**UNIVERSIDAD DE CHILE**
**FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS**
**DEPARTAMENTO DE INGENIERÍA ELÉCTRICA**

# IMPLEMENTACIÓN DE LA TRANSFORMADA RÁPIDA DE GAUSS EN UNA UNIDAD DE PROCESAMIENTO GRÁFICO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRICISTA

**RAFAEL IGNACIO RODRÍGUEZ OLIVOS**

PROFESOR GUÍA:
PABLO ESTÉVEZ VALENCIA

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
PABLO ZEGER FERNANDEZ

SANTIAGO DE CHILE
2008

**UNIVERSIDAD DE CHILE**
**FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS**
**DEPARTAMENTO DE INGENIERÍA ELÉCTRICA**

# IMPLEMENTACIÓN DE LA TRANSFORMADA RÁPIDA DE GAUSS EN UNA UNIDAD DE PROCESAMIENTO GRÁFICO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRICISTA

**RAFAEL IGNACIO RODRÍGUEZ OLIVOS**

PROFESOR GUÍA:
PABLO ESTÉVEZ VALENCIA

MIEMBROS DE LA COMISIÓN:
HECTOR AGUSTO ALEGRÍA
PABLO ZEGERS FERNANDEZ

SANTIAGO DE CHILE
OCTUBRE, 2008

RESUMEN DEL INFORME FINAL PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: RAFAEL IGNACIO RODRÍGUEZ OLIVOS
FECHA: 25 de octubre de 2008
PROFESOR GUÍA: DR. PABLO ESTÉVEZ V.

# IMPLEMENTACIÓN DE LA TRANSFORMADA RÁPIDA DE GAUSS EN UNA UNIDAD DE PROCESAMIENTO GRÁFICO

En el presente trabajo de memoria se realiza una implementación de la Transformada de Gauss y la Transformada Rápida de Gauss Mejorada (cuyo acrónimo en inglés es IFGT) mediante la utilización de una Unidad de Procesamiento Gráfico (en inglés Graphics Processor Unit, GPU). El objetivo es reducir el tiempo de cómputo mediante el procesamiento paralelo. Esto permitiría acelerar las estimaciones de Funciones de Densidad de Probabilidad (FDP) que se utilizan dentro de la Teoría de la Información en las medidas de divergencia e información mutua.

La implementación se realiza sobre una tarjeta de video modelo Nvidia 7900 GT, utilizando los lenguajes de programación C y Cg (C for graphics), donde este último se utiliza sólo sobre la tarjeta. La implemetación de la Transformada de Guass se realiza directamente sobre la GPU. En cambio la implementación de la IFGT se realiza una primera parte en lenguaje C, donde se determinan parámetros que se utilizan posteriormente en la GPU. Además de las implementaciones sobre la GPU se realizan implementaciones sobre una CPU de un computador estandar con el fin de comparar los tiempos y el error relativo dado por el valor absoluto de $\frac{Valor\ Real - Valor\ Implementacin}{Valor\ Real}$.

El resultado obtenido es una aceleración en el tiempo de cálculo en la Transformada Rápida de Gauss Mejorada sólo para el caso de dimensión 8 y con una cantidad de datos de 65536 . Se observa que la transferencia de los datos, es decir, la lectura de los datos desde la GPU al computador, es sumamente lenta en la implementación realizada llegando a ser de 85 % del tiempo total del programa. Se observa además que los errores de la implementación sobre la GPU son del orden de $10^{-6}$.

En conclusión, la implementación de la Transformada de Gauss y la Transformada Rápida de Gauss Mejorada sobre la GPU funciona correctamente pero no eficientemente, dado el alto tiempo que se tienen en la transferencia de los datos desde la GPU al computador. Este factor debe mejorarse en la implementaciones futuras, por lo que se proponene posibles soluciones.

*A mis padres*

# Agradecimientos

En primer lugar quiero agradecer a las personas que me acogieron durante todo el período universitario, mis tías.

Aprovecho además de agradecer a mis amigos: Daniel Baeza, Paula Carrillo, Rodrigo Inostrosa, Isao Parra, Carlos Toro, Jorge Vergara, que gracia a su apoyo y consejos en momentos en que no había luz me dieron ánimo para seguir.

También quiero agradecer al Profesor Pablo Estevéz por haber tenido la confianza de darme este proyecto. Además de agradecer a los profesores Héctor Agusto y Pablo Zegers, por haber accedido a ser parte de la comisión de este trabajo de título.

Por último un agradecimiento a todos mis compañeros, en particular a los del laboratorio Pablo, Schulz, Carloncho y Alonso con los que compartí todo este período de trabajo.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

En la actualidad dado el avance tecnológico, uno de los problemas que se presentan es la gran cantidad de datos que se generan como audio, video, imágenes, etc. y que a su vez se deben manejar. Éstos usualmente pueden no aportar nueva información, siendo redundantes y por ende es necesario discriminar seleccionando lo relevante. Toda esta información es posible cuantificarla utilizando una serie de medidas presentes en lo que se denomina Teoría de la Información (ITL).

La Teoría de la Información ha tenido impacto en el diseño de eficientes y confiables sistemas de comunicación utilizando criterios como la Divergencia, Información Mutua y la Entropía para cuantificar la cantidad de información presente en un conjunto de datos. Estas medidas dependen de la Función de Distribución de Probabilidad (FDP) que siguen los datos, la que por lo general no es posible estimarla de forma paramétrica, por ello se deben utilizar métodos matemáticos no paramétricos para el cálculo de la FDP . El método más utilizado es el Estimador de Parzen, el cual a través de la superposición de una serie de funciones gaussianas, similar a lo que se realiza en una transformanda de Fourier, permite obtener una estimación de la FDP de los datos. La estimación de la FDP y su la aplicación en el cálculo de medidas tiene un alto costo computacional del orden $O\left(N^2\right)$, siendo $N$ la cantidad de datos. En este contexto la Transformada de Gauss es una transformada que presenta un kernel gaussiano y particularmente, presenta una forma matemáticamente similar al Estimador de Parzen. Sobre esta transformada se han realizado una serie de algoritmos para disminuir su orden $O\left(N^2\right)$ y así reducir el costo computacional.

En particular se observan 2 algoritmos que permiten una disminución del orden de la Transfor-

mada de Gauss, el primero es la Transformada Rápida de Gauss presentada en por Greengard y Strain [23] que realiza una expacion de la transformada utilizando desarrollo de Series de Taylor y Hermite, obteniendo así una disminución en los tiempo trabajando con datos hasta dimensión 3. Un segundo algortimo es la Transformada Rápidad de Gauss Mejorada, la cual mejora los tiempos para dimensiones superiores a 3 pero tieniendo un limite de dimensión 10, el cambio principal se debe en la realización de *clustering* sobre los datos, además de la descomposición en Serie de Taylor [1].

Paralelamente se ha observado un avence vertiginoso en el hardware gáfico en los últimos años. Las tarjetas de video, hoy en día denominadas Unidad Procesamiento Gráfico (del inglés Graphics Processor Unit, GPU), han aumentado considerablemente su poder de procesamiento en imagenes alcanzando efectos más reales y en tiempo real. El gran rendimiento que presentan las GPUs se debe a la arquitectura de procesamiento paralelo que esta especializada para el trabajo gráfico. Este poder de procesamiento ha motivado a muchos investigadores a utilizar las GPUs como una unidad de co-procesamiento, al entregarle gran parte de la carga del procesamiento de los datos.

Dado lo anterior se han realizado una serie de implementaciones sobre las GPUs, dando inicio a lo que se denomina GPGPU (*General Programing GPU*) [24]. Los primeros programas se centraron en problemas de algebra lineal [10], para luego avanzar a programas más complejos como la programación de algoritmos genéticos en la actualidad [11] .

En esta memoria se implementa la Transformada Rápida de Gauss en la unidad de Procesamiento Gráfico con el fin de analizar la aceleración, correspondiente a la razon entre el tiempo que toma el cálculo sobre la GPU y el tiempo que tomo el cálculo sobre el procesador de un computador estandar, de la implementación dentro de ésta con respecto a una Procesador normal de un computador.

## 1.1. Objetivo General

Esta memoria tiene como objetivo general implementar la Transformada de Gauss y la Transformada Rápida de Gauss Mejorada (IFGT, por sus siglas en inglés) sobre una Unidad de Procesamiento Gráfico (GPU). La idea es acelerar los cálculos mediante la utilización de la GPU con respecto a un procesador normal de un computador.

## 1.2. Objetivos Específicos

Se plantean como objetivos específicos:

1. Implementación de la Transformada de Gauss y la Transformada Rápida de Gauss Mejorada en un Computador, en un lenguaje como C o sobre MATLAB.

2. Implementación de la Transformada de Gauss sobre la Unidad de Procesamiento Gráfico.

3. Realizar comparaciones de aceleración, entre las implementaciones dentro de una CPU y una GPU.

4. Desarrollo de una aplicación específica.

# Capítulo 2

# Antecedentes

## 2.1. Tarjeta Gráfica

Una tarjeta gráfica, tarjeta de vídeo, tarjeta aceleradora de gráficos o adaptador de pantalla, es una tarjeta de expansión para un computador, encargada de procesar los datos provenientes de la CPU (*Central Process Unit*) y transformarlos en información comprensible y representable en un dispositivo de salida, como por ejemplo el monitor.

### 2.1.1. GPU

El término GPU (*Graphics Process Unit*) fue introducido por NVIDIA a finales de los años 90 [20] cuando el término "Controlador VGA" no era lo suficientemente exacto para describir el hardware gráfico presente en una computadora.

En 1987 IBM introdujo el hardware VGA (*Video Graphics Array*) en donde la CPU era responsable de todo el trabajo de los pixeles, lo que producía un alto costo.Esto no se observa con las tarjetas actuales, donde la CPU no trabaja con los pixeles directamente y todo el manejo de éstos es realizado dentro de la GPU.

La GPU es un procesador (como la CPU) dedicado al procesamiento de gráficos; donde su función principal es disminuir la carga de trabajo de la CPU y, por ello, dado que para la imagen no se necesita una alta precisión, está optimizada para el cálculo en puntoflotante, el cual es predominante en las funciones gráficas 3D. La mayor parte de la información ofrecida en la especificación de una

tarjeta gráfica se refiere a las características de la GPU, pues constituye la parte más importante de la tarjeta gráfica. Las dos características más importantes son la frecuencia de reloj del núcleo, que en 2006 oscilaba alrededor de los 250 [MHz], y el número de pipelines (*cantidad de Vertex y Fragment Shaders*), encargadas de traducir una imagen 3D compuesta por vértices y líneas en una imagen 2D compuesta por pixeles. En la actualidad se cuentan con velocidades mucho mayores por ejemplo, 612 [MHz] en el modelo de NVIDIA GeForce 8800 Ultra [8].

La industria ha identificado 4 generaciones de GPU. Cada generación conlleva una mejora en el rendimiento y en el nivel de la programabilidad de las GPU. En el avance de cada generación también mejoran las 2 mayores interfaces de programación gráfica, OpenGL y DirectX. OpenGL es una estándar abierto para la programación gráfica en computadores con sistema operativos como Windows, Linux, UNIX y Machintosh. DirectX es una evolución de un conjunto de interfaces de programación multimedia para la programación 3D.

#### 2.1.1.1. Pre-GPU

Antes de la introducción del término GPU compañías como Silicon Graphics (SGI) y Evans & Sutherland diseñaron costosos y especializados hardware gráficos. Los sistemas gráficos desarrollados en este período introdujeron mucho de los conceptos utilizados hoy en día como lo son vértices *(vertex*), transformación y mapeos de texturas.

Las tarjetas gráficas o de video nacen a finales de los años 60 cuando se deja de usar impresoras como elemento de visualización y se utilizan monitores. La primera tarjeta gráfica, se lanzó con los primeros computadores IBM , en 1981. La MDA (*Monochrome Display Adapter*) trabajaba en modo texto y era capaz de representar 25 líneas de 80 caracteres en pantalla. Contaba con una memoria de vídeo de 4KB, por lo que sólo podía trabajar con una página de memoria. Se utilizaba con monitores monocromáticos, de tonalidad normalmente verde. Posteriormente la aparición de VGA (*Video Graphics Array*) tuvo una aceptación masiva, lo que llevó a compañías como ATI, Cirrus Logic y S3 Graphics, a trabajar sobre dicha tarjeta para mejorar la resolución y la cantidad de colores. Así nació el estándar SVGA (*Super VGA*) el cual fue definido en 1989 y en su primera versión se estableció para una resolución de 800 $\times$ 600 pixels y 4 bits de color por pixel, es decir, hasta 16 colores por pixel. Con dicho estándar se alcanzaron los 2 MB de memoria de vídeo, así como resoluciones de 1024 x 768 puntos a 256 colores. La evolución de las tarjetas gráficas dio un giro importante en 1995 con la aparición de las primeras tarjetas 2D/3D, fabricadas por Matrox,

Creative, S3 y ATI, entre otros. Dichas tarjetas cumplían el estándar SVGA, pero incorporaban funciones 3D.

A mediados de los años 90, el hardware gráfico más rápido consistía de un múltiples chips que trabajaban juntos para procesar imágenes y mostrarlas en pantalla. El sistema más complejo se componía de una docena de chips pero con el progreso del tiempo y la mejora en la tecnología de los semiconductores permitió que la complejidad de múltiples chips pasaran a estar contenida en sólo uno, desencadenando una reducción de costos y disminución en los tamaños.

#### 2.1.1.2. Primera Generación

La primera generación de GPUs, la cual se considera hasta 1998, incluye a los modelos TNT2 de NVIDIA, Rage de ATI y Voodoo de 3dfx. Estas GPU eran capaces de realizar rasterizado y transformaciones y aplicarlas a 1 ó 2 texturas. Cuando se ejecuta la mayoría de las aplicaciones 3D y 2D, las GPU alivian completamente a la CPU de la actualización de pixels individuales. Sin embargo, la GPU de esta generación presenta dos claras limitaciones. En primer lugar, éstas que carecen de la capacidad de transformar los vértices de objetos 3D, luego los vértices se producen en la CPU y en segundo lugar, tienen un muy limitado conjunto de operaciones matemáticas para combinar texturas. La potencia alcanzada por GPU fue tal que el puerto PCI (*Peripheral Component Interconnect*), dejo de tener la capacidad suficiente para las nuevas tasas de transferencias entre GPU y CPU. Por ello Intel desarrolló el puerto AGP (*Accelerated Graphics Port*) que solucionaría en parte los cuellos de botella que se empezaban a observar.

#### 2.1.1.3. Segunda Generación

La segunda Generación de GPU (1999 - 2000) incluye a los modelos GeForce 256 y GeForce2 de NVIDIA, Radeon 7500 de ATI y la Savage3D de S3. Estas GPUs podían descargar vertices 3D y realizar transformaciones e iluminaciones (*T & L*) desde la CPU. La rápida transformación de vértices fue una de las principales capacidades que diferenciaron las GPUs de primera y segunda generación. Ambos APIs (*Application Programming Interface*) gráficos OpenGL y DirectX 7 dan apoyo al hardware en las transformaciones de vértices. A pesar de que el conjunto de operaciones matemáticas para combinar texturas y colorido de pixels es ampliado en esta generación las posibilidades son todavía limitadas. Dicho de otro modo, esta generación es más configurable, pero aún no programable.

#### 2.1.1.4. Tercera generación

La tercera generación de GPU (2001) incluye a los modelos GeForce3 y Geforce4 de NVIDIA, Xbox de Microsoft y la Radeon 8500 de ATI. Esta generación proporciona una programabilidad de los vértices más que otorgar una mayor configurabilidad como es en el caso de la segunda generación de éstos. En vez de apoyar la transformación convencional de modos de iluminación dados por las interfaces de OpenGL y Directx7, las GPUs de tercera generación daban la posibilidad de especificar una secuencia de instrucciones para la transformación de los vértices. A nivel de pixeles es posible configurar algunas opciones a través de los APIs gráficos, pero de todas formas las opciones disponibles no son suficientes como para considerarlas unidades programables. Dado que la GPU de esta generación posee programabilidad para el procesador de vértices (*Vertex Processor*) pero carencias en el procesador de pixeles se considera como una generación de transición. Las memorias de las GPU debieron mejorar su velocidad, por lo que se incorporaron las memorias DDR (*Double Data Rate*) a las tarjetas gráficas. Las capacidades de memoria de vídeo en la esta generación pasan de los 32 MB del modelo GeForce, hasta los 64 y 128 MB del modelo GeForce 4.

#### 2.1.1.5. Cuarta Generación

La cuarta generación de GPU (a partir del 2002) incluye los modelos de la familia GeForce FX de NVIDIA y la Radeon 9700 de ATI. Estas GPU poseen programabilidad tanto a nivel de procesador de vértices como al procesador de pixeles. Este nivel de programabilidad abre la posibilidad de reducir la carga en el procesamiento de complejas transformaciones de vértices y operaciones de pixeles realizadas por la CPU y traspasarlas totalmente a la GPU. El desarrollo de DirectX9 y de extensiones de OpenGL proporcionaron un nivel más alto en la programabilidad para los procesadores de vértices y pixeles. En este punto se observa la aparición de lenguajes de programación para las GPU. Las tarjetas aumentaron considerablemente su número de transistores, desde una cantidad de 63 millones de transistores en 2002 con la familia GeForce 4 a una cantidad de 681 millones de transistores en 2006 con la familia GeForce 8, llegando incluso a ir duplicando su cantidad de transistores cada 12 meses lo que es mayor que la ley de Moore para las CPUs [20, 25].

En la Tabla 2.1.1se muestran las distintas generaciones con modelos de tarjetas representativos, además de la cantidad de transitores presentes en cada una de ellas.

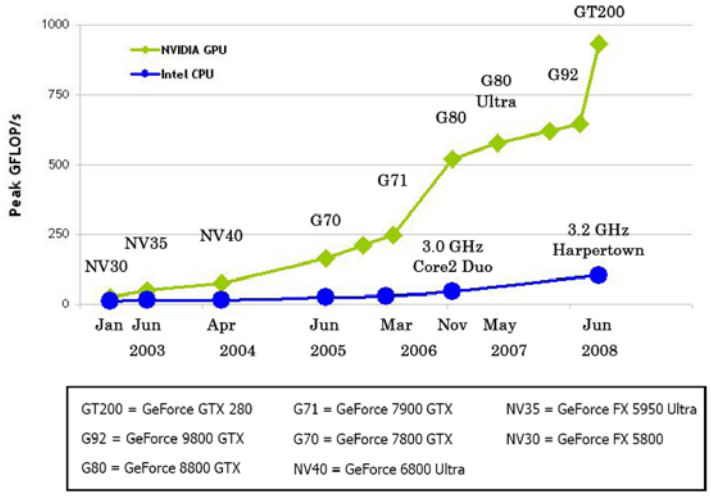
Tabla 2.1.1: Evolución de CaracterísticasGPUs de NVIDIA

| Generación | Año | Producto | Transistores (Millones) |
|---|---|---|---|
| Primera | Fines 1998 | RIVA TNT | 7 M |
| Primera | Principio 1999 | RIVA TNT2 | 9 M |
| Segunda | Fines 1999 | GeForce 256 | 23 M |
| Segunda | Principio 2000 | GeForce 2 | 25 M |
| Tercera | Principio 2001 | GeForce 3 | 57 M |
| Tercera | Principio 2002 | GeForce 4 | 63 M |
| Cuarta | Principio 2003 | GeForce FX | 125 M |

#### 2.1.1.6. Actualidad

Posterior a la Cuarta generación en que ambos Procesadores, de Vértices y de Fragmentos, pueden progrables surge una nueva variedad de tarjetas a partir del 2007, la cual posee una arquitectura totalmente distinta. Esta nueva de generación de GPUs presenta lo que se denomina una arquitectura unifica en donde no existe una diferenciación entre los Procesadores de Vértices y de Fragmentos, siendo unificados. Además se presenta un aumento en la cantidad de transistores a 754 millones, y un aumento en la cantidad de procesadores presentes 128, como en el caso del modelo Geforce 9800GTX . También hay que mencionar la última serie de GPUs producida por Nvidia, la serie 200 lanzada el 2008, la cual para el modelo GeForce GTX 280 presenta 240 procesadores.

Figura 2.1.1: Evolución en rendimiento de las GPUs Nvidia [25]



En la figura 2.1.1 se observa como ha evolucionado la cantidad de operaciones por segundo (Floating point Operation Per Seconds) que pueden realizar las GPU comparadas con los procesadores Intel. Se observa que con una nueva serie de tarjetas se produce un aumento considerable en la capacidad de cálculo, separándose cada vez más de los procesadores.

### 2.1.2. Arquitectura GPU

La GPU está compuesta por diferentes componentes que permiten que tenga un gran rendimiento ante el procesamiento de los datos de imágenes que le son entregadas para procesar. [12, 13, 15, 17, 20]

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales las que conforman el denominado *pipeline* ("tubería"). El *pipeline* gráfico es una secuencia de etapas que operan en paralelo y en un orden fijo, cada etapa recibe la entrada de la etapa anterior y envía la salida correspondiente como entrada a la etapa siguiente.

Dentro del *pipeline* gráfico se pueden detallar una serie de operaciones que se realizan entre cuales se encuentran:

- Transformación de modelos: en donde la geometría provista como una entrada es establecida en lo que se conoce como un espacio tridimensional.
- Iluminación de vértices: la geometría en una escena 3D es iluminada de acuerdo a la ubicación de las fuentes de luz, reflejo y otras propiedades de las superficies iluminadas.
- Transformación de vista: los datos son transformados de coordenadas en un modelo 3D a un sistema de coordenadas 3D basadas en la posición y orientación de una cámara virtual, es decir, resulta una escena 3D vista con la perspectiva de la cámara virtual.
- Proyección y transformación: la geometría es transformada del espacio visual de la cámara a un espacio de imagen 2D.
- Clippping: la primitivas geométricas (líneas, triángulos, puntos, cuadrados) que quedan fuera del rango de visión son descartadas del escenario.
- Texturización: los fragmentos son sombreados en este punto, los fragmentos individuales o pre-pixeles se les asigna un color basado en valores interpolados de los vértices durante la rasterización o de una textura de memoria.

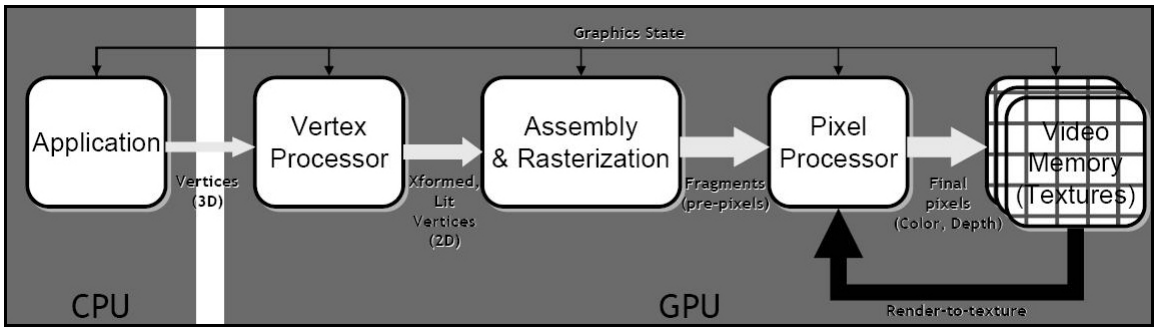



Figura 2.1.2: Pipeline gráfico

- Rasterización: es el proceso en el cual la representación de la escena en el espacio de la imagen 2D es convertida en un formato de mallas (matriz) y se determina el correcto valor del pixel.
- Despliegue: que es en donde los pixeles coloreados finalmente pueden ser desplegados en el dispositivo de salida.

Desde el punto de vista del hardware gráfico se distinguen 3 unidades que definen el *pipeline* de la GPU: donde se procesan vértices, donde se realiza la rasterización y donde se procesan los pixeles. Dos de estas unidades son programables y entregan posibilidades para el procesamiento de las imágenes, Procesadores de Vértices (*Vertex Processors*) y Procesadores de Pixeles (*Pixel Processor*).

#### 2.1.2.1. Procesador de Vértices *(Vertex Processor)*

El Procesador de Vértices o a veces llamado *Vertex Shaders* recibe los datos de los vértices y los datos de texturas desde la CPU para ser procesados por el Progrmama de Vértices *(Vertex Program)*. Desde el punto de vista gráfico en esta etapa se puede incluir transformaciones geométricas en el espacio local como traslaciones y rotaciones, con lo cual la imagen 3D es poligonalizada utilizando los vértices de entrada que luego darán origen a los triángulos. La geometría completa es iluminada definiendo la posición de las fuentes de luz y las características de la superficie. En la mayoría de las implementaciones de hardware en esta etapa del *pipeline* se calcula sólo la luz en los vértices de los polígonos y se interpolan los valores entre éstos. Luego cada uno de los triángulos son proyectados en el plano 2D considerando los datos de profundidad e iluminación calculados mediante dos tipos de proyecciones, la ortogonal y la de perspectiva.

El modelo de flujo de datos para el procesamiento de vértices comienza por cargar los atributos de los vértices (como son la posición y el color) dentro del procesador de vértices. Luego el procesador de vértices repetidamente obtiene la siguiente instrucción y la ejecuta hasta finalizar el programa de vértices..

Los registros de los atributos de los vértices son sólo de lectura y contienen la aplicación específica de atributos para el vértice. Los temporales de registros pueden ser escritos y/o leídos y se utilizan para el cálculo de resultados intermedios. Los registros de resultados son sólo de escritura. El programa ( programa de vértices) se encarga de escribir sus resultados en los registros de salida. Cuando el Procesador de Vértices termina, el resultado se encuentra en el registro de salida y contiene el vértice recién transformado. Después de la creación de triángulos y la rasterización, los valores interpolados para cada registro son enviados al procesador de fragmento [15, 20].

El procesador de vértices utiliza un conjunto limitado de instrucciones, las operaciones matemáticas. Las operaciones se realizan en estándar punto flotante de 32 bit de precisión por componente, dentro de las operaciones que puede realizar el *Vertex Shaders* se incluyen: sumar, multiplicar, producto punto, mínimo y máximo. Además de las operaciones mencionadas se cuentan con operaciones especiales dentro de las que se encuentran:

- Funciones exponenciales: $\exp x$, $\log x$.
- Recíprocos: $\frac{1}{\sqrt{r}}$
- Funciones trigonométricas: $\sin x$,$\cos x$

#### 2.1.2.2. Procesador de Fragmentos o *Pixel Shader*

El Procesador de Fragmentos a veces llamado *Pixel Shaders*, corresponde a la segunda unidad programable dentro de una GPU. Esta unidad posee las mismas operaciones matemáticas que el Procesador de Vértices pero además soporta operaciones con texturas. Las operaciones con texturas requieren acceder a una imagen de textura utilizando un conjunto de coordenadas de texturas para poder acceder.

Esta unidad trabaja al igual que el Procesador de Vértices soportando el trabajo con tipos numéricos punto flotante de 32 bit, pero las operaciones en esta unidad pueden ser más eficientes utilizando una menor precisión en el tipo de datos.

El digrama de flujo de la unidad, al igual que el Procesador de Vértices, involucra ejecutar una serie de instrucciones hasta que el programa termina. La serie de conjuntos de instrucciones son cargadas al inicio, sin embargo, en vez de los atributos de vértices, el *Pixel Processor* recibe como registros de entrada fragmentos interpolados de los parámetros derivados de los vértices. La unidad además posee registros intermedios temporales de lectura/escritura. Las operaciones de escritura en los registros de salida son el color y opcionalmente la nueva profundidad de los fragmentos. En resumen el conjunto de elementos que el *Fragment Processor* procesa son puntos, lo que cuales denominan *fragmentos* y son principalmente píxeles con información sobre el color y la profundidad.

En esta etapa, se aplica una acción sobre cada fragmento independiente, se accede a los datos de textura y a los valores interpolados de los vértices y se calculan los valores que permiten una imagen con un gran nivel de detalle. El *Fragment Processor* trabaja sobre un grupo de cientos de pixeles al mismo tiempo con una sola instrucción lo que es conocido como SIMD (una instrucción múltiples datos), es decir, una vez que se tiene definido el código del *Fragment Program* y se ejecuta, la misma operación va a ser aplicada sobre un conjunto de fragmentos.

El alto rendimiento del *Fragment Processor*, se refleja en lo siguiente:

- Las operaciones en punto flotante de 16 y 32 bit se realizan a alta velocidad, aunque el almacenamiento y las limitaciones de ancho de banda de la tarjeta pueden favorecer el rendimiento de punto flotante de 16 bit. Para ejecutar ocho operaciones matemáticas y una búsqueda en una textura se utilice un solo ciclo de reloj.

- Debido a que las GPUs disponen de una mayor cantidad de *Fragment Processors* que *Vertex Processor* para procesamiento del *pipeline*, se tiene un mayor poder de cálculo en esta unidad.

Como resultado del proceso de este bloque se obtiene el valor que designa el color final de cada píxel, los cuales son almacenados en el *framebuffer*, la cual es una unidad intermedia en donde se almacenan termporalmente los datos, para luego ser desplegados en la pantalla [15, 20].

La descripción anterior queda algo obsoleta con la aparición del modelo unificado de shaders, lo que se observa en los modelos de GPUs NVIDIA serie 8 y superiores, el cual utiliza un mismo conjunto de intrucciones en todos los tipos de *shaders* (*Vertex y Pixe*l), guardando ciertas diferencias, por ejemplo sólo el *pixel shader* puede leer implícitamente coordenadas de texturas, en donde se caracteriza por que todos los *shaders* tienen las mismas capacidades, es decir, pueden leer desde texturas, buffer de datos y realizar el mismo conjunto de intrucciones aritméticas.

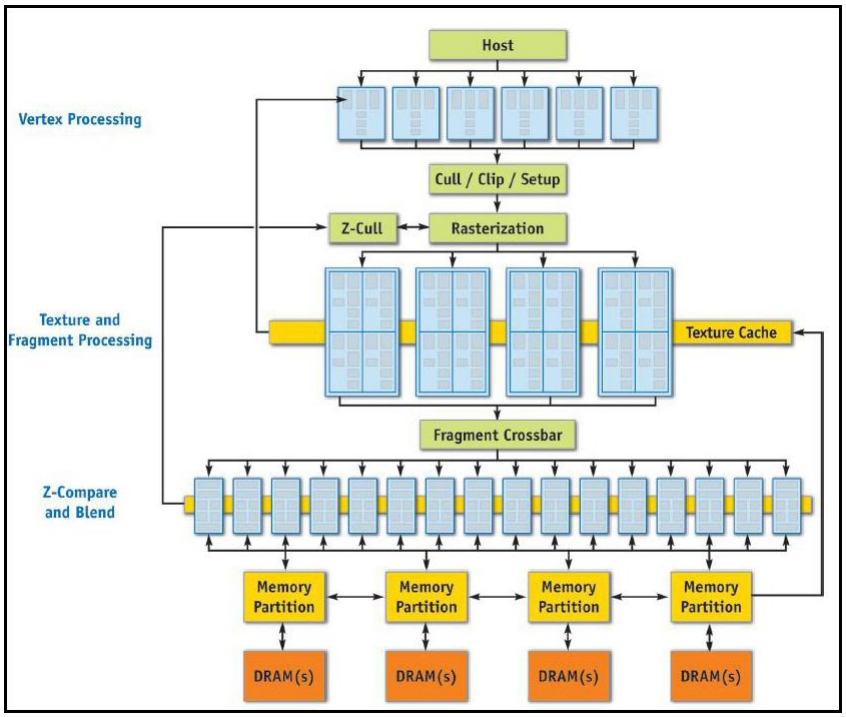



Figura 2.1.3: Detalle de la arquitectura de una GPU, NVIDIA Serie 6000 [15].
En la figura 2.1.3 se muestra la arquitectura presente en la GeForce serie 6000 de Nvidia, la cual es muy similiar a la Serie 7 salvo la variación en la cantidad de *Fragment Processor* y *Vertex Processor. [20]*

El modelo Unificado permite un uso más flexible de la funcionalidades del hardware gráfico. Por ejemplo, en una situación con una gran carga de geometría el sistema puede asignar más unidades para realizar los calculos en el Programa de Vértices y en el Programa de geometría. En los casos con menos carga de trabajo en el Programa de Vértices y una mayor carga para el Programa de Pixeles, más unidades de la GPU podrían destinarse para ejecutar el Programa de Pixeles más eficientemente.

Los sistemas que incluyen esta arquitectura unificada se encuentran en la serie GeForce 8 y GeForce 9 de Nvidia, en las series Radeon 2000 HD y HD Radeon 3000 de ATI, en la serie GMA X3000 de Intel y en la GPU que utiliza la Xbox 360.

#### 2.1.2.3. Memorias de las GPU

Dependiendo si la tarjeta gráfica está integrada a la placa madre (en donde se observa un bajo rendimiento) o no, utilizará la memoria RAM (*Random Access Memory*) propia del computador o

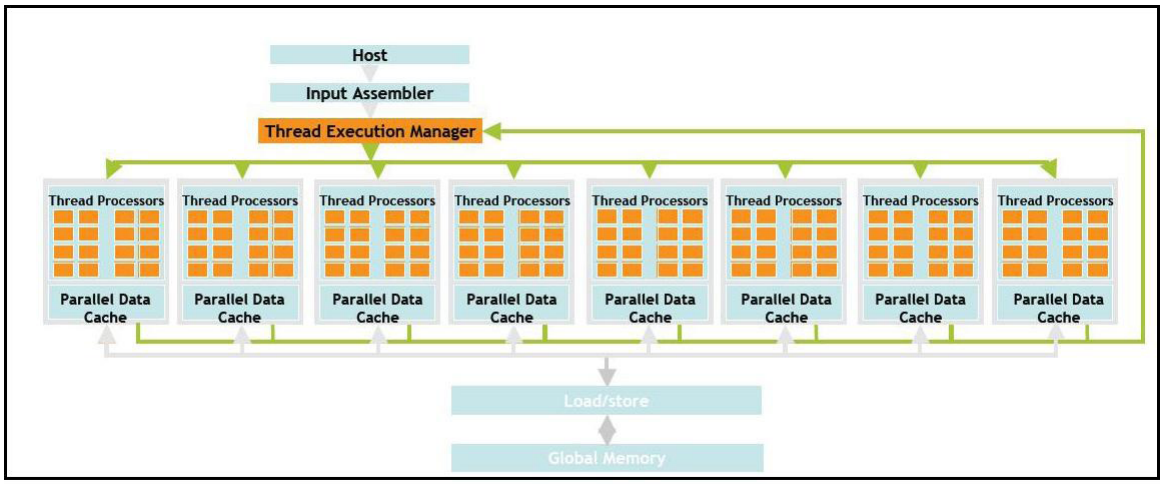



Figura 2.1.4: Arquitectura Unificada

De la Figura 2.1.4 se observa que hay una separación visible de los distintos procesadores dentro de la GPU, como se puede observar en la figura 2.1.3 en donde se identifican claramente los Procesadores de Vértices y Fragmentos [17].

dispondrá de una propia dependiendo del caso. Dicha memoria es la memoria de vídeo o VRAM. Su tamaño oscila entre 128 MB y 1 GB. La memoria empleada en las tarjetas gráficas está basada en la tecnología DDR (*Double Data Rate*), destacando DDR2, GDDR3 (*Graphics Double Data Rate*, Version 3) y GDDR4 (*Graphics Double Data Rate*, Version 4). La frecuencia de reloj de la memoria se encuentra entre 400 MHz y 1,8 GHz [17].

Las memorias en una GPU destacan por su rapidez, y tienen un papel relevante a la hora de almacenar los resultados intermedios de las operaciones y las texturas que se utilicen. Inicialmente, a la GPU le llega la información de la CPU en forma de vértices. El primer tratamiento que reciben estos vértices se realiza en el *Vertex Shader*, como se mencionó anteriormente. Aquí se realizan transformaciones como la rotación o el movimiento de las figuras. Tras esto, los vértices se transforman en pixeles mediante el proceso de rasterización. Estas etapas no poseen una carga relevante para la GPU. Donde sí se encuentra el principal cuello de botella del chip gráfico es en el siguiente paso: el *Pixel Shader*. Aquí se realizan las transformaciones referentes a los píxeles, tales como la aplicación de texturas.

El modelo de memoria de las GPU es sumamente restrictivo:

- Asigna la memoria disponible sólo antes de empezar con el programa.
- El acceso a memoria es limitado durante la ejecución

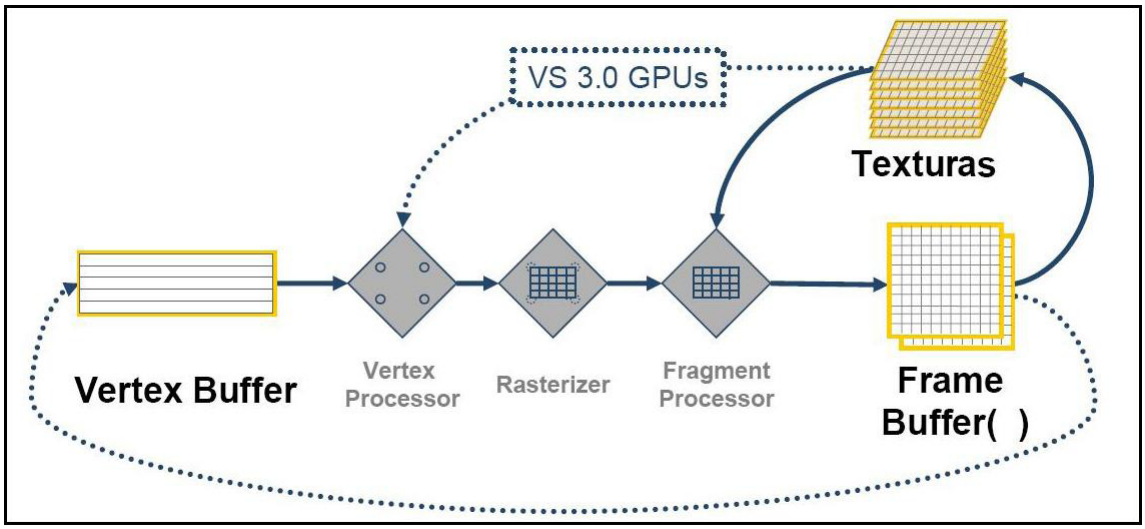



Figura 2.1.5: Estructura de Acceso Memoria en una GPU [12].

- los registros son de lectura y escritura.
- no posee memoria local
- la memoria global sólo puede ser leída durante la ejecución del programa y escrita sólo al terminar la ejecución.
- no posee acceso al disco

Los datos en una GPU son almacenados principalmente en 3 partes:

- *Vertex Buffer*, almacena los datos de los vértices y son los datos requeridos para iniciar el pipeline, dentro de la GPU sólo el *Programa de Vértices* puede tener acceso a esta memoria. Dentro de las limitaciones que presentes está el que no se puede leer aleatoriamente y los programas realizados por el procesador de fragmentos no tienen acceso.
- *Frame Buffer*, sólo puede ser escrito por la GPU, específicamente por el *Fragment Processor*.
- Texturas, donde se almacenan los datos de los pixeles, ya sea cargados desde la CPU a GPU o internamente como resultado del *Fragment Program* por la GPU. Además pueden ser accesados por la CPU y presenta la posibilidad de ser leídos por el *Vertex Processor* y por el *Pixel Processor*.

### 2.1.3. Diferencias entre GPU y CPU

La gran diferencia que se observa entre las CPUs y las GPUs es principalmente el objetivo con que se desarrollaron. Las CPU estan diseñadas para fines generales, los programas que se implementan en una CPU tienen bajo paralelismo y complejos requisitos de control. En cierto modo el diseño de la CPU hace que tenga un bajo rendimiento en lo que concierne a realizar las operaciones que ocurren durante el pipeline y otras operaciones de similares características.

La programación en la CPU, por lo general corresponden a modelos seriales los que se deben implementar de forma eficiente. La CPU procesa sólo una parte de los datos en un instante y no aprovecha el posible paralelismo aplicable, a pesar de que las CPU han integrado a su conjunto de instrucciones, instrucciones como Intel SSE (*Streaming SIMD Extensions*) para permitir la ejecución en paralelo de algunos procesos no logra llegar al grado de paralelismo presnente en una GPU.

Una de las razones por que en las CPU es menos frecuente el nivel de paralelismo, es debido a decisiones de diseño que al dedican más transistores para controlar el hardware.Los programas de la CPU tienen requerimientos de control más complejos que en el caso de las GPUs, por lo que gran parte de los transistores presentes en la CPU son asignados a esa tarea, lo que provoca que sólo una parte de la CPU este realmente dedicada para el cálculo.

El objetivo de las CPUs son los programas de propósito general, los cuales no contienen por lo general hardware especializado para funciones en particulares, en cambio la GPU, utiliza un hardware especial para tareas particulares, lo cual es más eficiente que la solución que puede dar un dispositivo de propósito general.

El sistema de memoria de la CPU está optimizado para la mínima latencia en lugar del máximo rendimiento que se observa en el sistema de memoria de la GPU. A falta de paralelismo, la CPU debe retornar las referencias de memoria lo más rápido que sea posible para poder continuar. Por ello, los sistemas de memoria de la CPU contienen varios niveles de memoria caché (que constituyen una parte importante de los transistores del chip) para reducir al mínimo este tiempo de latencia. Sin embargo, la utilización de memoria caché es ineficiente para muchos tipos de datos, incluyendo los datos y entradas gráficas a las que se accede sólo una vez. Para el pipeline gráfico, maximizar el rendimiento de todos los elementos en lugar de minimizar la latencia de cada uno de los elementos resulta en una mejor utilización del sistema de memoria y un mayor rendimiento general de aplicación.

Pero la potencia de las GPU y su dramático ritmo de desarrollo reciente se deben a dos factores diferentes. El primer factor es la alta especialización de las GPU, ya que al desarrollar una sola tarea, es posible realizar un diseño acorde para llevar a cabo esa tarea más eficientemente.

Por otro lado, muchas aplicaciones gráficas conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo (vértices y píxeles) completamente independientes. Por tanto, es una buena estrategia usar la fuerza bruta en las GPU para completar más cálculos en el mismo tiempo. Los modelos actuales de GPU suelen tener una media docena de procesadores de vértices (que ejecutan *vertex shaders*), y hasta dos o tres veces más procesadores de fragmentos o píxeles (que ejecutan *fragment shaders*). De este modo, una frecuencia de reloj de unos 500-600 [MHz] (el estándar hoy en día en las GPU de más potencia), muy baja en comparación con lo ofrecido por las CPU (3.8-4 [GHz] en los modelos más potentes), se traduce en una potencia de cálculo mucho mayor gracias a su arquitectura en paralelo. Una de las mayores diferencias con la CPU estriba en su arquitectura, mencionada anteriormente. A diferencia de la CPU, que tiene una arquitectura Eckert-Mauchly, la GPU se basa en el Modelo Circulante [14]. Este último modelo facilita el procesamiento en paralelo, y la gran segmentación que posee la GPU para sus tareas.

### 2.1.4. Analogías con la CPU

Al trabajar con una GPU se realizan una serie de analogías con el trabajo sobre una CPU [4], las que hacen más abordable la programación de estos dispositivos.

Para trabajar en la GPU se consideraran las siguientes analogías:

- Datos

La primera analogía que se puede realizar es decir que los Arreglos utilizados en la CPU corresponden a las Texturas utilizadas en la GPU. Los arreglos son el diseño de dato nativo dentro de una CPU, los cuales son de relativamente fácil acceso. Para la GPU la forma básica de representación de los datos corresponde a las texturas, es decir un diseño bidimensional que puede ser visto como matrices. Las texturas (matrices en la GPU) presentan un límite físico a la cantidad de datos que se pueden cargar. En la actualidad el límite varía según el modelo de la GPU que se tiene estando dentro del rango de 2048 a 4096 datos por dimensión.

Dentro de una CPU se hace referencia a los índices de una matriz en cambio dentro de la GPU uno se refiere a las coordenadas de texturas para acceder a los valores almacenados. Generalmente

las GPU trabajan con tetrapletas de datos simultáneamente (ésto corresponde a la utilización de los cuatro canales de colores RGBA).

- Kernels presentes en la CPU son análogos a los *Shader* de la GPU

Los kernels en la CPU corresponden a las instrucciones cíclicas sobre los datos que están guardados en arreglos, en la GPU se escriben instrucciones similares dentro de un Programa de Pixeles las cuales son aplicadas a las texturas, la cantidad de paralelismo corresponde al número de Procesadores de Fragmentos que posea la GPU.

- Feedback

En el caso de la CPU se puede acceder a la memoria para lectura y escritura cuantas veces sea necesaria durante la ejecución del programa, en cambio en GPU se puede acceder a los datos sólo una vez que ya se haya terminado la ejecución de los *shaders*, por ello es necesario utilizan texturas auxiliares en donde almacenar los datos y realizar la técnica denominada "ping pong" [4], la cual consiste en ir alternando la ubicación de la memoria en cada una de las iteraciones.

### 2.1.5. Lenguajes de programación

Antes de la aparición de lenguajes de alto nivel para la programación de *shaders*, la programación de las GPU se debía realizar directamente en lenguaje ensamblador. Dado que la programación en lenguaje ensamblador puede llegar a ser muy complicada surgió la necesidad de contar con lenguajes de alto nivel que permitiera una programación más amigable.

Dentro de los lenguajes de programación utilizados en la GPU se encuentran:

#### 2.1.5.1. HLSL (*High Level Shading Language*)

HLSL apareció en el año 2002, creado por la empresa Microsoft y por algunos fabricantes de chips gráficos. Con este lenguaje los programas ganaron legibilidad, lo que se tradujo en una menor cantidad de errores, mejor depuración y se hicieron más fáciles de mantener y limpiar. Uno de los principales ventajas que introdujo este lenguaje fue elevar el lenguaje de programación a medio - alto nivel, evitando así programar en lenguaje ensamblador.

#### 2.1.5.2. Cg (*C for Graphics*)

Este lenguaje fue desarrollado por NVIDIA a finales del 2002 en colaboración con Microsoft para la programación de *pixel* y *vertex shader*. Cg está basado en el lenguaje de programación C, y aunque comparten la misma sintaxis, algunas características de C fueron modificadas y los nuevos tipos de datos se añadieron para hacer Cg más adecuado para la programación de GPU. Cabe destacar que este lenguaje es apropiado sólo para la programación de las GPUs y no constituye un lenguaje de programación general [20].

#### 2.1.5.3. GLSL (*OpenGL Shading Language*)

El lenguaje GLSL se origina a finales del año 2003, tras una evolución en paralelo con Cg, tomando conceptos bases del lenguaje C. El GLSL se basa en ANSI C, y muchos de los rasgos se han mantenido, excepto cuando entran en conflicto con el cumplimiento o la facilidad de aplicación. C se ha ampliado con la matriz de vectores y tipos para que sea más conciso paralas operaciones típicas llevadas a cabo en gráficos 3D. Algunos mecanismos de C + + también se han tomado en cuenta, tales como la sobrecarga de funciones basadas en argumento de tipos, y la capacidad de declarar todas las variables donde primero se necesita en lugar de a principios de bloques [9].

#### 2.1.5.4. Otros lenguajes de programación

Los lenguajes anteriormente mencionados corresponden a los lenguajes más utilizados para la programación, pero existen lenguajes de programación que han sido desarrollados dentro de un ambiente universitario no comercial , orientado a la experimentación e investigación.

- Sh

Lenguaje de meta programación de *shader*, desarrollado por la universidad de Waterloo en Canadá. Está orientado tanto a GPU como procesadores no puramente *streaming* como los basado en la arquitectura *Cell*. Este lenguaje pretende dar una mayor abstracción respecto a la arquitectura, pero su implementación actual (como librería para lenguajes orientados a objetos) no es capaz de obtener el mismo rendimiento que otras opciones de programación de *shaders* más cercanas al hardware [7].

- BrookGPU

Es un lenguaje alternativo desarrollado por la Universidad de Stanford. Está orientado a realizar aplicaciones de tipo general que aprovechen la potencia de cálculo de la GPU. Se podría considerar una variante al lenguaje C, necesita un API gráfico (OpenGL o DirectX) para cargar sus programas en los procesadores de la GPU [6].

Cabe mencionar la aparición de nuevos lenguajes dado el avance de la tecnología y de la arquitectura de las GPU, siendo uno de los más destacados el lenguaje CUDA.

- CUDA (Compute Unified Device Architecture)

CUDA es un compilador y conjunto de herramientas de desarrollo que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos para ejecutar en GPUs. CUDA fue desarrollado por NVIDIA y esta diseñado para trabajar en GPUs que presentan la Arquitectura Unificada, es decir, las GPUs posteriores a las Serie 8000 incluida.

Esta tecnología permite que los programadores escriban software para resolver problemas computacionales complejos en una fracción del tiempo utilizando el poder de procesamiento paralelo de múltiples núcleos de la GPU. De hecho se usa ya para acelerar aplicaciones, desde codificación de audio y video, exploración de gas y petróleo, hasta imágenes médicas e investigación científica [16].

Algunas de las características de CUDA son:

Lenguaje C estándar para desarrollo de aplicación paralelo en la GPU.

Bibliotecas numéricas estándar para FFT (Fast Fourier Transform) y BLAS (Basic Linear Algebra Subroutines).

Driver CUDA dedicado para computación con un camino rápido de transferencia de datos entre la GPU y la CPU.

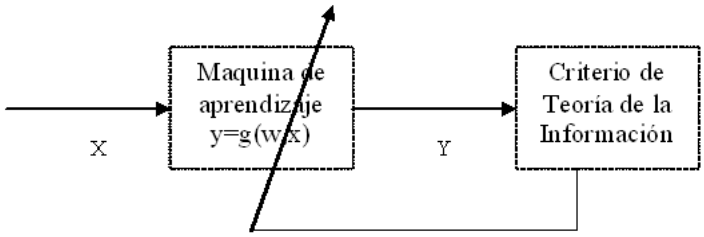
El driver CUDA inter-opera con los drivers gráficos OpenGL y DirectX.

Compatible con los sistemas operativos Linux de 32/64 bits y Windows XP de 32/64 bits.

## 2.2. Teoría de la Información

En la actualidad las comunicaciones juegan un papel muy importante en la vida cotidiana. El gran desarrollo de la tecnología ha permitido que millones de personas puedan navegar por internet, descargar recursos audiovisuales y estar en contacto con otras personas . Esto genera grandes flujos de datos e información. Por otro lado grandes cantidades de datos son generados en todo orden de

Figura 2.2.1: Esquema de Entrenamiento




cosas como fábricas, servicios gubernamentales, laboratorios de investigación etc, los cuales por si solos no dicen nada pero que pueden contener una gran cantidad de información que requiere ser procesada.

Uno de los principales problemas que se observan en las tecnologías actuales es la gran cantidad de datos que se generan desde distintos puntos, ya sea, por comunidades o por individuos.

La información tiene muchos significados en el punto de vista del lenguaje, pero se puede encontrar una caracterización específica de lo que es la información dentro del marco de la teoría de la información. En ella, la información es tratada como datos que entre si están relacionados a través de una función de probabilidad. ITL trabaja sobre este postulado aprovechando las ventajas que ofrecen las funciones de probabilidad para extraer medidas de información o conocimiento (entropía, información mutua) que son utilizadas para la interpretación de los datos. Esta característica de ITL la hace muy apropiada en la utilización de algoritmos que modelan ciertos sistemas con la finalidad de predecir futuros comportamientos de éste. En términos matemáticos se tiene que los datos son mapeados desde el espacio original $R^K$, con $X \in R^K$, a otro espacio de interés $R^M$, con $Y \in R^M$ (favorable para la manipulación e interpretación) a través de una función $g : R^K \longrightarrow R^M$, la cual tiene como condición mantener toda la información desde el espacio original al espacio de interés. Esto se representa mediante la siguiente ecuación:

$$Y = g\left(X, W\right) \tag{2.2.1}$$

donde $W$ son los parámetros necesarios para ajustar adecuadamente la función $g$. La utilización de ITL es precisamente traspasar toda la información de los datos a través del conjunto de parámetros de la función $g$. En un esquema de entrenamiento utilizando el descenso por gradiente se tiene:

Mediante la obtención de lo que se denomina entropía de los datos es posible obtener una

cuantificación de la cantidad de información obtenida en un conjunto de datos. Shannon [22] define la entropía de una función de distribución de probabilidad $P$ como:

$$H_s(P) = \sum_{k=1}^{N} p_k \log\left(\frac{1}{p_k}\right) \qquad \sum_{k=1}^{N} p_k = 1, \quad p_k \geq 0 \tag{2.2.2}$$

La entropía corresponde a una cantidad promedio de información contenida en una observación de una variable aleatoria que toma valores $x_1, x_2, \ldots, x_n$ con probabilidades $p_k = P(x = x_k)$ con $k = 1, 2, \ldots, N$. La entropía mide la cantidad promedio de información contenida por un evento $x$, o alternativamente, la cantidad de información que falta cuando sólo la distribución a *priori* está dada.

Una definición alternatica puede obtener la entropía de Renyi [19]. La entropía de Renyi de orden $\alpha$se define como:

$$H_\alpha = \frac{1}{1-\alpha} \log\left(\sum_{k=1}^{N} p_k^\alpha\right) \qquad \alpha > 0, \, \alpha \neq 1 \tag{2.2.3}$$

Al ser esta una versión más general de la entropía, la entropía de Shannon se puede obtener bajo la siguiente relación:

$$\lim_{\alpha \to 1} H_\alpha = H_s \tag{2.2.4}$$

La versión continua de la entropía de Renyi sale directamente de la expresión anterior al integrar sobre el espacio de la FDP (Función Densidad Probabilidad). En especial es interesante observar el caso con $\alpha$= 2, que equivale a la entropía cuadrática de Renyi:

$$\begin{cases} H_\alpha(Y) = \frac{1}{1-\alpha} \log\left(\int (f_Y(z))^\alpha \, dz\right) \\ H_2(Y) = -\log\left(\int (f_Y(z))^2 \, dz\right) \end{cases} \tag{2.2.5}$$

Una observación importante es que la entropía de Renyi posee el mismo óptimo global que la entropía de Shannon en términos de maximización, sin embargo la ventaja está en que la versión cuadrática permite estimar de una forma mucho más simple la entropía a partir de muestras de datos sin la necesidad de imponer o asumir una FDP a priori.

Para evitar asumir una FDP se utiliza un estimador basado en kernels, en particular en kernels gaussianos. Este estimador es conocido como ventana de Parzen, donde dado un conjunto de datos

$Y = \{y_1, ..., y_N\}$ $y_i \in \Re^d$ la FDP puede ser estimada como:

$$\begin{cases} \hat{f_Y}(z, \{y\}) = \frac{1}{N} \sum_{i=1}^{N} G\left(z - y_i, \sigma^2 I\right) \\ G(z, \Sigma) = \frac{1}{(2\pi)^{M/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2} z^T \Sigma^{-1} z\right) \end{cases} \tag{2.2.6}$$

donde $G(.,.)$ es el kernel gausiano y $\Sigma = \sigma^2 I$ es la matriz de covarianzas. Cuando se combinan la definición de entropía de Renyi con el estimador de Parzen, se puede conseguir un estimador de entropía basado en las muestras discretas de datos $\{y\}$ como:

$$\begin{cases} H(\{y\}) = H_2(Y \mid \{y\}) = -\log\left(\int f_Y(z)^2 dz\right) = -\log V(\{y\}) \\ V(\{y\}) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} \int G\left(z - y_i, \sigma^2 I\right) G\left(z - y_j, \sigma^2 I\right) dz = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} G(y_i - y_j, 2\sigma^2 I) \end{cases} \tag{2.2.7}$$

La última igualdad se justifica utilizando la siguiente propiedad de las funciones Gausianas. Sea $G(z, \Sigma) = \frac{1}{(2\pi)^{M/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2} z^T \Sigma^{-1} z\right)$ el kernel gaussiano en el espacio M-dimensional, donde $\Sigma$ es la matriz de covarianza, $z \in \Re^M$. Sean $y_i \in \Re^M$ y $y_j \in \Re^M$ dos muestras de datos en el espacio, $\Sigma_1$y $\Sigma_2$ las matrices de covarianza para dos kernel gaussianos en el espacio. Se puede demostrar [3] que se cumple la siguiente relación:

$$\int G(z - y_i, \Sigma_1) G(z - y_j, \Sigma_2) dz = G\left((y_i - y_j), (\Sigma_1 + \Sigma_2)\right) \tag{2.2.8}$$

La información mutua mide la cantidad de información entre dos variables aleatorias. La información mutua a la salida de un mapeo puede ser calculada como una diferencia de entropías de Shannon $I(x, y) = H(y) - H(y/x)$. La estimación de la expresión $I(x, y)$ , es un caso particular de la divergencia de Kullback-Leibler (KL) [19], la cual estima la distancia entre 2 funciones de densidad de probabilidad $f(x)$ y $g(x)$ mediante:

$$K(f, g) = \int f(x) \log \frac{f(x)}{g(x)} dx \tag{2.2.9}$$

Donde la entropía de Shannon es utilizada de forma implícita. Al igual que en la entropía se obtiene la entropía de Renyi de orden $\alpha$, para la información mutua se define la medida de

divergencia de Renyi de orden $\alpha$ para dos funciones de densidad de probabilidad $f(x)$ y $g(x)$, obteniéndose:

$$H_{R\alpha}(f,g) = \frac{1}{1-\alpha}\int \log\left(\frac{f(x)^{\alpha}}{g(x)^{\alpha}}\right)dx \tag{2.2.10}$$

Y en el límite cuando $\alpha \to 1$, se tiene tiene la divergencia de KL:

$$\lim_{\alpha\to 1} H_{R\alpha}(f,g) = K(f,g) \tag{2.2.11}$$

La obtención de la entropía y la divergencia de Renyi, existen una serie de métodos tanto paramétricos como no paramétricos.

## 2.3. Estimación de Funciones de Distribución de Probabilidad

Como se vió en la sección 2.2 Teoría de la Información trabaja con la FDP, y por lo tanto es requisito conocerla para su aplicación, por ello es fundamental tener algún método para determinarla. Dentro de los principales métodos de estimación paramétrica de funciones de densidad de probabilidad se encuentran: el método de los Momentos y el método de Máxima Verosimilitud y dentro de los métodos no paramétricos se encuentra el Estimador de Parzen:

### 2.3.1. Método de los Momentos

Los momentos [2] permiten caracterizar una función de probabilidad. Luego si dos variables aleatorias poseen los mismos momentos, se concluye que dichas variables tienen o siguen la misma función de densidad de probabilidad. Por lo tanto se puede emplear para estimar los respectivos parámetros que definen las funciones.

El método consiste en igualar los momentos apropiados de la distribución de una población con los correspondientes momentos muestrales, para estimar un parámetro desconocido de la distribución.

### 2.3.2. Método de Máxima Verosimilitud

El método de estimación [2] por máxima verosimilitud, selecciona como estimador a aquel valor del parámetro que tienen la propiedad de maximizar el valor de la probabilidad de la muestra aleatoria observada, es decir, el método consiste en encontrar el valor del parámetro que maximice la función de verosimilitud.

Luego se define como:

Sea $X_1, X_2, ..., X_N$ una muestra aleatorea de una distribución de densidad de probabilidad $f(x, \theta)$ y $L(x_1, x_2, \ldots, x_N)$ la verosimilitud de la muestra en funcion de $\theta$. Si $t = u(x_1, x_2, \ldots, x_N)$es el valor de para el cual el valor de la función de verosimilitud es máxima, entonces $T = u(X, X_2, \ldots, X_N)$ es el estimador de máxima verosimilitud de y $t$ .

Los métodos mencionados se basan en el conocimiento previo de la FDP que se quiere aproximar que en algunos casos no es posible conocer con antelación. Un método que permite obtener la funciones de probabilidad mediante la estimación no paramétrica corresponde al estimador de Parzen.

### 2.3.3. Estimador de Parzen

El estimador de Parzen está dentro de lo que se denomina estimación de densidad de kernel (KDE, Kernel Density Estimation). En este método la FDP es estimada mediante la suma de funciones de kernel, las cuales típicamente son funciones gaussianas que se encuentran centradas sobre los puntos que se quiere determinar su FDP. Es necesario definir un ancho de banda asociado a la función de kernel para manejar la suavidad de la estimación.

Este método para la estimación de FDP es sumamente útil para manipular las funciones presentadas por ITL, se puede observar que al utilizar el criterio de la entropía de Renyi de orden 2 la estimación de Parzen se integra fácilmente permitiendo un mejor trabajo matemático, al utilizar la propiedad 2.2.

El estimador de Parzen [18] se define:

Si $X_1, X_2, ..., X_N$ son variables aleatoreas independientes e identicamente distribuidas (IID), entonces la densidad de kernel de la FDP está dada por:

$$\hat{f}_k(x) = \frac{1}{Nh} \sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right) \tag{2.3.1}$$

donde $K$ es algún kernel y $h$ es el ancho de banda o también conocido como la varianza.

Usualmente como kernel se considera la función gaussiana con media cero y varianza unitaria.

$$K(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) \tag{2.3.2}$$

Varios métodos han sido propuestos para la estimación de densidad de kernel de forma eficiente. Entre los distintos métodos que se han empleado se encuentra la estimación basada en los k-vecinos más cercanos y los basados en la transformada rápida de fourier para la estimación de la densidad en grillas de datos.

Recientemente métodos como la Transformada de Gauss y *Fast Multipole Method* (FMM) han proporcionado una alternativa más eficiente para la estimación de la FDP reduciendo la cantidad de cálculos [1, 26].

En particular como la estimación de Parzen se basa en suma de N gaussianas se tienen métodos como la Transformada Rápida de Gauss (FGT) y la Transformada Rápida de Gauss Mejorada (IFGT) que reducen el cálculo de un orden O(NM) a un orden O(N+M), en donde N corresponde a la cantidad de variables aleatorias y M corresponde a la cantidad de centros donde se estimara la FDP. Cabe destacar que los métodos basados en series de Taylor como lo son la *Fast Gauss Transform* (FGT) y la IFGT, poseen una buena aceleración para anchos de bandas $h$ grandes.

## 2.4. Transformada de Gauss

La transformada de Gauss posee una estructura similar a la presentada por la estimación de Parzen y se define como sigue:

$$G_\delta f(y) = \int_\Gamma \exp\left(-\frac{\|y - x_i\|^2}{h^2}\right) f(x)\, dx \qquad (h > 0) \tag{2.4.1}$$

donde la función se define en $\Gamma \subset \Re^d$. Para propósitos numéricos se utiliza la versión discreta de la transformada de Gauss. Dado los valores de $f$ como un conjunto de $N$ puntos $x_i \in \Re^d$, se puede expresar de la forma que sigue:

$$G(y) = \sum_{i=1}^{N} q_i \exp\left(-\frac{\|y - x_i\|^2}{h^2}\right) \tag{2.4.2}$$

donde los coeficientes $q_i$ son dependientes de los valores de $f(x_i)$. Para evaluar la expresion anterior sobre $M$ puntos $(y_1, y_2, \ldots, y_M)$ la expresión queda:

$$G(y_j) = \sum_{i=1}^{N} q_i \exp\left(-\frac{\|y_j - x_i\|^2}{h^2}\right) \tag{2.4.3}$$

Al observar la ecuación 2.4.3 se ve que para obtener los $M$ $G(y_i)$ es necesario realizar $MN$ operaciones.

### 2.4.1. Transformada Rápida de Gauss (FGT)

La Transformada Rápida de Gauss (Fast Gauss Transform), es una variante de Fast Multipole Method (FMM) [23], el cual fue desarrollado para realizar sumas rápidas de campos potenciales generados por una gran cantidad de partículas, las cuales se encuentran sobre potenciales electrostáticos o gravitacionales en 2 o 3 dimensiones.

El primer paso de la FGT es la subdivisión espacial de un hipercubo unitario dentro de $N_{side}^d$ "cajas" de lado $\sqrt{2}rh$, en donde $r$ representa el largo de la "caja" y $h$ el ancho de banda o varianza. Se sugiere escoger un $r \leq 0{,}5$ tal que $N_{side} = \frac{1}{\sqrt{2}rh}$sea un entero. Los conjuntos de datos $\{x_i\}$ e $\{y_j\}$ son asignados a "cajas" diferentes, dados el conjunto de datos $\{x_i\}$ en una "caja" y los valores de $\{y_j\}$ en una "caja" vecina, el cálculo es realizado usando uno de los siguientes métodos dependiendo del número de $\{x_i\}$ e $\{y_j\}$ presentes en las "cajas".

La FGT presenta un orden $O((N+M)p^d)$ donde N y M corresponden a la cantidad de datos $\{x_i\}$e $\{y_j\}$respectivamente y $p$ indica hasta que término de la expansión de Hermite se considerará para el cálculo, $d$ corresponde a la dimensionalidad de los datos. La FGT [23] realiza 2 expansiones diferentes para la gaussiana, una dada por la serie de Hermite la que entrega una aproximación para campo lejano y otra que está dado por la serie de Taylor, la que resulta de intercambiar $\{x_i\}$ e $\{y_j\}$ en la expansión de Hermite, la que se realiza para el campo cercano. Se observa que al aumentar la dimensionalidad $d$, el orden del problema aumenta potencialmente lo que lo hace costoso computacionalmente para dimensiones altas.

Un inconveniente que presenta la FGT, es el uso de una estructura de datos en forma de "cajas", es decir, la FGT subdivide el espacio de los datos utilizando una grilla, lo que en altas dimensiones puede ser un problema, en particular si los datos tienen algun tipo característica en común formando cúmulos.

### 2.4.2. Transformada Rápida de Gauss Mejorada (IFGT)

La IFGT, trata de eliminar los defectos presentes en la FGT, los cuales pueden ser atribuidos por la aplicación de FMM, dado que FMM fue desarrollado para funciones potenciales singulares, que para funciones gaussianas están lejos de ser singulares y la forma de dividir el espacio en "cajas", lo que para altas dimensiones es costoso. Luego para evitar la descomposición realizada en la FGT, la IFGT realiza una expansión distinta.

La IFGT utiliza la serie de Taylor para descomponer los Kernels Gaussianos presentes, asumiendo que se poseen $N$ elementos $\{x_i\}$centrados en $x_*$ y una cantidad de $M$ puntos $\{y_i\}$, el término exponencial 2.4.3, puede ser escrito como

$$\exp\left(-\frac{\|y_j - x_i\|^2}{h^2}\right) = \exp\left(-\frac{\|y_j - x_*\|^2}{h^2}\right)\exp\left(-\frac{\|x_i - x_*\|^2}{h^2}\right)\exp\left(-\frac{2\,(x_i - x_*)\,(y_j - x_*)}{h^2}\right) \tag{2.4.4}$$

De la expresión 2.4.4 se observa que en el tercer término de la derecha hay términos de ambos conjuntos de datos $\{x_i\}$e $\{y_i\}$. La idea principal del algoritmo es reemplazar la expresión este término en una expansión en Serie de Taylor quedando la expresión como sigue:

$$\exp\left(-\frac{2\,(x_i - x_*)\,(y_j - x_*)}{h^2}\right) = \sum_{n=0}^{p-1}\frac{2^n}{n!}\left[\left(\frac{x_i - x_*}{h}\right)\left(\frac{y_j - x_*}{h}\right)\right]^n + error_p \tag{2.4.5}$$

utilizando la notación multi-índice, la expansión en serie de Taylor puede ser reescrita como sigue:

$$\exp\left(-\frac{2\,(x_i - x_*)\,(y_j - x_*)}{h^2}\right) = \sum_{|\alpha|\leq p-1}\frac{2^\alpha}{\alpha!}\left(\frac{x_i - x_*}{h}\right)^\alpha\left(\frac{y_j - x_*}{h}\right)^\alpha + error_p \tag{2.4.6}$$

Ignorando el *error* producto de la aproximación de Taylor, la expresión 2.4.3 puede ser aproximada como sigue:

$$\hat{G}\,(y_j) = \sum_{i=1}^{N} q_i \exp\left(-\frac{\|y_j - x_*\|^2}{h^2}\right)\exp\left(-\frac{\|x_i - x_*\|^2}{h^2}\right)\left[\sum_{|\alpha|\leq p-1}\frac{2^\alpha}{\alpha!}\left(\frac{x_i - x_*}{h}\right)^\alpha\left(\frac{y_j - x_*}{h}\right)^\alpha\right] \tag{2.4.7}$$

Reordenando los términos de la expresión 2.4.7 se obtiene:

$$\hat{G}(y_j) = \sum_{|\alpha|\leq p-1} \left[ \frac{2^\alpha}{\alpha!} \sum_{i=1}^{N} q_i \exp\left( -\frac{\|x_i - x_*\|^2}{h^2} \right) \left( \frac{x_i - x_*}{h} \right)^\alpha \right] \exp\left( -\frac{\|x_i - x_*\|^2}{h^2} \right) \left( \frac{y_j - x_*}{h} \right)^\alpha$$

$$= \sum_{\alpha\leq p-1} C_\alpha \exp\left( -\frac{\|x_i - x_*\|^2}{h^2} \right) \left( \frac{y_j - x_*}{h} \right)^\alpha \tag{2.4.8}$$

donde

$$C_\alpha = \left[ \frac{2^\alpha}{\alpha!} \sum_{i=1}^{N} q_i \exp\left( -\frac{\|x_i - x_*\|^2}{h^2} \right) \left( \frac{x_i - x_*}{h} \right)^\alpha \right] \tag{2.4.9}$$

Realizando una partición sobre el espacio de los datos de entrada, se divide $\{x_i\}$ en $K$ cúmulos, $S_k$ con $k = 1, ..., K$ y con $c_k$ el centro respectivo de cada cúmulo $S_k$, utilizando el algoritmo de *clustering* [5]. Luego la expresión 2.4.7 se reescribe como:

$$\hat{G}(y_j) = \sum_{k=1}^{K} \sum_{|\alpha|\leq p-1} C_\alpha^k \exp\left( -\frac{\|y_j - c_k\|^2}{h^2} \right) \left( \frac{y_j - c_k}{h} \right)^\alpha$$

donde

$$C_\alpha^k = \left[ \frac{2^\alpha}{\alpha!} \sum_{x_i \in S_k} q_i \exp\left( -\frac{\|x_i - c_k\|^2}{h^2} \right) \left( \frac{x_i - c_k}{h} \right)^\alpha \right] \tag{2.4.10}$$

Considerando el rápido decaimiento que posee la exponencial e ignorando elementos que se encuentran alejados de cierto radio al centro del cúmulo, se crea un radio de corte para los elementos $\{y_j\}$ tal que $\|y_j - c_k\| > r_y^k$, en donde el radio de corte depende del error deseado para la aproximación. Luego se obtiene que la Transformada de Gauss es estimada mediante:

$$\hat{G}(y_j) = \sum_{\|y_j - c_k\| \leq r_y^k} \sum_{|\alpha|\leq p-1} C_\alpha^k \exp\left( -\frac{\|y_j - c_k\|^2}{h^2} \right) \left( \frac{y_j - c_k}{h} \right)^\alpha \tag{2.4.11}$$

. Luego el algoritmo utilizado por C. Yang, R. Duraiswami and N. Gumerov [1] para calcular la Transformada de Gauss es el siguiente:

**Algoritmo 1** Escoger los parámetros para la IFGT

Definir:$\delta(p,a,b) = \frac{1}{p!}\left(\frac{2ab}{h^2}\right)^p e^{-(a-b)^2}$ $b_*(a,p) = \frac{-a+\sqrt{a^2-2ph^2}}{2}$ $r_{pd} = \binom{p-1+d}{d}$
Escoger el radio de corte: $r = \text{mín}\left(R, h\sqrt{\ln\left(1/\epsilon\right)}\right)$, con $\epsilon$ error deseado, $R$ corresponde al rango de los datos y $h$ varianza o ancho de banda
Escoger número límite de cúmulos $K_{limite} = \frac{20R}{h}$
Para $k = 1...K_{limite}$

- Calcular un valor estimado para radio máximo de los cúmulos como $r_x = k^{-\frac{1}{d}}$
- Calcular un valor estimado para el número de vecinos como $n = \left(\frac{r}{r_x}\right)^d$
- Escoger el $p(k)$tal que $\delta(p = p(k), a = r_x, b = \text{mín}(b_*(r_x, p(k)), r + r_x)) \leq \epsilon$
- Calcular el término constante $c(k) = \log(k) + (1+n) r_{(p(k)-1)d}$

Fin
una vez realizado escoger:
$K = k_*$para el $c(k_*)$es mínimo.
$P_{max} = p(k_*)$

Una vez obtenidos los parámetros iniciales se procede a realizar el cálculo de la IFGT

**Algoritmo 2** IFGT

1. Definir: $\delta(p,a,b) = \frac{1}{p!}\left(\frac{2ab}{h^2}\right)^p e^{-(a-b)^2}$ $b_*(a,p) = \frac{-a+\sqrt{a^2-2ph^2}}{2}$ .
2. Escoger el radio de corte $r$, el número de cúmulos $K$ y el máximo número para la truncar la Serie de Taylor $P_{max}$ usando el algoritmo 1.
3. Dividir los $N$ elementos $\{x_i\}$ dentro de $K$ cúmulos mediante la un algoritmo de clustering, indicando los centros $c_k$ y los radios $r_x^k$ de cada cluster .
4. Para cada cúmulo $S_k$ con centro $c_k$ calcular los coeficientes $C_\alpha^k$

$$C_\alpha^k = \left[\frac{2^\alpha}{\alpha!}\sum_{x_i \in S_k} q_i \exp\left(-\frac{\|x_i - c_k\|^2}{h^2}\right)\left(\frac{x_i - c_k}{h}\right)^\alpha\right] \tag{2.4.12}$$

5. Para cada elemento $\{y_j\}$ evaluar la Transformada Discreta de Gauss

$$\hat{G}(y_j) = \sum_{\|y_j - c_k\| \leq r_y^k} \sum_{|\alpha| \leq p_{max}-1} C_\alpha^k \exp\left(-\frac{\|y_j - c_k\|^2}{h^2}\right)\left(\frac{y_j - c_k}{h}\right)^\alpha \tag{2.4.13}$$

# Capítulo 3

# Implementación

En este capítulo se describe como se aborda la implementación de la Transformada de Gauss y la IFGT sobre la CPU y la GPU y el planteamiento del algoritmo para la implementación en la GPU.

## 3.1. Características del equipo

La implementación del algoritmo se realiza sobre un computador que posee las siguientes características de hardware, las que se indican en la tabla 3.1.1:

Dentro del equipo se cuenta con una GPU que posee las características técnicas presentadas en la tabla 3.1.2

## 3.2. Implementación sobre la CPU

La implementación sobre la CPU se realiza utilizando el lenguaje C y el editor Visual Studio 2005, sobre el cual se realiza la implementación de la Transformada de Gauss y de la IFGT

Tabla 3.1.1: Características generales del Equipo

| Procesador | AMD Athlon 64 X2 3800+ |
|---|---|
| Velocidad del Procesador | 2.0 [GHz] |
| Memoria RAM | 1 [GB] |

Tabla 3.1.2: Características de la GPU

| Modelo | GeForce 7900 GT |
|---|---|
| Bus Interface | PCIe x16 |
| Configuración del Núcleo | 8:24 |
| Memoria de Video | 256 [MB] GDDR3 |
| Soporte de librerías gráficas | OpenGL 2.0 - DirectX 9.0 |

### 3.2.1. Implementación de la Transformada de Gauss

La implementación de la transformada de Gauss en la CPU se realiza de forma directa, es decir, se escribe las líneas de código correspondientes al cálculo de

$$G(y_j) = \sum_{i=1}^{N} q_i \exp\left(-\frac{\|y_j - x_i\|^2}{h^2}\right) \tag{3.2.1}$$

La implementación de la Transformada de Gauss es técnicamente directa, siendo necesario realizar sólo $\sum_{i=1}^{N} q_i \exp\left(-\frac{\|y_j - x_i\|^2}{h^2}\right)$ para un $y_j$ dado y luego repetirlo la cantidad de elementos que posea $\{y_j\}$.

### 3.2.2. Implementación de la IFGT

La implementación de la IFGT se realiza utilizando el algoritmo presentado en la sección anterior propuesto por C. Yang, R. Duraiswami and N. Gumerov [1] y basándose en su implementación para Matlab [21]. La implementación de la IFGT se hace sobre lenguaje C utilizando el editor Visual Studio 2005.

Una vez implementada la Transformada de Gauss y la IFGT se realizan pruebas para comprobar el error de la IFGT con respecto a la Transformada de Gauss y pruebas de aceleración respectivas a modo de comprobar la programación de los algoritmos.

## 3.3. Implementación sobre la GPU

Para la programación de la GPU, se utiliza el editor Visual Studio 2005, el API gráfico OpenGL y los lenguajes de programación C para la correspondiente parte en C y Cg para la parte del algoritmo que se implementa sobre la GPU. La implementación de la transformada de Gauss e

IFGT sobre la GPU se realiza utilizando el formato de Textura GL_LUMINANCE, el cual permite cargar sólo un elemento dentro de un pixel y una precisión de Float 32 bit.

### 3.3.1. Implementación Transformada de Gauss

La implementación sobre la GPU se basa en la utilización de texturas para el manejo de los datos, dado que las texturas representan un arreglo bidimensional o matriz de datos se introducen los elementos $\{x_i\}$ , $\{y_j\}$ y $\{q_i\}$ sobre una textura.

La textura se completa de forma horizontal, es decir, se van introduciendo los valores de los elementos $\{x_i\}$ , $\{y_j\}$ y $\{q_i\}$ hacia el lado completando la textura de la forma que se muestra en la figura 3.3.1. La textura se completa de la siguiente forma: se considera la dimensión de los elementos$\{x_i\}$ e $\{y_j\}$, los cuales se concatenan uno del lado de otro hasta completar la cantidad de columnas que uno desea, una vez llegado al límite se prosigue con la siguiente fila. Para el conjunto de datos $\{q_i\}$, la textura se rellena de igual forma, con la salvedad que $\{q_i\}$ son unidimensionales.

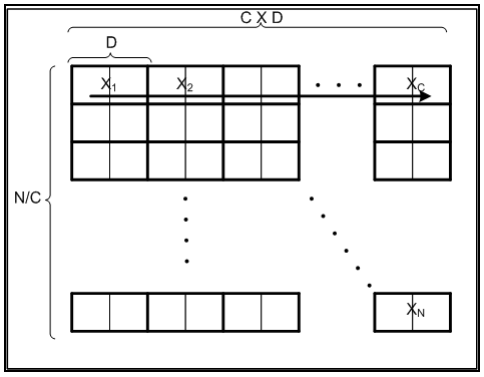



Figura 3.3.1: Ejemplo de relleno de la textura con los elementos $\{x_i\}$

El cálculo de $G(y_j)$ se realiza mediante la implementación de 2 programas en la GPU. El primer programa (Programa 1) calcula lo correspondiente a la expresión $q_i e^{\frac{\|x_i - y_j\|^2}{h^2}}$ para $i = 1, ..., N$ y un segundo programa (Programa 2) entrega lo correspondiente a $\sum_{i=1}^{N} q_i e^{\frac{\|x_i - y_j\|^2}{h^2}}$ para un $y_j$ dado. Luego para calcular la Transformada de Gauss para todos los $\{y_j\}$ se deben repetir los programas tantas veces como cantidad de elementos tenga el conjunto de datos $\{y_j\}$.

Como se mencionó el Programa 1 realiza el cálculo de $q_i e^{\frac{\|x_i - y_j\|^2}{h^2}}$ para un $y_j$, para ello recibe los datos $\{x_i\}$, $\{y_j\}$ y $\{q_i\}$ almacenados en sus respectivas texturas, luego se ejecuta el programa y se entrega como resultado una textura en donde cada pixel de la textura tiene almacenado el valor

correspondiente a un $q_i e^{\frac{\|x_i - y_j\|^2}{h^2}}$, la figura 3.3.2 gráfica lo anterior.

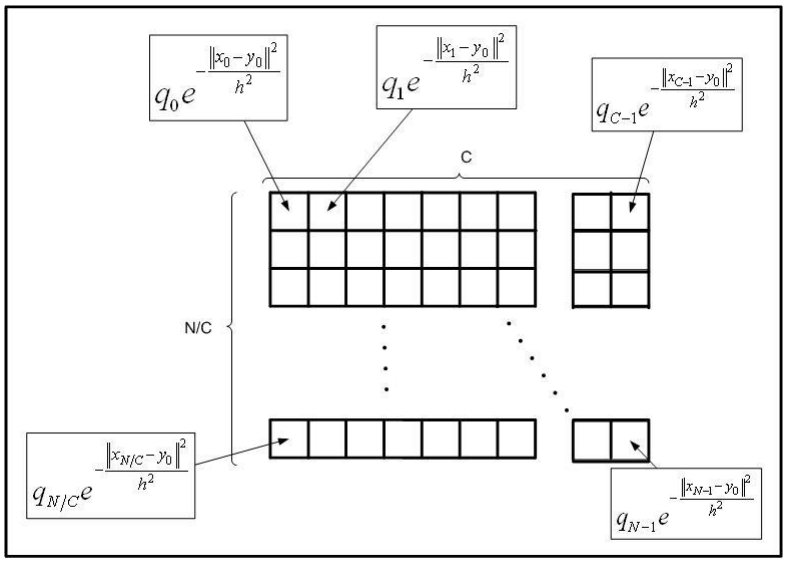



Figura 3.3.2: Textura resultado Programa 1

Luego de obtener la textura presentada en la figura 3.3.2 se procede a calcular $\sum_{i=1}^{N} q_i e^{\frac{\|x_i - y_j\|^2}{h^2}}$, lo que es realizado por el Programa 2. Para obtener el término se requiere sumar todos los elementos (pixeles) de la textura, y para realizarlo se utiliza la técnica denominada reducción.

La reducción consiste en dividir la textura en 4 partes y sumar 4 elementos, y el resultado almacenarlo en una textura de tamaño un cuarto de la original y así sucesivamente hasta obtener sólo un pixel. Por esta razón las texturas que se cargan deben de tener dimensiones de potencia de 2, dado que cada paso de la reducción divide a la mitad los lados de la textura.

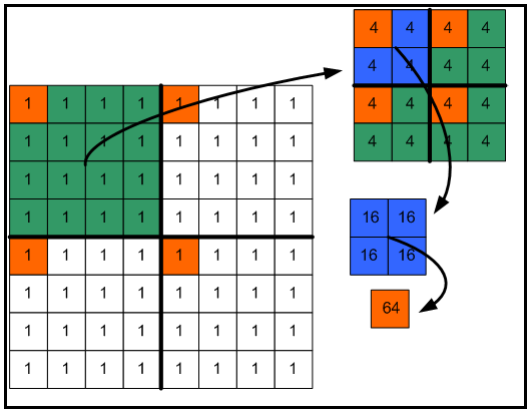



Figura 3.3.3: Técnica de reducción

En la figura 3.3.3 se muestra como se realiza la reducción. Supongamos que se tiene una textura rellena de 1. Como primer paso se toma la textura inicial (en este caso color blanco) y se divide en 4. Luego se realiza la suma siguiendo los pixeles naranjos, hasta recorrer toda la textura y el resultado es guardado en una textura de un cuarto del tamaño original (en el caso de la figura color verde), y para el ejemplo tendrá en cada pixel el valor 4. Luego a esta nueva textura se realiza el mismo procedimiento se divide en cuatro y el resultado se almacena una textura de menor tamaño, hasta llegar a un solo pixel, con el resultado final que para fines del ejemplo es 64. Cabe destacar que para realizar el procedimiento es necesario utilizar 2 texturas, una inicial donde vienen los datos originales y otra auxiliar, luego las texturas se van alternando cada vez que se realiza el proceso.

Con la utilización de la técnica anterior se logra obtener el valor de $\sum_{i=1}^{N} e^{\frac{\|x_i - y_j\|^2}{h^2}}$ para un $y_j$, luego de iterar tantas veces como elementos posee $\{y_j\}$ se obtiene la Transformada de Gauss.

### 3.3.2. Implementación IFGT

Sobre la GPU se implementa la parte de la IFGT que requiere un mayor tiempo de cálculo, que corresponde a:.

$$\hat{G}(y_j) = \sum_{\|y_j - c_k\| < r_y^k} e^{\frac{-\|y_j - c_k\|^2}{h^2}} \sum_{x_i \in S_k} q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[ \frac{(x_i - c_k)}{h} \frac{(y_j - c_k)}{h} \right]^n \tag{3.3.1}$$

Lo que corresponde a la parte del algoritmo de la IFGT a la selección de los parámetros y a el particionamiento de los elementos $\{x_i\}$ se realiza fuera de la GPU por tener un tiempo muy bajo en relación con la expresión 3.3.1.

Al igual que en la implementación de la Transformada de Gauss, en primer lugar se realiza la carga de las texturas con los datos necesarios para el cálculo, partiendo por $\{x_i\}$ , $\{y_j\}$ y $\{q_i\}$. La carga de las texturas sigue el mismo esquema presentado en la sección 3.3.1. Dado que la expresión 3.3.1 incluye términos adicionales para el cálculo, éstos son entregados como texturas a la GPU para ser incluidos en el proceso, dentro de los cuales se incluyen $r_y^k$, un término que representa $x_i \in S_k$, $c_k$ y $\frac{1}{n!}\left(\frac{2}{h}\right)^n$.

Tabla 3.3.1: Valores calculados en la CPU para ser ingresados como constantes en texturas sobre la GPU

| Expresiones | | Tamaño de la Variable como textura |
|---|---|---|
| $\frac{1}{n!}\left(\frac{2}{h^2}\right)^n$ | Se genera un vector de largo que depende de la cantidad de términos que posee la expansión de Taylor, creando un vector, que se denominará $ST$. | P x 1 |
| $r_y^k$ | Es un valor que se obtiene del algoritmo de *clustering*, los datos se ingresan en un vector que depende la cantidad de cúmulos estimada. | K x 1 |
| $x_i \in S_k$ | Esta expresión es obtenida después de haber realizado el algoritmo de *clustering*, se genera una matriz que indica la pertenencia de un elemento $x_i$ al respectivo *cluster* $S_k$con un 1 si pertenece o un 0 si no, que se denominará $CI$. | N/C x C K |
| $c_k$ | Corresponde a los centros de los *clusters*, es el resultado del algoritmo de *clustering* y se genera un vector que depende de la cantidad de *clusters* formados. | K x D |

Luego para el cálculo de la expresión 3.3.1 en la GPU, se realizan 7 pasos los que se llevan a cabo sobre una serie de programas. El primer programa (Programa 1) realiza el cómputo de la expresión $\|y_j - c_k\|^2$entre todos los elementos $y_j$ y $c_k$ obteniéndose un una textura de tamaño $\left[\frac{N}{C}, C \cdot K\right]$. Un segundo programa (Programa 2) se encarga de realizar la comparación $\|y_j - c_k\| < r_y^k$ y a su vez calcular el término $e^{-\frac{\|y_j - c_k\|^2}{h^2}}$, obteniendo una textura del mismo tamaño que la resultante del Programa 1, sólo que esta vez contiene todos los términos $e^{-\frac{\|y_j - c_k\|^2}{h^2}}$, como se muestra en la Figura 3.3.4.

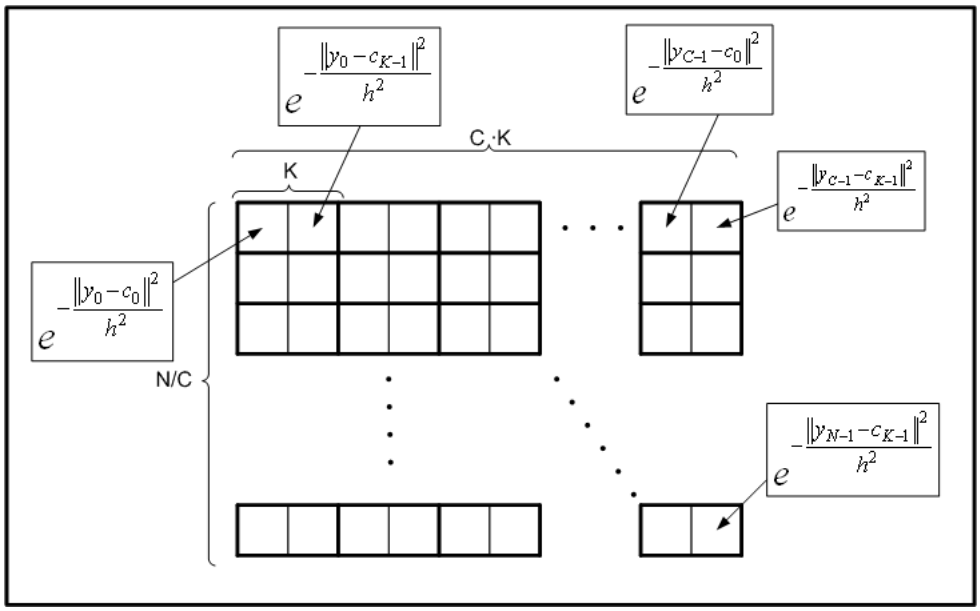



Figura 3.3.4: Resultado Programa 2 IFGT en GPU

Luego de similar forma al paso o Programa 1, un tercer programa (Programa 3), realiza el cálculo de $\|x_i - c_k\|^2$ para cada todas las combinaciones de $x_i$ con $c_k$. Dentro del mismo programa se realiza el cálculo $e^{\frac{-\|y_j - c_k\|^2}{h^2}}$, obteniéndose una textura de dimensiones $\left[\frac{N}{C}, C \cdot K\right]$. Posteriormente un cuarto programa (Programa 4) multiplica la textura resultante que contiene los valores $e^{\frac{-\|x_i - c_k\|^2}{h^2}}$ por la textura que contiene los valores de $q_i$ y por la textura $CI$ que contiene 1 o 0 dependiendo si el elemento $x_i$ pertenece o no al cluster $S_k$. Luego se obtiene una textura que en cada pixel tiene almacenado el valor $q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}}$ si $x_i \in S_k$ ó 0 si no pertenece.

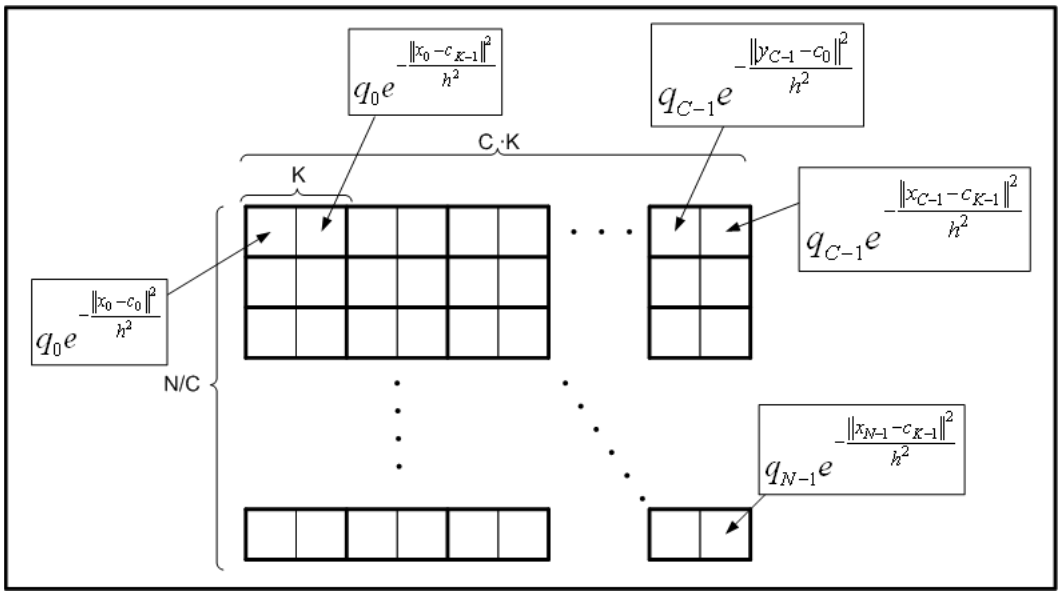



Figura 3.3.5: Resultado Programa 4 IFGT en GPU

Hasta el momento se tiene calculado lo que corresponde a los términos $e^{-\frac{\|y_j - c_k\|^2}{h^2}}$ y $q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}}$ de lo que se encuentra en la expresión 3.3.1, luego falta calcular el término $\sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j - c_k)}{h}\right]^n$, el cual se realiza de forma parecida a la implementación de la Transformada de Gauss sobre la GPU.

Se procede de un $y_j$ a la vez, debido a que el cálculo del término $(x_i - c_k)(y_j - c_k)$ para todos los índices genera un cubo, es decir, se tiene una variable con dimensiones $[N, N, K]$ si es que se realiza todas las combinaciones, lo que no es factible de trabajar sobre la GPU, dado que tiene una limitante de hardware sobre el tamaño de las texturas. Por ello un quinto programa (Programa 5) toma un $y_j*$ y realiza el cálculo de $(x_i - c_k)(y_j * -c_k)$ y a su vez lee la textura $ST$ y realiza el cálculo de $\sum_{n=0}^{p-1} \frac{1}{n!} \left(\frac{2}{h^2}\right)^n [(x_i - c_k)(y_j * -c_k)]^n$, luego como resultado de éste programa se obtiene una textura de tamaño $\left[\frac{N}{C}, K \cdot C\right]$, lo que se bosqueja en la figura 3.3.6. Se debe notar que cada cuadrado presente en la figura 3.3.6, corresponde a un pixel.

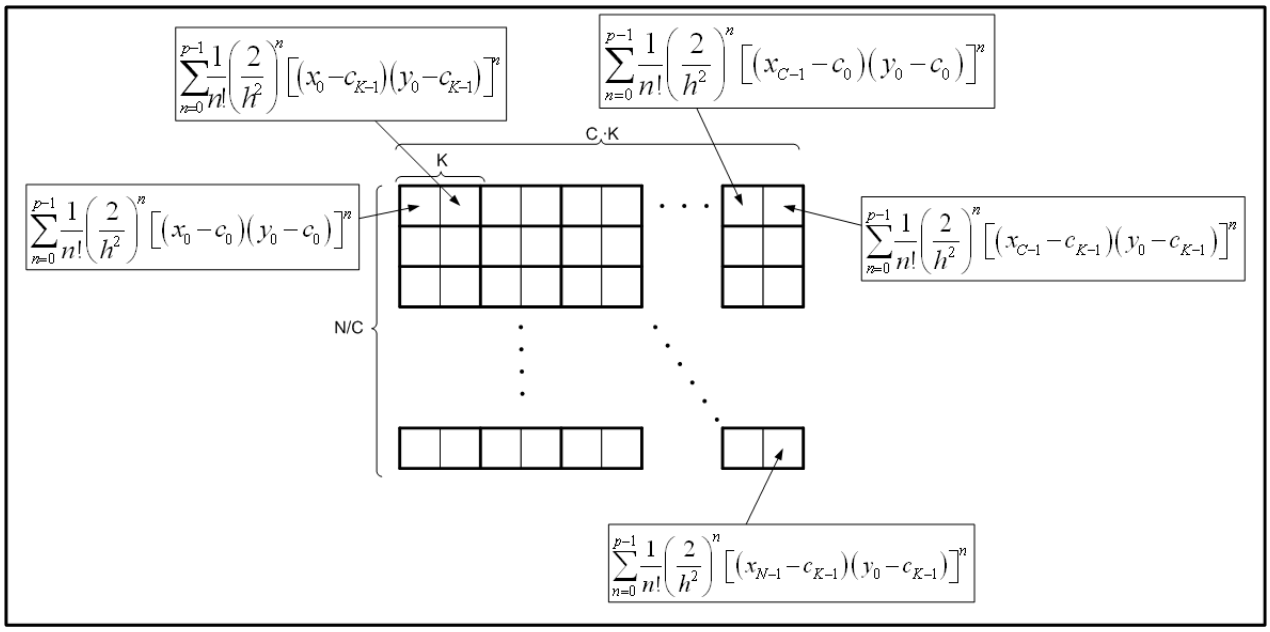



Figura 3.3.6: Resultado Programa 5 IFGT en GPU

Una vez finalizado el programa se procede a calcular el término $q_i e^{\frac{-\|x_i-c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i-c_k)}{h} \frac{(y_j*-c_k)}{h}\right]^n$, lo que se realiza en el Programa 6. Éste programa recibe los resultados del Programa 4: $(q_i e^{\frac{-\|x_i-c_k\|^2}{h^2}})$ y Programa 5: $(\sum_{n=0}^{p-1} \frac{1}{n!} \left(\frac{2}{h^2}\right)^n [(x_i - c_k)(y_j * -c_k)]^n)$ en forma de 2 texturas de igual tamaño $(\left[\frac{N}{C}, K \cdot C\right])$ y los multiplica, obteniendo el resultado deseado en una textura de tamaño $\left[\frac{N}{C}, K \cdot C\right]$. La figura 3.3.7 representa lo que se realiza en esta etapa.

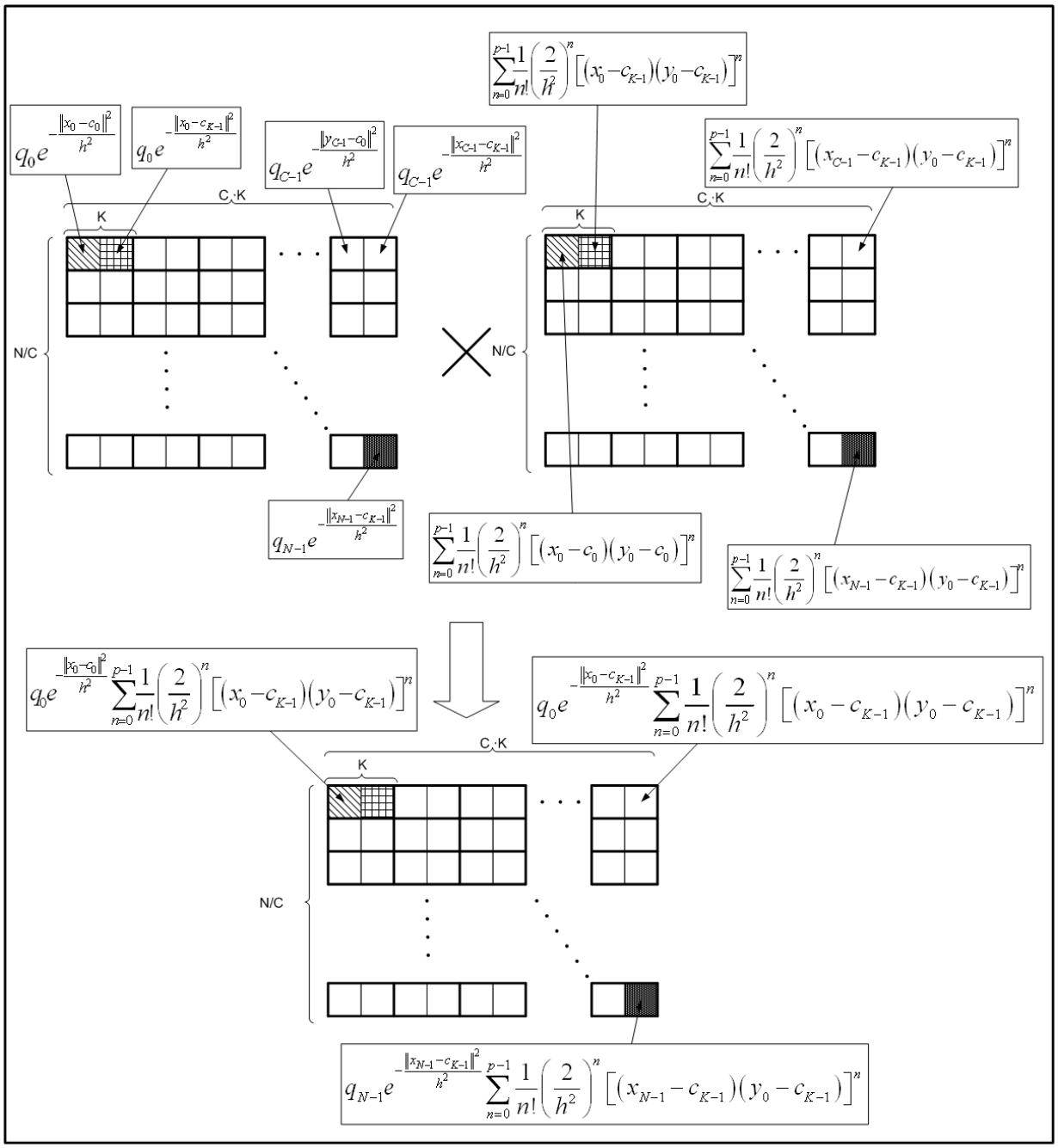



Figura 3.3.7: Resultado del Programa 6

Luego se obtiene en el pixel $i$ el resultado de $q_i e^{\frac{-\|x_i-c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i-c_k)}{h} \frac{(y_j*-c_k)}{h}\right]^n$ luego a través de la ejecución del programa 7 se realiza el método de reducción utilizado en la Transformada de Gauss obteniéndose finalmente el resultado de $\sum_{x_i \in S_k} q_i e^{\frac{-\|x_i-c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i-c_k)}{h} \frac{(y_j*-c_k)}{h}\right]^n$, lo que es

almacenado en una textura de tamaño $[1, K]$. El resultado obtenido es leído de la GPU y almacenado en una variable dentro de la GPU.

Los pasos presentados con los programas 5, 6 y 7 se deben repetir para cada elemento de $\{y_j\}$. Una vez obtenidos todos los valores de $\sum_{x_i \in S_k} q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j * - c_k)}{h}\right]^n$ para cada elemento de $\{y_j\}$, los resultados son almacenados en una textura la cual se multiplica por la textura obtenida como resultado del programa 2, la cual contiene $e^{-\frac{\|y_j - c_k\|^2}{h^2}}$ , obteniendo así el resultado de la IFGT. A continuación se resume lo anterior en lo que corresponde al algoritmo 3.

**Algoritmo 3** Algoritmo de la IFGT en GPU

1. Realizar los pasos 1, 2 y 3 presentados en el Algoritmo 2.
2. Cargar los datos $x_i$, $y_j$, $q_i$ $\frac{1}{n!}\left(\frac{2}{h^2}\right)^n$, $r_y^k$, $x_i \in S_k$ y $c_k$ sobre las texturas correspondientes.
3. Ejecutar el Programa 1 para obtener obtener $\|y_j - c_k\|^2$.
4. Ejecutar el Programa 2 y obtener $e^{-\frac{\|y_j - c_k\|^2}{h^2}}$ si $\|y_j - c_k\| < r_y^k$ y 0 si no cumple la condición.
5. Ejecutar el Programa 3 y obtener $e^{\frac{-\|x_i - c_k\|^2}{h^2}}$.
6. Ejecutar el Programa 4 y multiplicar $q_i$,$e^{\frac{-\|x_i - c_k\|^2}{h^2}}$ y $CI$ para obtener $q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}}$ si $x_i \in S_k$ y 0 si no.
7. Para $j = 1, ...N$
   a) Calcular $\sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j - c_k)}{h}\right]^n$.
   b) multiplicar $\sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j - c_k)}{h}\right]^n$ por el resultado de 6.
   c) realizar la reducción del resultado $7b$ y obtener $\sum_{x_i \in S_k} q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j * - c_k)}{h}\right]^n$.
   d) Descargar el valor $7c$.
   e) Fin.Cargar todos los valores obtenidos en $7d$ y cargarlos en una textura y multiplicarlos por el resultado obtenido en 4 y se obtener:

$$\hat{G}(y_j) = \sum_{\|y_j - c_k\| < r_y^k} e^{\frac{-\|y_j - c_k\|^2}{h^2}} \sum_{x_i \in S_k} q_i e^{\frac{-\|x_i - c_k\|^2}{h^2}} \sum_{n=0}^{p-1} \frac{2^n}{n!} \left[\frac{(x_i - c_k)}{h} \frac{(y_j - c_k)}{h}\right]^n \quad (3.3.2)$$

## 3.4. Pruebas Experimentales

Las pruebas a realizar se refieren básicamente a comprobar la velocidad del cálculo de las distintas implementaciones de la Transformada de Gauss y de la IFGT en la GPU.

Para ello se realizan pruebas considerando las limitaciones de cantidad de datos que se pueden cargar sobre una textura dada la implementación realizada de la Transformada de Gauss y la IFGT, por ello se realiza una prueba manteniendo la cantidad de datos fijos en 65536 lo que corresponde a $256 \cdot 256$ y variando la dimensión de los datos de 1 hasta 8. El límite de 8 se debe a que cuando se tiene $256 \cdot 8$ se alcanza 2048 que es el límite del tamaño de la textura que se carga.

Una segunda prueba se realiza considerando una cantidad de elementos variables partiendo desde 256, 1024, 4096, 16384 y 65536 donde los valores corresponden a los factores $16 \cdot 16$, $32 \cdot 32$, $64 \cdot 64$, $128 \cdot 128$ y $256 \cdot 256$ , lo que está dado por utilizar la reducción explicada en la implementación.

Las razones de velocidad se calculan mediante el cuociente:

$$Aceleracin = \frac{Tiempo\,Ejecucin\,GPU}{Tiempo\,Ejecucin\,CPU} \tag{3.4.1}$$

Donde el tiempo de Ejecución GPU corresponde al tiempo que toma la implementación sobre la GPU, la cual se compara con la implementación respectiva sobre la CPU.

Además de las pruebas de velocidad se revisan los errores de las implementaciones, comparando el error relativo máximo comparado con la implementación de la Transformada de Gauss en C, dado por:

$$err = max\left(\frac{TG_i - Implementacin_i}{TG_i}\right) \tag{3.4.2}$$

Las pruebas anteriores se realizan para la Transformada de Gauss y la IFGT en CPU, la Transformada de Gauss y la IFGT en GPU.

Las pruebas se realizan con los datos $\{q_i\}$ generados aleatoriamente, $\{x_i\}$ distribuidas uniformemente en un hipercubo unitario normalizado entre 0 y 1, y en caso de las variables $\{y_j\}$ generado aleatoriamente.

Dentro de las pruebas a realizar se considera el corroborar la implementación de la IFGT en C con la implementación de Vikas C. Raykar and Changjiang Yang [21] para Matlab. Para ello se realizan 2 pruebas en donde se prueba el algoritmo implementado en C con el que se posee en

MATLAB. La primera es con dimensión 3 y ancho de banda 4, en donde se varía la cantidad de datos desde N=10000 a N=960000. La segunda es con se mantiene constante el ancho de banda y se aumenta la dimensión de los datos a 4 y los datos varían de N=10000 a N=960000.

Para implementación sobre la GPU, se realiza una primera prueba manteniendo cantidad constante de datos y variando la dimensión de éstos de 1 a 8. Dado que la cantidad de datos que se puede manejar sobre la tarjeta es limitada se trabaja con un máximo de datos de 65536. La segunda prueba se obtiene al mantener la dimensión constante y a variar la cantidad de datos desde los valores 256, 1024, 4096, 16384, 65536.

# Capítulo 4

# Resultados

En esta sección se presenta los resultados de las pruebas realizadas a las implementaciones de la Transformada de de Gauss y la Transformada Rápida de Gauss Mejorada y las pruebas presentadas en el Capítulo 3.

## 4.1. Transformada de Gauss e IFGT en CPU

La primera prueba es comparar implementación de la IFGT en C con la implementación de IFGT realizada por Vikas C. Raykar and Changjiang Yang [21] para Matlab.

En las Tablas 4.1.1 y 4.1.3 se indican los datos obtenidos para un ancho de banda $H = 0{,}4$ y dimensiones $D = 3$ y $D = 4$ respectivamente. El error corresponde al error relativo de la implementación de la IFGT en C y la Transformada de Gauss en C. Los tiempos se encuentran en segundos y el Speed Up 1 y Speed Up 2 corresponde a las razones $TiempoGT/TiempoIFGTC$ y $TiempoIFGTMatlab/TiempoIFGTC$.

A la Izquierda de la figura 4.1.1 se grafican los tiempos para el cálculo de la Transformada de Gauss, IFGT en Matlab e IFGT en C, en el eje de las ordenadas se tiene la cantidad de datos y en la abscisas el tiempo en segundos. Se observa que el cálculo directo de la Transformada de Gauss es considerablemente mayor que a medida que aumenta la cantidad de datos. Además se observa que la implementación de la IFGT en es levemente menor que la versión en Matlab, lo que es debido a que la IFGT implementada en C esta íntegramente en lenguaje C en cambio la Implementación en Matlab, a pesar de que esta enlazada con el lenguaje C es mayor, dado que

Tabla 4.1.1: Tiempos y Error de la implementación de Transformada de Gauss e IFGT con d=3 y h=0.4

| N | GT [s] | IFGT MATLAB [s] | IFGT C [s] | Error Relativo GT con IFGT C | Speed Up 1 | Speed Up 2 |
|---|---|---|---|---|---|---|
| 10000 | 8,172 | 2,265 | 1,359 | 3,10E-07 | 6,01 | 1,67 |
| 20000 | 33,281 | 6,172 | 2,656 | 3,10E-07 | 12,53 | 2,32 |
| 40000 | 133,688 | 14,203 | 5,765 | 3,00E-07 | 23,19 | 2,46 |
| 80000 | 533,985 | 32,625 | 10,375 | 3,10E-07 | 51,47 | 3,14 |
| 120000 | 1204,031 | 32,016 | 18,515 | 3,10E-07 | 65,03 | 1,73 |
| 240000 | 4825,046 | 97,657 | 43,812 | 3,10E-07 | 110,13 | 2,23 |
| 480000 | 19257,469 | 128,374 | 69,921 | 3,10E-07 | 275,42 | 1,84 |
| 960000 | 77606,125 | 256,484 | 178,812 | 3,20E-07 | 434,01 | 1,43 |

ocupa como intermediario otro programa, que en este caso es Matlab. A la derecha se observa el error relativo de la Transformada Rápida de Gauss Mejorada en C con respecto a la Transformada de Gauss en Matlab, el cual es bastante pequeño y cumple con el requerimiento de tener un error menor a $1e^{-6}$.

Tabla 4.1.3: Tiempos y error de la implementación de Transformada de Gauss e IFGT con d=4 y h=0,4

| N | GT [s] | IFGT MATLAB [s] | IFGT C [s] | Error Relativo GT con IFGT C | Speed Up 1 | Speed Up 2 |
|---|---|---|---|---|---|---|
| 10000 | 8,906 | 37,391 | 1,359 | 2,00E-07 | 0,47 | 1,96 |
| 20000 | 36,188 | 74,344 | 2,656 | 2,10E-07 | 0,95 | 1,95 |
| 40000 | 145,312 | 182,094 | 5,765 | 2,20E-07 | 1,92 | 2,41 |
| 80000 | 582,219 | 364,437 | 10,375 | 2,20E-07 | 3,85 | 2,41 |
| 120000 | 1310,391 | 546,687 | 18,515 | 2,20E-07 | 4,60 | 1,92 |
| 240000 | 5248,25 | 1093,375 | 43,812 | 2,20E-07 | 7,45 | 1,55 |
| 480000 | 21075,219 | 2189,359 | 69,921 | 2,20E-07 | 18,55 | 1,93 |
| 960000 | 83558,328 | 5385,171 | 178,812 | 2,30E-07 | 29,65 | 1,91 |

A la Izquierda de la figura 4.1.2se grafican los tiempos para el cálculo de la Transformada de Gauss, IFGT en Matlab e IFGT en C, se observa que el cálculo directo de la Transformada de

Figura 4.1.1: Tiempos de GT, IFGT en Matlab e IFGT en C, D=3,H=0.4

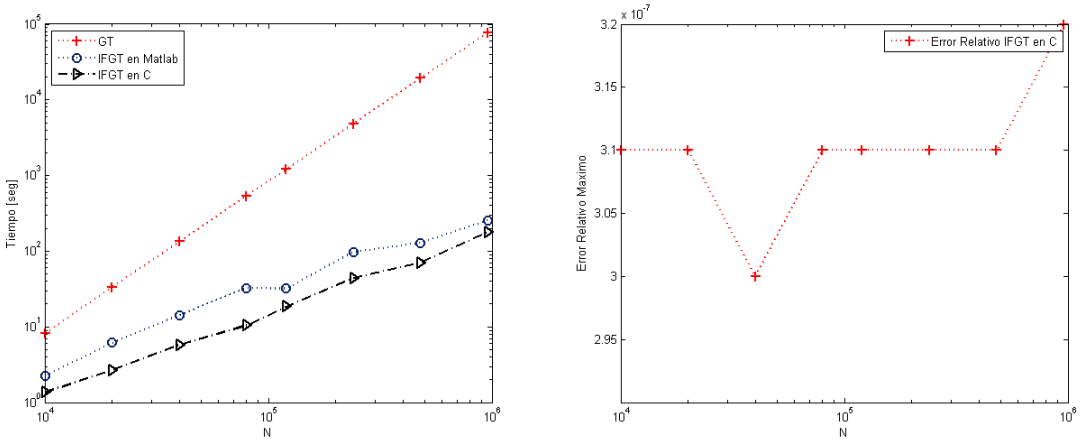
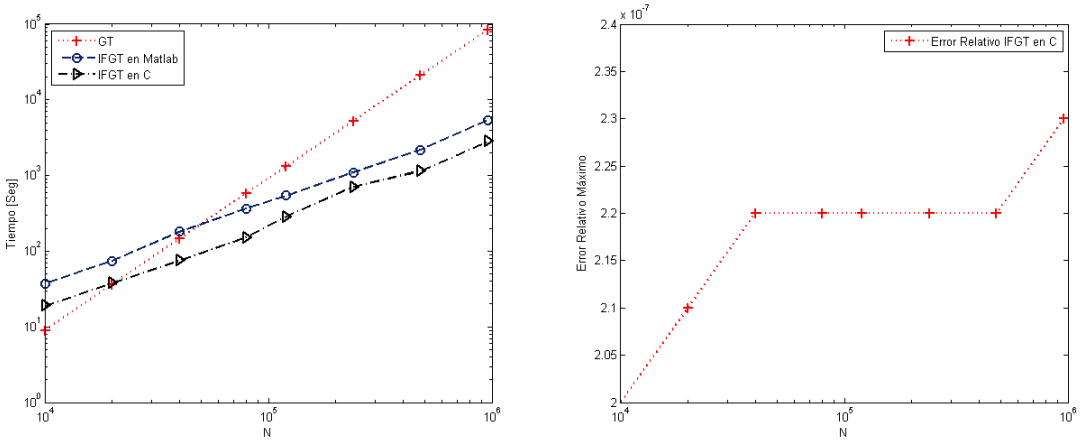



Figura 4.1.2: Tiempos de GT, IFGT en Matlab e IFGT en C, D=4 y H=0.4




Gauss es levemente menor para los primeros elementos, pero a partir de N=40000 pasa a ser más rápido el calculo a través de ambas implementaciones de la IFGT. A la derecha se observa el error relativo, es pequeño y cumple con el requerimiento de ser menor a $1e^{-6}$.

Al comparar los gráficos 4.1.1 y 4.1.2 se observa de cierta forma como influye el tener el ancho de banda h fijo en la aceleración, lo que es comentado en [1] indicando que la aceleración de las aproximaciones realizadas por series es menor para anchos de bandas pequeños.

En la tabla 4.1.4 el error presentado es menor que en el caso de las tablas 4.1.1 y 4.1.3 dado que se encuentran fijados los parámetros de P y K, es decir, al estar fijando los valores de K (número de *cluster*) y P (cantidad de términos de la Serie de Taylor) se esta afectando directamente en la

Tabla 4.1.4: Tiempos de ejecución y error de la Transformada de Gauss e IFGT en la CPU
Con $N = M = 65536$ $(256 \cdot 256)$
con un ancho de banda $H = 0{,}3D$ y para el caso de la IFGT se fijaron los parámetros P=14, y K=3.

| Dimensión | GT [s] | IFGT [s] | Error |
|---|---|---|---|
| 1 | 220,375 | 1,000 | 1,14e-10 |
| 2 | 217,704 | 1,421 | 1,10e-10 |
| 3 | 227,984 | 3,406 | 6,41e-11 |
| 4 | 244,641 | 9,312 | 5,53e-11 |
| 5 | 250,828 | 22,484 | 5,07e-11 |
| 6 | 262,484 | 155,562 | 4,48e-11 |
| 7 | 276,485 | 145,765 | 4,18e-11 |
| 8 | 284,625 | 240,718 | 4,10e-11 |

precisión de los datos.

Al comparar la aceleración de las tablas 4.1.4 y 4.1.5 se observa un menor rendimiento a medida para misma cantidad de puntos y dimensión pero a diferentes anchos de bandas lo que es observado como un defecto de los métodos de estimación por series como lo es la IFGT y FGT para anchos de banda pequeños [1] .

Tabla 4.1.5: Tiempos de ejecución de la Transformada de Gauss en la GPU
Con un $N = M = 65536$ $(256 \cdot 256)$ con un ancho de banda $H = 0{,}5D$ y para el caso de la IFGT se fijaron los parámetros P=14, y K=3.

| Dimensión | GT [s] | IFGT [s] |
|---|---|---|
| 1 | 189,875 | 0,968 |
| 2 | 190,532 | 1,296 |
| 3 | 227,797 | 1,953 |
| 4 | 213,516 | 2,734 |
| 5 | 250,594 | 4,015 |
| 6 | 262,391 | 5,234 |
| 7 | 277,016 | 10,234 |
| 8 | 252,688 | 10,390 |

Las siguientes Tablas 4.1.6, 4.1.7 y 4.1.8 muestran los datos obtenidos para distintas cantidades de datos.

Tabla 4.1.6: Resultados de la Transformada de Gauss e IFGT CPU
En la siguiente tabla se muestran los datos obtenidos para dimensión 3 y 4, ancho de banda H=1, P=14, K=3

| | D=3 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,062 | 9,6e-09 | – |
| 1024 | 0,0470 | 0,078 | 9,5e-09 | 0,60 |
| 4096 | 0,7500 | 0,140 | 1,0e-10 | 5,36 |
| 16384 | 12,0780 | 0,437 | 1,0e-10 | 27,64 |
| 65536 | 199,2500 | 1,640 | 3,6e-11 | 121,49 |

| | D=4 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,062 | 9,7e-09 | – |
| 1024 | 0,0470 | 0,093 | 9,5e-09 | 0,51 |
| 4096 | 0,7810 | 0,203 | 1,1e-10 | 3,85 |
| 16384 | 13,1600 | 0,720 | 1,4e-10 | 18,33 |
| 65536 | 213,3590 | 2,437 | 3,8e-11 | 87,55 |

Tabla 4.1.7: Resultados de la Transformada de Gauss e IFGT CPU
En la siguiente tabla se muestran los datos obtenidos para dimensión 5 y 6, ancho de banda H=2, P=9, K=2

| | D=5 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,078 | 9,7e-10 | – |
| 1024 | 0,0470 | 0,109 | 9,5e-09 | 0,74 |
| 4096 | 0,7970 | 0,39 | 1,1e-10 | 2,43 |
| 16384 | 13,4840 | 1,859 | 9,5e-10 | 7,51 |
| 65536 | 219,5630 | 5,687 | 3,9e-11 | 38,92 |

| | D=6 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,109 | 9,8e-10 | – |
| 1024 | 0,047 | 0,578 | 9,5e-10 | 0,088 |
| 4096 | 0,844 | 1,484 | 1,1e-10 | 0,6 |
| 16384 | 14,078 | 5,703 | 1,5e-10 | 2,571 |
| 65536 | 231,125 | 33,421 | 4,4e-11 | 6,93 |

Tabla 4.1.8: Resultados de la Transformada de Gauss e IFGT CPU
En la siguiente tabla se muestran los datos obtenidos para dimensión 7 y 8, ancho de banda H=3, P=7, K=2

| | D=7 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,109 | 9,7e-10 | – |
| 1024 | 0,0620 | 0,281 | 9,5e-09 | 0,28 |
| 4096 | 1,0000 | 0,875 | 3,1e-10 | 1,21 |
| 16384 | 16,7500 | 3,484 | 1,1e-11 | 4,87 |
| 65536 | 277,0620 | 13,234 | 3,6e-11 | 21,00 |

| | D=8 | | | |
|---|---|---|---|---|
| N | GT [s] | IFGT [s] | Error | Speed Up |
| 256 | 0,0001 | 0,171 | 9,8e-09 | – |
| 1024 | 0,0620 | 0,484 | 9,9e-10 | 0,14 |
| 4096 | 1,0000 | 1,843 | 1,1e-10 | 0,51 |
| 16384 | 17,3130 | 6,968 | 1,3e-10 | 2,50 |
| 65536 | 286,7500 | 86,359 | 3,6e-11 | 3,32 |

Los valores de ancho de banda H, fueron escogido de modo que se lograra un ancho de banda adecuado para la IFGT, lo cual se escogió revisando las pruebas realizadas por Vikas C. Raykar and Changjiang Yang [1]

Se observa en los distintos casos que la IFGT tiene un aumento en la aceleración a medida que se va aumentando las cantidades de datos, y se observa además que el ancho de banda influye sobre la aceleración . En todos los casos se tiene un buen valor del error relativo, suficientemente bajo.

## 4.2. Transformada de Gauss e IFGT en GPU

A Continuación se presentan los datos obtenidos con las implementaciones de la Transformada de Gauss e IFGT sobre la GPU.

Tabla 4.2.1: Tiempos de ejecución de la Transformada de Gauss en la GPU

| Dimensión | Tiempo ejecución Programa 1 (N veces) [segundos] | Tiempo ejecución Programa 2 (N veces) [segundos] | Tiempo de leer 1 pixel (N veces) [segundos] | Tiempo Total |
|---|---|---|---|---|
| 1 | 21,850 | 9,120 | 32,293 | 63,438 |
| 2 | 5,575 | 12,011 | 50,930 | 68,531 |
| 3 | 6,157 | 13,562 | 58,546 | 78,328 |
| 4 | 4,532 | 16,191 | 74,011 | 94,750 |
| 5 | 6,926 | 18,081 | 531,852 | 556,937 |
| 6 | 7,348 | 16,682 | 633,83 | 657,985 |
| 7 | 6,695 | 19,063 | 735,631 | 761,516 |
| 8 | 7,440 | 20,216 | 832,517 | 860,266 |

Los datos presentados en la Tabla 4.2.1 corresponden a una cantidad fija de $N = M = 65536$ elementos, el ancho de banda $h$ varía en función de la dimensión de los datos de forma $h = 0{,}3D$. La cantidad de elementos esta definida por la capacidad máxima que permite la textura para el caso de tener $D = 8$ y 256 elementos se tiene un total de 2048 pixeles utilizados en una dirección de la textura.

Los tiempos que se entregan representan el tiempo que tomó realizar las distintas operaciones N veces, como se comentó en el capítulo 3, se observa que lo más considerable en proporción es la lectura de los pixeles desde la la GPU hacia la CPU y a medida que se aumenta la dimensión del problema aumenta este tiempo.

De la tabla además se observa que el programa 2, aquel que realiza la reducción en la implementación de la Transformada de Gauss en la GPU, toma algunos segundos, y en caso del programa 1, aquel que realiza el cálculo de $e^{\frac{-\|x_i - y_j\|^2}{h^2}}$, también toma poco tiempo para un N=65536, y ambos se ven poco afectados con el aumento de la dimensión.

Los datos presentados en la Tabla 4.2.2 corresponden a una cantidad fija de $N = M = 65536$ elementos, el ancho de banda $h$ varía en función de la dimensión de los datos de forma $h = 0{,}3D$. La cantidad de elementos esta definida por la capacidad máxima que permite la textura para el

Figura 4.2.1: Gráfico Tiempos IFGT con N=65536

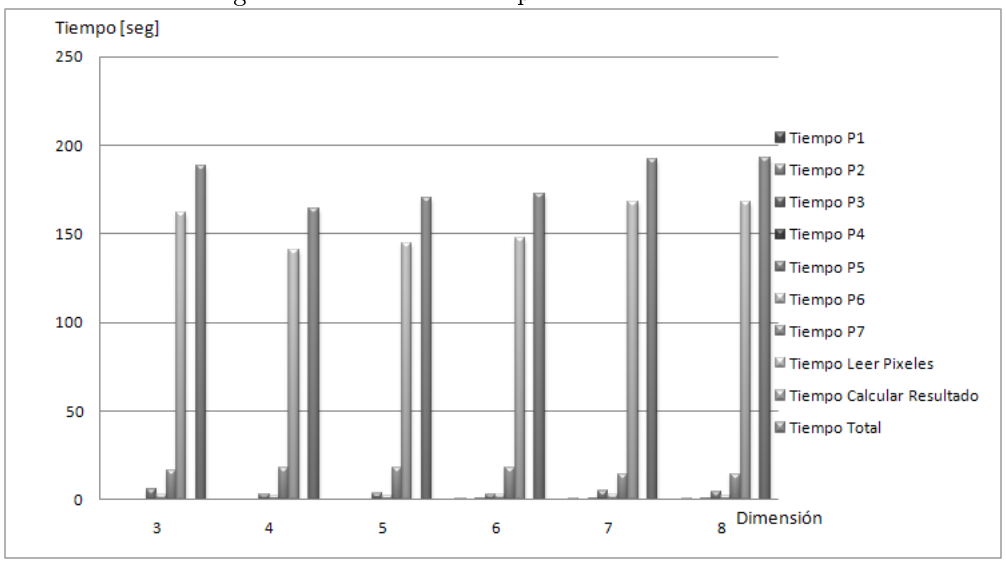



Figura 4.2.2: Gráfico Tiempos IFGT con N=65536

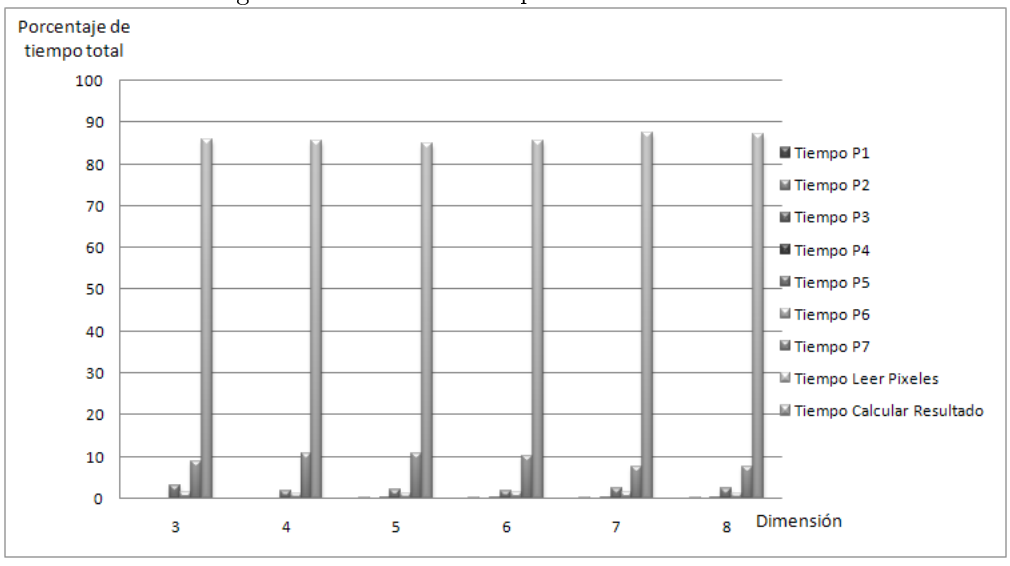


Tabla 4.2.2: Tiempos de Ejecución de cada programa de la IFGT en GPU

| D | P1 [s] | P2 [s] | P3 [s] | P4 [s] | P5 [s] | P6 [s] | P7 [s] | Leer pixels [s] | Calcular Resultado [s] | Total [s] |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 0,016 | 0,062 | 0,016 | 0,078 | 5,956 | 2,589 | 16,684 | 161,818 | 0,016 | 188,344 |
| 4 | 0,016 | 0,078 | 0,016 | 0,093 | 3,033 | 1,944 | 17,604 | 140,419 | 0,016 | 164,344 |
| 5 | 0,016 | 0,141 | 0,046 | 0,11 | 3,489 | 2,206 | 18,072 | 144,498 | 0,062 | 169,891 |
| 6 | 0,016 | 0,188 | 0,046 | 0,172 | 3,073 | 2,455 | 17,507 | 147,231 | 0,063 | 172,125 |
| 7 | 0,016 | 0,25 | 0,047 | 0,219 | 4,767 | 2,754 | 14,403 | 167,825 | 0,047 | 191,828 |
| 8 | 0,016 | 0,25 | 0,047 | 0,203 | 4,517 | 2,302 | 14,4 | 168,123 | 0,063 | 192,735 |

caso de tener $D = 8$ y 256 elementos se tiene un total de 2048 pixeles utilizados en una dirección de la textura. Los programas 5, 6 y 7 se realizan $N$ veces siendo $N$ la cantidad de elementos, en este caso 65536. Los Programas o Pasos 5, 6 ,7 y la lectura de pixeles se realiza N veces como se muestra en el Capítulo 3.

Se observa que al igual que en la implementación de la Transformada de Gauss el hecho de realizar el traspaso de los datos desde la GPU a las CPU al leer los pixeles pasa a ser bastante importante al ser realizado N veces. Cabe destacar que el resto de los cálculos realizados toma sumamente poco tiempo siendo el que mayor tiempo toma es el programa 7, que es en donde se realiza la reducción de los datos como se indico en el capítulo anterior.

Del análisis de los tiempos de las tablas 4.2.1 y 4.2.2 se observa que el traspaso de los datos desde la GPU a la CPU es algo sumamente costoso a pesar de que la GPU se encuentra conectada a través de un bus PCI-e que reduce el problema del "cuello de botella" en el traspaso de los datos desde la GPU a CPU.

Llama la atención en la tabla 4.2.2 que el tiempo de ejecución de los Programas 1, 2, 3 y 4 es casi invariante al aumento de la dimensión de los datos, al igual que el cálculo del resultado en donde se obtiene el valor de la IFGT, lo que resulta sumamente atractivo. Se observa algo relativamente similiar para el caso de los Programas 5, 6 y 7, en donde a pesar de que se obtienen tiempos del orden de segundos y decenas de segundos no se ve afectado por el aumento de la dimensión.

Algo similar se observa en la tabla 4.2.1 en donde el Programa 1, salvo para el caso de D=1, los tiempos se encuentran bajo los 10 segundos. Para el Programa 2 se observa algo levemente diferente dado que el tiempo aumenta levemente con la dimensión. El salto que se observa entre la dimensión

4 y 5 en el programa 2 se debe a que se define una textura de mayor tamaño para la carga de los datos dado que pasamos a tener un tamaño mayor en la textura dado que todas las pruebas anteriores para dimensiones 1 a 4 se define una tamaño de textura máximo para trabajar de 1024 después de la dimensión 5 este valor es aumentado a 2048 para que se pueda realizar los cálculos hasta la dimensión 8.

Tabla 4.2.3: Tiempos de ejecución y error de la Transformada de Gauss e IFGT en la GPU
Los datos presentados corresponde a la implementación de la Transformada de Gauss y a la IFGT, donde los parámetros de la IFGT son P=14 y K=3, para un ancho de banda H=1, en ambas implementaciones.

| | D=3 | | | | | D=4 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| N | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up |
| 256 | 0,438 | 0,750 | 4,17e-07 | 8,44e-07 | 0,58 | 0,407 | 0,765 | 3,92e-07 | 7,21e-07 | 0,53 |
| 1024 | 0,688 | 1,324 | 4,82e-07 | 8,49e-07 | 0,51 | 0,64 | 1,438 | 5,85e-07 | 8,33e-07 | 0,45 |
| 4096 | 1,703 | 4,359 | 5,67e-07 | 9,29e-07 | 0,39 | 1,656 | 4,281 | 5,43e-07 | 9,03e-07 | 0,39 |
| 16384 | 8,093 | 14,435 | 6,35e-07 | 1,08e-06 | 0,56 | 9,375 | 17,500 | 6,44e-07 | 1,04e-06 | 0,54 |
| 65536 | 80,86 | 166,015 | 5,35e-07 | 1,20e-06 | 0,48 | 96,59 | 207,828 | 5,44e-07 | 1,10e-06 | 0,46 |

Tabla 4.2.4: Tiempos de ejecución y error de la Transformada de Gauss e IFGT en la GPU
Los datos presentados corresponde a la implementación de la Transformada de Gauss y a la IFGT, donde los parámetros de la IFGT son P=9 y K=2, para un ancho de banda H=2, en ambas implementaciones.

| | D=5 | | | | | D=6 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| N | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up |
| 256 | 0,550 | 0,718 | 3,65e-07 | 6,41e-07 | 0,77 | 0,453 | 0,953 | 3,74e-07 | 7,38e-07 | 0,48 |
| 1024 | 0,750 | 1,391 | 4,52e-07 | 7,37e-07 | 0,54 | 0,718 | 1,593 | 4,60e-07 | 7,02e-07 | 0,45 |
| 4096 | 1,766 | 4,188 | 5,40e-07 | 8,31e-07 | 0,42 | 1,766 | 4,469 | 5,09e-07 | 8,68e-07 | 0,40 |
| 16384 | 38,938 | 16,109 | 6,05e-07 | 9,73e-07 | 2,42 | 11,015 | 16,125 | 4,55e-07 | 9,16e-07 | 0,68 |
| 65536 | 557,016 | 110,547 | 3,38e-06 | 1,02e-06 | 5,04 | 127,766 | 119,219 | 5,23e-07 | 9,73e-07 | 1,07 |

Tabla 4.2.5: Tiempos de ejecución de la Transformada de Gauss e IFGT en la GPU
Los datos presentados corresponde a la implementación de la Transformada de Gauss y a la IFGT, donde los parámetros de la IFGT son P=7 y K=2, para un ancho de banda H=3, en ambas implementaciones.

| | D=7 | | | | | D=8 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| N | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up | GT GPU [s] | IFGT GPU [s] | Error GT | Error IFGT | Speed Up |
| 256 | 0,532 | 1,187 | 3,68e-07 | 6,38e-07 | 0,45 | 1,094 | 1,187 | 3,63e-07 | 4,63e-07 | 0,92 |
| 1024 | 0,813 | 1,844 | 4,37e-07 | 7,57e-07 | 0,44 | 1,485 | 1,829 | 4,40e-07 | 5,33e-07 | 0,81 |
| 4096 | 1,875 | 4,703 | 5,02e-07 | 7,92e-07 | 0,40 | 5,25 | 4,704 | 5,14e-07 | 6,21e-07 | 1,12 |
| 16384 | 52,078 | 16,985 | 5,75e-07 | 8,89e-07 | 3,07 | 57,594 | 17,031 | 7,64e-07 | 7,33e-07 | 3,38 |
| 65536 | 758,516 | 125,907 | 1,41e-06 | 1,03e-06 | 6,02 | 667,437 | 139,75 | 9,35e-07 | 9,36e-07 | 4,78 |

En las Tablas 4.2.3,4.2.4 y 4.2.5, se presentan los datos obtenidos para distintos tamaños de datos, dimensiones y anchos de bandas. Se observa que el error de la implementación de la TG y de la IFGT son bastante aceptables siendo del orden de $1e^{-6}$.

## 4.3. Comparaciones entre Implementaciones en CPU y GPU

Los datos presentados en las secciones 4.1 y 4.2 se puede comparar

Tabla 4.3.1: Comparaciones de Tiempos en Cálculo de las implementaciones en CPU y GPU

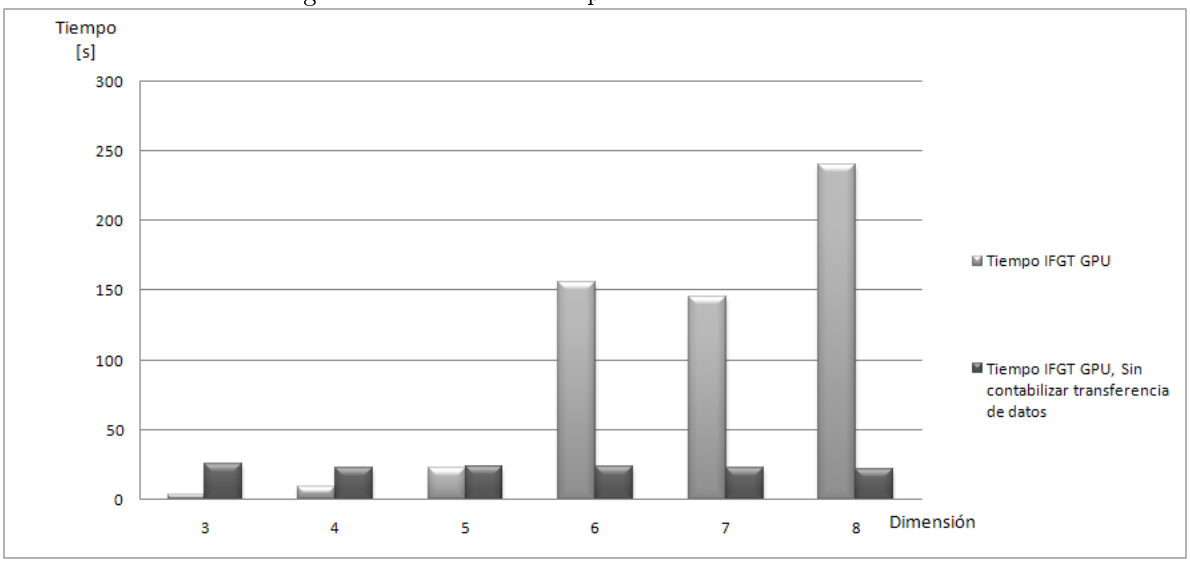
| | GT | | IFGT | | Aceleración | |
|---|---|---|---|---|---|---|
| Dimensión | GPU [s] | CPU [s] | GPU [s] | CPU [s] | GT CPU/ GT GPU | IFGT CPU/ IFGT GPU |
| 3 | 78,328 | 227,984 | 188,344 | 3,406 | 2,91 | 0,02 |
| 4 | 94,750 | 244,641 | 164,344 | 9,312 | 2,58 | 0,06 |
| 5 | 556,937 | 250,828 | 169,891 | 22,484 | 0,45 | 0,13 |
| 6 | 657,985 | 262,484 | 172,125 | 155,562 | 0,40 | 0,90 |
| 7 | 761,516 | 276,485 | 191,828 | 145,765 | 0,36 | 0,76 |
| 8 | 860,266 | 284,625 | 192,735 | 240,718 | 0,33 | 1,25 |

En la Tabla 4.3.1 se puede observar comparaciones de velocidad de las distintas implementaciones. Se observa que para dimensiones bajas la IFGT implementada en la GPU posee una muy baja aceleración comparado con la IFGT implementada en C, lo cual se tiende a invertir a medida que aumentan los datos logrando una aceleración superior a 1 sólo en el caso de dimensión 8.

Tabla 4.3.2: Tiempos de Ejecución de cada programa de la IFGT en GPU

| D | P1 [s] | P2 [s] | P3 [s] | P4 [s] | P5 [s] | P6 [s] | P7 [s] | Calcular Resultado [s] | Total [s] |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0,016 | 0,062 | 0,016 | 0,078 | 5,956 | 2,589 | 16,684 | 0,016 | 25,42 |
| 4 | 0,016 | 0,078 | 0,016 | 0,093 | 3,033 | 1,944 | 17,604 | 0,016 | 22,80 |
| 5 | 0,016 | 0,141 | 0,046 | 0,110 | 3,489 | 2,206 | 18,072 | 0,062 | 24,14 |
| 6 | 0,016 | 0,188 | 0,046 | 0,172 | 3,073 | 2,455 | 17,507 | 0,063 | 23,52 |
| 7 | 0,016 | 0,250 | 0,047 | 0,219 | 4,767 | 2,754 | 14,403 | 0,047 | 22,50 |
| 8 | 0,016 | 0,250 | 0,047 | 0,203 | 4,517 | 2,302 | 14,400 | 0,063 | 21,80 |

Figura 4.3.1: Gráfico Tiempos IFGT con N=65536




Se observa además que la implementación de la Transformada de Gauss sobre la GPU tiene un comportamiento pésimo al pasar de la dimensión 4 a 5 en donde la aceleración decrece considerablemente, lo que es provocado por la transferencia de datos y de como esta implementada.

Haciendo un análisis al no considerar la transferencia de datos en la IFTG implementada en la GPU se obtiene lo siguiente:

De los resultados presentados se puede concluir lo siguiente:

- La Transferencia de los datos entre la GPU y CPU es realmente un problema en particular al realizarlo N veces, como es el caso de la implementación presentada. En este punto es tangible

el “cuello de botella” que existe en la transferencia de datos entre ambas unidades. De la figura 4.2.1 y 4.2.2se puede observar que el tiempo de transferencia de los datos es sumamente alta llegando a ser del 85 %. Se observa del gráfico 4.3.1 que al no considerar la transferencia de datos

- Se observa en la figura 4.3.1 que la implementación de la se mantiene un tiempo constante.
- El hecho de que el tamaño de la textura que se cuenta sea limitado produce un *trade-off* entre la cantidad de elementos que se cargan y la dimensión de éstos, como trabajo futuro se propone la implementación de alguna aplicación en el marco de la Teoría de la Información, siendo este trabajo de título una herramienta para la estimación de FDP utilizadas en las medidas de Divergencia e Información Mutua. Una limitante a la hora de operar con la implementación realizada, además que utilizar la técnica de reducción es necesario trabajar con texturas de tamaño potencia de 2, lo que limita las pruebas realizadas a cantidades de datos fijas de 256, 1024, 4096, 16384 y 65536.
- Si no se contabiliza el tiempo de transferencia de los datos de entre la GPU y la CPU para la IFGT se logra tiempos bastante bajos encontrándose bajo los 26 segundos para las distintas dimensiones en que se probó.

# Capítulo 5

# Conclusiones

- Se implementó satisfactoriamente la IFGT en C en la CPU, teniendo con este algoritmo un punto de comparación para el desarrollo del código en la GPU. Los tiempos y errores estuvieron de acuerdo a los resultados presentados en las bibliografías [1].
- Se implementaron satisfactoriamente la DGT e IFGT en la GPU, dando una mejora sólo para dimensión 8 y con una cantidad de datos de 65536 para el caso de la IFGT, en donde la implementación sobre la GPU tiene una aceleración mayor que 1 comparado con la implementación sobre la CPU.
- El número de datos y el costo de la transferencia de éstos entre la GPU y la CPU es una de las limitantes más importantes que se encontraron en el desarrollo de la IFGT. Esto se debe a que el hardware esta orientado principalmente a manejo gráfico dentro del computador . En tarjetas más recientes, específicamente los modelos que presentan la nueva arquitectura, como el modelo Geforce 8800 GTX y supriores, este problema es disminuido, ya que, se aumentan considerablemente la capacidad de memoria y el ancho de banda para la transferencia de datos entre la GPU a la CPU.
- Entre los problemas presentados durante el desarrollo del trabajo fueron la programación sobre la tarjeta, dado que al realizar las implementaciones sobre la GPU es necesario ingresar los datos como texturas sobre la tarjeta, lo que no es intuitivo. Para solucionar este problema se puede proponer como trabajo futuro la implementación de la IFGT sobre CUDA, lo que implica la adquisición de GPUs, pero de fácil programación. Otro problema es el costo en

la transferencia de los datos desde la GPU a la CPU, debido a la diferencia en los anchos de banda que presentan, esto produjo que en la implementación fuesen mayores los tiempos de computo sobre la GPU que la CPU siendo cerca del 85 % del tiempo. Se propone como trabajo futuro la utilización de una extensión de OpenGL llamada Pixel Buffer Object (PBO), la cual acelera la tasa de transferencia de los datos desde la memoria. La desventaja sería que al utilizar esta extensión aumentan los requerimientos de memoria además de hacer más compleja la programación, ya que no es directa su aplicación..

- El desarrollo de una aplicación específica queda pendiente debido a que no se obtuvieron los resultados esperados con respecto a la aceleraciones en los cálculos de la Transformada de Gauss.

- Un siguiente paso en la implementación sobre la GPU de la IFGT es evitar el traspaso de datos entre la GPU y la CPU, ya sea, modificando el algoritmo presentado en este trabajo de título o realizando la implementación en otra GPU que minimice el traspaso de datos.

# Bibliografía


[1] R. Duraiswami C. Yang and N. Gumerov. Improved fast gauss transform. Technical report, Dept. of Computer Science, University of Maryland, College Park, 2003.

[2] George C Canovos. *Probabilidad y Estadística - Aplicaciones y Métodos*. Virginia Commonwealth University, 1988.

[3] Xu D. *Energy, Entropy and Information Potential in NeuroComputing*. PhD thesis, U. of Florida, 1998.

[4] Dominik Göddeke. *GPGPU: Basic Math Tutorial, Dominik Göddeke*.

[5] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical-Computer Science*, 38:293 306, 1985.

[6] http://graphics.stanford.edu/projects/brookgpu/.

[7] http://libsh.org, Ultima visita 01/09/2008.

[8] http://www.nvidia.com/page/geforce8.html.

[9] http://www.opengl.org/documentation/glsl/.

[10] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms.

[11] W. B. Langdon.

[12] Aaron Lefohn. Gpu memory model overview. In *"General Purpose Computation on Graphics Hardware"*., 2005.

[13] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[14] Manuel Ujaldón Martínez. *Procesadores gráficos para PC.* 2005.

[15] Randima Fernando Matt Pharr. *GPU Gemps 2: Programming Techniques for high-performance graphics and General-purpose computation.* Addison - Wesley, 2005.

[16] Nvidia. www.nvidia.com/cuda.

[17] John Owens. Gpu architecture overview. In *SIGGRAPH 2007*, 2007.

[18] E. Parzen. On estimation of a probability density function and mode. *Ann. Math. Stat.*, 33:pp. 1065–1076., 1962.

[19] Jose C. Principe. *Information-Theoretic Learning - Tutorial.*

[20] Mark J. Kilgard Randima Fernando. *Cg, The Cg Tutorial.* Addison - Wesley, 2003.

[21] Vikas C. Raykar and Changjiang Yang. Improved fast gauss transform - code.

[22] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379 423;623 656, 1948.

[23] Leslie Greengard & John Strain. The fast gauss transform. *SIAM J. ScI. STAT. COMPUT.*, Vol. 12 No. 1:pp. 79–94, January 1991.

[24] www.gpgpu.org.

[25] www.nvidia.com.

[26] C. Yang, R. Duraiswami, N.A. Gumerov, and L. Davis. Improved fast gauss transform and efficient kernel density estimation. *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 664–671 vol.1, Oct. 2003.