UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

“IMPLEMENTACIÓN DE AUTÓMATAS CELULARES CON ARQUITECTURA GPU”

PROYECTO FINAL

QUE PARA OBTENER EL GRADO DE

ESPECIALISTA EN CÓMPUTO DE ALTO RENDIMIENTO

PRESENTA.

GUILLERMO ALONSO BARRÓN

Director de tesina:

DRA. MARÍA ELENA LÁRRAGA RAMÍREZ

Instituto de Ingeniería, UNAM

Ciudad Universitaria, Ciudad de México.   Noviembre, 2018

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Tabla de contenido

Introducción. 4
Capítulo 1. 7
Marco Teórico. 7
1.1 Definición de los autómatas celulares. 7
1.2 Elementos principales de los autómatas celulares 7
1.3 Aplicaciones de los autómatas celulares 9
1.3.1 El Juego de la Vida 10
1.4 Conceptos básicos del cómputo de alto desempeño 12
1.4.1 Arquitecturas paralelas 12
1.5 Arquitecturas de memoria compartida. 13
1.5.1 Unidades de Procesamiento Gráficas (GPU, Graphics Processing Units) . 13
1.6 Métricas para evaluación del desempeño. 15
1.6.1 Tiempo de ejecución. 15
1.6.2 Costo. 15
1.6.3 Aceleración o Speedup. 15
1.6.4 Eficiencia. 15
1.6.5 Ley de Amdahl. 16
1.6.6 Ley de Gustafson. 17
1.6.7 Escalabilidad. 18
Capítulo 2 Trabajos relacionados. 19
2.1 Implementacion de GoL usando NUMA 19
2.2 Un análisis del uso de cuatro tarjetas GPU. 20
Capítulo 3 Desarrollo del modelo de AC basado en GPU. 21
3.1 Caso de estudio. 21
3.1.1 El Juego de la Vida. 21
3.2 Implementación paralela. 22
3.2.1 Estructura general. 22
3.2.2 Configuración inicial. 23
3.2.3 Memoria en GPU y transferencia de datos. 23
3.2.4 Funciones Kernel 24
3.2.5 Puntos de sincronización. 25
Capítulo 4 Resultados obtenidos. 27

4.1 Características de la plataforma de ejecución. 27
4.2 Descripción de los parámetros de entrada. 28
4.3 Evaluación experimental. 28
4.3.1 Tiempo Acumulado. 28
4.3.3 Aceleración 29
Conclusiones y trabajo futuro. 32
Trabajo futuro. 32

Introducción.

En las últimas décadas, el incremento continuo del poder computacional ha permitido la extensión de la aplicación de las metodologías computacionales en la investigación y la industria, así como el estudio cualitativo de fenómenos con comportamiento complejo. Uno de los paradigmas bien establecidos para el estudio, entendimiento y análisis de la dinámica de este tipo de sistemas son los Autómatas Celulares (de aquí en adelante referidos como AC).

De manera informal se puede decir que los AC son sistemas dinámicos, discretos tanto en el espacio como en el tiempo, formados por un gran número de elementos simples y homogéneos llamados celdas, dispuestas uniformemente generando una estructura topológica regular *d*-dimensional denominado enmallado; en donde cada una de sus celdas, puede representar, no simultáneamente, un número finito *k* de estados específicos. Dichas celdas, interactúan entre sí, con las celdas dentro de su vecindad, a través de un conjunto de reglas locales de transición que representan la dinámica del sistema que se está modelando. Así, los modelos de AC se han usado en varias áreas de investigación porque involucran modelos simples que se pueden usar para simular comportamientos complejos, que incluyen materiales metálicos, flujo de agua, solidificación de estructuras de grano, modelos de crecimiento urbano, simulaciones de corazón, propagación de incendios, detección de bordes en imágenes, etcétera.

Particularmente, la computación de alto rendimiento (HPC por sus siglas en inglés) ha permitido resolver problemas de gran magnitud mediante AC que eran inaccesibles antes del advenimiento de software innovador que toma la ventaja del hardware potente del HPC. Por lo que en los últimos años, se han realizado varios estudios enfocados a la implementación paralela de diferentes modelos basados en AC para procesar gran cantidad de información en un menor tiempo, en comparación con sus versiones secuenciales, y permitir manejar espacios de dominio grandes que no sería posible manejar de manera secuencial. Estos estudios se han hecho con base en el uso de diferentes arquitecturas de cómputo paralelas, tales como clúster [1, 2, 3], procesadores multicore y manycore [1, 2, 4, 5, 6], GPUs [4, 6, 7, 8, 9, 10], FPGAs [7, 10, 11, 12], etc., y su combinación [2, 9].

En el trabajo realizado por Rybacky [6], se hace una comparación entre varias implementaciones de modelos AC, evaluados en arquitecturas multicore y unidades de procesamiento gráfico (GPU por sus siglas en inglés). Se encontró que los algoritmos basados en multicore son, en su mayoría, superados por los basados en GPU, pero en algunos problemas más específicos ocurre lo contrario. Por lo que se concluye que, si bien es cierto, los GPU son una buena alternativa para el desarrollo de implementaciones paralelas, su utilidad depende del modelo a simular. Bezbradica [13] realizaron un estudio sobre estrategias de paralelización de autómatas celulares para la industria farmacéutica. Para esto, utilizan implementaciones de memoria compartida haciendo uso del API OpenMP, de memoria distribuida usando la librería de paso de mensajes MPI y un híbrido de ambas implementaciones. Los resultados muestran que la implementación híbrida ofrece mejor rendimiento, seguida de la implementación desarrollada en MPI. También

sugieren realizar mejoras al experimentar con métodos de balanceo de carga. Posteriormente, Millán et. al en [14] analizan el rendimiento de implementaciones paralelas tanto para el modelo de AC conocido como “Juego de la Vida” (GoL,Game of Life por sus siglas en inglés), como de un modelo Lattice Boltzman, en varias arquitecturas. En ese estudio, utilizaron contadores de hardware, esto para verificar la eficiencia del algoritmo basado en los accesos de memoria local, además, estudiaron la escalabilidad de la implementación donde para una malla de 64000 x 64000 y 1000 generaciones alcanzaron ~56x de aceleración con ~87% de eficiencia. También realizaron, aunque someramente, un análisis de los costos de comunicación entre los procesos remotos al hacer escalamiento, aunque solo se limita a reducir el espacio del dominio hasta un punto que no afecte el rendimiento. Por otra parte, Sánchez Solchaga [15], en su tesis de maestría, propone tres implementaciones paralelas híbridas de modelos de AC a gran escala para arquitecturas paralelas de acceso no uniforme a memoria (NUMA, No Uniform Memory Access), utilizando varias configuraciones de dominio, en las que analiza el desempeño en términos de escalabilidad y eficiencia. Además considera la estructura topológica del sistema donde se ejecutan, lo que permite utilizar de manera adecuado los recursos existentes. En su trabajo concluye que al utilizar de mejor manera los recursos proporcionados por la arquitectura se puede mejorar el rendimiento se las implementaciones paralelas de los modelos a gran escala basados en AC. Recientemente, Millán et. Al [16] realizaron un análisis comparativo del desempeño de cuatro arquitecturas de GPU’s realizando implementaciones paralelas del modelo del GoL. Sin embargo, las implementaciones que utilizan no son a gran escala, concluyendo que el rendimiento de las implementaciones va depender de las características de cada arquitectura de GPU.

Con el objetivo de dar continuidad al trabajo realizado en [15] para sistemas a gran escala, este trabajo de tesina se enfoca en realizar la implementación paralela del Juego de la Vida usando aceleradores de hardware GPU para analizar su desempeño. Para este propósito se hará uso de una tarjeta gráfica NVIDIA Tesla K40 y se desarrolla una implementación con una función kernel y dos funciones device.

Este trabajo se presenta organizado en cuatro capítulos organizados de la siguiente manera:

Capítulo 1. Marco Teórico. En este capítulo se presenta una introducción al paradigma de modelado de los AC, así como su definición, propiedades y se describen brevemente algunas aplicaciones. Así mismo se describen conceptos básicos sobre Cómputo de Alto Rendimiento (HPC, High Performance Computing), tales como las principales arquitecturas, métricas para la evaluación del desempeño de aplicaciones paralelas.

Capítulo 2. Trabajos relacionados. En este capítulo, se analizan algunos trabajos de interés que se relacionan con este proyecto final, estos trabajos fueron seleccionados por ser recientes en el área.

Capítulo 3 Desarrollo del modelo de AC basado en GPU. En este capítulo, se presenta una propuesta para el diseño de implementación paralela del modelo de AC. El

objetivo principal es realizar una implementación de un AC conocido, que se usa como caso de estudio, utilizando de una tarjeta gráfica.

Capítulo 4 Resultados obtenidos. En este capítulo, se presenta la evaluación de la propuesta de implementación paralela en GPU para el modelo de AC presentado en el capítulo anterior, para esto se realizan y analizan los resultados obtenidos en diversos experimentos, con base a estos resultados se realiza la comparativa de ambas implementaciones, secuencial y en GPU.

Finalmente, se presentan las conclusiones obtenidas con base en los resultados de simulación obtenidos y su análisis.

Capítulo 1.
Marco Teórico.

Con base en el objetivo de este trabajo, y para un mejor entendimiento del mismo, en este capítulo se introducen los conceptos básicos necesarios para ello. Primeramente, se introducen los Autómatas Celulares (AC), se describen sus propiedades y algunos conceptos relacionados. También se mencionan brevemente algunas de sus aplicaciones. Por otro lado, también se describen algunos conceptos básicos sobre el cómputo de alto rendimiento (HPC, High Performance Computing, por sus siglas en inglés), a fin de tener una base para comprender este trabajo. Se presentan algunos aspectos principales relacionados, como las arquitecturas y métricas para la evaluación de aplicaciones paralelas.

1.1 Definición de los autómatas celulares.

Los AC son sistemas dinámicos discretos tanto en el tiempo como el espacio, formados por un gran número de elementos simples y homogéneos llamados celdas, dispuestas uniformemente generando una estructura topológica regular de d-dimensiones llamado enmallado, en donde cada una de sus celdas, puede representar, no simultáneamente, un número finito k de estados específicos. Dichas celdas, interactúan entre sí, con las celdas dentro de su vecindad, a través de un conjunto de reglas locales que representan la dinámica del sistema que se está modelando.

1.2 Elementos principales de los autómatas celulares

Los elementos básicos de un autómata celular son:

a) Espacio celular regular. La estructura topológica del enmallado depende de la cantidad de dimensiones del espacio de dominio (ver Figura 1), como de la forma de las celdas (ver Figura 2) las cuales pueden tener una de tantas formas, donde la elección se basa en el objetivo del estudio a realizar.

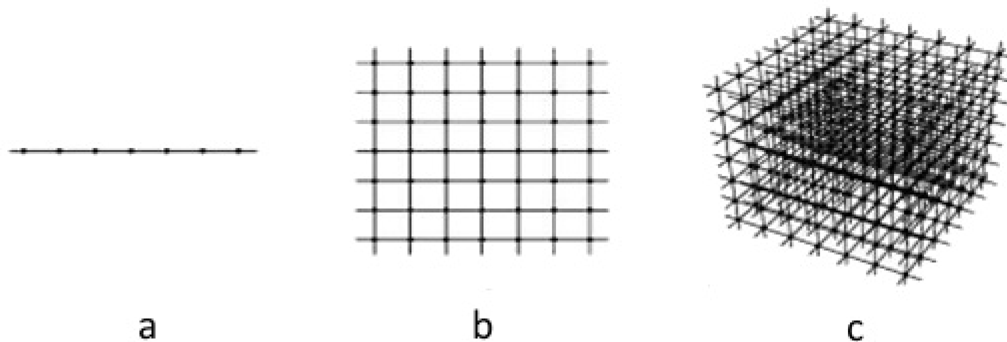


Figura 1. Espacio celular en 1, 2 y 3 dimensiones en el espacio celular de un AC con celda cuadrada.

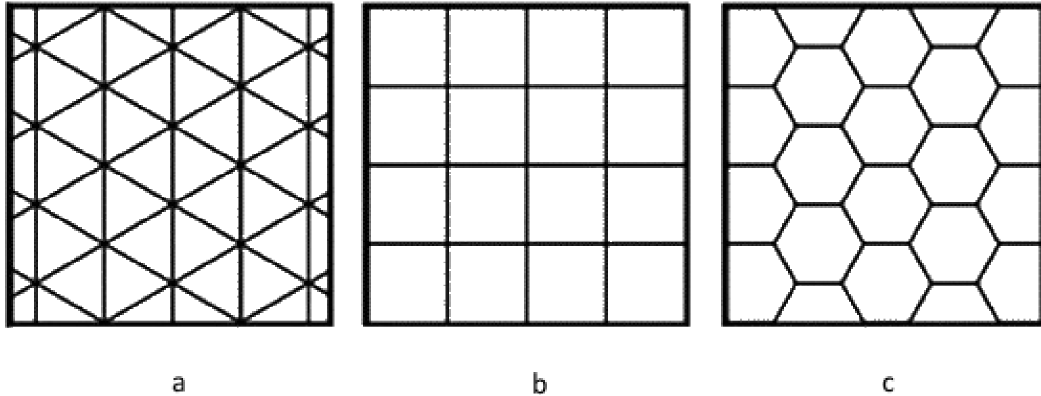


Figura.2. Algunas formas de celdas y la estructura topológica generada por las mismas en el espacio celular de un AC bidimensional.
a) Triangular, b) Cuadrada, c) Hexagonal

b) Conjunto de Estados. Es finito y cada elemento toma un valor de ese conjunto, puede denominarse también alfabeto. Puede ser expresado en valores o colores.
c) Configuración Inicial. Es la asignación inicial de valores a cada elemento o célula del espacio.
d) Vecindades. Son las celdas o células adyacentes que son capaces de influenciar la evolución de una celda específica (comúnmente se incluye a la misma celda). Una vecindad no cuenta con restricción de tamaño o ubicación, solo que debe ser la misma para todas las celdas en el AC. En la Figura 3 se muestran las vecindades más utilizadas.

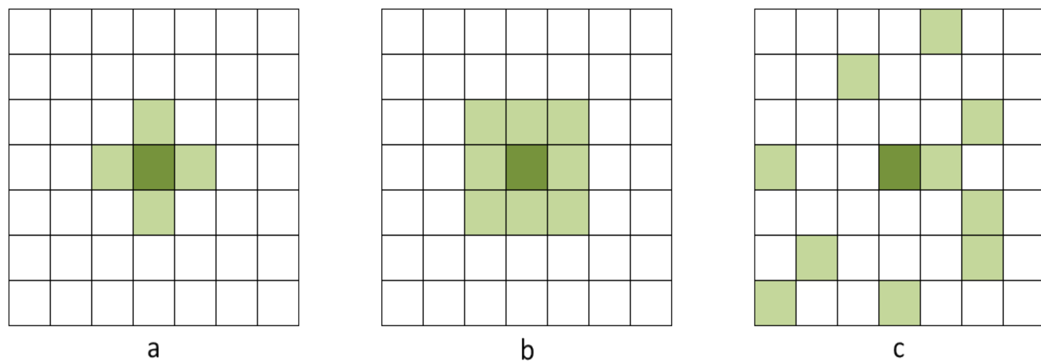


Figura 3. Algunos tipos de vecindades en el espacio celular de un AC con celda cuadrada.
a) Von Neumann, b) Moore, c) Aleatoria

e) Función de Transición Local. La evolución del estado de cada celda en el AC está definida por un conjunto de reglas de transición, las cuales son usadas por cada una de las celdas en el espacio de dominio en cada paso de tiempo o generación actualizando su estado al siguiente tiempo, con respecto al estado actual de las celdas en su vecindad.

f) Condiciones de frontera. En teoría el espacio de dominio es infinito. Pero en la práctica el enmallado siempre es acotado, por este motivo, en lugar de tener una regla de transición diferente para las celdas de los límites, se extiende la vecindad utilizando celdas virtuales que permitan completar la vecindad. Para definir los estados que utilizarán las celdas virtuales se utilizan varias condiciones de frontera, a saber (ver Figura 4):

   i. Frontera fija: es definida al completar la vecindad con valores fijos en las celdas virtuales.
   ii. Frontera adiabática: es definida al duplicar el valor de la celda en la celda virtual.
   iii. Frontera reflectiva: es definida por replicar el valor de la celda opuesta en el la celda virtual.
   iv. Frontera periódica: es definida por replicar el valor de la celda ubicada en el extremo opuesto en la celda virtual, formando un toroide.

   Dependiendo de la condición de frontera usada, serán los resultados que nos dé el AC.

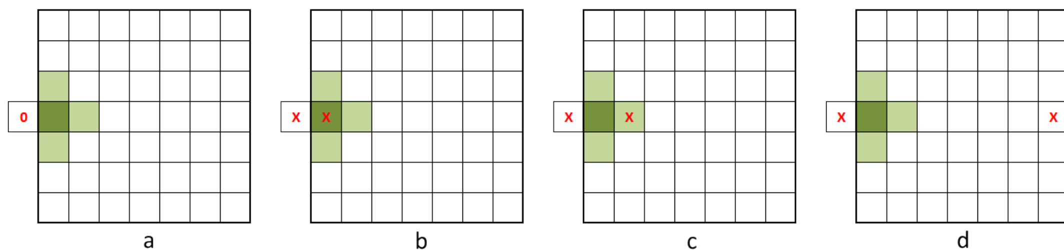


Figura 4. Condiciones de frontera de un AC.
a) Fija, b) Adiabática, c) Reflectiva, d) Periódica

g) Condición inicial. Es el valor inicial de cada celda que define el comportamiento inicial del AC y, por lo tanto, afecta el comportamiento del modelo.

A pesar que de los AC son simples, no es tan fácil analizar su comportamiento desde un inicio, a menos que se recurra a la simulación, partiendo de un estado inicial e ir cambiando los estados de todas las celdas de forma síncrona con forme pasa el tiempo, lo hace altamente complejo y difícil de predecir.

1.3 Aplicaciones de los autómatas celulares

Los autómatas celulares presentan un enfoque muy poderoso para el estudio de fenómenos emergentes complejos creados a partir de muchas interacciones locales simples. Debido a esto, los AC son muy populares para estudiar aplicaciones reales y han generado un gran interés tanto en el mundo académico como en la industria por parte de investigadores que trabajan en diferentes campos científicos y se ocupan de aspectos teóricos y aplicaciones prácticas. Así, los AC se reconocen como una de las herramientas

más importantes para el modelado de sistemas complejos en diversas áreas como la geografía [17-19], para componer música [20], para encriptación de información [21], detección de bordes en imágenes de interés, entre muchas otras.

1.3.1 El Juego de la Vida

El juego de la vida (referido como GoL por sus sigla en inglés), es el ejemplo de AC más conocido. Fue formulado por el matemático británico John H. Conway en 1970. Desde un punto de vista teórico, es interesante porque es equivalente a una máquina universal de Turing, es decir, todo lo que se puede computar algorítmicamente se puede computar en el juego de la vida.

Desde su publicación, ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones. Se considera que la vida es un buen ejemplo de emergencia y autoorganización. La autoorganización, que refiere a una forma global de orden que surge de la interacción de los componentes individuales de un sistema inicialmente desordenado, es interesante para los científicos matemáticos, economistas y otros, para observar cómo patrones complejos pueden provenir de la implementación de reglas muy sencillas.

El Juego de la vida no es una cuestión filosófica, es un sencillo juego con tres reglas muy simples donde nadie gana ni pierde, pero que, entre otras muchas cosas, nos puede orientar sobre la evolución de distintas formas de vida o, incluso, de la evolución de una epidemia. Así, en el juego de la vida se simulan células vivas y muertas en una cuadrícula (enmallado) bidimensional, en donde cada celda (*i, j*) o célula puede tener uno de dos estados (vivo o muerto). El estado de la celda cambiará en el siguiente paso de tiempo, dependiendo del estado actual de cada celda y su entorno (vecindad) que lo formarán las ocho celdas que la rodean (esto se conoce como un entorno de Moore), de acuerdo al siguiente conjunto de reglas locales:

a) **Muerte por soledad:** Una celda viva con menos de dos vecinos vivos, muere.
b) **Supervivencia:** Una celda viva con dos o tres vecinos vivos, permanece viva para la siguiente generación.
c) **Muerte por sobrepoblación:** Una celda viva con más de tres vecinos vivos, muere.
d) **Nacimiento por reproducción:** Una celda muerta con exactamente tres vecinos vivos, nace .

La configuración inicial es la primera generación del sistema, la segunda generación se forma al aplicar las reglas anteriores a cada una de las celdas de manera simultánea y así sucesivamente para crear las generaciones posteriores (ver Figura 5).

Teóricamente, el enmallado es infinito, pero para poderlo representar en una computadora se utiliza una malla finita con una condición de frontera periódica, lo que significa que, para completar la vecindad de las celdas del borde del enmallado, se consideran los valores de las celdas de la posición opuesta del enmallado.

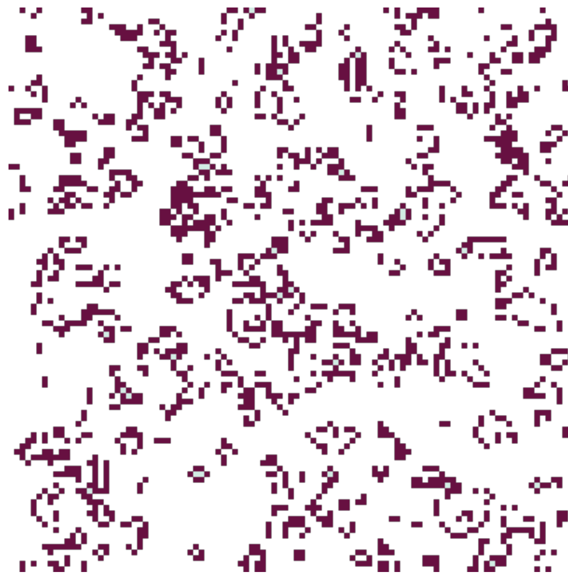


Figura 5. Configuración de un estado de tiempo del GoL.

Veamos un ejemplo simple. En el espacio siguiente, se han pintado de verde las células vivas.

	1	2	3	4	5
A					
B					
C					
D					
E					

Si nos fijamos en la célula A1, como en su entorno solo hay una célula viva, en la siguiente etapa A1 morirá; la célula A2, en cambio, nacerá, puesto que tiene 3 células vivas (verdes) en su entorno; las células A3, A4, A5 y B1, se quedan iguales porque ninguna de ellas tiene exactamente 3 células vivas en su entorno; la B2, morirá por tener más de 3 células vivas en su entorno... y así, sucesivamente.

	1	2	3	4	5
A					
B					
C					
D					
E					

Así, este simple juego puede generar tantas formas de 'vida artificial' diferentes. Evidentemente, además de ser un juego entretenido y adictivo, el juego de la vida ha dado

lugar a investigaciones mucho más interesantes y se usa para simular comportamientos naturales de evolución. Para ello se usan distintos espacios celulares, se definen distintas células, distintos entornos, se permiten infinitos estados, etc.

1.4 Conceptos básicos del cómputo de alto desempeño

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (en paralelo). A pesar de que el paralelismo se ha utilizado desde hace años, recientemente, el interés por él ha aumentado debido a las limitaciones que existen para procesar grandes cantidades de datos en equipos secuenciales.

Un sistema paralelo en cluster combina múltiples sistemas de cómputo, conectados mediante una red de alta velocidad y un esquema de programación específico para obtener un poder de cómputo semejante a una supercomputadora. El éxito de los clusters de computadoras se debe a su habilidad de combinar múltiples sistemas de cómputo, de tal manera que proveen de capacidades de procesamiento que una sola computadora no podría resolver.

En las últimas décadas, la investigación del cómputo de alto rendimiento incluyó nuevos desarrollos en tecnologías paralelas de hardware y software, cada una con características específicas y ventajas que las hacen especiales para resolver cierto tipo de problemas; las cuales se clasifican de diferentes maneras: algunas en base al número de instrucciones concurrentes y en los flujos de datos disponibles en la arquitectura, otras según el nivel de paralelismo que admite el hardware, o en base a su organización de memoria. A continuación, se presentan algunas clasificaciones, así como algunas de las plataformas convencionales actuales de interés para este trabajo de las cuales es necesario conocer las características que estas tecnologías brindan.

El Cómputo de Alto Rendimiento (High performance Computing o HPC en inglés) es la agregación de potencia de cálculo para resolver problemas complejos en ciencia, ingeniería o gestión. Para lograr este objetivo, la computación de alto rendimiento se apoya en tecnologías computacionales como los clústers, los supercomputadores o la computación paralela. La mayoría de las ideas actuales de la computación distribuida se han basado en la computación de alto rendimiento.

Actualmente, el Cómputo de Alto Desempeño (HPC por sus siglas en inglés) permite realizar experimentos que por medios convencionales podrían resultar costos, peligrosos o incluso imposibles de realizar. No obstante, hacer aplicaciones paralelas no es tarea fácil ya que la concurrencia da lugar a posibles errores que hay que considerar. Por este motivo hacer simulaciones eficientes, escalables y que den un alto desempeño no es tan fácil.

1.4.1 Arquitecturas paralelas

En 1972 Michael J. Flynn propuso una de las primeras clasificaciones de las arquitecturas de cómputo paralelas conocida como *taxonomía de Flynn* [22], en esta se

clasifica a las arquitecturas de cómputo respecto al flujo de las instrucciones que se ejecuta una plataforma y en la secuencia de datos disponible en la arquitectura. A continuación se muestran las categorías principales:

- **SIMD – Single Instruction, Multiple Data.** Se tiene un flujo único de instrucciones, ya sea en un procesador (núcleo), el paralelismo es proporcionado operando múltiples flujos de datos de manera simultánea. Ejemplos de esta categoría son las unidades de procesamiento gráfico GPU.
- **MIMD – Multiple Instruction, Multiple Data.** Varias secuencias de instrucciones en varios procesadores que operan en diferentes elementos de datos de manera simultánea.
- **SISD – Single Instruction, Single Data.** Es la ejecución normal, es decir, no paralela de un solo procesador.
- **MISD – Multiple Instruction, Multiple Data.** Es donde múltiples procesadores operan sobre un arreglo de datos, esta categoría se considera un paradigma útil en la práctica.

1.5 Arquitecturas de memoria compartida.

Consisten en un conjunto de Unidades de Proceso (UP) que comparten una única memoria global. Los módulos de memoria, que forman parte de la memoria global, proveen un espacio de direcciones común, el cual es accedido por todas las UP del sistema. Las UP pueden intercambiar los datos haciendo uso de la memoria, al escribir y leer variables compartidas.

1.5.1 Unidades de Procesamiento Gráficas (GPU, Graphics Processing Units)

Estos procesadores fueron diseñados principalmente para acelerar tareas de gráficos como la presentación de imágenes y surgieron a principios de los años 80's pero eran muy costosos. Con el paso del tiempo, a finales de los 90's, se popularizaron por los videojuegos ya que estos incluían más y mejores imágenes que requerían mayor procesamiento; con esto los costos se redujeron mucho, lo que permitió ser una tecnología más asequible.

En sus inicios se conformaba por un arreglo de hardware que era configurable pero no programable, la tarjeta gráfica recibía comandos y datos desde el CPU y los procesaba principalmente con librería de OpenGL y DirectX. En la figura 6 se muestra un ejemplo de esta estructura.

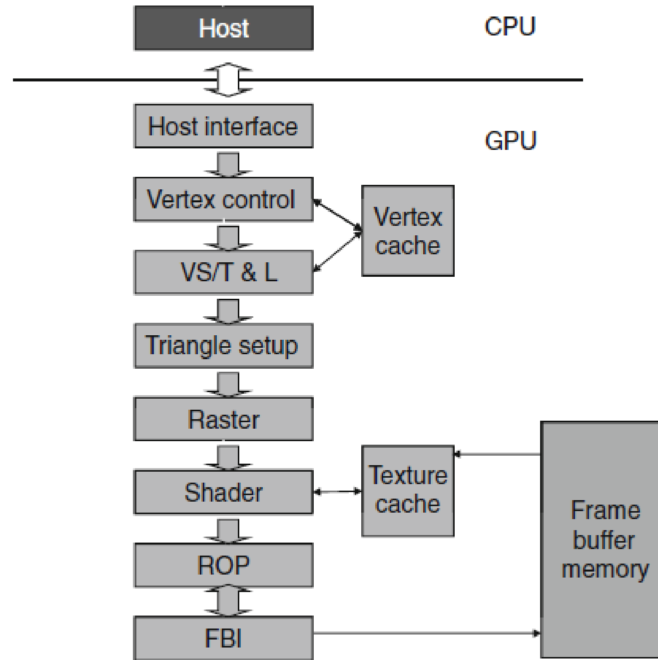


Figura 6. NVIDIA GeForce pipeline

A finales de los 90´s, debido a que se necesitaban tarjetas gráficas con soporte 3D, se modifica un poco la estructura para permitir que cada etapa en el proceso tuviera accesos a la memoria de la tarjeta para almacenar los datos intermedios requeridos para el cálculo. A mediados de los 2000's surgen los GPU de Propósito General (GPGPU, General Purpose GPU), que permiten procesar operaciones de punto flotante rápidamente y que pueden ser programadas en C en lugar de librerías gráficas y que siguen el modelo de programación SIMT.

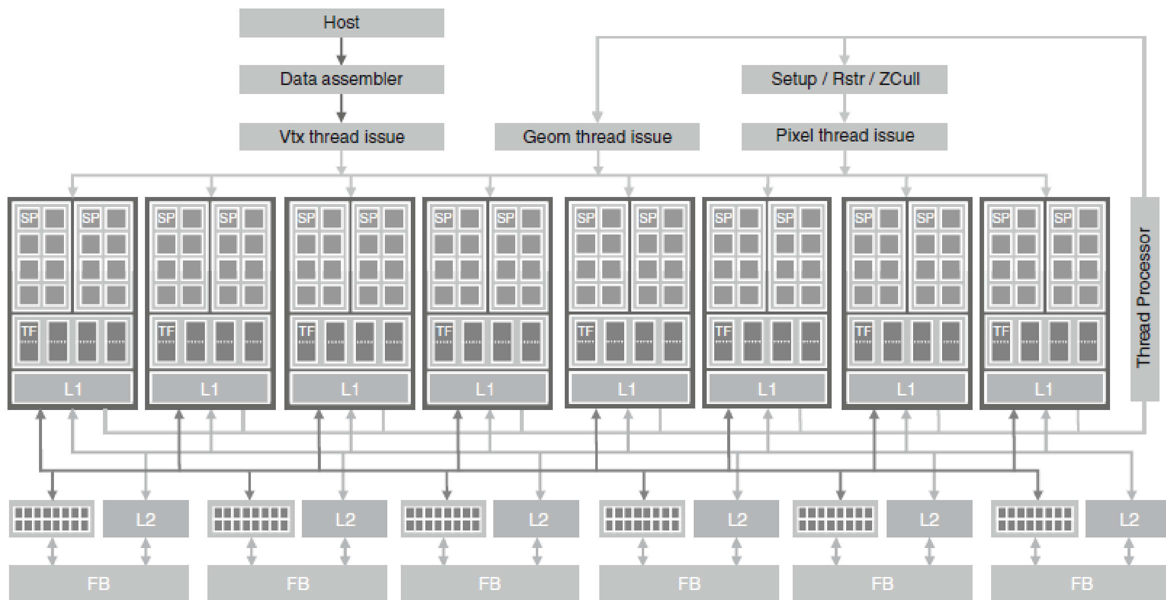


Figura 7 GeForce 8800 GTX programable.

1.6 Métricas para evaluación del desempeño.

Uno de los principales problemas en los sistemas paralelos es el decaimiento del rendimiento, que idealmente a partir de una paralelización crecería de manera lineal; por ejemplo, al momento de duplicar el número de procesadores el tiempo de ejecución se reduciría a la mitad. Sin embargo, esto no sucede y en realidad son pocos los algoritmos que logran una aceleración óptima. La mayoría de ellos tiene una aceleración casi lineal para un grupo reducido de elementos de procesamiento, después puede ser constante o en el peor de los casos decrece. Por tal motivo es necesario analizar el desempeño de un programa paralelo, ya que permite evaluar el hardware usado y determinar la escalabilidad y efectividad del algoritmo, además de evaluar la ganancia al usar paralelismo. A continuación se mencionan algunas métricas que facilitan el análisis y desarrollo de programas que se ejecutan en paralelo.

1.6.1 Tiempo de ejecución.

Se dice que es la medida principal del desempeño de un programa. Se define como el tiempo que transcurre desde que inicia hasta que termina la ejecución, en términos de sistemas paralelos el tiempo de ejecución T_p, es el tiempo que tarda desde que empieza el primer proceso hasta que termina el último proceso en ejecución. Debido al no determinismo en la ejecución de sistemas paralelos, es decir, que al ejecutarse varias veces es poco probable que tarde exactamente el mismo tiempo cada vez. Por lo que, generalmente, se obtiene de un promedio de varias ejecuciones hechas en la misma plataforma.

1.6.2 Costo.

Cuantifica la cantidad total de trabajo que realiza cada uno de los procesadores que participan en la ejecución de un programa [23]. El costo de un programa paralelo que tiene tiempo de ejecución $T_p(n)$, con tamaño de entrada n y ejecutado en p procesadores es:

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

1.6.3 Aceleración o Speedup.

Es la medida que indica la ganancia en velocidad al paralelizar una aplicación, en comparación con la implementación secuencial de la misma [23]. La aceleración $S_p(n)$ de un programa paralelo con tiempo de ejecución $T_p(n)$ se define como:

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

Donde p es el número de procesadores utilizados para resolver el problema de tamaño n y $T_s(n)$ es el tiempo de ejecución de la mejor implementación secuencial para resolver el problema.

1.6.4 Eficiencia.

Mide la porción útil del trabajo total realizado por n procesadores, indica que tan bien se están utilizando los recursos del sistema y se expresa como:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_s(n)}{pT_p(n)}$$

Donde $T_s(n)$ es el tiempo de ejecución de la mejor implementación secuencial y $T_p(n)$ es el tiempo de ejecución paralelo en p procesadores. Un programa con aceleración lineal, es decir, $S_p(n)=p$ tiene una eficiencia $E_p(n) = 1$[23].

1.6.5 Ley de Amdahl.

Determina la aceleración potencial de un algoritmo en una plataforma paralela, esta ley supone que el problema es de tamaño fijo, por lo que el trabajo que se hará será independiente de los recursos de cómputo; además señala que la porción que no se pueda paralelizar limita la aceleración que pueda lograr la implementación paralela, es decir, que la parte secuencial se mantiene respecto al número de procesadores usados y la parte paralela se distribuye equitativamente en los *P* procesadores reduciendo el tiempo, como se muestra en la figura 8.

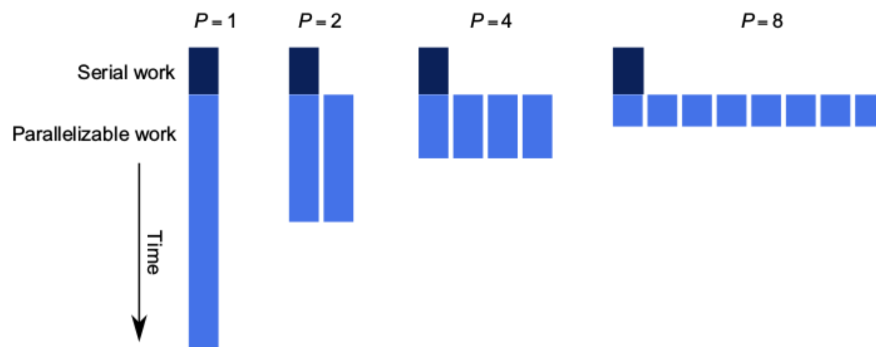


Figura 8. La aceleración está limitada por la sección de trabajo secuencial.

El tiempo de ejecución de un programa paralelo se compone del tiempo de ejecución de la fracción secuencial *f* (en donde, $0 \leq f \leq 1$) y el tiempo el tiempo de la fracción paralela $\frac{(1-p)}{p}$[23]. De acuerdo a la ley de Amdahl, la máxima aceleración alcanzable es:

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

Si la parte secuencial *f* del programa demanda el 15% del tiempo de ejecución, obtenemos un límite superior a no más de 6.67x, independientemente de los procesadores que se utilicen. En la figura 8 se muestran las aceleraciones de varios valores de *f* y *p,* y en la figura 9 se muestra el uso ineficiente de los recursos paralelos de distintas fracciones seriales.

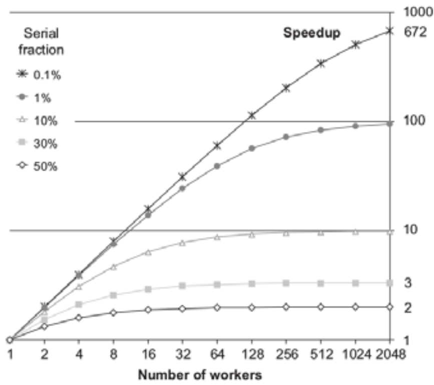


Figura 9. Aceleración – La escalabilidad de la paralelización está limitada por la fracción serial de la carga de trabajo..

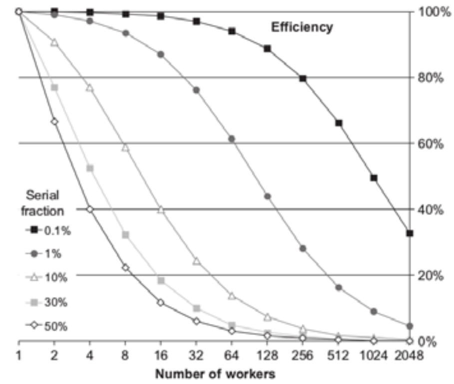


Figura 10. Eficiencia – Aun cuando mayores aceleraciones son posibles, la eficiencia puede fácilmente llegar a ser deficiente.

1.6.6 Ley de Gustafson.

En la figura 11 se muestra que la ley considera un incremento en el tamaño de los datos que es proporcional al incremento en número de recursos paralelos y calcula la máxima aceleración de la aplicación. Gustafson y Barris observaron que al momento de que el problema crece, el trabajo requerido por la parte paralela crece de manera rápida, por lo que la fracción serial decrece y la aceleración mejora.

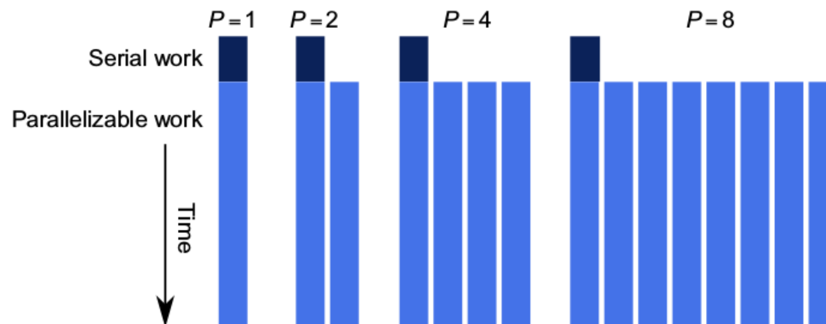


Figura 11. Ley de Gustafson-Barsis . Sí el tamaño del problema se incrementa con respecto a p mientras la porción serial crece lentamente o permanece fija, la aceleración crece conforme se agregan recursos.

Por lo que podemos decir que la ley de Amdahl predice la máxima aceleración que puede alcanzar la paralelización de un código secuencial y la ley de Gustafson-Barsis es utilizada para calcular la aceleración de un código paralelo existente, mediante la siguiente ecuación:

$$S_p(n) \leq p + (1-p)\, s$$

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

Por otro lado ambas leyes asumen que el tiempo de ejecución de la parte secuencial es independiente del número de procesadores. Sin embargo, la ley de Gustafson suponer que la cantidad total de trabajo es directamente proporcional con el número de procesadores. Ambas leyes no se contraponen, ya que, mientras una se refiere a que un programa corra más rápido con la misma carga de trabajo la otra trata de que el programa corra en el mismo tiempo con una carga de trabajo mayor.

1.6.7 Escalabilidad.

Es una medida que se refiere a si se puede alcanzar una mejora en rendimiento proporcional al número de procesadores empleados. Se dice que un sistema es escalables para un rango de procesadores [1…n], si la eficiencia *E(n)* se mantiene constante y mayor a un factor 0.5. Normalmente, se tiene, en cualquier sistema, un número determinado de procesadores que al llegar a él la eficiencia empieza a disminuir considerablemente. Tomando en cuenta lo anterior, podemos decir, un sistema es más escalable que otro si este número de procesadores es menor que el otro.

Capítulo 2 Trabajos relacionados.

En este capítulo se describen algunos trabajos que de interés para el desarrollo de esta tesina.

Aunque recientemente ha surgido el interés por el desarrollo de implementaciones de AC haciendo uso de arquitecturas paralelas, como se mencionó en la introducción de esta tesina, que permitan su uso para la modelación de sistemas con millones de partículas de manera eficiente; en este capítulo nos enfocados a dos trabajos que motivan y se toman como base para la realización de este trabajo.

2.1 Implementación de GoL usando NUMA

En [15], Sánchez Solchaga se enfoca en realizar un análisis del desempeño de implementaciones paralelas híbridas con diseño NUMA de modelos de autómatas celulares a gran escala, utilizando diversas configuraciones del espacio de dominio, en términos de escalabilidad y eficiencia. Con la finalidad de determinar si al considerar la estructura topológica del sistema donde se ejecutan, permite una estrategia que no solo brinde un mejor desempeño al minimizar los tiempos de ejecución sino también de utilizar de manera adecuada los recursos existentes. Para este propósito propone tres implementaciones, las cuales se distinguen por la manera de mapear los procesos y colocar los datos en memoria.

Las implementaciones tomadas en cuenta en [15] son así:

a) Base: es una versión paralela híbrida de la versión “secuencial”. Hace uso tanto de memoria distribuida como de memoria compartida. El espacio de dominio global se divide entre los nodos de procesamiento especificados de acuerdo a la cartografía indicada.
b) Implícita: A diferencia de la implementación anterior es que posterior a la reserva de memoria, se enlazan los procesos a unidades de procesamiento específicas, de acuerdo a la topología del sistema. De esta manera se evita la migración de los procesos.
c) Explícita: aquí además de enlazar los procesos, el subdominio de cada nodo de procesamiento es dividido físicamente y cada uno de estos bloques de memoria son reservados colocándolos en el nodo NUMA donde se encuentras los hilos que procesaran los datos, a fin de incrementar la localidad.

Mediante un análisis computacional de rendimiento, basado en escalamiento fuerte y débil, Solchaga mostró que el rendimiento de las implementaciones propuestas se puede incrementar significativamente al tomar en cuenta aspectos de la arquitectura de hardware en donde son ejecutadas; debido a que la localidad de los accesos a memoria tiene un alto impacto en el rendimiento de las aplicaciones. Particularmente, los resultados de la evaluación del rendimiento realizada en [15] mostraron que tanto la colocación de procesos como la asignación de memoria conscientes de la topología NUMA del sistema, mejoraron el rendimiento de la implementación paralela de los modelos de AC al utilizar un alto número de procesos e incrementar el tamaño del espacio de dominio; principalmente para el caso de la agrupación compacta. Como consecuencia, los tiempos de ejecución de la simulación

se reducen debido a que al asignar la memoria en el mismo nodo en el que se ejecuta el proceso que la accede, se reduce la latencia causada por los accesos remotos y la contención por el ancho de banda de la memoria; logrando así, un incremento en la aceleración y la eficiencia. Sin duda el trabajo realizado por Solchaga es importante y diferente a los trabajos existentes en la literatura, sin embargo no analiza el uso de otro tipo de arquitecturas, como aquellas basadas en GPUs.

2.2 Un análisis del uso de cuatro tarjetas GPU.

Más recientemente, Millán et. al en [16], se enfocan en el análisis y comparativa del desempeño la implementación en paralelo del GoL usando cuatro tarjetas gráficas GPU's diferentes. Sin embargo, las implementaciones presentadas no son a gran escala. Básicamente generan cuatro implementaciones, la primera "Baseline" que usa únicamente memoria global para realizar los cálculos, dos más ambas usando memoria compartida, difiriendo solo en el patrón de acceso a memoria y la última usando una implementación multicelda. Millán et. al concluyen que el rendimiento de las implementaciones depende de las características de cada arquitectura de GPU, como es de esperarse.

Los trabajos descritos en este capítulo, motivaron esta tesina. En el siguiente capítulo se presenta así una implementación GoL en una tarjeta gráfica NVIDIA Telsa K40, con el fin de permitir el manejo de sistemas a gran escala y verificar su rendimiento cuando se hace uso de GPUs.

Capítulo 3 Desarrollo del modelo de AC basado en GPU

En este capítulo, se presenta una propuesta para el diseño de implementaciones paralelas de modelos de AC en GPU. El objetivo general es realizar una implementación de una AC conocido, el Juego de la Vida, en una tarjeta gráfica y analizar su desempeño.

Debido al paralelismo de grano fino de los AC, sabemos que el espacio celular puede ser fácilmente particionado en las entidades más pequeñas que lo forman; de tal manera que cada celda dentro del espacio de dominio puede evolucionar simultáneamente y ser procesada por una tarea independiente. Sin embargo no son independientes ya que las celdas de los AC requieren obtener información de sus vecinos. Esta comunicación depende directamente de la vecindad utilizada en el modelo, ya que la actualización de cada una de las celdas en el paso de tiempo actual (*t*), requiere conocer los estados de las celdas vecinas y su mismo estado para utilizarlos como argumentos en la función de transición definida en el modelo. De esta manera, obtener el estado de la celda *c[i, j]* en el siguiente paso de tiempo *(t+1).*

3.1 Caso de estudio.

A continuación, se presenta una descripción de la implementación secuencial del modelo estudiado basado en autómatas celulares. El juego de la Vida (GoL).

3.1.1 El Juego de la Vida.

Descrito en el capítulo 1, el juego de la vida, GoL [24], es uno de los AC más conocidos, formulado por el matemático británico John H. Conway en 1970. El particular interés en el GoL es que logra un resultado similar al comportamiento real de la vida, con un conjunto de reglas lo más simple posible, por este motivo ha recibido mucha atención por parte de los investigadores; por otro lado, una de sus características más importantes es que permite realizar cómputo universal, es decir, que con una configuración inicial apropiada, el GoL, se puede convertir en una computadora de propósito general o máquina de Turing [25].

El Pseudocódigo 3.1, muestra la versión secuencial standard del GoL utilizando dos arreglos de tamaño N x N, cada uno para representar el paso de tiempo actual y siguiente *(t y t+1* respectivamente), de esta manera se evitan errores causados por la sobre escritura de los estados de las celdas en el enmallado entre cada pasa del tiempo. POr simplicidad y para un mejor entendimiento se presenta en inglés.

```
1    for every generation in Life do
2        for every Cell in DomainSpace do
3            Neighbors ← 0
4            for every Cell in Neighborhood do
5                Neighbors ← Neighbors + Cell [i, j]
```

```
6           end for
7           if (Cell [i, j] is alive and neighbors == 3)
8               Cell[t+1] ← ALIVE //viva
9           else if (Cell is alive and (neighbors == 2 or neighbors == 3))
10              Cell[t+1] ← ALIVE
11          else
12              Cell[t+1] ← DEAD //muerta
13          end if
14      end for
15  end for
```

Pseudocódigo 3.1. "El juego de la vida".

A pesar de la simplicidad, el GoL es un modelo que genera patrones interesantes, además de que provee los fundamentos básicos para crear simulaciones que muestran características y comportamientos reproductivos de sistemas biológicos. Así mismo, la implementación secuencial y paralela de este modelo brinda las bases para el desarrollo de modelos de AC más complejos. Por lo que en este trabajo nos enfocaremos en el mismo.

3.2 Implementación paralela.

La paralelización de la implementación secuencial del modelo de AC descrito en la sección anterior, se realiza utilizando una técnica de descomposición de dominio basada en el modelo Single Program Multiple Data (SPMD), en donde cada proceso ejecuta el mismo programa en su propio subconjunto de datos.

3.2.1 Estructura general.

La implementación paralela, como todo programa hecho en CUDA, está formada por una o más fases que pueden ser bloques ejecutados en el CPU (host) o en el GPU (device) y el encargado de realizar la separación de cada código es el compilador NVIDIA C (nvcc). A continuación se muestra la estructura de la implementación realizada en este trabajo:

```
1    Allocate and initializate Matrices Lattice 1 y Lattice 2
2    Allocate device memory Curr y Next
3    Function Kernel invocation code
4    Interchanging Matrices Curr ← Next
5    Free device and host matrices
```

3.2.2 Configuración inicial.

Para establecer la configuración inicial, se puede optar por establecer de manera estática, aplicando una configuración predefinida, con el objetivo de verificar el buen comportamiento del modelo, o establecerse de manera aleatoria, a través del uso de funciones generadoras de número aleatorios. De acuerdo con el estudio realizado por Gibson et al [4], la mayor actividad de las celdas para el modelo del GoL se da al utilizar una probabilidad de existencia de celdas vivas de entre 20% y 60%, por lo que en este trabajo se utiliza una probabilidad de 50% de celdas vivas, para poder alcanzar un alto nivel de actividad el paso de 1000 generaciones.

3.2.3 Memoria en GPU y transferencia de datos.

Tanto el host como el GPU tienen memorias separadas, por lo que es necesario reservar memoria en el GPU para transferir los datos necesarios desde la memoria del CPU a la memoria reservada del GPU. De igual manera al terminar la ejecución se requiere regresar los resultados desde la memoria del GPU a la memoria del host. En la Figura 12 se muestra un esquema del modelo de memoria en CUDA.

- Device code can:
  - – R/W per-thread registers
  - – R/W per-thread local memory
  - – R/W per-block shared memory
  - – R/W per-grid global memory
  - – Read only per-grid constant memory

- Host code can
  - – Transfer data to/from per-grid global and constant memories

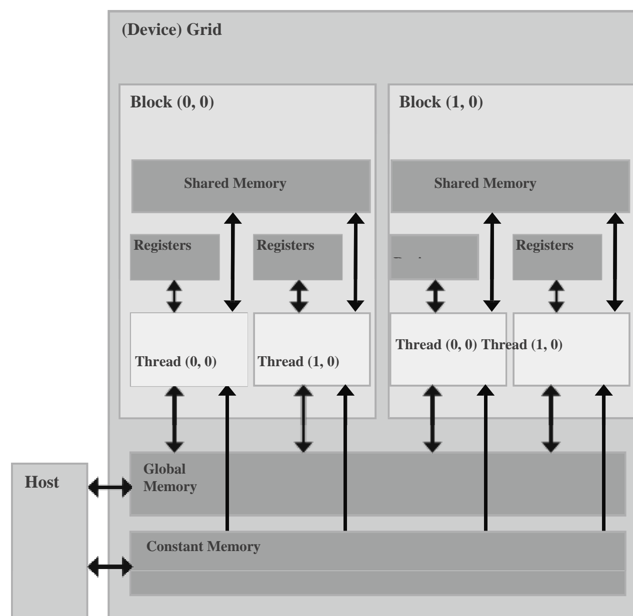


Figura 12 Esquema del Modelo de Memoria en CUDA

La función utilizada, que permite reservar y administrar la memoria tanto del CPU como del GPU, es *cudaMallocManaged()*. El uso de esta función depende de las características que presentan las tarjetas gráficas, para este trabajo la tarjeta NVIDIA Tesla K40 cumple con los requerimientos para usarla. En la implementación realizada se usa de la siguiente manera *cudaMallocManaged(&Curr,(height * width) * sizeof(int *)).*

3.2.4 Funciones Kernel

Se tienen dos funciones que se ejecutan en la tarjeta gráfica, también conocidas como funciones *device*. Una de ellas se encarga de verificar el estado de las celdas vecinas y la otra realiza la suma de los vecinos con estado vivo. La otra, llamada función *global*, es la encargada de ser el enlace entre el CPU y el GPU y de recibir los datos de las funciones *device* para tomar la decisión del estado de la celda actual en el siguiente tiempo *(t +1)*.

Cuando se invoca una función *kernel global*, esta se ejecuta sobre una matriz de hilos paralelos (ver Figura 13). Los hilos dentro de la matriz están organizados en una jerarquía de dos niveles como se ve en la Figura 13. Se tienen, para ejemplificar, una matriz de bloques (grid) de 3 x 2, es decir 6 bloques; y cada boque esta, a su vez, organizado en una matriz de hilos de 4 x 3. Entonces, para la Figura 13, se tienen 6 x 12 = 72 hilos para procesar la información.

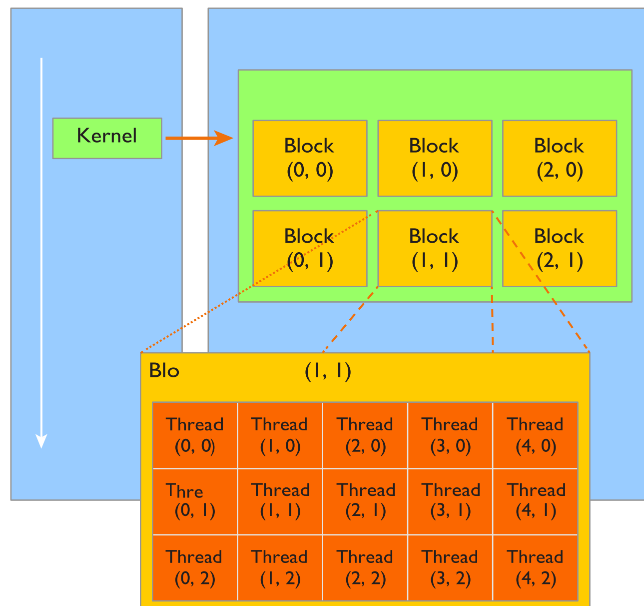


Figura 13 Estructura de bloque e hilos en un GPU

Para obtener la referencia de cada uno de ellos, se tienen variables establecidas dentro del lenguaje de programación, la cuales son:

- blockDim.x, blockDim.y, blockDim.z: para obtener la dimensión del grid que contiene los bloques.
- blockIdx.x, blockIdx.y,blockIdx.z: para obtener la organización de los bloques.
- threadIdx.x, threadIdx.y, threadIdx.z: para obtener la organización de los hilos dentro de los bloques.

  Comúnmente una matriz se almacena en un arreglo unidimensional, por ejemplo como en la Figura 14, que muestra una matriz de dimensión 8 x 6. Así, para obtener el índice de un valor dentro de la matriz se siguen los siguiente dos pasos:

  1) Se mapea la ubicación del hilo y el bloque dentro de la matriz usando las siguientes fórmulas.

     ix = threadIdx.x + blockIdx.x * blockDim.x

     iy = threadIdx.y + blockIdx.y * blockDim.y

  2) Se mapea la ubicación de la matriz en la memoria mediante la siguiente fórmula.

     Idx = iy * nx + ix

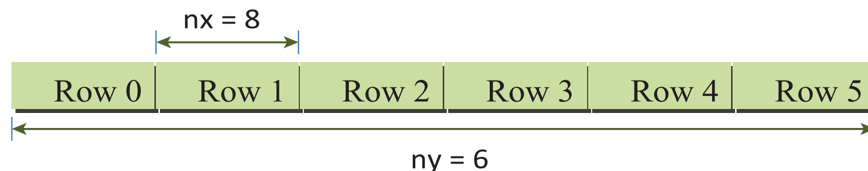


Figura 14 Representación de una matriz de 8 x 6 en la memoria.

Para la implementación hecha en este trabajo y definir el tamaño del grid de bloques y el tamaño del bloque se usó

```
dim3 blockside(NP,NP);
dim3 gridsize(width/BLOCK_SIDE,height/BLOCK_SIDE);
```

Y la llamada a la función *kernel global* es:

```
gol<<<gridsize,blockside>>>(height,width,curr,next);
```

3.2.5 Puntos de sincronización.

Para esta tesina, se utilizó la versión standard de implementación de modelos de AC, la cual consiste en dos arreglos para contener el estado actual y siguiente de las celdas,

e intercambiándolas en cada generación para evaluar la evolución del modelo para múltiples generaciones (ver Figura 15).

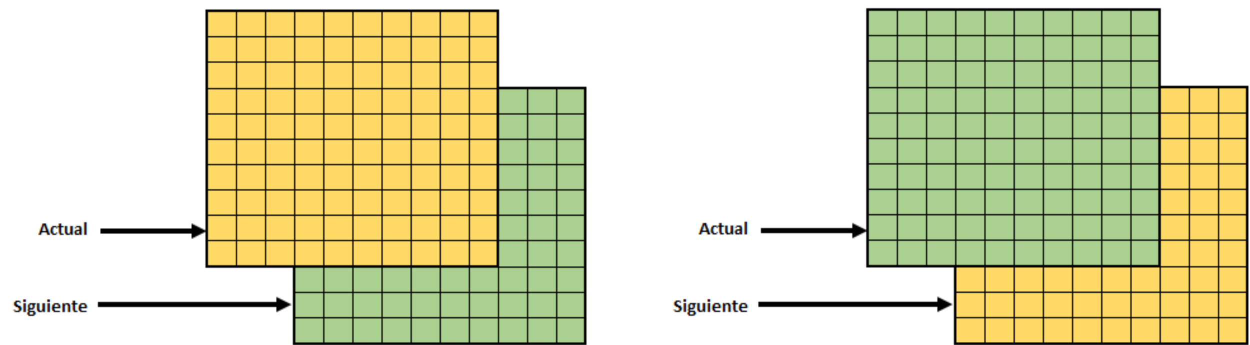


Figura 15 Intercambio de referencias entre mallas que representan el estado actual (t) y siguiente (t+1) del espacio de dominio.

Se establece un punto de sincronización antes de realizar el intercambio de mallas con el uso de la función cudaDeviceSynchronize(), esto permite que la tarjeta gráfica termine de realizar los cálculos de la función de transición aplicada al enmallado.

Capítulo 4 Resultados obtenidos.

En este capítulo se presentan los resultados de la evaluación de la implementación paralela del modelo de AC comentada en el capítulo anterior. Para este propósito se generan y analizan los resultados que se obtienen al realizar varios experimentos de la implementación realizada.

4.1 Características de la plataforma de ejecución.

Todas las simulaciones se realizaron utilizando el clúster Tonatiuh del Instituto de Ingeniería de la UNAM (IINGEN). Tonatiuh está formado por 5 nodos (uno de administración y cuatro de procesamiento), interconectados a través de un enlace Gigabit Ethernet a 1 Gbps. De tal manera que, los nodos de procesamiento suman un total de 200 núcleos, 448 GB de RAM y 6784 núcleos GPU. Tonatiuh usa como sistema operativo GNU/Linux, con la versión 2.6.32 del kernel, a través de la distribución de Rocks 6.2 basado en CentOS 6.9. En la Tabla 4.1 se muestran las características por cada nodo de Tonatiuh.

Nodo	Características
Nodo admin: tonatiuh	2 Intel Xeon CPU E5-2670 v3 @ 2.30GHz, 12 cores per socket, 1 thread per core; 128 GB RAM; 2 NUMA domains.
Nodo 0: compute-0-0	4 AMD Opteron Processor 6380 @ 2.50Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 8 NUMA domains.
Nodo 1: compute-0-1	4 AMD Opteron Processor 6276 @ 2.30Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 8 NUMA domains.
Nodo 2: compute-0-3	2 Intel Xeon CPU E5-2680 v2 @ 2.80GHz, 10 cores per socket, 2 threads per core; 64 GB RAM; 2 NUMA domains; 2 NVIDIA Tesla K40, 2880 GPU cores, 12 GB RAM.
Nodo 3: compute-0-4	2 AMD Opteron Processor 6272 @ 1.4 Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 4 NUMA domains; 2 NVIDIA Tesla M2090, 512 GPU cores, 6 GB RAM.

Tabla 4.1. Resumen de características de los nodos que conforman el cluster tonatiuh del IINGEN.

Para las implementaciones realizadas, se utilizaron las siguientes herramientas:

- Compilador gcc 4.4.7
- Compilador nvcc 8.0.61

Cabe mencionar que sólo se usó el nodo “compute-0-3” del clúster, esto debido a que es el nodo que tiene la tarjeta GPU con mejores prestaciones.

4.2 Descripción de los parámetros de entrada.

Para llevar a cabo la evaluación, se utilizaron los tamaños de malla mostrados en la Tabla 4.2 para el espacio de dominio global de la simulación.

Dimensiones	Total de elementos
1024 x 1024	1,048,576
2048 x 2048	4,194,304
4096 x 4096	16,777,216

Tabla 4.2. Tamaños de espacios de dominio utilizados para cada experimento.

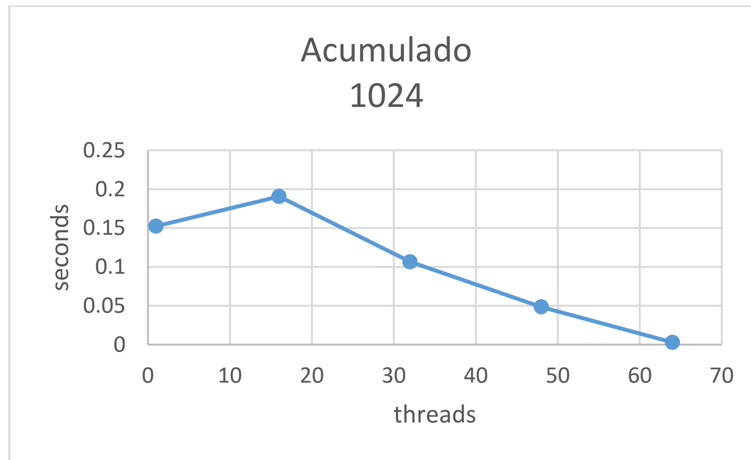
Para cada simulación se usó el mismo número de generaciones, en este caso, 1000.

El número de hilos evaluado para la implementación fueron NP: 16, 32, 48 y 64.

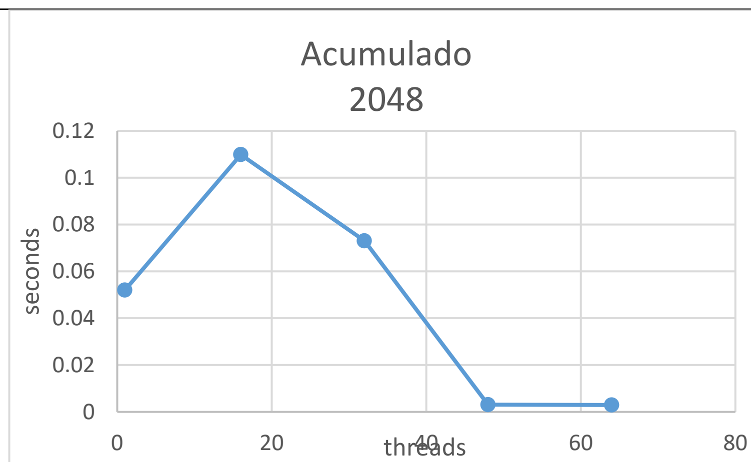
4.3 Evaluación experimental.

4.3.1 Tiempo Acumulado.

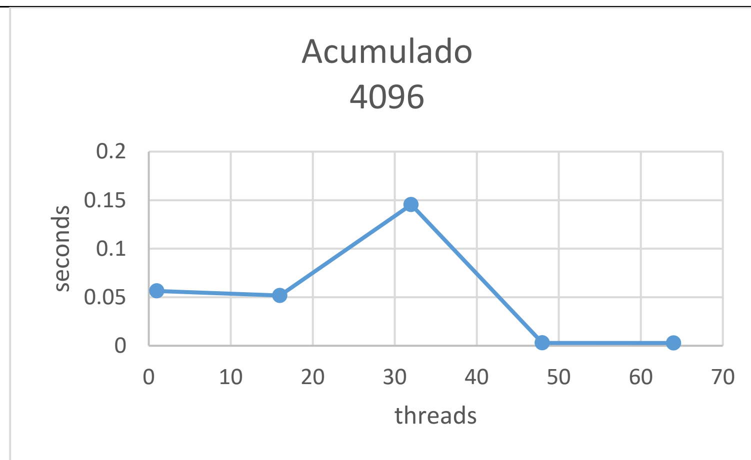
Primeramente se realizó una evaluación del tiempo acumulado con respecto a los hilos usados para los diferentes espacios de dominio, cuyos resultados se muestran en la Figura 15. Como se puede observar de esta figura, para los diferentes tamaños de los espacios de dominio considerados, se observa un incremento del tiempo al utilizar pocos hilos. Esto se debe a la sobrecarga de operaciones paralelas derivadas de la comunicación y sincronización intrínseca que requieren los AC y a la sobrecarga paralela de la implementación con GPUs. Sin embargo, hay un número de threads a partir del cual un incremento en el número de hilos utilizado implica una disminución en el tiempo acumulado Este desempeño es más notorio al incrementar el tamaño del espacio de dominio.



(a)



(b)

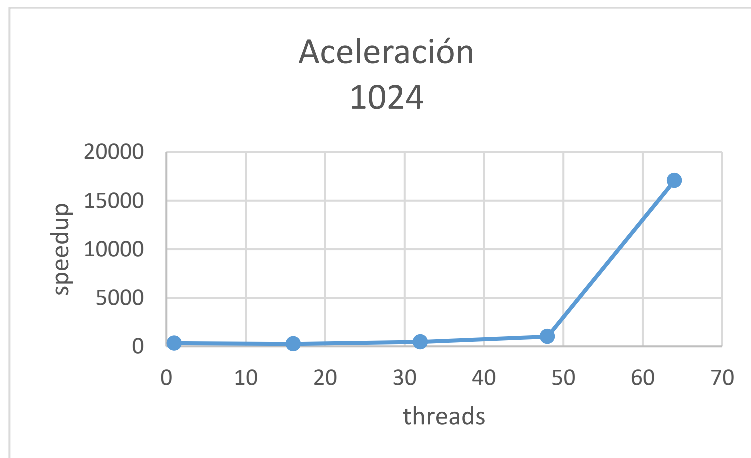


(c)

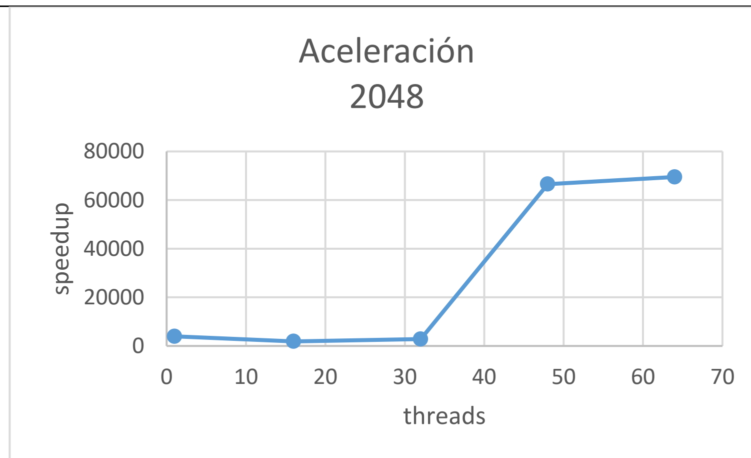
Figura 15 Tiempos acumulados utilizando un espacio de dominio de 1024 (a), 2048 (b), 4096 (c)

4.3.3 Aceleración

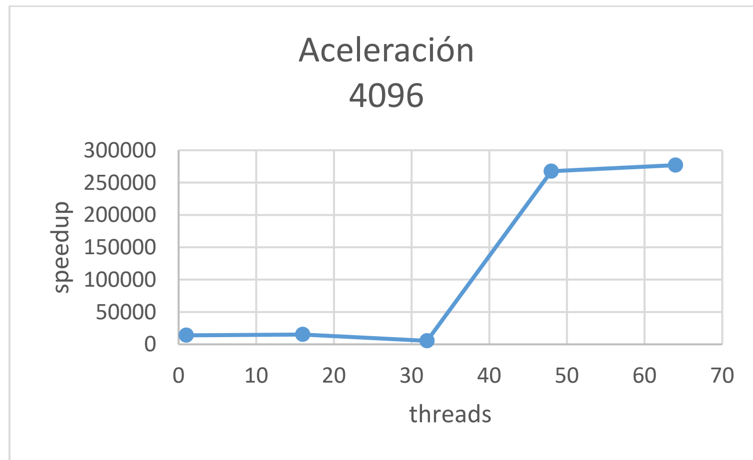
Segundo se estudió la aceleración de la implementación. Así, se calculó la aceleración de la implementación para los diferentes espacios de dominio utilizados, los cuales se muestran en la Figura 16. Como puede observarse de esta figura, la aceleración se incrementa en función del número de hilos considerados: un mayor número de hilos implica una aceleración mayor. Pero además esta es mayor conforme el espacio de dominio es más grande.



(a)



(b)



(c)

Figura 16 Aceleración utilizando un espacio de dominio de 1024 (a), 2048 (b), 4096 (c)

Conclusiones y trabajo futuro.

Tomando en cuenta los resultados mostrados en el capítulo anterior podemos observar que la implementación y la configuración usada en la tarjeta gráfica NVIDIA Tesla K40, generan tiempos de respuesta muy cortos al procesar espacios de dominio relativamente grandes. Esto debido, por un lado, a las características de la tarjeta que cuenta con 2880 Cuda-cores que permite agrupar en bloques para trabajar en conjunto y a su vez por cada bloque tener máximo 1024 hilos; aunado a la configuración de bloques que se haya definido.

Trabajo futuro.

A continuación se plantean propuestas a considerar para mejorar y/o ampliar la implementación hecha en este proyecto final.

- Realizar la asignación de memoria y la inicialización de las matrices mediante el uso de una función device, esto para minimizar el tiempo en la transferencia de la información del CPU al GPU.
- Realizar el intercambio del enmallado del estado actual con el siguiente dentro del dispositivo GPU ya que esto permite optimizar el tiempo ya que se hace uso de la memoria local del dispositivo, evitando hacer un uso excesivo de intercambio de información entre el dispositivo y el GPU.
- Realizar una implementación considerando la hecha por Sánchez Solchaga [15] para hacer un estudio comparativo entre ambas implementaciones.

Referencias.

[1] Bandman, O. (2013) Implementation of large-scale cellular automata models on multi-core computers and clusters, 2013 International Conference on High Performance Computing & Simulation (HPCS), Helsinki. 304-310.

[2] Chorley, M.J., Walker, D.W. (2010) Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters, In Journal of Computational Science, 1 (3): 168-174

[3] Guisado, J.L., Jiménez-Morales, F., Vega, F. (2007). Cellular Automata and Cluster Computing: an Application to the Simulation of Laser Dynamics. Advances in Complex Systems. 10: 167-190

[4] Gibson, M.J., Keedwell, E.C., Savić, D.A. (2015) An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware, In Journal of Parallel and Distributed Computing, 77: 11-25

[5] Kalgin, K.V. (2012) Parallel implementation of asynchronous cellular automata on a 32-core computer, Numerical Analysis and Applications. 5(1): 45–53

[6] Rybacki, S., Himmelspach, J., Uhrmacher, A.M. (2009) Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata, 2009 First International Conference on Advances in System Simulation, Porto, 62-67.

[7] Ntinas V.G., Moutafis B.E., Trunfio G.A., Sirakoulis G.C. (2016) GPU and FPGA Parallelization of Fuzzy Cellular Automata for the Simulation of Wildfire Spreading. In: Wyrzykowski R., Deelman E., Dongarra J., Karczewski K., Kitowski J., Wiatr K. (eds) Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, 9574: 560-569

[8] Topa, P. (2014) Cellular Automata Model Tuned for Efficient Computation on GPU with Global Memory Cache, 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Torino, 380-383.

[9] Millán, E.N., Martínez, P.C., Costa, G.V.G., Piccoli, M.F., Printista, A.M., Bederian, C. (2013). Parallel implementation of a cellular automata in a hybrid CPU/GPU environment. In: De Giusti A, editor. XVIII Congreso Argentino de Ciencias de la Computación, Red de Universidades con Carreras en Informática RedUNCI; 184–93

[10] Drieseberg, J., Siemers C. (2012) C to Cellular Automata and execution on CPU, GPU and FPGA, 2012 International Conference on High Performance Computing & Simulation (HPCS), 216-222.

[11] Dogaru, I., Dogaru, R. (2014) A comparative study of several 2D cellular automata implementations in FPGA, 2014 International Symposium on Fundamentals of Electrical Engineering (ISFEE), 1-4.

[12] Vourkas, I., Sirakoulis, G.C., (2012) FPGA based cellular automata for environmental modeling, 19th IEEE International Conference on Electronics, Circuits, and Systems, 93-96.

[13] Bezbradica, M., Crane, M., Ruskin, H.J. (2012) Parallelisation strategies for large scale cellular automata frameworks in pharmaceutical modelling, 2012 International Conference on High Performance Computing & Simulation (HPCS), 223-230

[14] Millán, E.N., Bederián, C., Piccoli, F., Garcia, C., Bringa, E. (2015). Performance analysis of Cellular Automata HPC implementations. Computers & Electrical Engineering. 48: 12-24

[15] Sánchez Solchaga, Manuel Eduardo (2018). Diseño y optimización de implementaciones de modelos de Autómatas Celulares para Arquitecturas Paralelas con Acceso no uniforme a Memoria. Tesis de Maestría. Universidad Nacional Autónoma de México.

[16] Millán, E.N., Wolovick,N, Piccoli, F., García, C, Bringa,E. (2017) Performance analysis and comparison and cellular automata GPU implementations. 20:2763-2777

[17] Sfa F., Nemiche M., and Lopez R., (2015). A Theoretical Learning Model Combining Stochastic Cellular Automata and Economic Indicators to Simulate Land Use Change. International Journal of Applied Evolutive Computing. 6 (3): 1-8.

[18] Fuglsang, M., Münier, B., Hansen, H. S. (2013). Modelling land-use effects of future urbanization using cellular automata: An Eastern Danish case. Environmental Modelling & Software, 50: 1-11.

[19] Chen, X., Yu, S. X. and Zhang, Y. P. (2013). Evaluation of Spatiotemporal Dynamics of Simulated Land Use/Cover in China Using a Probabilistic Cellular Automata-Markov Model, In Pedosphere, 23 (2): 243-255

[20] Burraston, D., Edmonds, E., Livingstone, D., Miranda, R.E. Cellular Automata in MIDI based Computer Music.

[21] Nandi, S., Kar, B.K., Chaudhuri, P.P. Theory and Applications of Cellular Automata in Cryptography. IEEE Transactions on Computers, Vol. 43, No.12 1994

[22] Flynn, M. (1972). Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., C-21: 948

[23] Hager, G. & Wellein, G. (2011). Introduction to high performance computing for scientists and engineers. Boca Raton, FL: CRC Press.

[24] Gardner M. (1970) The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. Scientific American, 223: 120–123.

[25] Wolfram, S. (1986) Theory and Applications of Cellular Automata. World Scientific, Singapore.