Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN

ALGORITMO DE OPTIMIZACIÓN DE COLONIA DE HORMIGAS (ACO) EN LA GPU APLICADO A PROBLEMAS DE SEÑALIZACIÓN CELULAR EN EL TRATAMIENTO DE ENFERMEDADES

Estudiante: Melisa Barro Hermida

Dirección: Patricia González Gómez

Margarita Amor López

A Coruña, Febrero de 2023.

A todas las personas importantes que me apoyaron a lo largo de esta etapa

Agradecimientos

A mi familia y amigos, por todo el apoyo y paciencia que me dieron. A mis tutoras, por todos los consejos, ayudas y material para realizar este trabajo.

Resumen

Los modelos computacionales se han convertido en modelos muy populares para el análisis del funcionamiento de redes bioquímicas complejas, como las involucradas en las redes de señalización celular. Estas soluciones construyen modelos lógicos predictivos entrenando una red de conocimiento previo con datos bioquímicos obtenidos a través de experimentos. El entrenamiento se presenta como un problema de optimización combinatoria que requiere métodos de solución eficientes.

La metaheurística denominada Optimización de Colonia de Hormigas, o *Ant Colony Optimization* (ACO), es uno de los métodos empleados en la resolución de problemas combinatorios de complejidad NP. El objetivo de este trabajo es el estudio y análisis de una sencilla implementación paralela de un ACO adaptado a los problemas descritos anteriormente haciendo uso de la plataforma CUDA (*Compute Unified Device Architecture*). Así, valoraremos la efectividad y eficiencia de optimización en la GPU (*Graphics Processing Unit*) obtenida gracias a esta plataforma, y las mejoras que se podrían conseguir de cara al futuro.

Abstract

Computational models became very popular for analysing the functioning of complex biochemical networks such as those involved in cell signalling networks. These solutions build complex predictive logic models by training a prior knowledge network with biochemical data obtained through experiments. Training is presented as a combinatorial optimisation problem that requires efficient solution methods.

The metaheuristic *Ant Colony Optimization* (ACO) is one of the methods used in solving NP-complex combinatorial problems. In this work we will study and analyse a simple parallel implementation of an ACO algorithm adapted to the problems described above using the CUDA (*Compute Unified Device Architecture*) framework. Thus, we will evaluate the effectiveness and efficiency of the optimisation in the GPU (*Graphics Processing Unit*) obtained using this framework, as well as the improvements that could be achieved in the future.

Palabras clave:

- ACO
- CUDA
- GPU
- Optimización
- Computación paralela

Keywords:

- ACO
- CUDA
- GPU
- Optimization
- Parallel computing

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo describe brevemente el proyecto y sus objetivos. También expone las tecnologías y herramientas empleadas en el mismo, y las fases de desarrollo con su planificación temporal. El capítulo termina mostrando la estructura del documento.

1.1 Objetivos del proyecto

El principal objetivo de este trabajo es estudiar la proyección de un algoritmo de optimización en la GPU, en concreto el algoritmo de Optimización por Colonia de Hormigas, utilizando como caso de estudio el entrenamiento de redes de señalización celular.

El algoritmo de Optimización por Colonia de Hormigas (*Ant Colony Optimization* - ACO) es una metaheurística empleada en la resolución de problemas de optimización de gran complejidad. Las metaheurísticas son métodos de resolución estocásticos que se utilizan en problemas muy complejos, donde los métodos deterministas son inviables por el coste computacional o de tiempo de ejecución. Aunque las metaheurísticas requieren un tiempo de ejecución menor que los métodos deterministas, siguen necesitando tiempos de cálculo elevados para obtener resultados razonables en problemas difíciles o de gran extensión. Por este motivo, explorar soluciones que exploten eficientemente los recursos computacionales, como las soluciones de computación de altas prestaciones (*High Performance Computing* - HPC), cobran gran importancia. Estas soluciones permiten reducir los tiempos de ejecución gracias a la paralelización del código. Entre las soluciones HPC posibles, destaca el uso de GPUs (*Graphics Processing Units*).

En este trabajo se estudia un algoritmo ACO aplicado al entrenamiento de redes de señalización celular. En este problema, una de las etapas necesita la ejecución de un método de optimización, que inicialmente estaba siendo resuelto con un algoritmo genético y que

posteriormente mejoró al ser sustituído por una versión secuencial del ACO. Con todo, para problemas grandes, los tiempos de ejecución son excesivos, por lo que se están explorando distintas estrategias de paralelización. En concreto, en este trabajo se estudia la proyección del algoritmo de optimización en la GPU para mejorar su eficiencia.

Para la proyección en la GPU se hará uso de la plataforma CUDA (*Compute Unified Device Architecture*) [5]. Esta plataforma, desarrollada por Nvidia, aprovecha las ventajas de la GPU frente a la CPU haciendo uso de del paralelismo que ofrecen sus múltiples núcleos, que permiten la ejecución de un gran número de hilos simultáneos. Así, cuando un programa está diseñado empleando varios hilos que realizan tareas independientes, su ejecución será más eficiente si tiene lugar en una GPU [2].

Para alcanzar el objetivo del proyecto, se empleará como base un código secuencial de ACO adaptado para el problemas de redes de señalización celular, implementado por el Grupo de Arquitectura de Computadores de la Universidad de A Coruña (UDC), y publicado en [6].

1.2 Metodología y herramientas

En esta sección se describe la metología seguida en el proyecto, así como las herramientas principales y de apoyo empleadas.

1.2.1 Metodología seguida

Para el desarrollo de este trabajo se ha seguido la metodología Scrum [7], ya que se adapta muy bien a las características del proyecto. Esta metodología está basada en realizar entregas parciales del producto, de forma que se pueden corregir fallos a medida que se localizan o adaptar el producto a los cambios que puedan surgir, con el objetivo de obtener el mejor resultado final posible.

En Scrum, los equipos son autónomos, es decir, se autoorganizan y no necesitan a una figura externa que los gestione. Además, cada miembro del equipo tiene un rol y funciones asignadas. Los equipos Scrum mantienen el control analizando los avances en los puntos de control establecidos a lo largo del proyecto.

De cara al funcionamiento de la metodología, es importante entender sus elementos y sus fases. El primer paso consiste en elaborar el *Product Backlog*, un documento genérico que recoge todos los requisitos y tareas necesarias para el proyecto. En el segundo paso se elabora el *Sprint Backlog*, que es el documento que recoge las tareas a realizar y las personas asignadas

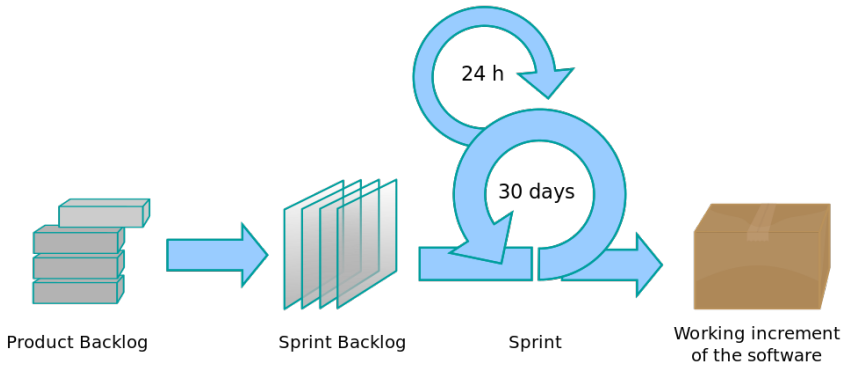


Figura 1.1: Fases en SCRUM.

a ellas, así como el número de horas dedicadas y el coste asociado. El elemento principal de Scrum es el *Sprint*, y es el período de tiempo en el que se realizan todas las tareas pactadas en el *Sprint Backlog*. Al final de cada *Sprint* tiene lugar una entrega parcial para realizar análisis y revisiones de cara al producto final. Este final de *Sprint* se conoce como *Sprint Review*, y ha de ser visto como una oportunidad para obtener *feedback* sobre el desarrollo del producto [7]. En la Figura 1.1 tenemos una vista en detalle de las fases de la metodología.

En este trabajo se ha adaptado la metodología, ya que en el equipo hay un único desarrollador y dos directoras. El trabajo se dividió en una serie de *Sprints* de duración aproximada de un mes. Al inicio de cada uno se realizó un *Sprint Planning* para establecer las tareas a realizar. Al final de cada *Sprint* tuvo lugar su *Sprint Review* correspondiente, en el cual, con la participación de las directoras, se revisaba el trabajo realizado y se establecían nuevos objetivos de acuerdo al avance obtenido. Los *Sprints* establecidos para el proyecto han sido 5, acordes a cada fase del trabajo:

- **Sprint 0:** documentación y búsqueda de información.
- **Sprint 1:** adaptación del algoritmo secuencial de cara a la proyección.
- **Sprint 2:** diseño e implementación de una primera versión.
- **Sprint 3:** desarrollo de una segunda versión mejorada.
- **Sprint 4:** elaboración de la documentación.

1.2.2 Herramientas empleadas

Durante el desarrollo de este trabajo, se usaron una serie de tecnologías y herramientas que se detallan a continuación:

- **Lenguaje de programación *C*:** el código secuencial que se usó de base está escrito en lenguaje *C*. En el resto del trabajo se mantuvo como lenguaje de programación principal, ya que también es uno de los lenguajes que soportan las librerías de CUDA.
- **CUDA:** para la proyección del código en GPU, se ha usado la plataforma CUDA, ya que nos permite alcanzar nuestro objetivo ofreciendo un modelo de programación sencillo e integrado con el lenguaje principal, *C*.
- **Git:** para el control de versiones se ha escodigo este sistema. Entre otras cosas, se ha escogido debido a su sencillez, popularidad y por ser el sistema empleado a lo largo de la carrera.
- **Github:** como repositorio final para presentar el código, se ha elegido esta herramienta por los mismos motivos que Git. El código lo podemos encontrar en el repositorio [8].

Además de las anteriores, se han usado como apoyo las siguientes herramientas:

- **Oracle VM Virtualbox:** todo el desarrollo del proyecto tuvo lugar en una máquina virtual Ubuntu 20.04, tanto para la edición del código como para la conexión al clúster donde se hicieron todas las pruebas. Virtualbox es la herramienta con la que más nos familiarizamos en la carrera de cara a la creación y gestión de máquinas virtuales.
- **Excel:** las tablas empleadas para almacenar los resultados de las pruebas han sido realizadas en hojas de cálculo Excel.
- **Latex:** herramienta escogida para desarrollar la memoria.
- **Overleaf:** plataforma online empleada para la elaboración de la documentación.

Por último, para las pruebas se ha utilizado la infraestructura del clúster *Pluton* [9] de la Universidad de A Coruña. *Pluton* cuenta con nodos que tienen GPUs Nvidia de arquitectura Kepler y Turing, necesarias para el desarrollo de este proyecto.

1.3 Planificación y costes

En esta sección se describirán las fases y tareas del trabajo, así como el presupuesto asociado al mismo.

1.3.1 Fases y tareas

Este proyecto se divide en 5 fases que fueron desarrolladas en orden cronológico:

1. **Estudio de ACO y CUDA:** aprendizaje e investigación sobre los conceptos básicos tanto de ACO como de CUDA.
2. **Adaptación de un algoritmo ACO en secuencial:** proceso de familiarización con el código secuencial que servirá de base en el proyecto. En esta fase se modificaron las estructuras de datos empleadas en el programa para facilitar su posterior proyección en GPU.
3. **Diseño e implementación de una versión en GPU:** partiendo del código secuencial, se realizó una primera versión para ejecutar en GPU haciendo uso de CUDA.
4. **Implementación de una versión mejorada:** tras realizar las pruebas de la primera versión, fueron evaluadas para implementar mejoras en el código y obtener mejores resultados.
5. **Documentación:** elaboración de la memoria en base a los resultados obtenidos.

En la Tabla 1.1 se muestran las tareas concretas llevadas a cabo dentro de cada fase, con la estimación de horas aproximada para cada una. A lo largo del desarrollo del trabajo, se han sufrido retrasos con respecto a la planificación inicial, que se pueden apreciar en la Tabla 1.2.

El proyecto se inicia a finales de marzo de 2022 e inicialmente estaba pensado para presentarse en la convocatoria de septiembre de ese mismo año. A parte de los propios retrasos, la coincidencia de períodos vacacionales, así como los trabajos externos, entregas finales y exámenes dieron lugar a que la fecha final de presentación sea en la convocatoria de febrero de 2023. Las fechas concretas relativas a cada tarea se describen en la Tabla 1.3.

El diagrama de Gantt correspondiente al proyecto se muestra en la Figura 1.2. En él se puede ver de forma detallada los períodos de tiempo que abarcaron cada tarea y las dependencias entre ellas. A continuación se describen las más importantes:

- Entre T1 y T2 hay una dependencia fin-comienzo, ya que es más lógico y sencillo entender primero el funcionamiento del algoritmo ACO antes que la propia programación paralela, ya que a medida que te familiarizas con el algoritmo, es más fácil entender y ver su naturaleza paralelizable. También se decidió empezar por ACO por motivos de comprensión, ya que en el propio grado no se estudia nada relativo a CUDA y familiarizarse con ello es un proceso más lento.

- Entre T2 y T3 existe otra dependencia fin-comienzo. En este caso no es estrictamente necesaria, ya que la configuración de *Pluton* podría haberse realizado paralelamente. Sin embargo, como la idea era estudiar un pequeño programa integrado con CUDA, se tomó la decisión de primero documentarse al respecto, luego configurar el entorno y así probar el programa ya en *Pluton*.
- Una vez terminada T3, se puede comenzar T4. De igual manera, al finalizar las adaptaciones en el código secuencial en T4, puede dar comienzo T5 y T6.
- T5 y T6 se realizan de forma paralela, de ahí la dependencia comienzo-comienzo. Una vez se empieza a implementar la primera versión, comienzan las pruebas para comprobar si se está implementando correctamente.
- Tras obtener la primera versión funcional, se puede empezar a implementar la segunda versión. Así, T7 y T8 siguen la misma filosofía que T5 y T6, con relaciones comienzo-comienzo. Una vez finalizada y probada la segunda versión, comienza la tarea de elaboración de la memoria, T9.
- TS, que es la tarea de supervisión por parte de las directoras, está presente a lo largo de todo el proyecto.

1.3.2 Costes

El equipo de desarrollo de este trabajo está formado por tres personas: la estudiante encargada de la realización del proyecto, y las dos directoras que se dedicaron a la supervisión. En la Tabla 1.4 se puede apreciar la estimación de las horas que cada recurso dedica al proyecto, el coste por hora y el coste total aproximado del trabajo. Para los cálculos se han empleado los sueldos de 11€/hora para la estudiante y 23€/h para cada directora. Esta información ha sido consultada en [10] y [11]. No se han tenido en cuenta otros recursos porque ya estaban disponibles al inicio del proyecto.

Fase	Tarea	Descripción	Horas
	T0	Planificación del trabajo	5
F1	T1	Estudio y documentación sobre ACO	10
F1	T2	Estudio y documentación sobre CUDA	10
F1	T3	Preparación de Pluton y estudio de un código en CUDA	20
F2	T4	Adaptaciones del código secuencial de cara a la proyección	40
F3	T5	Implementación de la primera versión	40
F4	T6	Pruebas de la primera versión	60
F3	T7	Implementación de la segunda versión	60
F4	T8	Pruebas de la segunda versión	80
F5	T9	Elaboración de la memoria	60
	TS	Tareas de supervisión	50
		Total	**435**

Tabla 1.1: Desglose de tareas con sus horas estimadas.

1.4 Estructura de la memoria

El documento se divide en 5 capítulos contando con este, descritos a continuación:

- **Capítulo 1:** consiste en una breve introducción, explicando los objetivos del proyecto, las herramientas empleadas en él y sus fases y planificación.
- **Capítulo 2:** contiene explicaciones sobre los conceptos necesarios para comprender el proyecto. Se describen los métodos de optimización y las metaheurísticas, y, en concreto, el algoritmo ACO y sus aplicaciones para la señalización celular. También se explican los fundamentos de la proyección del código en GPU y de la plataforma CUDA, así como las distintas arquitecturas de GPUs usadas de Nvidia.
- **Capítulo 3:** describe las implementaciones del algoritmo ACO. En primer lugar explica la implementación secuencial, que es la base para el desarrollo de la versión paralela. También recoge las adaptaciones realizadas a la versión secuencial para facilitar la implementación paralela. El capítulo termina describiendo dos versiones en GPU.

Tarea	Horas estimadas	Horas finales	Desviación
T0	5	5	0
T1	10	14	4
T2	10	27	7
T3	20	23	3
T4	40	51	11
T5	40	44	4
T6	60	62	2
T7	60	68	8
T8	80	81	1
T9	60	90	30
TS	50	54	4
Total	**435**	**509**	**74**

Tabla 1.2: Desviación en horas del proyecto.

Tarea	Fecha de inicio	Fecha de fin
T0	23/03/2022	23/03/2022
T1	1/04/2022	22/04/2022
T2	23/04/2022	31/05/2022
T3	01/06/2022	28/06/2022
T4	29/06/2022	27/07/2022
T5	28/07/2022	05/10/2022
T6	28/07/2022	05/10/2022
T7	06/10/2022	10/12/2022
T8	06/10/2022	10/12/2022
T9	11/12/2022	10/02/2023

Tabla 1.3: Vista en detalle de las fechas para cada tarea.

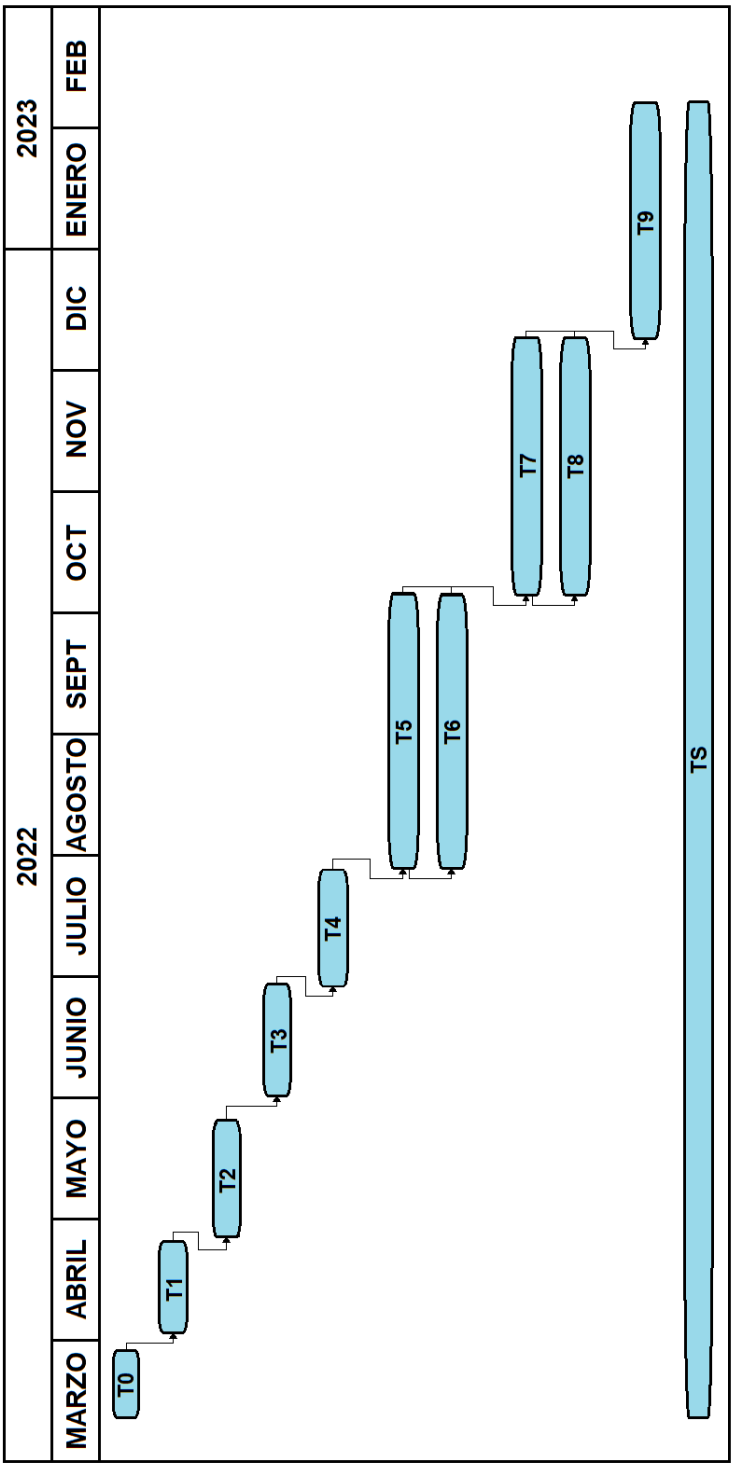


Figura 1.2: Diagrama de Gantt del proyecto.

Recurso	Horas estimadas	Coste por hora (€/h)	Coste (€)
Estudiante	435	11	4.785
Directora 1	25	23	575
Directora 2	25	23	575
	Total		**5.935**

Tabla 1.4: Coste del proyecto.

- **Capítulo 4:** expone los resultados obtenidos a lo largo del proyecto y una comparación entre ellos. También describe brevemente los problemas empleados para las pruebas, y la infraestructura en la que se llevaron a cabo.
- **Capítulo 5:** presenta las conclusiones finales alcanzadas tras el desarrollo del proyecto y las posibles líneas futuras.

Capítulo 2

Conceptos previos

Este capítulo describe los conceptos necesarios para la compresión del trabajo, profundizando en el algoritmo escogido para el proyecto. También explica las técnicas y los conceptos básicos de las arquitecturas de las tarjetas gráficas o GPUs (*Graphics Processing Unit* - GPU) y de su programación usando CUDA.

2.1 Métodos de optimización y metaheurísticas

Los problemas de optimización combinatoria son interesantes porque normalmente son sencillos de plantear, pero difíciles de resolver. En general, podemos clasificar los problemas según su complejidad de la siguiente forma [12]:

- **Problemas P:** las soluciones a estos problemas pueden ser calculadas por los ordenadores en una cantidad de tiempo razonable, es decir, de orden polinómica.
- **Problemas NP:** las soluciones a estos problemas no pueden ser calculadas en un tiempo de cálculo de orden polinómica.
- **Problemas NP-completos:** son una subclase de los problemas NP. Son considerados problemas clave, ya que encontrar una forma de resolverlos eficientemente para uno de ellos implica que todos los problemas NP se pueden resolver de forma eficiente.

Muchos de los problemas que podemos encontrar en la actualidad son NP-completos. Hay distintas maneras de abordar la solución a estos problemas complejos, clasificadas en métodos deterministas y métodos estocásticos. Los métodos deterministas normalmente exploran el espacio de búsqueda en su totalidad, devolviendo como solución el óptimo global. Por ello, el esfuerzo y coste computacional suele ser muy grande, de forma que estos métodos suelen no ser prácticos para problemas grandes.

Por lo tanto, para resolver estos problemas, muchas veces hay que emplear métodos estocásticos, que son métodos aproximados que devuelven soluciones casi óptimas en un tiempo relativamente corto. En esta categoría se encuentran las heurísticas, que hacen uso de los conocimientos específicos de cada problema para construir o mejorar sus soluciones [13]. Además de las heurísticas, también se incluyen las metaheurísticas, en las cuales nos centraremos a continuación.

Una metaheurística se puede definir como un método heurístico aplicable a un conjunto amplio de problemas diferentes. El uso de las metaheurísticas aumentó enormemente la capacidad de encontrar soluciones en un tiempo razonable y de alta calidad a problemas de optimización combinatoria difíciles. Las metaheurísticas operan mediante algoritmos especiales, considerados así principalmente porque no se rigen por un patrón predictivo, ni organizado, sino aleatorio. Estos algoritmos alcanzan su forma óptima mediante iteraciones o pruebas que aproximan la solución.

Aunque cada metaheurística tiene sus propias características, todas tienen en común una serie de componentes fundamentales y operaciones. Esto permite clasificarlas siguiendo una serie de criterios [13], como se puede apreciar en la Figura 2.1:

- **Búsqueda basada en población frente a búsqueda basada en una única solución:** los algoritmos basados en poblaciones, como la evolución diferencial (*Differential Evolution* - DE) o la búsqueda dispersa (*Scatter Search* - SS), almacenan un conjunto de soluciones en poblaciones, que son manejadas y modificadas por el algoritmo, mientras que los algoritmos basados en una única solución, como el recocido simulado (*Simulated Annealing* - SA), tienen una única solución que evoluciona durante la búsqueda.
- **Inspiradas en la naturaleza frente a las no inspiradas:** existen muchos algoritmos con estrategias inspirados en elementos de la naturaleza, como los algoritmos genéticos (*Genetic Algorithms* - GAs), la optimización de colonias de hormigas (*Ant Colony Optimization* - ACO), el enjambre de partículas (*Particle Swarm* - PSO) o el recocido simulado (SA). Otros métodos no están inspirados en la naturaleza, como la evolución diferencial (DE), la búsqueda dispersa (SS) o la búsqueda tabú (*Tabu Search* - TS).
- **Métodos que usan memoria frente a los que no usan:** algunos algoritmos hacen uso de una memoria para guardar la información obtenida durante la búsqueda, como las memorias en los métodos de búsqueda tabú (TS).
- **Métodos iterativos frente a codiciosos (*greedy*):** los algoritmos codiciosos parten de una solución vacía a la que, en cada paso, se le asigna una variable de decisión del problema hasta que se alcanza una solución completa. Los métodos iterativos parten de una

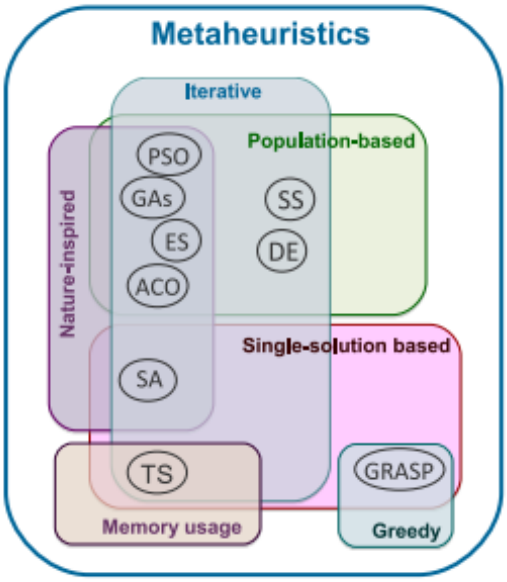


Figura 2.1: Clasificación de metaheurísticas [1].

o varias soluciones, que evolucionan en cada iteración mediante algunos operadores de búsqueda. La mayoría de las metaheurísticas son métodos iterativos.

La metaheurística en la que nos centramos en este trabajo, el ACO, es especialmente exitosa porque se inspira en el comportamiento de las hormigas en la vida real. A partir del sistema de hormigas, se desarrollaron varios enfoques algorítmicos basados en las mismas ideas y se aplicaron con un éxito considerable a una gran variedad de problemas de optimización combinatoria, para aplicaciones tanto académicas como relacionadas con el mundo real [14].

2.2 Problemas de señalización celular en el tratamiento de enfermedades

Los problemas de optimización aparecen en multitud de ámbitos de la vida real, desde problemas de ingeniería clásicos como sistemas de control, planificación de rutas, o problemas de predicción de comportamientos, y, en el caso que se refiere a este trabajo, diseño de fármacos personalizados para tratamiento de enfermedades como el cáncer.

En este contexto, existen diferentes modelos para el descubrimiento de fármacos. Uno de ellos consiste en el modelado lógico de la transducción de señales en las membranas celulares durante la enfermedad y durante la acción de un fármaco, y su comparación.

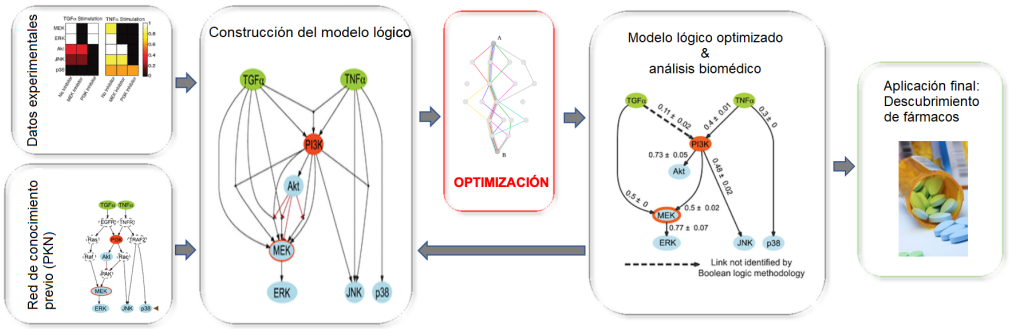


Figura 2.2: Aplicación de un modelo lógico al descubrimiento de fármacos, donde se señala el paso de optimización necesario.

En un contexto dado y para un tipo de células específico, el estudio de cómo éstas procesan señales a través de redes complejas y dinámicas es esencial para entender la señalización tanto a nivel fisiológico como referido a enfermedades. Existe un método que combina conocimiento estático y sin contexto específico sobre las redes de señalización con datos de perturbación [15]. Mediante el entrenamiento de redes de conocimiento previo (*Prior Knowledge Networks* - PKN) contra datos experimentales, se producen modelos con una capacidad predictiva mayor. La construcción del modelo está implementada empleando un formalismo basado en la lógica de que las relaciones entre especies están descritas mediante puertas lógicas que representan el estado de cada nodo teniendo en cuenta el estado de sus nodos padres. Posteriormente, los modelos generados automáticamente son entrenados contra los datos, siendo así útiles para entender cómo se procesan las señales en las células y como se pueden alterar para predecir el efecto de las perturbaciones. Esquemáticamente, en la Figura 2.2 se resume el método, donde se resalta el paso de optimización, que es lo que necesita el algoritmo ACO estudiado en este trabajo.

El código utilizado en este trabajo no ha sido todavía usado en el método completo, pero hace uso de una función que compara los resultados obtenidos por el propio programa contra datos reales. La diferencia entre estos resultados sirve para medir la eficacia del programa, de forma que el objetivo principal es minimizar esta función. Para minimizar este tipo de problemas, normalmente se requieren métodos de optimización global, y aunque se pueden emplear métodos determinísticos, el esfuerzo computacional que requieren para asegurar el óptimo global puede ser muy grande. Teniendo en cuenta esto, es evidente que estos métodos realmente no son eficaces para problemas reales, dando paso a los métodos estocásticos, particularmente a las metaheurísticas, que sí permiten encontrar soluciones cercanas a los

óptimos globales en tiempos de computación razonables.

Sin embargo, un problema en el entrenamiento de redes de señalización celular cuando se emplean metaheurísticas es que éstas no garantizan encontrar un óptimo global. Para poner solución a este problema, en [6] se propuso el uso del algoritmo ACO, ya que dentro de las metaheurísticas, algunas variantes de ACO tienen demostrada su convergencia. Sin embargo, es díficil estimar la velocidad teórica de esta convergencia, motivo por el cual en este trabajo nos centraremos en una implementación paralela del algoritmo, que ayuda a reducir el tiempo de ejecución. En [6] se proponían varias estrategias paralelas pero centradas en la programación de CPUs, usando paradigmas de memoria distribuída y memoria compartida. En este trabajo exploraremos la posibilidad de proyectar de forma eficiente este algoritmo ACO en la GPU.

2.2.1 Algoritmo ACO

El algoritmo de Optimización por Colonia de Hormigas (ACO) replica, de forma artificial, el comportamiento real que tienen las hormigas a la hora de buscar alimento. En la naturaleza, cuando buscan comida, las hormigas recorren caminos aleatorios partiendo de su hogar, de forma que van dejando un rastro de feromonas cuando vuelven al hormiguero. Este fenómeno se conoce como estigmergia.

Biológicamente, cuando las hormigas detectan las feromonas, se sienten atraídas hacia esos caminos, por lo que hay más posibilidades de que utilicen el recorrido con más concentración de feromonas. De esta forma, se favorece la aparición del camino más corto, que de forma iterativa acumulará más feromonas por el sucesivo paso de hormigas a través de él. Además, el rastro de feromonas también sufre un proceso de evaporación, por lo que los caminos más largos tienden a desaparecer, dado que las hormigas tardarán más tiempo en recorrerlos. Así, aunque inicialmente todas las hormigas escojan caminos aleatorios, llegará un punto en el que todas seguirán el mismo: el más corto. En la Figura 2.3 se puede entender de forma visual el funcionamiento de la estigmergia.

El algoritmo ACO es una metaheurística que utiliza una colonia de hormigas artificiales que, de forma iterativa, construyen nuevas soluciones (caminos) y cooperan entre sí a través de una matriz de feromonas para encontrar mejores soluciones en cada iteración del algoritmo.

Partiendo de un algoritmo ACO básico propuesto por Marco Dorigo en su tesis doctoral, en las últimas dos décadas se propusieron una multitud de variantes del ACO [14, 16]. Las más populares son:

- **Ant System (AS):** la actualización de feromonas tiene lugar después de que todas las

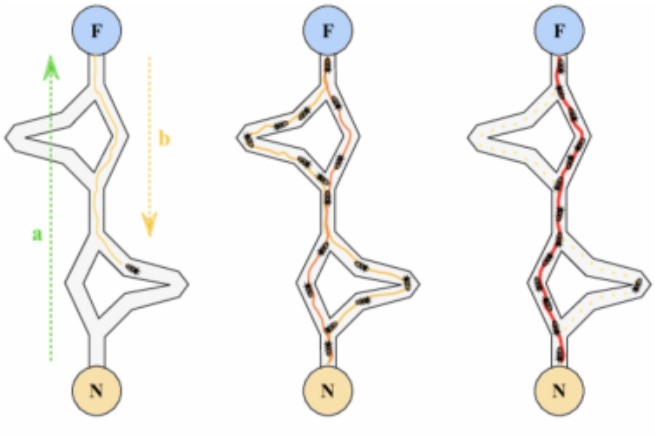


Figura 2.3: Representación de la estigmergia.

hormigas construyan sus recorridos. La cantidad de feromonas depositada por cada hormiga depende de la calidad del recorrido.

- **Ant Colony System (ACS):** en esta versión, cada vez que una hormiga recorre un camino, elimina una pequeña cantidad de feromonas de él para aumentar la exploración de caminos alternativos.
- **Elitist Ant System (EAS):** en este caso, la idea es reforzar los caminos que pertenecen al mejor recorrido encontrado desde el inicio del algoritmo. Esto se lleva a cabo teniendo en cuenta que las hormigas que obtienen mejores soluciones, es decir, las elitistas, son las que depositan una cantidad mayor de feromonas.
- **Rank-based Ant System (RBAS):** esta variante clasifica las distintas hormigas, de forma que cada una deposita una cantidad de feromonas que decrece de forma proporcional a su rango. De la misma forma que en la variante EAS, son las mejores hormigas las que depositan una mayor cantidad de feromonas.
- **MAX-MIN Ant System (MMAS):** únicamente las mejores hormigas de cada iteración son las que depositan feromonas en sus caminos. A mayores, el valor que puede tomar el rastro de feromonas está acotado dentro de un intervalo.

En este proyecto trabajaremos con la variante MMAS. El principal motivo es que, a parte de ser una de las variantes más populares del ACO debido a su eficiencia, también se trata de una metaheurística que garantiza la convergencia a un óptimo global.

2.3 Sistemas heterogéneos CPU-GPU

Desde hace años, la mayoría de aplicaciones software se escribían como programas secuenciales fácilmente comprendidos por un ser humano que recorra línea a línea el código. De forma histórica, la mayoría de desarrolladores software confiaron en los avances del hardware para aumentar la velocidad de sus aplicaciones secuenciales, así, el mismo programa se ejecutaba más rápido a medida que aparecían nuevas generaciones de procesadores. Sin embargo, esta expectativa no es válida a día de hoy, ya que un programa secuencial sólo se ejecutará en uno de los núcleos del procesador, que no tiene por qué ser más rápido que los que se usan hoy en día. Esto implica que las aplicaciones que realmente sacarán provecho de cada nueva generación de microprocesadores serán los programas paralelos, donde múltiples hilos de ejecución cooperan para trabajar más rápido [2].

En el año 2003, la industria de semiconductores se divide principalmente en dos trayectorias para el diseño de microprocesadores. Por una parte, la trayectoria multinúcleo busca mantener la velocidad de ejecución de los programas secuenciales mientras se avanza hacia múltiples núcleos. Los multinúcleos empezaron siendo procesadores de dos núcleos, de forma que el número de éstes aumentaba con cada nueva generación de semiconductores. Por el contrario, la trayectoria multihilo está centrada en el rendimiento de ejecución de las aplicaciones paralelas. Los multihilos empezaron con un gran número de hilos, y de nuevo, el número de estos aumenta con cada generación. En el año 2020, la relación de cálculo en punto flotante entre las GPUs multihilo y las CPUs multinúcleo es de aproximadamente 10. Esta relación no se refiere a velocidad de aplicación, sino a una velocidad bruta que los recursos en ejecución pueden soportar en estos sistemas: por ejemplo, la GPU Nvidia A100 soporta hasta 10 TFLOPS en simple precisión frente a la CPU AMD Ryzen 9 3950X, que soporta 1 TFLOPS [17].

Esta diferencia de rendimiento viene dada por las distintas filosofías de diseños entre los dos tipos de procesadores, como se puede apreciar en la Figura 2.4.

Por una parte, las CPUs están diseñadas para minimizar la latencia de ejecución de un único hilo. Las cachés de último nivel de los *chips* están diseñadas para capturar los datos a los que se accede con frecuencia, convirtiendo así los accesos a caché en accesos de poca latencia. Las unidades lógicas aritméticas (*Arithmetic Logic Unit* - ALU) y la lógica de transferencia de datos también están diseñados para minimizar la latencia efectiva de operación a coste de un uso mayor del área y potencia del *chip*. Reduciendo la latencia de las operaciones dentro del mismo hilo, el hardware de la CPU reduce la latencia de ejecución de cada hilo individual. Sin embargo, todas estas características consumen área de *chip* y energía que podrían utilizarse

para proporcionar más unidades lógicas aritméticas y canales de acceso a memoria.

Por otra parte, la filosofía de diseño de las GPUs viene determinada por el rápido crecimiento de la industria de los videojuegos, que ejerce una gran presión económica para realizar un número masivo de cálculos en punto flotante por fotograma en juegos avanzados. Esta demanda motiva a los fabricantes de GPUs a buscar formas de maximizar el área del *chip* dedicado a los cálculos en punto flotante de la misma operación sobre diferentes datos. La solución predominante consiste en optimizar el rendimiento de ejecución de múltiples hilos. La reducción de superficie y potencia del hardware de los canales dedicados a memoria permite a los diseñadores tener un mayor número de unidades aritméticas en un *chip*, aumentando así el rendimiento total de ejecución.

Con todo, está claro que las GPUs están diseñadas como motores de cálculo paralelo orientados al rendimiento y que no funcionarán bien en algunas tareas en las que las CPUs están diseñadas para funcionar bien. De cara a los programas que cuentan con uno o pocos subprocesos, las CPUs con latencias de operación más bajas pueden obtener un mayor rendimiento que las GPUs. Pero cuando un programa tiene un gran número de subprocesos, las GPUs ganarán en rendimiento a las CPUs. Entonces, podemos esperar que muchas aplicaciones empleen tanto la CPU como la GPU, ejecutando las partes secuenciales en CPU y aquellas con cálculos paralelos en GPU. Este es el motivo por el cual el modelo de programación CUDA, introducido por Nvidia en el año 2007, está diseñado para soportar la ejecución conjunta de CPU y GPU de una aplicación [2].

La arquitectura de una GPU está compuesta por un conjunto de *Streaming Multiprocessors*

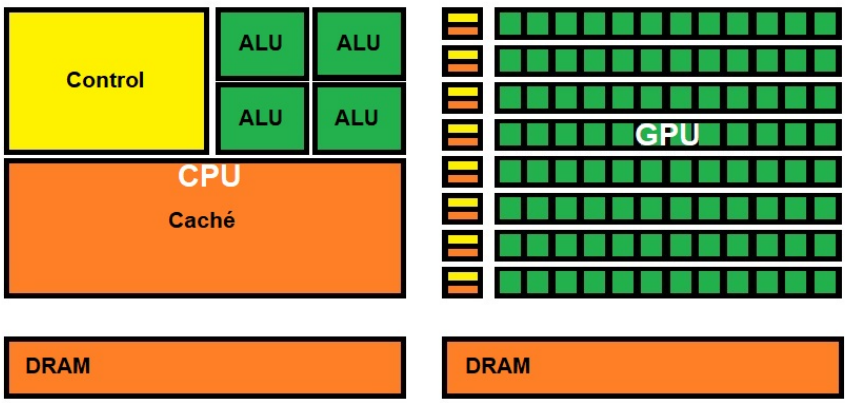


Figura 2.4: Diferencias de diseño entre CPU y GPU. Imagen basada en [2].

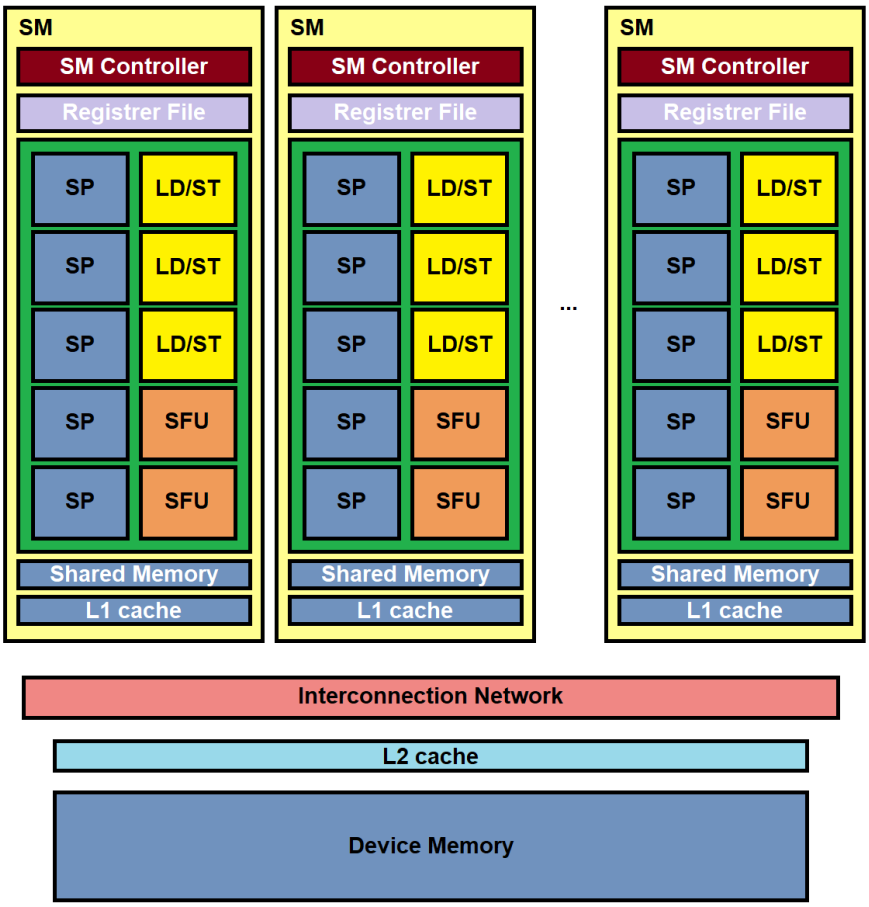


Figura 2.5: Arquitectura de una GPU de Nvidia. Imagen basada en [3].

(SMs). Cada uno de estos SM cuenta con varios núcleos CUDA de simple y doble precisión, también llamados *Streaming Processors* (SPs), una caché L1, memoria compartida, un fichero de registros, unidades de almacenamiento y carga, unidades de funciones especiales (*Special Function Units* - SFUs) y controladores de memoria. En la Figura 2.5 tenemos una vista en detalle.

2.3.1 Distintas arquitecturas

A lo largo de los años, Nvidia ha ido creando sucesivas generaciones de GPUs capaces de soportar CUDA con distintas arquitecturas. Aunque existe una gran variedad de arquitecturas, trataremos con las dos que más conciernen a este proyecto.

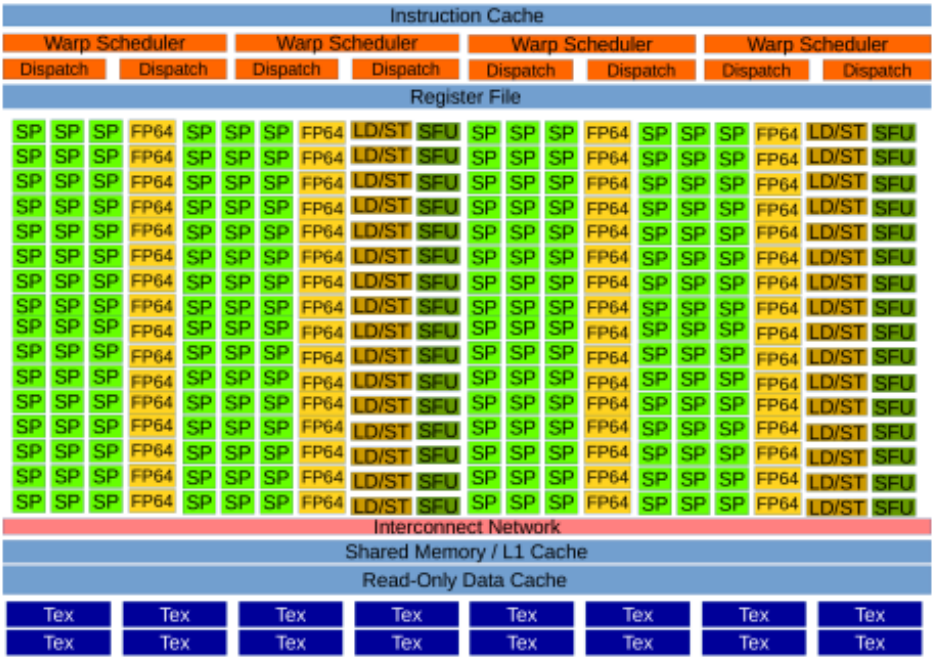


Figura 2.6: Estructura de un SM en la arquitectura Kepler [3].

Arquitectura Kepler

Esta arquitectura es la sucesora de la Fermi 2012. Kepler cuenta con hasta 15 SMs, cada uno de ellos con 192 núcleos CUDA de simple precisión, 64 unidades de doble precisión, 32 SFUs, 32 unidades de carga/almacenamento y 16 unidades de textura. También incluye 6 controladores de memoria de 64 bits. En la Figura 2.6 tenemos una vista detallada.

El número de registros por hilo puede ser de hasta 255, incluye instrucciones *shuffle*, unas instrucciones especiales que permiten a los hilos compartir información sin necesitar la memoria compartida si pertenecen al mismo *warp*, y soporte nativo para operaciones atómicas FP64 en memoria global. También incluye la funcionalidad *CUDA Dynamic Parallelism* que permite el lanzamiento de un *kernel* desde otro *kernel*. En cuanto a los tamaños de las memorias, la caché L2 aumenta a 1536KB. La memoria compartida/caché L1 es de 64KB y permite tres configuraciones: 32KB/32KB, 16KB/48KB y 48KB/16KB. Además, el ancho de banda de la memoria compartida es de 64 bits e introduce una caché de datos de sólo lectura de 48KB. En la Figura 2.7 se puede apreciar en detalle la jerarquía de memoria de esta arquitectura.

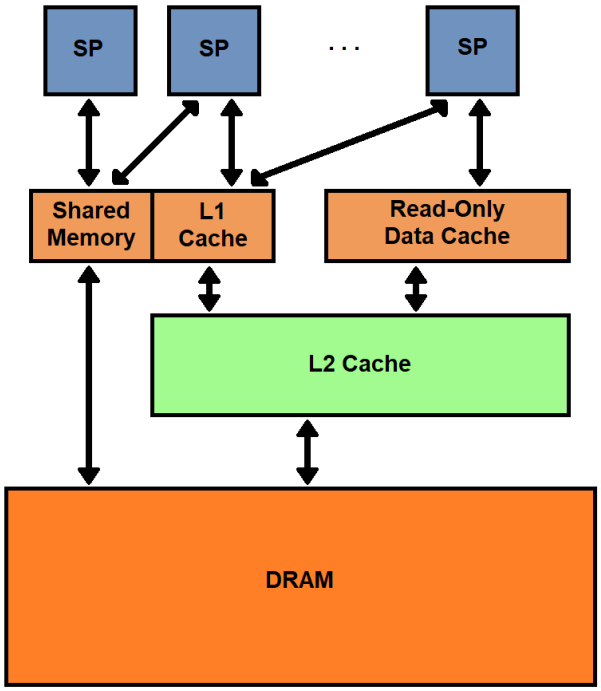


Figura 2.7: Jerarquía de memoria en arquitecturas Kepler. Imagen basada en [2].

Arquitectura Turing

Esta arquitectura se lanzó en 2018 como sucesora de la arquitectura Pascal y una de sus principales características son los *Tensor Cores*, unidades especializadas para realizar la operación tensor/matriz, que es uno de los núcleos de las funciones usadas en aprendizaje profundo. Soporta trazado de rayos en tiempo real (*Ray tracing* - RTX) con núcleos dedicados específicamente a él, lo cual es muy importante para aplicaciones computacionalmente pesadas como la realidad virtual.

La arquitectura cuenta con 72 SMs, cada uno de ellos con 64 núcleos CUDA, 8 *Tensor Cores*, un archivo de registros de 256KB, 4 unidades de texturas y 96KB de memoria compartida/caché L1. En total, la arquitectura cuenta con 4608 núcleos CUDA, 72 núcleos de trazado de rayos, 576 *Tensor Cores*, 288 unidades de textura y 12 controladores de memoria GDDR6 de 32 bits. En la Figura 2.8 tenemos una vista detallada.

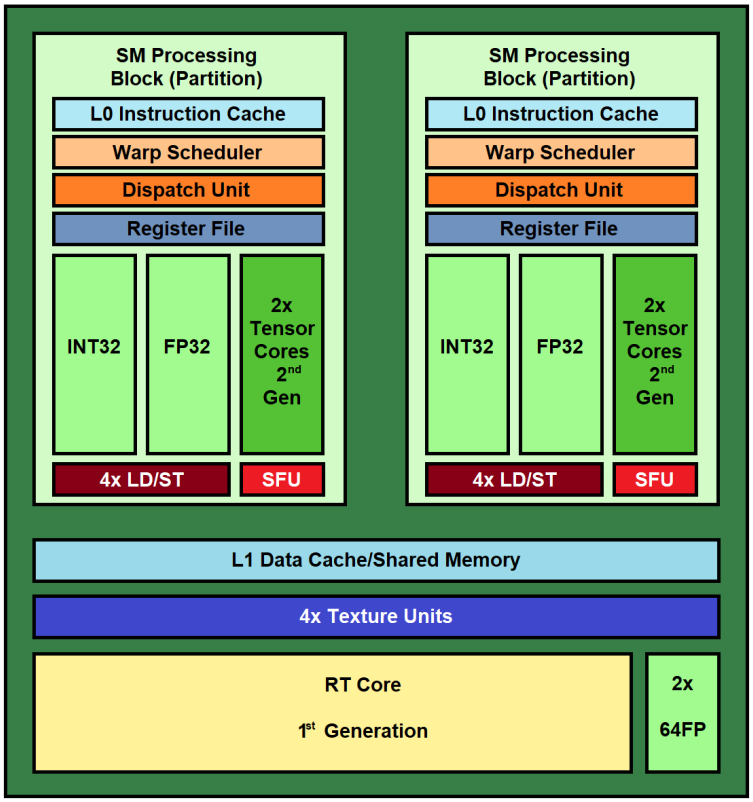


Figura 2.8: Arquitectura Turing.

2.4 Plataforma CUDA

CUDA (*Compute Unified Device Architecture*) [5] es una plataforma de computación paralela que incluye un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia que permiten programar empleando una variación del lenguaje de programación *C* para codificar algoritmos en GPU.

A continuación se explican una serie de conceptos necesarios para entender el funcionamiento de CUDA. Por una parte, el término *host* hace referencia a la CPU, mientras que *device* se refiere a la GPU. A mayores, empleamos el término *kernel* para referirnos a código que cuando es invocado se ejecuta N veces en paralelo por N hilos diferentes [18].

En CUDA los hilos se agrupan en bloques, y los hilos sólo pueden comunicarse con aquellos que se encuentren en el mismo bloque. A su vez, los bloques se agrupan en un *grid*. Un *kernel* se ejecuta en un *grid* de bloques de hilos, como vemos en la Figura 2.9. Es importante que los programadores organicen correctamente el *grid* y los bloques de hilos cuando se

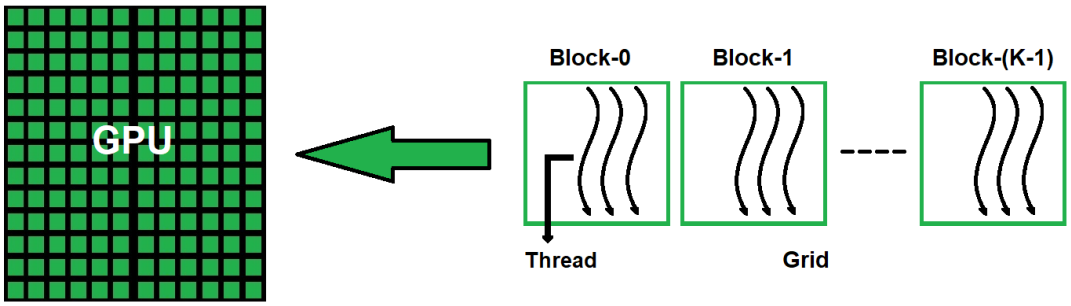


Figura 2.9: Organización de los hilos en CUDA. Imagen basada en [4]

hace una llamada a un *kernel*, ya que tiene un gran impacto en el rendimiento de la aplicación.

De forma interna, CUDA gestiona la ejecución de un programa organizando y procesando los hilos lógicos en núcleos físicos de CUDA [3]. Un flujo de proceso típico de un programa en CUDA cuenta con las siguientes fases:

- Reservar espacio en la memoria del *device*.
- Copiar los datos necesarios de la memoria del *host* a la del *device*.
- Realizar la llamada a los *kernels* para ejecutar las funciones en la GPU.
- Copiar los resultados desde la memoria del *device* de vuelta a la del *host*.
- Liberar el espacio de memoria do *device*.

Otro aspecto importante para obtener un buen rendimiento en CUDA es la jerarquía de memoria. A través de un subsistema de baja latencia y capacidad, se organizan varios niveles de memoria con distintas capacidades, latencias y anchos de banda. En la Figura 2.10 podemos apreciar visualmente este subsistema. Los distintos tipos de memoria se describen a continuación [4]:

- **Memoria global:** es la memoria más grande, pero también la de mayor latencia de la GPU. Puede ser accedida por cualquier hilo, incluso fuera de la ejecución de un *kernel*. Se encuentra en la DRAM del *device*. De forma análoga a la CPU, la memoria global es comparable a la memoria principal.
- **Memorias de sólo lectura:** cada SM cuenta con una memoria constante y una memoria de texturas. Se tratan de memorias de sólo lectura para los hilos de la GPU, y sólo la CPU puede escribir en ellas. Todas estas memorias sólo pueden ser leídas por el código del *kernel* y están destinadas a tareas específicas. Dependiendo de las características de

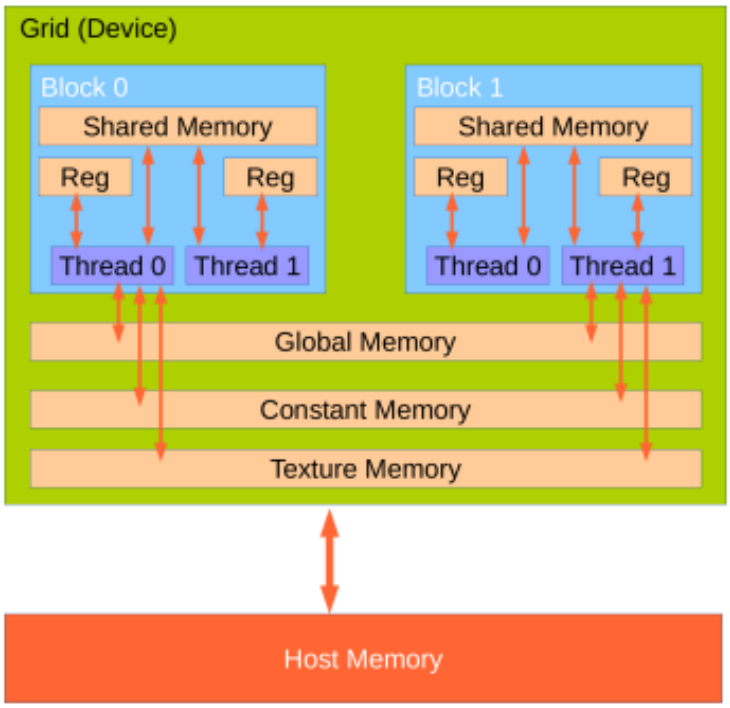


Figura 2.10: Subsistema de memoria en CUDA [3].

los accesos, acceder a los datos en esta memoria puede ser más rápido y eficiente que empleando la memoria global.

- **L1/Memoria compartida (*Shared Memory* - SMEM):** se trata de una memoria programable que cuenta con un mayor ancho de banda y una menor latencia que la memoria global. El tiempo de vida de la memoria compartida es el mismo que el del *kernel*. Funciona como una interconexión entre los hilos dentro de un bloque, así, sólo los hilos de un mismo bloque pueden acceder a ella. La principal diferencia entre L1 y memoria compartida es que la memoria compartida se controla a través de software, mientras que la caché L1 es controlada por hardware. En algunas arquitecturas estas memorias se encuentran en la misma unidad de memoria y el usuario tiene que configurar el sistema para indicar el tamaño de cada una.

- **Registros:** es la memoria más rápida que existe. Es privada para cada hilo y gestionada por el propio compilador. En general, una variable declarada en el *kernel* estará almacenada en un registro. Cuando finaliza la ejecución del *kernel*, ya no se puede acceder a una variable guardada en un registro. Además, las variables de un *kernel* que no pueden ser almacenadas en un registro por el compilador, se guardan en la memoria local, una parte de la memoria global que sólo puede ser accedida por el hilo correspondiente. Los

accesos a memoria local tienen una mayor latencia que los de los registros.

2.5 Implementaciones paralelas de ACO

ACO es un algoritmo en el cual cada hormiga construye su camino de forma independiente a las demás. Esta naturaleza independiente permite paralelizar al algoritmo tanto en los datos como en los dominios de población. Existen muchas estrategias paralelas diferentes que se pueden adaptar al ACO. La mayoría de ellas pueden clasificarse en estrategias de grano fino y de grano grueso.

En los enfoques de grano fino, las hormigas son asignadas a procesadores individuales que intercambian información con frecuencia. Las estrategias de grano fino intentan encontrar paralelismo en el algoritmo secuencial manteniendo su comportamiento asignando pocas y pequeñas tareas a procesos únicos que realicen un intercambio frecuente de información entre ellos. En la metaheurística ACO, encontrar el paralelismo en el algoritmo secuencial es sencillo, ya que la mayoría de las operaciones se encuentran en bucles fácilmente paralelizables. Sin embargo, el frecuente intercambio de información afecta negativamente a la escalabilidad de estos enfoques.

En las estrategias de grano grueso, subpoblaciones más grandes, o incluso poblaciones enteras, son asignadas a procesadores individuales y el intercambio de información es menor. Estos enfoques son más eficientes y se basan en encontrar una variante paralela del algoritmo secuencial. En muchas metaheurísticas esta variante consiste en implementar un modelo basado en islas. Así, la población inicial se distribuye en distintas islas en las que el algoritmo original se ejecuta aisladamente y, en ocasiones, las islas intercambian información que les permite beneficiar sus búsquedas de conocimiento. Esta solución reduce en gran medida las comunicaciones entre los procesos distribuídos, lo cual mejora la escalabilidad. En ACO, las islas se denominan colonias y cada una se asigna a un procesador. Cada colonia ejecuta el algoritmo secuencial de forma remota, hasta que se alcanza un paso de comunicación, en el que la información sobre las mejores soluciones encontradas hasta el momento y/o la matriz de feromonas son intercambiadas entre los procesadores. Este tipo de implementación paralela de ACO adopta el nombre de modelo multicolonia [6].

En este trabajo hemos seguido una implementación de grano fino asignando tareas pequeñas a cada hilo de ejecución, que representa a cada hormiga. Sin embargo, será muy interesante, como trabajo futuro, explorar la posibilidad de hacer una implmentación a grano grueso, explotando los recursos de la GPU.

Capítulo 3

Implementación

Este capítulo explica en profundidad el funcionamiento del algoritmo ACO apoyándonos en el pseudocódigo de la versión secuencial. Posteriormente, se describe la implementación paralela.

3.1 Implementación secuencial

El ACO es una metaheurística que consiste en que las hormigas de una colonia cooperen para encontrar buenas soluciones a problemas de optimización difíciles. Una hormiga artificial en ACO es un procedimiento estocástico que construye una solución de forma gradual añadiendo elementos a una solución bajo construcción. Esto permite al ACO ser aplicado a problemas de optimización combinatoria en los que se puede definir una heurística constructiva, como las redes de señalización celular.

Algorithm 1 Pseudocódigo de la metaheurística ACO

```
1: Inicializar parámetros iniciales y las feromonas
2: while condición de finalización no alcanzada do
3:     Construcción de soluciones
4:     Actualización de feromonas
5:     Acciones complementarias
6: end while
```

El algoritmo empieza inicializando las variables necesarias y las feromonas. Dentro del bucle, se ejecuta la construcción de soluciones, donde cada hormiga construye de forma incremental las soluciones al problema de optimización. Después tiene lugar la actualización de feromonas, basada tanto en la evaluación de nuevas soluciones como en la evaporación de feromonas. También cuenta con una serie de funciones adicionales, que se encargan de realizar acciones específicas del problema que no pueden ser llevadas a cabo por hormigas. Normal-

mente, se refieren a optimizaciones locales o globales para aproximar aún más la solución [6].

Este trabajo está enfocado hacia un ACO que emplea la variante MMAS (*Max-Min Ant System*), cuyo funcionamiento se describe siguiendo el algoritmo 2.

Algorithm 2 Pseudocódigo de la metaheurística ACO empleando la variante MMAS

```
Inicializar parámetros iniciales y las feromonas
while condición de finalización no alcanzada do
    Construcción de soluciones
    Selección de la mejor hormiga
    Actualización de feromonas
    Reinicio de la matriz de feromonas
end while
```

A mayores de las funciones que tienen lugar en el ACO básico, tras la construcción de soluciones tiene lugar la selección de la mejor hormiga y luego la actualización de feromonas, que precisamente la hará esa mejor hormiga determinada anteriormente. Por último, también existe una función de reinicio de feromonas en caso de que el algoritmo se quede atascado. A continuación, se explican en detalle las fases concretas del algoritmo.

3.1.1 Construcción de soluciones

El algoritmo ACO se aplica sobretodo en problemas relacionados con grafos. Las hormigas artificiales se mueven de un nodo del grafo a otro construyendo soluciones basándose en una función de probabilidad que depende de la cantidad de feromonas depositadas en cada arco y de algunas heurísticas que el usuario puede añadir.

Cuando se entrena una red de señalización celular, cada hormiga construye una solución que consiste en un vector binario que representa si atraviesa (1) o no (0) un camino. Cada nodo del grafo tiene sólo dos caminos, P_1 y P_0, que representan la inclusión o no del camino correspondiente. En comparación con el algoritmo básico de ACO, que utiliza un grafo completo, este enfoque ahorra memoria y tiempo de cálculo. Podemos apreciarlo en la Figura 3.1.

En cada iteración, cada hormiga escoge el nodo al que avanza en función de una regla de transición probabilística. Esta regla depende directamente de los valores de las feromonas:

$$p_{ij}^{k} = \frac{[t_{ij}]}{[t_{i0}] + [t_{i1}]} \tag{3.1}$$

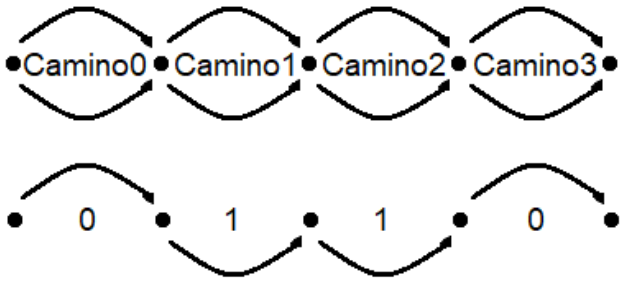


Figura 3.1: Representación de la construcción de soluciones de un ACO aplicado a un problema binario como el de la señalización celular.

- p_{ij}^k representa la probabilidad de que una hormiga *k* realice la transición de un nodo *i* a un nodo *j*.
- t_{ij} hace referencia a la inclinación de una hormiga de cruzar el camino desde *i* hasta *j*, dependiendo de la cantidad de feromonas.
- t_{i0} representa la decisión de no tomar el camino *i*.
- t_{i1} representa la decisión de tomar el camino *i*.

Una diferencia de esta implementación con el ACO tradicional es que dos de los parámetros usados en la regla de transición, α y β, no son necesarios. α es un parámetro positivo y real que controla la influencia de las feromonas en la decisión y β es un parámetro positivo y real que controla la influencia de la información heurística en la decisión.

Tras conocer el camino de cada hormiga, se emplea una función para computar la puntuación obtenida, que será el valor que queremos minimizar. Se puede apreciar este proceso en el algoritmo 3.

Algorithm 3 Pseudocódigo de la construcción de soluciones iniciales

```
for 1:N_h do
    Inicialización
    Construcción del camino
    Computar puntuación obtenida
end for
```

3.1.2 Actualización de la mejor solución

Como estamos trabajando con la variante MMAS, necesitamos guardar siempre la mejor solución obtenida hasta el momento. Así, distinguimos dos soluciones importantes: la mejor de cada iteración, y la mejor obtenida hasta el momento. En cada iteración, y una vez la hormiga termina su contribución a la construcción de soluciones, se ejecuta una función que recorre todas las soluciones de la iteración actual y encuentra la mejor de todas ellas, en este caso la correspondiente con una puntuación menor. Una vez localizada la mejor solución de esa iteración, se compara con la mejor obtenida hasta el momento, de forma que si es mejor, se actualiza el valor de la mejor solución hasta el momento con el de la mejor solución de esa iteración. En el algoritmo 4 se describe el proceso en detalle.

Algorithm 4 Pseudocódigo de la actualización de la mejor solución

```
1:  min := colonia(0).puntuacion
2:  for i = 1:N_h do
3:      if colonia(i).puntuacion < min then
4:          min := colonia(i).puntuacion
5:          mejor := i
6:      end if
7:  end for
8:  if colonia(mejor).puntuacion < mejorHormigaGlobal.puntuacion then
9:      mejorHormigaGlobal.puntuacion := colonia(mejor).puntuacion
10: end if
```

3.1.3 Actualización de las feromonas

Una vez construídas las soluciones por cada hormiga, tiene lugar la actualización de feromonas. Éstas pueden ver su valor aumentado cuando las hormigas depositan feromonas en los mejores caminos, o disminuído debido a la evaporación de feromonas para evitar una acumulación masiva de éstas y para olvidar malas decisiones tomadas a la hora de escoger caminos. Las hormigas tienen preferencia por los caminos que tengan más feromonas depositadas en ellos.

El proceso de evaporación comentado anteriormente ayuda a evitar una convergencia prematura o a que el algoritmo se quede atascado en un óptimo local disminuyendo las feromonas a un ritmo constante. La siguiente ecuación define el proceso:

$$t_{i,j} \leftarrow (1-p)t_{i,j} \tag{3.2}$$

donde p es la tasa de evaporación de feromonas y $t_{i,j}$ es el elemento (i,j) en la matriz de fero-

monas.

Después del proceso de evaporación, tiene lugar la actualización de feromonas. Las feromonas depositadas vienen dadas por la siguiente ecuación:

$$t_{i,j} \leftarrow t_{i,j} + \triangle(t_{i,j})^{mejor} \tag{3.3}$$

donde $\triangle(t_{i,j})^{mejor}$ es la cantidad de feromonas depositadas por la mejor hormiga de la iteración o por la mejor hormiga hasta el momento, que puede ser:

- $1/f(S^{mejor})$, si el camino pertenece a S^{mejor}
- 0, en cualquier otro caso

donde S^{mejor} es la puntuación de la solución encontrado por la mejor hormiga de la iteración o por la mejor hormiga hasta el momento. En el algoritmo 5 se describe el proceso de actualización de feromonas, incluyendo la función de control de límites de la matriz de feromonas necesaria para la variante MMAS.

Algorithm 5 Pseudocódigo de la actualización de feromonas

```
1: Evaporación de feromonas
2: Actualización de la matriz de feromonas
3: Control de los límites de la matriz de feromonas
```

3.1.4 Acciones complementarias

Las acciones complementarias se usan para implementar funciones centralizadas que no pueden ser realizadas por hormigas porque requieren información que no es local a ellas. Una acción complementaria consiste en recoger información global para decidir si depositar feromonas o no para guiar el proceso de búsqueda desde una perspectiva global. Los procedimientos de búsqueda local suelen ser acciones complementarias.

En la variante MMAS sólo la mejor hormiga de cada iteración, es decir, la que obtuvo la mejor puntuación en esa iteración, o la mejor hormiga hasta el momento pueden depositar feromonas en cada iteración, causando una acumulación mayor en los mejores caminos. Sin embargo, esta estrategia puede dar lugar a una convergencia demasiado rápida, ya que la acumulación excesiva de feromonas en buenos caminos puede dar lugar a que las hormigas escojan siempre esos caminos, que no tienen por qué ser el óptimo. Para intentar evitar que esto suceda, la variante MMAS limita el rango de valores de las feromonas a un intervalo con un máximo y un mínimo, brindando así una pequeña posibilidad para cada camino de

ser escogido. Además, como las feromonas se inicializan en primer lugar al valor máximo, el algoritmo experimenta una mayor diversificación ya que en las primeras iteraciones la diferencia entre todos los caminos será menor que si las feromonas se inicializasen a 0.

Es importante tener en cuenta qué hormiga deposita feromonas en esta variante, ya que dependiendo de cuál sea, la búsqueda funciona distinto. Si las actualizaciones las realiza siempre la mejor hormiga hasta el momento, la búsqueda estará dirigida hacia la mejor solución global, mientras que cuando es la mejor hormiga de la iteración, es menos dirigida. Las investigaciones en este tema apuntan que en problemas de menor tamaño puede ser mejor emplear sólo actualizaciones de feromonas de la mejor hormiga de la iteración, mientras que en los problemas grandes se obtiene un mayor rendimiento centrándose en la mejor solución global. Esto se puede controlar aumentando gradualmente la frecuencia con la que la mejor hormiga hasta el momento actualiza las feromonas. En este trabajo empleamos el parámetro *ugb* para decidir cual de las dos hormigas se encarga de la actualización. Este parámetro lo establece el usuario y sirve para indicar qué número de veces se usa la actualización por la mejor hormiga local, y qué número de veces la actualización por la mejor hormiga global. En el algoritmo 6 se puede apreciar esta toma de decisión.

Algorithm 6 Pseudocódigo de la elección de la hormiga que deposita feromonas

```
1: if iteracion % ugb then
2:     actualizarFeromonas(mejorHormigaIter)
3: else
4:     actualizarFeromonas(mejorHormigaGlobal)
5: end if
```

3.1.5 Reinicio de la matriz de feromonas

En la variante MMAS, las rutas de feromonas se reinician cuando el programa corre riesgo de entrar en una fase de estancamiento. Esta fase se calcula empleando un factor de ramificación que computa la cantidad de caminos explorados, o si no se encontró ninguna ruta mejor en un cierto número de iteraciones consecutivas. En este caso, como vemos en el algoritmo 7, controlamos el estancamiento cuando no se encuentra una solución mejor tras un número de iteraciones definido. Este mecanismo favorece la búsqueda de caminos alternativos, permitiendo al algoritmo no estancarse en un mínimo local.

3.2 Transformación de la versión secuencial

A continuación, se describen en detalle las adaptaciones realizadas en el código secuencial para facilitar una implementación eficiente sobre una GPU.

Algorithm 7 Pseudocódigo del reinicio de feromonas

```
1: if algoritmo no mejora tras X iteraciones then
2:     Reiniciar feromonas
3:     Guardar la iteración donde tiene lugar el reinicio
4: end if
```

En la versión secuencial empleamos una matriz para representar las feromonas, de forma que tenemos dos entradas para cada camino (0 o 1). Además, cada hormiga está representada como una estructura que cuenta, por una parte, con la puntuación obtenida para cada hormiga, y por otra con un *array* que almacena la solución para cada camino. De cara a la implementación sobre la GPU, para facilitar el acceso a los datos en GPU, se hicieron los siguientes cambios:

- Las feromonas pasan a estar representadas como dos *arrays* de tamaño número de caminos, cada uno de ellos representando la opción 0 o 1.
- Ya no existe la estructura para representar cada hormiga, sino que tenemos varios *arrays* para representar toda la información necesaria: tenemos el *array* de tamaño número de hormigas para almacenar las puntuaciones de cada una, y tenemos un *array* de tamaño número de hormigas multiplicado por el número de caminos para almacenar 0 o 1, dependiendo de si se escogió ese camino o no. Además, esta desestructuración permitirá realizar de forma más eficiente la asignación de los distintos espacios de memoria a cada una de las variables.
- En el algoritmo secuencial sólo era necesaria una única semilla, pero en la implementación paralela tenemos distintos hilos de ejecución y cada uno de ellos necesita su propia semilla. Así, para la primera versión en GPU se ha creado un *array* de semillas para poder tener una por cada hilo en GPU. Empleando el número de hilo en la definición inicial de la semilla, aseguramos que todas las hormigas son realmente diferentes.

3.3 Paralelización de ACO empleando CUDA

En esta sección se explican las diferentes implementaciones y optimizaciones del código usando CUDA.

Previamente a los detalles de implementación, se definen las siguientes variables que serán usadas durante la explicación:

- ***N_t***: número de hilos (*threads*) que posee cada bloque.

- ***N_h***: número de hormigas.
- ***n***: número de caminos del problema.
- ****ant_solution***: *array* de tamaño número de hormigas multiplicado por el número de caminos ($N\_h \times n$) que almacena 0 o 1, dependiendo de si toma ese camino o no.
- ****ant_score***: *array* de tamaño número de hormigas (*N_h*) que guarda la puntuación de cada una.
- ****best_so_far_ant_solution***: *array* con la solución al problema, que tiene tantos elementos como número de caminos.
- ***best_so_far_ant_score***: puntuación final obtenida por la mejor hormiga.

También, se explican los siguientes conceptos relativos a la programación en CUDA:

- ***__global__***: prefijo empleado para referirse a las funciones que constituyen los *kernels* y que son invocadas desde la CPU y ejecutadas por todos los hilos en paralelo.
- ***__device__***: prefijo empleado para referirse a funciones que se ejecutan en *kernels*, pero sólo pueden ser invocadas desde la propia GPU, o para referirse a variables almacenadas en la memoria global de la GPU.
- ***kernel«<num_bloques, num_hilos, num_BS»>***: invocación de un *kernel* desde la CPU. Se especifican el número de bloques de hilos y el número de hilos por bloque que lo ejecutarán. Como se usa memoria compartida, el tercer parámetro indica cuántos Bytes se van a utilizar.
- ***cudaMalloc y cudaFree***: funciones empleadas para almacenar y liberar memoria, respectivamente, en la GPU.
- ***cudaMemcpy***: función empleada para copiar datos entre GPU y CPU. Recibe como parámetros la variable destino, la variable origen, el tamaño de la variable y la dirección de copia (CPU a CPU, CPU a GPU, GPU a GPU, GPU a CPU), en ese orden. En este trabajo sólo copiamos de CPU a GPU, o viceversa.
- ***threadIdx.x***: variable predefinida en CUDA que contiene los índices del hilo dentro de su bloque de hilos.

3.3.1 Versión inicial

La primera versión en CUDA es una adaptación muy simple de la versión secuencial en C. Se trataba de buscar una versión inicial de CUDA que funcionase y con la que se obtuviesen

resultados correctos, pero no necesariamente la más eficiente posible.

```
1    while ( !termination_condition() ) {
2        if ( iteration == 1 ) {
3            init_ants_device();
4        } else {
5            construct_solutions_device();
6        }
7        update_statistics();
8        pheromone_trail_update();
9        iteration++;
10   }
11   score = best_so_far_ant_score;
```

Listado 3.1: Iteración principal de la primera versión de ACO en CUDA.

El algoritmo, como se ha comentado en su descripción, realiza varias iteraciones hasta que se alcanza la condición de terminación. En esta primera versión, que podemos apreciar en el Listado 3.1, están implementadas en la GPU con CUDA, únicamente las funciones *init_ants_device* (línea 3), donde se inicializan las hormigas asignándoles valores aleatorios a sus soluciones, y *construct_solutions_device* (línea 5), que es la función en la que cada hormiga realiza su trabajo y construye la solución. El código de esta función se muestra en el Listado 3.2 para su versión secuencial, y en el Listado 3.3 la versión con CUDA.

```
1  void construct_solutions() {
2      int k, j;
3
4      for ( k = 0 ; k < N_h ; k++ ) {
5          for ( j = 0 ; j < n ; j++ ) {
6              select_gate( &ant[k], j);
7          }
8          ant[k].score = obj_function(&ant[k]);
9      }
10 }
```

Listado 3.2: Función de construcción de soluciones en ACO secuencial.

Como se puede ver en el *kernel* (líneas 1-13 del Listado 3.3), éste es ejecutado de forma que $N\_h = N\_t$. En la versión secuencial, existe un bucle que recorre las hormigas (línea 4 del Listado 3.2) para que cada una construya la solución. En la primera versión en CUDA, este bucle es eliminado, ya que es sustituido por la invocación del *kernel* por cada hilo, siendo el id de cada hilo calculado en la línea 3 del Listado 3.3. Además, en esta versión se realizan las siguientes transferencias de datos en cada iteración:

- Se copian ambos *arrays* relativos a las feromonas de la memoria de la CPU a la GPU (líneas 16 y 17), ya que están inicializados en la CPU, pero son necesarios para la construcción de soluciones en GPU.
- Se copian los resultados calculados en la GPU a la CPU (líneas 23 y 24), ya que las funciones *update_statistics* (línea 8 del Listado 3.1) y *pheromone_trail_update* (línea 9 del Listado 3.1) se realizan todavía en la CPU y necesitan esta información actualizada.

```
1  __global__ void construct_solutions_dev(int* ant_solution_dev,
       float* ant_score_dev, int* bs_optimum_dev, int n, long int
       *seed_dev, float q_0, float *pheromone0_dev, float
       *pheromone1_dev) {
2
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j, pos;
5
6      for ( j = 0 ; j < n ; j++ ) {
7          pos = i*n + j;
8          select_gate_dev( ant_solution_dev, j, pos, i, seed_dev,
       q_0, pheromone0_dev, pheromone1_dev);
9      }
10     ant_score_dev[i] = obj_function_dev(ant_solution_dev,
       bs_optimum_dev, i, n);
11 }
12
13 void construct_solutions_device() {
14     cudaMemcpy(pheromone0_dev, pheromone0, n * sizeof(float),
       cudaMemcpyHostToDevice);
15     cudaMemcpy(pheromone1_dev, pheromone1, n * sizeof(float),
       cudaMemcpyHostToDevice);
16
17     construct_solutions_dev<<<1,N_h>>>(ant_solution_dev,
       ant_score_dev, bs_optimum_dev, n, seed_dev, q_0, pheromone0_dev,
       pheromone1_dev);
18     cudaError_t cudaerr = cudaDeviceSynchronize();
19
20     cudaMemcpy(ant_solution, ant_solution_dev, (n_ants*n) *
       sizeof(int), cudaMemcpyDeviceToHost);
21     cudaMemcpy(ant_score, ant_score_dev, n_ants * sizeof(float),
       cudaMemcpyDeviceToHost);
22 }
```

Listado 3.3: Función de construcción de soluciones en la primera versión en CUDA.

```
1  __global__ void init_ants_dev(int* ant_solution_dev, float*
       ant_score_dev, int* bs_optimum_dev, int n, long int seed) {
2
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j, rnd, pos;
5      seed = seed * (i+1) ;
6
7      for ( j = 0 ; j < n ; j++ ) {
8          rnd = (int) round( ran01_dev( &seed ) );
9          pos = i*n+j;
10         ant_solution_dev[pos] = rnd;
11     }
12     ant_score_dev[i] = obj_function_dev(ant_solution_dev,
       bs_optimum_dev, i, n);
13 }
14
15 void init_ants_device() {
16
17     cudaMemcpy(bs_optimum_dev, bs_optimum, n * sizeof(int),
       cudaMemcpyHostToDevice);
18
19     seed = (long int) time(NULL)*(ntry+1);
20
21     init_ants_dev<<<1,32>>>(ant_solution_dev, ant_score_dev,
       bs_optimum_dev, n, seed);
22     cudaError_t cudaerr = cudaDeviceSynchronize();
23
24     cudaMemcpy(ant_solution, ant_solution_dev, (n_ants*n) *
       sizeof(int), cudaMemcpyDeviceToHost);
25     cudaMemcpy(ant_score, ant_score_dev, n_ants * sizeof(float),
       cudaMemcpyDeviceToHost);
26 }
```

Listado 3.4: Función de inicialización de hormigas en la primera versión en CUDA.

Además, en el *kernel* (líneas 1-14 del Listado 3.4) que se invoca en la primera iteración, también se transfieren los siguientes datos:

- Se copia el *array* que contiene la solución óptima de la CPU a la GPU, ya que se emplea para calcular la puntuación inicial de cada hormiga (línea 13).
- Se copian de vuelta a la CPU los *arrays* relativos a las soluciones y puntuación de cada hormiga, por el mismo motivo explicado para el caso del Listado 3.3.

Es importante tener en cuenta que estas transferencias de datos entre CPU y GPU son muy costosas en tiempo computacional. Además, todos los datos residen en la memoria global de

la GPU.

En esta primera versión, finalmente, se ha eliminado la función de *init_ants_device*, ya que realmente no estaba teniendo ningún impacto. Al estar inicializadas todas las feromonas al mismo valor, no hay ningún camino que tenga mayor prioridad con respecto a los demás en la primera iteración, por lo que se usa la misma función de construcción para todas las iteraciones.

3.3.2 Versión optimizada

Esta segunda versión ya ha sido implementada haciendo uso del potencial de la GPU y una utilización más eficiente de su jerarquía de memoria. Parte del nuevo *kernel* se puede ver en el Listado 3.5.

Esta versión sigue usando $N\_h = N\_t$, pero hace uso de la memoria compartida de la GPU, de la sincronización de hilos, de los registros de la GPU y de variables globales de la memoria global de forma eficiente. Además, todas las iteraciones se realizan en la GPU, minimizando ampliamente la comunicación entre la CPU y GPU ahorrando, en consecuencia, tiempo computacional. Se han analizado cada una de las variables para emplear el tipo de memoria más adecuado para cada una.

La nueva versión del *kernel* crea en memoria compartida un *array* para almacenar un dado número de variables. Así ahorramos tiempo computacional, ya que la memoria compartida es casi tan rápida como los registros y además permite la comunicación de información entre hilos. En concreto, las variables *pheromone0_dev*, *pheromone1_dev* y *ant_score_dev* hacen uso de la utilización dinámica de la memoria compartida (líneas 1-3).

```
1
2     extern __shared__ float pheromone0_dev[];
3   float *pheromone1_dev = &pheromone0_dev[n];
4   float *ant_score_dev = &pheromone1_dev[n];
5
6     if ( i == 0 ) {
7         init_aco_device(n, N_h, pheromone0_dev, pheromone1_dev,
    &trail_max_dev, &trail_min_dev, &trail_0_dev,
    &best_iteration_dev, &restart_best_dev, &n_restarts_dev);
8     }
9
10    __syncthreads();
11
12    while ( !( (iteration_dev >= max_iters_dev) ||
    (best_so_far_ant_score_dev <= optimal_dev) ) ) {
13
14        construct_solutions_device( i, bs_optimum_device,
    seed_device, n, pheromone0_dev, pheromone1_dev, ant_score_dev,
    ant_solution_dev );
15        __syncthreads();
16
17        if (i == 0) {
18            update_statistics_device(
    best_so_far_ant_solution_device, n, N_h, iteration_dev,
    pheromone0_dev, pheromone1_dev, ant_score_dev, ant_solution_dev,
    &trail_max_dev, &trail_min_dev, &trail_0_dev,
    &best_iteration_dev, &restart_best_dev, &n_restarts_dev );
19            pheromone_trail_update_device(
    best_so_far_ant_solution_device, iteration_dev, N_h, n,
    u_gb_dev, pheromone0_dev, pheromone1_dev, ant_score_dev,
    ant_solution_dev, trail_max_dev, trail_min_dev, restart_best_dev
    );
20        }
21
22        iteration_dev++;
23        __syncthreads();
24
25    }
26
27    if ( i == 0 ) {
28        iteration_device[0]=iteration_dev;
29        best_so_far_ant_score_device[0]=best_so_far_ant_score_dev;
30    }
```

Listado 3.5: Parte del kernel la segunda versión en CUDA.

Estas variables son accedidas por todos los hilos del bloque, pero sólo las modifica el hilo 0 (líneas 6 y 16). Por tanto, después de las modificaciones es necesario emplear una barrera de sincronización (líneas 10, 15 y 23) para asegurarse de que cuando los hilos lean las variables en la siguiente iteración, estén leyendo el dato correcto. Es importante tener en cuenta que estas barreras hacen que el tiempo computacional sea mayor, ya que detienen la ejecución hasta que todos los hilos terminan su trabajo.

El número máximo de tamaño de memoria compartida que necesitaremos es $(2 \times n + N_h) \times sizeof(float)$, que son 36,5KB. Este valor es menor que 48KB, que es el tamaño que en general permiten las GPU actuales para usar memoria compartida por bloque. Para aumentar el rendimiento se ha configurado la memoria L1/*Shared Memory* para que tenga el mayor valor la *Shared Memory* y el menor tamaño la L1:

cudaFuncSetCacheConfig(aco_device, cudaFuncCachePreferShared);

Por otro lado, un conjunto de variables tales como *iteration_dev*, *u_gb_dev* o *max_iters_dev* han sido almacenadas en los registros de la GPU. El motivo es que son variables con valores constantes y de las que sólo nos interesa el valor del hilo 0 o sólo las modifica cada uno de los hilos. El registro es un tipo de memoria rápida y privada para cada hilo, que normalmente se emplea para almacenar datos usados con frecuencia, por lo que es el tipo de memoria ideal para estas variables.

```
1    cudaMalloc((void**)&seed_device, (n_ants) * sizeof(long int));
2    cudaMalloc((void**)&bs_optimum_device, n * sizeof(int));
3    cudaMalloc((void**)&best_so_far_ant_solution_device, n *
     sizeof(int));
4    cudaMalloc((void**)&iteration_device, sizeof(int));
5    cudaMalloc((void**)&best_so_far_ant_score_device,
     sizeof(float));
6    cudaMalloc((void**)&ant_solution, n*n_ants*sizeof(int));
7
8    cudaMemcpy(bs_optimum_device, bs_optimum, n * sizeof(int),
     cudaMemcpyHostToDevice);
9    cudaMemcpy(seed_device, seed, n_ants * sizeof(long int),
     cudaMemcpyHostToDevice);
```

Listado 3.6: Parte de la inicialización en la segunda versión en CUDA donde se reserva memoria para las variables en GPU y se copian las variables necesarias.

Además, en el Listado 3.6, se puede apreciar cómo se transfieren los datos necesarios de CPU a GPU (líneas 8 y 9):

- Se copia el *bs_optimum* que contiene la solución óptima de la CPU a la GPU, necesaria para que cada hormiga construya su solución.
- Se copia el *array* de semillas (*seed*) necesarias para cada hilo.

```
1    cudaMemcpy(&best_so_far_ant_score,
     best_so_far_ant_score_device, sizeof(float),
     cudaMemcpyDeviceToHost);
2    cudaMemcpy(&iteration, iteration_device, sizeof(int),
     cudaMemcpyDeviceToHost);
3
4    exit_aco();
```

Listado 3.7: Parte del final del algoritmo en la segunda versión en CUDA donde se devuelven las variables necesarias a CPU para realizar los informes.

En el Listado 3.7 se pueden apreciar las variables que devolvemos a CPU. En este caso, como todo el algoritmo tiene lugar en GPU, sólo devolvemos dos variables necesarias para escribir el informe que nos permitirá analizar los resultados. Este informe se escribe en CPU para no consumir tiempo de computación en la GPU.

Por último, la variable *ant_solution* (línea 6 del Listado 3.6) se almacena en la memoria global. No se utiliza la memoria compartida ya que supera su capacidad. El tamaño de esta variable es $n \times N_h \times sizeof(int)$, siendo éste un valor comprendido entre 72KB para el caso de 32 hormigas y 579 caminos, que es el problema más pequeño, y 8310KB para el caso de 1024 hormigas y 4155 caminos, que es el problema más grande con el que trabajaremos. Por tanto, se tomó la decisión de reservar desde la CPU el tamaño de memoria global necesario para esta variable.

Capítulo 4

Resultados experimentales

Este capítulo expone los resultados obtenidos en cada versión implementada, contrastándolos contra los del algoritmo secuencial original.

4.1 Entorno de pruebas

En esta sección se describirán en detalle los tipos de problema empleados para las pruebas y la infraestructura en la que se llevaron a cabo.

4.1.1 Problemas empleados en las pruebas y parametrización del ACO

El objetivo de este trabajo consistía en estudiar una proyección del algoritmo ACO aplicado a un problema binario en GPU, siguiendo una implementación de grano fino y evaluando el rendimiento obtenido para un problema de señalización celular en el tratamiento de enfermedades. El algoritmo que se usó de base está adaptado a este tipo de problemas. Para la evaluación, sin embargo, no fue posible usar los benchmarks reales porque la función de coste de estos problemas, cedida por el grupo del profesor Saez-Rodríguez en el Instituto de Biomedicina Computacional de la Universidad de Heidelberg [1], estaba escrita en *R* y encontramos que estaba fuera del alcance del proyecto traducirla a *C* para su ejecución en la GPU, o gestionar código *R* en la GPU. En su lugar decidimos usar una función de coste que compara los caminos de cada iteración con el camino final de un problema real. Esa función de coste, lógicamente, tiene un coste computacional muchísimo más bajo que la función de coste real, que es un simulador. Sin embargo, el objetivo de este trabajo es probar la paralelización del algoritmo de optimización y para las pruebas estructurales y la depuración no hay diferencia con usar la función de coste simple.

[1] Saez-Rodriguez Group. Institute for Computational Biomedicine. Heidelberg University. https://saezlab.org

Nombre del problema	n
579n	579
1116n	1116
2154n	2154
4155n	4155

Tabla 4.1: Problemas empleados en las pruebas.

Para realizar esta evaluación, hemos empleado los problemas descritos en la Tabla 4.1. Se han escogido estos ya que son varios problemas con tamaños que van desde pequeños a medio-grandes, de forma que podemos realizar una evaluación más exhaustiva y conocer cómo afecta la solución cuando aumentamos el tamaño del problema.

Para entender la eficiencia que se puede conseguir al ejecutar nuestro problema en GPU, es importante tener en cuenta no solo el tamaño del problema, sino también su naturaleza. En este caso tratamos con un algoritmo estocástico. Esto significa que tiene cierta base aleatoria, por lo que para analizar exhaustivamente los resultados obtenidos, es necesario realizar una serie de ejecuciones, no una única. Esto hace que el tiempo de ejecución también se vea incrementado. Una vez recogidos los resultados, se realiza un análisis estadístico para sacar conclusiones.

Durante la recogida de datos, es importante tener en cuenta ciertos parámetros de control establecidos por el usuario:

- ***max_tries***: número de ejecuciones que realizamos por cada experimento.
- ***max_iters***: número máximo de iteraciones por ejecución. Forma parte de la condición de fin: si el algoritmo no encuentra una solución óptima antes del número máximo de iteraciones, se detendrá.
- ***max_time***: tiempo máximo permitido para la ejecución actual. Nos permite controlar la convergencia del algoritmo, haciendo que éste se detenga si no encuentra una solución antes de este tiempo.

Nosotros ejecutamos el algoritmo hasta que alcanza la convergencia, por lo que marcamos un máximo de tiempo y número de iteraciones lo suficientemente alto para que nunca se alcancen. Por otro lado, realizamos 30 ejecuciones por experimento. Al final del programa se

	Plataforma 1
CPU Model	2 × Intel Xeon E5-2660 Sandy Bridge-EP
CPU Speed/Turbo	2.20 GHz/3.0 GHz
#Cores por CPU	8
#Threads por core	2
#Cores/Threads por nodo	16/32
Cache L1/L2/L3	32 KiB/256 KiB/20 MiB
Memoria RAM	64 GiB DDR3 1600 Mhz (8 × 8 GiB)
Aceleradoras	NVIDIA Tesla Kepler K20m 5 GiB GDDR5 (0-0-12)

Tabla 4.2: Descripción de la Plataforma 1.

recoge un informe que incluye, para cada ejecución, el mejor valor obtenido, las iteraciones totales y el tiempo total de esa ejecución. Con todos los resultados se calcula la media y la desviación típica del experimento que informamos en las tablas de este capítulo.

Además, de cara a la realización de pruebas hay otros parámetros de configuración que debemos tener en cuenta. Como en este proyecto el interés está realmente en el estudio de la proyección en GPU, estos parámetros se dejan con valores constantes para todos los experimentos, de forma que se puede comparar el comportamiento de las versiones paralelas frente a la secuencial. A continuación, se definen estos parámetros:

- ***rho***: hace referencia al factor de evaporación de las feromonas. Tiene un valor de 0.5.
- ***q0***: representa la probabilidad de construir la mejor solución posible y está influenciada por los rastros de feromonas y por la información heurística. Tiene el valor de 0.

4.1.2 Infraestructura usada para las pruebas

Para realizar las pruebas se utilizó el clúster *Pluton* del Grupo de Arquitectura de Computadores de la UDC [9], que se describe a continuación. *Pluton* tiene un punto de entrada único accesible desde el exterior (nodo de *login* o nodo *frontend*). Los usuarios se conectan al clúster a través de este nodo para editar o compilar sus códigos y enviar trabajos al planificador o al sistema de colas para su posterior ejecución en los nodos de cómputo.

	Plataforma 2
CPU Model	2 × Intel Xeon Silver 4216 Cascade Lake-SP
CPU Speed/Turbo	2.1 GHz/3.2 GHz
#Cores por CPU	16
#Threads por core	2
#Cores/Threads por nodo	32/64
Cache L1/L2/L3	32 KiB/1 MiB/22 MiB
Memoria RAM	256 GiB DDR4 2933 Mhz (8 × 32 GiB)
Aceleradoras	NVIDIA Tesla T4 16 GiB GDDR6 (2-0)

Tabla 4.3: Descripción de la Plataforma 2.

Los nodos de cómputo empleados en las pruebas han sido la Plataforma 1, cuyas especificaciones se definen en la Tabla 4.2, por tener tarjetas gráficas con arquitectura Kepler; y la Plataforma 2, que podemos ver en detalle en la Tabla 4.3, por tener tarjetas gráficas con arquitectura Turing.

La primera plataforma se usó para las pruebas iniciales y durante la depuración de los desarrollos, porque al ser una arquitectura más antigua es menos demandada por los usuarios del clúster y se podían ejecutar las pruebas más rápidamente en las colas y sin molestias para otros investigadores. Sin embargo, la versión final se ejecutó en la plataforma 2 por ser la más moderna, y estos son los resultados que se muestran en esta memoria final. Con todo, los resultados obtenidos en la plataforma 1 están disponibles en el repositorio del código para que puedan ser consultados por cualquier persona interesada.

4.2 Resultados experimentales

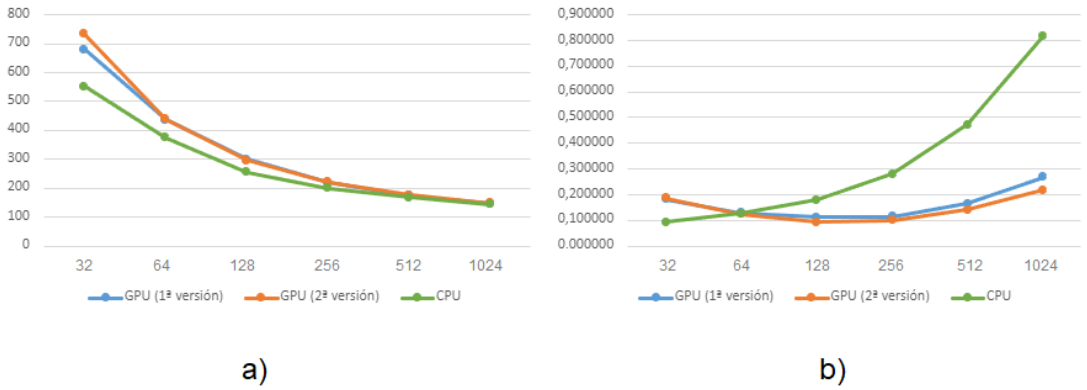
En esta sección se comentan los resultados obtenidos para ambas versiones de CUDA empleando la arquitectura Turing, y se comparan con el algoritmo original. En las tablas 4.4, 4.5, 4.6 y 4.7 se muestran los resultados obtenidos para ambas versiones en CUDA junto con los de la versión secuencial para los problemas 579, 1116, 2154 y 4155, respectivamente, y variando el número de hormigas que se utilizan. El número de hormigas afecta al trabajo que se realiza en paralelo en la GPU en cada iteración. Por lo tanto, aumentar el número de hormigas beneficia a las versiones de la GPU, mientras que perjudica a la versión de la CPU. En ambos casos

siempre hay que buscar un equilibrio entre la carga computacional de cada iteración (número de hormigas) y el número de iteraciones que se necesitará para converger (generalmente a mayor número de hormigas, mejor convergencia).

En general, podemos observar que al aumentar el tamaño del problema el tiempo necesario para construir las soluciones aumenta. Al aumentar el número de hormigas se hace más trabajo en paralelo en la GPU y más trabajo en secuencial en la CPU dentro de una única iteración, por lo que la eficiencia de la GPU aumenta y también la diferencia entre la versión secuencial y la paralela. Además, al aumentar el número de hormigas, la convergencia, medida en número de iteraciones necesarias para alcanzar el óptimo, mejora. Para visualizar mejor los resultados y la necesidad de llegar a un compromiso entre el número de hormigas y la velocidad de convergencia, en las figuras 4.1, 4.2, 4.3 y 4.4 se muestra gráficamente el número de iteraciones necesarias para la convergencia y el tiempo necesario al aumentar el número de hormigas, para todas las versiones en todos los problemas.

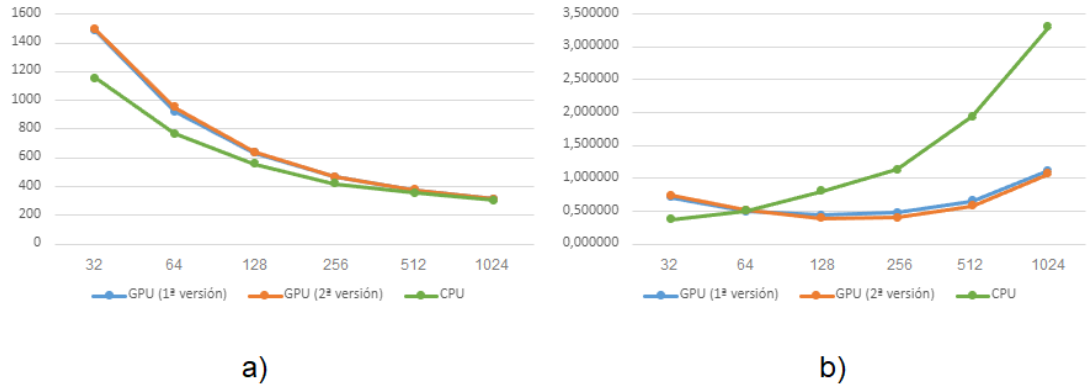
Tras analizar los resultados obtenidos, podemos sacar una serie de conclusiones generales. Por una parte, la versión 1 nunca es mejor que la versión 2 como se esperaba, debido al gran número de transferencias y a tener todas las variables en la memoria global. Además, hay que valorar un compromiso entre el número de hormigas y el tiempo computacional requerido. En estos problemas, para el caso secuencial es mejor usar un número pequeño de hormigas, pero para el caso de la GPU, los mejores resultados se obtienen para 128 y 256 hilos.

Los resultados anteriores fueron tomados con la función de coste simple que no representa apenas carga computacional. Para estudiar el comportamiento de la paralelización del algoritmo desde un punto de vista estructural, esa función es válida, pero si lo que queremos es valorar el rendimiento de la paralelización cuando usemos el algoritmo de optimización en un caso real necesitamos utilizar una función de coste con una carga computacional más realista. En este caso, para mantener el mismo comportamiento en cuanto a convergencia de los problemas anteriores, hemos modificado la función de coste para que realice más operaciones y simule más carga computacional manteniendo el mismo resultado que la función simple que comparaba las soluciones con las soluciones reales de los problemas biológicos. Hemos utilizado tres versiones, una donde la carga era baja, otra con carga media y otra con carga alta. El objetivo es mostrar la eficiencia de la proyección sobre la GPU cuando los problemas a los que nos enfrentemos sean grandes y complejos. En las figuras 4.5 y 4.6 se muestran los resultados de tiempo de ejecución medios en experimentos similares a los anteriores con las nuevas funciones de coste. Se puede ver como la versión de la GPU siempre supera en eficiencia a la versión de la CPU. Además, se aprecia cómo esta ganancia es mayor cuanto mayor es



a) b)

Figura 4.1: a) Media de iteraciones para cada versión con los distintos números de hormigas en el problema 579. b) Media de tiempo para cada versión con los distintos números de hormigas en el problema 579.



a) b)

Figura 4.2: a) Media de iteraciones para cada versión con los distintos números de hormigas en el problema 1116. b) Media de tiempo para cada versión con los distintos números de hormigas en el problema 1116.

la carga que introduce la función de coste, y cuanto mayor es el problema a resolver. La Tabla 4.8 muestra la aceleración conseguida en estos experimentos, siguiendo la fórmula:

$$sp = T_{avg\ sequential}/T_{avg\ parallel} \tag{4.1}$$

En un caso real, como los problemas biológicos cedidos por el grupo del profesor Saez-Rodriguez, la evaluación de cada función de coste puede tardar entre 5 y 10 minutos [6], lo que resultaría en resultados más impactantes. Lamentablemente, como ya se ha comentado, no se han podido recodificar estas funciones de coste para su uso en la GPU.

	CPU		GPU 1ª versión		GPU 2ª versión	
Nh	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**
32	552	0,093277 ±0,009744	681	0,181787 ±0,035012	733	0,185601 ±0,035489
64	376	0,127558 ±0,012399	437	0,128293 ±0,034741	440	0,123805 ±0,022452
128	256	0,180056 ±0,009822	300	0,111921 ±0,022659	297	0,093606 ±0,017526
256	201	0,280714 ±0,01088	220	0,114678 ±0,020943	221	0,099356 ±0,018728
512	168	0,472082 ±0,022117	174	0,163764 ±0,03081	176	0,140697 ±0,011204
1024	142	0,817141 ±0,023836	147	0,268026 ±0,027368	148	0,218141 ±0,011698

Tabla 4.4: Resultados problema 579 para cada número de hormigas (Nh).

	CPU		GPU 1ª versión		GPU 2ª versión	
Nh	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**
32	1154	0,371327 ±0,026515	1482	0,709941 ±0,107492	1494	0,735823 ±0,11721
64	766	0,500770 ±0,041393	922	0,491454 ±0,074653	950	0,516123 ±0,084879
128	558	0,808359 ±0,051568	632	0,440450 ±0,052451	638	0,387014 ±0,02584
256	420	1,137542 ±0,0393211	466	0,477123 ±0,044046	467	0,401355 ±0,032808
512	356	1,939113 ±0,051893	373	0,649679 ±0,042634	377	0,57609 ±0,033074
1024	302	3,300517 ±0,091807	312	1,112307 ±0,042353	310	1,065188 ±0,024449

Tabla 4.5: Resultados problema 1116 para cada número de hormigas (Nh).

	CPU		GPU 1ª versión		GPU 2ª versión	
Nh	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**
32	2497	1,522384 ±0,147326	3126	2,790936 ±0,387696	3211	3,018979 ±0,468926
64	1606	2,010970 ±0,1662	2019	2,056467 ±0,205862	1978	2,035179 ±0,235546
128	1169	2,996292 ±0,168587	1314	1,725829 ±0,160566	1320	1,497212 ±0,155717
256	894	4,680954 ±0,200293	976	1,757534 ±0,094296	982	1,605993 ±0,115162
512	729	7,744583 ±0,151052	775	2,554136 ±0,07512	774	2,80621 ±0,092221
1024	419	10,008290 ±0,005636	651	4,480775 ±0,114758	658	4,499286 ±0,133712

Tabla 4.6: Resultados problema 2154 para cada número de hormigas (Nh).

	CPU		GPU 1ª versión		GPU 2ª versión	
Nh	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**	**Iters**	**Tiempo(s)**
32	4960	5,832763 ±0,482824	6635	11,398932 ±1,195087	6854	12,466164 ±1,403509
64	3352	8,057153 ±0,586116	4219	8,202767 ±0,670088	4200	8,375860 ±1,277574
128	2376	11,759037 ±0,752455	2763	6,725598 ±0,527405	2792	6,149265 ±0,337549
256	1801	18,322033 ±0,369613	2033	6,962670 ±0,39215	1990	7,023181 ±0,254703
512	1486	30,790632 ±0,556982	1587	10,422734 ±0,362755	1583	11,348921 ±0,402721
1024	1284	53,419034 ±0,633181	1341	17,862157 ±0,298149	1333	18,382698 ±0,595398

Tabla 4.7: Resultados problema 4155 para cada número de hormigas (Nh).

Problema	Mejor tiempo CPU	Mejor tiempo GPU	Aceleración
579-Fcarga-baja	0,180201	0,093194	1,93
579-Fcarga-media	0,367566	0,095225	3,86
579-Fcarga-alta	0,537281	0,091889	5,85
4155-Fcarga-baja	6,659820	6,028639	1,10
4155-Fcarga-media	8,337070	6,157604	1,35
4155-Fcarga-alta	9,961956	6,141797	1,62

Tabla 4.8: Aceleración obtenida en GPU empleando la nueva función de carga.

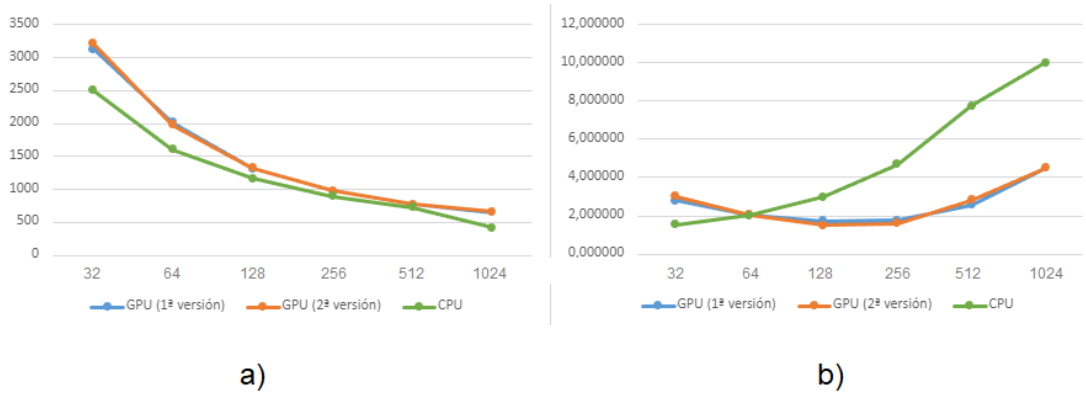


Figura 4.3: a) Media de iteraciones para cada versión con los distintos números de hormigas en el problema 2154. b) Media de tiempo para cada versión con los distintos números de hormigas en el problema 2154.

Figura 4.4: a) Media de iteraciones para cada versión con los distintos números de hormigas en el problema 4155. b) Media de tiempo para cada versión con los distintos números de hormigas en el problema 4155.

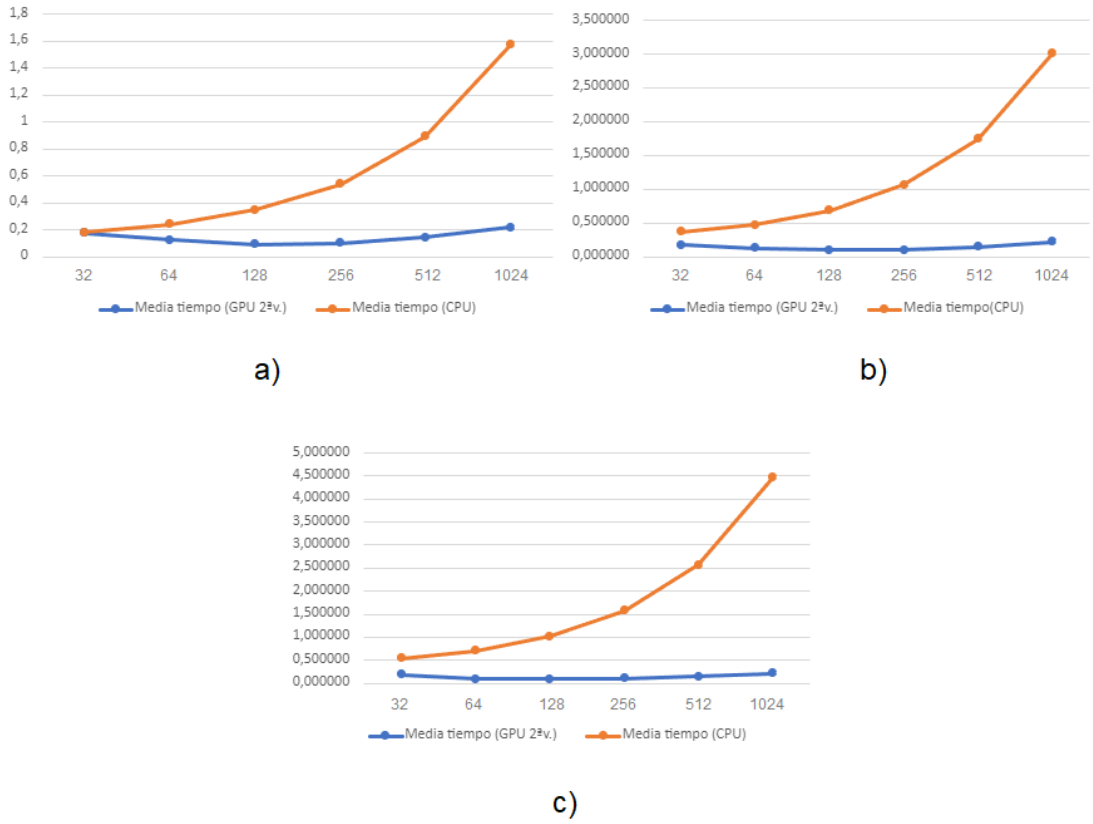


Figura 4.5: Media de tiempo de ejecución (GPU vs CPU) del problema 579 con una función de coste con a) carga baja, b) carga media y c) carga alta.

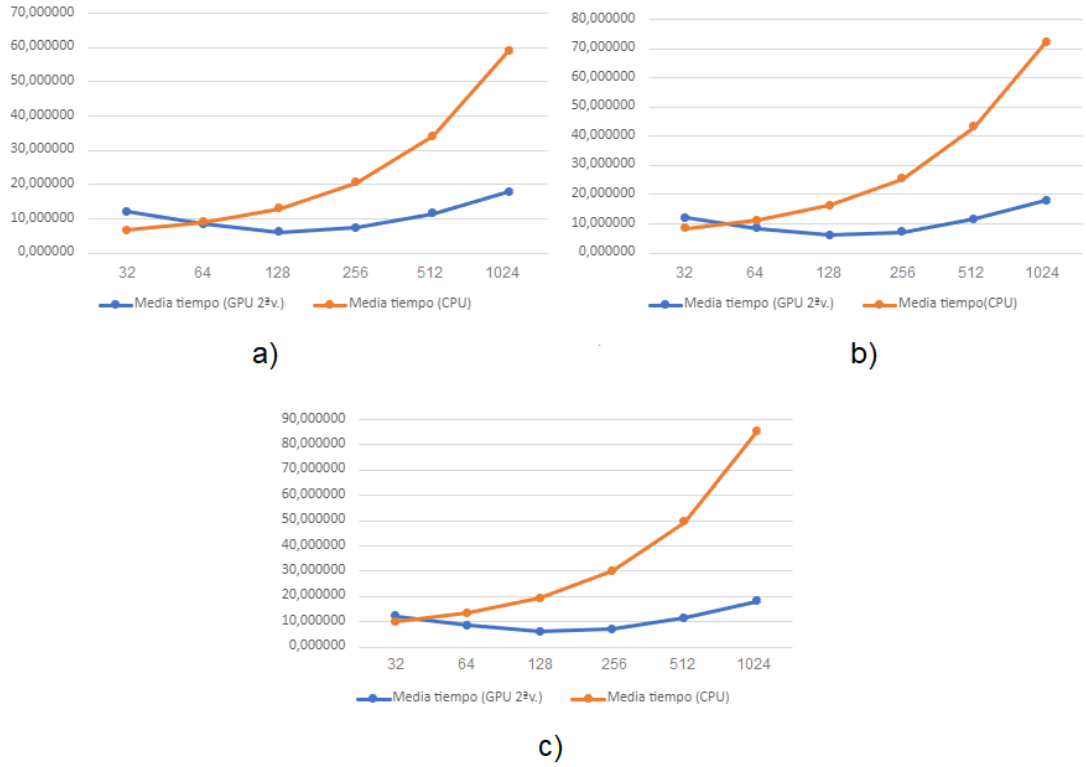


Figura 4.6: Media de tiempo de ejecución (GPU vs CPU) del problema 4155 con una función de coste con a) carga baja, b) carga media y c) carga alta.

Capítulo 5

Conclusiones y trabajo futuro

Este capítulo expone las conclusiones alcanzadas tras realizar el proyecto. Las conclusiones se presentan desde dos puntos de vista: desde el plano académico, explicando la relación entre la titulación y el trabajo y los conocimientos adquiridos durante su desarrollo, y desde el plano de la investigación, es decir, las conclusiones alcanzadas tras probar la proyección del algoritmo en la GPU. Por último, se describen las posibles líneas de trabajo futuro.

5.1 Plano académico

El desarrollo de este trabajo me ha permitido asimilar los conocimientos aprendidos a lo largo del grado, así como adquirir otros nuevos. A pesar de que CUDA era una plataforma nueva para mí, muchos de los conocimientos adquiridos a lo largo del grado han sido de gran ayuda. Aunque todas las asignaturas han sido importantes en el desarrollo del proyecto, a continuación comentaré aquellas más destacadas.

La más importante es *Concurrencia y Paralelismo*. Como he comentado anteriormente, CUDA no formaba parte del temario estudiado en la asignatura, pero la formación adquirida con respecto al paralelismo y las prácticas realizadas, me ha facilitado entender y ”visualizar” el algoritmo ACO paralelo. He ampliado los conocimientos que obtuve de esta asignatura estudiando CUDA y el uso de memoria compartida.

A la hora de realizar el apartado de planificación y costes, la asignatura *Gestión de Proyectos* ha sido muy útil, sobretodo para hacer el diagrama de Gantt.

De cara a la parte de programación, destacan varias asignaturas. *Programación I* y *Programación II* han servido como base para entender el código y realizar las propias implementaciones. Además, *Sistemas Operativos* y *Algoritmos* han sido muy útiles por varios motivos, el

principal porque en ellos se ha programado usando el mismo lenguaje de programación que en este trabajo, *C*. A mayores, gracias a *Sistemas Operativos* ha sido más sencillo entender y trabajar con los punteros, y *Algoritmos* me ha facilitado la comprensión a la hora de trabajar con algoritmos y cómo analizarlos en base a las mediciones de tiempo.

En lo que se refiere al entorno de trabajo, como ha sido desarrollado dentro de una Máquina Virtual Ubuntu 20.04, asignaturas como *Administración de Sistemas Operativos* y *Calidad en la Gestión TIC*, propias de la mención de *Tecnologías de la Información* me han ayudado mucho ya que en ellas se ha trabajado en entornos Unix. También han sido útiles para comprender y trabajar de forma más ágil en el clúster *Pluton*. Además, para la elaboración de la memoria del proyecto, me ha sido de gran utilidad la asignatura *Diseño de Redes*, también de esta mención, ya que para las prácticas de esta materia se realizaban memorias en *Overleaf*.

Por último, en general muchas de las asignaturas del grado, entre las que destaco *Internet y Sistemas Distribuídos* me han proporcionado los conocimientos para emplear un sistema de control de versiones, ya que es una de las cosas más importantes en nuestro campo.

5.2 Plano de la investigación

El objetivo principal del proyecto, que es el estudio de una proyección del algoritmo ACO en la GPU, se ha alcanzado. Se han realizado dos versiones en CUDA y se han probado, demostrando que la segunda versión tiene un mejor rendimiento.

Para la proyección del ACO en GPU se ha planteado una paralelización de grano fino donde cada hormiga, encargada de realizar la construcción de cada nueva solución, se corresponde con un hilo. La primera versión incluye una única función en la GPU además de una sobrecarga de transferencias de datos mayor en comparación a la segunda versión. A mayores, la segunda versión ejecuta todo el bucle principal del algoritmo en GPU, aunque no todas se pueden realizar en paralelo, y solo las lleva a cabo un hilo. En esta segunda versión se reducen las comunicaciones entre CPU y GPU, obteniendo así mejores resultados. Para valorar no solo la corrección de la solución, sino también su eficiencia, se han realizado experimentos variando la carga computacional de la función de coste, que arrojan resultados muy prometedores.

El trabajo ha resultado interesante como punto de partida para el desarrollo de nuevas versiones que exploten todavía más los recursos de la GPU. En concreto, este trabajo sirve como base para realizar una implementación de grano grueso más eficiente.

5.3 Trabajo futuro

La implementación de grano fino realizada en este proyecto sirve como base para futuros trabajos. En concreto, a corto plazo se podría realizar una implementación de grano grueso multicolonia. Una implementación de grano grueso explotaría mejor los recursos de la GPU, aprovechando los distintos bloques de hilos. En este caso, en cada bloque residiría una colonia que se comunicaría con el resto haciendo uso de la memoria compartida. Además, en lugar de que las diferentes ejecuciones realizadas debido a la naturaleza estocástica del problema se realicen de forma secuencial, se podrían hacer todas a la vez, ejecutando cada una en un bloque distinto.

Bibliografía

[1] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms.* Wiley-Interscience, 2005.

[2] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors*, 2nd ed. Morgan Kaufmann, 2013.

[3] A. P. Diéguez, *Parallel Prefix Operations on Heterogeneous Platforms*, PhD thesis, Universidade da Coruña, 2018.

[4] "CUDA kernels are subdivided into blocks." [En línea]. Disponible en: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[5] "CUDA (Compute Unified Device Architecture)." [En línea]. Disponible en: https://developer.nvidia.com/cuda-zone

[6] P. González, R. Prado-Rodriguez, A. Gábor, J. Saez-Rodriguez, J. R. Banga, and R. Doallo, "Parallel ant colony optimization for the training of cell signaling networks," *Expert Systems with Applications*, vol. 208, p. 118199, 2022. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0957417422013586

[7] K. Schwaber and J. S. November, *The Definitive Guide to Scrum: The Rules of the Game*, 2020.

[8] "Repositorio del proyecto con el código y los resultados obtenidos." [En línea]. Disponible en: https://github.com/Melisa-barro/TFG

[9] "Clúster Pluton." [En línea]. Disponible en: https://pluton.dec.udc.es/

[10] "Salarios de los programadores junior en españa (2023)." [En línea]. Disponible en: https://es.indeed.com/career/programador-junior/salaries

[11] "Salarios de los directores de proyecto en españa (2023)." [En línea]. Disponible en: https://es.indeed.com/career/director-de-proyecto/salaries

[12] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, 1st ed. Cambridge University Press, 2009, ch. 1 and 2.

[13] D. R. Penas, *Optimization in computational systems biology via high performance computing techniques*, Doctoral Thesis, Universidade da Coruña, 2017.

[14] M. Dorigo and T. Stützle, *Ant Colony Optimization.* Massachusetts Institute of Technology, 2004.

[15] J. Sáez-Rodríguez, L. G. Alexopoulos, J. P. Epperlein, R. Samaga, D. A. Lauffenburger, S. Klamt, and P. K. Sorger, "Discrete logic modelling as a means to link protein signalling networks with functional analysis of mammalian signal transduction," *Molecular Systems Biology*, 2009.

[16] M. Dorigo and T. Stützle, "Ant colony optimization: overview and recent advances," *Handbook of Metaheuristics*, pp. 311–351, 2019.

[17] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead," *IEEE Access*, vol. 8, pp. 225 134–225 180, 2020.

[18] M. W. Michael Kenzel, Bernhard Kerbl and M. Steinberger, *CUDA and Applications to Task-based Programming.* Tutorial in Eurographics'22, 2022.