con CUDA® [35] . Dado que la implementación requiere algunas operaciones especiales en CUDA (las funciones atómicas), es imprescindible que posea **capability**[1] versión 1.1.

Al mismo tiempo se utilizó programación orientada a objetos. Por otra parte, el código realizado en CUDA tuvo que ser transformado en una biblioteca estática para

[1]La capacidad de cómputo (*compute capability*) describe las características del hardware y refleja el conjunto de instrucciones soportadas por el dispositivo así como otras especificaciones [36].

poder así enlazarse con la interfaz gráfica hecha en QT. A continuación presentaremos algunos elementos importantes a considerar en la implementación.

3.2. Implementación 3D-GrabCut

3.2.1. Estructuras de Datos

Entre las estructuras de datos utilizadas tenemos `GrabCut`, `PíxelNormalizado`, `point`, `GMM` y `ComponenteGaussiano`.

Una de las clases principales de la aplicación es la clase `GrabCut` dentro de la cual tenemos las variables *inicio* y *final* que son de tipo *point* y sirven para representar dos puntos los cuales se utilizan para delimitar el cubo contenedor del volumen a segmentar. Dentro de esta clase existen dos objetos de la clase `GMM`, el del *foreground* y el del *background*.

Usualmente las imágenes en formato RGB se leen con valores entre 0 y 255, debido a que en la técnica GrabCut se trabajo con valores entre 0 y 1, cada píxel es normalizado y convertido a la estructura PíxelNormalizado que tiene 3 componentes, Rojo, Verde y Azul, los cuales son tomados a partir de la función de transferencia ya que la imagen de entrada está en escala de grises.

El grafo está contenido en dos variables de tipo apuntador a **Entero** (*en C se representa como int* var;*) debido a que las funciones en CUDA trabajan con estos tipos de variables. Para los T-Links se tiene una lista de tamaño igual al número de nodos y para los N-links se cuenta con una lista cuyo tamaño es igual al numero de nodos $\times$ 8, dado que cada nodo tiene posibles conexiones con 6 vecinos, la fuente y el destino. Debido a que las funciones atómicas trabajan en principio con enteros y por presentarse problemas con la representación del 0 con variables de tipo **Flotante** en los valores tomados de los GMM los cuales son de este mismo tipo, los mismos son multiplicados por una constante k (para la ejecución de las pruebas se utilizó un $k = 1000$) y posteriormente son convertidos a variables de tipo **Entero**.

Existen 2 matrices adicionales del tamaño del volumen, en la clase `GrabCut`, *trimaps* y *mattes*. La primera almacena por cada posición el valor *trimap* del vóxel (*background*, *foreground* ó *unknown*) y la segunda matriz almacena la información referente al *matte* del vóxel (*foreground* ó *background*).

Existen dos objetos de clase `GMM`, uno para el foreground y otro para el background. En cada GMM se tiene un valor con el número de píxeles agregados a él, y 5 componentes gaussianos.

Los componentes gaussianos se representan con la clase `ComponenteGaussiano` Dentro de cada componente se almacena la media de este componente, la matriz de covarianza, la inversa de la matriz de covarianza, el determinante de la matriz de covarianza, el peso del componente dentro del GMM, los autovalores, autovectores y un contador con el número de vóxeles agregados a ese componente.

3.2.2. Algoritmo pseudo-formal e implementación

A continuación se presenta un pequeño algoritmo en pseudo-formal de la implementación:

1. Inicializar la clase GrabCut.
2. Seleccionar el Sub-Volumen a Segmentar.
3. GrabCut.set_foreground(Point inicio, Point final).
4. Fin del Algoritmo.

Para la inicialización de la clase se llama al constructor de la clase `GrabCut`, luego de seleccionar el sub-volumen a segmentar se debe llamar a la función **set_foreground(Point inicio, Point final)**

```
//Constructor de la clase GrabCut
GrabCut::GrabCut(void)

//Función que inicializa los trimaps y los mattes del volumen
//a partir del sub-volumen seleccionado
//Los parametros son de tipo Point _inicio y _final
//sirven para delimitar el cubo que contiene el sub-volumen
void GrabCut::set_foreground(Point _inicio, Point _final)
```

Para la inicialización de los modelos gaussianos, se llama al constructor de la clase `GMM`:

```
//Constructor de la clase GMM
GMM::GMM()

//Función que calcula la componente que se le pase por parametro
void GMM::calcularComponente(int compNum)
```

Para el paso 4 del algoritmo de GrabCut es necesario conocer la probabilidad de que un píxel pertenezca a un componente específico de su GMM respectivo, para esto la clase `GMM` provee la siguiente función:

```
//Permite conocer la probabilidad de que un vóxel pertenezca
//a un componente específico de este GMM
//Utiliza los parametros componente el cual es el
//número del componente
// y el valor de color del vóxel al que queremos calcular
//su probabilidad
//retorna la probabilidad de que el vòxel pertenezca al componente
float GMM::probabilidad(int componente, VoxelNormalizado c)
```

Para el cálculo del peso de los T-Links es necesario conocer la probabilidad de que un píxel pertenezca a un GMM, para esto la clase `GMM` provee la siguiente función:

```
//Permite conocer la probabilidad de que un vóxel
//pertenezca a este GMM
//recibe de parametro el voxel "c" para calcular su probabilidad
//retorna el valor de la probabilidad
float GMM::probabilidad(VoxelNormalizado c)
```

En los pasos 3, 4 y 5 del algoritmo de GrabCut, es necesario agregar información de píxeles a los diferentes componentes de un GMM, para esto la clase `ComponenteGaussiano` provee la siguiente función:

```
//Esta función agrega un voxel a este componente
//Se le pasa por parametro el voxel "c" a ser agregado
void ComponenteGaussiano::AgregarPixel(VoxelNormalizado c)
```

Luego de agregar todos los píxeles a un componente, se invoca la función **calcularComponente(int totalPíxeles)** para calcular las variables necesarias del mismo.

```
//Calcula la matriz de covarianza, su inversa, el determinante
//los autovalores y autovectores, la media y el peso del componente
void ComponenteGaussiano::CalcularComponente(int totalPixeles)
```

3.2.3. Implementación en CUDA

Para la aceleración del algoritmo de Max-flow se utilizó esta herramienta, la implementación anterior y la que va a ser descrita fueron transformadas en una biblioteca

estática (*.lib*) para poder ser utilizadas por la interfaz gráfica, las funciones que se necesitan fueron importadas para poder ser utilizadas por la clase interfaz mediante la declaración:

```
__declspec (dllexport)valor-devuelto nombre-funcion (argumentos)
```

Aquí se tiene contenida a la clase GrabCut, ya que esta es la encargada de la inicialización de la memoria, los datos y ejecución de las funciones de CUDA (*kernels*).

Después que el usuario ha escogido el sub-volumen esta se llama a la siguiente función que reserva los espacios de memoria necesarios, inicializa los datos copiando los necesarios a la memoria del dispositivo que van a ser utilizados por los *kernels* y crea el grafo respectivo.

```
//Función encargada de reservar los espacios de memoria
//inicializar las variables, copia en la memoria del dispositivo las que sean necesarias
//y por ultimo crea el grafo
// recibe como entrada el volumen la funcion de transferencia y las dimensiones del volumen
__declspec(dllexport) void init_memory(GLubyte* image, GLubyte* ft2, int width, int height, int deep)
```

Otra de las funciones exportadas a la interfaz es **set_foreground(int x1, int y1, int z1, int x2, int y2, int z2)**, la cual transforma estos parámetros en variables de tipo **point** para después pasarlas a la función de la clase GrabCut **set_foreground(Point inicio, Point final)**.

```
//Transforma los puntos seleccionados por el usuario
//y se los pasa a la clase grabcut para seleccionar el sub-volumen
__declspec(dllexport) void set_foreground(int x1, int y1, int z1, int x2, int y2, int z2)
```

Por ultimo la función **GrabCut()** es la encargada de llamar a las funciones destinadas al desarrollo del Max-flow y la invocación de los *kernels* de CUDA.

```
//Aqui se efectua el algoritmo GrabCut
//se realiza el Max-flow
//y se despliega el volumen resultante
__declspec(dllexport) void GrabCut()
```

3.2.4. Interfaz de Usuario

La interfaz gráfica de usuario (*GUI*) se desarrolló utilizando la biblioteca QT. En la Figura 3.1 se puede observar la interfaz de la aplicación la cual cuenta con 2 vistas

del tipo *QWidget* para mostrar diferentes planos de profundidad en dirección de un eje (*Z y Y respectivamente*), de esta manera el usuario podrá definir un rectángulo en la primera vista el cual va a servir para delimitar el sub-volumen, en la segunda vista se seleccionará la profundidad a la cual llegará el cubo delimitador. Después tenemos una vista del tipo *GlWidget* para mostrar el despliegue del volumen utilizando *OpenGL*®. Por otro lado tenemos un *QWidget* que sirve para mostrar y modificar la función de transferencia que va a ser aplicada al volumen, notese que los cambios afectan también a las vistas de los ejes.

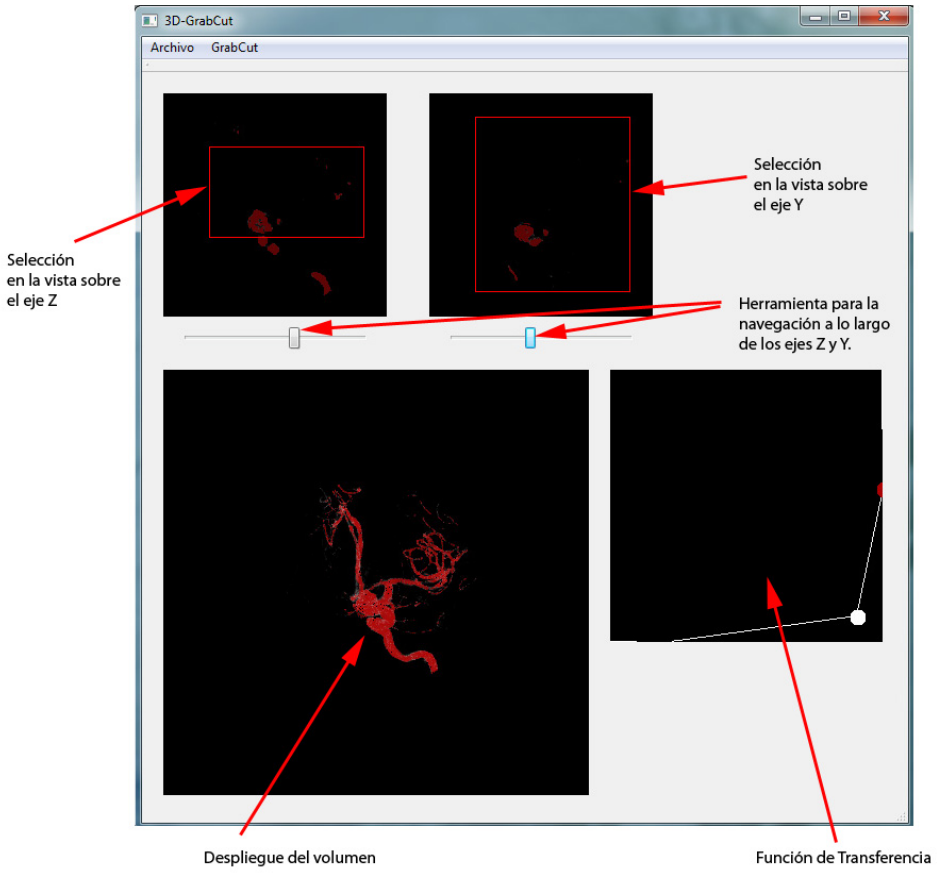


Figura 3.1: Interfaz de 3D-GrabCut

Al iniciar el programa se procede a cargar el volumen elegido mediante el menú **Archivo→Abrir** el cual desplegará una ventana en la cual procederemos a buscar la ruta del archivo de volumen (esta aplicaión solo contempla archivos de volumen con profundidad de 8 bits *.raw*) mediante el explorador, también se debe introducir las dimensiones del volumen que va a ser cargado.

El menú **Archivo** permite también guardar un volumen resultante a la segmentación; así como también, abrir o guardar una función de transferencia que se haya editado.

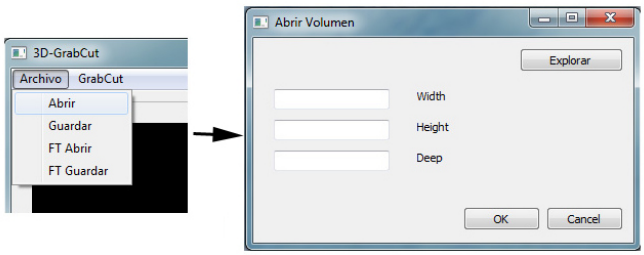
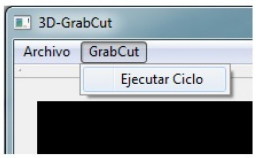


Figura 3.2: Menú para abrir el volumen, y luego la ventana donde se especifican las dimensiones.

En el menú **GrabCut** esta la opción **iniciar ciclo** el cual dará comienzo a la ejecución del algoritmo de segmentación.



Una vez finalizado la implementación de la versión paralela de GrabCut para volúmenes, 3D-GrabCut, se realizaron una serie de pruebas para comprobar su correcto funcionamiento. En el siguiente capítulo se muestran las pruebas realizadas así como el análisis de éstas.

Capítulo 4

Pruebas y Resultados

Una vez finalizada la etapa de diseño y desarrollo es necesario poner a prueba el sistema para de esta forma evaluar el redimiento, eficiencia y precisión para determinar si se alcanzan los objetivos planteados. En este capítulo se presentarán las pruebas realizadas para la segmentación de un volúmen mostrando los resultados cualitativos y cuantitativos.

4.1. Descripción del ambiente de pruebas

Para las pruebas se requirió de un hardware en particular para poder ejecutar el algoritmo. En la Tabla 4.1 se muestran las arquitecturas utilizadas.

Arq.	Procesador	RAM	Sistema Operativo
1	Intel(R) i3 540 3.70 GHz	4.00 GB	Windows 7 64 bits
2	Intel(R) i5 650 3.20 GHz	4.00 GB	Windows 7 64 bits

Cuadro 4.1: Arquitecturas utilizadas en el ambiente de pruebas

Es importante notar que para el caso de la arquitectura 1 se realizaron dos implementaciones para estas pruebas, la primera fue realizada con el objetivo de que se ejecutara en la GPU y la segunda para ser ejecutada el el CPU, para comparar los resultados de las mismas. Para el caso de las implementaciones realizadas para ser ejecutadas en la GPU se utilizó paralelismo en el dominio de los datos.

Arq.	Tarjeta Gráfica	# Cores	Ancho de Banda	Memoria	capability
1	GTX 470	448	113.9 GB/s	1280 MB	2.0
2	GT 240	96	57.6 GB/s	1024MB	1.2

Cuadro 4.2: Tarjetas gráficas de cada arquitectura del ambiente de pruebas

Los volúmenes utilizados en todas las pruebas tiene el formato de archivo crudo *.raw*, con una precisión de muestreo de 8 bits. Las características de los volúmenes empleados se presentan en la Tabla 4.3 a continuación.

Volumen	1	2	3	4
Dimensiones (w×h×d)	41×41×41	256×256×128	256×256×256	256×256×256
Tamaño MB	0.06MB	8MB	16MB	16MB
N^{ro} de Vóxeles	68.921	8.388.608	16.777.216	16.777.216

Cuadro 4.3: Carácteristicas de los volúmenes utilizados en los casos de prueba

En la Figura 4.1 podemos apreciar los volúmenes que fueron utilizados para las pruebas de esta implementación:

Volumen 1: es una simulación de la distribución de probabilidad de dos cuerpos de un nucleón en el núcleo atómico $16O$

Volumen 2: es una tomografía computarizada de un motor

Volumen 3: es una tomografía computarizada de un bonsai

Volumen 4: es una resonancia magnética de una cabeza humana

La fuente de donde se consiguieron los archivos fue la siguiente: para los volúmenes 1 , 2 y 3 fueron tomados de http://www.volvis.org/ y el volumen 4 fue tomado de http://www9.informatik.uni-erlangen.de/External/vollib/ y transformado a formato (*.raw*).

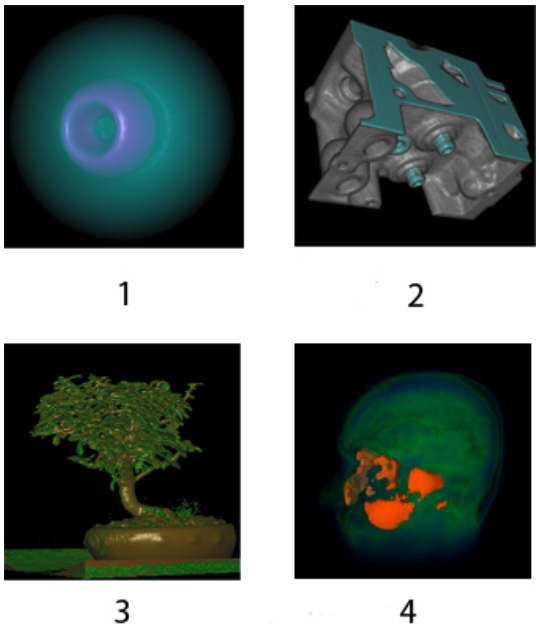


Figura 4.1: Volúmenes utilizados en los casos de prueba

4.2. Resultados cuantitativos

Para las mediciones se corrió el algoritmo al menos 10 veces con la misma configuración, midiéndose el tiempo en el caso de las implementaciones hechas en CUDA utilizando funciones propias del lenguaje, y para las mediciones para el caso de la versión en CPU se utilizó la biblioteca *timer.h*. Luego se promediaron las mediciones que se presentan en las tablas a continuación.

Las mediciones se hicieron en base a dos etapas del algoritmo: la inicialización de las variables y la posterior creación del grafo (**e1**) la cual se mostrará en milisegundos (ms) y la ejecución del algoritmo *max-flow* (**e2**) se mostrará en segundos(s), y para cada volumen se hizo 3 casos de prueba diferentes modificando la función de transferencia (las cuales denominaremos: identidad, ft1 y ft2). Para la implementación en CPU sólo se utilizó la identidad como función de transferencia debido a que el tiempo necesitado para la culminación del algoritmo crecía considerablemente en relación con los datos de entrada.

De los resultados mostrados en la Tabla 4.4 se puede notar que los tiempos los cuales son muy cortos para las 3 arquitecturas fueron más favorables para 1 y 3, cabe destacar que la arquitectura 3 es una implementación hecha en CPU pero la misma arrojó mejores resultados que la implementación realizada en GPU en la arquitectura 2, esto no quiere

Volumen 1						
Arq.	Función de Transferencia					
	identidad		ft 1		ft 2	
	e1	**e2**	**e1**	**e2**	**e1**	**e2**
1-GPU	68.157(ms)	0.663(s)	76.316(ms)	0.487(s)	69.237(ms)	0.455(s)
2-GPU	113.022(ms)	0.948(s)	104.338(ms)	0.974(s)	102.949(ms)	0.655(s)
1-CPU	72.840(ms)	0.642(s)	—	—	—	—

Cuadro 4.4: Tabla de tiempos para el volumen 1.

decir que dicha implementación sea mejor en CPU utilizando estas arquitecturas, la razón por la cual esto sucede es porque el volumen utilizado es pequeño, por lo tanto, el tiempo que necesita el CPU para culminar el algoritmo es menor al tiempo que se necesita en la arquitectura 2 para copiar los datos en el dispositvo lo cual es un costo elevado en comparación con el tiempo que necesita el dispositivo para realizar los cálculos. Es importante notar que para las pruebas hechas en la arquitectura 3, la función de transferencia 1 y 2 no fueron realizadas debido al gran tiempo que se requeria para finalizar el algoritmo en el caso del volumen 4 (4 horas y media en promedio). Se decidió no tomar los tiempos para los otros volúmenes estudiados.

Volumen 2						
Arq.	Función de Transferencia					
	identidad		ft 1		ft 2	
	e1	**e2**	**e1**	**e2**	**e1**	**e2**
1-GPU	647.678(ms)	61.311(s)	661.710(ms)	41.296(s)	619.244(ms)	68.134(s)
2-GPU	909.793(ms)	327.307(s)	778.193(ms)	110.842(s)	740.112(ms)	184.653(s)
1-CPU	4470.720(ms)	895.619(s)	—	—	—	—

Cuadro 4.5: Tabla de tiempos para el volumen 2.

Para los resultados mostrados en la Tabla 4.5 se nota el cambio de los tiempos con respecto a la arquitectura 3 debido a que los datos de prueba son de mayor tamaño, podemos apreciar en los tiempos necesarios para la culminación de cada etapa que la arquitectura 1 presenta mejores resultados con respecto a la arquitectura 2, debido a que posee una capacidad de procesamiento en paralelo mas grande.

Volumen 3						
Arq.	Función de Transferencia					
	identidad		ft 1		ft 2	
	e1	**e2**	**e1**	**e2**	**e1**	**e2**
1-GPU	1161.866(ms)	112.101(s)	1114.512(ms)	83.015(s)	1140.115(ms)	161.234(s)
2-GPU	*	*	*	*	*	*
1-CPU	7488.969(ms)	2742.491(s)	—	—	—	—

Cuadro 4.6: Tabla de tiempos para el volumen 3.

La arquitectura 2 (indicada con un asterisco) no podía cargar los volúmenes 3 y 4 por falta de capacidad de memoria en el dispositivo. Esta memoria corresponde a la cantidad de memoria compartida asignada por la arquitectura CUDA para dicho modelo de tarjeta gráfica.

Volumen 4						
Arq.	Función de Transferencia					
	identidad		ft 1		ft 2	
	e1	**e2**	**e1**	**e2**	**e1**	**e2**
1-GPU	400.731(ms)	353.644(s)	826.943(ms)	584.227(s)	807.170(ms)	644.066(s)
2-GPU	*	*	*	*	*	*
1CPU	7875.332(ms)	19633.242(s)	—	—	—	—

Cuadro 4.7: Tabla de tiempos para el volumen 4.

Los resultados obtenidos en las Tablas 4.6 y 4.7 reflejan que aunque los volúmenes son del mismo tamaño, también influye en el tiempo de culminación otros factores entre ellos cabe destacar como uno de los más importantes es el tamaño del sub-volumen de selección el cual influye en la cantidad de hilos que estarán activos en las primeras iteraciones del algoritmo. Para estos casos de prueba el número de vóxeles dentro de cada sub-volumen de selección fueron: 1,085,312 para el volumen 3 y 3,681,600 para el volumen 4.

A continuación se mostrarán unas gráficas a partir de los resultados obtenidos de las pruebas de tiempo.

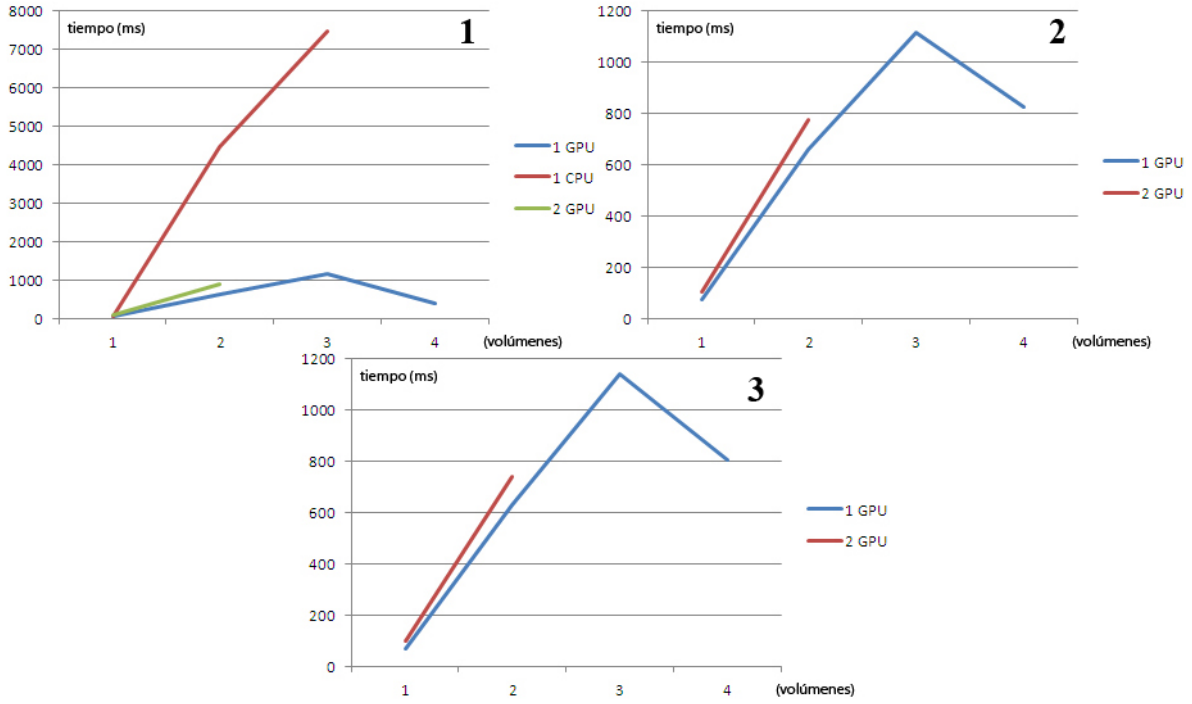


Figura 4.2: Resultados de tiempo para las funciones de transferencia 1, 2 y 3 en la etapa 1 del algoritmo.

En las gráficas mostradas en la Figura 4.2, las cuales son los tiempos de ejecución de la etapa de inicialización del algoritmo, se puede observar como el tiempo utilizado por el algoritmo implementado en CPU crece rapidamente a medida que el volumen de datos es más grande. Por otro lado las implementaciones realizadas en GPU prensentan mejores resultados en cuestión de tiempo con respecto al tamaño de los datos de entrada. en todo momento la arquitectura 1 se mantiene por debajo de la arquitectura 2 gracias a su capacidad de computo en paralelo puede ejecutar más bloques de hilos simultaneamente.

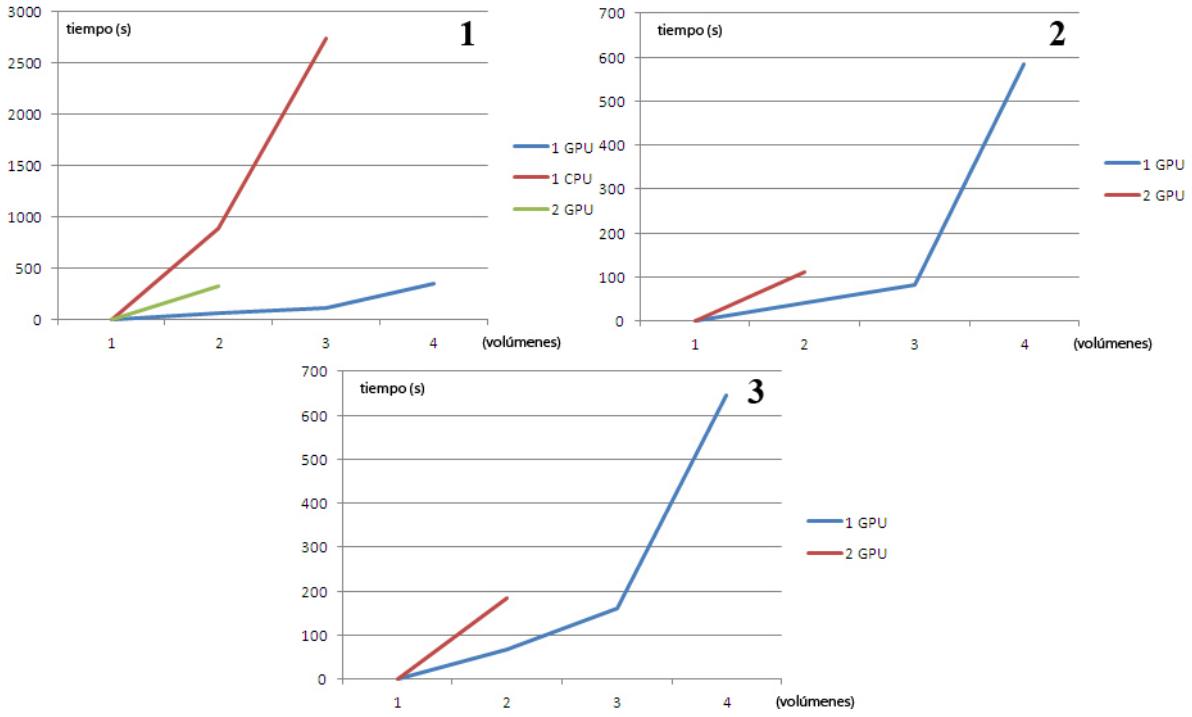


Figura 4.3: Resultados de tiempo para las funciones de transferencia 1, 2 y 3 en la etapa 2 del algoritmo.

Al observar los resultados obtenidos en las gráficas mostradas en la Figura 4.3 de la etapa correspondiente a la ejecución del *max-flow*, el CPU demostró no ser un buen candidato para la ejecución del algoritmo dando tiempos demasiado altos, llegando hasta necesitar alrededor de 5 horas para culminar su ejecución para el volumen 4. Con respecto a las arquitecturas 1 y 2 vemos que la primera de estas sigue dando un mejor resultado en cuestiones de tiempo, otro detalle importante es el crecimiento en el tiempo necesario para dos volúmenes de igual tamaño (volumen 3 y 4) variando el sub-volumen de selección para el volumen 4 el sub-volumen de selección fue más grande implicando mayor tiempo de calculo necesario.

4.3. Resultados cualitativos

A continuación se presentan los resultados visuales de la aplicación utilizando las 3 diferentes funciones de transferencia utilizadas en las pruebas de medición de tiempo. Para cada una de las pruebas se utilizó 3 funciones de transferencia diferentes ($f1$, $f2$ y $f3$) siendo $f1$ igual al valor de la función identidad, $f2$ la cual denominamos **ft1**

anteriormente, es una función que busca optimizar el resultado del algoritmo tanto en tiempo como visualmente, y por ultimo $f3$ la cual denominamos **ft2** es una función creada al igual que $f2$ de manera manual pero agregando puntos de control de una manera donde no se busca obtener un resultado en particular. Dichas funciones pueden observarse en la Figura 4.4, las cuales fueron utilizadas para la ejecución del algoritmo para el volumen 1, siendo la función identidad utilizada para todos los volúmenes en los casos de prueba, mientras que $f2$ y $f3$ varian para los volúmenes en cada caso.

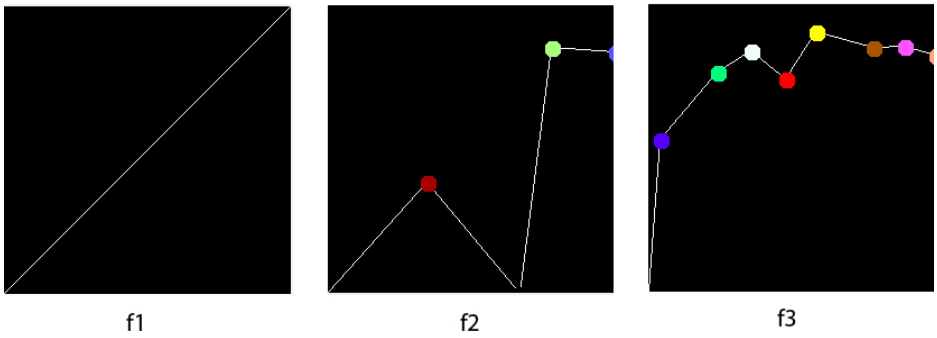


Figura 4.4: Funciones de transferencia utilizadas para el volumen 1

4.3.1. Volumen 1

En la Figura 4.5 se tienen las imágenes originales para las funciones de transferencia utilizadas ($f1$, $f2$ y $f3$) que se muestran en la Figura 4.4, para todos los casos de prueba se utiliza el mismo sub-volumen de selección, para esta prueba se busca segmentar un cuerpo con forma de anillo que se encuentra en un extremo del volumen en dirección del eje Z.

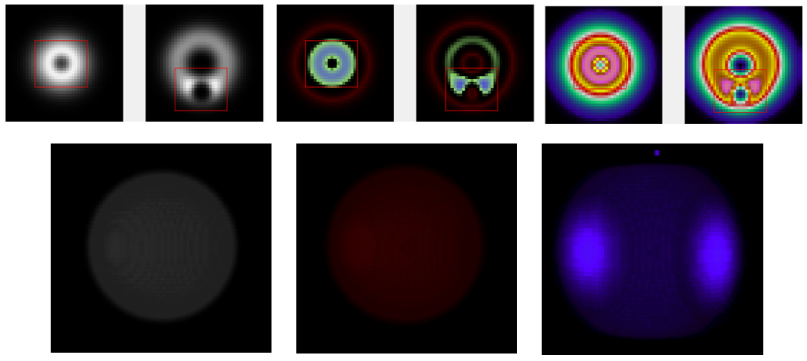


Figura 4.5: Volumen 1 con $f1$, $f2$ y $f3$.

Se puede apreciar el cambio obtenido en cada volumen despues de aplicar cada función de transferencia, siendo en el caso de $f1$ un cuerpo con forma esférica con un espacio vacio en su interior y el objeto de interés buscado con colores muy cercanos al blanco. Para el caso de $f2$ se forma una cascara esferica en color rojo en cuyo interior se forma otro cascarón en color verde claro el cual tiene en su superficie el objeto de interes. Utilizando $f3$ se tiene un solo cuerpo con diferentes capas de las cuales la más externa (en azul) va perdiendo su forma esférica y el objeto de interes disminuyo su tamaño al ir obteniendo un color similar a una de las capas internas.

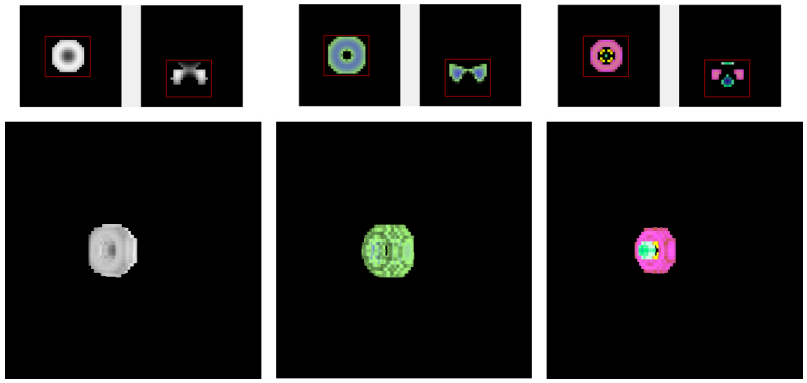


Figura 4.6: 3D-GrabCut del volumen 1.

Los resultados provenientes del primer caso de prueba se muestran en la Figura 4.6, aquí podemos ver que dichos resultados son diferentes para cada función de transferencia, debido a que el algoritmo utiliza la información de color obtenida a partir de esta, para el caso de $f1$ la función identidad da buenos resultados a pesar que solo sean colores en escala de grises. Para $f2$ el algoritmo incluyó una capa de color verde perteneciente al cascaron interno que se puede apreciar en la imagen original (vea Figura 4.5) a pesar de esto el resultado es muy aceptable. Y por ultimo para $f3$ aunque se mantiene la forma esperada fueron descartados algunos voxeles debido a que su color se asemejaba con parte del *background* y se incluyeron otros que fueron tomados como si pertenecieran al *foreground*.

4.3.2. Volumen 2

En la Figura 4.7 se encuentran las imágenes del volumen original aplicando las funciones $f1$, $f2$ y $f3$. Para este caso se busca segmentar dos piezas cilindricas y dos anillos presentes en el volumen.

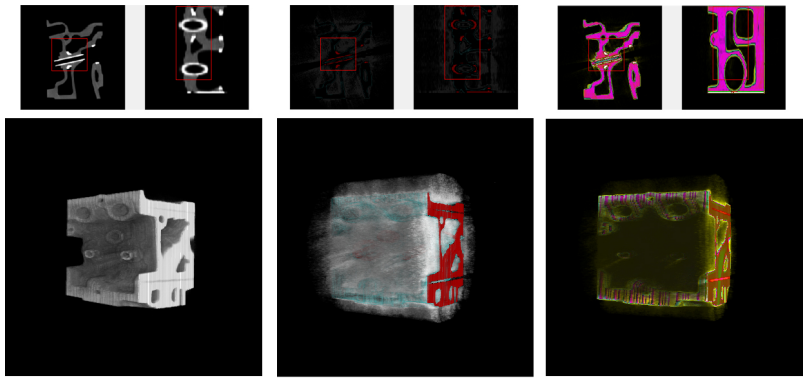


Figura 4.7: Volumenen 2 con $f1$, $f2$ y $f3$.

Después de aplicar las funciones de transferencia notamos que $f2$ y $f3$ presentan un cambio significativo con respecto a $f1$, agregando información de ruido al volumen, la cual podría interferir en la busqueda del objeto de interes deseado.

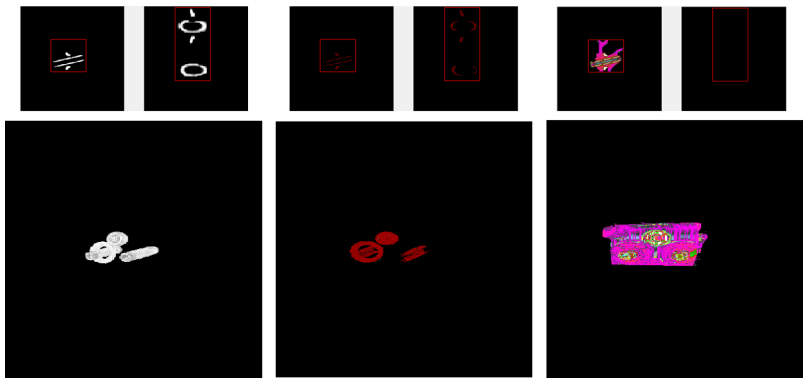


Figura 4.8: 3D-GrabCut volumen 2.

La Figura 4.8 expone los resultados obtenidos de aplicar 3D-GrabCut al volumen 2 formado a partir de $f1$, $f2$ y $f3$ que se muestra en la Figura 4.7. De dichos resultados $f1$ logró resultados consistentes con realición a lo que se andaba buscando, $f2$ logró resultados muy similares aunque con una erosión en las partes cilindricas del objeto de interés, mientras que $f3$ logró la iclusion del volumen que es parte del *background*, mostrando que no es una buena elección para este caso de segmentación.

4.3.3. Volumen 3

A medida que se van realizando las pruebas los volúmenes de entrada son más grandes y complejos, de esta manera podemos analizar el desempeño del algoritmo con estos datos. En la Figura 4.9 mostraremos el volumen 3 el cual es mas complejo que los dos primeros que habiamos analizados aplicando $f1$, $f2$ y $f3$.

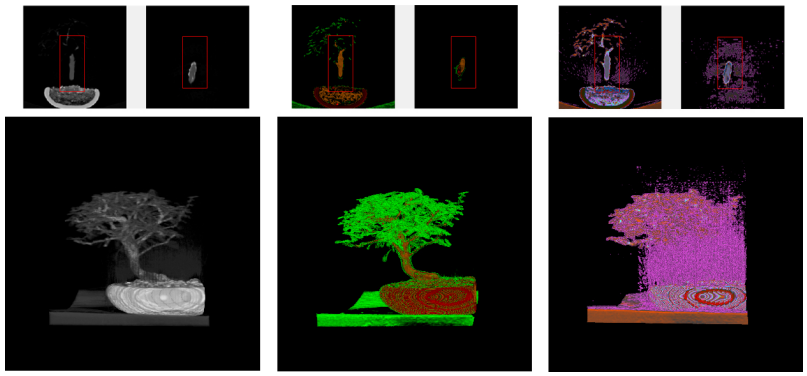


Figura 4.9: Volumenen 3 con $f1$, $f2$ y $f3$.

Para los datos de entrada en este caso de prueba notamos que $f1$ presenta un buen volumen inicial, excepto que aporta un poco de ruido proveniente de la imagen original en la parte del tallo. Para solucionar esto en $f2$ se buscó eliminar dicho ruido y ofrecer una visión mas común del volumen quese esta estudiando. Por otro lado $f3$ resalta y aumenta el ruido anteriormente mencionado y ofrece colores similares para cada parte del volumen.

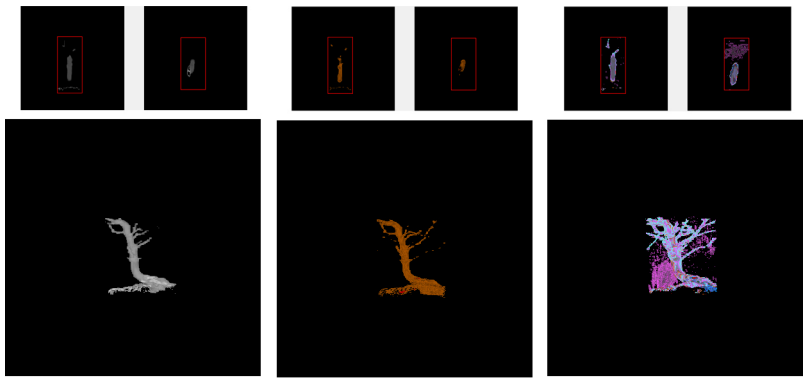


Figura 4.10: 3D-GrabCut volumen 3.

Los resultados obtenidos mostrados en la Figura 4.10 muestran que a pesar del ruido introducido en $f1$ este fue eliminado por el algoritmo y ofrece un buen resultado, $f2$ por

otra parte muestra un mejor resultado como se puede notar al incluir mas ramas que el resultado obtenido con $f1$. El resultado utilizando $f3$ incluyó el ruido introducido del volumen original. En los tres casos el algoritmo añadió una parte de la superficie de la tierra donde se encotraba el bonsai debido a que el color de esta es muy parecido al color del objeto de interés.

4.3.4. Volumen 4

Este último caso de prueba a parte de ser uno de los más grandes se busca segmentar una parte de mayor tamaño que en los otros casos de prueba. El objeto de interés que se trata de obtener en esta segmentación es el cerebro de la persona el cual serviría para un posterior análisis el cual mostará una dificultad mayor para la ejecución del algoritmo. Los datos de entrada a los cuales se les aplicó $f1$, $f2$ y $f3$ se muestran en la Figura 4.11.

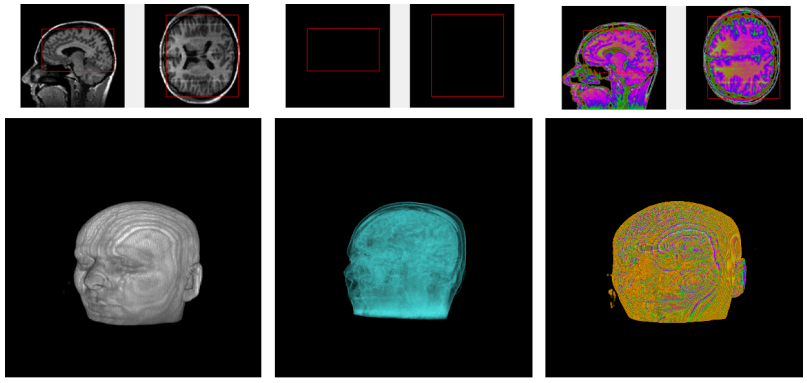


Figura 4.11: Volumenen 4 con $f1$, $f2$ y $f3$.

Para el caso $f1$ los colores son muy similares debido a que esta en escala de grises, por otra parte con $f2$ a pesar de que se utilice un solo color se nota que la región de interes (cerebro) se nota mejor que con $f1$ y con $f3$ el cerebro tiene colores muy similares con el resto del objeto.

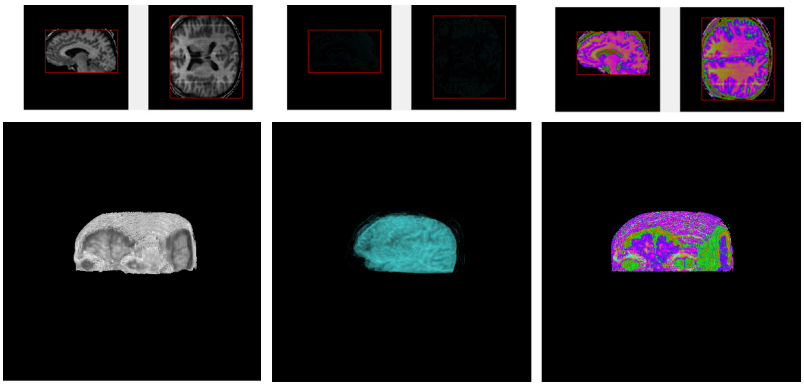


Figura 4.12: 3D-GrabCut volumen 4.

Los resultados obtenidos con $f1$ y $f2$ fueron muy similares dejando en los datos de salida una parte que no debería encontrarse en el resultado esperado. En cambio, $f2$ logró un resultado bastante bueno eliminando en gran parte los vóxeles pertenecientes al *background* que $f1$ y $f2$ incluyeron en los datos de salida.

Se puede concluir que la edición de la función de transferencia es muy importante, tanto en el tiempo de computo como el resultado visual.

4.4. Mediciones de memoria

Para las mediciones de memoria se tomaron en cuenta las estructuras de datos que ocupan mayor espacio de memoria tales como el volumen, el grafo, las listas que guardan los valores del tipo de nodo (*trimap matte*) la lista de excesos, la cola de hilos activos y la lista que guarda el valor de la conexión entre la fuente (s) con los nodos. Los valores de la memoria utilizada para los casos de prueba se muestra en la Tabla 4.8.

Volumen	1	2	3	4
tamaño del volumen (MB)	0,06 MB	8 MB	16 MB	16 MB
memoria necesaria (MB)	3,5 MB	424 MB	848 MB	848 MB

Cuadro 4.8: Memoria necesaria para alojar las estructuras de datos importantes.

Se puede observar que a medida que los volúmenes de datos aumentan la memoria requeria aumenta de manera muy considerable, siendo esto una de las principales desventajas del algoritmo.

4.5. Optimización en la carga de hilos en el dispositivo

Como mencionamos en el capitulo 3, en nuestra implementación desarrollamos una cola para almacenar los hilos que se encontrarán activos para la siguiente iteración del algoritmo en paralelo, sin dicha optimización siempre se estaría creando un número de hilos igual al número de nodos en el grafo, de los cuales a medida que avanza el algoritmo la cantidad de hilos que se mantendrían inactivos sería muy grande, siendo esto un desperdicio de uso computacional por parte del dispositivo. A continuación se mostrará unos gráficos que reflejan la cantidad de hilos activos a lo largo de las iteraciones del algoritmo en los casos de prueba para los volúmenes 1, 2 y 3.

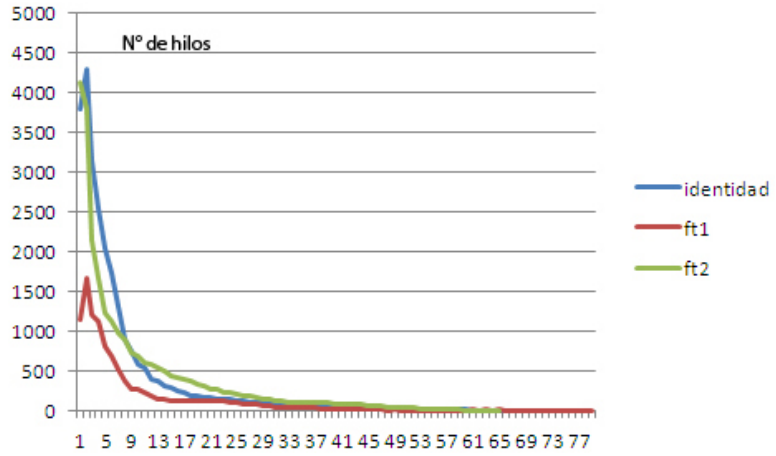


Figura 4.13: Hilos utilizados para la ejecución del algoritmo en el volumen 1

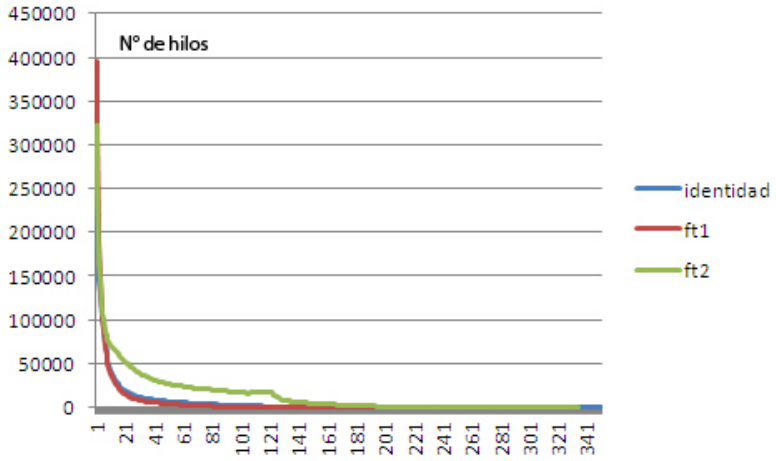


Figura 4.14: Hilos utilizados para la ejecución del algoritmo en el volumen 2

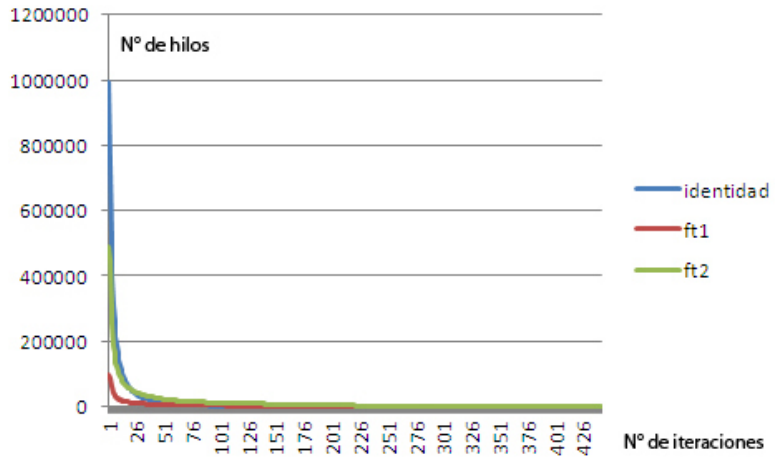


Figura 4.15: Hilos utilizados para la ejecución del algoritmo en el volumen 3

4.6. GPU vs CPU

Los resultados visuales mostrados anteriormente fueron obtenidos utilizando la arquitectura 1, para la arquitectura 2 los datos de salida fueron iguales en los casos de prueba. Con respecto a los resultados obtenidos con el CPU se analizó la diferencia entre los resultados obtenidos entre la arquitectura 1 y 3, los cuales se muestran en la Figura 4.16.

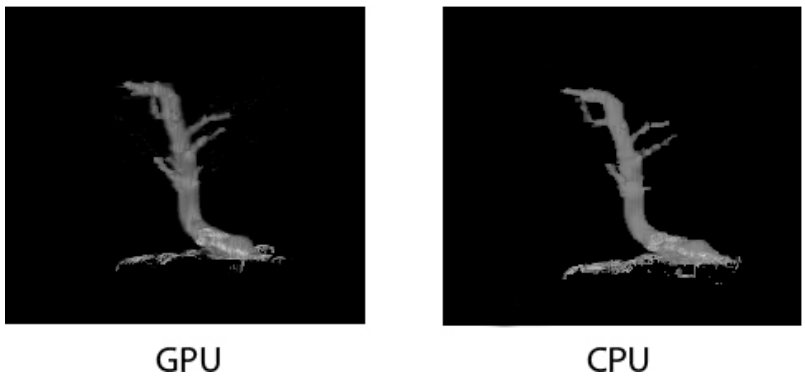


Figura 4.16: Diferencia entre los resultados obtenidos por las arquitectura 1 y 3.

Estos resultados muestran que los datos de salida pueden ser diferentes utilizando el mismo enfoque, la diferencia está en la manera de ejecución del algoritmo, al ser en paralelo el orden en que cada nodo actualiza su exceso modifica de manera diferente el

grafo que cuando se ejecuta en modo secuencial, como lo hace el CPU, esto hace que el grafo se vaya volviendo diferente cuando se ejecuta en forma paralela en comparación con una forma secuencial. Igualmente los resultados que se obtienen cumplen con los requisitos para ser considerados correctos, debido a que un grafo puede tener mas de un corte mínimo.

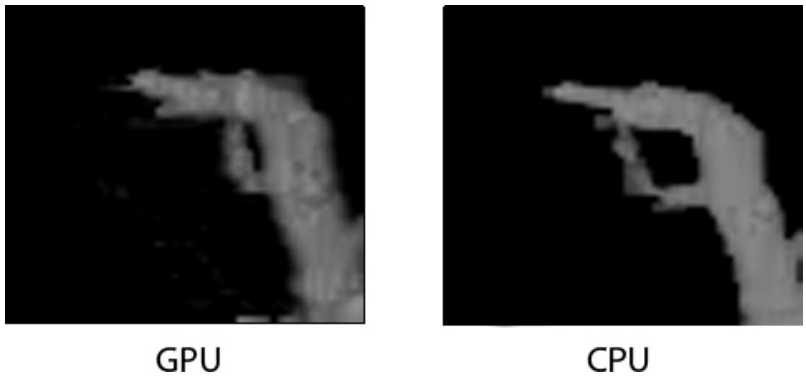


Figura 4.17: Zoom aplicado a los resultados obtenidos en la arquitectura 1 y 3.

En la figura 4.17 podemos observar con mayor detalle la diferencia entre los resultados obtenidos entre la arquitectura 1 y 3.

Capítulo 5

Conclusiones y Trabajos Futuros

5.1. Conclusiones

En este trabajo se desarrolló una implementación de la técnica de GrabCut para volúmenes llamada 3D-GrabCut lo cual es un gran aporte debido a que existen solo implementaciones de la misma en 2D. Se utilizaron técnicas de paralelismo sobre el hardware paralelo de la tarjeta gráfica, el cual demostró el incremento en el rendimiento de la solución. En las pruebas realizadas se obtuvo excelentes resultados tanto visuales como en tiempo.

Dada que la implementación se realizó bajo la arquitectura CUDA, se requiere la existencia de una tarjeta de video marca NVidia con soporte mínimo de su *capability* 1.1 lo cual es una limitante al momento de realizar las pruebas con diversos equipos. Sin embargo, para tener tiempos y resultados visuales de referencia se implemento el mismo algoritmo en su forma secuencial en el CPU.

Al aplicar el algoritmo sobre los volúmenes de prueba, utilizando la identidad como función de transferencia, se obtenían resultados visuales aceptables pero con presencia de ruido. Dichos resultados fueron cambiando a medida que se editaba la función de transferencia tanto en el tiempo (mayor o menor tiempo) como visualmente. Por ello, se puede afirmar que la efectividad de los resultados está afectada directamente por la función de transferencia que se esté utilizando.

Las estructuras de datos empleadas fueron creadas para ser utilizadas en un ambiente de ejecución paralela. Las aristas, los T-links y los excesos del algoritmo de flujo máximo fueron adaptados de la versión realizada en CPU para el GPU. Igualmente, se

implementó una versión de una cola con acceso múltiple empleando funciones atómicas en CUDA.

En cuanto a la interfaz de usuario, se creó la parte gráfica y la parte lógica como dos módulos independientes. La interfaz fue desarrollada en QT, que cuenta con diferentes vistas del volumen para realizar el proceso de selección del sub-volumen de manera eficaz. La lógica, la parte que realiza el cómputo, se trabajo como un librería estática para así ser independiente del la GUI.

Una de las principales desventajas es el tiempo de cómputo para volúmenes grandes o cuando el sub-volumen seleccionado sea de gran tamaño. Esto se debe por una parte a la gran cantidad de operaciones necesarias para el algoritmo de GrabCut, y por otra parte la cantidad de memoria disponible en la tarjeta gráfica para almacenar las estructuras de datos.

3D-GrabCut obtiene buenos resultados con poca intervención humana para la segmentación de volúmenes. La utilización de la arquitectura CUDA permitió realizar los algoritmos paralelos de una manera más rápida que su versión secuencial. Las pruebas efectuadas demuestran que el tiempo empleando el GPU es menor en comparación con el CPU a medida que el número de vóxeles aumenta (volúmenes más grandes). Para el caso de volúmenes de poco tamaño, el tiempo de transmisión de los datos desde el *host* (CPU) al *device* (GPU) es un factor importante, lo cual ocasiona un mayor tiempo de ejecución de 3D-GrabCut sobre la versión en CPU.

5.2. Trabajos futuros

Como parte de los trabajos futuros se propone ampliar las funcionalidades del programa implementando técnicas de pre-procesamiento del volumen como eliminación de ruido, aplicación de algún tipo de filtrado, etc.

Otro trabajo futuro consistiría en realizar mayores pruebas dentro del GPU como utilización de variables del tipo flotante de 32-bits y 64-bits para mayor precisión del algoritmo de flujo máximo. Estas pruebas deben ser realizadas en tarjetas gráficas que soporten dichas variables.

Dado que para volúmenes que ocupen más de 16 megabytes, el consumo de memoria en la creación del grafo ($16 \times 8 \times 4 + EstructurasDatos \sim 800$Mb), lo cual se acerca a la capacidad total de la memoria de la tarjeta gráfica disponible para el momento de las pruebas. Se podría mejorar la técnica creando un grafo solo a partir del sub-volumen seleccionado utilizando la información obtenida del volumen original.

Realizar una implementación en otros lenguajes como por ejemplo OpenCL para lograr portabilidad con otros GPU.

En el algoritmo original de GrabCut [1], se implementa el algoritmo de *matte* para mejorar el resultado en los bordes de los volúmenes. Al mismo tiempo, en el mismo trabajo, se recomienda utilizar un pincel de selección que asigna un valor a un vóxel para llevarlo al conjunto de *foreground* ó *background* de manera manual.

Bibliografía

[1] C. Rother, V. Kolmogorov, y A. Blake, "Grabcut: Interactive foreground extraction using iterated graph cuts," *ACM Transactions on Graphics*, vol. 23, pp. 309–314, 2004.

[2] Y. Y. Boykov y M.-P. Jolly, "Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images," 2001.

[3] C. Hansen y C. Johnson, *The Visualization Handbook*. Elsevier, 2005.

[4] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, núm. 3, pp. 29–37, 1988.

[5] A. Kaufman, D. Cohen, y R. Yagel, "Volume graphics," *Computer*, vol. 26, núm. 7, pp. 51–64, 1993.

[6] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, núm. 2, pp. 129–147, 1982.

[7] P. L. Williams y N. Max, "A volume density optical model," 1992.

[8] K. Engel, M. Kraus, y T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," en *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. New York, NY, USA: ACM, 2001, pp. 9–16.

[9] D. L. Pham, C. Xu, y J. L. Prince, "Current methods in medical image segmentation," *Annual Review of Biomedical Engineering*, vol. 2, núm. 1, pp. 315–337, 2000.

[10] T. Mcinerney y D. Terzopoulos, "Deformable models in medical image analysis: A survey," *Medical Image Analysis*, vol. 1, pp. 91–108, 1996.

[11] J. Sijbers, M. Verhoye, P. Scheunders, A. V. der Linden, D. V. Dyck, y E. Raman, "Watershed-based segmentation of 3d rm data for volume quantization," *Magnetic Resonance Imaging*, vol. 15, pp. 679–688, 1997.

[12] J. B. T. M. Roerdink y A. Meijster, "The watershed transform: Definitions, algorithms and parallelization strategies," 2001.

[13] T. Collins., "Graph cut matching in computer vision," 2004.

[14] Y. Boykov, O. Veksler, y R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, p. 2001, 2001.

[15] V. Kolmogorov y R. Zabih, "What energy functions can be minimized via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 65–81, 2004.

[16] D. M. Greig, B. T. Porteous, y A. H. Seheult, "Exact maximum a posteriori estimation for binary images," 1989.

[17] R. Sedgewick y K. Wayne, *Algorithms*. Boston: Addison Wesley, 2011.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, y C. Stein, *Introduction to Algorithms*. New York: The MIT Press, 2001.

[19] NVIDIA, "Cuda programming model overview."

[20] ——, "Cuda programming basics - part I."

[21] J. F. Talbot, "Abstract implementing grabcut," 2006.

[22] M. Geoffrey y P. David, "Finite mixture models," *Johm Wiley and Sons, inc.*

[23] R. Sheldon, "Introductory statistics," *Elsevier/Academic Press.*, 2005.

[24] L. R. Ford y D. R. Fulkerson, "Flows in networks," *Princeton Univ.*, 1962.

[25] A. V. Goldberg y R. E. Tarjan, "A new approach to the maximum flow problem," *Journal of the ACM*, vol. 35, pp. 921–940, 1988.

[26] J. Edmonds y R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems." en *Combinatorial Optimization'01*, 2001, pp. 31–33.

[27] R. J. Anderson y J. C. Setubal., "On the parallel implementation of goldberg maximum flow algorithm," *In SPAA*, pp. 168–177, 1992.

[28] D. A. Bader y V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," 2005.

[29] F. Alizadeh y A. Goldberg., "Implementing the pushrelabel method for the maximum flow problem on a connection machine." 1992.

[30] N. D. R. Keriven y N. Paragios, "Gpu-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction." 2005.

[31] M. H. A. Varshney y L. Davis., "On implementing graph cuts on cuda. in first workshop on general purpose processing on graphics processing units," 2007.

[32] V. Vineet y P. Narayanan, "Cuda cuts: Fast graph cuts on the gpu," 2008.

[33] NVIDIA, "Cuda programming guide."

[34] QT, "http://qt.nokia.com/products/."

[35] "http://ve.nvidia.com/," NVIDIA.

[36] NVIDIA, "Best practice guide version 2.3."

Capítulo 6

Anexos

Funciones utilizadas en el Algoritmo *Push-Relabel*

Algoritmo 3 Algoritmo Preflow

```
1:  funcion PREFLOW(S, E, A, H, C, index)
2:      id ← (blockIdx.x * blockDim.x) + threadIdx.x        ▷ obteniendo el id del vóxel
3:      si id < N°_voxeles entonces        ▷ evitar errores cuando sobran hilos de ejecución
4:          si (S[id] > 0) Y (H[id] < n) entonces
5:              E[id] ← S[id]
6:              A[id * 8] ← S[id]
7:              S[id] ← 0
8:              C[index] ← id                ▷ agregamos el indice id en la cola C
9:              index + +                    ▷ incrementamos el indice de la cola
10:         fin si
11:     fin si
12:     syncthreads()
13: fin funcion
```

Algoritmo 4 Algoritmo Push

```
1: funcion PUSH(S, E, A, H, C)
2:     ind ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si ind < N°_voxeles_activos entonces          ▷ evitar errores cuando sobran hilos de ejecución
4:         id ← C[ind]
5:         para Cada_arista hacer
6:             si (A[arista_actual] > 0) Y (H[id] > H[nodo_vecino]) entonces
7:                 si E[id] ≥ A[arista_actual] entonces
8:                     E[id] ← E[id] − A[arista_actual]
9:                     A[nodo_vecino] ← A[nodo_vecino] + A[arista_actual]
10:                    E[nodo_vecino] ← E[nodo_vecino] + A[arista_actual]
11:                    A[arista_actual] ← 0
12:                si no
13:                    A[arista_actual] ← A[arista_actual] − E[id]
14:                    A[nodo_vecino] ← A[nodo_vecino] + E[id]
15:                    E[nodo_vecino] ← E[nodo_vecino] + E[id]
16:                    E[nodo_vecino] ← 0
17:                fin si
18:            fin si
19:        fin para
20:    fin si
21:    syncthreads()
22: fin funcion
```

Algoritmo 5 Algoritmo Relabel

```
1: funcion RELABEL(A, H)
2:     H[fuente] ← N°_nodos
3:     H[destino] ← 0
4:     Init_Relabel_Global(H)
5:     mientras Exista_conexion_hacia_destino hacer
6:         Marcar_Nodos_destino(A, H, C, index)
7:         Relabel_Nodos_destino(A, H, C)
8:     fin mientras
9:     mientras Exista_conexion_hacia_fuente hacer
10:        Marcar_Nodos_fuente(A, H, C, index)
11:        Relabel_Nodos_fuente(A, H, C, index)
12:    fin mientras
13: fin funcion
```

Algoritmo 6 Algoritmo Init Relabel

```
1: funcion INIT_RELABEL_GLOBAL(H)
2:     id ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si id < N°_voxeles entonces          ▷ evitar errores cuando sobran hilos de ejecución
4:         H[id] ← −1
5:     fin si
6: fin funcion
```

Algoritmo 7 Algoritmo Marcar Nodos Destino

```
1: funcion MARCAR_NODOS_DESTINO(A, H, C, index)
2:     id ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si id < N°_voxeles entonces          ▷ evitar errores cuando sobran hilos de ejecución
4:         si H[id] < 0 entonces
5:             para Cada_arista hacer
6:                 si (A[arista_actual] > 0) Y (H[nodo_vecino] < N°_nodos) Y (H[nodo_vecino] > −1)
   entonces
7:                     C[index] ← id
8:                     index + +
9:                     break
10:                fin si
11:            fin para
12:        fin si
13:    fin si
14: fin funcion
```

Algoritmo 8 Algoritmo Relabel Nodos Destino

```
1: funcion RELABEL_NODOS_DESTINO(A, H, C)
2:     ind ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si ind < N°_voxeles_activos entonces          ▷ evitar errores cuando sobran hilos de ejecución
4:         id ← C[ind]
5:         min ← INF                                  ▷ inicializamos con un numero muy grande
6:         para Cada_arista hacer
7:             si (A[arista_actual] > 0) Y (H[nodo_vecino] < N°_nodos) Y (H[nodo_vecino] > −1)
   entonces
8:                 si min > H[nodo_vecino] entonces
9:                     min ← H[nodo_vecino]
10:                fin si
11:            fin si
12:        fin para
13:        min++
14:        H[id] ← min
15:    fin si
16: fin funcion
```

Algoritmo 9 Algoritmo Marcar Nodos Fuente

```
1: funcion MARCAR_NODOS_FUENTE(A, H, C, index)
2:     id ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si id < N°_voxeles entonces          ▷ evitar errores cuando sobran hilos de ejecución
4:         si H[id] < 0 entonces
5:             para Cada_arista hacer
6:                 si (A[arista_actual] > 0) Y (H[nodo_vecino] > −1) entonces
7:                     C[index] ← id
8:                     index++
9:                     break
10:                fin si
11:            fin para
12:        fin si
13:    fin si
14: fin funcion
```

Algoritmo 10 Algoritmo Relabel Nodos Fuente

```
1: funcion RELABEL_NODOS_FUENTE(A, H, C)
2:     ind ← (blockIdx.x * blockDim.x) + threadIdx.x          ▷ obteniendo el id del vóxel
3:     si ind < N°_voxeles_activos entonces                   ▷ evitar errores cuando sobran hilos de ejecución
4:         id ← C[ind]
5:         min ← INF                                          ▷ inicializamos con un numero muy grande
6:         para Cada_arista hacer
7:             si (A[arista_actual] > 0) Y (H[nodo_vecino] > −1) entonces
8:                 si min > H[nodo_vecino] entonces
9:                     min ← H[nodo_vecino]
10:                fin si
11:            fin si
12:        fin para
13:        min + +
14:        H[id] ← min
15:        M[id] ← usado
16:    fin si
17: fin funcion
```

Algoritmo 11 Algoritmo Cola

```
1: funcion CREAR_COLA(E, C, index)
2:     id ← (blockIdx.x * blockDim.x) + threadIdx.x           ▷ obteniendo el id del vóxel
3:     si id < N°_voxeles entonces                            ▷ evitar errores cuando sobran hilos de ejecución
4:         si E[id] > 0 entonces
5:             C[index] ← id
6:             index + +
7:         fin si
8:     fin si
9: fin funcion
```