UNIVERSIDAD NACIONAL DEL ALTIPLANO

ESCUELA DE POSGRADO

PROGRAMA DE MAESTRÍA

MAESTRÍA EN INFORMÁTICA

TESIS

MODELO DE PROGRAMACIÓN EN PLATAFORMA DE COMPUTACIÓN PARALELA BASADO EN EL DESEMPEÑO DEL GPU

PRESENTADA POR:

NÉLIDA RIVERA ENRIQUEZ

PARA OPTAR EL GRADO ACADÉMICO DE:

MAGISTER SCIENTIAE EN INFORMÁTICA

PUNO, PERÚ

2014

UNIVERSIDAD NACIONAL DEL ALTIPLANO

ESCUELA DE POSGRADO

PROGRAMA DE MAESTRÍA

MAESTRÍA EN INFORMÁTICA

TESIS

MODELO DE PROGRAMACIÓN EN PLATAFORMA DE COMPUTACIÓN PARALELA BASADO EN EL DESEMPEÑO DEL GPU

PRESENTADA POR:

NÉLIDA RIVERA ENRIQUEZ

PARA OPTAR EL GRADO ACADÉMICO DE:

MAGISTER SCIENTIAE EN INFORMÁTICA

APROBADA POR EL SIGUIENTE JURADO:

PRESIDENTE

.....

M.Sc. Ernesto Nayer Tumi Figueroa

PRIMER MIEMBRO

.....

M.Sc. Pedro Leonardo Quispe Ticona

SEGUNDO MIEMBRO

.....

M.Sc. Paco Wilson Marconi Quispe

ASESOR DE TESIS

.....

M.Sc. Leonid Alemán Gonzales

Puno, 27 de marzo del 2014

ÁREA: Ingeniería de software
TEMA: Desarrollo de sistema informatico

DEDICATORIA

El presente trabajo lo dedico a mis hijos, que son mi fuerza para salir adelante con la bendición y protección de Dios.

AGRADECIMIENTOS

- A Dios por bendecirme para llegar a este objetivo anhelado y darme unos hijos maravillosos que son mi fuerza para seguir adelante.
- A mis padres, esposo y amigos por su apoyo incondicional en este momento tan importante.
- A la Universidad Nacional del Altiplano, a la Maestría en Informática por permitirme lograr un nivel más en mi formación profesional.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ANEXOS

RESUMEN

La computación paralela permite acelerar los procesos de cálculo computacional superando en un 100% con cada generación de nuevos procesadores, en esta investigación en particular se ha centrado este aprovechamiento de la computación paralela usado la Unidad de Proceso Grafico (GPU), que ha superado en desempeño, velocidad y procesadores internos, llegando a tener hasta 256 procesadores en el año 2012, lo que se ha incrementado a 1024 procesadores en 2014. Una de las librerías más portables para el trabajo de aplicaciones de computación paralela es la librería OpenCL (Open Computing Language, en español lenguaje de computación abierto) consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en unidades centrales de procesamiento como unidades de procesamiento gráfico. El lenguaje está basado en C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales. La Empresa NVIDIA también ha desarrollado su propia librería para sus poderosas tarjetas gráficas GeForce aunque caras y de acceso con licencia la librería CUDA. Finalmente se propone en esta investigación el modelo correcto de programación para este nuevo tipo de computación y calculo en la cada vez más incrementable información gráfica y visual multimedia.

Palabras Clave: OpenCL, CUDA, Paralelismo, Alto desempeño, GPU.

ABSTRACT

Parallel computing allows to accelerate computationally processes exceeding 100% with each new generation of processors, in this particular research has focused the use of parallel computing used Graphic Processing Unit (GPU) has surpassed performance, internal processor speed, reaching up to 256 processors in 2012, which has increased to 1024 processors in 2014. One of the most portable libraries for the work of parallel computing applications is the (Open Computing Language) library consists of application programming interface and a programming language OpenCL. Together they allow you to create applications with data -level parallelism and task that can run on both CPUs and graphics processing units. The language is based on C99, eliminating certain functionality and extending it with vector operations. The NVIDIA Company has also developed its own library for its powerful but expensive graphics cards GeForce CUDA Seller and licensed access. Finally, the correct programming model for this new kind of computation and calculation in the increasingly graphic and visual increasing multimedia information is proposed in this research.

Keywords: OpenCL , CUDA , Parallelism , High performance, GPU.

INTRODUCCIÓN

Puede definirse como el uso de una unidad de procesamiento gráfico (GPU) en combinación con una CPU para acelerar aplicaciones de cálculo científico, ingeniería y empresa. NVIDIA lo introdujo en 2007 y, desde entonces, las GPU han pasado a instalarse en centros de datos energéticamente eficientes de laboratorios, universidades, grandes compañías y PYMEs de todo el mundo. El cálculo acelerado en la GPU ofrece un rendimiento sin precedentes ya que traslada las partes de la aplicación con mayor carga computacional a la GPU y deja el resto del código ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Es necesario especificar un modelo de programación adecuado para adaptar las futuras aplicaciones que logren aprovechar al máximo la potencia de este procesamiento, debe tenerse en cuenta que este procesamiento acelerado ayuda en áreas como Computación Grafico, Visión Computacional, Inteligencia Artificial y Procesamiento de Imágenes y Patrones, así como el aceleramiento de búsqueda y detección de grandes cantidades de datos tal vez rostros en una estación de trenes.

CAPÍTULO I

PROBLEMÁTICA DE INVESTIGACIÓN

1.1 PLANTEAMIENTO DEL PROBLEMA

Actualmente los procesadores, tienen desarrollada una arquitectura que permite hasta 3 GHz por núcleo, pero a pesar de todo ese desarrollo aun no superan los 8 núcleos por procesador, si bien para programas o aplicaciones normales esto es más que suficiente, para la computación de alto desempeño como programas CAD, procesadores de Imágenes, editores de audio y video, renderizadores de animaciones 3D y Computación Gráfica el CPU no abastece para procesar tal cantidad de información y algoritmos que toman horas en finalizar el procesamiento de datos como la generación de mallas para una superficie no uniforme y la renderización con los contrastes de intensidad de luz, sombra y aún más lento para procesar movimiento en tiempo real.

En cambio los GPU ofrecen mayo rendimiento además de más de 250 núcleos y teniendo en cuenta que cada año duplica su potencia, es decir que para fines del 2014 ya se tendrá GPUs de 1024 núcleos, reduciendo el tiempo de procesamiento y mejorando la calidad de resultado no solo gráfico, sino de procesamiento de información, actualmente el GPU se usa para procesar

grandes cantidades de datos como el suavizamiento y renderización de mallas triangulares para procesamiento de imágenes y renderización de la misma en tiempos relativamente cortos. Pues cada Thread o hilo creado se procesa a una velocidad 10 veces superior a la de los microprocesadores normales.

Esto nos conduce a la pregunta de investigación: ¿El Planteamiento de un modelo de programación para plataformas de computación paralela basada en el desempeño del GPU, mejorará el desempeño de los actuales enfoques computacionales?

1.2 OBJETIVOS

1.2.1 Objetivo General

- Contrastar y definir un modelo de programación para Plataformas de Computación Paralela basado en el desempeño del GPU.

1.2.2 Objetivos Específicos

- Analizar el costo computacional del proceso de cálculo en el CPU y el GPU.
- Establecer el modelo más óptimo para codificar aplicaciones que desempeñen ejecución en paralelo.
- Documentar las interfaces más óptimas para usar OpenCL y poder usar toda la capacidad de un procesador GPU.

CAPÍTULO II

MARCO TEÓRICO

2.1 ANTECEDENTES DE LA INVESTIGACIÓN

Meneses (2011) Es un compendio de las diversas libreras para computación paralela, mostrando las rutinas generales para lograr la programación correcta en este tipo de ambientes, muestra además cuadros de desempeño de distintas tarjetas gráficas y la importancia de la GPGPU o GPU para computación de propósito general, no limitando su desempeño el proceso de video sino en la posibilidad de aplicarla en diversos cálculos puesto que tiene procesadores de sobra para realizar cálculos.

Ponce (2013) presenta los entornos de configuración en Windows y Linux, además de la implementación de teoremas de Física en C++ y agregar conceptos como la detección de colisión, simulación de cuerpos rígidos, simulación de cuerpos suaves o esponjosos y para lograr la gran cantidad de calculo que requiere sin ralentizar la renderización hace uso de la librería OpenCL para computación paralela y los resultados son óptimos al visualizar los resultados.

2.2 BASE TEÓRICA

2.2.1 Computación Paralela

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que los 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 paralelismo se ha empleado durante muchos años, sobre todo en la computación de altas prestaciones, pero el interés en ella ha crecido últimamente debido a las limitaciones físicas que impiden el aumento de la frecuencia. Como el consumo de energía y por consiguiente la generación de calor de las computadoras constituye una preocupación en los últimos años, la computación en paralelo se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en forma de procesadores multinúcleo. Las computadoras paralelas pueden clasificarse según el nivel de paralelismo que admite su hardware: equipos con procesadores multinúcleo y multiprocesador que tienen múltiples elementos de procesamiento dentro de una sola máquina y los clústeres, MPPS y grids que utilizan varios equipos para trabajar en la misma tarea. Muchas veces, para acelerar tareas específicas, se utilizan arquitecturas especializadas de computación en paralelo junto a procesadores tradicionales. (Kirk D. 2012).

Los programas informáticos paralelos son más difíciles de escribir que los secuenciales porque la concurrencia introduce nuevos tipos de errores de software, siendo las condiciones de carrera los más comunes. La comunicación y sincronización entre diferentes subtareas son algunos de los mayores obstáculos para obtener un buen rendimiento del programa paralelo. La máxima aceleración posible de un programa como resultado de la paralelización se conoce como la ley de Amdahl. (Culler D.,1997)

2.2.2 CPU vs GPU

EL CPU o Unidad Central de Proceso es el encargado de realizar la ejecución de mnemo-códigos de los que constan los programas y las sentencias del mismo sistema operativo host (Kirk D., 2012), cabe destacar que este dispositivo actualmente ha desarrollado velocidades de hasta 3 GHz y se han integrado hasta 8 núcleos por chip, pero a pesar de ello la latencia de procesamiento aún es muy baja comparada con los procesadores GPU.

El GPU o Unidad de Procesamiento Gráfico ha crecido en tecnología y latencia de procesamiento en esencia por la ingente cantidad de información gráfica que procesa una tarjeta de video, los videojuegos has tenido gran parte de la contribución en su desarrollo pues cada nuevo videojuego exigía más recursos gráficos, con la llegada de los juegos tridimensionales y de realidad aumentada, no bastaba uno ni ocho procesadores, es por ello que actualmente las tarjetas como la GeForce 8800 tiene hasta 150 procesadores con una latencia 10 veces más alta que los CPU, todo este poder de procesamiento se incrementa cada año,

por la misma exigencia de procesamiento de imágenes ahora de HD a mas fidelidad. Los Conceptos de programación paralela pueden ser desarrollados teóricamente y demostrados usando esta plataforma de procesamiento y para la implementación usarse las librerías de propósito general. (NVIDIA, 2010).

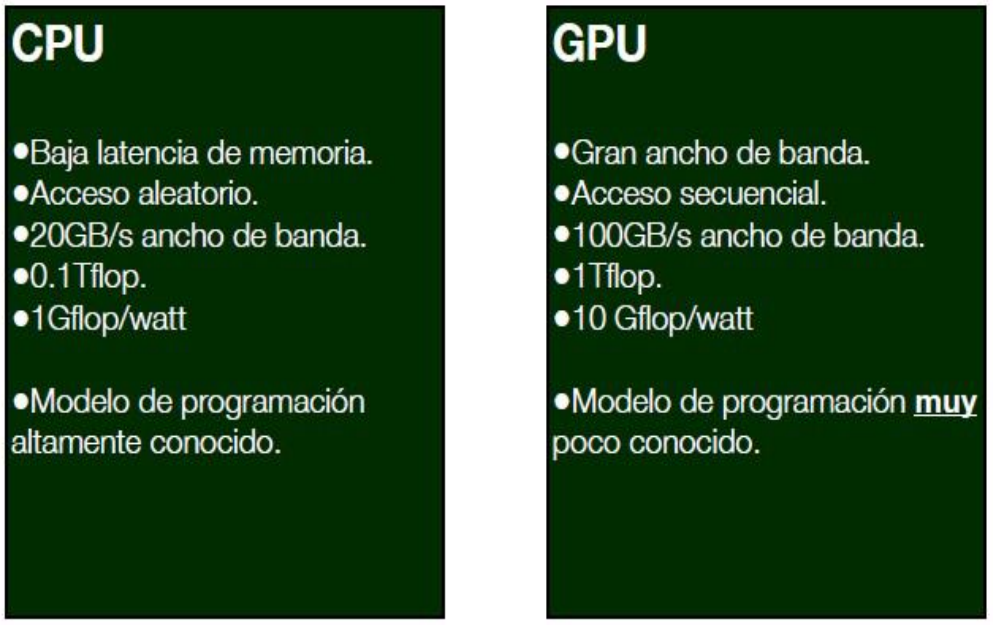
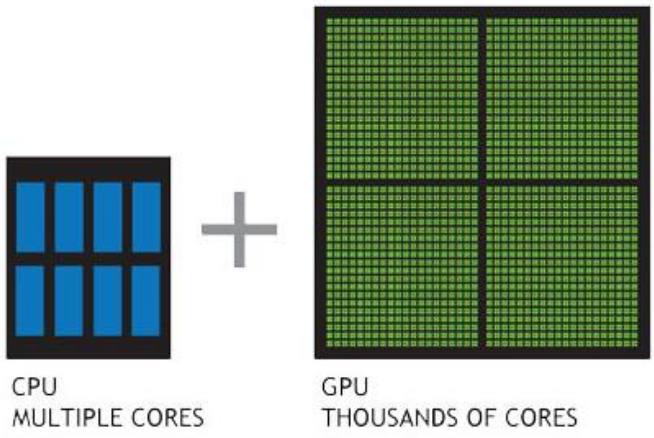


Figura 1. Comparación de performance entre un CPU y GPU.

Fuente: (NVIDIA, 2010)



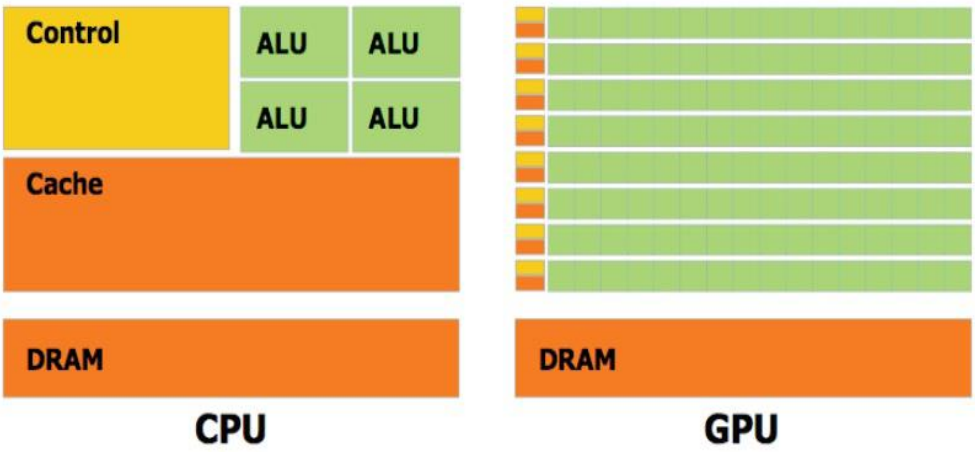


Figura 2. Comparación gráfica de la distribución de núcleos de procesamiento.

Fuente: (NVIDIA, 2010)

Una forma sencilla de entender la diferencia entre la CPU y la GPU es comparar la forma en que procesan las tareas. Una CPU consta de unos cuantos núcleos optimizados para el procesamiento secuencial de las instrucciones, mientras que la GPU se compone de miles de núcleos, más pequeños y eficientes, diseñados para manejar múltiples tareas de forma simultánea. (NVIDIA, 2010)

2.2.3 Calculo acelerado en la GPU

Puede definirse como el uso de una unidad de procesamiento gráfico (GPU) en combinación con una CPU para acelerar aplicaciones de cálculo científico, ingeniería y empresa. NVIDIA lo introdujo en 2007 y, desde entonces, las GPU han pasado a instalarse en centros de datos energéticamente eficientes de laboratorios, universidades, grandes compañías y PYMEs de todo el mundo. Cómo se aceleran las aplicaciones con las GPUS el cálculo acelerado en la GPU ofrece un rendimiento sin precedentes ya que traslada las partes de la aplicación

con mayor carga computacional a la GPU y deja el resto del código ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido. (NVIDIA, 2010).

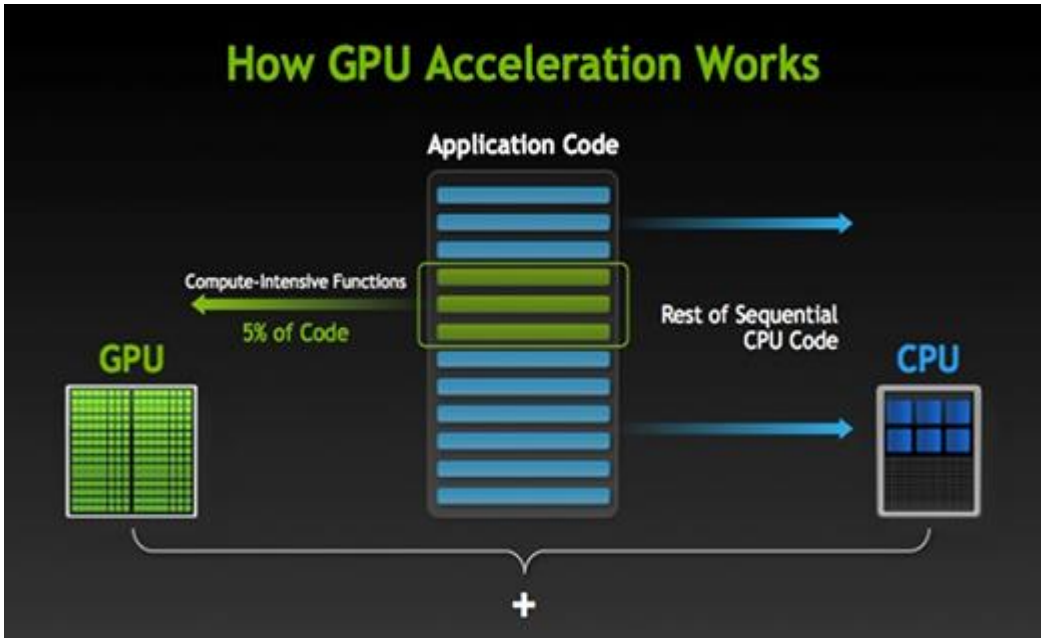


Figura 3. Mientras en GPU es capaz de ejecutar 5% de código en un ciclo de reloj en cambio el CPU el 1% por ciclo de reloj.
Fuente: (NVIDIA, 2010)

2.2.4 Plataforma de desarrollo sobre GPU

Entre los más importantes podemos citar:

- OpenCL (Creado por Apple, IBM, AMD, NVidia, Intel)
- N-Vidia CUDA (API Privativo de alto desempeño)
- Brook+
- DirectCompute
- CAPS

Tradicionalmente, los programas informáticos se han escrito para el cómputo en serie. Para resolver un problema, se construye un algoritmo y se implementa como un flujo en serie de instrucciones. Estas instrucciones se ejecutan en una unidad central de procesamiento en un ordenador. Sólo puede ejecutarse una instrucción a la vez y un tiempo después de que la instrucción ha terminado, se ejecuta la siguiente.

La computación en paralelo, por el contrario, utiliza simultáneamente múltiples elementos de procesamiento para resolver un problema. Esto se logra mediante la división del problema en partes independientes de modo que cada elemento de procesamiento pueda ejecutar su parte del algoritmo de manera simultánea con los otros. Los elementos de procesamiento son diversos e incluyen recursos tales como una computadora con múltiples procesadores, varios ordenadores en red, hardware especializado, o cualquier combinación de los anteriores. (NVIDIA, 2010).

2.2.5 LEY DE AMDAHI Y LEY DE GUSTAFSON

Idealmente, la aceleración a partir de la paralelización es lineal, doblar el número de elementos de procesamiento debe reducir a la mitad el tiempo de ejecución y doblarlo por segunda vez debe nuevamente reducir el tiempo a la mitad. Sin embargo, muy pocos algoritmos paralelos logran una aceleración óptima. La mayoría tienen una aceleración casi lineal para un pequeño número de elementos de procesamiento, y pasa a ser constante para un gran número de elementos de procesamiento. (Amdahl, 1967).

La aceleración potencial de un algoritmo en una plataforma de cómputo en paralelo está dada por la ley de Amdahl, formulada originalmente por Gene

Amdahl en la década de 1960. Esta señala que una pequeña porción del programa que no pueda paralelizarse va a limitar la aceleración que se logra con la paralelización. Los programas que resuelven problemas matemáticos o ingenieriles típicamente consisten en varias partes paralelizables y varias no paralelizables (secuenciales). Si alpha (α) es la fracción de tiempo que un programa gasta en partes no paralelizables, luego

$$S = \frac{1}{\alpha} \qquad = \lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha}$$

Es la máxima aceleración que se puede alcanzar con la paralelización del programa. Si la parte secuencial del programa abarca el 10% del tiempo de ejecución, se puede obtener no más de 10× de aceleración, independientemente de cuántos procesadores se añadan. Esto pone un límite superior a la utilidad de añadir más unidades de ejecución paralelas. (Brooks, 1996)

«Cuando una tarea no puede dividirse debido a las limitaciones secuenciales, la aplicación de un mayor esfuerzo no tiene efecto sobre la programación. La gestación de un niño toma nueve meses, no importa cuántas mujeres se le asigne”. (Brooks, 1996).

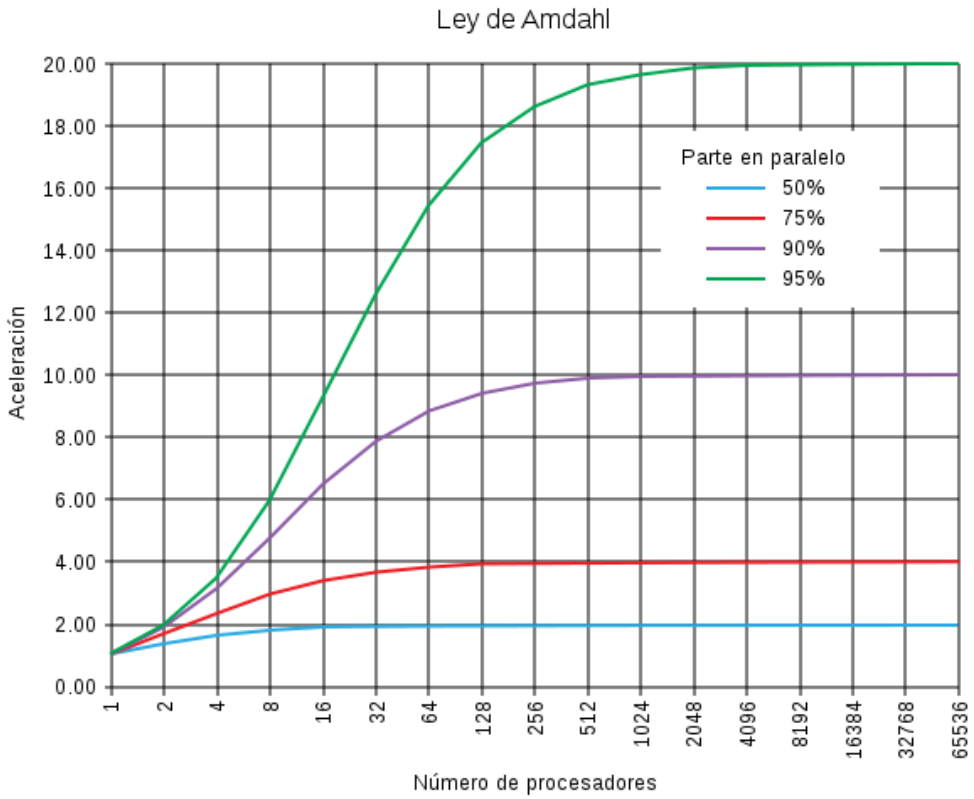


Figura 4. Representación gráfica de la ley de Amdahl.
Fuente: (Brooks, 1996)

La mejora en la velocidad de ejecución de un programa como resultado de la paralelización está limitada por la porción del programa que no se puede paralelizar. Por ejemplo, si el 10% del programa no puede paralelizarse, el máximo teórico de aceleración utilizando la computación en paralelo sería de 10x no importa cuántos procesadores se utilicen. (Brooks, 1996).

2.2.6 OPENCL

OpenCL (Open Computing Language, en español lenguaje de computación abierto) consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden

ejecutarse tanto en unidades centrales de procesamiento como unidades de procesamiento gráfico. El lenguaje está basado en C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales. Apple creó la especificación original y fue desarrollada en conjunto con AMD, IBM, Intel y NVIDIA. Apple la propuso al Grupo Khronos para convertirla en un estándar abierto y libre de derechos. El 16 de junio de 2008 Khronos creó el Compute Working Group3 para llevar a cabo el proceso de estandarización. En 2013 se publicó la versión 2.0 del estándar OpenCL forma parte de Mac OS X v10.6 ('Snow Leopard'), mientras que AMD decidió apoyar OpenCL en lugar de su antigua API Close to Metal. Intel también dispone de su propio entorno de desarrollo y NVIDIA además de tener su propia API para chips gráficos llamada CUDA, también soporta OpenCL. (Tay, 2013)

Este ejemplo calcula una Transformada rápida de Fourier. Las llamadas a la API son las siguientes:

```
// create a compute context with GPU device

context = clCreateContextFromType(CL_DEVICE_TYPE_GPU);


// create a work-queue

queue = clCreateWorkQueue(context, NULL, NULL, 0);


// allocate the buffer memory objects

memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA);

memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(float)*2*num_entries, NULL);
```

```
// create the compute program

program = clCreateProgramFromSource(context, 1,
&fft1D_1024_kernel_src, NULL);


// build the compute program executable

clBuildProgramExecutable(program, false, NULL, NULL);


// create the compute kernel

kernel = clCreateKernel(program, "fft1D_1024");


// create N-D range object with work-item dimensions

global_work_size[0] = n;

local_work_size[0] = 64;

range = clCreateNDRangeContainer(context, 0, 1, global_work_size,
local_work_size);


// set the args values

clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem),
NULL);

clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem),
NULL);

clSetKernelArg(kernel, 2, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);

clSetKernelArg(kernel, 3, NULL,
sizeof(float)*(local_work_size[0]+1)*16, NULL);


 // execute kernel

clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

El cómputo en sí es este:

```
// This kernel computes FFT of length 1024. The 1024 length FFT is decomposed
// into calls to a radix 16 function, another radix 16 function and then a
// radix 4 function
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy)
{
    int tid = get_local_id(0);
```

```
    int blockIdx = get_group_id(0) * 1024 + tid;

    globalStores(data, out, 64);
}
```

Código fuente ejemplo usando los kernel de OpenCL. (Tay, 2013)

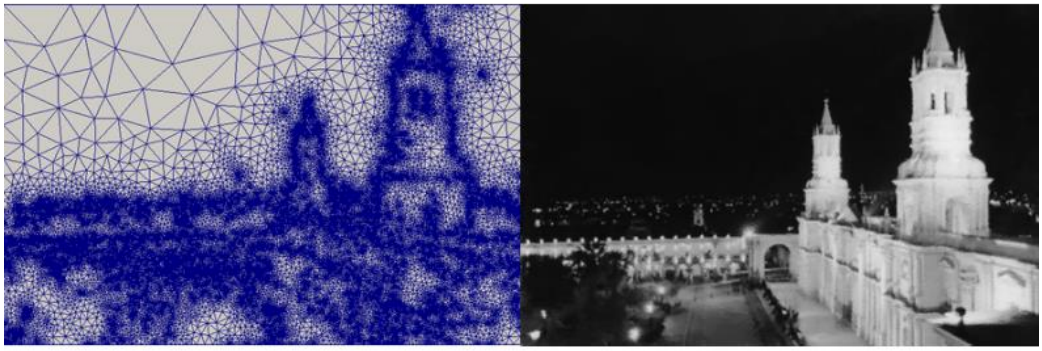


Figura 5. Ejemplo de programa escrito en C/C++ y CUDA para acelerar el procesamiento de cálculo de colores en mallas triangulares. Teniendo en cuenta que hay más de 30,000 triángulos y miles de cálculo por renderización.

Fuente: (Harris, 2012)

2.2.7 HILOS DE EJECUCION (THREADS)

En sistemas operativos, un hilo de ejecución, hebra o subproceso es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. (Stallings, 2011)

La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente). Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente. (Stallings, 2011)

Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo con otra tarea. (Stallings, 2011)

Los hilos de ejecución que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un proceso. El hecho de que los hilos de ejecución de un mismo proceso compartan los recursos hace que cualquiera de estos hilos pueda modificar éstos. Cuando un hilo modifica un dato en la memoria, los otros hilos acceden a ese dato modificado inmediatamente. Lo que es propio de cada hilo es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros). El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo. Cuando el proceso finaliza, todos sus hilos de ejecución también han terminado. Asimismo, en el momento en el que todos los hilos de ejecución finalizan, el proceso no existe más y todos sus recursos son liberados. Algunos lenguajes de programación tienen características de diseño expresamente creadas para permitir a los programadores lidiar con hilos de ejecución (como Java o Delphi). Otros (la mayoría) desconocen la existencia de hilos de ejecución y éstos deben ser creados mediante llamadas de biblioteca especiales que dependen del sistema operativo en el que estos lenguajes están siendo utilizados (como es el caso del C y del C++).(Stallings, 2011).

CAPÍTULO III

METODOLOGÍA

3.1 MÉTODO DE RECOLECCIÓN DE DATOS

Se realiza una investigación de tipo exploratoria-descriptiva, la realización de un estudio diagnóstico descriptivo que permita indagar sobre el comportamiento de los sistemas existentes y en estudio.

3.2 METODOLOGÍA DE DESARROLLO

Son procesos basados en la documentación y el orden para conseguir productos de software de calidad, a lo que los que desarrolladores de software que no les gustan del papeleo lo han denominado metodologías burocráticas por ocupar el mayor tiempo en la documentación y quitando el encanto a la programación.

3.3 MODELO DE REQUERIMIENTOS

Requisitos funcionales.

1) Poner a disposición de los usuarios un programa que permite el manejo de imágenes procesado por GPU.

2) Funciones de aceleración el efecto de imágenes.

3) Diseño de una interfaz de usuario amigable y de fácil modificación.

Requisitos No Funcionales

Apariencia o Interfaz Externa:

El diseño de la interfaz deberá ser agradable y sobre todo lo más sencillo posible, pues será utilizada tanto por usuarios con una preparación integral como por algunos con conocimientos básicos de computación, por lo que, además, deberá ser sencilla, aprovechando las facilidades del ambiente Web en el que se desarrollará.

Usabilidad:

El sistema podrá ser usado por cualquier tipo de personas que posean conocimientos básicos en el manejo de la computadora y el ambiente Web en sentido general. Rendimiento: Aunque no se requiere una velocidad de respuesta comparada con los sistemas de tiempo real, se debe garantizar la rapidez de respuesta del sistema ante las solicitudes de los usuarios.

Portabilidad:

El programa podrá implantarse para las distintas versiones de Windows, de forma tal que no haya dificultad en cambiar de una a otra plataforma sin necesidad de efectuar cambios significativos. Lo anterior se debe a que la aplicación está implementada sobre PHP que es un lenguaje multiplataforma.

Requerimientos de Software:

Para la implantación del sistema se requiere de un entorno de desarrollo de programación para el lenguaje C++ Visual para lo que se ha recurrido al entorno de programación C++ Builder. Los requerimientos en el lado del cliente para la utilización del sistema solo se limitan a tener disponible un navegador Web.

Requerimientos de Hardware:

La computadora dedicada a servidor debe tener como mínimo las siguientes características de hardware: Procesador Dual Core, o similar, a 3.8 GHz, 2 Gb de memoria RAM y 250 Gb de capacidad en disco duro. Las computadoras que pueden ser utilizadas por los usuarios para acceder al sistema solo deben de estar conectadas en red con el servidor.

3.4 LENGUAJE DE MODELAMIENTO UNIFICADO

Lenguaje Unificado de Modelado (LUM o 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 OMG (Object Management Group). 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 de negocio, 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 o RUP), pero no especifica en sí mismo qué metodología o proceso usar.

UML no puede compararse con la programación estructurada, pues UML significa Lenguaje Unificado de Modelado, no es programación, solo se diagrama la realidad de una utilización en un requerimiento. Mientras que, programación estructurada, es una forma de programar como lo es la orientación a objetos, sin embargo, la programación orientada a objetos viene siendo un complemento perfecto de UML, pero no por eso se toma UML sólo para lenguajes orientados a objetos. UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas.

3.5 MÉTRICA DE VALIDACIÓN DE SOFTWARE

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. Este estándar es el más usado. 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. Idoneidad - Atributos del software relacionados con la presencia y aptitud de un conjunto de funciones para tareas especificadas.
- **Exactitud -** Atributos del software relacionados con la disposición de resultados o efectos correctos o acordados.
- **Interoperabilidad -** Atributos del software que se relacionan con su habilidad para la interacción con sistemas especificados
- **Seguridad -** Atributos del software relacionados con su habilidad para prevenir acceso no autorizado ya sea accidental o deliberado, a programas y datos.
- Cumplimiento de normas.
- **Fiabilidad -** Un conjunto de atributos relacionados con la capacidad del software de mantener su nivel de prestación bajo condiciones establecidas durante un período establecido.
- **Madurez -** Atributos del software que se relacionan con la frecuencia de falla por fallas en el software.
- **Recuperabilidad -** Atributos del software que se relacionan con la capacidad para restablecer su nivel de desempeño y recuperar los

datos directamente afectos en caso de falla y en el tiempo y esfuerzo relacionado para ello.

- **Tolerancia a fallos -** Atributos del software que se relacionan con su habilidad para mantener un nivel especificado de desempeño en casos de fallas de software o de una infracción a su interfaz especificada.
- **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.
- **Aprendizaje cc-** Atributos del software que se relacionan al esfuerzo de los usuarios para reconocer el concepto lógico y sus aplicaciones.
- **Comprensión -** Atributos del software que se relacionan al esfuerzo de los usuarios para reconocer el concepto lógico y sus aplicaciones.
- **Operatividad -** Atributos del software que se relacionan con el esfuerzo de los usuarios para la operación y control del software.
- Atractividad.
- **Eficiencia -** Conjunto de atributos relacionados con la relación entre el nivel de desempeño del software y la cantidad de recursos necesitados bajo condiciones establecidas.
- **Comportamiento en el tiempo -** Atributos del software que se relacionan con los tiempos de respuesta y procesamiento y en las tasas de rendimientos en desempeñar su función.
- Comportamiento de recursos
- **Mantenibilidad -** Conjunto de atributos relacionados con la facilidad de extender, modificar o corregir errores en un sistema software.

- **Estabilidad -** Atributos del software relacionados con el riesgo de efectos inesperados por modificaciones.
- **Facilidad de análisis -** Atributos del software relacionados con el esfuerzo necesario para el diagnóstico de deficiencias o causas de fallos, o identificaciones de partes a modificar.
- **Facilidad de cambio -** Atributos del software relacionados con el esfuerzo necesario para la modificación, corrección de falla, o cambio de ambiente.
- **Facilidad de pruebas -** Atributos del software relacionados con el esfuerzo necesario para validar el software modificado.
- **Portabilidad -** Conjunto de atributos relacionados con la capacidad de un sistema software para ser transferido desde una plataforma a otra.
- **Capacidad de instalación -** Atributos del software relacionados con el esfuerzo necesario para instalar el software en un ambiente especificado.
- **Capacidad de reemplazamiento -** Atributos del software relacionados con la oportunidad y esfuerzo de usar el software en lugar de otro software especificado en el ambiente de dicho software especificado.
- **Adaptabilidad -** Atributos del software relacionados con la oportunidad para su adaptación a diferentes ambientes especificados sin aplicar otras acciones o medios que los proporcionados para este propósito por el software considerado.
- Co-Existencia

La sub característica Conformidad no está listada arriba ya que se aplica a todas las características. Ejemplos son conformidad a la legislación referente a usabilidad y fiabilidad.

Cada subcaracterística (como adaptabilidad) está dividida en atributos. Un atributo es una entidad la cual puede ser verificada o medida en el producto software. Los atributos no están definidos en el estándar, ya que varían entre diferentes productos software.

Un producto software está definido en un sentido amplio como: los ejecutables, código fuente, descripciones de arquitectura, y así. Como resultado, la noción de usuario se amplía tanto a operadores como a programadores, los cuales son usuarios de componentes como son bibliotecas software.

El estándar provee un entorno para que las organizaciones definan un modelo de calidad para el producto software. Haciendo esto así, sin embargo, se lleva a cada organización la tarea de especificar precisamente su propio modelo. Esto podría ser hecho, por ejemplo, especificando los objetivos para las métricas de calidad las cuales evalúan el grado de presencia de los atributos de calidad.

- Métricas internas son aquellas que no dependen de la ejecución del software (medidas estáticas).
- Métricas externas son aquellas aplicables al software en ejecución.

Las calidades en las métricas de uso están sólo disponibles cuando el producto final es usado en condiciones reales. Idealmente, la calidad interna no necesariamente implica calidad externa y esta a su vez la calidad en el uso. Este estándar proviene desde el modelo establecido en 1977 por McCall y sus

colegas, los cuales propusieron un modelo para especificar la calidad del software. El modelo de calidad McCall está organizado sobre tres tipos de Características de Calidad:

- Factores (especificar): Describen la visión externa del software, como es visto por los usuarios.
- Criterios (construir): Describen la visión interna del software, como es visto por el desarrollador.
- Métricas (controlar): Se definen y se usan para proveer una escala y método para la medida.

ISO 9126 distingue entre fallo y no conformidad. Un fallo es el incumplimiento de los requisitos previos, mientras que la no conformidad es el incumplimiento de los requisitos especificados. Una distinción similar es la que se establece entre validación y verificación.

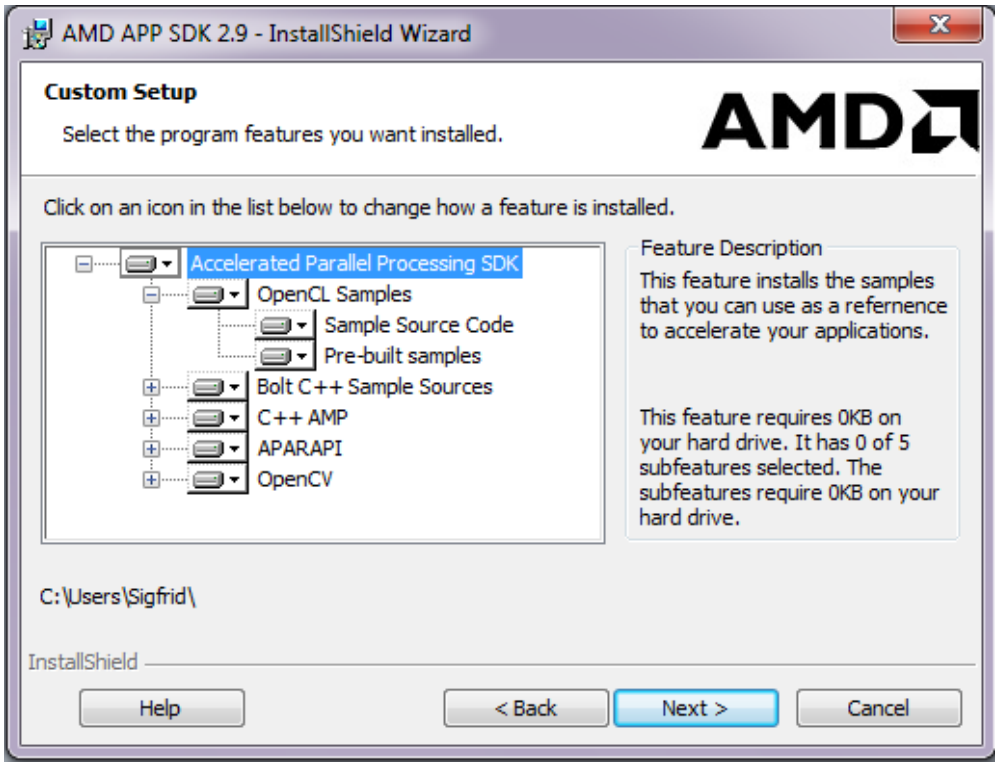
CAPÍTULO IV

RESULTADOS Y DISCUSIÓN

4.1 INSTALACIÓN DE OPENCL APP SDK

Puede descargarse desde el sitio web, una vez descargada la aplicación ejecute el instalador.

Website:http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-partner/amd-app-sdk-2-9-windows-321.exe



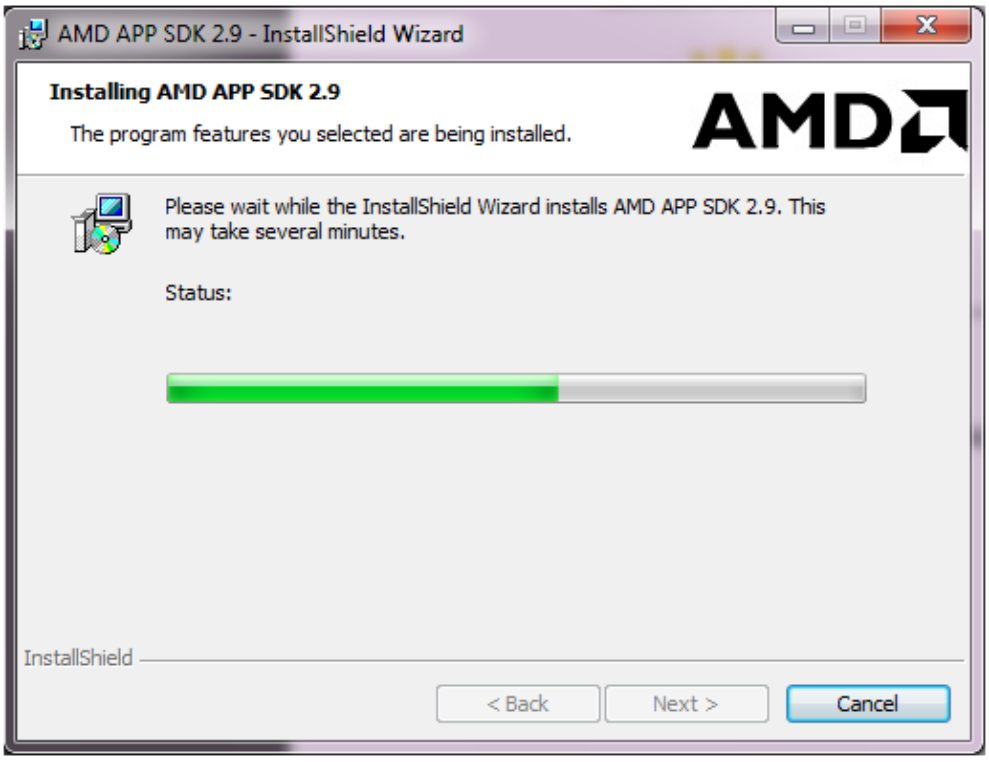


Figura 6. Proceso de instalación de OpenCL APP SDK para C++ y Visual Studio 2010, contiene librerías y un pack extra para OpenCV (Computing Vision SDK for C++ App).

Comparación de Código Fuente

Caso 1: Multiplicación del contenido de los vectores A y B de tipo float de 8 bytes para ampliar los datos de procesamiento. La implementación clásica se vería como:

```
void tradicional_multiplicacion(
int n,
const float *a,
const float *b,
float *c )
{
for( int i=0; i<n i++ )
c[i] = a[i] * b[i];
}
```

Evaluación de desempeño

Estas pruebas se realizaron en una PC AMD E-300 1,3 GHz (2 Cpus) con memoria RAM de 3GB, sobre Windows 7 Ultimate Edition

Tabla 1. Evaluación de desempeño de muestra de imágenes empleando un algoritmo con implementación clásica.

Tamaño del vector	Operación con punto flotante	Total de Iteraciones	Tiempo de procesamiento
1000	SI	n(1000)	0,012 ms
1’000,000	SI	n(1000000)	0,800 ms

Notas: en base de pruebas con el software de elaboración de prueba

Caso 2: Multiplicación del contenido de los vectores A y B de tipo float de 8 bytes implementando en OpenCL, basada en kernels:

```
kernel void paralelo_multiplicacion(
     const float *a,
     const float *b,
     float *c )
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
// ejecutar N kernels N-items
```

Evaluación de desempeño

Estas pruebas se realizaron: AMD Radeon 6310 de 256 núcleos.

Tabla 2. Evaluación de desempeño de muestra de imágenes empleando

Tamaño del vector	Operación con punto flotante	Total de Iteraciones	Tiempo de procesamiento
1000	SI	n(1000)	0,012 ms
1’000,000	SI	n(1000000)	0,014 ms

Notas: en base de pruebas con el software de elaboración de prueba

Claramente el tiempo de procesamiento es abismal, imaginemos que se tenga que calcular y renderizar una escena de 100 cuadros este proceso tomara 10% del tiempo total de una CPU.

4.2 PROPUESTA DE MODELO DE PROGRAMACIÓN

Si bien es cierto el modelo de programación parece sencillo, pero en realidad solo se muestra la función kernel de cálculo, veamos este segundo ejemplo.

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Hasta este punto no parece tener mayor complejidad, pero para inicializar el uso de la GPU es necesario crear los Buffers para los argumentos, establecer el Stack a usar, copiar a memoria valores globales y registrarlos, luego seleccionar y activar los GPUs para poder usarlos durante un periodo de tiempo que el GPU estime conveniente para un Thread o todos. Pero el código en la función main es enormemente engorroso, como se muestra a continuación:

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcA, NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
                             NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0,
n*sizeof(cl_float),
                          dst, 0, NULL, NULL);
```

a. Crear el contexto y obtener la info

b. Crear espacio en memoria y establecer permisos para el GPU

c. Crear los recursos del programa base con los hilos a ser generados

d. Crear los Kernels y establecer el gancho a la función recurrente

e. Enviar los argumentos

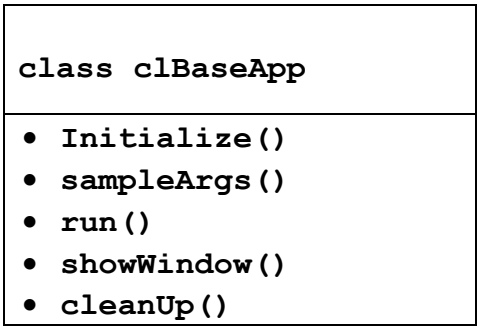
f. Se ejecuta

g. Recoger argumentos de retorno

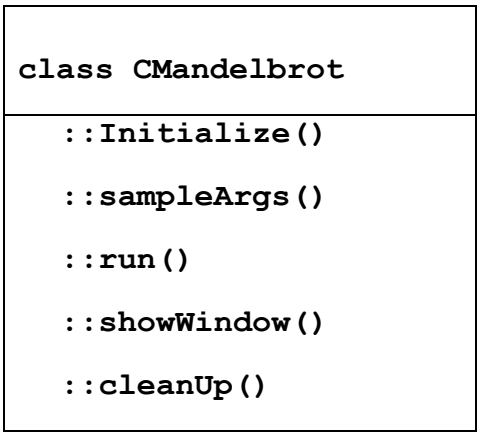
4.3 PROPUESTA DEL MODELO DE PROGRAMACION

Nos basaremos en los conceptos de la Programación Orientada a Objetos POO, para modularizar un modelo sencillo y recurrente para su uso. Gráficamente tendríamos.

Clase base definida como:



Clase derivada definida como:



El nuevo ***main*** tendría el siguiente cuerpo:

```
int main (
     int argc,
     char *argv[] )
{
    CMandelbrot *clMandelbrot = new CMandelbrot;
    clMandelbrot->initialize();
    clMandelbrot->sampleArgs->parseCommandLine(argc, argv);
    clMandelbrot->run();
}
// codigo limpio sin complejidad mas que en la libreria
```

4.4 EJEMPLOS DE PROGRAMACIÓN PARALELA

Tenemos los siguientes:

- **Conjunto de Mandelbrot:** El conjunto de Mandelbrot es un conjunto matemático de puntos en el plano complejo, cuyo borde forma un fractal.

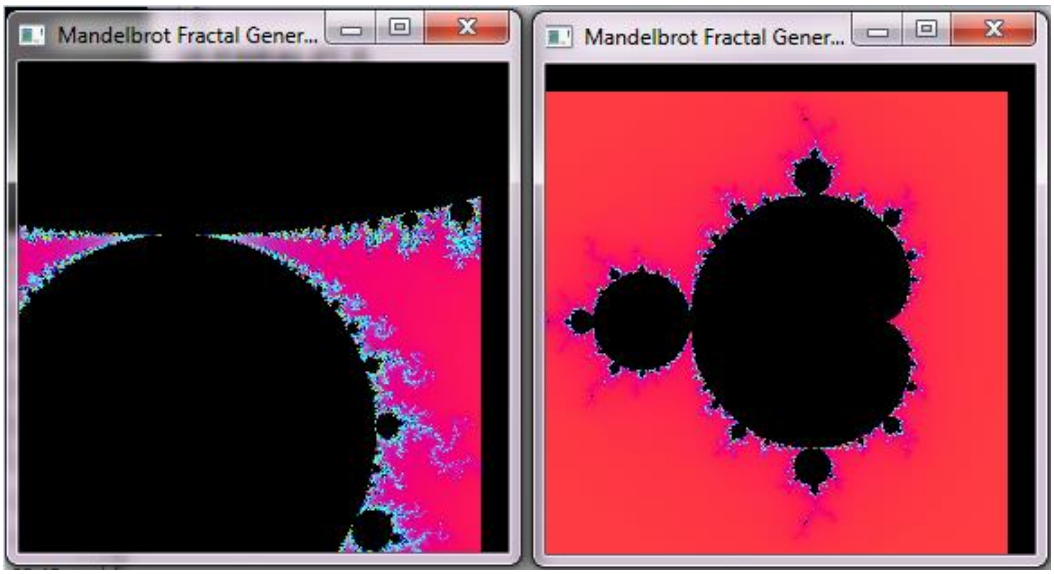


Figura 7. Figura del conjunto de Mandelbrot.

- **OpenCL renderizando gráficos en movimiento.**

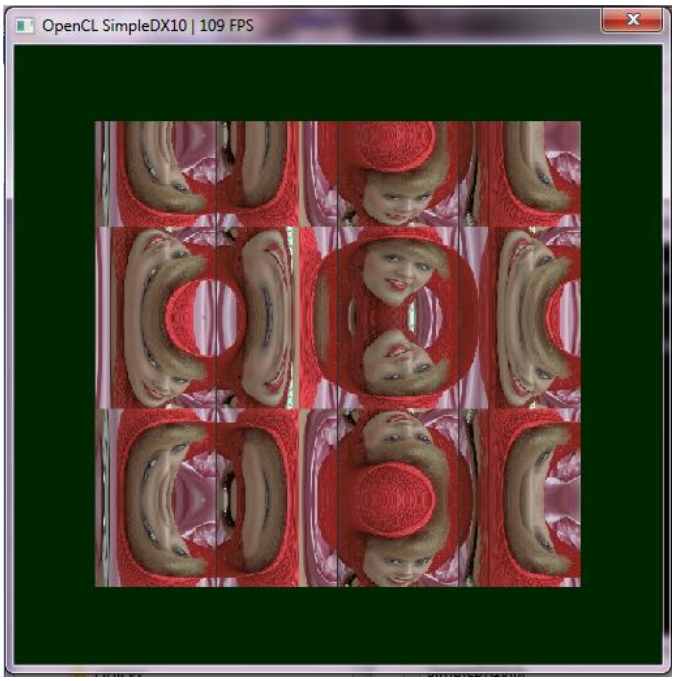


Figura 8. OpenCL renderizando gráfico en movimiento.

- **Ruido Gaussiano en OpenCL**

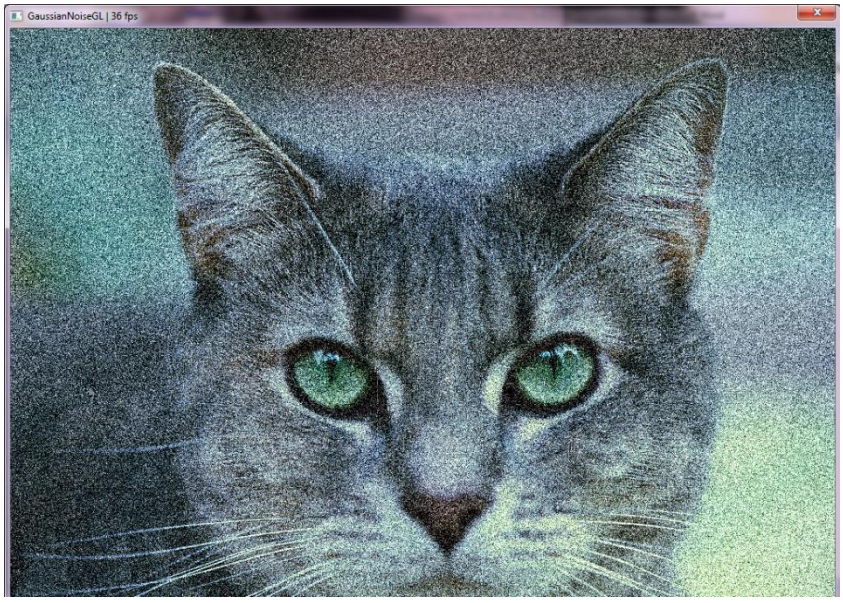


Figura 9. Creando Ruido Gaussiano en OpenCL. Sobreponer Imágenes

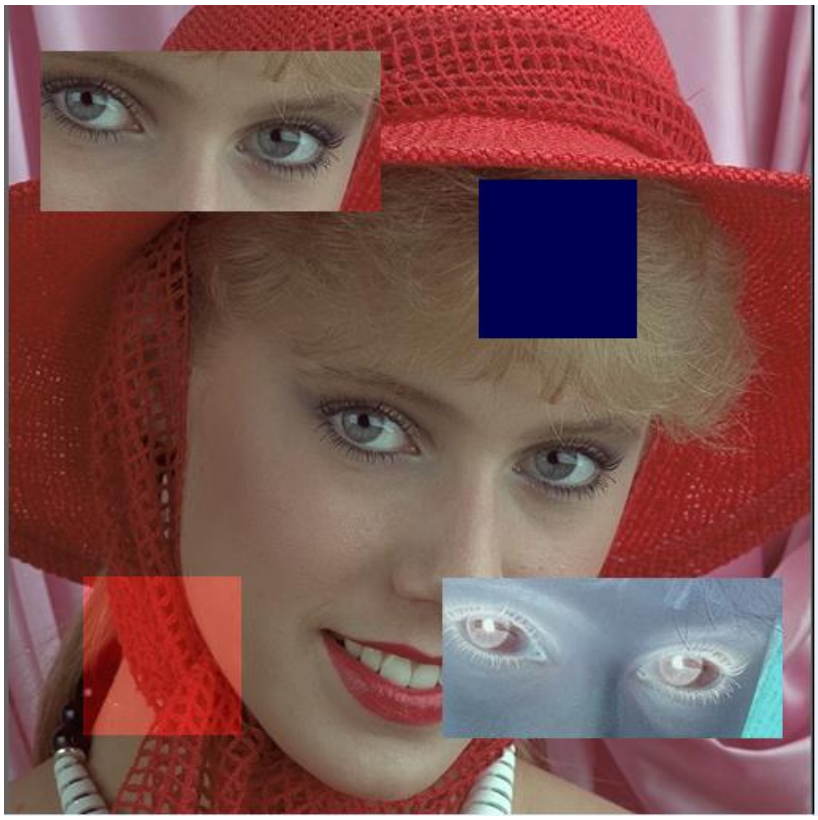


Figura 10. Creando sobre posición de imágenes.

CONCLUSIONES

- Analizar el costo computacional del proceso de cálculo en el CPU y el GPU, para el cálculo se ha hecho uso de un procesador de 2.3GHz sobre un procesador AMD Vision 1.3 Ghz Duo, mostrando que el tiempo se incrementa dramáticamente mientras más grande se vuelve la imagen. Como se muestra en los cuadros mostrados.
- Para establecer el modelo más óptimo para codificar aplicaciones que desempeñen ejecución en paralelo. Se ha revisado todos los ejemplos de la SDK y se ha optimizado el proceso proponiendo una librería base para las aplicaciones basadas en computación paralela basada en el GPU, optimizando el proceso de escritura de más aplicaciones.
- Se ha documentado el costo computacional de procesamiento de un CPU vs un GPU mostrando el mejor caso y el peor caso, así como un apéndice de programas que realizan proceso y para ver los resultados se ha hecho uso de ejemplos gráficos, modificación y renderizado en tiempo real de las mismas para aprovechar al máximo el procesamiento del GPU.

RECOMENDACIONES

- Se recomienda implementar una librería base App que permita generar la conexión al GPU, dejando así un esquema simple de programar, no debe repetirse el código en un ejemplo y otro lo adecuado es usar una librería base y a partir de ella generar las aplicaciones asi se aprovecha correctamente los tópicos de programación paralela y el gran desempeño que nos brinda el uso de la GPU
- La programación paralela está convirtiéndose en el estándar de programación de alto desempeño, generalmente para cálculos ingentes, se debe tener en cuenta que el costo computacional del uso general del CPU genera sobrecarga sobre el mismo, se propone seguir investigando en estos temas para mejorar las investigaciones

BIBLIOGRAFÍA

Amdahl, Gene M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities.* California: IBM Sunnyvale

Brooks, F. (1996). *The mythical man month essays on software engineering.* (5ta ed.). U.S.: Addison-Wesley.

Culler D. S., Pal, J. (1997). *Parallel computer architecture.* San Francisco: Morgan Kaufmann.

Dongara, J. (2003). *Sourcebook of Parallel Computing,* Massachusetts: Morgan Kaufmann Ed.

Giles, M. (2010). *Course on CUDA Programming on NVIDIA GPUs*. U.S.

Harris, M. (2012). *How to Overlap Data Transfers in CUDA C/C++.* U.S.: NVIDIA Corporation.

Hennessy, J. L., Patterson, D. (2006). *Computer Architecture: A Quantitative Approach* (4ta ed.). Massachusetts: Morgan Kaufmann.

Kirk, D., Hwu, W. (2012). *Programming Massively Parallel Processors,* Second Edition: A Hands-on Approach (2da Ed.). U.S.: Morgan Kaufmann.

Meneses, A. (2011). *Introducción a GPU's y Programación CUDA para HPC, Departamento de Computación*. CINVESTAV-IPN / LUFAC.

NVIDIA Corporation (2010). *NVIDIA CUDA C, Programming Guide,* Version 3.2, septiembre. Santa Clara, California, Estados Unidos:

NVIDIA Corporation (2010). *NVIDIA CUDA, CUDA C Best Practices Guide*, Version 3.2, agosto 2010. Santa Clara, California, U.S.: NVidia

NVIDIA Corporation; NVIDIA CUDA, *Reference Manual,* Version 3.2 Beta, agosto 2010. Santa Clara, California, Estados Unidos: NVidia

NVIDIA Corporation (2011). *CUDA Presentation 4.0; Parallel Programming in C with MPI and OpenMP.* U.S.: McGrawHill,

Ortega J., Anguita, M., y Prieto, A. (2005). *Arquitectura de Computadores.* España: Ed. Paraninfo S.A.

Ponce, Y. (2013), *Particle Based Simulations Using GPUs,* Arequipa Peru: SIBGRAPI.

Sanders J., Kandrot E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. España: Addison-Wesley Professional.

Stallings W. (2011). *Operating Systems: Internals and Design Principles* (7ma. Ed.). NY U.S.A.: Prentice Hall by PEARSON EDUCATION

Tay R. (2013). *OpenCL Parallel Programming Development Cookbook.* U.S.: Packt Publishing.

Yauri, M. P. (2013). *Aceleración en GPU del Proceso de Cálculo de Patrones de Color en Mallas Triangulares Generadas a Partir de Imágenes.* Arequipa: SIBGRAPI.

ANEXOS

Anexo 1.Código Fuente

```
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <math.h>

using namespace appsdk;
#define SAMPLE_VERSION "AMD-APP-SDK-v2.9.233.1"
#define MAX_ITER 16384
#define MIN_ITER 32
#define MAX_DEVICES 4

class clBaseApp
{
        cl_uint
        seed;
        cl_double
        setupTime;
        cl_double
        totalKernelTime;
        cl_double
        totalProgramTime;
        cl_uint
        *verificationOutput;

        SDKTimer    *sampleTimer;      /**< SDKTimer object */

    public:

        CLCommandArgs   *sampleArgs;
        cl_uint *output;


        Mandelbrot()
        {
            seed = 123;
            output = NULL;
            verificationOutput = NULL;
            sampleArgs->sampleVerStr = SAMPLE_VERSION;
        }

        int setupCL();
        int runCLKernels();

        void mandelbrotRefFloat(cl_uint * verificationOutput,
                                cl_float leftx,
                                cl_float topy,
                                cl_float xstep,
                                cl_float ystep,
                                cl_int maxIterations,
                                cl_int width,
                                cl_int bench);

        int initialize();
        int setup();
        int run();
        int cleanup();


        inline cl_uint getMaxIterations(void)
        {
            return maxIterations;
        }
```

```
        inline void setMaxIterations(cl_uint maxIter)
        {
            maxIterations = maxIter;
        }

        cl_bool showWindow(void);
};

#endif
```

```
#include "MandelbrotDisplay.hpp"
#include "Mandelbrot.hpp"
#ifdef linux
# define GL_GLEXT_PROTOTYPES
#endif // !linux
#include <GL/glew.h>
#include <GL/glut.h>
#include <cstdlib>
#include <cstdio>

// An instance of the Mandelbrot Class
Mandelbrot clMandelbrot;
// Window height, Window Width and the pixels to be displayed
int width;
int height;
unsigned char * output;

int mouseX = 0;
int mouseY = 0;
bool panning = false;
bool zoomIn = false;
bool zoomOut = false;
// display function
void
displayFunc()
{
    if (!clMandelbrot.getBenched())
    {
        glClear(GL_COLOR_BUFFER_BIT);
        glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, output);
        glFlush();
        glutSwapBuffers();
    }
}
// idle function
void
idleFunc(void)
{
    if (panning)
    {
        if (mouseX < (width / 4))
        {
            clMandelbrot.setXPos(clMandelbrot.getXPos()                    -
clMandelbrot.getXStep());
        }
        else if (mouseX > (3 * width / 4))
        {
            clMandelbrot.setXPos(clMandelbrot.getXPos()                    +
clMandelbrot.getXStep());
        }
        if (mouseY < (height / 4))
        {
            clMandelbrot.setYPos(clMandelbrot.getYPos()                    +
clMandelbrot.getYStep());
        }
        else if (mouseY > (3 * height / 4))
```

```
        {
            clMandelbrot.setYPos(clMandelbrot.getYPos()                    -
clMandelbrot.getYStep());
        }
        if (zoomIn)
        {
            clMandelbrot.setXSize(clMandelbrot.getXSize() * 0.99);
        }
        else if (zoomOut)
        {
            clMandelbrot.setXSize(clMandelbrot.getXSize() / 0.99);
        }
    }
    clMandelbrot.run();
    //clMandelbrot.verifyResults();

    glutPostRedisplay();
}
// keyboard function
void
keyboardFunc(unsigned char key, int mouseX, int mouseY)
{
    switch(key)
    {
        // If the user hits escape or Q, then exit
    case GLUT_ESCAPE_KEY:
    case 'q':
    case 'Q':
    {
        cleanup();
        exit(0);
        break;
    }
    case 'c':
    {
        printf("center     (%.13f,     %.13f),     window     width     %.13f\n",
clMandelbrot.getXPos(),
               clMandelbrot.getYPos(),
               clMandelbrot.getXStep());
        break;
    }
    case 'i':
    {
        cl_uint maxIterations = clMandelbrot.getMaxIterations();
        maxIterations = ((maxIterations * 2) < MAX_ITER) ? maxIterations * 2 :
MAX_ITER;
        printf("Setting maxIterations to %d\n", maxIterations);
        clMandelbrot.setMaxIterations(maxIterations);
        break;
    }
    case 'I':
    {
        cl_uint maxIterations = clMandelbrot.getMaxIterations();
        maxIterations = ((maxIterations / 2) > MIN_ITER) ? maxIterations / 2 :
MIN_ITER;
        printf("Setting maxIterations to %d\n", maxIterations);
        clMandelbrot.setMaxIterations(maxIterations);
        break;
    }
    case 'b':
    {
        if (clMandelbrot.getTiming())
        {
            clMandelbrot.setBench(1);
        }
    }
    case 'p':
```

```
    {
        panning = (panning == false)? true : false;
        break;
    }
    default:
        break;
    }
}
void mouseEntry(int state)
{
    if (state == GLUT_LEFT)
    {
        panning = false;
    }
    else
    {
        panning = true;
    }
}
void mouseFunc(int button, int state, int x, int y)
{
    switch (button)
    {
    case GLUT_LEFT_BUTTON:
    {
        if ((state == GLUT_DOWN) && !zoomOut)
        {
            zoomIn = true;
        }
        else
        {
            zoomIn = false;
        }
        break;
    }
    case GLUT_RIGHT_BUTTON:
    {
        if ((state == GLUT_DOWN) && !zoomIn)
        {
            zoomOut = true;
        }
        else
        {
            zoomOut = false;
        }
        break;
    }
    default:
        break;
    }
    mouseX = x;
    mouseY = y;
}

void motionFunc(int x, int y)
{
    mouseX = x;
    mouseY = y;
}
void passiveMotionFunc(int x, int y)
{
    mouseX = x;
    mouseY = y;
}
// initialise display
void initDisplay(int argc, char *argv[])
{
```

```
    initGlut(argc, argv);
    initGL();
}
// initialise glut
void initGlut(int argc, char *argv[])
{
    /* initialising the window */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);

    // Print available keyboard and mouse options.
    printf("\n\tKeyboard Options :\n"
           "\t  'i'  double the number of iterations\n"
           "\t  'I'  halve the number of iterations\n"
           "\t  'b'  benchmark the current frame (when -t is enabled)\n"
           "\t  'c'  print the current center position\n"
           "\t  'p'  toggle panning\n"
           "\n\tMouse Options :\n"
           "\t  Move mouse to pan image\n"
           "\t  Left click to zoom in\n"
           "\t  Right click to zoom out\n\n");

    printf("width %d, height %d\n", width, height);
    glutInitWindowSize(width, height);
    mouseX = width / 2;
    mouseY = height / 2;
    glutInitWindowPosition(0,0);
    glutCreateWindow("Mandelbrot Fractal Generator");

    // the various glut callbacks
    glutDisplayFunc(displayFunc);
    glutIdleFunc(idleFunc);
    glutKeyboardFunc(keyboardFunc);
    glutMouseFunc(mouseFunc);
    glutMotionFunc(motionFunc);
    glutPassiveMotionFunc(passiveMotionFunc);
    glutEntryFunc(mouseEntry);
}

// initialise OpenGL
void
initGL(void)
{
    glewInit();
}

void
mainLoopGL(void)
{
    glutMainLoop();
}

// free any allocated resources
void
cleanup(void)
{
    clMandelbrot.cleanup();
    clMandelbrot.printStats();

}

int main(int argc, char * argv[])
{
    // initialise and run the Mandelbrot kernel
    clMandelbrot.initialize();
    if(clMandelbrot.sampleArgs->parseCommandLine(argc, argv))
    {
```

```
            return SDK_FAILURE;
        }

        if(clMandelbrot.sampleArgs->isDumpBinaryEnabled())
        {
            return clMandelbrot.genBinaryImage();
        }
        else
        {
            int returnVal = clMandelbrot.setup();
            if(returnVal != SDK_SUCCESS)
            {
                return (returnVal == SDK_EXPECTED_FAILURE)? SDK_SUCCESS :
SDK_FAILURE;
            }

            if(clMandelbrot.run()!=SDK_SUCCESS)
            {
                return SDK_FAILURE;
            }
            if(clMandelbrot.verifyResults()!=SDK_SUCCESS)
            {
                return SDK_FAILURE;
            }

            // show window if it is not running in quiet mode
            if(clMandelbrot.showWindow())
            {
                width = clMandelbrot.getWidth();
                height = clMandelbrot.getHeight();
                output = (unsigned char *)clMandelbrot.getPixels();

                initDisplay(argc, argv);
                mainLoopGL();
            }
            cleanup();
        }
        return SDK_SUCCESS;
}
//-------------------------------------------------------------------------
```

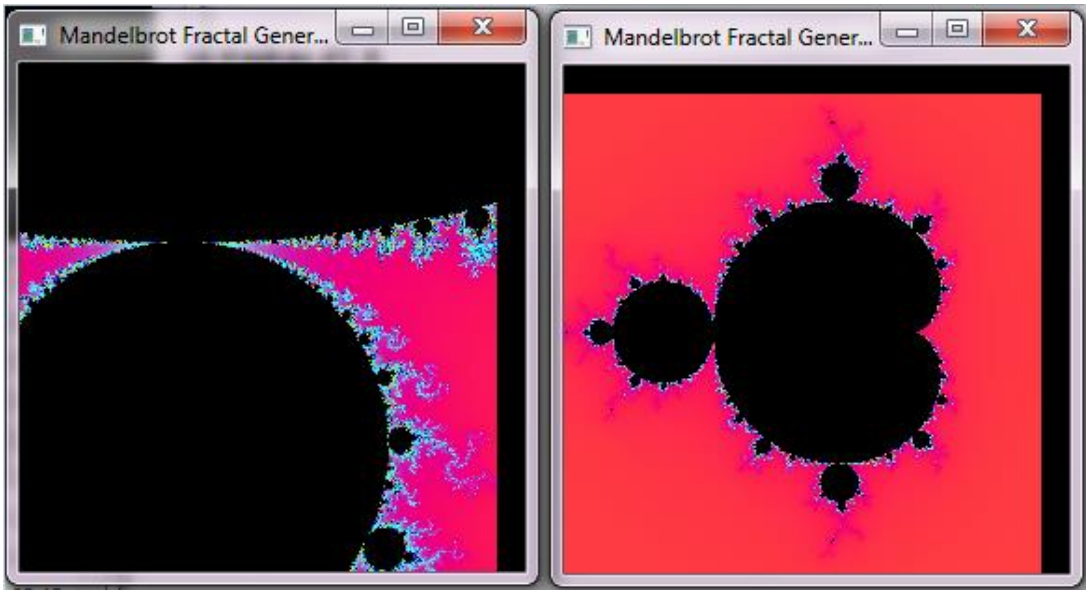


Figura 11. Resultado en pantalla del software que crea imagen fractal del conjunto de Mandelbrot.