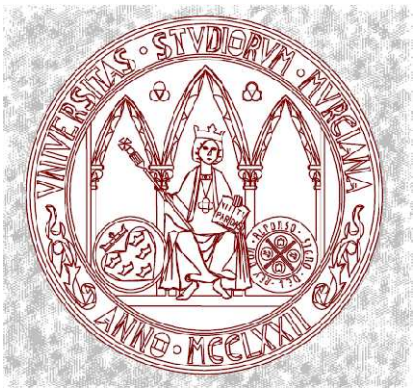

Universidad de Murcia
Facultad de Informática
Murcia. España

Proyecto Final de Carrera

Análisis y Diseño de un Simulador de Sistemas P con Membranas Activas en CUDA

Ginés David Guerrero Hernández
Ingeniería Informática
gines.guerrero@ditec.um.es

Directores:
Jose Manuel García Carrasco
Jose María Cecilia Canal

7 de septiembre de 2009

A mi familia y Guadalupe
por estar siempre a mi lado.

Nota legal

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

He de expresar mi profundo agradecimiento a todas aquellas personas que de una o otra manera han estado a mi lado durante esta etapa de mi vida.

Al Grupo de Investigación en Computación Natural de la Universidad de Sevilla, en especial a Mario e Ignacio, por por su colaboración con la Universidad de Murcia y el buen trato recibido durante todo el año, sobre todo, durante mi corta estancia en Sevilla.

A Chema y Miguel Ángel, por la gran ayuda que me han ofrecido durante el desarrollo del proyecto y por los ratos que hemos pasado juntos, no sólo en la universidad, sino también fuera de ella. Y como no, por ser los responsables de que el trabajo realizado haya tenido tantos frutos.

A Jose Manuel, por su confianza depositada en mí y por ofrecerse a ser mi director de proyecto, dándome la oportunidad de realizar este trabajo, ya que ha sido una grata experiencia su realización y me ha aportado multitud de conocimientos que me han introducido en el mundo de la investigación.

A mis compañeros y amigos de la universidad, en especial a Eneko, Juanlu y Pedro, porque sin todos vosotros, vuestros consejos, vuestra ayuda, y como no, las *actividades extra-universitarias*, no estaría escribiendo estas líneas.

Y por supuesto a mi familia y Guadalupe, porque sin ellos no sería como soy.

Gracias.

Resumen

Las Unidades de Procesamiento Gráfico (GPUs) se han consolidado como coprocesadores masivamente paralelos, capaces de ejecutar aplicaciones de propósito general. Permiten a los programadores utilizar distintos niveles de paralelismo para obtener mayor rendimiento de sus aplicaciones. La gran capacidad de cómputo y el bajo coste de las GPUs hacen pensar en estas como una buena alternativa a otros enfoques paralelos.

Los *sistemas P* o sistemas de membranas son dispositivos teóricos inspirados en el comportamiento de las células vivas. Estos sistemas nos brindan nuevos modelos computacionales, así como la posibilidad de modelar computacionalmente sistemas en el marco de la Biología de Sistemas. Los sistemas P tienen una naturaleza masivamente paralela y no determinista.

En este trabajo evaluamos la GPU como la arquitectura subyacente para simular la clase de sistemas P reconocedores con membranas activas. Analizamos el rendimiento de tres simuladores de sistemas P implementados en diversas arquitecturas: sobre CPU, sobre CPU-GPU, y por último, sobre GPU. Como *benchmarks* hemos usado dos sistemas P: el primero es un *benchmark* sintético específicamente diseñado para explotar el paralelismo de la GPU, y el segundo es un caso real: el problema clásico de las N-Reinas. Demostramos que la GPU está mejor adaptada que la CPU para simular estos sistemas P debido a su naturaleza masivamente paralela.

Palabras clave

GPGPUs, CUDA, Sistemas P, Computación de Membranas.

Abstract

GPUs have been consolidated as a massively data-parallel coprocessor to develop many general purpose computations, and enable developers to utilize several levels of parallelism to obtain higher performance of their applications. The massively parallel nature of certain computations leads to use GPUs as an underlying architecture, becoming a good alternative to other parallel approaches.

P systems or membrane systems are theoretical devices inspired in the way that living cells work, providing computational models and a high level computational modeling framework for biological systems. P systems are massively parallel distributed and non-deterministic systems.

In this document, we evaluate the GPU as the underlying architecture to simulate the class of recognizer P systems with active membranes. We analyze the performance of three simulators implemented on CPU, GPU-GPU and GPU respectively. We compare them using a presented P system as a benchmark, showing that the GPU is better suited than the CPU to simulate those P systems due to its massively parallel nature.

Key Words

GPGPUs, CUDA, P System, Membrane Computing.

1

Introducción

Este proyecto ha tenido lugar gracias a la incorporación de Miguel Ángel Martínez del Amor (ex-alumno de la Universidad de Murcia) a la Universidad de Sevilla, propiciando así la colaboración entre el Grupo de Arquitectura y Programación Paralela (GACOP) del Departmento de Ingeniería y Tecnología de Computadores (DITEC) de la Universidad de Murcia y el Grupo de Investigación en Computación Natural (RGNC) del Departamento de Ciencias de la Computación e Inteligencia Artificial (CCIA) de la Universidad de Sevilla.

Este primer capítulo nos sirve para mostrar el porqué ha surgido la necesidad de emprender este proyecto y que objetivos son los que hemos llevado a cabo durante el desarrollo del trabajo.

También mostraremos la estructura de este documento y cual será el contenido de cada uno de los capítulos.

1.1. Motivación del Proyecto

La computación de membranas (o computación celular) es una rama emergente en la Computación Natural que fue introducida por Gh. Păun [23]. La principal idea está en considerar los procesos bioquímicos que tienen lugar dentro de las células vivas desde un punto de vista computacional, obteniendo un nuevo modelo no determinístico y masívamente paralelo

de computación usando máquinas celulares. Estos dispositivos son conocidos como *sistemas P*, los cuales tienen una estructura de membranas similar a la de las células. Estas membranas delimitan regiones, donde se almacenan multiconjuntos de objetos que evolucionan de acuerdo a unas reglas dadas de forma masivamente paralela, siguiendo un esquema síncrono y no determinístico.

Los estudios más recientes en la computación de membranas [5] se han centrado en la capacidad computacional y en la eficiencia de los sistemas P, pero últimamente se usan con éxito en el modelado de procesos biológicos, siendo complementarios y una alternativa a los enfoques clásicos usados en la Biología de Sistemas (ODEs, Redes de Petri, etc.).

Hay diferentes modelos computacionales que están siendo estudiados en esta área: sistemas P de transición, sistemas P con membranas activas, sistemas P probabilísticos, sistemas P estocásticos, etc. Todos estos modelos están teóricamente diseñados para resolver distintos problemas. Por ejemplo, puede verse en [3] como para el modelado de un ecosistema relacionado con el quebrantahuesos en los Pirineos se utiliza un sistema P probabilístico. Este modelo es simulado utilizando un simulador de sistemas P secuencial, que permite analizar la evolución del ecosistema partiendo de diferentes condiciones iniciales, obteniendo resultados cercanos a los reales. Por otra parte, en [4] se introduce el modelo de la *apoptosis* inducido por la proteína FAS, concluyendo que los sistemas de membranas son una herramienta útil en los Sistemas Biológicos.

En este trabajo nos centramos en la capacidad de los sistemas P de construir un espacio de trabajo exponencial (expresado en número de membranas y objetos) en tiempo polinomial (incluso lineal). En este sentido, el modelo de sistemas P con membranas abstrae el modo de obtener nuevas membranas a través del proceso de *mitosis* (división de membranas). Este modelo ha sido usado con éxito para diseñar soluciones a los conocidos problemas **NP**-completos, como son SAT [27] y *Suma de Subconjuntos* [25].

Hasta ahora no ha sido posible sintetizar ninguna implementación real de los sistemas P, por lo que la computación y análisis de estos dispositivos se realiza mediante simuladores. Así pues, los simuladores de sistemas P son herramientas que ayudan a los investigadores a obtener resultados de un determinado modelo. Estos simuladores deben de ser lo más eficientes posibles para poder manejar tamaños grandes de instancias de problemas. La naturaleza masivamente paralela de los sistemas P nos lleva a buscar una tecnología masivamente paralela donde el simulador puede ejecutarse de forma eficiente.

Las nuevas generaciones de unidades de procesamiento gráfico (GPUs) son procesadores masivamente paralelos que pueden soportar varios miles de hilos concurrentes. Muchas aplicaciones de propósito general han sido portadas a esta plataforma debido a su gran rendimiento [32] [29]. Las GPUs de NVIDIA actuales, que pueden contener unos 240 procesadores escalares por chip [17], están optimizadas para realizar cálculos de propósito general, y son programadas mediante C y Compute Unified Device Architecture (CUDA) [42] [20].

Se puede ver como los sistemas P y las GPUs tienen una naturaleza masivamente paralela, de ahí surge la motivación de este proyecto: el hecho de comprobar como se comportan las GPUs para desarrollar un simulador para sistemas P con membranas activas.

1.2. Objetivos

A lo largo de este trabajo evaluaremos la GPU como arquitectura subyacente para la simulación masivamente paralela de los sistemas P con membranas activas.

Analizaremos tres simuladores, que serán ejecutados en distintas plataformas. El primer simulador será implementado completamente sobre una CPU, el segundo simulador será ejecutado parcialmente en la GPU, y por último un tercer simulador que lleva a cabo la simulación completa en la GPU.

Para poder llevar a cabo un análisis de los tres simuladores empleamos dos *benchmarks*. El primero de ellos es un sistema P que está diseñado específicamente para poder someter a los simuladores a situaciones teóricas donde haya gran paralelismo. Por otro lado, el segundo *benchmark* está enfocado para ver como se comportan los simuladores ante un caso real, en concreto veremos como se comportan con un sistema P que resuelve el problema de las N-Reinas, el cual no está tan adaptado a la arquitectura de la GPU.

Como conclusión, hemos obtenido que la GPU es más adecuada para simular estos sistemas P cuando el paralelismo crece.

1.3. Contenido de la Memoria

Esta memoria está estructurada en ocho capítulos, cuyos contenidos pasamos a describir sucintamente.

El capítulo 2 está dedicado a la presentación de la Computación celular con membranas, donde veremos en profundidad los sistemas P reconocedores con membranas activas.

En el capítulo 3 hacemos una introducción a CUDA, mostrando algunos conceptos sobre programación en GPUs, donde para su mejor entendimiento se examinará previamente la arquitectura de dichos dispositivos.

Veremos algunos aspectos genéricos de la simulación de sistemas P con membranas activas en el capítulo 4, presentando un simulador secuencial ya existente.

En los capítulos 5 y 6 se explican los simuladores paralelos, centrándonos en cuestiones de diseño e implementación. Primero se muestra el simulador que está parcialmente desarrollado en la GPU, y finalmente el que está implementado para su completa ejecución en la GPU.

En el capítulo 7 se hace un análisis del funcionamiento de los tres simuladores, para lo cual previamente se mostrarán los sistemas P que se utilizarán como entrada para realizar las pruebas.

Para finalizar, en el capítulo 8 se detallan algunas conclusiones que se pueden extraer de los resultados obtenidos y se presentan una serie de ideas para trabajo futuro.

2

Computación Celular con Membranas

En este capítulo daremos una introducción a los sistemas P, profundizando en los sistemas P con Membranas Activas.

Un repaso similar aunque algo más formalizado (dirigido a lectores familiarizados con el área de la Computación Celular con Membranas) puede encontrarse en [24].

2.1. Introducción

Al buscar en la Naturaleza viva esquemas de comportamiento o de organización que nos puedan inspirar desde el punto de vista computacional, parece razonable centrar nuestra atención en la célula. Es la unidad más pequeña en cuanto a forma de vida se refiere, y la actividad que ocurre en su interior puede ser considerada, en cierto sentido, como procesamiento de información, en cierto sentido. Esto nos conduce al concepto de Computación Celular con Membranas.

En esta línea, para llegar a los *sistemas de membranas* (también conocidos como *sistemas celulares* o *sistemas P*), introducidos a finales de 1998 por Gh. Păun [23], hay que abstraer el complicado mundo interior de la célula simplificándolo hasta reducirlo a un sencillo sistema

de compartimentos. Al fin y al cabo, toda célula consta en su interior de varias vesículas, dentro de las cuales tienen lugar reacciones que pueden transformar los compuestos químicos presentes, o bien pueden originar un flujo de materiales de un compartimento a otro, atravesando la membrana que los separa.

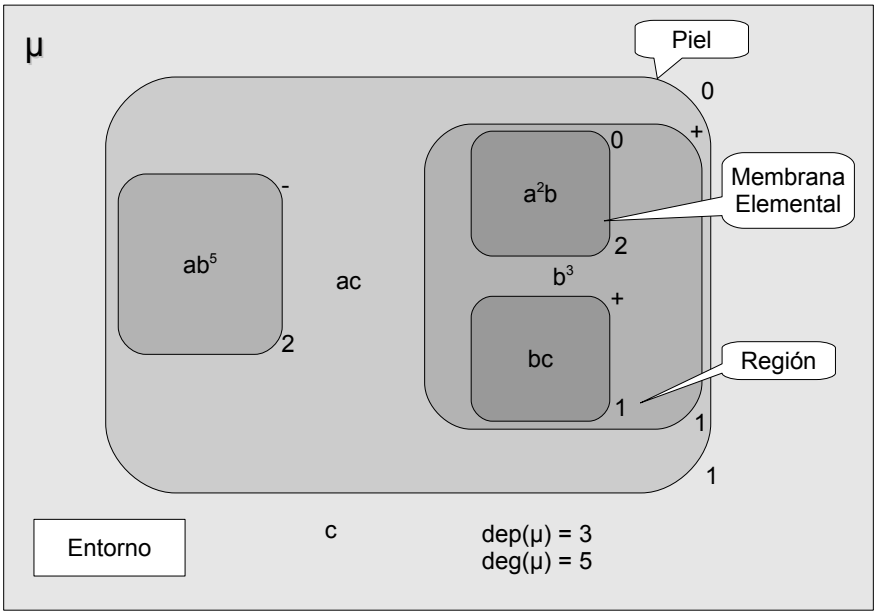

Figura 2.1: Sistema de computación celular con membranas

Básicamente, como ilustra la Figura 2.1, un P sistema consta de un conjunto de *membranas*, organizado jerárquicamente en una *estructura de membranas*. Existe una *membrana piel* que engloba todas las demás, separando al sistema del *entorno externo*. Las membranas que no contienen otras membranas en su interior se denominan *membranas elementales*. Las *regiones* delimitadas por las membranas (es decir, el espacio acotado por una membrana y las membranas inmediatamente interiores, si existe alguna) pueden contener ciertos *objetos* que, en general, se permite que aparezcan repetidos (y que son abstracciones de los compuestos químicos presentes en las vesículas). Mediante la aplicación de determinadas *reglas de evolución* asociadas a las membranas (que representan las reacciones químicas entre los distintos compuestos), estos objetos pueden transformarse en otros, e incluso pueden pasar de una región a otra adyacente, atravesando la membrana que las separa.

2.2. Sistemas P de Transición

En la versión más simple de este modelo, los sistemas P de transición, se considera que las reglas de evolución están asociadas a las membranas, lo cual representa que en distintas vesículas de una célula pueden tener lugar diferentes tipos de reacciones químicas. Además, se considera que las reglas pueden incorporar *indicadores de destino*, y así los productos de la aplicación de la regla pueden permanecer en la misma región en la que se encontraban los reactivos (*here*), o bien pueden ser enviados a otra región, tanto hacia afuera (*out*), a

la membrana inmediatamente exterior; como hacia dentro (*in*), a *una* membrana inmediatamente interior. Los casos extremos se dan cuando un objeto en la membrana `piel` recibe la indicación *out*, con lo cual es expulsado al entorno externo, no pudiendo ser recuperado por el sistema, y cuando en una membrana elemental tenemos una regla que asigna a uno de los productos la indicación *in*, en cuyo caso dicha regla no se puede ejecutar.

Obsérvese que las membranas del sistema actúan tanto de separadores como de canales de comunicación entre regiones, pero son sujetos pasivos en el proceso, la responsabilidad del funcionamiento del sistema recae exclusivamente sobre las reglas de evolución.

Intuitivamente, el significado de una regla de evolución $u \rightarrow v$ es que los objetos de u se transforman en los objetos que "aparecen" en v, de tal manera que los nuevos objetos pueden dirigirse a otras membranas, en función de lo que marquen los correspondientes indicadores de destino.

Además, las reglas se aplican de manera *maximal* en cada paso, en el siguiente sentido: tras la aplicación de las reglas, no deben quedar objetos en ninguna membrana que pudieran haber evolucionado por la acción de alguna regla asociada a esa membrana y que sea aplicable.

A continuación vamos a precisar lo que entenderemos por un *paso de transición*. Dadas dos configuraciones, C y C', decimos que C' se obtiene a partir de C en un paso de transición si la segunda configuración es el resultado de aplicar sobre la primera, en paralelo (de manera simultánea), y sobre todas las membranas que aparecen en C a la vez, algunas de las reglas de evolución asociadas a dichas membranas, respetando la condición de aplicación de manera maximal en el sentido antes indicado.

Dado que para cada configuración suele existir más de un (multi)conjunto de reglas aplicables, los pasos de transición se ejecutan de manera *no determinista*; es decir, a lo largo de la computación, una configuración del sistema puede tener más de una configuración siguiente. Si todas las computaciones con la misma configuración inicial (y con la misma entrada) devuelven el mismo resultado (tienen la misma salida), diremos que el sistema P es confluente. Si no es posible obtener ninguna configuración siguiente, porque ninguna regla es aplicable en ninguna membrana de la configuración, diremos que la configuración es *de parada* o *final*.

2.3. Sistemas P Symport/Antiport

Existe un modelo de sistemas P que hace especial énfasis en el papel de las membranas como canales de comunicación, y de hecho no se permite sustituir multiconjuntos de objetos por otros objetos, sino únicamente mover los objetos de una región a otra adyacente.

Son los llamados sistemas P con reglas de tipo *symport/antiport* [36] que se denotan mediante tuplas $(i, u/w, j)$, con la siguiente interpretación: si la región i contiene al multiconjunto u y la región j contiene al multiconjunto w, entonces u y w serán intercambiados de lugar.

Si alguno de los dos multiconjuntos es vacío, se habla de regla *symport*, en la que varios objetos que se encuentran en la misma región cooperan para moverse a otra. En caso contrario,

se habla de regla *antiport*, en la que objetos que se encuentran en regiones diferentes cooperan a través de la membrana que los separa para cruzarla en sentidos opuestos.

2.4. Sistemas P con Membranas Activas

En los sistemas P de transición, presentados anteriormente, el número de membranas puede permanecer invariable o reducirse durante la computación (debido a la posible disolución de alguna membrana). Una alternativa natural consiste en permitir que dicho número pueda también aumentar (lo que está justificado desde el punto de vista biológico, por ejemplo, mediante la creación o división de membranas). Esta idea fue explorada por Gh. Păun [37] dando origen a un nuevo modelo: los *sistemas celulares con membranas activas*.

Las diferencias fundamentales con el modelo de transición son que las membranas tienen asociada además de la etiqueta una *carga eléctrica* (que puede ser positiva, negativa o neutra), y que no hay cooperación en las reglas (es decir, que las reglas son lanzadas por un único objeto).

Formalmente un sistema P con membranas activas está definido por la construcción: $\Pi = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$, donde:

- $m \geq 1$ indica el grado inicial del sistema.
- O es el alfabeto de los objetos.
- H es un conjunto finito de etiquetas para las membranas.
- μ es una estructura de membranas, compuesta por m membranas etiquetadas con los elementos de H.
- $\omega_1, \ldots, \omega_m$ son cadenas sobre O, que describen los multiconjuntos de objetos situados en las m regiones de μ.
- R es un conjunto finito de reglas

En concreto, las reglas utilizadas en el modelo de membranas activas con división pueden ser de uno de los tipos siguientes:

(*a*) $[a \rightarrow v]_l^\alpha$ (reglas de evolución de objetos). Es una regla interna que no modifica nada fuera de la membrana l, ni tampoco la carga eléctrica de ésta. Su ejecución produce la sustitución de un objeto a por un multiconjunto v, dentro de una membrana etiquetada por l y con carga α.

(*b*) $[a]_l^\alpha \rightarrow b[\ ]_l^\beta$ (reglas de tipo *send-out*). Un objeto a puede abandonar una membrana etiquetada por l y con carga α pasando a la membrana padre, pudiendo transformarse en un nuevo objeto b en el proceso. Al mismo tiempo, la ejecución de la regla puede producir un cambio en la carga de la membrana (de α a β), pero conservándose la etiqueta.

(*c*) $a[\ ]_l^\alpha \rightarrow [b]_l^\beta$ (reglas de tipo *send-in*). Un objeto a puede penetrar en una membrana etiquetada por l y con carga α desde la región inmediatamente superior (es decir, desde la membrana padre), pudiendo transformarse en uno nuevo b en el proceso y, al mismo tiempo, cambiar la carga de la membrana atravesada (de α a β), pero conservándose la etiqueta.

(*d*) $[a]_l^\alpha \rightarrow b$, $l \neq skin$ (reglas de disolución). Su ejecución produce la transformación de un objeto a dentro de una membrana etiquetada por l y con carga α en otro nuevo b. Además, simultáneamente, la membrana es disuelta (la `piel` no se puede disolver).

(*e*) $[a]_l^\alpha \rightarrow [b]_l^\beta [c]_l^\gamma$, $l \neq skin$ (reglas de 2-división para membranas elementales). Un objeto a situado en una membrana etiquetada por l y con carga α puede provocar la división de dicha membrana en otras dos, con la misma etiqueta pero, eventualmente, con distintas cargas eléctricas, de tal manera que el objeto a se transforma de manera independiente en cada una de ellas en nuevos objetos b y c (aparte de esos objetos, los contenidos de las membranas resultantes son idénticos). Este tipo de reglas no pueden ser ejecutadas en la `piel`.

Obsérvese que las reglas del sistema están asociadas a etiquetas (es decir, la regla $[a \rightarrow v]_l^\alpha$ está asociada a la etiqueta l) y no, propiamente, a las regiones del sistema. Nótese también que las reglas del tipo (*e*) permiten la existencia de varias membranas en el sistema con la misma etiqueta.

En cuanto a la manera de aplicar las reglas, se aplican de manera paralela maximal con la restricción de que sobre la misma membrana no se pueden aplicar simultáneamente más de una regla de los tipos $(b) - (e)$.

Los sistemas P con creación de membranas fueron introducidos por primera vez en [34, 35]. El modelo es esencialmente el mismo al descrito para la división, salvo que no se utilizan cargas eléctricas, y que en lugar de reglas de división de tipo (*e*) se consideran reglas de la forma $[a \rightarrow [v]_{l_2}]_{l_1}$. Este nuevo tipo de regla permite que un objeto a que se encuentre en una membrana l_1 produzca la aparición de una nueva membrana etiquetada por l_2 que estará contenida dentro de la anterior.

2.5. Sistemas P Estocásticos y Probabilísticos

La condición de paralelismo maximal no determinista en la aplicación de las reglas no se impone de manera sistemática en todos los modelos de computación celular. Por el contrario, existen diversas alternativas que pueden ser consideradas. Por ejemplo, A. Obtulowicz introdujo en [38] los sistemas P probabilísticos, en los que cada regla lleva asociada una probabilidad. Así, en lugar de dejar que el sistema haga elecciones no deterministas, los objetos disponibles se "reparten" entre las reglas aplicables proporcionalmente atendiendo a la probabilidad de cada una.

Otro caso en el que se utiliza una semántica distinta es el de los P sistemas estocásticos. Este modelo se introdujo con el principal objetivo de modelar en el marco de la computación celular con membranas un amplio abanico de fenómenos biológicos. Con este nuevo enfoque se

han obtenido muy buenos resultados, especialmente al simular procesos en los que el número de partículas (moléculas, bacterias, individuos de un ecosistema, etc) implicadas es muy bajo. En este caso la evolución del sistema está controlada por un algoritmo estocástico (por ejemplo, algoritmo de Gillespie), tratando de capturar la dinámica de los sistemas biológicos que se desea simular.

3

Unidad de Procesamiento Gráfico (GPU)

Antes de discutir el diseño de nuestro simulador para sistemas P con membranas activas, introduciremos la arquitectura de la GPU de NVIDIA utilizada en nuestras pruebas y el modelo de programación paralela CUDA.

3.1. GPU como Procesador de Propósito General

Las GPUs modernas ofrecen gran potencia y alta flexibilidad a bajo costo, lo cual las convierte en una alternativa ideal para resolver problemas que efectúan un elevado volumen de operaciones aritméticas. Una GPU es un procesador de múltiples núcleos que puede ser usado como coprocesador, ya que tiene su propio espacio de memoria y ejecuta paralelamente un gran número de hilos. Debido a esto surge el concepto de Unidad de Procesamiento Gráfico de Propósito General (GPGPU) [43], que hace referencia al aprovechamiento de la GPU para tareas no gráficas.

Típicamente, el desarrollo de aplicaciones para la GPGPU se ha realizado por medio de la codificación de *shaders* usando APIs gráficas, como son OpenGL [45] y DirectX [46], lo

cual puede llegar a hacer muy tedioso el desarrollo, por lo que han surgido nuevos lenguajes específicos para GPGPU que permiten crear programas sin apenas conocer el *pipeline* gráfico, como es el caso de Brook [40] y Sh [41], ambos desarrollados en universidades, y por otro lado también hay alternativas desarrolladas por grandes empresas, en este grupo podemos encontrar a CTM de ATI [47], Accelerator de Microsoft [39] y CUDA de NVIDIA [44], entre otras. En todos estos lenguajes de programación, a pesar de hacer sencilla la creación de un programa de propósito general, sigue siendo complejo sacar todo el potencial *hardware* que tienen estas tarjetas.

Por otro lado, cabe decir que no cualquier problema se adapta a la arquitectura de la GPU. Para que sea eficiente la ejecución en este tipo de arquitectura, el programa que se quiere paralelizar debe de tener mucha carga computacional e independencia de datos, pues así se puede aprovechar la gran cantidad de procesadores.

3.2. Arquitectura de la tarjeta Tesla

La tarjeta Tesla de NVIDIA tiene una arquitectura masivamente paralela, donde su procesador multihilo puede llegar a manejar hasta 30720 hilos paralelos de forma altamente eficiente tanto para aplicaciones gráficas, como para aplicaciones de propósito general.

La Figura 3.1 muestra la arquitectura de la GPU Tesla, en particular la Tesla C1060 [17] que tiene un `array` de 240 Procesadores de flujo (SPs) organizados en 30 Multiprocesadores de flujo (SMs) y 4GB de memoria GDDR3 (llamada *device memory* o *global memory*). Las aplicaciones comienzan en la parte del *host* (CPU) y se comunican con *device* (GPU) a través del bus PCI Express x16 (proporcionando un ancho de banda de 4 GB/s en cada dirección).

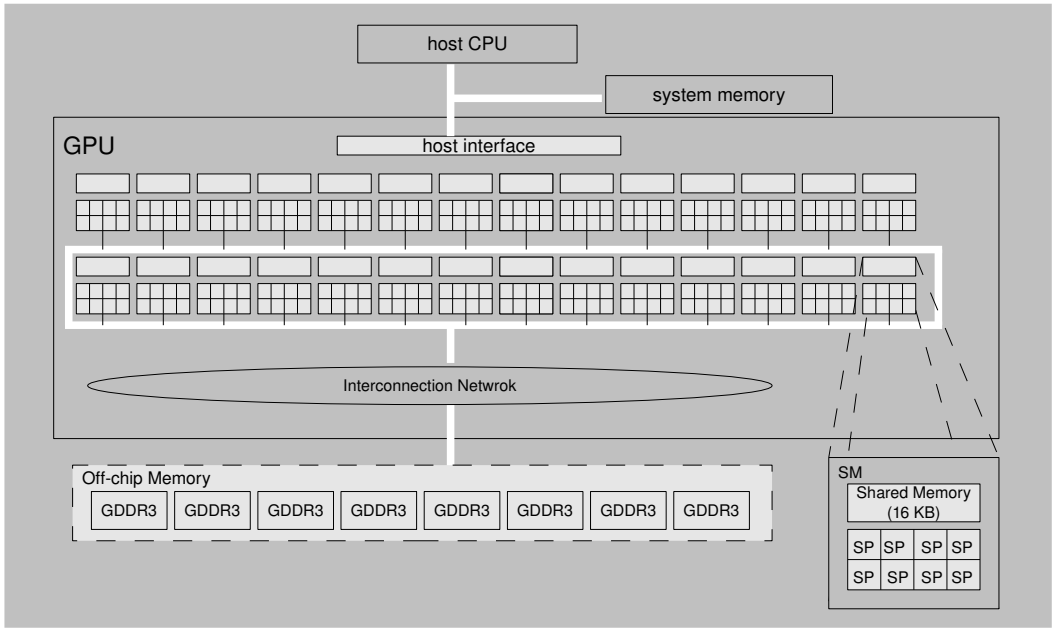

Figura 3.1: GPU Tesla C1060 con 240 SPs organizados en 30 SMs

3.2.1. Elementos de Procesamiento

La Figura 3.2 muestra la arquitectura de un SM, el cual tiene ocho SPs, un conjunto de 16384 registros de 32 bits y dos Unidades de Funciones Especiales (SFUs) que se encargan de realizar las operaciones en punto flotante, como son la raíz cuadrada, seno o coseno. Además, cuenta con un MT IU que permite controlar la ejecución y planificación de hasta 1024 hilos, y una memoria de 16 KB de lectura/escritura que tiene muy baja latencia de acceso (llamada *shared memory*).

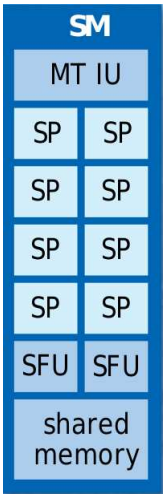

Figura 3.2: Arquitectura de un SM

Los SPs son capaces de ejecutar tres instrucciones de forma paralela en un ciclo de reloj y la frecuencia de la Tesla C1060 es de 1.296 GHz, permitiendo así alcanzar a esta una capacidad de proceso de 933 GFLOPS pico teóricos (240 SPs * 3 instrucciones * 1.296 GHz).

3.2.2. Organización del Sistema de Memoria

Además del uso de registros en la GPU podemos encontrar distintos tipos de memoria [44] (ver Figura 3.3), cada uno de ellos optimizado para un uso concreto. Es importante conocer bien estos tipos de memoria, ya que un mal uso de estos puede causar que el rendimiento de un programa decrezca en gran medida.

En la Tabla 3.1 se pueden observar los distintos tipos de memoria, junto a su tamaño y principales características. A continuación se detallan dichos tipos:

- ***Global memory* o *device memory*.**

  Como mencionamos anteriormente, en la tarjeta Tesla C1060 disponemos de 4GB de este tipo de memoria. Antes de comenzar la ejecución en la GPU todos los datos necesarios para la ejecución deben de ser almacenados en la *global memory*.

  El bus de datos de esta memoria tiene un ancho de 512 bits, divida en 8 particiones de 64 bits.

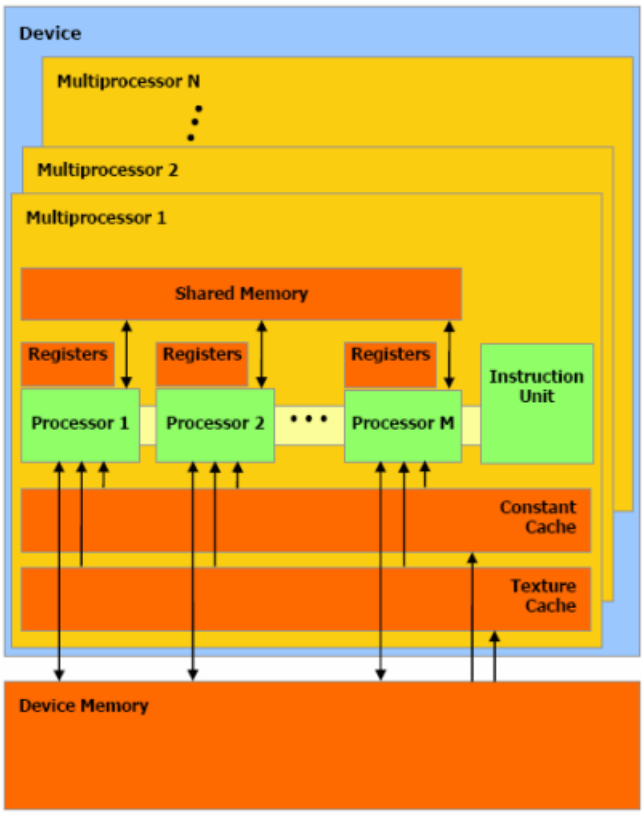

Figura 3.3: Jerarquía de memorias en la GPU

Para un uso eficiente de esta memoria debemos conseguir que se acceda de manera simultanea a posiciones contiguas de memoria permitiendo al hardware accesos fusionados a memoria (*coalesce memory access*).

- **Memoria local**.

  Esta memoria se utiliza cuando un hilo hace una reserva excesiva de registros, o para estructuras o `arrays` que consumen demasiados registros. Estos datos son almacenados en *device memory* sin ser cacheados.

- **Memoria de texturas**.

  La memoria de texturas está cacheada a memoria global. Por tanto, cuando el dato al que queremos acceder no está en la memoria de texturas tendremos que acceder a *device memory*.

  La memoria de texturas está optimizada para el almacenamiento 2D, aprovechando la localidad espacial. Así, al acceder a direcciones de memoria contiguas se pueden obtener beneficios frente a la memoria global o de constantes.

- **Memoria de constantes**.

  Hay 64 KB de memoria de constantes, la cual también está cacheada a *device memory*. Cuando se accede con frecuencia a una misma dirección de memoria se puede conseguir que el coste de los accesos a lectura sean tan rápidos como el coste del acceso a un registro. Este coste aumenta linealmente conforme los hilos acceden a diferentes posiciones de memoria.

- ***Shared Memory***.

  Tenemos 16 KB en cada SM que permiten un acceso rápido a los datos. Esta memoria está divida en 16 bancos de 1 KB y es tan rápida como los registros, siempre y cuando no haya conflictos de bancos. Cada banco puede ser accedido simultáneamente por hilos distintos, pero accesos al mismo banco por hilos distintos son serializados. Por tanto, para conseguir un rendimiento óptimo debemos intentar que hilos que se ejecutan a la vez en el mismo SM accedan a bancos distintos.

Memoria	**Situación**	**Tamaño**	**Latencia**	**Acceso**
Device Memory	Off-Chip	4 GB	400-600 ciclos	Lectura/Escritura
Memoria Local	Off-Chip	4 GB	400-600 ciclos	Lectura/Escritura
Memoria de Texturas	On-Chip	4 GB	>100 ciclos	Lectura
Memoria de Constantes	On-Chip	64 KB	$\simeq$ 0 ciclos	Lectura
Shared Memory	On-Chip	16 KB/SM	$\simeq$ 0 ciclos	Lectura/Escritura
Registros	On-Chip	16384/SM	$\simeq$ 0 ciclos	Lectura/Escritura

Tabla 3.1: Tipos de Memoria en la Tesla C1060

3.3. Modelo de Ejecución Multihilo

Subiendo de nivel, ahora vamos a ver como se ejecutan los hilos en los distintos SM, la organización lógica de hilos y el modo de planificación para la ejecución.

Un SM es un dispositivo específicamente diseñado para procesar gran cantidad de hilos, es capaz de ejecutar hasta 1024 hilos de forma concurrente. Cada hilo tiene su propio estado de ejecución por lo que cada uno puede ejecutar distintas partes del código con independencia del resto de hilos. Los SMs ejecutan hilos siguiendo el modelo Simple-Instrucción Múltiples-Hilos (SIMT) [17]. Los SMs crean, manejan y ejecutan los hilos en grupos de 32 hilos simultáneamente llamado *warp*. Cada SM puede llegar a manejar 32 *warps* (1024 hilos en total). Todos los hilos del *warp* deben de comenzar en la misma parte del programa, pero son libres de dividirse y ejecutarse con independencia

El flujo de ejecución comienza con un conjunto de *warps* listos para ser seleccionados. Uno de ellos (*warp* activo) es seleccionado por la unidad de instrucción que esté lista para ejecutar instrucciones.

El SM mapea todos los hilos del *warp* activo a los SPs, y cada uno de estos hilos se ejecuta con su propio estado de instrucciones y registros.

Algunos hilos en el *warp* pueden ser desactivados durante una predicción de salto, y este es el punto crítico en el proceso de optimización, donde el máximo rendimiento es obtenido cuando todos los hilos activos en el *warp* ejecutan las mismas instrucciones. Si por el contrario los hilos del *warp* divergen, este serializará su ejecución cada vez que se realice un salto en el recorrido, deshabilitando todos los hilos que no lo realicen, para que finalmente, cuando termine dicho recorrido, los hilos reconverjan para seguir con la ejecución normal [17].

El planificador de *warp* de cada SM trabaja a la mitad de la frecuencia que los procesadores, seleccionando en cada ciclo uno de los 32 *warps* posibles y ejecutándolo como dos subconjuntos de 16 hilos [33]. Además es importante destacar que el cambio de *warp* teóricamente tiene un coste de 0 ciclos [44].

3.4. Modelo de Programación en CUDA

El lenguaje de programación utilizado para la GPU es C y C++ con las extensiones que provee CUDA. En el modelo de programación CUDA [42] [20] encontramos dos partes bien diferenciadas, por un lado la parte *host* que se corresponde con la CPU, y la parte *device* que hace referencia a la GPU. Un programa escrito en el lenguaje de programación CUDA tiene una parte secuencial (*host code*) que ejecuta programas paralelos (conocidos como *kernels*) en *device*. Los *host programs* se ejecutan en la Unidad Central de Procesamiento (CPU) y los *kernels* son ejecutados en la GPU.

Un *kernel* sigue el modelo Simple-Programa Múltiples-Datos (SPMD), donde un gran número de hilos se ejecutan en paralelo. El programador organiza estos hilos en un *grid* de *bloques de hilos* (ver Figura 3.4). Un bloque de hilos en CUDA es un conjunto de hilos que pueden ejecutar el mismo programa (*kernel*) y cooperar para obtener un resultado gracias al uso de barreras de sincronización y una *shared memory* que comparten los hilos del bloque.

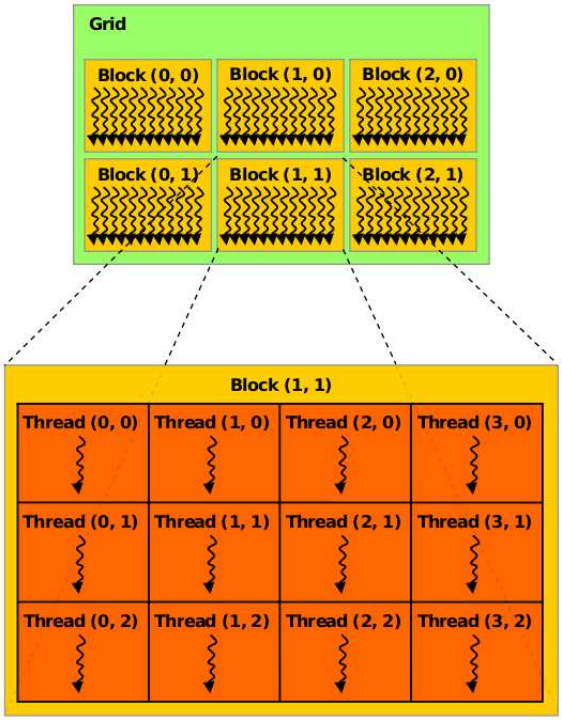

Figura 3.4: *Grid* de bloques de hilos

El programador decide el número de bloques de hilos que hay en un *grid* y el número de hilos que hay en un bloque. Los bloques de hilos en un *grid* pueden usar una o dos dimensiones y todos ellos tienen su único identificador. De forma similar, los hilos que están dentro de un bloque de hilos pueden ser declarados utilizando una, dos o tres dimensiones, todos con su

identificador propio. Por otro lado, el número máximo de hilos en un bloque es de 512. Usando una combinación del identificador dentro del *grid* y del identificador dentro del bloque, los hilos pueden seleccionar los distintos datos a los que deben acceder, o seleccionar que partes del programa deben ejecutar.

Los bloques de hilos en el modelo de programación CUDA son vistos como *multiprocesadores virtuales*, ya que ellos tienen fijada un asignación en la *shared memory* y cada hilo en el bloque tiene también reservado unos registros [32]. La comunicación entre bloques de hilos es llevada a cabo a través de la *global memory*, y la sincronización entre ellos sólo es conseguida cuando el *kernel* finaliza.

3.4.1. Capacidad de Cómputo

Algo a tener en cuenta a la hora de programar en CUDA es la Capacidad de cómputo (CC) de la tarjeta que estamos utilizando. La capacidad de cómputo está definida mediante dos números, el primero de ellos hace referencia a la arquitectura de los procesadores, y el segundo número sirve para indicar pequeños añadidos en los procesadores. Así pues, en nuestro caso la tarjeta Tesla C1060 dispone de la CC 1.3, gracias a la cual, entre otras cosas, vamos a poder hacer uso de las operaciones atómicas tanto a nivel de *global memory* (a partir de la CC 1.1), e incluso a nivel de *shared memory* (a partir de la CC 1.2).

Gracias a las operaciones atómicas se puede conseguir una lectura, modificación y/o escritura de una dirección de memoria de 32 o 64 bits, garantizando que se realice sin que otros hilos puedan interferir hasta que las operaciones sean completadas.

El uso de las operaciones atómicas será un requisito indispensable para poder llevar a cabo algunos aspectos de este trabajo.

Para finalizar este capítulo, en la Tabla 3.2 se muestra un resumen de algunas de las características más importantes en relación al `hardware` y `software` de la tarjeta Tesla C1060.

Componente Hardware	**Limitación**
Capacidad de Cómputo	1.3
Multiprocesador de flujo (SM)	30
Procesador de flujo (SP)/SM	8
Número de *cores*	240
Registros de 32 bits/SM	16384
Device Memory	4 GB
Memoria de Constantes	54 kB
Shared Memory/SM	16 KB
Hilos/SM	1024
Hilos por bloque	512
Hilos por *Warp*	32
Frecuencia de reloj	1.30 GHz

Tabla 3.2: Resumen de características hardware y software en la Tesla C1060

4

Simulación de Sistemas P con Membranas Activas

Para la realización del simulador de sistemas P con membranas activas hemos tomado como punto de partida el simulador secuencial JAVA definido en P-Lingua 1.0 [6][1].

El simulador secuencial toma como entrada un fichero binario generado por el compilador de P-Lingua, y obtiene como salida la respuesta contenida en el entorno, teniendo la posibilidad de mostrar la información de los pasos ejecutados, incluyendo la configuración de todas las membranas del sistema P en los distintos pasos.

El diseño del simulador presentado en [6] está basado en la división de la simulación en dos fases: fase de selección y fase de ejecución.

Tanto la forma de manejar la entrada de datos, como la diferenciación de fases especificadas anteriormente serán mantenidas en los diseños de los simuladores paralelizados en la GPU.

4.1. Simulador Secuencial

Para realizar una comparación más justa con los simuladores parelelizados en la GPU se recodificará en C++ el simulador secuencial programado en JAVA. Además, este será

[1] P-Lingua es un framework desarrollado por investigadores del RGNC, el cual provee un lenguaje de especificación de sistemas P, parsers, compiladores y simuladores para distintos modelos [7].

adaptado para tener las mismas estructuras que los simuladores CUDA explicados en los siguientes capítulos.

La versión secuencial en C++ obtiene un *speed up* de hasta 120x con respecto a la versión JAVA.

4.2. Entrada del Simulador

La Figura 4.1 muestra el proceso de generación de un fichero binario mediante el compilador de P-Lingua. Para poder generar este fichero binario, es necesario definir un sistema P siguiendo el lenguaje de programación P-Lingua.

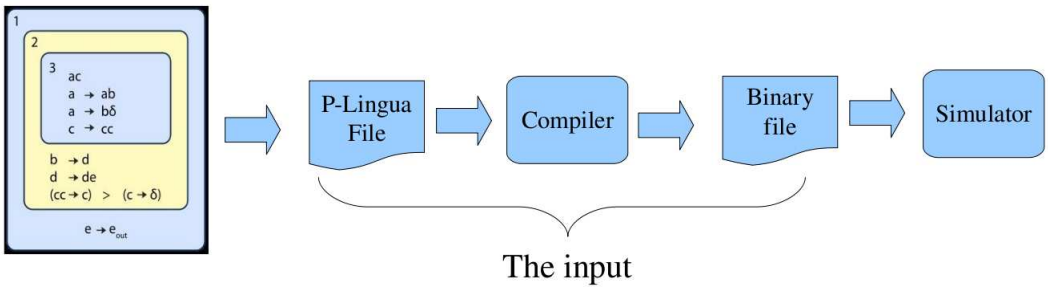

Figura 4.1: Generación de la entrada de los simuladores

El fichero binario que genera el compilador de P-Lingua contiene toda la información del sistema P (alfabeto, etiquetas, reglas, ...), totalmente parseada y libre de errores sintácticos y semánticos desde el punto de vista de especificación de sistemas P. Nuestro simulador se encarga de leer este fichero e introducir toda la información que contiene en las estructuras adecuadas para poder trabajar con ellas. Dado que se usa un fichero binario, este proceso es eficientemente implementado en C++.

4.3. Estructura del Simulador

El algoritmo de simulación de sistemas P con membranas activas aquí estudiado está diseñado teniendo en cuenta que hay dos fases conceptualmente bien diferenciadas:

- **Fase de Selección**. En esta fase se buscan todas las reglas que se pueden ejecutar en cada membrana. Los datos de entrada que recibe la fase de selección consisten en la descripción de las membranas con sus multiconjuntos (cadenas sobre el alfabeto O, etiquetas en H asociadas con las membranas, etc.) y las reglas R que pueden ser seleccionadas. La salida de esta fase será el conjunto de reglas seleccionadas por todas las membranas.
- **Fase de Ejecución**. Se ejecutan las reglas que hayan sido seleccionadas en la fase anterior. La fase de ejecución tomará como entrada la información de todas las membranas del sistema P y las reglas seleccionadas en la fase previa, las cuales ejecutará una a una

reflejando las modificaciones pertinentes en el sistema P, y obteniendo así una nueva configuración del sistema P. Esta nueva configuración será la entrada para la fase de selección de la siguiente iteración (siguiente configuración del sistema P).

Se da un proceso de interacción entre estas dos fases hasta que la configuración final sea obtenida o se llegue a un límite de pasos previamente especificado.

Teniendo en cuenta esta distinción de fases surgirán dos simuladores paralelos. El primero de ellos tan sólo paralelizará la fase de selección (ver capítulo 5), en cambio, el segundo simulador paralelo será completamente ejecutado en la GPU (ver capítulo 6).

Los simuladores presentados en este trabajo suponen que los sistemas P que van a tratar son confluentes (definido en la sección 2.2), y por lo tanto resuelven el problema del no-determinismo seleccionando una rama cualquiera de computación, ya que todas llevan a la misma respuesta del sistema.

5

Simulador de Sistemas P con Selección en GPU

En este capítulo mostramos un primer simulador paralelo de sistemas P con membranas activas.

Este simulador está basado en la separación conceptual de las fases de selección y de ejecución. En esta primera versión del simulador paralelizamos la fase de selección en la GPU, mateniendo la fase de ejecución en la CPU.

Durante el capítulo se presentan tanto aspectos del diseño como de la implementación del simulador.

5.1. Diseño

En el diseño de aplicaciones en el modelo de programación CUDA necesitamos identificar el trabajo a realizar por cada hilo independiente y por cada bloque de hilos.

En esta cuestión podemos encontrar principalmente dos alternativas:

1. Identificar cada uno de los hilos con una de las posibles reglas que pueden ser seleccionadas. y por otro lado identificar los bloques de hilos con las membranas. De este modo, cada bloque de hilos comprobará todas las reglas que podrían ser ejecutadas en la membrana, obteniendo como salida las reglas que se deben de ejecutar en cada una de las membranas.

2. La otra opción se basa en la similitud del doble paralelismo que podemos encontrar en los sistemas P (a nivel de objetos y a nivel de membranas) y en el doble paralelismo en el modelo de programación CUDA. A partir de esta idea relacionamos los conceptos de hilo y objeto, y los de bloque de hilos y membrana.

En este diseño nos centramos en las segunda alternativa porque, a priori, el número de objetos es menor que el número de reglas y así evitamos tener un mayor número de hilos por bloque, que haría más complejo el diseño y por tanto más ineficiente. Además, el número de objetos distintos que puede haber en una membrana es común a todas las membranas (tamaño del alfabeto).

Es importante destacar que es posible que haya más de 512 objetos en una membrana de un sistema P, por lo que para poder ejecutar dicha membrana en la GPU necesitamos disponer de más de 512 hilos, lo cual es físicamente imposible en la tarjeta Tesla C1060 en la que trabajamos [17]; así pues, la solución adoptada es que un hilo pueda asociarse a varios objetos. En cuanto a lo que respecta al número de membranas no hay ningún problema, ya que el número de bloques de hilos que se pueden ejecutar en la GPU es muy elevado, en total 2^{32} bloques (2^{16} bloques en la dimensión x del grid multiplicados por 2^{16} bloques en la dimensión y del grid).

La Figura 5.1 muestra como cada hilo se corresponde con un objeto (objetos) del multiconjunto perteneciente a una membrana dada. Este se encarga de comprobar si para ese objeto hay alguna posible regla (reglas) a ejecutar y, de ser así, añadirla al conjunto de reglas que deben de ser ejecutadas.

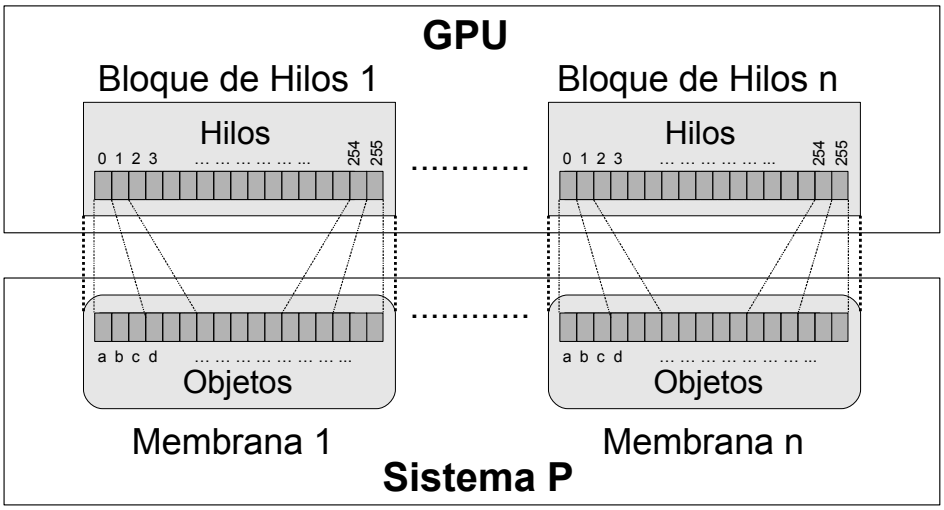

Figura 5.1: Correspondencia entre objetos/hilos y membranas/bloques de hilos

A continuación vemos la información que necesita el *kernel* para poder realizar los cálculos

oportunos:

- Los datos sobre los objetos que hay en cada uno de los multiconjuntos de las membranas (exceptuando la `piel`).
- La carga y etiqueta de cada una de las membranas (exceptuando la `piel`).
- La información necesaria para saber si hay reglas de un determinado tipo asociadas a un objeto.
- El multiconjunto de la `piel`, que es tratado como una membrana especial.
- Además hacen falta otros datos auxiliares: el número de membranas, el número de etiquetas y el número de objetos.

Por otro lado el *kernel* devuelve la siguiente información:

- Para cada uno de los objetos se indica las veces que es seleccionada la regla de evolución que tiene asociada.
- Devolvemos si se ha seleccionado alguna regla de los tipos ***send in***, ***send out***, `disolución` y `división` para una determinada membrana, ya que sólo se puede seleccionar una regla de estos tipos por membrana.

En el siguiente apartado se puede ver mejor el funcionamiento del simulador mediante un pseudocódigo y su explicación.

5.2. Implementación del Simulador

En esta sección nos centramos en cuestiones relacionadas con la codificación del simulador, así pues, se puede observar el funcionamiento del bucle principal donde se realizan las llamadas al *kernel* y las transferencias de datos entre la CPU y la GPU.

Además, se muestra una explicación de las estructuras que son utilizadas dentro del *kernel*, mediante el uso de la nomenclatura también descrita en esta sección.

5.2.1. Pseudocódigo

El Algoritmo 1 resume el funcionamiento del simulador, abstrayendo los detalles de implementación.

Al comenzar se transfiere la información correspondiente a las reglas desde la CPU a la GPU, la cual permanece invariante durante toda la simulación.

A continuación se entra en el bucle principal del algoritmo, donde se realiza la transferencia de los datos de las membranas a la GPU.

Con la información transferida se llama al *kernel* donde se realiza la selección de las reglas que pueden y deben ejecutarse.

Una vez obtenidas las reglas seleccionadas hay que pasar dicha información desde la GPU a la CPU, para así poder ejecutar las reglas de forma secuencial en la CPU. Esto hace que se modifiquen las membranas y, por lo tanto, que en el siguiente paso haya que transferir de nuevo esta información a la GPU.

En el primer paso partimos de configuración inicial del sistema P, paso tras paso se van produciendo modificaciones en las membranas hasta llegar a la configuración final o hasta que sea alcanzado un número máximo de pasos que se ha definido antes del comienzo de la simulación.

Algoritmo 1 Simulador Paralelo con Selección en la GPU

Entrada: *membranas*: Configuración inicial del sistema P.
reglas: Reglas disponibles.
limitePasos: límite máximo de pasos a ejecutar.
Salida: *membranas*: Configuración final del sistema P.

```
1: CopiarDatosDesdeCPUaGPU(reglas)
2: mientras NO limitePasos Y NO configuracionFinal hacer
3:    CopiarDatosDesdeCPUaGPU(membranas)
4:    kernelSeleccion(reglas, membranas, reglasSeleccionadas)
5:    CopiarDatosDesdeGPUaCPU(reglasSeleccionadas)
6:    ejecutarReglas(reglas, membranas, reglasSeleccionadas)
7: fin mientras
```

5.2.2. Nomenclatura

A continuación describimos la nomenclatura usada para explicar con mayor facilidad ciertos conceptos en la codificación de las estructuras:

- m: Número de membranas máximo que hay en el sistema P de entrada.
- o: Número de objetos que hay en el alfabeto O del sistema P de entrada.
- r: Número de reglas que hay en el sistema P de entrada.
- c: Número de cargas posibles que hay en el sistema P de entrada.
- e: Número de etiquetas que hay en el sistema P de entrada.
- *mid*: Identificador de una membrana en el sistema P de entrada.
- *oid*: Identificador de un objeto del alfabeto O dentro de un membrana del sistema P de entrada.
- b: Número de bloques con los que se lanza un *kernel* en CUDA.
- h: Número de hilos en cada bloque con los que se lanza un *kernel* en CUDA.

- *bid*: Identificador de un bloque en CUDA.
- *hid*: Identificador de un hilo dentro de un bloque en CUDA.

5.2.3. Datos de Entrada

A continuación se describen las diferentes estructuras que son utilizadas para pasar información al *kernel* de selección.

`Array` con información de los multiconjuntos

`Array` unidimensional que contiene toda la información relativa a los multiconjuntos de todas las *membranas*. Tiene un tamaño de $O(m * o)$, y en cada posición se almacena un `ushort` que contiene la multiplicidad del objeto al cual pertenece esa posición.

La Figura 5.2 muestra dicha estructura con m membranas, cada una de ellas con un alfabeto de tamaño o. Por simplicidad se ha dibujado un `array` bidimensional, aunque en realidad el objeto *oid* de la membrana *mid* es direccionado de la forma $mid * o + oid$.

	O_1	O_2	O_3	O_4	O_5	O_6	O_7		O_{o-1}	O_o
M_1	1	0	0	6	0	0	0	...	0	3
...	...	...	...	...	...	...	...	...	...	...
M_n	3	0	0	0	0	0	0	...	2	4

Figura 5.2: `Array` con la información de los multiconjuntos

`Array` con información de cargas y etiquetas

Este `array` sirve para almacenar las cargas y etiquetas asociadas a cada una de las *membranas*, y tiene un tamaño de $O(m)$. La Figura 5.3 muestra este `array` donde cada posición contiene una estructura de dos `shorts`, uno almacena la carga de la membrana y el otro la etiqueta de la misma.

	M_1	M_2	M_3	M_4	M_5	M_6	M_7		M_{m-1}	M_n
Etiq	2	1	0	0	1	0	2	...	1	2
Carg	0	0	1	2	2	2	1	...	2	0

Figura 5.3: `Array` con información de cargas y etiquetas

Estructura con las reglas

Ahora pasaremos a explicar la estructura utilizada para pasar la información de las reglas disponibles en el sistema P (ver Figura 5.4).

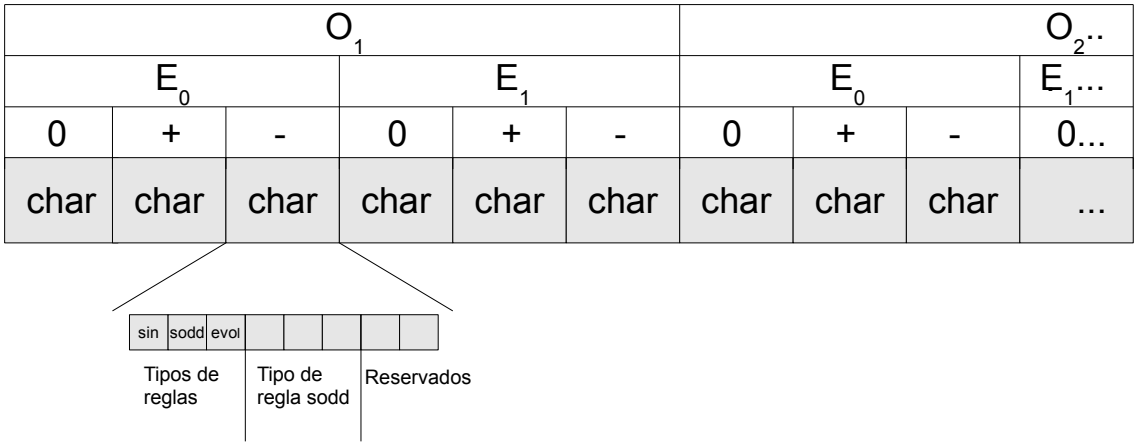

Figura 5.4: Estructura con las reglas

La estructura presenta una posición para cada uno de los objeto del alfabeto O, teniendo en cuenta que ese objeto puede estar en una membrana con cualquier etiqueta del conjunto H y con tres posibles cargas (+, -, 0). Por tanto la estructura tiene un tamaño de $O(e*c*o)$.

En cada posición de la estructura almacenamos la información necesaria para saber que reglas hay asociadas para un objeto dado de una determinada membrana. Para ello tendremos en consideración que:

- En una membrana se puede ejecutar solo una regla de los tipos ***send in***, ***send out***, `disolución` y `división`[1]. Teniendo esto en cuenta surge la duda de cual añadir a la estructura en el caso de que haya varias reglas aplicables a un objeto. La solución a este problema ha sido crear una jerarquía de prioridades entre los distintos tipos de reglas, veámosla teniendo en cuenta que están ordenadas de mayor a menor prioridad:

  1. `Disolución`. Es beneficioso trabajar con el menor número de membranas, y si tenemos la opción de poder eliminar una membrana debemos de considerarlo.
  2. `Evolución`. Son reglas fáciles de aplicar ya que únicamente provocan cambios a nivel local de la membrana.
  3. ***Send out***. Las reglas de este tipo tienen una dependencia con el padre.
  4. ***Send in***. Este tipo de regla es el más problemático de aplicar, ya que al igual que en el caso anterior tiene una dependencia con el padre, pero con el agravante de que si varios hilos, ya sean de una misma membrana o de distintas membranas, intentan traer un objeto del padre simultáneamente se podrían provocar incoherencias.
  5. `División`. Son un tipo de reglas que, a pesar de no tener dependencias, provocan el incremento del número de membranas, lo cual debe de evitarse.

[1] A los tres últimos tipos nos referiremos como tipos `sodd`, si en cambio también incluimos las reglas de tipo ***send in*** hablaremos de tipos `siodd`.

- Un objeto puede tener varias reglas de un mismo tipo asociadas, es válido elegir aleatoriamente una de esas reglas, dado que suponemos que el sistema P es confluente.

A partir de esto se podría llegar a la conclusión de que en cada posición de la estructura sólo necesitamos un `bit` que indique si hay alguna regla para ser seleccionada de tipo `evolución` y otro `bit` para indicar si hay alguna regla del resto de tipos. Pero con estos dos `bits` no es suficiente debido a que las reglas de tipo *`send in`* no se va a saber si se pueden ejecutar hasta el momento de ejecución, ya que las reglas de los tipos `sodd` dependen del mismo objeto local a la membrana, pero las de tipo `send-in` no.

Por ejemplo, podría darse el caso de que un objeto tenga una regla de tipo `división` y otra de tipo *`send in`* asociadas, y al tener ésta última mayor prioridad se introduzca en nuestra estructura, pero en tiempo de ejecución no pueda seleccionarse y la membrana pueda quedar sin ninguna regla seleccionada, cuando podría haberse seleccionado la regla de tipo `división`.

Por este motivo cada campo de la estructura debe tener tres `bits` para indicar los posibles tipos que se pueden seleccionar, y otros tantos para poder distinguir el tipo exacto de regla de entre los tipos `sodd` que hay, y así saber la prioridad que tiene frente a las demás reglas asociadas a otros objetos de la misma membrana.

5.2.4. Datos de Salida

Al final, el kernel de selección devuelve cierta información codificada en las estructuras detalladas a continuación.

`Array` con información de las reglas de evolución seleccionadas

Para cada regla de tipo evolución tenemos que indicar cuantas veces ha sido seleccionada, para ello necesitamos una estructura de tamaño de $O(m * o)$, donde en cada posición se almacena un `ushort`. La estructura mostrada por la Figura 5.2 se adecua a estas necesidades, y ya que sólo es necesitada para introducir datos al *kernel* podría ser reutilizada para la salida con este propósito.

`Array` con información del resto de reglas seleccionadas

Para cada una de las membranas indicamos si se ha seleccionado alguna regla de los tipos *`send in`*, *`send out`*, `disolución` y `división`.

Para esto es necesario un `array` de longitud $O(m)$, donde cada posición contiene un `uint` para poder almacenar el número que identifica a la regla seleccionada (ver Figura 5.5).

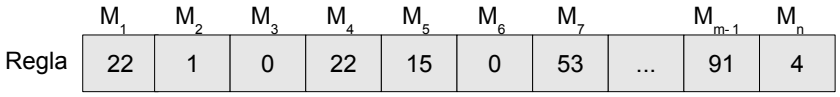

Figura 5.5: `Array` con las reglas seleccionadas de los tipos `siodd`.

5.2.5. Kernel de `Selección`

En esta versión, el simulador tiene un único *kernel*, el cual se encarga de realizar la fase de selección. A continuación describiremos como se lleva a cabo esta fase separándola conceptualmente en obtención de datos, obtención de las reglas seleccionadas y almacenamiento de estas.

Obtención de Datos

Partiendo de las estructuras de entrada anteriormente descritas, cada uno de los bloques de hilos accede a la posición *bid* de la estructura que contiene la información de las cargas y etiquetas, obteniendo la información de la membrana que representan.

Posteriormente se accede desde la posición $o/h * hid * bid$ hasta la posición $o/h * hid * bid + o/h - 1$ del `array` que contiene la información de los multiconjuntos, para así obtener la multiplicidad del objeto (objetos) que le corresponde (corresponden) al hilo en cuestión. Por ejemplo, si se da el caso de un alfabeto que es cuatro veces el número de hilos por bloque, el quinto hilo del bloque que está situado en sexto lugar accederá desde la posición 120 (4*5*6) a la 123 (4*5*6+4-1).

Una vez que cada hilo tenga la información sobre la multiplicidad del objeto (objetos) que le corresponde (corresponden), y de la carga y etiqueta de la membrana que representa, debe de acceder a la estructura que contiene las reglas. En este caso cada hilo accede desde la posición $idh * e * c + eid * c + cid$ a la posición $idh * e * c + eid * c + cid + o/h - 1$, donde *eid* y *cid* son respectivamente la etiqueta y carga asociadas a la membrana a la que pertenece el objeto (objetos).

Obtención de Reglas Seleccionadas

Aquí se describe la parte principal del *kernel* de selección, donde se comprueba si hay alguna regla disponible para ser seleccionada. Esto se hace siguiendo el orden de tipos de reglas especificado en el subapartado 5.2.3 de este capítulo, lo cual se consigue mediante sincronizaciones de los hilos de un bloque tras realizar la comprobación de la existencia de una regla de un determinado tipo. Realizando este proceso en el orden adecuado podemos asegurar el cumplimiento de dicha jerarquía de prioridades.

También debemos de tener en cuenta que todos los hilos de un bloque comprueban a la vez un determinado tipo de reglas, y se podría dar el caso de que dos o más hilos seleccionaran simultáneamente varias reglas de los tipos `siodd`. En este caso recurri-

mos a las operaciones atómicas mencionadas en la sección 3.4.1, concretamente a la instrucción atómica `atomicCAS(int* direc, int compara, int val)`, gracias a la cual conseguimos realizar atómicamente una lectura de una zona de memoria (`direc`), computar ($antiguo == compara?val : antiguo$) y almacenar el resultado en la misma zona de memoria. De este modo, cuando una regla de tipo `siodd` es seleccionada se cambia el valor de la variable `antiguo` almacenada en `direc` y ya no se podrán almacenar más reglas.

En cuanto a las reglas de tipo `evolución`, al poder seleccionarse varias por hilo y membrana, se deben tratar de diferente manera, de tal modo que tras comprobar las reglas de mayor prioridad (únicamente las de tipo `disolución`), se siguen comprobando si existen reglas del tipo `evolución`, y de ser así se seleccionan de forma maximal, es decir, cada hilo seleccionara todas las reglas de evolución que pueda para el objeto (objetos) que tenga asociados, sin necesidad de realizar comprobaciones atómicas.

Por otro lado, con las reglas del tipo ***send in*** se podría dar el caso de que varios bloques de hilos intentaran seleccionar simultáneamente reglas de este tipo para introducir en su membrana el mismo objeto de la membrana `piel`, y que este tuviera una multiplicidad menor al número de bloques de hilos que están accediendo a él. En este caso, todos los bloques seleccionarían dicha regla, lo que provocaría una situación no deseada, que debe de controlarse mediante el uso de la operación atómica `atomicDec`, que asegura que el objeto de la membrana `piel` sea decrementado atómicamente por los distintos bloques de hilos.

Almacenado de Reglas Seleccionadas

Para el almacenado de las reglas que han sido seleccionadas nos valemos de las estructuras mencionadas en el subapartado 5.2.4. La primera estructura que se menciona en ese subapartado es utilizada durante la fase de selección de las reglas de tipo `evolución`, cada hilo introduce desde la posición $o/h * hid * bid$ hasta la posición $o/h * hid * bid + o/h - 1$ del `array` las reglas de dicho tipo que ha seleccionado para cada objeto (objetos).

En cuanto a las reglas de los tipos `siodd`, cuando una es seleccionada, se introduce el identificador de la regla en el segundo array presentado en el subaparatado 5.2.4, concretamente en la posición *bid*.

6

Simulador de Sistemas P con Selección y Ejecución en GPU

A partir del simulador que paraleliza la fase de selección, ahora se va a mostrar las modificaciones necesarias que se han realizado para conseguir que las dos fases del simulador sean ejecutadas en la GPU.

Al igual que en el capítulo anterior, a lo largo de este capítulo iremos detallando información sobre el diseño y la implementación del simulador.

6.1. Diseño

Uno de los objetivos de esta versión del simulador es disminuir la cantidad de datos transferidos entre la GPU y la CPU. Por ello deberemos adaptar la versión anterior para que sólo se transfiera la configuración inicial al *kernel* y, a partir de ese momento se realicen

los cálculos necesarios en la GPU, únicamente dando el control a la CPU para tareas de sincronización y cálculos cuya paralelización no sea factible.

Ahora, al poder ejecutarse reglas de tipo `división` en el *kernel*, hay que reservar a priori tanto espacio en la GPU como se vaya a necesitar tras la ejecución de la división. No obstante. antes de comenzar a realizar la ejecución, se puede saber cual será el número máximo de membranas que se van a generar para un sistema P dado, y así evitar realizar una reserva de memoria cada vez que se prevea que se puede realizar una división, ya que tan sólo con una única reserva de memoria se podría tener la certeza de que la ejecución se va a poder realizar sin problemas de este tipo.

Hay varias opciones a la hora de implementar la ejecución: en un solo *kernel* o en varios. Veamos las ventajas de realizarlo en varios *kernels*:

1. Es más modular y por tanto facilita la organización del código, el reparto del trabajo y el mantenimiento de la aplicación. Además, permite localizar errores más fácilmente.
2. Se consigue que el *kernel* tenga un menor número de bifurcaciones en el código.

Por otro lado, si se implementa en un único *kernel*, y durante una ejecución se tienen que ejecutar varias reglas de distintos tipos, se ahorraría la sobrecarga que conlleva la invocación de varios *kernels*. Pero aun así, es más eficiente la otra opción, ya que la sobrecarga de llamar a un *kernel* no es tan elevada. Asimismo, los sistemas P suelen estar diseñados para que en un determinado paso solo ejecuten reglas de un determinado tipo (exceptuando las reglas de tipo `evolución`).

Ahora, en el *kernel* de selección, la información que le pasamos de los multiconjuntos no debe ser sobreescrita para introducir los datos de las reglas de `evolución` seleccionadas, ya que esa información es modificada en cada paso a partir de las reglas ejecutadas, hasta llegar a la configuración final. Por este motivo, si queremos almacenar las reglas de tipo `evolución` seleccionadas habría que crear otra estructura de igual dimensión a la que contiene los multiconjuntos, lo cual es mejor evitar, debido a que esa estructura puede llegar a ocupar un gran espacio en memoria, y tenerla por duplicado podría impedir realizar ejecuciones de instancias de sistemas P de cierta envergadura. Pero hemos ideado una alternativa que evita esto: consiste en ejecutar las reglas de evolución en el `kernel` de selección. Esto es posible debido a que los cambios que realizan este tipo de reglas afectan localmente a la membrana, pero jamás realizan un cambio en la *piel*, y por lo tanto, no requieren sincronización global. Además, las reglas de tipo `evolución` suelen ejecutarse en cada paso, evitando así llamadas extra a otro *kernel* y, en consecuencia, mejorando la eficiencia.

Los datos que tenemos que pasarle al *kernel* de selección no varían prácticamente nada con respecto a la versión anterior, eso sí, la estructura que contiene la información de las reglas deberá de contener una cantidad mayor de información, pues ya no es suficiente con que se indique las reglas que hay disponibles, sino que debe contener toda la información relativa a las reglas, para que estas se puedan ejecutar en la GPU.

A la hora de llamar a los *kernels* que ejecutan los distintos tipos de reglas sería poco eficiente llamar a todos, cuando a lo mejor sólo se van a ejecutar reglas de un determinado

tipo, y por lo tanto con hacer la llamada a un solo *kernel* sería suficiente. Esto se consigue gracias a una variable que pasamos al *kernel* de selección, donde marcando unos determinados `bits` se indican los tipos de las reglas seleccionadas. Para evitar incoherencias esto se realiza mediante el uso de una operación atómica, en concreto la instrucción `atomicOr` que opera atómicamente a nivel de `bits`.

Se utilizan los siguientes datos de entrada en los *kernels* de ejecución:

- Los datos sobre los objetos que hay en cada uno de los multiconjuntos de las membranas (exceptuando la `piel`).
- La carga y etiqueta de cada una de las membranas (exceptuando la `piel`).
- La información necesaria para poder ejecutar cualquiera de las reglas disponibles.
- Los identificadores de las reglas que han sido seleccionadas para ser ejecutadas en el *kernel* de selección.
- En algunos de estos *kernels* se producen modificaciones en la membrana `piel`, por lo que también deberemos pasarle los datos referentes a su multiconjunto.
- Otros datos auxiliares son necesarios también, como es el caso del número de membranas, el número de etiquetas o el número de objetos.

Cuando finalice cualquier *kernel* de ejecución se habrán producido modificaciones que afectarán a:

- Los multiconjuntos de las membranas (exceptuando la `piel`).
- La carga y etiqueta de las membranas donde se han ejecutado las reglas (exceptuando la `piel`).
- La membrana `piel` es modificada en algunos casos, según el tipo de regla que sea ejecutado.
- Al producirse una o más divisiones, aumenta el número de membranas que hay en el sistema P.

La única membrana que es ejecutada en la CPU es la `piel`, por su necesidad de una sincronización global no puede seleccionarse/ejecutarse en paralelo junto con el resto de membranas, y lanzar un *kernel* para ejecutar tan sólo una membrana, con el escaso nivel de paralelismo que se podría conseguir, no merece la pena. Por lo tanto, se ha optado por realizar la ejecución de la membrana `piel` en la CPU.

El proceso de ejecución del resto de membranas puede verse en la siguiente sección, junto a un pseudocódigo explicativo.

6.2. Implementación del Simulador

En este apartado queda reflejada información sobre todos los *kernels* y la parte principal del algoritmo desde donde son invocados.

También se muestra una explicación de los cambios realizados en las estructuras con respecto a la anterior versión.

6.2.1. Pseudocódigo

El Algoritmo 2 describe el comportamiento de este simulador, el cual describimos a continuación.

La configuración inicial de las membranas y las reglas son transferidas de la CPU a la GPU.

Después, accedemos al bucle donde tienen lugar las llamadas a los *kernels*. El primero en ser llamado es el de selección. Una vez seleccionadas la reglas y teniendo conocimiento de su tipo, se invocará el *kernel* (los *kernels*) de ejecución que sea necesario. Este proceso se repetirá hasta salir del bucle, por haber obtenido la configuración final o haber llegado al número máximo de pasos.

Una vez terminada la ejecución, se pasa la configuración de las membranas obtenida en la GPU a la CPU.

Algoritmo 2 Simulador Paralelo con Selección+Ejecución en la GPU

Entrada: $membranas$: Estado inicial del sistema P.
$reglas$: Reglas disponibles.
$limitePasos$: límite máximo de pasos a ejecutar.
Salida: $membranas$: Estado final del sistema P.

```
1: CopiarDatosDesdeCPUaGPU(membranas)
2: CopiarDatosDesdeCPUaGPU(reglas)
3: mientras NO limitePasos Y NO configuracionFinal hacer
4:    kernelSeleccion(reglas, membranas, reglasSeleccionadas)
5:    si DISOLUCION ∈ reglasSeleccionadas entonces
6:       kernelDisolucion(reglas, membranas, reglasSeleccionadas)
7:    si no, si DIVISION ∈ reglasSeleccionadas entonces
8:       kernelDivision(reglas, membranas, reglasSeleccionadas)
9:    si no, si SEND_OUT ∈ reglasSeleccionadas entonces
10:      kernelSendOut(reglas, membranas, reglasSeleccionadas)
11:   si no, si SEND_IN ∈ reglasSeleccionadas entonces
12:      kernelSendIn(reglas, membranas, reglasSeleccionadas)
13:   fin si
14: fin mientras
15: CopiarDatosDesdeGPUaCPU(membranas)
```

6.2.2. Estructuras de Entrada

Las estructuras a utilizar son básicamente las mismas que las explicadas en el subapartado 5.2.3 de el capítulo anterior, con la diferencia de que ahora la estructura de reglas tiene mayor información, y que ya no es necesario reutilizar la estructura que se usaba antes para devolver las reglas de `evolución`. En cuanto al resto de estructuras no hay variaciones, por lo que sólo procedemos a explicar la estructura que contiene las reglas.

Estructura con las reglas

La Figura 6.1 muestra la nueva estructura de las reglas. Esta conserva toda la información que había en la versión anterior, es decir, lo necesario para poder indicar si en una posición hay una regla de tipo ***send in***, de tipo `evolución` o del resto de tipos. Asimismo, contiene información sobre las cargas que se le tienen que asignar a las membranas después de aplicar las reglas, ya sea una regla de tipo ***send in*** (campo `carga`), división (campos `carga1` y `carga2`), o del resto de tipos de reglas sodd (campo `carga1`).

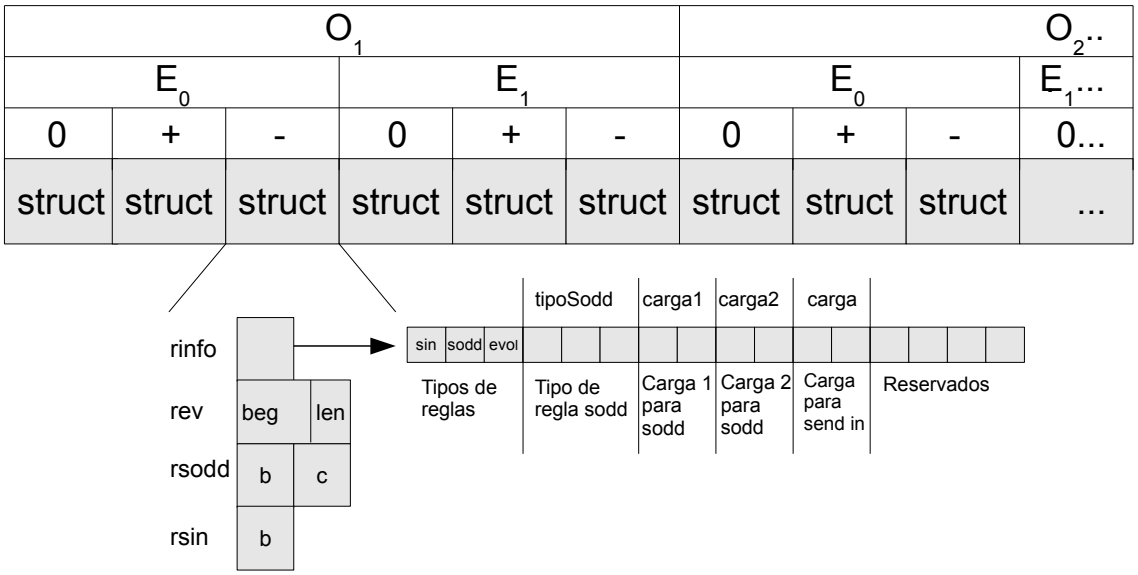

Figura 6.1: Estructura con las reglas ampliada

Igualmente, tenemos otros campos de tipo `ushort` donde indicamos el cambio de elemento que se produce al aplicar la regla, es decir, si se ejecuta una regla de tipo ***send in*** el objeto con el que se corresponde el hilo que lo ejecuta queda decrementado en la membrana `piel`, y en la membrana que está relacionada con ese hilo se introduce el elemento que contenga el campo `rsin`. De manera similar se hace con los tipos de reglas `sodd`, teniendo en cuenta que las reglas de tipo `división` conllevan dos objetos, uno para cada una de las nuevas membranas creadas (campo `rsodd`).

Finalmente, sólo queda ver como se ejecutan las reglas de `evolución`, en las cuales no hay cambios de carga, pero lo que si se produce es una introducción de una cantidad indeterminada de elementos. Para esto nos servimos de una estructura auxiliar (ver Figura 6.2), en la cual quedarán indicados los elementos que deben introducirse mediante una posición de inicio y

otra de fin. Estas posiciones se indican en el campo `rev` de la estructura de las reglas. También debemos de tener en cuenta que se pueden introducir varios objetos de un determinado tipo, por lo que la estructura auxiliar también contiene el campo `multiplicidad` para este propósito.

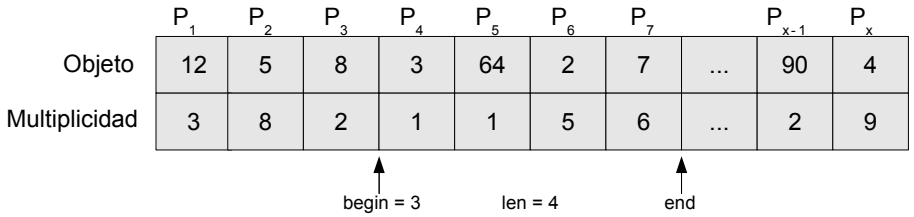

Figura 6.2: Estructura auxiliar para las reglas de tipo `evolución`

6.2.3. Kernel de `Selección`

En este *kernel* el proceso de selección es similar al descrito en el subapartado 5.2.5 del capítulo anterior, sólo que ahora, cuando se ejecuta una regla se decrementa la multiplicad del objeto que esté involucrado (siempre es una unidad en todas las reglas, menos en las de tipo `evolución`).

Además, ahora también se ejecutan las reglas de evolución en este *kernel*. Por lo que una vez terminado el proceso de selección, cada hilo comprueba si para alguno de los objetos con los que está relacionado hay seleccionada alguna regla de tipo `evolución`, y de ser así añade los nuevos objetos pertenecientes al `array` de la Figura 6.2, tantas veces como así esté indicado en esa misma estructura.

6.2.4. Kernel de Ejecución de `Disolución`

Este *kernel*, al igual que todos los de ejecución, su funcionalidad es muy específica y por lo tanto es poco complejo.

En este caso concreto, cada bloque de hilos se corresponde con una membrana y su funcionamiento se puede observar en los siguientes pasos:

1. Obtención de los datos genéricos de toda la membrana (bloque de hilos):
   - De la estructura que contiene la información de las membranas obtenemos la carga y etiqueta de la membrana que es tratada.
   - También se obtiene la regla asociada y su tipo.
2. Si la regla a tratar es de tipo `disolución`:
   - Se añade a la membrana el elemento que le corresponde por ejecutar la regla.
   - Se marca la membrana como disuelta.

- Cada uno de los hilos del bloque añade los elementos que le corresponden de la membrana actual a la membrana padre, en este caso la `piel`. Esto se hace mediante el uso de la operación de adición atómica (`atomicAdd`).

6.2.5. Kernel de Ejecución de `División`

En el *kernel* que se encarga de ejecutar las reglas de división ocurre igual que en el caso anterior, es decir, los bloques de hilos se corresponden con las membranas. La descripción de como se ejecuta en este *kernel* se puede ver a continuación:

1. Obtención de los datos genéricos de toda la membrana (bloque de hilos):
   - De la estructura que contiene la información de las membranas obtenemos la carga y etiqueta de la membrana que es tratada.
   - También se obtiene la regla asociada y su tipo.
2. Si la regla a tratar es de tipo `división`:
   - Se crea una nueva membrana y, en consecuencia, se incrementa de manera atómica el número de membranas en una unidad (utilizando la instrucción `AtomicAdd`).
   - Modificamos la carga de la membrana actual.
   - Le asignamos la carga y etiqueta a la nueva membrana creada.
   - Cada uno de los hilos del bloque se encarga de copiar los objetos que le corresponden en la nueva membrana creada, para así conseguir una réplica de la membrana actual.
   - A cada membrana se le añade el objeto que le corresponde por haber ejecutado una división.

6.2.6. Kernel de Ejecución de *`Send Out`*

En este caso, con que un solo hilo se encargue de una membrana es suficiente, por lo que los bloques de hilos pueden ser de cualquier tamaño, en nuestro caso optamos por 256.

En este `kernel` se realiza lo siguiente:

1. Obtención de los datos de cada hilo:
   - De la estructura que contiene la información de las membranas obtenemos la carga y etiqueta de la membrana que es tratada por el hilo.
   - También se obtiene la regla asociada a la membrana y su tipo.
2. Si la regla a tratar es de tipo *`send out`*:
   - Se modifica la carga de la membrana actual.
   - El debido objeto de la membrana `piel`, indicado en la estructura de las reglas, se incrementa atómicamente en uno.

6.2.7. Kernel de Ejecución de *Send In*

Estamos en las mismas condiciones que en el *kernel* de ejecución de *send out*, salvo que cambian algunos pequeños aspectos:

1. Obtención de los datos de cada hilo:
   - De la estructura que contiene la información de las membranas obtenemos la carga y etiqueta de la membrana que es tratada por el hilo.
   - También se obtiene la regla asociada a la membrana y su tipo.
2. Si la regla a tratar es de tipo *send in*:
   - Se modifica la carga de la membrana actual.
   - El debido objeto de la membrana actual, indicado en la estructura de las reglas, se incrementa atómicamente en uno.

7

Pruebas realizadas y Resultados

En este capítulo se muestra el escenario donde hemos realizados las pruebas, dando las especificaciones del hardware y software utilizado.

También describimos los *sistemas P* que vamos a utilizar como entradas a los tres simuladores que han sido vistos a lo largo de este trabajo.

Finalmente mostramos los resultados obtenidos en los distintos simuladores, haciendo una comparación entre ellos.

7.1. Escenario de Pruebas

El ordenador utilizado para las pruebas tiene las siguientes características hardware:

- Procesador Intel Core2 Quad Q9550 @ 2,83 GHz.
- 8 GB de memoria RAM.

- GPU Tesla C1060, cuyas características pueden ser vistas en la Tabla 3.2.

En cuanto al software utilizado podemos verlo a continuación:

- Sistema Operativo Ubuntu Server 8.04.2 de 32 bits.
- Driver CUDA 185.18.14.
- Toolkit CUDA 2.1.
- SDK CUDA 2.1.
- Los simuladores han sido compilados con la opción -O3.
- Sin instalar entorno gráfico (modo texto).

7.2. P Sistemas Diseñados para las Pruebas

Las pruebas realizadas se basan en probar dos tipos de sistemas P:

- Sistemas P sintético creado exclusivamente para poner a prueba el simulador.
- Sistemas P que resuelven el problema de las N-Reinas.

En el resto de la sección daremos una descripción de ambos tipos.

7.2.1. Sistemas P Sintéticos

Hemos diseñado una familia de sistemas P específicos, donde hay un gran nivel de paralelismo y permite analizar el rendimiento del simulador variando distintos parámetros. Esta familia de sistemas P está basada en ejecutar reglas de tipo `evolución` y así obtener paralelismo a nivel de objeto (hilo) dentro de una membrana (bloque de hilos), y reglas de tipo `división` que permiten crear nuevas membranas en cada paso y así explotar el paralelismo a nivel de bloques de hilos (membranas). Las reglas definidas son las siguientes:

(a) Reglas de evolución: $[o\{i\} \rightarrow o\{i\}]_2^0, 0 <= i < n$

(b) Reglas de división: $[d]_2^0 \rightarrow [d]_2^0\,[d]_2^0$

Solo hay dos niveles de membranas: la membrana `piel` (con la etiqueta 1) y el resto de membranas (con la etiqueta 2).

El sistema P nos permite controlar el número de objetos en el sistema modificando el parámetro n. Por otro lado, el número de reglas aumenta en la misma medida que aumenta el número de objetos. El número de membranas está definido en cada paso como 2^s, donde s es el número de paso. Por último, el número de reglas de tipo `evolución` selecionadas y ejecutadas en cada paso en una membrana es siempre el mismo, tantas como objetos hay en la membrana.

7.2.2. Sistemas P que resuelven las N-Reinas

Por otro lado, hemos probado el comportamiento de los simuladores trabajando con sistemas P que resuelven el problema de las N-Reinas, diseñado por Miguel A. Gutiérrez–Naranjo en [13].

En el trabajo citado se estudia como resolver el problema de las N-Reinas mediante una familia de sistemas P. Para ello se traduce el problema a lógica proposicional, usando la forma normal conjuntiva (pasando a ser el problema SAT). De esta forma, se busca en paralelo, mediante un sistema P, si alguna cláusula es cierta (contiene solución).

El sistema P diseñado es una modificación del sistema P para resolver SAT [12] como un problema de decisión, es decir, devolviendo en el último paso de computación una respuesta `si` o `no`.

Utilizando P-Lingua 2.0 se definen los sistemas P utilizados para resolver los problemas de las 3-Reinas y las 4-Reinas. Para el primero se llegan a tener 512 membranas y hasta 1883 objetos distintos, en cambio, el segundo puede crear hasta 65536 membranas con hasta 8120 objetos diferentes.

Con el problema de las 5-Reinas se generarían hasta 2^{25} membranas y 25574 objetos, lo cual es bastante difícil de conseguir ejecutar por las limitaciones de memoria que tenemos en nuestro sistema. Y las 2-Reinas quedan descartadas debido a que con sólo 4 membranas no se consigue explotar el paralelismo de la GPU.

7.3. Resultados de las Pruebas

Tras ejecutar los sistemas P de prueba que hemos explicado, obtenemos una serie de resultados. Estos quedan reflejados en las gráficas del resto de la sección (en escala logarítmica) junto a una explicación.

Además, todos los datos mostrados en las gráficas pueden ser visto en el Anexo B.

7.3.1. Sistema P Sintético

Usando el sistema P visto en el subapartado 7.2.1 hemos realizado pruebas para comprobar el paralelismo a nivel de membranas, tendremos desde 2 hasta 16384 membranas. Por otro lado comprobaremos el paralelismo entre objetos, incrementado exponencialmente el número de objetos hasta 32768.

Las Figuras 7.1, 7.2 y 7.3 muestran los resultados obtenidos en las simulaciones. En la primera figura puede verse el tiempo que se emplea para llevar a cabo la fase de selección en los tres simuladores. Se puede observar como los simuladores paralelos emplean un menor tiempo que el simulador secuencial. Quizás pueda extrañar que el simulador que es ejecutado completamente en la GPU (simulador paralelo V2) emplea un mayor tiempo que el simulador que sólo lleva a cabo la fase de selección en la GPU (simulador paralelo V1). Esto es debido

a que el simulador V2 además de realizar la selección también ejecuta las reglas de tipo `evolución`. A pesar de esto, se llega a obtener un *speed up* de hasta 7x.

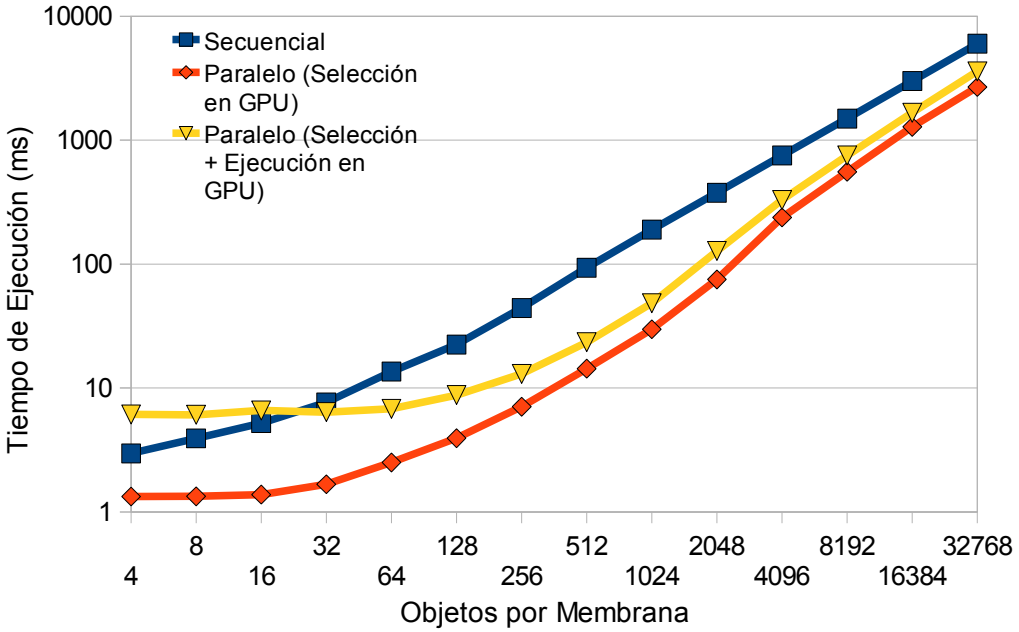

Figura 7.1: Tiempo de Selección en los tres simuladores para el sistema P sintético

Por otro lado, la Figura 7.2 muestra los tiempos tomados para la fase de ejecución, en la que se puede ver como el simulador paralelo V2 obtiene un resultado mejor que los otros dos simuladores, los cuales llevan a cabo la fase de ejecución en la CPU. Esta mejora es obtenida a partir del momento que la membrana tiene más de 32 objetos por membrana, lo cual implica 32 hilos por bloque, que es el tamaño del *warp* en la Tesla C1060. Cuando se utilizan en mayor medida los recursos de la GPU, el simulador paralelo V2 llega a tener un *speed up* de hasta 24x con respecto a la ejecución secuencial de los otros dos simuladores, que tienen tiempos muy similares debido a que el código de la fase de ejecución es prácticamente igual en ambos.

Sin embargo, en los dos simuladores implementados en la GPU hay un tiempo de sobrecarga debido a las transferencias que son realizada a través del bus PCI-Express. En el Anexo B pueden ser vistos todos los datos de tiempos obtenidos tras las simulaciones realizadas, incluidos los tiempos de las transferencias en los simuladores paralelos. La Figura 7.3 ha sido creada teniendo en cuenta los tiempos totales de simulación (también las transferencias), donde se puede observar que las transferencias hacen que el simulador paralelo V1 obtenga unos tiempos similares al simulador secuencial, no obstante el simulador paralelo V2 sigue siendo hasta 8 veces más rápido.

7.3.2. Sistema P que resuelve las N-Reinas

Las tres figuras de esta subsección contienen la información de las simulaciones realizadas con los sistemas P para resolver 3-Reinas y 4-Reinas.

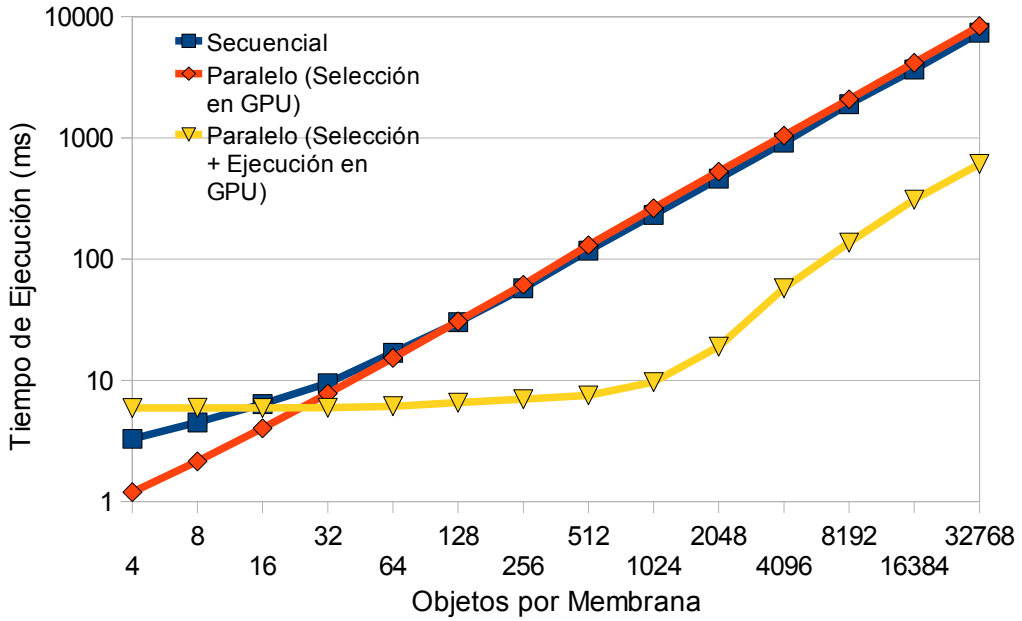

Figura 7.2: Tiempo de Ejecución en los tres simuladores para el sistema P sintético

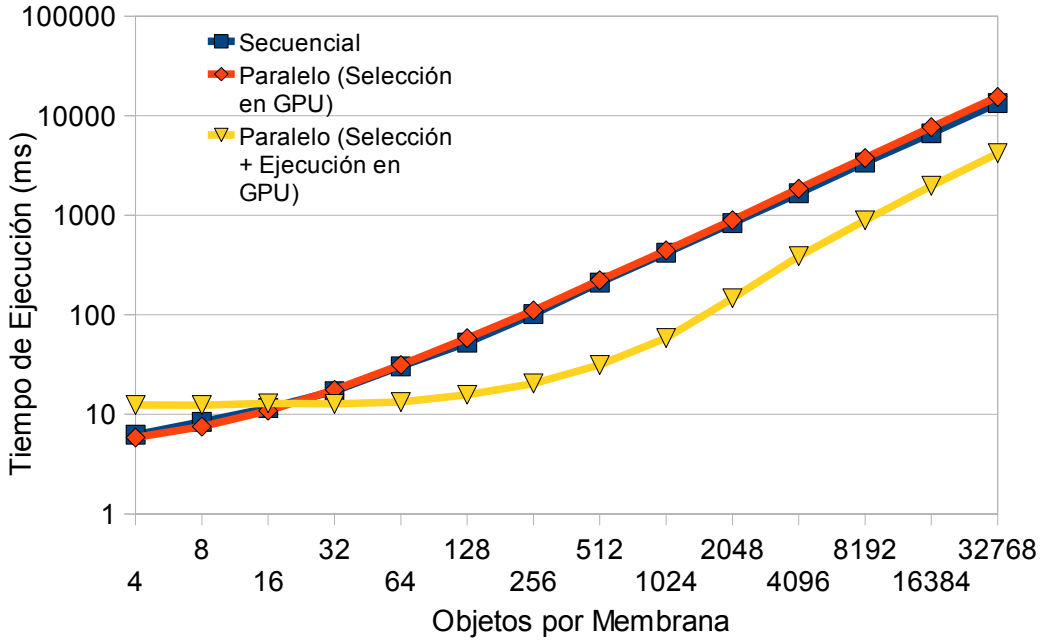

Figura 7.3: Tiempo de simulación total para los tres simuladores para el sistema P sintético

En primer lugar, puede observarse en la Figura 7.4 los tiempos empleados para realizar la fase de selección en ambos sistemas P. Por la misma razón que en los sistemas P sintéticos, el tiempo que tarda en realizar la fase de selección el simulador V2 es algo mayor que el V1. Pero también puede observarse como los simuladores programados en CUDA obtienen un resultado mejor: hasta 3 veces más rápido.

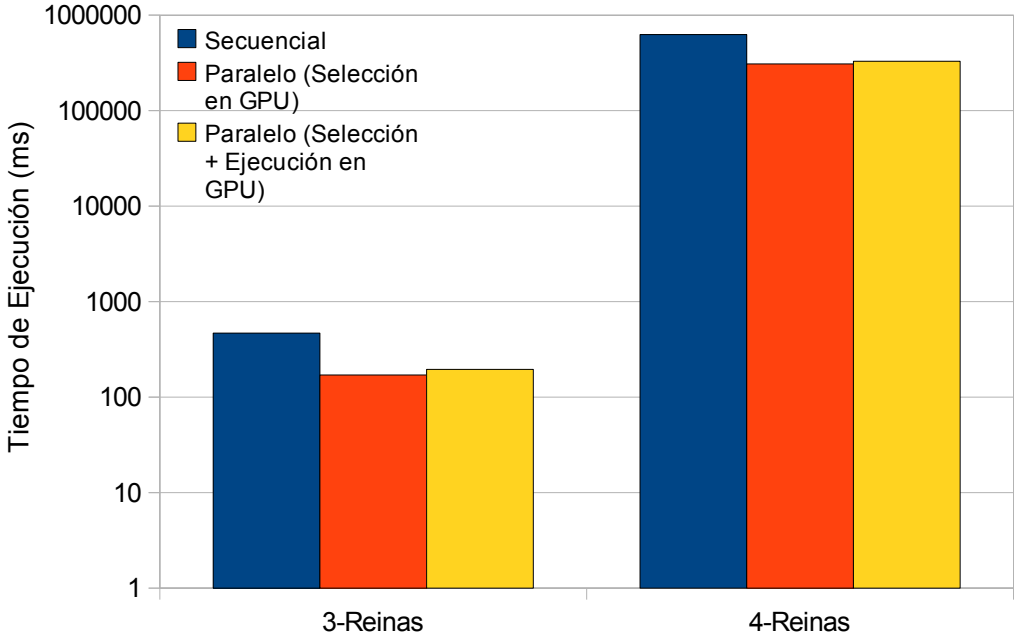

Figura 7.4: Tiempo de Selección de los tres simuladores para las N-Reinas

Por otro lado la Figura 7.5 muestra los tiempos que han sido empleados en la fase de ejecución. Hay una gran simulitud de tiempos entre la version secuencial y la paralela V1, ya que ambas realizan esta fase secuencialmente. En cambio, la versión paralela V2 reduce considerablemente esos tiempos de cómputo. En este caso, el sistema P tiene muchas operaciones de tipo evolución en los últimos pasos, con una gran cantidad de objetos y membranas, las cuales únicamente en la versión paralela V2 son ejecutadas en el *kernel* de selección, haciendo que la mejora obtenida en el tiempo de ejecución llegue a ser del orden de 100x.

Al igual que con las pruebas realizadas en el subapartado anterior, tenemos que tener en cuenta que las versiones realizadas en la GPU hay que realizar transferencias de datos, lo cual conlleva un incremento en el tiempo final de la simulación (ver Figura 7.6). Llegando a tener un *speed up* de casi 4x.

Se puede observar como el *speed up* que se obtiene finalmente en estas pruebas no llega a ser tan elevado como en los sistemas P sintéticos, lo cual es lógico, ya que estos aprovechan mucho mejor los recursos de la tarjeta Tesla. No obstante, también sería interesante estudiar la adaptación del sistema P para resolver el problema de las N-Reinas a la arquitectura de la GPU.

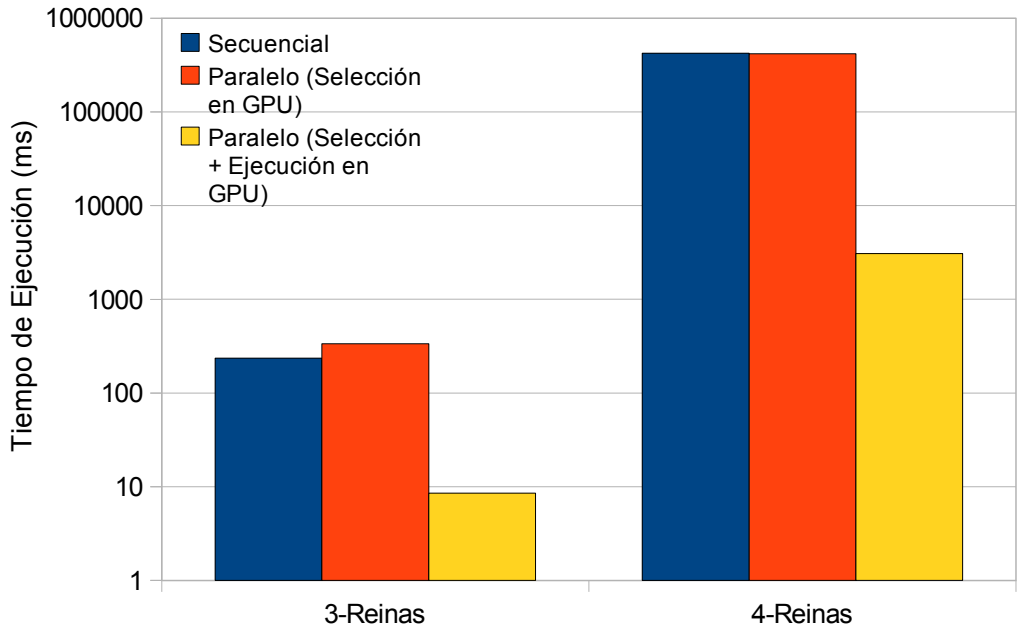

Figura 7.5: Tiempo de Ejecución de los tres simuladores para las N-Reinas

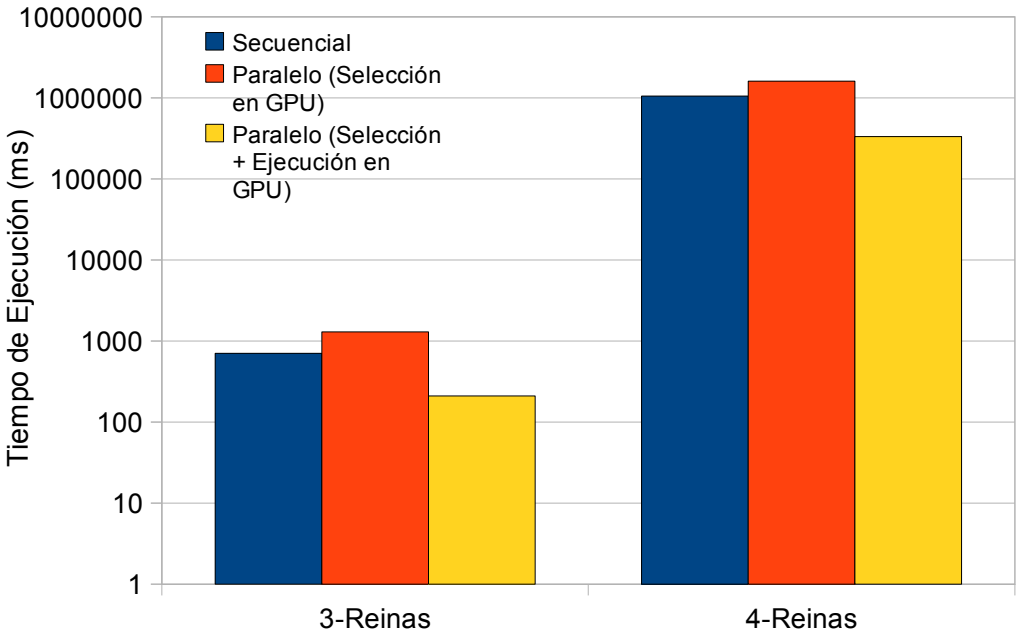

Figura 7.6: Tiempo de simulación total en los tres simuladores para las N-Reinas

8

Conclusiones y Trabajo Futuro

Este capítulo será utilizado para exponer las conclusiones finales a las que se han llegado a lo largo de este trabajo, y como no, indicar cual será la línea de trabajo que se seguirá en el futuro en lo que respecta a este proyecto.

8.1. Conclusiones y Aportaciones

En este documento hemos analizado tres simuladores de sistemas P con membranas activas. El primer simulador ha sido implementando completamente en la CPU, el segundo ha sido implementado usando la CPU y la GPU, y finalmente, un tercer simulador implementado completamente en la GPU.

Las pruebas realizadas demuestran que las GPUs son plataformas eficientes para la simulación de los sistemas P, debido al doble paralelismo que presentan. Un primer nivel de paralelismo lo encontramos entre los objetos que hay dentro de las membranas, que se corresponde con el que hay entre los hilos de la GPU. Y un segundo nivel lo encontramos entre las membranas, las cuales son representadas con los bloques de hilos del modelo de programación CUDA.

Además, las pruebas demuestran que la versión que es completamente ejecutada en GPU consigue mejorar los resultados de la otra versión paralela, puesto que reduce la cantidad de datos transferidos a través del bus PCI-Express, y por otro lado, se consigue aprovechar

mejor el potencial de la GPU, al llevar a cabo ambas fases del algoritmo en esta arquitectura.

Los sistemas P son una herramienta muy interesante para poder abordar problemas NP-Completos, inspirándose en el comportamiento de las células vivas (creando un número exponencial de nuevas membranas en un tiempo polinomial). Asimismo, la computación de membranas está siendo utilizada para modelar sistemas biológicos, y así complementar otros métodos clásicos, es decir, simular el comportamiento de ciertos ecosistemas o de ciertos procesos dentro de las células, como es el caso de la *apoptosis*. Por lo tanto, consideramos que obtener una implementación eficiente de un simulador de sistemas P es muy interesante para futuras investigaciones.

Usando la potencia y el paralelismo que ofrecen las GPUs para simular los sistemas P con membranas activas, los desarrolladores de aplicaciones para la computación de membranas pueden encontrar un apoyo en su investigación al poder realizar las simulaciones más rápidamente, pudiendo ver este simulador como una alternativa a los implementados secuencialmente.

Se han realizado diversas publicaciones derivadas de este trabajo. A continuación pasamos a enumerarlas:

1. **José M. Cecilia, José M. García, Ginés D. Guerrero, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez.**
   Parallel and Ubiquitous methods and tools in Systems Biology.
   Enviado a Brifings in Bioinformatics.
   Special Issue on Parallel and ubiquitous, method and tools.

2. **José M. Cecilia, Ginés D. Guerrero, José M. García, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez.**
   Simulation of P systems with active membranes on CUDA.
   International Workshop on High Performance Computational Systems Biology.
   Trento (Italia). 14 al 16 de Octubre.

3. **Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, José M. Cecilia, Ginés D. Guerrero, José M. García.**
   Simulating active membranes systems using GPUs.
   10th Workshop on Membrane Computing.
   Curtea de Arges (Rumania). 24 al 27 de Agosto, 2009.

4. **José M. Cecilia, Ginés D. Guerrero, José M. García, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez.**
   A massively parallel framework using P systems and GPUs.
   Symposium on Application Accelerators in High Performance Computing.
   Urbana, Illinois (Estados Unidos). 28 al 30 de Julio, 2009.

5. **Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, José M. Cecilia, Ginés D. Guerrero, José M. García.**
   Simulation of Recognizer P Systems by Using Manycore GPUs.
   Proceedings of the 7th Brainstorming Week on Membrane Computing.
   Sevilla, Volumen II, paginas 369-384. Julio, 2009.

6. **Ginés D. Guerrero, José M. Cecilia, José M. García, Miguel A. Martínez-del-Amor, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez.**
Analysis of P systems simulation on CUDA.
XX Jornadas de Paralelismo.
A Coruña. 16 al 18 de Septiembre, 2009.

8.2. Trabajo Futuro

En las siguientes versiones el simulador podrá ser adaptado para resolver casos más específicos (por ejemplo, las N-Reinas), y así poder obtener el máximo rendimiento posible. Por otro lado, también está la posibilidad de ampliar el simulador para poder simular otras variantes de sistemas P, como son los sistemas P probabilísticos o sistemas P estocásticos, los cuales son útiles para hacer modelado computacional dentro del marco de la Biología de Sistemas.

Es también importante remarcar que nuestro simulador está limitado tanto por los recursos de la GPU, como de la CPU (RAM, Memoria de la GPU, etc.). Estos limitan en gran medida el tamaño de las instancias de los problemas NP-completos, y por lo tanto en las siguientes versiones será interesante reducir el uso de memoria.

Es cierto que las GPUs ofrecen altas prestaciones para implementar el simulador, pero hay que ir más allá y pensar en utilizar cluster de GPUs que nos ofrecerán un mayor nivel de paralelismo y mayor cantidad de memoria.

A

Uso de los Simuladores

En este anexo vamos a mostrar como se utilizan los simuladores, dando unos ejemplos de uso, en los que se podrá ver una pequeña muestra de los datos de salida producidos.

Los simuladores son lanzados mediante la línea de comandos invocando al ejecutable `pcuda`, al cual se debe llamar junto a varios parámetros:

- **-v X**: indica el nivel de detalle de la salida de nuestro programa. Por defecto no está activada. Los niveles de detalle son:
  - **-v 1**: muestra solo la última configuración.
  - **-v 2**: muestra la configuración de la membrana `piel` en cada paso, y además, en el último paso la configuración del resto de las membranas.
  - **-v 3**: muestra la información de todas las membranas en todos los pasos, y además, muestra el alfabeto de los objetos y el conjunto de reglas disponibles.
- **-l**: indica el número máximo de pasos que se deben ejecutar. Por defecto toma el valor 256.
- **-p**: indica el simulador con el que se va a llevar a cabo la ejecución. Por defecto es el secuencial. Su uso es el siguiente:
  - **-p 1**: ejecuta el simulador secuencial.
  - **-p 2**: ejecuta el simulador que paraleliza la fase de selección.
  - **-p 3**: ejecuta el simulador completamente paralelizado.
- Los siguientes parámetros son obligatorios:
  - **-i**: indica el fichero binario generado por el compilador P-Lingua que se le pasa como entrada.

- **-m**: indica el número máximo de membranas que se van a crear en el sistema P.
- **-o**: indica el número máximo de objetos que una membrana puede tener en un determinado paso.
- **-b**: indica el número de hilos que va a tener un bloque (este parámetro solo será utilizado cuando se lance una versión paralela).

Supongamos que tenemos un sistema P con membranas activas cuyo fichero binario contiene la siguiente información:

```
Número de objetos: 137
Número de etiquetas: 2
Número de membranas: 2
Número de inicial de multiconjuntos: 1
Número de reglas de evolución: 172
Número de reglas sendin: 7
Número de reglas sendout: 17
Número de reglas de disolution: 0
Número de reglas de division: 6
Número total de reglas: 202
```

Para realizar la simulación secuencial de este sistema P introducimos el siguiente comando: `pcuda -v 3 -i ../pcuda1.4/data/f.bin -m 64 -o 137`. Obteniendo la siguiente información de salida:

```
ALFABETO (objeto, id): (#,0), (No,1), (Si,2), (e,3), (t,4), (c{1},5), (c{2},6),
 (c{3},7), (c{4},8), (c{5},9), (c{6},10), (d{1},11), (d{2},12), (d{3},13), (d{4},14),
 (d{5},15), (d{6},16), (d{7},17), (d{8},18),......

ETIQUETAS (etiqueta, id): (2,0), (1,1)

REGLAS:
  ID57:  [r{1,1} -> r{1,2}]'2
  ID61:  [r{1,2} -> r{1,3}]'2
  ...
  ID174: d{1} []'2 ->  [d{2}]
  ID175: d{2} []'2 ->  [d{3}]
  ...
  ID110: +[nx{1,2} -> nx{1,1}]'2
  ID118: +[nx{1,3} -> nx{1,2}]'2
  ...

CONFIGURACIÓN: 0

ENTORNO:
```

```
MEMBRANA PIEL ID: 0, Etiqueta: 1, Carga: 0
  Multiconjunto:

MEMBRANA ID: 1, Etiqueta: 2, Carga: 0
  Multiconjunto: d{1}, nx{1,2}, nx{2,2}, nx{2,4}, nx{4,6}, x{1,1}, x{2,3}, x{3,5}

REGLAS SELECCIONADAS:
  ID197:  [d{1}]'2 -> +[d{1}] -[d{1}], Membrane id: 1, Times: 1
...

CONFIGURACIÓN: 41

ENTORNO: Si, t

MEMBRANA PIEL ID: 0, Etiqueta: 1, Carga: 2
  Multiconjunto: t*10, c{6}*10, d{29}*64

MEMBRANA ID: 1, Etiqueta: 2, Carga: 1
  Multiconjunto: c{4}, r{0,12}*3

...

MEMBRANA ID: 64, Etiqueta: 2, Carga: 1
  Multiconjunto: c{3}, r{0,12}*3, r{2,12}
```

Como puede observarse la salida muestra información sobre el alfabeto, etiquetas y reglas. Además tras cada paso se va mostrando la información de la configuración de las membranas. En este caso se partirá de la configuración inicial en el paso 0, que sólo tiene una membrana `piel` y una membrana hija, hasta la configuración final del paso 41, donde la membrana `piel` pasa a tener 64 membranas hijas. En esta configuración el entorno contiene el objeto `Si` que nos indica que hay al menos una posible solución.

Sin embargo, si lo que quisiéramos ejecutar fuera el problema de las 3-Reinas (descrito en el subapartado 7.2.2) con el simulador que paraleliza ambas fases, deberíamos ejecutar el siguiente comando:

```
pcuda -i 3-Reinas.bin -p 3 -m 512 -o 1883 -b 269
```

En este caso puede verse como el parámetro **-b** es utilizado para indicar que cada bloque tenga 269 hilos, y por lo tanto a cada hilo le corresponderán 7 de los 1883 objetos.

Para lanzar cualquier otra simulación tan solo habría que adaptar los parámetros según lo que se desee ejecutar.

B

Datos de las Gráficas

En este anexo se podrá encontrar una serie de tablas que se corresponden con los datos de las figuras del capítulo 7. Así se puede realizar un análisis más exhaustivo de los resultados.

Nº Objetos	**Tº Selección**	**Tº Ejecución**	**Tº Total**
4	2,97	3,29	6,25
8	3,92	4,5	8,41
16	5,21	6,37	11,58
32	7,64	9,47	17,11
64	13,55	16,89	30,45
128	22,39	30,11	52,5
256	44,15	57,3	101,45
512	93,42	117,4	210,83
1024	189,11	231,44	420,55
2048	375,86	459,78	835,63
4096	750,93	913,53	1664,45
8192	1490,96	1896,79	3387,75
16384	2993,08	3654,29	6647,37
32768	5996,82	7371,48	13368,3

Tabla B.1: Tiempos de simulación del sistema P sintético en el Simulador Secuencial

Nº Objetos	Tº Selección	Tº Ejecución	Tº Transferencia	Tº Total
4	1,33	1,19	3,33	5,85
8	1,33	2,14	4,06	7,54
16	1,38	4,02	5,49	10,89
32	1,67	7,79	8,22	17,68
64	2,5	15,29	13,61	31,4
128	3,93	30,7	23,59	58,23
256	7,05	61,45	42,39	110,9
512	14,31	130,09	78,68	223,08
1024	29,76	262,59	151,1	443,45
2048	75,01	527,61	291,26	893,89
4096	237,49	1044,94	569,87	1852,3
8192	554,7	2081,83	1125,59	3762,12
16384	1279,52	4163,13	2235,26	7677,91
32768	2680,28	8355,44	4453,17	5488,89

Tabla B.2: Tiempos de simulación del sistema P sintético en el Simulador Paralelo V1

Nº Objetos	Tº Selección	Tº Ejecución	Tº Transferencia	Tº Total
4	6,12	5,91	0,41	12,43
8	6,08	5,93	0,4	12,41
16	6,59	5,91	0,41	12,91
32	6,37	5,94	0,41	12,72
64	6,78	6,1	0,41	13,29
128	8,76	6,54	0,43	15,73
256	12,99	6,99	0,45	20,43
512	23,36	7,51	0,49	31,36
1024	48,32	9,65	0,58	58,55
2048	127,23	18,97	0,75	146,94
4096	329,07	57,52	1,09	387,68
8192	751,38	136,83	1,76	889,98
16384	1665,33	308,95	2,96	1977,24
32768	3584,92	607,59	5,37	4197,88

Tabla B.3: Tiempos de simulación del sistema P sintético en el Simulador Paralelo V2

Problema	Tº Selección	Tº Ejecución	Tº Total
3-Reinas	2468,34	235,74	704,09
4-Reinas	626794	423457	1050251

Tabla B.4: Tiempos de simulación del sistema P de las N-Reinas en el Simulador Secuencial

Problema	Tº Selección	Tº Ejecución	Tº Transferencia	Tº Total
3-Reinas	170,96	336,28	788,67	1295,9
4-Reinas	309952	418984	878109	1607045

Tabla B.5: Tiempos de simulación del sistema P de las N-Reinas en el Simulador Paralelo V1

Problema	Tº Selección	Tº Ejecución	Tº Transferencia	Tº Total
3-Reinas	195,61	8,56	6,2	210,37
4-Reinas	330105	3076,95	37,98	333219,93

Tabla B.6: Tiempos de simulación del sistema P de las N-Reinas en el Simulador Paralelo V2

Glosario de términos

apoptosis
Autodestrucción de las células de forma programada, codificada genéticamente. 2, 50

CC
Capacidad de cómputo. 17

CCIA
Departamento de Ciencias de la Computación e Inteligencia Artificial. 1

CPU
Unidad Central de Procesamiento. 3, 12, 16, 23, 25, 26, 33–36, 44, 49, 51

CUDA
Compute Unified Device Architecture. 2, 3, 11, 16, 17, 20, 23, 24, 26, 27, 46, 49

DITEC
Departmento de Ingeniería y Tecnología de Computadores. 1

GACOP
Grupo de Arquitectura y Programación Paralela. 1

GPGPU
Unidad de Procesamiento Gráfico de Propósito General. 11, 12

GPU
Unidad de Procesamiento Gráfico. 2, 3, 11–13, 16, 19, 21, 23–26, 33, 34, 36, 43, 44, 46, 49–51

mitosis
División celular característica de las células somáticas, que produce dos células hijas que serán genéticamente idénticas a la célula progenitora. 2

MT IU
Unidad de procesamiento multihilo. 13

ODE
Ecuaciones Diferenciales Ordinarias. 2

RGNC
Grupo de Investigación en Computación Natural. 1

SAT
Problema de satisfacibilidad booleana. 2

SFU
Unidad de Funciones Especiales. 13

shader
Procedimiento de sombreado e iluminación que permite al artista/programador especificar el renderizado de un vertex o de un pixel. 11

SIMT
Simple-Instrucción Múltiples-Hilos. 15

SM
Multiprocesador de flujo. 12, 13, 15, 17

SP
Procesador de flujo. 12, 13, 15, 17

SPMD
Simple-Programa Múltiples-Datos. 16

warp
Un conjunto de 32 hilos que se ejecuta en paralelo físicamente dentro del mismo SM. 15, 16

Bibliografía

[1] A. Alhazov, M.J. Pérez–Jiménez. Uniform solution of QSAT using polarizationless active membranes. In J. Durand–Lose and M. Margenstern (eds.) Machines, Computations, and Universality. Lecture Notes in Computer Science, 4664 (2007), pp.122–133.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. SIGGRAPH '04, ACM Press (2004), pp. 777–786.

[3] M. Cardona, M. Angels Colomer, M.J. Pérez-Jiménez, D. Sanuy, A. Margalida. Modeling Ecosystems Using P Systems: The Bearded Vulture,a Case Study. In Proceedings of Workshop on Membrane Computing, Edinburgh, UK (2008), pp. 137–156. 2

[4] S. Cheruku, A. Paun, F.J. Romero–Campero, M.J. Pérez–Jiménez, O.H. Ibarra. Simulating FAS–induced apoptosis by using P systems. Progress in Natural Science, 17, 4 (2007), 424–431 2

[5] G. Ciobanu, M.J. Pérez–Jiménez, G. Paun, editors. Applications of membrane computing. Natural Computing Series, Springer, (2006). 2

[6] D. Díaz–Pernil, I. Pérez–Hurtado, M.J. Pérez–Jiménez, A. Riscos–Núñez. A P–Lingua programming environment for Membrane Computing. In Membrane Computing: 9th International Workshop (WMC08), Lecture Notes in Computer Science, 2009, 187-203. 19

[7] M. García–Quismondo, R. Gutiérrez–Escudero, I. Pérez–Hurtado, M.J. Pérez–Jiménez. P–Lingua 2.0: A software framework for cell-like P systems. International Journal of Computers, Communications and Control, vol. IV (3), 2009, 234-243. 19

[8] M. Garland, S.L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov. Parallel computing experiences with CUDA. IEEE Micro, 28, 4 (2008), 13–27.

[9] N.K. Govindaraju, D. Manocha. Cache–efficient numerical algorithms using graphics hardware. Parallel Comput., 33, 10–11 (2007), 663–684.

[10] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Available membrane computing software. In G. Ciobau, Gh. Păun, M.J. Péréz–Jiménez (eds.) Applications of Membrane Computing, Natural Computing Series, Springer–Verlag, 2006. Chapter 15 (2006), pp. 411–436.

[11] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Towards a programming language in cellular computing. Electronic Notes in Theoretical Computer Science, Elsevier B.V., 123 (2005), 93–110.

[12] M. J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. A polynomial complexity class in P systems using membrane division. In E. Csuhaj–Varjú, C. Kintala, D. Wotschke, G. Vaszil (eds.), Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems, DCFS 2003, Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, (2003), pp. 284–294. 43

[13] M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez, F.J. Romero–Campero. Computational efficiency of dissolution rules in membrane systems. International Journal of Computer Mathematics, 83, 7 (2006), 593–611. 43

[14] M. Harris, S. Sengupta, J.D. Owens. Parallel prefix sum (Scan) with CUDA. GPU Gems 3 (2007).

[15] T.D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, ACM (2008). pp. 15–25.

[16] M.D. Lam, E.E. Rothberg, M.E. Wolf. The cache performance and optimizations of blocked algorithms. ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, ACM (1991), pp. 63–74.

[17] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28, 2 (2008), 39–55. 2, 12, 15, 24

[18] W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard. Cg: a system for programming graphics hardware in a C–like language. SIGGRAPH '03, ACM (2003), pp. 896–907.

[19] J. Michalakes, M. Vachharajani. GPU acceleration of numerical weather prediction. IPDPS, (2008), pp. 1–7.

[20] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable parallel programming with CUDA. Queue, 6, 2 (2008), 40–53. 2, 16

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. Gpu computing. Proceedings of the IEEE, 96, 5 (2008), 879–899.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, T.J. Purcell. A survey of general–purpose computation on graphics hardware. Computer Graphics Forum, 26, 1 (2007), 80–113.

[23] G. Păun. Computing with membranes. Journal of Computer and System Sciences, 61, 1 (2000), pp. 108–143, and Turku Center for Computer Science-TUCS Report No 208. 1, 5

[24] G. Păun: Membrane Computing, An introduction. Springer-Verlag, Berlín (2002). 5

[25] M.J. Pérez–Jiménez, A. Riscos–Núñez. Solving the Subset–Sum problem by active membranes. New Generation Computing, 23 (2005), 367–384. 2

[26] M.J. Pérez–Jiménez, F.J. Romero–Campero. An efficient family of P systems for packing items into bins. Journal of Universal Computer Science, 10, 5 (2004), 650–670.

[27] M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrinini. A polynomial complexity class in P systems using membrane division. Journal of Automata, Languages and Combinatorics, 11, 4 (2006), 423–434. 2

[28] M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. Complexity classes in models of cellular computing with membranes. Natural Computing, 2, 3 (2003), 265–285.

[29] A. Ruiz, M. Ujaldon, J.A. Andrades, J. Becerra, K. Huang, T. Pan, J.H. Saltz. The GPU on biomedical image processing for color and phenotype analysis. BIBE, (2007), pp. 1124–1128. 2

[30] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, W. mei Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (2008), pp. 73–82.

[31] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, Sain-Zee Ueng, S.S. Baghsorkhi, W.W. Hwu. Program optimization carving for GPU computing. J. Parallel Distrib. Comput., 68, 10 (2008), 1389–1401.

[32] N. Satish, M. Harris, M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. To Appear in Proceedings of the 23rd IEEE 4, May 2009. 2, 17

[33] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, vol. 28, no. 2, pp. 39-55, March/April, 2008. 16

[34] M. Ito, C. Martín-Vide, Gh. Păun. A characterization of Parikh sets of ET0L languages in terms of P systems. In Words, semigroups and transducers (M- Ito, Gh. Păun, S. Yu, eds.), 239-254, Word Scientific, Singapore 2001. 9

[35] M. Madhu, K. Krithivasan. P systems with membrane creation: Universality and efficiency. Lecture Notes in Computer Science, 2055, 276-287, 2001 9

[36] A. Păun, Gh. Păun. The power of communication: P systems with symport/antiport. New Generation Computing, 20, 3, 295-305, 2002. 7

[37] Păun, Gh. P Systems with active membranes: attacking NP complete problems, *Journal of Automata, Languages and Combinatorics* **6** (1), 2001, 75–90. 8

[38] A. Obtulowicz. Probabilistic P systems. Lecture Notes in Computer Science, 2597, 377-387, 2002. 9

[39] D. Tarditi, S. Puri, J. Oglesby, Accelerator: using data parallelism to program GPUs for general-purpose uses. Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006. 12

[40] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan. Brook for GPUs: stream computing on graphics hardware, ACM Transactions on Graphics 23 (3) (2004) 777–786. 12

[41] M. D. McCool, Metaprogramming GPUs with Sh, AK Peters, 2004. 12

[42] NVIDIA CUDA Programming Guide 2.0, (2008): http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf 2, 16

[43] GPGPU organization. World Wide Web electronic publication: www.gpgpu.org 11

[44] NVIDIA CUDA. World Wide Web electronic publication: www.nvidia.com/cuda 12, 13, 16

[45] J. Kessenich, D. Baldwin, R. Rost, The OpenGL shading language. http://www.opengl.org/documentation/glsl 11

[46] Microsoft, DirectX 10. http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx 11

[47] ATI CTM Guide 1.01 (2006) http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf 12