UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CARRERA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

TRABAJO DE GRADO PREVIO APROBACIÓN PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERA EN SISTEMAS COMPUTACIONALES

TEMA:

COMPARATIVA DE LOS PARADIGMAS DE PROGRAMACIÓN PARALELA: POR MANEJO DE THREADS, POR PASO DE MENSAJES E HÍBRIDA.

AUTORA: GONZAGA NOGUERA ROSA MERCEDES

DIRECTOR: ING. PABLO LANDETA

IBARRA-ECUADOR

2015

CERTIFICACIÓN DEL DIRECTOR

Certifico que la tesis **"Comparativa de los paradigmas de programación paralela: por manejo de threads, por paso de mensajes e híbrida"**, ha sido realizada con interés profesional y responsabilidad por la señorita: Rosa Mercedes Gonzaga Noguera, portadora de la cédula de identidad número: 100459541-7; previo a la obtención del Título de Ingeniera en Sistemas Computacionales.

Ing. Pablo Landeta
Director de la Tesis

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

La UNIVERSIDAD TÉCNICA DEL NORTE dentro del proyecto Repositorio Digital institucional determina la necesidad de disponer los textos completos de forma digital con la finalidad de apoyar los procesos de investigación, docencia y extensión de la universidad.

Por medio del presente documento dejo sentada mi voluntad de participar en este proyecto, para lo cual pongo a disposición la siguiente investigación:

DATOS DE CONTACTO	
CÉDULA DE IDENTIDAD	100459541-7
APELLIDOS Y NOMBRES	ROSA MERCEDES GONZAGA NOGUERA
DIRECCIÓN	San Miguel de Urcuquí, Tumbabiro, Calle Sucre y Marieta Veintimilla esq. S/N
EMAIL	rositamg1993@gmail.com
TELÉFONO FIJO	06 2934 118
TELÉFONO MÓVIL	0994896858

DATOS DE LA OBRA	
TÍTULO	"COMPARATIVA DE LOS PARADIGMAS DE PROGRAMACIÓN PARALELA: POR MANEJO DE THREADS, POR PASO DE MENSAJES E HÍBRIDA"
AUTORA	ROSA MERCEDES GONZAGA NOGUERA
FECHA	14 de diciembre de 2015
SÓLO PARA PROYECTOS DE GRADO	
PROGRAMA	☒ PREGRADO ☐ POSTGRADO
TÍTULO POR EL QUE OPTA	INGENIERÍA EN SISTEMAS COMPUTACIONALES
DIRECTOR	ING. PABLO LANDETA

2. AUTORIZACIÓN DE USO A FAVOR DE LA UNIVERSIDAD

Yo, ROSA MERCEDES GONZAGA NOGUERA, con cédula de identidad Nro. 100459541-7, en calidad de autora y titular de los derechos patrimoniales del proyecto de grado descrito anteriormente, hago entrega del ejemplar respectivo en forma digital y autorizo a la Universidad Técnica del Norte, la publicación de la obra en el Repositorio Digital Institucional y el uso del archivo digital en la Biblioteca de la Universidad con fines académicos, para ampliar la disponibilidad del material y como apoyo a la educación, investigación y extensión, en concordancia con la Ley de Educación Superior Artículo 143.

UNIVERSIDAD TÉCNICA DEL NORTE

CESIÓN DE DERECHOS DE AUTOR DEL TRABAJO DE INVESTIGACIÓN

A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

Yo, ROSA MERCEDES GONZAGA NOGUERA, con cédula de identidad Nro. 100459541-7, manifiesto mi voluntad de ceder a la Universidad Técnica del Norte los derechos patrimoniales consagrados en la ley de propiedad intelectual del Ecuador, artículo 4, 5 y 6, en calidad de autora del proyecto de grado denominado: **"COMPARATIVA DE LOS PARADIGMAS DE PROGRAMACIÓN PARALELA: POR MANEJO DE THREADS, POR PASO DE MENSAJES E HÍBRIDA"**, que ha sido desarrollado para optar por el título de Ingeniera en Sistemas Computacionales, en la Universidad Técnica del Norte, quedando la Universidad facultada para ejercer plenamente los derechos cedidos anteriormente.

En mi condición de autor me reservo los derechos morales de la obra antes citada. En concordancia suscribo este documento en el momento que hago entrega del trabajo final en formato impreso y digital a la Biblioteca de la Universidad Técnica del Norte

.....

Nombre: ROSA MERCEDES GONZAGA NOGUERA

Cédula: 100459541-7

Ibarra, 14 de diciembre de 2015

3. CONSTANCIAS

La autora (as) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es (son) la (las) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, 14 de diciembre de 2015

LA AUTORA:

.....

Firma

Nombre: ROSA MERCEDES GONZAGA NOGUERA

Cédula: 100459541-7

DEDICATORIA

A **Dios**, que me ha guíado en cada uno de los pasos que he dado en vida.

A mis padres: **Isabel y Alfonso**, quiénes supieron inculcar en mí valores, y siempre me apoyaron en las metas propuestas y que nunca me dejaron renunciar.

A mis hermanos: **Christian, David, Gabriel y Wladimir**, quiénes con consejos y guías supieron hacer que cada día sea una nueva experiencia.

A **toda mi familia**, quienes de una u otra manera se hicieron presentes en diferentes actos y con palabras de aliento siempre me orientaron a seguir adelante.

Rosa Mercedes Gonzaga Noguera

AGRADECIMIENTO

A todas las personas que colaboraron y guiaron en la elaboración del presente trabajo de investigación.

Facultad de Ingeniería en Ciencias Aplicadas de la Universidad Técnica del Norte y a cada uno de los señores docentes y personal administrativo.

De manera especial al **Ingeniero Pablo Landeta**, director de tesis quien con sus conocimientos me guío para la elaboración del presente trabajo.

A **mis compañeros** quienes a lo largo de la carrera me brindaron la ayuda necesaria en cada paso que se dio.

A grandes amigos como: **Erika, José, Elsita, Alejandro, Jenny y Álvaro** quiénes me acompañaron en grandes retos e hicieron que la estadía en la universidad sea inolvidable.

A la **Universidad Técnica del Norte** por brindarme sus conocimientos y poder conocer a tan buenos catedráticos y amigos.

ÍNDICE DE CONTENIDOS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

RESUMEN

El presente trabajo tiene como fin realizar una comparación entre los diferentes paradigmas de programación paralela, en los cuales se encuentran: por manejo de threads, por paso de mensajes, e híbrida; esto se lleva a cabo a través de una investigación explorativa y luego descriptiva. La programación paralela ha ido tomado fuerza con la aparición de las nuevas arquitecturas de multiprocesadores, es por eso que varias empresas han desarrollado lenguajes que ayuden al aprovechamiento de estos nuevos recursos entre los que se encuentran también las GPUs o Tarjetas Gráficas.

Cada lenguaje de programación paralela presenta diferentes características que indican la compatibilidad que tienen con las arquitecturas existentes, en este documento se detalla la descripción de cada paradigma y un lenguaje referencial que se adapta a cada uno de ellos y las ventajas que se tiene al trabajar con las diferentes librerías.

Al final encontramos un análisis construido a partir de la investigación realizada, detallando cuál de los paradigmas y el lenguaje es más conveniente a utilizar, tomando en cuenta los tiempos de ejecución y la interacción con el usuario, basado en la arquitectura utilizada.

SUMMARY

The present work has as objetive realize a comparison between different parallel paradigms of the programming , in which include: by handling of threads, message passing, and hybrid; this is accomplished through an exploratory research and then descriptive. Parallel programming has gone taking strength with the appearance, of new multiprocessor architectures, so several companies have developed languages to assist the use of these new resources among which are also the GPU or Graphics Cards.

Each parallel programming language has different characteristics that indicate the compatibility with the existing architectures, in this document describe the description of each paradigm and referential language that adapts to each of them and the advantages you have when working with detailed different libraries.

To the end we found an analysis constructed from the investigation realized, detailing which of the paradigms and language is more convenient to use, taking into account the execution times and interaction with the user, based on the architecture used.

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

1.- Planteamiento del problema

1.1.- Antecedentes

Desde la creación de la primera computadora, la humanidad ha visto la necesidad de ir avanzando tecnológicamente cada día, buscando facilidades para las tareas que realiza el usuario; y así en conjunto con la electrónica se ha ido mejorando la creación de microprocesadores, teniendo de esta manera un alto rendimiento en los computadores.

De la mano de la computadora vienen los lenguajes de programación, que desde su creación han variado desde FORTRAN el cual fue el primer lenguaje de Programación Estructural conocido hasta llegar a la Programación Orientada a Objetos que se enfoca en escribir nuestros programas en términos de objetos, propiedades y métodos; y la Programación Paralela, teniendo como principal función el manejo de las tareas que necesita el usuario y aprovechando los recursos que ofrecen los procesadores multicore.

1.2.- Situación Actual

Las computadoras en su mayoría contienen procesadores que son mono-núcleo, es decir tienen un solo cerebro para ejecutar todos los procesos, para ello nacen los procesadores multicore, estos pueden repartir los procesos a través de varios cerebros para una mejor ejecución, mejorando así la capacidad de procesamiento.

Con el avance de la tecnología en el área de computación, hoy en día se tiene una amplia gama de lenguajes de programación que gozan de una alta popularidad, con ellos se puede desarrollar desde pequeñas aplicaciones hasta amplios sistemas que son necesarios para la sociedad actual.

La Programación Paralela se ha tomado un amplio campo en el desarrollo de aplicaciones, está tiene como principio la división de un problema de gran

tamaño en problemas más pequeños, así de esta forma se puedan ejecutar varias instrucciones de una manera simultánea, ampliando el uso de los cerebros de un procesador.

1.3.- Prospectiva

La presente investigación pretende establecer las características de los paradigmas de Programación Paralela que existen para un mejor aprovechamiento de los recursos del computador, especialmente los núcleos del procesador.

Después de realizada la investigación, se realizará una comparación entre los diferentes paradigmas, a través del desarrollo de una aplicación en donde se pueda apreciar las ventajas y desventajas que tengan cada uno, y elegir cual es el mejor.

1.4.- Problema

La Programación Paralela ha obtenido mayor importancia debido a la aparición de arquitecturas multicore, teniendo como objetivo el aprovechar de una mejor manera la memoria que los procesadores poseen, y así obtener mejores tiempos de respuesta al correr una aplicación.

Las nuevas aplicaciones exigen mayores recursos que al tener un solo procesador impide un rápido procesamiento de estas, impidiendo la resolución de grandes problemas en un menor tiempo, por lo cual al unir múltiples procesadores se tendrá una mejor ejecución de estas, dando así inicio al paralelismo.

1.5.- Justificación

La mayoría de computadores actuales vienen con procesadores de varios núcleos, pero los programas que se utilizan sobre ellos han sido creados para el uso de un solo núcleo, hoy en día la Programación Paralela tiene como fin el aprovechamiento de todos los núcleos del procesador con la división de procesos y tareas.

La Programación Orientada a Objetos sigue siendo un ente muy fuerte, por lo cual los programas de un solo núcleo siguen vigentes, la presente investigación tiene como meta establecer los beneficios de la Programación Paralela para una correcta repartición de los recursos al momento de estar trabajando sobre un computador.

1.6.- Objetivos

1.6.1.- Objetivo General

Comparar los paradigmas de programación paralela para mejorar el uso de los recursos del computador mediante una investigación explorativa y descriptiva.

1.6.2.- Objetivos Específicos

- ✓ Realizar el planteamiento de por qué es necesario paralelizar
- ✓ Investigar los tipos de paralelismo y sus arquitecturas
- ✓ Definir los paradigmas de programación paralela
- ✓ Establecer bases de comparación
- ✓ Determinar cuál es el mejor paradigma de programación paralela

1.7.- Alcance

Esta investigación se encargará de explorar los diferentes paradigmas de programación paralela existentes, y las ventajas que tienen al aplicarlas en un computador, para ello se determinaran sus características y pondrán a prueba los estándares que presenta cada una.

Para empezar el estudio se realizará una investigación exploratoria, así se entrará en contacto con el tema, y se podrá ir obteniendo la información suficiente para luego realizar una investigación a mayor profundidad, para la cual posteriormente se aplicará el tipo de investigación descriptiva, en donde se obtendrá de manera más específica los datos deseados de cada paradigma.

A continuación se detalla los tipos de paradigmas y los lenguajes de programación a utilizar en cada uno.

Programación paralela por Manejo de Threads

El sistema ejecuta varios threads (hilos) simultáneamente, y estos aprovechan los recursos disponibles.

Lenguaje a utilizar: OpenCL.

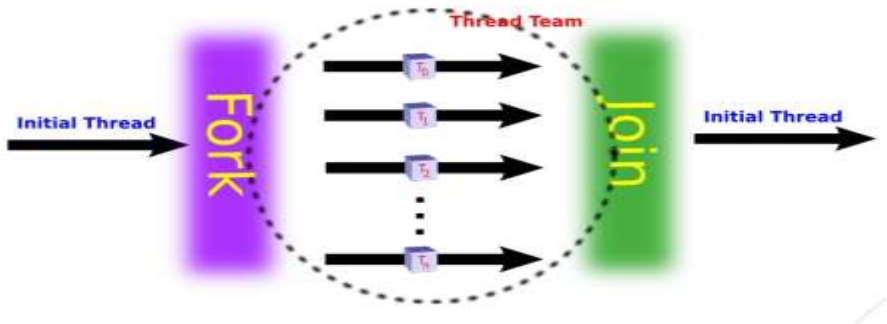

Ilustración 1 Programación Paralela por Manejo de Threads

Fuente: Diapositivas de Programación Paralela y Distribuida desarrolladas por Pedro Antonio Varo Herrero

Programación Paralela por Paso de Mensajes

Tiene que haber como mínimo dos procesos: enviar un mensaje y recibir un mensaje.

Lenguaje a utilizar: MPI.

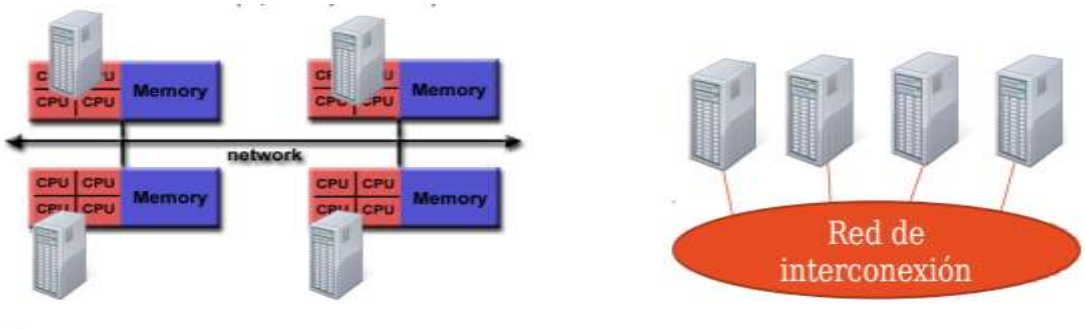

Ilustración 2 Programación Paralela por Paso de Mensajes

Fuente: Diapositivas de Programación Paralela y Distribuida desarrolladas por Pedro Antonio Varo Herrero

Programación Paralela Híbrida

Es la combinación de la Programación Paralela por manejo de Threads y por paso de mensajes.

Lenguaje a utilizar: CUDA

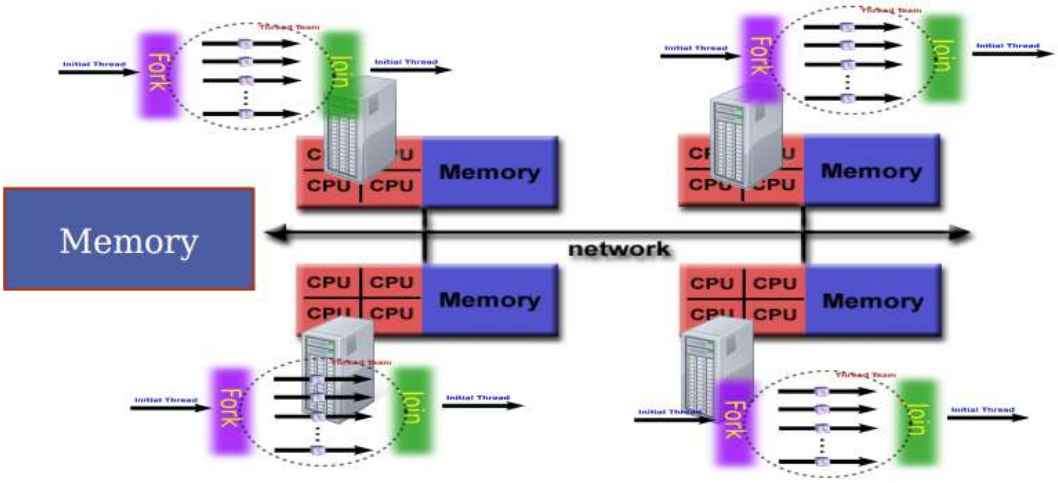

Ilustración 3 Programación Paralela Híbrida

Fuente: Diapositivas de Programación Paralela y Distribuida desarrolladas por Pedro Antonio Varo Herrero

Al final de la investigación se utilizará una herramienta benchmark que monitoree el rendimiento del computador, por medio de la cual se utilizará los datos para realizar las comparativas entre los paradigmas de programación paralela.

CAPÍTULO II

MARCO TEÓRICO

2.- Marco Teórico

2.1.- Tipos de Investigación

2.1.1.- Investigación Explorativa

"*Los estudios exploratorios nos permiten aproximarnos a fenómenos desconocidos, con el fin de aumentar el grado de familiaridad y contribuyen con ideas respecto a la forma correcta de abordar una investigación en particular. Con el propósito de que estos estudios no se constituyan en pérdida de tiempo y recursos, es indispensable aproximarnos a ellos, con una adecuada revisión de la literatura."* (Grajales, 2000)

Al no contar con una información exacta del tema, se debe aplicar una investigación explorativa, y de esta manera ir adquiriendo conocimiento para adentrarse en el tema.

2.1.2.- Investigación Descriptiva

"*Los estudios descriptivos buscan desarrollar una imagen o fiel representación (descripción) del fenómeno estudiado a partir de sus características. Describir en este caso es sinónimo de medir. Miden variables o conceptos con el fin de especificar las propiedades importantes de comunidades, personas, grupos o fenómeno bajo análisis." (Grajales, 2000)*

Con la obtención de datos en la investigación exploratoria se puede acercar a la investigación descriptiva, detallando la información encontrada.

2.2.- ¿Qué es una comparativa de herramientas?

Una comparativa es el estudio de las características de dos o más herramientas con el fin de analizar las ventajas y desventajas que estas tengan, a través de pruebas e información que se obtenga.

La comparativa de herramientas ayuda a tener una mejor idea de lo que se está comparando, pudiendo dar conclusiones sobre el manejo o las características que se tenga.

2.3.- Arquitectura de Computación Paralela

Para el trabajo de programación paralela se debe tomar en muy en cuenta las características del computador en el que se va a trabajar, la combinación de varias configuraciones nos pueden llevar a obtener distintas arquitecturas en las cuales de acuerdo al número de procesadores o el tamaño de memoria se puede elegir la más adecuada para resolver las diferentes aplicaciones.

Una de las más conocidas clasificaciones de la arquitectura de computadores es la taxonomía de Flynn (Flynn & Rudd, 1996) quien tomó dos aspectos a consideración: el flujo de instrucciones y el flujo de datos, de acuerdo a la multiplicidad de cada uno, en estas encontramos:

SISD[1]: estás computadoras manejan un único flujo de instrucciones y un único flujo de datos, son computadoras de una CPU; éstas son especialmente usadas para la ejecución de programas secuenciales en donde todas las instrucciones están bajo el mando de una única unidad de control.

SIMD[2]: se caracterizan por tener un único flujo de instrucciones y múltiple flujo de datos, su arquitectura se caracteriza por contar con varias CPU, estás pueden ejecutar el mismo flujo de instrucciones sobre distintos datos; las SIMD incluyen los computadores vectoriales, la unidad de control se encarga de enviar la orden para ejecutar la instrucción sobre diferentes grupos de datos que provienen de varios flujos.

MISD[3]: éstas computadoras tienen múltiple flujo de instrucciones y un único flujo de datos, para ello las diferentes unidades de procesamiento trabajan sobre el único flujo de datos con diferentes instrucciones, ninguna de las arquitecturas que existen implementan este tipo de computadora.

MIMD[4]: este tipo de computadoras tienen un múltiple flujo de instrucciones y un múltiple flujo de datos, las unidades de instrucciones son muy

[1] **SISD:** Single Instruction Single Data.
[2] **SIMD:** Single Instruction Multiple Data
[3] **MISD:** Multiple Instruction Single Data
[4] **MIMD:** Multiple Instruction Multiple Data

independientes y cada una maneja su propio flujo de datos, en MIMD existen dos tipos de computadoras: las de Multiprocesadores, son arquitecturas con memoria compartida; y las Multicomputadoras que manejan la arquitectura con memoria distribuida. MIMD se caracteriza por ser fuertemente acoplado a los `procesos si su grado de interacción es alto, caso contrario es muy débil su acoplación si la interacción es poco frecuente. Hoy en día la mayoría de las computadoras en el mercado son MIMD.

Según Piccoli (Piccoli, 2011) existe otra taxonomía, esta se deriva de considerar a la memoria y la forma de comunicación entre los distintos procesadores como las características discriminantes. En esta clasificación hay dos grupos: los multiprocesadores con memoria compartida y los multiprocesadores con memoria distribuida, el primero agrupa a todas las arquitecturas integradas por un conjunto de procesadores, los cuales comparten la arquitectura paralela, el segundo grupo tiene a todas las arquitecturas formadas por varios procesadores y sus memorias conectados mediante una red de interconexión. Cada procesador tiene su memoria local y se comunica con el resto por medio de mensajes a través de la red, a esta arquitectura también se la conoce como multicomputadoras.

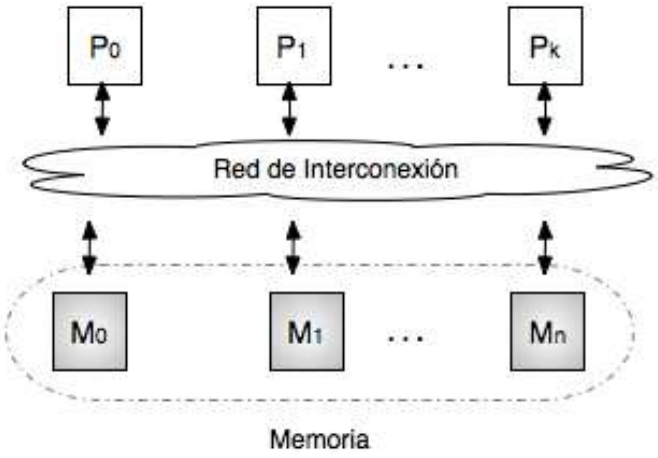

Ilustración 4: Arquitectura de una máquina paralela con memoria distribuida

Fuente:(Piccoli, 2011)

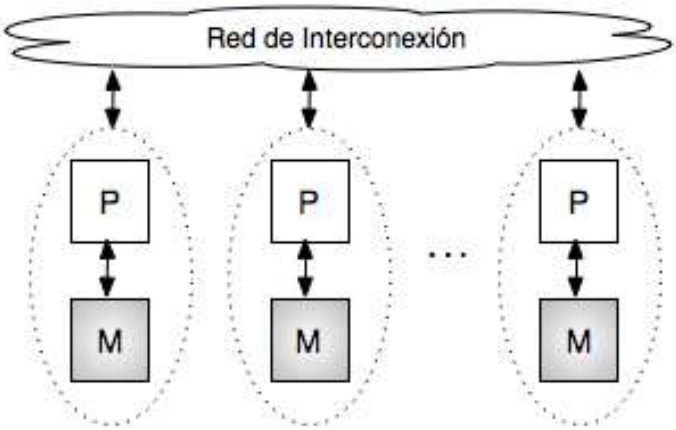

Ilustración 5: Arquitectura de una máquina con paso de mensajes

Fuente: (Piccoli, 2011)

2.4.- ¿Qué son las GPU's?

Una GPU[5] actual se la puede describir como un vector de varios procesadores, que tienen la facilidad de ejecutar instrucciones sobre diferentes datos de una manera simultánea o paralela.

"Para los programadores de CUDA C y OpenCL, Las GPU's son procesadores de computación masivamente numéricas paralelas programadas en C con extensiones. Uno no necesita entender los algoritmos gráficos o terminología para poder programar estos procesadores. Sin embargo, la comprensión de la herencia de estos procesadores gráficos ilumina las fortalezas y debilidades de ellos con respecto a las principales pautas de cálculo. En particular, la historia ayuda a aclarar la razón de ser de las principales decisiones de diseño arquitectónico de las GPU's modernas programables: lectura masiva múltiple, relativamente pequeñas memorias caché en comparación con las CPU, y el diseño de interfaz de memoria de ancho de banda céntrica. Mirar en torno a la evolución histórica también es probable como dar a los lectores el contexto necesario para proyectar la evolución futura de la GPU como dispositivos informáticos".(Kirk & Wen-mei, 2012)

Piccoli relata que la GPU desde un inicio fue un procesador con muchos recursos computacionales que ha adquirido notoriedad para su uso en aplicaciones

[5] **GPU:** Unidad de Procesamiento Gráfico con sus siglas en Inglés

de propósito general, y así paso de ser un procesador con funciones especiales a ser considerado un procesador masivamente paralelo, convirtiéndose en la arquitectura base de aplicaciones paralelas.

2.4.1.- Estructura del Pipeline Gráfico

Es necesario entender a la GPU como una unidad de procesamiento de propósito general, así como también su funcionamiento desde el punto de vista de la especialidad del hardware; esto permitirá realizar una analogía de cada uno de los mecanismos propios con los que se aplican en GPGPU.

El funcionamiento de las GPU es resumido como un pipeline[6] de procesamiento formado por etapas con una función especializada en cada uno, siendo estás ejecutadas en paralelo y dadas un orden pre establecido, cada etapa recibe como entrada la salida de la etapa anterior y su salida es la entrada de la siguiente etapa. La aplicación envía a la GPU una secuencia de vértices, agrupados en lo que se denomina primitivas geométricas[7]: polígonos, líneas y puntos, los cuales son tratados en las siguientes etapas:

Primera etapa: Transformación de Vértices, en ella se lleva a cabo una secuencia de operaciones matemáticas sobre cada uno de los vértices suministrados en la aplicación, estas operaciones son: transformación de la posición del vértice en una posición en pantalla, generación de las coordenadas para la aplicación de texturas y asignación de color a cada vértice.

Segunda etapa: Ensamblado de primitivas y rasterización, aquí los vértices generados en la etapa anterior son agrupados en primitivas geométricas basándose en la secuencia inicial de vértices obteniendo como resultado una secuencia de triángulos, líneas y puntos siendo los últimos sometidos a rasterización[8], con esta

[6] **Pipeline:** es un conjunto de elementos procesadores de datos conectados en serie.
[7] **Primitivas geométricas:** son grupos de diversos objetos básicos, por ejemplo: los de tipo bidimensional o 2d: son el círculo, el cuadrado y otras formas básicas.
[8] **Rasterización:** proceso por el cual se determina el conjunto de píxeles cubiertos por una primitiva determinada.

se obtienen un conjunto de localizaciones de píxeles y conjuntos de fragmentos[9], si estos superan con éxito el resto de etapas se actualizará la información del pixel.

Tercera etapa: Interpolación, texturas y colores, los fragmentos que se han obtenido de la segunda etapa son sometidos a operaciones de rasterización, operaciones matemáticas y de textura y determinación del color final del fragmento, esta etapa permite descartar fragmentos y que su valor en memoria no sea actualizado, como resultado final esta etapa emite uno o ningún fragmento actualizado.

Cuarta etapa: Operaciones Raster, finalmente se analiza cada fragmento sometiéndolo a un conjunto de pruebas relacionadas con aspectos gráficos del mismo, determinando el valor que tomará el pixel generado en memoria a partir del fragmento original, si las pruebas fallan se descarta el pixel, caso contrario se realiza la escritura en memoria como paso final.

Un pipeline gráfico recibe como entrada la representación de una escena en 3D y se tiene como resultado una imagen 2D que se presenta en pantalla, realizando cada una de las transformaciones en el pipeline, estas pueden implementarse de acuerdo al software o hardware.

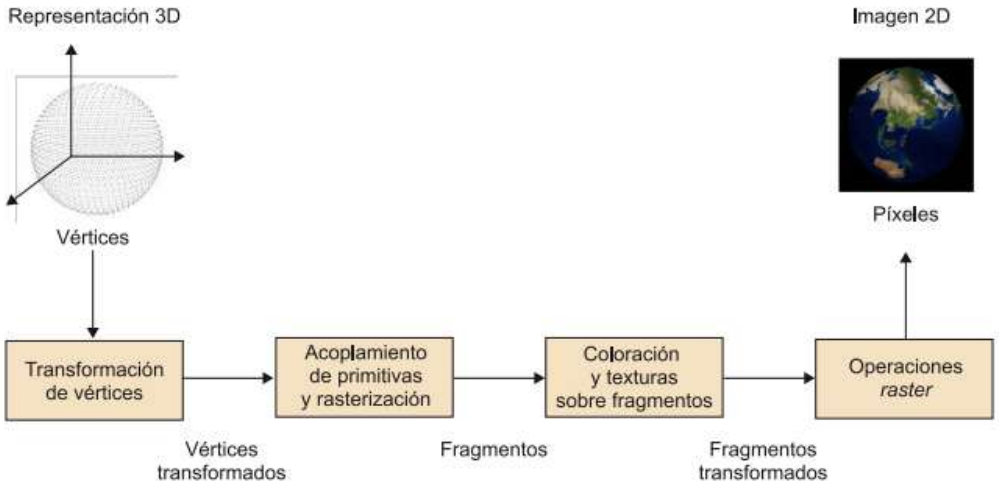

Ilustración 6 Funcionalidades del Pipeline Gráfico

Fuente Guim, F.

[9] **Fragmentos:** en este se asocia una localización de pixel e información relativa a su color y una o más coordenadas de textura.

2.4.2.- Arquitecturas orientadas al procesamiento gráfico GPU

Los fabricantes de procesadores han ido buscando alternativas con el fin de mejorar el rendimiento de los computadores con lo que han tomado en cuenta varios factores:

- ✓ Limitaciones fundamentales en la fabricación de circuitos integrados como, por ejemplo, el límite físico de integración de factores
- ✓ Restricciones de energía y calor debidas, por ejemplo a los límites de tecnología CMOS y a la elevada densidad de potencia eléctrica.
- ✓ Limitaciones en el nivel de paralelismo a nivel de instrucción.

La industria encontró una solución al desarrollar procesadores con múltiples núcleos que se centran en el rendimiento de ejecución de aplicaciones paralelas en contra de programas secuenciales. Actualmente casi todos los ordenadores poseen procesadores multinúcleo, desde dispositivos móviles con procesadores duales hasta procesadores con más de docena de núcleos para servidores y estaciones de trabajo por lo cual la computación paralela ha dejado de ser exclusiva para supercomputadores y sistemas de altas prestaciones, y así estos los dispositivos brindan funcionalidades más sofisticadas que sus predecesores en computación paralela.

Al existir una alta demanda de cálculo por parte de los usuarios en el ámbito de generación de gráficos tridimensionales ha hecho que el hardware evolucione de manera rápida en la computación gráfica, a esto se suma la industria de los videojuegos que ejerce una gran presión económica para mejorar la capacidad de realizar una gran cantidad de cálculos en coma flotante para procesamiento gráfico.

Los fabricantes de GPU buscan la manera de optimizar el rendimiento de los cálculos y han optado por explotar un número masivo de flujos de ejecución, y así mientras un flujo está en espera para el acceso a memoria, el resto pueda seguir ejecutando una tarea pendiente. El modelo GPU requiere menos lógica de control para cada flujo de ejecución por lo disponen una memoria caché que permite que los flujos que comparten memoria tenga el suficiente ancho de banda sin tener que

ir a la DRAM, por lo cual mucha más área del chip se dedica al procesamiento de datos en coma flotante.

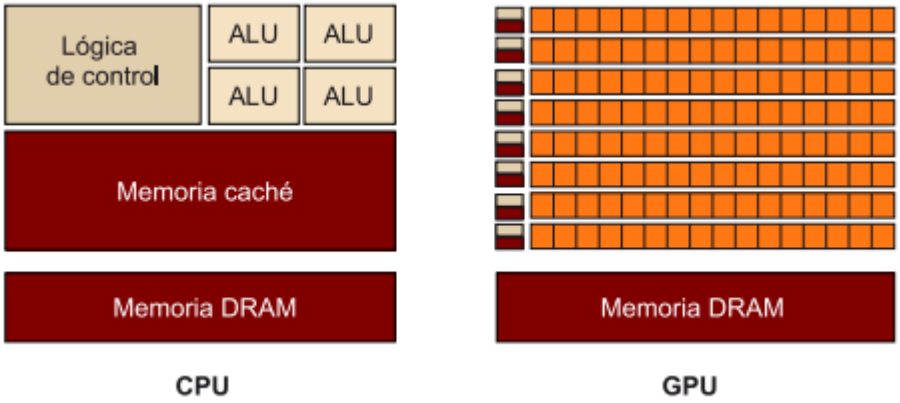

Ilustración 7 Comparativa de la superficie dedicada típicamente a computación, memoria y lógica de control para CPU y GPU

Fuente Guim, F

Los GPU presentan algunas características básicas como:

- ✓ Siguen el modelo SIMD esto quiere decir que todos los flujos ejecutan la misma instrucción, por lo tanto los núcleos decodifican la instrucción una única vez.
- ✓ La velocidad de ejecución se basa en la explotación de la localidad de los datos, tanto en la localidad temporal como en la localidad espacial.
- ✓ La memoria de una GPU se organiza en varios tipos de memoria, tiempos de acceso y modos de acceso.
- ✓ El ancho de banda de la memoria es mayor.

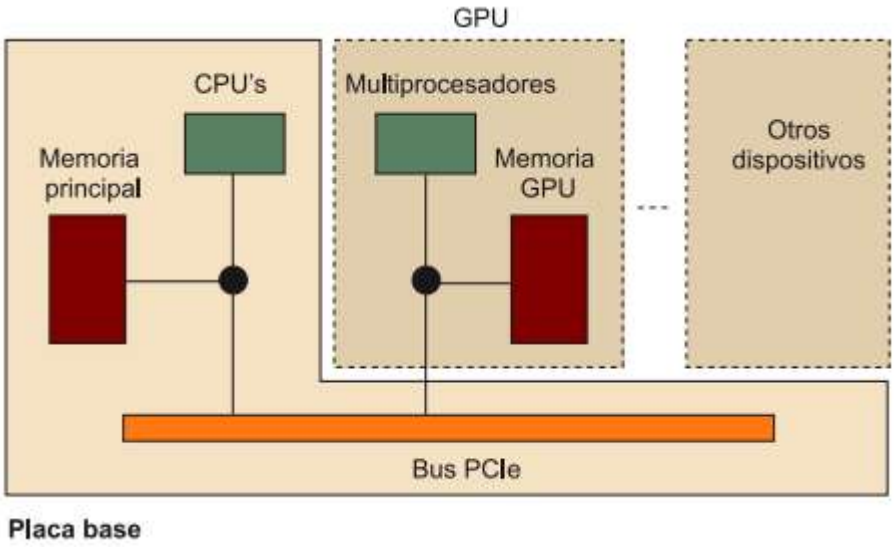

Ilustración 8 Interconexión entre CPU y GPU mediante PCIe

Fuente Guim, F

2.4.2.1.- Arquitectura Nvidia

En la industria comercial, Nvidia ofrece diferentes productos, estos se dividen en tres familias básicamente:

- ✓ GeForce: está orientada al mercado de consumo multimedia (videojuegos, edición de vídeo, fotografía digital, entre otros).
- ✓ Cuadro: se dedica a soluciones profesionales que requieren modelos 3D, como los sectores de la ingeniería o la arquitectura
- ✓ Tesla: orientada a la computación de altas prestaciones, como el procesamiento de información sísmica, simulaciones de bioquímica, modelos meteorológicos y de cambio climático, computación financiera o análisis de datos.

Nvidia GeForce 6

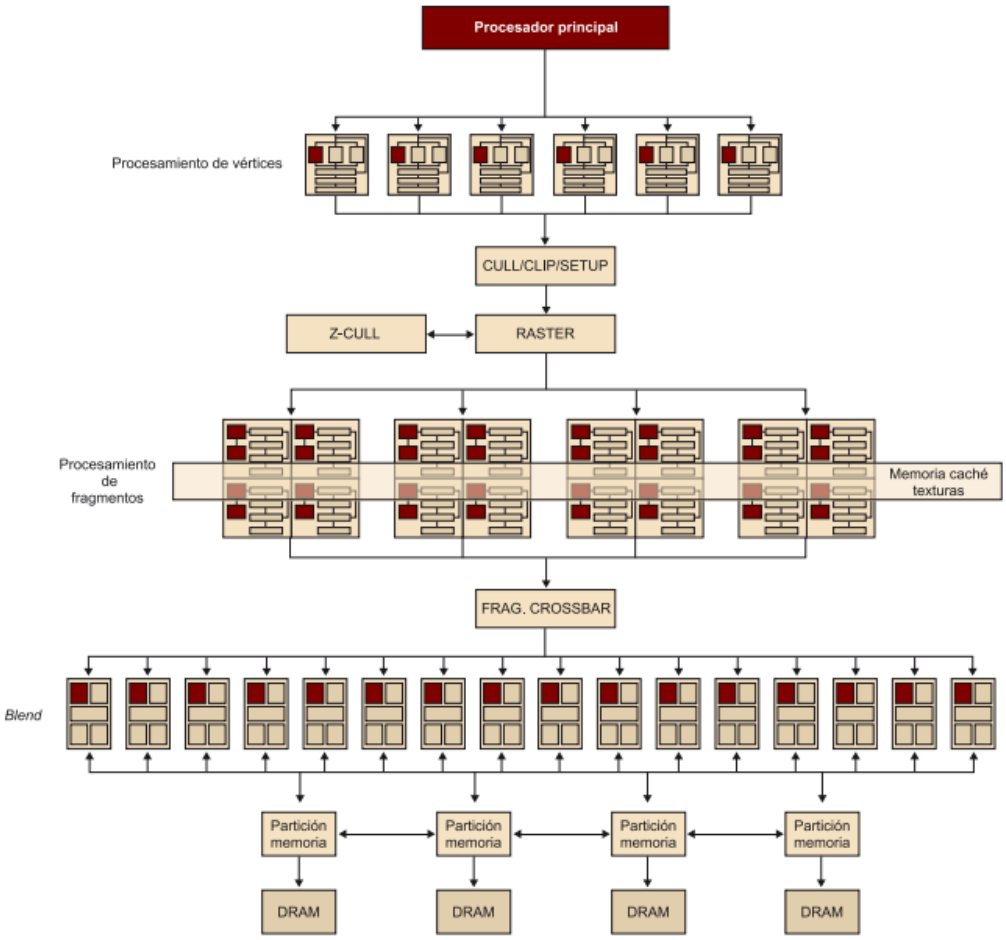

Ilustración 9 Esquema de la arquitectura de la GPU GeForce 6 de Nvidia

Fuente Guim, F

El procesador principal envía a la unidad gráfica tres tipos de datos: instrucciones, texturas y vértices. Los procesadores de vértices aplican un programa específico que se dedica a ejecutar las transformaciones sobre cada uno de los vértices de la entrada. GeForce 6 fue la primera tarjeta gráfica que permitía que un programa ejecutado en el procesador de vértices fuera capaz de consultar datos de textura. Todas las operaciones se llevan a cabo con una precisión de 32 bits en coma flotante. El número de procesadores de vértices disponibles es variable en función del modelo de procesador, a pesar de que suele estar entre dos y dieciséis.

Cada uno de los procesadores de vértices tienen conexión a la memoria caché de vértices, en ella se almacena datos relativos a vértices antes y después de haber sido procesados por el procesador de vértices. Siguiendo el proceso los

vértices se agrupan en primitivas, tomando en cuenta la Ilustración 9 el bloque denominado CULL/CLIP/SETUP ejecuta operaciones específicas para cada primitiva, que elimina, transforma o prepara para la etapa de rasterización. El bloque de rasterización calcula que píxeles son afectados con cada primitiva y hace uso del bloque Z-CULL para descartar píxeles. Una vez después de completar estás acciones los fragmentos se vuelven candidatos a píxeles y pueden ser transformados por el procesador de fragmentos.

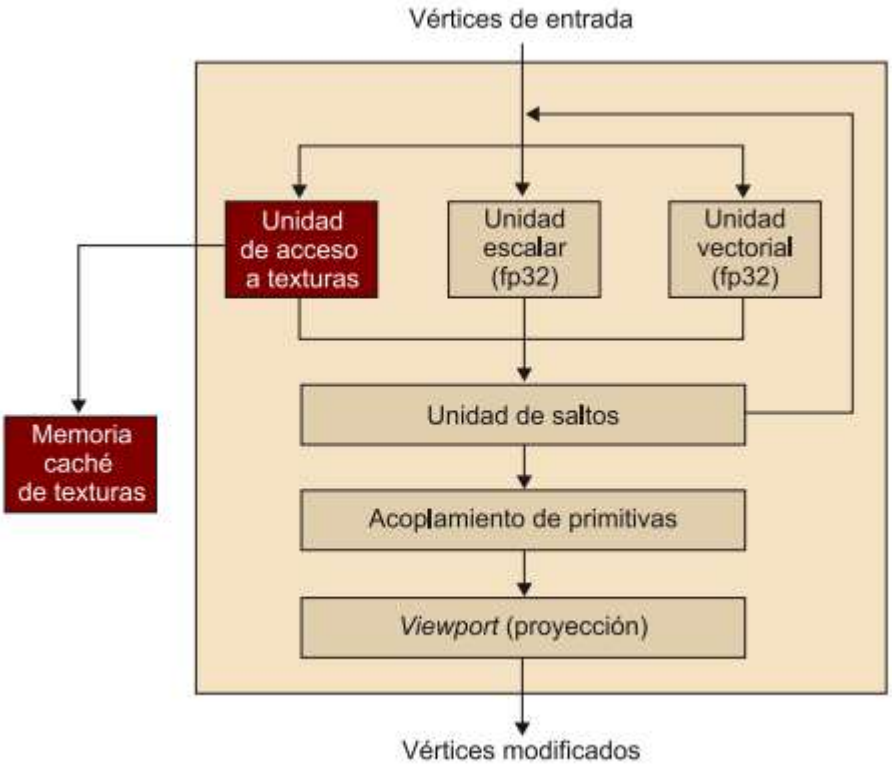

Ilustración 10 Procesador de vértices de la serie GeForce 6 de Nvidia

Fuente Guim, F

En la Ilustración 10 se puede observar la arquitectura de los procesadores de fragmentos típicos de la serie GeForce 6 de Nvidia, estos se dividen en dos partes: la unidad de textura, que se dedica al trabajo con texturas, y la unidad de procesamiento de fragmentos, que opera junto a la unidad de texturas, sobre cada uno de los fragmentos de entrada. Estas unidades operan de manera simultánea para aplicar un mismo programa a cada uno de los fragmentos de manera independiente.

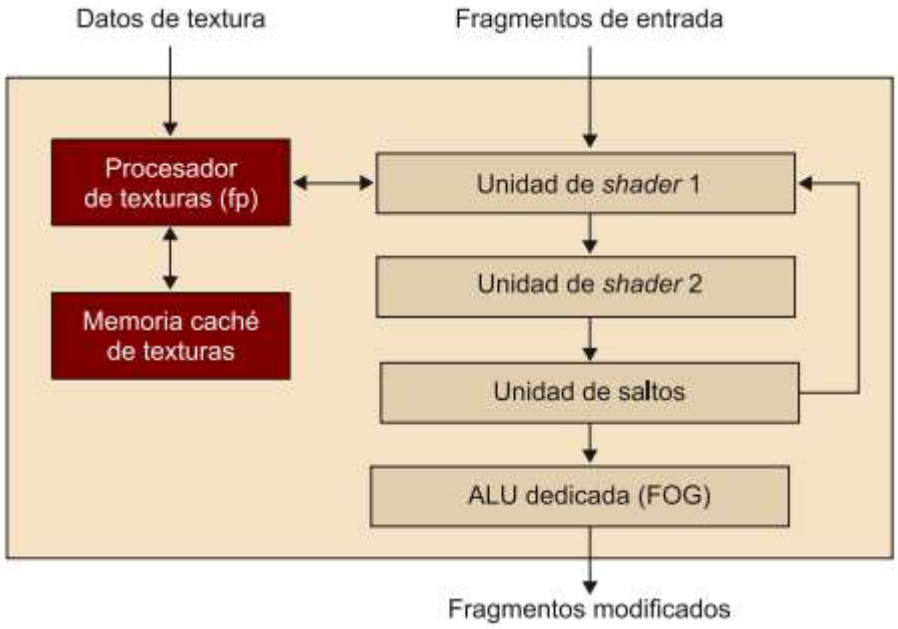

Ilustración 11 Procesador de fragmentos de la serie GeForce 6 de Nvidia

Fuente Guim, F

Al igual que en el procesador de vértices, los datos de textura se pueden almacenar en memorias caché en el chip mismo con el fin de reducir el ancho de banda den la memoria y aumentar así el rendimiento del sistema.

El procesador de fragmentos carga datos desde la memoria a través de la unidad de texturas, y de manera opcional se pueden filtrar los fragmentos antes de ser recibidos por el procesador de fragmentos. El procesador de fragmentos recibe los datos en formato fp32 o fp16, mientras que la unidad de textura soporta una gran cantidad de formatos de datos y tipo de filtrados. Los operadores de fragmentos poseen dos unidades de procesamiento que operan con una precisión de 32 bits. Estos fragmentos viajan por las unidades de shader (Ilustración 11) y por la unidad de saltos, después vuelven hacia las unidades de shader para seguir ejecutando las operaciones. Esta operación sucede una vez por cada ciclo de reloj, en general, es posible realizar ocho operaciones matemáticas en el procesador de fragmentos por ciclo reloj o cuatro en el supuesto de que se produzca una lectura de datos de textura en la primera unidad de shader.

La memoria del sistema se divide en cuatro partes independientes, que se construyen a partir de memorias dinámicas para así reducir los costos de fabricación. Todos los datos procesados en el pipeline gráfico se almacenan en

memoria DRAM, mientras que las texturas y los datos de entrada (vértices) se pueden guardar tanto en la memoria del sistema como la memoria DRAM. Así estas particiones de memoria proporcionan un subsistema de memoria de bastante anchura (256 bits) y flexible, logrando velocidades de transferencia cercanas a los 35GB/s, en caso de ser memorias DDR con velocidad de reloj de 550Mhz, 256 bits por ciclo reloj y dos transferencias por ciclo.

Nvidia G80

Nvidia G80 se define como una arquitectura totalmente unificada, sin diferenciación a nivel de hardware entre las diferentes etapas que forman el pipeline gráfico, está orientada a la ejecución masiva de flujos y que se ajusta al estándar IEEE 754.

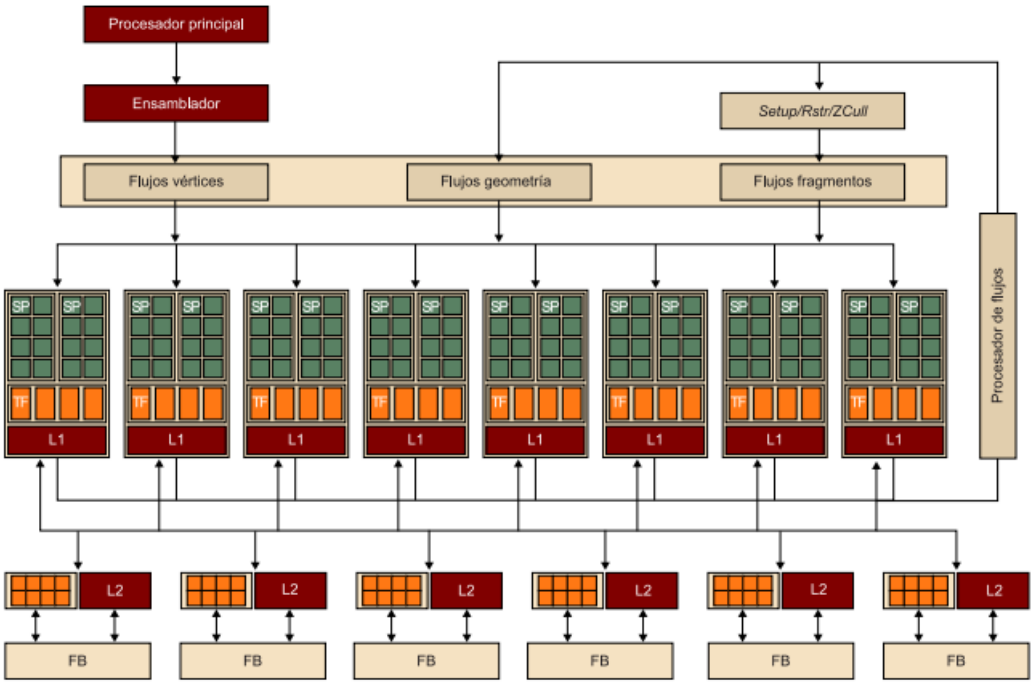

Ilustración 12 Arquitectura unificada de la serie G80 de Nvidia

Fuente Guim, F

Cuando Nvidia lanzó la arquitectura G80 presentó una mejora en la capacidad de procesamiento gráfico y un incremento de las prestaciones respecto a la generación anterior de GPU, y a la vez fue clave para la mejora de la capacidad de cálculo en coma flotante. Debido a la arquitectura unificada, el número de etapas

del pipeline se reduce de manera significativa y pasa de un modelo secuencial a un modo cíclico. El pipeline clásico utiliza diferentes tipos de shaders, por medio de los cuáles los datos se procesan secuencialmente, a diferencia que en la arquitectura unificada existe una única unidad de shaders no especializados que procesan los vértices como datos de entrada en diferentes pasos.

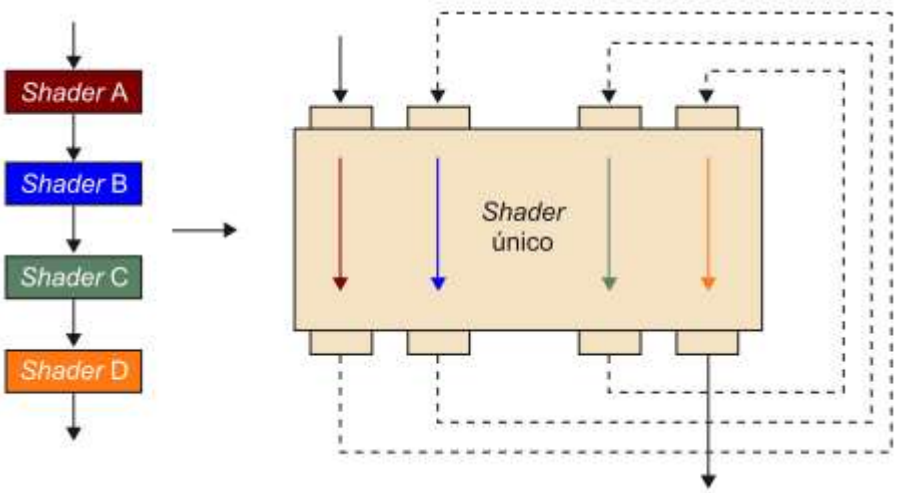

Ilustración 13 Comparación entre un pipeline secuencial y el cíclico de la arquitectura unificada

Fuente Guim, F

Para ejecutar un programa en la arquitectura G80 hay que tomar en cuenta dos etapas diferenciales: la primera etapa procesa los datos de entrada mediante un hardware especializado, este se encarga de distribuir los datos de tal manera que se puedan utilizar el máximo número de unidades funcionales para obtener la máxima capacidad de cálculo mientras se ejecuta el programa. En la segunda etapa, el controlador global de flujos se encarga de controlar la ejecución de los flujos que ejecutan los cálculos de manera coordinada, este también determina en cada momento qué flujos y su tipo serán enviados a cada unidad de procesamiento que componen la GPU. En las unidades de procesamiento existe un planificador de flujos que se encarga de dirigir la gestión interna de los flujos y de los datos.

El núcleo de procesamiento de los shaders consta de ocho bloques de procesamiento, cada uno contiene dieciséis unidades de procesamiento principal, denominadas streaming processors (SP), cada bloque posee u planificador de tareas, memoria caché de nivel 1 y unidades de acceso y filtrado de texturas propias, de esta manera cada grupo comparte unidades de textura y memoria caché.

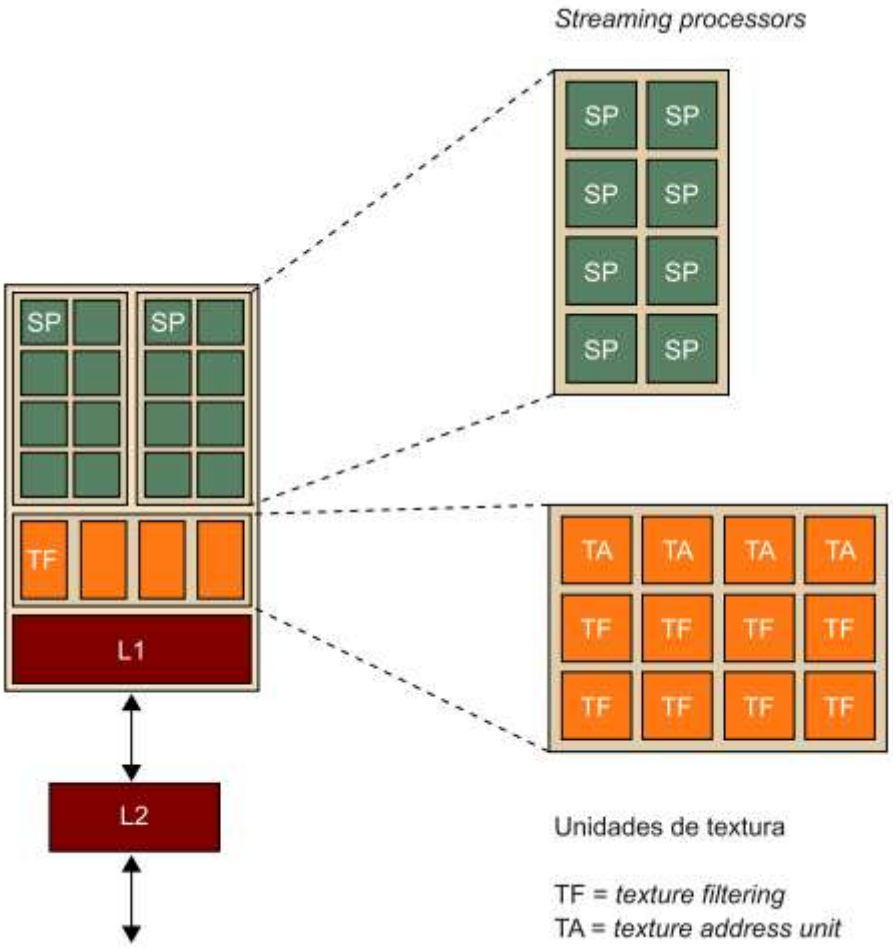

Ilustración 14 Streaming processors y unidades de textura que forman un bloque en la arquitectura G80 de Nvidia

Fuente Guim, F

Cada SP puede llevar a cabo operaciones matemáticas o bien direccionamiento de datos en memoria y la transferencia posterior, casa procesador es una ALU que opera sobre datos escalares de 32 bits de precisión, contrastando con el apoyo vectorial que ofrecían las arquitecturas anteriores a los procesadores de fragmentos.

Internamente, cada bloque está organizado en dos grupos de ocho SP. El planificador interno planifica la misma instrucción sobre los dos subconjuntos de SP que forman el bloque y cada uno se encarga de un cierto número de ciclos de reloj, dependiendo del flujo que esté en ejecución en ese momento, además tiene acceso al conjunto de registros propio de este, los bloques también pueden acceder tanto al conjunto de registros global como a dos zonas de memoria adicionales, solo

de lectura, estas se llaman memoria caché global de constantes y memoria caché global de texturas.

Además de las operaciones antes mencionadas los bloques pueden compartir información con el resto de bloques por medio sel segundo nivel de la memoria caché. Para compartir datos en modo lectura/escritura es necesario el uso de la memoria DRAM de vídeo, junto a la penalización consecuente que representa el tiempo de ejecución.

Existen implementaciones posteriores a G80 que pertenecen a la familia de Tesla que ofrecen más prestaciones, y un número más elevado se SP, más memoria y ancho de banda. A continuación se presenta las características de algunas de estas implementaciones.

Modelo	Arqui-tectura	Reloj (MHz)	Núcleos		Memoria				Rendi-miento - pico teórico (GFLOP)
			Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
C870	G80	600	128	1.350	GDDR3	1.536	1.600	76	518
C1060	GT200	602	240	1.300	GDDR3	4.096	1.600	102	933
C2070	GF100	575	448	1.150	GDDR5	6.144	3.000	144	1.288
M2090	GF110	650	512	1.300	GDDR5	6.144	3.700	177	1.664

Ilustración 15 Comparativa de los modelos de la familia Tesla

Fuente Guim, F

Fermi es la arquitectura que implementan las últimas generaciones de Nvidia, esta proporciona soluciones cada vez más masivas a nivel de paralelismo, tanto desde el número de SP comode flujos que se pueden ejecutar en paralelo.

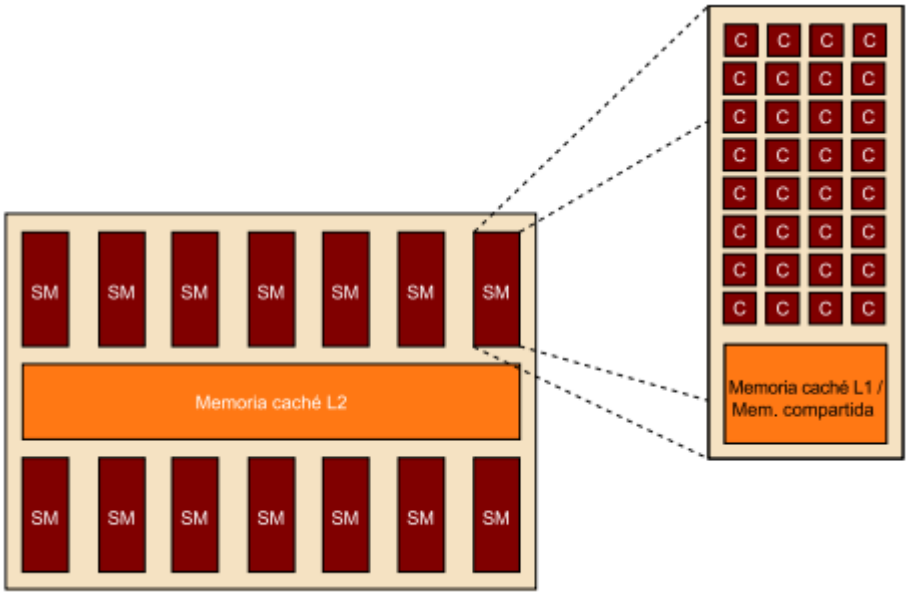

Ilustración 16 Esquema simplificado de la arquitectura Fermi

Fuente Guim, F

En la Ilustración 16 se observa un esquema simplificado de la arquitectura Fermi, este dispositivo está formado por un conjunto de Streaming Multiprocessors (SM) que comparten memoria caché L2. Cada SM contiene 32 núcleos y cada uno de ellos puede ejecutar una instrucción entera o en punto flotante por ciclo reloj. Además de la memoria caché también incluye una interfaz con el procesador principal, un planificador de flujos y múltiples interfaces de DRAM.

La GPU en fermi está compuesta por un conjunto de unidades computacionales con unos pocos elementos que la apoyan a nivel de abstracción. Este modelo tiene como objetivo principal dedicar la mayor parte de la superficie del chip y de la corriente eléctrica a la aplicación en cuestión y maximizar así el rendimiento en coma flotante.

2.4.2.2.- Arquitectura AMD (ATI)

La tecnología GPU desarrollada por AMD/ATI ha ido en crecimiento al igual que la tecnología Nvidia hacia la computación GPGPU.

AMD R600

Esta serie es la equivalente a la Nvidia G80 y la arquitectura CU que fue implementada por la familia Evergreen de AMD, esta es un desarrollo nuevo y está asociada al modelo de programación en OpenCL.

La arquitectura unificada de AMD al igual que la de Nvidia G80 incluye un shader de geometría que permite ejecutar operaciones de creación de geometría en la GPU y así no sobrecargar a la CPU. Esta serie incorpora 320SP, un valor mayor a los 128 de la G80. Esto no quiere decir que sea más potente, ya que los SP en las dos arquitecturas funcionan de una forma muy diferente.

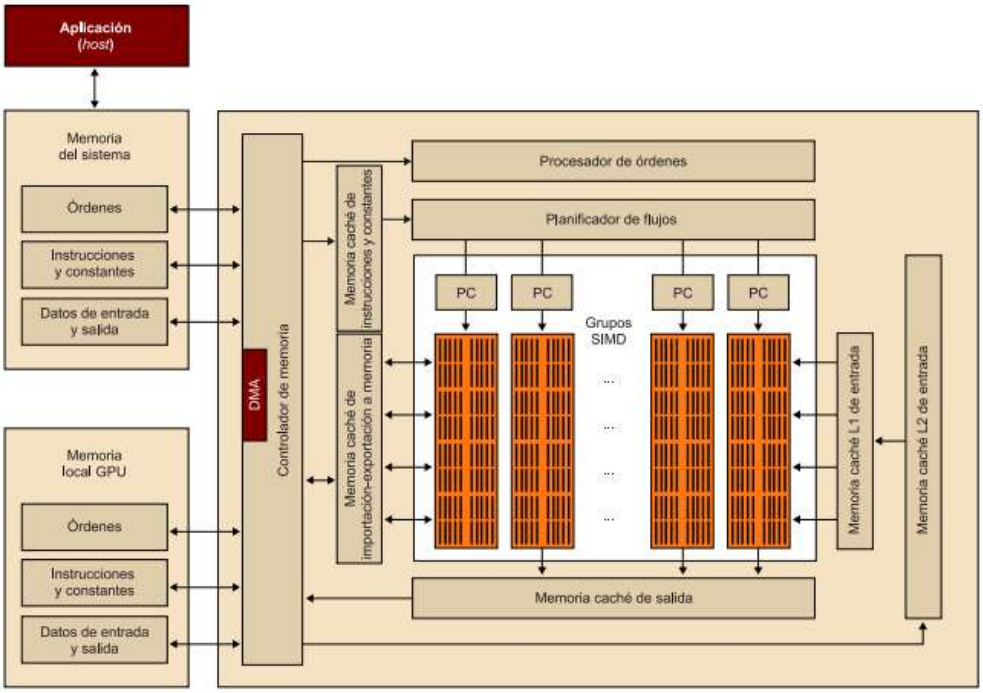

Ilustración 17 Diagrama de bloques de la arquitectura de la serie R600 de AMD

Fuente Guim, F

Los SP de la arquitectura R600 están organizados en grupos de cinco (grupos SIMD), de los cuáles uno puede ejecutar operaciones complejas tales como operaciones en coma flotante de 32 bits, mientras los otros cuatro se dedican solo a operaciones simples con enteros, está arquitectura SIMD híbrida también se la conoce como VLIW-5D. Al estar los SP organizados de esta manera, la arquitectura R600 solo puede ejecutar 64 flujos, cuando se podría pensar que soporta 320. Desde

otra perspectiva estos 64 flujos podrían ejecutar cinco instrucciones en cada ciclo reloj, pero hay que tener en cuenta que cada una de estas instrucciones tiene que ser completamente independientes de las demás. Así el planificador de flujos de esta arquitectura lleva la responsabilidad además de gestionar una cantidad muy grande de flujos. El planificador de flujos se encarga de seleccionar dónde hay que ejecutar cada flujo mediante una serie de árbitros, dos por cada matriz de dieciséis grupos SIMD, además determina la urgencia de ejecución en un flujo, esto significa que el planificador de flujos puede asignar a un flujo un grupo SIMD ocupado, mantener los datos que este grupo SIMD estaba utilizando y, una vez que el flujo con mayor prioridad ha finalizado, seguir ejecutando el flujo original con normalidad.

Una diferencia más de los SP de las arquitecturas Nvidia y AMD es la frecuencia de reloj en la que trabajan. Así los SP de Nvidia utilizan un dominio específico de reloj que funciona a una velocidad más elevada que el resto de elementos mientras que en AMD los SP utilizan la misma frecuencia de reloj que el resto de elementos y no tienen el concepto de dominios de reloj.

Modelo	Arquitectura (familia)	Núcleos		Memoria				Rendimiento - pico teórico (GFLOP)
		Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
R580	X1000	48	600	GDDR3	1.024	650	83	375
RV670	R600	320	800	GDDR3	2.048	800	51	512
RV770	R700	800	750	GDDR5	2.048	850	108	1.200
Cypress (RV870)	Evergreen	1.600	825	GDDR5	4.096	1.150	147	2.640

Ilustración 18 Comparativa de varios modelos de GPU AMD

Fuente Guim, F

2.5.- ¿Qué es la Programación Paralela?

La Programación Paralela es la manera de ejecutar varias instrucciones de una manera simultánea, basada en el principio de dividir en problemas pequeños uno grande y de esta manera permitir mejorar la velocidad en su resolución, mejorando así el rendimiento del computador.

La Programación Paralela nace con la necesidad de mejorar la capacidad de procesamiento de un computador al momento de ejecutar instrucciones, de esta manera se ha vuelto una herramienta muy importante en la tecnología de la época actual.

"Las tareas principales que enfrenta la programación paralela y distribuida consisten en comunicar y sincronizar un conjunto de procesos que se ejecutan concurrentemente en diferentes procesadores."(Nesmachnow, 2004)

"La aplicación anfitriona maneja sus dispositivos a través de un contenedor llamado contexto. Existe otro contenedor de kernels[10] (funciones) llamado programa. La aplicación dispara cada kernel hacia una estructura llamada fila de comandos. La lista de comandos es un mecanismo a través del cual la aplicación principal les indica a los dispositivos disponibles que kernel va a ejecutar."(Tello & Huesca, 2012)

El procesamiento paralelo cada vez toma más fuerza en la actualidad, el hecho de necesitar una mayor velocidad de procesamiento ha hecho que el procesamiento secuencial se vaya convirtiendo en innecesario. Al usar el paralelismo nos encontramos con algunos hechos: primeramente encontramos la necesidad de tener una mayor potencia de cálculo que independientemente de los procesadores que se tenga todo depende de la tecnología del momento, estás van de la mano de paradigmas basados en cálculo paralelo, al final la idea sería tener un n número de computadores mejorando así la velocidad de cálculo y que el problema pueda resolverse en 1/n veces del tiempo requerido por el sistema secuencial; como

[10] **Kernels:** Un kernel es una función especial diseñada para ser ejecutada en uno o más dispositivos

segundo hecho tenemos el mejorar la relación costo/rendimiento ya que al realizar una gran cantidad de cálculos en el computador puede generar una alta explosión de costos, para esto es conveniente involucrar elementos de bajo poder de cálculo, como último hecho se puede apreciar una potencia expresiva de los modelos de procesamiento paralelo, existen muchos problemas que son más fáciles de modelar gracias a paradigmas paralelos, ya sea por usar una estructura paralela o por que el problema sea intrínsecamente paralelo, al plantear inicialmente un mecanismo paralelo para resolver un problema, se puede facilitar la implantación de un modelo computacional, al final se podría tener mejores soluciones para el problema en un menor tiempo de implementación. El mayor problema a enfrentarse con la programación paralela es que no son ideas fáciles de entender o implementar.

2.6.- Características de la Programación Paralela

Debido al rendimiento obtenido en los procesadores de un solo núcleo, se ve la necesidad de la construcción de arquitecturas paralelas, pero a pesar de ello no existe todavía un aprovechamiento total de estas, por lo que nace la programación paralela con las siguientes características:

- ✓ Realiza la ejecución de varias instrucciones de manera simultánea.
- ✓ La programación paralela emplea elementos de procesamiento múltiple simultáneamente para resolver un problema. Esto se logra dividiendo el problema en partes independientes de tal manera que cada elemento de procesamiento pueda ejecutar su parte del algoritmo a la misma vez que los demás.
- ✓ Los elementos que van a ser procesados pueden ser diversos e incluir recursos tales como un único ordenador con muchos procesadores, varios ordenadores en red, hardware especializado o una combinación de los anteriores.

Además de las características presentadas anteriormente, al momento de construir un sistema paralelo se debe tomar en cuenta los siguientes aspectos:

- ✓ Los cálculos deben realizarse en periodos de ejecución razonables.
- ✓ Se requiere hardware rápido y software eficiente.
- ✓ Se debe utilizar lenguajes familiares o variaciones fáciles de entender.
- ✓ La tasa de cálculo no debe variar al momento entre un computador u otro.
- ✓ El código debe ser fácil de mover entre máquinas.

2.7.- ¿Por qué paralelizar?

Hoy en día la tecnología ha tenido un gran avance, y con la introducción de la programación paralela se puede llegar a un gran aprovechamiento de los recursos teniendo en cuenta los siguientes factores:

- ✓ Los programas de ordenador paralelos reducen la complejidad de procesamiento de las arquitecturas paralelas.
- ✓ El incremento de velocidad que consigue un programa como resultado de la paralelización viene dado por La Ley de Amdahl[11] que dice que "la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente".
- ✓ Los programas en paralelo ayudan al estudio de algún fenómeno de una forma más rápida mediante un modelo computacional.
- ✓ A mediados de 1990, lo fabricantes de supercomputadoras determinaron que era más fácil y barato construir máquinas con muchos procesadores en vez de construir procesadores más potentes.
- ✓ A mediados del año 2000, los fabricantes de procesadores determinaron que es más eficiente usar procesadores multicore que construir procesadores más rápidos.

[11] **La Ley de Amdahl**, llamada así por el arquitecto de ordenadores Gene Amdahl, se usa para averiguar la mejora máxima de un sistema cuando solo una parte de éste es mejorado.

2.8.- Áreas de aplicación de programación paralela

Existen algunos campos en donde es necesario un mayor poder de cálculo, en el cuál un computador simple no sería el correcto para obtener el resultado deseado, entre ellos tenemos:

- ✓ Procesamiento de imágenes: existe una gran cantidad de datos envueltos en el procesamiento, y exige una mayor velocidad para procesar las imágenes, lo que hace que sea necesario usar estructuras de datos y operaciones paralelas.
- ✓ Modelo matemático: para la resolución de ecuaciones diferenciales parciales o elementos finitos se necesita grandes poderes de cálculo.
- ✓ Computación Inteligente: Si se quiere emular la estructura del cerebro se necesita el procesamiento paralelo, como por ejemplo una interacción de neuronas van describiendo la dinámica del sistema aprovechando el paralelismo y haciendo más eficiente el tiempo de ejecución
- ✓ Manipulación de Base de Datos: la ventaja del paralelismo es reducir tiempos de ejecución con la utilización de estructuras de datos adecuadas y operaciones transaccionales en los elementos de las bases de datos.
- ✓ Predicción del tiempo: requiere un alto poder de cálculo y una mayor rapidez en su ejecución.

CAPÍTULO III

PARADIGMAS DE PROGRAMACIÓN PARALELA

3. Paradigmas de Programación Paralela

3.1 Programación Paralela por Manejo de Threads

3.1.1.- Manejo de Threads

Las nuevas arquitecturas de los procesadores necesitan explícitamente aplicaciones paralelas que aprovechen los recursos asignados, a esto se le llama paralelismo a nivel de hilo, ya que los flujos de control necesarios se los conoce como hilos.

"El lanzamiento de hilos consume algo de tiempo del procesador, por lo que es importante que los hilos tengan una buena cantidad de trabajo para hacer que esta carga sea insignificante en comparación con el trabajo que se realiza por el hilo. Si la cantidad de trabajo que hace un hilo es baja, la sobrecarga de hilos puede dominar la aplicación. Esta sobrecarga es causada usualmente por tener demasiado fino una granularidad de trabajo. Esto se puede solucionar mediante el aumento de la cantidad de trabajo que cada hilo hace".(Blair-Chappell & Stokes, 2012)

3.1.2.- OpenCL

OpenCL[12] es un lenguaje de programación desarrollado por Khronos Group para trabajar sobre GPU's, este es de código abierto y se lo puede utilizar en tarjetas Nvidia o ATI mientras se dispongan de todos los drivers.

"OpenCL es un marco estándar de la industria para los equipos de programación compuesto por una combinación de CPU, GPU, y otros procesadores. Estos llamados sistemas heterogéneos se han convertido en una clase importante de plataformas, y OpenCL es el primer estándar de la industria que se dirige directamente a sus necesidades. Lanzado por primera vez en diciembre de 2008 con los primeros productos disponibles en el otoño de 2009, OpenCL es una tecnología relativamente nueva."(Munshi, Gaster, Mattson, & Ginsburg, 2011)

"OpenCL ™ es el primer estándar sin derechos de autor abierto para multiplataforma, la programación en paralelo de los procesadores modernos se encuentran en los ordenadores personales, servidores y dispositivos de mano. OpenCL (Open Computing Language) mejora en gran medida la velocidad y capacidad de

[12] **OpenCL:** Lenguaje de Computación Abierto con sus siglas en inglés.

respuesta para una amplia gama de aplicaciones en numerosas categorías de mercado de los juegos y el entretenimiento a software científico y médico.

OpenCL 2.1 es una evolución significativa de esta norma ampliamente utilizada que ofrece la funcionalidad demandada por la comunidad de desarrolladores - preservando al mismo tiempo la compatibilidad con otros dispositivos y ser capaz de ser usado en cualquier hardware compatible con OpenCL 2.0.

- ✓ *Nuevo lenguaje C++, con un kernel basado en C ++ 14 para mejorar significativamente la productividad del programador, la portabilidad y el rendimiento con código reutilizable limpio.*
- ✓ *Soporte para el nuevo lenguaje Khronos[13] SPIR-V ™ [14]que ayuda como intermediario en el núcleo dando flexibilidad al lenguaje de kernel, compartiendo así los elementos de entrada y salida de un compilador común, y la capacidad para enviarlos al kernel sin exponer el código fuente.*
- ✓ *Mejoras en la API OpenCL incluyendo subgrupos, que exponen el enhebrado del hardware en el núcleo, junto con las operaciones de consulta de subgrupos adicionales para mayor flexibilidad"(OpenCL)*

[13] Khronos: Es una industria sin fines de lucro que elabora estándares abiertos para la creación y aceleración de la computación paralela.
[14] SPIR-V: Es un lenguaje intermedio que ayuda a la representación del cómputo paralelo y gráficos. Es el primer estándar abierto.

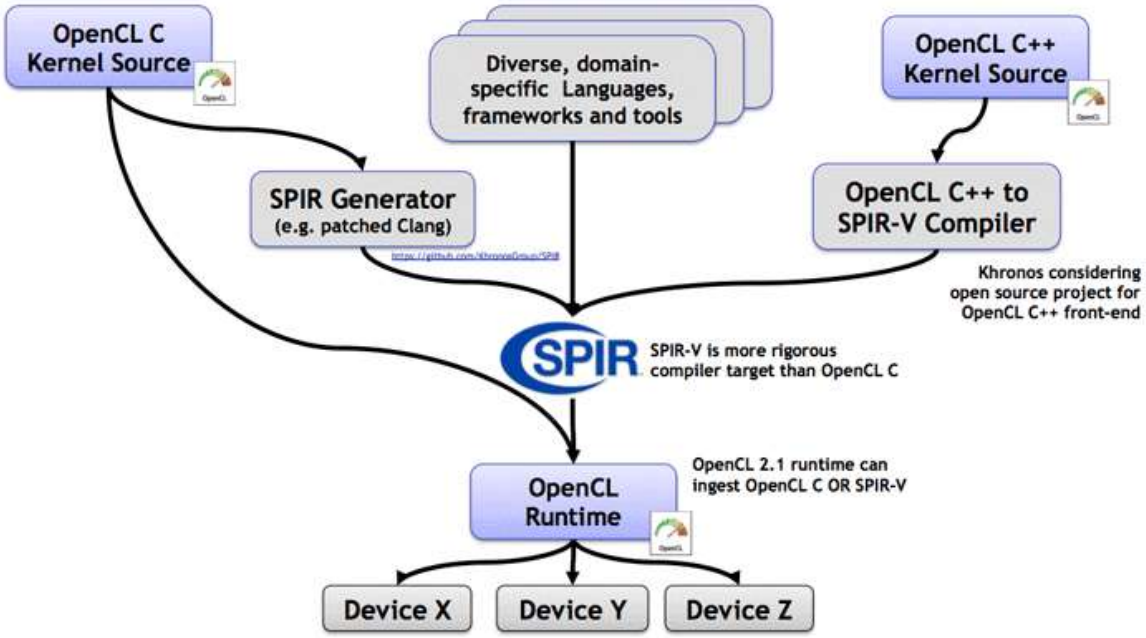

Ilustración 19 Ecosistema de Compilación de OpenCL 2.1

Fuente: https://www.khronos.org/opencl/

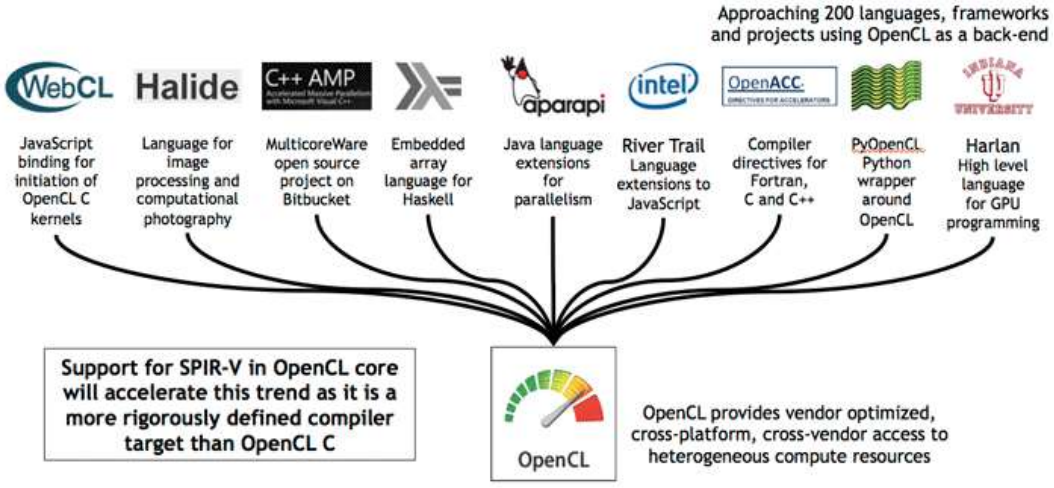

Ilustración 20 OpenCL como lenguaje de programación paralelo

Fuente: https://www.khronos.org/opencl/

3.1.3.- Compatibilidad de OpenCL

- ✓ Intel
- ✓ MediaTek Inc
- ✓ QUALCOMM
- ✓ Texas Instruments
- ✓ AMD
- ✓ Altera Corporation
- ✓ Vivante Corporation
- ✓ Xilinx, Inc.
- ✓ ARM Limited
- ✓ Imagination Technologies
- ✓ Apple, Inc.
- ✓ STMicroelectronics International NV
- ✓ ARM
- ✓ IBM Corporation
- ✓ Creative Labs
- ✓ NVIDIA
- ✓ Samsung Electronics

3.1.4.- Programación en OpenCL

La especificación de OpenCL define sus funciones, estructuras de datos y también las características necesarias para el desarrollo de herramientas para diferentes dispositivos, así mismo define los aspectos que hacen compatible al dispositivo con el framework.

3.1.4.1.- Tipos de Datos en OpenCL

Tipos de Datos Escalares

Tipo	Descripción
bool	Es un tipo de dato condicional, puede ser verdadero o falso
Char	Entero con signo de 8 bits
unsigned char, uchar	Entero sin signo de 8 bits
Short	Entero con signo de 16 bits
unsigned short, ushort	Entero sin signo de 16 bits
Int	Entero con signo de 32 bits.
unsigned int, uint	Entero sin signo de32 bits.
Long	Entero con signo de 64 bits
unsigned long, ulong	Entero sin signo de 64 bits
Float	Dato decimal de 32 bits.
Double	Dato decimal de 64 bits
Half	Dato decimal de 16 bits
size_t	Entero sin signo del tamaño del espacio de dirección puede ser 32 o 64 bits
ptrdiff_t	Entero con signo del tamaño del espacio de dirección puede ser 32 o 64 bits
Void	Constituye un conjunto de valores vacíos

Tabla 1 Tipos de Datos Escalares en OpenCL

Datos Tipo Vector

Tipo	Descripción
Charn	Vector de n enteros con signo de 8 bits
Ucharn	Vector de n enteros sin signo de 8 bits
Shortn	Vector de n enteros con signo de 16 bits
ushortn	Vector de n enteros sin signo de 16 bits
Intn	Vector de n enteros con signo de 32 bits
Uintn	Vector de n enteros sin signo de 32 bits
Longn	Vector de n enteros con signo de 64 bits
Ulongn	Vector de n enteros sin signo de 64 bits
Floatn	Vector de n decimales de 32 bits
doublen	Vector de n decimales de 64 bits
Halfn	Vector de n decimales de 16 bits

Tabla 2 Datos Tipo Vector en OpenCL

Tipos de Datos de Aplicación

Tipo de Dato en OpenCL	Tipo API para Aplicación
Char	cl_char
Uchar	cl_uchar
Short	cl_short
ushort	cl_ushort
Int	cl_int
Uint	cl_uint
Long	cl_long
ulong	cl_ulong
Float	cl_float
double	cl_double
Half	cl_half
charn	cl_charn
ucharn	cl_ucharn
shortn	cl_shortn
ushortn	cl_ushortn
Intn	cl_intn
uintn	cl_uintn
longn	cl_longn
ulongn	cl_ulongn
floatn	cl_floatn
doublen	cl_doublen
halfn	cl_halfn

Tabla 3 Datos de Aplicación en OpenCL

3.1.4.2.- Programación básica en OpenCL

Dependiendo sobre el proveedor que se vaya a programar se debe tomar en cuenta el SDK propietario, por ejemplo Nvidia y AMD cuentan con sus propios SDK, por lo que es necesario definir desde un inicio la plataforma en que se va a trabajar; además de la plataforma también se debe conocer los dispositicos que se encuentran disponible para la misma.

A continuación se presenta el código de algunas funciones básicas:

Instrucción for: cuando se tiene una gran estructura de datos, lo principal a hacer es iterarlos para poder ejecutar diversas funciones sobre esos datos, por lo que se usa ciclos anidados:

Código Secuencial

```
for(i=0; i<x ; i++){
        for(j=0; j<y; j++){
                for(k=0; k<z; k++){
                        procesar(arr[i][j][k]);
                }
        }
}
```

Código OpenCL

```
int i = get_global_id(0);
int j = get_global_id(1);
 int k = get_global_id(2);
 procesar(arr[i][j][k]);
```

Cómputo de π: el cálculo de Pi es muy común, a continuación se presenta el código escrito en C y en OpenCL, aunque este último nos dará una mayor velocidad de cálculo cabe recalcar que OpenCL no es muy fácil de escribir y muchas veces tampoco leer.

Código en C

```
long num_steps = 100000000000;
double step = 1.0/num_steps;
double x, pi, sum = 0.0;
for(long i = 0; i<num_steps; i++){
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x*x);
}
pi = sum * step;
```

Código en OpenCL

```
#define _num_steps 100000000000
#define _divisor 40000
#define _step 1.0/_num_steps
#define _intrnCnt _num_steps / _divisor
__kernel void pi( __global float *out )
{
        int i = get_global_id(0);
        float partsum = 0.0;
        float x = 0.0;
        long from = i * _intrnCnt;
        long to = from + _intrnCnt;
        for(long j = from; j<to; j++)
        {
                x = ( j + 0.5 ) * _step;
                partsum += 4.0 / ( 1. + x * x);
        }
        out[i] = partsum;
}
```

3.1.4.3.- Hello World en OpenCL

Hello World - kernel (hello.cl)

```
__kernel void hello(__global char* string)
{
string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = ',';
string[6] = ' ';
string[7] = 'W';
string[8] = 'o';
string[9] = 'r';
string[10] = 'l';
string[11] = 'd';
string[12] = '!';
string[13] = '\0';
}
```

Hello World - host (hello.c)

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
```

```
#include <CL/cl.h>
#endif

#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_mem memobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;
cl_platform_id platform_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;

char string[MEM_SIZE];

FILE *fp;
char fileName[] = "./hello.cl";
char *source_str;
size_t source_size;

/* Cargar el código fuente que contiene el kernel*/
fp = fopen(fileName, "r");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

/* Obtener la plataforma e información del dispositivo */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
&device_id, &ret_num_devices);

/* Crear el contexto OpenCL */
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

/* Crear el Comando Queue */
command_queue = clCreateCommandQueue(context, device_id, 0,
&ret);

/* Crear el Buffer de Memoria */
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE *
sizeof(char), NULL, &ret);
```

```
45. /* Crear el Programa Kernel de la fuente */
46. program = clCreateProgramWithSource(context, 1, (const char
    **)&source_str,
47. (const size_t *)&source_size, &ret);

48. /* Construir el Programa  Kernel */
49. ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

50. /* Crear OpenCL Kernel */
51. kernel = clCreateKernel(program, "hello", &ret);

52. /* Establecer los parámetros de OpenCL Kernel */
53. ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

54. /* Ejecutar OpenCL Kernel */
55. ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);

56. /* Copiar los resultados del buffer de memoria */
57. ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
58. MEM_SIZE * sizeof(char),string, 0, NULL, NULL);

59. /* Mostrar Resultados */
60. puts(string);

61. /* Finalizar */
62. ret = clFlush(command_queue);
63. ret = clFinish(command_queue);
64. ret = clReleaseKernel(kernel);
65. ret = clReleaseProgram(program);
66. ret = clReleaseMemObject(memobj);
67. ret = clReleaseCommandQueue(command_queue);
68. ret = clReleaseContext(context);

69. free(source_str);

70. return 0;
71. }
```

3.2.- Programación Paralela por Paso de Mensajes

3.2.1.- Paso de Mensajes

Este es el modelo más usado, los programadores realizan las tareas mediante el envío de mensajes, en donde las direcciones tienen un espacio ya distribuido. Para el paso de mensajes se puede usar la red o una memoria distribuida, dependiendo del modelo de arquitectura que se tenga.

"Las comunicaciones entre dos tareas ocurren a dos bandas, donde los participantes tienen que invocar una operación. Podemos denominar a estas comunicaciones como operaciones cooperativas ya que deben ser realizadas por cada proceso, el que tiene los datos y el proceso que quiere acceder a los datos".(Jorba i Esteve, Margalef, & Morajko, 2006)

En las CPU modernas el modelo por paso de mensajes, deja que el programador tenga el control sobre la localidad de los datos.

3.2.2.- MPI

Para la adaptación de paso de mensajes se utilizará el lenguaje C++ en conjunto con la librería de mpi, que incluye la realización del tránsito de operaciones sobre datos y la sincronización de tareas.

"MPI especifica la funcionalidad de las rutinas de comunicación de alto nivel

Las funciones de MPI brindan acceso a una aplicación de bajo nivel que toma cuidado de los sockets, buffering, copia de datos, enrutamiento de mensajes, etc."(Barker, 2015)

Para realizar un programa en MPI tenemos los siguientes recursos básicos:

- ✓ Incluir en la cabecera del programa la librería específica
  #include <mpi.h> inserta los tipos y definiciones básicas
- ✓ Inicialización de comunicaciones
  MPI_Init inicializa el ambiente de MPI
  MPI_Comm_size retorna el número de procesos
  MPI_Comm_rank Devuelve el número de un proceso específico
- ✓ Comunicación para compartir los datos entre procesos
  MPI_Send envía un mensaje
  MPI_Recv recibe un mensaje
- ✓ Salir del sistema de paso de mensajes
  MPI_Finalize

3.2.3.- Tipos de datos en C++

Tipo de Dato	Descripción	Número de bytes típico	Rango
short	Entero corto	2	-32768 a 32767
int	Entero	4	-2147483648 a +2147483647
long	Entero largo	4	-2147483648 a +2147483647
char	Carácter	1	-128 a 127
signed short	Entero corto	2	-32768 a 32767
unsigned short	Entero corto sin signo	2	0 a 65535
signed int	Entero	4	-2147483648 a +2147483647
unsigned int	Entero sin signo	4	0 a 4294967295
signed long	Entero largo	4	-2147483648 a +2147483647
unsigned long	Entero largo sin signo	4	0 a 4294967295
signed char	Carácter	1	-128 a 127
unsigned char	Carácter sin signo	1	0 a 255
float	Real (Número en coma flotante)	4	Positivos: 3.4E-38 a 3.4E38 Negativos: -3.4E-38 a -3.4E38
double	Real doble(Número en coma flotante de doble precisión)	8	Positivos: 1.7E-308 a 1.7E308 Negativos: -1.7E-308 a -1.7E308
long double	Real doble largo	10	Positivos: 3.4E-4932 a 1.1E4932 Negativos: -3.4E-4932 a -1.1E4932
bool	Dato de tipo lógico	1	0, 1
wchar_t	Carácter Unicode	2	0 a 65535

Tabla 4 Tipos de Datos en C++

3.2.4.- Hello World MPI

```
#include <stdio.h>
#include "mpi.h"
#include <string.h>
main(int argc, char **argv)
{
      char message[20];
      int i, rank, size, tag = 99;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == 0) {
         strcpy(message, "Hello, world!");
         for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
         MPI_Recv(message,  20,  MPI_CHAR,  0,  tag,  MPI_COMM_WORLD,
         &status);
      }
      printf("Message from process %d : %.13s\n", rank, message);
      MPI_Finalize();
}
```

3.2.5.- Descripción del código Hello World

Inicio y Cierre del ambiente de MPI

```
#include <stdio.h>
#include "mpi.h"
#include <string.h>
main(int argc, char **argv)
{
      char message[20];
      int i, rank, size, tag = 99;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == 0) {
         strcpy(message, "Hello, world!");
         for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
         MPI_Recv(message,  20,  MPI_CHAR,  0,  tag,  MPI_COMM_WORLD,
         &status);
      }
      printf("Message from process %d : %.13s\n", rank, message);
      MPI_Finalize();
}
```

Inicializa el ambiente de MPI

Cierra el ambiente de MPI

Manejo de Procesos

```
#include <stdio.h>
#include "mpi.h"
#include <string.h>
main(int argc, char **argv)
{
      char message[20];
      int i, rank, size, tag = 99;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == 0) {
         strcpy(message, "Hello, world!");
         for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
         MPI_Recv(message,  20,  MPI_CHAR,  0,  tag,  MPI_COMM_WORLD,
         &status);
      }
      printf("Message from process %d : %.13s\n", rank, message);
      MPI_Finalize();
}
```

Retorna el número de procesos

Devuelve el número de un proceso específico

Uso de mensajes

```
#include <stdio.h>
#include "mpi.h"
#include <string.h>
main(int argc, char **argv)
```

```
{
      char message[20];
      int i, rank, size, tag = 99;
      MPI_Status status;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == 0) {
         strcpy(message, "Hello, world!");
         for (i = 1; i < size; i++)
         MPI_Send(message, 13,  MPI_CHAR, i, tag, MPI_COMM_WORLD);
      } else {
         MPI_Recv  (message,  20,  MPI_CHAR,  0,  tag,  MPI_COMM_WORLD,
         &status);
      }
      printf("Message from process %d : %.13s\n", rank, message);
      MPI_Finalize();
}
```

Envía un mensaje

Recibe un mensaje

3.2.6.- Cálculo de PI con MPI

El siguiente código utiliza Parallel Patterns Library (PPL), en lugar de OpenMP, y usa las operaciones colectivas de MPI en lugar de operaciones punto a punto.

```
// ParallelPI.cpp : Define el punto de entrada para la aplicación
MPI.
//
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "limits.h"
#include <ppl.h>
#include <random>
```

```
#include <time.h>

using namespace Concurrency;

int ThrowDarts(int iterations)
{

combinable<int> count;

int result = 0;


parallel_for(0, iterations, [&](int i){

std::tr1::uniform_real<double> MyRandom;
double RandMax = MyRandom.max();
std::tr1::minstd_rand0 MyEngine;
double x, y;

MyEngine.seed((unsigned int)time(NULL));

x = MyRandom(MyEngine)/RandMax;
y = MyRandom(MyEngine)/RandMax;

if(x*x + y*y < 1.0)
{
count.local() += 1;
}
});

result = count.combine([](int left, int right) { return left +
right; });

return result;
}

void main(int argc, char* argv[])
{
int rank;
int size;
int iterations;
int count;
int result;
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

if(rank == 0)
{
//Rango 0 lee el número de comandos en la línea de iteraciones
//50M iteraciones es el número por defecto.
iterations = 50000000;
if(argc > 1)
{
iterations = atoi(argv[argc-1]);
}
printf("Ejecutando %d iteraciones en %d nodos.\n", iterations,
size);
fflush(stdout);
}
//Emite el número de iteraciones a ejecutar.
MPI_Bcast(&iterations, 1, MPI_INT, 0, MPI_COMM_WORLD);

count = ThrowDarts(iterations);

//Reúne y suma los resultados
MPI_Reduce(&count, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if(rank == 0)
{
printf("El valor de PI es aproximado a: %16f",
4*((double)result/(double)(iterations*size)));
}

MPI_Barrier(MPI_COMM_WORLD);

MPI_Finalize();

}
```

3.3.- Programación Paralela Híbrida

En la programación híbrida nos encontramos el uso de la memoria compartida junto al paso de mensajes, en este caso los datos se encuentra en una memoria global de fácil acceso para los procesadores paralelos, así de esta forma cualquier procesador puede buscar y almacenar datos en cualquier lugar de la memoria de una manera independiente, preservando la integridad de las estructuras de datos compartidas; por otro lado el paso de mensajes se asocia a un proceso en particular, y de esa manera al acceder a un dato remoto necesitará la comunicación por mensajes, siendo las encargadas de la sincronización las operaciones de envío y recepción.

"*La arquitectura de cluster de multicores permite introducir un nuevo modelo de programación paralela como es la programación híbrida… La comunicación entre procesos que pertenecen al mismo procesador físico puede realizarse utilizando memoria compartida (nivel micro), mientras que la comunicación entre procesadores físicos (nivel macro) se suele realizar por medio del pasaje de mensajes*".(Leibovich et al., 2011)

3.3.1.- CUDA

CUDA[15] es una arquitectura de programación paralela desarrollada por NVIDIA.

"CUDA es el motor de computación en las unidades de procesamiento gráfico o GPU de NVIDIA, ésta es accesible para los desarrolladores de software a través de los lenguajes de programación estándar de la industria. La arquitectura de CUDA soporta una gama de interfaces computacionales incluyendo OpenGL y Direct Compute. El modelo de programación paralela de CUDA está diseñado para superar este reto, manteniendo una curva de aprendizaje mínima para los programadores familiarizados con los lenguajes de programación estándar como C. En su esencia son tres abstracciones clave - una jerarquía de grupos de hilos, memoria compartida y sincronización de barrera [16]- que son simplemente expuestas al programador como un conjunto mínimo de extensiones del lenguaje."(Yang, Huang, & Lin, 2011)

[15] **CUDA**: Compute Unified Device Architecture

[16] **Sincronización de barrera:** significa que todos los que implementen esta barrera deberán parar en ese punto sin poder ejecutar las siguientes líneas de código hasta que todos los restantes hilos/procesos hayan alcanzado esta barrera.

3.3.2.- Programación en CUDA

CUDA propone un modelo de programación SIMD[17] con funcionalidades de procesamiento de vector. La programación en GPU se la puede realizar con una extensión de C/C++ con constructores y palabras clave, está extensión incluye la organización de trabajo paralelo a través de threads y la jerarquía de memoria de la GPU.

"Los threads en el modelo CUDA son agrupados en Bloques, los cuales se caracterizan por:

- ✓ *El tamaño del bloque: cantidad de threads que lo componen. Es determinado por el programador.*
- ✓ *Todos los threads de un bloque se ejecutan sobre el mismo SM[18].*
- ✓ *Los threads de un bloque comparten la memoria, la cual pueden usar como medio de comunicación entre ellos."*(Piccoli, 2011)

3.3.2.1.- Bloques de Threads

Cada hilo se identifica con un índice ThreadIdx, este puede ser unidimensional, bidimensional o tridimensional dentro de un bloque de threads. Las características que presenta el bloque de threads son:

- ✓ Tiene una unidad de asignación de threads a procesadores.
- ✓ Cada multiprocesador puede tener asignados 8 bloques y cada bloque puede incluir hasta 512 hilos.
- ✓ Los threads de un bloque pueden comunicarse por memoria compartida.
- ✓ La variable predefinida blockDim (de tipo dim3) permite acceder a las dimensiones del bloque dentro de un kernel.

3.3.2.2.- Modelo de la Memoria en CUDA

CUDA proporciona un direccionamiento de memoria que permite leer y escribir en cualquier localidad de la memoria DRAM.

[17] **SIMD**: Computadoras con un único flujo de instrucciones y múltiple flujo de datos
[18] **SM:** Multiprocesadores de Streaming con sus siglas en inglés

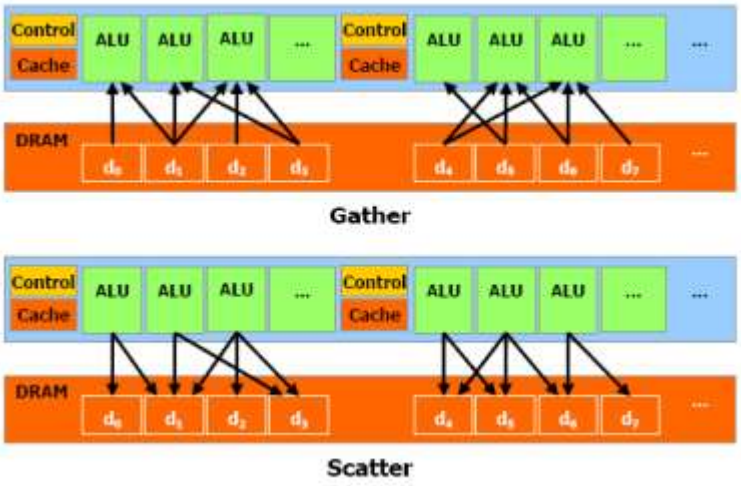

Ilustración 21 Repartición de memoria en CUDA

CUDA ofrece una memoria compartida que hace más rápido el acceso de lectura y escritura, permitiendo que varios threads compartan datos entre sí, minimizando los accesos a la memoria DRAM.

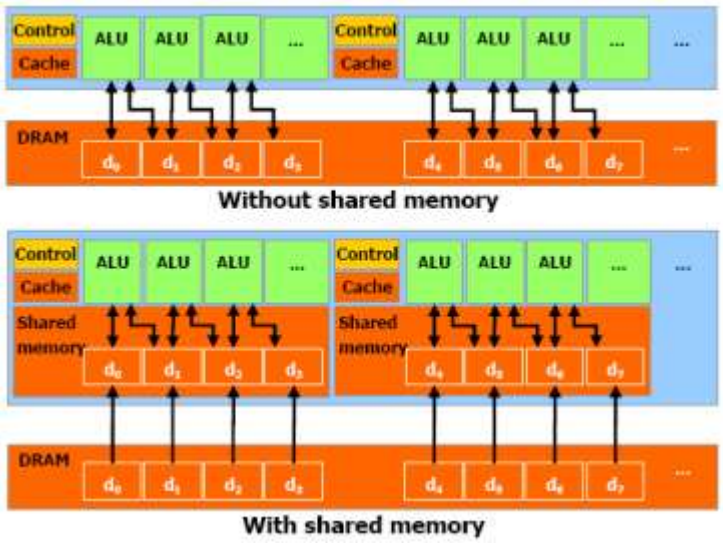

Ilustración 22 División de memoria en CUDA

3.3.2.3.- Instrucciones para el Manejo de Memoria

Crear memoria

- ✓ cudaMalloc ((void**) devPtr, size_t size)
- ✓ cudaMallocHost ((void**) hostPtr, size_t size)
- ✓ cudaFree (void *devPtr)
- ✓ cudaFreeHost (void *hostPtr)

Copiar memoria

- ✓ cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
- ✓ cudaMemcpy2D(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)
- ✓ cudaMemcpyToSymbol(const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind) H-->D D-->D
- ✓ cudaMemcpyFromSymbol(void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind) D-->H D-->D

Calificadores de una función

- ✓ __device__
  - Se ejecuta en el dispositivo
  - Llamada solamente desde el dispositivo
- ✓ __global__
  - Se ejecuta en el dispositivo
  - Llamada solamente desde el host
- ✓ __host__
  - Se ejecuta en el host
  - Llamada solamente desde el host

Calificadores de una variable

- ✓ __device__
  - Reside en el espacio de la memoria global
  - Tiene el tiempo de vida de una aplicación
  - Es accesible a partir de todos los hilos dentro del grid, y a partir del host a través de la biblioteca en tiempo de ejecución
- ✓ __constant__ (Opcionalmente se utiliza junto con __device__)
  - Reside en el espacio de la memoria constante
  - Tiene el tiempo de vida de una aplicación
  - Es accesible a partir de todos los hilos dentro del grid, y a partir del host a través de la biblioteca en tiempo de ejecución
- ✓ __shared__ (Opcionalmente se utiliza junto con __device__)

- Reside en el espacio de memoria compartida de un bloque de hilos
- Tiene el tiempo de vida de un bloque
- Solamente accesible a partir de los hilos que están dentro del bloque

Llamada a una función

Una función, por ejemplo:

__global__ void NameFunc(float *parametro);

Debe ser llamada como sigue:

NameFunc <<< Dg, Db, Ns, St >>> (parametro);

- ✓ Dg: Es de tipo dim3 dimensión y tamaño del grid
- ✓ Db: Es de tipo dim3 dimensión y tamaño de cada bloque
- ✓ Ns: Es de tipo size_t número de bytes en memoria compartida
- ✓ St: Es de tipo cudaStream_t el cuál indica que stream va a utilizar la función kernel

(Ns y St son argumentos opcionales)

Variables definidas automáticamente

Todas las funciones __global__ y __device__ tienen acceso a las siguientes variables:

- ✓ gridDim es de tipo dim3, indica la dimensión del grid
- ✓ blockIdx es de tipo uint3, indica el índice del bloque dentro del grid
- ✓ blockDim es de tipo dim3, indica la dimensión del bloque
- ✓ threadIdx es de tipo uint3, indica el índice del hilo dentro del bloque

Tipos de Datos

char1, uchar1, char2, uchar2, char3, char3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, int4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, longlong1, longlong2, float1, float2, float3, float4, double1,double2

Funciones Matemáticas

- ✓ __NombreFuncion()
  - A nivel de hardware

- Mayor velocidad pero menor precisión
- Ejemplos: __sinf(x), __expf(x), __logf(x),…

✓ NombreFuncion()

- Menor velocidad pero mayor precisión
- Ejemplos: sinf(x), expf(x), logf(x),…

✓ -use_fast_math: Opción del compilador nvcc

3.3.3.- Hello World en CUDA

```
#include <stdio.h>
__global__ void childKernel()
{
 printf("Hello ");
}
__global__ void parentKernel()
{
// launch child
childKernel<<<1,1>>>();
if (cudaSuccess != cudaGetLastError()) {
return;
}
// wait for child to complete
if (cudaSuccess != cudaDeviceSynchronize()) {
return;
}
printf("World!\n");
}
int main(int argc, char *argv[]) {
// launch parent
parentKernel<<<1,1>>>();
```

```
 if (cudaSuccess != cudaGetLastError()) {
 return 1;
}
// wait for parent to complete
if (cudaSuccess != cudaDeviceSynchronize()) {
return 2;
}
return 0;
}
```

CAPÍTULO IV

BASES DE COMPARACIÓN

4.- Bases de Comparación

Después de haber realizado las respectivas investigaciones a los diferentes paradigmas de programación paralela que se han planteado anteriormente se realizará la comparación entre las diferentes características que presentaron cada uno.

4.1.- Arquitecturas Compatibles

ARQUITECTURAS/LENGUAJES	OPENCL	MPI	CUDA
Intel	X	X	
MediaTek Inc	X		
QUALCOMM	X		
Texas Instruments	X		
AMD	X	X	
Altera Corporation	X		
Vivante Corporation	X		
Xilinx, Inc.	X		
ARM Limited	X		
Imagination Technologies	X		
Apple, Inc.	X		
STMicroelectronics International NV	X		
ARM	X		
IBM Corporation	X		
Creative Labs	X		
NVIDIA	X	X	X
Samsung Electronics	X		

Tabla 5 Arquitecturas compatibles con los lenguajes de programación paralela

Fuente: Propia

Si bien OpenCL posee una gran cantidad de arquitecturas compatibles, eso no quiere decir que el código funcione de manera óptima en todas, en el caso de

MPI, este es un protocolo de comunicación que se emplea para programar sistemas paralelos, de esta manera permite que en modelos SPMD se ejecute el mismo programa en cada núcleo. A diferencia de la programación en CUDA u OpenCL que tienen acceso directamente a las unidades de procesamiento, MPI permite transmitir un lenguaje escrito en otro programa a cada núcleo de procesamiento.

Dependiendo de la arquitectura en que se trabaje, si se toma como ejemplo a NVIDIA, se puede decir que CUDA toma una gran ventaja sobre ella, pues fue diseñado directamente para trabajar sobre esta arquitectura, además de que junto a este lenguaje se puede conectar MPI para la transmisión de mensajes.

4.2.- Ventajas de los paradigmas de programación paralela

Programación paralela por manejo de threads/tareas

El paralelismo a nivel de tareas ofrece las siguientes ventajas:

- ✓ Un uso más eficaz y más escalable de los recursos del sistema.
- ✓ Un mayor control mediante programación del que se puede conseguir con un subproceso o un elemento de trabajo.
- ✓ Brinda a las empresas, instituciones y usuarios en general el beneficio de la velocidad.
- ✓ Ventaja competitiva, provee una mejora de los tiempos para la producción de nuevos productos y servicios.
- ✓ Colaboración y flexibilidad operacional.

Programación paralela por paso de mensajes

El paralelismo por paso de mensajes ofrece las siguientes ventajas:

- ✓ MPI está implementado para todas las máquinas de memoria distribuida.
- ✓ MPI es el modelo natural para máquinas de memoria distribuida
- ✓ MPI ofrece alto rendimiento para DSMS [19]

[19] **DSMS**: Data stream management system, es un programa informático para gestionar los flujos de datos continuos.

- ✓ MPI es útil en sistemas SMPS[20] que forman clusters
- ✓ MPI puede ser usado sobre máquinas de memoria compartida
- ✓ MPI es grande, ya que contiene 125 funciones las cuales dan al programador control fino sobre las comunicaciones
- ✓ MPI es pequeño, ya que los programas de paso de mensaje pueden ser escritos usando sólo seis funciones

Programación paralela híbrida

La programación paralela híbrida ofrece las siguientes ventajas:

Para mejorar la escalabilidad

- ✓ Cuando muchas tareas producen desbalanceo
- ✓ Aplicaciones que combinan paralelismo de grano grueso y fino
- ✓ Reducción del tiempo de desarrollo de código
- ✓ Cuando el número de procesos MPI es fijo
- ✓ En caso de mezcla de paralelismo funcional y de datos

4.3.- Comparación de paradigmas

El manejo de threads da una gran ventaja en la distribución de tareas, pero está no es aprovechada cuando las tareas son muy pequeñas, para que se disminuya el costo tiene que asignarse a cada hilo una tarea grande para que así aproveche los recursos que le han sido asignados.

MPI es el paradigma más utilizado, debido al uso de mensajes entre los recursos de las arquitecturas permite una mayor comunicación en menor costo.

La programación paralela híbrida brinda mayores facilidades para el programador al tener los recursos ya asignados en memoria distribuida y al incluir MPI realiza tareas en menos tiempo disminuyendo los costos.

[20] **SMPS:** Switching Mode Power Supply, son fuentes que pueden aumentar la frecuencia de corriente.

Creación de HelloWorld en los diferentes lenguajes

HelloWorld Secuencial

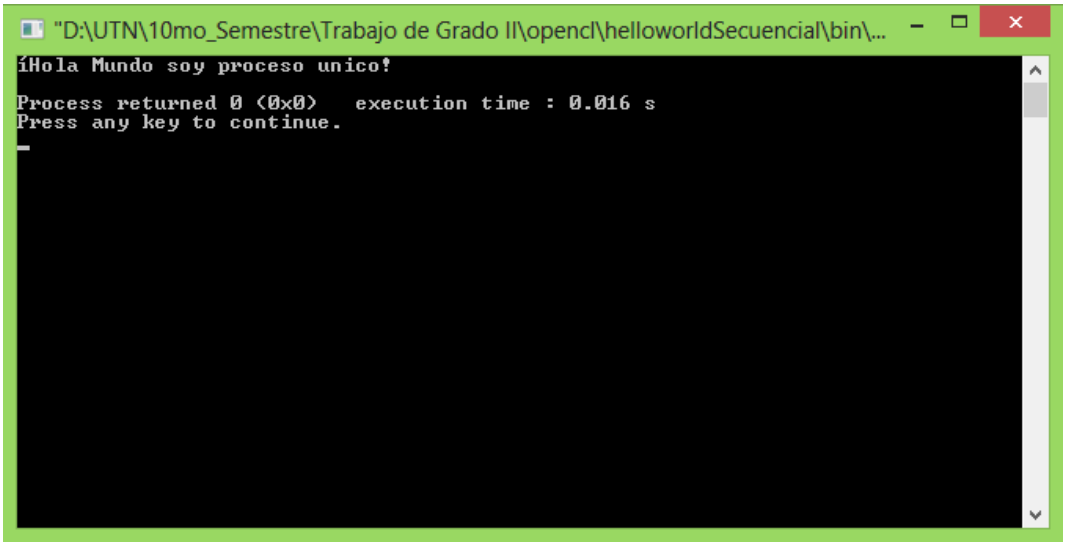

Ilustración 23 Ejecución Hello World de manera secuencial en c++

Fuente: Propia

HelloWorld con MPI

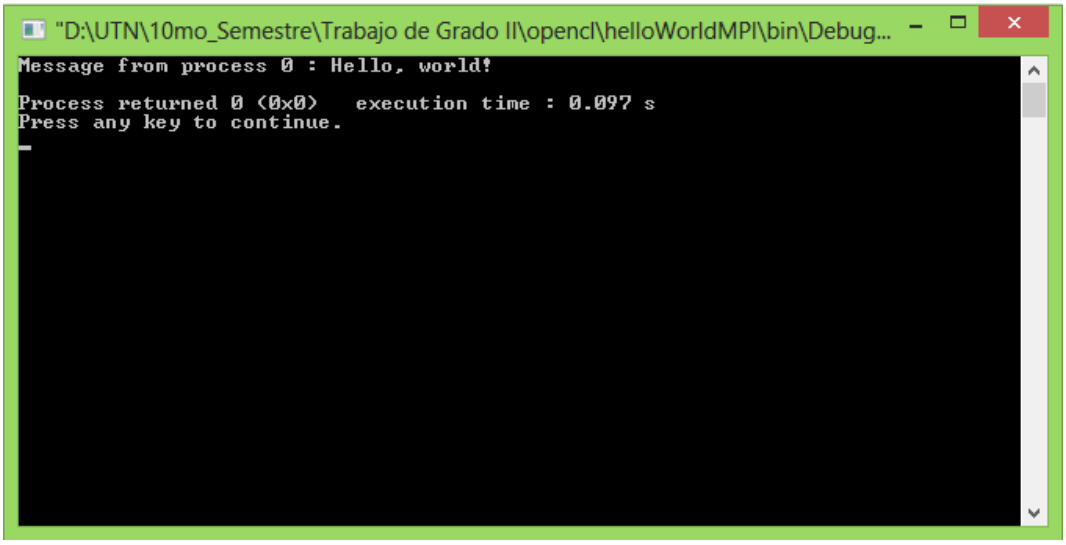

Ilustración 24 Ejecución Hello World con MPI

Fuente: Propia

HelloWorld OpenCL

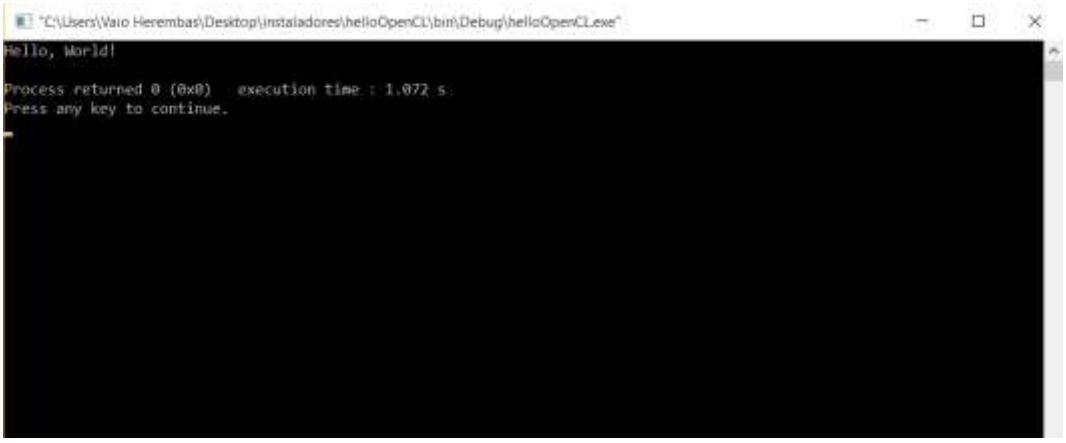

Ilustración 25 Ejecución Hello World con OpenCL

Fuente: Propia

HelloWorld CUDA

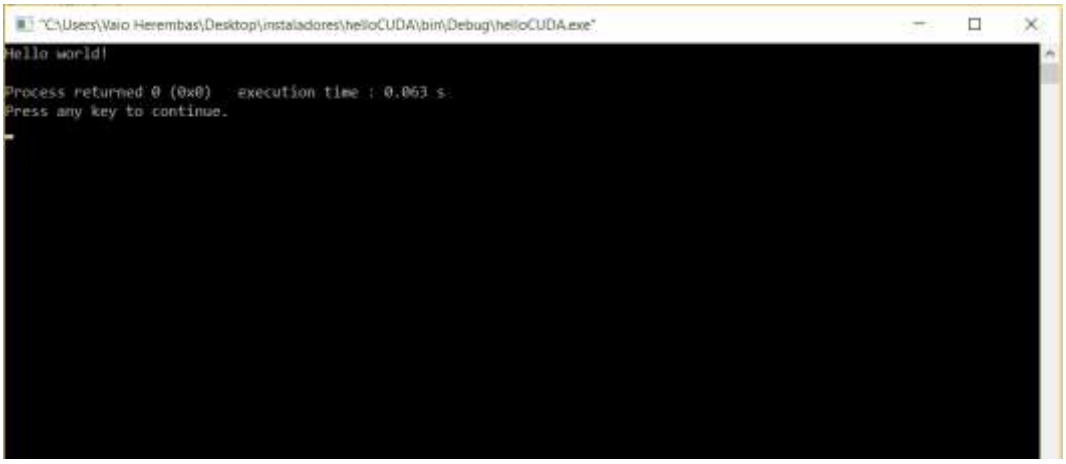

Ilustración 26 Ejecución Hello World con OpenCL

Fuente: Propia

Tiempos de ejecución

Código Secuencial	MPI	OpenCL	CUDA
0.016s	0.097s	1.072s	0.063s

Tabla 6 Comparación tiempos de ejecución Hello World

Fuente: Propia

Calculo de PI

Cálculo de PI Secuencial

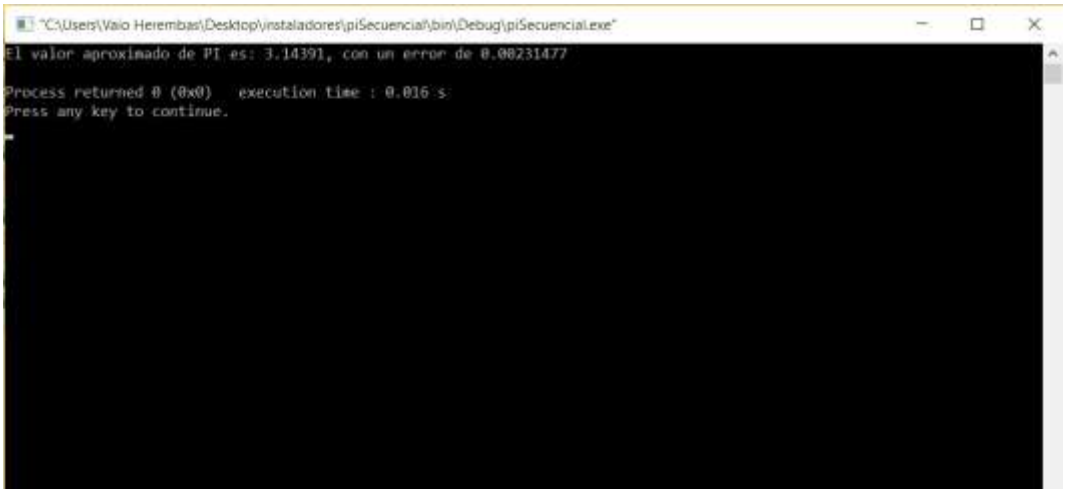

Ilustración 27 Ejecución código PI secuencial

Fuente: Propia

Cálculo de PI con MPI

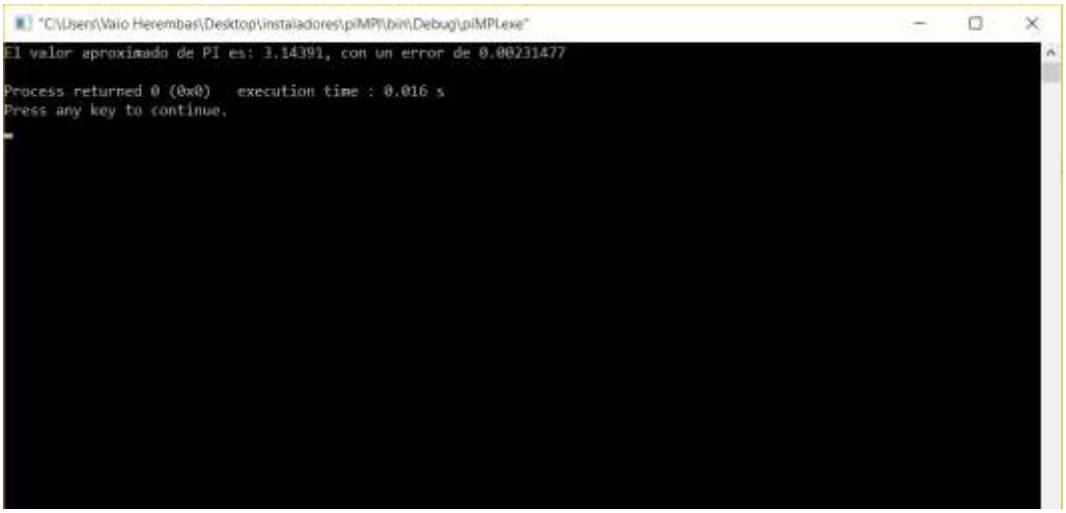

Ilustración 28 Ejecución código PI con MPI

Fuente: Propia

Cálculo de PI con OpenCL

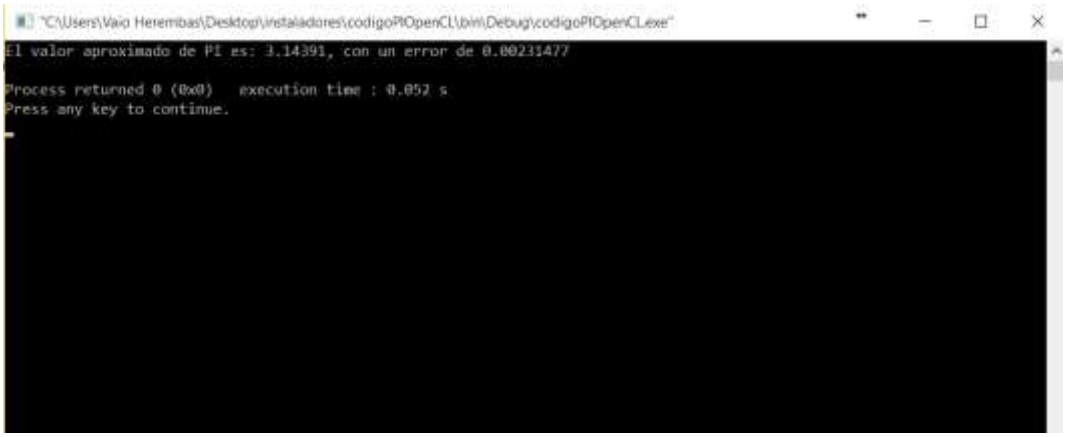

Ilustración 29 Ejecución código PI con OpenCL

Fuente: Propia

Cálculo de PI con CUDA

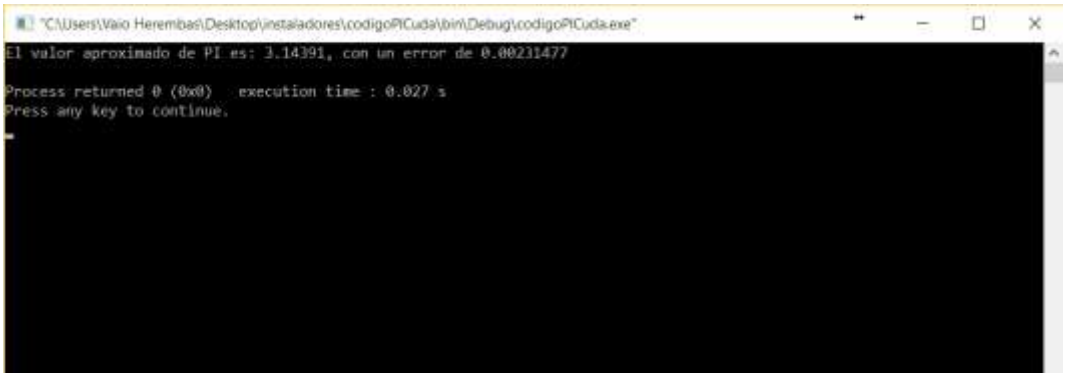

Ilustración 30 Ejecución código PI con CUDA

Fuente: Propia

Tiempos de ejecución

Código Secuencial	Código MPI	Código OpenCL	Código CUDA
0.016s	0.016s	0.052s	0.027s

Tabla 7 Comparación tiempos de ejecución código PI

Fuente: Propia

CAPÍTULO V

CONCLUSIONES Y RECOMENDACIONES

5.- Conclusiones y Recomendaciones

5.1.- Conclusiones

- ✓ La computación paralela ha ido tomando fuerza en el hardware y software actuales, tomando en cuenta que varios de los dispositivos que hoy en día se usa tengan más de un solo núcleo.
- ✓ Las GPU's tienen una gran acogida en el mercado financiero debido al crecimiento de videojuegos y aplicaciones gráficas.
- ✓ La programación paralela puede basarse en tareas o en la arquitectura de los computadores.
- ✓ OpenCL es un lenguaje que a simple vista resulta de difícil comprensión, pero cuando se logré aprender todas las ventajas que este tiene se tendrá muchas ventajas sobre el tema de programación paralela.
- ✓ El paso de mensajes ayuda a una mejor repartición de memoria entre los procesos y flujos de instrucción.
- ✓ MPI es una librería que nos ayuda en el aprovechamiento de los recursos del procesador, y brinda la conexión entre la GPU y CPU.
- ✓ CUDA es un lenguaje especial para GPU's Nvidia pero sus métodos han hecho que se límite la programación en otras arquitecturas.
- ✓ El paradigma de programación paralela híbrida es el más desarrollado en el campo de CPUs y GPUs, pues aprovecha al máximo los recursos de la arquitectura que se posee.

5.2.- Recomendaciones

- ✓ Revisar las características del hardware que se obtiene y de esta manera realizar un análisis correcto del lenguaje a utilizar.
- ✓ La programación paralela requiere de un gran tiempo de aprendizaje y concentración, por lo cual es necesario realizar varias investigaciones en el campo.
- ✓ Estudiar las diferentes GPU's que se tenga como alternativas para elegir el software adecuado para realizar la aplicación.
- ✓ Definir desde un inicio las estructuras paralelas a usar en la aplicación para que la solución al problema sea más efectiva.
- ✓ Estudiar correctamente las funciones que ofrece MPI para un mayor aprovechamiento de los recursos al momento de utilizarlo, ya sea independiente o en la programación híbrida.
- ✓ Analizarlas ventajas que ofrece OpenCL junto a MPI para poder realizar una programación paralela híbrida en arquitecturas que no sean Nvidia.
- ✓ Al programar en arquitecturas Nvidia aprovechar las herramientas y librerías que ofrece CUDA, ya que fue un software desarrollado especialmente para esta arquitectura.

BIBLIOGRAFÍA

[1] Barker, B. (2015). *Message passing interface (mpi).* Paper presented at the Workshop: High Performance Computing on Stampede.

[2] Blair-Chappell , S., & Stokes, A. (2012). *Parallel Programming with Intel Parallel Studio XE* Retrieved from http://utn.eblib.com/patron/FullRecord.aspx?p=817946

[3] Flynn, M. J., & Rudd, K. W. (1996). Parallel architectures. *ACM Computing Surveys (CSUR), 28*(1), 67-70.

[4] Grajales, T. (2000). Tipos de investigación. *On line)(27/03/2.000). Revisado el*.

[5] Jorba i Esteve, J., Margalef, T., & Morajko, A. (2006). *Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes*: Universitat Autònoma de Barcelona.

[6] Kirk, D. B., & Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*: Newnes.

[7] Leibovich, F., Gallo, S., De Giusti, A. E., De Giusti, L. C., Chichizola, F., & Naiouf, M. (2011). *Comparación de paradigmas de programación paralela en cluster de multicores: pasaje de mensajes e híbrido.* Paper presented at the XVII Congreso Argentino de Ciencias de la Computación.

[8] Munshi, A., Gaster, B., Mattson, T. G., & Ginsburg, D. (2011). *OpenCL programming guide*: Pearson Education.

[9] Nesmachnow, S. (2004). Algoritmos genéticos paralelos y su aplicación al diseño de redes de comunicaciones confiables.

[10] OpenCL, K. The open standard for parallel programming of heterogeneous systems.

[11] Piccoli, M. F. (2011). Computación de alto desempeño en GPU.

[12] Tello, A. A., & Huesca, J. d. J. T. (2012). OpenCL, Programación concurrente y paralela. *Editor: Prof. Ariel Ortiz Ramírez*, 75.

[13] Yang, C.-T., Huang, C.-L., & Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications, 182*(1), 266-269.

[14] Guim, F., & Rodero, I. Arquitecturas basadas en computación gráfica (GPU).

LINKOGRAFÍA

[15] https://www.khronos.org/

[16] http://www.nvidia.es/

[17] http://www.intel.es/

[18] http://www.amd.com/

[19] Les pages d'Alexandre, (2011). Les pages d'Alexandre. Recuperado de: http://www.obellianne.fr/alexandre/tutorials/OpenCL/tuto_opencl_codeblocks.php

[20] http://www.mpich.org/

[21] Mohammad ZeinEddin. (2012). MPICH2 on Code::Blocks. Recuperado de: http://zeineddin.blogspot.com/2012/10/mpich2-on-codeblocks.html

[22] Wes Kendall. (2015). MPI Hello World. Recuperado de: http://mpitutorial.com/tutorials/mpi-hello-world/

[23] Jonathan Tompson. (2012). An Introduction to the OpenCL Programming Model. Recuperado de: http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf

[24] Whisky & Rum bestellen. (2012). Cuda Programming. An Introduction to the OpenCL Programming Model. Recuperado de: http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf