EXPLOTACIÓN DE LA CONCURRENCIA DE AMBIENTES DE MODELADO Y SIMULACIÓN, BASADOS EN OBJETOS Y REGLAS

ÁNDERSON YAHIR VEGA CASTILLO

RUBÉN DARÍO HERNÁNDEZ RODRÍGUEZ

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA DE SISTEMAS
BUCARAMANGA
2014

EXPLOTACIÓN DE LA CONCURRENCIA DE AMBIENTES DE MODELADO Y SIMULACIÓN, BASADOS EN OBJETOS Y REGLAS

ÁNDERSON YAHIR VEGA CASTILLO
RUBÉN DARÍO HERNÁNDEZ RODRÍGUEZ

TRABAJO DE GRADO PARA OPTAR AL TÍTULO DE INGENIEROS DE SISTEMAS

Director
Ph.D. CARLOS JAIME BARRIOS HERNÁNDEZ

Codirector
M.Sc. HUGO HERNANDO ANDRADE SOSA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA DE SISTEMAS
BUCARAMANGA
2014

DEDICATORIA

Para mis padres principalmente, personas responsables de
Haber conseguido llegar al final de este tramo de vida
Para la UIS, SC3 y grupo SIMON
Para Rubén, mi compañero de trabajo y amigo
Para todos los profesores que me compartieron su conocimiento
Para todos mis amigos que estuvieron conmigo en este difícil camino
Para todos ellos muchas gracias

Anderson Yahir Vega Castilo

A todas aquellas personas que me apoyaron durante
Todo este proceso, a mis padres, a mi pareja
A mis amigos y compañeros
A mis profesores, al grupo de supercomputación,
Al grupo SIMON y la UIS.
A mi compañero Anderson, por su trabajo en equipo
y su apoyo.
A todos ellos infinitas gracias

Rubén Darío Hernández Rodríguez

AGRADECIMIENTOS

El éxito de este proyecto ha sido gracias al trabajo en conjunto de varias personas que prestaron tiempo de su vida para dedicarle a la realización del mismo:

Al director Carlos Barrios y al codirector Hugo Andrade, por la oportunidad que nos dieron, por la confianza depositada en nosotros, por su tiempo, a los conocimientos que compartieron con nosotros, por su constante seguimiento y a su apoyo durante todo el proceso.

Al grupo de supercomputación y calculo científico de la UIS, SC3 por sus conocimientos compartidos y colaboración.

Al grupo SIMON de investigaciones en modelado y simulación, por permitirnos trabajar con su software HOMOS, por la colaboración prestada y su constante apoyo.

Al Ingeniero Ángel María Valdés, autor del prototipo de HOMOS 2.0, por toda su colaboración, los conocimientos prestados, su interés y constante apoyo con el proyecto.

Al laboratorio de Cómputo de Alto Rendimiento y Calculo científico de la Universidad industrial de Santander (UIS). Por facilitar los recursos de hardware y prestarnos soporte necesario.

Por último, pero no menos importante a nuestra alma mater, la Universidad Industrial de Santander, por esta experiencia de vida que ha sido toda nuestra carrera, por su calidad educativa y por permitirnos ser parte de esta gran comunidad de profesionales UIS.

CONTENIDO

LISTA DE FIGURAS

Página

LISTA DE TABLAS

Página

LISTA DE ANEXOS

Página

ACRÓNIMOS

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphic Processing Unit
OpenGL	Open Graphics Library
SIMON	Grupo SIMON de Investigaciones en Modelado y Simulación
SISD	Single Instruction stream -Single Data stream
SIMD	Single Instruction stream - Multiple Data stream
MISD	Multiple Instruction stream - Single Data stream
MIMD	Multiple Instruction stream - Multiple Data stream

GLOSARIO

Computación de alto rendimiento: Herramienta para solución de problemas complejos, que permite la reducción en el tiempo de cálculo. Comercialmente, la computación de alto rendimiento es conocido como una computadora que tiene una alta potencia comparada con otras del mercado actual.

Concurrencia: En computación, la concurrencia es la propiedad de los sistemas que permiten que múltiples procesos sean ejecutados al mismo tiempo, y que potencialmente puedan interactuar entre sí. Los procesos concurrentes pueden ser ejecutados realmente de forma simultánea, sólo cuando cada uno es ejecutado en diferentes procesadores. En cambio, la concurrencia es simulada si sólo existe un procesador encargado de ejecutar los procesos concurrentes, simulando la concurrencia, ocupándose de forma alternada en uno y otro proceso a pequeñísimos intervalos de tiempo. De esta manera simula que se están ejecutando a la vez.

Programación paralela: Forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos mas pequeños, que luego son resueltos simultáneamente.

Paralelismo: es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente, basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.

RESUMEN

Título:

EXPLOTACIÓN DE LA CONCURRENCIA DE AMBIENTES DE MODELADO Y SIMULACIÓN, BASADOS EN OBJETOS Y REGLAS.*

Autores:

ÁNDERSON YAHIR VEGA CASTILLO
RUBÉN DARÍO HERNÁNDEZ RODRÍGUEZ**

Palabras Clave:

CPU, GPU, PROCESAMIENTO EN PARALEO, HOMOS, CONCURRENCIA, CUDA, MULTIPLES NUCLEOS

Durante hace varios años se han venido desarrollando proyectos en distintas áreas utilizando programación en paralelo, este tipo de computación se caracteriza por ejecutar varias instrucciones simultáneamente, denotando que al dividir un gran conjunto de procedimientos en fragmentos pequeños para ser tratados en forma paralela, se puede obtener una mayor eficiencia en tiempo de ejecución de la aplicación. Existen actualmente millones de aplicaciones dedicadas a realizar cualesquiera tipos de procesos, por esta razón es necesario diseñar código más eficiente, además teniendo en cuenta el auge de las CPU y GPU de múltiples núcleos, ya que este tipo de procesadores están cada vez más al alcance del público en general, se hace aún más evidente dicha necesidad. Es en este punto donde nace este proyecto, pretendiendo detectar oportunidades de concurrencia en los procedimientos implementados en la aplicación HOMOS, este software se basa en algunas reglas que rigen el comportamiento de unos objetos con el fin de modelar y simular un ambiente y esta manera obtener ciertos resultados semejantes a los dados por la situación en un ambiente real.

HOMOS es un software desarrollado por el grupo de investigación SIMON de la Universidad Industrial de Santander en 1998, está desarrollado en DELPHI, porque está desarrollado en este lenguaje de programación se debe seleccionar cuidadosamente la técnica más apropiada para aplicar mecanismos de supercomputación en algunas partes del código. Se decidió utilizar CUDA para así aprovechar los múltiples núcleos que proveen las poderosas tarjetas gráficas NVIDIA.

* Proyecto de Grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería de Sistemas. Director: Ph.D. Carlos Jaime Barrios. Codirector: M.Sc. Hugo Hernando Andrade Sosa

ABSTRACT

TITLE:
EXPLOITATION OF THE CONCURRENCY OF MODELING AND SIMULATION ENVIRONMENTS, BASED ON OBJECTS AND RULES.[1]

AUTHORS:

ÁNDERSON YAHIR VEGA CASTILLO
RUBÉN DARÍO HERNÁNDEZ RODRÍGUEZ[**]

KEYWORDS:

CPU, GPU, PARALLEL COMPUTING, HOMOS, CONCURRENCY, CUDA, MULTI CORE

DESCRIPTION:
During several years have been developing projects in different areas using parallel programming, this type of programming is characterized by running multiple instructions simultaneously , denoting that by dividing a large set of procedures in small fragments to be processed in parallel, can be greater efficiency in execution time of the application. There are currently millions of applications dedicated to perform any types of processes , therefore it is necessary to design more efficient code , also taking into account the rise of multicore CPU and GPU and that such processors are increasingly available to the public in general, is made even more evident the need. It is at this point that this project was , pretending identify opportunities for competition in the procedures implemented in the HOMOS application, this software is based on some rules that govern the behavior of each object in order to model and simulate an environment and thus obtain some similar to those given by the situation results in a real environment .

HOMOS was developed by the research group SIMON Universidad Industrial de Santander in 1998 software, is developed in Delphi, that was developed in this programming language should carefully select the most appropriate technique to implement mechanisms supercomputing in some parts of code. We decided to use CUDA to take advantage of multicores that provide powerful NVIDIA graphics cards.

[1] Bachelor Thesis
[**] Physicomechanical Engineering Faculty. System Engineering School. Director: PhD. Carlos Jaime Barrios. Codirector: MsC. Hugo Hernando Andrade Sosa

INTRODUCCIÓN

En Santander se encuentra actualmente la plataforma de supercomputación basada en GPU más importante del país, más específicamente en el valle de Guatiguará donde está ubicado el laboratorio de Supercomputación y Cálculo Científico de la Universidad Industrial de Santander, dicha supercomputadora es llamada GUANE-1. Además el SC3 posee los recursos esenciales para realizar el procesamiento en paralelo en la UIS, tales componentes se interconectan en una plataforma denominada GRIDUIS-2.

Teniendo en cuenta que para desarrollar un proyecto dentro del área de alto rendimiento computacional se cuenta con estos poderosos recursos, nace la intención de estudiar la posibilidad y viabilidad de acelerar ciertas aplicaciones, tal es el caso de HOMOS, una solución para el modelamiento basado en objetos y reglas. Se pretende estudiar cómo fue desarrollado el software, bajo que lenguaje y su arquitectura. Una vez que se logre interpretar como se estructuró HOMOS, cuales son las dependencias entre sus módulos, la principal acción a ejecutar es identificar aquel método que consume mas recursos en tiempos de ejecución debido a los procesos concurrentes que maneja y atacar esa parte del código mediante el uso de CUDA para poder aprovechar los múltiples procesadores en las GPU.

Para implementar código CUDA en proyecto HOMOS se debe realizar un proceso delicado, debido a la complejidad que presenta este hecho. Este software esta desarrollado en Delphi, un lenguaje de programación bastante visual y por lo tanto el uso de clases y objetos es la clave del desarrollo de cualquier proyecto basado en este tipo de lenguaje, debido al uso de estos objetos se presenta gran cantidad de concurrencia entre las distintas clases y este factor dificulta aún más el tratamiento del código. Luego de determinar los procesos que se van a atacar, es necesario llevar ese fragmento de código a un lenguaje C/C++ para que así se pueda implementar definitivamente el código de procesamiento en paralelo,

presentado por NVIDIA y de esta manera verificar que el software puede trabajar con problemas de simulación bastantes complejos de una forma más eficiente.

En el siguiente texto se encuentra consignada la justificación de este proyecto como también los objetivos propuesto del mismo, seguidos por aquellos conceptos necesarios para el desarrollo de la propuesta y algunos proyectos que aplicaron soluciones afines a sus respectivas aplicaciones. Además se describe la metodología utilizada a lo largo del proceso y también las distintas pruebas y resultados obtenidos de ellas. Por último se anexan las limitaciones encontradas en el desarrollo del proyecto, las conclusiones que resultaron y como aspecto bastante importante cabe resaltar las recomendaciones planteadas para futuros desarrollos de investigación que se quieran realizar en esta área.

1. DESCRIPCIÓN DEL PROYECTO

1.1 JUSTIFICACIÓN

El modelamiento y simulación basados en objetos y reglas posee procesos naturales que pueden tener concurrencia, es decir, existen ciertas tareas que claramente pueden aprovechar el procesamiento en paralelo. Los ambientes generados por el uso de esta metodología exigen grandes recursos computacionales, cuando se trata de capacidad de procesamiento y de memoria.

Hoy en día existen plataformas que soportan dichas exigencias, garantizando altas prestaciones a diferentes escalas: desde computadores personales, que poseen unidades de procesamiento de múltiples núcleos, hasta computadores con cientos de unidades de procesamiento y gran capacidad de memoria.

Teniendo en cuenta herramientas robustas para el modelado y simulación basados en objetos y reglas, como HOMOS y las arquitecturas existentes que soportan el procesamiento en paralelo, se plantea la explotación de la concurrencia al mapear los procesos eficientemente sobre estas arquitecturas.

Al implementar el procesamiento en paralelo en aquellos procesos concurrentes de HOMOS, podremos obtener una mayor eficiencia en los tiempos de procesamiento y un mejor aprovechamiento de los recursos de hardware. Logrando así, que HOMOS deje de ser una herramienta que trabaje de forma secuencial, motivo por el cual, se ve limitado cuando se trabaja con modelos de gran complejidad.

1.2 DESCRIPCIÓN DEL PROBLEMA

Cuando se desarrolla software bajo el paradigma de programación secuencial se trabajan bloques de código en los cuales se realiza una tarea por vez, según el orden en el cual se realiza la ejecución. Teniendo en cuenta que actualmente la tecnología de múltiples núcleos, se ha ido posicionando en la población promedio es evidente que las aplicaciones que se usan no aprovechan al máximo el rendimiento proporcionado por dicha configuración, por esto es posible predecir que dentro de poco la programación en paralelo llegará a ser un paradigma primordial a tener en cuenta en el desarrollo de software de gran envergadura.

Si bien es cierto que no toda aplicación requiere que sus procesos sean divididos para luego ser resueltos simultáneamente, existe una gran cantidad de ellas en diversas áreas del conocimiento que por el contrario si se implementara esta clase de programación se obtendrían resultados bastante acelerados donde antes eran realmente costosos en términos de tiempo, aumentando demasiado la eficiencia del programa

Hoy en día la Universidad Industrial de Santander cuenta con el Parque Tecnológico de Guatiguará, centro especializado en distintas áreas tecnológicas y dentro de este se encuentra ubicado el laboratorio de Supercomputación y Cálculo Científico (SC3). Este laboratorio reúne los recursos más importantes que permiten el procesamiento en paralelo en la UIS, cuenta además con plataformas de alto rendimiento que posicionan el SC3 como uno de los centros de cómputo más importantes del país, una de estas tecnologías y además la más importante cuando se habla de procesamiento con GPU's a nivel nacional es el GUANE-1, recurso bastante poderoso en materia de procesamiento en paralelo.

Observando los recursos disponibles y la posibilidad de mejorar el rendimiento del software HOMOS se evaluará la viabilidad de explotar la concurrencia en los

procesos que maneja esta aplicación y así poder modelar ambientes de gran tamaño interactuando con grandes cantidades de información.

1.3 OBJETIVO GENERAL

Identificar las oportunidades de concurrencia en la herramienta HOMOS, la cual trabaja con ambientes de modelado y simulación basada en objetos y reglas, y así proponer mecanismos de implementación que soportan el procesamiento en paralelo.

1.4 OBJETIVOS ESPECÍFICOS

- Analizar arquitectura de la aplicación, identificar oportunidades de implementación de procesamiento en paralelo.
- Definir una estrategia para aprovechar la concurrencia y las oportunidades de implementar un modelo en paralelo.
- Desarrollar un prototipo de acuerdo a la estrategia seleccionada y evaluar su rendimiento.
- Establecer lineamientos de procesamiento en paralelo y explotación de concurrencia.

2. MARCO TEÓRICO

2.1 HOMOS[2]

Es una herramienta que aborda una nueva área de investigación para el desarrollo de la teoría de los sistemas dinámicos y el modelamiento de sistemas basados en objetos y reglas, y de esta manera sumar una característica importante en el modelado y simulación, la distribución espacial de los objetos que interactúan en el sistema evaluado. Homos provee la facilidad de diseñar las clases que servirán como plantilla para crear objetos que intervendrán en el proceso de simulación, también es posible crear las reglas que rigen el comportamiento de cada objeto y el comportamiento grupal. La gran ventaja que Homos tiene es que hace mucho más fácil el proceso de modelamiento sistémico al usuario ya que lo exime de preocupaciones matemáticas de la teoría de autómatas celulares, además del desarrollo de complejos algoritmos que rigen el comportamiento de los objetos, permitiendo así centrarse en el ambiente de modelado tal cual como si estuviera pintando un paisaje donde los objetos interactúan entre ellos y reflejan la realidad sistémica que resulta de esta.

La Metodología de Modelamiento basada en Objetos y reglas, se basa en algunos principios de **Autómatas Celulares[3]**. Asume como espacio, en el cual se desenvuelve la dinámica del fenómeno, una matriz de celdas donde cada una contiene o no un objeto, y a cada nuevo paso de simulación la celda determina su nuevo estado y el de sus vecinos siguiendo reglas de comportamiento e interacción específicas entre ellos. Junto a lo anterior, la **Metodología Orientada**

[2] Mogotocoro, Carmen; Lozano, Oscar. HOMOS 1.0 Herramienta software para el modelamiento y simulación basado en objetos y reglas. TESIS (INGENIERO DE SISTEMAS) - UIS., ESCUELA DE INGENIERIA DE SISTEMAS E INFORMATICA, 1998.
[3] Martínez, Genaro. Introducción a la simulación de procesos con autómata celular. [En línea]. Disponible en: <http://2006.igem.org/wiki/images/1/1b/IntoCA.pdf>

a Objetos[4] define objetos de una misma clase, donde cada clase tiene atributos y métodos que pueden relacionarlos con otros objetos. La interacción entre los objetos se define por reglas. Por ejemplo, la regla "reproducción" define que si dos objetos de la misma clase se encuentran en celdas de la misma vecindad, se genera un nuevo objeto de la misma clase, asociando a la regla una probabilidad de ocurrencia. Así se pueden definir otras reglas entre objetos de la misma clase y clases diferentes.

2.2 PROCESAMIENTO EN PARALELO[5]

El desempeño de las computadoras determina la clase de problemas que se pueden resolver. Existen problemas tan complejos, que su solución podría tomar muchos años, por ello se han buscado soluciones computacionales que puedan dar una solución rápida a estos problemas. Uno de los métodos desarrollado para lograr las soluciones es el procesamiento en paralelo.

La idea básica detrás del procesamiento paralelo es que varios dispositivos (procesadores), ejecutando simultánea y coordinadamente las tareas, pueden rendir más que un único dispositivo. El problema fundamental son las innovaciones tecnológicas que se requieren para obtener ese rendimiento mejorado.

Si bien el procesamiento paralelo ofrece una ventaja definitiva en cuanto a costos, su principal beneficio, la escalabilidad (capacidad de crecimiento), puede ser difícil de alcanzar. Esto se debe a que conforme se añaden procesadores, las disputas por los recursos compartidos se intensifican.

---

[4] Franco, Ricardo. Metodología para el desarrollo de aplicaciones orientadas a objetos. [En línea]. Disponible en: <http://3d-tips.wikispaces.com/file/view/Desarrollo+de+aplicaciones+basado+en+UML.pdf>

[5] Universidad Internacional del Ecuador, Facultad de Informática y multimedia. Procesamiento en paralelo. [En línea]. Disponible en: <http://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r62976.PDF>

Algunas alternativas de arquitecturas de procesamiento paralelo enfrentan este problema fundamental, con diferentes resultados, entre las que se puede mencionar:

• Multiprocesamiento simétrico
• Procesamiento masivamente paralelo
• Procesamiento paralelo escalable

Cada diseño tiene sus propias ventajas y desventajas. Cada diseño tiene sus propias ventajas y desventajas. La programación en paralelo también se ha venido utilizando de diferentes formas en computadores secuenciales:

- Segmentación encauzada
- Paralelismo a nivel de instrucción
- Ejecución fuera de orden

2.2.1 Clasificación[6]

Paralelismo implícito – bajo nivel

- Mejora de la concurrencia de la CPU
- Oculta a la arquitectura computacional
- Aprovechamiento de los recursos de paralelismo de la CPU
- Segmentación o pipeline
- Ejecución fuera de orden
- Procesadores auxiliares (Video – FPU)
- SIMD

Paralelismo explícito – alto nivel

[6] Universidad Tecnológica Nacional. Procesamiento Paralelo. [En línea]. Disponible en: <http://www.electron.frba.utn.edu.ar/materias/950419/archivos/20100614_Procesamiento_Paralelo.pdf>

- Varios procesadores con memoria dedicada
- Varios procesadores con memoria compartida
- Ofrecen una infraestructura explícita para el desarrollo de software

2.2.2 Modelos de computación en paralelo[7]

En 1972 Flynn estableció una clasificación según la cual todo sistema de cómputo pertenece a una de las siguientes cuatro categorías: **SISD**, **SIMD**, **MISD** y **MIMD**.

SISD: A la categoría SISD corresponde el modelo de computador tradicional (Von Neumann), en el que más allá de las variantes, se lleva a cabo un procesamiento secuencial mediante la ejecución de una única instrucción sobre un único dato por vez.

SIMD: Dentro del tipo SIMD se consideran los sistemas que a partir de una única instrucción son capaces de procesar un conjunto o *"array"* de datos en forma simultánea, con la particularidad de ejecutar la misma instrucción sobre cada uno de sus elementos.

MISD: La categoría MISD corresponde a configuraciones en las que un tren de unidades de procesamiento se surte por uno de sus extremos de un dato extraído de memoria y devuelve un resultado a la misma memoria por el extremo opuesto, en tanto cada unidad del tren toma por entrada al dato procesado por la unidad anterior y da por salida el que será tomado como entrada por la unidad siguiente.

MIMD: MIMD es un modelo en el cual el sistema puede considerarse de alguna forma particionado en unidades con capacidad de, desarrollar su propio cómputo y Comunicarse con las demás; de modo que al ejecutar cada una de ellas un determinado flujo de instrucciones sobre su propio flujo de

[7] Murray, Leslie. El Procesamiento Paralelo, Enfoque Cualitativo y Simulación. [En línea]. Disponible en:<http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR0102.pdf>

datos se logra que en conjunto, múltiples flujos de instrucciones actúen sobre otros tantos flujos de datos.

Estos sistemas a su vez pueden clasificarse en dos grandes categorías, los **Multiprocesadores** o sistemas de memoria compartida en los cuales la comunicación entre las distintas unidades se realiza, en forma implícita a través de variables compartidas dentro de una única memoria principal y los **Multicomputadores** o sistemas de memoria distribuida en los que la comunicación explícita entre unidades se realiza a través de una Red de paso de mensajes y donde cada unidad de procesamiento opera sobre su propia memoria.

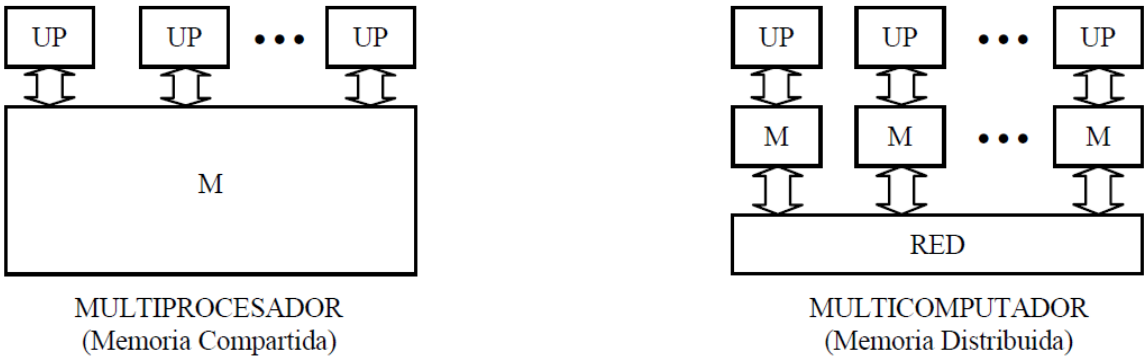


Figura 1 Multiprocesador y Multicomputador. Fuente: Leslie Murray, El Procesamiento en Paralelo Enfoque Cualitativo y Simulación

2.2.3 Espacios de direccionamiento[8]

Cuando se resuelve un problema entre varios procesadores es necesario que éstos se comuniquen entre sí para intercambiar información. El paso de mensajes y la memoria compartida proveen dos formas de comunicación:

[8] Arce Misael, Becerra Georgina, Vidal Antonio. Implementación CPU – GPU y comparativa de las bibliotecas BLAS-CUBLAS, LAPAK-CULA. [En línea]. Disponible en: <http://riunet.upv.es/bitstream/handle/10251/11735/INCO2-2011-01.pdf?sequence=1>

- Paso de mensajes
- Memoria compartida

2.2.4 Granularidad

Una computadora paralela puede estar compuesta por un pequeño número de procesadores muy potentes o por un gran número de procesadores poco potentes. La Granularidad se puede definir como la relación entre el tiempo requerido por una operación de comunicación básica y el tiempo requerido por un cómputo básico. Las computadoras paralelas en las que esta relación es pequeña son apropiadas para algoritmos que requieren comunicación frecuente, es decir, en los algoritmos en los que el tamaño de grano es pequeño (antes de la comunicación requerida). Estos algoritmos contienen paralelismo de grano fino, comúnmente estas computadoras son llamadas computadoras de grano fino. En contraste las computadoras en las que dicha relación es grande son adecuadas para algoritmos que no requieren de mucha comunicación, estas computadoras son referidas como computadoras de grano grueso.

También se considera el número y la potencia de los procesadores de una computadora para determinar la Granularidad.

- **Grano Grueso** Pocos procesadores de gran potencia.
- **Grano Mediano** Poco más de mil procesadores de mediana o baja potencia.
- **Grano Fino:** Del orden de las decenas de millar de procesadores de mediana o baja potencia.

2.2.5 Métricas de desempeño

Los algoritmos secuenciales son evaluados en términos de su tiempo de ejecución en función del tamaño del problema. Se define el tamaño del problema como el número de datos involucrados en una ejecución.

Cuando se trata de algoritmos paralelos el tiempo de ejecución depende no solamente del tamaño del problema sino también de la arquitectura y el número de procesadores de la(s) computadora(s). Por lo tanto para evaluar el rendimiento de un algoritmo dado, es necesario usar métricas de desempeño como:

- **Tiempos de ejecución**
- **Aceleración**

2.2.6 CPU vs GPU[9]

La computación paralela usando aceleradores ha ganado la atención la investigación extensa en los últimos años. En particular, el uso de GPUs para computación de propósito general ha dado a luz varios casos de éxito con respecto al tiempo tomado, costo, poder y otras métricas significativamente, sin embargo, acelerador basado en la informática ha relegado el papel de la CPU en la computación. Como CPUs evolucionan y también ofrecen adecuar los recursos computacionales, es importante incluir también CPUs en el cálculo. A esto le llamamos el modelo de computación híbrida. De hecho, la mayoría de los sistemas informáticos de la época actual ofrecen un grado de heterogeneidad y, por tanto, un modelo de este tipo es bastante natural.

[9] Programación y Paralelismo (CPU vs GPU). [En línea]. Disponible en: <http://fcharte.com/Default.asp?noticias=2&a=2010&m=9&d=22>

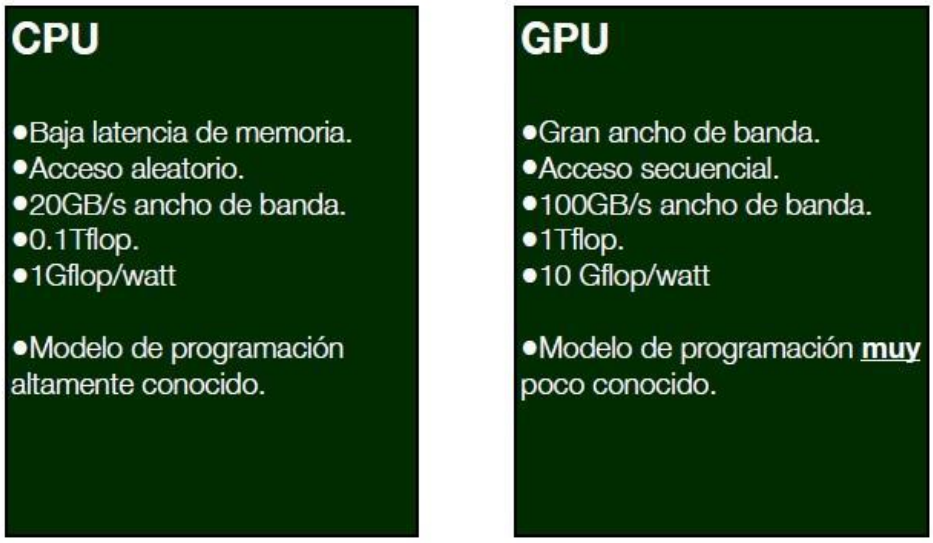


Figura 2. Comparación entre CPU y GPU. Fuente: Amilcar Meneses, Introducción a GPU's y Programación CUDA para HPC

2.3 CUDA

CUDA[10] es una plataforma de computación en paralelo y el modelo de programación inventado por NVIDIA. Permite a un aumento espectacular en rendimiento informático mediante el aprovechamiento de la potencia de la unidad de procesamiento gráfico (GPU). Con millones de GPUs habilitadas con CUDA vendidas hasta la fecha, los desarrolladores de software, los científicos y los investigadores están encontrando usos de amplio alcance para la computación GPU CUDA.

2.3.1 Funcionamiento de CUDA[11]

CUDA es un modelo de programación de propósito general el cual posee una

[10] Nvidia Corporation. What is CUDA? [En línea]. Disponible en: <http://www.nvidia.com/object/cuda_home_new.html>
[11] Meneses Amilcar. Introducción a GPU's y programación CUDA para HPC. [En línea]. Disponible en: <http://computacion.cs.cinvestav.mx/~ameneses/pub/notas/cuda_taller.pdf>

serie de características:

- El usuario inicializa conjuntos de threads en el GPU
- GPU = super-threaded dedicado para procesamiento masivo de datos
- (co-processor)
- Conjunto de software dedicado
- Manejadores de dispositivos, lenguaje y herramientas.
- Manejador para carga de programas al GPU
- Manejador Independiente - Optimizado para computaciones
- Interfaz diseñada para computaciones - API no gráfica
- Comparte datos con objetos OpenGL en buffer
- Aceleración garantizada en los accesos a memoria
- Manejo explícito de la memoria del GPU

El código paralelo o *Kernel* es lanzado y ejecutado en el dispositivo (GPU) por varios *threads*. Dentro del funcionamiento del código, cada *thread* es contenido en un *bloque* de *threads* y cada *bloque* está organizado dentro de una malla o *grid*, una *grid* solo puede ejecutar un *kernel*, para ello el código en paralelo debe ser escrito para un thread y será replicado según la configuración de la ejecución.

En el ámbito de CUDA:
Device = GPU
Host = CPU
Kernel = Función llamada desde el Host que se ejecuta en Device

Un CUDA Kernel se ejecuta mediante un array de Threads y todos los Threads ejecutan el mismo código. Así mismo cada Thread tiene un ID que se usa para direccionar la memoria y tomar las decisiones de control. La siguiente Figura muestra como se encuentra estructurada la GRID de CUDA.

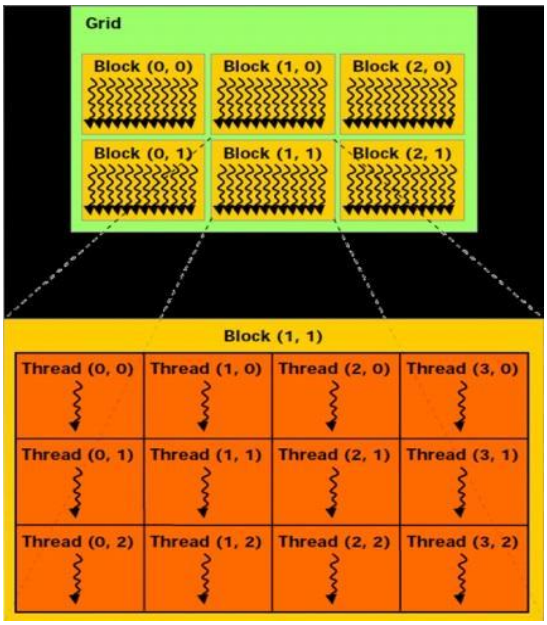


Figura 3. GRID de CUDA. Fuente: CUDA: Modelo de programación. Rondán Andrés

Los hilos o threads son identificados mediante **threadidx:** que es un vector de elementos 3D, así cada hilo puede ser identificado por un índice de 1,2 ó 3 dimensiones. Los hilos de un bloque pueden cooperar entre sí mediante el uso de memoria compartida dentro del bloque y sincronizando su ejecución para coordinar los accesos a memoria. Las grids pueden ser de 1 o 2 dimensiones, luego cada bloque dentro de un grid puede ser direccionado por un índice de 1 o 2 dimensiones mediante **blockidx.** Asimismo, la dimensión del bloque también se puede obtener desde dentro del kernel mediante **blockDim.**

2.3.2 Jerarquía de memoria

Los threads de cuda, pueden acceder a los datos de múltiples espacios de memoria durante su ejecución. Cada thread posee su propia memoria local. Cada

bloque tiene su propia memoria compartida por todos los threads del bloque y con el mismo tiempo de vida que los threads que lo componen. Todos los hilos tienen acceso a la memoria global, además existen otros 2 espacios de memoria adicionales de solo lectura: constant y texture memory. La figura 4 muestra la jerarquía de memoria dentro de CUDA.

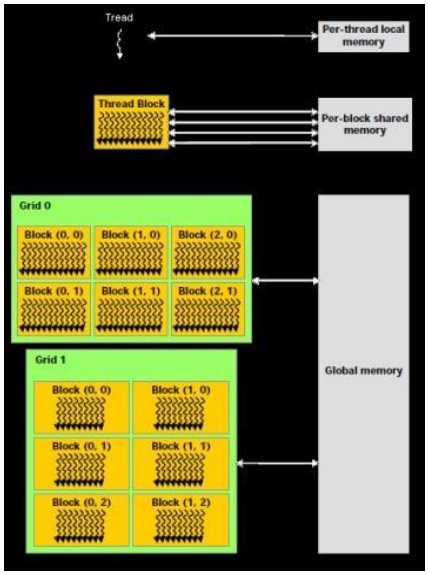


Figura 4. Jerarquía de memoria CUDA. Fuente: CUDA: Modelo de programación. Rondán Andrés

El modelo de programación de CUDA asume que los threads de CUDA se ejecutan en un device, que actúa como coprocesador de un host que ejecuta un programa. También asume que host y device poseen su propia DRAM, host memory y device memory. CUDA proporciona instrucciones para reservar, liberar, copiar memoria en la memoria del device, así como transferir datos entre el *host* y el *device*.

3. ESTADO DEL ARTE

A medida que transcurren los años van surgiendo nuevas arquitecturas capaces de soportar gran capacidad de procesamiento gracias a los múltiples procesadores comprendidos en una misma pieza. Un ejemplo de este constante avance son las tarjetas gráficas desarrolladas por NVIDIA. Las primeras GPU fueron diseñadas como aceleradoras de gráficos y solo permitían algunos procesos específicos de funcionamiento no dinámico. Es terminando el siglo 20 cuando las arquitecturas se volvieron más programables y exactamente en 1999 se logra conseguir el desarrollo de la primera GPU de NVIDIA.

En el año 2003, un equipo de investigadores dirigidos por Ian Buck lanzó "Brook", el primer modelo de programación ampliamente adoptado para extender C con constructos con datos paralelos. Usando conceptos como los flujos, los kernels y los operadores de reducción, el compilador Brook y el sistema de tiempo de ejecución expusieron la GPU como un procesador con fines generales en un lenguaje de alto nivel. Y lo más importante es que los programas de Brook no sólo eran más fáciles de escribir que el código de la GPU ajustado a mano, sino que eran siete veces más rápidos que el código similar existente.
NVIDIA sabía que un hardware impresionantemente rápido tenía que combinarse con herramientas de hardware y software intuitivas e invitó a Ian Buck a unirse a la compañía y a empezar a hacer evolucionar una solución que ejecutara C a la perfección en la GPU. Al reunir el software y el hardware, NVIDIA lanzó CUDA en 2006, la primera solución del mundo para computación general en las GPU.[12]

A continuación se mencionan algunos proyectos que implementaron CUDA como

[12] Nvidia Corporation. GPU COMPUTING: THE REVOLUTION [En línea]. Disponible en: <http://www.nvidia.com/object/cuda_home_new.html>

solución para acelerar sus aplicaciones.

3.1 NAMD SCALABLE MOLECULAR DYNAMICS

Descripción: Es un código paralelo de dinámica molecular diseñado para simulaciones de altas prestaciones de grandes sistemas biomoleculares. Basado en objetos Charm ++, NAMD recurre a cientos de núcleos para las simulaciones típicas e incluso más de 200.000 núcleos para las simulaciones más grandes. NAMD usa el programa de gráficos moleculares populares VMD para la configuración de la simulación y el análisis de la trayectoria, pero también en ficheros compatibles con AMBER, CHARMM, y X- PLOR. NAMD es código libre.

Desarrollado por: Profesor Kale y profesor Robert Skeel del Grupo de biofísica teórica y computacional del Instituto Beckman.

3.2 SYNTHESIS OF ARTIFICIAL NEURAL CIRCUITRY

Descripción: Es evidente que los circuitos neuronales de los cerebros biológicos son bastante complejos de mapear. Además, la investigación en neurociencia ha establecido recientemente que estos circuitos neuronales se reconfiguran a sí mismos durante toda la vida, incorporando información sobre el mundo exterior y la actividad interna por igual. Evolved Machines es el primer grupo de investigación en el mundo, ya sea en la academia o la industria, en simular el crecimiento de una malla de neuronas y, además, utilizar este proceso para sintetizar matrices neuronales que realizan la función sensorial.

Este trabajo de investigación requiere de gran capacidad de computación paralela ya que por ejemplo una sola neurona conlleva a evaluar una ecuación diferencial 200’000.000 de veces por segundo, lo que supone la necesidad de uso de 4 gigaflops. Un arreglo neuronal en este tipo de circuitos de compone de miles de

neuronas, en conclusión se necesita más de 10 teraflops de poder. Evolved Machines comenzó a trabajar con GPUs NVIDIA en 2006. Alcanzaron aceleraciones de aproximadamente 130 veces con respecto a los resultados obtenidos con los procesadores x86.

Desarrollado por: Evolved Machines

3.3 ABINIT

Descripción: ABINIT es un paquete cuyo programa principal permite encontrar la energía total, densidad de carga y la estructura electrónica de los sistemas hechos de electrones y núcleos (moléculas y sólidos periódicos) dentro de la Teoría de la densidad funcional (TDF), utilizando pseudopotenciales. ABINIT también incluye opciones para optimizar la geometría de acuerdo con las fuerzas de DFT, o para realizar simulaciones de dinámica molecular usando estas fuerzas, o para generar matrices dinámicas, cargas efectivas Born, y tensores dieléctricos, basado en la Teoría de la Perturbación de la Densidad Funcional, y muchas más propiedades. Con la implementación de CUDA se logro obtener una aceleración de hasta 2.7 X.

Desarrollado por: The Abinit group

3.4 PATHWISE

Descripción: PathWise es una solución de gestión de riesgos de anualidad que apoyar el desarrollo de nuevos productos, la información financiera y reglamentaria, las actividades de gestión de riesgos empresariales, y la gestión y presentación de informes del programa de coberturas. La plataforma PathWise es una solución informática de alto rendimiento integrado basado en las tecnologías de computación en paralelo avanzadas, como la GPU Tesla de NVIDIA, y una interactiva grid middleware especialmente diseñada, que junto a acelerar

significativamente la gestión de riesgos de simulación basado en semanas y días, a minutos o segundos, la entrega de los niveles de rendimiento en tiempo real, y el apoyo para la previsión de la próxima generación y la toma de decisiones.

PathWise es la solución empresarial de gestión de riesgos más rápida y altamente integrada en su género.

Desarrollado por: AON

3.5 INSIGHT EARTH

Descripción: Es un software de interpretación sísmica 3D que permite identificar las regiones más productivas en recursos de esquisto. También sirve para descubrir aspectos ocultos en las zonas actuales de producción mediante la eliminación de la estructura actual sobreimpresa de todo el volumen, para ampliar la productividad más allá de las limitaciones convencionales. Además para que sea posible definir rápidamente los límites de las masas de sal con alta precisión necesariamente se debe usar esta tecnología para la interpretación. Por último este software permite Interpretar los fallos en las obras de gran complejidad con una claridad sin precedentes, la eliminación de ruido y las huellas en los datos sísmicos

Desarrollado por: Originalmente dentro de TerraSpark, financiado por el Consorcio de Interpretación y Visualización de Geociencia (CIVG). Insight Earth fue lanzado al mercado en 2008, y desde entonces ha proporcionado a los clientes mejoras en la velocidad y la precisión de la interpretación.

4. METODOLOGÍA

La metodología que se usó dentro del proyecto, se trata de una variante del método científico, con el cual se obtuvo una concepción y diseño del sistema. Este se realizó en 4 diferentes fases que estuvieron en constante realimentación.

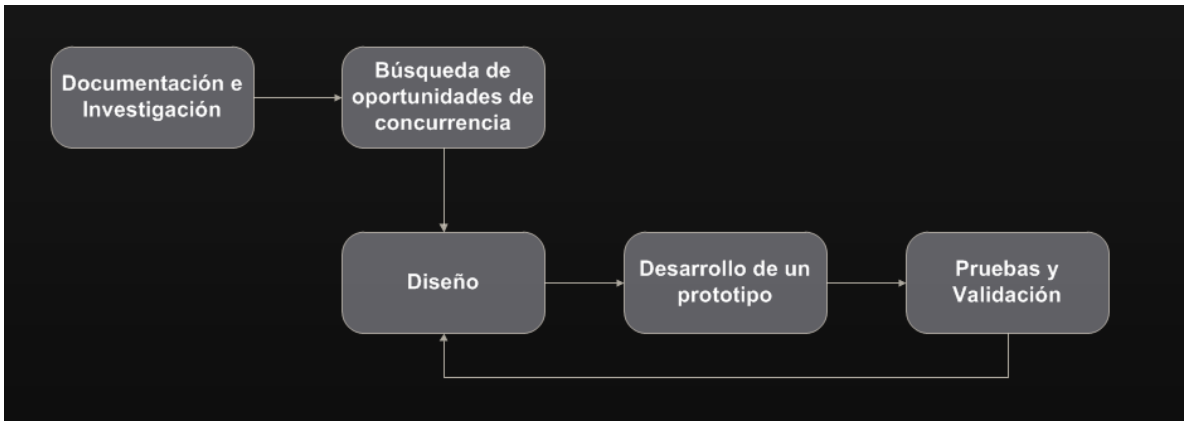


Figura 5. Metodología. Fuente: Autores

PRIMERA FASE: *DOCUMENTACIÓN E INVESTIGACIÓN*

La primera fase inició con la búsqueda y recopilación de información sobre el software en el cual se trabajara, en este caso HOMOS. El software fue desarrollado en el grupo SIMON de Investigaciones en Modelado y Simulación, se enfocó la búsqueda en este grupo. Gracias a la colaboración de miembros del grupo, tanto actuales como antiguos, se logró consolidar una serie de archivos donde se concentraba la mayor información sobre la primera versión de HOMOS, así mismo se obtuvo información del prototipo de la versión 2.0 de HOMOS, que aunque se encuentra sin implementar, resaltaba en gran medida las posibilidades dentro de lo que puede llegar hacer HOMOS, que justifica en medida la necesidad de una mejora en su rendimiento. La información recopilada en resumen revela

que HOMOS es un software desarrollado a través de una IDE enfocada en interfaz gráfica como lo es DELPHI, HOMOS ha sido construido a partir de diferentes clases con funciones específicas, que en conjunto hacen que el software funcione. El prototipo de HOMOS 2.0 ayudó a comprender las debilidades del sistema actual, así como posibles mejoras para aplicar en un futuro incluyendo la versión 2.0.

En esta fase se obtuvo el diagrama UML de Clases del sistema, el cual permitió el inicio de un proceso para la obtención de la arquitectura del software, que inicio con la elaboración del diagrama de casos de uso, así como un diagrama de componentes del software.

La documentación continúo con un rastreo del origen de la compilación de HOMOS, del cual se logró obtener el código fuente de la versión final de HOMOS 1.0.0.73, el cual se compiló con la IDE Borland Delphi 3 Client/Server Edition. A partir del código fuente, se realizó un análisis del grado de dependencia que existía entre los elementos que conforman el software, de este análisis se obtuvo un extenso diagrama de dependencias, que muestra como es la red de trabajo en el funcionamiento de HOMOS. (Véase anexo A)

A partir del análisis del funcionamiento del software y con referencia a los diagramas ya existentes se obtuvo una idea más clara de la arquitectura del mismo. De este modo se obtuvo un diagrama que sintetiza la organización de HOMOS. (Véase anexo B)

La recopilación de información sobre las técnicas y modelos de programación en paralelo, fueron estudiados por medio de las fuentes bibliográficas ofrecidas por la universidad, libros, internet y el conocimiento de las diferentes personas que participaron del proceso, además de esto se tomaron pequeños cursos sobre el procesamiento en paralelo. Gracias a esta curva de aprendizaje se pudieron hacer predicciones sobre cuáles serían las técnicas que mejor se acoplaran al software.

Resultados alcanzados

1. Se localizaron varias carpetas con archivos afines con el software HOMOS, además se obtuvo acceso a carpetas sobre el prototipo de HOMOS 2.0
2. Se logró consolidar diferentes diagramas que permiten ver de una manera mas clara el funcionamiento y la arquitectura de HOMOS.
3. Se logró un aprendizaje acerca de las técnicas de procesamiento en paralelo, así como ideas preliminares de una posible solución.

Principales dificultades:

Uno de los mayores inconvenientes en el momento de buscar y recopilar información sobre HOMOS, debido a la antigüedad del software, luego a partir de una búsqueda más detallada, se logró contactar al autor del prototipo de HOMOS 2.0, el cual compartió la información perteneciente al proyecto HOMOS 1.0.
Otra dificultad se presentó al momento de visualizar la dependencia de los elementos del sistema de HOMOS, debido al tamaño del mismo que se convirtió en algo extenso y de una difícil comprensión, pero que ayudó bastante al momento de obtener la arquitectura del software.
Cumplir la curva de aprendizaje sobre el procesamiento en paralelo, en el tiempo disponible para el proyecto se consideró un reto y una carrera contra el reloj.

SEGUNDA FASE: *BÚSQUEDA DE OPORTUNIDADES DE PROCESAMIENTO CONCURRENTE Y DISEÑOS PRELIMINARES*

En primer lugar se inició con pruebas sencillas de funcionamiento, donde se encontraron varias limitaciones y problemas que tiene el software HOMOS, que concuerdan con las mencionadas en la documentación de HOMOS 2.0, tales como problemas de desbordamiento. A partir del análisis de la arquitectura del

software, se pudo extraer los principales módulos o componentes con los cuales trabaja HOMOS. De estos dos análisis se presentó una primera hipótesis de cuál debía ser el componente a atacar, el cual se trataba del módulo o motor de simulación.
Para demostrar dicha hipótesis, se utilizaron dos herramientas para realizar pruebas de rendimiento y tiempo de cómputo del software HOMOS: ProDelphi y Automatized AQTime 6, que revelaron datos que ayudaron a concluir finalmente que el simulador, era quien tomaba mayor tiempo de cómputo, por lo cual se decidió ser el objetivo de ataque.

Se inició el análisis de código fuente del motor de simulación, el cual contenía varios métodos para el funcionamiento del software. A través de las pruebas de rendimiento se logró detectar que método influía más dentro de la carga computacional y tiempo de ejecución del software HOMOS. Este método hace llamadas a dos métodos, los cuales se encargan de controlar los estados de la simulación y la posible dirección o vecindad que tienen cada objeto dentro de la malla de HOMOS. Específicamente el método encargado de las vecindades es quien presenta una carga computacional considerable con respecto al software, además este método era aquel tenía el suficiente grado de paralelismo para ser atacado, esto debido a que este dentro de su algoritmo, permite el manejo de datos en diferentes hilos y así poder implementar un procesamiento en paralelo.

Dentro se buscaron los ciclos anidados y el grado de dependencia que puedan tener las variables dentro del algoritmo, además de esto se tuvo en cuenta la granularidad del mismo. Para un objeto de estudio mas detallado, se aisló el código del algoritmo en un proyecto Delphi aparte para futuras pruebas.

Dentro de las posibilidades de aplicar una técnica de programación en paralelo se presentaron varias opciones que se ajustaban al algoritmo, todas ellas tenían sus ventajas, pero finalmente se tomo la decisión de utilizar CUDA, un modelo de

programación desarrollado por Nvidia para procesamiento de alto rendimiento en tarjetas graficas. Este modelo se escogió primero, por su ya conocido rendimiento, segundo por que en términos de hardware, es mas probable que en computadoras personales de uso cotidiano, se cuente con una tarjeta de video de múltiples núcleos y tercero debido a la visión de que en un futuro serán mayormente aprovechadas las tarjetas graficas para el cálculo computacional en términos de software cotidiano.

Resultados alcanzados:

1. Se comprobaron y encontraron fallos dentro del software, además se encontraron otras falencias del mismo.
2. Se lograron realizar diferentes pruebas de rendimiento, que se reflejaron en métricas para la toma de decisiones.
3. Se definió que el motor de simulación, es donde se requería realmente la optimización por medio de procesamiento en paralelo.
4. Dentro del motor de simulación se logró definir qué método atacar específicamente, para diseñar e implementar un modelo de procesamiento en paralelo para el mismo.
5. Se aisló el código del método objetivo para un análisis más sencillo y detallado.
6. Se concluyo que CUDA seria el modelo de programación en paralelo a usar.

Principales dificultades:

Dentro de las principales dificultades dentro de esta fase, se encuentra el seguimiento de los datos requeridos por el método escogido, debido a que la dependencia del código entre clases es bastante alta.

La replicación de datos para el aislamiento del algoritmo, requirió que estos se extrajeran del código original, para que el funcionamiento del mismo fuera exacto.

TERCERA FASE: *DISEÑO E IMPLEMENTACIÓN*

Con el modelo de programación CUDA, se empezó un análisis del funcionamiento y el flujo de datos hacia el método seleccionado. Debido al modelo de programación escogido se llegó a la necesidad de trasladar el algoritmo al lenguaje de programación C++, eso se llevó a cabo por medio de la replicación de los datos utilizados en el programa original, específicamente en el método en análisis. De este modo se llevó a cabo la traducción del algoritmo de Delphi a C++, se insertaron los mismos datos y se realizaron pruebas de ejecución. Una vez el algoritmo estuvo aislado, se inició el diseño del algoritmo en el modelo de programación de CUDA, para un correcto aprovechamiento del mismo.

El diseño se enfocó en hacer que cada una de las búsquedas que realiza el algoritmo en la búsqueda de una nueva vecindad disponibles, se ejecutará en diferentes hilos por separado, para que de este modo los diferentes hilos ejecutando en paralelo, y cada uno realice una búsqueda individual de cada uno de los datos. La implementación de CUDA inició con pruebas y prototipos sencillos con manejo de datos pequeños, para luego hacer una implementación completa del algoritmo, al final se desarrollo un algoritmo que puede manejar tantos datos, como la tarjeta lo permita.

Resultados alcanzados:

1. Se lograron replicar datos del software para la utilización en las pruebas del algoritmo aislado.
2. Se logró implementar el algoritmo en otro lenguaje de programación tal como C++.
3. Se realizó un diseño del algoritmo dentro del modelo de CUDA, para

ejecutarlo en paralelo.

4. Se implementó exitosamente el algoritmo en CUDA, utilizando el diseño propuesto y los datos replicados del software original.

Principales dificultades:

Nuevamente debido al alto grado de dependencia, fue necesario trasladar únicamente el algoritmo en cuestión y realizar una replicación de datos para ejecutar el mismo, como lo haría dentro del software original. Debido a las clases y métodos propios de Delphi, se realizaron ajustes homologables dentro del algoritmo en C++, pero sin afectar su funcionamiento, ya que estos ajustes se debían principalmente al tipo de dato utilizado.

El manejo de memoria dentro del modelo de programación CUDA, presento una dificultad al momento de desarrollar el software, esto se logro corregir con ensayo y error, además de la constante realimentación de conocimientos.

CUARTA FASE: *VALIDACIÓN Y EXPERIMENTACIÓN*

Una vez se obtuvo el prototipo en una fase eficiente de ejecución, se inició el proceso de pruebas para el prototipo y para el algoritmo del software original, usando las herramientas de profiling de Nvidia Nsight y Smartbear AQTime 6. Inicialmente se realizaron pruebas con datos extraídos de proyectos de HOMOS, donde se evidenciaba un rendimiento bajo. Luego de esto se realizaron las pruebas con una cantidad mucho mayor de datos, para así tener una visión más concreta sobre el rendimiento sobre el prototipo realizado y el modelo de programación original de HOMOS. Estas pruebas permitieron realizar conclusiones y lineamientos sobre el procesamiento en paralelo en programas secuenciales, como lo es HOMOS 1.0.

Resultados alcanzados:

1. Se completaron las pruebas para ambos algoritmos implementados, tanto en DELPHI como en CUDA.
2. Se lograron obtener conclusiones sobre la mejora de rendimiento obtenido con el prototipo y el diseño desarrollado.
3. A partir de los resultados se pudieron realizar lineamientos sobre el procesamiento en paralelo y futuros proyectos aplicables dentro del ámbito de ingeniería de sistemas en la Universidad Industrial de Santander.

Principales dificultades:

Dentro de las principales dificultades se encuentra que el tiempo no fue suficiente para que el prototipo desarrollado pudiera aplicarse totalmente al software, así logrando una actualización total de HOMOS.

El modelo de programación de Delphi actualmente no cuenta con un sistema de compatibilidad absoluto con CUDA, para este debe hacerse uso de librerías, por ello como delante de las recomendaciones se sugiere que todo HOMOS sea trasladado a C o C++.

5. BÚSQUEDA DE OPORTUNIDADES DE PROCESAMIENTO CONCURRENTE

La búsqueda de concurrencia y oportunidades de paralelismo en HOMOS, inicia con un análisis general a la estructura del software y su funcionamiento.

El software HOMOS fue desarrollado en el entorno en programación Borland DELPHI 3.0 Client/Server, el cual se encuentra dividido en aproximadamente 88 unidades y *Forms*, escritas en lenguaje PASCAL, que componen el funcionamiento de HOMOS, dividiéndose en dos grandes categorías: **Unidades de manejo de interfaz gráfica"**, que se encargan de interactuar con los *Forms* que componen la interfaz gráfica de HOMOS, además de ser el enlace con las **Unidades lógicas**, que son aquellas que componen los algoritmos y manejo de datos para el funcionamiento de HOMOS. La figura 6 muestra a grandes rasgos cómo se dividen las unidades del sistema.

Figura 6. Unidades HOMOS. Fuente: Autores

Una unidad[13] es una colección de constantes, tipos de datos, variables, procedimientos y funciones. Cada unidad es casi como un programa Pascal independiente: tiene un cuerpo principal que es llamado antes que el programa comience y hace todo lo que sea necesario para su inicialización. En resumen una unidad es una librería de declaraciones y de rutinas que se pueden usar dentro de un programa, que permite hacer programas por partes que se compilan

[13] Unidades Delphi. [En línea]. Disponible en: <http://www.aabcomp.com/delphi/cap_4.htm>

separadamente.

Se concluye fácilmente que las unidades lógicas son aquellas que pueden poseer oportunidades de paralelismo y un grado de concurrencia, por tanto la búsqueda de concurrencia ahora se ve localizada hacia este tipo de unidades de HOMOS. Las unidades lógicas son aquellas que se encargan de partes específicas y rutinas de referencia para otras unidades, tales como las unidades que controlan las reglas del ambiente, repositorios, registros, etc. La unidad de control que se encuentra más alto en la jerarquía es USistema, esta unidad es llamada por FHomos, que es la unidad que interactúa con la interfaz gráfica y a su vez con el usuario, de este modo FHomos hace la llamada a USistema, sobre una acción específica, para que USistema sea quien llame a la o las unidades necesarias para ejecutar. En un escalón más abajo dentro de la jerarquía se encuentran unidades que controlan los aspectos donde se centra el funcionamiento y ejecución del software, tales como la unidad que controla los proyectos ***UProyecto***, la unidad que define el ambiente ***UAmbiente***, la unidad que controla el escenario ***UEscenario*** y la unidad que controla la simulación ***USimulador***.

Debió a la cantidad de unidades presentes en el software, sería un proceso muy largo analizar cada una de ellas y buscar concurrencia en sus estructuras, para que la búsqueda de un foco de ataque, se filtrara la búsqueda, teniendo en cuenta parámetros como mayor tiempo de ejecución y su relevancia dentro de la ejecución del software. Para ello se llegó la necesidad de hacer pruebas de rendimiento.

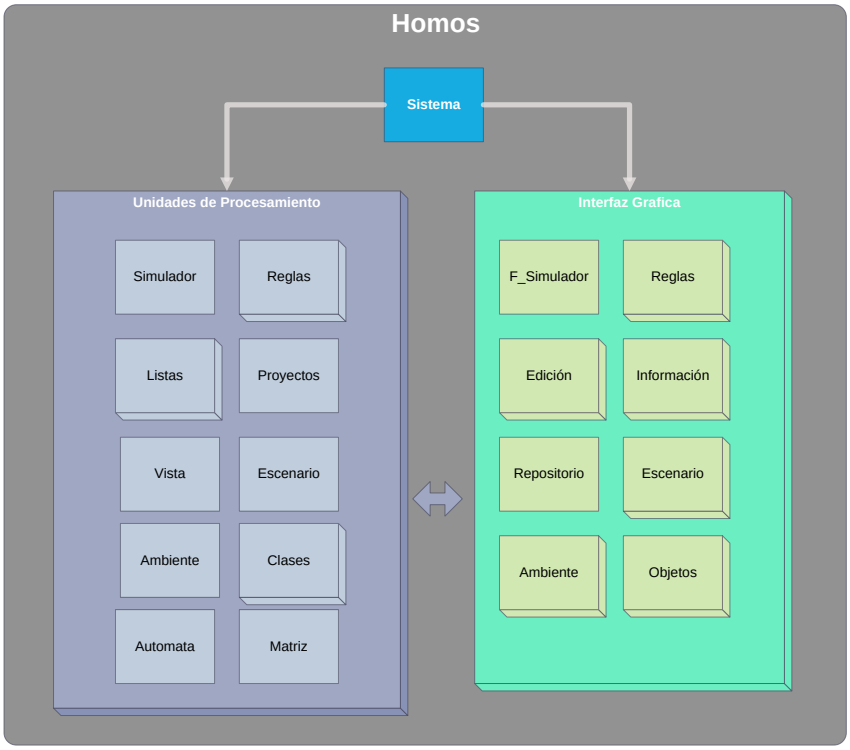


Figura 7. Arquitectura HOMOS. Fuente: Autores

HOMOS debido a que se encuentra escrito en Delphi 3.0 se requería un software de profiling que soportara dicha versión, en este proyecto se usó Automated AQtime 6 de Smartbear Software en una versión de prueba, que permitía toda su funcionalidad por un tiempo limitado.

Las pruebas se llevaron a cabo en una máquina virtual de windows XP Service Pack 2, usando el software VirtualBox de Oracle en el equipo presentados en la siguiente tabla:

Equipo de Computo
CPU: Intel Pentium Dual Core E5800 Frecuencia: 3.2 GHz Número de núcleos: 2
Memoria RAM: 4GB DDR3
GPU: Nvidia Geforce GT 610 2 GB DDR3 RAM
Sistema Operativo: Windows 7 Ultimate 32 bits

Tabla 1 Equipos Utilizados. Fuente: Autores.

Las pruebas se realizaron ejecutando en HOMOS dos proyectos predefinidos que tenían un evidente tiempo de ejecución lento, “ciclo del agua” y “Lobos y conejos”, así en cada uno de los equipos se realizaron los test y arrojaron resultados concluyentes.

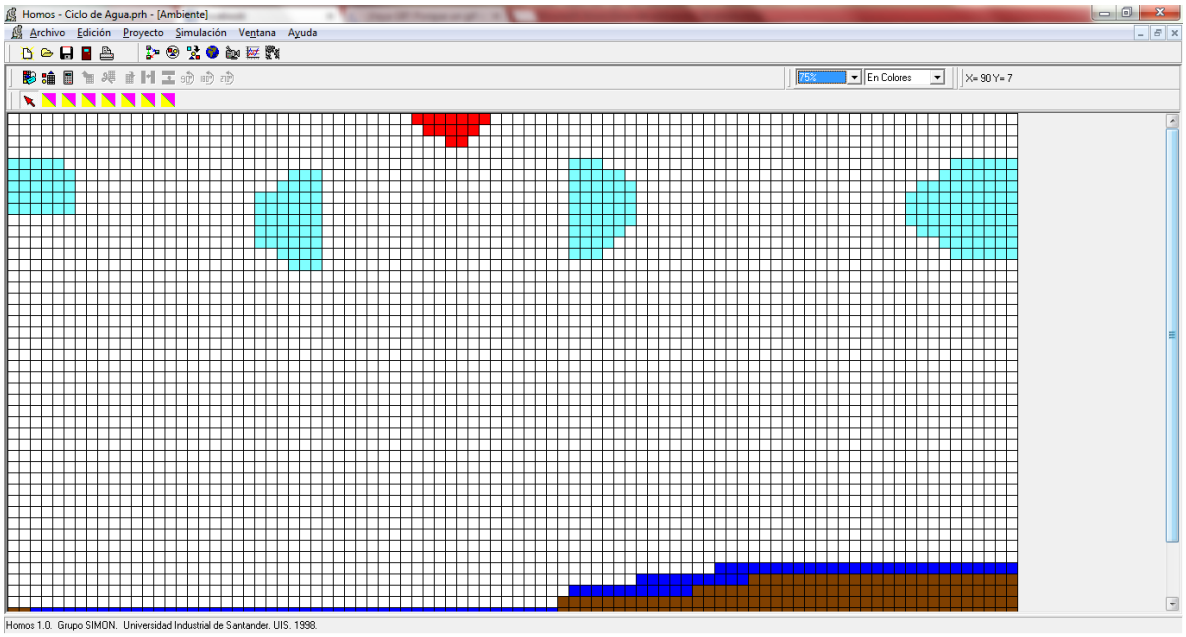


Figura 8. Ciclo de agua, HOMOS. Fuente: HOMOS 1.0

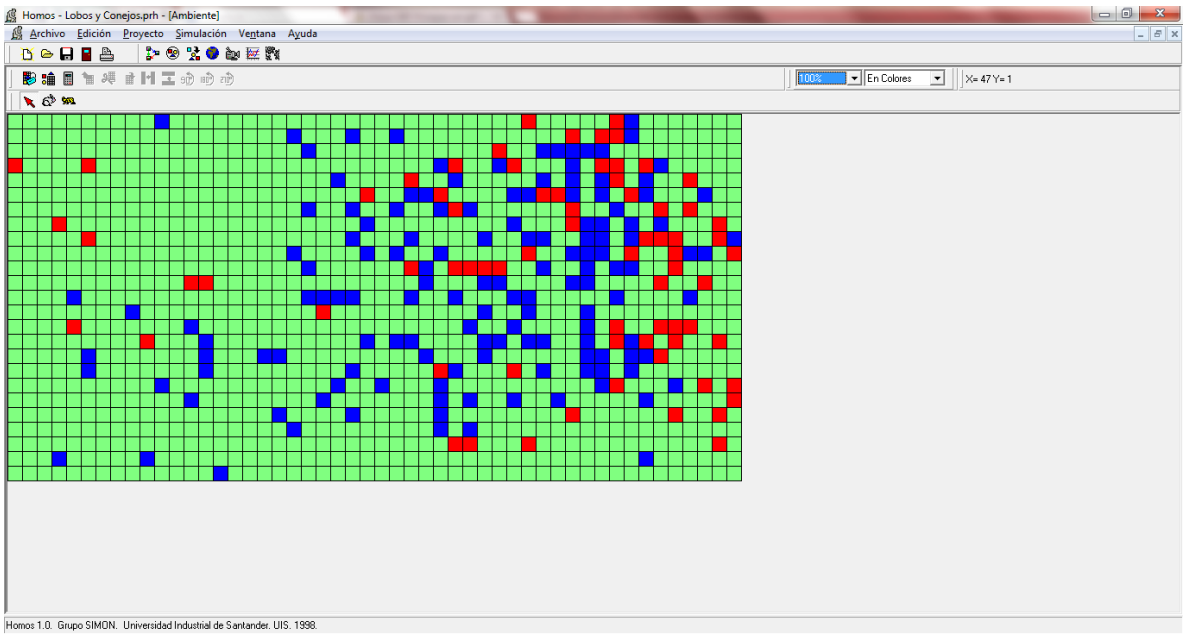
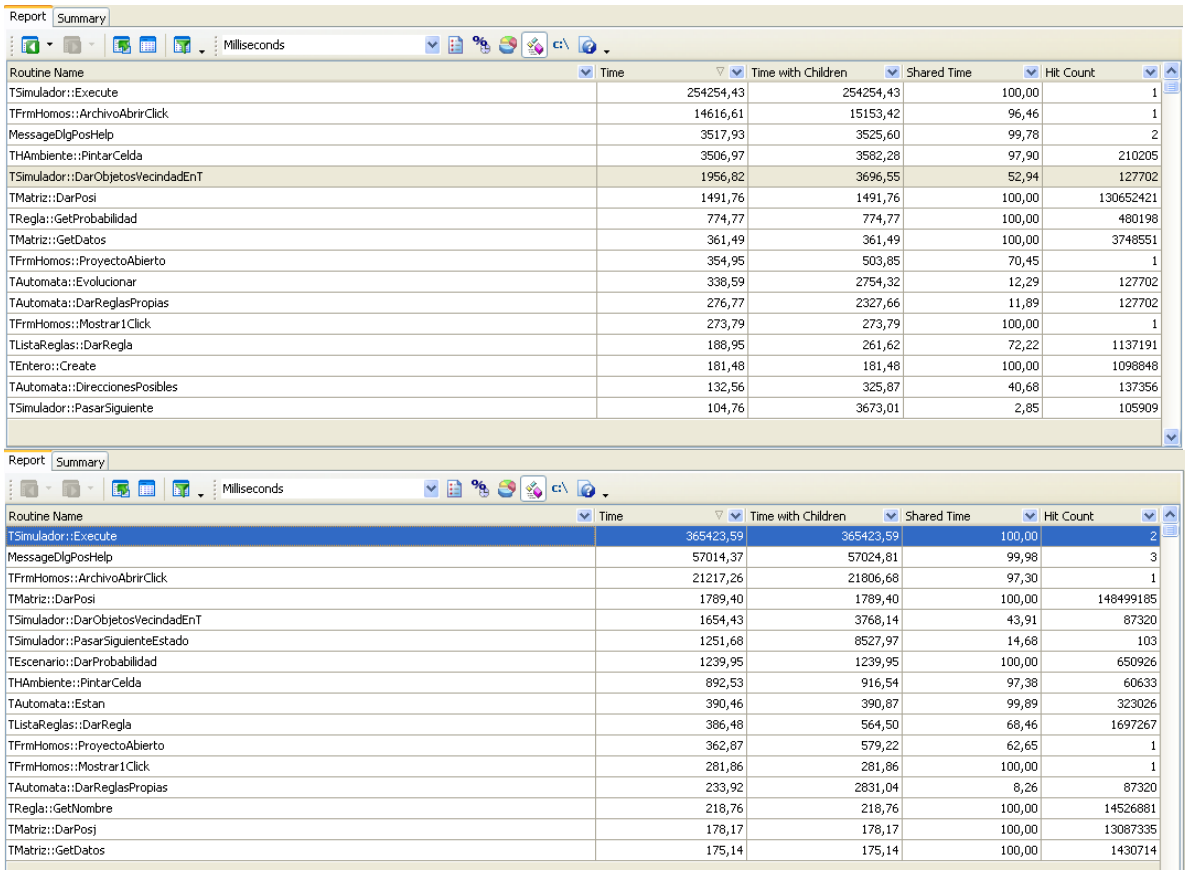


Figura 9. Lobos y conejos, HOMOS. Fuente: HOMOS 1.0



Report | Summary

Milliseconds

Routine Name	Time	Time with Children	Shared Time	Hit Count
TSimulador::Execute	254254,43	254254,43	100,00	1
TFrmHomos::ArchivoAbrirClick	14616,61	15153,42	96,46	1
MessageDlgPosHelp	3517,93	3525,60	99,78	2
THAmbiente::PintarCelda	3506,97	3582,28	97,90	210205
TSimulador::DarObjetosVecindadEnT	1956,82	3696,55	52,94	127702
TMatriz::DarPosi	1491,76	1491,76	100,00	130652421
TRegla::GetProbabilidad	774,77	774,77	100,00	480198
TMatriz::GetDatos	361,49	361,49	100,00	3748551
TFrmHomos::ProyectoAbierto	354,95	503,85	70,45	1
TAutomata::Evolucionar	338,59	2754,32	12,29	127702
TAutomata::DarReglasPropias	276,77	2327,66	11,89	127702
TFrmHomos::Mostrar1Click	273,79	273,79	100,00	1
TListaReglas::DarRegla	188,95	261,62	72,22	1137191
TEntero::Create	181,48	181,48	100,00	1098848
TAutomata::DireccionesPosibles	132,56	325,87	40,68	137356
TSimulador::PasarSiguiente	104,76	3673,01	2,85	105909

Report | Summary

Milliseconds

Routine Name	Time	Time with Children	Shared Time	Hit Count
TSimulador::Execute	365423,59	365423,59	100,00	2
MessageDlgPosHelp	57014,37	57024,81	99,98	3
TFrmHomos::ArchivoAbrirClick	21217,26	21806,68	97,30	1
TMatriz::DarPosi	1789,40	1789,40	100,00	148499185
TSimulador::DarObjetosVecindadEnT	1654,43	3768,14	43,91	87320
TSimulador::PasarSiguienteEstado	1251,68	8527,97	14,68	103
TEscenario::DarProbabilidad	1239,95	1239,95	100,00	650926
THAmbiente::PintarCelda	892,53	916,54	97,38	60633
TAutomata::Estan	390,46	390,87	99,89	323026
TListaReglas::DarRegla	386,48	564,50	68,46	1697267
TFrmHomos::ProyectoAbierto	362,87	579,22	62,65	1
TFrmHomos::Mostrar1Click	281,86	281,86	100,00	1
TAutomata::DarReglasPropias	233,92	2831,04	8,26	87320
TRegla::GetNombre	218,76	218,76	100,00	14526881
TMatriz::DarPosj	178,17	178,17	100,00	13087335
TMatriz::GetDatos	175,14	175,14	100,00	1430714

Figura 10. Pruebas AQTime. Fuente: Autores

No	Unit	Class	Method	RT-Sum *	% *	Percentage of Runtime
1	USimulador	TSimulador	Execute	**1.027 m**	57.85	
2	USimulador	TSimulador	PasarSiguienteEstado	**43.117 s**	40.49	
3	USimulador	TSimulador	DarObjetosVecindadEnT	**28.055 s**	26.34	
4	USimulador	TSimulador	PasarSiguiente	**7.218 s**	6.78	
5	UAutomata	TAutomata	Evolucionar	**6.554 s**	6.15	
6	CHAmbiente	THAmbiente	PintarCelda	**5.934 s**	5.57	
7	UAutomata	TAutomata	DarReglasPropias	**5.180 s**	4.86	
8	UAutomata	TAutomata	PuedeDireccion	**2.698 s**	2.53	
9	USimulador	TSimulador	ActualizarAmbiente	**1.733 s**	1.63	
10	FSimulador	TFrmSimulador	RetornarAmbiente	**1.733 s**	1.63	
11	UMatriz	TMatriz	DarPosi	**1.524 s**	1.43	
12	UListaReglas	TListaReglas	DarTipoRegla	**1.430 s**	1.34	

Tabla 2. Pruebas ProDelphi. Fuente: Angel Maria Valdes

Además de estas pruebas realizadas por los autores, se logró tener acceso a la documentación de HOMOS 2.0 donde se realizaron pruebas de rendimiento a

HOMOS 1.0.0.73, usando el software ProDelphi, que arrojó resultados igualmente concluyentes y similares a los obtenidos con AQtime. (Ver anexo C, resultados pruebas HOMOS)

Los resultados de las pruebas revelaron que la unidad que controla el Simulador, era aquella que tiene un mayor tiempo durante la ejecución del software, ya que como lo dice su nombre es quien realiza el proceso de simulación, así mismo las pruebas detallaron que dentro del Simulador existe un método que es aquel que lleva la mayor carga computacional. De este modo el rango de búsqueda se reduce aún más, tomando como foco este método llamado **Execute**, para así buscar las posibles opciones de paralelismo. También es posible observar que las demás unidades tienen un tiempo de ejecución en promedio similar, a excepción de la unidad Proyecto, que suele ser ligeramente mayor, pero sin acercarse al tiempo del simulador.

La estructura del método Execute revisando su código realiza tres acciones importantes:

1. Llama el método de la unidad que controla la interfaz del simulador, para que sea visible el estado del simulador.
2. Revisa si el estado del simulador no es “terminado”, entonces llama el método **PasarSiguienteEstado.**
3. Si el estado es “terminado” termina el método y la ejecución del simulador.

Ahora bien se sabe que el llamamiento al método *PasarSiguienteEstado* es aquel que esta generando mayor tiempo de ejecución, por tanto se traslado el análisis a este método. Ahora bien el método *PasarSiguienteEstado*, no posee un grado de concurrencia que soporte un procesamiento en paralelo, esto se debe a que el método hace llamamiento a otros 5 métodos, lo cual lo hace un método de control de ejecución dentro de la ejecución del simulador. De esos 5 métodos, sólo uno de

ellos posee la concurrencia necesaria para soportar un procesamiento en paralelo, este método se llama ***DarObjetoVecindadEnt***, el cual contiene el algoritmo que realizar la búsqueda de cada una de las vecindades de los objeto que cumpla el criterio de la regla y pueda realizar su ejecución, todo esto por medio de un ciclo anidado.

El algoritmo toma cada objeto de la simulación y calcula las ubicaciones de la vecindad inmediata, tomando como centro el objeto según la vecindad de Moore.

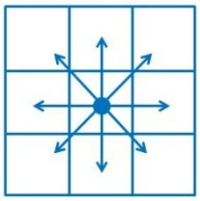


Figura 11. Vecindad de Moore. Fuente: Documentación HOMOS 1.0

Luego de esto para cada objeto de su vecindad, busca el intervalo de objetos válidos, luego para cada una de las celdas busca si se encuentra un objeto presente, de ser este el caso, el algoritmo indica que encontró un objeto en dicha ubicación.

Este algoritmo puede ser procesado en paralelo, ya que cada una de estas búsquedas, por objetos, se puede dividir en una unidad de procesamiento, para que solo se realicen, múltiples búsquedas secuenciales al mismo tiempo y no una sola que deba ejecutarse múltiples veces en una misma línea de tiempo. Los demás métodos no cumplían con el requisito, ya que estos solo actualizaban diferentes variables, pero no poseían oportunidades de aplicar paralelismo.

5.1 OPCIONES DE PARALELISMO

Para lograr interconectar diversos procesadores en un sistema se pueden aprovechar distintas técnicas, dentro de las cuales se pueden destacar los sistemas de memoria compartida y de memoria distribuida.
Algunas características de los sistemas de memoria compartida que caben resaltar son que están compuestos de procesadores y módulos de memoria interconectados, existe un direccionamiento de memoria común para todos los procesadores, la comunicación entre procesos es muy rápida y también escalan a un máximo del orden de 100 procesadores por problemas de rendimiento y costo.
En los sistemas de memoria distribuida cada procesador cuenta con su propia memoria, existe un sistema de interconexión que permite acceder a la memoria de los otros procesadores y pueden escalar al orden de miles de procesadores.

Para desarrollar aplicaciones que se ejecuten en múltiples procesadores se pueden utilizar varias alternativas, estas se diferencian en complejidad y escalabilidad.

5.1.1 OpenMP

Es una API especialmente diseñada para trabajar programación multiproceso con memoria compartida en múltiples plataformas. OpenMP permite implementar código en paralelo a aquellas aplicaciones desarrolladas en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Esta compuesta por un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que determinan el comportamiento en tiempo de ejecución.

Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones

paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras.

OpenMP se basa en el modelo fork-join, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en K hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join).

5.1.2 Multithread

Otra forma de plantear paralelismo es por medio de la ejecución de nuestras aplicaciones en diferentes y concurrentes hilos de ejecución (Threads). Para muchos lenguajes de programación existen distintas clases propias de cada una para la implementación de multithreading. Este paradigma arroja algunas desventajas importantes tales como:

- Los hilos pueden interferir entre sí al momento de compartir recursos de hardware como la memoria caché.
- Los tiempos de ejecución de un solo hilo no es mejorado pero si puede ser empeorado.
- Comparado con el multiprocesamiento el soporte de hardware para multihilo es más visible al software, por lo tanto requiere más cambios tanto en las aplicaciones como al sistema operativo.

5.1.3 Interfaz de Paso de Mensajes (MPI)

MPI es una especificación para programación de paso de mensajes, que proporciona una librería de funciones para C, C++ o Fortran que son empleadas en los programas para comunicar datos entre procesos. Es una técnica empleada en la programación en paralelo para aportar sincronización entre procesos y permitir la exclusión mutua.

La principal característica de MPI es que no precisa de memoria compartida, por lo que es muy importante en la programación de sistemas distribuidos.
MPI es el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida. La ventaja de MPI sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son portables y rápidos dado que MPI ha sido implementado para casi toda arquitectura de memoria distribuida y cada implementación de la biblioteca ha sido optimizada para el hardware en la cual se ejecuta.

5.1.4 Multiprocesamiento en GPU's

Hasta ahora se mencionaron alternativas de programación en paralelo usando los múltiples procesadores contenidos en CPU's. Pero evidentemente para obtener excelentes resultados en la aceleración de aplicaciones es necesario aprovechar el gran poder de procesamiento que poseen las GPU's. Para acelerar aplicaciones mediante el uso de tarjetas gráficas existen tres alternativas básicas; el uso de librerías, agregar directivas de OpenACC y/o utilizar lenguajes de programación orientados especializados.

5.1.5 Librerías CUDA[14]

Es posible acelerar aplicaciones realizando sencillos llamados a funciones comprendidas en ciertas librerías en este caso para GPU's NVIDIA.

- AmgX
- cuFFT
- CUblas
- CULA Tools

[14] Librerías CUDA. [En línea]. Disponible en: <https://developer.nvidia.com/gpu-accelerated-libraries>

- MAGMA
- ArrayFire
- Thrust

5.1.6 OpenACC[15]

Es un estándar de programación para computación en paralelo que permite a científicos y técnicos Fortran, C y los programadores de C++ implementar programación paralela en sistemas heterogéneos CPU/GPU. OpenACC permite utilizar directivas de compilación simples para identificar qué áreas de código pueden ser aceleradas, sin necesidad de modificar el mismo código. A través de la identificación de los bloques de código en paralelo, las directivas permiten al compilador que haga el trabajo detallado de la computación en el procesador encargado de la aceleración.

OpenACC fue desarrollado por el PGI, Cray y NVIDIA con el apoyo del CAPS Enterprise, las directivas OpenACC son una visión compartida de cómo las directivas pueden simplificar el modelo de programación para distintos tipos de aceleradores, donde cada proveedor se ha comprometido a apoyar un estándar de programación común.

5.1.7 Arquitecturas de hardware y software

5.1.7.1 CUDA

Esta es una plataforma capaz de soportar computación en paralelo y a su vez es un modelo de programación desarrollado por NVIDIA. Permite aumentos

[15] OpenACC. [En línea]. Disponible en: <http://www.openacc-standard.org/>

impresionantes en el rendimiento del procesamiento al aprovechar la potencia de la unidad de procesamiento de gráficos.

5.1.7.2 Stream

La tecnología Stream de ATI es un conjunto de hardware y software que permiten a los procesadores gráficos de AMD junto con la CPU, acelerar las aplicaciones aprovechando la alta capacidad de procesamiento de dichas tarjetas. Esto permite una plataforma más equilibrada, capaz de ejecutar exigentes tareas de computación más rápido.

5.1.7.3 OpenCL

Open Computing Language, (lenguaje de computación abierto) posee una interfaz y un lenguaje de programación. Dicho conjunto permite crear aplicaciones en paralelo a nivel de datos y de tareas que pueden ejecutarse tanto en CPU's como en GPU's. El lenguaje está basado en C.

Apple creó la especificación original y fue desarrollada en conjunto con AMD, IBM, Intel y NVIDIA. Apple le propuso al Grupo Khronos[16] convertir OpenCL en un estándar abierto y libre de derechos para convertirla en un estándar abierto y libre de derechos. El 16 de junio de 2008 Khronos creó el *Compute Working Group* para llevar a cabo el proceso de estandarización. En 2013 se publicó la versión 2.0.

OpenCL[17] es una solución libre a diferencia de CUDA de NVIDIA o STream de ATI las cuales intentan aprovechar la potencia de los procesadores gráficos para llevar

[16] Khronos [En línea]. Disponible en: <https://www.khronos.org/opencl/>

[17] OpenCL. [En línea]. Disponible en: <https://developer.nvidia.com/opencl>

a cabo tareas costosas en términos de ejecución repartidas entre la CPU y la GPU de cualquier tarjeta gráfica compatible. Al contrario de CUDA o Stream, OpenCL fue creado originalmente para que no dependa de algún hardware determinado. Además el hecho de que OpenCL sea abierto nos permite utilizarlo en múltiples plataformas y distintos sistemas operativos.

5.2 SELECCIÓN DE ALTERNATIVA

Teniendo en cuenta todas las soluciones estudiadas era necesario optar por una de ellas que nos pudiera brindar importantes ventajas principalmente en el aspecto de la aceleración de HOMOS. Inicialmente se analizó la posibilidad del manejo de multihilos propias del lenguaje Delphi, incluso en la implementación original de HOMOS se trabajó con esta clase thread, intentando dividir tareas en distintos hilos de ejecución. Por este motivo decidimos enfocar nuestro proyecto hacia opciones más potentes desde el punto de vista del multiprocesamiento, por ello se decidió poner como objetivo de estudio la posibilidad de acelerar las tareas ejecutadas por la aplicación aprovechando el alto rendimiento que nos pueden brindar las GPU. Como se ha hablado a lo largo del documento el SC3 posee importantes recursos en materia de procesamiento con GPU's de NVIDIA y presenta además una interesante alternativa de procesamiento en paralelo para conseguir nuestros objetivos planteados en este proyecto. Al decidir trabajar con tarjetas gráficas de NVIDIA se nos plantean las posibilidades de trabajar con CUDA o hacer uso de una librería llamada Thrust[18], el trabajo con la librería es un proceso que facilita la escritura del código quizá haciendo un poco más sencillas las tareas de paralelización. Pero dentro de los propósitos personales estaba el poder aprovechar la investigación realizada para aprender acerca de un lenguaje de programación bastante poderoso y además en su "estado puro", que soportara

[18] Thrust. [En línea]. Disponible en: <http://www.ib.cnea.gov.ar/~gpgpu/2013/Clases/clase5_thrust.pdf>

procesos paralelizados; es por ello entonces que se decidió finalmente enfocarnos en desarrollar este proyecto con CUDA para así aprovechar los recursos que nos puede facilitar la Universidad y así mismo aprender mucho más acerca del área de supercomputación basada en GPU's desarrolladas por NVIDIA.

6. DISEÑO DEL PROTOTIPO DEL SIMULADOR DE BUSQUEDA

El prototipo del algoritmo de búsqueda del método "*DarVecindadEnt*", fue pensado para que funcione con el modelo de programación CUDA, este modelo requiere que el código está desarrollado en C o C++, por tanto es necesario que trasladar de Pascal a C o C++ el nuevo algoritmo y así luego implementar CUDA. Para ello en el diseño preliminar se planteó separar el algoritmo del código original, y replicar los datos que son utilizados por el mismo, para que del mismo modo, los algoritmos implementados en C o C++, y CUDA, tenga la misma base de información y llegar al mismo resultado.

El algoritmo original realiza una búsqueda dentro del ambiente del proyecto en ejecución, en una vecindad de 8 celdas, se busca el objeto válido para la ejecución, vale la pena aclarar que el número de celdas se ve limitado por la malla cuadrada con la que fue desarrollado HOMOS 1.0, pero como demostró el prototipo de HOMOS 2.0, esta misma puede llegar a extenderse de muchas más formas y tener vecindades de n posibilidades, por tanto se comprueba que, una mejora en el rendimiento del simulador, puede ser de gran ayuda para aliviar la carga computacional, cuando se tengan grandes cantidades de datos. Por tanto se planteó un diseño del algoritmo en CUDA de la siguiente manera:

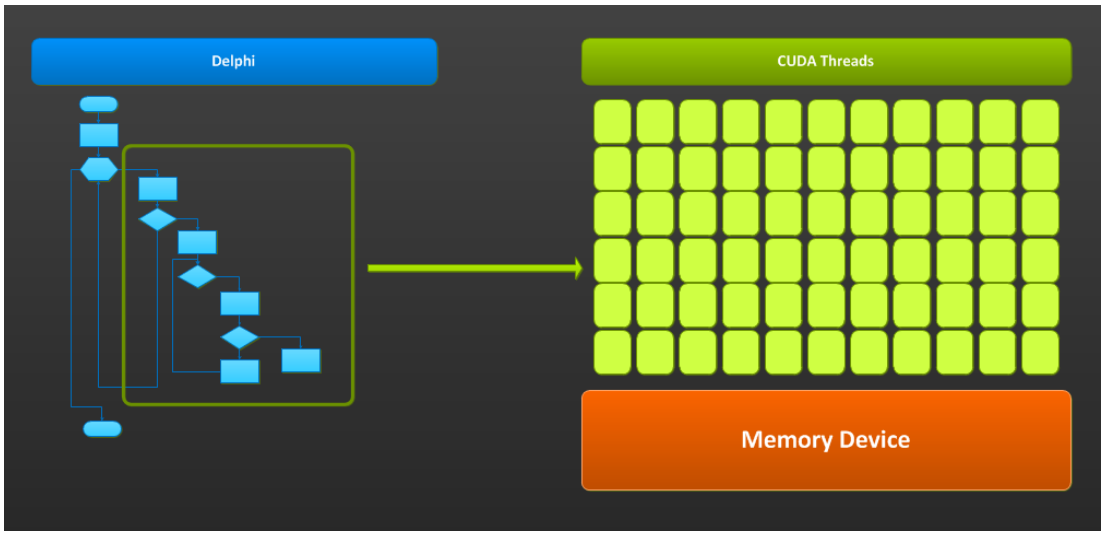


Figura 12. Ejecución Algoritmo CUDA. Fuente: Autores

Donde al lado izquierdo de la figura 9, se puede apreciar el ciclo anidado de ***8*n*** búsquedas por cada objeto, en contraste con el algoritmo del lado derecho, que solo realiza una búsqueda, pero este se realiza por separado, por múltiples CUDA threads, una búsqueda en paralelo, y encontrara de una manera más eficiente la igualdad.

Además de esto gracias a la arquitectura de las GPU es posible utilizar gran cantidad de datos sin importar su dimensión, ya que esta puede trabajarse hasta en tres dimensiones.

7. IMPLEMENTACIÓN DEL PROTOTIPO DEL SIMULADOR DE BUSQUEDA

La implementación se divide en 3 fases, las cuales se alimentan entre ellas para lograr que se obtenga un código limpio, para poder ejecutar pruebas de rendimiento, que permitan comparar el algoritmo implementado en CUDA con el algoritmo original en DELPHI.

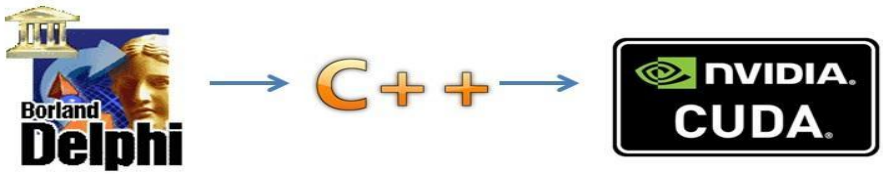


Figura 13. Modelo de Implementación. Fuente: Autores

La primera fase consistió en aislar el algoritmo del método, en una nueva unidad, en la cual se replicaron los datos usados en la ejecución de HOMOS, se verificó el correcto funcionamiento del código y se continuó a la siguiente fase.

La segunda fase prosiguió con la programación del algoritmo en C++; se escogió C++, debido a que es más a fin con DELPHI que C, además se escogió debido a que se logró obtener la licencia de Visual Studio 2012, por medio del convenio de la Universidad Industrial de Santander con Microsoft. De este modo se pudo contar con una herramienta robusta para el desarrollo del código, por último se usaron los mismos datos que el algoritmo de DELPHI y se comprobó su correcta ejecución.

La fase final de la implementación, inició con la programación del algoritmo en CUDA, haciendo que este trabajara en cada CUDA thread, se utilizó la versión de

CUDA 5.5 en una tarjeta de video Geforce GT 610 de arquitectura Fermi, luego el algoritmo se programó y se comprobó su correcta ejecución. Por último se amplió la cantidad de datos en cada implementación, para que asi sea más evidente al momento de comparar su ejecución, la diferencia de rendimiento. Cabe aclarar que en la fase final de la implementación, se tuvo especial cuidado con el manejo de memoria entre el dispositivo y la CPU.

8. LINEAMIENTOS

La implementación de programación en paralelo en un software completamente desarrollado en cualquier lenguaje de programación requiere la planificación exhaustiva de muchos detalles que se deben tener en cuenta en el momento de elegir la técnica que nos proporcione mayores ventajas en términos de supercomputación.

En esta parte del documento se mencionan las recomendaciones que se deben considerar en futuros proyectos relacionados o afines a este. Todos estos puntos nacen de la experiencia que se obtuvo a partir de la investigación realizada y con el posterior desarrollo de este proyecto.

Para trabajar en un proyecto que involucre programación en paralelo es recomendable tener en cuenta los siguientes factores:

- **Verificar viabilidad:** Si la idea es trabajar en la aceleración de un software, es decir, teniendo el código se pretende hacer de este uno mucho más eficiente; lo mejor sería revisar si realmente es conveniente aplicar métodos que involucren paralelismo. Para ello es indispensable trabajar con el mayor número de datos posibles y así determinar si se producen desbordes u otro tipo de errores debido a la poca capacidad de procesamiento. Si este es el caso, podríamos decir que se necesita un nivel de procesamiento mucho más elevado y de dicha forma se justificaría el uso de múltiples núcleos que se encarguen de las tareas por separado.

- **Búsqueda de oportunidades de procesamiento concurrente:** Cuando la necesidad de aplicar metodología paralela está clara, se procede a buscar

aquellos procesos que nos generan grandes consumos de tiempo en su ejecución. Al determinar estos métodos se pueden identificar las áreas que se van a atacar. Se pueden hacer distintas pruebas al ejecutar el software para observar y analizar el tiempo que gastan los distintos procesos realizados por el programa y de esta manera tomar las decisiones necesarias para el desarrollo del proyecto.

- **Analizar dependencias:** Al tener identificadas las zonas donde se presenta la concurrencia en el código es importante estudiar las dependencias dentro del mismo. De acuerdo al paradigma de programación que se esté manejando el grado de dependencia entre métodos será mayor o menor. A lo largo del documento se ha planteado que el lenguaje en el cual se desarrolló HOMOS, presenta un alto grado de dependencias entre clases, ya que Delphi es un lenguaje de programación orientado a objetos y además permite al programador facilidades para manejar la parte gráfica; por lo tanto sumando las distintas clases y los forms utilizados resulta un software bastante dependiente entre sí. Este factor es determinante al momento de paralelizar código ya que se debe contar con aquellos métodos que pueden estar involucrados y a partir de allí se deben tomar acciones para no afectar el flujo normal del programa.

- **Elección de alternativa:** Para finalizar el proyecto con éxito y alcanzando los objetivos propuestos en su totalidad, un punto clave es elegir la forma de implementar supercomputación de tal manera que se logre obtener los mejores resultados en cuanto a la diferencia en la aceleración del código. Esto depende de diversos aspectos que anteriormente fueron mencionados y dónde además cabe resaltar que se debe tener en cuenta el hecho de conocer el lenguaje de programación a utilizar, porque si no es así se tiene que considerar la curva de aprendizaje requerida para familiarizarse con esta tecnología. Tal detalle representaría una demora importante en el

desarrollo del proyecto por tanto implica un detenido análisis para determinar si afecta o no a lo propuesto.

- **Realizar pruebas:** Cuando se complete el proceso obviamente debe ser constatado con resultados que aprueben la mejora en el rendimiento del software, ya sea ejecutándose totalmente o simplemente el método en el cual se decidió enfocar el esfuerzo. En este proyecto de usaron distintas arquitecturas de tarjetas gráfica NVIDIA para realizar las pruebas, las cuales arrojaron ciertos resultados y a partir de ellos sacar las conclusiones necesarias para cumplir con los objetivos del proyecto. A raíz de este tipo de pruebas se determinará básicamente si se logró aumentar la capacidad de procesamiento requerida por los procesos ejecutados por la aplicación.

9. PRUEBAS DEL PROTOTIPO DEL SIMULADOR

El prototipo se puso a prueba en diferentes configuraciones de software y hardware, en la siguiente tabla se muestran los equipos usados para hacer las pruebas de rendimiento.

EQUIPO 1	**FICOMACO**	**GUANE-2**
CPU: Intel Pentium Dual Core E5800 Frecuencia: 3.2 GHz Número de núcleos: 2 Número de hilos: 2	**CPU:** Intel Core 2 Q9450 Frecuencia: 2.66 GHz Número de núcleos: 4 Número de hilos: 4	**CPU:** Intel Xeon E5640 Frecuencia: 2.67 GHz Número de núcleos: 4 Número de hilos: 12
Memoria RAM: 4 GB	**Memoria RAM:** 4 GB	**Memoria RAM:** 102.5 GB
GPU: Nvidia Geforce GT 610 2 GB DDR3 RAM	**GPU:** Nvidia Quadro FX 570 256 MB DDR2 RAM	**GPU:** Nvidia Tesla S2050 12 GB DDR5 RAM
Sistema Operativo: Windows 7 Ultimate 32 bits	**Sistema Operativo:** Debian 6.0.9	**Sistema Operativo:** Debian 6.0.4

Tabla 3. Equipos usados para pruebas de prototipo. Fuente: Autores

Siguiente a esto se realizaron pruebas del algoritmo en DELPHI, para realizar una comparación de rendimiento y poder tener un valor aproximado de la mejora en el rendimiento en contraste con el prototipo en CUDA. Para las pruebas se tomaron

varias muestras variando el número de búsquedas que se debían realizar llegando a un límite de 2000.

En la siguiente grafica puede verse una comparación de rendimiento del algoritmo de búsqueda extraído de HOMOS 1.0 contra el algoritmo de CUDA en los equipos usados.

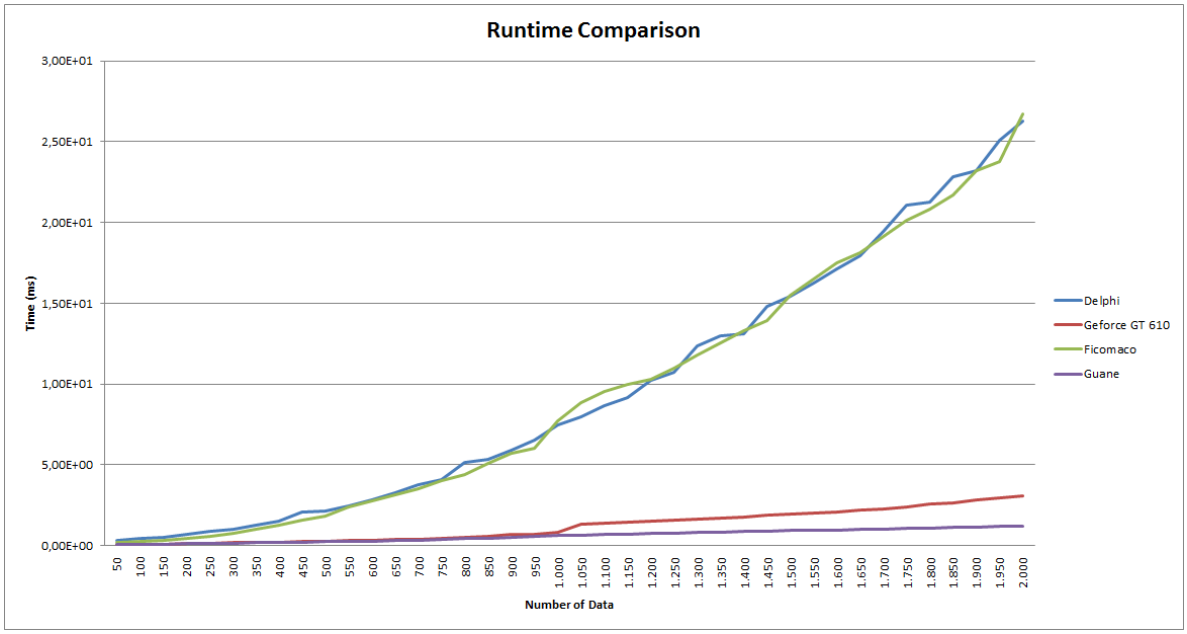


Figura 14. Comparación en tiempos de ejecución. Fuente: Autores

Los resultados mostraron que el algoritmo obtiene una sustancial mejora en el tiempo de ejecución usando la misma cantidad de datos y que el modelo de programación usado, puede ser una opción viable para una futura versión de HOMOS.

10. LIMITACIONES DEL PROYECTO

- El prototipo desarrollado en este proyecto se realizo de forma exploratoria, por lo cual no se consolido una nueva versión de HOMOS. Se desarrollo un prototipo en CUDA basado en el motor de simulación de HOMOS, específicamente el método encargado de la búsqueda de objetos en cada una de las vecindades del ambiente.
- El uso de CUDA es exclusivo de equipos que posean una GPU nvidia, que soporten dicho modelo de programación, por tanto en equipos con tarjetas de otros fabricantes no funcionará.
- Debido al alto grado de dependencia que existe en la arquitectura de HOMOS, se requeriría para implementar el prototipo del simulador de búsquedas que se cree un puente de comunicación entre el prototipo escrito en CUDA y el software original desarrollado en Delphi 3, esto puede lograrse por medio de la creación de una librería que posea el prototipo escrito en CUDA y que esta misma sea invocada por el software original y retorne los resultados esperados.
- El rendimiento del algoritmo implementado en el prototipo presenta una mejora sustancial en el rendimiento del mismo, pero a su vez se ve limitado por el tiempo que tarda CUDA en hacer copiado de memoria de la CPU a la GPU. El prototipo desarrollado en este proyecto se realizo en la versión 5.5 de CUDA, pero en el momento de su culminación, se libero la versión 6.0 que corrige el error de memoria, pero solo se encuentra disponible para GPU tesla, por lo cual no se implemento en el prototipo.
- La integración de diversas tecnologías sobre HOMOS, deberá ser de manera modular, buscando aquellos procesos y nodos del software que puedan usarse para tales fines, por ello la arquitectura de HOMOS puede presentar dificultades en la comunicación de datos.

11. RECOMENDACIONES

Para futuros desarrollos relacionados con la programación en paralelo es importante tener en cuenta algunas recomendaciones esenciales. Para implementar metodologías de procesamiento es necesario que el software a optimizar, en este caso HOMOS, se encuentre estructurado en módulos funcionales, que permitan su modificación por separado y así se quiere modificar o mejorar, se puede lograr sin necesidad de alterar los demás componentes del mismo. Esto ayudara a que futuras investigaciones y posibles mejoras a HOMOS puedan hacerse de manera incremental. Se recomienda establecer adecuadamente la metodología con la cual se decide implementar procesamiento en paralelo dentro de un proyecto considerando los recursos que se tienen, en términos de capacidad computacional, manejo de datos y la forma en que los distintos hilos de ejecución se distribuyen las tareas en cada núcleo de la unidad encargada del procesamiento. Si se trata de utilizar dispositivos aceleradores, se debe estudiar la arquitectura que se acomode mejor a la cantidad de datos que se trabaje y aquella que proporcione resultados más óptimos en tiempos de ejecución del software.

El prototipo desarrollado en este proyecto no se incluyo en el software original de HOMOS, es decir no sé desarrollo una nueva versión , esto debido al tiempo que requiere su implementación en el software, el cual debe hacerse por medio de un puente entre los dos lenguajes, PASCAL y C++, esto requiere que el código del simulador de HOMOS invoque la rutina del prototipo y transfiera los datos al mismo, la segunda razón por la cual no se implemento fue debido a que el prototipo del simulador en CUDA logra sintetizar el modelo de procesamiento en paralelo que refleja CUDA y logra demostrar la mejora de rendimiento en contraste con el algoritmo secuencial, la implementación en HOMOS se deja como propuesta para futuros proyectos que deseen implementarla en versiones futuras

de HOMOS. En una versión futura de HOMOS se recomienda la implementación del prototipo desarrollado por el ingeniero Ángel Valdez, el cual permite que el ambiente de simulación no este limitado a las 8 vecindades de HOMOS actual, esto a su vez permitirá que la implementación del prototipo desarrollado en este proyecto sea mas provechosa, ya que gracias al mismo podrían pensarse en vecindades de tamaño N, e incluso de manera ambiciosa puede pensarse en usar ambientes de simulación en tres dimensiones.

En este proyecto se escogió CUDA debido a su ya conocido rendimiento frente a la computación secuencial y mirando el futuro de la computación mediante GPU, pero en distintas partes del software no seria posible aplicar dicha tecnología, por ello debe estudiarse muy bien el área atacar y el método propuesto, para así obtener el máximo provecho de los recursos hardware con los que se cuente, sin importar de que tipo sean, logrando así que en futuras investigaciones se obtenga un mayor grado de portabilidad y escalabilidad.

12. CONCLUSIONES

La arquitectura del software HOMOS, demostró ser un claro caso de una agrupación de objetos y clases que trabajan conjuntamente para el funcionamiento del software, dividiéndose en distintos tipos de unidades especializadas en una tarea concreta, que son llamadas entre ellas mismas para ejecutar los procesos deseados, además de esto su grado de dependencia es un factor influyente al momento de encontrar procesos que se puedan extraer y optimizar. Esto llevo a que las posibilidades de implementar procesos concurrentes se vean influenciadas por la estructura de los objetos o unidades, haciendo que la intervención y mejoras al software tengan que hacerse necesariamente en módulos separados y específicos. En este proyecto se desarrolló una versión del simulador de búsquedas en CUDA, pero no se implementó en el software original. Se puede extraer de la investigación y procesos realizados; que en aquellos procesos que necesiten de grandes iteraciones de cálculo, es provechoso el uso de procesamiento en paralelo por medio de GPU.

Durante el desarrollo de este proyecto se estudiaron distintos modelos para la implementación de procesamiento en el software, el cual luego de distintas investigaciones y referencias, se escogió el uso de CUDA gracias a distintos factores tales como; la capacidad computacional que brindan las GPU para el manejo de una gran cantidad de procesos y datos en distintos módulos del dispositivo. Además de esto la tecnología CUDA permite que los procesos ejecutados de forma paralela, no sufran de conflictos de coherencia y el uso de memoria no afecte los demás procesos del software HOMOS, ya que la GPU utiliza su memoria interna para dichos procesos. Por último se eligió CUDA pensando en la posibilidad de que en futuras versiones sea necesaria una metodología de alto rendimiento, al llegar a necesitarse gran capacidad de cómputo en el simulador de HOMOS.

La implementación de la lógica en paralelo en un software desarrollado completamente en Delphi es un proceso que requiere gran esfuerzo, dedicación y constante investigación ya que la idea de utilizar CUDA presenta un gran reto porque se hace necesario transcribir código a lenguaje C/C++ para poder hacer uso de esta tecnología NVIDIA. Debido a esta necesidad y la amplia dependencia entre clases del software, tuvo que aislarse el método esencial para la ejecución de la simulación dentro del programa y replicar los datos que requería dicho proceso para su funcionamiento normal, de esta forma se logró optimizar parte del código que da muestras claras de los importantes logros que se pueden conseguir en materia de aceleración de aplicaciones por medio de las GPU, lo cual demuestra que programas secuenciales pueden obtener mejoras de rendimiento a través de metodologías en paralelo sin necesidad de cambiar totalmente su estructura ni hacer cambios radicales.

Al trabajar con la tecnología NVIDIA CUDA se debe tener en cuenta que no cualquier equipo está en capacidad de ejecutar este tipo de aplicaciones, por lo tanto se debe evaluar si es de provecho manejar estas soluciones de acuerdo a los usuarios que en últimas son los que trabajarán con el programa y son ellos los que deben contar con el hardware necesario para hacer uso de este, aunque en la actualidad esto se ha convertido en algo mas común, debido a que actualmente los computadores personales traen tarjetas de video NVIDIA integradas que soportan el uso de CUDA, pero es una restricción al uso del software que debe considerarse a futuro y que permite la posibilidad de pensar en software especializado.

Al finalizar la fase de desarrollo de este proyecto se iniciaron las pruebas que permitirían saber qué tipo de GPU proporcionan los mejores resultados de acuerdo a la cantidad de datos con los que se trabajaba. Las pruebas se realizaron tomando como control el algoritmo original extraído de HOMOS, el cual

fue ejecutado con la misma cantidad de datos para realizar una comparación uno a uno con el prototipo del simulador desarrollado, ejecutado un equipo dedicado al cálculo científico como lo es el GUANE con una tarjeta TESLA, un equipo promedio que posee una tarjeta gráfica G-FORCE de gama baja y un equipo perteneciente al grupo SC3 llamado FICOMACO con una tarjeta QUADRO. Finalmente al observar los resultados de este estudio se determinó que el rendimiento del GUANE, obtuvo una aceleración aproximadamente de 520% con respecto al simulador original en Delphi. De igual forma se comprobó para la tarjeta G-FORCE, el rendimiento fue en promedio 500% superior en contraste al algoritmo original; por el contrario no se obtuvo lo mismo en el FICOMACO, ya que éste presentó un rendimiento bastante cercano al algoritmo en DELPHI (ver anexo E, H), se pudo concluir que el rendimiento del equipo FICOMACO no presento una aceleración en frente del algoritmo original debido a su hardware, ya que la GPU de este equipo posee una arquitectura especializada para el diseño y no el computo. Se puede establecer que para obtener un rendimiento superior en computación GPU por medio de la metodología CUDA, es necesario contar con el recurso computacional que lo soporte de forma adecuada, esto lo hace al mismo tiempo una limitación del mismo en términos de portabilidad.

Es posible implementar diferentes modelos de paralelismo en HOMOS, como OpenMP, Ompss, OpenCL, etc. Pero para que sea posible implementar estos modelos se requeriría como se dijo anteriormente trabajar sobre diferentes módulos separados del software. HOMOS puede llegar a tener un entorno de simulación mucho mas basto, y como se demostró en el prototipo de HOMOS 2.0 las mallas donde los objetos se mueven pueden tener infinitas formas y número de vecinos, lo cual aprovecharía en gran medida el prototipo desarrollado en este proyecto, se podría incluso pensar en implementar más de dos dimensiones y obtener ambientes de simulación más reales, pero para se repite la recomendación de una nueva versión.

Con la investigación realizada para lograr los objetivos propuestos en este proyecto se logró proponer los lineamientos para la implementación de programación en paralelo en un software o para proyectos afines a esta área, que permiten ayudar a realizar proyectos simulares donde se busque optimizar software por medio de procesamiento en paralelo. Estos puntos permitirán tener alguna idea de los factores a tener en cuenta en el momento que se decida trabajar con este paradigma de programación. La propuesta de estos parámetros era un punto clave dentro de lo esperado al finalizar el desarrollo de este proyecto.

Por último se logró demostrar que sin necesidad de cambiar bruscamente la estructura de un software puede lograrse una mejora significativa en su rendimiento, lo cual permite afirmar que el procesamiento en paralelo y la computación en GPU, hacen parte del futuro de la computación, tanto en software científico como cotidiano.

BIBLIOGRAFÍA

1. Cantú, Marco. La Biblia de Delphi 7. Anaya Multimedia 2003
2. Gupta, Nitin. CUDA Array in CUDA, How to use CUDA Array in CUDA [En línea] <http://cuda-programming.blogspot.com/2013/02/cuda-array-in-cuda-how-to-use-cuda.htmll> Consultado en [28/03/14]
3. Gupta, Nitin. Texture Memory in CUDA What is texture memory in CUDA Programming [En línea] <http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html> Consultado en [24/03/14]
4. Jason, Sanders. Kantdrot, Edward. CUDA by Example, An introduction to General-Purpose GPU Programming. Boston: Pearson Education, 2011.
5. Jesús R. Peinado, Jesús M. García. Autómatas celulares [En línea] <http://www.enelnombredetux.com/project.php?project=autcel> Consultado en [22/03/14]
6. Rennich, Steve. CUDA C/C++ Streams and Concurrency [En línea] <http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf> Consultado en [24/03/14]
7. Rodríguez, Miguel. DELPHI - Programación Orientada a Objetos. Entorno Visual [En línea] <http://coba.dc.fi.udc.es/~penabad/delphi6.pdf> Consultado en [29/03/14]
8. Velthuis, Rudy. Using C++ Objects in Delphi. [En línea] <http://rvelthuis.de/articles/articles-cppobjs.html> Consultado en [29/03/14]
9. W.Hwu, wen-mei. GPU Computing gems, Jade Edition. Waltham: Elsevier, Inc, 2012.

ANEXOS

A. DIAGRAMA DE DEPENDENCIAS VISTA GENERAL

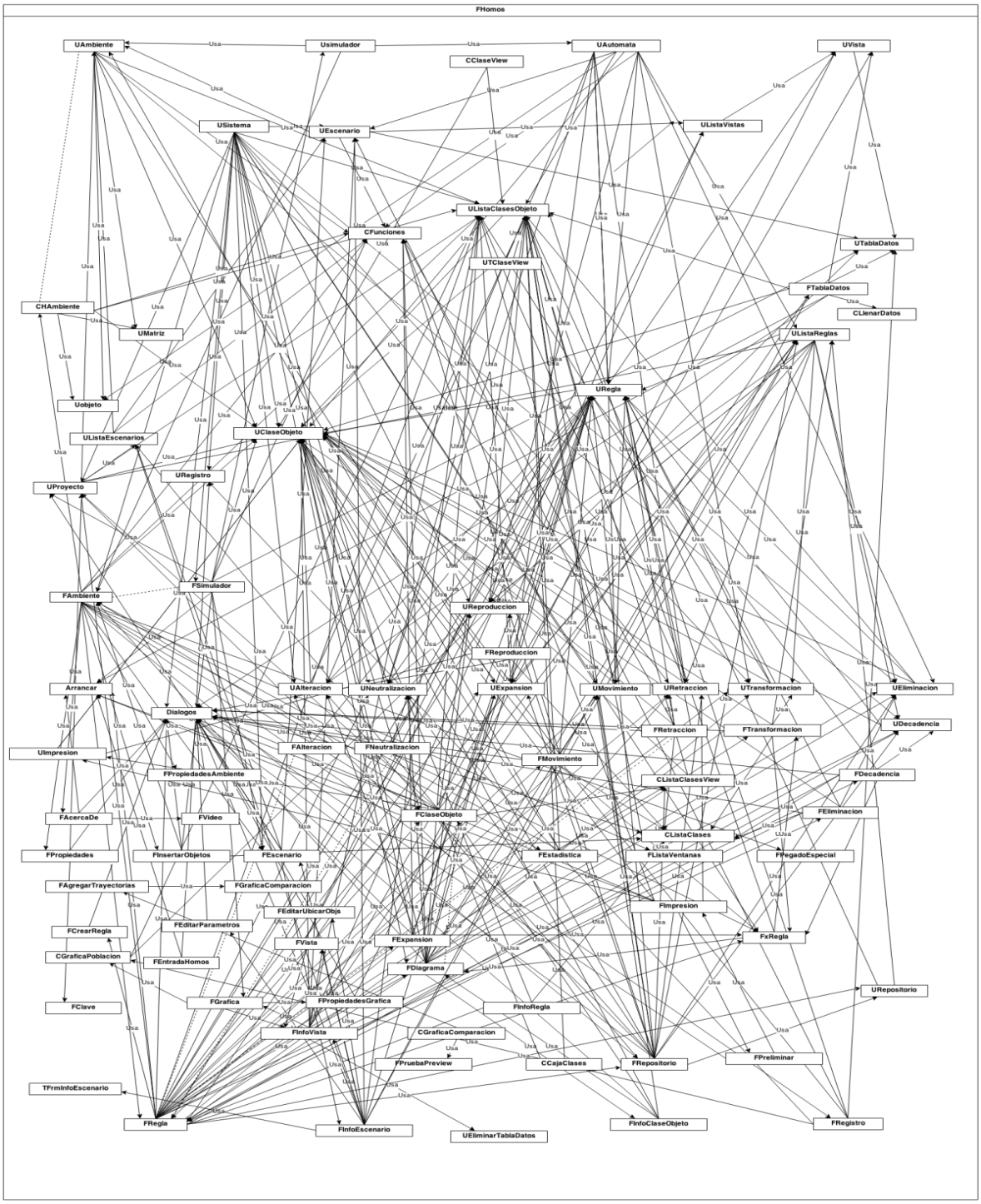


DIAGRAMA DE DEPENDENCIAS VISTA 1

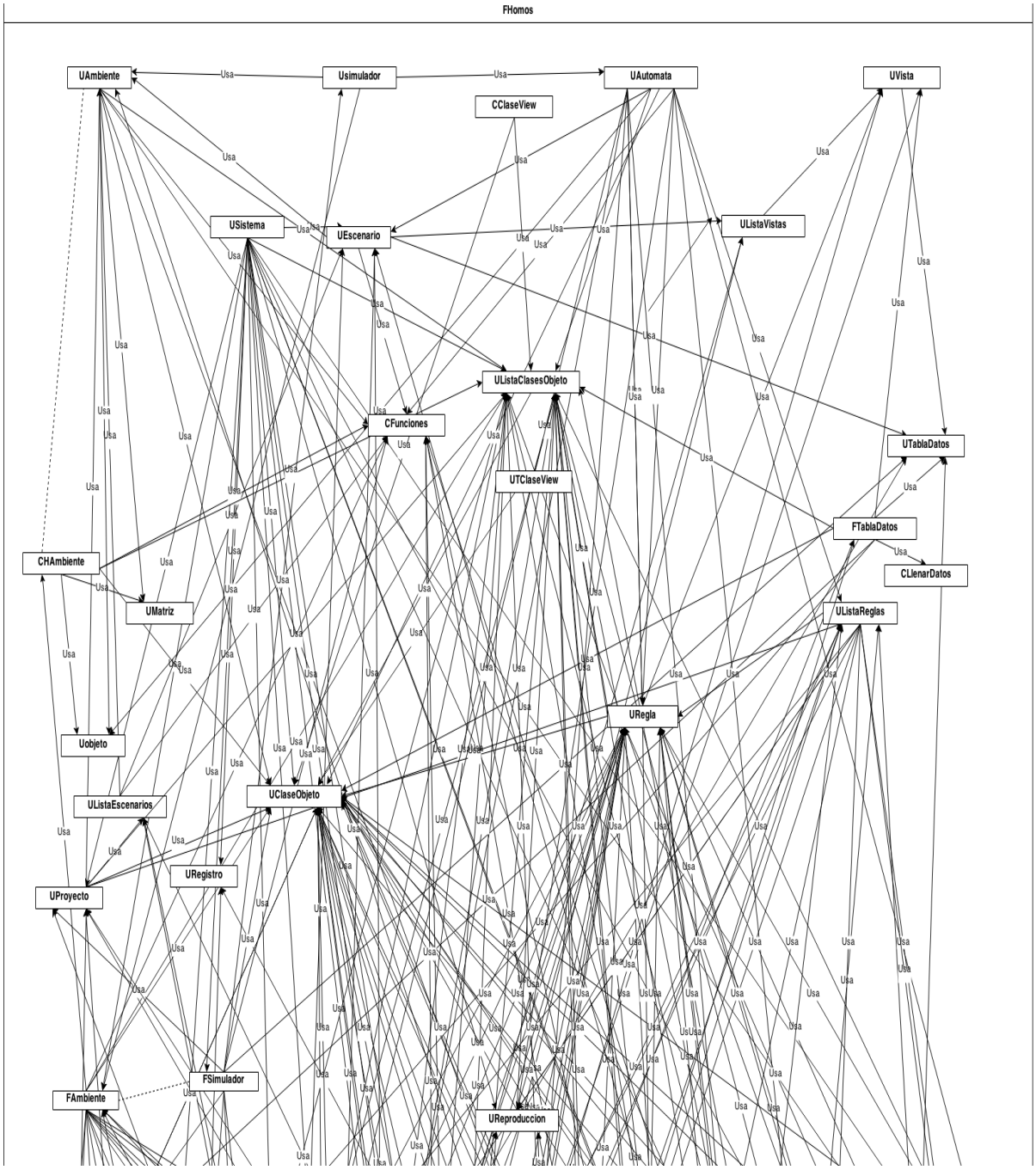
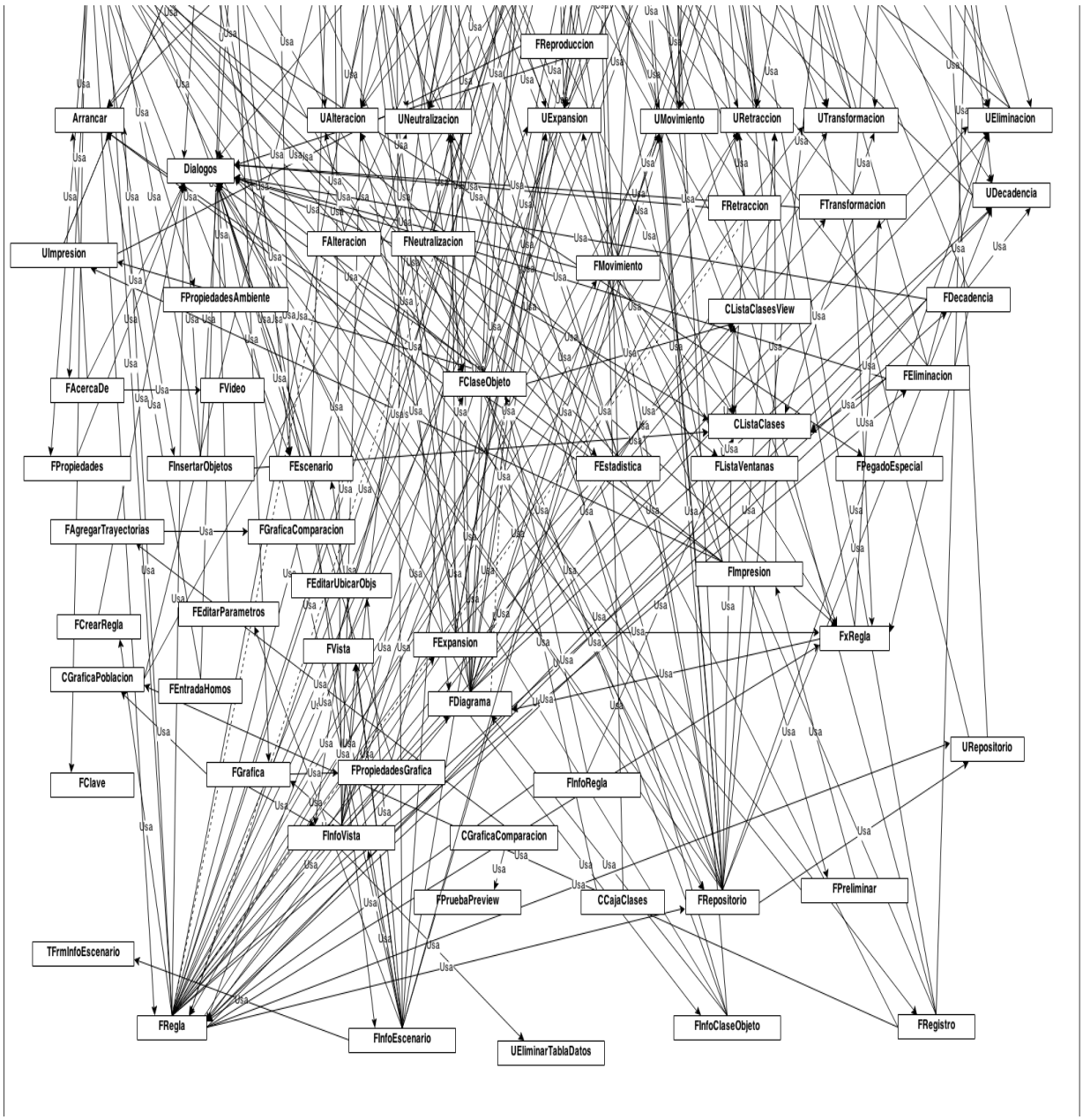


DIAGRAMA DE DEPENDENCIAS VISTA 2



B. DIAGRAMA DE CASOS DE USO

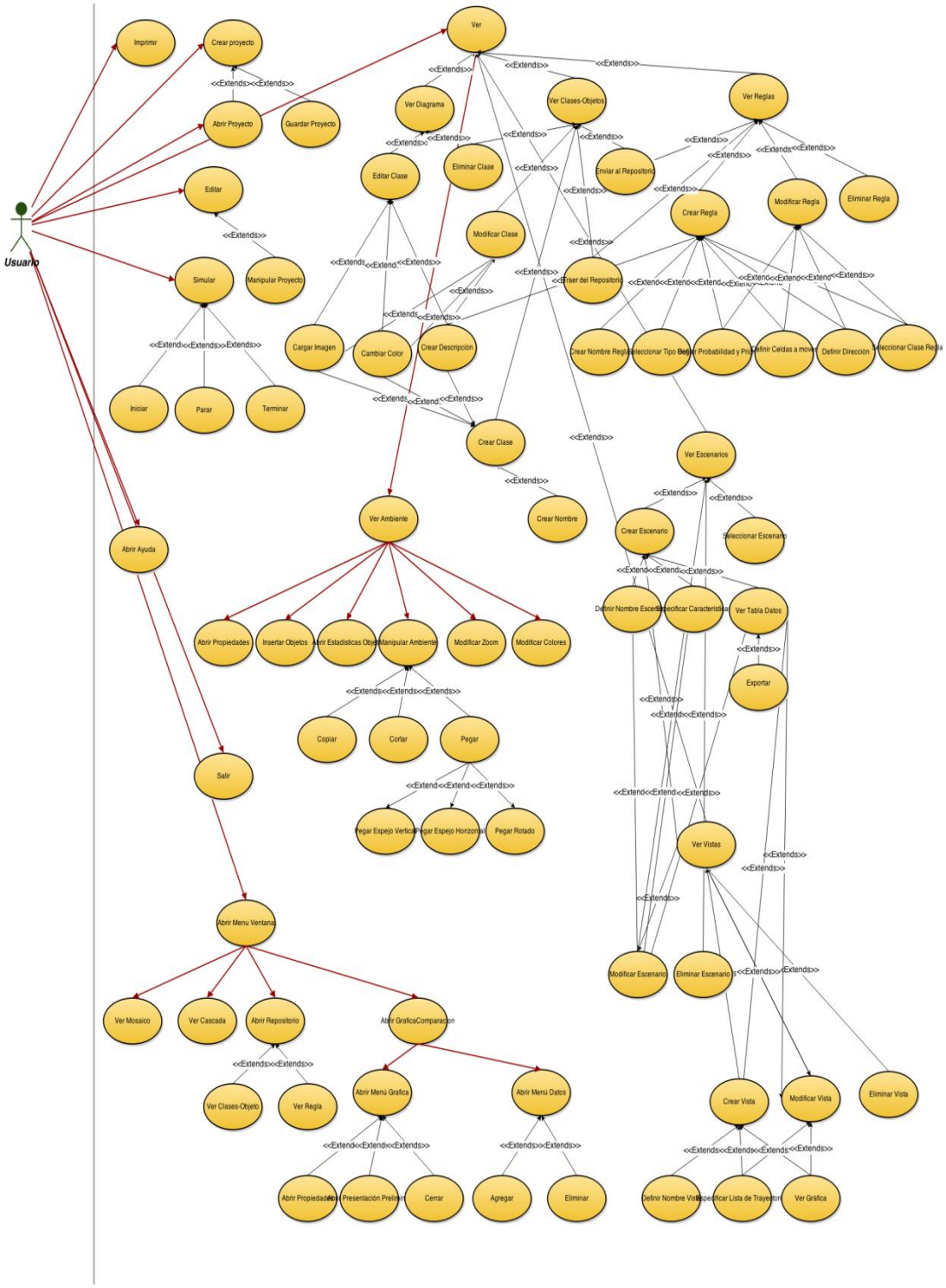


DIAGRAMA DE CASOS DE USO VISTA 1

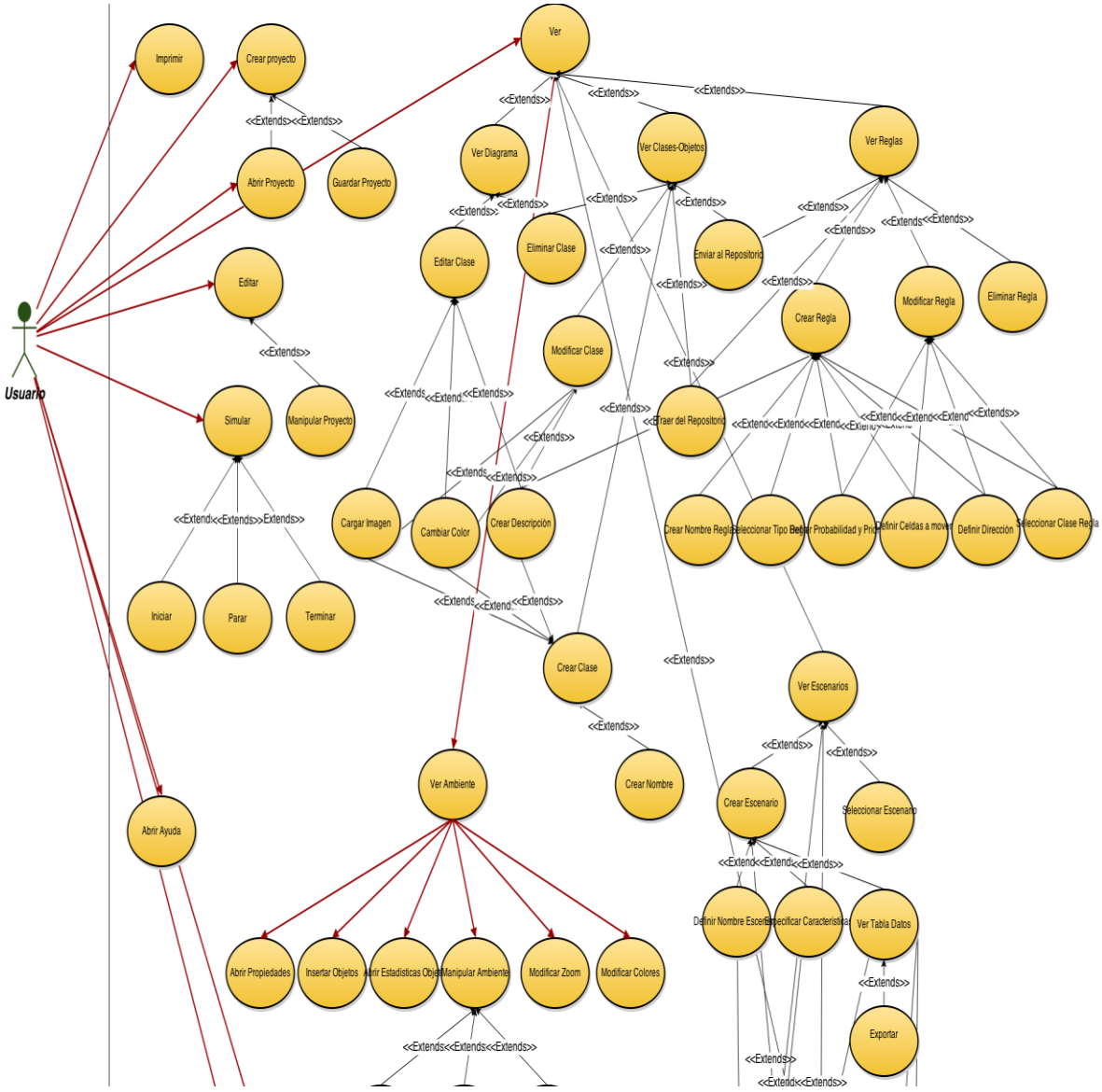
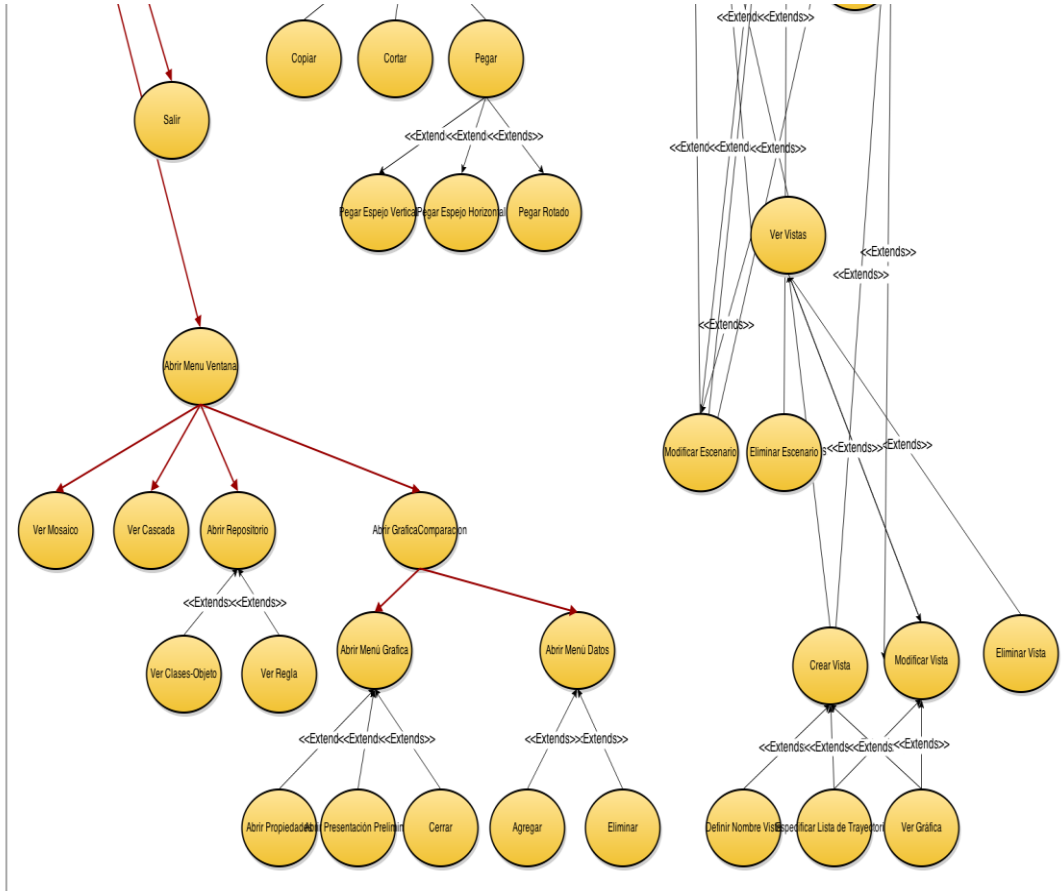


DIAGRAMA DE CASOS DE USO VISTA 2



C. PRUEBAS AQTIME

Routine Name	Time	Time with Children	Shared Time	Hit Count
TSimulador::Execute	96456,07	96456,07	100,00	1
TFrmHomos::ArchivoAbrirClick	20002,72	20542,42	97,37	1
MessageDlgPosHelp	4142,19	4151,70	99,77	2
TSimulador::DarObjetosVecindadEnT	557,10	960,86	57,98	30823
TRegla::GetProbabilidad	372,61	372,61	100,00	391816
TListaReglas::DarRegla	365,17	556,73	65,59	1069611
TFrmHomos::ProyectoAbierto	355,75	525,32	67,72	1
TMatriz::DarPosi	354,60	354,60	100,00	31213672
TAutomata::Estan	309,52	311,36	99,41	266865
TRegla::GetNombre	209,93	209,93	100,00	12904962
THAmbiente::PintarCelda	132,52	135,55	97,76	11067
TAutomata::DarReglasPropias	131,85	1589,57	8,29	30823
TGraficaPoblacion::Pintar	80,42	124,21	64,75	70
TAutomata::PuedeProbabilidad	71,12	71,12	100,00	391816
TListaReglas::DarTipoRegla	58,12	638,29	9,10	1069611
TFrmEntradaHomos::FormCreate	54,42	54,42	100,00	1
TFrmInfoVista::Button5Click	53,55	58,23	91,95	1
TFrmHomos::ProyectoCerrado	48,04	58,76	81,75	2
TMatriz::GetDatos	39,46	39,46	100,00	426669
TGraficaPoblacion::Paint	37,91	47,60	79,65	71
TAutomata::Evolucionar	30,04	1626,98	1,85	30823
TFrmHomos::FormShow	27,15	27,15	99,99	1
TFrmHomos::ActivarBotonesdelProyecto	25,00	25,00	100,00	4
TFrmDiagrama::CargarClasesObjeto	23,55	30,14	78,12	1
TListaReglas::AsignarActual	23,44	580,17	4,04	1069611
TEntero::Create	15,03	15,03	100,00	63302
TMatriz::DarPosj	14,96	14,96	100,00	1562827

TFrmAmbiente::ActualizarBotonesClases	13,61	49,29	27,62	1
TAutomata::DireccionesPosibles	13,53	29,28	46,20	15506
TAmbiente::GetMatriz	11,47	50,64	22,66	425469
TFrmHomos::Simular1Click	9,98	10,16	98,19	1
CreateMessageDialog	9,49	9,49	99,93	2
TReglaAplicable::Create	8,45	8,45	100,00	17290
TFrmDiagrama::CargarReglas	6,65	9,24	71,96	1
TFrmDiagrama::AgregarClaseEstructura	6,25	6,29	99,45	10
TAmbiente::DarCantidades	6,19	12,88	48,09	71
TTablaDatos::GetDatos	5,73	5,73	100,00	16032
TSimulador::PasarSiguiente	5,52	102,50	5,39	6079
TAutomata::ValidarEspacio	5,06	14,16	35,73	15506
TListaClasesView::CrearLista	4,53	4,53	99,96	1
TEntero::Abrir	3,46	3,46	100,00	710
TAutomata::PuedeDireccion	3,45	46,92	7,36	15506
TWord::Abrir	3,42	3,42	100,00	1200
TFrmRegla::CrearLista	3,19	3,19	99,89	1
TSimulador::Actualizar	2,76	143,44	1,92	71
TListaClasesObjeto::DarClaseObjeto	2,65	4,07	65,24	11334
TFrmSimulador::ActualizarEscenario	2,42	140,68	1,72	71
TFrmHomos::AbrirProyecto	2,39	539,70	0,44	1
TFrmDiagrama::AgregarReglaEstructura	2,33	2,56	90,79	22
TFrmHomos::CerrarProyecto	2,11	61,23	3,44	1
TAutomata::EstaLibre	2,09	9,10	22,95	63302
TSimulador::PasarSiguienteEstado	1,98	2832,88	0,07	71
TFrmAmbiente::SeleccionarZoom	1,94	16,65	11,63	3
TMatriz::SetDatos	1,65	1,65	100,00	11267
TDato::Abrir	1,58	5,04	31,35	71
TFrmAmbiente::HAmbiente1Marcado	1,51	1,51	100,00	6
TClaseObjeto::GetNombre	1,43	1,43	100,00	79269
TFrmGrafica::ActualizarGrafica	1,34	4,65	28,71	1

TAutomata::DarPosicion	1,33	1,33	100,00	73653
TMatriz::Destroy	1,21	1,21	100,00	2
TChartPreview::FormCreate	1,20	1,22	98,34	1
TMatriz::Iniciar	1,18	1,45	81,86	3
TMatriz::Abrir	0,92	4,34	21,28	1
TClaseObjeto::Abrir	0,87	0,87	99,69	10
TDato::Destroy	0,84	0,84	100,00	142
TFrmAmbiente::SeleccionarModo	0,80	19,25	4,18	3
TChartPreview::FormDestroy	0,77	0,77	99,71	1
ActivadaDireccion	0,72	0,72	100,00	124048
TRepositorio::Guardar	0,68	0,68	99,65	1
TFrmSimulador::BtnEjecutarClick	0,63	6,29	10,01	1
initialization	0,61	0,61	100,00	1
TListaClases::CrearLista	0,59	0,59	100,00	1
TRegistro::SetUltimoArchivo	0,55	0,55	100,00	1
THAmbiente::DarCoordInfX	0,51	0,51	100,00	18483
TSpinButton::SetUpGlyph	0,49	0,49	100,00	4
PintarFlecha	0,48	0,48	100,00	26
TFrmHomos::FormCreate	0,46	1,90	24,32	1
TProyecto::Abrir	0,43	11,85	3,61	1
TDato::Create	0,40	0,40	100,00	71
TFrmSimulador::FormShow	0,38	0,38	100,00	2
TObjeto::Destroy	0,37	0,37	100,00	1343
TSimulador::Create	0,37	0,37	100,00	1
TParametro::Abrir	0,31	0,31	100,00	22
TFrmHomos::ActivarBotonesEjecutando	0,30	0,30	100,00	3
TSpinButton::SetDownGlyph	0,29	0,29	100,00	4

D. PRUEBAS PRODELPHI

Run	Unit	Class	Method	%	Calls	Av. RT	RT-Sum	Av. RT *	RT-Sum *	% *
1	Arrancar	---	Verificar	0.05	1	13.789 µs	13.789 µs	13.789 µs	13.789 µs	0.05
1	CListaClases	TListaClases	CrearLista	4.83	1	1.285 ms	1.285 ms	1.285 ms	1.285 ms	4.83
1	CListaClasesView	TListaClasesView	Create	3.98	2	529.277 µs	1.059 ms	529.277 µs	1.059 ms	3.98
1	FAcercaDe	TFrmAcercaDe	FormCreate	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	FAlteracion	TFrmAlteracion	FormCreate	0.01	1	2.757 µs	2.757 µs	2.757 µs	2.757 µs	0.01
1	FDecadencia	TFrmDecadencia	FormCreate	0.01	1	2.237 µs	2.237 µs	2.237 µs	2.237 µs	0.01
1	FEditarUbicarObjs	TFrmEditarUbicarObjs	FormCreate	0.01	1	1.580 µs	1.580 µs	1.286 ms	1.286 ms	4.83
1	FEliminacion	TFrmEliminacion	FormCreate	0.01	1	2.997 µs	2.997 µs	2.997 µs	2.997 µs	0.01
1	FEntradaHomos	TFrmEntradaHomos	FormCreate	47.66	1	12.684 ms	12.684 ms	12.684 ms	12.684 ms	47.66
1	FEscenario	TFrmEscenario	FormCreate	0.00	1	0.003 µs	0.003 µs	0.003 µs	0.003 µs	0.00
1	FExpansion	TFrmExpansion	FormCreate	0.02	1	4.077 µs	4.077 µs	4.077 µs	4.077 µs	0.02
1	FHomos	TFrmHomos	ActivarBotonesdelProyecto	5.66	2	753.629 µs	1.507 ms	753.629 µs	1.507 ms	5.66
1	FHomos	TFrmHomos	ActivarBotonesEjecutando	2.70	1	719.069 µs	719.069 µs	719.069 µs	719.069 µs	2.70
1	FHomos	TFrmHomos	FormActivate	0.01	1	1.417 µs	1.417 µs	275.994 µs	275.994 µs	1.04
1	FHomos	TFrmHomos	FormCreate	9.83	1	2.617 ms	2.617 ms	7.282 ms	7.282 ms	27.36
1	FHomos	TFrmHomos	FormShow	10.97	1	2.919 ms	2.919 ms	2.919 ms	2.919 ms	10.97
1	FHomos	TFrmHomos	ProyectoCerrado	3.09	1	821.889 µs	821.889 µs	2.326 ms	2.326 ms	8.74
1	FInfoEscenario	TFrmInfoEscenario	FormCreate	0.34	1	89.580 µs	89.580 µs	89.580 µs	89.580 µs	0.34
1	FInfoVista	TFrmInfoVista	FormCreate	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	FJuego	TFrmJuego	FormCreate	0.00	1	0.003 µs	0.003 µs	0.003 µs	0.003 µs	0.00
1	FMovimiento	TFrmMovimiento	FormCreate	0.01	1	3.006 µs	3.006 µs	3.006 µs	3.006 µs	0.01
1	FNeutralizacion	TFrmNeutralizacion	FormCreate	0.01	1	3.769 µs	3.769 µs	3.769 µs	3.769 µs	0.01
1	FPropiedades	TFrmPropiedades	FormCreate	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	FRepositorio	TFrmRepositorio	FormCreate	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	FReproduccion	TFrmReproduccion	FormCreate	0.01	1	2.677 µs	2.677 µs	2.677 µs	2.677 µs	0.01
1	FRetraccion	TFrmRetraccion	FormCreate	0.01	1	2.620 µs	2.620 µs	2.620 µs	2.620 µs	0.01
1	FSimulador	TFrmSimulador	FormCreate	0.01	1	1.771 µs	1.771 µs	720.840 µs	720.840 µs	2.71
1	FSimulador	TFrmSimulador	FormShow	1.03	1	274.577 µs	274.577 µs	274.577 µs	274.577 µs	1.03
1	FTransformacion	TFrmTransformacion	FormCreate	0.01	1	2.906 µs	2.906 µs	2.906 µs	2.906 µs	0.01
1	FVista	TFrmVista	FormCreate	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	Hilo	---	INITIALIZATION	0.08	1	20.849 µs	20.849 µs	20.849 µs	20.849 µs	0.08
1	UAutomata	---	INITIALIZATION	0.46	1	122.289 µs	122.289 µs	122.289 µs	122.289 µs	0.46
1	UClaseObjeto	TClaseObjeto	Abrir	3.85	2	512.026 µs	1.024 ms	512.363 µs	1.025 ms	3.85
1	UClaseObjeto	TClaseObjeto	GetImagen	0.00	2	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	UClaseObjeto	TClaseObjeto	GetNombre	0.00	6	0.170 µs	1.023 µs	0.171 µs	1.023 µs	0.00
1	UClaseObjeto	TClaseObjeto	SetColor	0.00	2	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	UClaseObjeto	TClaseObjeto	SetImagen	0.00	2	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	UClaseObjeto	TClaseObjeto	SetNombre	0.00	2	0.336 µs	0.671 µs	0.337 µs	0.671 µs	0.00
1	UImpresion	---	INITIALIZATION	0.00	1	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	Uintelig3	---	INITIALIZATION	0.42	1	111.791 µs	111.791 µs	111.791 µs	111.791 µs	0.42
1	UListaClasesObjeto	TListaClasesObjeto	Abrir	0.53	1	141.966 µs	141.966 µs	1.167 ms	1.167 ms	4.38
1	UListaClasesObjeto	TListaClasesObjeto	DarClaseObjeto	0.02	3	1.612 µs	4.837 µs	1.954 µs	5.860 µs	0.02
1	UListaReglas	TListaReglas	Abrir	0.17	1	45.583 µs	45.583 µs	124.197 µs	124.197 µs	0.47
1	UListaReglas	TListaReglas	Abrir(DarTipoRegla)	0.04	3	3.510 µs	10.529 µs	26.206 µs	78.614 µs	0.30
1	UMovimiento	TMovimiento	Abrir	0.22	3	19.752 µs	59.257 µs	22.694 µs	68.086 µs	0.26
1	UMovimiento	TMovimiento	SetDireccion	0.00	3	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	UMovimiento	TMovimiento	SetNumPasos	0.00	3	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	URegistro	TRegistro	GetDirectorioHomos	0.26	1	68.351 µs	68.351 µs	68.351 µs	68.351 µs	0.26
1	URegistro	TRegistro	GetDirectorioTrabajo	1.07	1	283.911 µs	283.911 µs	283.911 µs	283.911 µs	1.07
1	URegistro	TRegistro	GetUltimoArchivo	0.22	1	58.046 µs	58.046 µs	58.046 µs	58.046 µs	0.22
1	URegla	TRegla	DarClaseObjeto	0.00	3	0.435 µs	1.306 µs	2.389 µs	7.166 µs	0.03
1	URegla	TRegla	SetClaseObjeto1	0.00	3	0.000 µs	0.000 µs	0.000 µs	0.000 µs	0.00
1	URegla	TRegla	SetNombre	0.00	3	0.206 µs	0.617 µs	0.206 µs	0.617 µs	0.00
1	URegla	TRegla	SetProbabilidad	0.00	3	0.349 µs	1.046 µs	0.349 µs	1.046 µs	0.00
1	URepositorio	TRepositorio	Abrir	2.39	1	635.114 µs	635.114 µs	1.926 ms	1.926 ms	7.24

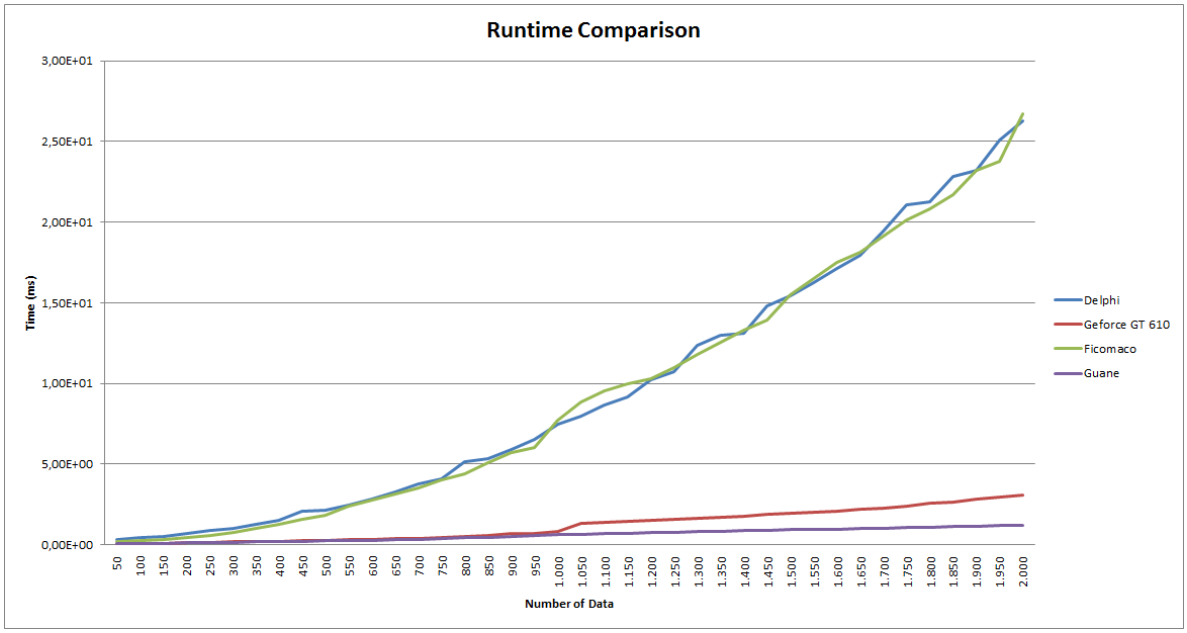
Jueves, Febrero 17, 2005, 04:32 PM Page 1
sim (Run: 3) / CPU: 350 MHz / Total RT: 1.775 m
* : Values include child methods

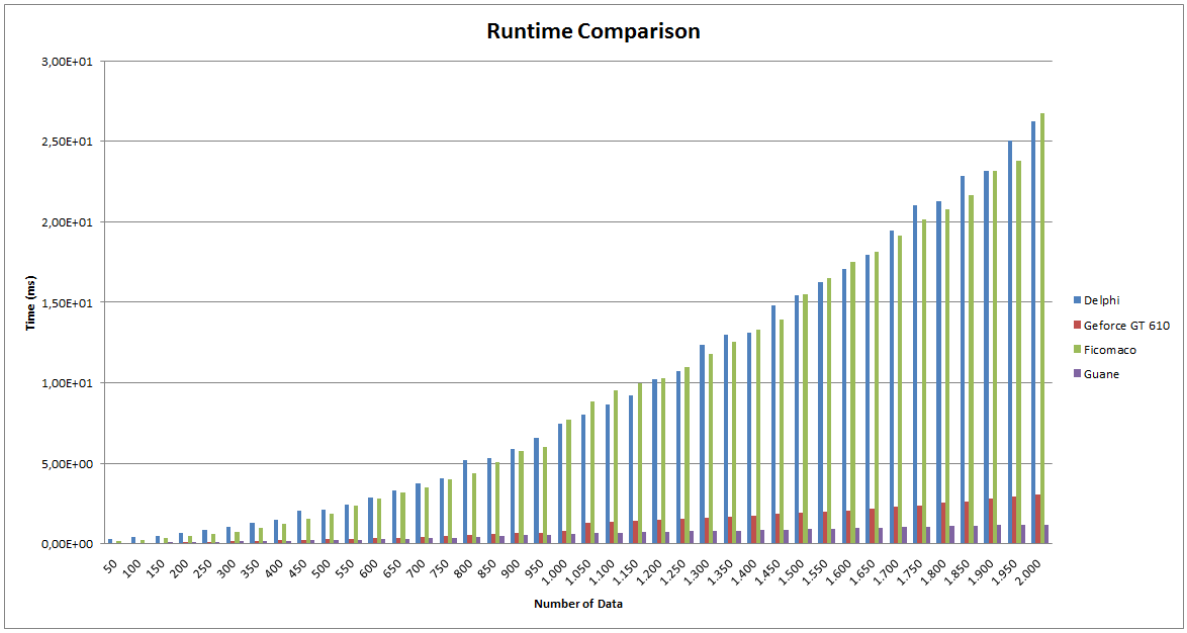
No	Unit	Class	Method	RT-Sum *	% *	Percentage of Runtime
1	USimulador	TSimulador	Execute	**1.027 m**	57.85	
2	USimulador	TSimulador	PasarSiguienteEstado	**43.117 s**	40.49	
3	USimulador	TSimulador	DarObjetosVecindadEnT	**28.055 s**	26.34	
4	USimulador	TSimulador	PasarSiguiente	**7.218 s**	6.78	
5	UAutomata	TAutomata	Evolucionar	**6.554 s**	6.15	
6	CHAmbiente	THAmbiente	PintarCelda	**5.934 s**	5.57	
7	UAutomata	TAutomata	DarReglasPropias	**5.180 s**	4.86	
8	UAutomata	TAutomata	PuedeDireccion	**2.698 s**	2.53	
9	USimulador	TSimulador	ActualizarAmbiente	**1.733 s**	1.63	
10	FSimulador	TFrmSimulador	RetornarAmbiente	**1.733 s**	1.63	
11	UMatriz	TMatriz	DarPosi	**1.524 s**	1.43	
12	UListaReglas	TListaReglas	DarTipoRegla	**1.430 s**	1.34	

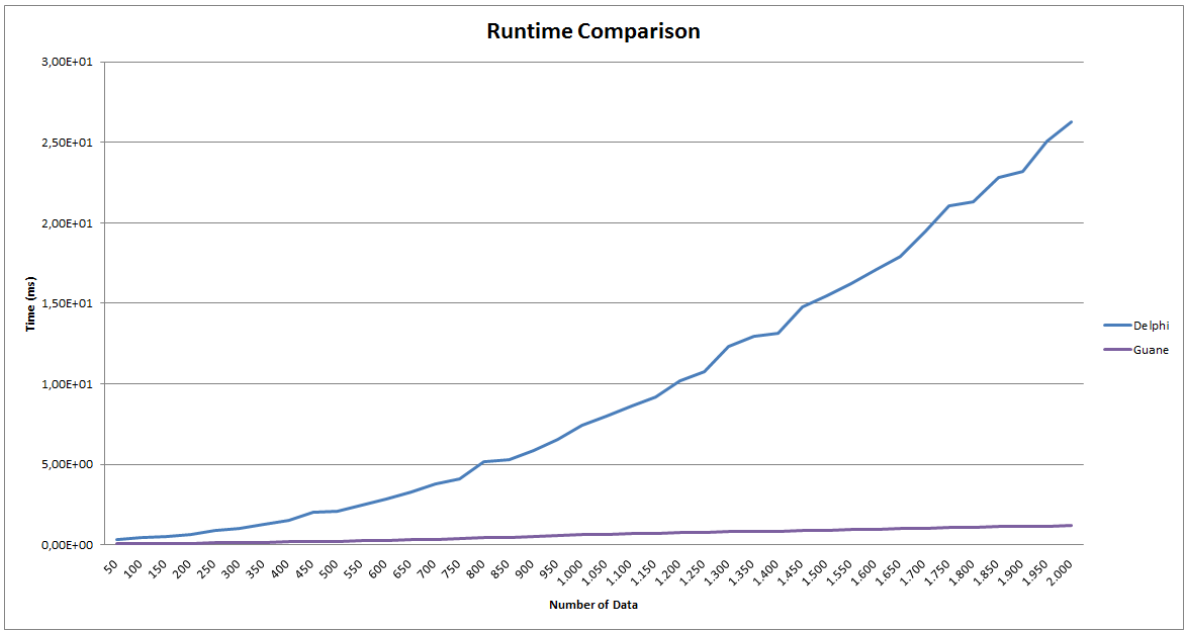
Jueves, Febrero 17, 2005, 04:26 PM Page 1
inicial (Run: 1) / CPU: 350 MHz / Total RT: 26.612 ms
* : Values include child methods

No	Unit	Class	Method	Unit-Sum	%	Percentage of Runtime
1	FEntradaHomos	TFrmEntradaHomos	FormCreate	**12.684 ms**	47.66	
2	FHomos	TFrmHomos	FormShow	**8.585 ms**	32.26	
3	CListaClases	TListaClases	CrearLista	**1.285 ms**	4.83	
4	CListaClasesView	TListaClasesView	Create	**1.059 ms**	3.98	
5	UClaseObjeto	TClaseObjeto	Abrir	**1.026 ms**	3.85	
6	URepositorio	TRepositorio	Abrir	**635.114 µs**	2.39	
7	URegistro	TRegistro	GetDirectorioTrabajo	**410.309 µs**	1.54	
8	FSimulador	TFrmSimulador	FormShow	**276.349 µs**	1.04	
9	UListaClasesObjeto	TListaClasesObjeto	Abrir	**146.803 µs**	0.55	
10	UAutomata	---	INITIALIZATION	**122.289 µs**	0.46	
11	Uintelig3	---	INITIALIZATION	**111.791 µs**	0.42	
12	FInfoEscenario	TFrmInfoEscenario	FormCreate	**89.580 µs**	0.34	

E. Comparación prototipo CUDA




Runtime Comparison
Time (ms)
3,00E+01
2,50E+01
2,00E+01
1,50E+01
1,00E+01
5,00E+00
0,00E+00
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1.000 1.050 1.100 1.150 1.200 1.250 1.300 1.350 1.400 1.450 1.500 1.550 1.600 1.650 1.700 1.750 1.800 1.850 1.900 1.950 2.000
Number of Data
Delphi
Geforce GT 610
Ficomaco
Guane

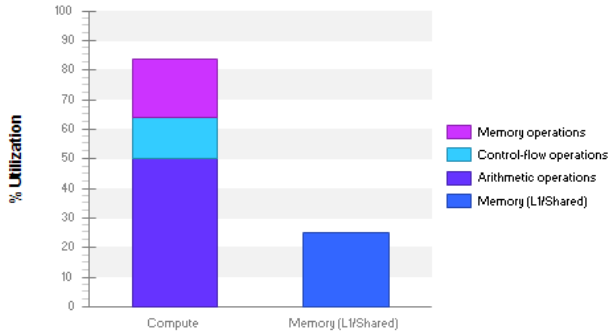

Runtime Comparison
Time (ms)
3,00E+01
2,50E+01
2,00E+01
1,50E+01
1,00E+01
5,00E+00
0,00E+00
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1.000 1.050 1.100 1.150 1.200 1.250 1.300 1.350 1.400 1.450 1.500 1.550 1.600 1.650 1.700 1.750 1.800 1.850 1.900 1.950 2.000
Number of Data
Delphi
Guane

F. Pruebas Nvidia Visual Profiler

Rendimiento computacional del método Kernel

Kernel Performance Is Bound By Compute

For device "GeForce GT 610" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

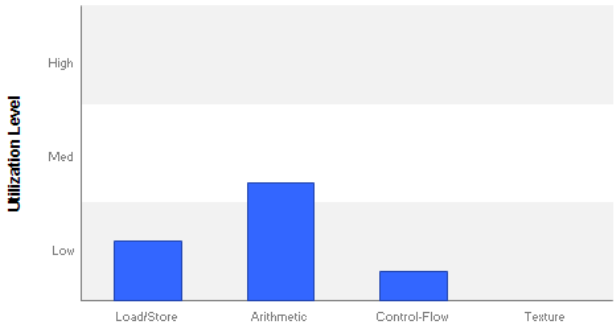


Utilización de la unidad de funciones

Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.
Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.
Texture - Texture operations.



Utilización de la GPU

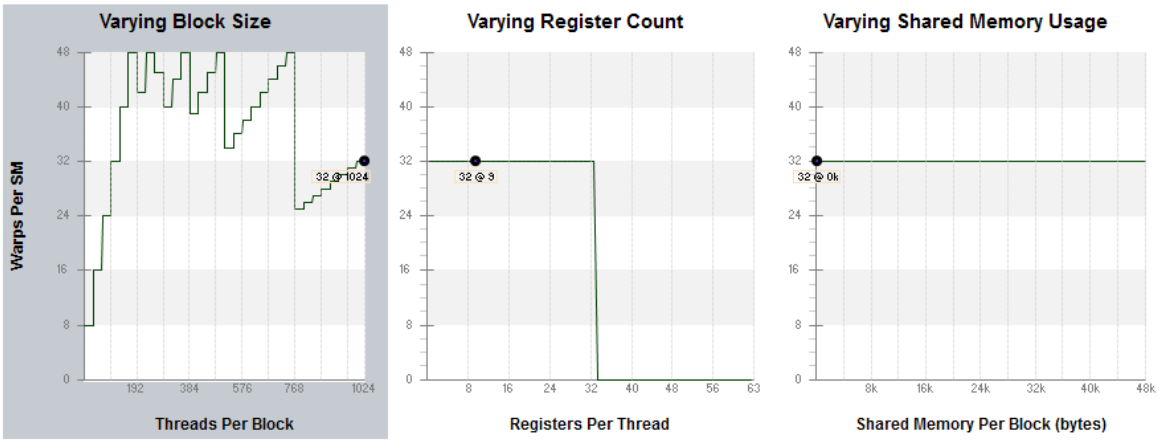
GPU Utilization May Be Limited By Block Size

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel has a block size of 1024 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GT 610" can simultaneously execute up to 8 blocks on each SM. Because each block uses 32 warps to execute the block's 1024 threads, the kernel is using only 32 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [2,1,1] (2 blocks) Block Size: [1024,1,1] (1024 threads)
Occupancy Per SM				
Active Blocks		1	8	0 1 2 3 4 5 6 7 8
Active Warps	28,91	32	48	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Active Threads		1024	1536	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536
Occupancy	60,2%	66,7%	100%	0% 15% 30% 45% 60% 75% 90% 100%
Warps				
Threads/Block		1024	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		32	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Block Limit		1	8	0 1 2 3 4 5 6 7 8
Registers				
Registers/Thread		9	63	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 63
Registers/Block		10240	32768	0 4k 8k 12k 16k 20k 24k 28k 32k
Block Limit		3	8	0 1 2 3 4 5 6 7 8
Shared Memory				
Shared Memory/Block		0	49152	0 8k 16k 24k 32k 40k 48k
Block Limit		8	8	0 1 2 3 4 5 6 7 8



Latencia de instrucciones

Instruction Latencies

Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle. The following chart shows the break-down of stalls reasons averaged over the entire execution of the kernel. The kernel has low theoretical or achieved occupancy. Therefore, it is likely that the instruction stall reasons described below are not the primary limiters of performance and so should not be considered until any occupancy issues are resolved.

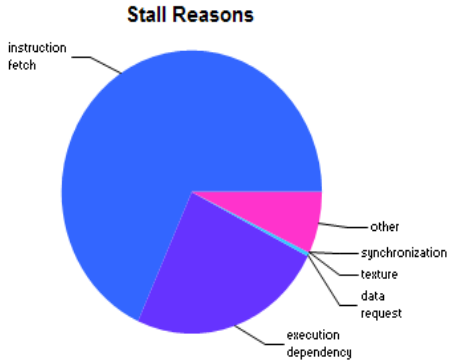
Instruction Fetch - The next assembly instruction has not yet been fetched.

Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Data Request - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.



Resultados de pruebas, método Kernel

addKernel(int*, int*, int*, int*, int*, int*, int*, int*, i...		
Start	152,408 ms	
End	155,486 ms	
Duration	3,078 ms	
Grid Size	[2,1,1]	
Block Size	[1024,1,1]	
Registers/Thread	9	
Shared Memory/Block	0 bytes	
◢ Efficiency		
Warp Execution Efficiency	⚠ 79,9%	
◢ Occupancy		
Achieved	60,3%	
Theoretical	66,7%	
◢ Shared Memory Configuration		
Shared Memory Requested	48 KB	
Shared Memory Executed	48 KB	
Shared Memory Bank Size	4 bytes	

G. Pruebas de capacidad Prototipo

Tiempo de Ejecución (ms)			
Número de Datos	**Equipo 1**	**Ficomaco**	**Guane**
100	0,05	0,24	0,07
1.000	0,78	7,64	0,62
10.000	74,33	438,63	5,84
100.000		9909,07	190,59
250.000		25223,52	477,19
500.000		49687,94	929,54
1.000.000		99490,29	1840,13
2.000.000		202041,14	3670,77

H. Tabla de Rendimiento

Numero de Datos	Delphi	GT 610	Ficomaco	Guane
50	3,03030303	17,6678445	5,07872016	13,1061599
100	2,22222222	17,4825175	3,94321767	12,5944584
150	1,92307692	12,7226463	2,91036088	11,1607143
200	1,49253731	8,77192982	2,16684724	9,04159132
250	1,13636364	7,69230769	1,65098233	7,82472613
300	0,97087379	5,82411182	1,30123617	6,57462196
350	0,78125	5,12295082	0,98892405	5,71428571
400	0,66225166	4,71920717	0,80070462	5,32481363
450	0,48543689	4,12541254	0,64036885	4,73484848
500	0,47169811	3,61271676	0,53789468	4,34782609
550	0,40816327	3,32115576	0,41895345	4,03225806
600	0,34965035	2,96296296	0,35816619	3,47826087
650	0,3030303	2,68384326	0,3160856	3,07219662
700	0,26595745	2,43249818	0,28381677	2,82167043
750	0,24449878	2,11193242	0,25015009	2,52143217
800	0,19305019	1,87793427	0,22752093	2,32288037
850	0,18832392	1,68833361	0,19642121	2,08333333
900	0,16977929	1,48170099	0,17428283	1,84365782
950	0,15243902	1,40627197	0,16652235	1,77525297
1.000	0,13422819	1,21624909	0,12981787	1,59642401
1.050	0,125	0,76213703	0,11285535	1,51745068
1.100	0,11587486	0,72674419	0,10484928	1,45264381
1.150	0,10881393	0,69141948	0,1002999	1,39004726
1.200	0,09803922	0,66159444	0,09719212	1,33815068
1.250	0,09319664	0,6339948	0,09109625	1,28518185
1.300	0,08090615	0,61061244	0,08469695	1,23900384
1.350	0,07716049	0,58534301	0,07988433	1,19631535
1.400	0,0761035	0,56350727	0,07514729	1,1554015

1.450	0,06756757	0,52991362	0,07170721	1,11919418
1.500	0,06472492	0,51751798	0,06454778	1,08061379
1.550	0,06153846	0,49751244	0,06051584	1,05086171
1.600	0,05847953	0,48141729	0,05707013	1,01843365
1.650	0,05577245	0,45905252	0,05511282	0,98902186
1.700	0,0514668	0,43729229	0,05229141	0,96302003
1.750	0,04750594	0,41573127	0,04959924	0,93703148
1.800	0,04699248	0,39099155	0,04807299	0,91491308
1.850	0,04380201	0,3784152	0,04611483	0,888494
1.900	0,04314064	0,35640459	0,04312799	0,86715227
1.950	0,03988831	0,34028652	0,04202511	0,8469552
2.000	0,03808073	0,324412	0,03739352	0,82767754