UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y EN SISTEMAS

"ANÁLISIS DEL RENDIMIENTO DE LA PARALELIZACIÓN DEL JUEGO DE LA VIDA MEDIANTE EL USO DE GPUs Y CUDA"

TESINA

QUE PARA OBTENER EL GRADO DE

ESPECIALISTA EN CÓMPUTO DE ALTO RENDIMIENTO

PRESENTA.

ISRAEL VELÁZQUEZ GUTIÉRREZ

Directora de tesina:

DRA. MARÍA ELENA LÁRRAGA RAMÍREZ

Instituto de Ingeniería, UNAM

Ciudad Universitaria, Ciudad de México. Diciembre, 2020.

Dedicatoria.

A la memoria mi padre.

Agradecimientos:

A mi familia, por darme el apoyo necesario para realizar este posgrado y ser mi motor motivacional principal.

A mi tutora, la Dra. María Elena Lárraga Ramírez, por su tiempo, dedicación, paciencia e invaluables consejos para mejorar este trabajo. Asimismo; por haberme motivado a continuar con el proyecto en este año tan complicado que se nos ha presentado, poniendo de manifiesto el profesionalismo y liderazgo que la caracterizan como persona.

Al Instituto de Ingeniería (IINGEN) y al proyecto PAPIIT IN112619, por la beca SICOE que me fue otorgada para tener acceso a un lugar de trabajo, equipo de cómputo necesario y demás infraestructura.

A la Dirección General de Cómputo de Tecnologías de la Información y Comunicación (DGTIC), y en especial, a la Coordinación de Supercómputo, por el apoyo económico e infraestructura que me fueron otorgados durante el curso de este programa de posgrado.

A mis sinodales, el Dr. Ernesto Rubio Acosta y al Ing. Adrián Durán Chavesti, por su dedicación, tiempo y experiencia compartida en la realización de este proyecto.

A mis compañeros de la especialidad: Lety, Toño, Roberto, Miguel y Hermilo, fue muy grata experiencia convivir y apoyarnos juntos en esta travesía académica que indudablemente deja una huella indeleble en mi vida.

A mis profesores, Dra. Ma. E. Lárraga, Dr. Héctor Benítez, Dr. Jorge L. Ortega, Dr. Lukas Nellen, Dr. Ernesto Rubio, Mtra. Yolanda Flores, Mtro. José Luis Gordillo y al Ing. Adrián Durán, a todos ustedes les estoy muy agradecido por haberme compartido más que sus conocimientos y experiencia, la pasión por el Cómputo de Alto Rendimiento.

Y finalmente para cerrar con broche de oro este apartado, a todos los que hicieron posible este programa de Posgrado en Ciencia e Ingeniería de la Computación, empezando por el Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS), por ser la entidad principal donde se imparten los estudios. Asimismo, al coordinador principal el Dr. Javier Gómez Castellanos y a la asistente de procesos Ma. de Lourdes González Lora, quienes incluso desde antes de iniciar con los estudios correspondientes ya están siempre al pendiente de proporcionarnos el apoyo necesario para cualquier trámite administrativo y consejos para sobrevivir en el posgrado.

Índice

Introducción

Los Autómatas Celulares (AC) han proliferado en distintas áreas de la investigación porque pueden modelar comportamientos complejos empleando reglas simples. Los AC son sistemas dinámicos discretos que se componen de agrupaciones de células en un determinado espacio d-dimensional con formaciones regulares o irregulares pero que cada una interactúa con sus vecinas en un conjunto de reglas que determinan la evolución total del sistema simulado. Hay una amplia gama de aplicaciones de AC, tráfico vehicular, modelado de crecimiento urbano, modelado de propagación de epidemias, la propagación de incendios forestales, la detección de bordes en las imágenes, etc. Recientemente hay un gran interés, en el estudio de sistemas de tamaño muy grande con millones de partículas para su uso en la realidad; ante ello, el uso de Cómputo de Alto Rendimiento es fundamental.

La idea del Cómputo de Alto Rendimiento (HPC [1]) surgió desde el principio de la computación convencional que se basó en el modelo Von Neumann, como un reto para diseñar arquitecturas capaces de realizar múltiples operaciones simultáneamente y para resolver problemas científicos más velozmente; tal que a finales de los 50's, ya se tenían las primeras arquitecturas diseñadas. Desde entonces, el HPC ha avanzado en dos grandes líneas de investigación: el *hardware* y el *software*; considerando para el primer caso, la integración de una gran gamma de arquitecturas heterogéneas, y para el segundo caso, el diseño cada vez más eficiente de algoritmos capaces de explotar dichas arquitecturas. Una de las plataformas hardware más utilizadas en la actualidad, son los procesadores masivamente paralelos en forma de GPUs y uno de los lenguajes y conjunto de herramientas de mayor éxito para explotar la capacidad de dichos procesadores es NVIDIA CUDA.

Uno de los aspectos en la programación de GPUs es el uso de las memorias. Un uso incorrecto de la jerarquía de memorias disponibles en las GPUs puede ocasionar que no se consiga acelerar el algoritmo que se implemente de forma paralela y que su tiempo de ejecución sea superior al de la implementación secuencial en la CPU.

El ***objetivo*** de este trabajo es *realizar un análisis del rendimiento de implementaciones paralelas en una arquitectura heterogénea moderna basada en GPUs de uno de los modelos de AC más conocidos, el Juego de la Vida, con respecto a aquella en una arquitectura secuencial convencional tipo CPU.* De tal manera que, las implementaciones paralelas que se proponen no sólo reduzcan los tiempos de ejecución, sino que utilicen de manera adecuada los recursos que se disponen y la jerarquía de memoria disponibles mediante el uso de NVIDIA CUDA. Para ello se desarrollan tres implementaciones del Juego de la Vida, una secuencial, y dos paralelas. Mediante simulación computacional se analizan diferentes tamaños del Juego de la vida y se analiza el desempeño computacional del mismo.

El resto de esta tesina consiste en 4 capítulos:

[1] HPC (High Performace Computing)

En el **Capítulo 1: Marco Teórico**, damos una breve introducción de los conceptos más relevantes que se involucran en este trabajo, tanto de los AC como del HPC, asimismo se abordan algunos ejemplos que fueron considerados como casos de estudio.

En el **Capítulo 2: Desarrollo del modelo**, en este capítulo se explica el desarrollo de los programas realizados para la medición de los factores que fueron requeridos para el análisis de rendimiento.

En el **Capítulo 3: Resultados obtenidos,** se presentan en gráficas comparativas los resultados obtenidos de la ejecución de las pruebas de los programas implementados.

En el **Capítulo 4: Conclusiones y trabajos futuros**, en este capítulo hacemos la exposición de las conclusiones a las que se llegaron dados los resultados obtenidos.

También es importante mencionar que la profundidad de los temas abordados está acotada a la explicación básica de los conceptos requeridos que sustentan este trabajo.

Capítulo **1**

Marco Teórico

En este capítulo se presenta el marco teórico en el que se ubica esta tesina y se definen los conceptos fundamentales para su entendimiento.

1.1 Cómputo paralelo

El paralelismo, consiste en la división de una tarea en unidades discretas computables para que estas subtareas puedan distribuirse entre varios procesadores [1], con el objetivo de que cada uno simultáneamente realice sus operaciones y de esa manera lograr el procesamiento de un problema más eficientemente, su principal ventaja radica en que permite abordar problemas que por su complejidad o tamaño son intratables computacionalmente bajo el enfoque secuencial tradicional.

Sin embargo, implementar algoritmos paralelos para solucionar un problema, no es una tarea fácil; dado que la misma naturaleza de lo que se desea paralelizar contiene en ocasiones, dependencia entre los pasos necesarios para su ejecución, lo que ha significado desde los inicios del cómputo paralelo uno de los principales retos. Por lo que los temas de estudio se han bifurcado en la esencia de lo que se divide: el *dato* o la *acción* [2].

Con base en lo anterior, de manera general, el paralelismo se clasifica en dos categorías principales: ***paralelismo de datos*** y ***paralelismo funcional*** [3].

El paralelismo de datos se enfoca en distribuir datos de una aplicación en varios procesadores para que éstos de manera simultánea realicen los cálculos necesarios bajo una misma función, pero con porciones de datos diferentes cada uno. En este enfoque es necesario que exista un proceso que controle la distribución de datos y después la concentración de resultados. En contraste, el paralelismo funcional consiste en dividir las tareas en subtareas, aunque no necesariamente sean independientes entre ellas, para distribuirlas a través de diferentes procesadores interconectados, las subtareas distribuidas se comunican entre sí en forma directa punto a punto, para coordinarse en la solución de un problema en específico. Este tipo de paralelismo es muy útil cuando las subtareas tienen cargas de trabajo heterogéneas y, por lo tanto, tiempos de ejecución diferentes. Existen otros criterios de clasificación del paralelismo [4, 5], pero que, por la naturaleza de este trabajo, no las vamos a mencionar.

Debido a que el cómputo paralelo tiene su origen en el cómputo serial [5], vamos a iniciar dando un breve repaso al modelo arquitectónico de hardware más relevante que sustenta a los sistemas tradicionales secuenciales, el modelo Von Neumann [6].

1.2 Arquitectura Von Neumann

Es la arquitectura en la que están basados los sistemas uniprocesador tradicionales [6], la cual consiste en:

a) Memoria principal,
b) Una Unidad Central de Procesamiento CPU (procesador o core), mismo que está integrado por una Unidad de Control (CU) y una Unidad Lógica Aritmetica (ALU), y
c) Un canal de interconexión de entradas y salidas.

Como se muestra en la figura 1.2

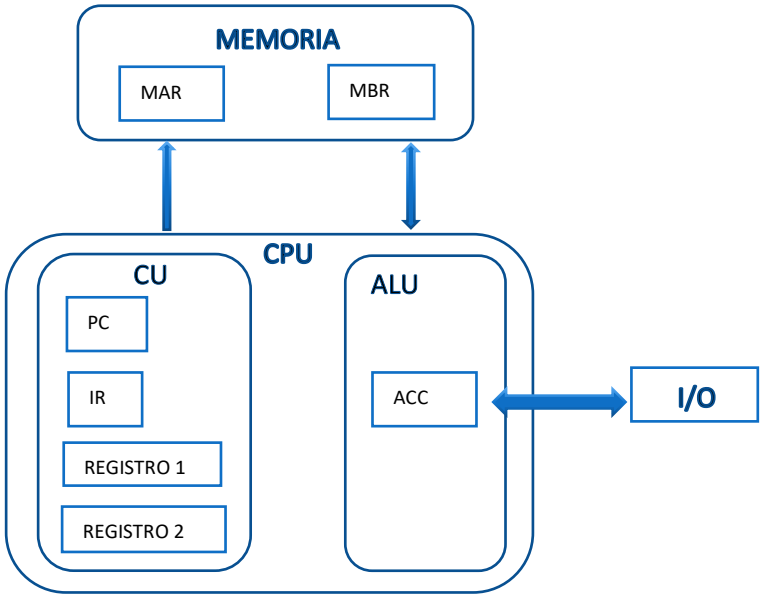

Fig. 1.2 Modelo Von Neumann

En una máquina Von Neumann, se ejecuta una y solo una instruccion simultáneamente y cada instruccion opera sólo en una fracción del total de datos del problema [7].

1.3 Taxonomia de Flynn

La taxonomía más popular de arquitecturas de computadoras fue definida por Michael J. Flynn desde 1966 y oficialmente publicada en 1972 [8], y está basada en la noción de dos entidades fundamentales en la computación: la *instrucción* y el *dato*. La instrucción la define como bloques minímos de operaciones básicas con una secuencia lógica; mientras que el *dato* es el tráfico intercambiado entre la memoria y la Unidad de Procesamiento. Tanto la instrucción como el dato pueden ser simples o múltiples; con base en su combinación se clasifican en las siguientes 4 categorías:

Single Instruccion Single Data (SISD, Instrucción Simple, Dato Simple)
Single Instruccion Multiple Data (SIMD, Instrucción Simple, Múltiples Datos)
Multiple Instruccion Single Data (MISD. Instrucción Múltiple, Dato Simple)
Multiple Instruccion Multiple Data (MIMD, Instrucción Múltiple, Datos Múltiples)

Las computadoras convencionales (Secuenciales) Von Neumann son clasificadas como SISD, mientras que computadoras paralelas pueden ser SIMD o MIMD. Por otro lado, las arquitecturas tipo MISD no han proliferado debido a la complejidad que representan, por lo que en este trabajo no se considerarán a pesar de ser paralelas [1, 2, 7].

1.4 Arquitecturas paralelas

1.4.1 Arquitecturas Paralelas de Hardware

1.4.1.1 SIMD

Esta arquitectura consiste en una sola unidad de control y un arreglo de *n* (P_1 … P_n) procesadores Aritméticos Lógicos *(*Arithmetic Logic Units (*ALU*'s)) cada uno con su bloque de memoria (M_1 … M_n), pudiendo ésta ser individual para cada procesador o global para todos los procesadores. La unidad de control, en la práctica, es un monoprocesador completo modelo Von Neumann, que recupera instrucciones de la memoria y las envía al arreglo de ALUs. Los procesadores (P_1 … P_n) ejecutan la misma instrucción simultáneamente, cada uno toma el dato de su memoria asignada por lo que el dato difiere entre ellos, es decir, cada uno tiene su propio flujo de datos. Algunos de los procesadores pueden desactivarse durante determinadas operaciones. Dicha activación y desactivación de procesadores es manejada por la unidad de control [4, 5, 7].

El arreglo de procesadores ALU está conectado al procesador principal mediante un bus de datos y éste accede a ellos de forma aleatoria. En la fig. 1.4.1A se muestra un modelo conceptual de la topología de esta arquitectura y en la figura 1.4.1B mostramos un esquema de su topología funcional.

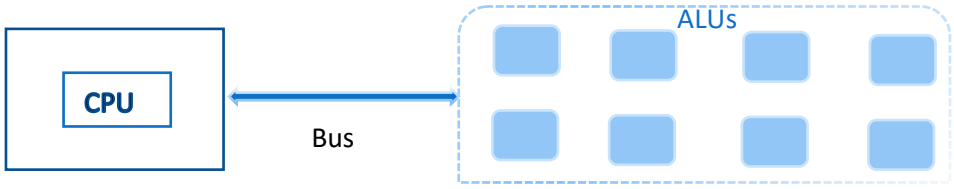

Fig. 1.4.1A Modelo conceptual de la Arquitectura SIMD

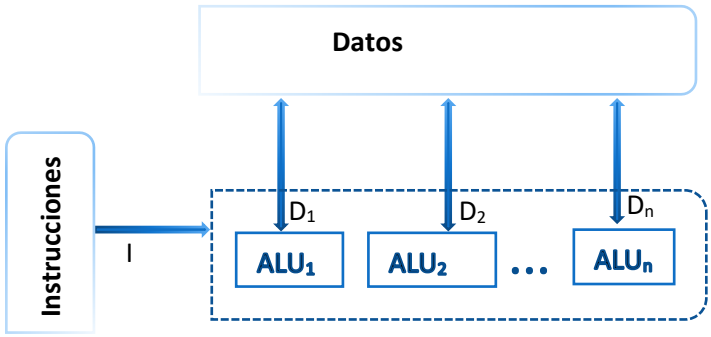

Fig. 1.4.1B Topología Funcional de la Arquitectura SIMD

En esta categoría se contemplan las unidades de procesamiento especializado [5, 7], como Graphics Processing Unit (GPU), Streaming SIMD Extensions (SSE) o Advanced Vector Extension (AVX) [7].

Por su practicidad, el modelo de arquitecturas SIMD ha proliferado en la mayoría de las computadoras de uso personal y dispositivos móviles multicore la incluyen, en ocasiones de forma simple y otras veces combinada con algún tipo de unidad de procesamiento especializado, como las descritas anteriormente.

De igual manera, otra ventaja que ofrece es que un programa puede estar codificado usando un lenguaje serial tradicional y al momento de ejecutarse la maquina puede segmentar ciertas instrucciones que no tengan dependencia funcional entre ellas y ejecutarlas en paralelo. Éste es uno de los puntos fuertes del paralelismo de datos.

1.4.1.2 MIMD

Este tipo de arquitecturas contienen múltiples CPU's Von Neumann completos [1, 4, 5, 7], es decir; cada uno con su propia unidad de control y su propia ALU, por lo que una de sus principales diferencias con la arquitectura SIMD, es la capacidad de procesar múltiples instrucciones en múltiples datos, es decir, funcionan independientemente entre ellos, en la figura 1.4.1.2 se muestra un ejemplo de su topología funcional.

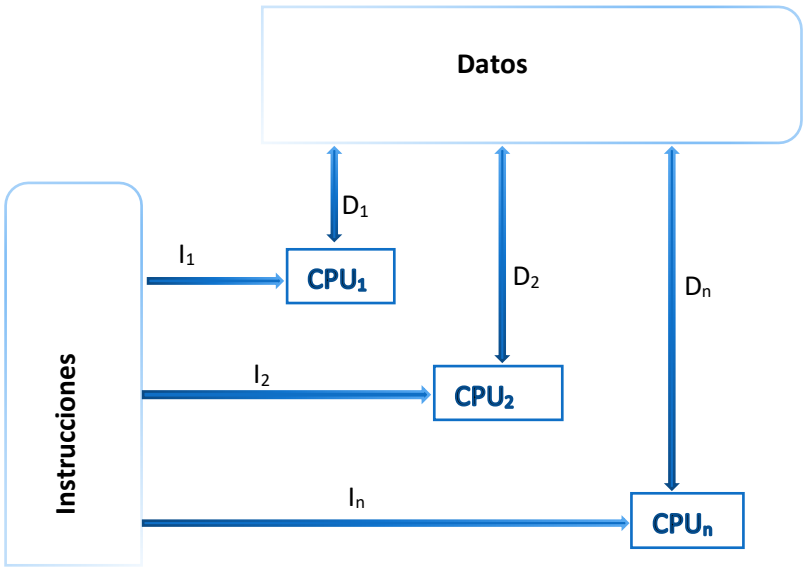

Fig. 1.4.1.2 Topología Funcional de la Arquitectura

Asimismo, la ventaja que esta arquitectura presenta respecto a la SIMD es su fiabilidad, que quiere decir que, si un procesador falla, la tarea a procesarse puede ser realizada por otro procesador solo con una pequeña degradación en su tiempo de ejecución, a esto se le conoce como tolerancia a fallas. En esta categoría se contemplan diseños bastante heterogéneos, desde arquitecturas de multiprocesador hasta clúster y supercomputadoras tipo Grid [5].

1.4.2 Arquitecturas paralelas de software

De acuerdo con [9], la Taxonomia de Flynn puede ser extendida desde la perspectiva del software a las siguientes arquitecturas de programación:

Single Program, Single Data (SPSD; un programa, un dato)
Single Program, Multiple Data (SPMD; un programa, múltiples datos)
Multiple Program, Multiple Data (MPMD; múltiples programas, múltiples datos)
Multiple Program, Single Data (MPSD; múltiples programas, un dato)

1.4.2.1 SPMD

En esta arquitectura, un programa se ejecuta simultáneamente por múltiples procesadores autónomos explotando al máximo las arquitecturas de hardware SIMD, donde un proceso enviado a un procesador puede a la vez fragmentarse y ejecutar varios subprocesos o hilos.

Map Reduce y CUDA, son ejemplos de programas SPMD [5, 7, 10].

1.4.2.2 MPMD

Esta arquitectura tiene la característica de poder ejecutar simultáneamente diferentes programas, con diferentes datos, por múltiples procesadores autónomos para resolver un problema común. Esta característica es de mucha ayuda cuando se trata de resolver problemas grandes y complejos con fuerte dependencia funcional entre sus procesos y que requieren de su intercomunicación para concluir la tarea final. El Cloud Computing tanto en Servicios Orientados a la Arquitectura (SOA's) como en Software como Servicio (SaaS) son ejemplos de la implementación de arquitecturas MPMD [7, 11].

1.5 Organización de la Memoria

En [12], Jorge L. Ortega Arjona menciona que la memoria y los periféricos son recursos comunes o compartidos que emplean los procesadores para poder comunicarse entre sí durante la ejecución de un programa paralelo o distribuido, y dependiendo del grado de interacción (entre los procesos y los periféricos) se clasifican en dos tipos: **fuertemente acoplados** (*tightly-coupled*) o **débilmente acoplados** (*loosely-coupled*) para grados de interacción alto y bajo respectivamente.

También menciona que la manera en cómo la memoria es repartida entre los procesadores, determina el mecanismo de comunicación entre ellos, el cual podría ser por ***memoria compartida*** o por ***paso de mensajes***.

1.5.1 Memoria compartida

Los sistemas con esta arquitectura son *fuertemente acoplados* [12], consisten en módulos de memoria común interconectados entre sí, como se muestra en la figura 1.5.1A, donde

todos los procesadores del sistema tienen igual oportunidad de acceso para intercambiar datos. Los módulos de memoria se pueden organizar de manera local y compartida en cada procesador o como un espacio global y común para todos los procesadores, en el segundo esquema: el medio de conexión, la topología de red que implementen y la distancia al procesador juegan un rol importante y relevante para determinar el tiempo de acceso de los procesadores al módulo de memoria requerido.

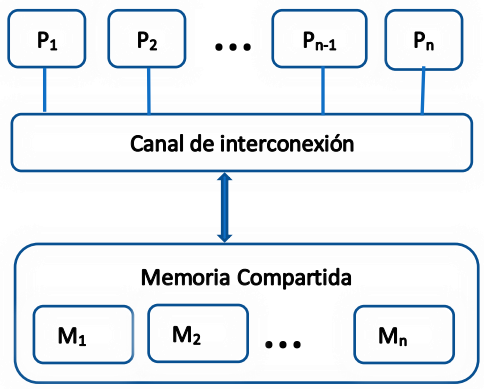

Fig. 1.5.1A Taxonomia de Memoria Compartida

Desde el punto de vista del *tiempo de acceso* a dicha memoria, la plataforma como se muestra en la figura 1.5.1B, se clasifica en tres categorías que a continuación se describen. UMA (Uniform Memory Access) si es homogéneo para todos los procesadores, sí el tiempo difiere, se clasifica como NUMA (Non Uniform Memory Access). En las plataformas NUMA cada procesador tiene una parte de la memoria compartida, por lo que también se les conoce como arquitecturas de memoria distribuida compartida. Una tercera clasificación es COMA (Cache Only Memory Access), que es similar a NUMA, donde cada procesador tiene su propia memoria, pero en este caso la memoria compartida es sólo memoria caché [4, 5, 7].

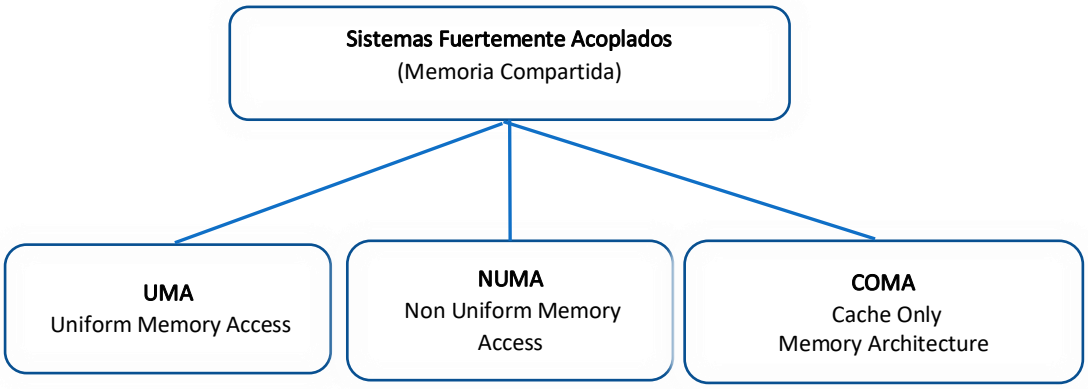

Fig. 1.5.1B Implementación de memoria en sistemas fuertemente acoplados

1.5.2 Paso de mensajes

Los sistemas *débilmente acoplados* presentan una distribución de la memoria con este esquema de comunicación entre los procesadores [12]. También se les conoce como sistemas de Memoria Distribuida [5], y consisten en que cada procesador tiene su propia memoria y ésta es privada, y la forma en cómo se comunican es mediante el paso de mensajes a través de una red de interconexión. La comunicación entre los procesos es punto a punto y unidireccional, se lleva a través de primitivas elementales de *send/receive* que se establen desde la programación del software que se desee ejecutar en dichas plataformas.

1.6 Arquitecturas Heterogéneas GPGPU

A principios del año 2000, la industria del hardware se bifurcó en dos grandes enfoques para proporcionar paralelismo a las aplicaciones y disminuir los tiempos de ejecución de los programas, siendo estos: el ***multicore*** y el ***multithreading*** [13, 14].

El multicore, con un enfoque de CPU modelo Von Neumann completo, basado en la idea de colocar varios CPU's en un solo chip, pero individualmente con procesamiento en serie, que al tratarse de CPU's completos serían de propósito general con la capacidad de realizar las operaciones complejas de optimización propias de un CPU. Mientras que el multithreading, basado en las Graphics Processing Units (GPU), consisten en arreglos masivos de unidades ALU's simples, con el objetivo de realizar operaciones de punto flotante a gran velocidad, bajo el esquema de sistemas fuertemente acoplados, y a pesar de que las GPU's surgieron a principios de los 80's, no fue sino hasta finales de los 90's con el auge de los videojuegos y las películas 3D, que se empezó a aplicar como coadyuvante del procesamiento general de las CPU's, con un enfoque de realizar las operaciones livianas para que el CPU se ocupará de procesamientos más complejos.

Ambos enfoques, el *multicore* y el *multithreading* presentaron su mayor rendimiento cuando se fusionaron en una arquitectura combinada tipo SIMD, a este tipo de arquitecturas hibridas CPU–GPU se les denominó GPGPU (General Purpose Programming Using a Graphics Processing Unit) [13, 14, 15], y hoy en día han proliferado tanto que es bastante común encontrarla hasta en equipos portátiles, ya sea cooperando independientemente o integradas como aceleradores gráficos. En la figura 1.6 se muestra un ejemplo de su topología estructural.

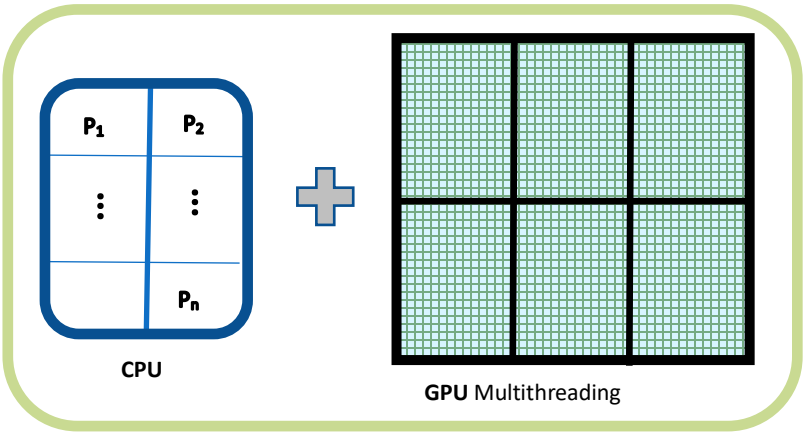

Fig. 1.6 Arquitectura GPGPU

1.6.1 CUDA

CUDA (Compute Unified Device Architecture) es un tipo de programa SPMD específico para tarjetas gráficas de la marca NVIDIA. Por el enfoque original de multithreading se le clasificó bajo el modelo SIMT (Single Instrucción Multiple Threads) que a la vez éste se basa en las arquitecturas SIMD, y fue diseñado para aprovechar al máximo las arquitecturas heterogéneas GPGPU. Por lo que, en la actualidad, ha proliferado extensamente en aplicaciones de problemas relacionados al paralelismo de datos, como renderización, machine learning, deep learning, ciencia de datos, robótica, tecnología 5G, IoT, juegos y vehículos autónomos entre muchas otras [16, 17].

CUDA soporta la combinación de distintos lenguajes de programación tradicionales, pero en este trabajo nos enfocamos a su aplicación en el lenguaje C. CUDA C es una extensión de las librerías del estándar ANSI C más las propias para explotar las arquitecturas GPGPU, incluyendo de igual manera las necesarias para la administración de los dispositivos y la memoria; todo bajo un modelo de programación modular y escalable [14, 16]. Dada su naturaleza hibrida, CUDA combina código para ser ejecutado tanto por la parte del CPU como por la GPU; siendo su compilador ***nvcc*** que está basado en ***LLVM***[2], el responsable de direccionar los fragmentos de código a uno u otro de los componentes de procesamiento CPU o GPU [13]. En la figura 1.6.1 se muestra un esquema de su integración y operación.

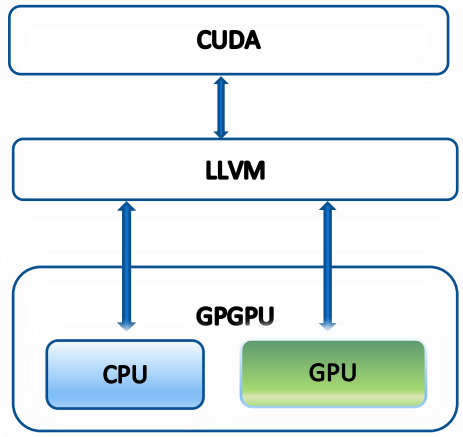

Fig. 1.6.1 Integración y operación de CUDA

1.6.1.1 Organización de la Memoria.

En CUDA, similar a la distribución de las jerarquías de memoria caché que se implementan en CPU's tradicionales, también se cuenta con distintos niveles de memoria que difieren en su capacidad de tamaños, velocidad de ancho de banda y latencias. En la figura 1.6.1.1, se muestra la representación esquemática de los 6 niveles de memorias que se implementan en CUDA.

[2] LLVM (Low Level Virtual Machine) Más que ser un acrónimo de máquinas virtuales, es una colección de proyectos de compilación de código reusable. Fuente: https://llvm.org/

En esta imagen que muestra la distribución de las memorias en las GPU's Nvidia, es importante observar que hay tres tipos de memoria que se encuentran fuera del Chip de cada Streaming Multiprocessor, y otros tres tipos que se encuentran dentro de él. También es importante observar el alcance que cada una de ellas tiene.

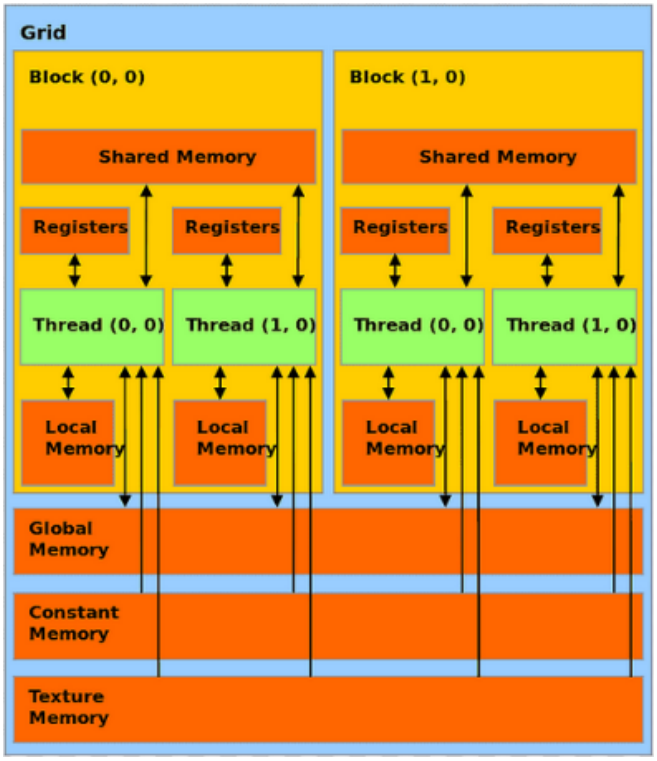

Fig. 1.6.1.1 Organización de la memoria en CUDA.

1.6.1.2 Estructura de programación.

En CUDA, al tratarse de un lenguaje diseñado para explotar al máximo las características de arquitecturas GPGPU, implementa un esquema *Host-Device*[3], donde se asume que la plataforma es híbrida y que por lo menos coexistirán un CPU y uno o varios dispositivos GPU's. Los programas en realidad contienen a la vez, código híbrido, declarándose desde el inicio con la directiva del preprocesador C *#include <cuda.h>*, que nos permitirá más adelante establecer explícitamente por medio de palabras clave, que parte deberá ser procesada por los GPU's. El compilador ***nvcc,*** al detectar dichas instrucciones, realiza la separación del código organizando la parte del GPU's en "*paquetes de procesamiento*" llamados ***kernels***, los cuales se declaran e invocan de la siguiente manera:

Declaración:

*__global__ void **funcionX** (int a, int b, int c) {/* código CUDA */}*

Invocación:

***funcionX** <<<n,m>>>( );*

Cuando se invoca a un kernel, se genera un conjunto de ***threads*** (hilos) que cooperan colectivamente para realizar el procesamiento del kernel, a este conjunto de threads se les conoce como ***grids*** (cuadriculas) y se corresponden con la arquitectura de hardware del GPU que se activa para llevar el procesamiento del kernel que fue invocado [13].

[3] Host-Device se corresponden con CPU y GPU respectivamente.

En las arquitecturas GPGPU, el Host y el Device tienen memorias separadas[4], por lo que, para la ejecución de un kernel, se deben implementar las siguientes acciones:

1. Reservar la Memoria en el CPU y en el GPU.
2. Copiar los datos del CPU al GPU.
3. Ejecutar el kernel en el GPU.
4. Copiar los resultados del GPU al CPU.
5. Liberar la memoria del CPU y del GPU.

1.6.1.3 Transformación de Arreglos Bidimensionales.

En CUDA, no se soporta el manejo de arreglos bidimensionales [16], por lo que es necesario manejarlos como un Vector. De tal manera que para referenciar al elemento $a_{i,j}$, de una matriz $A_{m,n}$ con dimensiones m,n, debemos transformar los índices a una expresión vectorial dada por la siguiente fórmula:

$$K=i*n+j$$

Por ejemplo, supongamos que tenemos una matriz A de dimensiones $A_{5,6}$, como se muestra en la figura 1.6.1.3A

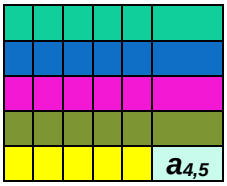

Fig. 1.6.1.3A Matriz $A_{5,6}$

Y supongamos que necesitamos referenciar al elemento a con índices $a_{4,5}$, nuestra expresión quedaría como se muestra en la figura 1.6.1.3B

$$K=4*6+5=29$$

Fig. 1.6.1.3B Transformación vectorial en CUDA.

[4] Existen arquitecturas GPGPU con memorias fusionadas y con sus propias librerías de programación.

1.7 Métricas del rendimiento

En esta sección se describen algunas métricas importantes para evaluar el desempeño de una implementación en paralelo.

1.7.1 Tiempo de ejecución

Es el tiempo transcurrido desde el inicio hasta el fin de un programa en todos los procesos que lo integran. Para medirlo, es importante tomar como parámetro la media de un número considerable de mediciones, para contrarrestar la imprecisión debido al no determinismo de la elección de los procesos por el Scheduler del sistema operativo [2, 18, 19].

1.7.2 Costo

Cuantifica la cantidad total de trabajo que realiza cada uno de los procesadores que participan en la ejecución de un programa [2]. El costo de un programa paralelo que tiene tiempo de ejecución T_p (n), con tamaño de entrada n y ejecutado en p procesadores se expresa como C_p y se define como sigue:

*C_p (n)= número de procesadores * tiempo de ejecución con p procesadores*

$$C_p\,(n) = p \cdot T_p(n)$$

1.7.3 Speedup (Aceleración)

Esta medida indica la ganancia de velocidad que se obtiene al paralelizar una aplicación, con respecto a su implementación final y se define como sigue:

$$S_p(n) = \frac{tiempo\ de\ ejecución\ serial}{tiempo\ de\ ejecución\ paralela} \qquad S_p(n) = \frac{T_s\,(n)}{T_p\,(n)}$$

donde *p* representa al número de procesadores que se usaron para resolver un problema paralelo con tamaño de entrada *n*, y $T_s(n)$ es la aceleración absoluta, es decir, el tiempo de ejecución correspondiente a la mejor implementación secuencial para resolver el mismo problema.

1.7.4 Eficiencia

La eficiencia indica que tan bien se están utilizando los recursos computacionales del sistema [1, 2]. Sean $T_s\,(n)$, $T_p(n)$, que denotan el mejor tiempo de ejecución serial y el tiempo de ejecución paralela cuando se usan *p* procesadores y se considera un tamaño de entrada *n* entonces, entonces la eficiencia $E(n)$ se define como:

$$E(n) = \frac{tiempo\ de\ ejecución\ serial}{(tiempo\ de\ ejecución\ paralela\ ) * (numero\ de\ procesadores)}$$

También pueden ser descrita como:

$$E(n) = \frac{T_s\,(n)}{T_P(n) \cdot p} * 100\%$$

1.7.5 Escalabilidad

Es la métrica que nos permite conocer el comportamiento de la eficiencia de una solución paralela incrementando el número de procesadores empleados (Escalabilidad de hardware) o el incremento del problema, es decir; la cantidad de datos procesados (Escalabilidad Algorítmica) [2, 4, 19].

Intuitivamente, la escalabilidad de hardware se refiere al incremento de procesadores empleados mientras que la escalabilidad algorítmica es muy compleja de medir; dado que su comportamiento no es lineal y éste depende de la cantidad de datos a procesar y el número de pasos necesarios para poderlo lograr. Esto la convierte en una cualidad intrínseca del problema que se esté analizando; por ejemplo, en una multiplicación de matrices, si se duplica su tamaño, el número de pasos necesarios para resolverla se cuadruplica, a continuación, se describirán las dos leyes más conocidas que abordan esta temática.

1.7.5.1 Ley de Amdahl.

Determina la aceleración máxima S_p de un algoritmo en una plataforma paralela con un tamaño fijo de su capacidad de trabajo [2]. Asimismo, se establece que la aceleración tiende a la proporción serial que no se puede paralelizar cuando se incrementa el número de procesadores *P*. Por lo tanto, la aceleración de un programa paralelo queda compuesta del tiempo de ejecución de la fracción serial, *f* y el tiempo de ejecución de la fracción paralela $\frac{(1-f)}{P}$, quedando expresada de la siguiente manera:

$$S_p(n) = \frac{T_s\,(n)}{[f * T_s\,(n) + \left(\left(\frac{1-f}{P}\right) * T_s\,(n)\right)]} = \frac{1}{(f + \frac{(1-f)}{P})} \leq \frac{1}{f}$$

donde P representa al número de procesadores utilizados para resolver un problema de tamaño *n.*

1.7.5.2 Ley de Gustafson.

Está Ley fue formulada John L. Gustafson y Barris en 1988, como una alternativa a la limitante establecida en la Ley de Amdahl, donde la aceleración máxima S_p está limitada por la parte serial que integra el programa, pero con una carga fija de trabajo. Gustafson y Barris, analizaron que muchos de esos procesadores que trabajaban la parte paralela, no ocupaban todo su tiempo, es decir, tenían espacios de espera muy prolongados. Por lo que propusieron aumentar la carga de trabajo a los procesadores, de esa manera el trabajo

realizado por la parte paralela crece y la fracción serial decrece, y por lo tanto la aceleración mejora [2, 7], quedando expresada de la siguiente manera:

$$S_p \leq p + (1-p)\, t_s$$

donde p es el número de procesadores y t_s es el porcentaje de tiempo total que una aplicación toma en la ejecución serial.

1.8 Autómatas Celulares

Como este trabajo se enfoca en implementaciones en paralelo para modelos de autómatas celulares, se proporciona una introducción al concepto del mismo, para una mejor comprensión de lo que se realizó.

1.8.1 Antecedentes

La Teoría de Autómatas Celulares (aquí referidos como AC) tiene su origen en la teoría de Autómatas de Estado Finito definida a finales de los 40's por John Von Neumann en su publicación titulada "The General and Logic Theory of Automata" [20].

Von Neumann, inspirado por los propios avances sobre teoría de campos, probabilidad, expresiones regulares y lógica matemática que presentaban sus colegas matemáticos contemporáneos, como Kurt Gödel, Warren S. McCulloch, Walter Pitts, Norbert Wiener y S. C. Kleene, entre muchos más, se enfocó desde principios de los 50's en crear un modelo capaz de auto reproducirse. Sin embargo, su trabajo se vio concretado hasta que Konrad Zuse y Stanislaw Ulam introdujeron un enfoque topológico basado en un espacio bidimensional; donde las entradas mismas de cada elemento consistían en los valores de los elementos vecinos y estos valores pertenecían al número finito de posibilidades que podía tomar cada elemento comprendido en el espacio definido. Sin embargo, este campo del conocimiento no mostró auge durante varios años, hasta que a principios de los 70's Martin Gardner presentó un diseño hecho por el matemático británico John Horton Conway, denominado "Game of Life" (GoL)[5] [21].

El ***Juego de la Vida*** alcanzó de inmediato una gran aceptación en la comunidad científica porque resolvía en parte, una proeza planteada por el mismo Von Neumann, de encontrar una máquina universal de Turing factible algorítmicamente de computarse. Además, este modelo matemático presentaba dos cualidades indispensables para entender fenómenos naturales: ***emergencia*** y ***autoorganización*** [22].

También, en el Juego de la Vida, Conway identificó que, dadas las condiciones iniciales del modelo, este tendía a un patrón de comportamiento evolutivo. Dicha identificación inspiró años más tarde a Stephen Wolfram, otro matemático británico, a realizar una exhaustiva identificación y clasificación de Autómatas Celulares Elementales con 2 y 3 estados;

[5] GoL, en lo sucesivo nos referiremos a este modelo en español "El Juego de la Vida".

culminando parcialmente su trabajo en el 2002, con la publicación del libro "*New Kind of Science*" [23].

1.8.2 Definición

No existe una definición matemáticamente exacta de los AC, pero en esencia, se establecen los elementos que los conforman, la forma en cómo se organizan, las reglas de operación, un estado inicial (predefinido o aleatorio) y sus posibles valores de estados; siendo estos, una colección de elementos simples organizados homogéneamente con una topología y dimensión establecida denominada el espacio o la malla celular. Donde cada elemento comprendido en la colección es un Autómata de Estado Finito (célula) que toma sus valores de entrada de sus vecinos y tiene definido un rango finito de estados posibles, además; cambia sus valores dada una regla global y de aplicación simultánea, por lo que son estructuras dinámicas discretas temporal y espacialmente [20], quedando expresados de la siguiente manera:

$$A = (L, S, N, f)$$

donde:

L (Lattice):

Se refiere a la dimensión *d* del espacio celular en el que evoluciona el sistema a través del tiempo, donde $d \in Z+$, aunque en la práctica, son universalmente empleadas las dimensiones d=1 y d=2.

S (States):

Especifica el conjunto finito de estados por los que los elementos en L pueden iterar en cada transición.

N (Neighborhood):

Determina el vector $N=(n_1, n_2 \ldots n_m)$ de vecinos en la dimensión d, donde la célula *n* tiene *m* vecinos.

f (function):

Es la función de transición local $\boldsymbol{f}: S^m{}_i \rightarrow Sn_{(i+1)}$ o simplemente "*La Regla*" para cada celda con vecinos *m* y estados *S*.

1.8.3 Vecindad

Son el conjunto de células adyacentes a la célula $\boldsymbol{C_{i,j}}$[6], con radio r; donde r es el número de elementos adyacentes a $\boldsymbol{C_{i,j}}$ a considerar dentro de la vecindad ***N***. Los valores de cada uno de los elementos de la vecindad fungen como entrada a la función que determina la

[6] $\boldsymbol{C_{i,j}}$ Notación válida solamente para d =2, es decir; un espacio bidimensional.

evolución de $C_{i,j}$. La Figura 1.8.3 muestra un ejemplo de las vecindades de Neumann y Moore con radio r =1.

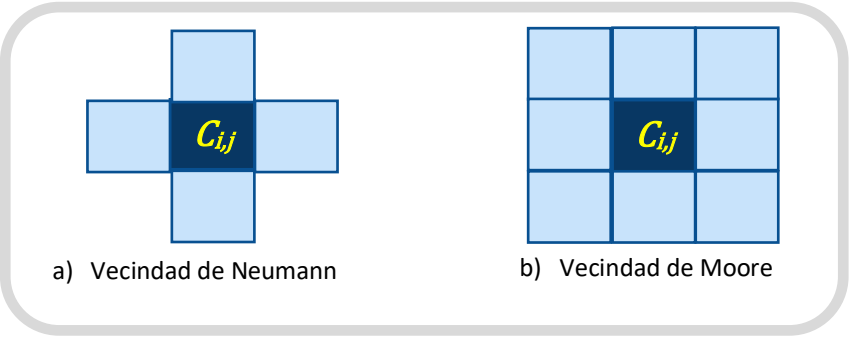

Fig. 1.8.3 Vecindades de una Célula, con *r = 1.*

1.8.4 Condiciones de Frontera

En los límites del espacio celular, para la definición de las vecindades se establecen condiciones especiales llamadas *condiciones de frontera* [24], que prácticamente consisten en establecer los criterios para completar la vecindad de las celdas de la periferia; ya sea agregando *r* celdas adicionales (virtuales) para completar la vecindad de radio *r* que se definida en el vector N, o tomando valores de otras celdas existentes en el espacio celular L. De acuerdo con esos criterios, se clasifican como:

a) *Frontera fija*: Consiste en agregar r celdas virtuales con un valor predefinido.

b) *Frontera adiabática*: Se establece el valor del estado de la celda del límite en las r celdas virtuales necesarias para establecer su vecindad.

c) *Frontera reflectiva*: Se replican los valores de las r celdas adyacentes definidas en L, a las r celdas virtuales agregadas para completar la vecindad.

d) *Frontera periódica*: Similar a la anterior, pero en este caso las r celdas virtuales tomarán el valor de las r celdas existentes en L del lado opuesto, creando un anillo o un toroide, para dimensiones d=1 y d=2 respectivamente.

En la figura 1.8.4 mostramos una representación de las fronteras descritas.

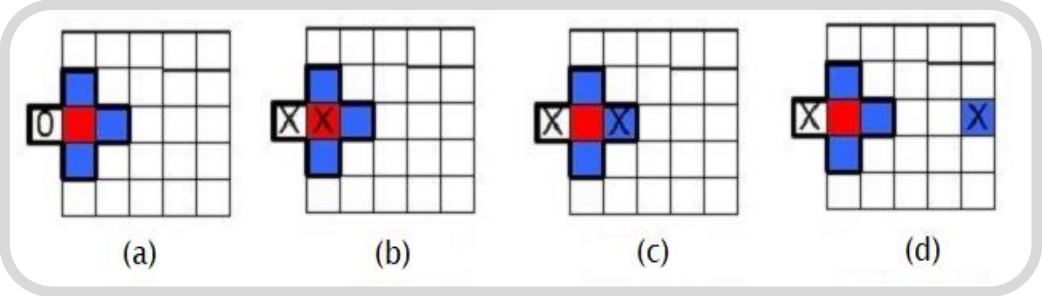

Fig. 1.8.4 Fronteras de un Autómata Celular

1.8.5 El Juego de la Vida

Uno de los AC más representativos y conocidos, es el Juego de la Vida. Fue diseñado por John Horton Conway[7] y publicado por Martin Gardner en la revista Scientific American en 1970, como un pasatiempo matemático que obedece al esquema de la máquina universal de Turing [21]. Sin embargo, este *"pasatiempo"* (como él mismo lo clasificó) cobró mucho interés en la comunidad científica debido a las propiedades emergentes y de autoorganización que proporcionó, mismas que desde entonces han sido útiles para comprender el comportamiento complejo de múltiples sistemas de diversas disciplinas del conocimiento, entre ellas biología, física, mecánica, economía y ciencias sociales, entre muchas tantas más [22].

El Juego de la Vida, es un Autómata Celular bidimensional que establece un conjunto de estados de dos valores posibles para cada una de sus células, siendo estos VIVO o MUERTO y su evolución en cada generación dependerá del valor que tengan sus vecinos definidos bajo una vecindad de Moore y de la función de transición aplicada simultáneamente a cada célula. La función de transición se compone de las reglas para nacimientos, muertes y supervivencias [20, 21], que se describe a continuación:

1. Nacimiento:
   Se presenta cuando una célula muerta, tiene exactamente 3 vecinos vivos.

2. Supervivencia:
   Permanecerán vivas las células que tengan 2 o 3 vecinos vivos.

3. Muerte:
   Morirán las células que presenten cualquiera de los siguientes escenarios:
   a) Sí tienen 1 o ninguna célula vecina viva (Por soledad) y
   b) Sí presentan 4 o más vecinos vivos (Por sobrepoblación).

La evolución de este modelo de Autómata va a depender no únicamente de lo descrito anteriormente, sino también y muy relevante mencionarlo, del ***estado inicial*** con el que sea configurado; dado que dependiendo de ese valor el sistema puede presentar los siguientes comportamientos que fueron mencionados por el mismo Conway como "*divertidos patrones de comportamiento evolutivo*" [21], siendo estos: (a) patrones de oscilación, (b) patrones estáticos, (c) patrones tipo nave espacial y (d) patrones repetitivos longevos.

[7] John Horton Conway, matemático británico de la Universidad de Cambridge, su legado más relevante fue en el campo de Teoría de Números, Teoría de Campos, Teoría de Juegos y Teoría de Códigos.

1.8.6 Aplicaciones de los Autómatas Celulares

En esencia, los AC se integraron a las herramientas matemáticas disponibles para modelar Sistemas Complejos [21, 22], mismos que surgían desde distintas disciplinas, pero las más significativas en cuanto al empleo de estas técnicas durante muchos años fueron la epidemiología, la inmunología, la medicina, la biología, la física y las matemáticas.

En la actualidad, es casi indispensable su conocimiento en el estudio de Sistemas Complejos, ya sea para modelar comportamientos exclusivamente con AC o combinándolos con otras técnicas, y recientemente han tenido un gran auge en áreas de las ciencias sociales, geografía, finanzas y cifrado de información.

1.9 Trabajos relacionados

En lo siguiente se describen algunos trabajos relacionados con el trabajo que aquí se propone.

1.9.1 Implementación del *Juego de la Vida* empleando NUMA

En [25], Sánchez Solchaga realizó un estudio de análisis del rendimiento de la memoria en arquitecturas hibridas tipo NUMA, empleando la metodología de diseño de algoritmos paralelos propuesta por Ian Foster y Autómatas Celulares como modelo de medición, en su trabajo se enfocó a la dependencia de los programas con la arquitectura de hardware donde se ejecutan para mejorar la eficiencia.

Cómo se describió en la introducción, los Autómatas Celulares son modelos matemáticos intrínsicamente paralelos, pero se debe tomar muy en cuenta la desventaja que representa la granularidad de cada celda; dado que al tratarse de granularidad fina se aumentan las comunicaciones interprocesos. En el trabajo de Solchaga, se cuidó bastante ese aspecto de acuerdo con la metodología empleada, haciendo para tal efecto, una descomposición del dominio de grano grueso; qué significa agrupar celdas continuas para ser tratadas como un macroproceso que será enviado a un nodo en específico de la arquitectura NUMA y éste a su vez, dentro de ese nodo, lanzará los hilos necesarios para aprovechar al máximo la memoria compartida que tenga asignada, logrando disminuir las comunicaciones y aprovechar de mejor manera el tiempo del procesador.

Otro aspecto relevante para el desempeño que se tomó en cuenta fue el mapeo explícito de procesos a la arquitectura y la reservación explicita de la memoria requerida; logrando con ello acoplar proceso-nodo y empleando más eficientemente la memoria compartida de cada nodo.

Las pruebas del rendimiento paralelo que se realizaron en el trabajo de Solchaga fueron combinadas empleando tanto MPI para paso de mensajes, como OpenMP para la memoria compartida; así como la librería libnuma para la asignación explicita de memoria distribuida, para lo cual, se llevaron a cabo las siguientes implementaciones:

a) Base: se trata de una versión paralela híbrida que combina el empleo de memoria distribuida con memoria compartida, pero deja al propio sistema que realice la reserva de la memoria empleada. No se consideró el mapeo proceso-nodo.

b) Implícita: En esta prueba, se consideró el mapeo proceso-nodo de acuerdo con la topología del sistema, obligando al sistema operativo a realizar una reservación de memoria en el mismo donde se estén ejecutando los procesos, evitando así las latencias de la comunicación.
c) Explícita: En este caso, se consideró tanto el mapeo proceso-nodo como la asignación explicita de memoria a cada proceso a fin de incrementar la localidad.

Se obtuvo como resultado la demostración de que si se emplea adecuadamente la arquitectura de hardware donde se ejecutan los procesos, se puede tanto mejorar el rendimiento como la eficiencia. Sin embargo; en este trabajo se consideraron arquitecturas hibridas fuertemente acopladas y débilmente acopladas, pero compuestas de CPU Von Neumann completas, no se realizaron pruebas empleando arquitecturas GPGPU.

1.9.2 Implementación del Juego de la Vida en GPU

En [26], Millán y Wolovick, realizaron un análisis del Juego de la Vida en arquitecturas GPU's, con un enfoque principal basado en el empleo de la memoria integrada a la tarjeta GPU, variando el tipo y los patrones de acceso, para comprobar que la eficiencia del rendimiento depende del modelo empleado en cada arquitectura GPU, para lo cual, las mediciones se hicieron en cinco arquitecturas diferentes de GPU de la familia NVIDIA a través de las cuatro implementaciones, descritas a continuación:

1) *Baseline*: En esta implementación se emplea la memoria global del dispositivo en cada evolución, los hilos van por datos, realizan los cálculos y vuelven a regresar a la memoria a actualizar los valores, es evidente que se incrementan las comunicaciones interprocesos.
2) *Shared v1*: A diferencia de la primera, en este modelo cada hilo copia sus datos a la memoria compartida y realiza los cálculos, es decir, implementan granularidad fina.
3) *Shared v2*: También de memoria compartida como el anterior, con la diferencia que en este modelo se cambia la granularidad del dominio.
4) *Cells*: Esta implementación se basa en que cada hilo comparte 6 valores de los 8 vecinos que le rodean, de esa forma disminuye la comunicación y aumenta el rendimiento.

En este trabajo se concluye que como trabajos futuros se podrían hacer pruebas que incluyan arquitecturas heterogéneas CPU multicore y GPU's, siendo justo ese el objetivo de nuestro proyecto.

Capítulo **2**

Desarrollo del Modelo

Aunque los AC son muy simples y no requieren de una gran capacidad de cómputo para poder realizarlos, cuando se usan para modelar sistemas con un gran número e partículas o células, el número de operaciones derivadas de las simulaciones pueden consumir mucho tiempo de cómputo. Por lo que recientemente, el desarrollo de implementaciones de modelos de AC mediante el uso de arquitecturas paralelas orientadas a reducir los tiempos de ejecución y a manejar de manera eficiente grandes espacios de dominio, ha generado mucho interés.

En este capítulo, se presenta una propuesta para el diseño y desarrollo de implementaciones paralelas de modelos de AC mediante el uso de GPUs y CUDA, a fin de permitir el manejo de sistemas a gran escala. El *objetivo* es *realizar un análisis del rendimiento de implementaciones paralelas en una arquitectura heterogénea moderna basada en GPUs de uno de los modelos de AC más conocidos, el Juego de la Vida, con respecto a aquella en una arquitectura secuencial convencional tipo CPU.* De tal manera que las implementaciones paralelas que se proponen no sólo reduzcan los tiempos de ejecución, sino que utilicen de manera adecuada los recursos que se disponen y la jerarquía de memoria disponibles mediante el uso de NVIDIA CUDA.

2.1 Pseudocódigo secuencial.

El Juego de la Vida se toma como modelo base en este trabajo de tesina, ya que la implementación secuencial y paralela de este modelo brindan las bases para el desarrollo de modelos de AC más complejos. A continuación, se muestran el pseudocódigo secuencial del modelo "*El Juego de la Vida*" y una malla bidimensional que representa un fragmento del espacio celular donde se observa una célula $C_{[i,j]}$ con vecindad de Moore r=1 y los índices de los vecinos que influyen en su evolución.

```
for every generation in Life do
    for every Cell in DomainSpace do
        Neighbors ← 0
        for every Cell in Neighborhood do
            Neighbors ← Neighbors + Cell [i, j]
        end for
        if (Cell [i, j] is alive and neighbors == 3)
            Cell[t+1] ← ALIVE
        else if (Cell is alive and (neighbors == 2 or neighbors == 3))
            Cell[t+1] ← ALIVE
        else
            Cell[t+1] ← DEAD
        end if
    end for
end for
```

i-1, j-1	i-1, j	i-1, j+1
i, j-1	$C_{i,j}$	i, j+1
i+1, j-1	i+1, j	i+1, j+1

2.2 Implementaciones desarrolladas.

En lo siguiente se describen las implementaciones desarrolladas para el estudio que se realizó. Se consideraron 3 implementaciones: una secuencial, una nombrada CUDA implícita y una nombrada CUDA explícita, las cuales se describen a continuación.

Implementación	Descripción
Secuencial	Se desarrolló una versión secuencial del modelo, para que, a partir de este, poder perfilar las zonas de paralelización y nos sirviera como referencia en las métricas de comparación.
CUDA implícita	Se desarrolló una versión paralela de CUDA empleando el modelo por default de la Memoria Global con que cuenta la tarjeta GPU.
CUDA explicita	Se desarrolló una versión paralela de CUDA empleando la Memoria Shared que implementa la tarjeta GPU.

2.3 Perfilado del programa.

Se le conoce como el perfilado de un programa a la identificación de los segmentos de código secuencial más costosos computacionalmente hablando, para que con vistas a realizar una optimización se analice la factibilidad de paralelizarlos.

A continuación, se muestran los dos segmentos del código más costosos que fueron detectados en nuestro modelo secuencial.

Bloque 1: *Sumatoria de los estados de los vecinos para una celda* $A_{i,j}$

```
int obtenerVecinosVivos()
    {
            vecinosVivos  =  espacioCelular (A[i-1][j-1])
            vecinosVivos += espacioCelular (A[i-1][j])
            vecinosVivos += espacioCelular (A[i-1][j+1])
            vecinosVivos += espacioCelular (A[i][j-1])
            vecinosVivos += espacioCelular (A[i][j+1])
            vecinosVivos += espacioCelular (A[i+1][j-1])
            vecinosVivos += espacioCelular (A[i+1][j])
            vecinosVivos += espacioCelular (A[i+1][j+1])
     }
```

Bloque 2: *Reglas de Evolución*

```
If (espacioCelular(A[i][j]))
    {
            If(vecinosVivos < 2  || vecinosVivos > 3 ){
                    A[i][j] = 0
            }
    } else {
            If(vecinosVivos == 3)
    {
            A[i][j] = 1
    }
}
```

2.4 Desarrollo de las implementaciones paralelas implícita y explicita.

Para las dos implementaciones paralelas que se desarrollaron, tanto para la memoria global como para la memoria shared, se crearon las funciones kernel que fueron mencionadas en el punto anterior; con el fin de enviar a la tarjeta GPU la mayor cantidad de cálculos computacionales y optimizar el modelo. En lo siguiente, se muestran las firmas de las funciones kernel que fueron desarrolladas para el dispositivo GPU.

```
__global__ void calcularFronteras(int DIMENSION, int *d_espacioCelular)
    {
       // código cuda C.
    }

__global__ void evolucionarModelo(int DIMENSION, int *d_C1, int *d_C2)
    {
       // código cuda C.
    }
```

Asimismo, fue necesario hacer una transformación del espacio celular bidimensional a una representación vectorial, en ambos casos; los índices de cada una de las células se calcularon como se expresa en el siguiente fragmento de código:

```
int coordenadaX = blockDim.x * blockIdx.x + threadIdx.x + 1;
int coordenadaY = blockDim.x * blockIdx.x + threadIdx.x;
```

Capítulo **3**

Resultados obtenidos

En este capítulo, se presentan las evaluaciones obtenidas de los desarrollos realizados del modelo de AC, tanto de la implementación serial como las dos versiones en paralelo; la explicita con memoria global y la implícita con memoria compartida.

3.1 Características de la plataforma

A continuación, se presenta una tabla que detalla las características de la plataforma donde se ejecutaron las pruebas y de donde se obtuvieron los resultados que se publican en este trabajo.

Hardware	Software
Workstation Dell Precision T920 Procesador: Intel Xeon Silver 42162, 3.2 GHz, 16C, 22MB Caché. RAM: 32GB, DDR4 2666MHz. GPU: Nvidia GeForce RTX 2080 super. Memoria: GDDR6 8 GB Cuda cores: 3072	**Sistema Operativo:** GNU/Linux Slackware 14.2 (64 bits) GCC versión 5.5.0 CUDA 11.0 Capability 7.5 Driver versión: 450.66 Warp Size: 32

Tabla 3.1 *Características de la plataforma empleada.*

3.2 Pruebas de evaluación

En la tabla siguiente, se describen las pruebas que se llevaron a cabo para cada una de las implementaciones empleadas para la recolección de resultados. Se consideraron 5 tamaños de la entrada del problema con la finalidad de evaluar de mejor manera el desempeño de las implementaciones y sus limitaciones.

Dimensión	Total, de elementos	Espacio (por arreglo)
512x512	262,144	1MB
1024x1024	1,048,576	4 MB
2048x2048	4,194,304	16 MB
4096x4096	16,777,216	64 MB
8192x8192	67,108,864	256 MB
16384x16384	268,435,456	1 GB

Tabla 3.2 *Pruebas realizadas.*

3.3 Resultados de Runtime

Como punto inicial, se hizo un análisis de tiempo de ejecución considerando diferentes tamaños de entrada del problema. En la Tabla 3.3 se muestran los resultados obtenidos del tiempo de ejecución expresado en segundos para las tres implementaciones consideradas. Estos resultados se muestran graficados en la figura 3.3A.

Dimensión	**Serial** (s)	**Cuda Global Memory** (s)	**Cuda Shared Memory** (s)
512x512	9.97131	0.01287	0.00574
1024x1024	40.44972	0.04462	0.01771
2048x2048	159.25496	0.14727	0.06230
4096x4096	633.18595	0.46421	0.24220
8192x8192	2,665.90789	1.62815	0.99064
16384x16384	10,651.69519	6.42101	4.06012

Tabla 3.3 *Tiempo promedio de 10 resultados de la ejecución (segundos)*

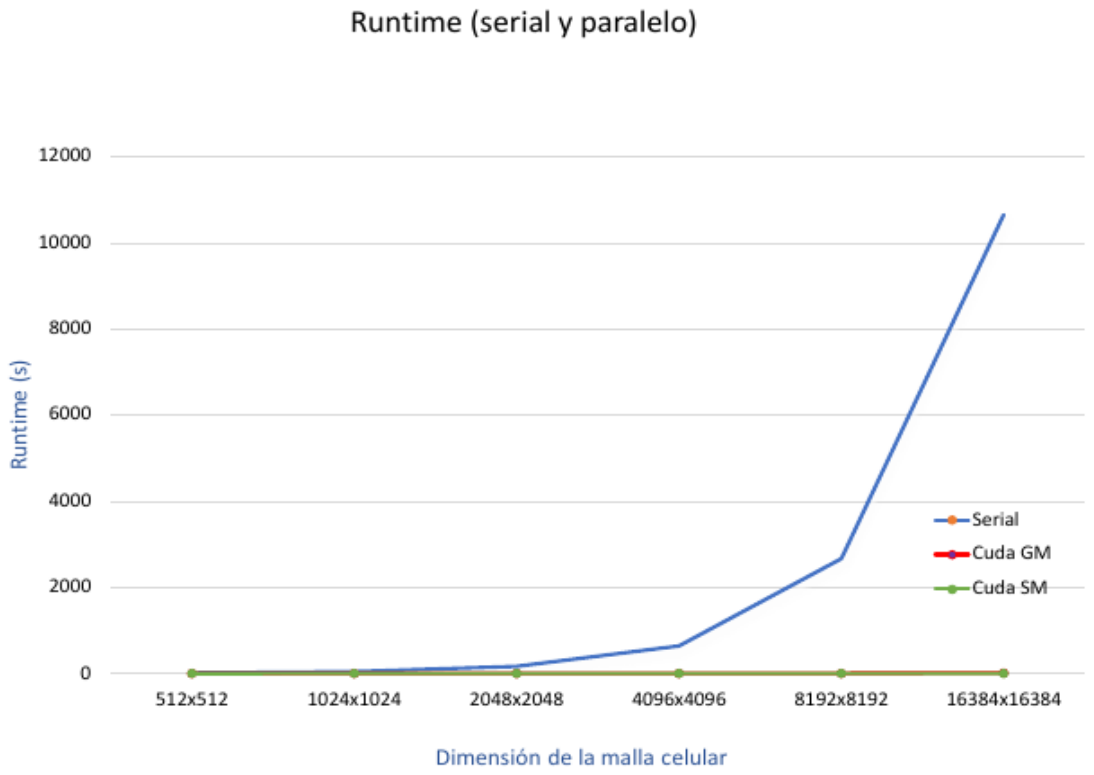

Fig. 3.3A Gráfica de los tiempos de ejecución de las implementaciones serial y paralelas.

Como se puede observar de la figura 3.3A, los tiempos que corresponden a las implementaciones paralelas, no son representativos en esta comparación. Por lo que en la figura 3.3B, se muestran el tiempo de ejecución con respecto al tamaño de la entrada, para las dos implementaciones paralelas consideradas en este trabajo. Nótese que la ganancia de rendimiento obtenida en la implementación explicita con Cuda Shared Memory respecto a la implícita que fue desarrollada con Cuda Global Memory. Se observa que la brecha que marca la diferencia entre los tiempos de ejecución en ambas implementaciones no es demasiada grande, sin embargo, es bastante ilustrativa para representar como sería el comportamiento de ambas implementaciones

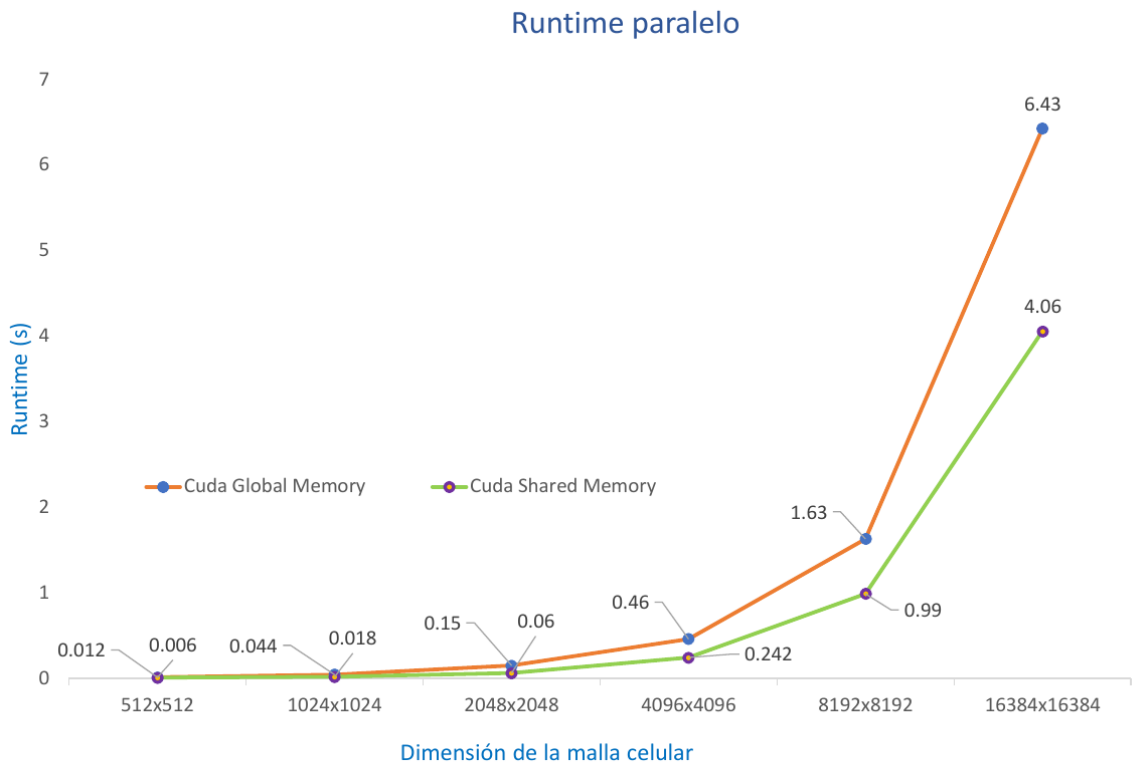

Fig. 3.3B Gráfica de los tiempos de ejecución de las implementaciones paralelas.

3.4 Resultados de Speedup.

Posteriormente se analizó el Speedup (aceleración) para las diferentes implementaciones consideradas. La métrica lo que nos indica es que tanto crece la aceleración de una implementación secuencial al ser paralelizada, se mide como el factor de aceleración X o simplemente el Speedup.

En la Tabla 3.4 se muestran los resultados obtenidos para los diferentes tamaños considerados como entrada del problema, tanto para Cuda Global Memory y Cuda Shared Memory.

Dimensión	Cuda Global Memory	Cuda Shared Memory
512x512	774.6	1,736.9
1024x1024	906.5	2,284
2048x2048	1,081.3	2,556.2
4096x4096	1,364	2,614.3
8192x8192	1,637.3	2,691
16384x16384	1,658.8	2,623.4

Tabla 3.4 *Factor de aceleración X*

Estos mismos resultados se muestran graficados en la figura 3.4, de donde se aprecia la diferencia del Factor de Aceleración que presentan ambas implementaciones paralelas, lo cual nos muestra que en ambos casos el rendimiento mejora conforme aumenta la carga de trabajo. Sin embargo, al incrementarse paulatinamente la carga se aprecia que llega a un límite superior del cual ya no hay mejora, y ese comportamiento es debido a los limites propios de los tamaños de las memorias en las tarjetas GPU's; lo cual indica que en otras tarjetas se obtendrían resultados similares con distintas cargas de trabajo, dependiendo propiamente de cada tarjeta GPU.

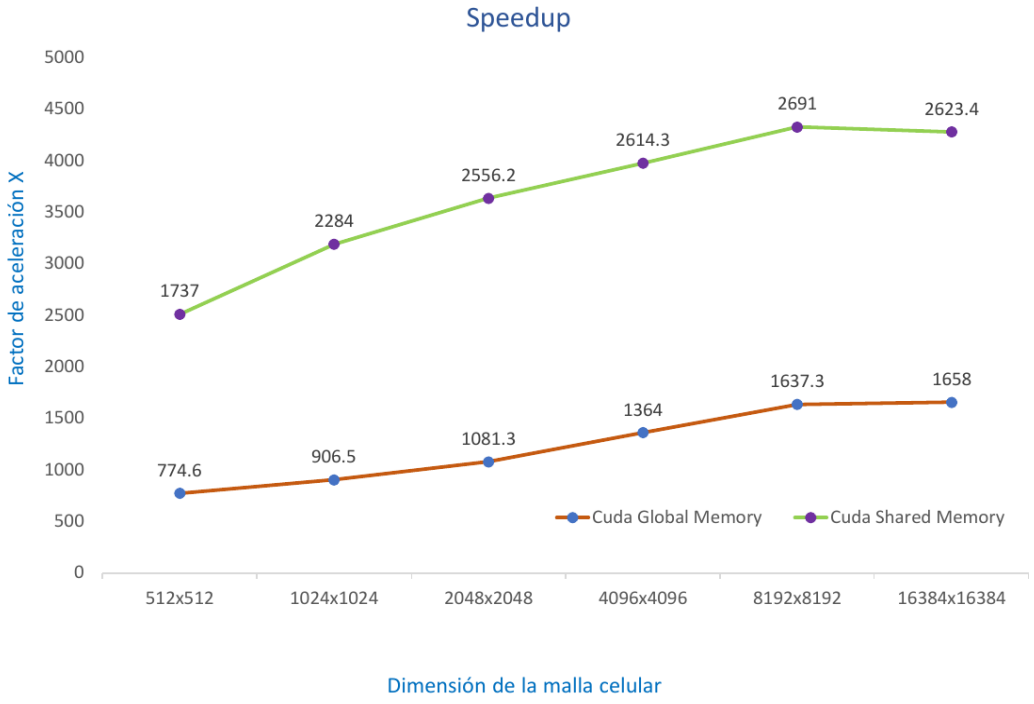

Fig. 3.4 Gráfica del Factor de Aceleración de las implementaciones paralelas.

3.5 Resultados de Eficiencia

También se analizó la eficiencia de los dos tipos de implementaciones desarrolladas y diferentes tamaños de la malla. En la Tabla 3.5 se muestran los resultados correspondientes a esta métrica, asimismo se muestran graficados en la figura 3.5, de donde se puede percibir un comportamiento congruente con la medición del punto anterior, ya que también mejora la eficiencia conforme se aumenta la carga de trabajo, hasta que llega a un punto límite máximo distinto para cada implementación desarrollada.

Dimensión	Cuda Global Memory %	Cuda Shared Memory %
512x512	25.2	56.5
1024x1024	29.5	74.3
2048x2048	35.2	83.2
4096x4096	44.4	85.1
8192x8192	53.3	87.6
16384x16384	54.0	85.4

Tabla 3.5 *Eficiencia paralela respecto a la ejecución serial.*

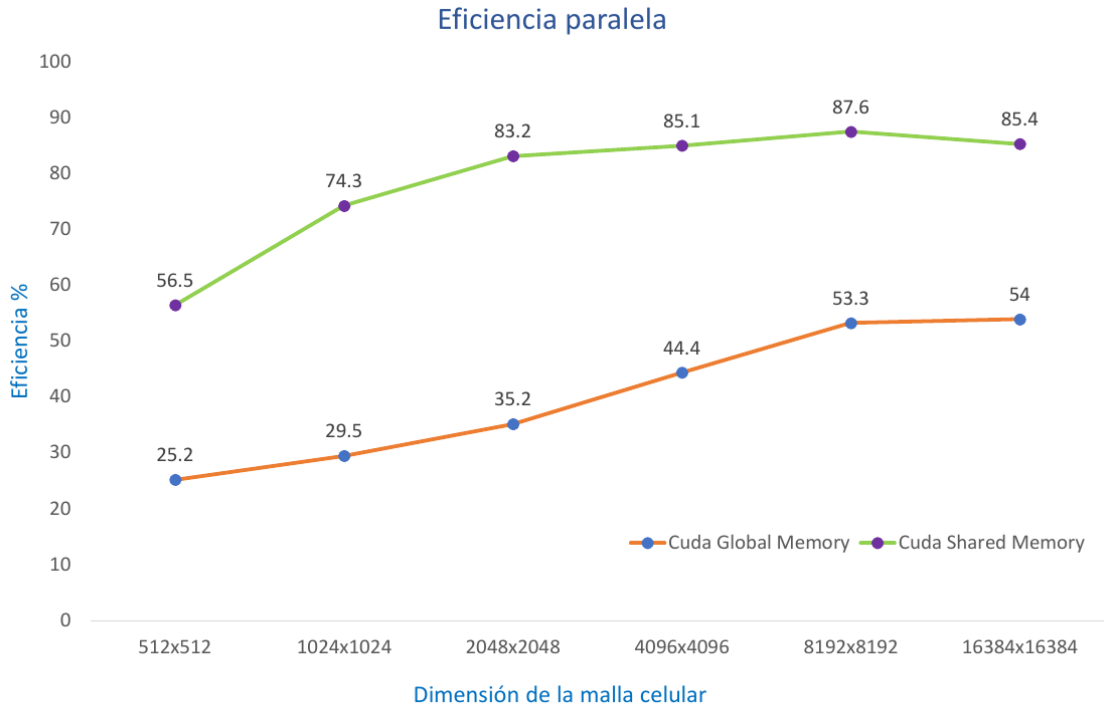

Fig. 3.5 Gráfica de la Eficiencia de las implementaciones paralelas.

Capítulo **4**

Conclusiones y trabajo futuro.

4.1 Conclusiones.

En esta tesina se realizó un análisis del rendimiento de implementaciones paralelas en una arquitectura heterogénea moderna basada en GPUs de uno de los modelos de AC más conocidos, el Juego de la Vida, con respecto a aquella en una arquitectura secuencial convencional tipo CPU. Para ello se diseñaron dos implementaciones paralelas con la finalidad de manera adecuada los recursos que se disponen y la jerarquía de memoria disponibles mediante el uso de NVIDIA CUDA. Así se realizó un análisis del rendimiento de implementaciones paralelas en una arquitectura heterogénea moderna basada en GPUs de uno de los modelos de AC más conocidos, el Juego de la Vida, con respecto a aquella en una arquitectura secuencial convencional tipo CPU. De tal manera que, que las implementaciones paralelas que se proponen no sólo reduzcan los tiempos de ejecución, sino que utilicen de manera adecuada los recursos que se disponen y la jerarquía de memoria disponibles mediante el uso de NVIDIA CUDA. Mediante simulación computacional se analizaron diferentes tamaños del Juego de la vida y el desempeño computacional del mismo.

Los resultados indican que, desde la perspectiva única del rendimiento, las implementaciones paralelas mediante las tarjetas GPU's de la marca Nvidia y su framework de programación CUDA, ofrecen una alternativa bastante atractiva a considerarse favorablemente cuando se requieren realizar optimizaciones paralelizando los programas, y no se cuenta con acceso a un clúster de grandes dimensiones de CPU multicore. Además, incluso mejora el rendimiento si se considera y emplea desde la conceptualización de la programación, las características de memoria que tienen las tarjetas GPU's Nvidia donde serán ejecutadas las simulaciones.

También se concluye que en los escenarios donde se cuente con clusters con arquitecturas heterogéneas incluyendo múltiples nodos multicore y a la vez con tarjetas GPU's Nvidia, es posible realizar implementaciones que sean capaces de aprovechar de mejor manera los recursos disponibles realizando para el efecto, programación hibrida para cada una de las partes de hardware disponibles.

4.2 Trabajos Futuros

A continuación, se presentan algunas sugerencias que se consideran factibles para tomarse como líneas de estudio con el fin de profundizar en estos temas de pruebas de rendimiento en arquitecturas heterogéneas.

- Desarrollar modelos de medición que involucre la totalidad de tipos de memoria que tienen las tarjetas GPU Nvidia.

- Desarrollar modelos de comparación en otras arquitecturas de GPU con otros frameworks, como por ejemplo los de la familia AMD y FPGA.
- Desarrollar modelos de medición híbridos para clusters de GPU's, ya que, en la actualidad, es cada vez más común este tipo de arquitecturas donde se incluyen múltiples nodos con CPU's multiprocesor y a la vez con varias tarjetas GPU's, por lo que se considera pertinente realizar modelos de medición que empleen tecnologías hibridas con MPI, OpenMP y CUDA.
- Y como consecuencia del punto anterior, y considerando la complejidad que representa el realizar modelos híbridos de programación, sería bastante interesante que se abriera un proyecto que tuviera como objetivo principal encapsular en un solo framework, todo lo necesario para que dado un programa serial, éste fuera capaz de paralelizarlo de la mejor manera posible solamente detectando el hardware disponible y el código secuencial otorgado, de esa manera se apoyaría más significativamente a los investigadores en sus modelos y simulaciones que requieran ejecutar.

Referencias

[1] P. Pacheco, An introduction To Parallel Programming, Morgan Kaufman, 2011.

[2] M. A. Barry Wilkinson, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2 ed., Pearson, 2004.

[3] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2003.

[4] G. &. W. G. Hager, Introduction to High Performance Computing for Scientists and Engineers, Chapman & Hall/CRC Computational Science, 2011.

[5] Hesham&Mostafa, Advanced Computer Architecture and Parallel Processing, Wiley-Interscience, 2005.

[6] S. G. Shiva, Computer Organization, Design and Architecture; 4 edition, CRC Press, 2007.

[7] S. G. Shiva, Advanced Computer Architectures, 1st Edition, Kindle Edition, CRC Press, 2018.

[8] M. Flynn, «Some Computer Organizations and Their Effectiveness,» IEEE Trans. Comput., C-21: 948, 1972.

[9] G. D. N. V. P. G. Darema F, «A single-program-multiple-data computational model for EPEX/FORTRAN.,» de *Parallel Computing*, 1988; 7(1):11–24.

[10] M. G. T. M. John Cheng, CUDA C Programming, Wrox, 2014.

[11] C. G. v. E. T. K. C. Chang CC, «Evaluating the performance limitations of MPMD Communication. InProceedings of the 1997,» de *Conference on Supercomputing ACM/IEEE* , NewYork,NY,USA, 1997; 1–10.

[12] J. L. Ortega Arjona, Estudio y Evaluación de la Programación Orientada a Objetos en una Arquitectura Paralela, Cd. Mex.: UNAM, 1996.

[13] W.-m. W. H. David B. Kirk, Programming Massively Parallel Processors: A Hands-on Approach 3rd Edition, Morgan Kaufmann, December 21, 2016.

[14] S. T. &. P. Collet, Massively Parallel Evolutionary Computation on GPGPUs (Natural Computing Series), Kindle Edition, Springer, December 5, 2013.

[15] R. V. S. B. J. C. W.-m. H. Hyesoon Kim, «Performance Analysis and Tuning for General Purpose Graphics Processing Units 1st Edition,» Morgan & Claypool Publishers, November 26, 2012.

[16] J. S. &. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming 1st Edition, Addison-Wesley Professional;, July 19, 2010.

[17] NVIDIA, «CUDA-Developers,» [En línea]. Available: https://developer.nvidia.com/cuda-zone. [Último acceso: 21 Julio 2020].

[18] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd Edition,, Springer, 2011.

[19] J. Shun, Shared Memory Can Be Simple, Fast, and Scalable, ACM Books, 2017.

[20] J. Kari, Cellular Automata, Springer, 2013.

[21] G. Martin, «The Fantastic Combinations of John Conway’s New Solitaire Game "Life".,» *Scientific American,* nº 223, p. 120–123, 1970.

[22] H. Sayama, Introduction to the Modeling and Analisys of Complex Systems, Binghamton University: Open Textbooks, SUNY, 2015.

[23] WOLFRAM Science, «Some Historical Notes,» [En línea]. Available: https://www.wolframscience.com/reference/notes/876b. [Último acceso: 28 Julio 2020].

[24] J. C. &. Gursaran, «Cellular Automata,» de *International Journal of Scientific & Engineering Research*, London, England, 2013.

[25] M. E. S. S. &. M. E. L. Ramírez, Diseño y optimización de implementaciones de modelos de Autómatas Celulares para Arquitecturas Paralelas con Acceso no uniforme a Memoria., Cd. Mex.: UNAM, 2018.

[26] E. W. P. F. G. C. B. Millán, «Performance analysis and comparison and cellular automata GPU implementations.,» *CLUSTER COMPUTING-THE JOURNAL OF NETWORKS SOFTWARE TOOLS AND APPLICATIONS,* vol. 20, pp. 1-15, 2017.