UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Paralelización de autómatas celulares en tarjetas gráficas y su métrica de rendimiento

TESIS

Que para obtener el título de

Ingeniero en computación

P R E S E N T A

José Ángel de Jesús García Vázquez

DIRECTOR DE TESIS

Ing. José Antonio Ayala Barbosa

Ciudad Universitaria, Cd. Mx. 2022

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

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.

Agradecimientos

En primer lugar quiero agradecer a mi tutor el Ing. José Antonio Ayala Barbosa, quien con sus conocimientos y apoyo me guió a través de cada una de las etapas de este proyecto para alcanzar los resultados que buscaba.

Quiero agradecer a la UNAM por brindarme todas las herramientas y recursos para mi formación académica. También quiero agradecerle a los profesores de la carrera que contribuyeron a mi formación profesional.

Gracias a todos mis amigos y amigas, por apoyarme aún cuando mis ánimos decaían, por alentarme cuando lo necesitaba y no dejarme solo.

En especial, quiero agradecer a mis familiares, que siempre estuvieron allí para apoyarme continuamente con sus palabras y aliento, pero sobre todo con aquello que me decían cuando creía no poder más.

Muchas gracias a todos.

Índice general

Introducción **10**

1. Antecedentes **12**

1.1. Cómputo de alto rendimiento 12
1.1.1. GPU en cómputo de alto rendimiento 13
1.1.2. Taxonomía de Flynn en el cómputo de alto rendimiento 13
1.2. Impacto de las tarjetas gráficas 16
1.3. GPGPU 17
1.3.1. Cómputo gráfico y científico con GPUs 18
1.4. Paralelización en tarjetas gráficas 18
1.4.1. Kernels 19
1.4.2. Malla, bloques e hilos 20
1.4.3. Memoria compartida 21
1.4.4. Multiprocesadores SM 23
1.5. Modelos y Sistemas 24
1.5.1. Modelo 24
1.5.2. Sistema 24
1.5.3. Sistema Multiagente 24
1.5.4. Sistema Complejo 25
1.5.5. Sistema Caótico 25
1.5.6. Sistema Estáticos 26
1.5.7. Sistema Dinámico 26
1.6. Autómatas Celulares 26
1.6.1. Espacio Celular 28
1.6.2. Condición de frontera 28
1.6.3. Reglas de transición 29

1.6.4. Geometría de la vecindad 29
1.7. Dimensionalidad 30
1.7.1. Clasificación de Wolfram 30
1.7.2. Reglas de Wolfram 31
1.8. Tipos de Autómatas celulares 32
1.9. Herramientas para simular AC 33
1.10. Requerimientos 33
1.10.1. Requerimientos funcionales y no funcionales 33
1.11. Métricas 34
1.11.1. Tiempo de ejecución (Runtime) 35
1.11.2. Aceleración (Speedup) 36
1.11.3. Factor de costo (Costfactor) 37
1.11.4. Eficiencia (Efficiency) 38
1.11.5. Ley de Amdahl 39
1.11.6. Ley de Gustafson 41

2. Selección del caso **43**
2.1. El juego de la vida 43
2.1.1. Patrones 44
2.1.2. Sistemas en el Juego de la vida 46
2.1.3. Modelos el Juego de la vida 46
2.2. ¿Por qué el juego de la vida? 47

3. Implementación en secuencial **48**
3.1. Programación secuencial 48
3.2. Desarrollo 48
3.2.1. Equipo empleado 49
3.2.2. Código en secuencial 52

4. Implementación en paralelo **58**
4.1. Desarrollo 58
4.1.1. Diagramas 60
4.1.2. Código 62
4.1.3. Explicación del algoritmo en paralelo 67

5. Métricas de rendimiento **70**
5.1. Primer conjunto de datos 71
5.1.1. Aceleración 72
5.2. Segundo conjunto de datos 74
5.2.1. Aceleración 76
5.3. Ley de Amdahl 77
5.3.1. Aceleración con la ley de Amdahl 80

6. Conclusiones y trabajo futuro **82**

Bibliografía **105**

Índice de figuras

1.1. CPU vs GPU [11] 13
1.2. Diagrama de SISD[18] 14
1.3. Diagrama de SIMD[18] 15
1.4. Diagrama de MISD[18] 16
1.5. Diagrama de MIMD[18] 17
1.6. Malla con NxM bloques de OxP hilos. Imagen basada en [7] 20
1.7. Diagrama de la distribución de la malla, los bloques y los hilos[18] 21
1.8. Diagrama de la memoria en CUDA[30] 23
1.9. Ejemplo de un autómata celular[10] 27
1.10. Espacio celular triangular, cuadrado y hexagonal[25] 28
1.11. Geometría de Moore y Von-Neuman. Basado en [26] 30
1.12. Posibles dimensiones de un AC. Basado en [26] 30
1.13. Regla 30[31] 32
1.14. Regla 30 después de 15 iteraciones[31] 32
1.15. Gráfica de runtime variando el número de procesadores 35
1.16. Gráfica de la aceleración variando el número de procesadores 36
1.17. Gráfica del factor de costo 38
1.18. Gráfica de eficiencia 39
1.19. Gráfica de la ley de Amdahl, toma su forma dependiendo de α y el número de procesadores. 41
1.20. Gráfica de la ley de Gustafson, toma su forma dependiendo de α y el número de procesadores de forma lineal. 42

2.1. Oscilador, patrón más sencillo de obtener en el juego de la vida[26] . . . 44
2.2. Ejemplo de un patrón estático[26] 45
2.3. Evolución de una nave espacial[26] 45

2.4. Ejemplo de un matusalén estabilizado[26] 45

3.1. Diagrama de flujo general del programa en secuencial 50
3.2. Diagrama de flujo del funcionamiento del AC en secuencial 51
3.3. Cálculo de las células vecinas. 54
3.4. Ejecución de un AC de tamaño 100x100 57

4.1. Diagrama de flujo del funcionamiento del AC en paralelo. 61
4.2. Ejemplo del funcionamiento del algoritmo. 68

5.1. Gráfica reducida del tiempo de ejecución del primer conjunto de datos . . 71
5.2. Gráfica reducida de la aceleración del primer conjunto de datos. 73
5.3. Gráfica del tiempo de ejecución del segundo conjunto de datos. 74
5.4. Gráfica ampliada de la aceleración del segundo conjunto de datos. 77
5.5. Línea de tiempo de ejecución del algoritmo. 78
5.6. Gráfica que representa el porcentaje de la fracción del algoritmo en secuencial. 79
5.7. Gráfica que representa la máxima aceleración del algoritmo. 81

6.1. Gráfica de aceleración 83
6.2. Gráfica del tiempo de ejecución del primer conjunto de datos. 87
6.3. Gráfica de la aceleración primer conjunto de datos. 89
6.4. Gráfica del tiempo de ejecución del segundo conjunto de datos 91
6.5. Gráfica de aceleración del primer conjunto de datos 95
6.6. Gráfica que representa el porcentaje de la fracción del algoritmo en secuencial. 98
6.7. Gráfica que representa la máxima aceleración del algoritmo. 100
6.8. Diagrama que representa las actividades realizadas. 102
6.9. Diagrama que representa las actividades realizadas. 103
6.10. Diagrama que representa las actividades realizadas. 104

Lista de tablas

1.1. Tabla de transferencia de datos en CUDA[23] 22

5.1. Tabla reducida del primer conjunto de datos. 71
5.2. Tabla reducida de la aceleración para el primer conjunto de datos. 73
5.3. Tabla reducida del tiempo de ejecución para el segundo conjunto de datos. 74
5.4. Tabla comparativa de los tipos de datos utilizados 75
5.5. Tabla reducida de la aceleración para el segundo conjunto de datos. 77
5.6. Tabla del tiempo de ejecución del algoritmo en segundos. 78
5.7. Tabla con los porcentajes de ejecución de tiempo en paralelo y secuencial 79
5.8. Tabla de los resultados de la ley de Amdahl 81

6.1. Tabla del primer conjunto de datos. 88
6.2. Tabla del primer conjunto de datos. 90
6.3. Tabla del tiempo de ejecución para el segundo conjunto de datos CPU (Secuencial) 92
6.4. Tabla del tiempo de ejecución para el segundo conjunto de datos GTX 1050 Ti (Paralelo) 93
6.5. Tabla del tiempo de ejecución para el segundo conjunto de datos GTX 1660 Super (Paralelo) 94
6.6. Tabla de los resultados de la aceleración del segundo conjunto de datos. . 96
6.7. Tabla del tiempo de ejecución del algoritmo en segundos 97
6.8. Tabla con los porcentajes de ejecución de tiempo en paralelo y secuencial 99
6.9. Tabla de los resultados de la ley de Amdahl 101

Índice de códigos

1.1. Código de creación de kernels 19
1.2. Código para determinar el número de hilos en un kernel 19
1.3. Código de creación de kernels 21
1.4. Código de creación de kernels 22
3.1. Apartado de memoria dinámica en la memoria RAM de la computadora. . 52
3.2. Lectura de datos e iteraciones del AC. 52
3.3. Cálculo de las células vecinas. 54
4.1. Lectura de los datos e inicialización de variables. 62
4.2. Transferencia de datos entre host y device. 63
4.3. Cálculo de la dimensión de los bloques y la malla. 63
4.4. Lectura de los datos e inicialización de variables. 64
4.5. Lectura de los datos e inicialización de variables. 65

Introducción

Los autómatas celulares no son un concepto nuevo, son una herramienta utilizada para representar sistemas dinámicos que evolucionan de forma discreta, son muy útiles para modelar fenómenos. Un problema que tienen todos los autómatas celulares es que debido a la gran cantidad de datos que trabaja se vuelve una tarea lenta de procesar incluso para las computadoras más actuales.

La computación de alto rendimiento o HPC (High Performance Computing) por sus siglas en inglés, se apoya de distintas técnicas y tecnologías para resolver problemas con una gran cantidad de datos en el menor tiempo posible, esto es lo que se quiere lograr en un autómata celular, reducir los tiempos de ejecución.

Actualmente, lo más común es encontrar simuladores computacionales de autómatas celulares que realizan todo el procedimiento de forma secuencial, es decir, una instrucción a la vez. Aunque esta implementación es útil, los tiempos de espera pueden llegar a ser muy grandes, esto depende directamente de cómo esté construido el simulador, del tamaño del autómata y del número de iteraciones que se realizan.

Utilizando la programación en paralelo se pueden resolver este y muchos otros problemas en menor tiempo. Debido a que este tipo de algoritmos divide el problema en tareas más pequeñas y ejecuta más de una instrucción a la vez, podría reducir notoriamente los tiempos de ejecución.

La programación en paralelo más común es la que se realiza en procesadores que cuentan con múltiples núcleos y generan múltiples hilos, pero dentro de este trabajo se quiere utilizar otro tipo de hardware, las tarjetas gráficas. Las tarjetas gráficas actuales poseen gran cantidad de núcleo, pudiendo generar miles de hilos de forma paralela, volviéndolas una mejor herramienta que un procesador a la hora de ejecutar tareas de este tipo.

En este trabajo se mostrarán las ventajas que tiene utilizar las tarjetas gráficas a la hora de paralelizar autómatas celulares, mostrando métricas que comprueben su validez de forma cuantitativa. Para ello se divide el trabajo en 7 capítulos:

- **Capítulo 1. Antecedentes.** Se explican las bases de HPC, las diferencias entre CPU y GPU, arquitecturas, las herramientas y conceptos que serán utilizados en los demás capítulos.
- **Capítulo 2. Selección del caso.** En este capítulo se explica mejor el autómata celular elegido y las razones por las que se eligió este modelo.
- **Capítulo 3. Implementación en secuencial.** Dentro de este capítulo se muestran diagramas y secciones de código que se utilizaron para implementar el algoritmo que resuelve el autómata celular seleccionado de forma secuencial.
- **Capítulo 4. Implementación en paralelo.** En este capítulo se muestran los diagramas y las secciones de código más importantes que se utilizaron para implementar el algoritmo de forma paralela. También se muestra la diferencia entre como es la ejecución y la implementación secuencial contra su similar en la paralelo.
- **Capítulo 5. Métricas de rendimiento.** Se exponen los resultados con métricas de rendimiento para determinar si en verdad es útil realizar el procesamiento de un autómata celular en paralelo y con tarjeta gráfica o no.
- **Capítulo 6. Conclusiones.** En este capítulo, con ayuda de los datos y métricas del anterior, se explican los resultados obtenidos y se determina si la implementación realizada fue o no útil.
- **Capitulo 7. Trabajo futuro.** Dentro de este capítulo se generan ideas que podrían ser tomadas para mejorar este trabajo.

Capítulo 1

Antecedentes

1.1. Cómputo de alto rendimiento

Para poder comprender un concepto tan importante se puede comenzar imaginando un problema en el cual, se invertiría gran cantidad de tiempo para poder llegar a resolverlo, aún con una computadora ya que, aunque las computadoras han sido destinadas a resolverlos de una forma más rápida y precisa, existen algunos que requieren una inversión de tiempo extensa para poder llegar a dicha solución.

El cómputo de alto rendimiento tiene como objetivo reducir significativamente la inversión de tiempo para resolver los problemas de forma confiable y eficiente, lo que puede lograrse con la ayuda de muchos procesos, los cuales trabajan al mismo tiempo de forma paralela.

De acuerdo con los datos obtenidos de la página oficial del INEGI en colaboración con el IIMAS de la UNAM[9], para que se pueda considerar a una computadora dentro del cómputo de alto rendimiento, esta debe contar con un sistema que opere arriba de un teraflops, es decir, que realice 10^{12} o más operaciones de punto flotante por segundo.

La empresa AMD[2] explica que existe la computación heterogénea, que es aquella en la que podemos encontrar procesos que son ejecutados en las unidades centrales de procesamiento (CPU) y/o en las unidades de procesamiento gráfico (GPU). Actualmente la computación heterogénea está presente en nuestro día a día, ya que la podemos encontrar en las consolas de videojuegos, los celulares, las computadoras personales, etc.

1.1.1. GPU en cómputo de alto rendimiento

Las GPU son unidades de procesamiento que de forma inicial, fueron diseñadas con el objetivo de procesar más rápido los gráficos en una computadora. Conforme se fueron utilizando, se pudo notar que además de dicho procesamiento, las GPU podían ofrecer más ya que toman una matriz de datos y trabajan sobre ella de forma simultánea, realizando una serie de tareas al mismo tiempo. Todo esto es lo que las diferencia de las CPU, las cuales en vez de trabajar en simultaneo, toman dato por dato, tal como se puede visualizar en la figura 1.1.

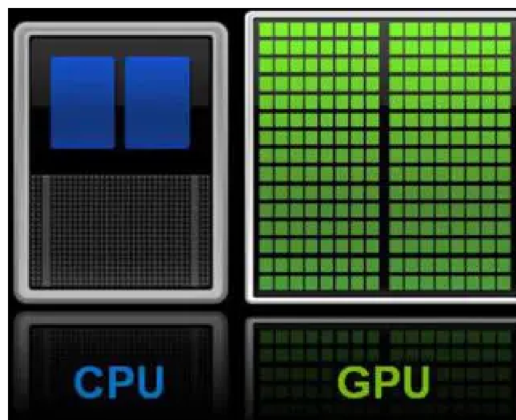


Figura 1.1: CPU vs GPU [11]

Dicha capacidad de tomar un conjunto de datos y trabajar sobre ellos de forma paralela, hace que las GPU se vuelvan una gran herramienta para el cómputo de alto rendimiento.

Es por ello que la computación heterogénea es tan utilizada hoy en día. Si todo el tiempo se utilizara la CPU, se tendrían tiempos de procesamiento muy extensos por otra parte, si únicamente se hiciera uso de la GPU se desperdiciarían recursos y su alto rendimiento. Puede considerarse como una opción viable y conveniente hacer uso de ambas unidades de procesamiento, para que de esta manera, según el número de datos a manipular, se realice una repartición de tareas.

1.1.2. Taxonomía de Flynn en el cómputo de alto rendimiento

Una taxonomía es la clasificación de cosas de acuerdo a características en común, ésta se utiliza para describir la cantidad de control concurrente[34]. En 1972, Michael J. Flynn propuso cuatro clasificaciones basadas en el número de instrucciones concurrentes y en

los flujos de datos según la arquitectura que posea el sistema, dichas clasificaciones son las siguientes:

- SISD Estas siglas provienen del idioma inglés, y hacen referencia a la existencia de una sola instrucción y de un solo dato. Se puede analizar su diagrama en la figura 1.2. Sus características son:
  - La unidad de proceso toma una instrucción a la vez por cada ciclo de reloj.
  - Por cada ciclo de reloj se toma un solo dato.
  - Es la forma más antigua en que se estableció para la computación.

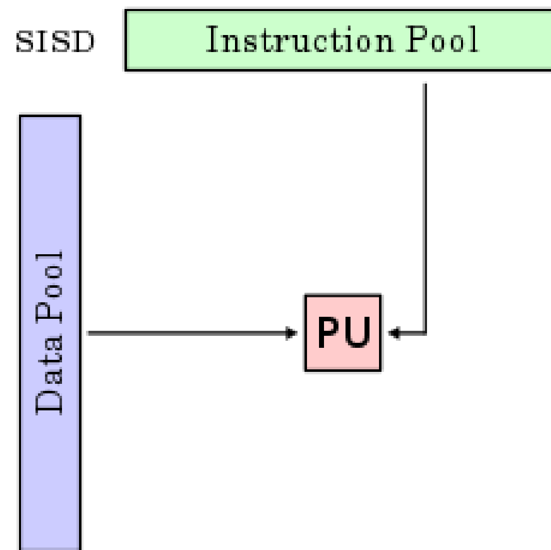


Figura 1.2: Diagrama de SISD[18]

- SIMD Estos sistemas tienen una sola instrucción pero múltiples datos, lo cual se utiliza para tener una paralelización en cuanto a datos. Con ayuda del diagrama de la figura 1.3 se observa mejor el comportamiento. Las características que ofrece son:
  - Cada unidad procesará la misma instrucción.
  - Cada unidad procesa un dato distinto.
  - Las unidades operan al mismo tiempo.

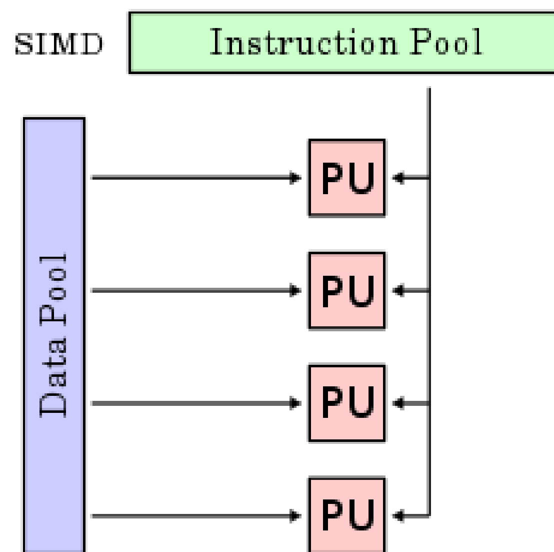


Figura 1.3: Diagrama de SIMD[18]

- MISD Es un tipo de arquitectura enfocada a los sistemas paralelos, donde se tienen múltiples instrucciones, pero sólo un dato como se muestra en el diagrama de la figura 1.4. Las principales características son:

  - Cada unidad procesará el mismo dato.
  - Cada unidad procesa una instrucción distinta.
  - Hay pocas aplicaciones que utilizan esta arquitectura.

- MIMD

  En este tipo de arquitectura existen múltiples instrucciones y múltiples datos, donde cada unidad puede procesar diferentes instrucciones en cualquier momento. El diagrama de la figura 1.5 ilustra mejor su funcionamiento. Las características con las que cuenta son:

  - Cada unidad procesará distintos datos.
  - Cada unidad procesa una instrucción distinta.
  - Cada unidad opera simultáneamente.

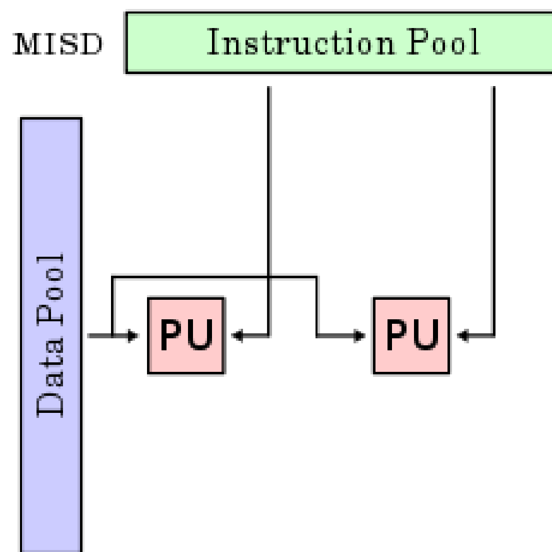


Figura 1.4: Diagrama de MISD[18]

1.2. Impacto de las tarjetas gráficas

Hoy en día, las computadoras no son un lujo, se han vuelto una necesidad del ser humano debido a las actividades que se ejecutan diariamente, desde trabajos de oficina hasta desarrollo de aplicaciones. Es por ello que dependiendo las tareas a realizar las computadoras cuentan con distintas características como velocidad de procesamiento, capacidad de almacenamiento y consumo de energía.

Aún con todo lo anterior, existen computadoras que no están diseñadas para uso comercial, es decir no solo para ejecutar videojuegos o manipular hojas de cálculo, sino para el procesamiento masivo de datos donde las operaciones a realizar son más complejas.

Año con año, se realizan pruebas para valuar las métricas de rendimiento, con los datos obtenidos se crean dos listas que posteriormente son publicadas, en las cuales aparecen las 500 computadoras más potentes del mundo, denominadas como supercomputadoras. La primera lista del año es publicada en junio, mientras que la segunda etapa se realiza en noviembre.

Tomando como referencia la primera lista anual (junio de 2020) se puede apreciar que las primeras posiciones de las supercomputadoras se apoyan de tarjetas gráficas para lograr el número de operaciones de punto flotante que las hacen estar en la cima.[16]

Se denota la importancia que han tomado las tarjetas gráficas y cómo pasaron de la

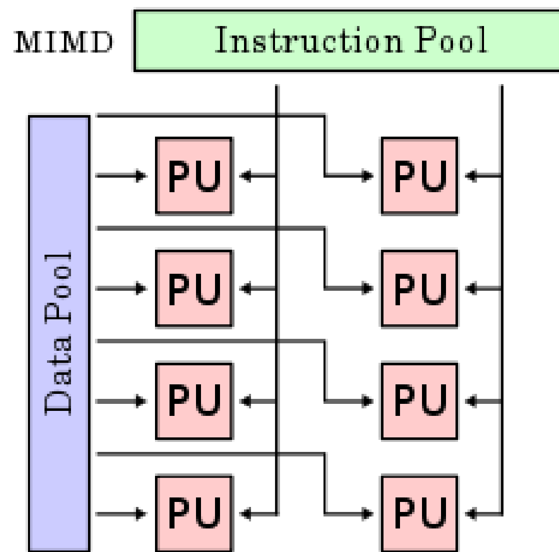


Figura 1.5: Diagrama de MIMD[18]

idea principal a ser parte de la computación de alto rendimiento.

1.3. GPGPU

Es el término que se le da al uso de las ventajas y desventajas que ofrece la GPU, las siglas derivan de su traducción del inglés "General-Purpose computation on Graphics Processing Units"[1] o en español Çomputación de propósito general en unidades de procesamiento de gráfico".

La GPU fue creada con el fin de mejorar la computación gráfica, mediante su capacidad de manejar grandes cantidades de píxeles al mismo tiempo. También han tenido gran impacto en el cómputo científico gracias a su destreza para trabajar sobre miles de datos; cuando una GPU es utilizada para un propósito distinto al de procesar los gráficos se puede decir que está haciendo uso de GPGPU.

Hay ocasiones en las que se debe trabajar sobre miles de datos de manera simultánea en tiempo real, ejecución que una CPU no podría realizar. Es prácticamente imposible trabajarlo por el tiempo de latencia que existe entre la comunicación de la obtención de datos, el procesamiento de los mismos y la entrega de resultados. Para poder mejorar el tiempo de procesamiento podemos hacer uso del GPGPU.

1.3.1. Cómputo gráfico y científico con GPUs

Una tarjeta gráfica actual puede ser utilizada para edición de videos, creación de modelos tridimensionales, desarrollo de videojuegos etc. así como también para la implementación de programas con fines distintos, desarrollo de algoritmos o ejecución de programas en forma paralela, aún cuando existen equipos especializados para dichas funciones.

La diferencia que existe entre un diseño y otro, desde el punto de vista del usuario, es la velocidad con que realizan las tareas, por ejemplo, una tarjeta gráfica enfocada a los gráficos conseguirá mejores tiempos de ejecución a la hora de implementar tareas como ejecución de videojuegos a comparación de una tarjeta gráfica destinada al cómputo científico, del mismo modo ocurre en el caso contrario.

1.4. Paralelización en tarjetas gráficas

Para lograr paralelizar un programa en una tarjeta gráfica se transfiere información entre la CPU y la tarjeta gráfica. Para lograr esto, se han creado distintas herramientas computacionales en cuanto a software.

Algunas de las herramientas más populares para trabajar sobre tarjetas gráficas son OpenCL, OpenGL, Metal y CUDA.

Aunque existen diversas herramientas, para este proyecto se eligió CUDA dado que es una plataforma que cuenta con gran soporte y facilidad de uso. Cuenta con la desventaja que solo se puede implementar en tarjetas gráficas de la marca Nvidia sin embargo, como se comentó anteriormente, este trabajo puede ser implementado en distintas plataformas tanto en hardware como en software haciendo uso de la misma lógica de programación.

CUDA (Compute Unified Device Architecture) es una API de programación en paralelo, creada como una variación del lenguaje de programación C diseñada para generar algoritmos enfocados en las GPU. Fue creada por NVIDIA para sus tarjetas gráficas a partir de la serie G8X.

CUDA aprovecha las características de las tarjetas gráficas Nvidia, como los múltiples núcleos, las memorias y los hilos que pueden trabajar sobre los datos de interés. Una de las ventajas que presenta CUDA con respecto a otras plataformas o lenguajes de programación en paralelo es que los desarrolladores pueden codificar utilizando lenguajes de programación populares como en el caso de Python, C, C++ y Fortran, lo que lo vuelve más sencillo para personas que no están tan familiarizadas con el paralelismo.[22]

Para este trabajo se eligió el lenguaje C para, ya que es donde más información se encontró a la hora de implementar el algoritmo.

1.4.1. Kernels

Cuando se utiliza CUDA, se cuenta con dos tipos de dispositivos para trabajar sobre ellos, la CPU y la GPU, en CUDA se les conoce como host y device respectivamente.

Al escribir el código compatible con CUDA, se utilizan los kernels para hacerle saber a la tarjeta gráfica que debe ejecutar una sección de código; esto podría ser lo equivalente a una función en un lenguaje de programación habitual, la diferencia es que las funciones son ejecutadas siempre en la CPU (host) y los kernels pueden hacer uso de la CPU o de la GPU(device).[21]

La estructura que debe tener un kernel es la que se muestra en la sección de código 1.1.

```
__global__ void myKernel(arg_1,arg_2,...,arg_n)
{
    //...Codigo para ejecutar en la GPU...
}
```

Código 1.1: Código de creación de kernels

Cuando se llama un kernel es necesario que se determine el número de hilos por bloque y el número de bloques por malla que son esenciales para ejecutar la sección de código, para pasar dicho parámetro al kernel debe ser llamado pasando los parámetros como se muestra en el código1.2.

```
dim3 dimBlock(O, P); // Bloque de dimensión O*P
dim3 dimGrid(N, M); // Malla de dimensión N*M
/*
 *  Procesos intermedios
 */
//Llamada del kernel
myKernel<<<dimGrid,dimBlock>>>(arg_1,arg_2,...,arg_n);
```

Código 1.2: Código para determinar el número de hilos en un kernel

Esto quiere decir, que se generarán OxP bloques y cada bloque tendría NxM bloques, en total habrán $NxMxOxP$ hilos[6].

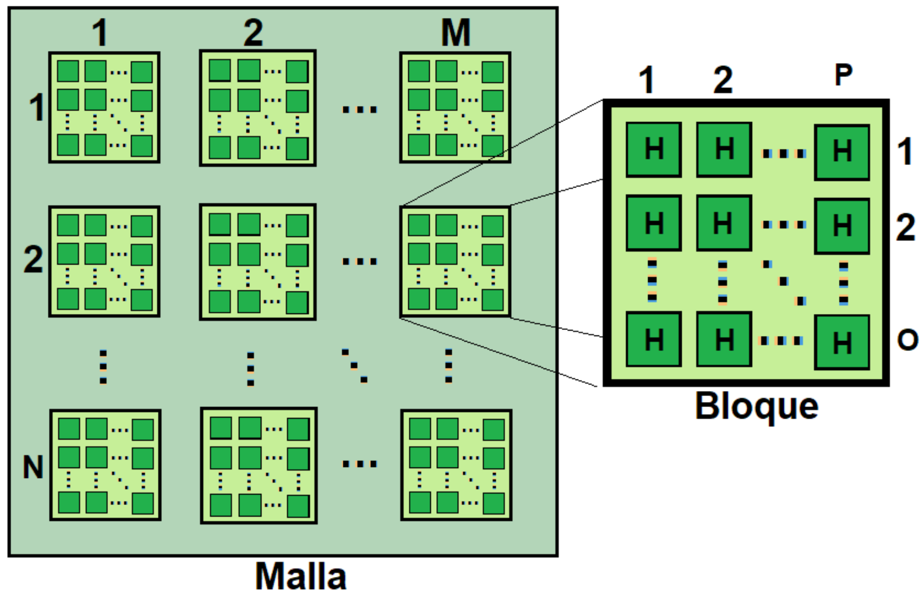


Figura 1.6: Malla con NxM bloques de OxP hilos. Imagen basada en [7]

1.4.2. Malla, bloques e hilos

Para hacer uso de todo el kit de desarrollo que ofrece CUDA, se deben considerar los recursos con los que cuenta el desarrollador, al momento de paralelizar un programa en CUDA, son los hilos los que en verdad realizan todo el trabajo, la forma de asignarles a cada uno su trabajo es mediante su identificador, de esta forma pueden realizar trabajos distintos de forma paralela.

En CUDA se tiene un bloque que es un conjunto de hilos y la malla que es el conjunto de todos los bloques que tiene la tarjeta gráfica, tal y como se visualiza en la figura 1.7. El número de bloques depende del modelo de cada tarjeta gráfica.[21]

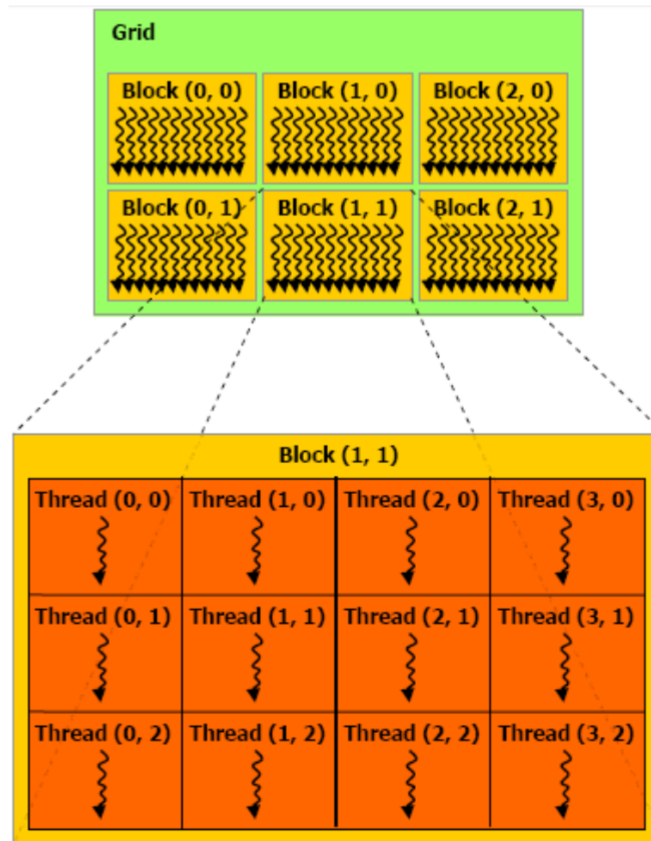


Figura 1.7: Diagrama de la distribución de la malla, los bloques y los hilos[18]

1.4.3. Memoria compartida

Con la diferencias definidas entre malla, bloque e hilo, es posible saber cuántos hilos trabajarán simultáneamente, sin embargo, se debe conocer que cada estructura tiene su propia memoria y que, incluso, la misma tarjeta gráfica cuenta con una memoria interna.[21]

Para acceder de forma inmediata a los datos que se encuentran en el host es necesario compartirlos al device para así poder trabajar sobre ellos.

Inicialmente, el programa tendrá los datos en el host, para poder transmitirlos se debe indicar a CUDA que se requiere memoria con la función mostrada en la la sección de código 1.3.

```
1 cudaMalloc(void **devPtr, size_t size);
```

Código 1.3: Código de creación de kernels

El primer argumento de la función *cudaMalloc()* es un apuntador a la dirección de la variable donde se quiere almacenar la memoria; el segundo argumento indica el tamaño de memoria que se requiere.

Una vez que se ha asignado la memoria, se realiza la transferencia de los datos del host al device, lo que puede llevarse a cabo con la función del código 1.4.

```
1 cudaMemcpy(void *dst, void *src, size_t size, memcpyKind tipo);
```

Código 1.4: Código de creación de kernels

El primer argumento de la función *cudaMemcpy()* representa el destino de la memoria, es decir, en donde se almacenarán los datos; el segundo, es el origen de los mismos, la variable donde originalmente se encuentra; en el tercer argumento se denota el tamaño de los datos y, por último, debe especificarse el tipo de transferencia que se realiza. En la tabla 1.1 se muestran los tipos de transferencia y el sentido en el que se transfieren los datos.

Tipo de transferencia	Sentido de transferencia
cudaMemcpyHostToHost	Del host al host
cudaMemcpyeviceToHost	Del device al host
cudaMemcpyHostToevice	Del host al device
cudaMemcpyeviceToDevice	Del device al device

Tabla 1.1: Tabla de transferencia de datos en CUDA[23]

Cuando se transfieren los datos al device, este los almacena en la memoria global y así, los hilos pueden acceder a los datos. Cada bloque cuenta con memoria compartida a la cual solo se puede acceder a través de los hilos que pertenecen a ese bloque y a través de los que cuentan con memoria local y registros; esta es la memoria más rápida que existe en CUDA, pero sólo puede ser accesada por su hilo.

Dentro de la malla también se encuentra la memoria constante, cuyo objetivo es almacenar los datos que no cambian durante el desarrollo de la aplicación.

Finalmente, se cuenta con la memoria de textura, cuya utilidad está enfocada a las aplicaciones gráficas, sin embargo, también se puede acceder a ella para otros fines computacionales. Estos dos tipos de memoria son utilizadas para mejorar el rendimiento de las

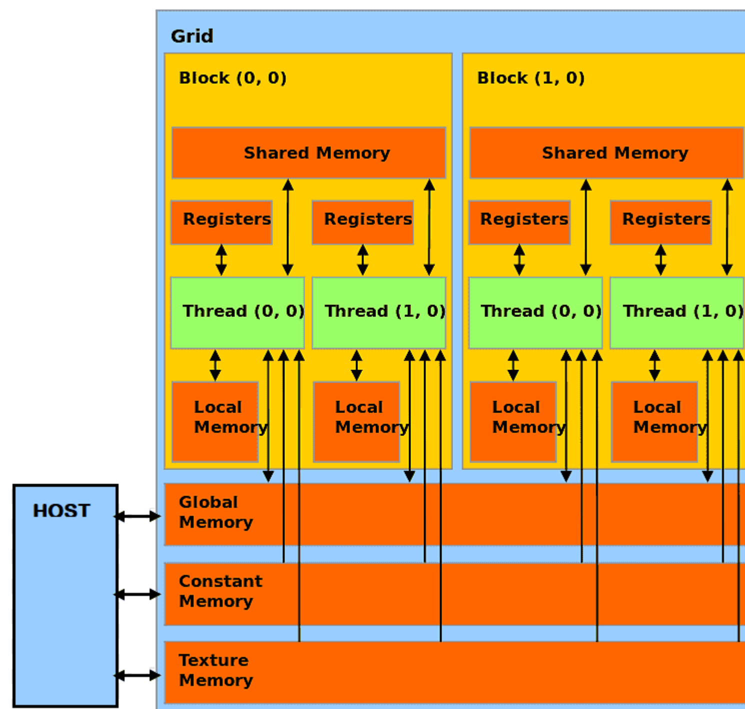


Figura 1.8: Diagrama de la memoria en CUDA[30]

aplicaciones desarrolladas, pues son capaces de disminuir el tráfico entre los otros buses de datos y se tiene mejor organización de los datos.

En la figura 1.8 se ilustra cómo es la comunicación entre las memorias; la flecha indica hacia dónde es el flujo de la información, si es doble quiere decir que puede transferir datos en ambas direcciones.

1.4.4. Multiprocesadores SM

La arquitectura CUDA se basa en una matriz escalable de multiprocesadores de transmisión (SM) multiproceso. Cuando un programa CUDA en la CPU del host invoca una cuadrícula del núcleo, los bloques de la cuadrícula se enumeran y distribuyen a los multiprocesadores con capacidad de ejecución disponible. Los subprocesos de un bloque de subprocesos se ejecutan simultáneamente en un multiprocesador, y varios bloques de subprocesos se pueden ejecutar simultáneamente en un multiprocesador. A medida que terminan los bloques de subprocesos, se lanzan nuevos bloques en los multiprocesadores desocupados.

Dependiendo del número de nucleos cuda y SM

1.5. Modelos y Sistemas

1.5.1. Modelo

El concepto "Modelo"puede tener significados diversos que dependen del área en donde se estudie o utilice; desde el punto de vista científico e ingenieril, es una abstracción de la realidad que se utiliza para analizar, describir y explicar un problema[29].

Dentro de los modelos en el ámbito científico se desprenden distintos tipos, los cuales se especifican en distintas áreas; en el presente trabajo se enfoca en los siguientes modelos:

- Modelo Conceptual: Es donde se establecen las bases del problema que está por resolverse; se abstraen los puntos claves. Puede decirse que es donde suministrar la información más relevante para resolver el problema.

- Modelo Matemático: El modelo matemático es la abstracción del problema en ecuaciones, funciones y cualquier otro método o herramienta matemática que ayude a solucionar el problema[24].

- Modelo Computacional: Es toda la implementación que se realiza en la computadora, por ejemplo, la codificación en algún lenguaje de programación, los pseudocódigos, los algoritmos y todas las herramientas computacionales que ayuden a dar solución el problema[29].

1.5.2. Sistema

En la actualidad la palabra sistema se puede encontrar en diversos ámbitos. Cotidianamente se considera como una forma de realizar algo, sin embargo, para fines de este trabajo se necesita una definición más completa.

Se puede describir a un sistema como un conjunto de elementos que trabajan juntos con un fin en común, esto se debe a que los elementos se afectan entre sí[15].

Los sistemas cuentan con varias clasificaciones, entra las cuales se destaca que un sistema puede pertenecer a más de una clasificación[15].

1.5.3. Sistema Multiagente

Este tipo de sistemas tienen la característica de trabajar con agentes. Los agentes son procesos autónomos e inteligentes que se encargan de tomar decisiones a través de lo que

captan sus sensores, con lo cual, activa sus actuadores. El agente está en constante análisis de las entradas en los sensores para seguir tomando decisiones [13].

En estos sistemas los agentes trabajan de forma separada e independiente, pero con un fin en común, resolver el problema.

1.5.4. Sistema Complejo

Para que un sistema pueda ser llamado complejo debe cumplir tres características únicas, las cuales son:

- Se componen por grandes cantidades de elementos que resultan ser muy similares.
- Sus elementos interactúan de forma local, por lo cual, no es posible analizar un elemento separado de los otros para dar una explicación a lo que realiza todo el sistema.
- Es prácticamente imposible predecir cómo evolucionará este sistema a partir de una entrada dada[15].

1.5.5. Sistema Caótico

Un sistema caótico puede llegar a confundirse con un sistema complejo aunque sean distintos; también cuentan con tres características que ayudan a clasificarlo como otro tipo de sistema, las cuales son:

- Si se hace un cambio mínimo a la entrada se obtendrá un resultado totalmente distinto al que se obtendría con la entrada original.
- Estos sistemas no son lineales, es decir, las reglas matemáticas que los modela y rigen no cumplen con la superposición.
- Predecir el comportamiento de estos sistemas es básicamente imposible, normalmente para saber el comportamiento del sistema ante una entrada debe calcularse o esperar a que suceda dicho proceso. Esta es la característica que comparte con los sistemas complejos[15].

1.5.6. Sistema Estáticos

Los sistemas estáticos son aquellos donde las salidas dependen únicamente de las entradas actuales, otra forma de describirlos puede ser como los sistemas cuyas salidas cambian con el tiempo, pero de manera simultánea cambiar las entradas en forma semejante. De esta forma, se puede decir que una entrada siempre tendrá el mismo resultado sin importar en qué instante de tiempo sea la entrada[27].

1.5.7. Sistema Dinámico

Estos sistemas cambian a partir de las entradas actuales, pero también dependen de las entradas previas. Estos cambian a través del tiempo dependiendo de las entradas y de las condiciones iniciales del sistema; las salidas de estos no siempre son las mismas aunque las entradas actuales sí lo sean[27].

1.6. Autómatas Celulares

Fueron introducidos inicialmente por Jon Von Neumann quien trató de idealizar y modelar un sistema biológico; la idea principal era que estos sistemas pudieran controlarse y reproducirse por sí mismos. Su nombre nace por su similitud con el crecimiento celular cuando se reproduce, de aquí mismo nace el nombre de la unidad más pequeña que contienen los AC[1], dicha unidad se le denominó el nombre de célula[5].

Después de algunas iteraciones los autómatas pueden o no cambiar su estado dependiendo de sus reglas, en la figura 1.9 se muestra un ejemplo de un AC que ha evolucionado después de algunas iteraciones.

[1] A partir de este momento se utilizarán las siglas AC para hacer referencia a los autómatas celulares.

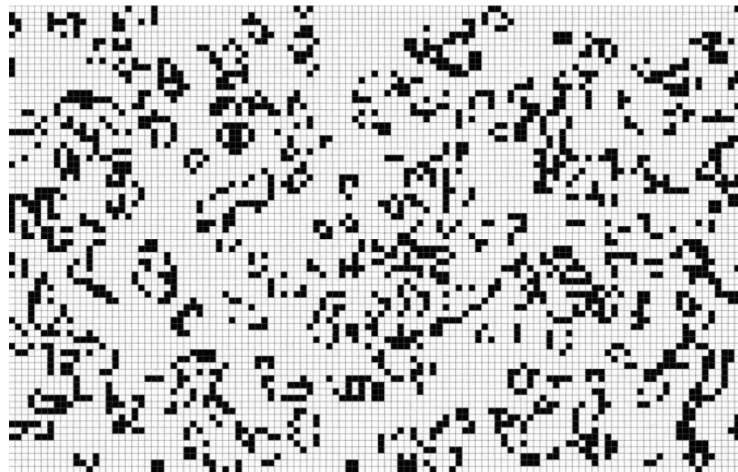


Figura 1.9: Ejemplo de un autómata celular[10]

Las aplicaciones más comunes que tienen los AC se dan en el cómputo científico para simular y modelar algunos problemas comunes, pueden ser muy precisos dependiendo si las reglas de transición están bien implementadas[4]. Algunos ejemplos son:

- Evolución de ecosistemas biológicos.
- Crecimiento de incendios forestales.
- Tráfico vehicular.
- Propagación de virus.
- Avance de una avalancha.

Ahora que se comprende la importancia de los AC, se puede dar inicio a la descripción de sus características, con el fin de dar al lector las bases de los términos que serán utilizados a lo largo del trabajo.

Los AC cambian sus estados de forma discreta mientras van evolucionando, por lo tanto, cada célula puede o no cambiar en cada pulso de reloj dependiendo de su estado actual, el de las células vecinas y las reglas de transición implementadas para el AC[3].

Un AC tiene un estado inicial, este estado es configurado antes de cualquier ejecución. Se sabe que un AC es un sistema invariante en el tiempo, lo cual quiere decir que no importa en qué momento se ejecute, si el estado inicial (la entrada al sistema) es la misma, siempre obtendremos el mismo resultado (respuesta del sistema) después del mismo número de iteraciones.

Un conjunto de células es el espacio donde los AC se desarrollan, a este espacio se le conoce como malla. Existen distintos tamaños y tipos de mallas, de las cuales se hablará más adelante.

1.6.1. Espacio Celular

El espacio celular hace referencia a la forma que tendrán las células y, por ende, a la forma que tendría la malla, lo más común es encontrar autómatas celulares cuadrados pero dependiendo del problema que se debe resolver se puede elegir un espacio celular diferente, por ejemplo un espacio triangular, cuadrado, hexagonal, etc[20].

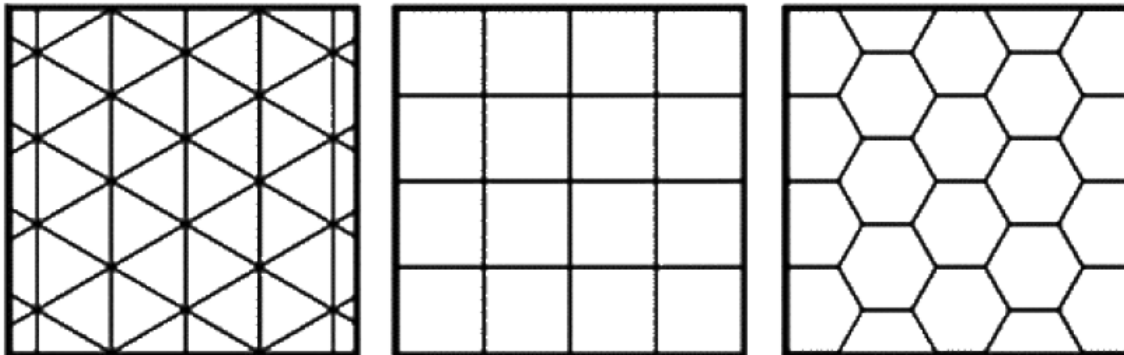


Figura 1.10: Espacio celular triangular, cuadrado y hexagonal[25]

1.6.2. Condición de frontera

Los AC al ser una combinación de modelos matemáticos y numéricos, pueden ser implementados y procesados con mayor facilidad en computadoras. El problema de ello es que las computadoras cuentan con recursos finitos en cuanto a memoria y a procesamiento. Si quisiéramos modelar un AC de dimensiones demasiado grandes no podríamos procesarlo en una computadora debido a que los recursos son limitados, se tendria que delimitar el AC.

Para resolver dicho problema se implementan distintas condiciones de frontera, las cuales se eligen dependiendo del modelo y el problema real a procesar. Para que la malla no sea de un tamaño infinito nos enfocamos en las células que se encuentran en las orillas de la malla.

Los tipos de frontera que existen en los AC son los siguientes[25]:

- Frontera cerrada: Las células que están en los límites de la malla únicamente revisan las que estén disponibles en ese momento. No toman valores extra, simplemente los valores que están a su alcance.

- Frontera abierta: Las células fuera de la malla toman un valor fijo. Podemos establecer un valor a las células dependiendo del problema a resolver.

- Frontera reflectora: Los valores fuera de la malla toman el valor que tiene la célula que quiere acceder. En este tipo de frontera se analiza una célula y si se encuentra en algún extremo de la malla, los valores vecinos fuera de la malla toman el mismo valor que el que tiene la célula analizada.

- Frontera circular: Una célula en la frontera revisa a sus vecinas y a las células que se encuentran hasta el otro extremo de la malla. Este tipo de frontera hace que no se establezca un valor fijo y a la vez que analice todos los valores.

- Sin frontera: No existen límites en la frontera.

1.6.3. Reglas de transición

Los autómatas cambian de forma discreta, pero no de forma aleatoria debido a que los resultados dependen de algunas propiedades ya vistas. Dentro de dichas propiedades se tienen las reglas de transición.

Para que el AC cambie de estado a manera que tenga sentido, se necesita al menos, una regla o un conjunto de ellas. A esta o su conjunto se le llaman reglas de transición o función de transición, las cuales se encargan de analizar el estado actual de cada una de las células y de sus vecinas; dependiendo de él, se decide el siguiente estado del AC.

1.6.4. Geometría de la vecindad

Se ha hablado mucho de la vecindad y se puede comprender que es el conjunto de células que rodea a la célula que está siendo analizada[8]. Se pueden encontrar 2 tipos de vecindad para los AC de espacio celular cuadrado:

- Von-Neuman: en este tipo de vecindad las células que nos interesan analizar son 4, las que se encuentran arriba, abajo, izquierda y derecha.

- Moor: este tipo de vecindad considera 8 células en total, trabaja sobre las mismas células que la vecindad de Von-Neuman y agrega las 4 esquinas.

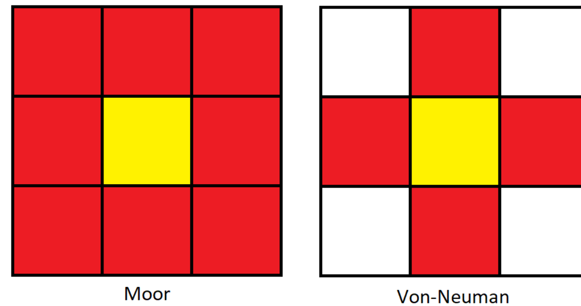


Figura 1.11: Geometría de Moore y Von-Neuman. Basado en [26]

1.7. Dimensionalidad

Un AC puede trabajar en el número de dimensiones que se requiere para darle solución al problema presentado. El más básico es un autómata celular unidimensional, los más usados son los bidimensionales y hay ocasiones donde se requiere utilizar modelos en n dimensiones.

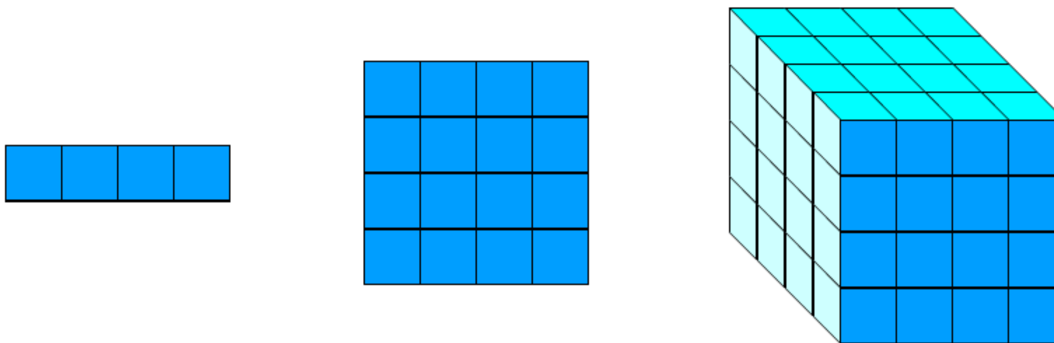


Figura 1.12: Posibles dimensiones de un AC. Basado en [26]

1.7.1. Clasificación de Wolfram

Stephen Wolfram es un cientifico britanico nacido en 1959 quien ha hecho grandes aportaciones a la computación, las matemáticas y la física. Entre estas aportaciones se

puede encontrar la clasificación de wolfram.

Wolfram define 4 clases para los autómatas celulares[34], estas son explicadas a continuación.

- Clase I: Casi todos los patrones iniciales evolucionan rápidamente en un estado estable y homogéneo. Cualquier aleatoriedad en el patrón inicial desaparece.

- Clase II: Casi todos los patrones iniciales evolucionan rápidamente hacia estructuras u oscilantes. Parte de la aleatoriedad del patrón inicial puede permanecer, pero solo algunos restos. Los cambios locales en el patrón inicial tienden a permanecer locales.

- Clase III: Casi todos los patrones iniciales evolucionan de forma pseudoaleatoria o caótica. Las estructuras estables que aparecen son destruidas rápidamente por el ruido circundante. Los cambios locales en el patrón inicial se propagan indefinidamente.

- Clase IV: Casi todos los patrones iniciales evolucionan en las estructuras que interactúan de manera compleja e interesante, con la formación de las estructuras locales que son capaces de sobrevivir largos periodos de tiempo. Podría ser el caso de que aparecieran estructuras estables u oscilantes, pero el número de pasos necesarios para llegar a este estado puede ser muy grande, incluso cuando el patrón inicial es relativamente simple. Los cambios locales en el patrón inicial pueden extenderse indefinidamente.

1.7.2. Reglas de Wolfram

Stephen Wolfram, en su libro .A new kind of science"[34] denomina a una regla como un algoritmo o una identidad simple, por lo que una regla se podría ser descrita como los pasos a seguir para evolucionar un autómata.

El autómata más sencillo de analizar e implementar es el unidimensional con células de dos posibles valores (viva o muerta) y con reglas de sus dos vecinos adyacentes, se puede concluir que depende de solo 3 células para determinar su estado siguiente. Como cada célula tiene dos posibles estados, tenemos $2x2x2 = 2^3 = 8$ posibles combinaciones para las tres células vecinas de una célula dada. Existen un total de $2^8 = 256$ autómatas celulares elementales o también conocidos como reglas de Wolfram.

Un AC elemental puede ser identificado por un índice denotado por su número binario correspondiente en 8 bits. La regla 30 (00011110_2) es una de las más populares, sus posibles estados son las mostradas en la figura 1.13.

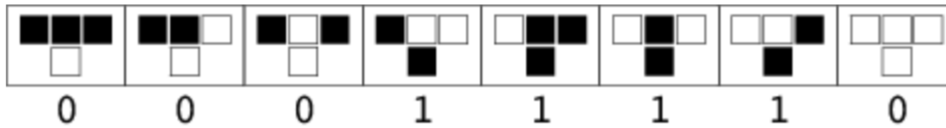


Figura 1.13: Regla 30[31]

Con los estados mostrados en la figura 1.13 y después de 15 iteraciones de este autómata celular elemental el resultado es el mostrado en la figura1.14.

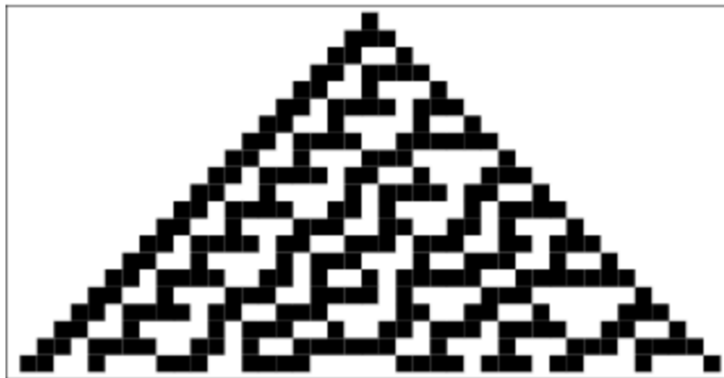


Figura 1.14: Regla 30 después de 15 iteraciones[31]

1.8. Tipos de Autómatas celulares

Al igual que en los sistemas y los modelos, en los AC existen distintos tipos que surgen a partir del modelo original variando algunas de sus propiedades o características[3]. Los tipos de AC más populares son:

- Probabilísticos: Surgen cuando la regla de transición se rige por una ley o un modelo probabilístico[17]. Por lo cual el valor de cada célula representa una probabilidad.
- Reversibles: Es aquel que puede calcular el estado anterior a partir del estado actual y sus reglas de transición.
- Totalístico: Son aquellos que su transición dependen únicamente de la suma de su vecindad[3].

- Elemental: Es unidimensional, en cuyos estados de la célula son 1 o 0 (vivo o muerto), por lo cual el estado siguiente depende de sus únicos dos vecinos.

1.9. Herramientas para simular AC

Existen herramientas bastante útiles para simular los estados y las reglas de un autómata un ejemplo es el sistema NetLogo.

NetLogo es una herramienta que se puede utilizar para modelar con ayuda de múltiples agentes y simular los datos de entrada. Esta herramienta fue creada por Uri Wilensky y se desarrolló por CCL[32].

La importancia de esta herramienta es que puede ser utilizada para simular autómatas celulares de forma gráfica. Este programa cuenta con dos modos de uso, puede ser usado al instalarlo en una computadora o trabajar en línea.

Otra de las ventajas que presenta NetLogo es que se trata de código abierto, esto quiere decir que es posible descargar el código para comprender su funcionamiento, realizar mejoras o crear modelos desde cero.

Como NetLogo, existen otros programas que ayudan a simular autómatas celulares, pero realizan la programación de forma secuencial y los que paralelizan los procesos siguen trabajando dentro de las CPU.

1.10. Requerimientos

Cuando se está desarrollando un sistema es necesario especificar cuales son los criterios que se deben cumplir para cubrir lo más esencial para que el sistema funcione, a estos criterios se les conoce como requerimientos. [12].

El objetivo del levantamiento de estos ayuda a entender lo que el cliente y el usuario necesitan que haga el sistema. Los requerimientos identifican el propósito, mientras que el diseño especifica el cómo.

1.10.1. Requerimientos funcionales y no funcionales

Es normal que los requerimientos se dividan en dos tipos, funcionales y no funcionales[12].

Los requerimientos funcionales especifican cómo es que debe trabajar el sistema, cómo es su interacción con el ambiente y cómo comportarse a determinadas acciones. De la

misma forma puede establecer que cosas no debe hacer el sistema.

Mientras que los no funcionales describen restricciones que pueden limitar la elección de solución de problemas. Todo esto puede incluir las restricciones como tipos de proceso por utilizar, fiabilidad, tiempo de respuesta, almacenamiento, seguridad, rendimiento, etc.

El cumplimiento de los requerimientos funcionales es relativamente sencillo de comprobar, ya que en la mayoría de los casos solo se requiere su correcto funcionamiento. Por otra parte,los no funcionales deben someterse a diversas pruebas dependiendo de lo que se requiera medir o probar.

A la hora de analizar el rendimiento se pueden implementar diversas métricas que muestren la tendencia del desempeño del sistema [12].

1.11. Métricas

Teóricamente es posible visualizar que hay casos donde la computación en paralelo es mejor que la secuencial, pero se requiere algo más que sólo palabras e imaginación para describir esto. La ingeniería y la ciencia se basan en números, estadísticas y gráficas para comprobar los hechos, es por ello que se implementan las métricas para verificar si realmente es mejor la implementación en paralelo.

Existen unas métricas para poder valuar el desempeño de un algoritmo en secuencial y poder compararlo con su equivalente en paralelo[20].

La computación en paralelo depende de la arquitectura con la que se trabaja, el número de datos, los procesadores que se utilizan, el algoritmo que se implementa para dicha arquitectura y las velocidades de comunicación entre los componentes. Teniendo en cuenta dichos factores se puede encontrar el punto que marca la pauta para comenzar a hacer uso de la implementación en paralelo o seguir usando la versión en secuencial.

En algunos problemas, aunque tengan muchos datos, sigue siendo factible hacer uso de un programa en secuencial debido a que la transferencia de datos en la versión en paralelo podría generar un retraso y hacer que se vuelva más lenta la ejecución, es por ello que se vuelve importante encontrar el punto donde es mejor la ejecución del programa en paralelo.

A continuación, se explican las métricas más utilizadas para medir el buen funcionamiento de los algoritmos en paralelo.

1.11.1. Tiempo de ejecución (Runtime)

El tiempo de ejecución o runtime es el tiempo que dura el algoritmo procesando los datos hasta encontrar la solución del problema o de terminar su ejecución. Es la medida más popular para comparar la velocidad de los algoritmos por ser una implementación bastante sencilla y porque los datos obtenidos son fáciles de interpretar[33].

Se puede interpretar que T(1) o Ts es el tiempo en secuencial ejecutado en un solo procesador, mientras que T(n) o Tp es el tiempo de ejecución en paralelo con n procesadores.

Existen 3 casos donde:

- Si $T(1) > T(n)$ quiere decir que el tiempo en paralelo es menor, por lo tanto es mejor hacer uso de dicha implementación.
- Si $T(1) = T(n)$ quiere decir que da lo mismo si se ejecuta el programa en paralelo o en secuencial porque el tiempo de espera para el resultado es el mismo.
- Si $T(1) < T(n)$ quiere decir que es mejor hacer uso del algoritmo secuencial porque está tardando más el algoritmo en paralelo dado a distintos factores.

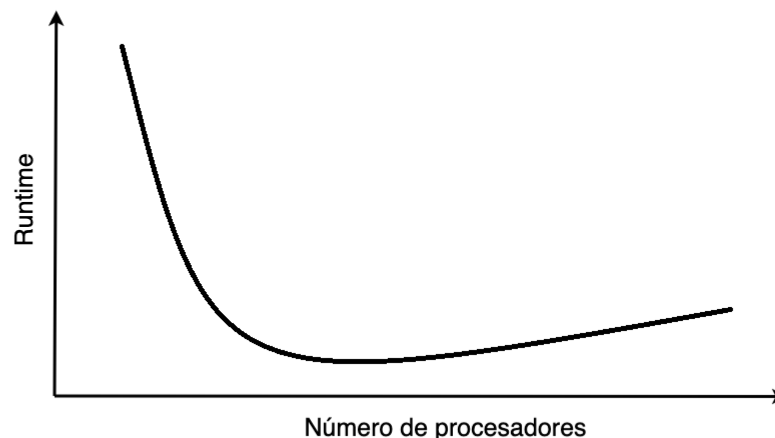


Figura 1.15: Gráfica de runtime variando el número de procesadores

En la figura 1.15 es posible visualizar cómo la curva comienza con una pendiente negativa, descendiendo de forma casi constante. Como el número de datos no cambia, es decir, el algoritmo siempre trabaja realizando el mismo número de procesos, al aumentar el número de los procesadores la carga se distribuye entre ellos, si en un inicio un solo procesador realizaba 100 operaciones, al aumentar el número de procesadores el trabajo se distribuye entre ellos bajando el tiempo de ejecución.

Llega un punto en el que la curva ya no baja de la misma forma y comienza a cambiar la pendiente a positiva, esto se debe a que el número de procesadores es mayor al número de procesos; cada procesador ahora realiza más trabajo porque se reparten mal los procesos y se desperdician los recursos.

1.11.2. Aceleración (Speedup)

El speedup, también conocido como aceleración, debe ser medido con el mismo número de datos para que la comparación sea equivalente. En esta medida se debe considerar el tiempo de ejecución de ambos algoritmos y el tiempo secuencial entre el tiempo en paralelo [35]. La forma de calcular la aceleración es como se muestra en la ecuación 1.1:

$$S(n) = \frac{T(1)}{T(n)} \tag{1.1}$$

Donde:

- $S(n)$ es el Speedup con n procesadores trabajando en paralelo.
- $T(1)$ es el tiempo del algoritmo en secuencial.
- $T(n)$ es el tiempo del algoritmo en paralelo con n número de procesadores.

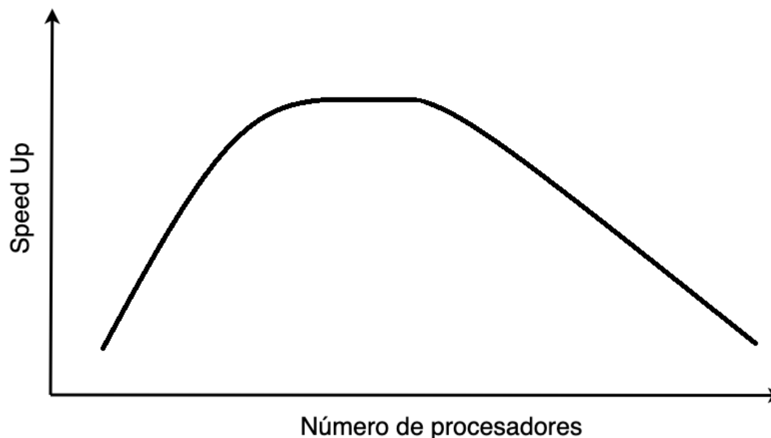


Figura 1.16: Gráfica de la aceleración variando el número de procesadores

Esta medida ayuda a saber cuántas veces es mayor o menor el tiempo de ejecución en su forma paralela, es decir, se nota si se volvió más rápido la ejecución o se redujo su velocidad.

Una forma sencilla de interpretar la aceleración es multiplicando el resultado por 100 para tener el resultado en porcentaje. Este porcentaje representa la velocidad perdida o ganada.

Por ejemplo, si el resultado es 100 % el algoritmo trabaja a la misma velocidad que su semejante en secuencial, no perdió ni ganó velocidad. Por otro lado, si el resultado es 150 % quiere decir que aumentó un 50 % de velocidad, pero si el resultado es menor a 100 por ejemplo 60 % la velocidad tuvo una perdida de un 40 %.

En la figura 1.16 se observa cómo la pendiente comienza positiva, es decir, la velocidad del algoritmo en paralelo aumenta con respecto a la velocidad de su similar en secuencial. Llega un punto en la gráfica en donde entra en una especie de meseta, en la que la pendiente es aproximadamente 0, esto quiere decir que la velocidad a partir de ese punto es constante.

Cuando la pendiente se vuelve negativa es porque la velocidad comienza a disminuir. Esta gráfica está fuertemente relacionada con la gráfica 1.15, por lo cual, en el momento en que la gráfica de la aceleración decrece es porque el tiempo de ejecución comienza a aumentar nuevamente.

1.11.3. Factor de costo (Costfactor)

Este dato debe calcularse porque aunque el tiempo que tarda en ejecutar el programa es menor, en un programa en paralelo debe tenerse en cuenta que el número de procesadores es mayor. Por ello, se debe calcular si es conveniente ahorrar tiempo a costa de gastar más recursos[33]. El cálculo del factor de costo es el mostrado en la ecuación 1.2:

$$C(n) = n * T(n) \tag{1.2}$$

Donde:

- $C(n)$ es el factor de costo.
- $T(n)$ es el tiempo de ejecución del algoritmo en paralelo con n número de procesadores.
- n es el número de procesadores.

Como el número de procesadores es 1 en la forma secuencial, al multiplicar 1 por el tiempo de ejecución el factor de costo se vuelve el mismo que el tiempo de ejecución. En la forma paralela cambia porque el número de procesadores es variable.

En las métricas anteriores se podía notar que hay ejecuciones que dan el mismo resultado si se realizan en paralelo o en secuencial, pero no es del todo cierto, dado que en la forma secuencial solo se está utilizando un solo procesador, mientras que en la forma paralela se usan n procesadores. Aunque el tiempo es el mismo, el gasto de recursos es mayor en la forma paralela.

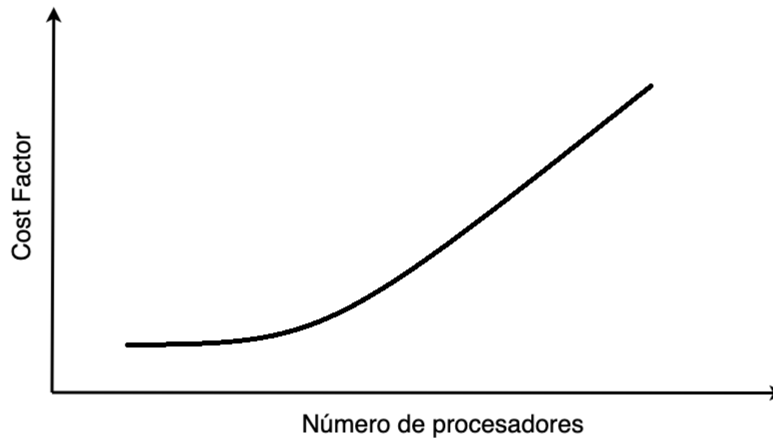


Figura 1.17: Gráfica del factor de costo

La figura 1.17 muestra la gráfica de forma teórica del factor de costo. Esta gráfica ayuda a ver si es conveniente ahorrar tiempo a costa del gasto de recursos. Esta aumenta porque a medida que avanza el eje horizontal, el número de procesadores es mayor, y a su vez, el producto también.

1.11.4. Eficiencia (Efficiency)

Esta medida es útil cuando queremos saber cuánto es el trabajo que realiza cada procesador en porcentaje. Se apoya de las medidas anteriores para conseguir los resultados. Para calcular la eficiencia se puede utilizar cualquiera de las siguientes ecuaciones 1.3:

$$
\begin{aligned}
E &= \frac{T(1)}{T(n) * n} * 100 \\
E &= \frac{T(1)}{C(n)} * 100 \\
E &= \frac{S(n)}{n} * 100
\end{aligned}
\tag{1.3}
$$

Donde:

- E es la eficiencia.
- $T(1)$ es el tiempo de ejecución del algoritmo en secuencial utilizando un solo procesador.
- $T(n)$ es el tiempo de ejecución del algoritmo en paralelo con n número de procesadores.
- $C(n)$ es el factor de costo.
- $S(n)$ es la aceleración.
- n es el número de procesadores.

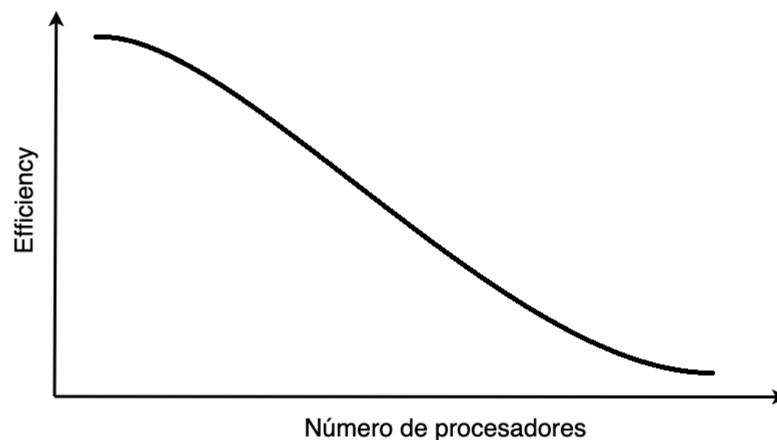


Figura 1.18: Gráfica de eficiencia

La figura 1.18 muestra la gráfica teórica de la eficiencia, en donde tiene una pendiente negativa porque entre mayor es el número de procesadores la carga de trabajo es menor.

1.11.5. Ley de Amdahl

Esta ley toma su nombre por su creador Gene Amdahl y muestra el aumento de rendimiento que puede tener una arquitectura mejorada. Se basa en la aceleración tradicional, pero se toma en cuenta que todo algoritmo, aún en paralelo, contiene una parte que

se ejecuta de forma secuencial, por lo cual aunque se aumente el número de procesadores infinitamente en un sistema, existe un límite en la aceleración debido a dicha parte secuencial[33].

De esta forma se puede entender que no solo el número de procesadores es el que determina la velocidad de un algoritmo, depende más del porcentaje del algoritmo que se puede o no paralelizar.

Con esto la ley de Amdahl quedaría representada matemáticamente con la ecuación 1.4:

$$S(n) = \frac{n}{1 + (n-1)\alpha} \tag{1.4}$$

Donde:

- n es el número de procesadores.
- α es la parte secuencial del código.

El mejor caso sería que α sea igual a cero, esto significaría que no hay partes secuenciales y por lo que todo podría paralelizarse sin embargo, la misma ley establece que esto es básicamente imposible.

A medida que el valor de α se acerca a cero, la aceleración máxima empieza a decaer con mayor rapidez.

Otra forma de escribir la ley de Amdahl es dividir la expresión 1.4 entre n quedando como se muestra en la ecuación 1.5.

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}} \tag{1.5}$$

Si se obtiene el límite cuando n tiende a infinito obtendríamos la ecuación 1.6, lo cual demuestra que aunque tengamos un número infinito de procesadores la aceleración tiene un tope.

La gráfica mostrada en la figura 1.19 muestra cómo es el incremento de la aceleración al aumentar el número de procesadores y con distintas alphas.

$$S(n) = \frac{1}{\alpha} \tag{1.6}$$

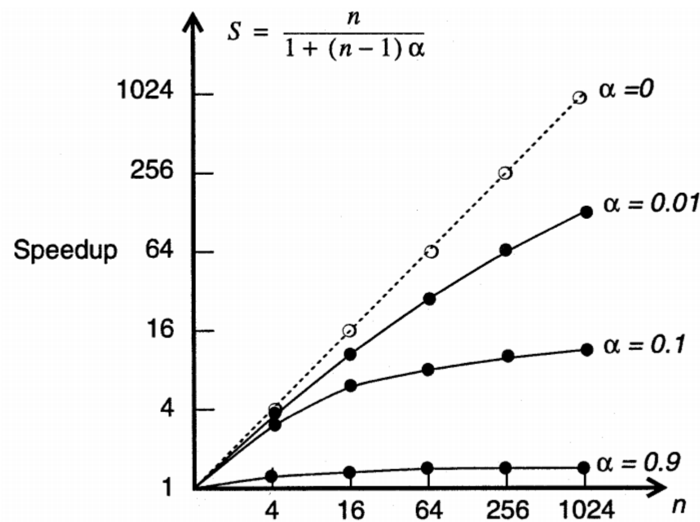


Figura 1.19: Gráfica de la ley de Amdahl, toma su forma dependiendo de α y el número de procesadores.

1.11.6. Ley de Gustafson

Esta ley establece que cualquier problema suficientemente grande puede ser paralelizado. Dicha ley va muy ligada a lo que propone Amdahl, mientras que Amdahl muestra las limitantes de la aceleración al determinar que un programa paralelizado tiene un límite, Gustafson muestra que el poder de computo puede escalarse de forma lineal.

La ley de Gustafson propone que los problemas cambien su tamaño y no solo cambiar el número de procesadores. De esta forma problemas con un tamaño mayor podrán resolverse en el mismo tiempo [19].

$$S(n) = n + (1 - n) * \alpha \tag{1.7}$$

Donde:

- n es el número de procesadores.
- α es la sección no paralelizable
- $S(n)$ es la aceleración

El resultado esperado es lineal como se muestra en la figura 1.20, la pendiente de la recta depende de α en la ecuación 1.7.

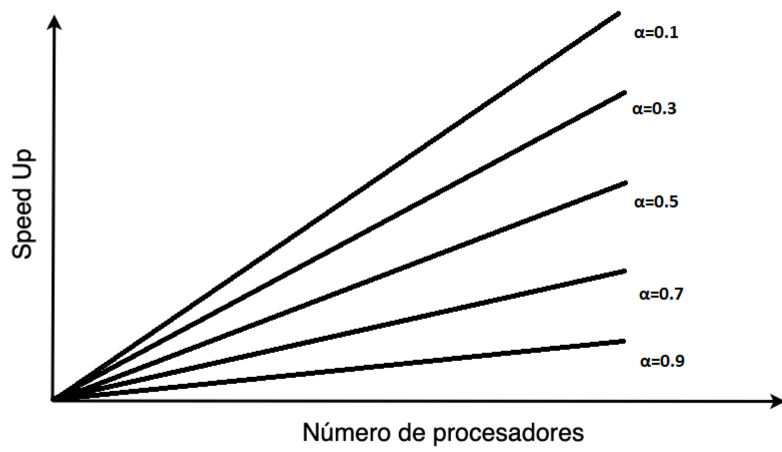


Figura 1.20: Gráfica de la ley de Gustafson, toma su forma dependiendo de α y el número de procesadores de forma lineal.

Capítulo 2

Selección del caso

Sabiendo qué es un autómata celular, sus características y los beneficios que ofrece se puede dar paso a explicar las especificaciones que tiene el AC que fue utilizado en este trabajo. Realmente se podría elegir cualquier AC, dado que el objetivo de este trabajo es poder paralelizar cualquiera y no uno en específico, pero las razones por las cuales se eligió ”El juego de la vida” se mencionan a continuación.

2.1. El juego de la vida

En 1970, el matemático John H. Conway propuso el AC que se volvería el más popular hasta la fecha. La razón principal por la cual se generó dicho AC es porque se quería simular matemáticamente un sistema complejo como lo es el comportamiento de la vida[5].

Por otro lado, se quería implementar un sistema que con cambios pequeños y reglas sencillas diera resultados completamente distintos o que implementaciones simples devolvieran patrones muy complejos.

El juego de la vida es un tablero representado en una malla bidimensional, con un espacio celular cuadrado donde podemos encontrar células que cuentan con dos posibles estados: vivas y muertas. Las células cambian su estado dependiendo de tres sencillas reglas y haciendo uso de la vecindad de Moore. Las reglas son las siguientes:

- Supervivencia: Una célula se mantiene viva si 2 ó 3 de sus vecinas están vivas.
- Muerte: Una célula viva muere si no se cumple la regla de supervivencia, si la suma de las células vecinas vivas son menores a 2 se dice que la célula que se está anali-

zando muere por soledad y si son más de 3 el número de vecinas vivas se dice que la célula analizada muere por sobrepoblación.

- Nacimiento: Una célula muerta cambia su estado a viva si tiene exactamente 3 vecinos vivos.

Una vez entendidas estas reglas, podemos establecer una configuración inicial en la malla, convirtiéndose en las condiciones iniciales del sistema, con ello se da paso a las reglas mencionadas anteriormente para crear las generaciones siguientes.

2.1.1. Patrones

Dentro del juego de la vida es normal que, después de cierto número de iteraciones, en el tablero se presenten figuras ya conocidad por los usuarios del juego de la vida, dichas figuras se les llama patrones[26]. Existen varios tipos de patrones, de los cuales los más comunes y populares son:

- Osciladores: Estos patrones se presentan cuando la figura después de *n* número de iteraciones vuelve a ser la figura inicial. Se han descubierto varios tipos de osciladores con periodos *n* distintos. Los osciladores cuentan con un rotor y un estator. El rotor son las células que cambian de estado en algún momento de la evolución del oscilador. El estator son las células que permanecen vivas durante todas las fases de la evolución del oscilador, es decir las que no cambian en ningún momento.

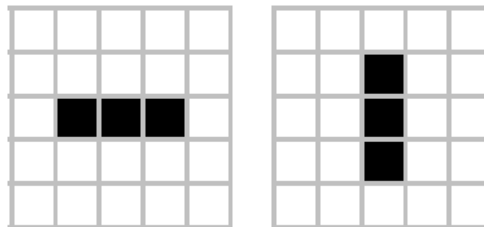


Figura 2.1: Oscilador, patrón más sencillo de obtener en el juego de la vida[26]

- Estáticos: Son figuras que nunca cambian su estado, por más iteraciones que se hagan en el AC. Hay quienes llaman a este tipo de patrones como osciladores de periodo 1, es decir, que tardan un turno en volver a su forma inicial.

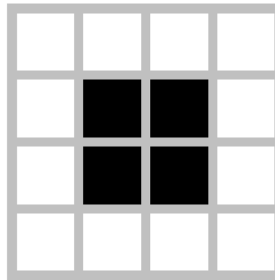


Figura 2.2: Ejemplo de un patrón estático[26]

- Naves espaciales: Este tipo de patrones son las figuras que se mueven dentro de la malla. En otras palabras, son osciladores que al terminar su periodo, se encuentran en una posición distinta en la malla. Incluso se puede medir la velocidad de las naves espaciales que depende del número de celdas que avanza cada iteración.

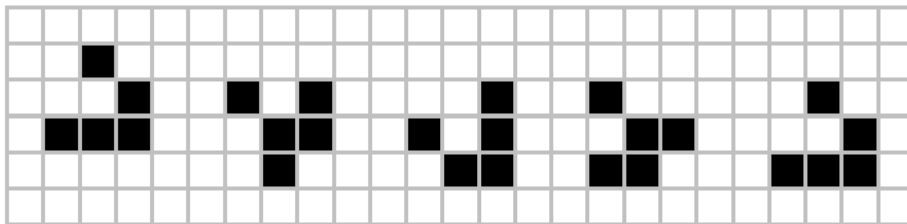


Figura 2.3: Evolución de una nave espacial[26]

- Matusalenes: Son patrones que pueden evolucionar a lo largo de muchas iteraciones para poder estabilizarse. En otras palabras, son condiciones iniciales en el AC que después de un gran número de iteraciones, generan otros patrones como osciladores, naves espaciales y estáticos, llegando así a una estabilidad.

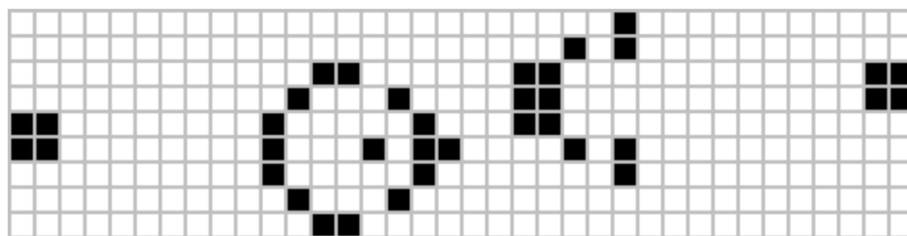


Figura 2.4: Ejemplo de un matusalén estabilizado[26]

2.1.2. Sistemas en el Juego de la vida

Un AC representa uno o varios sistemas, todo depende de la implementación que tenga el AC[3]. El juego de la vida no es la excepción, a continuación se muestran los tipos de sistema que representan a este famoso autómata celular.

- Sistema multiagente: Cumple con este tipo de sistema porque las células son el equivalente a los agentes que toman decisiones por sí solas con base a sus células vecinas.

- Sistema complejo: Cumple con este tipo de sistema por tener muchos elementos(células) que no se pueden analizar de forma separada. Tampoco se puede predecir el resultado hasta haberlo calculado o procesado.

- Sistema estático: Cumple con este tipo de sistema porque solamente depende de los elementos actuales para generar un cambio dentro de todo el sistema y siempre será la misma salida mientras la entrada también lo sea.

- Sistema Caótico: Se cumple este tipo de sistemas porque es imposible hasta ahora predecir el resultado de un AC después de un gran número de iteraciones, la única forma de saber el resultado de un AC es realizando una simulacion de sus n estados. Por otro lado, al hacer una pequeña perturbación en la entrada del sistema el resultado puede ser totalmente distinto al que se obtuvo con la entrada original.

2.1.3. Modelos el Juego de la vida

Un AC requiere de 3 modelos base para ser implementado correctamente: el conceptual, matemático y computacional. La representación de los modelos en el juego de la vida es la siguiente:

- Modelo conceptual: Son las bases que planteo John Conway para generar el juego de la vida. Desde elegir el fenómeno que quería representar hasta delimitar qué quiso incluir en dicho fenómeno.

- Modelo matemático: El modelo matemático en el juego de la vida son las reglas de transición que ocupa el AC, el tablero y las condiciones iniciales para poder evolucionar.

- Modelo computacional: Son todas las implementaciones computacionales que se ocupan para generar el simulador del juego de la vida, desde los diagramas de flujo hasta la implementación en código, incluso la ejecución del mismo.

El modelo computacional no es obligatorio para generar un AC, pero se ha vuelto indispensable dado que cuando se generan los otros dos modelos es necesario realizar pruebas y calcular las generaciones siguientes. Hacer esta tarea a mano sería algo en lo cual se tardaría un largo tiempo y, aunque se tuviera, los resultados podrían ser erróneos, es por ello que la mayoría de los autores agrega dicho modelo como indispensable a la hora de manejar sistemas complejos[14].

Entre mejor sean definidos e implementados los modelos los resultados serán mejores. Se podría generar un AC que evalúe la economía de un país dependiendo de distintos factores pero sus resultados y validez dependen totalmente de cómo se crean estos modelos.

2.2. ¿Por qué el juego de la vida?

El juego de la vida es el AC más popular hasta ahora, con ello la documentación que se tiene es basta, también ayuda a entender cómo es que pocas reglas y tan simples pueden dar resultados demasiado complejos.

Otra de las razones por las que se eligió este AC es por su sencillez de implementación y complejidad que puede tener a la hora de evolucionar. El juego de la vida es fácil de comprender para los usuarios por sus reglas sencillas y de forma análoga es fácil entender como evoluciona de una iteración a otra.

La finalidad de este trabajo es poner las bases para poder implementar cualquier AC de forma paralela utilizando una tarjeta gráfica, por ello no requerimos de un AC demasiado complejo para llevar a cabo dicha tarea. Todo esto vuelve a este autómata celular el elegido para representar de forma secuencial y paralela el caso de estudio.

Capítulo 3

Implementación en secuencial

3.1. Programación secuencial

Las primeras computadoras iniciaron con una arquitectura donde se tenía que realizar una instrucción a la vez. Para acelerar la arquitectura se implementaban diseños más sofisticados, donde el acceso a la memoria fuera más sencillo, las instrucciones a bajo nivel fueran menos complejas o se agregaban más componentes para aumentar su velocidad.

Con estas características es que la computación de inicio utilizando la programación secuencial, esta implementación es más sencilla porque está diseñada de modo que se realiza una instrucción a la vez, pero un proceso se atrasa, todos los procesos siguientes también lo hacen porque deben esperar a que termine la ejecución de los demás procesos.

Aunque los algoritmos diseñados para ser ejecutados de forma secuencial son muy útiles, no siempre son la mejor opción para resolver un problema, todo depende de lo que se quiere lograr.

3.2. Desarrollo

La codificación del algoritmo en secuencial de este trabajo fue hecha en el lenguaje de programación C debido a que la implementación en paralelo se realizó en CUDA, el cual utiliza un lenguaje basado en C, volviéndola una implementación muy similar. Esto ayuda a utilizar funciones similares o iguales para que la comparativa sea más justa.

3.2.1. Equipo empleado

Para el desarrollo del trabajo se utilizaron los siguientes componentes:

- Un procesador *AMD Ryzen 7 2700X* el cual cuenta con una velocidad base de 3.7 GHz.

- Dos memorias RAM DDR4 de 8 GB cada una con una frecuencia de 2666 MHz.

- Una tarjeta gráfica *GTX 1050 Ti* de la marca Nvidia la cual cuenta con 4 GB de RAM, 768 Núcleos CUDA y 6 SM (Stream Multiproccesor).

- Una tarjeta gráfica *GTX 1660 Super* de la marca Nvidia la cual cuenta con 6 GB de RAM, 1408 Núcleos CUDA y 22 SM (Stream Multiproccesor).

El sistema operativo empleado fue Windows 10 Pro de 64 bits para ambos casos (paralelo y secuencial) ya que se realizó en la misma computadora.

Cabe mencionar que para hacer uso del sistema no es necesario tener dos tarjetas gráficas, en este trabajo se utilizaron con el fin de demostrar el objetivo del trabajo, el cual es acelerar la ejecución de los AC.

Diagramas

Con ayuda de los siguientes diagramas de flujo se puede ver de forma general cómo va avanzando el flujo del programa, para obtener una idea de lo que está realizando sin necesidad de leer el código.

Se puede ver en el diagrama de la figura 3.1 que primero se inicializan las variables de forma general, se ejecutan 4 funciones que representan al caso de estudio, la única diferencia que existe entre cada una es que el AC ocupa distintos tipos de datos (int, float, long y double) esto con la finalidad de obtener datos y observar si existen diferencias. Cada función recibe como parámetro dos enteros. El primer parámetro *"N"* que representa la raíz del tamaño total de la matriz, es decir, si $N = 10$ el tamaño total de los datos sería 100 y el segundo parámetro es *"i"* el cual representa el número de iteraciones que realizará el AC, cuantas veces evolucionará.

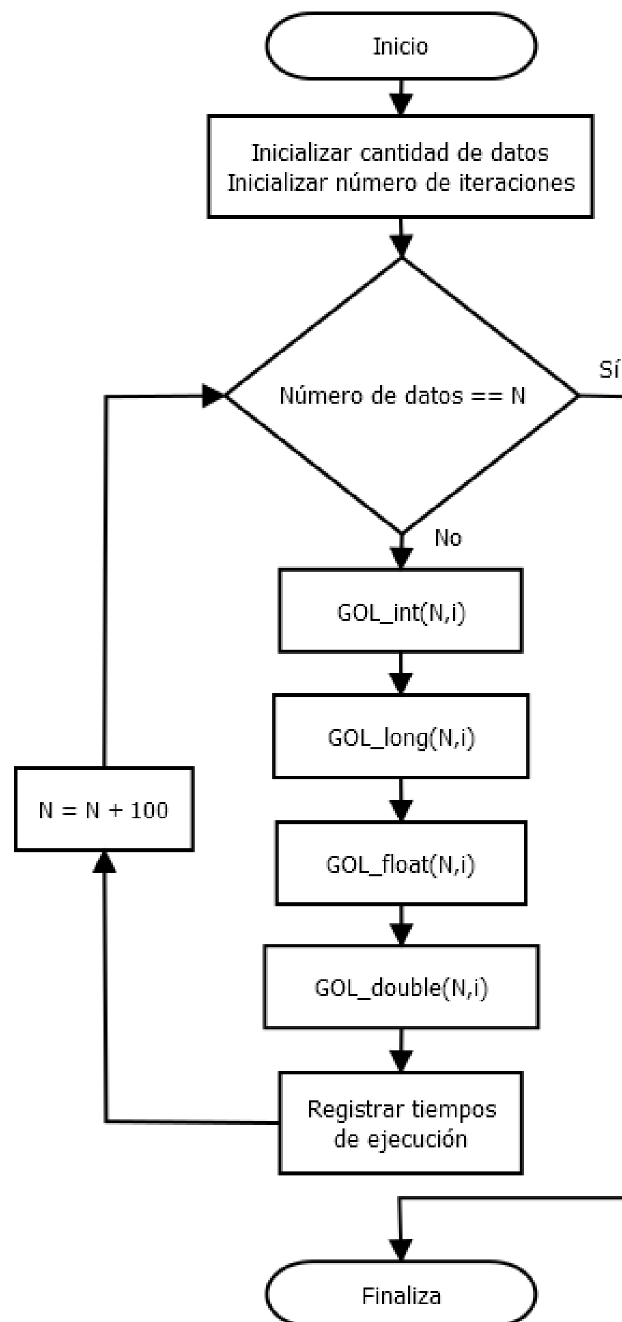


Figura 3.1: Diagrama de flujo general del programa en secuencial

Se guardan los tiempos de inicio y final para poder obtener el tiempo en segundos que tardó en ejecutarse el programa, de esta forma se puede comparar los tiempos con su equivalente en cuanto a tipo de dato y en ejecución en paralelo.

Como ya se ha mencionado las 4 funciones realizan los mismos procesos, la única

diferencia es el tipo de dato que contiene el AC, por lo cual describir una sola de esas funciones basta para comprender el procedimiento que realizan las otras.

El diagrama de la figura 3.2 representa cualquiera de las 4 funciones que se muestran en el diagrama de la figura 3.1 (*GOL_int(N,i), GOL_long(N,i), GOL_float(N,i) y GOL_double(N,i)*).

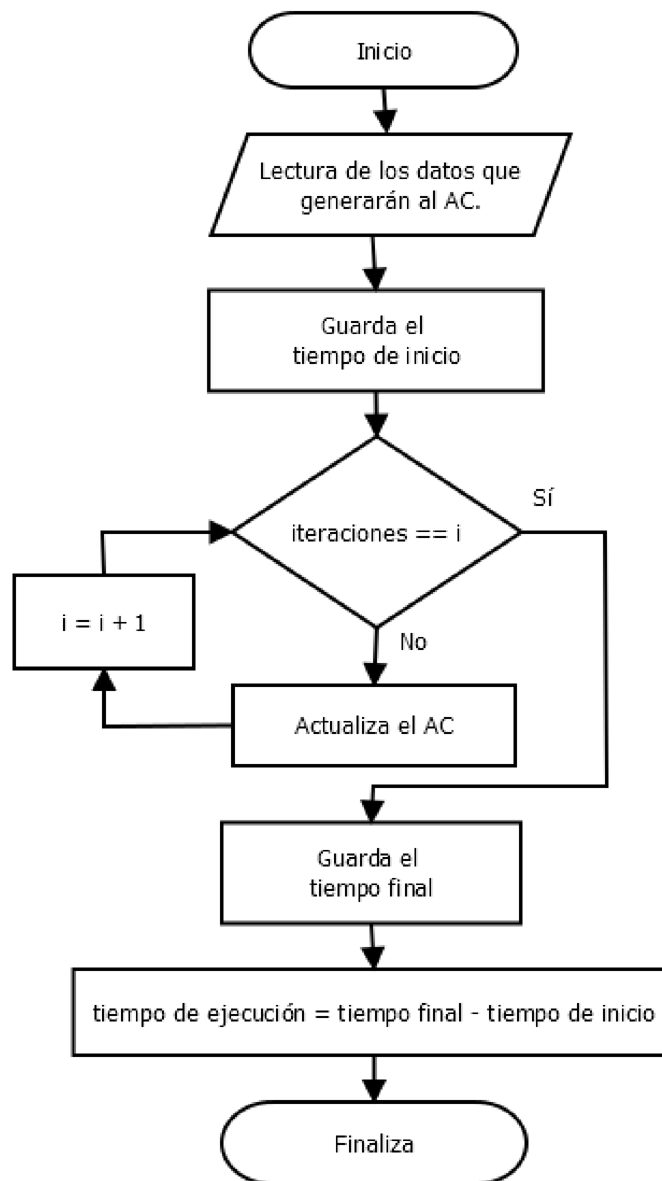


Figura 3.2: Diagrama de flujo del funcionamiento del AC en secuencial

3.2.2. Código en secuencial

A continuación, se presentan los pseudocódigos para la ejecución del AC.

Se debe inicializar las variables que se ocuparán a lo largo del programa, apartando la memoria de forma dinámica para las matrices que se van a utilizar como se muestra en el código 3.1.

```
1 int *malla, *aux;
2 malla = (int *)malloc(N * N * sizeof(int *));
3 aux = (int *)malloc(N * N * sizeof(int *));
```

Código 3.1: Apartado de memoria dinámica en la memoria RAM de la computadora.

Una vez que se tiene apartada la memoria se inicializan las matrices a partir de un archivo, de esta forma mantenemos siempre los mismos datos para no alterar los resultados de la forma secuencial o la forma paralela, realizando una comparación más justa. Este proceso lo logramos con la función *leeMint()* que se encuentra en la línea 3 del código 3.2.

En el ciclo *for* de la línea 8 del código 3.2 se realizan todas las iteraciones que realizará el AC, en caso de ser necesario ver el comportamiento del AC se activa la función *imprimeM()* como se puede apreciar en la figura.

Dentro de las variables *"inicio"* y *"final"* se guardarán los tiempos para poder calcular el tiempo que tarda en ejecutarse el programa.

Finalmente, se obtiene el tiempo en que terminan de ejecutarse los ciclos, y realizando la operación de la línea 16, se obtiene el tiempo total que tardaron en ejecutarse todas las iteraciones del AC.

```
1 double GOLint(int tamanio, int iteraciones)
2 {
3         leeMint(malla, aux,tamanio);
4         //imprimeMint(malla, N);
5 
6         //Inicio de conteo del tiempo
7         inicio = clock();
8         for (int i = 0; i < iteraciones; ++i)
9         {
```

```
10                  actualizaint(malla, aux, N);
11                  //printf("Iteracion %d\n", i + 1);
12                  //imprimeMint(malla, N);
13          }
14          final = clock();
15          double tiempo = ((double)final - inicio)/CLOCKS_PER_SEC;
16  }
```

Código 3.2: Lectura de datos e iteraciones del AC.

En el lenguaje CUDA no se utilizan las matrices bidimensionales ya que no cuentan con un soporte nativo, es por ello que Nvidia ofrece una técnica con la cual es posible acceder a una matriz unidimensional como si fuera bidimensional. En el programa secuencial, pese a hacer uso del lenguaje de programación C (donde sí existen las matrices bidimensionales), se hará uso de matrices unidimensionales para que esta operación no altere los resultados del trabajo.

Si recorremos una matriz unidimensional utilizando la ecuación 3.1 en un doble ciclo *for* podemos tratarla como una matriz bidimensional.

$$clula = i * N + j \tag{3.1}$$

Donde:

- i: es el índice para los renglones
- j: es el índice para las columnas
- N: es el tamaño de la matriz
- célula: es el número que se calculó para saber que célula se está analizando

Este método solo funciona con matrices bidimensionales, donde hay el mismo número de columnas y renglones, las cuales son utilizadas en este trabajo.

Para hacer que el AC vaya evolucionando se creó la función *actualizaMatriz()* con ellas se puede calcular las células vecinas sumando o restando 1 para revisar las células de los lados. También podemos sumar o restar *N* para revisar los vecinos de arriba o abajo, recordando que es una vecindad de Moore se tienen que calcular las células de las esquinas, para ello se hacen combinaciones de los otros tipos de suma. Por ejemplo, si

se quiere revisar la esquina superior izquierda sumamos 1 para avanzar a la derecha y restamos *N* para subir un renglón. Véase figura 3.3.

i*N+j-1-N	i*N+j-N	i*N+j+1-N
i*N+j-1	i*N+j	i*N+j+1
i*N+j-1+N	i*N+j+N	i*N+j+1+N

Figura 3.3: Cálculo de las células vecinas.

En el bloque de código 3.3 se puede ver cómo se realizó la revisión de las células vecinas para determinar el estado siguiente del autómata. Este bloque se encuentra dentro de una función llamada *"actualiza(int *malla, int *mallaAux, int N)"*. Dicha función cuenta con 3 argumentos: el argumento *malla*, contiene una matriz que representa el estado actual del AC; el argumento *mallaAux*, que será donde se guarden los resultados para determinar el estado siguiente del AC, y el argumento *"N"*, contiene el tamaño de la matriz. Dentro de dicha función, se requieren dos variables, la variable contador y *celActual*. La variable contador ayuda a determinar cuantas células vecinas siguen vivas y la variable celActual ayuda para determinar en qué célula nos encontramos y realizar la conversión de una matriz bidimensional a una matriz unidimensional como se muestra en la figura 3.3.

Se necesitan dos ciclos *for* para poder recorrer la matriz. Dentro de los for hay condicionales *if* para determinar las células vecinas (como se muestra en la figura 3.3) y también determinar si la célula que se está analizando se encuentra en la frontera. Recordando que un AC con una frontera cerrada está limitada, lo cual significa que una célula ubicada en la frontera tenga menos de 8 vecinos.

Ya que se calcula cuántas células vecinas se encuentran vivas, se determina si la célula queda viva o muerta para la siguiente ejecución. Para no alterar el AC y que los datos sean correctos se van guardando los resultados en un AC auxiliar.

```
1 void actualizaint(int *malla, int *mallaAux, int N){
```

```
2   int contador = 0; //Cuenta las células vivas vecinas
3   int celActual; //Se almacena el valor que representa
4                  //a la célula actual de forma unidimensional
5
6   for (int i = 0; i < N; ++i){
7       for (int j = 0; j < N; ++j){
8           celActual = i * N + j;
9
10          //Izquierda Arriba
11          if (i > 0 && j > 0 && malla[celActual-N-1] == 1){
12              contador++;}
13
14          //Arriba
15          if (i > 0 && malla[celActual-N] == 1){
16              contador++;}
17
18          //Arriba derecha
19          if (i>0 && j<N - 1 && malla[celActual+1-N] == 1){
20              contador++;}
21              .
22              .
23              .
24
25          if (malla[celActual] == 1){
26          //Actuamos sobre las células en la copia de la matriz
27              if (contador == 2 || contador == 3){
28              //Las células vivas con 2 o 3 células
29              //vivas vecinas, se mantiene vivas.
30                  aux[celActual] = 1;
31              }
32              else{ //Si no se cumple la condición, mueren.
33                  aux[celActual] = 0;
34              }
35          }
36          else{
37              if (contador == 3)
```

```
38                  { //Las células muertas con 3 células vecinas
39                    //vivas, resucitan.
40                  aux[celActual] = 1;
41              }
42          }
43          contador = 0;
44          }
45      }
46
47      for (int i = 0; i < N; i++)
48      {
49          for (int j = 0; j < N; j++)
50          {
51              celActual = i * N + j;
52              //Copiar la matriz origen en destino
53              malla[celActual] = mallaAux[celActual];
54          }
55      }
56  }
```

Código 3.3: Cálculo de las células vecinas.

En los datos recabados no se considera el tiempo que tardaba en proyectar el AC, solo se considera el tiempo que tarda en actualizar sus valores. Se decidió hacer esto porque los tiempos que se tomaban en ejecutar imprimiendo eran más tardados. Dado que se realizaron 1000 iteraciones por AC, considerar los tiempos de impresión aumentaría considerablemente los resultados.

Sin embargo, se implementó la función *imprimeMatriz()* para rectificar que ambas implementaciones fueron creadas y ejecutadas de forma correcta, comparando los resultados impresos en pantalla como se muestra en la figura 3.4. Los asteriscos (*) representan células vivas, mientras que los guiones (-) representan células muertas.

El código se encuentra disponible para ser consultado[1].

[1] Para consultar el código visite el siguiente enlace https://github.com/JoseAngel13/JuegoDeLaVidaCUDA

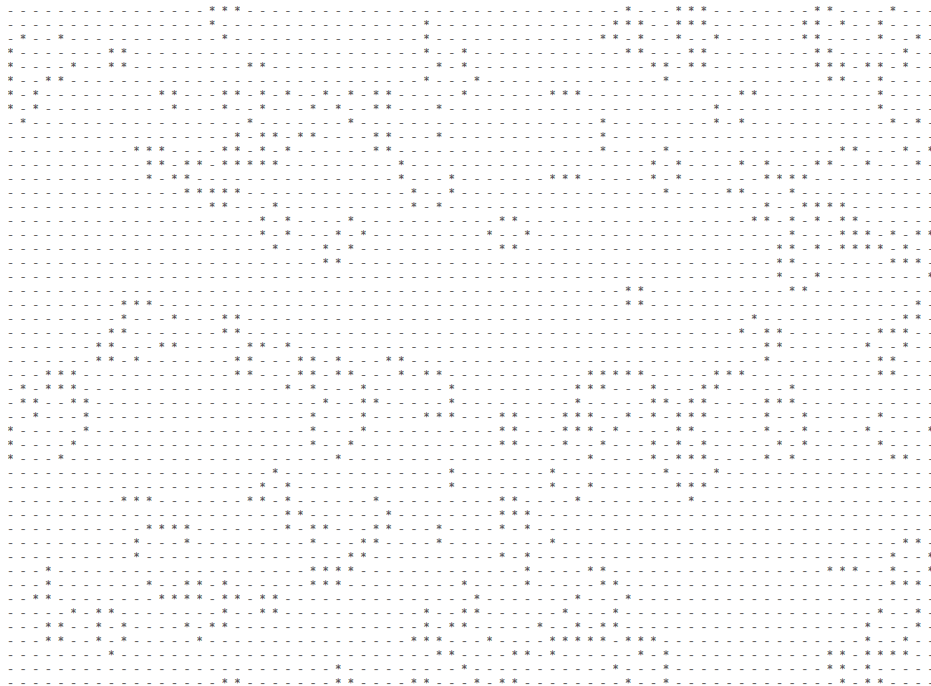


Figura 3.4: Ejecución de un AC de tamaño 100x100

Capítulo 4

Implementación en paralelo

A diferencia de la programación en secuencial, la programación en paralelo aprovecha las características de las unidades de procesamiento como son la CPU y GPU para realizar procesos al mismo tiempo, esto con la ayuda de los núcleos y los hilos que poseen. Gracias a este paradigma de programación podemos agilizar las tareas de una computadora.[33]

Para realizar la paralelización del autómata celular se eligió el lenguaje de programación CUDA. El lenguaje se eligió por las siguientes características: la gran documentación con la que cuenta, el soporte que tiene dicha herramienta y su facilidad de uso.

4.1. Desarrollo

La idea principal es muy similar a la implementación en secuencial. En términos generales, la diferencia radica en enviar los datos del host al device para que la tarjeta gráfica sea la encargada de realizar los procesos paralelizables y no la CPU como se acostumbra en la mayoría de los algoritmos.

El equipo empleado es el mismo que se utilizó para la implementación en secuencial, de esta forma mantenemos las mismas características para que los tiempos calculados puedan ser comparados.

CUDA genera cientos o miles de hilos dependiendo de dos factores. El primer factor es el número de SM (Stream Multiprocessor) y el segundo es el maximo de hilos por SM que dependen del modelo de la tarjeta. Para mayor información consulte el primer capitulo apartado *"Multiprocesadores SM"*

El número de hilos que se pueden ejecutar de forma paralela en una tarjeta gráfica está

dado por la ecuación 4.1.

$$\#Hilos = \#Max_SM * \#SM \tag{4.1}$$

Como ya se mencionó, en este proyecto se utilizaron dos tarjetas gráficas. Una de ellas es, la tarjeta 1050 Ti, la cual será usada como ejemplo para saber como se obtienen el máximo de hilos posibles. Esta tarjeta posee 6 SM y el máximo de hilos SM son 2048. Sustituidos en la ecuación anterior nos da:

$$\#Hilos = 2048 * 6 \tag{4.2}$$

$$12,288\, Hilos$$

Por otro lado, la tarjeta 1660 Súper posee 22 SM y el máximo de hilos SM son 1024. Sustituidos en la ecuación anterior nos da:

$$\#Hilos = 1024 * 22 \tag{4.3}$$

$$22,528\, Hilos$$

Estos hilos son los que podemos ejecutar de forma paralela en cada tarjeta gráfica.

Como la paralelización realizada en este trabajo está enfocada en que cada hilo se encargue únicamente de una célula, existe un limite en cuanto al tamaño del AC. Aunque son una gran cantidad de hilos que se ejecutan en paralelo solo nos da la posibilidad de ejecutar matrices que tengan 12,288 o menos células para la tarjeta 1050 Ti. Mientras que la 1660 Super puede ejecutar matrices con tamaño máximo de 22,528 células.

Como en este proyecto se están ejecutando matrices cuadradas, donde los renglones son del mismo tamaño que las columnas, se puede deducir que la matriz más grande que podría ejecutar la tarjeta 1050 Ti es de 110x110, dado que, si obtenemos la raíz cuadrada del número máximo de hilos es $\sqrt{12288} = 110{,}85$. Siguiendo el mismo procedimiento, la tarjeta 1660 Súper puede ejecutar matrices de 150x150.

Esto generaría un problema, debido a que si queremos ingresar una matriz más grande se deberían generar más hilos. Cuando se genera un programa que demanda más hilos de los que se pueden ejecutar, CUDA utiliza todos los hilos que tiene disponibles y cuando los hilos terminan su ejecución revisa si hay mas tareas por realizar y de ser así genera los

hilos faltantes[28].

Aunque ya realiza por sí mismo este proceso, es mejor controlar el flujo de los hilos porque cada que CUDA termina la ejecución de estos, revisa si hay tareas con mayor prioridad a la que se está realizando y las procesa, lo cual podría volver lenta la ejecución de AC.

Para controlar el flujo de los datos se tomó la decisión de generar siempre el máximo de hilos posibles, de esta forma, controlamos si un hilo debe ejecutarse más de una vez para no perder tiempo que se vuelvan a inicializar los hilos y revisar si hay otras tareas. Se puede utilizar una variable de control para dar el número máximo de hilos. De esto último surgen 3 posibles casos:

- **El número de células es menor al número máximo de hilos:** Cuando esto sucede, se utilizan únicamente los hilos que se necesitan y los sobrantes terminan su ejecución.

- **El número de células es igual al número máximo de hilos:** Este caso es el ideal, pero el menos probable de ocurrir. Se utiliza un hilo por cada célula.

- **El número de células es mayor al número máximo de hilos:** Se ejecutan todos los hilos, cada hilo se encarga de una célula única, cuando su ejecución está por terminar, se asigna una nueva célula a los hilos necesarios y los que sobren terminan su ejecución.

4.1.1. Diagramas

Se puede ver en el Capítulo 3 que se muestran diagramas para analizar la implementación de los algoritmos, el diagrama general es el mismo para ambos casos, donde existen cambios es en el funcionamiento particular de los algoritmos. Los cambios se pueden ver en los diagramas, que a grandes rasgos seria que en el algoritmo en paralelo envía los datos a la memoria de la tarjeta gráfica para ser procesados posteriormente.

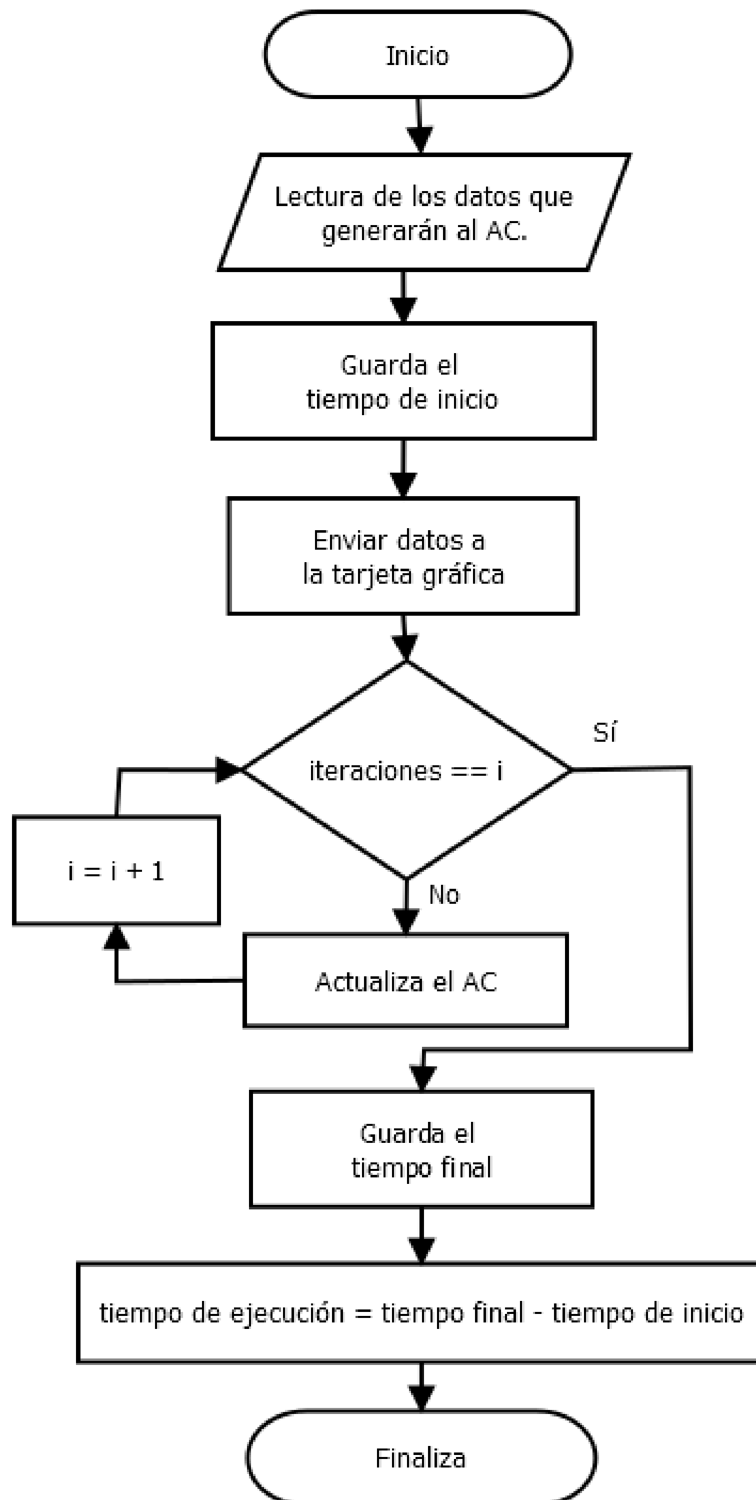


Figura 4.1: Diagrama de flujo del funcionamiento del AC en paralelo.

4.1.2. Código

A continuación se presentan los segmentos de pseudocódigo del programa en paralelo. Aunque la implementación base es muy similar, el algoritmo tiene cambios importantes.

Lo principal es declarar las variables, apartando el tamaño de la memoria que necesitamos para generar los arreglos. Al igual que el algoritmo en secuencial, se generan dos arreglos, uno funciona como auxiliar para ir guardando las actualizaciones del AC como se muestra en el bloque de código 4.1.

```
int *tablero, *tablero_aux;
int *d_tablero, *d_tablero_aux;
size_t size = N*N*sizeof(int);

//Asignación de memoria del lado del host
tablero = (int*)malloc(size);
tablero_aux = (int*)malloc(size);

//Asignación de memoria del lado de device
cudaMalloc(&d_tablero, size);
cudaMalloc(&d_tablero_aux, size);
```

Código 4.1: Lectura de los datos e inicialización de variables.

En la línea 2 de código 4.1 se declaran dos matrices más, esto se debe a que se necesita generar primero, la memoria necesaria en la memoria RAM de la computadora, y luego, generar la memoria en la RAM de la tarjeta gráfica. La función *malloc()* aparta la memoria, pero en la computadora, mientras que la función *cudaMalloc()* lo hace del lado de la tarjeta gráfica.

Los datos hasta ese momento se encuentran en la memoria de la computadora. Para realizar la transferencia de datos a la tarjeta se utiliza la función *cudaMemcpy()* como se muestra en el bloque 4.2, que requiere 4 argumentos. El primero es la dirección de origen, el segundo parámetro es la dirección de destino, el tercer parámetro es el tamaño en bytes de los datos que serán transferidos y, el último parámetro, es el que indica donde es el origen de los datos y cuál es el destino, en este caso es de la computadora a la tarjeta gráfica.

```
cudaMemcpy(d_tablero, tablero, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_tablero_aux, tablero_aux, size,
    cudaMemcpyHostToDevice);
```

Código 4.2: Transferencia de datos entre host y device.

Es necesario calcular el número máximo de hilos que la tarjeta gráfica puede ejecutar en paralelo. Es conveniente convertir este número en un número entero dado que no hay matrices que tengan tamaño en decimal. Para todo esto se hace uso del bloque de pseudocódigo 4.3.

```
int d_MAX = (int)sqrt(MAX_threads);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid((d_MAX + dimBlock.x-1)/dimBlock.x, (d_MAX+dimBlock.y
    -1)/dimBlock.y);
```

Código 4.3: Cálculo de la dimensión de los bloques y la malla.

Lo siguiente es determinar el tamaño de los bloques y malla, mediante la operación de la línea 3 del bloque de código 4.3. Se podría dejar de forma constante, pero esto solo serviría con un solo modelo de tarjeta gráfica, por ello es que se realiza de forma dinámica.

La función dimGrid() calcula el tamaño de la malla. Sus parametros son los siguientes:

$$((d_MAX + dimBlock.x - 1)/dimBlock.x, (d_MAX + dimBlock.y - 1)/dimBlock.y) \tag{4.4}$$

Donde:

- d_Max Es la Raíz cuadrada del máximo de hilos que se pueden ejecutar de forma paralela como se muestra en la ecuacion 4.1.
- dimBlock.x Contiene el número de hilos que se crearán en el eje X por bloque.
- dimBlock.y Contiene el número de hilos que se crearán en el eje Y por bloque.

Al igual que en su forma secuencial, el ciclo for de la línea 3 del código 4.4 ayuda para realizar todas las iteraciones que determine el usuario. La diferencia dentro del ciclo es que se utilizan kernels para poder hacer uso del paralelismo en la tarjeta gráfica.

Se recomienda hacer uso de la función *cudaThreadSynchronize()* cada que se llama un kernel, la función mencionada se utiliza para que el programa no siga su curso hasta que todos los hilos generados por los kernels terminen su ejecución. Si no se utiliza dicha función podría ejecutarse otro kernel al mismo tiempo que se esté ejecutando otro, por lo cual se podrían generar problemas como deadlock o condición de carrera.

El deadlock puede ocurrir cuando dos procesos se bloquean entre sí porque ambos toman un recurso que el otro necesita y no podrán concluir su tiempo de ejecución hasta que alguno de los dos libere el recurso que está utilizando. Por otro lado, la condición de carrera se da cuando dos procesos dependen de un mismo recurso y ninguno de los dos procesos bloquea el acceso al recurso, haciendo que ambos accedan y modifiquen la información sin saber que otro proceso desea modificarlo.

Estos son algunos de los problemas que podría generar un kernel si no son utilizados adecuadamente.

Para ver el funcionamiento del AC, se pueden activar las líneas comentadas dentro del ciclo for. Podríamos convertir la función *imprimeM()* en un kernel, esto evitaría la transferencia de datos entre la computadora y la tarjeta gráfica, pero dado que la función *printf()* no esta optimizada para trabajar en la tarjeta gráfica es mejor transferir completamente los datos.

Finalmente, se calcula el tiempo que tardó en ejecutarse el AC y se libera el espacio ocupado de la memoria RAM y de la tarjeta gráfica.

```
1  inicio = clock();//Toma del tiempo inicial
2  ...
3  for (int i = 0; i < iteraciones; ++i){
4          //printf("Iteracion %d\n",i+1);
5          //Función que actualiza el tablero
6          actualiza_Pdouble<<<dimGrid,dimBlock>>>(d_tablero,
               d_tablero_aux, N, dim_MAX);
7          cudaThreadSynchronize();//Espera a que todos los hilos
               terminen su ejecución
8          //Intercambio de los datos entre la malla auxiliar y la
               principal
9          copiaMatriz_Pdouble<<<dimGrid,dimBlock>>>(d_tablero_aux,
               d_tablero, N, dim_MAX);
```

```
10          cudaThreadSynchronize();
11          //cudaMemcpy(tablero,d_tablero,size,
                cudaMemcpyDeviceToHost);
12          //imprimeM_Pdouble(tablero,N);
13          //cudaThreadSynchronize();
14          //cudaMemcpy(d_tablero,tablero,size,
                cudaMemcpyHostToDevice);
15  }
16  final = clock();//Toma el tiempo final
17  double tiempo = ((double)final - inicio) / CLOCKS_PER_SEC;
```

Código 4.4: Lectura de los datos e inicialización de variables.

El código 4.4 muestra los cambios que se realizaron en el código principal, pero a la hora de paralelizar el AC el algoritmo cambia, ya que se debe asignar a cada hilo lo que debe ejecutar.

En el código 4.5 se muestra cómo trabaja cada hilo, cómo son configurados para que trabajen sobre cada célula. Lo principal es inicializar las variables y los hilos. En las líneas 4 y 5 del código 4.5 obtenemos el identificador de cada hilo, que con las operaciones que se realizan en las mismas líneas se convierten en un número de renglón y columna único para cada célula.

La condición *if* de la línea 8 es un filtro para dejar pasar únicamente a los hilos que puede crear la tarjeta gráfica de forma paralela, si por alguna razón creara más hilos este filtro termina con su ejecución.

En la línea 9 del código 4.5 se determina cuántas veces debe ser ejecutado cada hilo. Por ejemplo, si se crean 9 (3x3) hilos y el AC tiene 36 (6x6) células habrá hilos que deban ejecutarse 2 veces. La operación realizaría $\frac{6}{3}+1=3$ quiere decir que el AC es del triple de tamaño del que podemos ejecutar, por lo cual habrá hilos que ejecuten más de una célula.

```
1   __global__ void actualiza_Pdouble(double *malla, double *aux,
        int N, int dim_MAX){
2   int contador;
3   int celActual;
4   int id_i= blockDim.x * blockIdx.x + threadIdx.x;//fila
5   int id_j= blockDim.y * blockIdx.y + threadIdx.y;//columna
```

```
6  int i = id_i, j = id_j;
7
8  if (i < dim_MAX && j < dim_MAX) {
9      int ii = (int)((N)/(dim_MAX))+1;
10
11     for(int contador_i = 0 ; contador_i < ii; contador_i ++){
12         for(int contador_j = 0 ; contador_j < ii; contador_j ++){
13
14                 contador=0;
15                 i = id_i + contador_i * dim_MAX;
16                 j = id_j + contador_j * dim_MAX;
17                 celActual = i*N+j;
18
19                 if(celActual < N*N && i < N && j < N){
20                         //Izquierda Arriba
21                         if(i>0 && j>0 && malla[celActual-N-1]==1)
                               {
22                                 contador++;
23                         }
24                         //Arriba
25                         if(i>0 && malla[celActual-N]==1){
26                                 contador++;
27                         }
28                         //Arriba derecha
29                         if(i>0 && j<N-1 && malla[celActual+1-N
                               ]==1){
30                                 contador++;
31                         }
32
33                         ...
34
35                 //Se revisa si la célula está viva o muerta
36                         if(malla[celActual]==1){
37                             //Las células vivas con 2 o 3 células
                                   vecinas vivas, mantiene su estado
                                   .
```

```
38                                  if(contador==2 || contador==3){
39                                          aux[celActual]=1;
40                                  }
41                                  //Si no se cumple la condición
                                       mueren.
42                                  else{
43                                          aux[celActual]=0;
44                                          }
45                          }
46                          else{
47                              //Las células muertas con 3 células
                                   vecinas vivas, resucitan.
48                                  if(contador==3){
49                                          aux[celActual]=1;
50                                  }
51                          }
52                  }
53          }
54      }
55  }
56  }
```

Código 4.5: Lectura de los datos e inicialización de variables.

En el doble for de las líneas 11 y 12 del bloque de código 4.5 es donde se reasignan las células a calcular, para comprender mejor esto se tomará un ejemplo.

4.1.3. Explicación del algoritmo en paralelo

Para dejar más claro el funcionamiento del algoritmo se dará un ejemplo con datos pequeños, pero claros y precisos.

Supóngase que la tarjeta gráfica utilizada solo puede generar 9 hilos en paralelo, eso quiere decir que el tamaño máximo de matrices que se pueden ejecutar en paralelo son de tamaño NxN con $N = 3$. Si se ejecuta un AC de tamaño menor no habría problema, pero si el AC es de mayor tamaño, por ejemplo de *N=5* la matriz tendría un total de 25 células y solo hay 9 hilos.

Como el algoritmo analiza una célula por hilo, faltarían 16 hilos para poder analizar

completamente el AC. El algoritmo tiene contemplado este tipo de casos, cuando esto sucede se toman todos los hilos creados, en este caso 9 y se analizan las 9 primeras células, al terminar el análisis cada hilo analiza una nueva célula. Este proceso se repite hasta que todas las células son analizadas.

Los hilos no toman células al azar, llevan una secuencia para tener un orden y evitar que una célula sea revisada dos veces.

La figura 4.2 representa la organización que tienen los hilos para analizar el AC. Se tiene una malla de *N=5*, por lo cual hay 25 células por revisar. Si solo se pueden generar 9 hilos se divide la malla del AC en 4 partes que son analizadas en orden. La primera parte es la de color verde, la segunda parte es la azul, la tercera naranja y por último la roja. El AC es revisado en el mismo orden, primero verde y al último rojo, las células de color blanco son los hilos que salen del límite del AC, cuando esto ocurre los hilos no realizan acciones a esto se le conoce como un estado de ociosidad que ocurre cuando se activa un hilo, pero no se mantiene esperando a ser llamado y ejecutar un proceso.

	0	1	2	3	4	
0	H(0,0)	H(0,1)	H(0,2)	H(0,0+3)	H(0,1+3)	H(0,2+3)
1	H(1,0)	H(1,1)	H(1,2)	H(1,0+3)	H(1,1+3)	H(1,2+3)
2	H(2,0)	H(2,1)	H(2,2)	H(2,0+3)	H(2,1+3)	H(2,2+3)
3	H(0+3,0)	H(0+3,1)	H(0+3,2)	H(0+3,0+3)	H(0+3,1+3)	H(0+3,2+3)
4	H(1+3,0)	H(1+3,1)	H(1+3,2)	H(1+3,0+3)	H(1+3,1+3)	H(1+3,2+3)
	H(2+3,0)	H(2+3,1)	H(2+3,2)	H(2+3,0+3)	H(2+3,1+3)	H(2+3,2+3)

Figura 4.2: Ejemplo del funcionamiento del algoritmo.

El algoritmo en general funciona de forma muy parecida a su similar en secuencial, las diferencias principales son que en el secuencial ejecuta las NxN células en el procesador usando solo un núcleo, mientras que en su equivalente en paralelo dichas células se repar-

ten entre el número de hilos creados. El otro cambio es que el algoritmo en secuencial no realiza una repartición de tareas a diferencia del paralelo, simplemente termina de analizar una célula y toma la siguiente hasta haber terminado con todo el AC.

El código se encuentra disponible para ser consultado[1].

[1]Para consultar el código visite el siguiente enlace https://github.com/JoseAngel13/JuegoDeLaVidaCUDA

Capítulo 5

Métricas de rendimiento

Una vez que se comprende cómo funcionan ambos algoritmos, en secuencial y en paralelo, se procede a evaluar su rendimiento para saber cuál de los dos se desempeña mejor. Para evaluar ambos algoritmos se ejecutaron tomando el tiempo de inicio y tiempo final para obtener distintas métricas.

Las métricas que se usarán para evaluar el rendimiento de los algoritmos son el tiempo de ejecución, la aceleración y la ley de Amdahl.

Dentro del programa existe la función que imprime los datos de la matriz, pero se desactivó dicha función para generar los resultados de los tiempos de ejecución, ya que la ejecución se volvía más lenta en ambos casos.

La función para imprimir el AC también ayudó a verificar que las cosas se hacían correctamente en ambos algoritmos del AC. Imprimiendo la evolución del AC, se puede analizar si se ejecuta de forma correcta.

En las métricas se analiza el algoritmo en secuencial contra su similar en paralelo variando el número de procesadores. Normalmente el número de procesadores no se puede variar en la práctica cotidiana, lo que cambia en realidad es el número de datos a procesar. Por ejemplo, las supercomputadoras[16] tienen un número fijo de componentes, de vez en cuando se les agregan o quitan componentes, pero lo que más a menudo cambia es el número de datos. Por todo esto es que en las métricas utilizadas en este trabajo se cambia el número de datos y no el número de procesadores, dejando éste de forma constante para cada tarjeta gráfica.

5.1. Primer conjunto de datos

El primer conjunto de datos comienza con un AC de 10 x 10, el siguiente AC de 20 x 20; el siguiente, de 30 x 30 y así sucesivamente hasta llegar a 100 x 100. Todos los AC ejecutaron 1000 iteraciones y son de dimensión cuadrada (NXN).

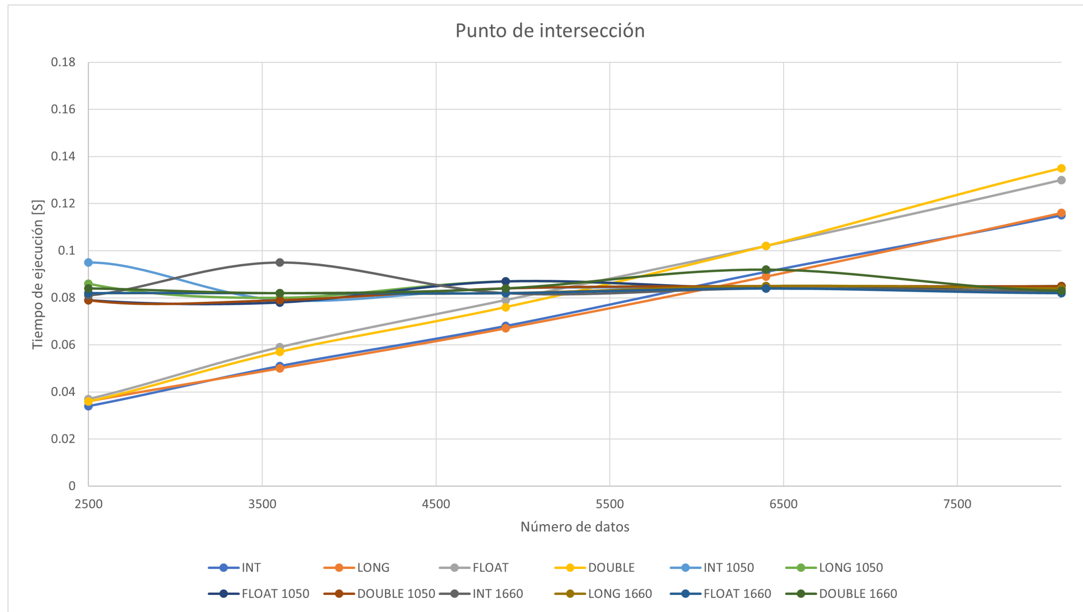


Figura 5.1: Gráfica reducida del tiempo de ejecución del primer conjunto de datos.[1]

N	NxN	INT	LONG	FLOAT	DOUBLE	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
50	2500	0.034	0.036	0.037	0.036	0.095	0.086	0.079	0.079	0.081	0.082	0.082	0.084
60	3600	0.051	0.05	0.059	0.057	0.079	0.08	0.078	0.079	0.095	0.082	0.082	0.082
70	4900	0.068	0.067	0.079	0.076	0.084	0.087	0.087	0.084	0.082	0.082	0.082	0.084
80	6400	0.091	0.089	0.102	0.102	0.084	0.084	0.084	0.085	0.084	0.085	0.084	0.092
90	8100	0.115	0.116	0.13	0.135	0.084	0.085	0.085	0.085	0.083	0.084	0.082	0.083

Tabla 5.1: Tabla reducida del primer conjunto de datos.[2]

La finalidad de este primer conjunto de datos fué encontrar el punto de intersección entre las gráficas, es decir , el punto donde comienza a ser más útil el uso de la paralelización.

Como la CPU y la GPU no son nodos dedicados[3]pueden presentarse picos dentro de las gráficas que representan que durante el tiempo de ejecución en dicho punto la CPU o la GPU (según el caso) tuvieron que realizar algún proceso alterno a la ejecución del AC.

[1]Esta gráfica es una ampliación de la gráfica 6.2 para denotar mejor el punto de corte.
[2]Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.1
[3]No se enfocan en una sola tarea, pueden llegar tareas con mayor prioridad y hacer uso de los núcleos.

Dentro de este primer análisis lo primordial es encontrar el punto donde la gráfica del tiempo de ejecución se intercepta con las otras funciones, porque este punto es donde se puede decir a partir de qué cantidad de datos es mejor la implementación en secuencial y en cuál la implementación en paralelo.

Como se mencionó en capítulos anteriores, se utilizaron dos tarjetas gráficas distintas, la tarjeta gráfica RTX 1050 Ti y RTX 1660 Super, utilizando el máximo de sus hilos y núcleos.

Recordando que la tarjeta gráfica GTX 1050 Ti cuenta con 4 GB de RAM, 768 Núcleos CUDA y 6 SM (Stream Multiproccesor), mientras que la tarjeta gráfica GTX 1660 Super cuenta con 6 GB de RAM, 1408 Núcleos CUDA y 22 SM (Stream Multiproccesor).

Los datos a analizar están en la tabla 5.1. La tabla con los datos completos es la tabla 6.1 la cual se encuentra en el anexo.

En la gráfica *Punto de intersección* de la figura 5.1 se puede notar que el punto de corte es aproximadamente entre 5000 y 6000 datos, es decir, está entre *N = 60* y *N = 70*.

De esta forma se puede decir que apartir de *N = 70* es recomendable utilizar el AC en su forma paralela para ambas tarjetas gráficas.

5.1.1. Aceleración

La aceleración ayuda a determinar cuántas veces mejoró o empeoró el algoritmo en paralelo respecto al algoritmo en secuencial. En este caso se consideró dividir el tiempo secuencial entre el tiempo en paralelo que le corresponde según la tarjeta gráfica y el tipo de dato que emplean. $S(n) = \frac{T(1)}{T(n)}$. Esto quiere decir que el tiempo en secuencial de 10x10 datos de tipo entero se dividió entre el tiempo en paralelo de 10x10 datos de tipo entero, haciendo esto para ambas tarjetas y todos los tipos de datos, dando como resultado la tabla 6.2.

En la tabla 5.2 y en la gráfica 5.2 se puede notar cuántas veces mejora el algoritmo en paralelo para ambas gráficas y para cada tipo de dato. Recordando que si el resultado es cercano a 1 la mejora es casi nula, mientras que si es menor no hay mejora, al contrario, es más tardado el algoritmo paralelo.

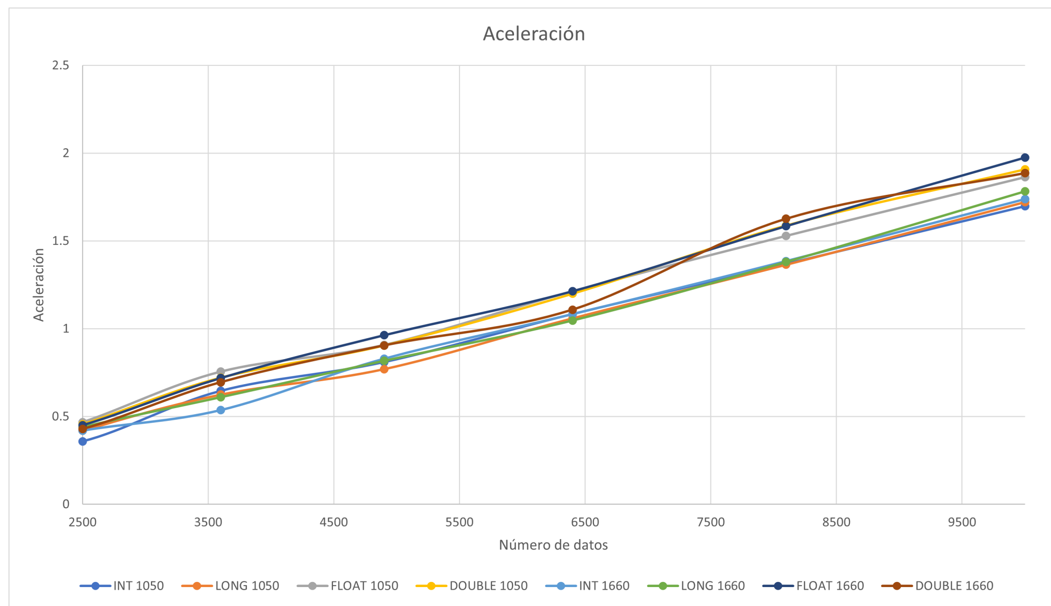


Figura 5.2: Gráfica reducida de la aceleración del primer conjunto de datos.[4]

N	NxN	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
50	2500	0.35789474	0.41860465	0.46835443	0.455696203	0.41975309	0.43902439	0.45121951	0.428571429
60	3600	0.64556962	0.625	0.75641026	0.721518987	0.53684211	0.6097561	0.7195122	0.695121951
70	4900	0.80952381	0.77011494	0.90804598	0.904761905	0.82926829	0.81707317	0.96341463	0.904761905
80	6400	1.08333333	1.05952381	1.21428571	1.2	1.08333333	1.04705882	1.21428571	1.108695652
90	8100	1.36904762	1.36470588	1.52941176	1.588235294	1.38554217	1.38095238	1.58536585	1.626506024
100	10000	1.69767442	1.72093023	1.86363636	1.908045977	1.73809524	1.78313253	1.97590361	1.886363636

Tabla 5.2: Tabla reducida de la aceleración para el primer conjunto de datos.[5]

A partir de 70x70 datos se comienza a acercar a 1, después de 80x80 ya es mayor que uno, esto nos dice que mejora su rendimiento a partir de este momento. La aceleración alcanzada con este conjunto de datos no es muy alta, aún así ya es visible una mejora. Hasta este punto no hay cambios drásticos entre los datos de una tarjeta y otra, de hecho son muy similares. La máxima aceleración alcanzada en este punto es con las matrices de 100x100 datos alcanzando 1.90 veces de mejora en la tarjeta 1050 Ti y 1.97 veces para la 1660 Super, es decir, casi el doble de velocidad.

[4]Esta gráfica es una ampliación de la gráfica 6.3

[5]Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.2

5.2. Segundo conjunto de datos

Para este segundo conjunto de datos se obtuvieron las gráficas más relevantes de este trabajo. Mientras que el conjunto anterior ayudó a entender en qué momento es mejor utilizar el algoritmo en paralelo, aquí se quiere encontrar los puntos máximos de las gráficas y las tendencias de las mismas.

Los datos que se utilizaron fueron con matrices que cambiaban su tamaño en un rango de N=250 hasta N=14,000. Con intervalos de 250 renglones y columnas entre cada muestra como se puede observar en la tabla 5.3.

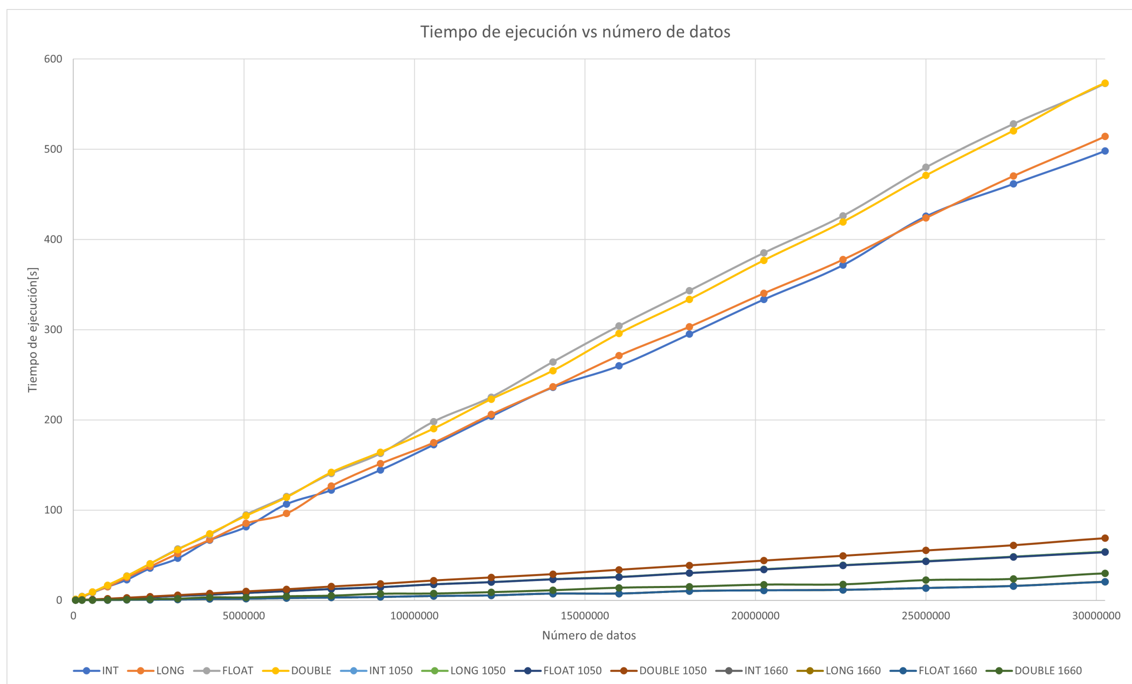


Figura 5.3: Gráfica del tiempo de ejecución del segundo conjunto de datos.[6]

N	NxN	INT	LONG	FLOAT	DOUBLE	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
250	62500	0.926	0.928	1.057	1.024	0.152	0.154	0.153	0.165	0.084	0.087	0.085	0.095
1250	1562500	22.867	25.536	26.91	26.767	2.4	2.416	2.393	2.816	0.465	0.465	0.466	0.853
2250	5062500	81.386	85.385	94.926	93.661	8.183	8.182	8.15	9.849	1.68	1.687	1.67	3.027
3250	10562500	172.527	174.885	198.086	190.298	17.739	17.751	17.649	21.969	4.718	4.72	4.724	7.472

Tabla 5.3: Tabla reducida del tiempo de ejecución para el segundo conjunto de datos.[7]

[6] Esta gráfica es una ampliación de la gráfica 6.4
[7] Esta tabla es solo un fragmento de los datos mostrados en las tablas 6.3, 6.4 y 6.5

El orden máximo de las matrices que se pudieron crear fue de 14,000x14,000 debido al desbordamiento de la memoria del modelo de tarjeta gráfica utilizada (GTX 1050 TI de 4 GB) pasando esta frontera las mediciones dieron resulados erroneos.

En la gráfica 6.4 se puede notar el cambio que hay en las pendientes de las funciones, con ambas tarjetas gráficas se consiguieron tiempos menores aunque el tiempo es aún menor en los casos de la tarjeta gráfica 1660 Super. Esto se debe a varios factores, el tiempo de comunicación entre la tarjeta y su memoria RAM, la velocidad de cada uno de sus núcleos y, sobre todo, al número total de hilos que puede ejecutar en paralelo.

Los tiempos de ejecución en el caso del algoritmo en secuencial son muy similares entre los tipos de dato INT y LONG, pero en el tipo de dato FLOAT y DOUBLE son mayores debido a que el procesador debe realizar más operaciones puesto que estamos tratando con datos que ocupan mayor memoria.

En el mismo caso, se nota que aunque FLOAT ocupa menor espacio de memoria que el tipo de dato DOUBLE como se muestra en la tabla 5.4, los tiempos son menores en el tipo de dato DOUBLE. Este comportamiento se debe a que el procesador es de doble punto flotante, lo cual quiere decir que cada que opera el procesador sobre un elemente de punto decimal siempre ocupa 8 bytes. Cuando llega un número que ocupa menos espacio, por ejemplo 4 bytes el procesador debe llenar los otros 4 bytes con 0 para poder procesar y no devolver datos erróneos. Por ello, tarda más en ejecutar el tipo de dato FLOAT que el DOUBLE.

TIPO DE DATOS	SE ESCRIBE	MEMORIA REQUERIDA	RANGO ORIENTATIVO
Entero	int	2 bytes	- 32768 a 32767
Entero largo	long	4 bytes	- 2147483648 a 2147483647
Decimal simple	float	4 bytes	- 3,4·1038 a 3,4·1038
Decimal doble	double	8 bytes	- 1,79·10308 a 1,79·10308

Tabla 5.4: Tabla comparativa de los tipos de datos utilizados

Para este segundo conjunto de datos los tiempos de ejecución se volvieron muy extensos cuando el algoritmo se ejecutaba de forma secuencial. Si se ejecutaba el programa para cada AC con 1000 iteraciones, los 4 tipos distintos de datos y con un AC cada vez mayor, se tendría que haber dejado la computadora ejecutando el programa por días continuos. Por ello se decidió ejecutar el programa solo en el rango de *N=250* hasta *N=10,000* los resultados siguientes fueron obtenidos realizando una regresión lineal para cada tipo de datos con la información obtenida anteriormente.

La regresión es lineal simple debido a que es el tipo de regresión que mejor se adaptó

a la curva que se generaba con los datos de *N=250* hasta *N=10,000*. Por lo ya mencionado se obtuvieron las siguientes ecuaciones y se generaron los datos restantes hasta *N=14,000*:

$$y_{int} = 0{,}0000169 * x_{int} - 3{,}8285 \tag{5.1}$$

$$y_{long} = 0{,}0000167 * x_{long} + 1{,}2361 \tag{5.2}$$

$$y_{float} = 0{,}0000191 * x_{float} - 3{,}0152 \tag{5.3}$$

$$y_{double} = 0{,}0000187 * x_{double} - 1{,}1588 \tag{5.4}$$

Donde:

- x: es el número de datos
- y: es el tiempo en segundos

5.2.1. Aceleración

Con los mismos datos del tiempo de ejecución se obtuvo la aceleración dividiendo el tiempo en secuencial entre el tiempo de su similar en paralelo.

La gráfica 5.4 de la aceleración crece rápidamente, pero vuelve a bajar del mismo modo, alcanza su punto máximo cuando el AC es de un tamaño de 1750x1750, obteniendo una aceleración de 61.53 como se puede ver en la tabla 5.5. Esto quiere decir que el algoritmo en paralelo pudo realizar el mismo proceso 61.53 veces más rápido que el algoritmo en secuencial.

En las misma gráfica (5.4), se puede notar que en cuanto la aceleración decrece comienza a tomar una tendencia muy similar a la de una recta con pendiente igual a 0. Que la pendiente se comporte de esta forma quiere decir que a partir de cierto número de datos la aceleración sera muy similar con estas tarjetas gráficas.

No se puede afirmar a partir de que cantidad de datos la aceleración se vuelve aproximadamente constante porque las dos tarjetas muestran datos y gráficas distintas, pero la misma tendencia (crecer, decrecer y volverse constante).

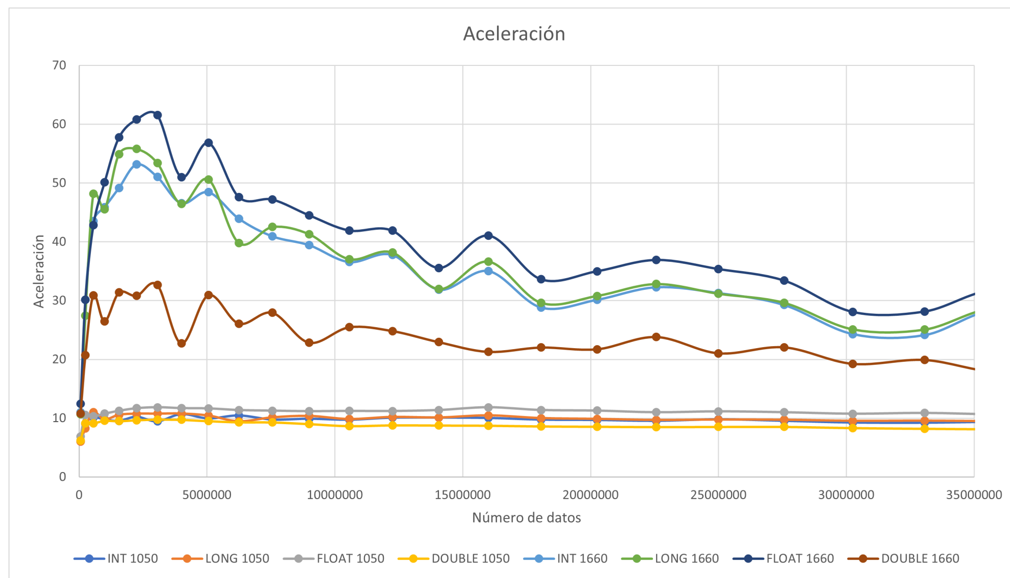


Figura 5.4: Gráfica ampliada de la aceleración del segundo conjunto de datos.[8]

N	NxN	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
250	62500	6.09210526	6.02597403	6.90849673	6.206060606	11.0238095	10.6666667	12.4352941	10.7789474
1750	3062500	9.43791776	10.7603563	11.8485101	9.771145144	51.0350493	53.3926002	61.5378788	32.6614311
3250	10562500	9.72585828	9.85212101	11.2236387	8.662114798	36.5678253	37.0519068	41.9318374	25.4681478
4750	22562500	9.5347632	9.68484615	10.9986579	8.501317229	32.2805524	32.8157255	36.9220239	23.8166231
5750	33062500	9.21643287	9.57472045	10.8888488	8.212223811	24.1428063	25.0707335	28.145205	19.9020922

Tabla 5.5: Tabla reducida de la aceleración para el segundo conjunto de datos.[9]

5.3. Ley de Amdahl

La ley de Amdahl establece que todos los algoritmos tienen un fragmento que se realiza de forma secuencial, aunque el algoritmo sea creado en paralelo tiene una sección que forzosamente se ejecuta de forma secuencial ya que no puede ser paralelizada. Por ello, es necesario encontrar el porcentaje del programa que se realiza en paralelo y la que se realiza en secuencial.

Por como está hecho el algoritmo, primero realiza la obtención de datos para construir el AC, genera las estructuras de datos y envía esta información a la memoria de la tarjeta gráfica. Todas estas tareas son realizadas en la CPU de forma secuencial. Los pa-

[8]Esta gráfica es una ampliación de la gráfica 6.5
[9]Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.6

sos siguientes son distribuir las células del AC a los distintos hilos de la tarjeta gráfica y procesarlos, tareas que se realizan de forma paralela.

La figura 5.5 muestra una línea de tiempo de como se ejecuta el algoritmo. La parte de color rojo hace referencia a al tiempo secuencial, la parte azul el tiempo que se puede paralelizar y la flecha representa el tiempo total.

Figura 5.5: Línea de tiempo de ejecución del algoritmo.

Para generar la tabla 5.6 se registraron el tiempo de ejecución del algoritmo en secuencial y el tiempo total.

N	NxN	INT		LONG		FLOAT		DOUBLE	
1000	1000000	0.002	0.307	0.002	0.271	0.002	0.271	0.005	0.549
5000	25000000	0.051	12.605	0.052	12.635	0.055	12.655	0.102	21.291
9000	81000000	0.168	64.127	0.166	64.236	0.165	64.128	0.331	91.608
13000	169000000	0.346	144.081	0.345	144.152	0.349	144.134	0.671	202.504
		Secuencial	Total	Secuencial	Total	Secuencial	Total	Secuencial	Total

Tabla 5.6: Tabla del tiempo de ejecución del algoritmo en segundos.[10]

La gráfica 5.6 representa el porcentaje del algoritmo en secuencial respecto al número de datos. A medida que el AC aumenta su tamaño disminuye su porcentaje en secuencial y aumenta el porcentaje en paralelo, esto se debe a que entre mayor es el AC se aprovecha más el paralelismo y el algoritmo tiene que enfocar más tiempo en la parte paralela que en la parte secuencial.

[10] Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.7

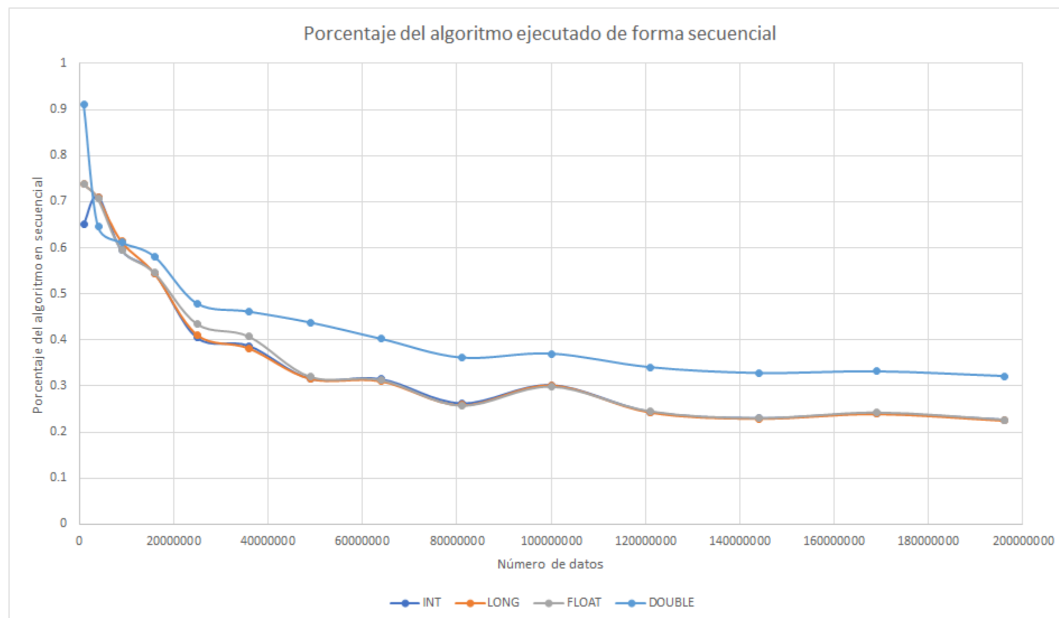


Figura 5.6: Gráfica que representa el porcentaje de la fracción del algoritmo en secuencial.[12]

N	NxN	INT_S %	LONG_S %	FLOAT_S %	DOUBLE_S %
1000	1000000	0.651465798	0.73800738	0.73800738	0.910746812
5000	25000000	0.404601349	0.411555204	0.434610826	0.479075666
9000	81000000	0.261980133	0.258422069	0.257297904	0.361322155
13000	169000000	0.240142698	0.239330706	0.24213579	0.331351479

Tabla 5.7: Tabla con los porcentajes de ejecución de tiempo en paralelo y secuencial[13]

El porcentaje secuencial de la tabla 5.7 se obtiene de dividir el tiempo secuencial entre el tiempo total de la tabla 5.6 y multiplicarlo por 100 como se muestra en la ecuación 5.5.

$$\alpha = \frac{Ts}{Tt} * 100 \tag{5.5}$$

Por otro lado, se puede notar en la gráfica 5.6 que el algoritmo mantiene un porcentaje similar para la mayoria de tipos de dato excepto para el tipo de dato DOUBLE, donde el porcentaje en secuencial es mayor al de los otros.

[12]Esta gráfica es una ampliación de la gráfica 6.6

[13]Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.8

5.3.1. Aceleración con la ley de Amdahl

La ley de Amdahl ayuda a encontrar la aceleración máxima teórica con un número de procesadores definido, esto quiere decir que con una pararelización adecuada de los datos se podría acelerar la velocidad de ejecución el número de veces resultante.

Para determinar la aceleración máxima se utilizó la ecuación mostrada en el primer capítulo, la ecuación 5.6

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}} \tag{5.6}$$

Donde: n representa el número de procesadores.

Con la ecuación 5.6 se generaron los resultados de la tabla 5.8. En este caso no se utilizaron procesadores sino hilos, debido a que la potencia de las tarjetas gráficas se basa en la paralelización sobre estos, por lo cual el número de hilos para esta función es $n = 22,528$. Por otro lado, α será igual a la parte secuencial que se obtuvo en la tabla 5.7, pero dividido entre 100 ya que necesitamos los valores entre 0 y 1, en la tabla están entre 0 y 100.

En la gráfica 5.7 y en la tabla 5.8 se puede notar que la aceleración máxima alcanzada es de 436.36 veces, es decir el algoritmo en este punto con 22,528 procesadores podria mejorar su velocidad 436.36 veces. Teóricamente significa que si este algoritmo tarda regularmente 100 segundos en ejecutarse en secuencial, la aceleración máxima que podria alcanzar es $\frac{100}{436,36} = 0{,}22s$. Es decir, baja 436.36 veces su tiempo o aumenta su velocidad, ambas conclusiones son correctas.

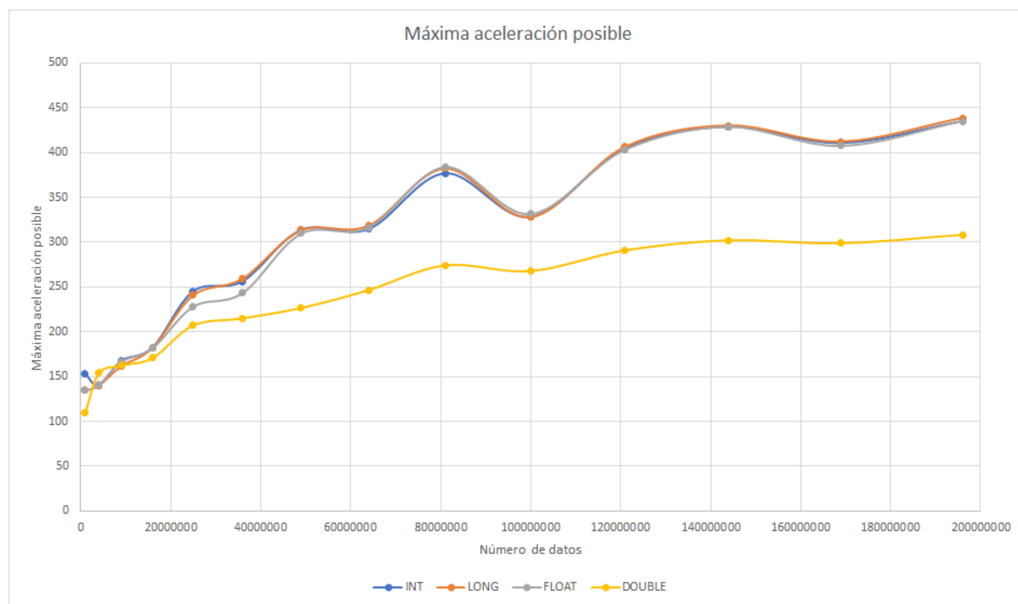


Figura 5.7: Gráfica que representa la máxima aceleración del algoritmo.[14]

N	NxN	INT	LONG	FLOAT	DOUBLE
1000	1000000	152.4678909	134.6958191	134.6958191	109.2722646
5000	25000000	244.4854419	240.398568	227.7746317	206.82809
9000	81000000	375.364914	380.4458185	382.079839	273.4145098
13000	169000000	408.879291	410.2412574	405.5742736	297.8178663

Tabla 5.8: Tabla de los resultados de la ley de Amdahl[15]

La gráfica 5.7 muestra un aumento rápido y llega a un punto donde comienza a estabilizarse, donde la pendiente es menor que al inicio. Estos resultados se deben al aumento de los datos, dado que entre más datos introducimos al algoritmo aprovecha más el paralelismo y, por ello, la aceleración esperada es mayor.

[14]Esta gráfica es una ampliación de la gráfica 6.7
[15]Esta tabla es solo un fragmento de los datos mostrados en la tabla 6.9

Capítulo 6

Conclusiones y trabajo futuro

Actualmente, tener una tarjeta gráfica es muy común, los precios son más accesibles que hace unos años y las características cada vez son mejores. Cada persona utiliza las tarjetas gráficas con fines distintos; en el mundo científico es bastante común hacer uso de sus especificaciones para mejorar las tareas designadas. Por otro lado, los AC representan distintos modelos y sistemas, pueden simular diferentes problemas del mundo real, volviéndolos una herramienta muy útil como se menciona en el primer capítulo.

El principal objetivo de este trabajo fue aumentar la velocidad con la que se ejecutan los autómatas celulares. La forma en la que se logra esto es mediante el análisis de más de una célula de manera simultánea.

Aprovechando las características de una tarjeta gráfica fue sencillo ejecutar más de una instrucción a la vez. Uno de los mayores retos se presentó al querer encontrar el modo en que se implementaran AC con un tamaño mayor al número máximo de hilos.

Para ejecutar los AC con un tamaño mayor al número de hilos, estos se reasignaron a células que no fueron analizadas a medida que terminaban. El proceso de reasignación no fue aleatorio, sino que se trató de un procedimiento controlado, el cual es explicado con mayor detalle en el cuarto capítulo.

La forma en que se midió el rendimiento entre el uso de la CPU en comparación a las tarjetas gráficas, fue con ayuda de tablas y gráficos, donde se registraron los tiempos de ejecución; con base en ellos se generaron métricas más elaboradas que sirvieron para su análisis.

El algoritmo desarrollado para paralelizar los procesos puede ser utilizado en distintas tarjetas gráficas. Dentro del trabajo se hizo uso dos modelos distintos de GPU, con el fin de demostrar que solo falta hacer un par de ajustes para que el algoritmo funcione

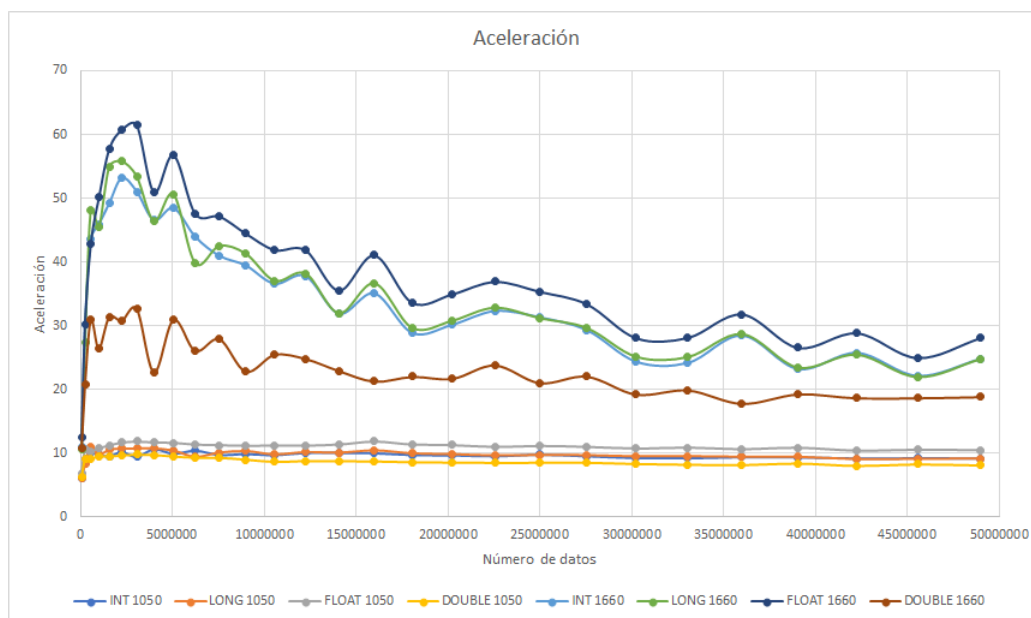


Figura 6.1: Gráfica de aceleración

correctamente.

Uno de los datos más relevantes que se obtuvo fue la aceleración máxima, alcanzada con la tarjeta gráfica GTX 1660 súper, como se muestra en la gráfica 6.1, se puede ver que se alcanzó un factor de aceleración de 56.84 unidades con un tamaño de 5,062,500 células. Esto quiere decir que realizó 56.84 veces más rápida la ejecución del AC en comparación del resultado que dio la CPU con el mismo número de datos.

Cabe mencionar que la tarjeta gráfica con mejor rendimiento en este trabajo (GTX 1660 súper), cuenta con especificaciones que pueden ser superadas fácilmente por GPUs más recientes, esto hace suponer que se pueden alcanzar tiempos menores, aceleraciones más altas y podrían ejecutarse autómatas de mayor tamaño gracias a que cuentan con mayor memoria.

Claro que no se puede comparar un núcleo de un procesador actual con uno de una tarjeta gráfica, ya que los procesadores trabajan a velocidades más altas, pero suelen ser pocos núcleos. Aún así, se mostró en este trabajo que la potencia de una tarjeta gráfica radica en la unión de todos los núcleos laborando en conjunto, utilizando y compartiendo sus recursos para resolver los problemas de forma paralela.

El software creado ayudaría a reducir el tiempo que llevaría crear y ejecutar un AC; desde hacer pruebas hasta generar los datos suficientes para solucionar el problema.

Antes de continuar, habría que recordar que los AC tienen distintas aplicaciones en

el mundo científico, tales como evolución de ecosistemas biológicos, crecimiento de incendios forestales, tráfico vehicular, etc. Otra de las aplicaciones es la propagación de los virus, se podría predecir el crecimiento del virus alrededor de un país, de un continente o incluso alrededor del mundo. Esta herramienta podría generar distintos panoramas, si se crean con exactitud los modelos y se cambian, ligeramente, las condiciones iniciales, de esta forma se podría calcular la expansión de un virus como el caso del Covid-19.

Las gráficas y tablas dentro de este trabajo muestran claramente las mejoras que tienen las tarjetas gráficas sobre el procesador a la hora de paralelizar los AC.

Se ha mostrado que las tarjetas gráficas son muy útiles para la paralelización de autómatas celulares, por lo que se abre la posibilidad de implementarlas en distintos tipos de algoritmos, desde aquellos para realizar búsquedas, hasta minar criptomonedas. Las propiedades que tienen las tarjetas gráficas y las características que tiene el AC se complementan perfectamente, pero esto únicamente es el comienzo de un proyecto que puede ir creciendo y mejorando.

Este trabajo podría tomarse como base y generar actualizaciones que ayuden a mejorarlo. Hasta ahora el trabajo reconoce autómatas celulares cuyo tamaño es de NXN, es decir, que tienen el mismo número de renglones y de filas. En este proyecto era útil para obtener los datos necesarios. Sin embargo, para analizar otro tipo de autómatas se podrían implementar modelos de tamaño NXM. Habrá casos donde se necesite ser más grande en renglones que en columnas o viceversa, todo depende de lo que se quiere analizar.

De igual forma, podría requerirse analizar autómatas que no sean de frontera cerrada, utilizando alguna otra de las que se mencionaron en el primer capítulo. La vecindad de Moore no es la única existente, podría requerirse la vecindad de Von-Neuman o quizá alguna personalizada. Esto también da paso a modificar el tipo de malla ya que se podría utilizar un espacio celular distinto (hexagonal, triangular, pentagonal, etc.).

Dejando a un lado las mejoras que se podrían hacer al AC, se puede pensar en generar datos para obtener estadísticas o gráficas que ayuden a determinar, automáticamente, cuántas células están vivas y cuántas muertas en cada evolución del autómata; si el AC está enfocado en otras reglas de transición y hay más de un estado, habría que determinar el número de células por estado, es decir se puede escalar en nivel estadístico.

Otro elemento importante a la hora de analizar AC son los gráficos. Se podrían generar imágenes con mejor presentación en cuanto a diseño y color de cada evolución del autómata, volviéndolo más legible y del agrado del usuario.

Pensando en que no todas las personas tienen conocimiento sobre la programación, es

importante generar interfaces que los usuarios puedan controlar fácilmente, incluso generar un software libre para los sistemas operativos que hagan uso de cómputo heterogéneo como lo son las computadoras, sistemas embebidos, consolas de videojuegos y/o celulares.

Anexo

Dentro de este apartado se encuentran las gráficas y tablas de los datos obtenidos en el trabajo, mismo en el que se hizo uso de dos conjuntos de datos y de dos tarjetas gráficas.

El primer conjunto de datos se utilizó para encontrar el punto de intersección entre el tiempo de ejecución; comparando el algoritmo en secuencia contra su equivalente en paralelo para ambas tarjetas gráficas. Además, para este grupo también se generó la siguiente gráfica de aceleración con el fin de analizar su tendencia.

El segundo conjunto de datos se utilizó para analizar la tendencia del tiempo de ejecución y la aceleración máxima alcanzada.

Por otro lado, se analizaron cuatro tipos de datos: int, long, float y double. Esto con el fin de comparar las tendencias de cada uno y, en caso de haber cambios drásticos, analizar a qué se debe.

La idea de hacer uso de dos tarjetas gráficas se concibió con el fin de comparar el rendimiento de modelos distintos y demostrar que el algoritmo no está hecho para un tipo en específico, sino que puede utilizarse en distintas tarjetas.

Gráfica del tiempo de ejecución del primer conjunto de datos

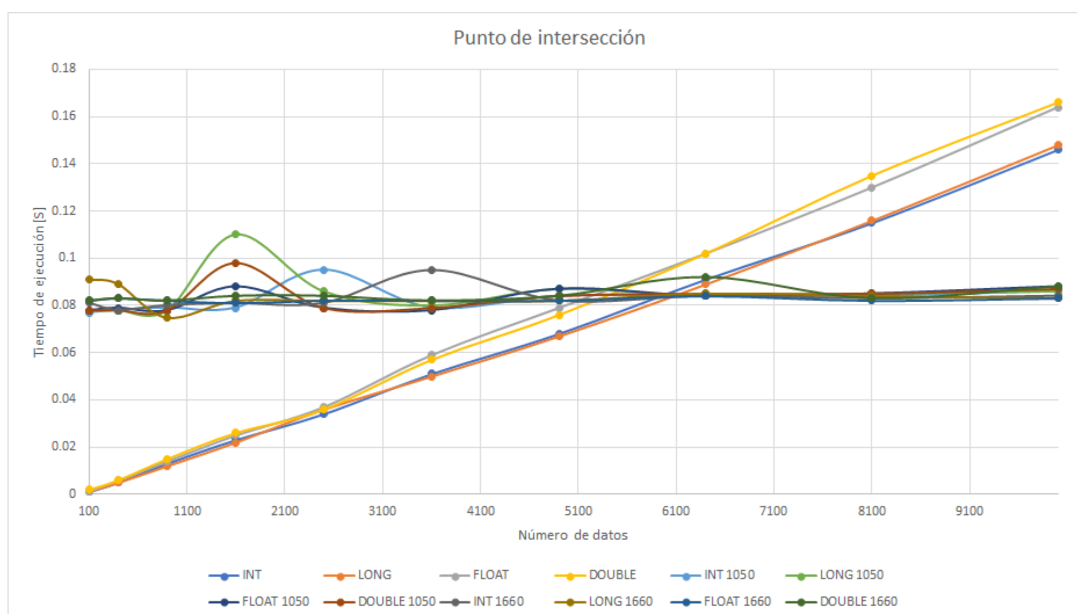


Figura 6.2: Gráfica del tiempo de ejecución del primer conjunto de datos.

Esta gráfica está relacionada a la tabla 6.1, muestra el punto de intersección entre los tiempos de ejecución en secuencial y los tiempos en paralelo para las tarjetas gráficas utilizadas. Con esta gráfica se puede notar a partir de cuantos datos es mejor hacer uso del algoritmo en paralelo y cuando sigue siendo más rápido su equivalente en secuencial.

Tabla con de los tiempo de ejecución del primer conjunto de datos

		Tiempo de ejecución en secuencial s CPU				Tiempo de ejecución en paralelo s GTX 1050 Ti				Tiempo de ejecución en paralelo s GTX 1660 Super			
N	NxN	INT	LONG	FLOAT	DOUBLE	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
10	100	0.001	0.001	0.001	0.002	0.077	0.078	0.078	0.078	0.081	0.091	0.082	0.082
20	400	0.005	0.005	0.006	0.006	0.078	0.078	0.079	0.078	0.078	0.089	0.083	0.083
30	900	0.013	0.012	0.014	0.015	0.079	0.078	0.078	0.078	0.08	0.075	0.082	0.082
40	1600	0.023	0.022	0.025	0.026	0.079	0.11	0.088	0.098	0.081	0.082	0.081	0.084
50	2500	0.034	0.036	0.037	0.036	0.095	0.086	0.079	0.079	0.081	0.082	0.082	0.084
60	3600	0.051	0.05	0.059	0.057	0.079	0.08	0.078	0.079	0.095	0.082	0.082	0.082
70	4900	0.068	0.067	0.079	0.076	0.084	0.087	0.087	0.084	0.082	0.082	0.082	0.084
80	6400	0.091	0.089	0.102	0.102	0.084	0.084	0.084	0.085	0.084	0.085	0.084	0.092
90	8100	0.115	0.116	0.13	0.135	0.084	0.085	0.085	0.085	0.083	0.084	0.082	0.083
100	10000	0.146	0.148	0.164	0.166	0.086	0.086	0.088	0.087	0.084	0.083	0.083	0.088

Tabla 6.1: Tabla del primer conjunto de datos.

Esta tabla contiene los tiempos de ejecución para los distintos tipos de datos, algoritmos y tarjetas gráficas. Los incrementos en los datos en el autómata celular son en intervalos de 10 en 10 renglones y columnas.

Gráfica de la aceleración del primer conjunto de datos

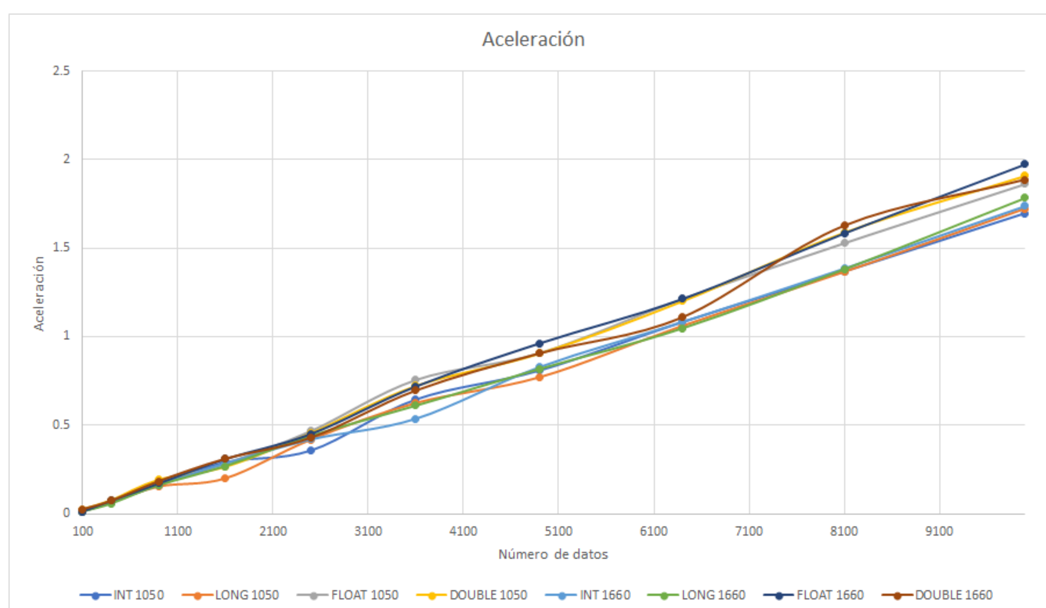


Figura 6.3: Gráfica de la aceleración primer conjunto de datos.

Esta gráfica está relacionada a la tabla 6.2, muestra la tendencia que toma la aceleración (tiempo secuencial entre el tiempo en paralelo) para el primer conjunto de datos. Están considerados los cuatro tipos de datos para las dos tarjetas gráficas empleadas en este trabajo.

Cuando la aceleración es menor a 1 significa que no existe una mejora de tiempo entre los algoritmos, por el contrario el tiempo es mayor para el caso paralelo, si la aceleración es igual a 1 los tiempos de ejecución son iguales y si es mayor a 1 existe una mejora en tiempo para el algoritmo paralelo con respecto al secuencial.

Tabla de la aceleración del primer conjunto de datos

		Aceleración Ts/Tp 1050				Aceleración Ts/Tp 1660			
N	NxN	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
10	100	0.01298701	0.01282051	0.01282051	0.025641026	0.01234568	0.01098901	0.01219512	0.024390244
20	400	0.06410256	0.06410256	0.07594937	0.076923077	0.06410256	0.05617978	0.07228916	0.072289157
30	900	0.16455696	0.15384615	0.17948718	0.192307692	0.1625	0.16	0.17073171	0.182926829
40	1600	0.29113924	0.2	0.28409091	0.265306122	0.28395062	0.26829268	0.30864198	0.30952381
50	2500	0.35789474	0.41860465	0.46835443	0.455696203	0.41975309	0.43902439	0.45121951	0.428571429
60	3600	0.64556962	0.625	0.75641026	0.721518987	0.53684211	0.6097561	0.7195122	0.695121951
70	4900	0.80952381	0.77011494	0.90804598	0.904761905	0.82926829	0.81707317	0.96341463	0.904761905
80	6400	1.08333333	1.05952381	1.21428571	1.2	1.08333333	1.04705882	1.21428571	1.108695652
90	8100	1.36904762	1.36470588	1.52941176	1.588235294	1.38554217	1.38095238	1.58536585	1.626506024
100	10000	1.69767442	1.72093023	1.86363636	1.908045977	1.73809524	1.78313253	1.97590361	1.886363636

Tabla 6.2: Tabla del primer conjunto de datos.

Esta tabla contiene los la relación entre el tiempo en secuencial y el tiempo en paralelo para ambas tarjetas y los cuatro tipos de datos. Para este conjunto de datos, los resultados no son tan distintos entre ambas tarjetas.

Gráfica del tiempo de ejecución del segundo conjunto de datos

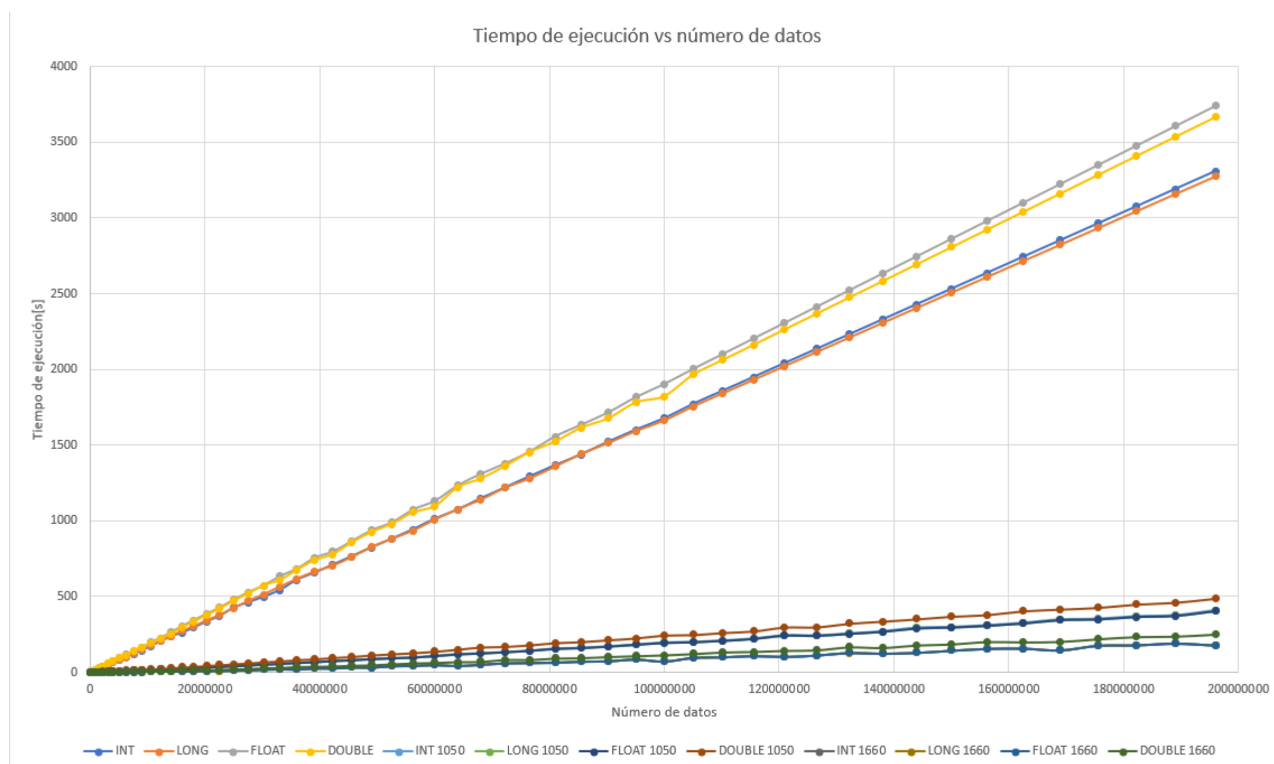


Figura 6.4: Gráfica del tiempo de ejecución del segundo conjunto de datos

Esta gráfica está relacionada a las tablas 6.3, 6.4 y 6.5; muestra la diferencia que hay entre ejecutar el algoritmo en secuencial y en paralelo para ambas tarjetas gráficas y con un conjunto de datos mayor. Los datos van creciendo en intervalos de 250 en 250 renglones y columnas.

La gráfica hace evidente la diferencia en tiempo de ejecución que existe entre el algoritmo en secuencial y paralelo.

Tabla del tiempo de ejecución segundo conjunto de datos en secuencial

		Tiempo de ejecución en secuencial s CPU			
N	NxN	INT	LONG	FLOAT	DOUBLE
250	62500	0.926	0.928	1.057	1.024
500	250000	3.741	3.513	4.284	4.23
750	562500	8.577	9.351	8.737	9.085
1000	1000000	15.097	14.929	16.386	16.752
1250	1562500	22.867	25.536	26.91	26.767
1500	2250000	35.62	37.493	40.623	40.512
1750	3062500	46.595	51.951	56.861	56.145
2000	4000000	66.707	67.182	72.736	73.859
2250	5062500	81.386	85.385	94.926	93.661
2500	6250000	106.777	96.408	115.098	114.127
2750	7562500	122.163	126.85	140.499	141.797
3000	9000000	144.489	151.51	162.737	164.142
3250	10562500	172.527	174.885	198.086	190.298
3500	12250000	204.028	206.217	225.295	222.952
3750	14062500	236.221	236.915	264.348	254.582
4000	16000000	259.958	271.443	304.213	295.908
4250	18062500	295.154	303.386	343.311	333.546
4500	20250000	333.715	340.503	385.212	376.978
4750	22562500	371.646	377.709	426.154	419.506
5000	25000000	425.752	423.997	479.977	471.068
5250	27562500	461.635	470.481	528.075	520.571
5500	30250000	498.221	514.313	572.975	573.408
5750	33062500	542.682	564.267	633.633	608.805
6000	36000000	610.172	616.148	682.469	675.457
6250	39062500	655.977	663.597	754.31	739.21
6500	42250000	709.136	702.663	795.932	778.643
6750	45562500	765.112	759.771	865.177	857.193
7000	49000000	824.615	825.475	936.822	924.806
7250	52562500	880.798	880.696	989.098	979.508
7500	56250000	943.052	934.767	1073.505	1054.786
7750	60062500	1013.204	1008.596	1131.669	1097.62
8000	64000000	1073.284	1075.488	1232.508	1220.824
8250	68062500	1147.726	1138.773	1309.461	1278.726
8500	72250000	1219.347	1217.138	1377.75	1361.178
8750	76562500	1292.965	1279.684	1457.05	1453.686
9000	81000000	1366.327	1357.657	1557.249	1524.153
9250	85562500	1438.603	1442.177	1634.701	1615.086
9500	90250000	1521.776	1513.079	1715.8	1676.596
9750	95062500	1601.272	1592.504	1816.715	1782.051
10000	100000000	1676.679	1662.334	1903.178	1823.487
10250	105062500	1771.728	1755.780	2003.679	1966.249
10500	110250000	1859.397	1842.411	2102.760	2063.255
10750	115562500	1949.178	1931.130	2204.229	2162.599
11000	121000000	2041.072	2021.936	2308.085	2264.280
11250	126562500	2135.078	2114.830	2414.329	2368.299
11500	132250000	2231.197	2209.811	2522.960	2474.655
11750	138062500	2329.428	2306.880	2633.979	2583.349
12000	144000000	2429.772	2406.036	2747.385	2694.380
12250	150062500	2532.228	2507.280	2863.179	2807.749
12500	156250000	2636.797	2610.611	2981.360	2923.455
12750	162562500	2743.478	2716.030	3101.929	3041.499
13000	169000000	2852.272	2823.536	3224.885	3161.880
13250	175562500	2963.178	2933.130	3350.229	3284.599
13500	182250000	3076.197	3044.811	3477.960	3409.655
13750	189062500	3191.328	3158.580	3608.079	3537.049
14000	196000000	3308.572	3274.436	3740.585	3666.780

Tabla 6.3: Tabla del tiempo de ejecución para el segundo conjunto de datos CPU (Secuencial)

Tabla del tiempo de ejecución segundo conjunto de datos en paralelo para la tarjeta 1050 TI

		Tiempo de ejecución en paralelo s GTX 1050 Ti			
N	NxN	INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050
250	62500	0.152	0.154	0.153	0.165
500	250000	0.413	0.424	0.406	0.463
750	562500	0.848	0.848	0.848	0.996
1000	1000000	1.542	1.525	1.523	1.751
1250	1562500	2.4	2.416	2.393	2.816
1500	2250000	3.485	3.485	3.478	4.191
1750	3062500	4.937	4.828	4.799	5.746
2000	4000000	6.248	6.234	6.218	7.59
2250	5062500	8.183	8.182	8.15	9.849
2500	6250000	10.218	10.191	10.12	12.252
2750	7562500	12.522	12.521	12.465	15.257
3000	9000000	14.608	14.608	14.558	18.224
3250	10562500	17.739	17.751	17.649	21.969
3500	12250000	20.251	20.204	20.101	25.344
3750	14062500	23.42	23.428	23.237	29.044
4000	16000000	25.929	25.928	25.684	33.884
4250	18062500	30.302	30.308	30.168	38.731
4500	20250000	34.477	34.485	34.153	44.058
4750	22562500	38.978	39	38.746	49.346
5000	25000000	43.458	43.441	43.039	55.249
5250	27562500	48.39	48.378	47.995	61.039
5500	30250000	53.857	53.883	53.312	68.843
5750	33062500	58.882	58.933	58.191	74.134
6000	36000000	64.988	65.034	64.076	82.858
6250	39062500	70.105	70.125	69.417	88.082
6500	42250000	77.175	77.161	76.447	96.929
6750	45562500	82.958	82.971	81.937	103.778
7000	49000000	89.815	89.819	89.273	114.142
7250	52562500	95.82	95.792	95.011	121.244
7500	56250000	100.599	100.596	99.646	128.653
7750	60062500	110.015	110.065	109.307	138.221
8000	64000000	122.164	122.163	120.169	151.625
8250	68062500	127.673	127.593	126.91	166.525
8500	72250000	135.148	135.097	134.679	170.021
8750	76562500	144.484	144.545	143.543	180.312
9000	81000000	157.799	157.803	156.871	195.058
9250	85562500	162.678	162.768	161.789	201.836
9500	90250000	172.77	172.853	171.345	214.207
9750	95062500	184.828	184.964	183.454	225.905
10000	100000000	197.597	197.602	195.732	245.253
10250	105062500	201.398	201.377	200.009	248.187
10500	110250000	210.68	210.804	209.148	262.093
10750	115562500	223.3	223.235	221.802	271.976
11000	121000000	244.633	244.63	243.631	298.16
11250	126562500	243.444	243.473	242.249	299.15
11500	132250000	256.756	256.66	255.395	322.925
11750	138062500	270.295	270.191	268.969	336.788
12000	144000000	294.241	294.162	290.186	351.834
12250	150062500	298.248	298.194	296.736	369.308
12500	156250000	309.674	309.739	307.871	379.838
12750	162562500	325.13	325.109	323.224	404.871
13000	169000000	347.77	347.685	345.458	416.626
13250	175562500	351.737	351.684	349.873	427.403
13500	182250000	368.441	368.293	366.357	449.316
13750	189062500	375.008	375.11	373.037	461.426
14000	196000000	407.354	407.175	404.335	488.079

Tabla 6.4: Tabla del tiempo de ejecución para el segundo conjunto de datos GTX 1050 Ti (Paralelo)

Tabla del tiempo de ejecución segundo conjunto de datos en paralelo para la tarjeta 1660 Súper

		Tiempo de ejecución en paralelo s GTX 1660 Super			
N	NxN	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
250	62500	0.084	0.087	0.085	0.095
500	250000	0.124	0.128	0.142	0.204
750	562500	0.197	0.194	0.204	0.294
1000	1000000	0.329	0.328	0.327	0.633
1250	1562500	0.465	0.465	0.466	0.853
1500	2250000	0.67	0.672	0.668	1.314
1750	3062500	0.913	0.973	0.924	1.719
2000	4000000	1.433	1.447	1.426	3.252
2250	5062500	1.68	1.687	1.67	3.027
2500	6250000	2.43	2.423	2.418	4.375
2750	7562500	2.984	2.983	2.977	5.076
3000	9000000	3.664	3.669	3.656	7.187
3250	10562500	4.718	4.72	4.724	7.472
3500	12250000	5.401	5.4	5.378	8.981
3750	14062500	7.413	7.412	7.433	11.099
4000	16000000	7.42	7.408	7.407	13.904
4250	18062500	10.24	10.242	10.212	15.147
4500	20250000	11.071	11.058	11.008	17.353
4750	22562500	11.513	11.51	11.542	17.614
5000	25000000	13.608	13.605	13.571	22.394
5250	27562500	15.775	15.883	15.807	23.606
5500	30250000	20.493	20.488	20.411	29.801
5750	33062500	22.478	22.507	22.513	30.59
6000	36000000	21.464	21.492	21.452	37.927
6250	39062500	28.323	28.345	28.411	38.339
6500	42250000	27.58	27.624	27.58	41.713
6750	45562500	34.63	34.666	34.662	45.905
7000	49000000	33.268	33.319	33.303	49.046
7250	52562500	43.098	43.043	43.121	55.438
7500	56250000	44.434	44.464	44.289	61.364
7750	60062500	50.489	50.493	50.603	64.408
8000	64000000	44.096	44.056	43.851	70.023
8250	68062500	52.78	52.769	52.916	70.716
8500	72250000	61.892	61.881	61.831	84.654
8750	76562500	64.225	64.223	64.23	83.857
9000	81000000	66.541	66.595	66.61	93.148
9250	85562500	73.125	73.14	73.21	95.905
9500	90250000	75.41	75.299	75.239	103.433
9750	95062500	86.926	86.826	86.914	109.272
10000	100000000	73.528	73.409	73.419	114.414
10250	105062500	97.954	97.936	98.006	122.996
10500	110250000	101.784	101.732	101.562	133.795
10750	115562500	109.932	109.94	110.007	137.222
11000	121000000	104.315	104.483	104.367	144.27
11250	126562500	113.372	113.378	113.467	147.379
11500	132250000	129.914	129.817	129.891	167.404
11750	138062500	125.11	124.981	125.16	162.999
12000	144000000	131.109	131.197	130.531	180.017
12250	150062500	145.021	144.989	145.138	186.273
12500	156250000	157.329	157.228	157.073	202.635
12750	162562500	159.137	159.077	159.118	200.32
13000	169000000	147.071	146.978	147	203.331
13250	175562500	178.023	177.9	178.07	220.593
13500	182250000	180.683	180.454	180.377	235.277
13750	189062500	192.139	192.162	192.119	238.126
14000	196000000	179.686	179.658	179.111	251.638

Tabla 6.5: Tabla del tiempo de ejecución para el segundo conjunto de datos GTX 1660 Super (Paralelo)

Gráfica de la aceleración para el segundo conjunto de datos

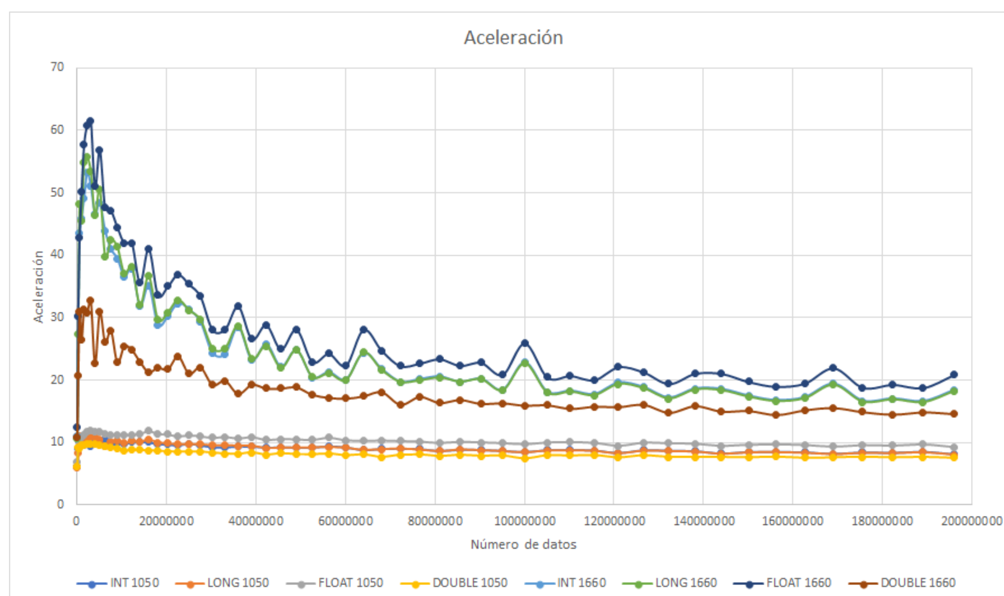


Figura 6.5: Gráfica de aceleración del primer conjunto de datos

Esta gráfica representa la aceleración para los datos de la tabla 6.6 que se obtuvieron al dividir el tiempo de ejecución en secuencial entre el tiempo de ejecución en paralelo.

Tabla de la aceleración para el segundo conjunto de datos

N	NxN	Aceleración Ts/Tp 1050				Aceleración Ts/Tp 1660			
		INT 1050	LONG 1050	FLOAT 1050	DOUBLE 1050	INT 1660	LONG 1660	FLOAT 1660	DOUBLE 1660
250	62500	6.09210526	6.02597403	6.90849673	6.206060606	11.0238095	10.6666667	12.4352941	10.77894737
500	250000	9.05811138	8.28537736	10.5517241	9.136069114	30.1693548	27.4453125	30.1690141	20.73529412
750	562500	10.1143868	11.0271226	10.303066	9.121485944	43.5380711	48.2010309	42.8284314	30.90136054
1000	1000000	9.79053178	9.7895082	10.7590282	9.567104512	45.887538	45.5152439	50.1100917	26.46445498
1250	1562500	9.52791667	10.5695364	11.2452988	9.505326705	49.1763441	54.916129	57.7467811	31.37983587
1500	2250000	10.2209469	10.7583931	11.6799885	9.66642806	53.1641791	55.7931548	60.8128743	30.83105023
1750	3062500	9.43791776	10.7603563	11.8485101	9.771145144	51.0350493	53.3926002	61.5378788	32.66143106
2000	4000000	10.6765365	10.7767084	11.697652	9.731093544	46.5505932	46.4284727	51.0070126	22.71186962
2250	5062500	9.94574117	10.4357125	11.647362	9.509696416	48.4440476	50.6135151	56.8419162	30.94185662
2500	6250000	10.4498923	9.46011186	11.3733202	9.314968985	43.9411523	39.7886917	47.6004963	26.08617143
2750	7562500	9.75586967	10.13098	11.2714801	9.293897883	40.9393432	42.5243044	47.194827	27.93479117
3000	9000000	9.89108708	10.3717141	11.1785273	9.00691396	39.4347707	41.2946307	44.5123085	22.83873661
3250	10562500	9.72585828	9.85212101	11.2236387	8.662114798	36.5678253	37.0519068	41.9318374	25.46814775
3500	12250000	10.0749593	10.2067412	11.2081488	8.797032828	37.7759674	38.1883333	41.8919673	24.82485247
3750	14062500	10.0862938	10.1124723	11.3761673	8.765390442	31.8657763	31.9637075	35.564106	22.93738175
4000	16000000	10.0257627	10.4691068	11.8444557	8.732971314	35.0347709	36.6418737	41.0710139	21.28222094
4250	18062500	9.74041317	10.0100963	11.3799722	8.6118613	28.8236328	29.6217536	33.6183901	22.02059814
4500	20250000	9.67935145	9.87394519	11.2790092	8.556402923	30.1431668	30.7924579	34.9938227	21.72408229
4750	22562500	9.5347632	9.68484615	10.9986579	8.501317229	32.2805524	32.8157255	36.9220239	23.81662314
5000	25000000	9.79686134	9.76029557	11.1521411	8.526271969	31.2868901	31.1647924	35.3678432	21.03545593
5250	27562500	9.53988427	9.72510232	11.0027086	8.528498173	29.2637084	29.621671	33.4076675	22.05248666
5500	30250000	9.25081234	9.54499564	10.7475803	8.329212847	24.311765	25.1031335	28.071873	19.24123352
5750	33062500	9.21643287	9.57472045	10.8888488	8.212223811	24.1428063	25.0707335	28.145205	19.90209219
6000	36000000	9.38899489	9.47424424	10.6509301	8.151982911	28.4276929	28.6687139	31.8137703	17.809397
6250	39062500	9.3570644	9.46305882	10.8663584	8.392293545	23.1605762	23.4114306	26.5499278	19.28088891
6500	42250000	9.18867509	9.10645274	10.4115531	8.033127341	25.7119652	25.4366855	28.8590283	18.66667466
6750	45562500	9.22288387	9.15706693	10.5590515	8.259872035	22.093907	21.9168926	24.9603889	18.67319464
7000	49000000	9.18126148	9.19042742	10.4939007	8.102241068	24.7870326	24.7749032	28.1302585	18.85589039
7250	52562500	9.19221457	9.19383665	10.4103525	8.078816271	20.4370968	20.4608415	22.9377334	17.66853061
7500	56250000	9.37436754	9.29228796	10.7731871	8.198689498	21.2236576	21.0230074	24.2386371	17.18900332
7750	60062500	9.20968959	9.16363967	10.3531247	7.941050926	20.0678168	19.9749668	22.3636741	17.04167184
8000	64000000	8.78559969	8.80371307	10.2564555	8.051600989	24.3397134	24.4118395	28.106725	17.43461434
8250	68062500	8.98957493	8.92504291	10.3180285	7.678883051	21.7454718	21.5803407	24.7460314	18.08255557
8500	72250000	9.02230888	9.00936364	10.2298799	8.005940443	19.7012053	19.6690099	22.282512	16.07931108
8750	76562500	8.94884555	8.85318759	10.1506169	8.062059098	20.1318023	19.9256341	22.6848825	17.33529699
9000	81000000	8.65865436	8.60349296	9.92693997	7.813845113	20.5336109	20.3867708	23.3786068	16.36270237
9250	85562500	8.84325477	8.86032267	10.1039069	8.001971898	19.6732034	19.7180339	22.3289305	16.84047756
9500	90250000	8.80810326	8.75355938	10.013715	7.826989781	20.1800292	20.0942775	22.8046625	16.20948827
9750	95062500	8.66357911	8.60980515	9.90283668	7.888497377	18.4210938	18.3413263	20.9024438	16.30839556
10000	100000000	8.48534644	8.41253631	9.72338708	7.435126176	22.8032722	22.6448256	25.9221455	15.93762127
10250	105062500	8.79714669	8.71886983	10.0179419	7.922448597	18.0873446	17.9278289	20.4444478	15.98628207
10500	110250000	8.82569062	8.73992476	10.0539321	7.872224745	18.2680628	18.1104382	20.7041984	15.42101723
10750	115562500	8.7289644	8.65065895	9.93782089	7.951432295	17.7307586	17.565307	20.0371663	15.75985447
11000	121000000	8.34340216	8.26528267	9.47369095	7.594177623	19.5664238	19.351819	22.1150824	15.69473903
11250	126562500	8.77030344	8.68609599	9.96630966	7.916759987	18.8324961	18.6529119	21.2778037	16.0694451
11500	132250000	8.68994882	8.60987727	9.87865777	7.663249981	17.1744115	17.0225094	19.4236691	14.78253208
11750	138062500	8.61809412	8.53795963	9.79287037	7.670548683	18.6190372	18.4578444	21.0448909	15.84886257
12000	144000000	8.2577598	8.1792893	9.46766832	7.658100127	18.5324539	18.3391091	21.0477572	14.96736419
12250	150062500	8.49034277	8.40821697	9.64890863	7.602729294	17.4611108	17.292897	19.727284	15.07329967
12500	156250000	8.51474938	8.42842232	9.68379549	7.696583807	16.7597614	16.6039834	18.9807274	14.42719668
12750	162562500	8.43809476	8.35421305	9.59683857	7.512266253	17.2397227	17.0736804	19.494517	15.18320063
13000	169000000	8.20160307	8.12096035	9.33509949	7.589252711	19.3938404	19.2106036	21.9379918	15.55040796
13250	175562500	8.42441298	8.34024252	9.57555613	7.685015664	16.6449153	16.4875202	18.8141099	14.88985938
13500	182250000	8.34922416	8.26736077	9.49336249	7.588545701	17.0253787	16.8730596	19.2816146	14.49208805
13750	189062500	8.51002579	8.42040961	9.6721734	7.665473445	16.6094741	16.4370679	18.7804358	14.85368565
14000	196000000	8.12210387	8.04183975	9.2512021	7.512677251	18.4130734	18.225941	20.8841713	14.57164657

Tabla 6.6: Tabla de los resultados de la aceleración del segundo conjunto de datos.

Tabla del porcentaje del algoritmo ejecutado en paralelo y en secuencial dependiendo del número de datos

TAM	NxN	INT		LONG		FLOAT		DOUBLE	
1000	1000000	0.002	0.307	0.002	0.271	0.002	0.271	0.005	0.549
2000	4000000	0.008	1.126	0.008	1.129	0.008	1.132	0.017	2.63
3000	9000000	0.019	3.196	0.02	3.258	0.019	3.188	0.038	6.213
4000	16000000	0.033	6.061	0.033	6.064	0.033	6.051	0.066	11.364
5000	25000000	0.051	12.605	0.052	12.635	0.055	12.655	0.102	21.291
6000	36000000	0.075	19.387	0.074	19.37	0.079	19.396	0.158	34.258
7000	49000000	0.1	31.699	0.1	31.663	0.101	31.604	0.208	47.527
8000	64000000	0.132	41.96	0.13	41.859	0.131	41.899	0.266	66.156
9000	81000000	0.168	64.127	0.166	64.236	0.165	64.128	0.331	91.608
10000	100000000	0.207	68.646	0.207	68.658	0.205	68.69	0.402	108.667
11000	121000000	0.248	101.725	0.247	101.751	0.249	101.766	0.485	142.502
12000	144000000	0.294	127.936	0.293	127.837	0.293	127.397	0.573	174.891
13000	169000000	0.346	144.081	0.345	144.152	0.349	144.134	0.671	202.504
14000	196000000	0.399	176.164	0.396	176.204	0.399	176.098	0.791	246.414
		Secuencial	Total	Secuencial	Total	Secuencial	Total	Secuencial	Total

Tabla 6.7: Tabla del tiempo de ejecución del algoritmo en segundos

Esta tabla contiene los tiempos de ejecución del algoritmo de su fracción secuencial y total cuando se ejecuta en la tarjeta gráfica RTX 1660 Super.

Gráfica del porcentaje del algoritmo ejecutado en secuencial dependiendo del número de datos

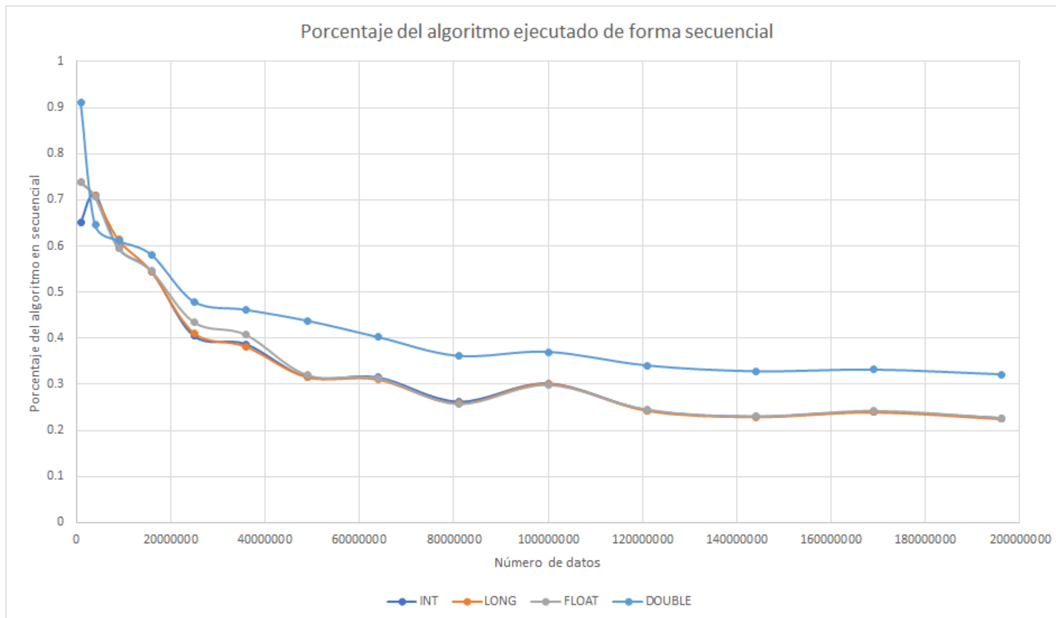


Figura 6.6: Gráfica que representa el porcentaje de la fracción del algoritmo en secuencial.

Esta gráfica representa el porcentaje del algoritmo en secuencial y como a mayor es el número de datos menor es el porcentaje de tiempo que se ejecuta el proceso secuencial.

Tabla del porcentaje del algoritmo ejecutado en paralelo y en secuencial dependiendo del número de datos

N	NxN	INT_S %	INT_P %	LONG_S %	LONG_P %	FLOAT_S %	FLOAT_P %	DOUBLE_S %	DOUBLE_P %
1000	1000000	0.651465798	99.3485342	0.73800738	99.26199262	0.73800738	99.26199262	0.910746812	99.08925319
2000	4000000	0.710479574	99.28952043	0.708591674	99.29140833	0.706713781	99.29328622	0.646387833	99.35361217
3000	9000000	0.594493116	99.40550688	0.613873542	99.38612646	0.595984944	99.40401506	0.611620795	99.3883792
4000	16000000	0.54446461	99.45553539	0.544195251	99.45580475	0.545364403	99.4546356	0.580781415	99.41921859
5000	25000000	0.404601349	99.59539865	0.411555204	99.5884448	0.434610826	99.56538917	0.479075666	99.52092433
6000	36000000	0.386857172	99.61314283	0.382034073	99.61796593	0.407300474	99.59269953	0.461206142	99.53879386
7000	49000000	0.315467365	99.68453264	0.315826043	99.68417396	0.3195798	99.6804202	0.43764597	99.56235403
8000	64000000	0.314585319	99.68541468	0.310566425	99.68943357	0.312656627	99.68734337	0.402079932	99.59792007
9000	81000000	0.261980133	99.73801987	0.258422069	99.74157793	0.257297904	99.7427021	0.361322155	99.63867784
10000	100000000	0.301547068	99.69845293	0.301494363	99.69850564	0.298442277	99.70155772	0.369937516	99.63006248
11000	121000000	0.243794544	99.75620546	0.242749457	99.75725054	0.244678969	99.75532103	0.3403461	99.6596539
12000	144000000	0.229802401	99.7701976	0.229198119	99.77080188	0.229989717	99.77001028	0.32763264	99.67236736
13000	169000000	0.240142698	99.7598573	0.239330706	99.76066929	0.24213579	99.75786421	0.331351479	99.66864852
14000	196000000	0.226493495	99.77350651	0.224739506	99.77526049	0.226578382	99.77342162	0.321004488	99.67899551

Tabla 6.8: Tabla con los porcentajes de ejecución de tiempo en paralelo y secuencial

Para este conjunto de datos se utilizo únicamente la tarjeta GTX 1660 Súper. Como todo algoritmo en paralelo cuenta con una parte en secuencial, se tomo el tiempo que hacia uso del paralelismo y el tiempo total con ello se pudo obtener el porcentaje. Los porcentajes van creciendo ya que entre mayor es el número de datos mayor es el tiempo que el algoritmo está haciendo uso se su programación paralela.

Gráfica de los resultados de la ley de Amdahl

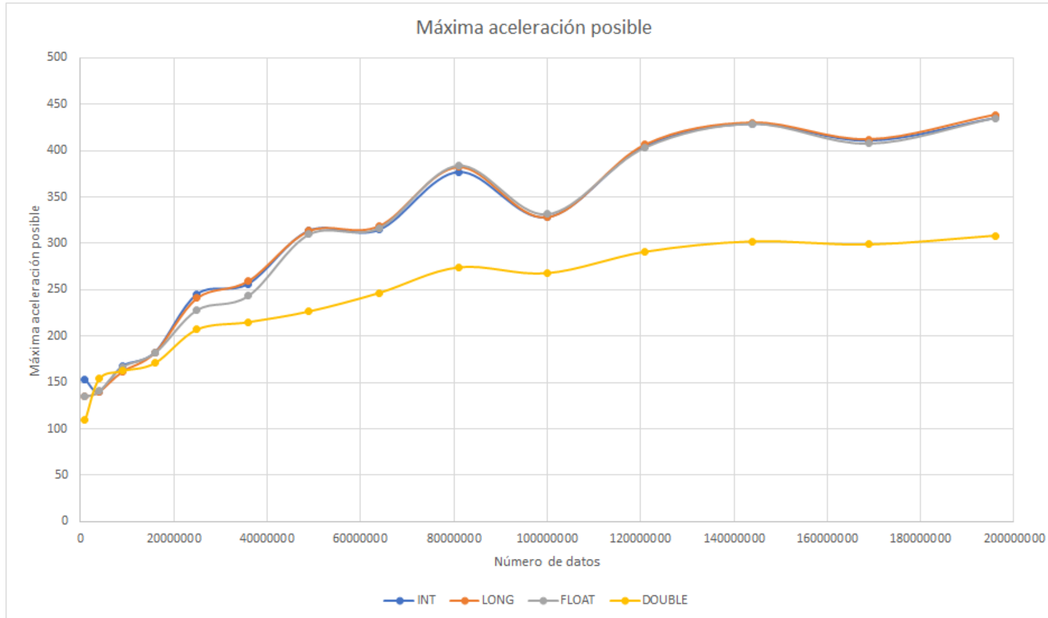


Figura 6.7: Gráfica que representa la máxima aceleración del algoritmo.

Esta gráfica está relacionada a los datos de la tabla 6.8, representa la aceleración máxima posible dependiendo el tamaño del AC y el tipo de datos.

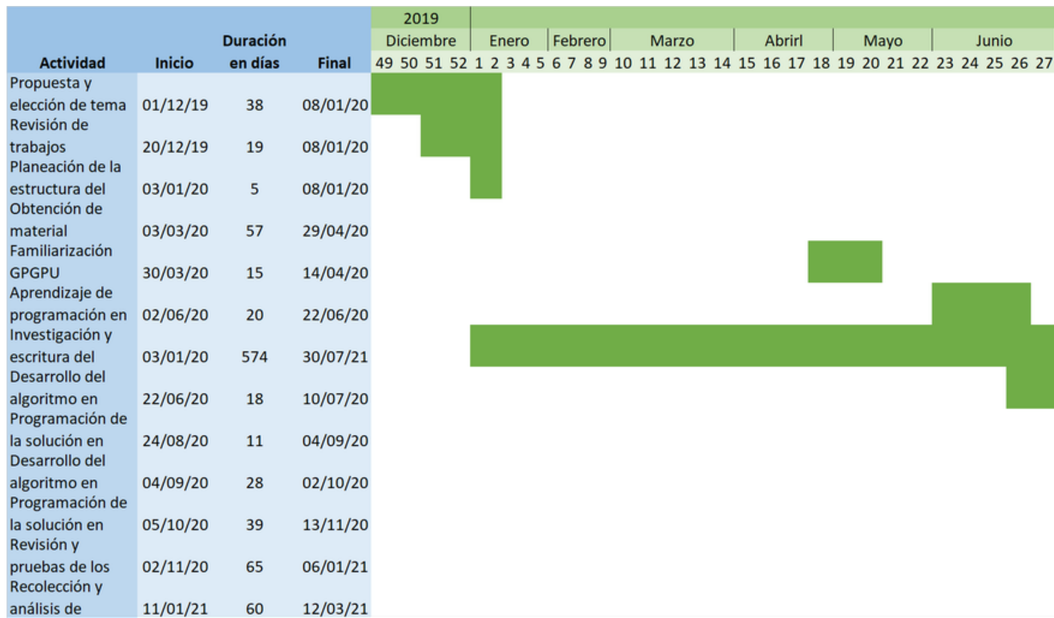
Tabla de los resultados de la ley de Amdahl

N	NxN	INT	LONG	FLOAT	DOUBLE
1000	1000000	152.4678909	134.6958191	134.6958191	109.2722646
2000	4000000	139.8822556	140.2526235	140.622979	153.6574954
3000	9000000	166.9712089	161.7376542	166.5563484	162.3290805
4000	16000000	182.1893971	182.2788453	181.8912311	170.8833398
5000	25000000	244.4854419	240.398568	227.7746317	206.82809
6000	36000000	255.5721629	258.7616463	242.8827373	214.76529
7000	49000000	312.6052288	312.2551293	308.6376557	226.2108404
8000	64000000	313.4695264	317.4688236	315.3761631	246.001852
9000	81000000	375.364914	380.4458185	382.079839	273.4145098
10000	100000000	326.8266477	326.8829513	330.1768975	267.1225453
11000	121000000	402.864134	404.5674009	401.4338756	290.0485111
12000	144000000	426.9287514	428.0329643	426.587622	301.1531
13000	169000000	408.879291	410.2412574	405.5742736	297.8178663
14000	196000000	433.0459789	436.3602389	432.8868554	307.2865385

Tabla 6.9: Tabla de los resultados de la ley de Amdahl

Resultados de los datos aplicando la ley de Amdahl para obtener la aceleración máxima teórica, dependiendo del porcentaje del algoritmo en secuencial.

Diagrama de Gantt parte 1 de 3



Actividad	Inicio	Duración en días	Final
Propuesta y elección de tema	01/12/19	38	08/01/20
Revisión de trabajos	20/12/19	19	08/01/20
Planeación de la estructura del	03/01/20	5	08/01/20
Obtención de material	03/03/20	57	29/04/20
Familiarización GPGPU	30/03/20	15	14/04/20
Aprendizaje de programación en	02/06/20	20	22/06/20
Investigación y escritura del	03/01/20	574	30/07/21
Desarrollo del algoritmo en	22/06/20	18	10/07/20
Programación de la solución en	24/08/20	11	04/09/20
Desarrollo del algoritmo en	04/09/20	28	02/10/20
Programación de la solución en	05/10/20	39	13/11/20
Revisión y pruebas de los	02/11/20	65	06/01/21
Recolección y análisis de	11/01/21	60	12/03/21

Figura 6.8: Diagrama que representa las actividades realizadas.

Este diagrama representa las principales actividades realizadas dentro de este proyecto, muestra la fecha de inicio, la fecha final y los días transcurridos. Dado a su tamaño se dividió en 3 partes.

Diagrama de Gantt parte 2 de 3

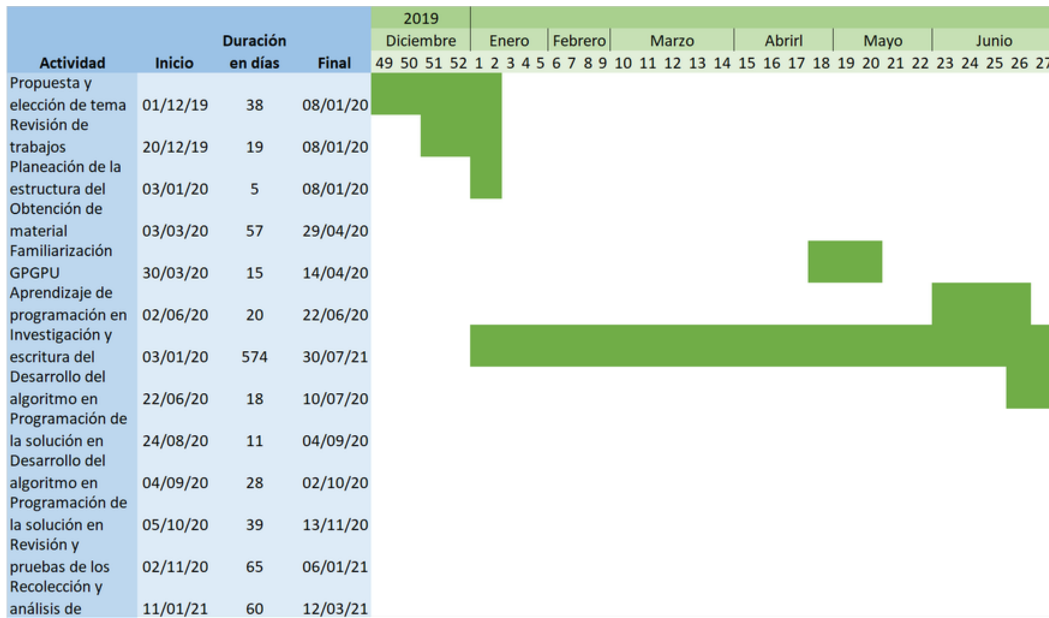
Actividad	Inicio	Duración en días	Final
Propuesta y elección de tema	01/12/19	38	08/01/20
Revisión de trabajos	20/12/19	19	08/01/20
Planeación de la estructura del	03/01/20	5	08/01/20
Obtención de material	03/03/20	57	29/04/20
Familiarización GPGPU	30/03/20	15	14/04/20
Aprendizaje de programación en	02/06/20	20	22/06/20
Investigación y escritura del	03/01/20	574	30/07/21
Desarrollo del algoritmo en	22/06/20	18	10/07/20
Programación de la solución en	24/08/20	11	04/09/20
Desarrollo del algoritmo en	04/09/20	28	02/10/20
Programación de la solución en	05/10/20	39	13/11/20
Revisión y pruebas de los	02/11/20	65	06/01/21
Recolección y análisis de	11/01/21	60	12/03/21

2020
Julio | Agosto | Septiembre | Octubre | Noviembre | Diciembre | Enero
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 1 2 3 4

Figura 6.9: Diagrama que representa las actividades realizadas.

Este diagrama representa las principales actividades realizadas dentro de este proyecto, muestra la fecha de inicio, la fecha final y los días transcurridos. Dado a su tamaño se dividió en 3 partes.

Diagrama de Gantt parte 3 de 3



2019
Diciembre | Enero | Febrero | Marzo | Abrirl | Mayo | Junio
49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Actividad	Inicio	Duración en días	Final
Propuesta y elección de tema	01/12/19	38	08/01/20
Revisión de trabajos	20/12/19	19	08/01/20
Planeación de la estructura del	03/01/20	5	08/01/20
Obtención de material	03/03/20	57	29/04/20
Familiarización GPGPU	30/03/20	15	14/04/20
Aprendizaje de programación en	02/06/20	20	22/06/20
Investigación y escritura del	03/01/20	574	30/07/21
Desarrollo del algoritmo en	22/06/20	18	10/07/20
Programación de la solución en	24/08/20	11	04/09/20
Desarrollo del algoritmo en	04/09/20	28	02/10/20
Programación de la solución en	05/10/20	39	13/11/20
Revisión y pruebas de los	02/11/20	65	06/01/21
Recolección y análisis de	11/01/21	60	12/03/21

Figura 6.10: Diagrama que representa las actividades realizadas.

Este diagrama representa las principales actividades realizadas dentro de este proyecto, muestra la fecha de inicio, la fecha final y los días transcurridos. Dado a su tamaño se dividió en 3 partes.

Bibliografía

[1] Texas advanced computing center. 8 things you should know about gpgpu technology. pág. 2. URL https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf.

[2] AMD. High-performance computing explained. *AMD*, pág. 2, 2019. URL https://www.amd.com/system/files/documents/hpc-explained.pdf.

[3] BioMates. Autómatas Celulares. 1997. URL https://www.sgapeio.es/INFORMEST/VICongreso/taller/applets/biomates/ace/ace_acl/ace_acl.htm.

[4] Alberto Cano Rojas y Ángela Rojas Matas. Autómatas celulares y aplicaciones. 2016. URL http://www.fisem.org/www/union/revistas/2016/46/01_13-307-1-ED.pdf.

[5] Tullio Ceccherini-Silberstein y Michel Coornaert. *Cellular Automata and Groups*. Cham, Suiza, 2010. ISBN 9783642140341.

[6] John Cheng, Max Grossman, y Ty McKercher. *Professional CUDA C Programming*. 2014.

[7] Shane Cook. *CUDA programming a developer's guide to parallel computing with GPUs*. Morgan Kaufmann, Waltham, 2013.

[8] Diego Escarlon. Autómatas celulares. 2005. URL http://axxon.com.ar/zap/278/c-zapping0278.htm.

[9] Fabian García. Cómputo de alto rendimiento (HPC) & big data. 2014.

[10] GOLHOOD. Game of life (el juego de la vida). 2015. URL http://www.golhood.com/2015/08/20/game-of-life-el-juego-de-la-vida/.

[11] Carlos Gonzáles. ¿Qué diferencia a una CPU y una GPU, si ambos son procesadores? 2018. URL https://www.adslzone.net/2018/01/12/cpu-vs-gpu-diferencias/.

[12] María del Carmen Gómez Fuentes. *Análisis de requerimientos*. México, CDMX, 2011.

[13] José Manuel Gómez Álvarez. Sistemas de información multiagente. 2005. URL https://web.archive.org/web/20071009182341/http://www.irisel.com/~jmgomez/IT/doctorate/taller_2.htm.

[14] National institute of biomedical imaging y bioengineering. Modelado Computacional. 2018. URL https://www.nibib.nih.gov/espanol/temas-cientificos/modelado-computacional.

[15] Milagros Jaramillo Rivas. Enfoque sistémico de la administración. teoría de sistemas. 2006. URL http://umc.edu.ve/mjaramillo/Unidades/Unidad%20VII/Guia/Guia.pdf.

[16] Top 500 The list. Top 500. 2020. URL https://top500.org/lists/top500/2020/06/.

[17] Pierre-Yves Louis. *Probabilistic Cellular Automata*. Springer, Cham, Suiza, 2018. ISBN 9783319655567.

[18] Luca. An introduction to parallel architectures. 2012. URL http://courses.eees.dei.unibo.it/mphseng-old/wp-content/uploads/2018/05/15_Parallel_Architectures_LUCA.pdf.

[19] Enrique Madridejos Zamora. *Diseño e implementacón de funcionalidades OpenMP, basadas en modelos de concurrencia desarrollados en lenguaje Go*. Tesis Doctoral, Universidad Politécnica de Madrid, Facultad de ciencias, España, Madrid, 2015.

[20] Johannes Müller y Karl-peter Hadeler. *Cellular Automata: Analysis and Applications*. Springer, Cham, Suiza, 2017. ISBN 9783319530420. URL http://www.springer.com/series/3733.

[21] Nvidia. Getting started with CUDA. 2008. URL https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf.

[22] Nvidia. CUDA Zone | NVIDIA Developer. 2019. URL https://developer.nvidia.com/cuda-zone.

[23] Cesar Perez, Jose Cámara, y Pedro Sanchez. *Introducción a la programacion en CUDA* . 2016. URL https://riubu.ubu.es/bitstream/handle/10259/3933/Programacion_en_CUDA.pdf?sequence=1.

[24] Luis Plaza. Modelación matemática en ingeniería. 2016. URL http://www.scielo.org.mx.pbidi.unam.mx:8080/scielo.php?script=sci_arttext&pid=S2448-85502016000200047&lng=es&nrm=iso&tlng=es.

[25] David Reyes. Descripción y aplicaciones de los autómatas celulares. pág. 6. URL http://delta.cs.cinvestav.mx/~mcintosh/cellularautomata/Summer_Research_files/Arti_Ver_Inv_2011_DARG.pdf.

[26] Manuel Romer. *El juego de la vida [Inteligencia en redes de comunicaciones].* URL http://www.it.uc3m.es/jvillena/irc/practicas/09-10/04mem.pdf.

[27] Fernando Sancho Caparrini. Sistemas Complejos, Sistemas Dinámicos y Redes Complejas. 2015. URL http://www.cs.us.es/~fsancho/?e=641.

[28] Amert Tanya, Otterness Nathan, Yang Ming, Anderson James H., y Smith F. Donelson. Gpu scheduling on the nvidia tx2: Hidden details revealed. pág. 12.

[29] Claudia. Vanney. Modelos cientifícos. 2016. URL http://dia.austral.edu.ar/Modelos_cient%C3%ADficos.

[30] Rafael Vejarano. Rendimiento y análisis energético en formación de haz con GPGPU-Sim. 2015. URL https://revistas.utp.ac.pa/index.php/id-tecnologico/article/view/589/html.

[31] Eric W Weisstein. Regla 30."De MathWorld: un recurso web de Wolfram. URL https://mathworld.wolfram.com/Rule30.html.

[32] Uri Wilensky. NetLogo. 1999. URL http://ccl.northwestern.edu/netlogo/index.shtml.

[33] Barry wilkinson y Allen Michael. *Parallel programming Techniques and applications using networked workstations and parallel computers*. 2 edón., 2005.

[34] Stephen Wolfram. *A new kind of science*. Wolfram Media, Inc, Canada, 2002. ISBN 1800943962. URL www.stephenwolfram.com.

[35] Jun Zhang. Parallel computing chapter 7 performance and scalability. pág. 26. URL https://www.cs.uky.edu/~jzhang/CS621/chapter7.pdf.