UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERIA ESTADÍSTICA E INFORMÁTICA

ESCUELA PROFESIONAL DE INGENIERIA ESTADÍSTICA E INFORMÁTICA

REFINAMIENTO PROGRESIVO Y MEJORAMIENTO DE IMÁGENES Y VIDEOS CON TÉCNICAS DE COMPUTACIÓN PARALELA

TESIS

PRESENTADA POR:

Bach. JUAN ALVARADO LOPE

PARA OPTAR EL TITULO PROFESIONAL DE:

INGENIERO ESTADISTICO E INFORMATICO

PUNO – PERU

2018

UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERIA ESTADÍSTICA E INFORMÁTICA

ESCUELA PROFESIONAL DE INGENIERIA ESTADÍSTICA E INFORMÁTICA

REFINAMIENTO PROGRESIVO Y MEJORAMIENTO DE IMÁGENES Y VIDEOS CON TÉCNICAS DE COMPUTACIÓN PARALELA

TESIS PRESENTADA POR:

Bach. JUAN ALVARADO LOPE

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO ESTADÍSTICO E INFORMÁTICO

APROBADA POR EL JURADO REVISOR CONFORMADOPOR:

PRESIDENTE : M.SC. ERNESTO NAYER TUMI FIGUEROA

PRIMER MIEMBRO : Dr. FIDEL ERNESTO TICONA YANQUI

SEGUNDO MIEMBRO : Dr. JOSE PÁNFILO TITO LIPA

DIRECTOR / ASESOR : Dr. VLADIMIRO IBAÑEZ QUISPE

Área : COMPUTACION GRÁFICA Y VISION COMPUTACIONAL
Tema : CIENCIAS DE LA COMPUTACION
Fecha de Sustentación : 01/06/2018

DEDICATORIA

A Dios, por estar conmigo en cada paso que doy, por fortalecer mi corazón; iluminar y haber puesto en mi camino a aquellas personas que han sido mi soporte y compañía durante todo el periodo de estudio.

A mi padre Pedro y a mi madre Francisca, por el apoyo incondicional y por los sabios consejos que me permitieron encaminar mi vida. Así poder cumplir mis metas.

A mis hermanos por brindarme momentos de alegría y apoyo para seguir mis estudios.

BACH. JUAN ALVARADO LOPE

AGRADECIMIENTOS

A las autoridades y docentes de la Universidad Nacional del Altiplano de Puno, así mismo a la Facultad de Ingeniería Estadística e Informática, personal administrativo, compañeros (as), para ellas y ellos mi profundo agradecimiento y reconocimiento por su colaboración en la presente investigación.

Al Director y Asesor de la investigación; Dr. Vladimiro Ibañez Quispe, quien con su amplia experiencia y trayectoria en el ejercicio profesional, dio una valiosa contribución en la concreción de este trabajo.

A los miembros del jurado, quienes en forma desprendida y con su excelente capacidad y conocimiento en la formación profesional, aportaron a través de sus observaciones respecto a la coherencia y metodología de la presente investigación.

A la familia Alvarado Lope, mi razón de ser, por darme el amor, la comprensión y la fortaleza para cumplir mis metas y superación profesional.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE ACRÓNIMOS

OpenCL : Librería desarrollada por Intel para procesamiento usando GPUs.

GPU : Graphic Processing Unit, unidad de procesamiento gráfico.

RPC : Remote Procedure Call o de sus siglas en inglés llamada de funciones o procedimientos remotos.

VRAM : Video Random Access Memory, memoria de acceso aleatorio dedicado para la memoria de video.

FPS : Fotograms per Second, cantidad de imágenes que se sobreponen en 1 segundo.

OCL **:** Object Constraint lenguaje, lenguaje de restricción de objetos

NCC **:** Norma de Control de Calidad.

GPGPU **:** General-Purpose on Graphics Processing Units; Computación de propósito general en unidades de procesamiento de gráficos.

DVI **:** Interfaz Visual Digital.

VGA **:** Video Graphics Array; o Arreglo de vídeo Gráfico.

HDMI **:** High Definition Multimedia Interface; Interfaz Multimedia de Alta Definicion.

VHS **:** Video Home System, sistema de videocasete y magnetoscopio.

HD-TV **:** high definition televisión Television de Alta Definicion.

RESUMEN

El área de visión computacional permite mejorar imágenes, utilizando la técnica de paralelización del proceso de computo, usando las ventajas de la unidad de procesamiento grafico presente en las modernas tarjetas gráficas, desde la Intel HD Graphics, hasta la poderosa NVidia GTX 1080 con sus 2560 procesadores que permitirán calcular cualquier rutina compleja, si este tiene un esquema correctamente adaptado para este tipo de computo, es por esta razón que el objetivo fue: Desarrollar un software basado en técnicas de paralelización mediante el GPU para la mejora y modificación de imágenes. El diseño de investigación se enmarca dentro del diseño descriptivo, la muestra estuvo constituida por 20 imágenes de distintos formatos. La metodología utilizada fue la Metodología de desarrollo Scrum, con las siguientes fases: el proceso de identificación del problema, lenguaje de Modelamiento Unificado UML y finalmente para la etapa de las pruebas, los resultados fueron: 1) Para la segmentación de los frames se ha hecho uso del software externo que obtiene las entradas en una colección de archivos separados por un índice guía, 2) Se ha generado una librería en modo kernel con el nombre BoxFilterGL_Kernels.cl, que es la que se encarga de realizar las iteraciones y operaciones de filtrado de las matrices que procesan y se generan en las iteraciones de ancho x alto, 3) Para la renderización de la imagen se realizó una salida del buffer completo sobre una imagen BMP output-salida, así se obtuvo el filtrado y aplicación de los cambios sobre la imagen de entrada.

Palabras clave: Paralelismo, rendimiento, procesamiento, imágenes.

ABSTRACT

The computational vision area allows to improve images, using the technique of parallelization of the computation process, using the advantages of the graphic processing unit present in the modern graphic cards, from the Intel HD Graphics, to the powerful NVidia GTX 1080 with its 2560 processors that will allow to calculate any complex routine, if this has a correctly adapted scheme for this type of computation, it is for this reason that the objective was: To develop a software based on techniques of parallelization by means of the GPU for the improvement and modification of images. The research design is part of the descriptive design, the sample consisted of 20 images of different formats. The methodology used was the Scrum Development Methodology, with the following phases: the problem identification process, UML Unified Modeling language and finally for the testing stage, the results were: 1) For the segmentation of the frames it has been made use of the external software that obtains the entries in a collection of files separated by a guide index, 2) A kernel-based library has been generated with the name BoxFilterGL_Kernels.cl, which is responsible for performing the iterations and operations of filtering of the matrices that are processed and generated in the iterations of width x height, 3) For the rendering of the image, an output of the complete buffer was made on a BMP output-output image, thus the filtering and application of the changes was obtained over the input image.

Keywords: Parallelism, performance, processing, images.

CAPITULO I

INTRODUCCIÓN

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (en paralelo). Hay varias formas diferentes de computación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas.

El despliegue de imágenes se ha realizado por rastreo desde la década de los 70s, esto internamente supone un costo computacional de enorme costo pues recorrer una imagen en ese entonces de 320x200 pixeles no suponía más de 64000 iteraciones, pero para las imágenes actuales con resoluciones mínimas de 1280x720 con 921600 iteraciones por Frame y/o imagen y resoluciones de 4K esta cantidad de iteraciones se cuadruplica y el costo de procesamiento para el CPU es técnicamente insostenible.

La computación paralela permite diseccionar labores de computo extenso como el caso de nuestra investigación, en lo que en los años 90s se conocía como Threads o hilos de procesos, que ahora concurren como tal en un procesador integro en este caso en la Unidad Grafica de Procesamiento o GPU, que ha desarrollado la capacidad de integrar básicamente desde 96 procesadores con la NVidia GT 650 a la suma y pasar a la NVidia GTX 1080 de 2560 procesador en el caso del fabricante de tarjetas de video.

1.1. Formulación del Problema

¿De qué manera se puede optimizar el desempeño de la Técnica de Mejoramiento de Imágenes Mediante la Paralelización en GPU?

1.2. Objetivos

1.2.1. Objetivo General

Desarrollar un software basado en técnicas de paralelización mediante el GPU para la mejora y modificación de imágenes.

1.2.2. Objetivos Específicos

- Segmentar una secuencia de imágenes y generar matrices de procesamiento.
- Implementar y añadir librerías que procesan imágenes en forma secuencial para adaptarlas al método paralelo usando la aceleración por GPU.

- Renderizar el modelo final bidimensional y aceleración de secuencias mediante paralelización en GPU.

1.3. Justificación

Se requiere un esquema de uso de la unidad grafica de proceso y su potencia para realizar tareas en paralelo, a diferencia de los micro-procesadores que realizan la ejecución de sentencias en modo secuencial o bus realizando un pequeño lote de sentencias de un task y de otro hasta finalizar la lista de tasks o tareas, pero en definitiva no hay procesamiento paralelo sino secuencial.

El GPU dependiendo de la cantidad de cores (núcleos de procesamiento) que posee realiza un cálculo o procesamiento de sentencias en modo paralelo aprovechando una función kernel en el caso de CUDA y OpenCL, las que son compiladas y ejecutadas, pero para diferencia a un proceso que corre en un core de otro se usa un esquema de separación por Matriz, Bloques e Hilos (Grids, Blocks & Threads), de este modo puede desmenuzar el procesamiento que generalmente va de un punto referencial y una cantidad corta de procesamiento o calculo, esta potencia la usaremos para realizar el cálculo del ruido del contenido de los fotogramas de las imágenes que procesaremos, pero como en la actualidad tenemos imágenes en 2K y 4K debe tenerse en cuenta que requerirá un enorme tiempo de procesamiento para dichas imágenes, con el uso de la potencia de cálculo de la GPU se reducirá el tiempo.

CAPITULO II

REVISIÓN DE LITERATURA

2.1 Antecedentes

Hennessy y Patteron (2006). Los tiempos de la GPU son mejores que los de la CPU cuando el uso de la CPU está cerca de los 9 núcleos o menos, pero cuando los algoritmos paralelos en CPU utilizan más de 9 los tiempos son mejores en CPU que en GPU. La estrategia usada para paralelizar en GPU posee mayor granularidad que en CPU debido a que la GPU posee una mayor cantidad de núcleos y por tal razón una mayor cantidad de hilos, lo cual permite pensar que si se aumentan la cantidad iteraciones o la cantidad de operaciones estadísticas; es probable decir que la aceleración en GPU podría llegar a dar mejor que en CPU. El problema que se trató en este capítulo es idóneo para paralelizar debido a que el análisis básico estadístico que se hizo es independiente para cada proceso, por tal razón según los resultados expuestos se lograron aceleraciones; que sentaron las bases de varios conceptos de la computación gráfica actual, como las transformaciones de vértices y el uso de texturas. Hoy en

día, las GPU accesibles han sobrepasado largamente las capacidades de aquellos sistemas exclusivos.

Guaycochea (2011). El trabajo desarrollado no deja de lado las características actuales del hardware donde se ejecuta la implementación resultante. Es por eso que se realiza una breve exposición de la evolución del hardware gráfico, y luego se hace hincapié en las características de las tarjetas gráficas o GPUs programables de la actualidad. Las GPUs actuales responden al modelo llamado Shader Model. Dentro de las nuevas características introducidas se destaca una nueva etapa llamada Geometry Shader. Tras estudiar la funcionalidad brindada por esta etapa, y analizar el rendimiento obtenido al utilizarla, se concluye que la misma no es útil para el renderizado de terrenos.

El uso de esta etapa degrada el rendimiento general de una técnica. Sin embargo, sí se encontraron útiles otras características, como la unificación de las unidades de procesamiento y la posibilidad de utilizar arreglos de texturas en combinación con la técnica de instanciación. Por otro lado, la solución desarrollada brinda una técnica de renderizado de terrenos apta para aplicaciones interactivas o tiempo real, como ser videojuegos o simuladores de vuelo. Con mayor precisión, la técnica apunta a ser utilizada en simuladores de vuelo, donde los requerimientos son más estrictos. La visualización producida debe representar fielmente terrenos del mundo real logrando una inmersión total del usuario dentro del simulador. Por ende, la técnica propuesta no muestra distorsiones ni artefactos, brindando una representación visualmente exacta del terreno.

Ponce (2011). Animación basada en física de partículas usando GPU, muestra las ventajas del uso del mismo mediante la paralelización aprovechando la tarjeta gráfica y la cantidad de procesadores que traen actualmente, así reducir el tiempo de proceso que requerimos para obtener resultados de renderización o cálculo de información y obtención de resultados visuales.

Ponce y Yalmar (2011). Implementando algoritmos utilizando GPUs, las ventajas que tiene son que la segmentación conseguida es precisa y que no añade sobrecoste al hacer la división o unión del contorno. Sin embargo, el algoritmo se basa en la resolución de ecuaciones diferenciales, lo que supone que el algoritmo sea costoso y lento. Se han desarrollado varias implementaciones en GPU, también sobre imágenes 3D, y en proyectos de carrera también se utilizan GPUs. De la misma manera, también se han desarrollado paralelizaciones del algoritmo para arquitecturas SMP (Symmetric Multi-Processing) o para arquitecturas de memoria distribuida con MPI (Message Passing Interface).

Hernández y Vargas (2013). Los modelos de memoria para la programación paralela fueron caracterizados en aras de distinguir sus particularidades, describiéndose los 5 tipos de memorias disponibles en CUDA. Se constató el acelerado ritmo de desarrollo de CUDA y de las aplicaciones en el área de la Bioinformática, donde se revisaron y comentaron varias aplicaciones. Se caracterizaron las variables a medir para el mejor desarrollo de una aplicación paralela a partir del estudio de varias herramientas disponibles para el análisis del rendimiento de

aplicaciones con CUDA, haciendo énfasis en el Visual Profiler Tools (y su versión de línea de comandos).

Se implementa el cálculo de la matriz de distancia de un conjunto de secuencias, que es la base de todo análisis filogenético. Como primera aproximación de solución al problema cumple con los elementos fundamentales de la programación usando CUDA.

Ruiz (2015). Clasificación Tisular en GPU, Respecto a las arquitecturas gráficas objeto de nuestro análisis, comenzamos nuestra andadura con modestas GeForce, prosiguiendo con Quadro de gama alta, y concluyendo con Tesla de propósito general, justo donde muchos se iniciaron en el mundo GPGPU (Computación de propósito general en unidades de procesamiento de gráficos); para tomar el relevo. La longevidad del algoritmo de detección de tumores nos ha permitido comparar evolutivamente todas estas arquitecturas, el registro de imágenes, ilustrar el beneficio de apoyarse en una popular librería como cuFFT, y los momentos de Zernike, desvelar las exigencias para optimizar el código en generaciones venideras (en nuestro caso, Fermi y Kepler).

Castillo, R (2016). El tiempo empleado por la GPUs en el procesamiento de los datos ráster, en ocasiones y teniendo en cuenta el nivel de resolución de los datos, no satisface las necesidades de los usuarios de SIG, en entornos computacionales con bajas prestaciones. A pesar de que se aprecia un avance científico en el área del procesamiento ráster, en su mayoría, las propuestas estudiadas, modelan el procesamiento paralelo en torno a la CPU a través de la utilización de

estrategias paralelas mediante OpenCL o a la GPU mediante CUDA, sin explotar al máximo la heterogeneidad entre diferentes plataformas de cómputo.

La aproximación satisface este factor, sin embargo el empleo de CUDA limita su campo a las GPU del fabricante NVIDIA. Adicionalmente, la configuración de hardware del entorno computacional empleado para el despliegue de la solución propuesta no es la más asequible, en términos de costo, a las posibilidades de las organizaciones cubanas especializadas en el desarrollo de software en GPUs y dedicadas al procesamiento de información geoespacial.

Teniendo en cuenta estos elementos, se inclina por el diseño de un método que permita garantizar la heterogeneidad entre diferentes plataformas de cómputo, tomando como base la arquitectura paralela de las GPU, además del bajo coste y las potencialidades de procesamiento que pueden llegar a alcanzar los sistemas distribuidos por computación voluntaria, con el fin complementar los medios disponibles para el procesamiento ráster.

2.2 Marco Teórico

2.2.1 Arquitecturas Gráficas

La popularidad alcanzada por las GPUs como procesadores gráficos programables se debe fundamentalmente a su bajo coste y al gran rendimiento alcanzado en muchas aplicaciones de propósito general. Sin embargo, la naturaleza de las arquitecturas gráficas es bien diferente

respecto a la actual evolución de estos procesadores dentro de la computación de altas prestaciones, (Culler, 1997).

Esta introducción a la arquitectura de la GPU se desarrolla en orden cronológico para conocer su origen y metamorfosis, lo que nos permitirá entender mucho mejor el modelo de programación implementado y su adecuación a ciertos tipos de algoritmos.

La evolución de las características implementadas para mejorar el rendimiento de arquitecturas gráficas cuando éstas se orientan a propósito general, Castillo (2016).

2.2.2 OpenCL

OpenCL (Open Computing Language) es el primer lenguaje estándar abierto, gratuito y multi-plataforma para la programación paralela de procesadores modernos usados en computadoras personales, servidores, dispositivos portátiles y embebidos. Los objetivos de OpenCL son: explicar de manera general como funciona un programa OCL, para ello veremos de que trata OCL. Castillo R. (2016)

- **OCL** (Object Constraint Language) es un lenguaje para la descripción formal de expresiones en los modelos UML. Fue adoptado en octubre de 2003, como parte de UML. Sus expresiones de este lenguaje pueden representar invariantes, precondiciones, postcondiciones, inicializaciones, guardias, reglas de derivación, así como consultas a objetos para determinar sus condiciones de estado.

OCL fue inicialmente desarrollado por IBM. Este lenguaje no causa efectos laterales, de manera que la verificación de una condición, que se presupone una operación instantánea, nunca altera los objetos del modelo. Su papel principal es el de completar los diferentes artefactos de la notación UML con requerimientos formalmente expresados. Espíndola, y Vargas, (2013).

Lenguaje OCL mejora en gran medida la velocidad y capacidad de respuesta para una amplia gama de aplicaciones en numerosos sectores del mercado de juegos y entretenimiento, así como software cuántico y médico.

Los objetivos de OpenCL son: explicar de manera general como funciona un programa OCL, justificar la computación paralela y crear una aplicación capaz de ejecutar este tipo de código en algunos de los lenguajes de programación más populares (ANSI C, Java y C#). Los ejemplos que se pueden usar para su programación (ANSI C, Java y C#) para poder usar OCL en Linux y Windows. Inicialización de los dispositivos OCL. Manejo de bueras (escribir y recuperar datos del dispositivo OCL). Programación básica de un Kernel. Ejecución de las funciones kernel. El uso de workers (hilos). Hennessy, (2006).

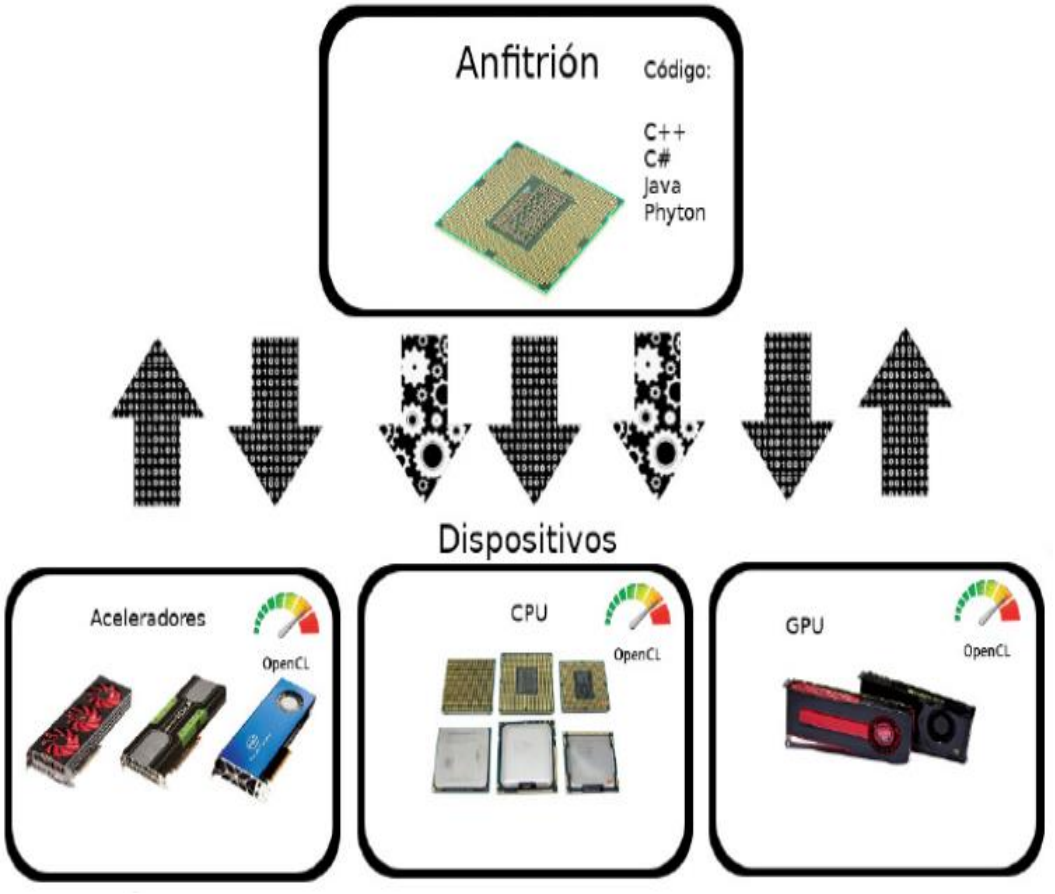


Figura 1 Modo de trabajo de la interacción entre OpenCL y las tarjetas gráficas con GPU

2.2.3 El Procesamiento Gráfico (GPU)

Es el encargado de mover los vértices iniciales y agruparlos en polígonos, para posteriormente rasterizarlos y transformarlos en píxeles aplicando texturas y colores, que finalmente se mezclará con los objetos ya existentes en la escena. En el argot gráfico, este proceso recibe el nombre de renderización, y sigue vigente en esencia, aunque las transformaciones de vértices y píxeles son programables mediante sombreadores (shaders) (2001), cuya polivalencia y unificación originaron CUDA (2006). La Figura 2.2.3 ilustra el proceso en su conjunto. Martínez, (2008).

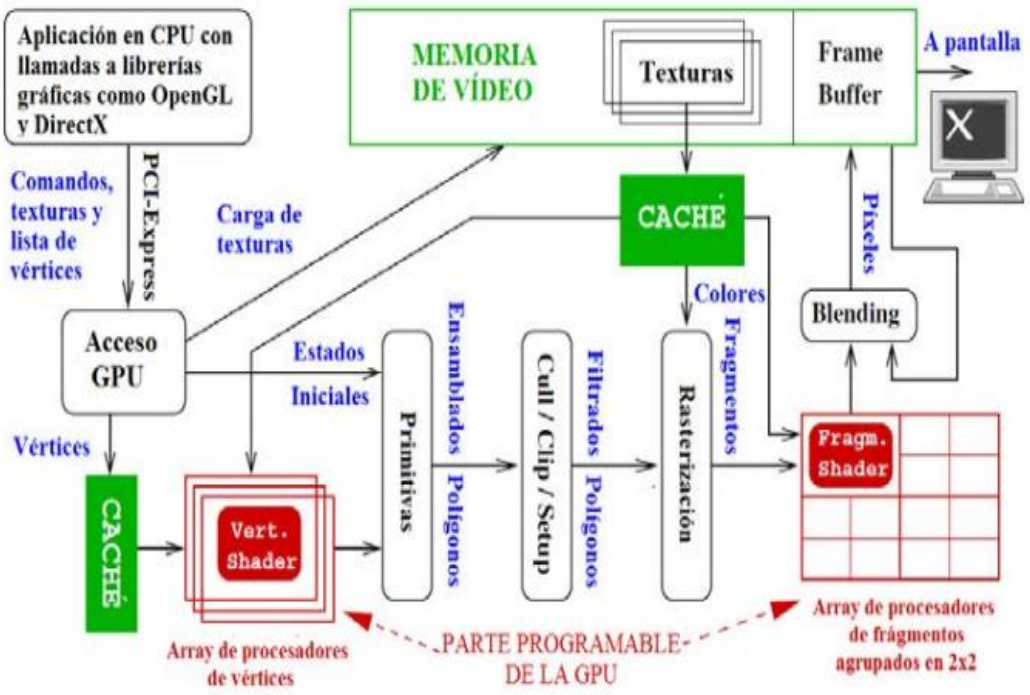


Figura 2 Como se firma electrónicamente.

La memoria de vídeo (VRAM), que aloja todos los datos necesarios para llevar a cabo la renderización, desde los vértices y sus atributos al comienzo del cauce de segmentación, hasta su presentación final en el frame buffer, que no es más que una representación interna de lo que vemos en pantalla. Además de estos dos elementos, se alojan en memoria de vídeo los mapas de texturas y el zbuffer1 o buffer de profundidad.

Los procesadores de señal, luz y color (RAMDAC), cuya principal misión es convertir la información digital almacenada en el frame buffer en señales aptas para los dispositivos externos de vídeo, analógica para VGA o digital en fechas más recientes para DVI o HDMI. García, (2012).

Los conectores externos, que permiten obtener el resultado de la información procesada en diferentes fuentes o formatos. Entre ellos

podemos citar como más populares DVI, VGA, HDMI, Super-VHS, HD-TV y vídeo compuesto.

La BIOS de vídeo. Es el firmware de configuración de la tarjeta gráfica. Esta función ha ido delegándose hacia los drivers de la tarjeta gráfica, existiendo utilidades que permiten manipular la configuración desde del propio sistema operativo (SO).

Los buses de comunicación, siendo el bus de memoria de vídeo y el bus que conecta el zócalo gráfico con la CPU (AGP o PCI-Express) los dos más relevantes. El primero comunica la GPU con su memoria de vídeo, y el segundo con el procesador central o CPU, normalmente a través del puente norte, el chip de la placa base que implementa sus controladores más importantes.

Los disipadores encargados de refrigerar todo el conjunto y mantener una temperatura de trabajo adecuada. Suelen ser los más aparatosos y los que ocultan el resto de elementos ya mencionados.

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 cientos de procesadores shader unificados que son capaces de actuar como vertex shaders, y como pixel shaders, o fragment shaders. De este modo, una frecuencia de reloj de unos 1-1,5 GHz (el estándar hoy en día en las GPU de más potencia), es muy baja en comparación con lo ofrecido

por las CPU (3,8-4 GHz en los modelos más potentes, no necesariamente más eficientes), se traduce en una potencia de cálculo mucho mayor gracias a su arquitectura en paralelo. García, (2012)

2.2.4 Algoritmos de Reducción en GPU

El secreto evidente de la buena aceleración que se obtiene en los resultados de las implementaciones presentadas en este trabajo, es la absoluta independencia de los cálculos entre cada una de las partículas simuladas. Por ello, no existe dependencia de datos en la GPU y prácticamente se asegura que todos los accesos a memoria son coalescentes, propiedad que mejora las prestaciones computacionales de la tarjeta gráfica, al mejorar el trasiego de información interno entre la memoria y los núcleos de la GPU.

Cálculos estadísticos para obtener estadísticas sobre el valor de las componentes de la velocidad de los electrones (perpendicular y paralela al campo magnético): caracterizado porque se deben siempre tener en consideración el valor de todas las partículas (por lo tanto, de todos los hilos participantes en el kernel). Espíndola y Vargas. (2013)

Recuento del número de electrones que en cada iteración pasan a ser electrones fugitivos (runaway), por exceder la velocidad crítica de confinamiento: se caracteriza porque, en condiciones normales del plasma, se trata de una baja proporción de electrones (hilos) los que deben computarse. Estas reducciones se pueden abordar al menos desde dos perspectivas diferentes:

Mediante algoritmos propios de reducción, generalmente con coste computacional de orden O (log2(N)): Son los indicados para cuando el porcentaje de hilos (partículas) a participar en la reducción es elevada respecto al total. Mediante operaciones atómicas que cada partícula realiza sobre una variable global común: Son los indicados cuando el porcentaje de hilos (partículas) a participar es despreciable frente al conjunto total de partículas.

La elección del método es muy importante por su impacto en el tiempo de computación. Esta decisión es un claro ejemplo de las dificultades que suelen tener los programadores que no son expertos en ingeniería informática, dado que frecuentemente se obvia la realización de un estudio previo o aunque sea posterior, experimental de costes del algoritmo a implementar. Navarro, (2015).

2.2.5 Arquitectura Gráfica NVIDIA CUDA G80

La computación de propósito general en procesadores tuvo un pronunciado crecimiento gracias a una arquitectura que cambió la relación entre el cauce de segmentación gráfico programable a nivel lógico y el procesador físico. En 2006, surgió una nueva arquitectura de GPU basada en la idea de unificar los procesadores de vértices y píxeles, de manera que las unidades de procesamiento pudiesen computar todas ellas. Desde un punto de vista gráfico, dicha transformación tiene como objetivo reducir el desequilibrio de carga entre los procesadores de vértices y de píxeles. Debido a este desequilibrio, muchas unidades funcionales estaban ociosas durante grandes periodos de tiempo. En la arquitectura unificada, sólo hay

un tipo de unidad de procesamiento que es capaz de ejecutar operaciones de vértices y píxeles. De esta manera, el cauce de segmentación se vuelve circular en torno a la unidad de procesamiento para cada tipo de operación, y la potencia de cálculo que no requieran los vértices en un momento dado pueden aprovecharla los píxeles (y viceversa). En el ámbito GPGPU, la arquitectura unificada ofrece al programador más unidades de procesamiento y una mayor funcionalidad. Sin embargo, la revolución en este sentido viene por parte de la arquitectura y nuevo paradigma de programación CUDA de Nvidia. Navarro, (2015).

El propósito de CUDA es crear una implementación para la arquitectura unificada basada en el rendimiento y en la facilidad de programación, abstrayendo los detalles de segmentación que son patrimonio exclusivo del entorno gráfico. La primera GPU que implementó la arquitectura unificada con las directrices de CUDA fue la G80 (ver Figura 2.2.5). Sobre ella, Nvidia ha ido incorporando nuevas funcionalidades agrupadas en forma de generaciones: Fermi es la segunda generación (2010), Kepler la tercera (2012), Maxwell la cuarta (2014) y Pascal la quinta (2016).

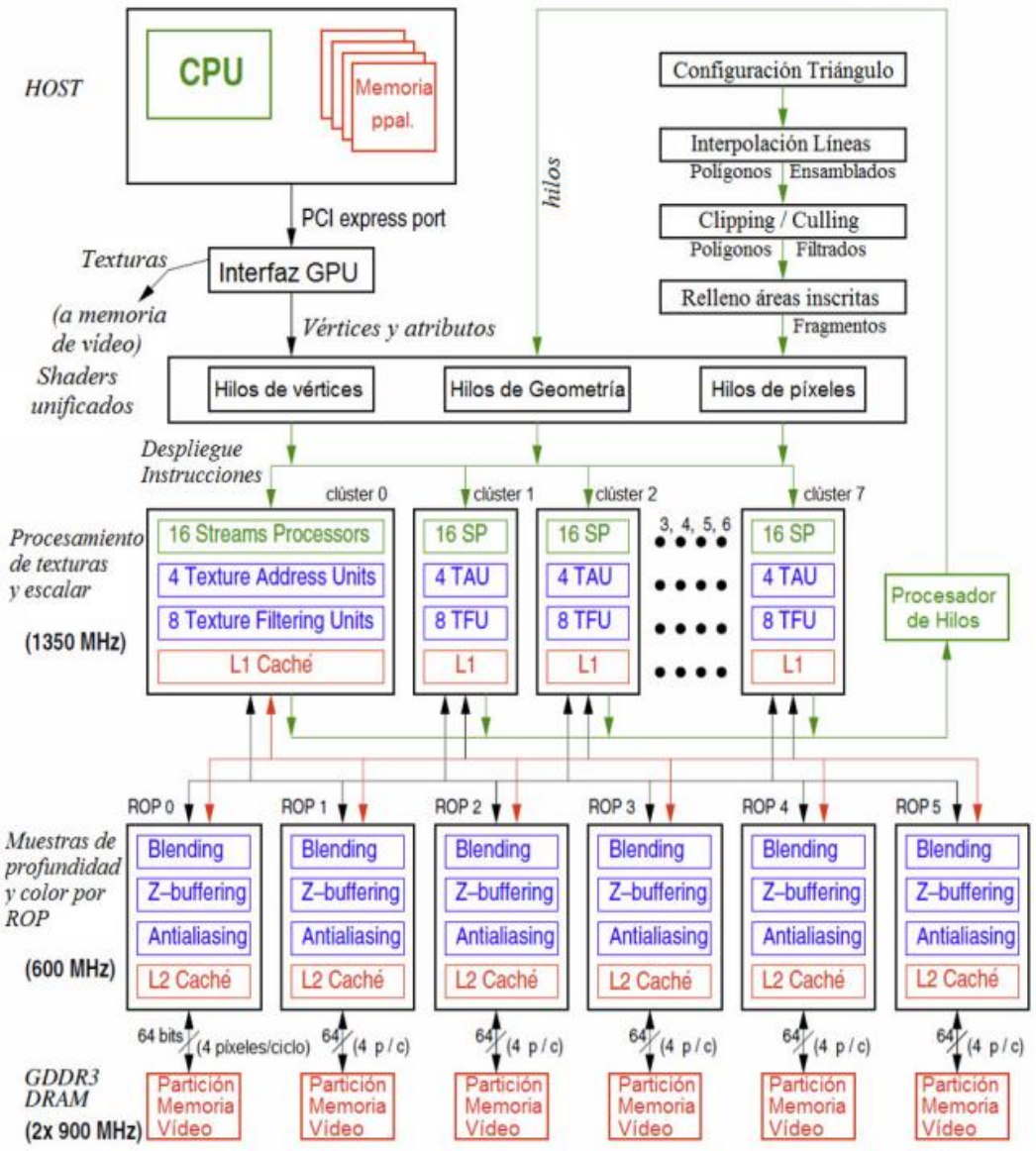


Figura 3 Diagrama de bloques de la arquitectura Nvidia CUDA G80.

2.2.6 Registro de la imagen en GPU

El flujo de trabajo del algoritmo de registro se recoge en la Figura 2.2.6 Desde el punto de vista del rendimiento, el conjunto de Transformadas Rápidas de Fourier (FFTs) empleadas para procesar la Norma de control de calidad (NCC), en un emparejamiento preciso del registro no rígido, es la parte más importante por ocupar la mayor parte del tiempo de ejecución.

Por ejemplo, en nuestro experimento de registro de un par de imágenes de 23 K x 62 K píxeles, más del 60 % del tiempo de ejecución se dedica al procesamiento de la NCC. (Norma de Control de Calidad). Navarro, (2015)

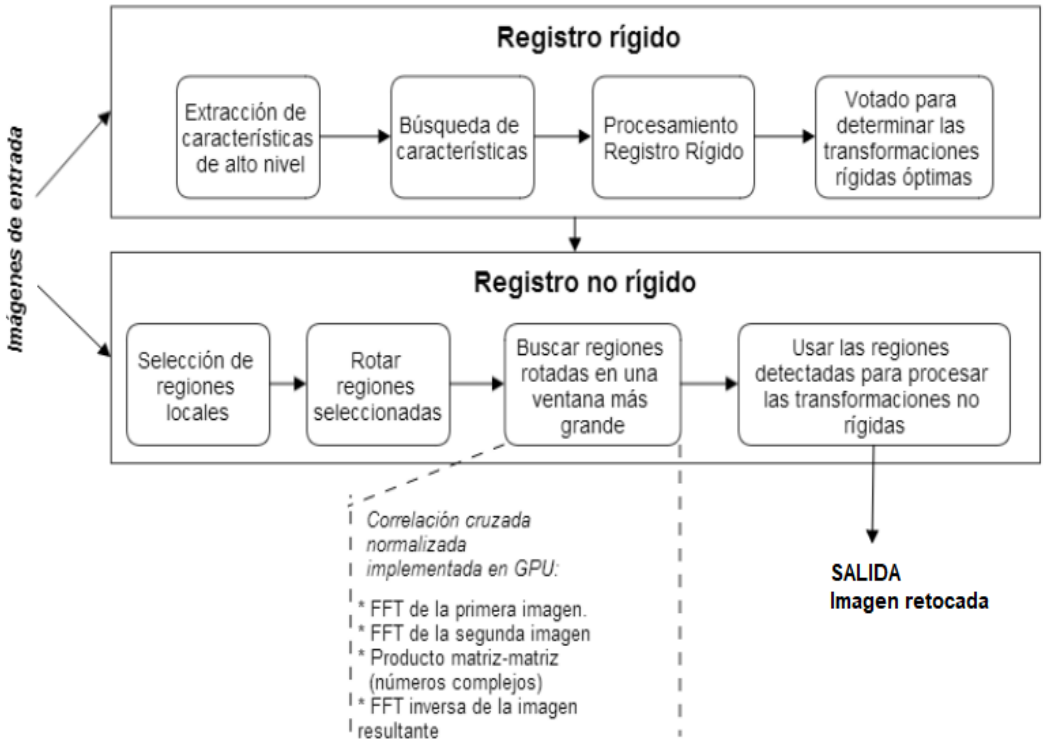


Figura 4 Flujo de trabajo del algoritmo de registro que consta de dos fases: el registro rígido y no rígido.

Cada caja representa una operación independiente que puede ser directamente paralelizada. La fase computacionalmente más exigente se ejecuta en la GPU para conseguir una ejecución mucho más rápida.

CAPITULO III

MATERIALES Y METODOS

3.1. Metodología de Desarrollo SCRUM

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

Scrum también se utiliza para resolver situaciones en que no se está entregando al cliente lo que necesita, cuando las entregas se alargan

demasiado, los costes se disparan o la calidad no es aceptable, cuando se necesita capacidad de reacción ante la competencia, cuando la moral de los equipos es baja y la rotación alta, cuando es necesario identificar y solucionar ineficiencias sistemáticamente o cuando se quiere trabajar utilizando un proceso especializado en el desarrollo de producto. Ver en detalle cuales son los beneficios de Scrum, sus fundamentos y sus requisitos.

3.2 El Proceso

En Scrum un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones que normalmente son de 2 semanas, aunque en algunos equipos son de 3 y hasta 4 semanas, límite máximo de feedback y reflexión). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto final que sea susceptible de ser entregado con el mínimo esfuerzo al cliente cuando lo solicite.

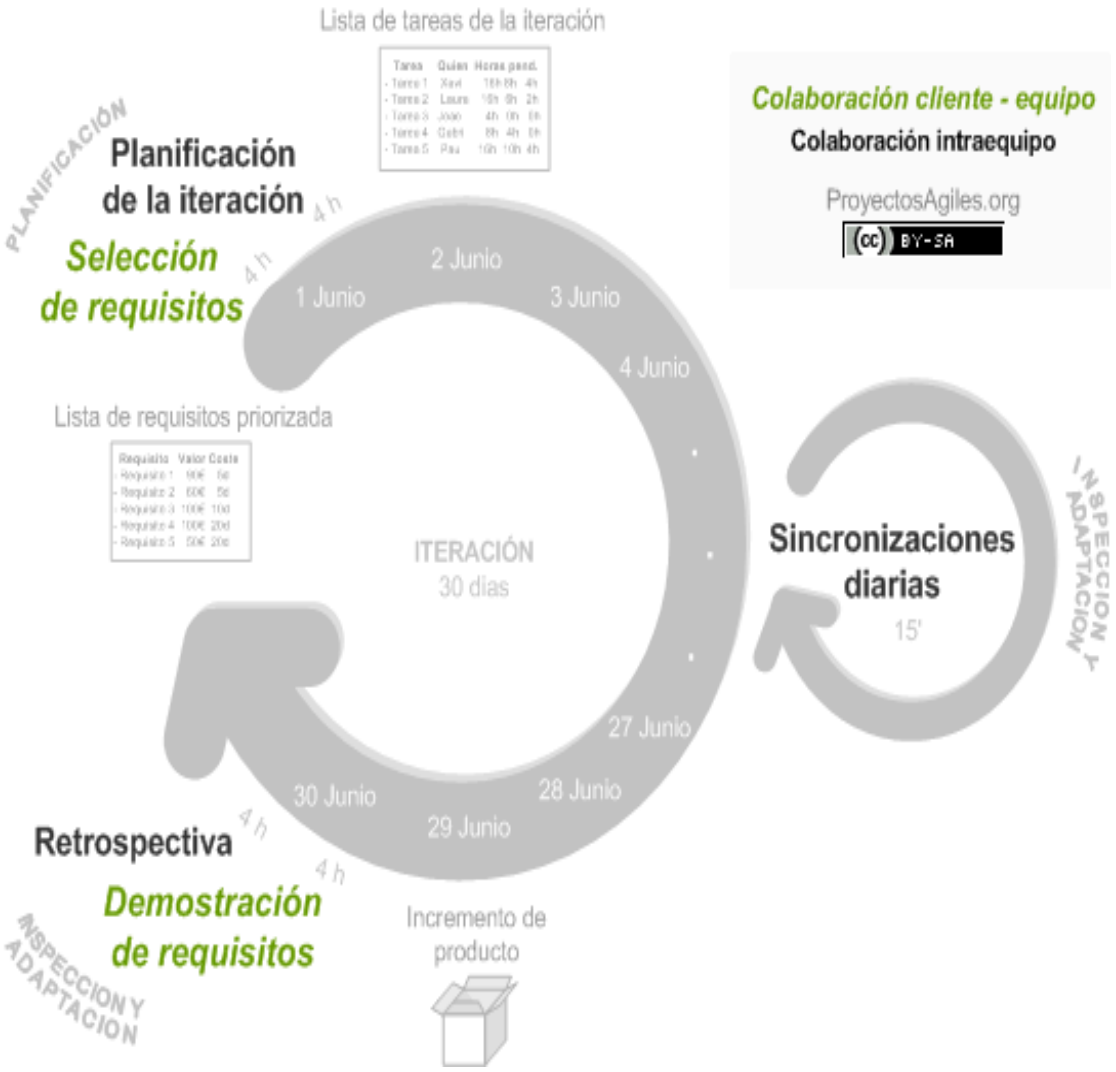


Figura 5 diagrama de proceso SCRUM.

Diagrama-proceso-scrum: El proceso parte de la lista de objetivos/requisitos priorizada del producto, que actúa como plan del proyecto. En esta lista el cliente prioriza los objetivos balanceando el valor que le aportan respecto a su coste y quedan repartidos en iteraciones y entregas.

Las actividades que se llevan a cabo en SCRUM son las siguientes:

3.3. Planificación de la Iteración

El primer día de la iteración se realiza la reunión de planificación de la iteración. Tiene dos partes:

Primero, Selección de requisitos (4 horas máximo). El cliente presenta al equipo la lista de requisitos priorizada del producto o proyecto. El equipo pregunta al cliente las dudas que surgen y selecciona los requisitos más prioritarios que se compromete a completar en la iteración, de manera que puedan ser entregados si el cliente lo solicita.

Segundo, Planificación de la iteración (4 horas máximo). El equipo elabora la lista de tareas de la iteración necesarias para desarrollar los requisitos a que se ha comprometido. La estimación de esfuerzo se hace de manera conjunta y los miembros del equipo se auto asignan las tareas.

3.3.1 Ejecución de la Iteración

Cada día el equipo realiza una reunión de sincronización (15 minutos máximo). Cada miembro del equipo inspecciona el trabajo que el resto está realizando (dependencias entre tareas, progreso hacia el objetivo de la iteración, obstáculos que pueden impedir este objetivo) para poder hacer las adaptaciones necesarias que permitan cumplir con el compromiso adquirido. En la reunión cada miembro del equipo responde a tres preguntas:

¿Qué he hecho desde la última reunión de sincronización?

¿Qué voy a hacer a partir de este momento?

¿Qué impedimentos tengo o voy a tener?

Durante la iteración el Facilitador (Scrum Master) se encarga de que el equipo pueda cumplir con su compromiso y de que no se merme su productividad.

- Elimina los obstáculos que el equipo no puede resolver por sí mismo.
- Protege al equipo de interrupciones externas que puedan afectar su compromiso o su productividad.

Durante la iteración, los clientes junto con el equipo refinan la lista de requisitos (para prepararlos para las siguientes iteraciones) y, si es necesario, cambian o replanifican los objetivos del proyecto para maximizar la utilidad de lo que se desarrolla y el retorno de inversión.

3.4. Inspección y Adaptación

El último día de la iteración se realiza la reunión de revisión de la iteración. Tiene dos partes:

Primero, Demostración (4 horas máximo). El equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, replanificando el proyecto.

Segundo, Retrospectiva (4 horas máximo). El equipo analiza cómo ha sido su manera de trabajar y cuáles son los problemas que podrían

impedirle progresar adecuadamente, mejorando de manera continua su productividad. El Facilitador se encargará de ir eliminando los obstáculos identificados.

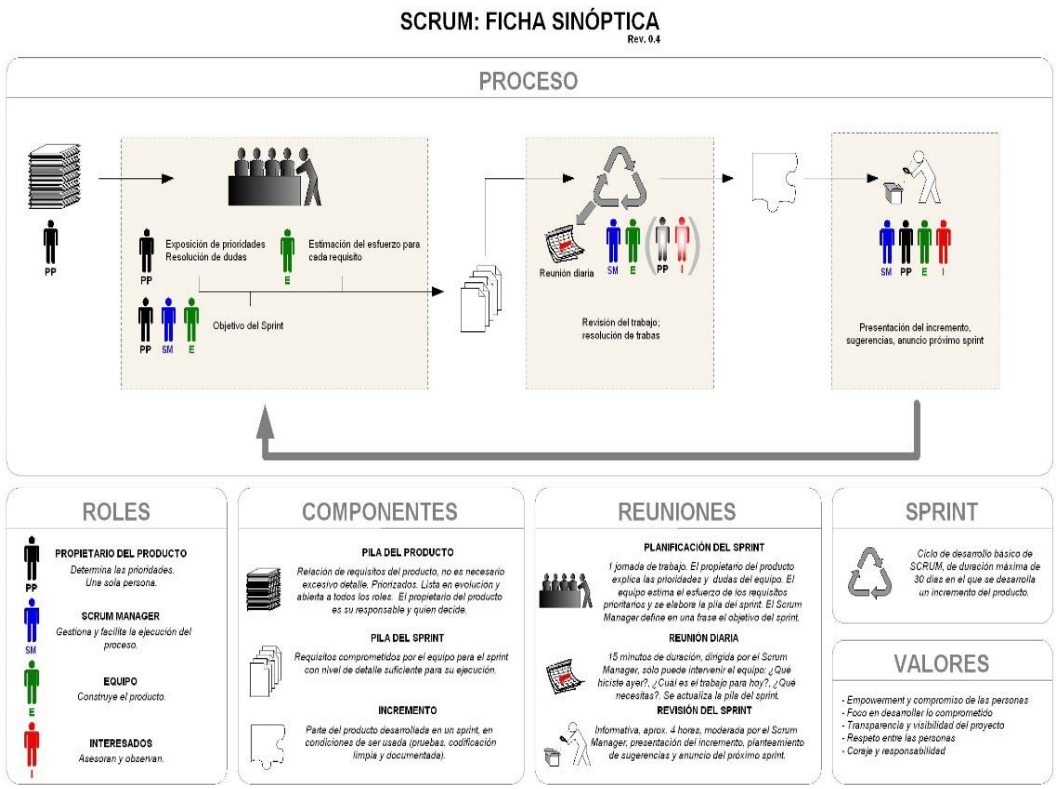


Figura 6 Inspección y adaptación

3.5 Lenguaje de Modelamiento Unificado – UML

El lenguaje unificado de modelado (UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el Object Management Group (OMG).

Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como

procesos, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Es importante remarcar que UML es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

Se puede aplicar en el desarrollo de software gran variedad de formas para dar soporte a una metodología de desarrollo de software (tal como el Proceso Unificado Racional, Rational Unified Process o RUP), pero no especifica en sí mismo qué metodología o proceso usar.

3.6 Métrica de Validación de Software ISO-9126

ISO 9126 es un estándar internacional para la evaluación de la calidad del software. Está reemplazado por el proyecto SQuaRE, ISO 25000:2005, el cual sigue los mismos conceptos.

El estándar está dividido en cuatro partes las cuales dirigen, realidad, métricas externas, métricas internas y calidad en las métricas de uso y expendido. El modelo de calidad establecido en la primera parte del estándar, ISO 9126-1, clasifica la calidad del software en un conjunto estructurado de características y sub-características de la siguiente manera:

Funcionalidad - Un conjunto de atributos que se relacionan con la existencia de un conjunto de funciones y sus propiedades específicas. Las funciones son aquellas que satisfacen las necesidades implícitas o explícitas.

Fiabilidad - Un conjunto de atributos relacionados con la capacidad del software de mantener su nivel de prestación bajo condiciones establecidas durante un período establecido.

Usabilidad - Un conjunto de atributos relacionados con el esfuerzo necesario para su uso, y en la valoración individual de tal uso, por un establecido o implicado conjunto de usuarios.

Eficiencia - Conjunto de atributos relacionados con la relación entre el nivel de desempeño del software y la cantidad de recursos necesitados bajo condiciones establecidas.

Mantenibilidad - Conjunto de atributos relacionados con la facilidad de extender, modificar o corregir errores en un sistema software.

Portabilidad - Conjunto de atributos relacionados con la capacidad de un sistema de software para ser transferido y adaptado desde una plataforma a otra.

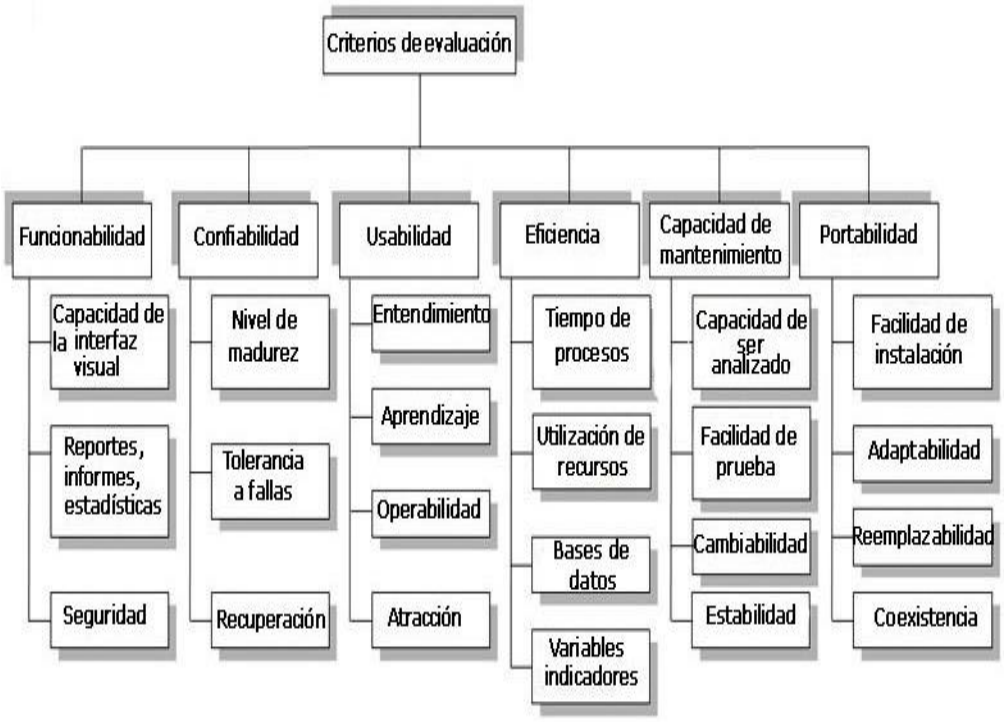


Figura 7 Cuadro de las métricas de validación

.

3.7 Tipo de Investigacion Descriptiva

El Diseño de investigación descriptiva es un método científico que implica observar y describir el comportamiento de un sujeto sin influir sobre él de ninguna manera. Los estudios descriptivos son aquellos que estudian situaciones que generalmente ocurren en condiciones naturales, más que aquellos que se basan en situaciones experimentales Por definición, los estudios descriptivos conciernen y son diseñados para describir la distribución de variables, sin considerar hipótesis. Un estudio descriptivo es un tipo de metodología a aplicar para deducir un bien o circunstancia que se esté presentando; se aplica describiendo todas sus dimensiones, en este caso se describe el funcionamiento del software que se desarrolla.

3.8 Los Alcances y las Limitaciones

A. Los Alcances:

La tracendencia de esta investigación radica en permitir concienciar a los profesionales de la Programacion, específicamente a los programadores usando GPUs; sobre la importancia de realizar estudios de programación en paralelo usando GPUs, para evaluar y brindar servicio de software de calidad. Además presentar un modelo metodológico para ser aplicado por los demás Programadores en el área.

B. Las Limitaciones:

En el desarrollo del presente trabajo de investigación se presentan las siguientes limitaciones:

- El costo del equipo, ya que para realizar el presente trabajo de investigación se necesita maquinas que tengan tarjeta gráfica de video GPUs; o por lo menos con una máquina que tenga una Tarjeta Gráfica separa NVIDIA GEFORCE.

- Escasez bibliográfica en cuanto a la programación paralela usando GPUs, ya sea en físico o mediante la WEB; Existe mediante la WEB pero la limitante es que esta en el idioma inglés.

- Presupuesto: según lo proponga el investigador.

A continuación se detalla el costo y presupuesto utilizado en el presente trabajo de investigación

Detalle	Costo estimado
Bibliografía sobre volumetrización.	1050.00
Laptop LENOVO ideapad Z470 procesador Intel(R) core(TM) i5-2410M CPU 2.3GHz ram 4GB	3000.00
Tarjeta grafica NVIDIA GeFORCE 950 GTX (80 cores)	500.00
Asesoramiento externo	1200.00
Imprevistos	1000.00
Total	6750.00

El costo del proyecto será íntegramente asumido por el ejecutor.

3.9 Ubicación del Experimento

- PAIS : Perú.
- Departamento : Puno.
- Distrito : puno.
- Universidad : Universidad Nacional Del Altiplano.
- Escuela Profesional : Ingeniería Estadística e Informática.
- Facultad : Ingeniería Estadística e Informática.

Figura 8 Ubicación del experimento.

CAPITULO IV

PRESENTACIÓN DE RESULTADOS

4.1. Diagrama de Flujo

Con este ejemplo sencillo hemos tratado de hacer explícito qué es y para qué sirve UML: un conjunto de normas que nos dicen cómo hay que representar esquemas de software. En el caso del software orientado a objetos, tendremos clases u objetos instanciados, y dispondremos de numerosos tipos de esquemas y diagramas para representar distintas cosas. Un esquema que cumple las normas UML podría tener este aspecto:

Hay que tener en cuenta que UML es un conjunto muy amplio de normas. Prácticamente nadie las conoce todas. Según la empresa o universidad, institución o centro de trabajo se usan determinados programas para crear diagramas y se conocen ciertas partes de UML, pero no el conjunto de UML.

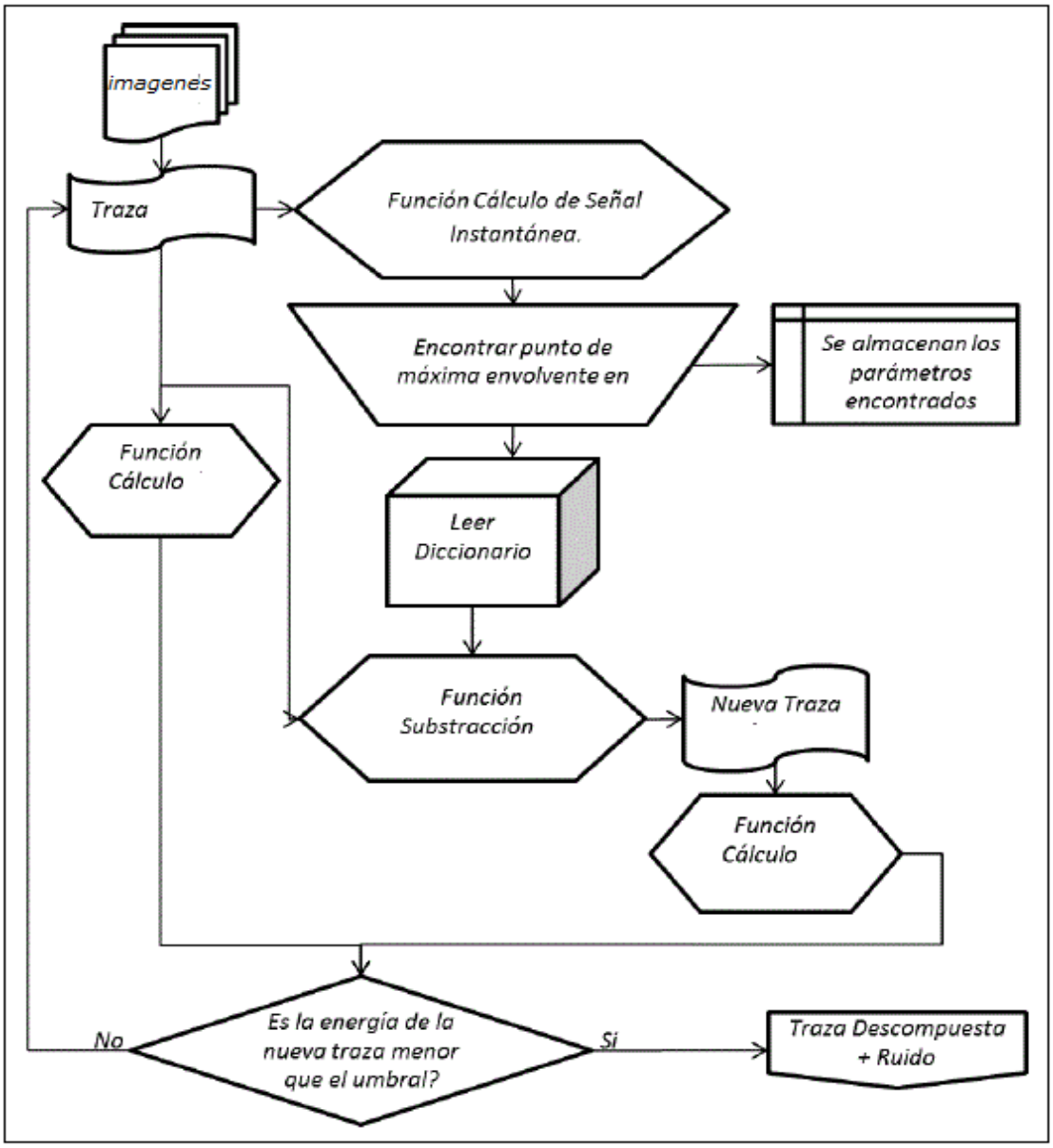


Figura 9 diagrama de flujo del filtro

Una vez obtenida la imagen pasa por estas diferentes etapas en ella se obtiene los elementos filtrados para la imagen final.

4.2. diagrama de Clases

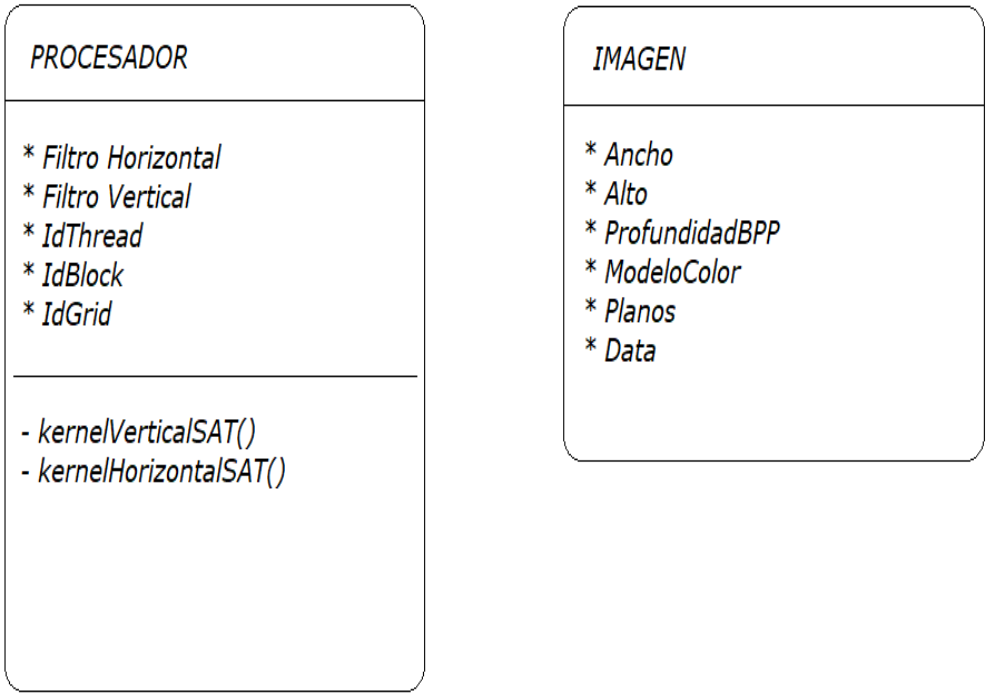


Figura 10 diagrama de clases que intervienen en el programa

Las clases definidas en el programa son basicamente la imagen como entidad que agrupa sus componentes de dimensión y calidad de la imagen, La clase Procesador que viene a ser el detalles de los componentes de las funciones kernel que son las que se ejecutan en paralelo y son invocadas en cada clock por el GPU.

4.3. OpenCL y la evolución en INTEL

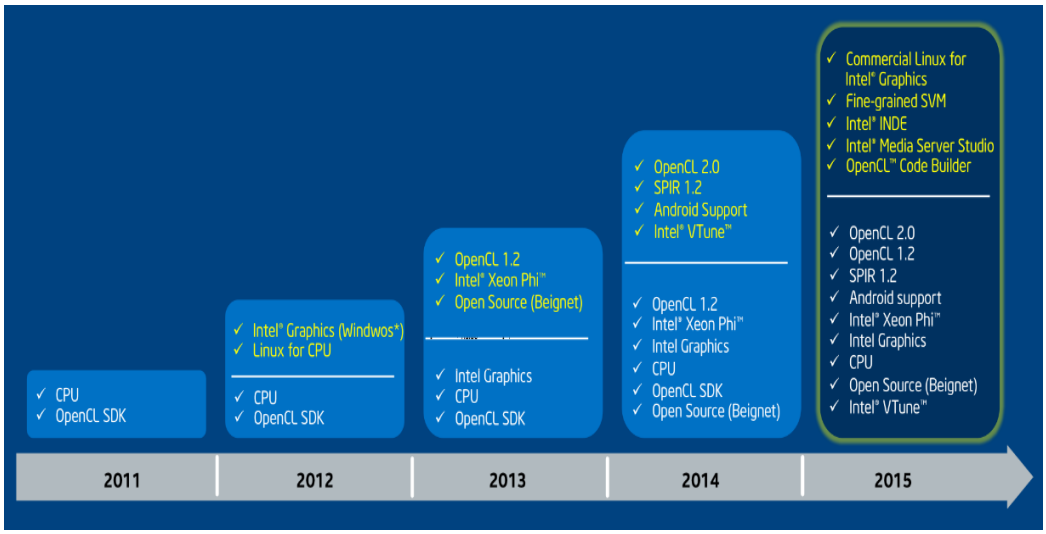


Figura 11 evolución de componentes para OpenCL

Se aprecia un incremento de características desde la versión de Intel HD Graphics 3000 a las versiones más actuales.

4.4. Interfaz de Software Demostrativo

Para la prueba se usa una imagen de tipo BMP con resolución de 1024x1024 con una profundidad de 32 bits de color RGBA en las dos etapas de la imagen.

Paso 1: se abre el software desarrollado.

Paso 2: se carga la imagen.

Paso 3: pulsamos en el botón ejecutar

Paso 4: con las teclas indicadas se realiza los filtros de mejora en FPS (fotogramas por segundo).

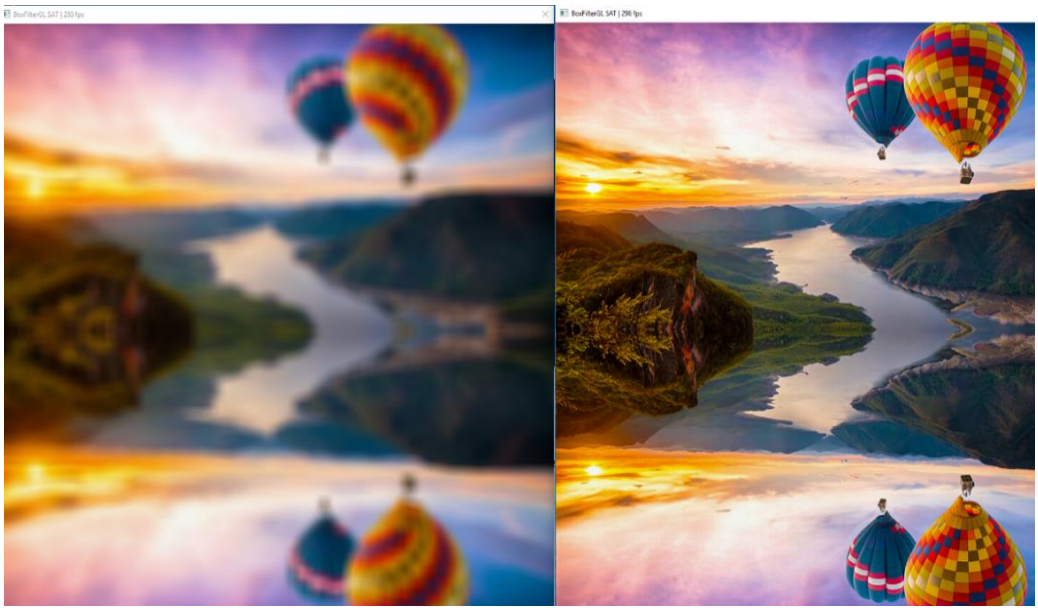
Paso 5: pulsamos el botón terminar.

Paso 6: pulsamos el botón salir.

Etapa 1: Imagen con desenfoque y una máscara de 3x3. Formato bmp

Etapa 2: Imagen filtrada y después de 3 barridos sobre la imagen. Formato bmp

Etapa 1 **Etapa 2**



Etapa 1 **Etapa 2**

Figura 12 Comparación de resultados de ambas imágenes de formato BMP

La captura fue realizada en la ejecución del software mediante impresión de pantalla y la unión se realizó mediante Paint

Para la prueba se usa una imagen de tipo JPG con resolución de 1024x1024 con una profundidad de 32 bits de color RGBA en las dos etapas de la imagen.

Etapa 1 **Etapa 2**

Etapa 1 **Etapa 2**

Comparación de resultados de ambas imágenes de formato JPG

Etapa 3: Imagen con desenfoque y una máscara de 3x3. Formato JPG

Etapa 4: Imagen filtrada y aclarada después de 3 barridos sobre la imagen. Formato JPG

Esquema de Reducción de Ruido en Imágenes

$$\frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

La aplicación de una matriz de consolación sobre la imagen fuente nos generara puntos más definidos sobre el resultado en una segunda y tercera pasada se reduce en buena cantidad la dispersión de los puntos de color sobre la imagen.

4.5. Estructura Básica de un Programa para GPU

Una vez que los datos son transferidos desde el anfitrión al dispositivo, se almacenan en el dispositivo de memoria global. Cualquier dato transferido en dirección opuesta también se almacena en la memoria global (pero esta vez en la memoria global del anfitrión). La palabra clave **_global**

(¡dos caracteres subrayados!) es un modificador que indica que los datos asociados con un cierto puntero se almacenan en la memoria global:

```
__kernel void foo( __global float *A ) { /// kernel code }
```

El archive deberá ir separado del código fuente en un archive de extensión por lo general .cl

```
#define COLS2     50

#define COLSROWS  50

#define REALTYPE  float



__kernel void matricesMul( __global REALTYPE *in1,

                           __global REALTYPE *in2,

                           __global REALTYPE *out )

{

    int r = get_global_id( 0 );

    int c = get_global_id( 1 );

    for( int cr = 0; cr < COLSROWS; cr ++ )

        out[ r * COLS2 + c ] +=  in1[ r * COLSROWS + cr ]

                              * in2[ cr * COLS2 + c ];

}
```

En general, los sistemas de memoria se diferencian bastante entre sí dependiendo de las plataformas. Por ejemplo, todas las CPU modernas soportan la recogida de datos automática, al contrario de las GPU donde este no es siempre el caso. Para garantizar la portabilidad del código, se ha adoptado un modelo de memoria abstracto en OpenCL que los programadores y distribuidores que necesitan implementarlo en el hardware real pueden adoptar. La memoria, como se define en OpenCL puede ilustrarse teóricamente como en la Figura siguiente:

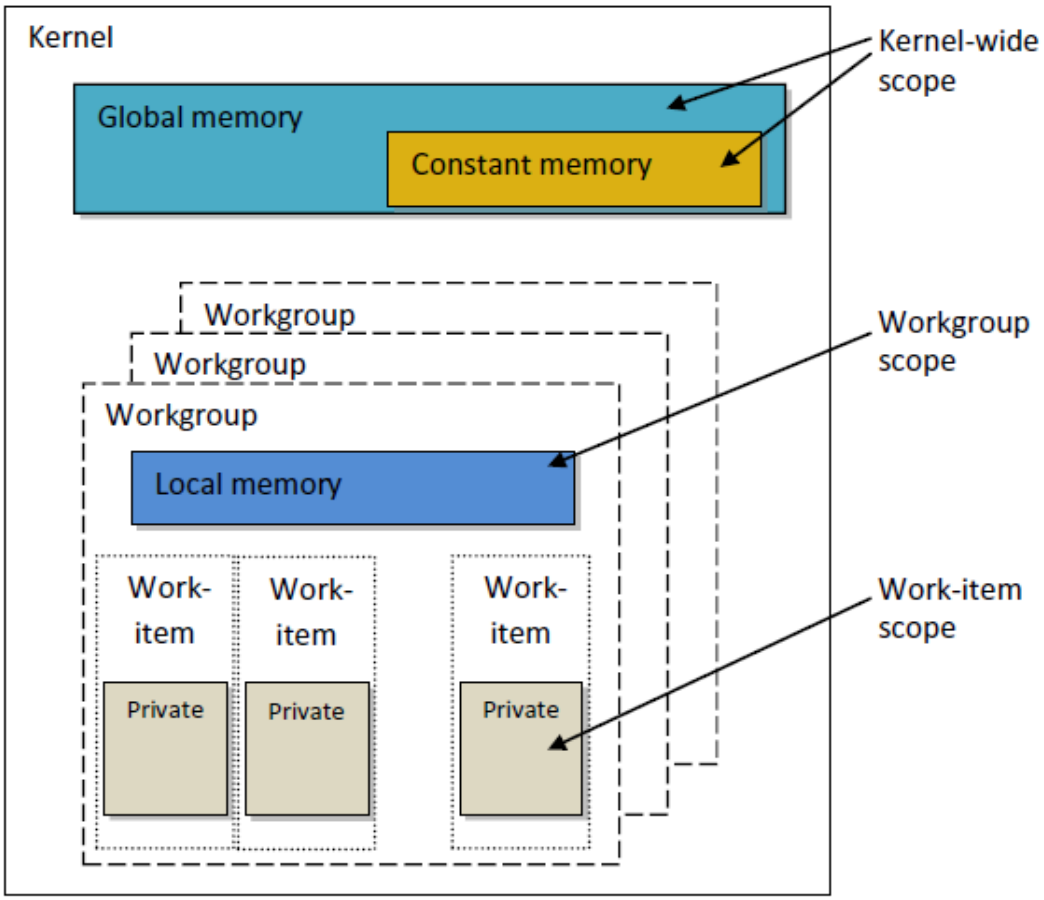


Figura 13 Modelo de memoria de video para OpenCL

4.6. Pruebas de Ejecución

Por consola de Windows.

Paso 1: abrir consola de Windows.

Paso 2: pulsamos Windows+R.

Paso 3: escribimos CMD.

Paso 4: pulsamos enter.

Paso 4: digitamos la dirección correcta a ejecutar.

`C:\Juancho\Procesador01.exe`

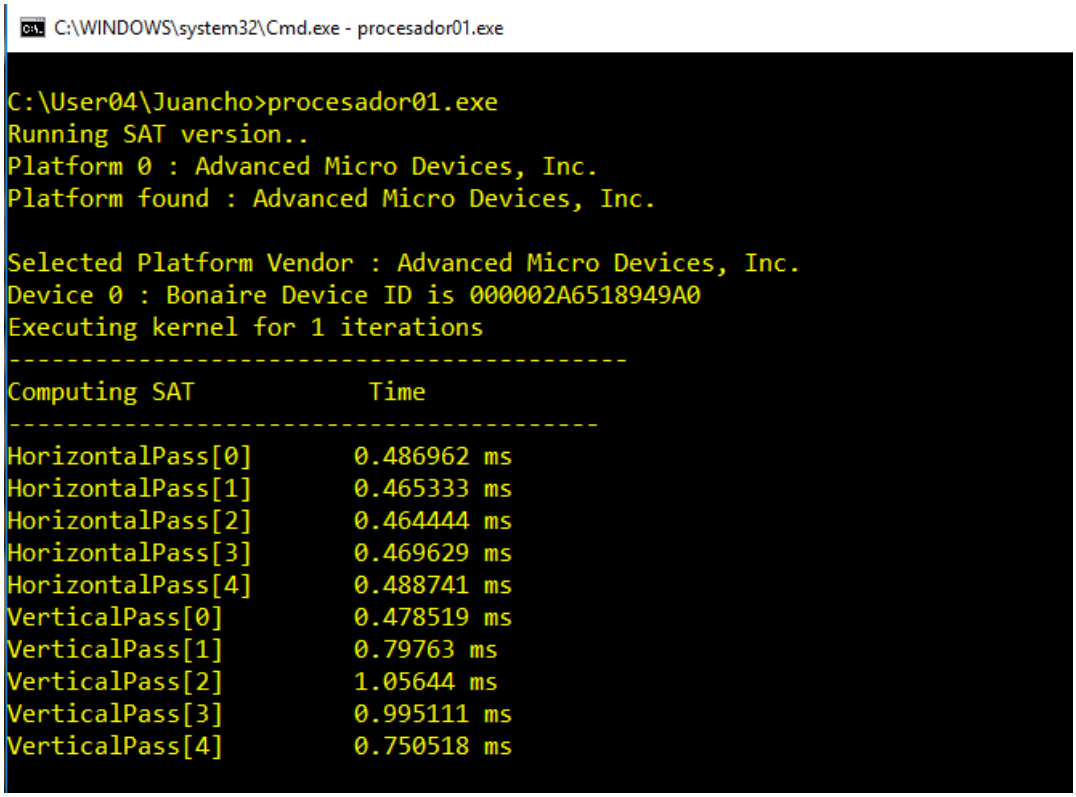


Figura 14 ejecución de la consola del programa con los resultados de tiempo de procesamiento para los barridos vertical y horizontal

CAPITULO V

CONCLUSIONES

- Para la segmentación de los frames se ha hecho uso del software externo que obtiene las entradas en una colección de archivos separados por un índice guía.

- Se ha generado una librería en modo kernel con el nombre BoxFilterGL_Kernels.cl, que es la que se encarga de realizar las iteraciones y operaciones de filtrado de las matrices que procesan y se generan en las iteraciones de ancho x alto, así preparar las imágenes en forma secuencial para adaptarlas al método paralelo, una vez procesado la matriz se envía a una matriz de salida que es la que finalmente contiene la imagen con el desenfoque limpiado.

- Para la renderización de la imagen se realizó una salida del buffer completo sobre una imagen BMP output-salida, así se obtuvo el filtrado y aplicación de los cambios sobre la imagen de entrada.

CAPITULO VI

RECOMENDACIONES

- Se recomienda realizar más estudios sobre las aplicaciones de propósito general basadas en GPU, puede tomarse como referencia la tendencia de aplicaciones GPGPU (General Purposse on GPU) las que tienen como fin mejorar el desempeño de aplicaciones que requieran gran cantidad de cálculo apoyados en la potencia del paralelizar funciones kernel las que tienen actualmente las GPUs.

- Se recomienda la implementación en un GPU CUDA, aunque la limitante sea el coste del equipo, comparado con las tarjetas de video ATI e Intel HD Graphics esto puede suponer un elevado coste, pero el desempeño es netamente más limpio y claro, así como mayor cantidad de prestaciones comparadas con las adaptaciones más generales.

CAPITULO VII

REFERENCIAS BIBLIOGRAFICAS

Castillo, R (2016). *Técnicas de programación paralela aplicadas al procesamiento de datos ráster mediante la biblioteca GDAL* (2016)

Chapman & Huang (2007). *Enhancing OpenCL and Its Implementation for Programming Multicore Systems*. Parallel Computing: Architectures, Algorithms and Applications. John von Neumann Institute for Computing, 2007.

Culler, J. (1997). *Procesamiento paralelo en clusters, rendimiento paralelo y creación de ambientes de desarrollo y ejecución de aplicaciones en clusters.* (1997)

Espíndola & Vargas (2013). *Many-Body Perturbation Theory to Second Order Applied on Confined Helium Like Atoms*. In COMPUTATIONAL AND EXPERIMENTAL CHEMISTRY: Developments and applications. CRC Press, New Jersey, 2013.

García, E. (2012). *Implementation of the electron propagator to second order on GPUs to estimate the ionization potentials of confined atoms.* J. Phys. B: At. Mol. Opt. Phys. 47 185007 (7pp) (2012).

Guaycochea, L. (2011). *Visualización de Terrenos con Tarjetas de Video Programables GPU.*

Hernández & Vargas (2013). *Four-index integral transformation in many-body perturbation theory and electron propagator to second order on GPUs for CUDA (2013).*

Hennessy, J. & Patterson, D. (2006). *Computer Architecture: Quantitative Approach [Arquitectura de computadoras*: un enfoque cuantitativo] (en inglés) (4ta edición). Morgan Kaufmann.

Martinez, C. (2008). *Técnicas de programación paralela aplicadas al procesamiento de datos ráster mediante las GPUs (2008).*

Navarro, H. (2015). *Rápidas y robustas, de las CPU´s a las GPU´s: Computación paralela y sus aplicaciones en problemas de datos paralelos utilizando arquitecturas GPU* (2015)

Patterson, D. & Hennessy, J. (2011). *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface [Organización y diseño de computadoras,* 4ta edición: La interfaz hardware/software] (en inglés) (4ta edición). Morgan Kaufmann.

Ponce A. & Yalmar (2011). *Animación Basada en Física Usando Partículas en GPUS.* Tesis Doctoral, Rio de Janeiro UFRJ / COPPE, Programa de Ingeniería de Sistemas y Computación.

Ruiz (2015). *Tissue Classification in GPU, with respect to graphic architectures (2015)*

Singh, J. & Culler, P. (1997). *Parallel computer architecture* (en inglés) (Nachdr. edición). San Francisco: Morgan Kaufmann Publ. p. 15.

Sipiran, I. & Bustos, B. (2012). *SHAPE MATCHING FOR 3D RETRIEVALAND RECOGNITION. PRISMA;* Research Group, Department of Cumputer Science, University of Chile.

Ufimtsev, I. & Martinez, T. (2008). *Graphical Processing Units for Quantum Chemistry.* Computing in Science and Engineering, 10, 26-34 (2008).

ANEXOS

ANEXO 1 CODIGO FUENTE DE LAS FUNCIONES KERNEL

```
/*
 * HorizontalSAT0 - primera pasada
 *                - input : 8 bpp imagen
 *            - output : 32 bpp data
 * HorizontalSAT
 * VerticalSAT
 * BoxFilter
 */


__kernel void box_filter(__global uint4* inputImage, __global
    uchar4* outputImage, int N)
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int width = get_global_size(0);
    int height = get_global_size(1);
    int k = (N - 1) / 2;

    if(x < k || y < k || x > width - k - 1 || y > height - k
    - 1)
    {
       outputImage[x + y * width] = (uchar4)(0);
       return;
    }

    /* N should be an odd number */
    int4 filterSize = (int4)(N * N);

    int2 posA = (int2)(x - k, y - k);
    int2 posB = (int2)(x + k, y - k);
    int2 posC = (int2)(x + k, y + k);
    int2 posD = (int2)(x - k, y + k);

    int4 sumA = (int4)(0);
    int4 sumB = (int4)(0);
    int4 sumC = (int4)(0);
    int4 sumD = (int4)(0);

    /* SAT */
    posA.x -= 1;
    posA.y -= 1;
    posB.y -= 1;
    posD.x -= 1;

    if(posA.x >= 0 && posA.y >= 0)
    {
```

```
        sumA = convert_int4(inputImage[posA.x + posA.y *
    width]);
    }
    if(posB.x >= 0 && posB.y >= 0)
    {
        sumB = convert_int4(inputImage[posB.x + posB.y *
    width]);
    }
    if(posD.x >= 0 && posD.y >= 0)
    {
        sumD = convert_int4(inputImage[posD.x + posD.y *
    width]);
    }
    sumC = convert_int4(inputImage[posC.x + posC.y *
    width]);

    outputImage[x + y * width] = convert_uchar4((sumA + sumC
    - sumB - sumD) / filterSize);

}



__kernel void horizontalSAT0(__global uchar4* input,
                                    __global uint4* output,
                                    int i, int r, int width)
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int pos = x + y * width;

    //Do a copy from input to output buffer
    //output[x + y * width] = convert_uint4(input[x + y *
    width]);
    //barrier(CLK_GLOBAL_MEM_FENCE);

    int c = pow((float)r, (float)i);

    uint4 sum = 0;

    for(int j = 0; j < r; j++)
    {
        if((x - (j * c)) < 0)
        {
              output[pos] = sum;
              return;
        }
```

```
        sum += convert_uint4(input[pos - (j * c)]);
    }

    output[pos] = sum;
}



__kernel void horizontalSAT(__global uint4* input,
                            __global uint4* output,
                             int i, int r, int width)
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int pos = x + y * width;

    int c = pow((float)r, (float)i);

    uint4 sum = 0;

    for(int j = 0; j < r; j++)
    {
        if(x - (j * c) < 0)
        {
            output[pos] = sum;
            return;
        }
        sum += input[pos - (j * c)];
    }

    output[pos] = sum;
}



__kernel void verticalSAT(__global uint4* input,
                          __global uint4* output,
                          int i, int r, int width)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int c = pow((float)r, (float)i);

    uint4 sum = (uint4)(0);

    for(int j = 0; j < r; j++)
```

```
    {
        if(y - (j * c) < 0)
        {
            output[x + y * width] = sum;
            return;
        }

        sum += input[x + width * (y - (j * c))];
    }

    output[x + y * width] = sum;
}



#define GROUP_SIZE 256

__kernel void box_filter_horizontal(__global uchar4*
    inputImage, __global uchar4* outputImage, int
    filterWidth)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int width = get_global_size(0);
    int height = get_global_size(1);

    int pos = x + y * width;
    int k = (filterWidth - 1)/2;

    /* Discard pixels in apron */
    if(x < k || x >= (width - k))
    {
        outputImage[pos] = (uchar4)(0);
        return;
    }

    int4 size = (int4)(filterWidth);

    int4 sum = 0;
    /* Read values from (filterWidth x filterWidth) sized
    kernel */
    for(int X = -k; X < k; X=X+2)
    {
        sum += convert_int4(inputImage[pos + X]);
        sum += convert_int4(inputImage[pos + X + 1]);
    }
```

```
    sum += convert_int4(inputImage[pos + k]);
    outputImage[pos] = convert_uchar4(sum / size);
}



__kernel void box_filter_vertical(__global uchar4*
    inputImage, __global uchar4* outputImage, int
    filterWidth)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int width = get_global_size(0);
    int height = get_global_size(1);

    int pos = x + y * width;
    int k = (filterWidth - 1)/2;

    /* Discard pixels in apron */
    if(y < k || y >= (height - k))
    {
       outputImage[pos] = (uchar4)(0);
       return;
    }

    int4 size = (int4)(filterWidth);

    int4 sum = 0;
    /* Read values from (filterWidth x filterWidth) sized
    kernel */
    for(int Y = -k; Y < k; Y=Y+2)
    {
       sum += convert_int4(inputImage[pos + Y * width]);
       sum += convert_int4(inputImage[pos + (Y + 1) *
    width]);
    }
    sum += convert_int4(inputImage[pos + k * width]);
    outputImage[pos] = convert_uchar4(sum / size);
}



__kernel void box_filter_horizontal_local(__global uchar4*
    inputImage, __global uchar4* outputImage, int
    filterWidth, __local uchar4 *lds)
```

```
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int width = get_global_size(0);
    int height = get_global_size(1);

    int pos = x + y * width;
    int k = (filterWidth - 1)/2;

    int lid = get_local_id(0);
    int gidX = get_group_id(0);
    int gidY = get_group_id(1);

    int gSizeX = get_local_size(0);
    int gSizeY = get_local_size(1);

    int firstElement = gSizeX * gidX + width * gidY *
    gSizeY;

    /* Load first k and last k into shared memory */
    if(lid < k)
    {
       lds[lid] = inputImage[firstElement - k + lid];
       lds[GROUP_SIZE + k + lid] = inputImage[firstElement +
    lid + GROUP_SIZE];
    }

    /* Load GROUP_SIZE  values into shared memory */
    lds[lid + k] = inputImage[firstElement + lid];

    barrier(CLK_LOCAL_MEM_FENCE);

    /* Discard pixels in apron */
    if(x < k || x >= (width - k))
       return;

    int4 size = (int4)(filterWidth);

    int4 sum = 0;
    /* Read values from (filterWidth x filterWidth) sized
    kernel */
    for(int X = -k; X <= k; X++)
    {
       sum += convert_int4(lds[lid + X + k]);
    }
    outputImage[pos] = convert_uchar4(sum / size);
}
```