

Prototipo de un Ambiente de Programación Paralelo para la Construcción de Autómatas Celulares de Propósito General

Aurelio Antonio Vivas Meza

1110348

Laboratorio de Redes y Sistemas Distribuidos (Lascilab)
Centro de Investigación en Bioinformática y Fotónica (CIBioFi)
Universidad del Valle, Cali, Colombia
aurelio.vivas@correounivalle.edu.co

Director:

Prof. John Sanabria Ph.D
Universidad del Valle, Cali, Colombia
john.sanabria@correounivalle.edu.co

Codirector:

Prof. Arturo Arguelles Ph.D
Universidad Santiago de Cali, Cali, Colombia
arturo.arguelles00@usc.edu.co

Codirector:

Carlos Arango
Estudiante Doctorado
Universidad del Valle, Cali, Colombia
carlos.arango.gutierrez@correounivalle.edu.co

Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Programa Académico de Ingeniería de Sistemas
Cali, Noviembre 10 de 2017

Agradecimientos

Me gustaría expresar mi más sincera gratitud a las siguientes personas y organizaciones.

- Mis supervisores, John, Arturo y Carlos por su invaluable retroalimentación durante el presente trabajo de investigación.
- Al Proyecto CIBioFi y el Sistema General de Regalías de Colombia (BPIN-2013000100007), COLCIENCIAS, Gobernación del Valle del Cauca, por haber financiado el presente proyecto de investigación.
- Laboratorio de Redes y Sistemas Distribuidos, Universidad del Valle, Cali, Colombia. Por ser el espacio que abrió las puertas del conocimiento.
- Observatorio Sismológico y Geofísico del Suroccidente Colombiano, Universidad del Valle, Cali, Colombia, el cual fue mi segundo espacio de trabajo.
- Laboratorio de Compiladores de la Universidad Federal de Minas Gerais, Brazil, en especial a Gleison Souza, por brindarme el primer impulso en el desarrollo de la herramienta Pragcc.
- A mi familia, por que gracias a su esfuerzo y motivación se han dado las facilidades para llegar en un principio a la universidad y luego, tener la posibilidad de culminarla.

Resumen

La programación en paralelo es una rama de la computación de alto rendimiento HPC (siglas en ingles), que se ha proliferado en los últimos años con mas fuerza en el ámbito científico, esta rama nace como un paradigma de programación ajustada a las nuevas arquitecturas de computadores paralelos (PCs, Supercomputadores, Estaciones de Trabajo, etc). Dada las nuevas características en los computadores y la creciente complejidad de los problemas tratados en el mundo científico, la programación en paralelo ha tomado un lugar significativo en el escalafón de alternativas para la solución de problemas.

El estudio de sistemas dinámicos como los fluidos, el clima, los gases, entre otros, ha permitido a través de la historia la observación y descubrimiento de gran variedad de fenómenos físicos. Entre los variados métodos existentes para llevar a cabo estas observaciones se encuentra el método de Autómata Celular. Este método en particular, permite observar el compartimento del sistema, y en muchos casos encontrar la solución numérica a la ecuación diferencial que gobierna el comportamiento del mismo. Si bien es cierto, este tipo de sistemas en el campo de la física son modelados normalmente bajo el paradigma de programación secuencial, lo cual, es contradictorio siendo que los modelos de Autómata Celular pueden tomar ventaja del poder computacional disponible en computadores paralelos actuales. Es por ello que el presente trabajo de investigación se enfocó en el desarrollo de una herramienta que permitiera a dichos modelos de automata celular, tomar provecho del paralelismo presente en computadores paralelos. Además como producto del proceso de investigación en el ámbito de autómatas celulares, surge un modelo de cambio climático para interpolación de temperaturas en el Valle del Cauca, descrito en la Sección 2.5.

En este orden de ideas, el **Capítulo 1** describe el contexto bajo el cual se plantea la iniciativa de implementar un ambiente de programación para la construcción de autómatas celulares de propósito general; El **Capítulo 2**, comprende lo que es el surgimiento de los automatas celulares, descripción de sus características, clasificación, propiedades y aplicaciones. El **Capítulo 3** describe la evolución en la arquitectura de computadores, desde el diseño básico de von Neumann hasta sus mejoras, dándole la bienvenida a las arquitecturas multiprocesador y multinúcleo. Así mismo se describe cómo las unidades de procesamiento de gráficos pasan de ser únicamente procesadores de gráficos a realizar computo de propósito general. Además, se describen las técnicas (modelos y patrones de programación paralela) que permiten acceder al poder computacional disponible en las arquitecturas anteriormente mencionadas. **Capítulo 4**, describe el ambiente de programación y cómo este

aplica las técnicas de programación en paralelo para optimizar la ejecución de modelos de automata celular de propósito general. En el **Capítulo 5**, se hace la validación del prototipo, llevando a cabo pruebas en tiempo de ejecución y utilización de recursos del código generado por el prototipo de ambiente de programación. Finalmente el presente documento concluye en el **Capítulo 6**, donde se discuten los resultados obtenidos en las pruebas llevadas a cabo en el capítulo anterior, finalizando con la discusión del trabajo futuro.

Índice general

1. Planteamiento del Problema	9
1.1. Descripción del Problema	9
1.2. Formulación del Problema	10
1.3. Justificación	10
1.4. Objetivo General	11
1.4.1. Objetivos Específicos	11
1.4.2. Resultados Esperados	12
1.5. Alcances	12
2. Autómatas Celulares	13
2.1. Una Breve Historia	13
2.2. Máquinas Auto Replicables	16
2.3. Autómata Celular	21
2.3.1. Espacio celular	22
2.3.2. Interacciones locales	23
2.3.3. Estados discretos	24
2.3.4. Dinámica discreta	24
2.3.5. Condiciones de Frontera	25
2.4. Clasificación	26
2.5. Aplicaciones	27
2.5.1. El Juego de la Vida	27
2.5.2. Solución Numérica de la Ecuación de Calor: Propuesta de un Modelo de Interpolación de Temperaturas para el Valle Geográfico del Río Cauca	29
3. Computación Paralela	37
3.1. Computadores Paralelos	38
3.1.1. Computadores MIMD	41
3.1.2. Computadores SIMD	43
3.2. Modelos de Programación Paralela	47
3.2.1. Clasificación	48

3.2.2.	OpenMP	50
3.2.3.	OpenACC	52
3.3.	Patrones de Programación Paralela	53
3.3.1.	Map	54
3.3.2.	Stencil	56
3.3.3.	Esqueleto Elemental de Autómata Celular Para el Pro- totipo	58
4.	Ambiente de Programación Paralela	61
4.1.	Plantillas	62
4.1.1.	Detalles de la Implementación	63
4.2.	Pragcc	64
4.2.1.	Detalles de la Implementación	64
4.3.	Coder	65
4.3.1.	Detalles de la Implementación	67
5.	Validación del Prototipo	71
5.1.	Estudio Modelo de Autómata Celular	71
5.1.1.	Descripción de los Modelos	72
5.1.2.	Detalles de Experimentación	74
5.1.3.	Hardware y Software usado en el Experimento	75
5.1.4.	Análisis y Resultados Implementación 1	76
5.1.5.	Análisis y Resultados Implementación 2	78
5.1.6.	Consolidación de Resultados	80
5.2.	Otros Ambientes de Programación	81
5.2.1.	CAMEL/CARPET	82
5.2.2.	Parallware Trainer	83
6.	Conclusión	85
6.1.	Resumen	85
6.2.	Resultados	85
6.3.	Trabajo Futuro	87
A.	Listados de Código	89
A.1.	Prototipo de Ambiente de Programación de Autómatas Celu- lares	89
A.1.1.	Plantillas	89
A.1.2.	Pragcc	90
A.2.	Estudio Modelo Autómata Celular	91

Índice de cuadros

2.1. Resumen de desarrollos históricos en el campo de los autóma- tas celulares[36]	17
5.1. Convenciones para los Experimentos	73

Listings

3.1.	Ejemplo de la implementación en Python del Patrón <i>Map</i> . . .	54
3.2.	Ejemplo de un Ciclo que Cumple con las Características del Patrón <i>Map</i>	54
3.3.	Ejemplo de un Ciclo que no Cumple con las Características del Patrón <i>Map</i>	55
A.1.	Ejemplo de una Plantilla en Lenguaje C99 con Sintaxis del Motor de Plantillas Jinja2. Observe que las variables se encuentran entre dobles llaves.	89
A.2.	Ejemplo de un Archivo <i>context.yml</i> para la plantilla <i>stencil</i> . .	90
A.3.	Ejemplo Esqueleto Elemental de la Función evolve de la plantilla <i>stencil</i>	90
A.4.	Ejemplo del Archivo <i>parallel.yml</i>	90
A.5.	Ejemplo de un Código Paralelizado con <i>pragcc</i>	91
A.6.	Implementación 1 Secuencial, 1_S	91
A.7.	Implementación 1 paralelizada con directivas OpenMP, 1_{MP} .	92
A.8.	Implementación 1 paralelizada con directivas OpenACC, 1_{ACC} .	92
A.9.	Implementación 2 Secuencial, 2_S . Basada en la implementación del método <i>Jacobi Iteration</i> [11].	92
A.10.	Implementación 2 paralelizada con directivas OpenMP, 2_{MP} . .	93
A.11.	Variación implementación 2, 1_{MP}	93
A.12.	Implementación 2 paralelizada con directivas OpenACC, 2_{ACC} .	93

Capítulo 1

Planteamiento del Problema

Esta sección introduce los antecedentes que se han presentado en cuanto al cambio de arquitecturas de computadores secuenciales a arquitecturas de computadores paralelos. Se plantea una problemática en cuanto a la programación secuencial de autómatas celulares y se propone una alternativa de programación paralela asistida por una herramienta que permita la programación en paralelo de autómatas celulares.

1.1. Descripción del Problema

En la última década y media, los diseñadores de computadoras, específicamente de microprocesadores, pasaron de diseñar arquitecturas mono núcleo al diseño de arquitecturas multinúcleo, con el objetivo de sobreponerse a las limitaciones físicas impuestas al intentar diseñar microprocesadores con frecuencias de reloj por encima de los 4 GHz [56]. A esta limitación se suma el elevado costo que impone el aumento en la densidad de transistores en un microprocesador, y el desequilibrio latente entre la velocidad a la cual el procesador procesa datos e instrucciones y la velocidad a la cual se obtienen datos e instrucciones de memoria principal [43].

Es por ello que las nuevas arquitecturas de computadores requieren entonces un replanteamiento de las soluciones algorítmicas que tradicionalmente se llevan a cabo de forma secuencial, dado que estas soluciones al ser ejecutadas bajo las nuevas arquitecturas presentan un bajo rendimiento en comparación al que podrían alcanzar siendo modeladas de forma paralela.

Sin embargo, construir aplicaciones programadas en paralelo es mucho mas complejo que construir una aplicación secuencial, quizás en un orden de magnitud [22]. Del mismo modo la curva de aprendizaje de un nuevo paradigma de programación para el investigador es muy alta. Lo anterior nos lleva a pensar en la implementación de una herramienta que permita al investigador en primera instancia, agilizar el proceso de programación de autómatas celulares, y como consecuencia, obtener un mejor rendimiento en la ejecución de sus experimentos, siendo el proceso de programación en paralelo un proceso transparente para el usuario.

1.2. Formulación del Problema

¿Cómo mapear un modelo de simulación basado en autómatas celulares especificado por medio de una interfaz gráfica amigable, a un lenguaje de programación paralelo capaz de explotar el paralelismo y la concurrencia de un computador paralelo?

1.3. Justificación

Los modelos de sistemas basados en autómatas celulares son altamente paralelizables [51], esto permite explotar de forma natural el paralelismo disponible en computadores paralelos. Sin duda es de gran ventaja cuando se desea explorar o simular sistemas cuyas matrices modelo son significativamente grandes y el modelo de programación secuencial o tradicional se queda corto en el procesamiento. Dada la complejidad que implica la programación en paralelo, es importante que el científico cuente con una herramienta que le permita acelerar el proceso de programación y adicionalmente el proceso de simulación, de esta forma, ocultando al usuario la complejidad de programar en paralelo y al mismo tiempo, entender la arquitectura paralela en la cual se ejecutará el modelo de autómatas celular.

1.4. Objetivo General

Desarrollar un prototipo de un ambiente de programación y simulación de autómatas celulares de propósito general con el fin de facilitar su programación paralela y al mismo tiempo los modelos generados exploten de manera eficiente el paralelismo subyacente en el computador paralelo en el cual serán ejecutados.

1.4.1. Objetivos Específicos

1. Conocer las propiedades, características, tipos y aplicaciones de autómatas celulares mas usados, para determinar el tipo, características y propiedades que soportará el prototipo de ambiente de programación paralela para autómatas celulares planteado.
2. Conocer los modelos y patrones de programación paralela que permitan la programación paralela de autómatas celulares de propósito general, para determinar qué modelo y patrón soportará el prototipo de ambiente de programación paralela planteado.
3. Implementar una interfaz que permita modelar gráficamente un autómata celular de propósito general dados los parámetros (tamaño de la grilla, estructura de la células, estructura del vecindario, conjunto de reglas) configurados por el usuario.
4. Plantear un conjunto de pruebas en tiempo de ejecución donde se consideren aspectos como las dimensiones del modelo y la utilización de recursos del computador paralelo donde se ejecute el modelo, con el fin de comparar su funcionamiento versus el funcionamiento de alternativas como JTrend, NetLogo, Golly, CAMEL/CARPET entre otras que en la actualidad se encuentren disponibles.

1.4.2. Resultados Esperados

1. Informe de los tipos, propiedades, características y aplicaciones de autómatas más comunes en los modelos de simulación basados en autómatas celulares.
2. Informe de los modelos y patrones de programación más comunes usados para la ejecución paralela de autómatas celulares.
3. Prototipo de un ambiente de programación paralela para autómatas celulares de propósito general junto con la documentación del prototipo.
4. Informe del conjunto de pruebas (tiempo de ejecución, dimensiones del modelo).

1.5. Alcances

El presente proyecto se limitará a la implementación de una interfaz de modelado que asistirá al usuario en la programación paralela de modelos de autómatas celulares. Adicionalmente, esta interfaz estará limitada a la programación de autómatas celulares de propósito general homogéneos¹, cuyo ambiente pueda ser descrito en grillas cuadradas de dos dimensiones. Dado que los sistemas distribuidos entran en la categoría de computadores paralelos, es necesario resaltar que este proyecto solo se enfocará a sistemas de memoria compartida.

¹Los modelos basados en autómatas celulares homogéneos se caracterizan por tener un único conjunto de reglas que rige el comportamiento del sistema

Capítulo 2

Autómatas Celulares

El presente capítulo detalla en primera instancia cuáles fueron aquellos pasos históricos que dieron origen a lo que se conoce hoy en día cómo autómatas celulares. En efecto el primer paso fue presentando por von Neumann y su estudio de las “máquinas auto replicables”. Al rededor de este estudio, se desencadenan una serie de modelos matemáticos basados en estas máquinas, justamente uno de estos modelos, con ayuda del matemático Stanislaw Ulam, se convierte en lo que conocemos como el método de autómata celular.

Antes de abordar las siguientes secciones es necesario hacer una aclaración importante en cuanto al termino *cellular automata*. La palabra *cellular*, a pesar de tener dos interpretaciones en el Español, **células** y **celdas**, en el contexto y origen de los automatas celulares, su interpretación debería ser **celdas**, por tanto en Español *cellular automata* se refiere a la palabra **automatas de celdas** o mejor conocido en al literatura como automatas celulares. Esta aclaración es relevante dado que en gran parte de la literatura respecto a este tema no se hace esta aclaración.

2.1. Una Breve Historia

Durante la Segunda Guerra Mundial (1939 - 1945), John von Neumann escuchó acerca de la creación del primer computador electrónico (ENIAC) construido para propósitos de la misma. Este se notó fascinado por el potencial de aquel computador, que para ese entonces fue considerado el más rápido en el conjunto de dispositivos que pudieron ser producidos en dicha época para hacer cálculos matemáticos. Entre los años 1945 y 1946, Neumann se vio involucrado en las modificaciones llevadas a cabo sobre este computador para realizar cálculos relacionados con la bomba de hidrógeno y el Proyecto Manhattan, de lo cual surge el concepto de programa almacenado para la

arquitectura de computadores. Su interés en resolver sistemas de ecuaciones diferenciales parciales no lineales en general y ecuaciones para predecir el clima en particular, lo condujeron naturalmente a desear computadoras cada vez más potentes. En consecuencia, dedicó gran parte de su vida a la búsqueda de una solución que permitiera eliminar aquellas limitaciones presentes en la construcción de estas poderosas máquinas [60, p. 19].

Von Neumann comparó los mejores computadores que pudieron ser contruidos en aquella época con los organismos naturales más inteligentes y concluyó que hay tres factores que limitaron a la ingeniería la construcción de computadores realmente poderosos: el tamaño de los componentes disponibles¹, la confiabilidad de estos componentes², y la falta de una teoría acerca de la organización lógica de sistemas que constan de elementos de computación³ [60, p. 20]. De estas circunstancias nace el hecho de que Neumann, se planteara la construcción de una nueva teoría de autómatas, esta nueva teoría, además de tener en cuenta la teoría existente, involucraría lo que, él denotó como autómatas naturales⁴. A continuación la teoría resultante se consolidó en el libro *Theory of Self-Reproducing Automata*. Este escrito, describe cómo él, daría solución a cada una de las tres problemáticas planteadas, en particular se centra en el fenómeno de la auto replicación, lo cual dado su parecido con la palabra auto reparación, se presume el estudio de este fenómeno tenía como objetivo solucionar el problema de la confiabilidad de los componentes. Sin embargo no se sabe a ciencia cierta por qué Neumann se involucra en la construcción de una máquina auto replicable, aunque gracias a este estudio se debe la creación de los autómatas celulares. [60, p. 19,20]

Theory of Self-Reproducing Automata se encuentra dividido en tres partes: *máquinas confiables y componentes no confiables*, *máquinas auto replicables* y por último, *comparación entre las máquinas de computación y el cerebro humano*. Aunque la temática general del libro gira en torno al estudio de autómatas o máquinas extremadamente complejas, este describe en gran parte el estudio de Neumann al rededor del fenómeno de la auto replicación en los seres vivos, y las problemáticas que conllevan la construcción de un modelo de autómata artificial capaz de emular la complejidad subyacente en

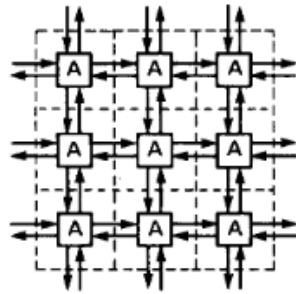
¹El tamaño de las piezas de un computador en aquella época era grande en comparación con los órganos que componen a un organismo inteligente como el ser humano.

²En comparación con el mecanismo de regeneración celular en los seres vivos, las máquinas o en sí sus componentes, no cuentan con la capacidad de regenerarse o en su defecto, su capacidad para recuperarse de errores es pobre.

³Probablemente, von Neumann creía que en el sistema nervioso humano el cual consta de elementos de computación (neuronas) podría estar la clave para construir procesadores más poderosos.

⁴Incluye el sistema nervioso, sistemas auto-reproductores y auto-reparables, también aspectos adaptativos y evolutivos de los organismos.

este fenómeno. Von Neumann durante su estudio desarrolla varios modelos de máquina auto replicable, entre los cuales se mencionan los siguientes: *máquina cinemática*, *máquina celular*, *máquina de tipo neuronal*, *máquina continua* y *máquina probabilista*. Gran parte de los modelos mencionados fueron descritos en forma de ecuaciones diferenciales parciales. El primer modelo de máquina auto replicable, el modelo cinemático, resultó ser un modelo muy complicado para ser tratado matemáticamente, en consecuencia en 1951 Stanislaw Ulam⁵ quien considera el problema de Neumann, le sugiere que su modelo de máquina auto replicable sería más amable para ser tratado si este fuese descrito en un *espacio celular* o *espacio de celdas*, parecido a un tablero de damas chinas [29, p. 192]. Esto significó el paso de un modelo complejo de autómatas basado en ecuaciones diferenciales parciales, a un modelo discreto en dos dimensiones planteado sobre un espacio celular equivalente en forma, a una matriz. El paso anterior dio origen al *modelo celular* de máquina auto replicable, Figura 2.2, el cual fue empleado por distintos autores en el futuro en numerosos proyectos de modelado y simulación [29, p. 195].



(a) Espacio celular de autómatas de estados finito

UNEXCITABLE	U			
ORDINARY TRANSMISSION				
				
SPECIAL TRANSMISSION				
				
CONFLUENT	c_{00}	c_{01}	c_{10}	c_{11}
SENSITIZED	s_0	s_0	s_1	s_{00}
	s_{01}	s_{10}	s_{11}	s_{000}

(b) Conjunto de 29 estados.

Figura 2.1: Modelo Celular de máquina auto replicable. El modelo resultante de la sugerencia hecha por Ulam a Neumann fue entonces un tablero rectangular compuesto por celdas, donde cada celda contaba de un *autómata finito idéntico*, Figura 2.1a, donde cada uno estos podía tomar un estado del conjunto de 29 estados posibles, Figura 2.1b. Así mismo, cada autómata finito podía establecer comunicación con cuatro de sus vecinos (arriba, abajo, izquierda y derecha) [29, p. 192-194].

⁵ Stanislaw Ulam trabajó en *Los Alamos National Laboratory* junto con Neumann en los años 1940s. Él se encontraba estudiando el crecimiento de los cristales, usando modelos simples basados en enrejados de dos o más dimensiones, de aquí que su sugerencia hacia Neumann fuera llevar su modelo a este tipo de enrejados (o modelos discretos).

En un principio, Von Neumann fue inspirado de cierta forma por las ideas de Turing al rededor de la construcción de una autómeta capaz de imitar el pensamiento humano [52]. Ambos buscaban encontrar aquella organización lógica presente en los humanos requerida para realizar algunas de las funciones que estos pueden llevar a cabo, para luego construir un modelo abstracto que emule dichas capacidades. Turing se enfocó en estudiar el comportamiento del cerebro humano y Neumann en el estudio del sistema nervioso humano en comparación con el funcionamiento de los computadores digitales. [36, p. xxix] [26, p. 22-23]. Finalmente, el modelo celular han sido estudiados en áreas como: ciencias de la computación, matemáticas, física, complejidad, biología, vida artificial entre otras, al mismo tiempo, han sido reinventados en gran cantidad de ocasiones bajo nombres como: *tessellation automata*, *cellular spaces*, *iterative arrays*, *homogeneous structures*, *uniform array*, *mosaic automata* y *universal spaces*, nombrados en ingles para evitar interpretaciones ambiguas. Sin embargo, a pesar de los avances disjuntos en este campo, el origen y desarrollo en cuanto a esta materia se le adjudica a Alan Turing (1936), John Von Neumann (1948), y Stanislaw Ulam (1950). Adicionalmente a Stephen Wolfram (1983) y John Conway (1969) cada uno de los cuales hace contribuciones originales en cuanto a su fundamento y significado [36]. Por otro lado, es importante resaltar que Neumann no pudo concretar su libro debido a su muerte prematura. De aquí que, Arthur Burks a mediados de 1966 publicará una recopilación de sus trabajos en el campo de la arquitectura de computadores y también en el ámbito de la teoría de autómetas. En el Cuadro 2.1 se presenta de forma cronológica las contribuciones originales hechas en el campo de los automatas celulares.

2.2. Máquinas Auto Replicables

“Can any automaton construct other automata that are exactly like it?”

— Von Neumann, *The Theory of Self-Reproducing Automata*

Un autómeta es una pieza de maquinaria o mecanismo de control diseñado para seguir de forma automática (o autónoma) una secuencia de operaciones o instrucciones codificadas [4]. Adicionalmente, el comportamiento de dicha maquinaria puede ser definido de manera precisa en términos matemáticos (modelo matemático) [60]. Von Neumann adicionalmente emplea el término *autómeta natural*, como aquel sistema natural que es autónomo o actúa sin intervención externa. En este orden de ideas, Neumann idea un modelo de

Año	Investigador	Descubrimiento
1936	Turing	Formaliza el concepto de computabilidad; Máquina de Turing Universal
1948	Von Neumann	Abstrajo la estructura lógica de la vida; introdujo su modelo de autómata auto replicable como un modelo inspirado en la biología, combinado con conceptos de computación
1950	Ulam	Propone a Neumann la necesidad de un modelo más realista para llevar a cabo su autómata
1966	Burks	Completó y describió el trabajo realizado por Von Neumann, que dada su muerte prematura no alcanzo a finalizar
1970	Conway	Introdujo el autómata celular denominado en la actualidad " <i>The Game of Life</i> "
1977	Toffoli	Aplica la teoría de autómatas celulares a modelamiento de leyes físicas.
1980 2002	Wolfram	Se centra en el estudio de lo autómatas lineales (o elementales), desglosando sus características y propiedades. Muestra la importancia de estos en los diferentes campos de la ciencia afirmando que este método de experimentación es un nuevo tipo de ciencia; publica los resultados de su investigación en su libro " <i>A New kind of Science</i> ".

Cuadro 2.1: Resumen de desarrollos históricos en el campo de los autómatas celulares[36]

autómata artificial inspirado en procesos evolutivos, específicamente hablando, el proceso de replicación de las células en los seres vivos. Este autómata fue descrito por él en ese entonces de la siguiente manera: *El autómata constructor flota en una superficie, rodeada por un suministro ilimitado de partes. El autómata constructor contiene en su memoria una descripción del autómata a ser construido (su replica). Operando bajo la dirección de esta descripción, este toma las piezas necesarias y las ensambla en el autómata deseado* [60, p. 81].

“Living organism are very complicated aggregations of elementary parts.”

— Von Neumann, *The Theory of Self-Reproducing Automata*

Teniendo en mente que un sistema natural es la composición de pequeños componentes, Neumann divide el problema de la construcción de esta máquina auto replicable en varios sub problemas tales como: monitoreo de partes cercanas, clasificación de las partes encontradas, recolección y fusión de partes, lectura de las instrucciones necesarias para construir un nuevo autómata, una unidad de control que envíe señales a los otros componentes y dirija el funcionamiento de la máquina, entre otras funcionalidades. Ya nos podemos imaginar qué tan complejo se torna dicho modelamiento, teniendo en cuenta que en principio Neumann obvia ciertos factores físicos que puedan afectar el funcionamiento de la máquina.

Dentro de este contexto, se obtienen cinco modelos: el cinemático, celular, neuronal, probabilista y continuo. El *Modelo Cinemático*, se ocupa de resolver problemas como el movimiento, contacto, posicionamiento, fusión y cortado. Consta de una cinta, la cual funciona como memoria de instrucciones, el programa entonces indica a la máquina buscar la pieza solicitada en la instrucción. Luego, la máquina entra en un procedimiento de identificación, para determinar si la pieza seleccionada es la indicada en la cinta de instrucciones. En caso de no ser así, esta pieza es arrojada en el ambiente. A medida que se encuentra la secuencia de piezas indicadas, la máquina procede a su ensamble. Cuando la máquina termina, habrá producido un duplicado de ella misma. El *Modelo Celular*, resulta del modelo cinemático, a razón de que el segundo no parecía ser muy amable hablando en términos matemáticos. Stanislaw Ulam quien trabajó con Neumann en el proyecto Manhattan le sugiere que su modelo sería más amable para ser tratado matemáticamente si este fuese planteado en un formato de *espacio celular* o también llamado, *espacio de celdas*, Figura 2.1a. El modelo resultante fue entonces un tablero rectangular compuesto por celdas, donde cada celda contaba con un *autómata finito* idéntico. Cada autómata del tablero podía asumir uno de 29 estados, Figura 2.1b, y a su vez cada uno de estos tenía la posibilidad de comunicarse con cuatro vecinos, de tal forma que el mismo autómata y sus vecinos conformarían un vecindario. Entonces, el nuevo estado de cada automata o celda en el espacio celular dependería del vecindario de la misma [29, p. 192-194].

En relación con lo anterior, el modelo de máquina auto replicable quedaría expresado en el espacio celular de manera similar a como se muestra en la Figura 2.2b, la cual muestra un ejemplo de cómo quedaría el brazo constructor de la máquina auto replicable embebida en el espacio celular. A

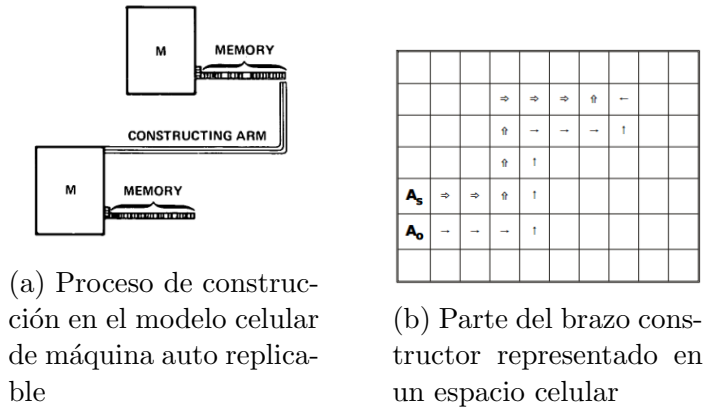


Figura 2.2: Modelo Celular de máquina auto replicable. La Figura 2.2b, muestra lo que sería el brazo constructor de la máquina auto replicable, es importante observar que el brazo es representado por una configuración particular del conjunto de estados en el espacio celular, así mismo sería representada la máquina completa. Por otro lado, la Figura 2.2a, muestra en general el modelo de máquina replicable deseado. El modelo completo de máquina auto replicable puede ser visto en [24, p. 6].

grandes rasgos, la máquina M se compone de una unidad de construcción (o unidad de control) y una memoria, Figura 2.2a. Entonces, el objetivo de la máquina constructora es cambiar el estado de aquellas celdas en el espacio celular que no están definidas, teniendo en cuenta las indicaciones de la memoria de instrucciones. Al finalizar la construcción de la replica, la máquina padre debe copiar lo que hay en su memoria a su replica, para que esta pueda empezar el proceso de construcción de una nueva máquina. A continuación, el modelo resultante termina siendo muy complejo y difícil de simular dada la capacidad de computación en dicha época. Es por ello que, se puede encontrar en la literatura modelos de autómatas celulares que intentan estudiar de forma simplificada el fenómeno de la auto replicación, por ejemplo se encuentra el modelo de autómatas auto replicable de Edgar F. Codd, que es un modelo simplificado del autómata de von Neumann. En este ámbito sobresale también el autómata de Chris Langton, Figura 2.3, quien en el proceso de investigación del mismo da surgimiento al campo de *Vida Artificial* [34, p. 2].

Dentro del contexto descrito anteriormente Neumann desarrolla el *Modelo Neuronal* de máquina auto replicable. Este modelo consideró de cierta forma la idealización de un grupo de neuronas en lugar de pequeños autómatas como espacio celular, las cuales eran estimuladas por señales de las neuronas

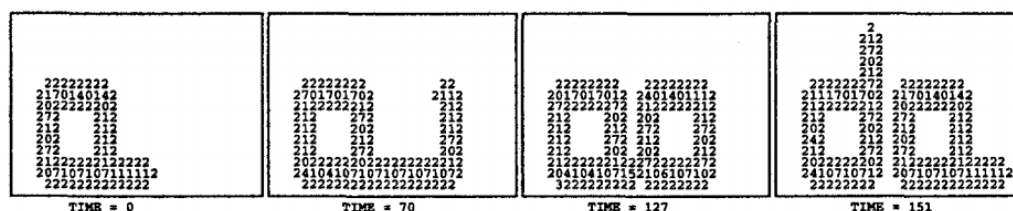


Figura 2.3: Modelo de autómatas celular auto replicable de Langton (o Loops de Langton) [57]

vecinas, de esta forma él buscó que su modelo se acercara un poco mas a lo que ocurre en los sistemas naturales como es el caso del sistema nervioso humano [29, p. 194]. Quizás para Neumann no fue suficiente la misma complejidad de su modelo y finalmente, se encamina en la creación de los modelos probabilista y continuo, buscando aproximar la idea de autómatas auto replicable a lo que sería mas cercano a un sistema biológico, tomando cada vez mas elementos del conjunto de variables que se podrían encontrar en este tipo de sistemas. De aquí que estos últimos modelos fueran tratados como ecuaciones diferenciales cada vez más complicadas.

Es importante concluir que como resultado de su estudio en el ámbito de las máquinas auto replicables, von Neumann determina que las siguientes características y capacidades pueden ser demostradas para éstas: (1) Lógica universal, la habilidad de funcionar como un computador de propósito general, capaz de simular una Máquina de Turing Universal. Esto era necesario porque el sistema auto replicable debía ser capaz de leer instrucciones para realizar cálculos complejos. (2) Capacidad de construir, para auto replicarse, una máquina debe ser capaz de manipular información, energía y materiales de la misma clase de la cual está compuesta. (3) Construcción Universal, implica la capacidad de construir cualquiera de las máquinas de tamaño finito que pueda ser formada a partir de diferentes tipos de piezas especificadas en un conjunto de instrucciones, dado un número finito de diferentes tipos de partes pero un suministro indefinidamente grande de partes de cada tipo. (4) Auto reproducción, esto es lo siguiente a la consideración anterior, desde que el constructor universal pueda ser construido a partir del conjunto de partes manufacturables. Si la máquina original está compuesta de estas partes, y el constructor universal cuenta con una descripción de sí mismo, este debe ser capaz de hacer copias de sí mismo [29, p. 191-192].

2.3. Autómata Celular

“One is free to think of CA as being nothing more than formal idealizations of partial differential equations.”

“CA are fundamentally the simplest mathematical representations of much broader class of complex systems.”

“CA are discrete information processors.”

“CA is a discrete dynamical system simulator.”

— Andrew Ilachinski, *Cellular Automata: A discrete universe*

La naturaleza no lineal presente en los sistemas dinámicos implican para su estudio el uso de ecuaciones diferenciales parciales (EDPs), las cuales son capaces de representar en gran medida el funcionamiento de dichos sistemas con el paso del tiempo. A este propósito, dichos sistemas pueden tornarse complejos cuando son pensados como la agrupación de muchos componentes, los cuales interactúan de forma local entre ellos y el medio; y por lo general en estos sistemas aparecen comportamientos difíciles de explicar, así mismo, el estudio de este tipo de sistemas se ha tornado una tarea atractiva para la comunidad científica [16]. Por consiguiente, para estudiar el comportamiento de estos sistemas se propone dar solución a sus respectivas EDPs, ya sea de forma analítica o por medio de un método numérico. La verdad es que en algunos casos, llevar a cabo el análisis matemático es muy complicado y no siempre existe la solución analítica para dichas ecuaciones [63, p. 4]. En este orden de ideas se han planteado en la historia varios algoritmos para dar solución numérica a estas EDPs tales como, *diferencias finitas*, *elemento finito*, *volumen finito*, *autómatas celulares*, *lattice gas* y el *método de Monte-Carlo*.

Un autómata celular (AC) es entonces, un modelo matemático discreto en espacio y tiempo de un sistema dinámico⁶. Este surge con la necesidad

⁶Es decir, un sistema que evoluciona con el paso del tiempo o de manera análoga un sistema cuyo estado cambia con el transcurso del tiempo.

de analizar y estudiar de forma simplificada el comportamiento que exhibe el gran número de sistemas complejos⁷ presentes en el mundo real, cuyo comportamiento en la mayoría de los casos es posible evidenciarlo por medio de una simulación o en otras palabras haciendo que las partes del sistema interactúen entre si, por una determinada cantidad de tiempo. De aquí que se tenga la posibilidad de estudiar el comportamiento del sistema de forma detallada y al mismo tiempo sea posible obtener la solución numérica del sistema en cuestión (si existe). “Los sistemas físicos que se componen de muchos elementos discretos con interacciones locales, a menudo se modelan convenientemente como autómatas celulares. Además, cualquier sistema físico que satisfaga ecuaciones diferenciales puede aproximarse como un autómata celular introduciendo diferencias finitas y variables discretas” [62, p. 6].

De manera formal, un AC es un arreglo n -dimensional de *máquinas de estados finito* (MEF) idénticas organizadas en un conjunto de celdas de forma regular⁸, como se observa en la Figura 2.1a. Cada celda encapsula una porción del estado del sistema completo, es decir cada una de estas puede representar (el estado de) una molécula en un sistema químico, una célula en un sistema biológico, una partícula de agua en un fluido, etc. La representación que se le da a cada celda depende meramente de la aplicación o el sistema que se esté modelando. Normalmente, la población de celdas es homogénea, es decir, cada una encapsula una porción igual del estado del sistema [65, p. 1]. En cada unidad de tiempo discreto, cada celda lee el estado de las celdas vecinas y consecuentemente cambia su propio estado [32, p. 404] [29, p. 192]. En la actualidad existe gran variedad de modelos de autómatas celulares, adaptados para diferentes tipos de aplicaciones. En general la mayoría de los modelos de AC poseen las siguientes propiedades.

2.3.1. Espacio celular

Normalmente un sistema es representado en una grilla de celdas, distribuidas en una, dos o tres dimensiones a lo cual se le denomina espacio celular o espacio de celdas. Este espacio en general puede tomar uno de los tipos de configuración descritos en la Figura 2.4 dependiendo de la naturaleza del sistema. La configuración más común es la *cuadrada* aplicada a modelos de *crecimiento poblacional* [39], *propagación de incendios forestales* [47], y *tráfico vehicular* [41], por nombrar unos pocos. En la literatura es posible encontrar generalizaciones de los modelos anteriormente mencionados, por

⁷Un sistema complejo significa cualquier sistema dinámico que consiste de partes que interactúan normalmente en una forma no lineal [36, p. 1].

⁸Gran parte de la literatura, usa el termino *celda* en lugar de *Máquina de Estado Finito* para referirse al elemento que encapsula una parte del estado del sistema completo.

ejemplo el modelo de *propagación de incendios forestales* [27] y el modelo de *crecimiento de copos de nieve* [49] [10] especificados en una configuración *hexagonal*. Otro tipo de configuración es la *triangular*, que es usualmente usada para modelar problemas de *dinámica de fluidos* y *lattice gas automatas*, se debe agregar que estas no son las únicas configuraciones posibles. De hecho, no sería equivocado decir que el número de configuraciones posibles se encuentra atado a la imaginación.

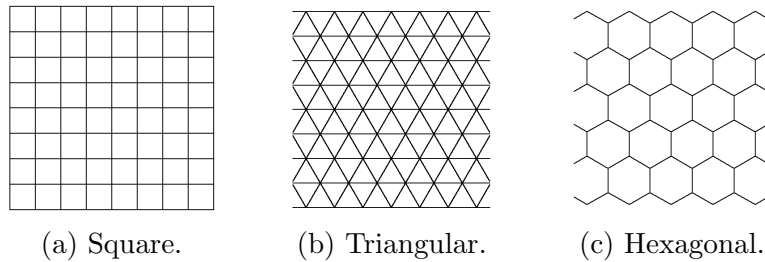


Figura 2.4: Ejemplos de configuraciones del espacio celular.

2.3.2. Interacciones locales

Es la forma en que cada celda del espacio celular interactúa con sus vecinos. La región que comprende una celda candidata y sus vecinos es denominada *vecindario*. El vecindario de una celda candidata puede ser afectado por el tipo de configuración que adquiera el espacio celular y el tipo de sistema que se desea modelar. La Figura 2.5 describe los tipos más comunes de vecindario. Así mismo, La Figura 2.6 describe algunas de sus generalizaciones.

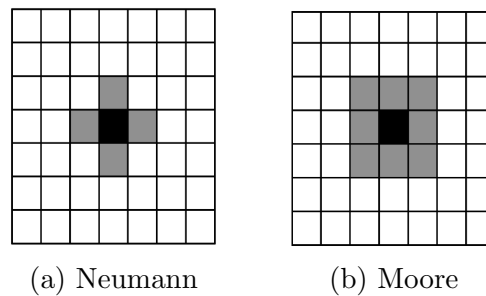


Figura 2.5: Tipos de vecindario en una configuración cuadrada [17], como se puede observar, las interacciones entre las partes de un sistema modelado con AC dependen del tipo de vecindario que represente el patrón de comunicación entre las mismas. Los vecindarios mas comunes son el de von Neumann y el de Edgar Moore.

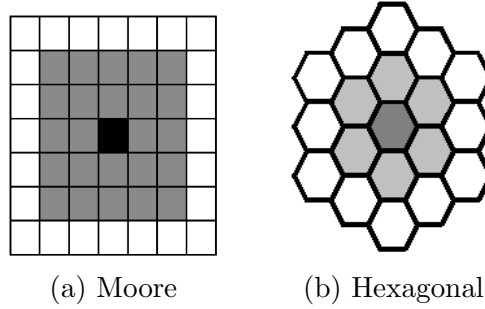


Figura 2.6: Generalizaciones en la especificación del vecindario, algunos modelos de AC incluyen una variable *radio* la cual representa la longitud del vecindario de una celda candidata, como se observa en la Figura 2.6a. Otros dependiendo de la configuración del espacio celular, modifican el patrón de interacción entre las partes del sistema como se muestra en la Figura 2.6b.

2.3.3. Estados discretos

Cada celda (o MEF) de un modelo de AC posee un estado (o valor) en un instante de tiempo discreto determinado. De manera análoga, el conjunto de estados representa los valores posibles que pueden tomar las partes constituyentes de un determinado sistema. Normalmente el conjunto de estados que puede tomar cada una de las partes es finito, sin embargo, cuando se trata con problemas cuyas partes constitutivas son de naturaleza continua, el número de estados posibles se torna infinito. Por ejemplo, el caso de la temperatura en un modelo de cambio climático.

2.3.4. Dinámica discreta

En cada unidad discreta de tiempo, cada celda (o MEF) del espacio celular es actualizada mediante un *función de transición*⁹. Esta función toma como entrada el vecindario de una celda candidata, incluyendo la misma. Luego, determina el nuevo estado de la celda candidata en función de los estados de cada celda en el vecindario. Una de las problemáticas estudiadas en el mundo de los AC es: Dado un sistema complejo, con sus respectivas EDPs, determinar un conjunto de reglas básicas o función de transición que permitan emular de forma computacional el comportamiento del sistema. El conjunto de reglas básicas puede ser pensado como el algoritmo o serie de pasos que debe seguir cada componente del sistema para interactuar con sus partes (micro dinámica), y por consiguiente, de esta interacción resulte el

⁹ Denominada en algunas literaturas como la dinámica del sistema. En física como la micro dinámica de sistema.

comportamiento global del sistema (macro dinámica). A manera de ejemplo, para modelar el comportamiento de los gases, se han descubierto reglas simples que preservan la energía y cantidad de movimiento de una partícula de gas, cuando esta colisiona con otras partículas, estos modelos se conocen como *Lattice Gas Automata*. Como es natural, para algunos sistemas ha sido posible determinar aquellas reglas simples que permiten modelarlo; para otros, el conjunto de reglas tiende a ser tan complejo que incluso con la potencia computacional con la cual se cuenta hoy en día, no es posible su simulación.

2.3.5. Condiciones de Frontera

“En la definición de autómata celular, generalmente se requiere que el espacio celular sea infinito en todas las dimensiones. Para consideraciones de computabilidad y complejidad, esto es razonable y necesario. Pero, por supuesto, es imposible simular un espacio celular verdaderamente infinito en un computadora. Por lo tanto, debemos prescribir algunas condiciones de contorno. Otra razón para usar ciertas condiciones de frontera puede ser que el sistema que queremos simular tiene ciertos límites naturales” [1]. Claramente, un sitio que pertenece al límite del enrejado no tiene el mismo vecindario que otros sitios internos. Para definir el comportamiento de estos sitios, se puede considerar una regla de evolución diferente, que ve el vecindario apropiado.

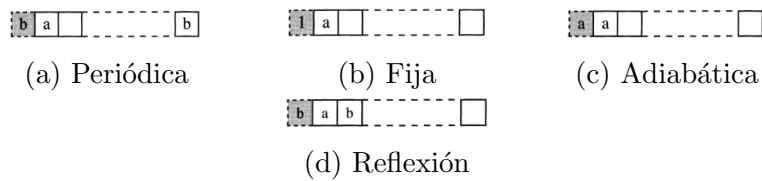


Figura 2.7: Condiciones de frontera ilustradas para un espacio celular unidimensional [23, p. 16]. La condición de límite *periódica*, supone que el lado izquierdo y el derecho están conectados formando una topología de toro o toroide (en geometría); La condición de límite *fija*, complementa el vecino faltante de las celdas izquierda y derecha con un valor fijo; El límite *adiabático*, se obtiene duplicando el valor de la celda. Por otro lado, la condición de límite *reflexión*, duplica el valor de la celda vecina.

2.4. Clasificación

Llegado este punto, es necesario considerar las formas en las cuales han sido clasificados los autómatas celulares en la historia. En vista de la gran cantidad de generalizaciones vistas en los modelos de AC, se hace necesario puntualizar el tipo de AC a ser tratado en el presente documento. Si echamos un vistazo a la historia, podemos observar en su contenido que gran parte de la literatura asociada hace alusión a diferentes tipos de AC sin mencionar de forma explícita a qué se debe dicha clasificación, por lo tanto, se hace necesario en esta sección brindar tanto el contexto como la relación que guardan dichas clasificaciones y los modelos de AC de forma breve. Un AC puede ser clasificado de muchas maneras, sin embargo, las clasificaciones más usadas giran en torno a sus características, *espacio celular*, *interacciones locales*, *estados discretos*, *dinámica del sistema*, y *condiciones de frontera*. Así mismo pueden ser clasificado teniendo en mente el comportamiento del sistema, lo cual corresponde a la clasificación ideada por Stephen Wolfram (la cual no será abordada en el presente documento). En lo que sigue se numeran las clasificaciones más comunes de AC.

- Clasificaciones que tienen en cuenta el espacio celular, dependiendo del tamaño un AC se puede caracterizar por tener una, dos o tres dimensiones. Así mismo, por el tipo de configuración del mismo, se tienen celdas ordenadas de forma *cuadrada o regular*, *hexagonal* y/o *triangular*.
- Clasificaciones que dependen del conjunto de estados, la teoría de autómatas celulares plantea los valores de cada celda del espacio celular como *discretos*, sin embargo, existen variaciones de AC donde cada uno de los estados toma valores *continuos*.
- Clasificaciones que dependen de la función de transición ó dinámica del sistema, por lo general cuando la función de transición del sistema toma componentes estocásticos para su tratamiento, o el modelo en general en su composición toma dichos componentes se dice que el modelo de AC es *estocástico* o *probabilista*. Por otro lado, se puede definir cierta clase de reglas que permitan al modelo de AC ser *reversible*, es decir, dado una configuración global del sistema en el instante $t + 1$ el autómata tiene la capacidad de volver a la configuración anterior en el instante t (en pocas palabras regresar a estados anteriores) [63, p. 273]. También, cuando el conjunto de reglas depende de la suma de los estados del vecindario se dice que el modelo de AC es *totalista*. Otra de las variaciones y no menos importante, es la forma en la cual es

aplicada la función de transición en cada una de las celdas (o MEFs) del espacio celular, en principio un autómata celular debe aplicar la misma función de transición a cada una de las celdas del espacio celular, a esta propiedad se le denomina homogeneidad, y por tanto se dice que el modelo de AC es *homogéneo*. Más aun, es posible encontrar un modelo en el cual las celdas siguen reglas diferentes, a este tipo de modelos se les denomina *no homogéneos*.

Dado lo anterior, se especifica un **autómata celular de propósito general** como un modelo con las siguientes características: *espacio celular*, cuadrado, hexagonal o triangular, esto es posible dado que las dos últimas configuraciones se pueden derivar de la primera, con una sutil modificación en la forma de interacción o vecindario de las celdas; *interacciones locales*, definidas a un radio de $r = 1$ que comprende la longitud o extensión del vecindario al rededor de una celda candidata dada; *estados discretos*, para este caso, los estados pueden tomar adicionalmente valores continuos; *dinámica del sistema* o también denominada *función de transición* deber ser homogénea; y *condiciones de frontera*, se pueden aplicar las variaciones descritas en la sección anterior.

2.5. Aplicaciones

Como se menciona en la Sección 2.3, los autómatas celulares o bien son **simuladores discretos de sistemas dinámicos** o **idealizaciones formales de ecuaciones diferenciales parciales**. Es por ello que en esta sección se pretende evidenciar la versatilidad que proveen los autómatas celulares en ambos escenarios. El primer sistema considerado, es el Juego de la Vida de John Conway, el cual exhibe un comportamiento complejo a partir de una dinámica del sistema relativamente sencilla. El segundo sistema a considerar, es uno de los aportes del presente trabajo, donde se propone un modelo de automata celular para interpolación de temperaturas en el Valle Geográfico del Río Cauca, Colombia.

2.5.1. El Juego de la Vida

Es un AC bidimensional diseñado por el matemático británico John Horton Conway en 1970, usado para simular la vida, muerte y supervivencia, a través de la evolución de un patrón inicial de celdas “vivas”. La configuración inicial de estados en el espacio celular, puede producir uno de tres comportamientos después de haber evolucionado a lo largo de varias generaciones:

extinción, donde no hay celdas “con vida” en el espacio celular; estabilidad, donde todas las celdas “con vida” permanecen en el mismo estado; oscilación, donde las celdas “con vida” entran en un ciclo sin fin [28, p. 32]. La simulación de este tipo de sistemas exhibe las características y el comportamiento presente en los sistemas biológicos donde se da el fenómeno de la reproducción [53, p. 342]. Además, es considerado un modelo discreto de AC, ya que cada celda toma valores discretos 0 (con vida) o 1 (sin vida).

La dinámica del sistema o función de transición se basa en la verificación de vida, muerte y sobre población, presente en cada vecindario del espacio celular. Cada celda se actualiza teniendo en cuenta el estado actual de sus ocho vecinos, siguiendo el patrón de vecindad de Moore, Figura 2.5b, sobre un espacio celular cuadrado. Éste sistema en particular, ha sido referenciado en gran cantidad de estudios, dada la riqueza de patrones generados en este a partir de reglas sencillas. Otro aspecto importante a considerar de este sistema es la flexibilidad que provee al permitir diferentes tipos de configuración y condiciones de frontera. La Figura 2.8, muestra continuación el conjunto de reglas básicas que describe la micro dinámica del sistema.

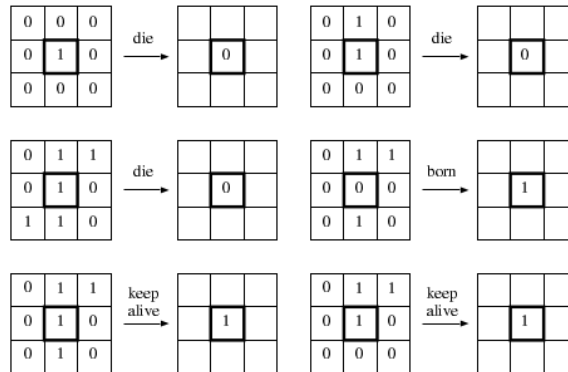


Figura 2.8: Función de transición del Juego de la Vida [30, p. 12], la función de transición consiste de cuatro reglas básicas: (Muerte) Una celda con vida, muere si tiene menos de dos o más de tres vecinos con vida; (Nacimiento) Una celda sin vida, llega a tener vida si tiene exactamente tres vecinos vivos; (Supervivencia) Una celda se mantiene con vida si tiene exactamente dos o tres vecinos con vida; (Permanece sin vida) Una celda sin vida, permanece en ese mismo estado si tiene menos de tres o más de tres vecinos sin vida.

Además de los diferentes comportamientos que emergen del sistema a partir de una reglas simples, es interesante desde un punto de vista teórico dado que el Juego de la Vida posee el poder computacional de una *Máquina*

de Turing Universal, es decir todo lo que puede ser computado en de forma algorítmica, puede ser computado por el Juego de la Vida [58, p. 54]. La demostración de este principio se encuentra en la siguiente literatura [50, p. 107-110].

2.5.2. Solución Numérica de la Ecuación de Calor: Propuesta de un Modelo de Interpolación de Temperaturas para el Valle Geográfico del Río Cauca

La agricultura es una actividad económica que depende en gran medida del estado de la atmósfera y el clima para producir los alimentos necesarios para mantener la vida humana. El cambio climático puede afectar la agricultura de varias maneras, incluso a través de cambios en la temperatura promedio, precipitación y cambios climáticos extremos [64]. Por consiguiente, se hace necesario el contar con los mecanismos adecuados para estudiar de cierta forma el cambio climático y de la misma manera poder generar predicciones que a su vez, permitan llevar a cabo un proceso de agricultura inteligente, con el objetivo de evitar en gran medida la pérdida de alimentos debido al cambio climático.

Hoy en día, específicamente en el Valle del Cauca se cuenta con estaciones climáticas, cuya instrumentación permite medir la temperatura de los lugares en los cuales fueron ubicadas. Estas de alguna forma pueden dar al agricultor un indicio de cómo esta variando la temperatura en la zona en la cual se encuentra ubicado, esto, si su zona agrícola es cercana a una o varias de las estaciones climáticas. En este orden de ideas, ¿Qué sucede con las zonas considerablemente retiradas de las estaciones?, en primera instancia, se podría pensar que aquellas zonas retiradas, tomarán datos de temperatura de las estaciones más cercanas. Sin embargo, no sería correcto hacer este tipo de suposiciones dado que la temperatura puede variar mucho entre puntos que se encuentran considerablemente alejados.

En lo que corresponde a esta sección, se pretende presentar un modelo de autómatas celulares, el cual permite determinar de forma numérica la temperatura en cada punto definido en el plano o *área de interés*, Figura 2.9b. Esto implica llegar a la solución numérica de la *ecuación de Fourier*, que describe el proceso de transferencia entre los puntos definidos en el plano. A continuación, la Figura 2.9 describe el área de estudio, adicionalmente se planea el *área de interés* (color pastel), de la cual se desea determinar el perfil de temperaturas. Seguido a esto, se describe la solución numérica de la *ecuación de Fourier* y cómo ésta es llevada al modelo de autómatas celulares.



Figura 2.9: Zona de Estudio y Área de Interés. Se estudió el comportamiento de temperaturas en la zona corresponde al valle geográfico del Río Cauca, es decir, la región comprendida entre Santander de Quilichao y Cartago. El área comprendida por el área de interés es de aproximadamente $15440,31 \text{ km}^2$, es decir $204,89 \text{ km}$ de alto por $75,36 \text{ km}$ de ancho.

La Ecuación del Calor de Fourier

La ecuación de Fourier, analiza el fenómeno de transferencia de calor en el interior de cuerpos sólidos en los cuales se da el proceso de transferencia de calor por *conducción*. Este último, es el fenómeno de transferencia térmica entre moléculas vecinas debido a un gradiente térmico y cuyo modelo matemático consiste de una ecuación diferencial parcial de segundo orden, Ecuación 2.1 [21, p. 54] [35, p. 1]. Para este caso en particular, se considera la versión bidimensional de la ecuación, donde $T(x, y, t)$ representa la temperatura en cada punto del interior del plano en cada instante de tiempo y α es una constante que depende del material [35, p. 5].

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \quad (2.1)$$

Dado que, en nuestro caso de estudio, el cambio de temperatura se presenta en un fluido (el aire), y se desea estudiar a una altura de 960 ± 50 metros sobre el nivel del mar, así mismo debe corresponder a la altura de un grupo de *estaciones climáticas de referencia*, de las cuales se conocen datos de temperatura. Para considerar conducción en el caso de los fluidos, es ne-

cesario poder despreciar su movimiento [21, p. 54]. Es oportuno mencionar en este punto que el sistema considerado no depende del tiempo, dado que el respectivo análisis se llevará a cabo sobre una fotografía del mismo, es decir, en principio el sistema estará en movimiento y en un instante determinado, se hace una captura de su estado, en pocas palabras, se hace un censo de la temperatura en ese instante de tiempo, para cada una de las *estaciones climáticas de referencia*. Esto se traduce a que el sistema en consideración está en estado estacionario, por lo tanto la temperatura de las estaciones no va a cambiar en el tiempo. Como consecuencia, la expresión que depende del tiempo en la Ecuación 2.1 se vuelve cero, luego la ecuación queda expresada como se muestra en Ecuación 2.2, donde la expresión resultante se conoce como la *ecuación de Laplace*, la cual puede ser resuelta de forma numérica haciendo una aproximación de esta en *diferencias finitas*.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \quad (2.2)$$

Diferencias Finitas

La Ecuación 2.2 puede ser tratada de forma numérica, si la expresamos de forma aproximada en diferencias finitas. El método de diferencias finitas es una aproximación de las derivadas parciales en una ecuación física por “diferencia entre” nodos separados por una distancia finita, una de sus características es que las ecuaciones diferenciales parciales son sustituidas por un conjunto de ecuaciones algebraicas para cada nodo. La solución para valores nodales de temperatura se obtiene por iteración o inversión de la matriz [3, p. 3]. Para aproximar a diferencias finitas, el sistema en consideración, Figura 2.9b, debe ser parcelado, Figura 2.10a. El sistema a considerar cuenta con las siguientes dimensiones, altura $y = 204,89km$ y ancho $x = 75,36km$. Para simplificar el tratamiento de la ecuación de Laplace en diferencias finitas, se parcela el sistema de tal forma que $\Delta x = \Delta y = 0,15km$, lo anterior como resultado de haber dividido la dimensión en x entre 500 puntos nodales y la dimensión en y entre 1397 puntos nodales.

Dado el paso de discretización o parcelado del sistema, ilustrado en la Figura 2.10a, y un primer perfil de temperaturas o **estado inicial del sistema**, Figura 2.10b. Se describe a continuación el paso de la ecuación de Laplace a diferencias finitas. La primera derivada en x e y de la temperatura se aproxima de la siguiente forma.

$$\frac{\partial T}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{T(x + \Delta x) - T(x)}{\Delta x} \quad (2.3)$$

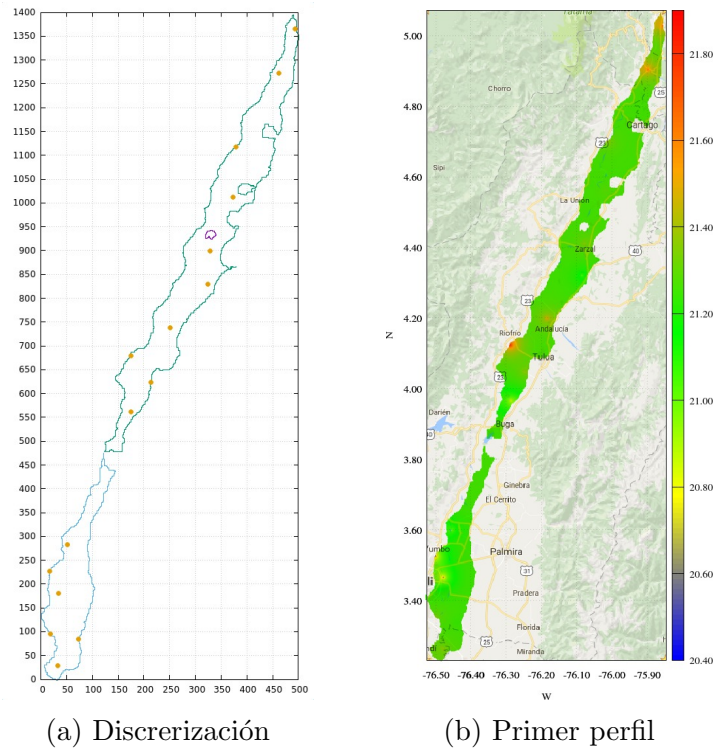


Figura 2.10: Discretización y Primer Perfil de Temperaturas. (a) Ejemplo de discretización del área de interés, Figura 2.9b, note que este es un ejemplo el cual no conserva el número de celdas especificado tanto para la dimensión x como para y dado que la cantidad de celdas real impide ver en detalle el modelo. Sin embargo es útil para fines ilustrativos. (b) se presenta una fotografía del sistema o como primer perfil de temperaturas.

$$\frac{\partial T}{\partial x} \approx \lim_{\Delta x \rightarrow 0} \frac{T(x + \Delta x) - T(x)}{\Delta x} \quad (2.4)$$

$$x = i * \Delta x; \quad \text{donde} \quad i = 0, 1, 2, 3, 4... \quad (2.5)$$

$$T(x) = T(i * \Delta x) = T_i \quad (2.6)$$

$$\frac{\partial T}{\partial x} \approx \lim_{\Delta x \rightarrow 0} \frac{T_{i+1} - T_i}{\Delta x} \quad (2.7)$$

De lo anterior, podemos aproximar la segunda derivada usando de igual forma la definición de la derivada, la cual queda en términos de la primera derivada que fue calculada en el paso anterior, Ecuación 2.7.

$$\frac{\partial^2 T}{\partial x^2} = \lim_{\Delta x \rightarrow 0} \frac{\frac{\partial T}{\partial x}|_{x+\Delta x} + \frac{\partial T}{\partial x}|_x}{\Delta x} \quad (2.8)$$

$$\frac{\partial^2 T}{\partial x^2} \approx \lim_{\Delta x \rightarrow 0} \frac{\frac{T_{i+1} - T_i}{\Delta x} + \frac{T_i - T_{i-1}}{\Delta x}}{\Delta x} \quad (2.9)$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} + T_{i-1} - 2T_i}{(\Delta x)^2} \quad (2.10)$$

Luego de encontrar la aproximación en diferencias finitas de la segunda derivada, Ecuación 2.10, procedemos a reemplazar los términos en la ecuación de Laplace, Ecuación 2.2, que se tenía inicialmente.

$$0 = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{T_{i+1,j} + T_{i-1,j} - 2T_{i,j}}{(\Delta x)^2} + \frac{T_{i,j+1} + T_{i,j-1} - 2T_{i,j}}{(\Delta y)^2} \quad (2.11)$$

$$\Delta x = \Delta y = h \quad (2.12)$$

$$0 = \frac{1}{h^2} [T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j}] \quad (2.13)$$

Finalmente se obtiene una ecuación algebraica, la cual nos permitirá encontrar la temperatura para cada punto nodal del área de interés en función de los nodos vecinos.

$$T_{i,j} = \frac{1}{4} [T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}] \quad (2.14)$$

Método de Relajación con Autómatas Celulares

El **método de relajación** es un método iterativo usado para resolver sistemas de ecuaciones, incluyendo sistemas no lineales. Al ser un método iterativo, genera una serie de aproximaciones donde la n -ésima aproximación deriva de la anterior. Además se debe cumplir que cada iteración se encuentra mas cerca a la solución. Dado lo anterior, se dice que el sistema converge. La implementación de este tipo de métodos requiere un **criterio de terminación**, para determinar en qué momento el sistema llega a un estado de convergencia. En general, las características de este problema sugieren un método el cual permita evolucionar el sistemas en busca de una solución, dada una **dinámica del sistema** descrita en la Ecuación 2.14, unas condiciones iniciales de temperatura, Figura 2.10b , y un patrón de comunicación entre las partes (nodos), lo cual podría llamarse **vecindario**, implícito en la ecuación que describe la dinámica del sistema. De esta forma, la naturaleza del problema sugiere sea abordado con el método de **autómata celular**.

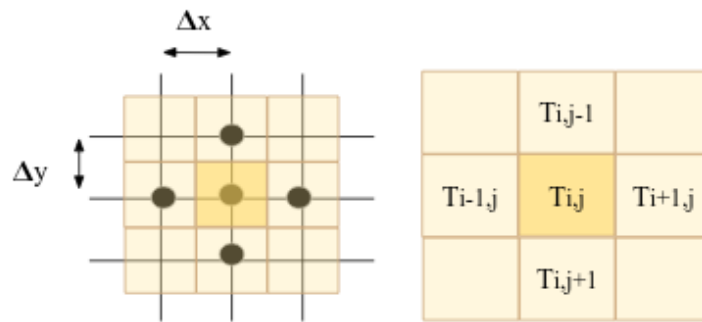


Figura 2.11: Equivalencia entre Nodos y celdas.

Para ello, debemos puntualizar el concepto de **espacio celular** y **condiciones de frontera**. En primera instancia, calcular la temperatura para cada nodo en el área de interés, es equivalente a calcular la temperatura de cada celda en un espacio celular, como se observa en la Figura 2.11. Por razones de simplicidad nos referiremos a estos puntos nodales como **celdas**. En este orden de ideas nuestro sistema resulta ser un espacio celular de 500 x 1397 celdas. Un punto importante a tener en cuenta, es que solo nos interesa saber cómo es el cambio de temperatura en la región comprendida en el borde azul de la Figura 2.10a (es decir, la región donde residen las estaciones climáticas de referencia). La región azul corresponde en este caso a la frontera de nuestro problema, donde la temperatura va ser equivalente a cero, lo cual corresponde a una **condición de frontera fija**. Finalmente, nuestro

modelo necesita una condición de parada la cual está dada por la siguiente Ecuación 2.15.

$$\sqrt{\sum_{i,j} (Tb_{i,j} - Ta_{i,j})^2} \leq 0,0000001 \quad (2.15)$$

$Tb_{i,j}$, representa la temperatura en la celda i, j del espacio celular en la iteración anterior del método, mientras que $Ta_{i,j}$ describe la temperatura de dicha celda en la iteración actual. En resumen, esta ecuación nos dice que cuando el cambio de temperatura en el estado actual del sistema $Ta_{i,j}$ respecto a su estado anterior $Tb_{i,j}$ sea muy pequeño, la evolución o iteración del sistema deber terminar. Después de haber evolucionado el sistema, se puede obtener los siguientes perfiles de temperatura, Figura 2.12. De igual forma para concluir, se muestra la gráfica, Figura 2.13, que evidencia el comportamiento de temperatura del sistema durante cien mil iteraciones.

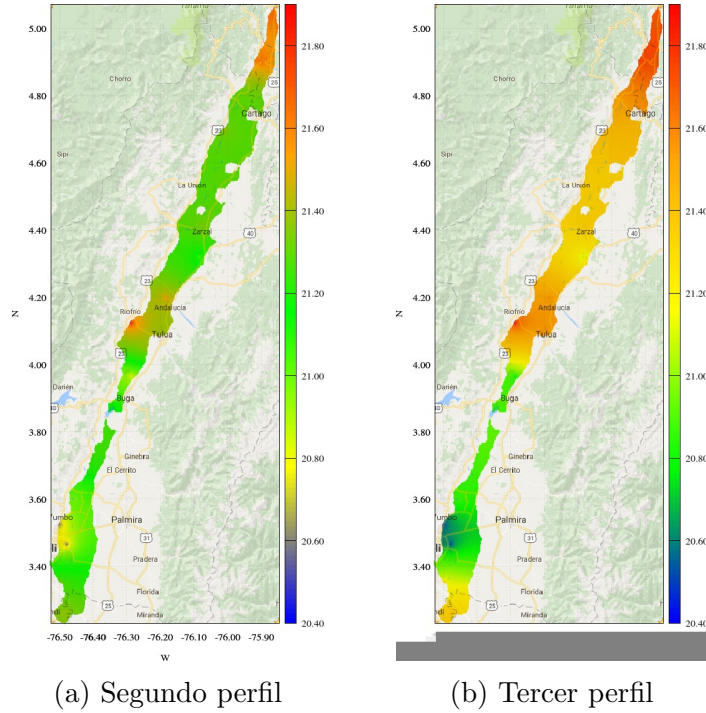
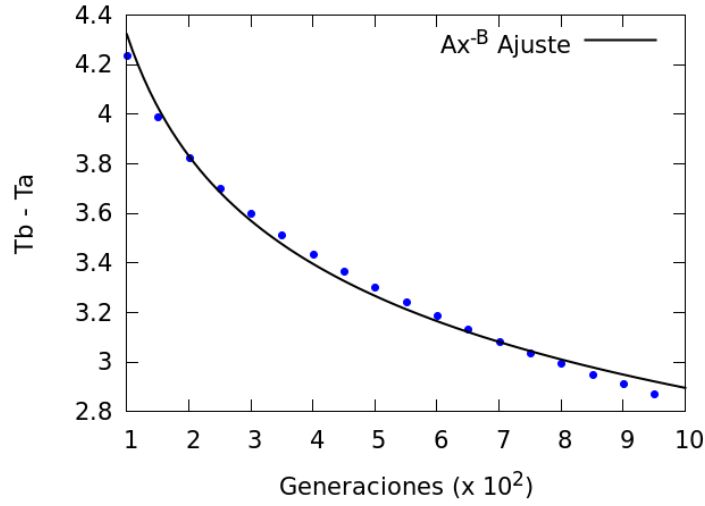
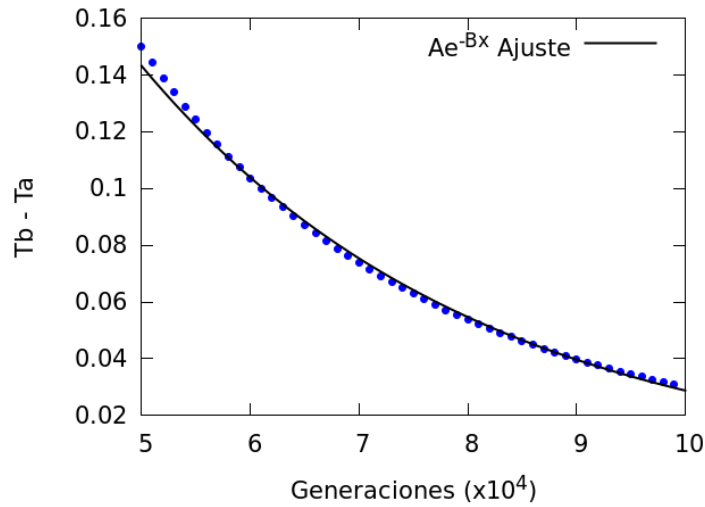


Figura 2.12: Perfiles de Temperatura Obtenidos.



(a) Convergencia Algebraica en las Primeras 1000 Generaciones.



(b) Convergencia Exponencial desde 50000 hasta 100000.

Figura 2.13: Gráfica Convergencia del Sistema. La gráfica describe el cambio de temperatura en el sistema durante 100000 Generaciones. Según se puede observar, en las primeras iteraciones el cambio temperatura del sistema es muy elevado, lo que conlleva a que la convergencia del sistema sea lenta al principio, Figura 2.13a. Luego, al aproximarse el sistema a un estado estable (bajo cambio de temperatura), la convergencia del sistema tiende a ser exponencial, Figura 2.13b.

Capítulo 3

Computación Paralela

En la última década y media las arquitecturas de computadores pasaron de ser sistemas mono procesador (o mono-núcleo) a sistemas multinúcleo con el propósito de sobreponerse a las limitaciones físicas que se imponían al aumentar el número de transistores sobre una sola unidad de procesamiento. Se incrementa entonces el número de núcleos por procesador y se disminuye la frecuencia del reloj con el fin de disminuir el consumo de energía y la generación de calor. Este cambio en el diseño de los procesadores causa que aplicaciones legadas ejecutadas bajo las nuevas arquitecturas tengan un desempeño peor que en procesadores anteriores o mono-núcleo. A este propósito, vale la pena preguntarse ¿Cómo resolver este problema?.

Las nuevas arquitecturas requieren entonces un repensarse las soluciones algorítmicas que tradicionalmente se llevan a cabo. El pensamiento algorítmico es en su esencia mono-hilo, pero este tipo de solución algorítmica no explota el potencial de las arquitecturas multinúcleo. En los últimos años ha tomado fuerza el desarrollo de soluciones basadas en técnicas de *programación en paralelo*, a razón de que el crecimiento en el poder computacional ha seguido el camino del paralelismo. De esta manera, el paralelismo ha permitido considerar la solución de problemas que en tiempos anteriores fueron considerados intratables. Hoy en día es posible tratar problemas de Modelamiento de Sistemas Físicos, Aprendizaje Profundo, Análisis de Datos, entre otros.

En lo que corresponde al presente capítulo, nos adentraremos en la historia de los computadores secuenciales, así mismo, se presentarán aquellos cambios sugeridos sobre éstos para llegar a lo que conocemos como **computadores paralelos**. El siguiente aspecto considerado, es la forma en la cual podemos acceder al poder computacional presente en los computadores paralelos, y es en este punto donde aparecen los **modelos de programación paralela**. Para finalizar, se introduce el concepto de **patrón de progra-**

mación paralela, una estrategia de programación para hacer que ciertos algoritmos y/o problemas en programación sean fácilmente paralelizables.

3.1. Computadores Paralelos

Por lo general, gran parte de los computadores paralelos que existen en la actualidad parten de los cambios llevados a cabo sobre la arquitectura planteada por von Neumann en la década de 1940. Esta arquitectura propone un grupo de componentes básicos para la construcción de computadores: memoria principal, unidad de procesamiento, dispositivos de entrada/salida, interconexión entre la memoria principal y la CPU como se muestra en la Figura 3.1. La memoria principal se compone de una colección de ubicaciones, cada una de las cuales es capaz de almacenar tanto instrucciones como datos de un programa. Cada ubicación consta de una dirección, que se utiliza para acceder su contenido¹. Por otro lado, la unidad de procesamiento central se divide en unidad de control y unidad lógica aritmética (ALU). La unidad de control es responsable de decidir qué instrucciones de un programa deben ser ejecutadas, y la ALU se encarga de la ejecución de instrucciones. La información acerca del estado de un programa en ejecución se mantiene en los registros, ubicados al interior de la CPU. Las instrucciones y los datos se transfieren entre la CPU y la memoria a través de una interconexión.

La idea principal de la arquitectura de Von Neumann es la distinción que se hace entre la memoria principal que contiene las instrucciones del programa y la CPU que lee y ejecuta cada instrucción. Esta división entre ambos componentes ha permitido a lo largo del tiempo realizar cambios en la estructura básica propuesta para optimizar, por un lado, la cantidad de procesadores dispuestos en un solo chip de procesador, lo que abre el camino a la creación de computadores paralelos. Por otro lado, estos cambios han permitido mejoras en aspectos relacionados con la lectura de instrucciones de la memoria principal.

En las últimas tres décadas del siglo XX, como resultado de los cambios realizados en la arquitectura de von Neumann, se introdujo una gran variedad de arquitecturas de computadores para el procesamiento paralelo. Así mismo, la proliferación de estas nuevas tecnologías han traído consigo nuevas tecnologías de hardware. En este orden de ideas, el campo de la informática se enfrenta a la dificultad de comprender cómo funcionan estas nuevas arquitecturas, cuáles son las diferencias entre ellas y cómo se relacionan con el paralelismo. Para ello, Michael J. Flynn en 1966 idea una taxonomía para clasificar sistemas de cómputo con capacidades de procesamiento paralelo. De

¹Datos o instrucciones

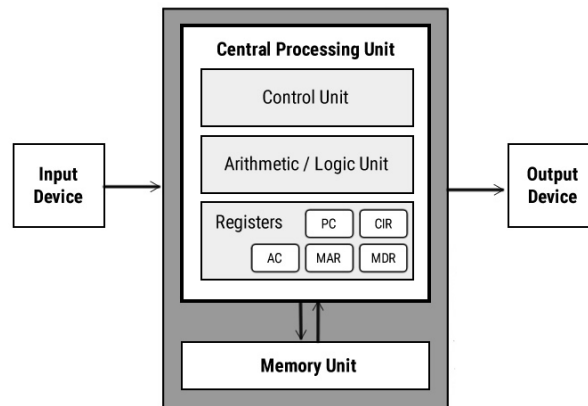


Figura 3.1: Arquitectura de Von Neumann [19], la CPU interactúa con la memoria principal y el sistema de entrada/salida, durante la ejecución de un programa el cual debe estar almacenado en la memoria principal, siguiendo el *concepto de programa almacenado* de Neumann al cual se debe la arquitectura mencionada.

esta taxonomía resultan 4 tipos de computadores paralelos, *Single Instruction, Single Data (SISD) Stream*; *Single Instruction, Multiple Data (SIMD) Stream*; *Multiple Instruction, Single Data (MISD) Stream*; y *Multiple Instruction, Multiple Data (MIMD) Stream* [55, p. 613].

Flynn basó su taxonomía teniendo en mente la cantidad de *flujos de instrucciones*² y *flujos de datos*³ manipulados por este tipo de máquinas de forma simultánea en el transcurso del tiempo. De esta manera, en computadores SISD, Figura 3.2a, un único flujo de datos está siendo procesado por un único flujo de instrucciones en el tiempo; en los SIMD, Figura 3.2b, diferentes flujos de datos están siendo procesados por único flujo de instrucciones en el tiempo; por otro lado, en los MISD, Figura 3.2c, un único flujo de datos está siendo procesado por varios flujos de instrucciones; finalmente en MIMD, 3.2d, múltiples flujos de datos están siendo procesados por múltiples flujos de instrucciones (o múltiples programas) en el tiempo. En la actualidad predominan las arquitecturas SIMD (GPUs, procesadores vectoriales) y MIMD (PCs, Estaciones de trabajo, Super computadores, etc).

²Secuencia de instrucciones pertenecientes a un determinado programa.

³Colección de datos sobre los que opera un determinado programa o flujo de instrucciones.

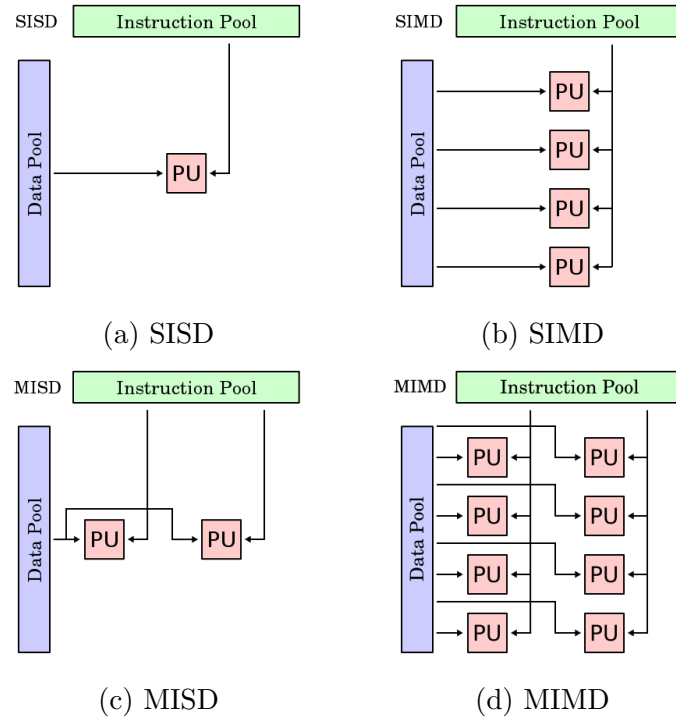


Figura 3.2: Taxonomía de Flynn [7], (a) SISD son clasificados como no paralelos, básicamente arquitecturas antiguas siguen este esquema de ejecución. (b) SIMD es el esquema de ejecución que siguen los procesadores de gráficos (GPUs). (c) MISD este tipo de esquema en realidad no es usado en la actualidad, sin embargo, se plantea su uso en escenarios donde se desee ejecutar varios programas que reciban como entrada los mismos datos, por ejemplo, varios algoritmos de criptografía intentando romper una contraseña o ejecutar varias instancias del mismo programa en paralelo sobre los mismos datos para tener redundancia y por ende tolerancia a fallos [9]. (d) MIMD es el esquema de ejecución usado por la mayoría de computadores en la actualidad, los teléfonos celulares, PCs, Clusters, Super Computadores, se encuentran en esta clasificación.

3.1.1. Computadores MIMD

Es un tipo de computador paralelo en el cual “cada procesador o núcleo puede estar trabajando en una secuencia de instrucciones diferente, así mismo, cada secuencia de instrucciones opera sobre una secuencia de datos diferente” [9]. Sin embargo, pueden haber casos en que varios procesadores cooperen en el procesamiento de una única secuencia de datos, lo que se conoce como *trabajo compartido*. Los computadores **multiprocesador** y **multinúcleo** caen en ésta categoría. Los primeros, representaron el primer paso hacia el paralelismo a nivel de procesos, además, se caracterizaron por tener varios procesadores equivalentes dispuestos en la misma placa madre, Figura 3.3a. La segunda organización de procesadores, por su parte buscó obtener mayor velocidad para la transferencia de datos entre procesadores cuando se tienen tareas compartidas, ubicando varios procesadores (núcleos) en el mismo chip⁴ de procesador, Figura 3.3c. Por último pero no menos importante, se encuentra una tecnología que se ha vuelto muy común en los procesadores multinúcleo actuales llamada *Hyper-Threading*. La tecnología *Hyper-Threading* crea dos núcleos de procesamiento virtual por cada núcleo físico presente en la CPU [20], Figura 3.3b, de esta forma un núcleo físico se comporta como dos en realidad, permitiendo de igual forma el procesamiento paralelo.

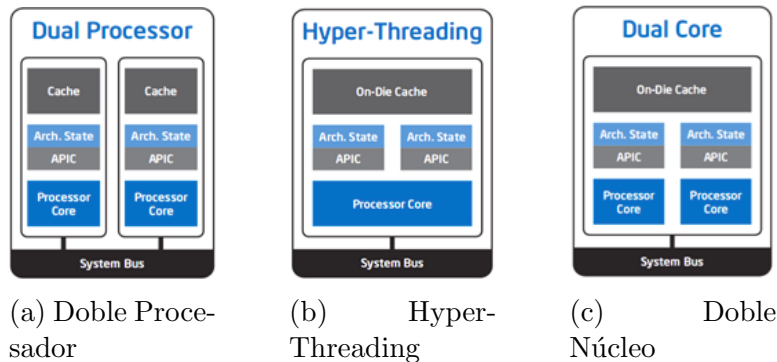


Figura 3.3: Organización de los procesadores en computadores multiprocesador y multinúcleo [15].

Cuando se habla de paralelismo, y en especial, al momento de diseñar algoritmos de procesamiento paralelo, debemos tener en consideración dos aspectos fundamentales, los cuales nos permitirá escoger el modelo de programación paralelo adecuado al momento de implementar el algoritmo. En primera instancia, la arquitectura de computador, basada en la taxonomía de

⁴Circuito integrado.

Flynn, y finalmente, cómo se llevará a cabo el acceso a los datos por entidades que pueden procesarlos de forma simultanea. La realidad es que dependiendo de la arquitectura de memoria, es posible determinar qué tan fácil es el acceso a los datos. Adicionalmente, para cada arquitectura de memoria se dispone de uno o varios *modelos de programación paralela* que abstraen el poder computacional presente en dichas arquitecturas.

Memoria compartida y memoria distribuida son ejemplos de las arquitecturas de memoria que puede seguir un sistema MIMD. Estas describen en esencia cómo un grupo de procesadores autónomos acceden a la memoria principal. En un arquitectura de memoria compartida, Figura 3.4a, una colección de procesadores autónomos están conectados a un sistema de memoria por medio de una interconexión, donde cada procesador puede acceder cada ubicación de memoria, naturalmente, los procesadores tienen la oportunidad de acceder los mismos datos. Los PCs o Estaciones de trabajo son ejemplos de este tipo de computadores. En arquitecturas de memoria distribuida, Figura 3.4b, cada procesador (probablemente multi-núcleo) está emparejado con su propia memoria privada, donde cada uno funciona de forma independiente⁵, es decir, los cambios que cada máquina hace a su memoria local no tiene efecto en la memoria de otros procesadores. Los pares procesador-memoria se comunican a través de una red de interconexión. Por lo tanto, sistemas que siguen la arquitectura de memoria distribuida se comunican explícitamente enviando mensajes por la red por medio de funciones especiales, que le permiten obtener información de la memoria de otro procesador. Vale la pena resaltar que el modelo de programación paralela OpenMPI hace honor a su nombre *Message Passing Interface*, dado que es la interfaz para llevar a cabo la tarea de paso de mensajes en este tipo de arquitecturas de memoria. Así mismo, este tipo de sistemas se caracterizan por ser una composición de sistemas de memoria compartida (PCs, Estaciones de trabajo, etc) conectados por medio de una red.

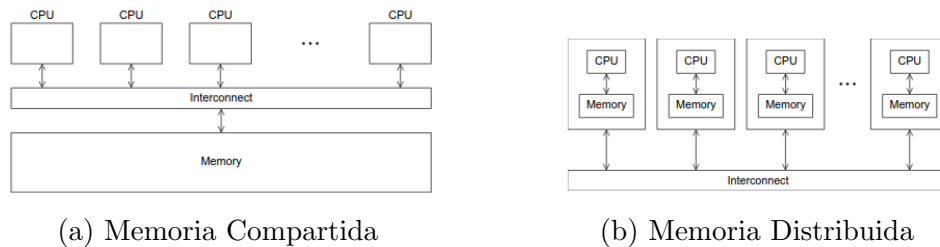


Figura 3.4: Arquitecturas de Memoria en Computadores Paralelos MIMD

⁵Por lo general cada par procesador-memoria representan un computador.

3.1.2. Computadores SIMD

A partir del año 2003, la industria de semiconductores se estableció en dos trayectorias principales para el diseño de microprocesadores. La trayectoria **multinúcleo** cuya arquitectura de memoria fue mencionada en la sección anterior, la cual en esencia busca mejorar la velocidad de ejecución de los programas siempre y cuando estos sean puestos a disposición de los múltiples núcleos. Abría que decir también que la era multinúcleo⁶ empezó entonces con procesadores *doble núcleo*, aumentando el número de núcleos con cada generación de procesadores. Un ejemplo en la actualidad es el microprocesador Intel® Xeon® Processor E5 v2 Family con 12 núcleos. Por otra parte, la trayectoria de **muchos núcleos** (o *many-core* en Inglés), comenzó con una gran cantidad de núcleos, por ejemplo el dispositivo Nvidia GeForce GT 650M con 384 CUDA⁷ cores, y una vez más, la cantidad de núcleos aumenta con cada generación de GPUs. Un ejemplo de una unidad de procesamiento de gráficos actual es el dispositivo Nvidia GTX680 con 1536 CUDA cores [38, p. 2-3].

A pesar de haber evolucionado de forma considerable el paralelismo en la trayectoria multinúcleo, el nivel de paralelismo liberado por la GPU es considerablemente abrumador respecto a su contraparte la CPU, Figura 3.5, esto de alguna manera llamó la atención en el mundo de la computación, el cual se plantea la siguiente interrogante, ¿Es posible llevar a cabo tareas de propósito general en un dispositivo de propósito particular como lo es la GPU?. Partiendo de esta posibilidad nace entonces el concepto de **Computación de Propósito General en Unidades de Procesamiento Gráfico** (GPGPU siglas en Inglés) apoyando a su vez lo que se conoce como **Computación Heterogénea**.

GPGPU y La Computación Heterogénea

Con el nacimiento de las GPUs, al mismo tiempo se desarrollaron APIs como OpenGL y DirectX para permitir al programador el uso de estos dispositivos. En principio, el objetivo principal de las GPU era realizar operaciones intensivas de punto flotante para el procesamiento de gráficos. Luego, para poder explotar el nivel de paralelismo presente en estos dispositivos en

⁶Es importante resaltar que existen tanto computadores multiprocesador como procesadores multinúcleo, su diferencia es sutil, mientras que en el primero los procesadores permanecen en chips independientes soldados a la misma placa madre, en el segundo, los procesadores, también llamados cores, residen en el mismo chip de procesador.

⁷CUDA, es una arquitectura de GPU presente en las GPUs modernas. Por lo tanto los núcleos procedentes de esta arquitectura son llamados CUDA cores y en arquitecturas anteriores cores de GPU, dado que difieren en diseño de los núcleos normales de una CPU.

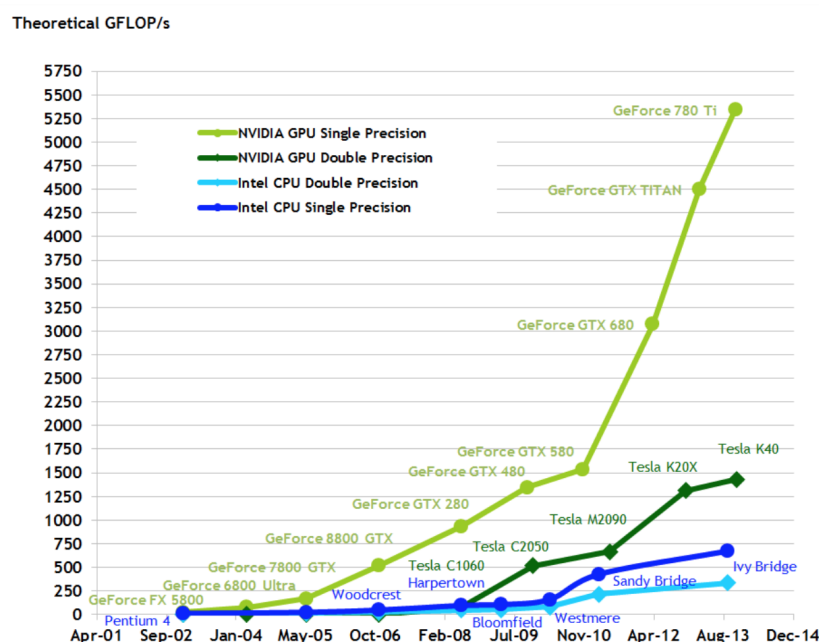


Figura 3.5: Evolución en rendimiento CPUs vs GPUs [2].

un propósito distinto, nació un concepto llamado GPGPU (*General-purpose computing on graphics processing units*), con el objetivo de realizar en GPU tareas que se ejecutan normalmente en una CPU. Sin embargo, al principio esta iniciativa no fue acogida en su totalidad por la comunidad, debido a la complejidad impuesta a los programadores para transformar algoritmos convencionales en operaciones de procesamiento de gráficos. Además en dicho tiempo, las mismas APIs limitaron el número de aplicaciones que pudieron ser abordadas con esta aproximación.

En el año 2007, con el lanzamiento de CUDA, Nvidia comenzó a implementar nuevas capacidades de hardware en sus chips para ayudar a programadores y científicos con el trabajo de escribir aplicaciones paralelas para GPUs. Esto le da al campo GPGPU un giro de 180 grados. Dado que al mismo tiempo de haber lanzado la **arquitectura CUDA**, lanzan el modelo de programación paralela CUDA, el cual expone al programador el poder computacional de las GPUs de Nvidia que cuentan con arquitectura CUDA. El modelo CUDA, entonces presenta un modelo de programación cooperativo, el cual combina el poder de dispositivos o procesadores de diferente naturaleza CPU y GPU. Este método aprovecha la capacidad del primero para llevar a cabo cálculos secuenciales y la velocidad del segundo para realizar cálculos masivos de punto flotante. Finalmente de este concepto de cooperación nace lo que se conoce hoy en día como *computación heterogénea*.

Hasta el 2012, el número de operaciones que podían soportar tanto GPU como CPU era respectivamente, 1.5 teraflops frente a 150 gigaflops. Esta diferencia en rendimiento motivó a los desarrolladores de aplicaciones a mover partes computacionalmente intensivas del programa a la GPU. Por lo regular cuanto mas trabajo hay por hacer, de cierta manera hay más oportunidad de encontrar paralelismo. Es natural preguntarse el por qué de una brecha de rendimiento tan grande entre GPU y CPU. La respuesta radica en la filosofía que prima sobre el diseño de cada procesador, como se observa en la Figura 3.6. La filosofía de diseño de la GPU esta enfocada en la realización de cargas de trabajo paralela al proporcionar memoria caché, unidades de control y aritméticas de tamaño reducido, lo que permite al diseñador colocar más unidades aritméticas para aumentar la cantidad de cálculos que pueden ser llevados a cabo de forma simultánea. Este diseño es correcto en dominios de problemas basados en el cómputo paralelo, incluyendo el procesamiento de gráficos en tiempo real, procesamiento de vídeo, análisis de imágenes médicas, dinámica molecular, simulación astrofísica y secuenciación de genes. De cierta forma el diseño de la GPU es *orientado a la cantidad*. Las CPU, por otro lado, son diseñadas para minimizar la latencia de ejecución de programas secuenciales. La complejidad envuelta tanto en la unidad de control como en la memoria caché, convierten los accesos de gran latencia a memoria en accesos de menor latencia a caché, buscando acelerar la ejecución del flujo de instrucciones. El gran tamaño de caché y unidad de control, justifican la complejidad presente en las mismas. Finalmente, se dice que este tipo de diseño es *orientado a la latencia* [38, p. 3] [18].

La Arquitectura CUDA

Como se menciona en la sección anterior, la arquitectura CUDA nace con la necesidad de explotar la capacidad de las tarjetas gráficas en un camino distinto al habitual. Vale la pena mencionar que para ello fue necesario un replanteamiento en el diseño de las mismas. En este caso en particular, el diseño de las tarjetas gráficas Nvidia fue replanteado en el camino de las GPGPU y a su vez permitir la Computación Heterogénea. La arquitectura que surge de este replanteamiento se conoce como la arquitectura CUDA. Esta arquitectura en general es un conjunto de diseños de GPU: Tesla, Fermi, Kepler, Maxwell y el más reciente Pascal, listado de forma cronológica. Cada diseño pretende de forma incremental, aumentar la cantidad de paralelismo soportado en el diseño anterior, así mismo, reducir el consumo de energía, lo cual es el principio que prima en las GPUs hoy en día, apoyando lo que se conoce como *green computing*.

La arquitectura Tesla, Figura 3.7, la cual es la base de las demás archi-

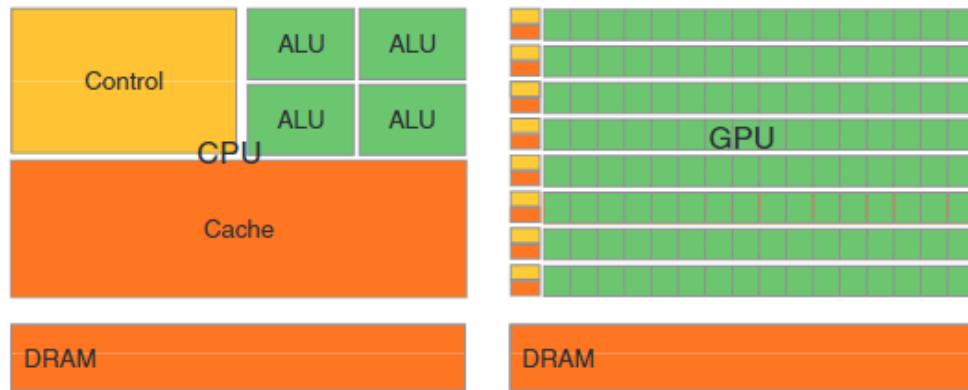


Figura 3.6: Tanto CPUs como GPUs tienen filosofías de diseño diferente, el diseño de las GPU ahorra energía y área del chip al reducir la memoria caché y la unidad de control, permitiendo al diseñador colocar mas unidades aritméticas (ALU) para tener la capacidad de realizar operaciones masivas de punto flotante. En contraste, la CPU esta optimizada para la ejecución de código secuencial. Proporciona unidad de control y memoria caché de gran tamaño para reducir la latencia de datos e instrucciones . [38, p. 4].

tecturas, se compone de un conjunto de 128 procesadores de transmisión (o *streaming processors (SPs)*). Estos SPs a su vez, son ordenados en grupos de 8 SP, este grupo de SP conforma lo que se conoce como multiprocesador de transmisión (o *streaming multiprocessor (SM)*). Finalmente estos 16 multiprocesadores de transmisión son organizados en 8 unidades de procesamiento independientes llamadas *clusters de texturas/procesadores (TCP)* [40, p. 40]. Fermi cuenta con hasta 512 SP organizados de la siguiente manera, 16 SM de 32 SP cada uno. Además ofrece mejoras en las operaciones de punto flotante para cálculos en precisión simple y doble [45]. Kepler emplea una nueva arquitectura de SM llamada “SMX”, presentando un consumo de energía de 90 % menos comparado con la arquitectura Fermi. Además, este diseño aumenta el rendimiento entregado por la GPU en cálculos de punto flotante en doble precisión [46]. Maxwell emplea un nuevo diseño de SP a partir del diseño SMX de Kepler llamado “SMM”, con el fin de reducir aún más el consumo de energía, introduce una nueva característica llamada paralelismo dinámico, y aumenta el número de SP disponibles a 640. Finalmente Pascal, el diseño más reciente, proporciona mejoras en rendimiento para aplicaciones relacionadas al Aprendizaje Profundo, Minería de Datos, Redes Neuronales, entre otros.

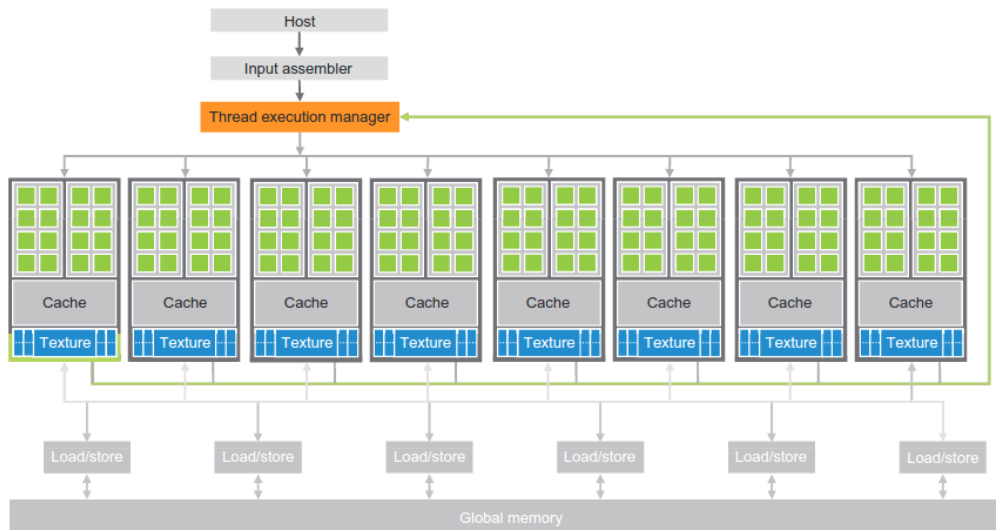


Figura 3.7: Arquitectura Básica de una GPU compatible con CUDA [38, p. 9]

3.2. Modelos de Programación Paralela

Parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel system [13].

Un modelo de programación paralela es una abstracción de las capacidades del hardware de un computador paralelo. Dado que un modelo de programación es simplemente una abstracción de la arquitectura del sistema subyacente, generalmente no está vinculado a alguna máquina en específico. Pero, un sistema en particular puede soportar más de un modelo de programación, aunque no siempre con la misma eficiencia [59, p. 3].

Los modelos de programación paralela por lo general se presentan en forma de librerías. Por lo tanto, para hacer uso de éstas, normalmente es necesario el uso de un lenguaje de programación. Por ejemplo, en el caso del lenguaje de programación C/C++, la librería PThreads, abstrae el proceso de creación de hilos de ejecución, lo cual es una capacidad del hardware de un computador paralelo, y en consecuencia los hilos de ejecución creados posteriormente podrán realizar tareas de forma simultánea. En resumen, es posible dar soporte a los modelos de programación paralela por medio de: **lenguajes de programación paralela**, **librerías optimizadas**, **directivas de compilación**, y **paralelización automática**.

- Lenguajes de Programación Paralela, lenguajes cuya estructuras de datos y sintaxis expone al programador las capacidades de un computador paralelo, ejemplo, *High Performance Fortran*.
- Librerías Optimizadas, pueden ser vistas de dos maneras:
 - Librerías que ocultan la complejidad del paralelismo, recibiendo un conjunto de datos de entrada, procesando éstos en paralelo y regresando al usuario un conjunto de resultados.
 - Librerías que exponen las capacidades de paralelismo del hardware, donde el programador se encarga de especificar qué tipo de tarea se debe llevar a cabo en paralelo y cómo. La librería PThreads de C/C++ es un ejemplo de este tipo de librerías. Por lo general, éste tipo de librerías deben ser incluidas en el código fuente.
- Directivas de Compilación, se basa en directivas de pre procesamiento o *pragmas*, colocados en un código secuencial para identificar secciones de código que pueden ser paralelizadas. Compiladores como GNU gcc/g++, PGI Compiler, Clang, ofrecen esta funcionalidad por medio de las librerías OpenMP, y OpenACC.
- Paralelización Automática, esta alternativa confía plenamente en la capacidad del compilador para analizar el código y determinar en qué partes se debe aplicar paralelismo, por lo general los compiladores que desarrollan este tipo de tareas son referidos como *transpilers* o *source-to-source compilers*. Ejemplos de este tipo de compiladores son Dawncc [44] y Parallware [31].

3.2.1. Clasificación

Dado que los modelos de programación son únicamente abstracciones, luego estos deben ser implementados en librerías, las mismas librerías son denominadas modelos de programación. Como resultado, cuando se hace referencia en la literatura a modelos de programación paralela, se habla de librerías como PThreads, OpenMP, OpenACC, OpenMPI, OpenCL, etc. En lo que corresponde a esta sección, nos interesa saber cómo estos modelos pueden ser clasificados, lo cual nos brindará un indicio de qué modelo utilizar dependiendo del tipo de sistema o computador paralelo considerado. En general, existen varias formas de clasificar los modelos de programación paralela, por ejemplo, se consideran aspectos como, la interacción entre los procesos, la forma de descomposición del problema y otros aspectos. En esta

sección, se considera clasificar los modelos de programación paralela teniendo en cuenta la arquitectura de memoria en la cual son soportados. De acuerdo a la arquitectura de memoria, se puede clasificar los modelos de programación paralela de la siguiente manera:

1. Para Computadores Paralelos MIMD.

- Modelos de Memoria Compartida. Los modelos de memoria compartida se basan en la suposición de que las entidades de ejecución (o trabajadores) que llevan la ejecución real de las instrucciones del programa tienen acceso a un área de memoria común, donde pueden almacenar objetos, accesibles de manera uniforme para todos. Pthreads y OpenMP son ejemplos de librerías que implementan este modelo [59, p. 7].
- Modelos de Memoria Distribuida. Los modelos de memoria distribuida se basan en la suposición de que las entidades (o trabajadores) que llevan a cabo la ejecución real de las instrucciones del programa no cuentan con un área de memoria común, por lo tanto estos deben hacer uso de algún mecanismo de paso de mensajes que les permita acceder a los datos que residen en una memoria remota. OpenMPI es un ejemplo de una librería que implementa este modelo.

2. Para Computadores Paralelos MIMD con dispositivos SIMD

- Modelos de Computación Heterogénea. Estos modelos asumen la existencia de dos entidades el *host* (CPU) y el *device* (GPU), cada uno con su memoria. Donde el primer ente envía porciones de código (llamadas *kernels*) al segundo, el cual posteriormente procede a su ejecución, lo anterior se conoce como programación heterogénea. OpenCL, CUDA y OpenACC son ejemplos de librerías que implementan dicho modelo [59, p. 10-12].

A pesar de la variedad de alternativas que implementan los modelos de programación, Lenguajes de Programación Paralela, Librerías Optimizadas, Directivas de Compilación, y Paralelización Automática, en la práctica, el concepto de Directiva de Compilación ha tomado mucha más fuerza, en comparación con las demás alternativas. En primera instancia, esta alternativa parte de un programa secuencial, luego el programador simplemente debe indicar dónde hay paralelismo. En alternativas como PThreads el programador debe lidiar con detalles como la creación de los hilos de ejecución y la

descomposición del problema en partes independientes. Por otro lado usando las directivas de compilación, el usuario únicamente debe preocuparse por aplicar de forma correcta las directivas y verificar que el comportamiento del programa sea el esperado. Ejemplos de librerías que brindan este nivel de abstracción de los modelos de programación son OpenMP y OpenACC.

3.2.2. OpenMP

OpenMP es un modelo de programación paralela diseñado para sistemas de memoria compartida. Se compone de una serie de directivas de compilación (o *pragmas*) y librerías de funciones, usadas para paralelizar código secuencial. Una característica importante de OpenMP es que soporta *paralelismo incremental*, es decir, la posibilidad de aplicar directivas de forma gradual al código secuencial, con el fin de brindar al compilador información más detallada del código a paralelizar [59, p. 8]. En la Figura 3.8a se muestra un ejemplo de código secuencial, seguidamente, la forma en la cual es ejecutado en un computador, Figura 3.8b. Luego si se aplica algo de paralelismo al programa secuencial de la Figura 3.8, obtendremos como resultado, el comportamiento mostrado en la Figura 3.9.

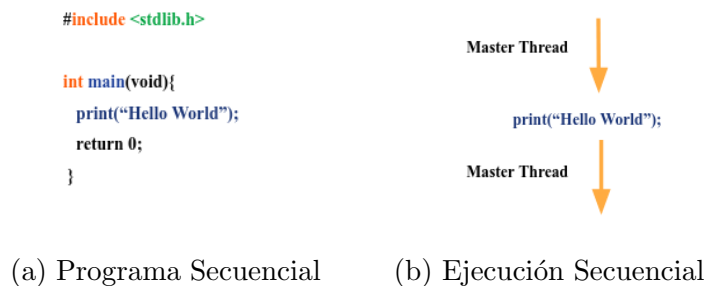


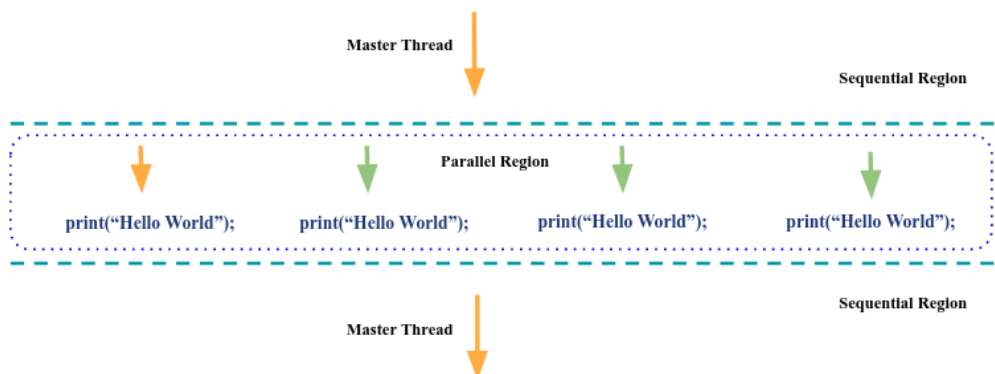
Figura 3.8: Ejecución de un Programa Secuencial, el paradigma de programación secuencial es el paradigma de programación más antiguo y de igual forma el más usado en la actualidad, dado que este paradigma nace en la era de los Computadores Secuenciales. Uno de los aspectos a considerar en la programación secuencial, es que incluso al ser ejecutado un programa en un computador paralelo, se obtiene el mismo comportamiento secuencial mostrado en la Figura 3.8b.

En general, la simplicidad que ofrecen las directivas de compilación, permiten llevar a cabo gran cantidad de trabajo, aplicando pequeños cambios al código de forma incremental. En particular, OpenMP fue diseñado para expresar paralelismo en sistemas multinúcleo. De igual forma soporta los

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

int main(void){
    #pragma omp parallel
    {
        print("Hello World");
    }
    return 0;
}
```

(a) Programa Paralelizado



(b) Ejecución en Paralelo

Figura 3.9: Ejecución de un Programa Paralelizado, note que el programa de la Figura 3.9a luce algo diferente respecto al programa de la Figura 3.8a. Aunque la diferencia es de cierto modo insignificante, el nivel de abstracción que hay detrás de las directivas es tanto, que al final obtenemos un programa paralelizado, dejando a un lado, procesos de creación y terminación de hilos, así como la división del dominio del problema.

modelos de paralelismo a nivel de datos, paralelismo a nivel de tareas y hilo de ejecución.

3.2.3. OpenACC

OpenACC (por *Open Accelerators*), al igual que OpenMP, es un modelo de programación basado en directivas de compilación. Diseñado principalmente para expresar paralelismo en sistemas heterogéneos CPU/GPU, también llamados Dispositivos Aceleradores de Vídeo (por ejemplo: Nvidia, AMD, Intel). Además, es una alternativa al modelo de programación CUDA. OpenACC al igual que OpenMP, busca reducir la complejidad presente cuando se desea añadir paralelismo a un programa en lenguaje de programación C/C++ [38, p. 315] [25, p. 9]. La librería de OpenACC puede ser habilitada en compiladores *GNU GCC/G++ Compiler*, y *PGI Compiler* por medio de la bandera de compilación “-acc”. La versión más estable de esta librería es ofrecida por el compilador *PGI Compiler*, el cual cuenta con herramientas de depuración y análisis de código avanzadas.

Los sistemas heterogéneos CPU/GPU normalmente se encuentran en sistemas de memoria compartida, dado que este tipo de sistemas funcionan como un esquema maestro-esclavo, donde debe existir un ente llamado *host* (o CPU) el cual es el encargado de coordinar o cargar el trabajo pesado a un *device* (o GPU), el ente encargado de llevar a cabo las tareas intensivas en cálculo. A pesar de ser necesario un esquema maestro-esclavo, entre ambos prima la cooperación, siguiendo la filosofía de la Computación Heterogénea, donde la CPU lleva a cabo tareas secuenciales y la GPU tareas intensivas. En la Figura 3.10 muestra la forma de cooperación entre CPU y GPU. Para GPUs, OpenACC soporta el modelo de hilo de ejecución y paralelismo a nivel de datos, adicionalmente soporta paralelismo a nivel de tareas con bajo rendimiento, dado que las GPUs no están diseñadas para llevar a cabo tareas secuenciales.

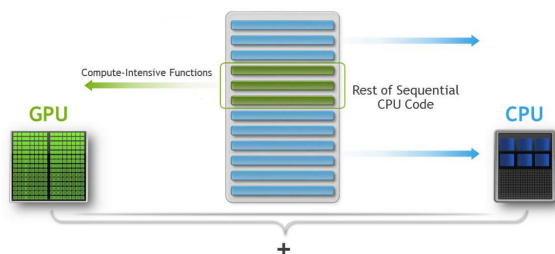


Figura 3.10: Cooperación entre CPU y GPU en la Computación Heterogénea [8].

3.3. Patrones de Programación Paralela

Los patrones pueden definirse vagamente como estrategias comúnmente recurrentes para tratar problemas particulares. Por lo regular, los patrones encapsulan buenas prácticas de programación y en cierta medida tienen la propiedad de ser reusables [43, p. 19]. En particular, se pueden aplicar patrones al diseño de aplicaciones paralelas, con el objetivo de hacer una aplicación fácilmente paralelizable y comprensible para otros programadores. Los patrones pueden afectar la forma en la cual los algoritmos están organizados, comúnmente para lograr escalabilidad, buena localidad⁸ de los datos y evitar el *overhead*⁹ [43, p. 20].

Antes de considerar los patrones de programación paralela, es importante considerar el concepto de **esqueleto algorítmico**, concepto del cual parte el estudio de los patrones de programación paralela. En efecto, “el término esqueleto algorítmico fue introducido por Murray Cole, y tiene su origen en la observación de que muchas aplicaciones paralelas comparten un conjunto común de interacciones conocidas”. El estudio de los esqueletos algorítmicos permite el análisis de los problemas que comparten un conjunto de interacciones y adicionalmente la creación de interfaces que sean portables y eficientes. Uno de los problemas tratados en esta área, es la definición de *esqueletos elementales* que sirvan como punto de partida para la construcción de programas paralelos. Por ejemplo, los bien conocidos operadores de listas *map* y *reduce* forman un conjunto de esqueletos elementales con paralelismo inherente. A pesar de que se proporcionan operadores equivalentes en la mayoría de los lenguajes y *Frameworks* de cómputo paralelo (el *map* corresponde a una operación a nivel de elemento y el *reduce* a una reducción), la ventaja principal de usar esqueletos elementales es la disponibilidad de un marco formal para la composición de un programa [48, p. xi].

De manera formal, “un patrón de diseño describe una buena solución para un problema recurrente en un contexto particular. El patrón sigue un formato prescrito que incluye el nombre del patrón, una descripción del contexto, las fuerzas (objetivos y restricciones) y la solución” [42]. Teniendo presentes los conceptos de patrón de diseño y esqueleto elemental, nos interesa entonces estudiar las características de los patrones de programación paralela *map* y *stencil*. Enfocando nuestro interés en el último, debido a que éste presenta

⁸La buena localidad en los datos en sentido figurado, significa que los datos están organizados de forma tal que optimiza el uso de la caché del procesador. Acelerando el proceso de lectura de datos e instrucciones de un programa.

⁹El concepto de *Overhead*, se relaciona a aquellos factores, ya sea de hardware, software (lenguajes de programación, librerías) o de pendientes de la misma implementación del código que incurren de forma negativa en el tiempo de ejecución de un programa paralelo.

como solución un esqueleto elemental para la construcción de **autómatas celulares** cuya estructura permite el problema sea fácilmente paralelizable. Al finalizar esta sección se define el esqueleto elemental que será usado por el prototipo de ambiente de programación paralela para abordar el modelamiento de autómatas celulares de propósito general.

3.3.1. Map

El patrón *map*, replica una función sobre cada elemento de una colección de datos. La función a ser replicada es llamada **función elemental**, esta función es aplicada a cada elemento de una colección de datos de entrada, generando de la misma forma una colección de datos de salida, los cuales conservan la misma estructura de los datos de entrada. Por ejemplo, en el Listado 3.1, podemos ver la implementación del patrón *map* en Python, donde la función `add_one` hace el rol de función elemental, la cual es aplicada a cada elemento de la colección de datos `collection`. Además, el patrón *map* puede ser visto en uno de los usos que tienen los ciclos en un programa secuencial, si se conoce de antemano tanto el número de iteraciones, como la independencia entre las iteraciones del mismo. En el Listado 3.2 se puede observar que la operación “sumar 1 al i-ésimo elemento”, posee el mismo patrón de interacción que la misma función *map* en el Listado 3.1.

```
def add_one(value):
    return value + 1

collection = [1,2,3,4,5,6,7,8]
collection = map(add_one,collection)
```

Listado 3.1: Ejemplo de la implementación en Python del Patrón *Map*.

```
collection = [1,2,3,4,5,6,7,8]

for i in range(len(collection)):
    collection[i] = collection[i] + 1
```

Listado 3.2: Ejemplo de un Ciclo que Cumple con las Características del Patrón *Map*.

Tanto en el Listado 3.1 como en el Listado 3.2 podemos distinguir una función elemental. Esto, nos indica de algún modo que en ambos casos podemos aplicar paralelismo sin mayor dificultad [43, p. 88-89]. Problemas que

presenta este patrón estructural de paralelismo, ya sea en forma de una función *map* o un ciclo, son considerados como fácilmente paralelizables, dado que la paralelización del problema consiste simplemente en la aplicación de la función elemental a cada uno de los elementos de la colección de forma simultánea. Finalmente el Listado 3.3 muestra un ejemplo de un problema que no cumple con las características del patrón *map*, dado que la función elemental debe depender únicamente del elemento de entrada, esto sugiere que el problema deba ser replanteado o no hace parte del conjunto de problemas que pueden ser abordados con este patrón de programación paralela.

```
collection = [1,2,3,4,5,6,7,8]

for i in range(len(collection)):
    collection[i] = collection[i] + collection[i-1] + 1
```

Listado 3.3: Ejemplo de un Ciclo que no Cumple con las Características del Patrón *Map*.

Para resumir lo visto en esta sección, la Figura 3.11, muestra en resumen cómo sería el proceso de aplicación de una función elemental sobre un conjunto de datos, tanto de forma secuencial como en paralelo. Además, se presenta un breve resumen del patrón a continuación.

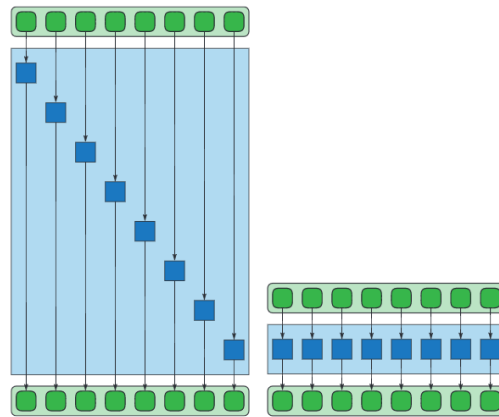


Figura 3.11: Patrón *Map*. En un patrón *map*, una función es aplicada a todos los elementos de una colección, produciendo usualmente una nueva colección con la misma forma de la colección de entrada [43]. En un programa secuencial a menudo el patrón *map* es implementado como un ciclo [8, p. 89].

1. Patrón: *Map*
2. Contexto: Replicar una función elemental sobre cada elemento de una colección de datos de entrada.
3. Objetivos:
 - Permitir la actualización de elementos de una colección de forma simultanea.
4. Restricciones:
 - Debe existir una función elemental o al menos un abstracción que cumpla con sus propiedades.
 - Se debe conocer de antemano el tamaño de la colección.
 - La colección de datos de salida debe tener la misma forma de los datos de entrada.
 - La salida de la función elemental para cada elemento únicamente depende de la aplicación de la misma en un único elemento de entrada.
 - La función elemental no debe tener efectos secundarios, es decir no debe afectar el estado del código en general.
5. Solución: Expresar la operación a realizar sobre cada elemento de una colección en términos de una función elemental.

3.3.2. Stencil

Es una generalización del patrón *map*, en la cual la **función elemental** no solo tiene acceso a un único elemento de una colección de datos de entrada, sino que adicionalmente puede acceder a un conjunto de **vecinos**, ver Figura 3.12. Al igual que el *map*, se cuenta con una *función elemental* que debe ser aplicada a cada elemento de una colección de datos, pero en este caso, la función elemental toma como entrada el vecindario correspondiente a cada elemento de la colección. Es decir el nuevo valor para cada elemento de la colección debe depender estrictamente de sus vecinos. Por lo general, en este patrón la colección de datos toma la forma de una matriz. Para este patrón se hace necesario establecer **condiciones de frontera**, dado que en los bordes o frontera de la matriz la indexación de los vecinos para cada elemento en la frontera cambia. En estos casos se puede optar por una de las siguientes alternativas, modificar la indexación para accesos fuera de los límites o ejecutando versiones de casos especiales de la **función elemental**.

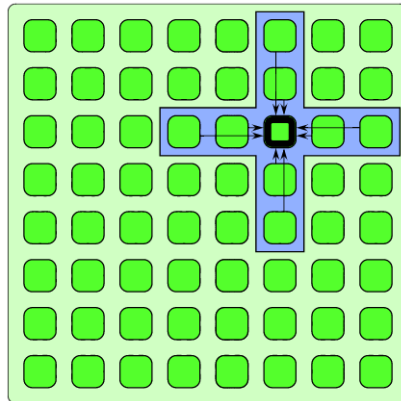


Figura 3.12: Patrón *Stencil*. Se genera un conjunto de resultados, cada uno como resultado de una función elemental que recibe como entrada un conjunto de vecinos. Normalmente los vecinos están ubicados a una distancia fija de cada una de las celdas o elementos de referencia. En este caso solo estamos considerando el vecindario de una celda. Sin embargo, la realidad es que cada casilla tiene un vecindario, con las mismas características al vecindario mostrado. De forma consecuente la función elemental puede ser aplicada en paralelo sobre el conjunto de celdas de la matriz [43, p. 90].

Este patrón es comúnmente usado para el estudio de sistemas dinámicos que pueden ser vistos como la composición de varios elementos y a su vez la interacción entre sus componentes puede ser modelada por medio de una función elemental sencilla. Entre los ejemplos de sistemas dinámicos que siguen éste patrón podemos encontrar sistemas relacionados con Dinámica de Fluidos, **Autómatas Celulares** incluyendo el modelo *Lattice Boltzman*. Adicionalmente es usado como solucionador de ecuaciones diferenciales parciales [43, p. 89]. Para finalizar la presente sección se muestra a continuación un resumen del patrón *stencil*.

1. Patrón: *Stencil*
2. Contexto: Replicar una función elemental sobre cada elemento de una colección de datos de entrada, donde la función elemental depende de un conjunto de vecinos, y cuyo patrón de vecindad es conocido.
3. Objetivos:
 - Permitir la actualización de elementos de una colección de forma simultánea.
4. Restricciones:
 - Debe existir una función elemental o al menos un abstracción que cumpla con sus propiedades.
 - Se debe conocer de antemano el tamaño de la colección.
 - La colección de datos salida debe tener la misma forma de los datos de entrada.
 - La salida de la función elemental para cada elemento únicamente depende la vecindad de cada elemento de la colección.
 - La función elemental no debe tener efectos secundarios, es decir, no debe afectar el estado del código en general.
5. Solución: Expresar la operación a realizar sobre cada elemento de una colección en términos de una función elemental que tome como entrada el vecindario de una celda o elemento dado.

3.3.3. Esqueleto Elemental de Autómata Celular Para el Prototipo

Después de haber estudiado las características y propiedades de los autómatas celulares, así mismo, los problemas que pueden ser abordados con los mismos. En esta sección se propone el esqueleto elemental o plantilla base, la cual será usada en el prototipo para el tratamiento de problemas que puedan ser modelados como un autómata celular de propósito general. De esta manera, el esqueleto elemental propuesto constará básicamente de cinco funciones: *initialize*, *function*, *neighborhood*, *evolve*, *main*. Cada descrita a continuación.

1. *initialize*

- Parámetros mínimos: Matriz, que representa el espacio celular.
- Funcionalidad: Debe inicializar la matriz.
- Salida: Sin retorno.

2. *function*

- Parámetros mínimos: Vecindario de una celda dada (estructura cuyos atributos sean los vecinos definidos para el sistema en consideración).
- Funcionalidad: Hace las veces de **función elemental o dinámica del sistema**. Dado un vecindario (estructura), el usuario debe calcular el siguiente estado de una celda dada.
- Salida: El nuevo estado de la celda en consideración.

3. *neighborhood*

- Parámetros mínimos: Matriz, representa el estado actual del espacio celular; posición, la ubicación de una celda en el espacio celular de la cual se desea obtener su vecindario.
- Funcionalidad: Dada una ubicación en el espacio celular, determinar cuáles son los vecinos de la celda que se encuentra en la ubicación dada como parámetro.
- Salida: El vecindario que corresponde a la celda ubicada en dicha posición.

4. *evolve*

- Parámetros mínimos: Matriz que guarda el estado actual del sistema.
- Funcionalidad: Debe aplicar *neighborhood* sobre cada celda para determinar su vecindario y llamar a *function*, la cual determinará el siguiente estado de la celda, dado su vecindario. En general, es la función encargada de la evolución del sistema y por ende donde se toma la mayor parte del tiempo de procesamiento. El Listado A.3, muestra un ejemplo de la implementación de esta función.
- Salida: Sin retorno.

5. *main*

- Parámetros mínimos: Sin parámetros.
- Funcionalidad: Debe llamar a *initialize* para inicializar la Matriz o espacio celular, además debe invocar a la función *evolve*, la cual llevará a cabo el proceso de evolución del sistema.
- Salida: Sin retorno.

Es natural preguntarse, el porqué de estas funciones, si claramente el código de un problema en C99, se puede escribir sin necesidad de usar funciones. Lo cierto es que al elaborar una plantilla en C99 basada en funciones, el nombre de cada función, le da al usuario un indicio de la utilidad de la misma en el código, lo que permite de cierta forma que la plantilla sea re utilizable. Además, expresar la **función elemental** del sistema, en términos de una función, en este caso **function**, cómo lo indica el patrón de programación paralela *stencil*, permitirá aplicar dicha función en paralelo en la colección de elementos del espacio celular. En el último capítulo del presente documento, Capítulo 5, se estudia qué implementación concreta en lenguaje C99 debería tener la función *evolve*, dado que la forma en la cual sea implementada dicha función tendrá implicaciones significativas en el tiempo de ejecución de los modelos generados a partir de este esqueleto elemental.

Capítulo 4

Prototipo Ambiente de Programación Paralela Para Autómatas Celulares de Propósito General

En capítulos anteriores fue posible describir en primera instancia, la utilidad que nos brindan los **autómatas celulares** como método numérico tanto para el estudio de sistemas que exhiben un comportamiento complejo (el juego de la vida) como para encontrar soluciones numéricas a ecuaciones diferenciales parciales (ecuación del calor de Fourier). Así mismo, se mostró que los problemas basados en autómatas celulares siguen un patrón de paralelismo llamado *stencil*, luego éste patrón nos permitiría implementar modelos de autómata celular fácilmente paralelizables. Sin embargo, el hecho de ser paralelizable un modelo de autómata celular no implica que haga uso eficiente de las múltiples unidades de procesamiento de un computador paralelo. Es en ésta parte del rompecabezas donde encajan **los modelos de programación paralela**, en particular OpenMP y OpenACC, los cuales son considerados, dado el alto nivel de abstracción de las capacidades de un computador paralelo que estos proveen. Finalmente estos permitirán que nuestros modelos tomen ventaja del paralelismo presente en los sistemas de memoria compartida homogéneos (múltiples CPUs) y/o heterogéneos (CPUs + GPUs).

El presente capítulo describe la implementación de tres herramientas: Plantillas, Pragcc y Coder. Las cuales pretenden asistir el proceso de programación paralela de autómatas celulares de propósito general. La primera, dada las características de un modelo de autómata celular, devuelve al usuario una plantilla en C99 de la cual puede partir para programar su modelo; La

segunda, permite la paralelización del modelo generado con la herramienta anterior; La tercera, funciona como editor de código WEB, además integra las funcionalidades de las dos primeras herramientas. Finalmente las tres herramientas constituyen el prototipo propuesto en el presente documento y cuyos detalles son descritos como sigue.

4.1. Plantillas

Es un repositorio y al mismo tiempo una herramienta WEB que reúne un conjunto de plantillas en C99 de patrones de programación en paralelo. El objetivo de esta herramienta es brindar al usuario esqueletos elementales que sirvan como punto de partida para la construcción de programas con procesamiento paralelo. Como se muestra en la Figura 4.1, cada plantilla cuenta básicamente con 4 archivos, *template.c*, *context.yml*, *parallel.yml* y *Makefile*.

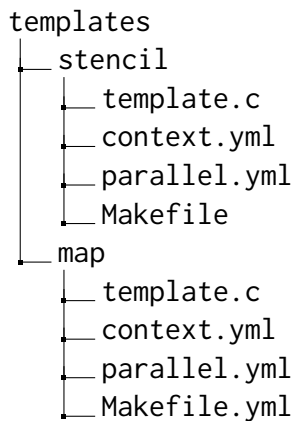


Figura 4.1: Repositorio de Plantillas Base.

El archivo **template.c**, es un código fuente escrito en C99, pero con sintaxis del motor de plantillas *Jinja2*, con el fin de dar mayor flexibilidad al usuario que usa la misma plantilla para diferentes problemas relacionados con el mismo patrón de programación paralelo. Esta plantilla debe tener implícito un patrón de programación paralelo, además, debe ser re-utilizable. Un ejemplo de plantilla se muestra en el Listado A.1. En particular el archivo *template.c* de la plantilla *stencil*, fue implementada según el esqueleto elemental propuesto en la Sección 3.3.3.

El archivo **context.yml**, es un archivo de meta-datos en formato YAML el cual permite que el archivo *template.c* se ajuste a las necesidades del problema a tratar. Por ejemplo, para el patrón *stencil*, se debe definir un

archivo *context.yml* como se muestra en el Listado A.2. Dados los parámetros mencionados en el Listado, la herramienta **Plantillas** devuelve un código fuente, Listado A.3, que cumple con las características presentadas en el archivo *context.yml*.

Los propiedades del archivo *context.yml* específicamente para la plantilla *stencil* son descritas a continuación.

- ***lattice***, contiene las propiedades relacionadas al espacio celular del modelo: *rowdim*, *coldim*, *type*, *neighborhood*.
 - *rowdim*, número de filas del espacio celular.
 - *coldim*, número de columnas del espacio celular.
 - *type*, el tipo de dato de las celdas del espacio celular. Por el momento, los tipos de datos soportados son (int,bool,double,float).
 - *neighborhood*, conjunto de vecinos, donde cada vecino tiene un *neighbor name* y la ubicación relativa del vecino a la celda de referencia. Ejemplo:
 - $\langle \text{neighbor name} \rangle : [\langle x \text{ offset} \rangle, \langle y \text{ offset} \rangle]$
- *generations*, número de generaciones que iterará el sistemas modelado.

El archivo *Makefile*, pertenece a una utilidad llamada *make* disponible para sistemas operativos Linux, la cual permite compilar y ejecutar programas a partir de simples reglas básicas. Permitiendo al usuario configurar y correr el programa fácilmente. El archivo *parallel.yml*, sera descrito posteriormente junto con la herramienta **Pragcc**.

4.1.1. Detalles de la Implementación

Plantillas es un proyecto de software libre cuyo código fuente se encuentra disponible en Github. Para facilitar la contribución y mantener una versión estable de la herramienta, el repositorio cuenta con un conjunto de pruebas, las cuales son ejecutadas automáticamente en la nube cuando se hace una contribución a la rama principal del proyecto. A continuación, se listan las herramientas usadas, así mismo, su objetivo en el desarrollo de la herramienta.

- Python 3.5.2, lenguaje de programación de la presente herramienta.
- C99, lenguaje base para las plantillas *template.c*.

- Jinja2, lenguaje de plantilla usado en los archivos *template.c*.
- Flask-restplus, una librería de Python, la cual permite diseñar una aplicación WEB con características de API Rest a partir de una aplicación Python.
- Github y Git, como sistema manejador de versiones para el desarrollo de software.
- Travis CI, es un sistema que permite correr el conjunto de pruebas del software de forma automatizada en la nube.
- Docker 17.09.1-ce, herramienta que permite el despliegue automatizado de la aplicación WEB en una infraestructura similar a una máquina virtual (Contenedor), la cual es provista de las librerías y paquetes necesarios para que la aplicación funcione correctamente en sistemas operativos que soporten virtualización con Docker.

4.2. Pragcc

Pragcc por *Pragmatic C99 Source Code*, es la iniciativa de un *Source-to-Source Compiler*, el cual, a partir de un archivo de meta-datos llamado *parallel.yml* permite paralelizar código en C99 con directivas de compilación OpenMP y OpenACC. Dicho lo anterior, éste archivo, permite conservar de forma independiente, información acerca de cómo debe ser paralelizado un determinado programa, en especial la plantilla *stencil*. Vale la pena resaltar que la correcta paralelización del código fuente, depende de cómo este definido el archivo, lo anterior sugiere este archivo sea creado por una persona que conozca de antemano la naturaleza del problema a paralelizar. Al igual que la herramienta Plantillas, Pragcc puede ser usado como una aplicación WEB, de esta forma éste toma como entrada tanto el archivo *parallel.yml*, Listado A.4, como el código fuente asociado a dicho archivo, Listado A.3, para generara el mismo código anterior, pero anotado con directivas de compilación OpenMP o OpenACC.

4.2.1. Detalles de la Implementación

Pragcc se encuentra desarrollado en Python, además se basa en la librería **pycparser**, la cual se encarga de construir el **árbol de sintaxis abstracta**¹

¹Un árbol de sintaxis abstracta, es una estructura de datos cuyo contenido representa la información relevante de un código dado en cualquier lenguaje de programación.

del código en lenguaje C99. A partir de la información que brinda el árbol, es posible llevar a cabo el anotado del código con directivas de OpenACC y OpenMP.

También es relevante mencionar que Pragcc es un proyecto de software libre cuyo código fuente se encuentra disponible en Github. Y al igual que Plantillas, cuenta con un conjunto de pruebas automatizado para mantener una versión estable de la herramienta. A continuación, se listan las herramientas usadas, así mismo, su objetivo en el desarrollo de la herramienta.

- Python 3.5.2, lenguaje de programación de la presente herramienta.
- Pycparser, librería de Python que permite analizar el código en C99 para su posterior paralelización.
- C99, lenguaje soportado para anotado de código con directivas de compilación.
- Flask-restplus, convierte a Pragcc en una aplicación WEB con características de API REST.
- Github y Git, como sistema manejador de versiones para el desarrollo de software.
- Travis CI, es un sistema que permite correr el conjunto de pruebas del software de forma automatizada en la nube.
- Docker 17.09.1-ce, provee el despliegue automatizado de la herramienta en sistemas operativos que soporten virtualización con Docker.

4.3. Coder

Es la herramienta que sirve como editor de código WEB e integra las funcionalidades de **Plantillas** y **Pragcc**. De forma particular, Coder permite crear un proyecto de autómatas celulares en C99 por medio de la interfaz mostrada en la Figura 4.2, donde el usuario especifica las características deseadas para el modelo de autómatas celulares: dimensiones del modelo, tipo de celdas para el espacio celular y tipo de vecindario (Ejemplo, Moore o Neumann). Luego de haber creado el proyecto, el usuario podrá editar, compilar, y paralelizar el modelo de autómatas celulares, como se muestra en la Figura 4.3.

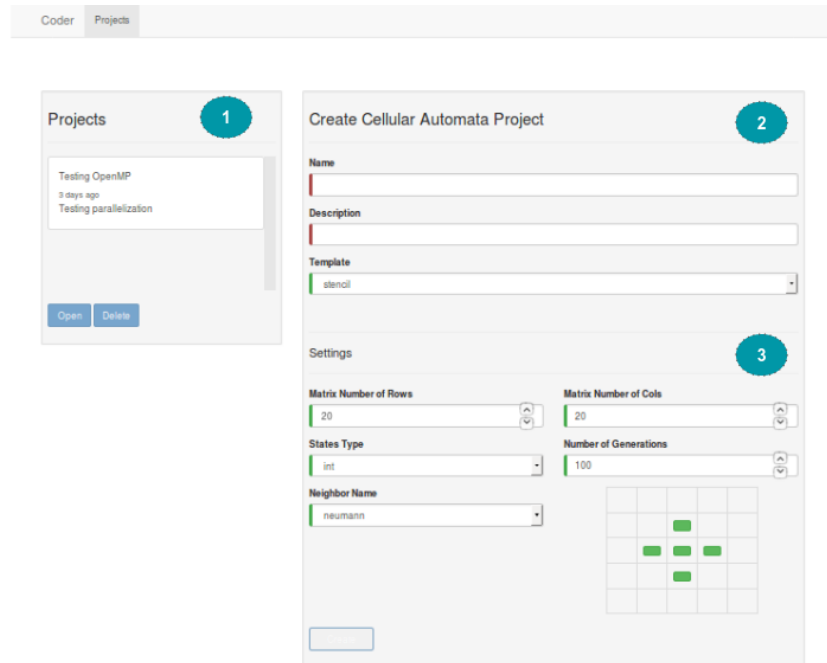


Figura 4.2: El editor de Proyectos consta de tres secciones, la sección 1, diseñada para listar, abrir y eliminar proyectos; Las secciones 2 y 3 se encargan de recolectar los detalles del proyecto a crear, así mismo se encargan de la validación de los datos de entrada.

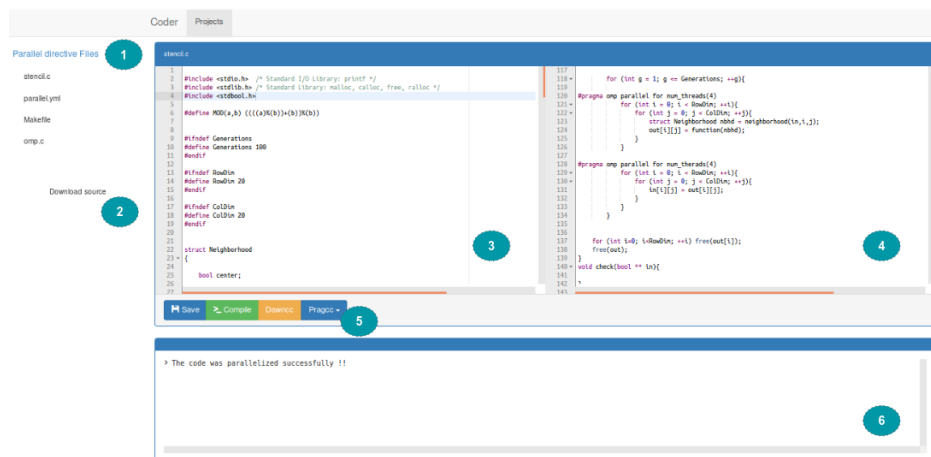


Figura 4.3: El Editor, se compone del visor de archivos, sección 1; un botón para descargar el proyecto en formato zip, sección 2; el editor de código, sección 3; el visor de código paralelizado, sección 4; las opciones disponibles para el archivo seleccionado, sección 5; consola con el fin de indicar al usuario el resultado de cada opción.

4.3.1. Detalles de la Implementación

Coder es una herramienta WEB desarrollada bajo el estilo de diseño WEB API REST, el cual permite a varias aplicaciones independientes (Coder, Pragcc y Plantillas) comunicarse por medio del protocolo HTTP mediante el paso de peticiones en formato JSON, como se muestra en la Figura 4.4. Además, el diseño de cada componente en **contenedores Docker** permite en general el bajo acoplamiento, es decir, al estar cada componente principal, **Coder**, **Plantillas** y **Pragcc** en contenedores independientes, se puede desplegar cada herramienta de forma individual o la totalidad de herramientas en conjunto, Figura 4.4. Lo cierto es que en cualquiera de los casos, se garantiza que cada componente o la herramienta en general funciona correctamente², lo anterior, facilita el desarrollo de cada componente de forma individual. Además al ser un proyecto de software libre, su código fuente se encuentra disponible en Github. A continuación se listan las herramientas usadas en la presente herramienta, así mismo, el objetivo que cumple cada una.

²Correctamente quiere decir que la herramienta integradora (Coder) al detectar que uno de sus componentes no está presente, es capaz de dar una respuesta adecuada al usuario. Por otro lado, los componentes adicionales (Plantillas y Pragcc), deberían presentar el mismo comportamiento funcionando cada uno de forma independiente.

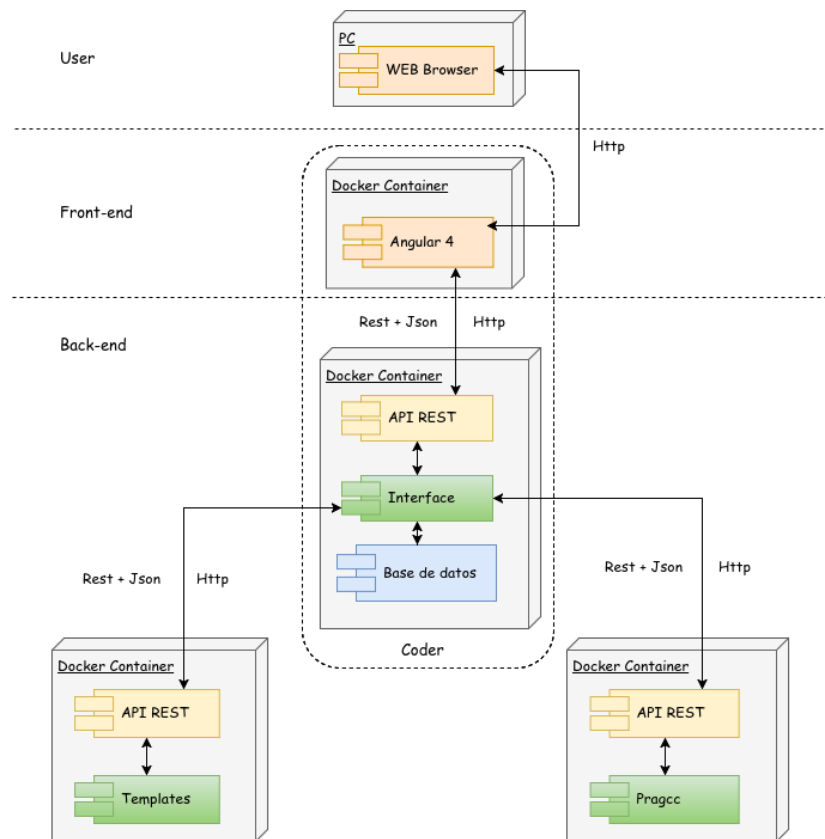


Figura 4.4: Diagrama de Despliegue de la Aplicación Web.

- Para el *front-end*:
 - HTML 5 y CSS, lenguajes usados para diseñar la parte visual de la aplicación web.
 - Angular 4, *Framework* de desarrollo WEB que convierte una pagina web en HTML 5 estática en una pagina web dinámica. Además permite hacer validación de formularios al lado del cliente, por ejemplo, el formulario para crear un proyecto de autómatas celulares es validado tanto al lado del cliente como al lado del servidor.
- Para el *back-end*:
 - Python 3.5.2, lenguaje de programación base de la presente herramienta.
 - Django 1.11.4, *Framework* de Python para hacer aplicaciones WEB.
 - Requests 2.18.4, librería de Python implementada en el módulo **Interface**, Figura 4.4 (color verde), para permitir la comunicación de **Coder** con herramientas externas Pragcc y Plantillas.
- Para el desarrollo y despliegue de la aplicación:
 - Github y Git, como sistema manejador de versiones para el desarrollo de software.
 - Docker-compose 1.15.0, permite desplegar y administrar una aplicación compuesta de múltiples contenedores. Funciona en sistemas operativos que tengan instalado Docker.

Capítulo 5

Validación del Prototipo

La primera parte del presente capítulo, demuestra qué implementación debería tener el esqueleto elemental del patrón *stencil*, para que los modelos de autómatas celulares generados a partir de éste tomen ventaja los recursos disponibles en un computador paralelo. Para finalizar, se describen a modo de comparación otras herramientas similares al presente prototipo.

5.1. Estudio del Modelo de Autómata Celular Generado por el Prototipo

En la Sección 3.3.3 se describe en palabras cómo debería ser implementado en código el esqueleto elemental usado por el prototipo, para abordar problemas de autómatas celulares de propósito general. Sin embargo, a partir de aquella descripción no es posible asegurar que el programa resultante sea eficiente en el uso de recursos de un computador paralelo. Dado lo anterior, el problema se reduce a encontrar una implementación en código C99 que haga uso eficiente de los recursos de un computador paralelo. De esta forma, dicha implementación en C99 será usada como esqueleto elemental para los modelos de autómatas celulares generados por el prototipo.

Lo contradictorio es que pueden existir infinitas implementaciones en C99 para el esqueleto elemental definido en la Sección 3.3.3. Entonces, el número de implementaciones posibles es equivalente a la cantidad de interpretaciones que puede dar cada persona a la descripción de dicho esqueleto elemental. De igual forma, sucede que cada implementación puede presentar mejor o peor rendimiento respecto a otra. Finalmente, esto se convierte en un problema combinatorio en el cual debemos comparar todas las implementaciones posibles para encontrar la mejor. Para acotar el espacio de búsqueda, se toma a bien considerar el estudio de dos implementaciones en particular, las cuales

se describen en la siguiente sección.

5.1.1. Descripción de los Modelos

De antemano, nos interesa estudiar de forma concreta, la implementación en C99 de la función **evolve** descrita en la Sección 3.3.3. Naturalmente la función **evolve** es seleccionada entre el grupo de funciones definidas para el esqueleto elemental por las siguientes razones:

1. Podemos asegurar que se puede paralelizar. Esto gracias a que sigue el patrón de programación paralela *stencil*.
2. Ésta función no va a ser manipulada por el usuario, es decir, su complejidad no es susceptible a cambios. En principio, el usuario debería manipular únicamente las funciones *function*, *neighborhood* e *initialize*.
3. La complejidad computacional de dicha función es $\mathcal{O}(N * X * Y)$, donde N es el número de generaciones que debe evolucionar el sistema, y $X * Y$ la dimensión del espacio celular en filas (*RowDim*) por columnas (*ColDim*). Además, **evolve** debe aplicar *function* y *neighborhood* $N * X * Y$ veces. Dado lo anterior, se puede asegurar que **evolve** es la función que domina en complejidad. Por lo tanto, siempre que se haga una mejora en alguna función que no sea **evolve**, la mejora en el tiempo de ejecución del programa resultante será insignificante. No obstante, si la mejora es llevada a cabo sobre la función **evolve**, el tiempo de ejecución del programa en general mejoraría de forma significativa.

Para determinar la implementación adecuada de la función **evolve**, se considera que dicha implementación en C99 debe cumplir las siguientes características.

1. En primer lugar, debe ser paralelizable con directivas de compilación OpenMP y OpenACC.
2. Luego dicha implementación, al momento de ser paralelizada con cualquiera de las directivas, debe tomar ventajas de los recursos del computador paralelo de manera eficiente, es decir, el tiempo de ejecución del programa debe presentar mejoras significativas, al mismo tiempo que aumenta el porcentaje de utilización de los recursos.

Dado lo anterior, se consideran las implementaciones de la función **evolve** presentes en el Cuadro 5.1, como objetos de estudio. El lector debe remitirse al Apéndice del presente documento para ver los detalles de cada implementación en código.

Cuadro 5.1: Convenciones para los Experimentos

Convención	Observación	Listado
1_S	Implementación 1 Secuencial	A.6
1_{MP}	Implementación 1 paralelizada con directivas OpenMP	A.7
1_{ACC}	Implementación 1 paralelizada con directivas OpenACC	A.8
2_S	Implementación 2 Secuencial	A.9
2_{MP}	Implementación 2 paralelizada con directivas OpenMP	A.10
$2,1_{MP}$	Variación implementación 2_{MP}	A.11
2_{ACC}	Implementación 2 paralelizada con directivas OpenACC	A.12

Cada implementación presente en el Cuadro 5.1, es una instancia de la función de evolución del modelo de autómata celular del Juego de la Vida, descrito en el Sección 2.5.1. Lo anterior, permite obtener estadísticas en tiempo de ejecución y comprobación del resultado de cada implementación. Se debe agregar que la implementación de las funciones *main*, *function*, *neighborhood* y *initialize*, es igual para todos los casos.

Por lo general cuando se considera un problema que sigue el patrón de programación *stencil*, se contempla el uso de dos matrices, *in* y *out*. La primera, que presenta el estado inicial del sistema, generación i ; La segunda donde residirá el nuevo estado del sistema, generación $i + 1$. Entonces, el algoritmo de evolución lee el estado actual *in* y escribe el estado siguiente en *out*, este primer paso se denomina **procesamiento del espacio celular**. Para el siguiente cálculo es necesario que *in* tenga el nuevo estado del sistema, y es por ello que se aplica un segundo paso, el **intercambio de información**, donde *out* pasa su información a *in*, de esta forma el siguiente cálculo se llevará a cabo sobre el nuevo estado del sistema en *in*. En este contexto *out* sirve el propósito de una variable auxiliar. A continuación se describen los detalles de experimentación.

5.1.2. Detalles de Experimentación

- Cada implementación posee 3 versiones: secuencial (S), con directivas OpenMP (MP), y con directivas OpenACC (ACC). Lo anterior, con el propósito de estudiar el comportamiento en tiempo de ejecución y uso de recursos en ausencia y presencia de paralelismo en ambas implementaciones.
- Todas las versiones son ejecutadas para dimensiones del espacio celular: 500^2 , 1000^2 , 1500^2 , 2000^2 , 2500^2 , y 3000^2 . Para estudiar, cómo se comporta cada versión a medida que crece el tamaño del problema.
- Para cada versión 100 generaciones son calculadas para cada dimensión del espacio celular.
- Cada versión es ejecutada 10 veces con el fin de tomar promedios y reducir el número de valores atípicos que puedan tomar lugar en los resultados.

En términos de complejidad computacional, la configuración del experimento sugiere las siguientes complejidades para las implementaciones secuenciales 1_S y 2_S . Siendo $N = 100$, se tiene:

$$\begin{aligned}
 &100 * (X * Y + C) \\
 &100 * X * Y + 100 * C \\
 &K = 100 * C \\
 &100 * X * Y + K \\
 &\mathcal{O}_{1_S}(X * Y)
 \end{aligned}$$

$$\begin{aligned}
 &100 * (X * Y + X * Y) \\
 &100 * (2 * X * Y) \\
 &200 * (X * Y) \\
 &\mathcal{O}_{2_S}(X * Y)
 \end{aligned}$$

Éste resultado implica que, siempre que el número de generaciones sea constante, ambas implementaciones 1_S y 2_S son equivalentes en complejidad¹.

¹La complejidad computacional es un concepto que mide la complejidad de un algoritmo teniendo en cuenta el número de pasos elementales que este lleva a cabo para desarrollar una tarea en particular. Este concepto es independiente de las del computador.

No obstante, lo anterior no implica que sean iguales en tiempo de ejecución². Es por ello que tomando como referencia los resultados obtenidos, nos interesa determinar cuál de las implementaciones 1 o 2 se comporta mejor en general.

5.1.3. Hardware y Software usado en el Experimento

A continuación se describen tanto el *hardware* como el *software* usado en el proceso de experimentación.

- CPU

- Modelo: Intel Core i5-3210M.
- Velocidad de Reloj: 2.50GHz.
- Núcleos Físicos: 2.
- Núcleos Virtuales: 2.
- Total Núcleos: 4.

- GPU

- Modelo: Nvidia GeForce GT 650M.
- Velocidad de Reloj: 950 MHz.
- Número de Multiprocesadores de Transmisión (SM): 2.
- Total Núcleos: 384 (192 per SM);
- Ancho de banda del bus de datos: 80 GB/s.

- Software

- Compilador: g++ desde un Contenedor Docker.

Ahora bien, si contamos con una CPU de 4 núcleos, el máximo porcentaje de CPU al que podría acceder un experimento secuencial es un valor cercano al 100 %. Por otro lado, para un experimento paralelo en CPU, éste porcentaje aumenta a un valor cercano al 400 %. Lo cierto es que en la CPU, conviven gran cantidad de procesos por parte del usuario y el sistema operativo, por lo anterior no es común ver que un proceso alcance el 100 % de forma secuencial o el 400 % con paralelismo en CPU. Para propósitos de el presente experimento, el término **Uso de CPU**, se define como el porcentaje de CPU al que pudo acceder un proceso durante su ejecución.

²El tiempo de ejecución de un programa, es el tiempo que éste toma para llevar a cabo una tarea. Esta característica depende del hardware del computador donde sea ejecutado el programa.

5.1.4. Análisis y Resultados Implementación 1

Al momento de paralelizar correctamente un programa 1_S con directivas OpenMP, implementación 1_{MP} respectivamente, se espera que al aumentar el porcentaje de uso del recurso CPU, de igual forma se reduzca el tiempo de ejecución. En las Figuras 5.1 y 5.2, se puede observar que dicho comportamiento se cumple. Sin embargo, aunque se hace uso de mas del 300% de la CPU, Figura 5.2, el tiempo de ejecución únicamente se reduce aproximadamente a la mitad. Como es natural, los programas que aplican paralelismo, deben incurrir en un costo por crear, administrar y terminar procesos adicionales. Éste costo se conoce como *parallel overhead*, éste *overhead* se hace más notorio cuando los procesos adicionales no tienen mucho trabajo por hacer. Si comparamos las dimensiones 500^2 , 1000^2 y 1500^2 en las implementaciones 1_S y 1_{MP} , sucede el fenómeno que estamos describiendo.

Tiempo de Ejecución

Implementación 1

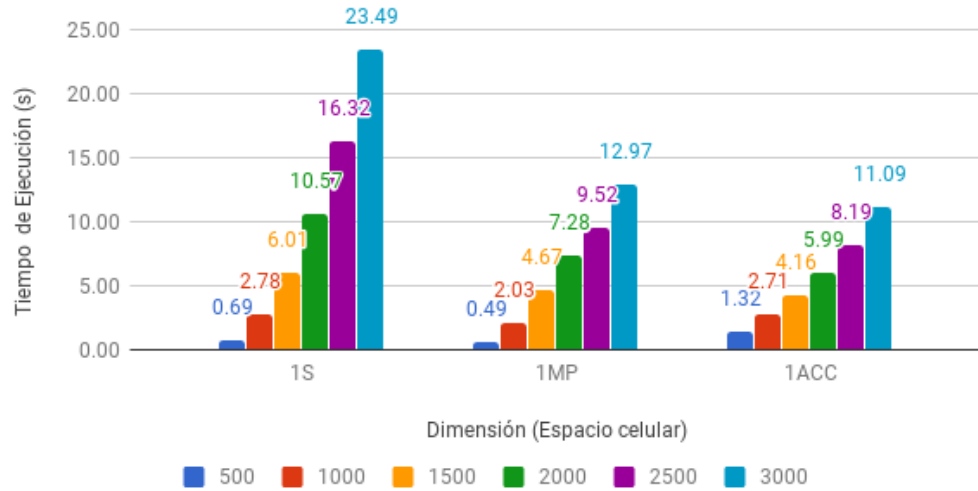


Figura 5.1: Gráfica Tiempo de Ejecución de Implementación 1

Por otro lado, al momento de paralelizar un programa 1_S con directivas OpenACC, implementación 1_{ACC} respectivamente, se espera que dada las capacidades de la GPU el tiempo de ejecución disminuya dramáticamente. De hecho, el tiempo de ejecución disminuye en comparación a la implementación secuencial 1_S . Sin embargo, respecto a la implementación paralela 1_{MP} , no se presentan cambios significativos. En efecto, la GPU a diferencia de la CPU, presenta bajo *parallel overhead*. En cambio, al estar presente en un sistema

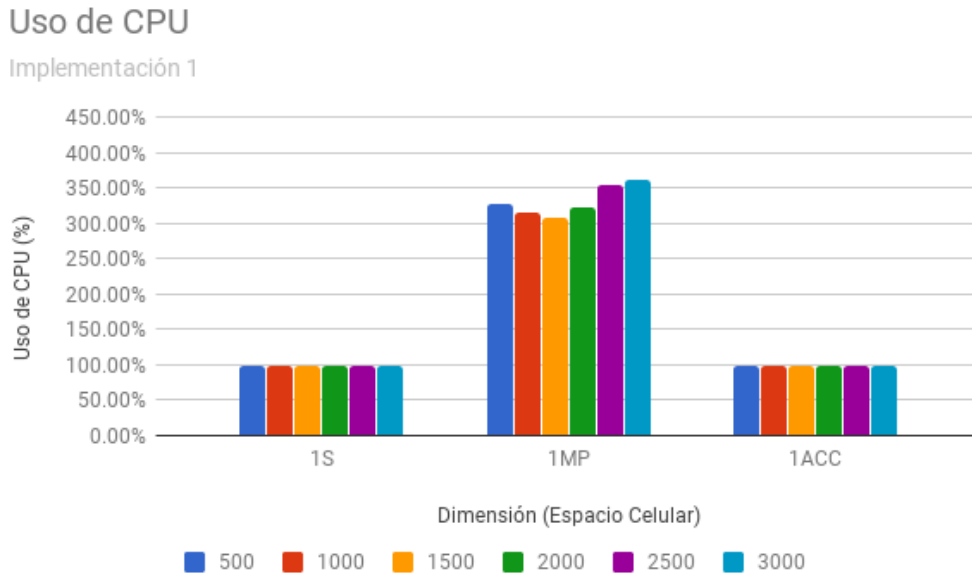


Figura 5.2: Gráfica Porcentaje Uso de CPU Implementación 1

heterogéneo donde tanto CPU como GPU por lo general gozan de memorias independientes, para hacer procesamiento en GPU, se debe incurrir en el costo de transferencia de datos entre las memorias de cada dispositivo.

La implementación 1_{ACC} en particular, trasfiere datos desde y hacia la GPU por cada generación. Lo anterior, debido a que el paso de procesamiento del espacio celular se hace en GPU y el paso de intercambio de información en CPU. Probablemente, podría llevarse el paso de intercambio a la GPU, sin embargo, OpenACC no provee los mecanismos (semáforos, barreras, etc) necesarios para asegurar que dicho paso sea llevado a cabo correctamente en ese lugar³. Finalmente, la implementación 1 no es apropiada para ser llevada a cabo en GPU, a pesar de ser la más intuitiva y probablemente eficiente en programación secuencial, no cumple los requerimientos necesarios para ser paralelizada en GPU. Dado que la CPU, en esta implementación realiza las tareas secuenciales y la GPU, las tareas intensivas, en CPU se puede alcanzar un porcentaje de uso cercano al 100%, como se muestra en la Figura 5.2, para la implementación 1_{ACC} .

³El hecho de detener la ejecución de una gran cantidad de hilos, para llevar a cabo una sección de código netamente secuencial, es demasiado costoso para la GPU. Es por ello que OpenACC no provee estos mecanismos. Para más información, el lector puede remitirse a [61, p. 9]

5.1.5. Análisis y Resultados Implementación 2

El problema presente en la implementación 1 sugiere un replanteamiento, en consecuencia se considera la implementación 2. Los resultados en tiempo de ejecución y utilización de CPU se muestran en las Figuras 5.3 y 5.4. En general, la implementación 2 cambia el paso de intercambio, de una operación con complejidad $\mathcal{O}(1)$, a una operación de copiado dato a dato con complejidad $\mathcal{O}(X * Y)$. Si se comparan las implementaciones 2_S y 2_{MP} se obtiene el mismo comportamiento visto comparando 1_S y 1_{MP} en la sección anterior. Este comportamiento debido al **parallel overhead** se conoce como la **Ley de Amdahl** y en general es una característica de los programas con procesamiento paralelo.

La siguiente implementación a considerar es la $2,1_{MP}$, donde se aplica paralelismo o directivas OpenMP al paso de intercambio de información. En éste experimento en particular, se observa que en algunos casos, añadir más paralelismo no siempre es conveniente, ver Figura 5.3 (implementaciones 2_{MP} y $2,1_{MP}$). Lo cierto es que existen problemas cuyo comportamiento mejora cuando se aplica más paralelismo así como aquellos que se comportan igual o peor. Finalmente, la implementación 2_{ACC} , presenta la reducción en tiempo de ejecución esperada, al llevar a cabo tanto el paso de procesamiento como el de intercambio en GPU, reduciendo el numero de transferencias de datos desde y hacia la memoria de la GPU. En concreto, se hace una transferencia de datos a GPU, seguida del procesamiento de datos durante 100 generaciones, para luego transferir los datos procesados de vuelta a la memoria principal de la CPU.

Tiempo de Ejecución

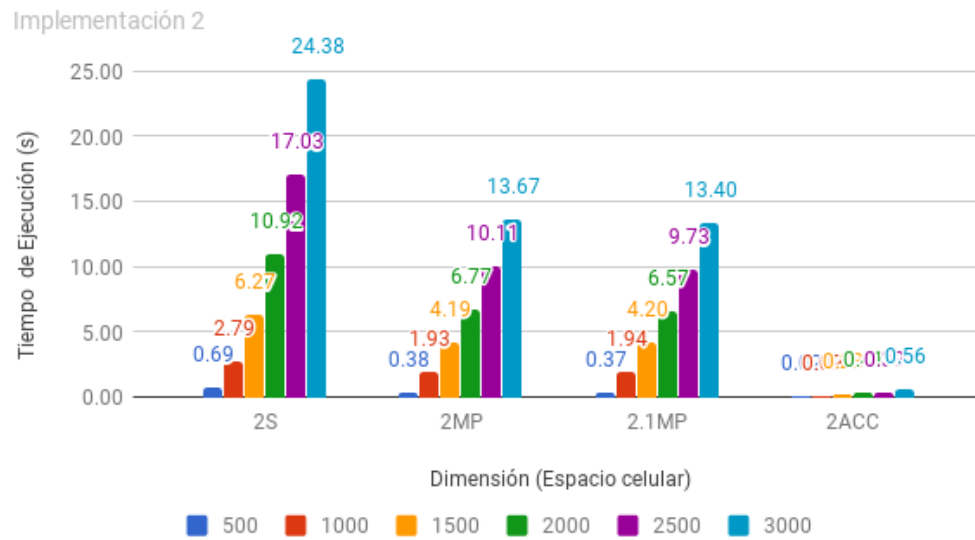


Figura 5.3: Gráfica Tiempo de Ejecución de Implementación 2

Uso de CPU

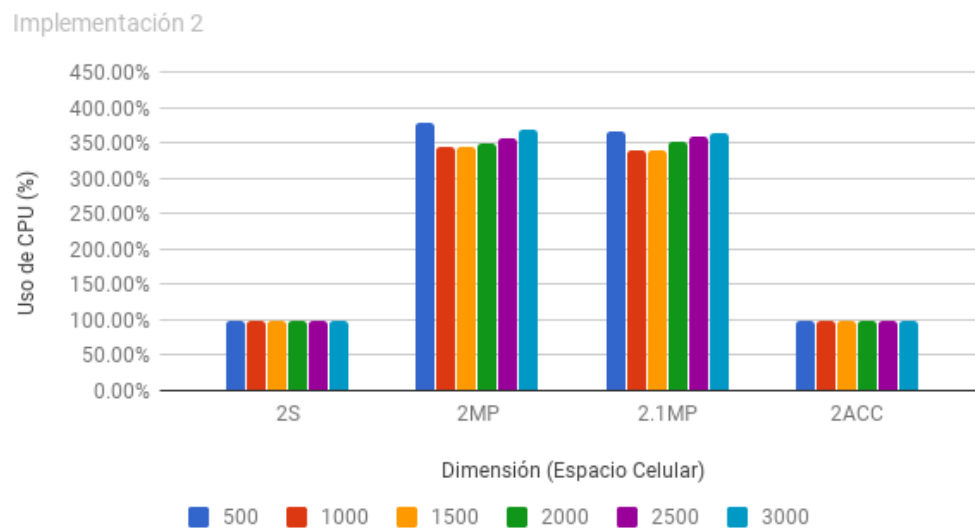


Figura 5.4: Gráfica Porcentaje Uso de CPU Implementación 2

5.1.6. Consolidación de Resultados

En las dos secciones anteriores se estudio el comportamiento de cada implementación de forma individual. En la presente sección intentaremos comparar la implementación 1 frente a la 2 en sus versiones S y MP donde no hay una diferencia clara en cuanto a los resultados obtenidos. Con respecto a la versión ACC, en la sección anterior quedó claro el porqué es superior a la implementación 1. Para introducir nuestro estudio, recordemos que ambas implementaciones 1 y 2 contemplan el mismo paso de procesamiento, pero difieren en el paso de intercambio de datos. La primera tiene un paso de intercambio con complejidad $\mathcal{O}(1)$ o constante y la segunda una complejidad $\mathcal{O}(X * Y)$. La Figura 5.5, permite comprobar las complejidades descritas, note que la implementación 1 para cada dimensión en la gráfica es cero (colores azul, amarillo y azul claro). Considerando estos resultados, es lógico pensar que la implementación 1 presentará menor tiempo de ejecución en sus versiones S y MP.

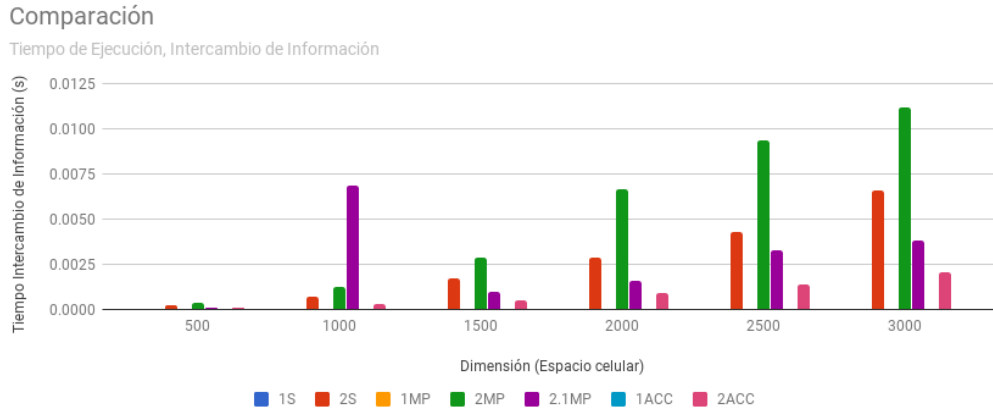


Figura 5.5: Gráfica Tiempo de Ejecución Paso de Intercambio de Información para cada Experimento

Al observar la Figura 5.6, podemos comprobar que 1_S y 2_S presentan el comportamiento esperado, es decir, la primera presenta un historial en tiempos de ejecución menor que la segunda. Además note en la Figura 5.7 que ambas 1_S y 2_S obtuvieron aproximadamente el mismo porcentaje del recuso CPU en el historial. A continuación, se comparan el tiempo de ejecución de las implementaciones 1_{MP} y 2_{MP} , note que para las dimensiones 500, 1000, 1500 y 2000, 1_{MP} no supera a 2_{MP} lo cual es un comportamiento no esperado. Pero, si observamos detenidamente la Figura 5.7, podremos observar que cuando se ejecutó el experimento 1_{MP} para las dimensiones

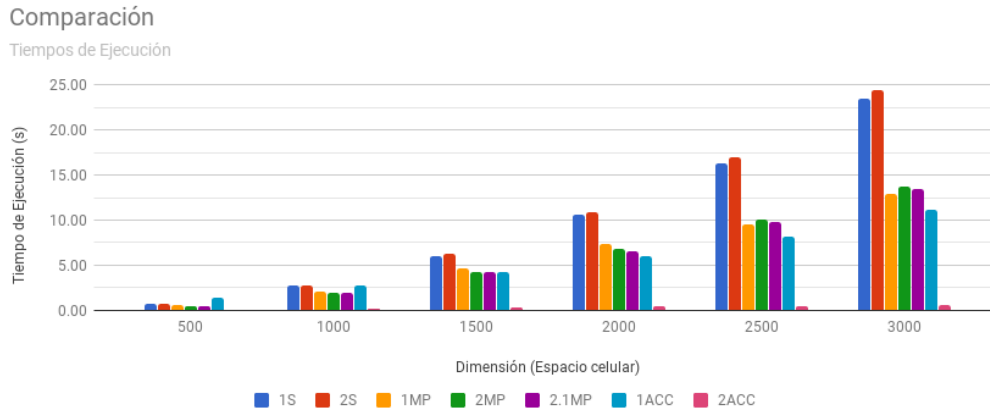


Figura 5.6: Gráfica Tiempo de Ejecución de cada Experimento

500, 1000, 1500 y 2000, obtuvo menor porcentaje de recurso CPU respecto al experimento 2_{MP} , de aquí que 2_{MP} tenga la oportunidad de terminar más rápido. Cuando 1_{MP} y 2_{MP} obtienen aproximadamente la misma cantidad de recurso CPU, se puede visualizar el comportamiento esperado en la Figura 5.6 (Tiempo de ejecución de cada experimento). Bajo las condiciones de los resultados mostrados, no es posible comparar las implementaciones 1 y 2. Es posible que el computador donde se realizaron los experimentos, no cumpliera con las condiciones ideales para llevar a cabo los mismos, es decir, estar libre en lo posible de procesos adicionales por parte del usuario y sistema operativo. Sin embargo, considerando los resultados individuales de cada implementación y la problemática que presenta la implementación 1 frente al paralelismo en GPU, se concluye que la implementación 2 es la más adecuada para implementar la función **evolve**. Por consiguiente, esta implementación en particular hará parte del esqueleto elemental usado en modelos de automata celular generados por el prototipo.

5.2. Otros Ambientes de Programación Paralela

En la literatura se documentada gran variedad de herramientas que permiten acelerar el proceso de programación en paralelo de autómatas celulares. La presente sección pretende mostrar a manera de comparación a CAMEL y Parallware, las cuales son herramientas que cercanas a lo que es el prototipo planteado en el presente trabajo.

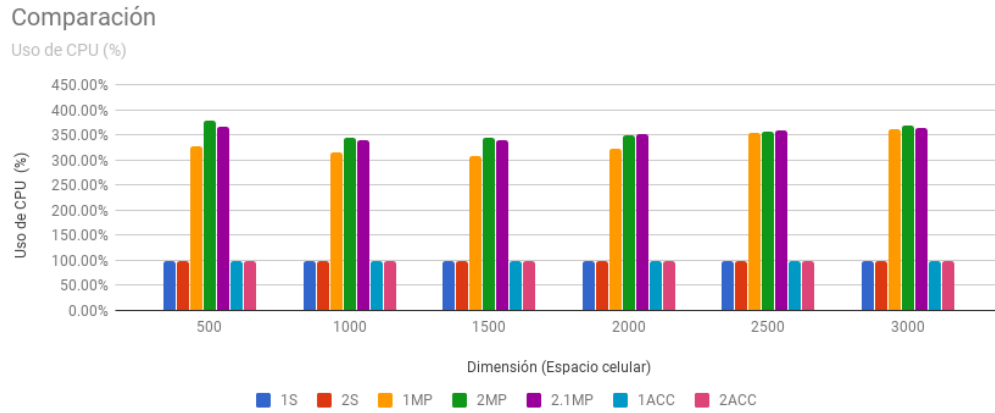


Figura 5.7: Gráfica Porcentaje de CPU Obtenido por cada Experimento. Al momento de ser ejecutado un proceso, el sistema operativo asigna un porcentaje de recursos disponibles en hardware a dicho proceso. Lo cierto es que aquel porcentaje depende de la cantidad de recursos disponibles en ese instante.

5.2.1. CAMEL/CARPET

CAMEL (por *Cellular Automata environMent for systEms modeLing*), es un ambiente de programación paralela interactivo que busca modelar fenómenos complejos por medio de autómatas celulares. Su diseño permite esconder tanto la complejidad de la programación paralela en el programa como la arquitectura de computador paralelo en el cual se ejecuta el modelo. Esta herramienta cuenta con un lenguaje de programación llamado CARPET usado específicamente para el modelado de autómatas, considerando que el usuario solo debe especificar la función de transición de estados de una celda y esta será replicada automáticamente en paralelo sobre el espacio celular del autómata [33]. Las características más relevantes de esta herramienta son descritas a continuación.

1. Interfaz de Usuario:

- Sección para visualización del sistema modelado durante una simulación.
- Sección para crear el modelo del sistema dinámico en lenguaje CARPET, solo se debe especificar las función de transición del sistema para una celda, luego esta función es replicada para procesar las celdas del sistema en paralelo.

2. Lenguaje de modelado: CARPET, un lenguaje parecido a C.
3. Tipo de Aplicación: Escritorio.
4. Computador Paralelo Soportado: computador paralelo MIMD de memoria distribuida, el cual se compone de *transputers*⁴ conectados a un nodo, donde cada *transputer* funciona como una unidad procesamiento [54, p. 56].
5. Ventajas:
 - El usuario tiene la oportunidad de visualizar la simulación del sistema.
 - Esconde el proceso de paralelización.
 - Ha sido usado para modelos de dinámica de fluidos, tráfico vehicular, procesos de combustión, entre otros [12] [54, p. 63].
 - Implementa un motor de paralelismo, el cual se encarga de balancear las cargas entre las unidades de procesamiento [6].
6. Desventajas:
 - Es una herramienta de escritorio. Lo cual hace difícil moverse en múltiples plataformas.
 - Fue diseñado para funcionar en una red de *transputers*. Se presume, no puede funcionar en arquitecturas de computadores actuales.
 - No hace uso de la GPU.

5.2.2. Parallware Trainer

Es un ambiente de programación paralela de uso comercial, el cual se basa en la **paralelización automática de código secuencial** de aplicaciones en lenguajes de programación C/C++ y FORTRAN. La paralelización de código secuencial es llevada a cabo por un *source-to-source compiler* llamado *parallware*, que es el encargado de analizar el código secuencial, y basado en un conjunto de heurísticas, intenta encontrar oportunidad de paralelismo principalmente en los ciclos del código dado. En esencia, la paralelización del código consiste en anotación del mismo con directivas de compilación de OpenMP y OpenACC. Las características más relevantes de esta herramienta son descritas a continuación [14] [5].

⁴Serie de microprocesadores de los años 1980, los cuales contaban con una memoria integrada y un canal de comunicación [37].

1. Interfaz de Usuario:

- Administrador de proyectos y archivos, permite administrar varios proyectos de código en lenguajes C/C++ y FORTRAN.
- Presenta un editor de código secuencial y editor de código paralelizado en OpenMP o OpenACC, donde el código paralelizado aparece de forma automática.
- Presenta dos consolas, tanto para ejecución de código secuencial, como para ejecución de código paralelizado.

2. Lenguaje de modelado: C/C++ y FORTRAN.

3. Tipo de Aplicación: Escritorio.

4. Computador Paralelo Soportado: MIMD y SIMD en sistemas de memoria compartida.

5. Ventajas:

- Permite comparar tiempos de ejecución, de código secuencial y paralelo.
- Brinda al usuario información acerca del proceso de paralelización del código secuencial.
- Hace uso tanto de las CPU, como GPU.
- Permite ajustar las directivas generadas de forma automática.

6. Desventajas:

- Es de uso comercial.
- Es una herramienta de escritorio. Lo cual hace difícil moverse en múltiples plataformas.

Capítulo 6

Conclusión

6.1. Resumen

En el presente trabajo de grado se desarrolló el prototipo de una herramienta WEB, para asistir el proceso de programación en paralelo de modelos de AC de propósito general. El prototipo, dada las características relevantes de un AC, retorna al usuario una plantilla en C99 de un modelo de un AC fácilmente paralelizable, el cual podrá presentar mejoras considerables aplicando paralelismo en CPU, y una mejora aun más significativa aplicando paralelismo en GPU. El presente capítulo concluye el trabajo de grado discutiendo los resultados de los experimentos llevados a cabo en la Sección 5 y presentando algunas iniciativas de trabajo futuro.

6.2. Resultados

- El prototipo garantiza que los modelos generados a partir de éste, tomarán ventaja del *hardware* disponible en un computador paralelo, de forma regular en CPU y de forma realmente significativa en GPU. Lo anterior, debido a que los modelos parten de la implementación 2, estudiada en la Sección 5.1. Donde se comprueba que a pesar de no ser la solución más intuitiva hablando en términos de programación secuencial, presenta igual complejidad a la implementación 1, y además mejores expectativas de paralelismo en GPU.
- La dramática mejora en GPU de la implementación 2_{ACC} se esperaba de antemano, considerando que el patrón *stencil* expone de forma explícita el paralelismo a nivel de datos del modelo de AC. Así mismo, éste nivel de paralelismo encaja perfectamente en la filosofía de diseño de la

GPU. Lo opuesto ocurre con las CPUs, soportan de forma eficiente el paralelismo a nivel de tareas, pero a nivel de datos con alto *overhead*, a razón de que deben crear gran cantidad de hilos para garantizar que cada hilo procese un dato, finalmente su diseño no está optimizado para ello.

- Cuando se desea lograr paralelismo en GPU, la cantidad de transferencias de datos desde y hacia la GPU, es un factor determinante para descartar el procesamiento de una implementación en particular en GPU, como es el caso de la implementación 1_{ACC} . Se llega a esta conclusión, a razón de que la lógica que maneja la arquitectura CUDA para administrar la ejecución de miles de hilos, garantiza la velocidad de ejecución de hilos, disminuyendo el número de obstáculos que se oponen al paralelismo. De esta forma, hace que los intentos de barra y/o semáforos en código no funcionen correctamente, provocando un comportamiento inesperado [61, p. 9]. Es por ello que OpenACC no implementa estos mecanismos.
- No siempre la implementación mas intuitiva hablando en términos de programación secuencial es la más apropiada en términos de paralelismo. Si bien es cierto, la implementación 1_S presenta mejor tiempo de ejecución que la implementación 2_S . Sin embargo, la implementación 2_{ACC} funciona mejor que la implementación 1_{ACC} en términos de paralelismo.
- Todo programa en presencia de paralelismo debe incurrir en un costo computacional, ya sea por administración de procesos adicionales (caso de la CPU), o por la transferencia de datos (caso de la GPU). Lo cierto, es que en ambos casos debe haber suficiente paralelismo, para justificar el costo computación en que se incurre.
- Al momento de realizar experimentos relacionados con el tiempo de ejecución de un programa, es fundamental que el computador donde se llevarán a cabo los experimentos esté libre en lo posible de procesos adicionales por parte del usuario y sistema operativo. Dado que lo anterior altera significativamente los resultados obtenidos en el proceso de experimentación, como se muestra en la Sección 5.1.6 del capítulo anterior.
- En cuanto al prototipo, el diseño de la aplicación WEB basado en servicios API REST permite el bajo acoplamiento de la aplicación, entonces al momento de fallar un componente, el funcionamiento de la

aplicación en general no es comprometido. Dada esta característica en diseño, el prototipo es capaz de responder correctamente ante un falla de sus componentes independientes.

- El despliegue en contenedores Docker, permite la portabilidad de la herramienta en plataformas que soporten Docker (Windows, y sistemas operativos basados en Linux), además permite que los servicios permanezcan en ambientes independientes, cada uno provisto de las herramientas necesarias para subsistir en su ambiente o contenedor.

6.3. Trabajo Futuro

- Desarrollar los requerimientos necesarios para que **Coder** sea una aplicación de software libre.
- Implementar en **Pragcc** un módulo de análisis estático de código C, con el fin de generar los meta datos necesarios para paralelizar el código analizado.
- Ampliar el conjunto de plantilla de patrones de programación paralela de la herramienta **Plantillas**, para brindar al usuario mayor variedad de plantillas fácilmente paralelizables.
- Habilitar el *source-to-source compiler* Dawncc y otros al conjunto de herramientas disponibles en esta aplicación. Para formar un ecosistema de herramientas (servicios) que puedan contribuir al programador, no solo en la programación de autómatas celulares si no también en la programación de otros modelos que soliciten programación de alto rendimiento.
- Extender la herramienta para que pueda enviar tareas a un Cluster de GPUs ó CPUs por medio de un servicio de ejecución, similar a un modelo de computación en la nube. Además, considerar un servicio de ejecución local, el cual pueda ser provisto por las librerías que necesita el programador para llevar a cabo la ejecución de su tareas.

Apéndice A

Listados de Código

A.1. Prototipo de Ambiente de Programación de Autómatas Celulares

A.1.1. Plantillas

Listado A.1: Ejemplo de una Plantilla en Lenguaje C99 con Sintaxis del Motor de Plantillas Jinja2. Observe que las variables se encuentran entre dobles llaves.

```
void evolve({{ lattice.type }} ** in){
    bool ** out = ({{ lattice.type }} **) malloc(RowDim*sizeof( {{ lattice.type }} *));
    for (int i=0; i<RowDim; ++i){
        out[i] = ({{ lattice.type }} *) malloc(ColDim*sizeof({{ lattice.type }}));
    }
    for (int g = 1; g <= Generations; ++g){
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                struct Neighborhood nbhd = neighborhood(in,i,j);
                out[i][j] = function(nbhd);
            }
        }
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                in[i][j] = out[i][j];
            }
        }
    }
    for (int i=0; i<RowDim; ++i) free(out[i]);
    free(out);
}
```

Listado A.2: Ejemplo de un Archivo *context.yml* para la plantilla *stencil*.

```
# The fields described here are mandatory to serve as a context
# to a cellular automata template.c
pattern_name: stencil
lattice:
  rowdim: 20
  coldim: 20
  type: bool
  neighborhood:
    up: [0,-1]
    down: [0,1]
    left: [-1,0]
    right: [0,1]
generations: 20
```

Listado A.3: Ejemplo Esqueleto Elemental de la Función **evolve** de la platilla *stencil*.

```
void evolve(bool ** in){
    bool ** out = (bool **) malloc(RowDim*sizeof( bool *));

    if( out == NULL ){
        perror("Not enough memory\n");
        exit(-1);
    }

    for (int i=0; i<RowDim; ++i){
        out[i] = (bool *) malloc(ColDim*sizeof(bool));
    }

    for (int g = 1; g <= Generations; ++g){
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                struct Neighborhood nbhd = neighborhood(in,i,j);
                out[i][j] = function(nbhd);
            }
        }

        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                in[i][j] = out[i][j];
            }
        }
    }

    for (int i=0; i<RowDim; ++i) free(out[i]);
    free(out);
}
```

A.1.2. Pragcc

Listado A.4: Ejemplo del Archivo *parallel.yml*.

```
version: 1.0
name: 'stencil'
description: |
    parallelization metadata corresponding to
    stencil.c c99 source code.
functs:
  all:
```

```

- main
- initialize
- function
- neighborhood
- evolve
parallel:
  evolve:
    mp:
      parallel_for:
        - nro: 2
          clauses:
            num_threads: 4
        - nro: 4
          clauses:
            num_threads: 4

```

Listado A.5: Ejemplo de un Código Paralelizado con *pragcc*.

```

void evolve(bool ** in){
    bool ** out = (bool **) malloc(RowDim*sizeof( bool *));

    if( out == NULL ){
        perror("Not enough memory\n");
        exit(-1);
    }

    for (int i=0; i<RowDim; ++i){
        out[i] = (bool *) malloc(ColDim*sizeof(bool));
    }

    for (int g = 1; g <= Generations; ++g){
        #pragma omp parallel for num_threads(4)
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                struct Neighborhood nbhd = neighborhood(in,i,j);
                out[i][j] = function(nbhd);
            }
        }

        #pragma omp parallel for num_threads(4)
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                in[i][j] = out[i][j];
            }
        }
    }

    for (int i=0; i<RowDim; ++i) free(out[i]);
    free(out);
}

```

A.2. Estudio Modelo Autómata Celular

Listado A.6: Implementación 1 Secuencial, 1_S.

```

void evolve(bool ** in){
    bool ** out = (bool **) malloc(RowDim*sizeof( bool *));

    if( out == NULL ){
        perror("Not enough memory\n");
        exit(-1);
    }

    for (int i=0; i<RowDim; ++i){
        out[i] = (bool *) malloc(ColDim*sizeof(bool));
    }
}

```

```

// Paso de procesameinto del espacio celular
for (int g = 1; g <= Generations; ++g){
    for (int i = 0; i < RowDim; ++i){
        for (int j = 0; j < ColDim; ++j){
            struct Neighborhood nbhd = neighborhood(in,i,j);
            out[i][j] = function(nbhd);
        }
    }

    // Paso de intercambio de informacion
    bool ** temp = in;
    in = out;
    out = temp;
}

for (int i=0; i<RowDim; ++i) free(out[i]);
free(out);
}

```

Listado A.7: Implementación 1 paralelizada con directivas OpenMP, 1_{MP}

```

void evolve(bool ** in){
    ...
    for (int g = 1; g <= Generations; ++g){
        #pragma omp parallel for num_threads(4)
        for (int i = 0; i < RowDim; ++i){
            for (int j = 0; j < ColDim; ++j){
                struct Neighborhood nbhd = neighborhood(in,i,j);
                out[i][j] = function(nbhd);
            }
        }

        bool ** temp = in;
        in = out;
        out = temp;
    }
    ...
}

```

Listado A.8: Implementación 1 paralelizada con directivas OpenACC, 1_{ACC}

```

void evolve(bool ** in){
    ...
    for (int g = 1; g <= Generations; ++g){
        #pragma acc data copy(in[0:RowDim][0:ColDim],out[0:RowDim][0:ColDim])
        #pragma acc parallel loop gang
        for (int i = 0; i < RowDim; ++i){
            #pragma acc loop vector
            for (int j = 0; j < ColDim; ++j){
                struct Neighborhood nbhd = neighborhood(in,i,j);
                out[i][j] = function(nbhd);
            }
        }

        bool ** temp = in;
        in = out;
        out = temp;
    }
    ...
}

```

Listado A.9: Implementación 2 Secuencial, 2_S . Basada en la implementación del método *Jacobi Iteration* [11].

```

void evolve(bool ** in){
    bool ** out = (bool **) malloc(RowDim*sizeof( bool *));

    if( out == NULL ){
        perror("Not enough memory\n");
        exit(-1);
    }

    for (int i=0; i<RowDim; ++i){
        out[i] = (bool *) malloc(ColDim*sizeof(bool));
    }
}

```

```

// Paso de procesameinto del espacio celular
for (int g = 1; g <= Generations; ++g){
  for (int i = 0; i < RowDim; ++i){
    for (int j = 0; j < ColDim; ++j){
      struct Neighborhood nbhd = neighborhood(in,i,j);
      out[i][j] = function(nbhd);
    }
  }

  // Paso de intercambio de informacion
  for (int i = 0; i < RowDim; ++i){
    for (int j = 0; j < ColDim; ++j){
      in[i][j] = out[i][j];
    }
  }
}

for (int i=0; i<RowDim; ++i) free(out[i]);
free(out);
}

```

Listado A.10: Implementación 2 paralelizada con directivas OpenMP, 2_{MP} .

```

void evolve(bool ** in){
  ...
  for (int g = 1; g <= Generations; ++g){
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < RowDim; ++i){
      for (int j = 0; j < ColDim; ++j){
        struct Neighborhood nbhd = neighborhood(in,i,j);
        out[i][j] = function(nbhd);
      }
    }

    for (int i = 0; i < RowDim; ++i){
      for (int j = 0; j < ColDim; ++j){
        in[i][j] = out[i][j];
      }
    }
  }
  ...
}

```

Listado A.11: Variación implementación $2,1_{MP}$

```

void evolve(bool ** in){
  ...
  for (int g = 1; g <= Generations; ++g){
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < RowDim; ++i){
      for (int j = 0; j < ColDim; ++j){
        struct Neighborhood nbhd = neighborhood(in,i,j);
        out[i][j] = function(nbhd);
      }
    }

    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < RowDim; ++i){
      for (int j = 0; j < ColDim; ++j){
        in[i][j] = out[i][j];
      }
    }
  }
  ...
}

```

Listado A.12: Implementación 2 paralelizada con directivas OpenACC, 2_{ACC}

```

void evolve(bool ** in){
  ...
  #pragma acc data copy(in[0:RowDim][0:ColDim]), create(out[0:RowDim][0:ColDim])
  {
    for (int g = 1; g <= Generations; ++g){
      #pragma acc parallel loop gang
      for (int i = 0; i < RowDim; ++i){
        #pragma acc loop vector

```

```

        for (int j = 0; j < ColDim; ++j){
            struct Neighborhood nbhd = neighborhood(in,i,j);
            out[i][j] = function(nbhd);
        }
    }

#pragma acc parallel loop
    for (int i = 0; i < RowDim; ++i){
        #pragma acc loop
        for (int j = 0; j < ColDim; ++j){
            in[i][j] = out[i][j];
        }
    }
}
...
}

```

Bibliografía

- [1] 2.3 boundary conditions. <http://www.jcasim.de/main/node6.html>. (Accessed on 10/29/2017).
- [2] 4. the evolution of computational power of gpus measured in gflops... https://www.researchgate.net/figure/271832389_fig4_Figure-14-The-evolution-of-computational-power-of-GPUs-measured-in-GFLOPs-against (Accessed on 11/04/2017).
- [3] Análisis de diferencias finitas usando la aproximación de expansión de la serie de taylor. ultimo. <https://es.scribd.com/document/358466680/Analisis-de-Diferencias-Finitas-Usando-La-Aproximacion-de-Expansion-de-La-Serie-de-Taylor-Ultimo>. (Accessed on 11/02/2017).
- [4] Automaton — definition of automaton by merriam-webster. <https://www.merriam-webster.com/dictionary/automaton>. (Accessed on 03/23/2017).
- [5] brochure_parallware_trainer.pdf. https://www.parallware.com/wp-content/uploads/2017/04/brochure_parallware_trainer.pdf. (Accessed on 11/10/2017).
- [6] Caboto. http://staff.icar.cnr.it/spezzano/intercab/l2_camel.htm. (Accessed on 11/10/2017).
- [7] Flynn's taxonomy - wikipedia. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy. (Accessed on 11/03/2017).
- [8] Introduction to openacc on x86 cpu and gpu - youtube. <https://www.youtube.com/watch?v=KgMJzmqenuc>. (Accessed on 11/06/2017).
- [9] Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/. (Accessed on 11/03/2017).

- [10] Li-jessica.pdf. <https://math.mit.edu/research/highschool/primes/materials/2014/Li-Jessica.pdf>. (Accessed on 06/15/2017).
- [11] An openacc example (part 1). <https://devblogs.nvidia.com/openacc-example-part-1/>. (Accessed on 01/20/2018).
- [12] Parallel cellular automata. <http://si.deis.unical.it/~talía/CA.html>. (Accessed on 11/10/2017).
- [13] Parallel programming model. <http://wiki.c2.com/?ParallelProgrammingModel>. (Accessed on 11/05/2017).
- [14] Parallware trainer — learning parallel programming is now different and easy! <https://www.parallware.com/>. (Accessed on 11/10/2017).
- [15] pp3tk.png (641×667). <https://i.stack.imgur.com/pp3tK.png>. (Accessed on 01/15/2018).
- [16] Sistemas complejos, sistemas dinámicos y redes complejas - fernando sancho caparrini. <http://www.cs.us.es/~fsancho/?e=64>. (Accessed on 10/26/2017).
- [17] Sustainability — free full-text — the improved cellular automata and its application in delineation of urban spheres of influence — html. <http://www.mdpi.com/2071-1050/6/12/8931/htm>. (Accessed on 06/15/2017).
- [18] Understanding throughput-oriented architectures — november 2010 — communications of the acm. <https://cacm.acm.org/magazines/2010/11/100622-understanding-throughput-oriented-architectures/fulltext>. (Accessed on 11/04/2017).
- [19] Von neumann architecture - computer science gcse guru. <https://www.computerscience.gcse.guru/theory/von-neumann-architecture>. (Accessed on 11/03/2017).
- [20] What is the difference between hyper threading & multi-core technology? — chron.com. <http://smallbusiness.chron.com/difference-between-hyper-threading-multicore-technology-58549.html>. (Accessed on 01/15/2018).
- [21] Enrique Torrella Alcaraz. *Transmisión de calor: procesos y equipos intercambiadores*. Universitat Politècnica de València, 2015.

- [22] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.
- [23] Chopard Bastien and Droz Michel. Cellular automata modeling of physical systems. *Cellular automata modeling of physical systems*, 1998.
- [24] William R Buckley. A catalog of self-replicating cellular automata. 2007.
- [25] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [26] Necia G Cooper. From turing and von neumann to the present. *Los Alamos Science*, 9:22–7, 1983.
- [27] L Hernández Encinas, S Hoya White, A Martin del Rey, and G Rodríguez Sanchez. Modelling forest fire spread using hexagonal cellular automata. *Applied mathematical modelling*, 31(6):1213–1227, 2007.
- [28] Ryno Fourie. *A parallel cellular automaton simulation framework using CUDA*. PhD thesis, Stellenbosch: Stellenbosch University, 2015.
- [29] RA Freitas and WP Gilbreath. Advanced automation for space missions: Proceedings of the 1980 nasa. *ASEE summer study*.
- [30] Toru Fujita, Daigo Nishikori, Koji Nakano, and Yasuaki Ito. Efficient gpu implementations for the conway’s game of life. In *Computing and Networking (CANDAR), 2015 Third International Symposium on*, pages 11–20. IEEE, 2015.
- [31] Hipólito Gómez-Sousa, Manuel Arenaz, Óscar Rubiños-López, and José Ángel Martínez-Lorenzo. Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments. In *Antennas and Propagation (EuCAP), 2015 9th European Conference on*, pages 1–6. IEEE, 2015.
- [32] Howard Gutowitz. *Cellular automata: theory and experiment*. MIT press, 1991.
- [33] Wei Huang. General purpose cellular automata programming. 2002.
- [34] Tim J Hutton. Codd’s self-replicating computer. *Artificial Life*, 16(2):99–117, 2010.

- [35] María del Carmen Ibarra. La ecuación de calor de fourier: resolución mediante métodos de análisis en variable real y en variable compleja. *UTN Facultad Regional Resistencia: II Jornadas de Investigación en Ingeniería del NEA y Países Limítrofes*, Consultado el, 10:3, 2012.
- [36] Andrew Ilachinski. *Cellular automata: a discrete universe*. World Scientific Publishing Co Inc, 2001.
- [37] Allen Kent and James G Williams. *Encyclopedia of Computer Science and Technology: Volume 39-Supplement 24-Entity Identification to Virtual Reality in Driving Simulation*, volume 39. CRC Press, 1998.
- [38] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [39] Uttam Kumar, Chiranjit Mukhopadhyay, and TV Ramachandra. Cellular automata calibration model to capture urban growth. *Boletín Geológico y Minero*, pages 285–99, 2004.
- [40] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.
- [41] Sven Maerivoet and Bart De Moor. Cellular automata models of road traffic. *Physics Reports*, 419(1):1–64, 2005.
- [42] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004.
- [43] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.
- [44] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):13, 2017.
- [45] C Nvidia. Nvidia’s next generation cuda compute architecture: Fermi. *Comput. Syst*, 26:63–72, 2009.
- [46] C Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. *Whitepaper*, 2012.

- [47] Sang Il Pak and Tomohisa Hayakawa. Forest fire modeling using cellular automata and percolation threshold analysis. In *American Control Conference (ACC), 2011*, pages 293–298. IEEE, 2011.
- [48] Fethi Rabhi. *Patterns and skeletons for parallel and distributed computing*. Springer Science & Business Media, 2003.
- [49] Clifford A Reiter. A local cellular model for snow crystal growth. *Chaos, Solitons & Fractals*, 23(4):1111–1119, 2005.
- [50] Joel L Schiff. *Cellular automata: a discrete view of the world*, volume 45. John Wiley & Sons, 2011.
- [51] Marcin Serebnyski and Pascal Bouvry. Block encryption using reversible cellular automata. In *International Conference on Cellular Automata*, pages 785–792. Springer, 2004.
- [52] Claude E Shannon. Von neumann’s contributions to automata theory.
- [53] Daniel Shiffman, Shannon Fry, and Zannah Marsh. *The nature of code*. D. Shiffman, 2012.
- [54] Giandomenico Spezzano and Domenico Talia. A high-level cellular programming model for massively parallel processing. In *High-Level Programming Models and Supportive Environments, 1997. Proceedings., Second International Workshop on*, pages 55–63. IEEE, 1997.
- [55] William Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 9 edition, 2000.
- [56] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs’s journal*, 30(3), 2005.
- [57] Gianluca Tempesti. A new self-reproducing cellular automaton capable of construction and computation. In *European Conference on Artificial Life*, pages 555–563. Springer, 1995.
- [58] Philip D Tetlow. *The Web’s awake: An introduction to the field of Web Science and the concept of Web life*. John Wiley & Sons, 2007.
- [59] Massimo Torquati, Koen Bertels, Sven Karlsson, and François Pacull. *Smart multicore embedded systems*. Springer, 2014.
- [60] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):1–23, 1966.

- [61] Michael Wolfe. The openacc application programming interface, 2013.
- [62] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601, 1983.
- [63] Xin-She Yang and Y Young. Cellular automata, pdes, and pattern formation. *arXiv preprint arXiv:1003.1983*, 2010.
- [64] H Yohannes. A review on relationship between climate change and agriculture. *Earth Science and Climate Change*, 7(2):335, 2016.
- [65] Ada Yuen and Robin Kay. Applications of cellular automata, 2009.