UNIVERSIDAD DE ALMERÍA

ESCUELA POLITÉCNICA SUPERIOR Y
FACULTAD DE CIENCIAS EXPERIMENTALES
INGENIERO EN INFORMÁTICA

PROYECTO FIN DE CARRERA

DISEÑO DE ALGORITMO EVOLUTIVO EN CUDA. COMPARATIVA Y APLICACIÓN A LA TOMA DE DECISIONES EN LA AGRICULTURA.

Alumno: Antonio José Páez Ruiz
Director: Dr. D. José Antonio Torres Arriaza

25/06/2014

Agradecimientos

En primer lugar agradecer a mis padres todo el esfuerzo y sacrificio que han depositado sobre mí a lo largo de todos estos años, así como, por transmitirme su perseverancia y fuerza de voluntad para con el trabajo bien hecho.

A mi hermana María, por su apoyo y por sacarme una sonrisa en los momentos arduos.

A mi novia Cristina, por compartir conmigo los buenos y malos momentos y brindarme todo su apoyo en mis decisiones durante estos años.

A mi tío Paco, por ser para mí un referente académico.

A mi director José Antonio Torres, por la oportunidad que me ofreció, por todas las horas dedicadas y por mostrarme el fascinante campo de la Neurocomputación.

Y por último, y no por ello menos importante, a mis compañeros de piso, los que son y los que fueron y a mis compañeros de prácticas por esas largas tardes que me han llevado hasta este punto tan importante de mi vida.

Tabla de contenido

Índice de figuras

Introducción

La capital almeriense y más concretamente la comarca del poniente, son consideradas la huerta de Europa. La provincia es el mayor exponente europeo, y probablemente mundial, de la agricultura intensiva bajo plástico. El desarrollo sin precedentes de esta actividad agrícola la ha convertido en la actualidad en el principal pilar de la economía con un impacto capital sobre la economía de la provincia. Esta agricultura intensiva viene desarrollándose desde los pasados años 50 del pasado siglo, cuando los agricultores almerienses aprendieron, con la ayuda de los invernaderos, a sacarle 3 cosechas al año a su árida tierra.

Desde nuestro punto de vista, como ingenieros que somos, el campo de la agricultura resulta muy interesante para insertar en ella nuestros conocimientos y las tecnologías actuales, con el fin de mejorar, optimizar y rentabilizar las técnicas que se aplican hoy día en la agricultura.

La estrecha relación de la que gozan hoy tecnología y agricultura comenzó en la década de 1840 donde los científicos Liebig y Johnston comenzaron a hablar de una agricultura científica, los cimientos de esta son el emplear distintos procedimientos químicos sobre la agricultura para aumentar la producción. Esta forma de generar mayores cosechas se considera como la primera aplicación tecnológica a la agricultura, generando hasta hoy mayor cantidad de alimentos y una carga menor de trabajo para obtener los mismos con una menor involucración humana en la producción/procesamiento de estos gracias al empleo de la robótica.

A corto plazo, los objetivos de la implementación tecnológica se verán reflejados en rendimiento, calidad y reducción de insumos, pero generando mayor producción alimenticia. A largo plazo, se estima que se podrían crear alimentos supernutritivos para animales, que serían plantas que producirían muchos más nutrientes, tendrían mejor adaptación fisiológica para aventajar en la competencia a especies cercanas, ser tolerantes al estrés por sequía, y hasta un mejoramiento general de su fotosíntesis.

Concretando algo más, se pueden destacar algunas actividades tecnológicas que han centrado su desarrollo en la actividad hortofrutícola, se encuentran invernaderos domóticos (1) que permiten un cultivo casi autónomo (ver Figura 1), pasando por sistemas de telemetría (ver Figura 2) capaces de controlar las operaciones de toda una flota de maquinaria y registrar desde la posición de la máquina hasta los consumos a lo largo de la jornada con el fin de incrementar la superficie trabajada por hora y obtener una mayor productividad de las máquinas que se traduce en una reducción de los costes fijos (2), así como, el uso de drones para la elaboración de imágenes multiespectrales (3) (ver Figura 3) cuyo análisis sirve a los robots para procesar sus actuaciones o incluso tractores que incorporan un software para llevar a cabo una siembra a la carta adaptando esta tanto a la densidad del cultivo como la profundidad del mismo, según las características del terreno (4).

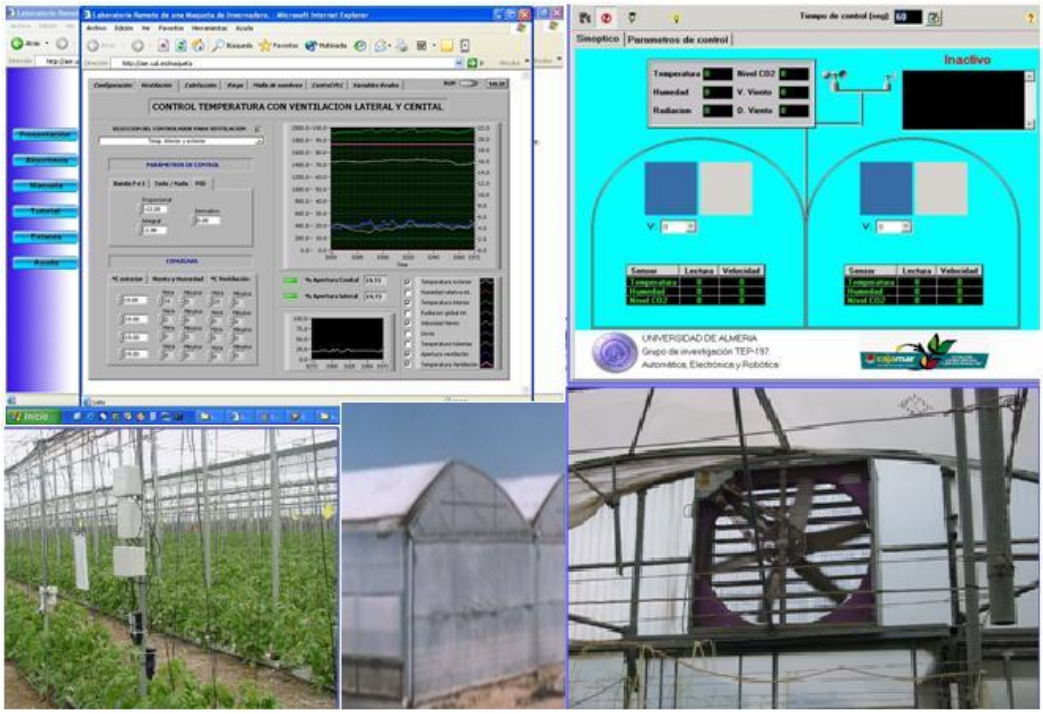

FIGURA 1: PANEL DE GESTIÓN PARA INVERNADERO DOMÓTICO

FIGURA 2: TRÁCTORES AUTÓNOMOS USADOS EN RECOLECTA Y SIEMBRA

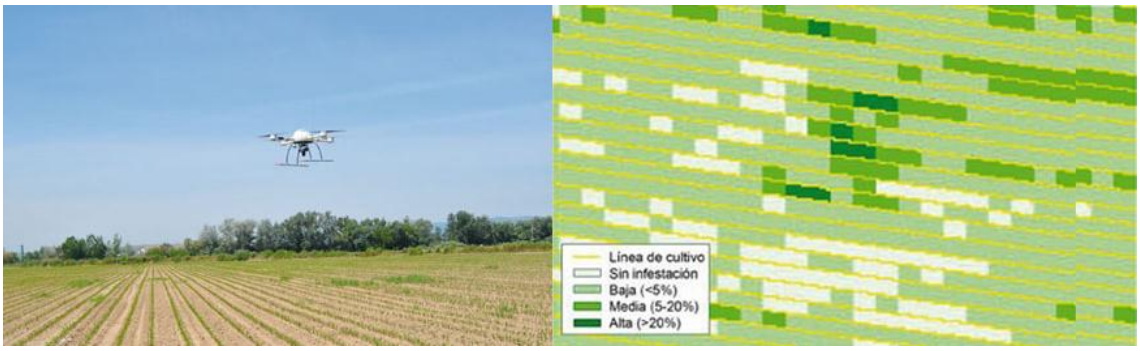

FIGURA 3: UAV Y SU MAPA OBTENIDO DE LAS INFESTACIONES DE LAS MALAS HIERBAS.

El fin de todas estas nuevas mejoras tecnológicas es reducir costes, mejorar la rentabilidad de los cultivos y disminuir el impacto ambiental que estos producen, a la par que se busca una mayor eficiencia en el cultivo mediante la gestión agronómica.

Estos argumentos han sido claves para centrar este trabajo en la ayuda a la toma de decisiones en la agricultura intensiva haciendo uso de la inteligencia artificial y del gran procesado de datos que nos brindan hoy los equipos informáticos y las nuevas tecnologías.

Idea del proyecto

Son diversas las etapas por la que una plantación pasa durante toda una campaña, desde el acondicionamiento de la finca, la siembra y puesta en producción, hasta la recogida del producto y finalización de la campaña, siendo la fases intermedias las más críticas y en las que más atención se necesita prestar a las plantas ya que una buena trayectoria puede suponer buenos ingresos o al contrario, la perdida de toda una campaña y del activo invertido en la plantación durante esos primeros meses. (5)

Dentro de los factores más sensibles y que al final son los que afectan a los precios del producto final se encuentran características de los cuidados de las plantas que pueden pasar desapercibidos como su correcto tutorado, hasta las más delicadas y catastróficas que pueden ocurrir tras un mal uso de fitosanitarios, considerándose estos factores siempre como los más impredecibles para los agricultores.

La principal idea del proyecto es la simulación de plantaciones con distintas características para la ayuda de toma de decisiones en el sector agrícola. Este simulador basado en algoritmos evolutivos será capaz de analizar y replicar posibles situaciones relacionadas con la siembra, recogida y tiempo de acondicionamiento de la finca entre campañas, las cuales, pueden afectar al precio del producto con el fin de minimizar costes y maximizar recursos. Para este simulador se ha usado un algoritmo evolutivo, más concretamente, una estrategia evolutiva (EE), en la cual, cada espécimen representará una plantación y a su vez un posible óptimo de la función de fitness ideada.

Se va a realizar la implementación de este algoritmo en dos lenguajes distintos. La primera versión se va a desarrollar en C de forma secuencial y para la parte paralela se pretende aprovechar la tecnología CUDA que nos ofrece NVIDIA para obtener unos tiempos de cómputo menores en la toma de decisiones, así como, brindar una mayor seguridad ante ciertas circunstancias y decisiones que se puedan presentar a los agricultores de la provincia.

Una vez realizadas ambas implementaciones se presentan los resultados obtenidos de eficiencia y tiempos devueltos de cada algoritmo, así como, una comparativa de ellos.

La agricultura en la provincia de Almería

Ya es una realidad que durante décadas la principal actividad que ha influenciado en el mercado económico de la provincia de Almería y más concretamente de la comarca del poniente ha sido, es y será la agricultura intensiva. Los años de despegue de este sector se enmarcan entre 1950 y 1975, donde la renta agraria almeriense pasó de ser la última de la región a ser la primera, con un crecimiento siete veces superior contra un crecimiento de dos veces y media del conjunto regional y dos veces de la media de España.

El clima subdesértico, los suelos áridos y la escasez de agua no fueron impedimento para que los agricultores y empresarios almerienses desarrollaran un amplio sistema de producción y comercialización que actualmente cuenta con cerca de 30.000 Has. de horticultura intensiva. De estas 30.000 Has, 500 son las últimas invernadas durante el año presente a lo largo de toda la costa almeriense, desde la comarca del poniente hasta la del levante. Estas últimas hectáreas llevan a la superficie invernada a un nuevo record histórico de la provincia de Almería.

En las Figura 4 y Figura 5 se ve el gran contraste durante el despegue de la agricultura y la actualidad. (6)

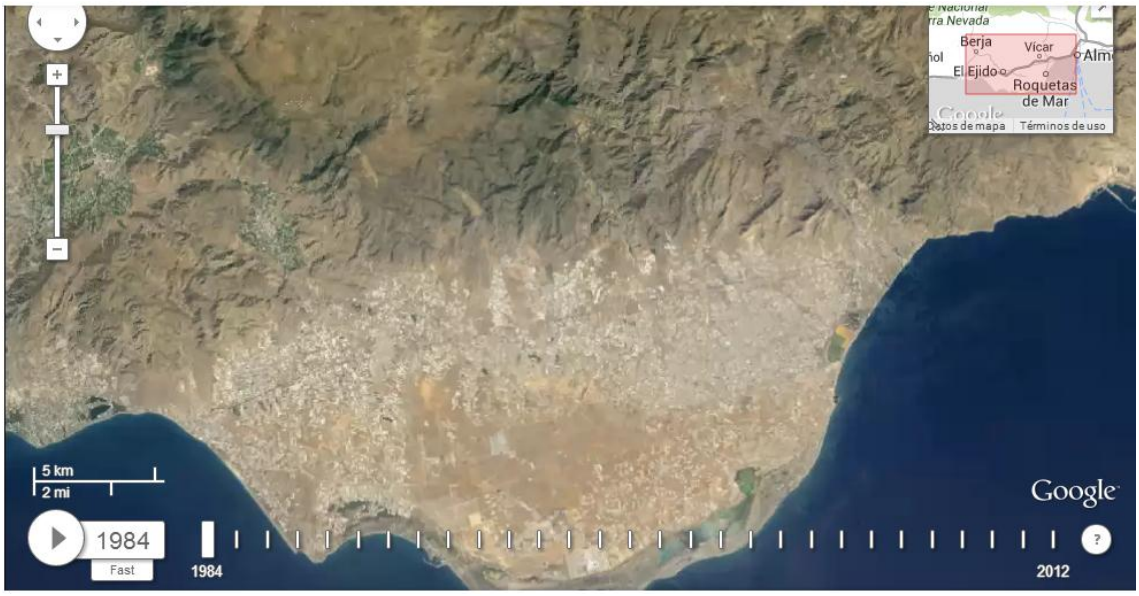

FIGURA 4: VISTA DE LA COMARCA DEL PONIENTE EN 1984

FIGURA 5: VISTA DE LA COMARCA DEL PONIENTE EN 2012

Durante todos estos años cabe destacas algunos hitos que han marcado la evolución de la agricultura, como pueden ser:

- Descubrimiento del enarenado (1956), hecho fortuito. Experimentación, verificación y validación de la técnica de enarenado (1957). Tras el sorprendente descubrimiento, se realizó una experiencia para enarenar varias parcelas en Roquetas de Mar, con resultados tan espectaculares que tierras prácticamente improductivas pasaron a ser cultivadas de hortalizas.
- Primeras experiencias del Instituto Nacional de Colonización INC sobre construcción de abrigos plásticos (1960), para aprovechar al máximo las invernales, y forzar los rendimientos.
- Construcción de los primeros invernaderos tipo “Parral de Almería” (1961).
- Desarrollo del sistema productivo y comercial primario (década de los sesenta), con una generalización de innovaciones para mejorar la productividad y la calidad de los productos.
- Consolidación del sistema productivo y comercial (década de los 80 y 90). El cerco de actividades que rodean la agricultura se vuelve muy complejo, incluyendo tanto actividades industriales como de servicios, que se vinculan con la agricultura intensiva abasteciéndola de todo tipo de productos plásticos, semillas, sistemas de riego, envases de cartón, madera y palets, abono orgánico y abejorros.

En la actualidad, y según los últimos datos recogidos, estos datos ratifican los primeros párrafos con los que inicia este documento. Según el último informe económico provincial "Almería en Cifras 2012" de la cámara de comercio (7) el año 2012 se cerró con un nuevo record histórico de las exportaciones provinciales, alcanzando un valor de 2.397,3 millones de euros. Este valor de exportación situó a la provincia en la tercera dentro de la comunidad autónoma. La evolución de exportaciones se ve en la Figura 6

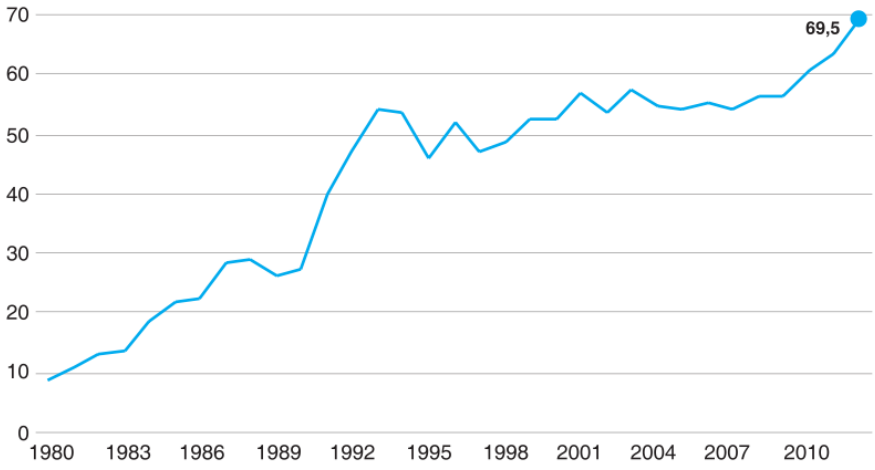

FIGURA 6: EVOLUCIÓN DE LAS EXPORTACIONES EN ALMERÍA (8)

Otro dato interesante de cara al trabajo futuro sobre el que se desarrolla este proyecto son los productos más demandados por las exportaciones, ya que esto es un buen indicador de producción de los mismos. Los productos usados para el estudio y desarrollo durante este trabajo son, ordenados de mayor demanda a menor, tomate Daniela, pimiento italiano, berenjena rayada, calabacín, judía stryke, melón amarillo y pepino español.

Producto	07/08	08/09	09/10	10/11	11/12	% var.
Berenjena	85.782	75.053	84.007	78.879	93.552	18,6
Calabacín	183.823	184.362	189.650	144.225	186.434	29,3
Col china	4.850	4.257	2.702	2.531	3.074	21,5
Judía verde	10.132	8.536	8.308	11.884	14.163	19,2
Lechuga	74.068	78.634	86.048	83.893	90.611	8,0
Melón	98.861	93.227	96.899	58.242	48.067	-17,5
Pepino	280.372	264.892	281.619	234.944	277.300	18,0
Pimiento	289.229	292.826	306.002	425.037	422.323	-0,6
Sandía	125.820	159.408	179.087	86.274	74.119	-14,1
Tomate	430.677	413.421	406.581	428.983	475.161	10,8
Otros	127.048	84.716	77.076	118.512	135.616	14,4
TOTAL	**1.710.662**	**1.659.332**	**1.717.979**	**1.673.406**	**1.820.419**	**8,8**

FIGURA 7: PRODUCTOS HORTÍCOLAS MÁS EXPORTADOS EN ALMERÍA, ALMERÍA EN CIFRAS 2012, PAG 59 (7)

Computación Evolutiva (CE)

Las técnicas de computación evolutiva (9) son un conjunto de heurísticas en vías emergentes, que han sido y son utilizadas de forma exitosa para la resolución de una variada gama de problemas en las áreas de optimización combinatoria, diseño de artefactos, búsqueda de información, control de dispositivos y aprendizaje automático, entre otros. Estas técnicas basan su operativa en la emulación de los mecanismos de la evolución natural identificados por Charles Darwin.

En la década de 1950 las relaciones detectadas entre los procesos de aprendizaje y la evolución natural dieron el marco para las primeras ideas sobre algoritmos evolutivos. Ya en 1948 Alan Turing había sugerido la conexión entre ambos aspectos, proponiendo desarrollar programas automodificables capaces de jugar ajedrez y simular otras actividades inteligentes desarrolladas por los seres humanos, utilizando técnicas evolutivas.

Las técnicas de CE nacieron en 1960, basándose en la naturaleza como máquina de resolver problemas y buscando el origen de dicho potencial para utilizarlo en la resolución de problemas complejos.

Desde los inicios de la era de la computación ya se buscaba la forma de aplicar los mecanismos naturales para diseñar dispositivos capaces de evolucionar y que pudiesen aplicarse a la resolución de complejos problemas en las áreas de aprendizaje automático, problemas de optimización y búsqueda de información. Cuando ya se afianzaron conceptos e ideas, tres modelos algorítmicos diferenciados se desarrollaron de forma simultánea. Las Estrategias de Evolución, la Programación Evolutiva y los Algoritmos Genéticos constituyen hoy en día las principales líneas de trabajo en el área de la computación evolutiva.

John von Neumann también se interesó en la combinación de técnicas evolutivas y computación. Von Neumann propuso mecanismos evolutivos basados en la programación para implementar autómatas con un poder computacional equivalente a una máquina universal de Turing. Además, conjeturó sobre el comportamiento de poblaciones de autómatas capaces de abordar problemas complejos comunicándose entre sí.

En el período comprendido entre los años 1953 y 1956 Nils Barricelli tomó el relevo y utilizó las ideas de Von Neumann, para llevar a cabo la primera simulación de “vida artificial” basada en principios evolutivos. Con su simulación de vida artificial, Barricelli sentó el precedente sobre el uso de los métodos evolutivos para la resolución de problemas.

Uno de los primeros intentos de aplicar técnicas evolutivas para la resolución de problemas prácticos de ingeniería, se propuso en el área de control estadísticos de procesos. En 1957, George Box propuso modificar los sistemas de operaciones estáticas tradicionales por mecanismos dinámicos capaces de realizar ajustes en las variables de control, evaluar sus efectos y modificar el proceso para mejorar los resultados obtenidos, siguiendo una analogía con el desarrollo de los procesos químicos en la naturaleza.

R. Friedberg trabajó en 1958 con un sistema de recompensas para calificar instrucciones que influían en la calidad de los resultados de programas sencillos. El mecanismo de aprendizaje se basaba, al igual que en la propuesta de Box, en realizar pequeñas modificaciones aleatorias y

luego evaluar los programas modificados. El sistema de recompensas de Friedberg se basaba en la idea de aplicar un mecanismo de selección de instrucciones asociado con la frecuencia con la cual producían resultados exitosos.

En la década de 1960, A. Newell y H. Simon propusieron un sistema al que llamaron "solucionador general de problemas" (General Problem Solver) que permitía al usuario especificar un escenario compuesto por objetos y definir operadores a aplicar sobre los objetos. El sistema se mostró capaz de resolver problemas sencillos definidos en espacios con número reducido de parámetros, utilizando técnicas heurísticas especificadas por el programador y que permitían al método evolucionar sus resultados utilizando un método al estilo del ensayo y error. El *General Problem Solver* consistió en una de las primeras propuestas de un sistema capaz de hallar soluciones genéricas, independientes del dominio de los problemas a resolver.

Sobre 1960-61 W. Bledsoe y H. Bremermann experimentaron con la idea de aplicar algoritmos genéticos para optimizar la eficiencia de los procesos de percepción: tratar los pesos de las conexiones sinápticas como nucleótidos de ADN, mutarlos, recombinarlos y seleccionarlos, como en la evolución darwiniana. Este trabajo se conoce como el precursor de los algoritmos genéticos y ellos son reconocidos como los padres de la inteligencia artificial.

La idea de codificación binaria vino de mano de Bledsoe y Bremermann, y el uso de un valor de aptitud en la aplicación de los algoritmos evolutivos para la resolución de problemas de optimización numérica. Bledsoe propuso el esquema de generar individuos, aplicar mutaciones y seleccionar los que produjeran mejores resultados y Bremermann lo extendió para considerar poblaciones de individuos. Con esto vieron la importancia del operador de mutación para evitar el estancamiento del proceso de búsqueda en mínimos locales del problema. El primer resultado teórico sobre la operativa de los algoritmos evolutivos fue alcanzado por Bremermann al determinar la probabilidad de mutación óptima para resolver problemas linealmente separables, presentado en Bremermann.

Una vez introducida la historia de la computación evolutiva se pasa a ver cómo trabajan estas.

Las técnicas de computación evolutiva trabajan sobre una población compuesta por un conjunto de codificaciones de soluciones candidatas para el problema a resolver. Estas soluciones interactúan entre sí, siguiendo los principios darwinianos de la evolución natural con la idea de producir iterativamente mejores soluciones al problema. Las soluciones potenciales se evalúan mediante una función de adecuación o función de fitness, que se crea de acuerdo al problema que se pretende resolver. En la naturaleza, durante el proceso evolutivo los seres vivos tratan de resolver los problemas relacionados con la supervivencia para garantizar la perpetuación de la especie. Mediante el mecanismo comentado, las técnicas de computación evolutiva emulan el proceso biológico de adaptación de los organismos vivos al entorno y las condiciones del medio, aplicándolo a la resolución de problemas en variadas áreas.

Como computación evolutiva se conoce a un conjunto de técnicas heurísticas de resolución de problemas complejos inspirados para su funcionamiento a los procesos de la evolución natural. Al trabajar sobre un conjunto de soluciones de un problema determinado, estas

técnicas basan su metodología en la selección de las mejores soluciones potenciales y la construcción de nuevas soluciones candidatas mediante recombinación de características de las soluciones seleccionadas.

En las literaturas referentes a computación evolutiva se ven varias propuestas para los algoritmos evolutivos. El algoritmo evolutivo más genérico es el siguiente, el cual, trabaja sobre una población inicial P;

```
CrearPoblacionInicial( P(0) );
generación = 0;
mientras (no CriterioParada) hacer
    Evaluar(P(generación));
    Padres = Seleccionar(P(generación));
    Hijos = Aplicar Operadores Evolutivos (Padres) ;
    NuevaPoblacion = Reemplazar(Hijos, P(generación));
    generación ++;
    P(generación) = NuevaPoblacion;
fin
retornar Mejor Solución Encontrada
```

El algoritmo evolutivo trabaja sobre individuos que representan potenciales soluciones al problema, codificados de acuerdo a un mecanismo prefijado. Los individuos son evaluados de acuerdo a una función de fitness que toma en cuenta la adecuación de cada solución al problema que se intenta resolver.

El diagrama de la estrategia evolutiva es mostrado en la Figura 8

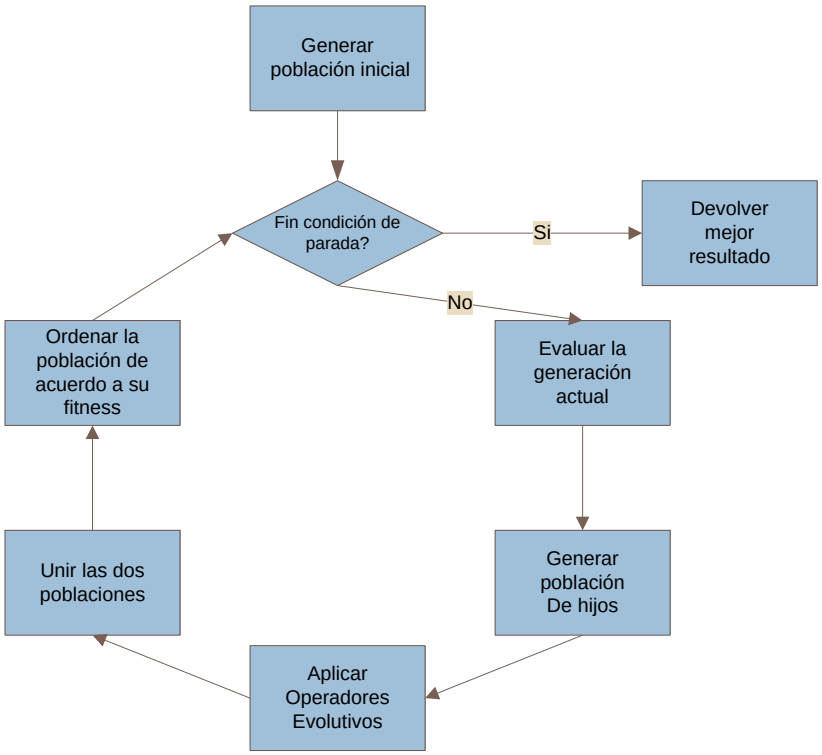

Figura 8: DIAGRAMA ESRATEGIA EVOLUTIVA GENÉRICA

El algoritmo evolutivo comienza con una etapa de inicialización de los individuos, que puede ser completamente aleatoria, muestreando al azar diferentes secciones del espacio de soluciones, o guiada de acuerdo a características del problema a resolver.

La evolución propiamente dicha se lleva a cabo en el bucle que genera nuevos individuos a partir de la población actual mediante un procedimiento de aplicación de operadores estocásticos. En este ciclo se distinguen cuatro etapas:

- Evaluación: etapa que consiste en asignar un valor de adecuación (fitness) a cada individuo en la población. Este valor evalúa que tan bien resuelve cada individuo el problema en cuestión, y es utilizado para guiar el mecanismo evolutivo.
- Selección: proceso que determina candidatos adecuados, de acuerdo a sus valores de fitness, para la aplicación de los operadores evolutivos con el objetivo de engendrar la siguiente generación de individuos.
- Aplicación de los operadores evolutivos: etapa que genera un conjunto de descendientes a partir de los individuos seleccionados en la etapa anterior.
- Reemplazo: mecanismo que realiza el recambio generacional, sustituyendo individuos de la generación anterior por descendientes creados en la etapa anterior.

Diversas políticas para la selección y el reemplazo de individuos permiten modificar las características del algoritmo evolutivo. Aplicando políticas adecuadas es posible privilegiar los individuos más adaptados en cada generación (estrategias elitistas), aumentar la presión selectiva sobre individuos mejor adaptados, generar un número reducido de descendientes en cada generación (modelo estacionario), y otras muchas variantes.

La condición de parada de la fase iterativa del algoritmo evolutivo usualmente toma en cuenta la cantidad de generaciones procesadas, deteniéndose el ciclo evolutivo al alcanzar un número prefijado de generaciones. Otras alternativas consideran la variación de los valores de fitness, deteniendo el ciclo evolutivo cuando el proceso se estanca y no obtiene mejoras considerables en los valores de fitness o estimaciones del error cometido respecto al valor óptimo del problema o una aproximación, en caso de conocerse.

Estrategias evolutivas

Las estrategias de evolución fueron introducidas por Rechenberg en 1965. En su propuesta inicial, consistía en un método de optimización que trabajaba sobre individuos compuestos por números reales para optimizar parámetros en problemas de diseño en ingeniería. El método modela la evolución al nivel de los propios valores a optimizar aplicando un mecanismo de selección determinística y un operador de mutación aleatorio basado en distribuciones de probabilidad, en general distribuciones gaussianas.

A partir de esa primera propuesta, los métodos de estrategias evolutivas se han difundido ampliamente. La utilización del operador de mutación como base del mecanismo evolutivo caracteriza a esta familia de métodos, aunque en versiones recientes se han propuesto modelos que incorporan mecanismos de recombinación como operador secundario, como la familia de cruzamientos aritméticos.

En su versión más simple, el proceso evolutivo se basa en la generación de un descendiente por parte de un individuo padre, mediante el operador de mutación, reemplazando el nuevo individuo generado a su progenitor en la población. Dos modelos avanzados de estrategias de evolución fueron formulados por Schwefel. Ambas variantes trabajan con un conjunto de padres que genera un conjunto de descendientes y se diferencian por el modo de reemplazar los individuos progenitores por sus descendientes. Estos modelos avanzados son actualmente conocidos como Estrategias de Evolución (μ, λ) (ver Figura 9) y Estrategias de Evolución (μ + λ) (ver Figura 10).

En (μ + λ), μ individuos producen λ hijos. La nueva población de (μ + λ) individuos se reduce, por un proceso de selección, nuevamente a μ individuos. Por otra parte, en (μ, λ), los μ individuos producen λ hijos (λ > μ) y el proceso de selección elige una nueva población de μ individuos desde el conjunto de λ hijos. Al hacer esto, la vida de cada individuo se limita a una generación. Esto permite que (μ, λ) trabaje mejor sobre problemas con un óptimo moviéndose sobre el tiempo, o en problemas donde la función objetivo presenta mucho ruido.

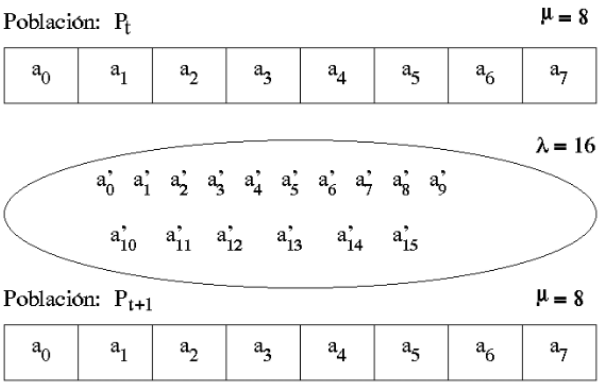

Figura 9: ESTRATEGIA EVOLUTIVA (μ, λ)

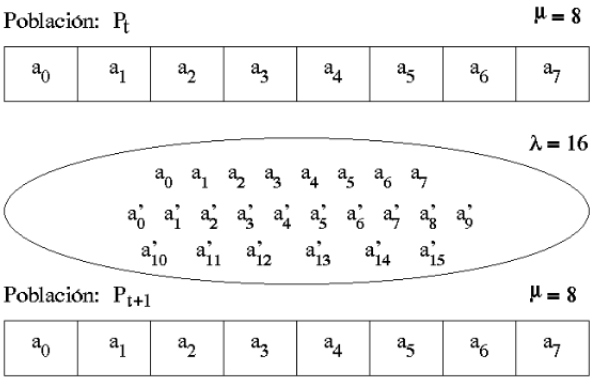

Figura 10: ESTRATEGIA EVOLUTIVA (μ + λ)

La selección (μ + λ), con la supervivencia garantizada de los mejores individuos, parece ser más efectiva, al realizar un proceso monótono de evolución. Sin embargo, este mecanismo de selección tiene varias desventajas cuando se compara con la selección (μ, λ), la cual restringe el tiempo de vida de los individuos a una generación:

- No es capaz de seguir un óptimo en movimiento ya que en el caso de medio ambientes cambiantes la selección (μ + λ) preserva la solución.
- La capacidad de la selección (μ, λ) para olvidar buenas soluciones es ventajosa en el caso de topologías multimodales.
- La selección (μ + λ) obstaculiza el mecanismo de auto adaptación para que trabaje adecuadamente con respecto a los parámetros estratégicos, porque los parámetros estratégicos mal adaptados pueden sobrevivir por un gran número de generaciones cuando producen mejoras en el fitness.

Una vez sabido todo esto y conocidos los puntos fuertes y débiles de cada una, se usará en este proyecto la selección (μ, λ), con el fin de obtener una exitosa auto adaptación de parámetros de estrategia a fin de facilitar la extinción de individuos mal adaptados.

CUDA

Este apartado está dedicado a la tecnología CUDA de NVDIA, a explicar que es, como funciona y las principales características que son necesarias para el desarrollo de este proyecto.

¿Qué es CUDA?

CUDA (10) es una tecnología desarrollada por la empresa NVIDIA. CUDA es una arquitectura de cálculo paralelo que aprovecha la gran potencia de las GPUs (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema. Esto trata, ni más ni menos, en usar las tarjetas que siempre se han dedicado al juego a otros campos, como son, el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos, la reconstrucción de imágenes de TC, el análisis sísmico o el trazado de rayos, entre otras.

Los distintos niveles hardware con los que trabaja CUDA son los que se ve en la Figura 11, en ella, se ve como se interconecta el host y los distintos componentes hardware de la gráfica, también se ven los diferentes bloques en los que divide los streaming multiprocessor (SM), y como estos están formados por múltiples procesadores que acceden a la misma memoria compartida. Todos estos componentes se ampliarán en las secciones siguientes.

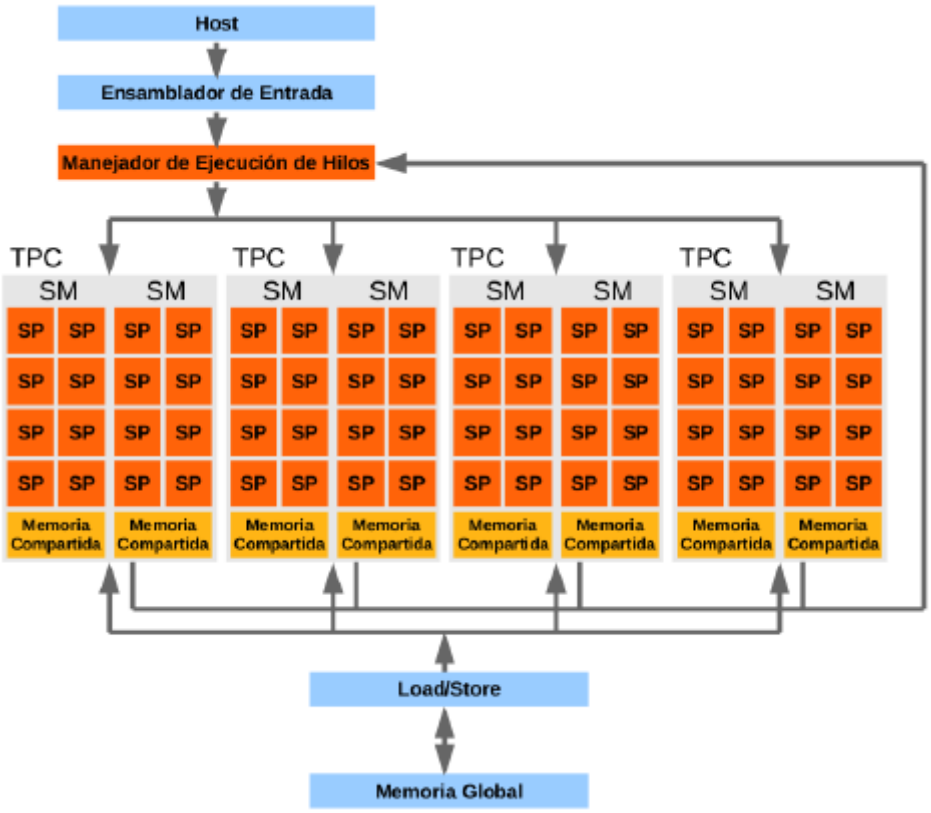

FIGURA 11: ARQUITECTURA CUDA

La forma en la que los sistemas informáticos están aprovechando esta tecnología es pasando de realizar el procesamiento central en la CPU a realizar cooprocesamiento repartido entre la CPU y la GPU. En la Figura 12, se ve como se realiza este cooprocesamiento.

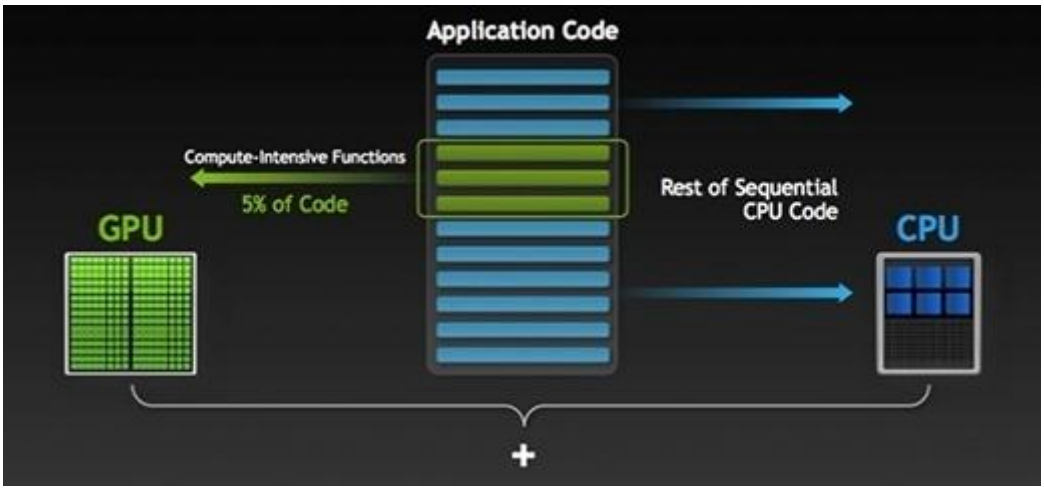

FIGURA 12: COOPROCESAMIENTO CPU-GPU

CUDA proporciona una extensión de C y C++ que permite implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad. El programador puede desarrollar ese paralelismo mediante diferentes lenguajes de alto nivel como C, C++ y Fortran o mediante estándares abiertos como las directivas de OpenACC. A demás, NVIDIA ha puesto en mano de los desarrolladores una gama completa de herramientas y soluciones pertenecientes al ecosistema CUDA, como entornos, librerías y soluciones para hacer debugging a nuestro código en paralelo, entre otras.

En la Figura 13 se ve una imagen en la que se compara un fragmento de código en C junto a ese mismo fragmento escrito en CUDA, este fragmento pertenece al algoritmo SAXPY (Single-precision real Alpha X Plus Y), muy utilizado en álgebra lineal.

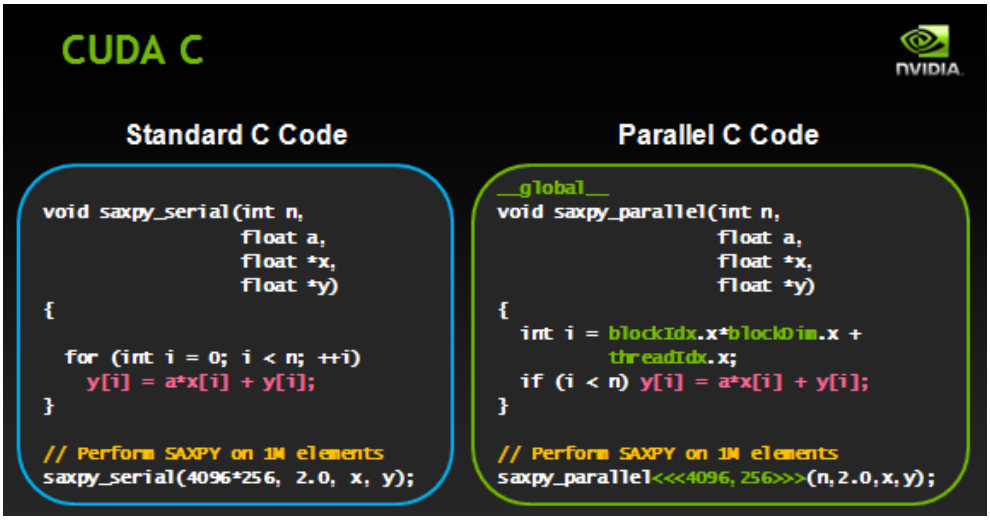

FIGURA 13: CÓDIGO C VS CÓDIGO CUDA

Taxonomía de Flynn de CUDA

Una de las clasificaciones para las arquitecturas paralelas existentes es la que definió el ingeniero Michael J. Flynn en 1972, conocida como la taxonomía de Flynn (11) por él. Esta clasificación distingue arquitecturas de multiprocesadores de acuerdo a dos dimensiones

independientes, datos e instrucciones, donde cada una de estas dimensiones puede tener sólo dos posibles estados, simple (single) y múltiple (múltiple).

En nuestro caso nos centraremos directamente en la que afecta a la tecnología CUDA, la clasificación SIMT.

Single Instruction, Multiple Thread (SIMT)

Antes de introducir la clasificación SIMT se debe de tener claro que esta proviene de la arquitectura Single Instruction, Multiple Thread -SIMD. Esta, instrucción única datos múltiples, puede verse ilustrada en la Figura 14– en la que se aprecia que todos los procesadores ejecutan la misma instrucción. Es un tipo de arquitectura paralela donde todas las unidades ejecutan la misma instrucción en un ciclo de reloj y donde cada unidad opera sobre una parte diferente de los datos. Esta arquitectura es especialmente apropiada para problemas de procesamiento de imágenes y gráficos. Dentro de SIMD se enmarcan los procesadores matriciales en los que existe más de una unidad de procesamiento trabajando sobre flujos de datos distintos, pero ejecutando la misma instrucción proporcionada por una única unidad de control.

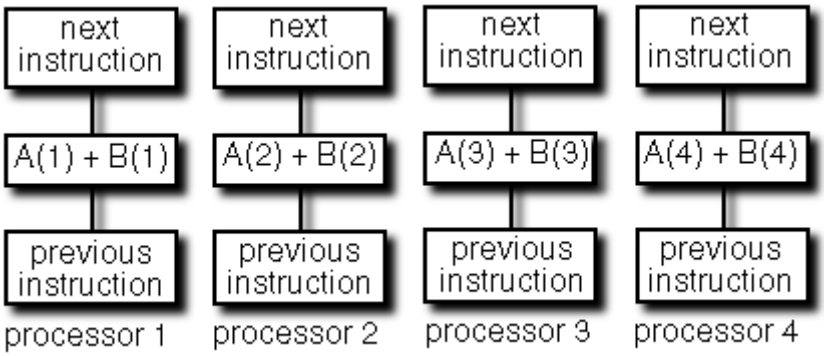

FIGURA 14: SIMD

La arquitectura SIMT (Figura 15) es similar a SIMD donde existen vector machines en las que una sola instrucción controla múltiples elementos de proceso. Una diferencia clave es que las organizaciones vectoriales SIMD exponen el ancho usado con el software, mientras que las instrucciones SIMT especifican la ejecución y el comportamiento de ramificación de un solo hilo. A diferencia de SIMD, SIMT permite a los programadores escribir código paralelo a nivel de hilo, hilos para escalares independientes, así como el código paralelo para la coordinación de hilos. Para ello los procesadores deberían ejecutar la instrucción leída por la Unidad de Instrucción en cada instante, de esta manera se sigue un modelo de arquitectura SIMT (Single Instruction Multiple Thread), donde un conjunto de threads ejecuta las misma instrucciones apuntadas por el cuerpo del código paralelo, explotando el paralelismo de datos, y en menor medida el de tareas. Además, existe una única unidad (hardware sequencer) que se encarga de gestionar el trabajo en paralelo sobre un grupo de esos hilos y una única instrucción se difunde a todos los elementos de procesamiento.

De forma resumida e puede decir que SIMT frente a SIMD permite a nivel de datos tener un paralelismo cuando los hilos son coherentes y a nivel de hilo cuando los hilos divergen y cada hilo se ejecuta de forma independiente.

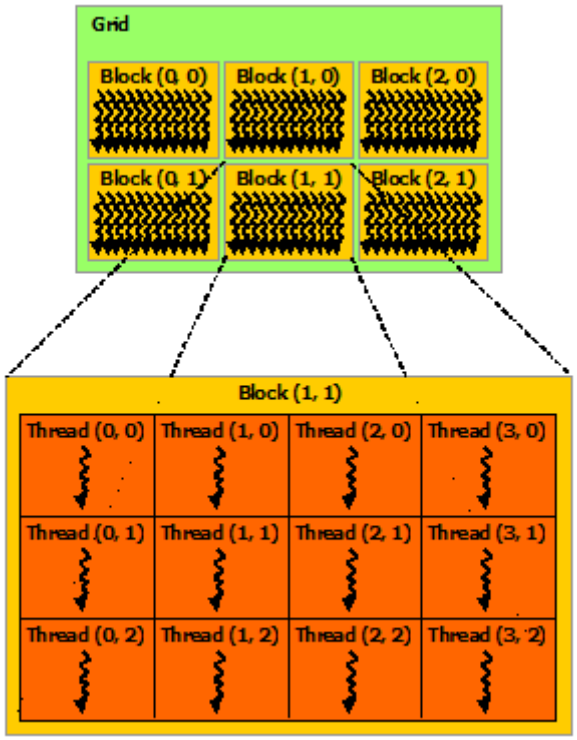

FIGURA 15: SIMT

Arquitectura de memorias en CUDA

A continuación se muestra la Arquitectura de memorias en CUDA (12).

Respecto a la jerarquía de memoria es imprescindible destacar que existen dos grupos de tipos de memoria que caracterizan la arquitectura CUDA, el primero de ellos es el ubicado de forma más local para los threads y denominados como *On-Chip* y el segundo situado de una forma más exterior o llamados *Off-Chip*, ya que estos tipos de memorias se usan para compartir datos entre todos los threads.

En el primer grupo, la primera memoria que se encuentra son los registros que son el tipo de memoria más rápida de la GPU y sólo accesible por cada hilo. Este espacio de memoria es gestionada por el compilador. El segundo grupo dentro de las *On-Chip* es la memoria local dedicada a cada thread. Ésta es sin duda la parte de la memoria que es privativa de un único thread y se encuentra dentro del ambiente de la función que implementa el kernel. Es la memoria para datos locales y utilizada exclusivamente para operaciones implementadas en la misma función, destacar que es la memoria de acceso más lento y no es cacheado. En tercer lugar está la memoria compartida por los threads. Este tipo de memoria permite que varios threads compartan información entre ellos cuando se ejecutan dentro de un mismo kernel, permitiendo principalmente optimizar el rendimiento.

En el segundo grupo de memorias – *Off-Chip* - coexisten tres tipos, más extensas y ubicadas en la memoria DRAM de la tarjeta. La primera de ellas la memoria global, reservada para propósitos de almacenamientos de datos de entrada y resultados del kernel. Tiene una capacidad de hasta 1.5 GB, permite tanto a los todos los threads como al host leer y escribir en ella; el segundo tipo de memoria en este grupo es la memoria constante, que ofrece un alto rendimiento con una capacidad de 64 KB con 8 KB de memoria caché lo que permite a todos

los threads leer el mismo valor de memoria simultáneamente en un ciclo de reloj. El tiempo de acceso a la memoria constante es similar al de los registros y sólo admite lectura desde los threads y lectura/escritura desde el host. Por último, la memoria de texturas explota la localidad espacial con vectores de 1, 2 y 3 dimensiones pero con un tiempo de acceso elevado aunque menor que el de la memoria global. Al igual que la memoria constante, sólo permite lectura desde los threads y lectura/escritura desde el host.

Todas estas memorias y su interacción con los hilos se muestran en la Figura 16

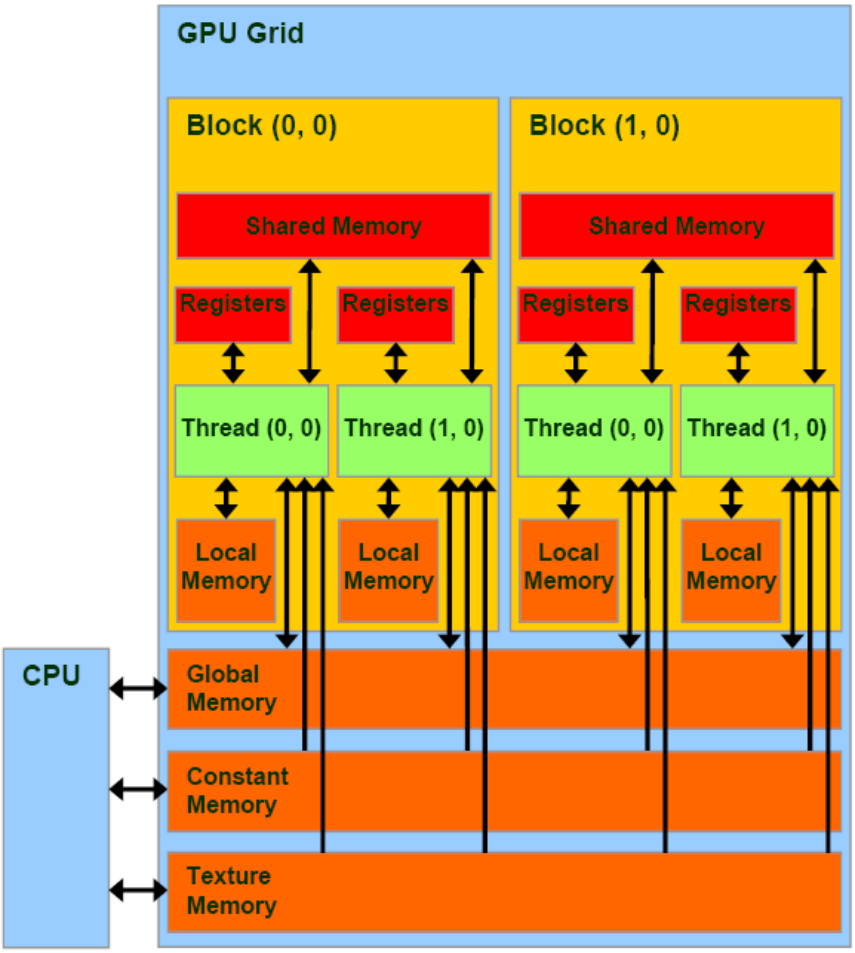

FIGURA 16: MODELO DE MEMORIAS CUDA

Como se ha ido explicando anteriormente no todas las memorias son iguales, permiten los mismos tipos de accesos o el mismo alcance para host o device, por ello la siguiente ilustración – Figura 17- muestra una tabla con esas particularidades de cada memoria. En ella se ve y en este orden, los siguientes campos:

- El tipo de memoria sobre el que se especifican las particularidades.
- La ubicación en la que se encuentra, dentro de los dos grupos principales antes mencionados
- Si dispone de cache o no.
- El tipo de acceso, es decir, solo lectura, lectura-escritura,…
- El alcance de la memoria, quien puede usarla, host o device.

- Duración de los datos en la memoria, mientras exista la aplicación, mientras se mantenga activo el bloque o el threads,…..

Memoria	Ubicación	Caché	Acceso	Alcance	Existencia
Constante	Off-chip	Sí	R	Device	Aplicación
Local	Off-chip	No	R-W	Thread	Aplicación
Global	Off-chip	No	R-W	Device	Aplicación
Compartida	Chip	No	R-W	Bloque	Bloque
Registros	Chip	-	R-W	Thread	Thread
Texturas	Off-chip	Sí	R??R-W??	Device	Aplicación

FIGURA 17: PROPIEDADES DE LAS DISTINTAS MEMORIAS CUDA

Modelo de programación paralela en CUDA

La característica principal de la programación en CUDA (10) es que permite combinar la implementación de código serie en el host con la implementación de código paralelo en el dispositivo, como ya se habló en el apartado ¿Qué es CUDA? De esta manera se consigue entrelazar ejecuciones serie con ejecuciones paralelas en los kernel. De forma visual queda reflejado en la Figura 18.

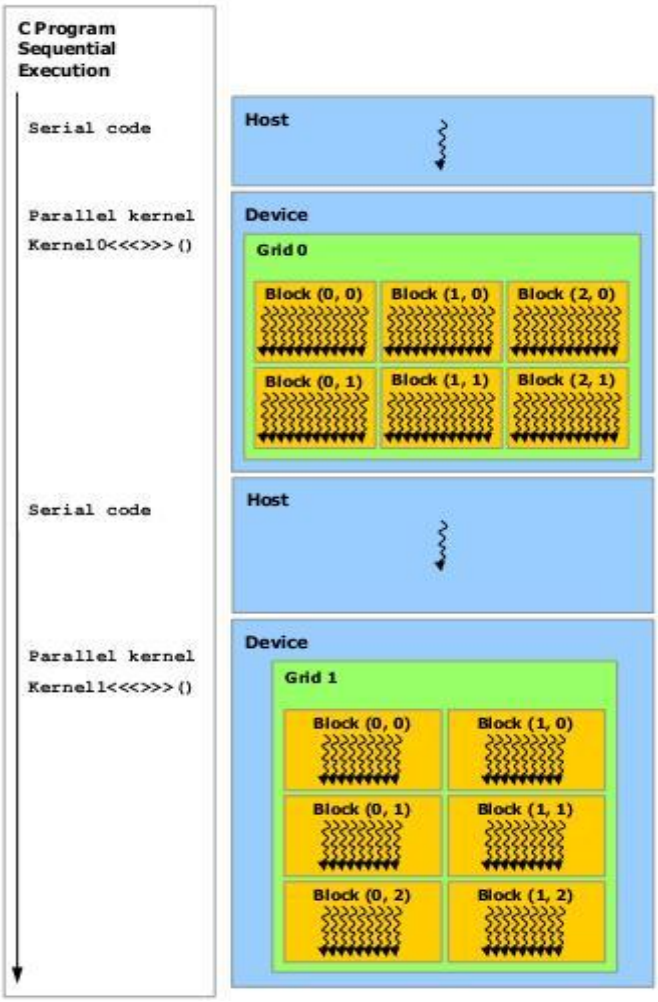

FIGURA 18: MODELO DE EJECUCIÓN EN CUDA

La ejecución del programa se lanza en el host (equipo nativo donde está instalada la tarjeta gráfica), en él se copian o generan los datos con los que trabajará en el device (el término device se refiere a la tarjeta gráfica), una vez copiados se lanza el kernel que ejecuta los hilos de forma paralela, con los resultados calculados se recuperan estos desde el device hacia el host y una vez allí estos se devuelven o se analizan. La ejecución en la parte del dispositivo se realiza por un conjunto finito de threads paralelos que actúan sobre una parte diferente de los datos. De esta forma se define un kernel como un conjunto determinado de muchos threads concurrentes, donde sólo un kernel es ejecutado al mismo tiempo en el dispositivo, y muchos threads participan en la ejecución del kernel.

En cada kernel cada thread tiene su propio identificador para poder realizar un paralelismo transparente sobre los datos. Los threads se agrupan en bloques, de esta forma se amplía el concepto kernel a un grid de bloques de threads, como se ve en la Figura 18. Cada uno de esos bloques será ejecutado en un Streaming Multiprocessor (SM) en forma independiente. Si hay suficientes SMs disponibles, todos los bloques de un grid son ejecutados en paralelo.

Otros aspectos de CUDA

(12) Dentro de este último apartado sobre CUDA se verán algunos términos en más profundidad, como Compute Capabilities, Warp, como CUDA identifica a cada hilo dentro de un kernel y la definición de las distintas funciones que permite el lenguaje de CUDA.

Para hacer uso óptimo de las GPUs, es necesario conocer las diferentes características de la tarjeta. Para ello NVIDIA utiliza un formato estandarizado que especifica estas características denominado compute capabilities - CC. La categorización incluye dos números, los cambios en la primera cifra implican cambios de generación, mientras que en la segunda implica una revisión. Las primeras GPUs de CUDA son de compute capability 1.0 mientras que las GPUs mar recientes – Maxwell, incorporan una CC de 5.0.

Algunas de las características técnicas de cada tarjeta que nos proporcionan las CC son: Warps residentes por SM, hilos residentes por SM, registros por SM, dimensión máxima del grid, máximo de instrucciones por kernel,….

Se conoce como warps a un grupo de 32 threads. Cada Streaming Multiprocessor (SM) crea, planifica y ejecuta hasta 24 warps pertenecientes a uno o más bloques (768 threads). De esta forma, cada warp ejecuta una instrucción en 4 ciclos de reloj. La ejecución en CUDA se planifica en base a warps y son los SM los encargados de crearlos, gestionarlos, planificarlos y ejecutarlo. Cuando los SMs deben ejecutar bloques de hilos, estos los dividen en warps y son lanzados. Después, cada conjunto de hilos dentro de un warp ejecuta una instrucción a la vez. Si los hilos de un warp divergen debido a una condición de bifurcación dependiente de los datos el warp serializa la ejecución de cada camino de la bifurcación, deshabilitando los hilos que no forman parte del camino de ejecución y posteriormente cuando todos los caminos completan su ejecución, los hilos convergen al mismo camino de ejecución. Por ello se dirá que la eficiencia máxima se logra cuando los hilos de un warp coinciden en su camino de ejecución.

Todos estos conceptos de SM, warp, grid, kernel que unifica CUDA junto con sus políticas de sincronización permite su ejecución en cualquier orden, concurrentemente o en serie. Por lo tanto, se considera que existe una barrera de sincronización implícita entre dos kernels lanzados consecutivamente. Como consecuencia, esta independencia proporciona una gran escalabilidad, donde los grids se escalan dependiendo del número de núcleos paralelos que existen en nuestra tarjeta gráfica. La Figura 19 muestra esto mismo, al comparar la ejecución de un mismo programa escrito en CUDA en una GPU con 2 SM o en otra con 4 SM.

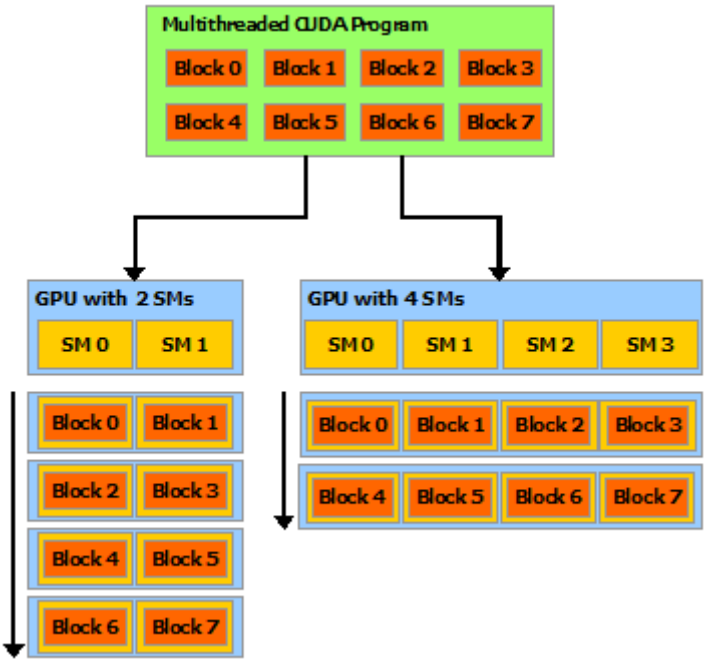

FIGURA 19: GESTIÓN DE ESCALABILIDAD EN CUDA

Lo siguiente que se muestra es como CUDA identifica a cada hilo dentro de un bloque dentro de un grid.

En CUDA los kernels que se lanzan se dividen en grid y en bloques de threads cada grid, de esta forma se entiende a un kernel como un grid de bloques de threads. Como ya se sabe tanto los threads como los bloques contienen un identificador unívoco que permite distinguirlos durante la ejecución, de esta forma, se puede por medio de los identificadores decidir sobre qué datos trabajar independientemente de lo que hagan otros threads. Así se dirá sobre qué parte de los datos proporcionados al kernel debe acceder un thread cuando se ejecute su porción de código que también ejecutarán otros threads con sus identificadores. Los identificadores de los bloques pueden ser unidimensionales (1D) o bidimensionales (2D), mientras que los identificadores de los threads puedes ser unidimensionales, bidimensionales o tridimensionales (3D). Esto simplifica enormemente el direccionamiento de memoria para datos multidimensionales, por lo que puede ser muy útil en procesamiento de imágenes y resolución de problemas matemáticos complejos que impliquen matrices.

La topología de un bloque se determinará dependiendo del problema a resolver y del mapeo a los threads sobre los datos con los que se trabajará.

Para poder acceder de una forma más cómoda a los datos, CUDA, ofrece algunas palabras reservadas:

- Para la identificación de threads, bloques y grid:
  - int threadIdx.x, threadIdx.y, threadIdx.z. Identificador (x, y, z) del thread dentro del bloque.
  - int blockIdx.x, blockIdx.y, blockIdx.z. Identificador (x, y, z) del bloque dentro del grid.
  - int blockDim.x, blockDim.y, blockDim.z. Tamaño (x, y, z) del bloque.

- Identificador único de cada thread:
  - threadIdx.x: Número de thread dentro de un bloque 1D
  - threadIdx.x + (blockIdx.x * blockDim.x): Identificador de thread único entre todos los bloques 1D (grid)
  - (threadIdx.x, threadIdx.y): Coordenadas del thread dentro de un bloque 2D
  - (threadIdx.x + (blockIdx.x * blockDim.x), threadIdx.y + (blockIdx.y * blockDim.y)): Coordenadas del thread dentro del grid (con bloques 2D)

Estos identificadores y su uso pueden verse de una manera más aplicada en la Figura 20

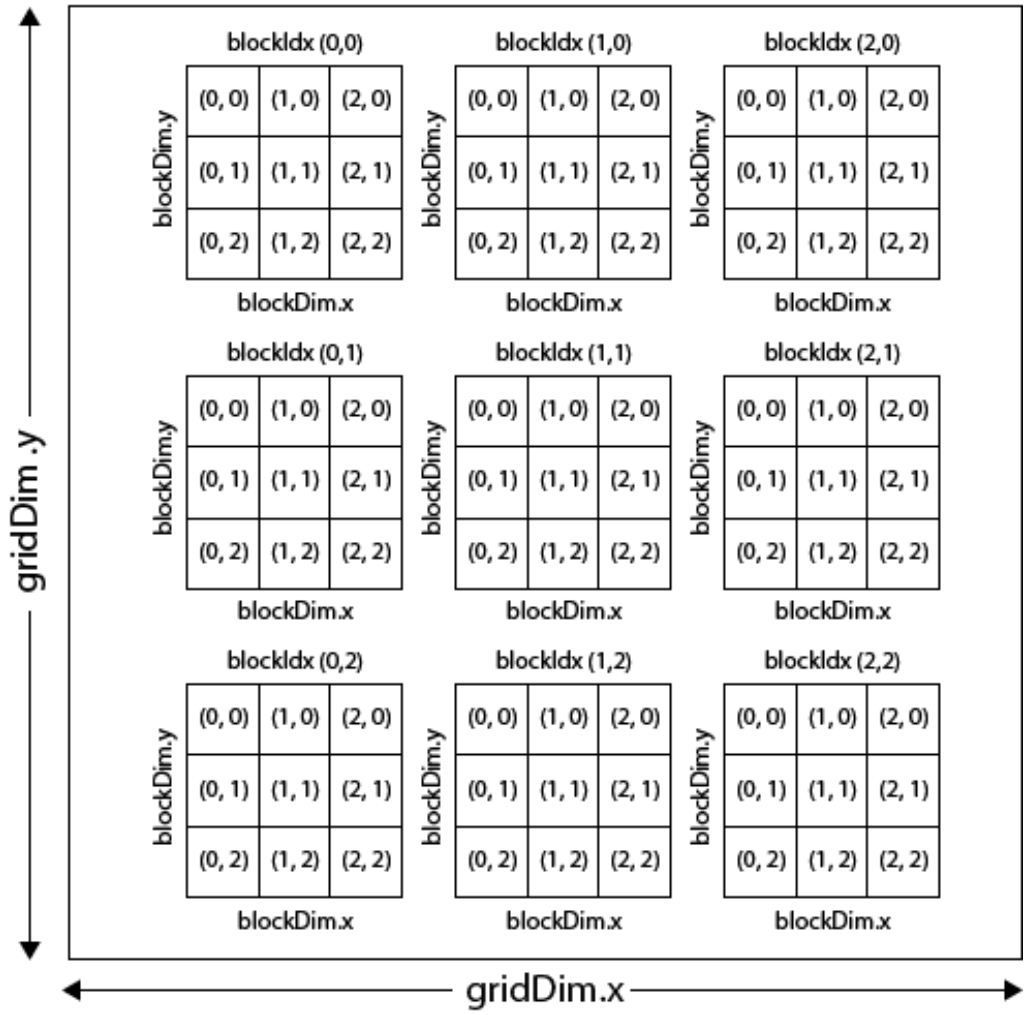

FIGURA 20: MANEJO DE IDENTIFICADORES DE HILOS

Otro aspecto de CUDA que también necesita ser aclarado es la nomenclatura usada para la creación de funciones. Son tres los tipos que existen: __global__, __device__ y __host__.

Funciones:	Ejecutable en:	Invocable desde:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

La función __global__ define funciones kernel, tiene que devolver void y no soportan llamadas recursivas. __device__ y __host__ pueden ser utilizados juntas (con compilación condicional), aunque lo normal es que no aparezcan funciones de tipo __host__.

El resto de conceptos y políticas de CUDA que se quedan en el tintero y que se aplicarán para el desarrollo de este proyecto, serán tratados en el apartado Diseño paralelo dentro del aparatado Estrategia Evolutiva .

Ejemplo de programa básico en CUDA

En este apartado se pretende mostrar un ejemplo básico de cómo el cooprocesamiento y todos los términos antes expuestos son capaces de acelerar la suma de vectores. Para ello se usa el código que incorpora la API de CUDA donde se muestra como se realiza la suma de dos vectores y se almacena el resultado en un tercero (13) . Gráficamente lo que se hace es lo que muestra la Figura 21, donde cada hilo será el encargado de realizar el cálculo de cada elemento en cada posición del vector.

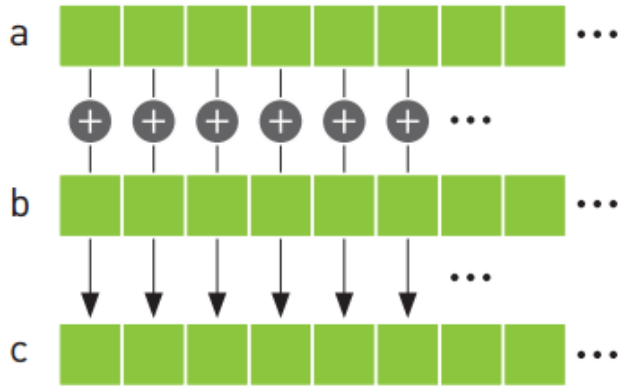

FIGURA 21: SUMA DE VECTORES

Lo primero que se hace es establecer la cantidad de elementos con los que se trabaja, crear los vectores en memoria local del host e inicializarlos. Lo que se pretende es sumar los vectores h_A y h_b y almacenar el resultado en h_c.

```
// Print the vector length to be used, and compute its size
int numElements = 50000;
size_t size = numElements * sizeof(float);
printf("[Vector addition of %d elements]\n", numElements);

// Allocate the host input vector A
float *h_A = (float *)malloc(size);

// Allocate the host input vector B
float *h_B = (float *)malloc(size);

// Allocate the host output vector C
float *h_C = (float *)malloc(size);

// Verify that allocations succeeded
if (h_A == NULL || h_B == NULL || h_C == NULL)
{
fprintf(stderr, "Failed to allocate host vectores!\n");
exit(EXIT_FAILURE);
}

// Initialize the host input vectores
for (int i = 0; i < numElements; ++i)
```

```
 {
 h_A[i] = rand()/(float)RAND_MAX;
 h_B[i] = rand()/(float)RAND_MAX;
 }
```

CÓDIGO 1: EJEMPLO SUMA DE VECTORES (I)

Una vez los datos estén listos en la memoria del host, pasa a preparar la memoria del device. Para ello lo que se hace es asignar la memoria del device que se crea según el número de elementos con los que trabaja y posteriormente copiar los datos desde la memoria del host hasta la memoria del device, con esto se consigue, almacenar los datos que tiene el vector h_A y h_B en los vectores que se encuentran en la memoria del device en d_A y d_B.

```
// Allocate the device input vector A
 float *d_A = NULL;
 err = cudaMalloc((void **)&d_A, size);
 if (err != cudaSuccess)
 {
     fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
     cudaGetErrorString(err));
     exit(EXIT_FAILURE);
 }

 // Allocate the device input vector B
 float *d_B = NULL;
 err = cudaMalloc((void **)&d_B, size);
 if (err != cudaSuccess)
 {
      fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
      cudaGetErrorString(err));
      exit(EXIT_FAILURE);
 }

 // Allocate the device output vector C
 float *d_C = NULL;
 err = cudaMalloc((void **)&d_C, size);
 if (err != cudaSuccess)
 {
       fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
      cudaGetErrorString(err));
       exit(EXIT_FAILURE);
 }

 // Copy the host input vectores A and B in host memory to the device input
vectores in
 // device memory
 printf("Copy input data from the host memory to the CUDA device\n");
 err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

 if (err != cudaSuccess)
 {
       fprintf(stderr, "Failed to copy vector A from host to device (error code
      %s)!\n", cudaGetErrorString(err));
       exit(EXIT_FAILURE);
 }

 err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

 if (err != cudaSuccess)
 {
```

```
        fprintf(stderr, "Failed to copy vector B from host to device (error
       code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}
```

CÓDIGO 2: EJEMPLO SUMA DE VECTORES (II)

Una vez los datos se encuentran en la memoria del device lo siguiente es lanzar el kernel. Para ello solo basta con indicar la configuración con la que se lanza el kernel. Al kernel se le pasan como argumentos los tres vectores asignados en la memoria del device y el número total de elementos.

```
// Launch the Vector Add CUDA Kernel
 int threadsPerBlock = 256;
 int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
 printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid,
threadsPerBlock);
 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
 err = cudaGetLastError();
 if (err != cudaSuccess)
 {
      fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
     cudaGetErrorString(err));
      exit(EXIT_FAILURE);
 }
```

CÓDIGO 3: EJEMPLO SUMA DE VECTORES (III)

Una vez terminada la ejecución paralela, por último, el código secuencial en el host recupera los datos y verifica que estos son correctos.

```
// Copy the device result vector in device memory to the host result vector
 // in host memory.
 printf("Copy output data from the CUDA device to the host memory\n");
 err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 if (err != cudaSuccess)
 {
     fprintf(stderr, "Failed to copy vector C from device to host (error code
    %s)!\n", cudaGetErrorString(err));
     exit(EXIT_FAILURE);
 }

 // Verify that the result vector is correct
 for (int i = 0; i < numElements; ++i)
 {
    if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
    {
       fprintf(stderr, "Result verification failed at element %d!\n", i);
       exit(EXIT_FAILURE);
    }
 }
```

CÓDIGO 4: EJEMPLO SUMA DE VECTORES (IV)

El kernel que realiza la suma de los dos vectores es el siguiente, donde se ve como se usa el identificar de cada hilo para recuperar los datos que se encuentran en ese índice y almacenarlos en su correspondiente posición, esto es lo que se muestra en la Figura 21.

Además se realiza una comprobación para que nunca se acceda a una posición que no existe con la instrucción if y la variable numElements

```
/**
 * CUDA Kernel Device code
 * Computes the vector addition of A and B into C. The 3 vectores have the same
 * number of elements numElements.
 **/
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
       C[i] = A[i] + B[i];
    }
}
```

CÓDIGO 5: EJEMPLO SUMA DE VECTORES (V)

VBA

Visual Basic para Aplicaciones o VBA (14), es una implementación del lenguaje de programación orientado a eventos de Microsoft Visual Basic 6 y asociado a un entorno de desarrollo integrado.

Visual Basic para aplicaciones permite la construcción de funciones definidas por el usuario, la automatización de procesos y el acceso a la API de Windows, así como, otras funciones de bajo nivel a través de bibliotecas (DLL). Con él se han sustituido y ampliado las capacidades de los lenguajes de programación de macros específicas para aplicaciones de Microsoft.

Este lenguaje es usado para controlar muchos aspectos de la aplicación que escapan a la aplicación que ejecuta estas macros, como pueden ser, la manipulación de la interfaz de usuario, menús y barras de herramientas y trabajo con formularios de usuario personalizados o cuadros de dialogo.

VBA usa la librería Visual Basic Runtime, pero normalmente solo se ejecuta código a través de la aplicación que invoca la subrutina, nunca como un programa independiente. Con él se manejan otras aplicaciones a través de los objetos OLE.

VBA está integrado en la mayoría de las aplicaciones de Microsoft Office, por ello su uso en este proyecto, además se puede encontrar en otras fuera de este paquete.

El código escrito en VBA se compila a un lenguaje intermedio llamado P-code que está optimizado para ejecutarse rápidamente en una plantilla física lo más pequeña posible y que se encuentra almacenado en ficheros con estructura COM separados de las informaciones de los documentos en los que se inserta.

De forma interna VBA es compatible con Visual Basic 6, donde el código fuente de los módulos y las clases de VBA puede ser importado directamente, ya que comparten la misma biblioteca y máquina virtual.

VBA es un lenguaje orientado a objetos, por ejemplo, se puede usar un objeto que corresponda a uno de los elementos de una aplicación Excel, como puede ser un libro, una hoja, una columna o una fila. Estos objetos están asociados con sus propiedades y métodos.

La codificación para cada objeto es muy simple, un objeto se asocia a un método o a una propiedad separándolos por un punto, por ejemplo: Objeto.Método u Objeto.Propiedad.

Todas las instrucciones que se escriban deberán de ir encerradas en macros o procedimientos, un procedimiento se caracteriza por la escritura de la instrucción Sub al comienzo y End Sub al final.

Al ser una modificación de Visual Basic las instrucciones, variables, sentencias y operaciones no difieren mucho de él, y por tanto tampoco del C estándar. Las nuevas funcionalidades que se añaden son los manejos de los objetos propios antes nombrados, como pueden ser, instrucciones para vaciar una tabla, para eliminar una columna o crear gráficas.

En él, también se usan sentencias del tipo if/elseif/else, select (similar a switch), bucles como for, while o do while.

Otro punto fuerte de VBA son los eventos, ya que como se dijo antes, es un lenguaje orientado a eventos. Un evento es el resultado de una acción y están asociados a los objetos y su estado después de la ocurrencia de este. Algunos de los eventos que registra VBA son: la selección de una celda o un rango, la selección de una hoja, si se abre o se cierra un libro, si se pulsa algún botón, si se ha calculado de nuevo la fórmula de una hoja de cálculo, etc...

Siguiendo por la línea de este proyecto, Excel incluye varios controladores de eventos que manejan acciones determinadas. Así, cuando ocurre uno de esos eventos y se le ha indicado a Excel que realice alguna acción cuando se produce, este ejecuta el código asociado a ese evento.

Por último hablar sobre las colecciones de objetos, una colección es un objeto utilizado para agrupar y administrar objetos relacionados. Por ejemplo, cada hoja tiene una colección de celdas, así, una colección es un objeto que representa todas las celdas de esa hoja y le permite recuperar una de la colección utilizando su índice y recorrer en iteración los elementos de la colección utilizando un bucle. Ventajas de utilizar colecciones frente a las matrices:

- Utilizan menos memoria.
- El acceso a los elementos de la colección es más flexible.
- Las colecciones tienen métodos para añadir nuevos elementos y eliminar otros.
- El tamaño de la colección se ajusta automáticamente.
- Pueden tener simultáneamente elementos de diferentes tipos.

Desarrollo del proyecto

En este apartado se habla sobre cómo se ha integrado cada uno de los puntos antes mencionados, como se aplican las distintas tecnologías descritas con anterioridad en el desarrollo del proyecto y porqué se han elegido para cada una de las partes donde se ven involucradas.

Comenzando por la agricultura, esta ya ha quedado bastante justificada en los primeros puntos de este documento, no obstante aquí se vuelve a recordar la idea fundamental. Cada cultivo pasa por varias etapas y cada plantación puede desarrollar varios cultivos, por ello, se pretende desarrollar una estrategia evolutiva que ayude a la planificación y toma de decisiones en lo que respecta a los factores más sensibles y que al final son los que afectan a los precios del producto, englobándolo a nivel de cultivo y año. Básicamente la idea principal del proyecto es la de simular millones de escenarios distintos y ver cual alcanza un máximo en su beneficio, con el fin de prestar una ayuda a la toma de decisiones en el sector agrícola. Este simulador analiza y replica posibles situaciones relacionadas con la siembra, recogida y tiempo de espera entre campañas, factores que afectan de forma directa al precio final del producto, ya que este oscila a lo largo de todo el año, con el fin de minimizar costes y maximizar recursos.

Una vez estudiadas las plantas que se simulan para las distintas campañas, fechas de siembra idóneas, kilos de producción, entre otras, se diseña y desarrolla la estrategia evolutiva.

La implementación de la estrategia evolutiva se realiza de dos formas distintas, de forma secuencial y de forma paralela, esto es, primero se desarrolla la EE en C, de forma secuencial, donde los bucles para las distintas operaciones sobre vectores de especímenes se llevan a cabo de forma normal, es decir, de espécimen en espécimen y posteriormente se traslada a CUDA, que no deja de ser una implementación en C, pero con todos los beneficios de la paralelización que esta ofrece, lo que permite una ejecución paralela para todos los especímenes y la posibilidad de eliminar los bucles internos de la versión en C.

El algoritmo genético se descarta al trabajar este sobre una codificación binaria. La programación evolutiva proviene una modificación de los algoritmos genéticos, donde lo que cambia es la representación de los individuos, aquí los individuos son ternas (tripletas) cuyos valores representan los estados de un autómata finito y esto tampoco se adapta a nuestro problema. Por tanto la mejor opción en la que respecta a la representación de especímenes es la estrategia evolutiva, la cual, permite una representación tanto discreta como continua. Otro motivo es que la programación evolutiva y los algoritmos genéticos usan normalmente selección estocástica, mientras que las estrategias evolutivas usan selección determinística, lo que lleva a seleccionar a los individuos basándose en su aptitud y no en una la probabilidad que depende de una heurística. También comentar que el algoritmo genético se descarta por enfatizar la importancia en el cruce y no sobre la mutación y la selección, factores más importantes para nuestra aplicación.

La elección del lenguaje de programación C para la parte secuencial viene dada, por su gran capacidad de ser altamente transportable a otros sistemas, por ser un lenguaje simple y por permitir un acceso a memoria de bajo nivel mediante el uso de punteros, agilizando más las operaciones a realizar. Otras cualidades que lo han hecho merecedor de esta elección es que pertenece a la familia de lenguajes estructurados, lo que lo hace entre otras cosas, más rápido

y más sencillo de entender para terceras personas, y además, permite la creación de nuevos tipos de datos mediante struct, sin que esto interfiera en la velocidad de ejecución.

Del otro lado, para el desarrollo de la EE en paralelo, la elección ha sido CUDA, tecnología puntera en la paralelización de procesos hoy día. Además, de permitir igual que C una gran portabilidad, brinda un gran grado de escalabilidad, lo que permite que en un futuro el código escrito corra a mayor velocidad si lo hace sobre una placa con mejores prestaciones que la actual. Otro motivo por el cual se ha elegido es que su programación es muy rápida y sencilla, ya que si se conoce C, la adaptación es inmediata.

Con todo esto ya se dispone del núcleo de la aplicación, falta dar el punto de interconexión entre el usuario y esta. Ahí es donde entra Microsoft Excel y el lenguaje de programación VBA. La interfaz gráfica diseñada trata de un libro de Excel con dos hojas, una para ofrecer datos de entrada y otra para los datos de salida. En la hoja de datos de entrada se insertan los parámetros con los que se lanza la EE. Una vez definidos, un botón es el encargado de generar un fichero con esos datos y lanzar el núcleo de la aplicación, que lee el fichero con los datos de entrada y devuelve otro con los datos de salida. Este último fichero es leído por otro botón en la hoja datos de salida, los cuales son interpretados por VBA e insertados en una tabla, para posteriormente ser dibujados sobre una gráfica también con VBA.

No olvidar que otro de los puntos de este proyecto es la comparativa, una vez todos los componentes estén listo, se realizarán distintas métricas sobre las cuales se describirán sus resultados, y nos permitirá comprobar si efectivamente la versión paralela de la EE tiene unos tiempos menores de ejecución que la secuencial.

Plan de trabajo

Tras la finalización de todas las pruebas, la estimación total de tiempo ha sido la esperada, pero con algunos cambios respecto a las horas dedicadas a cada ítem. El tiempo total de la realización del proyecto han sido 6 meses, de los cuales la mayor carga de tiempo la han tenido el estudio de la materia y la adquisición del conocimiento en el sector agrícola y la implementación de ambas versiones de la EE. El resto de tareas, como son el estudio de la tecnología CUDA, el montaje y puesta en marcha de los equipos o el diseño del algoritmo evolutivo si se han aproximado en horas a las estimadas. Así la realización del trabajo final queda como muestra la siguiente tabla:

#Fase	Tarea a realizar	Tiempo(horas)
1.1	Estudio arquitectura CUDA	56
1.2	Estudio lenguaje CUDA	56
2	Repaso conocimientos C	24
3	Montaje y puesta en marcha de los equipos	24
4	Estudio de los ecosistemas que se va a simular	200
5	Discretización de los datos de entrada	48
6	Diseño de la estrategia evolutiva	120
7.1	Diseño del núcleo de la aplicación paralelo	72
7.2	Diseño del núcleo de la aplicación secuencial	72
8	Diseño del GUI	24
9	Implementación del algoritmo evolutivo	80
10.1	Implementación del núcleo en paralelo	80
10.2	Implementación del núcleo secuencial	80
11	Implementación del GUI	24
12.1	Fase de pruebas del núcleo paralelo	80
12.2	Fase de pruebas del núcleo secuencial	80
12.3	Fase de pruebas del GUI	24
13	Unión del núcleo y GUI	56
	Fase de pruebas final	80
14	Realización de la memoria del proyecto	160

TABLA 1: ESTIMACIÓN FINAL

La línea de trabajo seguida se muestra en la tabla anterior a través de las numeraciones de las fases, pero a continuación se explican para entender con más detalla a que se refiere cada una de ellas.

Fases del desarrollo del proyecto.

Fase 1. Estudio arquitectura y lenguaje CUDA

Esta primera etapa se ha dedicado al estudio de todo lo relacionado con CUDA, ya que era una tecnología nueva para mí. En ella he leído la documentación de NVIDIA y varios manuales sobre la misma (15) (16), permitiéndome adquirir los conocimientos necesarios para la utilización de esta nueva tecnología.

Fase 2. Repaso conocimientos C

Esta fase fue una de las más rápidas, ya que durante mis estudios he dedicado prácticas al lenguaje de programación C y durante la realización del proyecto varias asignaturas impartieron sus prácticas en C. Por ello el lenguaje C no ha supuesto ningún retraso en la planificación.

Fase 3. Montaje y puesta en marcha del equipo

Para la realización de este proyecto ha sido necesaria la utilización de un servidor personal con capacidad para CUDA, el hardware de esté se detallará más adelante en el apartado Software, Tecnología y Hardware empleado, más concretamente en el apartado Equipo 1. Durante esta fase, se realiza el montaje del equipo y por supuesto de su tarjeta gráfica, una NVIDIA 9500 GT. También se instala el sistema operativo, Windows 7 Profesional, el IDE de desarrollo Microsoft Visual Studio Express, los drivers y el CUDA Toolkit el cual incluye el compilador de NVIDIA, bibliotecas matemáticas, y herramientas para la depuración y optimización del rendimiento de las aplicaciones. También se instalan otras aplicaciones complementarias, como son Excel y TeamViewer, un software para la utilización remota del equipo, lo que va a permitir utilizarlo como si estuviésemos delante de él y poder desarrollar nuestro código.

Fase 4. Estudio de los ecosistemas que se va a simular

Al contrario que la segunda fase, esta se alargó algo más de lo esperado, ya que el campo de la agricultura era desconocido para mí. Tuve que leer varios documentos sobre el tema, para comprender los especímenes que se simulan en la EE y diseñarlos de forma que estos sean los más fieles posibles a la realidad. El principal documento sobre el que me basé fue el publicado en 2003 por el Instituto Cajamar que habla sobre los distintos cultivos de la provincia y las distintas técnicas de producción que se aplican (17). De este se obtuvieron todos los datos necesarios para la implementación de los especímenes.

En esta fase también se realiza la recogida de datos de precios para cada una de las plantas que se van a estudiar, estos están accesibles en la web de fhalmeria y los usados corresponde a la alhóndiga la unión, que se pueden consultar desde el enlace en Bibliografía en el punto (18).

Fase 5. Discretización de los datos de entrada

Para la utilización de los datos recogidos ha sido necesario discretizarlos, esto es traducir valores infinitos o rangos de valores a unos valores que un ordenador y la EE sea capaz de manejar. Esta fase es una de las que menos duración tuvo ya que el estudio de los datos para ser insertados en el código fue rápido, y pocas modificaciones fueron necesarias a posteriori.

Fase 6. Diseño de la estrategia evolutiva

La fase 6 es la más delicada ya que todo lo restante dependerá de ella, por ello es una de las que más horas se ha llevado. En ella se realiza el diseño completo de la EE, se estudia su proceso de selección, de mutación y las alteraciones que se producirán en cada espécimen.

Además de, estudiar muy concienzudamente la función fitness, la cual es la causante de obtener un resultado u otro.

Fase 7. Diseño del núcleo de la aplicación secuencial e implementación

Esta fase trata de la traducción de lo definido en las fases cinco y seis al lenguaje de C para obtener la versión secuencial de la EE.

Fase 8. Diseño del núcleo de la aplicación paralela e implementación

Para la codificación de la EE en código paralelizado se parte del código escrito en la fase siete. Aquí se busca la forma de eliminar los bucles que trabajan con especímenes y lanzar tantos hilos CUDA como especímenes sean necesarios.

Fase 9. Diseño e implementación del GUI

En la fase número nueve se desarrolla la interfaz gráfica que enlaza al usuario final con la EE y sus resultados. Para ello se crea en Excel y haciendo uso del lenguaje VBA una interfaz que permita la inserción de los datos necesarios para la ejecución de la EE y tras finalizar esta muestre los resultados en forma de tabla y gráfica.

Fase 10. Pruebas

Para ir finalizando las distintas fases y antes de dar por concluida la implementación de la EE tanto en secuencial como en paralelo, se realizan una serie de batería de pruebas, que indiquen que ambas implementaciones están correctas. Estas pruebas son la verificación de la correcta generación de especímenes, su correcta evaluación y que los factores de mutación son los adecuados.

Fase 11. Recogida de datos para las estadísticas

Por último, se realizara una serie de ejecuciones con distintos valores en los parámetros de inicialización de la EE, pero ambos iguales para las dos versiones, lo que permite la recogida de datos de tiempo empleado en una y otra versión para su posterior comparación y obtención de conclusiones.

Software, Tecnología y Hardware empleado

En forma de esquema se presentan, ya que se han hablado de ellas en los puntos anteriores, el software, las tecnologías y el hardware usado durante la elaboración de este proyecto.

Software

El software necesario para el desarrollo de este proyecto es el siguiente:

- Microsoft Visual Studio 2010 Express
- CUDA Toolkit 5.5
- NVIDIA GPU Computing SDK 5.5
- Excel
- TeamViewer 8
- Procesador de textos Word

- Microsoft Project 2007

Tecnologías

La única y principal tecnología que se usará es CUDA, ya que el resto son recursos tanto software como lenguajes de programación.

Hardware

Se usarán dos equipos para las distintas baterías de pruebas del simulador:

- Equipo 1:
  - CPU: Intel Pentium D 2,80 GHZ
  - RAM: 2 GB
  - GPU: GeForce 9500 GT
    - La características más destacables de esta tarjeta gráfica son:
      - Cuenta con 32 Cores CUDA a una frecuencia de 1350 MHz
      - Permite 768 hilos por SM
      - Cuenta con 8192 registros por bloque
      - Ejecución de 512 hilos por bloque
      - Memoria global de 1024 Mb, memoria constante de 64 kb, memoria compartida por bloque de 16 kb
  - SO: Windows 7 Ultimate 32 bits
- Equipo 2:
  - CPU: Intel Core i7 4820K a 3,7GHz.
  - RAM: 32 GB
  - GPU: GeForce GTX 470
    - La características más destacables de esta tarjeta gráfica son:
      - Cuenta con 448 Cores CUDA a una frecuencia de 1215 MHz
      - Permite 1536 hilos por SM
      - Cuenta con 32768 registros por bloque
      - Ejecución de 1024 hilos por bloque
      - Memoria global de 1248 Mb, memoria constante de 65536 kb, memoria compartida por bloque de 49152 kb
  - SO: Windows 7 profesional 64 bits

Estrategia Evolutiva

En la estrategia evolutiva el principal elemento es cada uno de los especímenes, representados por Π, o sea cada uno de los invernaderos que se simularán. Cada uno de ellos está compuesto por varias campañas, dos o tres, $\{c_1, c_2\}$ o $\{c_1, c_2, c_3\}$ junto con los metros del invernadero y el fitness de dicho espécimen. Otros elementos adicionales son los protocolos de cada planta, que están representados por $\{h_1, h_2, ..., h_n\}$ y cada una de las pizarras de precios de cada planta, $\{P_{h1}, Ph2, ..., P_{hn}\}$.

Cada campaña c_i está compuesta por su identificador I_i, los kilos producidos por esa campaña k_i , y la fecha de siembra de la misma S_i.

Los componentes de cada protocolo son:

- Tiempo de maduración, que indica en días el tiempo que necesita una planta desde que se siembra su plántula hasta que está lista para ser recolectada por primera vez.
- Tiempo de recogida, indica en días el tiempo que permite el tipo de planta ser cosechada.
- Fecha inicial y final de siembra correcta, indica las fechas en las que se debe realizar la siembra para que la plantación ofrezca su mayor rendimiento en kilos de producción.
- Fecha inicial y final de siembra incorrecta, que indica las fechas en las que si se realiza la siembra de la plantación se obtiene el menor rendimiento en kilos de producción.
- Numero de kilos, se trata de dos valores, que indican el intervalo de kilos por cada mil metros de plantación que se obtienen en las mejores condiciones.
- Peores kilos se trata de dos valores, que indican el intervalo de kilos por cada mil metros de plantación que se obtienen en las peores condiciones.

La generación de cada uno de los atributos de una campaña se realiza de forma aleatoria, lo que da a todas las plantas la misma probabilidad de aparecer en una campaña. Para las fechas, se realiza con el mismo procedimiento, se crea una fecha de forma aleatoria entre las fechas de siembra de la planta antes obtenida al azar. Esto se hace así, ya que cada planta dispone de un protocolo y el agricultor debe respetarlo, por tanto la generación de un espécimen debe respetarlo también. La asignación de los kilos a cada campaña se realiza mediante una distribución uniforme continua, puesto que todos los kilos dentro del rango que indica el protocolo de la planta tienen igual longitud en la distribución de su rango y por tanto tiene la misma probabilidad.

Una distribución uniforme es aquella que puede tomar cualquier valor dentro de un intervalo, todos ellos con la misma probabilidad. Es una distribución continua porque puede tomar cualquier valor y no únicamente un número determinado (como ocurre en las distribuciones discretas). La gráfica de esta función de distribución de probabilidad es la siguiente (ver Figura 22).

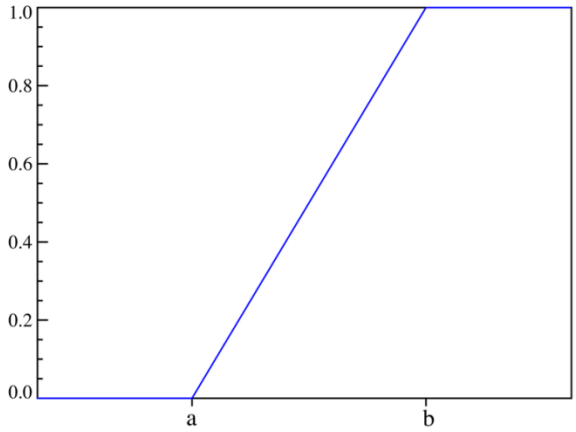

FIGURA 22: FUNCIÓN DE DISTRIBUCIÓN DE PROBABILIDAD

Además, se ha insertado un factor aleatorio al divisor de la función de densidad, el cual se multiplica por este aleatorio, quedando la función de distribución de la siguiente forma:

$$fda = \frac{a}{(b-a) * (R)}$$

Donde a y b son los kilos mínimos y máximos, respectivamente, indicados en el protocolo de la planta. R es un factor aleatorio entre 0 y 1.

Las características mutables para cada espécimen son: I_i - el identificador de una campaña - y K_i - los kilos que puede producir la misma -. Ambas características se mutan de acuerdo a un factor aleatorio y su disposición a través de una campana de Gauss, esto se debe, a que la naturaleza se comporta de forma gaussina. La probabilidad de mutación para ambos es del 32%. Esto se ha hecho así ya que la distribución que se usa es la normal – ver Figura 23- lo que quiere decir que la media (μ) se centra en cero y la varianza (σ) tiene un valor de uno, con estos parámetros y con el límite establecido en mu más una unidad, el 68% de los valores se quedan por encima del límite y el restante en la parte inferior. Todos los valores aleatorios que tras normalizarlos queden en esa parte inferior, indican que se debe mutar la característica en particular.

Siguiendo la misma táctica, se calcula el tiempo de esperas entre campañas. Este tiempo que puede ser de 15 días o un mes, será el que los agricultores dediquen a la limpieza de la finca y preparación para la próxima campaña. De experiencias recogidas, se ha observado que el tiempo de espera siempre se intenta que sea mínimo, por ello se ha dado una mayor probabilidad de aparición a una espera de 15 días que a la espera de un mes.

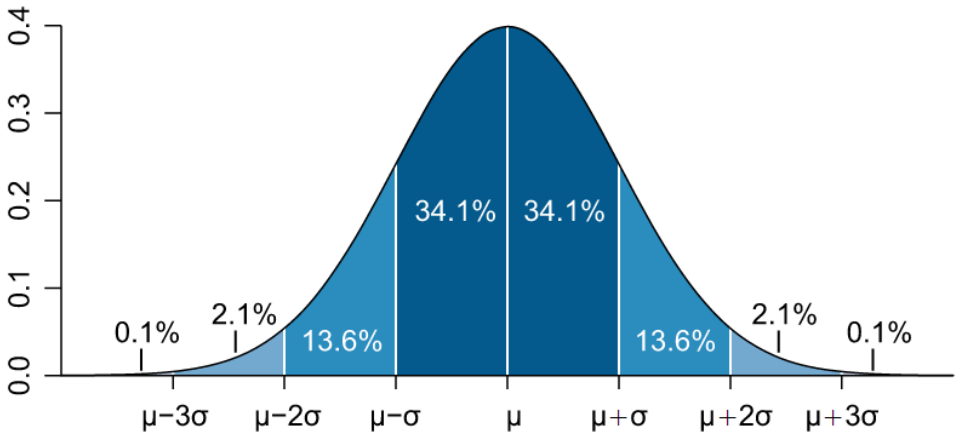

FIGURA 23: DISTRIBUCIÓN NORMAL TIPIFICADA

La función de densidad gaussiana usada para normalizar los valores obtenidos de forma aleatoria es:

$$e^{\frac{-1}{2}*\left(\frac{x-\mu}{\sigma}\right)^2}$$

Existen varias restricciones dentro de la generación aleatoria de especímenes, que son:

- La sumatoria de cada $M_i + R_i$ (tiempo de maduración y tiempo de recogida) debe de ser menor o igual que una constante T_{ci} que será el tiempo de duración de la campaña.
- La sumatoria de las duraciones (T_{ci}) de las campañas no podrán exceder nunca los 365 días.
- Las campañas no podrán solaparse, es decir, la fecha de siembra de la segunda campaña no podrá estar situada entre las fechas de siembra y la fecha final de campaña de la primera campaña.

En lo que se refiere a la función de fitness J siguiente $J(f_i)=(\Pi_i, h_i, P_{hi,} \lambda K_i, \lambda x_i)$ estos son los parámetros:

- Π_i espécimen a evaluar
- h_i protocolos de las plantas
- P_{hi} pizarras con los históricos de precios durante las recogidas de cada campaña.
- λK_i función aleatoria para variar la cantidad de kilos recogidos según las condiciones climatológicas.
- λx función aleatoria para insertar alteraciones como virus o contratiempos en la cosecha que disminuyen la cantidad de kilos que la plantación produce.

Con esta función de fitness se evalúa cuan bueno es un espécimen y es la encargada de calcular el valor del beneficio que se obtiene por todas las campañas en un año. Esta función es la que realmente muestra el resultado que se utiliza para la ayuda en la toma de decisiones, ya que gracias a ella se determina que sembrar, en qué fecha sembrarlo y el tiempo que se debe emplear en las labores de mantenimiento entre cosechas. Como ya se ha dicho el resultado final de esta función es el beneficio que se ha obtenido con cada uno de los especímenes en la sumatoria de beneficios para cada una de sus campañas, es decir, para cada

campaña se calcula la fecha cuando se comienza la recogida, a partir de la fecha de siembra y el tiempo de maduración necesario, una vez conseguidos el día y el mes en que se inicia la recolecta y según los días que indique el protocolo de la planta que se puede recolectar, comienza la sumatoria de precios. El total de esa sumatoria es multiplicado por el número de kilos que se esperan de la plantación y por el número de metros de la misma.

Una vez obtenidos los beneficios de cada campaña estos se suman para dar un valor de fitness final al espécimen.

Como ya se defendió en el apartado Estrategias evolutivas se usará una estrategia evolutiva (μ, λ). Para el proceso de selección, se producirá un swap de cultivos dentro de f_i. Nos hemos decantado por la estrategia evolutiva ya que las funciones con las que se trabajan son lineales.

En el proceso de selección se eligen los n mejores especímenes, se puede decir que es un proceso de selección elitista (reemplazo por inserción, los descendientes siempre pasan a la siguiente generación y sustituyen a los progenitores menos aptos).

Por último, antes de finalizar este apartado, destacar los datos recogidos y necesarios, comunes a ambas implementaciones que se pasan a desarrollar en los siguientes apartados.

Referente al número de campañas a lo largo de un año, más concretamente en la campaña 2011-2012, se detectó una tendencia en mayor uso por los agricultores al doble ciclo, en un intento evidente por maximizar los ingresos de las explotaciones. (8) Por esto, se ha decidido usar una función gaussiana que normalice valores aleatorios y si una vez normalizados se encuentran dentro del rango de los valores con un 70% de probabilidad de aparición, el espécimen que se crea es de dos campañas, sino se hará de tres.

Referente a las extensiones de las plantaciones, según una encuesta realizada a 27 explotaciones en El Ejido, La Mojonera, Roquetas de Mar, Vícar, Pechina, Almería y Níjar. Las superficies invernadas son variables yendo desde los 9.000m^2 a los 50.000m^2 y siendo la media de las explotaciones encuestadas 20.000m^2. Por ello para la asignación de estos metros se utilizará una función gaussiana que dado un valor aleatorio entre -10.000 y 10.000 normalice ese valor de acuerdo a la media dada por el estudio. (19)

Para cada una de las plantas usadas, tomate Daniela, pimiento italiano, berenjena rayada, calabacín, judía stryke, melón amarillo y pepino español, los protocolos de cada una de ellas se detallan a continuación:

*Tomate Daniela: ciclo de primavera, fecha de siembra del 15 de enero al 15 de febrero, la producción en este ciclo pueden alcanzar los 8–10 kg/m^2, el tiempo de recogida es de 60 días y el de maduración de 45. Por otro lado los valores negativos para el cultivo del tomate son, las fechas de siembra de cualquier día de mayo, donde los kilos que se obtendrán serán entre 4-5 kg/m^2.

*Pimiento italiano: ciclo de verano, comprende desde el 15 de septiembre hasta el 15 de octubre y su producción puede alcanzar los 6-7 kg/m^2, los días de recogida pueden alcanzar los 90 y su maduración se alcanza a los 60 días. En este caso los valores negativos para el cultivo

del pimiento son, las fechas de siembra de cualquier día de enero y la producción que se obtendrá será de 4-5 kg/m^2.

*Berenjena rayada: La producción puede alcanzar los 5-12 kg/m^2, la plantación se realiza del 1 al 15 de agosto, comenzando la recolección a finales de septiembre y finalizando en diciembre. Así se tiene, una duración de recogida de 90 días y un tiempo necesario de maduración de 30 días. Las peores fechas para plantar berenjena son cualquier día de diciembre, con la cual se pueden obtener entre 2.5-6 kg/m^2.

*Calabacín: cultivo temprano, el periodo de siembra comienza el 1 de agosto y finaliza el 15 del mismo mes, la producción en este ciclo pueden alcanzar los 12-15 kg/m^2. Esta campaña puede tener una duración de recogida de 90 días y un tiempo necesario de maduración de 30 días. Para el calabacín los peores meses de siembra son marzo, abril y mayo, con las que se obtienen una producción de 6-7.5 kg/m^2.

*Judía Strike: Plantación del 1 al 15 de agosto, comenzando la recolección a finales de septiembre y finalizando en diciembre, la producción en este ciclo pueden alcanzar los 5-6 kg/m2). Los días necesarios para una maduración son 45 días y los días que se puede recolectar el fruto son 60 días. Las peores fechas para la siembra de la judía son las comprendidas entre el 1 de abril y 31de mayo, dando una producción máxima de 1.5 -2 kg/m^2.

*Melón amarillo: Plantación de mediados de febrero a finales de marzo. Este tipo de planta puede alcanzar una producción s 4-5 kg/m^2. El tiempo necesario de maduración son 60 días y la recolecta puede durar 20 días. El rango de malas fechas de siembra para el melón es mayor, desde el 1 de agosto hasta el 30 de noviembre, obteniendo 2.5-3 kg/m^2 de producción.

*Pepino español: se puede realizar la siembra durante todo el mes de agosto, la producción esperada en este ciclo pueden alcanzar los 4-5 kg/m^2. Los días necesarios para una maduración son 30 días y los días que se puede recolectar el fruto son 60 días. En este último caso las fechas inadecuadas para la siembra del pepino son las comprendidas entre el 1 de marzo y el 30 de junio, resultado una producción de 0.5 – 1.5 kg/m^2.

Por último, indicar de forma ilustrativa, Figura 24, los ciclos de cada planta, con el fin de ver el solape que puede existir entre ellas y por tanto entre campañas a la hora de generar especímenes. La última columna corresponde a la suma de los tiempos de maduración y tiempo de recogida indicados en meses.

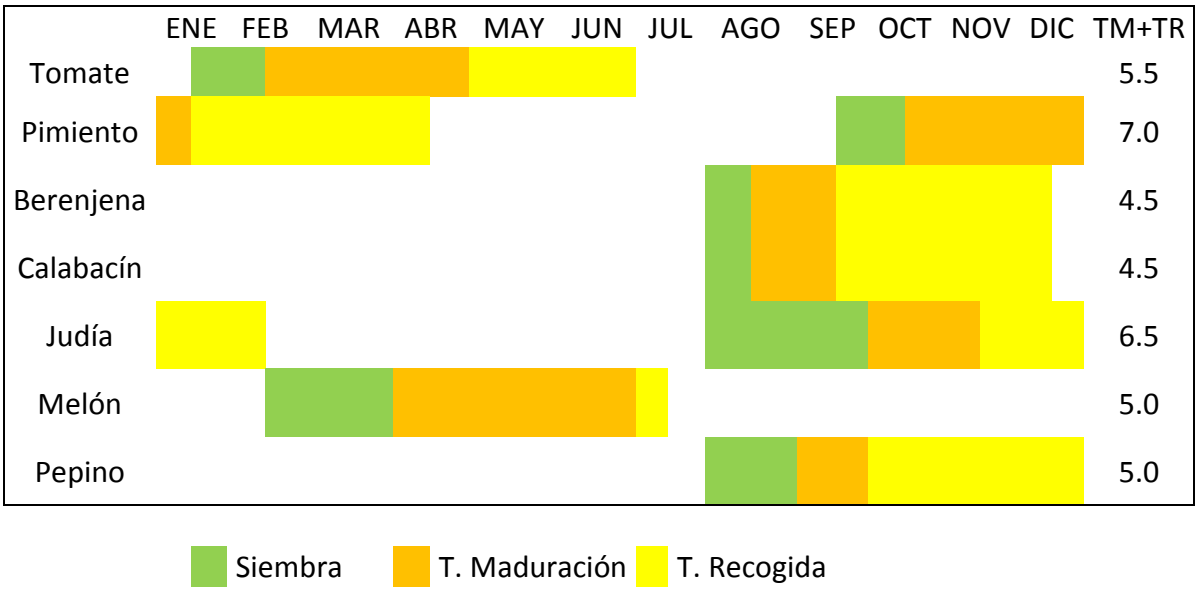

FIGURA 24: CICLOS DE COSECHAS

Todos estos datos han sido obtenidos del estudio técnico de producción en cultivos protegidos tomos I y II. (17)

Diseño secuencial de la EE

En este apartado se muestra cómo se ha llevado a cabo la implementación sobre el lenguaje C la estrategia evolutiva.

Lo primero que se muestra es el diagrama de flujo que representa la implementación, mostrando paso a paso como se desarrolla la EE en su versión secuencial.

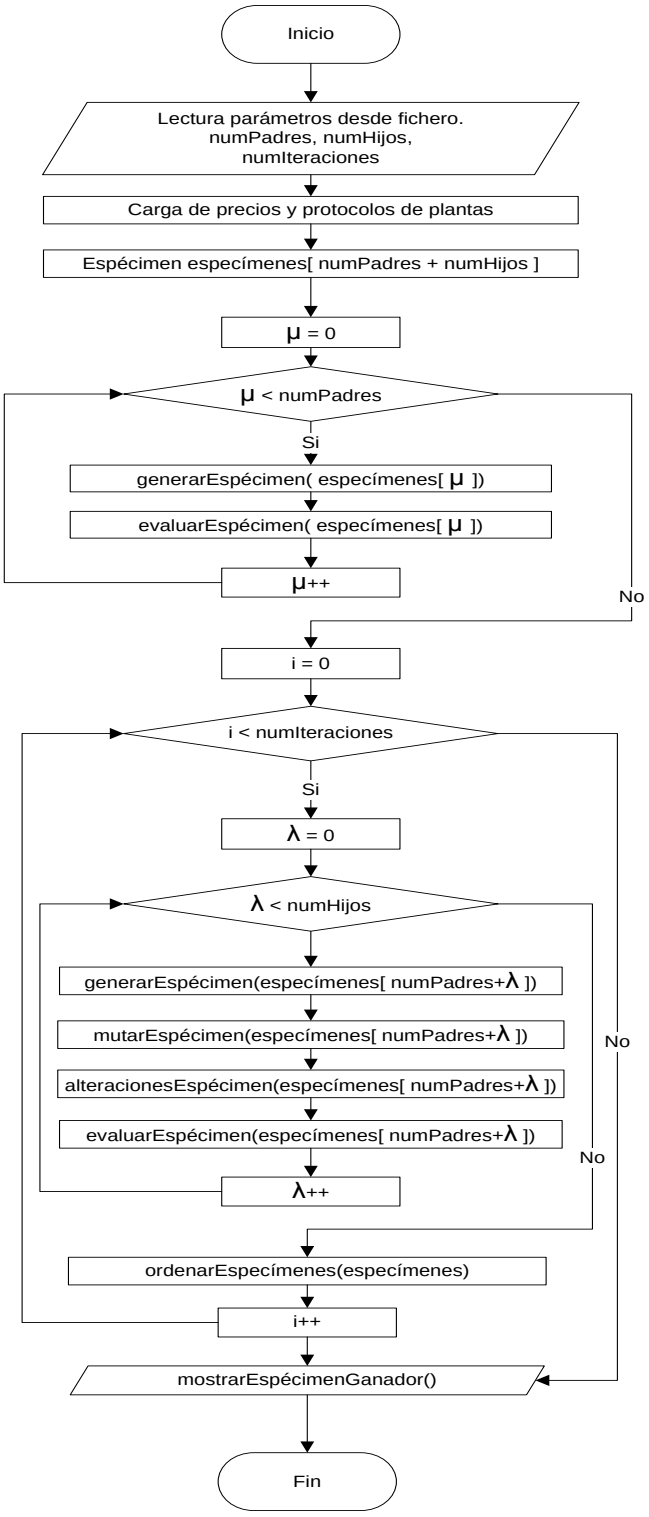

FIGURA 25: DIAGRAMA DE FLUJO VERSIÓN SECUENCIAL

El proyecto se divide en dos ficheros un .h que incluye todos los métodos y definiciones de variables usadas y un .c donde está escrito el main del programa. Dentro del fichero .h podemos encontrar las tres las estructuras usadas en la representación de los datos: campaña, espécimen y protocolo planta.

```
struct Campanya{
	int id;
	int kilos;
	tm fechasiembra;
};
struct Especimen{
	int metrosInvernadero;
	int numCampanyas;
	Campanya campanayas[3];
	float fitness;
	char esperas[2];
};
struct CaracteristicasPlanta {
	int t_maduracion;
	int t_recogida;
	struct tm f_inicialSiembraBueno;
	struct tm f_finalSiembraBueno;
	struct tm f_inicialSiembraMalo;
	struct tm f_finalSiembraMalo;
	int kilos[2];
	int peoreskilos[2];
};
```

CÓDIGO 6: ESTRUCTURAS USADAS

La estructura características planta representa a cada protocolo de actuación para cada planta, en él, cada fecha se representa mediante la estructura de tiempo *tm*, la cual está compuesta por nueve componentes de tipo entero que permite el almacenamiento de una fecha, su estructura se detalla a continuación:

```
int tm_sec; /* los segundos después del minuto -- [0,61] */
int tm_min; /* los minutos después de la hora -- [0,59] */
int tm_hour; /* las horas desde la medianoche -- [0,23] */
int tm_mday; /* el día del mes -- [1,31] */
int tm_mon; /* los meses desde Enero -- [0,11] */
int tm_year; /* los años desde 1900 */
int tm_wday; /* los días desde el Domingo -- [0,6] */
int tm_yday; /* los días desde Enero -- [0,365] */
int tm_isdst; /* el flag del Horario verano/invierno */
```

CÓDIGO 7: COMPONENTES DEL STRUCT TM

Kilos: se trata de un vector con dos valores, que indican el intervalo de kilos por cada mil metros de plantación que se pueden obtener en las mejores condiciones, almacenando en la posición 0 la menor cantidad y en la 1 la cantidad mayor.

Peores kilos: se trata de un vector con dos valores, que indican el intervalo de kilos por cada mil metros de plantación que se pueden obtener en las peores condiciones, almacenando en la posición 0 la menor cantidad y en la 1 la cantidad mayor.

Los datos que almacenan cada protocolo de planta, se han escrito en el código ya que estos no son variables, pues son datos reales que hay que respetar.

Pizarra de precios: {P_{h1}, P_{h2}, ..., P_{hn}}, en él está almacenado para cada planta h_1, el precio de venta para cada uno de los días que se hayan vendido ese género. Se trata de una matriz tridimensional en la que la primera dimensión representa la planta, la segunda el mes y la tercera el día, obteniendo del producto de las tres coordenadas el precio de venta para una planta en un mes y en un día. Antes de almacenar en esta matriz los datos se inicializa a cero.

```
float precios[numplantas][12][31];
```

Los precios se encuentran almacenados para cada planta en su correspondiente fichero, entonces, lo que se hace es leer cada uno de ellos y almacenarlos en su correspondiente dimensión, todos los archivos almacenan la información de la siguiente forma:

```
Mes1
dia1;mes1;precio1
dia2;mes1;precio2
Mes2
dia3;mes2;precio1
dia4;mes2;precio2
.
.
.
Mes12
dia1;mes12;precio1
dia2;mes12;precio2
```

Como se puede ver algunos meses no disponen de la información completa para todos sus días, esto no es problema. Los ficheros con la información de precios, se crean a partir de los precios recogidos por un historio de pizarra de precios para el año 2013 (18), se han insertado en un Excel y posteriormente con ayuda de VBA, se han creado los ficheros finales, esto se detalla en más profundidad en el capítulo Interfaz gráfica.

Para la representación de cada campaña: C_i, se usan tipos simples de datos. En ella se encuentra:

- Id: identifica que tipo de planta se siembra durante esa campaña. Este identificador varía y se genera de forma aleatoria dentro del rango [0-6].
- Kilos: representa los kilos que esa planta es capaz de producir por cada 1.000 metros de cultivo. Estos kilos se obtienen del protocolo de la planta antes definido y mediante una función gaussiana con un factor aleatorio.
- Fecha siembra: representa la fecha en la que será sembrada la plántula en el invernadero, también se obtiene de forma aleatoria pero siguiendo una distribución gaussiana. El rango de fechas en las que se puede sembrar cada planta se encuentra definido en los protocolos de planta. Se representa con la estructura tm.

La última estructura necesaria ha sido la que representa al espécimen, y se compone de:

- Metros Invernadero: entero que representa los metros del espécimen.
- numCampanyas: representa el número de campañas de las que se compone el espécimen.
- campanayas: vector de tres elementos de tipo campaña, que almacena cada una de las campañas de las que se compone un espécimen, si un espécimen tiene dos campañas la tercera se establece a null. Este vector se ha creado de forma estática y no de forma dinámica, porque las limitaciones de memoria de hoy día de los equipos son mayores y no existe problema de capacidad de esta, además, de forma estática se gana tiempo de ejecución al evitar gestionar la memoria.
- Fitness, este float representa al fitness de cada espécimen y el beneficio que aporta este.
- Esperas, este vector de char se ha creado con el fin de conocer las esperas que se han producido en ese espécimen, si son de 15 días o un mes. Los valores que almacena cada posición del vector son:
    - 0: para una espera de 15 días
    - 1: para una espera de un mes
    - X: que indica que no existen datos para esa espera, este valor se usa para los especímenes de dos campañas, en los que la espera entre la segunda y la tercera campaña no existe.

Una vez conocidos los datos con los que se va a trabajar y su representación, se muestra la parte más importante del código que forma parte del main:

```
 //carga de los precios y los valores con lo que se lanza la EE
 //inicialización de los protocolos
 .
 .
 .
int numUnion = (numespecimenes + numhijos);
 //Creación dinámica del vector que almacena los especímenes
Espécimen * especimenes = (Especimen *) malloc ( numUnion *
sizeof(Especimen));

srand (time(NULL));//necesario para que cada vez que se llame a rand(),
cambie la semilla y así parezcan más aleatorias las fechas

       /* algoritmo evolutivo */
       /* generar mu = numespecímenes especímenes aleatorios y
evaluarlos, representan los padres */
for(int i=0;i<numespecimenes;i++){
       especimenes[i]=generarEspecimen();
       evaluarEspecimen(especimenes,i);
}
printf("-Estrategia Evolutiva en marcha-\n");
int cont = 0;
do{
// generar landa hijos, mutarlos y evaluarlos
       for(int i=numespecimenes;i<numUnion;i++){
              especimenes[i]=generarEspecimen();
              mutarEspecimen(especimenes,i);
              alteracionesCosecha(especimenes,i);
```

```
            evaluarEspecimen(especimenes,i);
        }
 // ordenar problación y eliminar los menos aptos
        ordenarEspecimenes(especimenes,numUnion);
        cont++;
}while(cont < numiteraciones);

/* FIN algoritmo evolutivo */

 //Devolver el ganador
mostrarEspecimen(especimenes[0]);
 .
 .
 .
 .
//cálculo tiempo invertido y liberación de recursos
```

CÓDIGO 8: MAIN VERSIÓN SECUENCIAL

En esta versión en secuencial se puede ver como el manejo de especímenes se realiza dentro de bucles y de uno en uno. Esto nos da un orden total de O(numespecímenes) para el bucle de los padres y O(numiteraciones) más O(numhijos) para los bucles de la estrategia evolutiva más las operaciones con los especímenes hijo. Llegando a tener de complejidad lineal para los padres y un orden de complejidad cuadrática para los hijos.

Como se pueden ver son cuatro los métodos usados más importantes y de los que depende la EE de forma directa, además, del método de ordenación. Estos son: generarEspécimen(); mutarEspécimen(especímenes,i); alteracionesCosecha(especímenes,i); evaluarEspécimen(especímenes,i); y ordenarEspecímenes(especímenes,numUnion);

A continuación se pasa a detallar el comportamiento de cada método.

- generarEspécimen();
  Este método de pocas líneas simplemente decide si se crea un espécimen de dos o tres campañas, dependiendo de un valor aleatorio que se normaliza mediante la función de Gauss mostrada anteriormente.
  Un vez decidido si se crea un espécimen de dos o tres campañas se pasa a llamar a los correspondientes métodos creados para ello, la idea general de la creación de especímenes en ambos es la misma y se puede ver el algoritmo seguido a continuación:

```
Especimen especimen;//Se instancia un espécimen vacío
especimen.numCampanyas = 2;//Se asignan el número de campañas 2 o 3
tm f_finalCamp0; //representan las fechas en que finalizan la
tm f_finalCamp1; // campaña 0 y 1 respectivamente
// bucle que asegura que las campañas no se solapan
do{
 //generación de la primera campaña
 //generación de un id
       especimen.campanayas[0].id = rand () % (numplantas);
 //generación de una fecha aleatoria entre las fechas del protocolo de la planta para
la siembra
       especimen.campanayas[0].fechasiembra =
generarFechaAleatoriaEntreFechas(c_plantas[especimen.campanayas[0].id].f_inicialSiembr
aBueno, c_plantas[especimen.campanayas[0].id].f_finalSiembraBueno);
//cálculo fin de campaña 0, según la fecha de siembra y el tiempo de maduración + el
```

```
de recogida
	f_finalCamp0 = especimen.campanayas[0].fechasiembra;
	int meses = (int)(c_plantas[especimen.campanayas[0].id].t_maduracion +
c_plantas[especimen.campanayas[0].id].t_recogida) / 30;
	int dias = (c_plantas[especimen.campanayas[0].id].t_maduracion +
c_plantas[especimen.campanayas[0].id].t_recogida) % 30;
	dias = (f_finalCamp0.tm_mday + dias) %
diasmes[(especimen.campanayas[0].fechasiembra.tm_mon + meses) % 12];
	if ( dias == 0)
		dias = 1;
	f_finalCamp0.tm_mday = dias ;
	if(especimen.campanayas[0].fechasiembra.tm_mon + meses > 11){
		f_finalCamp0.tm_year += 1;
	}
	f_finalCamp0.tm_mon = (f_finalCamp0.tm_mon + meses) % 12;

//generación de la SEGUNDA campaña con un factor aleatorio para esperar 15 días o un
mes
	especimen.campanayas[1].id = rand () % (numplantas);
	especimen.campanayas[1].fechasiembra =
generarFechaAleatoriaEntreFechas(c_plantas[especimen.campanayas[1].id].f_inicialSiembr
aBueno, c_plantas[especimen.campanayas[1].id].f_finalSiembraBueno);
//Espera entre campañas
	double limite = gauss(1, 0, 1); // gauss de media+1*sigma, que es gauss(1) para
obtener una probabilidad del 70% aprox.
	double g = gauss(genNumAleatorioFloat(), 0, 1);
	if( g > limite ){//aumentar 15 días, mas probable que espere 15 días que un mes
		if(especimen.campanayas[1].fechasiembra.tm_mday + 15 >
diasmes[especimen.campanayas[1].fechasiembra.tm_mon]){
		//aumentar los días y un mes
		int findemes = diasmes[especimen.campanayas[1].fechasiembra.tm_mon] -
especimen.campanayas[1].fechasiembra.tm_mday ;
		int primerosmes = 15 - findemes;
		especimen.campanayas[1].fechasiembra.tm_mday = primerosmes;//día
asignado
		if(especimen.campanayas[1].fechasiembra.tm_mon + 1 > 11){
			especimen.campanayas[1].fechasiembra.tm_year += 1;
		}
		especimen.campanayas[1].fechasiembra.tm_mon =
(especimen.campanayas[1].fechasiembra.tm_mon + 1) % 12;
		}else{
			especimen.campanayas[1].fechasiembra.tm_mday += 15;
		}
		especimen.esperas[0]='0';//espera 15 dias
	}else{//Sino aumentar un mes
		if(especimen.campanayas[1].fechasiembra.tm_mon + 1 > 11){
			especimen.campanayas[1].fechasiembra.tm_year += 1;
		}
		especimen.campanayas[1].fechasiembra.tm_mon =
(especimen.campanayas[1].fechasiembra.tm_mon + 1) % 12;
		especimen.esperas[0]='1';//espera un mes
	}
	especimen.esperas[1]='X';
	//cálculo fin de campaña 1
	f_finalCamp1 = especimen.campanayas[1].fechasiembra;
	meses = (int)(c_plantas[especimen.campanayas[1].id].t_maduracion +
c_plantas[especimen.campanayas[1].id].t_recogida) / 30;
	dias = (c_plantas[especimen.campanayas[1].id].t_maduracion +
c_plantas[especimen.campanayas[1].id].t_recogida) % 30;
	dias = (f_finalCamp1.tm_mday + dias) %
diasmes[(especimen.campanayas[1].fechasiembra.tm_mon + meses) % 12];
	if ( dias == 0)
```

```
                dias = 1;
        f_finalCamp1.tm_mday = dias ;
        if(especimen.campanayas[1].fechasiembra.tm_mon + meses > 11){
                f_finalCamp1.tm_year += 1;
        }
        f_finalCamp1.tm_mon = (f_finalCamp1.tm_mon + meses) % 12;

}
 //comprobación que ninguna campaña se siembre a medias de la otra
 while( perteneceRango(especimen.campanayas[0].fechasiembra, f_finalCamp0,
 especimen.campanayas[1].fechasiembra) == 0 ||
                perteneceRango(especimen.campanayas[1].fechasiembra, f_finalCamp1,
 especimen.campanayas[0].fechasiembra) == 0 );

//asignación de kilos
for(int i=0;i< especimen.numCampanyas; i++){
        especimen.campanayas[i].kilos =
 uniforme(c_plantas[especimen.campanayas[i].id].kilos[0],
c_plantas[especimen.campanayas[i].id].kilos[1], genNumAleatorioFloat());
}

//aginación de metros para el invernadero
long media = 20000;
long var = 3500;
double g = 0.0;
double final = 0.0;
do{
        double r2 = (rand() % 20000) - 10000;//num aleatorio entre -10000 y 10000
        if(r2>0){//solo aceptamos los positivos
                g = gauss(r2, media, var);
                final = g * 1000000;
        }
}while( ( (int)(final) ) < 9000);
especimen.metrosInvernadero=(int)(final);
especimen.fitness=0.0;//inicializacion del fitness a 0
return espécimen; //se devuelve el espécimen creado
```

CÓDIGO 9: MÉTODO GENERAR ESPÉCIMEN, VERSIÓN SECUENCIAL

Para la creación de un espécimen de tres campañas simplemente se debe añadir la creación de una tercera campaña que se realiza de igual forma que la segunda y añadir las comprobaciones al buce do while para que ninguna campaña solape a otra, el código de la condición while es el siguiente para el caso de tres campañas,

```
perteneceRango(especimen.campanayas[0].fechasiembra, f_finalCamp0,
 especimen.campanayas[1].fechasiembra) == 0 ||
perteneceRango(especimen.campanayas[0].fechasiembra, f_finalCamp0,
 especimen.campanayas[2].fechasiembra) == 0 ||
perteneceRango(especimen.campanayas[1].fechasiembra, f_finalCamp1,
 especimen.campanayas[0].fechasiembra) == 0 ||
perteneceRango(especimen.campanayas[1].fechasiembra, f_finalCamp1,
 especimen.campanayas[2].fechasiembra) == 0 ||
perteneceRango(especimen.campanayas[2].fechasiembra, f_finalCamp2,
 especimen.campanayas[0].fechasiembra) == 0 ||
perteneceRango(especimen.campanayas[2].fechasiembra, f_finalCamp2,
 especimen.campanayas[1].fechasiembra) == 0
```

CÓDIGO 10: CONDICIÓN DE GENERACIÓN TRES CAMPAÑAS CORRECTAS

En este método se usan dos métodos que han aparecido por primera vez generarFechaAleatoriaEntreFechas, perteneceRango.

El primer método `tm generarFechaAleatoriaEntreFechas(struct tm f_inicial, struct tm f_final)` como su nombre indica genera una fecha de forma aleatoria entre f_inical y f_final y la devuelve en forma de struct tm.
El método `int perteneceRango(struct tm f_inicial, struct tm f_final, struct tm f_actual)` devuelve 0 si una fecha (f_actual) pertenece al rango [f_inicial - f_final], sino si, f_actual >= f_inicial devuelve 1 o si f_actual >= f_final devuelve 2.

El manejo de fechas, usado de forma interna en los métodos que lo necesitan se realiza haciendo uso del calendario juliano, para ello se transforma los struct tm a día juliano y se trabaja con los valores enteros devueltos por el método creado para tal fin. Estos métodos pueden verse en el apartado Método para convertir tm a día juliano y Método para convertir de día juliano a tm.

- mutarEspécimen(especímenes,i);
  Este método no va más allá de lo explicado para los factores de mutación en el apartado anterior, Estrategia Evolutiva. Tiene dos partes bien diferenciadas, la primera en la que se mutan los kilos y la segunda en la que se muta el tipo de planta.
  Para la primera parte, se reutiliza el código para la asignación de los kilos iníciales pero pasando como parámetros en esta vez los kilos marcados como peores en el protocolo de la planta que se encuentra mutando.
  Y para la segunda parte se ha utilizado las mismas líneas que se usan para generar un espécimen dependiendo del número de campañas, con las siguientes diferencias, se ha añadido el factor aleatorio para ver si se muta o no el id de la planta y se ha forzado el cambio del id de planta que tiene actualmente. Por último se ha generado un espécimen como se generaría un espécimen de forma normal.
  Los parámetros de este método son el vector de especímenes y el índice del espécimen que se va a mutar, lo que hace que la mutación se realice espécimen a espécimen.

- alteracionesCosecha(especímenes,i);
  La diferencia entre este método y el anterior en la parte para mutar los kilos es que, los kilos que se pasan a la función para alterar los kilos son los considerados como buenos. Este método se usa de igual forma que a la hora de asignar los kilos en el momento de creación de un espécimen con la salvedad de que a la función uniforme se le pasan los kilos que se consideran como buenos. Los parámetros de este método son los mismos que los del anterior.

- evaluarEspécimen(especímenes,i);
  Este método es el encargado de implementar la función de fitness.

```
for(int i=0;i< especimen[numespec].numCampanyas;i++){
       //cálculo del primer día de recogida de cada campaña
       f_inicioRecogida = especimen[numespec].campanayas[i].fechasiembra;
       meses =
(int)(c_plantas[especimen[numespec].campanayas[i].id].t_maduracion ) / 30;
```

```
	dias = (
c_plantas[especimen[numespec].campanayas[i].id].t_maduracion ) % 30;
	dias = (f_inicioRecogida.tm_mday + dias) %
diasmes[(especimen[numespec].campanayas[i].fechasiembra.tm_mon + meses) %
12];
	if ( dias == 0)
		dias = 1;
	 f_inicioRecogida.tm_mday = dias ;
	if(f_inicioRecogida.tm_mon + meses > 11){
		f_inicioRecogida.tm_year += 1;
	}
	f_inicioRecogida.tm_mon = (f_inicioRecogida.tm_mon + meses) % 12;
	//cálculo del beneficio
	int mesRecogida = f_inicioRecogida.tm_mon;
	int diaRecogida = f_inicioRecogida.tm_mday;
	int totalDiasRecogida =
c_plantas[especimen[numespec].campanayas[i].id].t_recogida;
	while (totalDiasRecogida > 0){
		precioCampanyas[i] +=
precios[especimen[numespec].campanayas[i].id][mesRecogida][diaRecogida] ;
		diaRecogida = (diaRecogida + 1) % diasmes[mesRecogida];
		if(diaRecogida == 0)
			mesRecogida = (mesRecogida + 1) % 12;
		totalDiasRecogida--;
	}
	 precioCampanyas[i] = precioCampanyas[i] *
especimen[numespec].campanayas[i].kilos *
(especimen[numespec].metrosInvernadero / 1000);
}//fin for campañas

for(int i=0;i< especimen[numespec].numCampanyas;i++){
	especimen[numespec].fitness += precioCampanyas[i];
}
```

CÓDIGO 11: MÉTODO EVALUAR ESPÉCIMEN, VERSIÓN SECUENCIAL

Este método tiene un bucle que recorre cada campaña y después calcula la fecha cuando se comienza la recogida a partir de la fecha de siembra y el tiempo de maduración necesario, con estos datos ya se tiene el día y el mes en que se inicia la recolecta. Una vez se conoce esta fecha y obtenidos los días de recolecta del protocolo de la planta se realiza un recorrido por la matriz de precios mediante un bucle while, la única aclaración que necesita este método es la forma en la que se recorre la matriz, en donde el día se va calculando según la siguiente formula `diaRecogida = (diaRecogida + 1) % diasmes[mesRecogida];` y para el mes se tiene en cuenta que si el día es 0 el mes hay que incrementarlo en una unidad `mesRecogida = (mesRecogida + 1) % 12;` Aclarar que el vector diasmes[] contiene el total de días de cada mes, es decir, enero 31, febrero 28, marzo 31, …. Los parámetros de este método son los mismos que los del anterior.

- ordenarEspecimenes(especimenes, numUnion);
  Este método es el último de los usados en la estrategia evolutiva y se encuentra fuera del bucle que trabaja con los hijos. Es el encargado de ordenar todos los especímenes que se encuentran en el vector y hacer el intercambio de los especímenes tanto padre como hijos con los padre menos favorables. El método de ordenación es sencillo, se trata del método de la burbuja, en el que se ordenan los especímenes de mayor a

menor valor fitness. Los parámetros de este método son los especímenes y el total de ellos que existen.

Problemas encontrados durante la implementación

Durante la implementación de la EE en C en su versión en secuencial no se ha encontrado ningún problema que interfiera con el correcto desarrollo de la misma sobre C.

Diseño paralelo de la EE

En este apartado se explica cómo se ha portado la EE escrita en C a la programación en CUDA. A grandes rasgos, lo que se ha hecho ha sido sustituir los bucles que operan con los especímenes por kernel CUDA, que como ya se explicó en el apartado Single Instruction, Multiple Thread (SIMT), la estrategia que sigue es operar con los datos de forma aislada entre ellos, en este caso los especímenes y que cada uno sea tratado en un hilo CUDA, puesto que todos los especímenes necesitan el mismo tratamiento y por tanto las mismas instrucciones. Otra característica que se aprovecha de CUDA es el uso de memoria compartida y los vectores de CUDA almacenados en la memoria de texturas, estos se explicarán a lo largo de este punto. La programación con CUDA es similar a C, en lo que respecta al IDE solamente es necesario que el main que ejecuta los kernel se encuentre en un fichero con extensión .cu, por ello al igual que en la implementación secuencial este proyecto se divide en dos ficheros un .h que incluye las funciones tanto de host como de device y el main.cu.

Antes de comenzar a ver el código se muestra el diagrama de flujo seguido por esta versión de la EE.

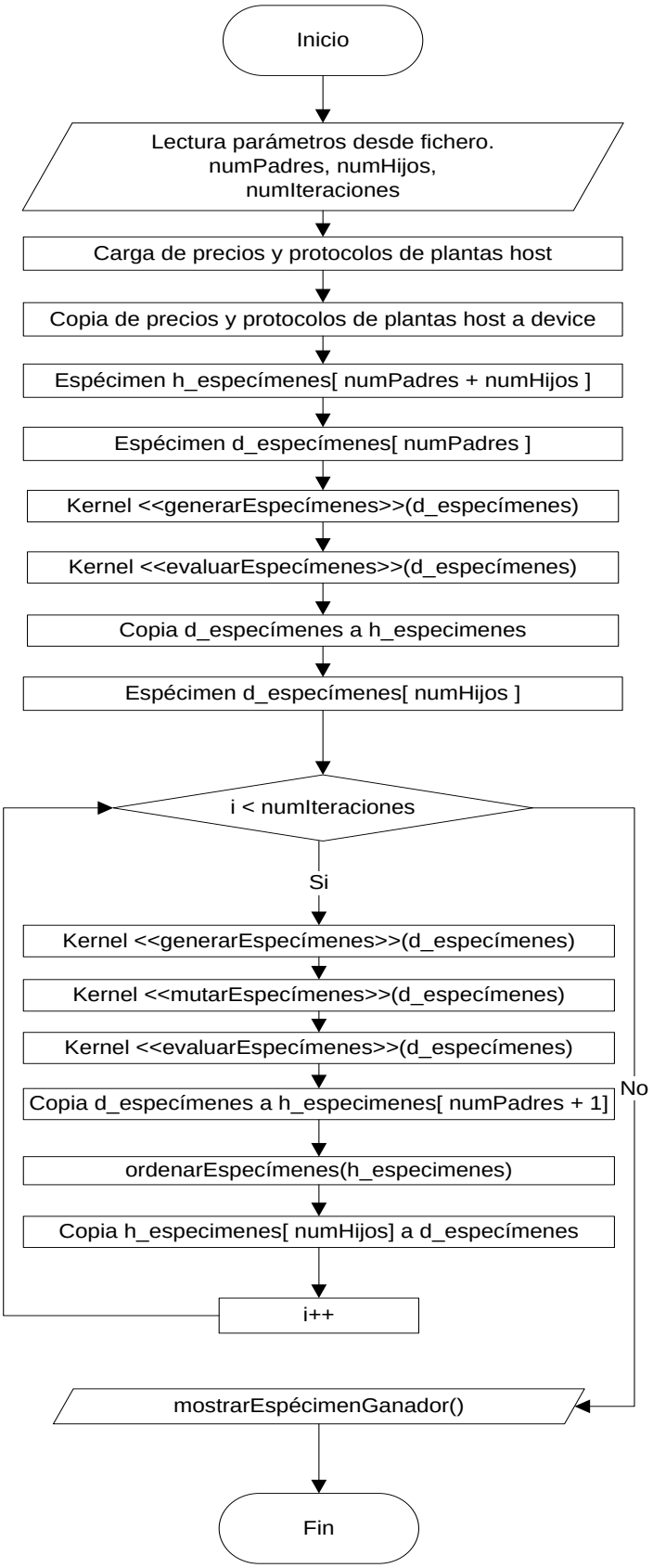

FIGURA 26: DIAGRAMA DE FLUJO VERSIÓN PARALELA

El código del main se detalla a continuación.

Antes de comentar las partes más importantes de la implementación se muestra el fragmento de código que se usa en todas las llamadas a funciones CUDA que se realizan y que permite detectar los errores que ocurran durante la ejecución de estas.

```
if (err != cudaSuccess) {
       fprintf(stderr, "Aclaración del error (error code %s)!\n",
 cudaGetErrorString(err));
}
```

CÓDIGO 12: MANEJO DE ERRORES EN CODIGO CUDA

La función `cudaGetErrorString(err))` nos permite pasar a una cadena de texto el mensaje correspondiente al código de error almacenado la variable err de tipo cudaError_t.

Para comenzar con la EE en CUDA lo primero que se necesita es la declaración de la memoria de texturas para almacenar los precios de los productos, mostrada en la primera línea (ver Código 13) se puede ver que se trata de una textura que almacena datos de tipo float, en tres dimensiones. También se declara la memoria compartida usada para almacenar las características o protocolos de las plantas.

```
texture<float, 3, cudaReadModeElementType> mytexture3D;//almacena los precios
__constant__ CaracteristicasPlanta D_c_plantas[numplantas]; //almacena los
protocolos de cada planta
```

CÓDIGO 13: DECLARACIÓN MEMORIAS CUDA

Lo siguiente es obtener el primer dispositivo CUDA, este se reconoce como el 0, ya que solo existe una tarjeta CUDA en el sistema y obtener sus características/propiedades.

```
int dev = 0 ;
err = cudaSetDevice(dev);
if (err != cudaSuccess) {
       fprintf(stderr, "No existe dispositivo CUDA (error code %s)!\n",
 cudaGetErrorString(err));
       getchar();exit(EXIT_FAILURE);
}
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
```

CÓDIGO 14: PROPIEDADES DE LA TARJETA CUDA

Acto seguido se leen los parámetros del fichero con los que el usuario ha indicado. Estos son usados para establecer el número de hilos necesarios y los bloques con lo que se lanza cada kernel. El número de threads por bloque se obtiene de la variable `deviceProp.maxThreadsPerBlock`, la cual devuelve el número máximo de threads que es capaz de ejecutar por bloque la gráfica que dispone nuestro sistema, la variable divisorThreads está incluida en los parámetros que indica el usuario.

```
//configuración del kernel
int threadsPerBlock = deviceProp.maxThreadsPerBlock / divisorThreads;
int blocksPerGrid = (numespecimenes + threadsPerBlock - 1) / threadsPerBlock;
```

CÓDIGO 15: CONFIGURACIÓN DE LANZAMIENTO KERNEL

Las siguientes instrucciones tienen que ver con la memoria de tipo compartida y constante y el almacenamiento de las características de cada planta en ella. Al igual que en la versión secuencial, los protocolos están escritos en el código de forma estática, solo es necesario copiarlos a la memoria del device, para ello la función cudaMemcpyToSymbol es la encargada. En ella se indican como parámetros la dirección de memoria de destino, la memoria de origen que se quiere copiar y el tamaño de los datos que se quieren copiar.

```
cudaMemcpyToSymbol(D_c_plantas, c_plantas, numplantas *
 sizeof(CaracteristicasPlanta));
```

CÓDIGO 16: FUNCIÓN COPIA DATOS MEMORIA CONSTANTE

Siguiendo con los datos necesarios para la ejecución de la estrategia evolutiva continuamos con los valores de los precios. Estos se han almacenado en una memoria de texturas. Se trata de una memoria, como la anterior, que se crea y define antes de la ejecución de los kernels pero a diferencia de ella, esta si es modificable, aunque para nuestro uso esto no influye ya que solo se usa como memoria de solo lectura, puesto que en ella están almacenados los precios y estos no se modifican a lo largo de la ejecución del programa. Para crear la memoria de texturas primer necesitamos declarar una textura como se vio al comienzo de este punto, además son necesarias, un variable de tipo cudaVector, cudaExtent y cudaChannelFormtDesc, estas van a condicionar la forma en la que se crea y se usa la memoria de texturas. Por último indicar que la configuración en los modos de acceso a la memoria es importante indicar la propiedad normalized a false, para que no normalice las coordenadas y añadir las tres dimensiones que se necesitan con el tipo de acceso `cudaAddressModeWrap`. Con todo esto ya podemos indicar a través de la función cudaBindTextureToVector que el vector creado con propiedades de vector 3D de CUDA va a usar la estructura de almacenamiento de la memoria de textura creada.

```
// creación del vector en la memoria cuda para los precios
cudaVector* cudavector = 0;
cudaExtent volumesize = make_cudaExtent( columnas, filas, numplantas);
cudaChannelFormatDesc channel = cudaCreateChannelDesc<float>();
cudaMalloc3DVector(&cudavector, &channel, volumesize);
//establece los parámetros de copia del vector
cudaMemcpy3DParms copyparms={0};
copyparms.srcPtr=make_cudaPitchedPtr((void*)precios, sizeof(float)* columnas,
columnas , filas );
copyparms.extent=volumesize;
copyparms.dstVector=cudavector;
copyparms.kind=cudaMemcpyHostToDevice;
err = cudaMemcpy3D(&copyparms);
if (err != cudaSuccess) {
       fprintf(stderr, "Error: (error code %s)!\n", cudaGetErrorString(err));
       getchar();exit(EXIT_FAILURE);
}
//establece los modos en los que se accede a la memoria
mytexture3D.normalized = false;
```

```
mytexture3D.filterMode=cudaFilterModePoint;
mytexture3D.addressMode[0]=cudaAddressModeWrap;
mytexture3D.addressMode[1]=cudaAddressModeWrap;
mytexture3D.addressMode[2]=cudaAddressModeWrap;
//asignación de la textura al vector
cudaBindTextureToVector(mytexture3D, cudavector, channel);
```

CÓDIGO 17: DEFINICIÓN Y CONFIGURACIÓN DE MEMORIA DE TEXTURAS

Algunas de las ventajas que tiene esta memoria es que dispone de una cache de nivel 1, permite una interpolación lineal, bilineal y trilineal por hardware y permite la autocorrección de accesos producidos fuera de la memoria. (20) (21)

En las siguientes líneas de código se muestra las reservas de espacio para los vectores de especímenes que se usan tanto en la memoria del host como en la del device y el vector de números aleatorios que se usa como semillas para las operaciones aleatorias dentro de los kernels.

```
//Reserva de espacio y copia a device para los padres
//Reserva de espacio en host
Especimen * h_Especimenes = (Especimen *) malloc ( numUnion *
sizeof(Especimen));
//Reserva de espacio en device
Especimen *d_Especímenes = NULL;
err = cudaMalloc(&d_Especimenes, numespecimenes * sizeof(Especimen) );
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al reservar espacio en device para vector
Especímenes (error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

//vector de semillas para la generación de aleatorios
long * h_Aleatorios = (long *) malloc ( numespecimenes * sizeof(long));
for(int i =0 ; i<numespecimenes;i++ ){
	h_Aleatorios[i]=rand();
}
//reserva y copia en device de las semillas
long *d_Aleatorios = NULL;
err = cudaMalloc(&d_Aleatorios, numespecimenes * sizeof(long) );
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al reservar espacio en device para vector
d_aleatorios (error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
err = cudaMemcpy(d_Aleatorios, h_Aleatorios, (numespecimenes * sizeof(long)),
cudaMemcpyHostToDevice);
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al copiar el vector h_Aleatorios de host a device
(error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
```

CÓDIGO 18: DEFINICIÓN Y COPIA DATOS ESPECÍMENES MU

En este fragmento de código se muestra dos funciones nuevas, cudaMalloc y cudaMemcpy, estas son las equivalentes a malloc y memcpy de C. La función cudaMalloc realiza una reserva de espacio en la memoria global del dispositivo del tamaño indicado en el segundo parámetro y almacena la dirección física de comienzo en el primer parámetro. La segunda función, realiza la copia de datos entre el host y el device o en sentido opuesto, dependiendo del tercer

parámetro, que puede ser cudaMemcpyHostToDevice para el primer caso cudaMemcpyDeviceToHost para el opuesto. Los primeros dos parámetros corresponde al destino y el origen de la copia en ese orden.

Tras todo esto, los datos están listos para ser tratados por los kernels CUDA. Los siguientes fragmentos de código corresponden a la generación y evaluación de los especímenes padres, también conocidos como μ. Existen dos kernels para cada una de las dos tareas que se realiza con los especímenes padre, generación y evaluación. La llamada a un kernel tiene el siguiente formato,

```
nombreKernel <<< numBloques, numThreadBloque >>>(arg1, … , argn)
```

Donde numBloques representa el número de bloques que ocupa el kernel y numThreadBloque el número de hilos que se ejecutan en cada bloque. Mencionar que las llamadas a kernels son asincrónicas y al realizar dos llamadas a dos kernels distintos es necesario utilizar sincronizaciones explícitas. Esta sincronización se lleva a cabo con la llamada a la función cudaDeviceSynchronize cuya función es bloquear la ejecución en el código host hasta que el device haya completado todas las tareas solicitadas precedentes. cudaDeviceSynchronize devuelve un error si una de las tareas precedentes fracasan.

Por último se realiza el copiado de los especímenes generados y evaluados a la memoria del host.

```
//se pasa numespécimens y no numunion para que no trabaje con especímenes que no
existen
generarEspecimenes<<<blocksPerGrid, threadsPerBlock>>>(d_Especimenes,
numespecimenes, time(0), d_Aleatorios);
err = cudaGetLastError();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al lanzar el kernel GenerarEscpecimen (error code
%s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
//sincronizamos para que los hilos que acaben antes esperen a los que más tardan
err = cudaDeviceSynchronize();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al sinconizar hilos 1(error code %s)!\n",
cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

//se pasa numespécimens y no numunion para que no trabaje con especímenes que no
existen
evaluarEspecimenes<<<blocksPerGrid, threadsPerBlock>>>(d_Especimenes,
numespecimenes);
err = cudaGetLastError();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al lanzar el kernel EvaluarEscpecimen (error code
%s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
//sincronizamos para que los hilos que acaben antes esperen a los que más tardan
err = cudaDeviceSynchronize();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al sinconizar hilos 2 (error code %s)!\n",
cudaGetErrorString(err));
```

```
	getchar();exit(EXIT_FAILURE);
}
// copiar resultados al host
err = cudaMemcpy(h_Especímenes, d_Especimenes, numespecimenes *
sizeof(Especimen), cudaMemcpyDeviceToHost);
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al copiar el vector d_Especímenes de device a host
(error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
```

CÓDIGO 19: KERNELS ESPECÍMENES MU

Lo siguiente en la implementación sobre CUDA es cambiar los tamaños del vector con el que se trabaja sobre la memoria del device para almacenar en ellos a los especímenes hijo, el número de bloques que se necesitan y el vector de semillas usado. También se vuelven a crear unas semillas nuevas para las ejecuciones con los especímenes hijo.

```
//modificamos la configuración del kernel para trabajar con los hijos
blocksPerGrid = (numhijos + threadsPerBlock - 1) / threadsPerBlock;

//cambiamos el tamaño del vector de semillas y generamos unas nuevas
h_Aleatorios = (long *) malloc ( numhijos * sizeof(long));
for(int i =0 ; i<numhijos;i++ ){
	h_Aleatorios[i]=rand();
}
//reservamos espacio en device y copiamos las semillas
d_Aleatorios = NULL;
err = cudaMalloc(&d_Aleatorios, numhijos * sizeof(long) );
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al reservar espacio en device para vector
d_aleatorios2 (error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
err = cudaMemcpy(d_Aleatorios, h_Aleatorios, (numhijos * sizeof(long)),
cudaMemcpyHostToDevice);
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al copiar el vector h_Aleatorios de host a device
(error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

//cambiamos el tamaño del vector d_Especímenes para trabajar ahora con los hijos
y los parámetros con los que se lanza el kernel
err = cudaMalloc(&d_Especimenes, numhijos * sizeof(Especimen) );
```

CÓDIGO 20: MODIFICACIÓN VARIABLES PADRES A HIJOS

La siguiente parte de código CUDA corresponde al bucle de la estrategia evolutiva donde se realizan los cálculos con los especímenes hijo. El contenido de este bucle que se puede ver en código 3 para la versión secuencial está implementado en paralelo siguiendo el mismo orden, pero sustituyendo el bucle for interno por kernel CUDA y el método de ordenación por un método que permite ordenar los especímenes haciendo uso de la librería Thrust (22) de CUDA.

Thrust es una biblioteca CUDA que implementa algoritmos paralelos con una interfaz similar a la Standard Template Library (STL) de C++. Thrust proporciona una biblioteca para la programación de GPU que permite hacer el proceso de desarrollo más sencillo. Con Thrust, las

rutinas como ordenamientos paralelos son de 5 a 100 más rápidos comparados con las versiones secuenciales. Además, añade la posibilidad de creación y manejo de vectores y manejo de ellos a través de iteradores. También incluye algoritmos para ordenar, transformar, reducir y otras operaciones para trabajar con vectores.

Para este problema se ha usado el algoritmo de ordenación (23) antes mencionado, haciendo uso de una particularidad que añade, la ordenación por clave. Para ello se ha almacenado en cada clave el fitness de cada espécimen y en su valor correspondiendo el índice donde está originalmente el espécimen. Una vez ordenadas las claves ya se conoce el orden por fitness que deben de tener los especímenes, con ello solo es necesario recorrer el vector de especímenes y ordenarlos de acuerdo a las claves.

```
//bucle EE
do{
//se pasa numhijos y no numunion para que no trabaje con especímenes que no
existen
generarEspecimenes<<<blocksPerGrid, threadsPerBlock>>>(d_Especimenes, numhijos,
time(0), d_Aleatorios);
err = cudaGetLastError();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al lanzar el kernel GenerarEscpecimen (error code
%s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}
err = cudaDeviceSynchronize();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al sinconizar hilos 3 (error code %s)!\n",
cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

mutarAlterarEspecimenesHijos<<<blocksPerGrid, threadsPerBlock>>>(d_Especimenes,
numhijos, time(0), d_Aleatorios);
err = cudaGetLastError();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al lanzar el kernel MutarEscpecimen (error code
%s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

err = cudaDeviceSynchronize();
if (err != cudaSuccess){
    fprintf(stderr, "Fallo al sinconizar hilos 4 (error code %s)!\n",
cudaGetErrorString(err));
    getchar();exit(EXIT_FAILURE);
}

evaluarEspecimenes<<<blocksPerGrid, threadsPerBlock>>>(d_Especimenes, numhijos);
err = cudaGetLastError();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al lanzar el kernel EvaluarEscpecimen (error code
%s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

err = cudaDeviceSynchronize();
if (err != cudaSuccess){
	fprintf(stderr, "Fallo al sinconizar hilos 5 (error code %s)!\n",
cudaGetErrorString(err));
```

```
	getchar();exit(EXIT_FAILURE);
}
//se almacenan los nuevos especímenes a partir de los creados en las enteriores
iteraciones, por ello el destino es h_Especimenes[numespecímenes]
err = cudaMemcpy(&h_Especimenes[numespecímenes], d_Especimenes, numhijos *
sizeof(Espécimen), cudaMemcpyDeviceToHost);
if (err != cudaSuccess) {
	fprintf(stderr, "Fallo al copiar el vector h_Especimes hijos de device a
host(error code %s)!\n", cudaGetErrorString(err));
	getchar();exit(EXIT_FAILURE);
}

double * keys = (double *) malloc ( numUnion * sizeof(double));
int * values = (int *) malloc ( numUnion * sizeof(int));
Especimen * especimenesAux = (Especimen *) malloc (numUnion *
sizeof(Especimen));
for(int i=0;i<numUnion;i++){
	keys[i] = h_Especimenes[i].fitness;
	values[i] = i;
}
thrust::sort_by_key(keys, keys + numUnion , values);
for(int i=0;i<numUnion;i++){
	especimenesAux[(numUnion-i-1)] = h_Especimenes[values[i]];
}
for(int i=0;i<numUnion;i++){
	h_Especimenes[i] = especimenesAux[i];
}
free(keys);free(values);free(especimenesAux);
.
.
.
cont++;
}while(cont < numiteraciones);
// FIN algoritmo evolutivo
```

CÓDIGO 21: BUCLE EE PARALELO

Por último solo queda devolver el espécimen ganador, y una vez terminadas las operaciones con los vectores liberar la memoria que ocupan con las funciones de cuda: cudaFree, cudaFreeVector y la liberación de la memoria de texturas con cudaUnbindTexture. Se liberan los vectores asignados en la memoria del host seguidamente y como última operación con la API de CUDA se ejecuta cudaDeviceReset que de forma explícita destruye y limpia todos los recursos asociados con el dispositivo actual en el proceso actual. Cualquier llamada posterior a la API para este dispositivo hará que se reinicie el dispositivo. Hay que tener en cuenta que esta función restablece el dispositivo de inmediato.

```
printf ("\n:::::::::::::::::::::::::::Ganador:::::::::::::::::::::::::::\n");
mostrarEspecimen(h_Especimenes[0]);

//liberación recursos device
cudaFree(d_Especimenes);
cudaFree(d_Aleatorios);
cudaFreeVector(cudavector);
cudaUnbindTexture(mytexture3D);

free(h_Especimenes);
free(h_Aleatorios);

// Reset the device
err = cudaDeviceReset();
```

```
if (err != cudaSuccess)    {
       fprintf(stderr, "Fallo al liberar el dispositivo! error=%s\n",
cudaGetErrorString(err));
       getchar();exit(EXIT_FAILURE);
}
```

CÓDIGO 22: LIERACIÓN DE RECURSOS PARALELO

Antes de terminar la explicación de cómo se ha realizado la implementación en paralelo de la EE se muestra cada uno de los tres kernels usados.

- Kernel generarEspecímenes

  Este kernel es el encargado de generar especímenes. Este kernel es una traducción a la versión de CUDA del método generarEspecimen de la versión secuencial (ver Código 9). La única diferencia entre el secuencial y el paralelo es que mientras en la primera se mantenía el programa en un bucle hasta que se generase un espécimen bueno, para la versión en paralelo esto no se puede llevar a cabo, ya que se excede el tiempo de uso de la gráfica y el programa finaliza de forma inesperada, por ello, en la versión paralela se ha limitado a dos ejecuciones para generar un espécimen, si en esas dos iteraciones no se crea se devuelve un espécimen vacío con fitness igual a menos uno, para su posterior manejo. Ver la sección Problemas encontrados durante la implementación punto 6 para despejar cualquier duda.

```
__global__ void generarEspecimenes(Especimen * e, int n, unsigned long
seed, long * aleatorios)
{
       int diasmes[12] = { 31,28,31,30,31,30,31 ,31,30,31,30,31 };
       int idx = blockIdx.x * blockDim.x + threadIdx.x;

       if(idx < n) {
              int tst=0;
              int meses; int dias; int findemes; int primerosmes;
              //representan las fechas que finalizan la campaña 0, 1 y 2
respectivamente
              tm f_finalCamp[3];
              double limite = CUDA_gauss(1, 0, 1); // gauss de
media+1*sigma, que es gauss(1) para obtener una probabilidad del 70%
aprox.
              double g =
CUDA_gauss(CUDA_genNumAleatorioFloat(aleatorios[idx]*89,idx), 0, 1);
              if( g > limite ){//genera espc dos campañas
                     //parte mutación
                     e[idx].numCampanyas=2;
                     e[idx].campanayas[0].id = ((int)
(CUDA_genNumAleatorioFloat(aleatorios[idx]*3,idx) * 1000000)) %
(numplantas);
                     e[idx].campanayas[1].id = ((int)
(CUDA_genNumAleatorioFloat(aleatorios[idx]*2,idx) * 1000000)) %
(numplantas);

              }else{//sino de tres campañas
                     e[idx].numCampanyas=3;
                     e[idx].campanayas[0].id = ((int)
(CUDA_genNumAleatorioFloat(aleatorios[idx]*3, idx) * 1000000)) %
(numplantas);
                     e[idx].campanayas[1].id = ((int)
(CUDA_genNumAleatorioFloat(aleatorios[idx]*2, idx) * 1000000)) %
```

```
(numplantas);
                    e[idx].campanayas[2].id = ((int)
(CUDA_genNumAleatorioFloat(aleatorios[idx]*4, idx) * 1000000)) %
(numplantas);
             }

             int maxIteraciones = 2;
             int cntSolape=0;
             do{
                    for(int i=0;i<e[idx].numCampanyas;i++){

                           e[idx].campanayas[i].fechasiembra =
CUDA_generarFechaAleatoriaEntreFechas(
D_c_plantas[e[idx].campanayas[i].id].f_inicialSiembraBueno,
D_c_plantas[e[idx].campanayas[i].id].f_finalSiembraBueno,
aleatorios[idx]*67, idx);
                           e[idx].esperas[i]='X';
                           if(i>0){
                                  //Espera entre campañas
                                  limite = CUDA_gauss(1, 0, 1); // gauss
de media+1*sigma, que es gauss(1) para obtener una probabilidad del 70%
aprox.
                                  g =
CUDA_gauss(CUDA_genNumAleatorioFloat(seed,idx), 0, 1);
                                  if( g > limite ){//aumentar 15 dias, más
probable que espere 15 dias que un mes

       if(e[idx].campanayas[i].fechasiembra.tm_mday + 15 >
diasmes[e[idx].campanayas[i].fechasiembra.tm_mon]){
                                                //aumentar los días y un
mes
                                                findemes =
diasmes[e[idx].campanayas[i].fechasiembra.tm_mon] -
e[idx].campanayas[i].fechasiembra.tm_mday ;
                                                primerosmes = 15 -
findemes;

       e[idx].campanayas[i].fechasiembra.tm_mday = primerosmes;//día
asignado

       if(e[idx].campanayas[i].fechasiembra.tm_mon + 1 > 11){

       e[idx].campanayas[i].fechasiembra.tm_year += 1;
                                                }

       e[idx].campanayas[i].fechasiembra.tm_mon =
(e[idx].campanayas[i].fechasiembra.tm_mon + 1) % 12;
                                          }else{

       e[idx].campanayas[i].fechasiembra.tm_mday += 15;
                                          }
                                          e[idx].esperas[i-1]='0';//espera
15 días
                                  }else{//Sino aumentar un mes
                           if(e[idx].campanayas[i].fechasiembra.tm_mon + 1
> 11){
       e[idx].campanayas[i].fechasiembra.tm_year += 1;
                                          }
              e[idx].campanayas[i].fechasiembra.tm_mon =
(e[idx].campanayas[i].fechasiembra.tm_mon + 1) % 12;
                                          e[idx].esperas[i-1]='1';//espera
un mes
```

```
                                    }
                                }//fin if esperas
                                //calculo fin de campaña
                                f_finalCamp[i] =
e[idx].campanayas[i].fechasiembra;
                                meses =
(int)(D_c_plantas[e[idx].campanayas[i].id].t_maduracion +
D_c_plantas[e[idx].campanayas[i].id].t_recogida) / 30;
                                dias =
(D_c_plantas[e[idx].campanayas[i].id].t_maduracion +
D_c_plantas[e[idx].campanayas[i].id].t_recogida) % 30;
                                dias = (f_finalCamp[i].tm_mday + dias) %
diasmes[(e[idx].campanayas[i].fechasiembra.tm_mon + meses) % 12];
                                if ( dias == 0)
                                       dias = 1;
                                f_finalCamp[i].tm_mday = dias ;
                                if(e[idx].campanayas[i].fechasiembra.tm_mon +
meses > 11){
                                       f_finalCamp[i].tm_year += 1;
                                }
                                f_finalCamp[i].tm_mon = (f_finalCamp[i].tm_mon
+ meses) % 12;
                         }//fin for
 //Comprobación de que el espécimen de dos campañas es correcto
                         if(e[idx].numCampanyas == 2 &&
                    CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[1].fechasiembra) != 0 &&

             CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[0].fechasiembra) != 0){
                                tst=1;
                         }
//Comprobación de que el espécimen de tres campañas es correcto
                         if(e[idx].numCampanyas == 3 &&
             CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[1].fechasiembra) != 0 &&
                    CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[2].fechasiembra) != 0 &&
                    CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[0].fechasiembra) != 0 &&
                    CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[2].fechasiembra) != 0 &&
                    CUDA_perteneceRango(e[idx].campanayas[2].fechasiembra,
f_finalCamp[2], e[idx].campanayas[0].fechasiembra) != 0 &&
                    CUDA_perteneceRango(e[idx].campanayas[2].fechasiembra,
f_finalCamp[2], e[idx].campanayas[1].fechasiembra) != 0 ){
                                tst=1;
                         }
                         cntSolape++;
                    }while(tst==0 && cntSolape < maxIteraciones);

                    //asignación de kilos
                    for(int i=0;i<e[idx].numCampanyas; i++){
                         e[idx].campanayas[i].kilos =
CUDA_uniforme(D_c_plantas[e[idx].campanayas[i].id].kilos[0],
D_c_plantas[e[idx].campanayas[i].id].kilos[1],
CUDA_genNumAleatorioFloat(aleatorios[idx]*54, idx));
                    }

                    //aginación de metros para el invernadero
                    long media = 20000;
                    long var = 3500;
```

```
		g = 0.0;
		double final = 0.0;

		double r2 = ( (int)(
CUDA_genNumAleatorioFloat(aleatorios[idx]*9, idx)*1000000 ) % 20000 ) -
10000;
		g = CUDA_gauss(r2, media, var);
		final = g*1000000;
		if((int)(final) < 9000)
			final += 9000;

		e[idx].metrosInvernadero = (int)(final);
		e[idx].fitness=0.0;//inicializacion del fitness a 0


		//si no se crea un espécimen valido se asigna fitness -1
		if(tst==0){
			e[idx].fitness = -1;
		}
	}//fin if idx
}
```

CÓDIGO 23: KERNEL GENERAR ESPÉCIMEN

- Kernel mutarEspecímenes

  Este kernel se basa en el anterior, al igual que en la versión secuencial en la versión en paralelo solo se muta el id de la planta y sus kilos. La parte en la que se basa este kernel en el anterior es la usada para la mutación de la planta. Antes de intentar mutar un espécimen se realiza una copia de este antes de mutarlo, por si en las dos iteraciones donde se genera uno nuevo con distinto id no se crea uno correcto para poder devolver el espécimen de tal forma que sea válido. Para la mutación de los kilos se utiliza una distribución uniforme al igual que la versión secuencial. Además este kernel también incluye la parte de la EE donde se realizan la alteración a una cosecha alterando sus kilos por posibles virus, o malas condiciones climatológicas.

```
__global__ void mutarAlterarEspecimenesHijos(Especimen * e, int n, unsigned
long seed, long * aleatorios)
{
	int diasmes[12] = { 31,28,31,30,31,30,31 ,31,30,31,30,31 };
	int idx = blockIdx.x * blockDim.x + threadIdx.x;

	int maxIteraciones = 2;
	if(idx < n) {
		//guardamos el original por si no se puede mutar el id y los
datos que se generen en el intento no son válidos.
		Especimen eOriginal;
		for(int i =0;i< e[idx].numCampanyas;i++){
			eOriginal.campanayas[i].fechasiembra =
e[idx].campanayas[i].fechasiembra;
			eOriginal.campanayas[i].id = e[idx].campanayas[i].id;
			eOriginal.campanayas[i].kilos =
e[idx].campanayas[i].kilos;
		}
		for(int i =0 ; i<2 ; i++){
			eOriginal.esperas[i] = e[idx].esperas[i];
		}
		eOriginal.fitness = e[idx].fitness;
eOriginal.metrosInvernadero = e[idx].metrosInvernadero;
eOriginal.numCampanyas = e[idx].numCampanyas;
```

```
		//parte mutación
		double limite;
		double g;
		int meses; int dias; int findemes; int primerosmes; int
idaux;
		//representan las fechas que finalizan la campaña 0, 1 y 2
respectivamente
		tm f_finalCamp[3];
		//mutación del tipo de planta
		limite = CUDA_gauss(1, 0, 1); // gauss de media+1*sigma, que
es gauss(1) para obtener una probabilidad del 70% aprox.
		g = CUDA_gauss(CUDA_genNumAleatorioFloat(aleatorios[idx]*4,
idx), 0, 1);
		int tst=0;
		if( g < limite ){//se muta el id planta
			int cntSolape=0;
			int cntID=0;
			do{
				for(int i=0;i<e[idx].numCampanyas;i++){
					//generación de la primera campaña
					idaux =e[idx].campanayas[i].id;
					 cntID=0;
					do{
						e[idx].campanayas[i].id =
(int)(CUDA_genNumAleatorioFloat(aleatorios[idx]*7, idx)*1000000) %
(numplantas);
						cntID++;
					}while(idaux ==e[idx].campanayas[i].id &&
cntID < maxIteraciones);
					e[idx].campanayas[i].fechasiembra =
CUDA_generarFechaAleatoriaEntreFechas(
D_c_plantas[e[idx].campanayas[i].id].f_inicialSiembraBueno,
D_c_plantas[e[idx].campanayas[i].id].f_finalSiembraBueno,aleatorios[idx]*2,
idx);

					e[idx].esperas[i]='X';
					if(i>0){
						//Espera entre campañas
						limite = CUDA_gauss(1, 0, 1); //
gauss de media+1*sigma, que es gauss(1) para obtener una probabilidad del
70% aprox.
						g =
CUDA_gauss(CUDA_genNumAleatorioFloat(aleatorios[idx]*9, idx), 0, 1);
						if( g > limite ){//aumentar 15
días, más probable que espere 15 días que un mes

	if(e[idx].campanayas[i].fechasiembra.tm_mday + 15 >
diasmes[e[idx].campanayas[i].fechasiembra.tm_mon]){
							//aumentar los días
y un mes
							findemes =
diasmes[e[idx].campanayas[i].fechasiembra.tm_mon] -
e[idx].campanayas[i].fechasiembra.tm_mday ;
							primerosmes = 15 -
findemes;

	e[idx].campanayas[i].fechasiembra.tm_mday = primerosmes;//día
asignado

	if(e[idx].campanayas[i].fechasiembra.tm_mon + 1 > 11){

	e[idx].campanayas[i].fechasiembra.tm_year += 1;
```

```
                                                        }

       e[idx].campanayas[i].fechasiembra.tm_mon =
(e[idx].campanayas[i].fechasiembra.tm_mon + 1) % 12;
                                                        }else{

       e[idx].campanayas[i].fechasiembra.tm_mday += 15;
                                                        }
                                                        e[idx].esperas[i-
1]='0';//espera 15 días
                                                }else{//Sino aumentar un mes

       if(e[idx].campanayas[i].fechasiembra.tm_mon + 1 > 11){

       e[idx].campanayas[i].fechasiembra.tm_year += 1;
                                                        }

       e[idx].campanayas[i].fechasiembra.tm_mon =
(e[idx].campanayas[i].fechasiembra.tm_mon + 1) % 12;
                                                        e[idx].esperas[i-
1]='1';//espera un mes
                                                }

                                        }//fin if esperas

                                        //cálculo fin de campaña
                                        f_finalCamp[i] =
e[idx].campanayas[i].fechasiembra;
                                        meses =
(int)(D_c_plantas[e[idx].campanayas[i].id].t_maduracion +
D_c_plantas[e[idx].campanayas[i].id].t_recogida) / 30;
                                        dias =
(D_c_plantas[e[idx].campanayas[i].id].t_maduracion +
D_c_plantas[e[idx].campanayas[i].id].t_recogida) % 30;
                                        dias = (f_finalCamp[i].tm_mday + dias) %
diasmes[(e[idx].campanayas[i].fechasiembra.tm_mon + meses) % 12];
                                        if ( dias == 0)
                                                dias = 1;
                                        f_finalCamp[i].tm_mday = dias ;

       if(e[idx].campanayas[i].fechasiembra.tm_mon + meses > 11){
                                                f_finalCamp[i].tm_year += 1;
                                        }
                                        f_finalCamp[i].tm_mon =
(f_finalCamp[i].tm_mon + meses) % 12;
                                }//fin for
                                if(e[idx].numCampanyas == 2 &&
                        CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[1].fechasiembra) != 0 &&
                        CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[0].fechasiembra) != 0){
                                        tst=1;
                                }
                                if(e[idx].numCampanyas == 3 &&
                CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[1].fechasiembra) != 0 &&
                CUDA_perteneceRango(e[idx].campanayas[0].fechasiembra,
f_finalCamp[0], e[idx].campanayas[2].fechasiembra) != 0 &&
                CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[0].fechasiembra) != 0 &&
                CUDA_perteneceRango(e[idx].campanayas[1].fechasiembra,
f_finalCamp[1], e[idx].campanayas[2].fechasiembra) != 0 &&
```

```
                CUDA_perteneceRango(e[idx].campanayas[2].fechasiembra,
f_finalCamp[2], e[idx].campanayas[0].fechasiembra) != 0 &&
                CUDA_perteneceRango(e[idx].campanayas[2].fechasiembra,
f_finalCamp[2], e[idx].campanayas[1].fechasiembra) != 0 ){
                                        tst=1;
                                }
                                cntSolape++;
                        }while(tst==0 && cntSolape < maxIteraciones);

                }//fin if mutación id planta

//si no se ha conseguido mutar el espécimen se restaura el original
                if(tst==0){
                        e[idx] = eOriginal;
                }


                //parte mutación para los kilos
                limite = CUDA_gauss(1, 0, 2); // gauss de media+1*sigma, que
es gauss(2sigma) para obtener una probabilidad del 5% aprox.
                g = CUDA_gauss(CUDA_genNumAleatorioFloat(aleatorios[idx]*5,
idx), 0, 2);
                if( g < limite ){//genera de dos campañas
                        for(int i=0;i< e[idx].numCampanyas;i++){
                                e[idx].campanayas[i].kilos =
CUDA_uniforme(D_c_plantas[e[idx].campanayas[i].id].peoreskilos[0],
D_c_plantas[e[idx].campanayas[i].id].peoreskilos[1],
CUDA_genNumAleatorioFloat(aleatorios[idx]*8, idx));
                        }
                }
                //fin parte mutación kilos

//parte alteraciones en la cosecha por virus u otros...
                limite = CUDA_gauss(1, 0, 2); // gauss de media+1*sigma, que
es gauss(2sigma) para obtener una probabilidad del 5% aprox.
                g = CUDA_gauss(CUDA_genNumAleatorioFloat(aleatorios[idx]*3,
idx), 0, 2);
                if( g < limite ){//genera de dos campañas
                        for(int i=0;i<e[idx].numCampanyas;i++){
                                e[idx].campanayas[i].kilos =
CUDA_uniforme(D_c_plantas[e[idx].campanayas[i].id].kilos[0],
D_c_plantas[e[idx].campanayas[i].id].kilos[1],
CUDA_genNumAleatorioFloat(aleatorios[idx]*3, idx));
                        }
                }

        }//fin if idx
}//fin kernel mutar hijos
```

CÓDIGO 24: KERNEL MUTACIÓN Y ALTERACIÓN ESPÉCIMEN

- Kernel evaluarEspécimen

  El último kernel creado es el que evalúa cada espécimen, en esta ocasión se trata de una copia exacta en la parte lógica a la usada en la versión en secuencial.

```
__global__ void evaluarEspecimen(Especimen * e, int n)
{
        int diasmes[12] = { 31,28,31,30,31,30,31 ,31,30,31,30,31 };
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if(idx < n) {
                if(e[idx].fitness != -1 ){//evaluar especímenes válidos
```

```
                        float precioCampanyas[3];
                        for(int i=0;i<e[idx].numCampanyas;i++){
                                precioCampanyas[i]=0;
                        }
                        int meses=0;
                        int dias = 0;
                        struct tm f_inicioRecogida;
                        for(int i=0;i<e[idx].numCampanyas;i++){
                                //cálculo del primer día de recogida de cada
campaña
                                f_inicioRecogida =
e[idx].campanayas[i].fechasiembra;
                                meses =
(int)(D_c_plantas[e[idx].campanayas[i].id].t_maduracion ) / 30;
                                dias = (
D_c_plantas[e[idx].campanayas[i].id].t_maduracion ) % 30;
                                dias = (f_inicioRecogida.tm_mday + dias) %
diasmes[(e[idx].campanayas[i].fechasiembra.tm_mon + meses) % 12];
                                if ( dias == 0)
                                        dias = 1;
                                f_inicioRecogida.tm_mday = dias ;
                                if(f_inicioRecogida.tm_mon + meses > 11){
                                        f_inicioRecogida.tm_year += 1;
                                }
                                f_inicioRecogida.tm_mon =
(f_inicioRecogida.tm_mon + meses) % 12;

                                //cálculo del beneficio
                                int mesRecogida = f_inicioRecogida.tm_mon;
                                int diaRecogida = f_inicioRecogida.tm_mday;

                                int totalDiasRecogida =
D_c_plantas[e[idx].campanayas[i].id].t_recogida;
                                while (totalDiasRecogida > 0){
                                        precioCampanyas[i] += tex3D(mytexture3D,
diaRecogida, mesRecogida, e[idx].campanayas[i].id);
                                        diaRecogida = (diaRecogida + 1) %
diasmes[mesRecogida];
                                        if(diaRecogida == 0)
                                                mesRecogida = (mesRecogida + 1) %
12;
                                        totalDiasRecogida--;
                                }
                                precioCampanyas[i] = precioCampanyas[i] *
e[idx].campanayas[i].kilos * (e[idx].metrosInvernadero / 1000);
                        }//fin for campañas

                        for(int i=0;i<e[idx].numCampanyas;i++){
                                e[idx].fitness += precioCampanyas[i];

                        }
                        //fin parte evaluar
                }//fin if espécimen correcto

        }//fin if idx
}//fin kernel evaluar especímenes
```

CÓDIGO 25: KERNEL EVALUAR ESPÉCIMEN

Por último comentar que el comportamiento de las funciones usadas dentro de los kernels es el mismo al de sus homónimas en la versión en secuencial, la única salvedad es que estas funciones se declaran como se mostró en el apartado Otros aspectos de CUDA.

Problemas encontrados durante la implementación

Los problemas encontrados durante la implementación han sido varios, a continuación se pasan a enunciarlos:

1. El uso de memoria de texturas a la hora de establecer los modos en los que se accede a la memoria.
   Ya que en un principio se usó un direccionamiento de tipo clamp, el cual a la hora de acceder a una coordenada realizaba una sustitución de las mismas a las coordenadas de límites más cercanos. Mientras que el método wrap, el utilizado, simplemente usa una aritmética modular para realizar el acceso a las coordenadas.
2. Almacenamiento de los protocolos de las plantas en memoria de texturas.
   La memoria de texturas no permite el almacenamiento en ella de objetos de tipo struct, solo de tipos de básicos (24), por ello fue necesario el cambio al uso de memoria constante.
3. Generación de números aleatorios.
   Dentro del código CUDA no se permite la utilización de funciones de C estándar, por ello para la generación de números aleatorios no se ha podido realizar con rand() sino con la librería que incorpora la API de CUDA cuRand (25). Esta librería hace más fácil la creación de números dentro del kernel del dispositivo. Los números generados por CURAND son pseudoaleatorios y/o cuasialeatorios. Una secuencia de pseudoaleatorios satisface la mayoría de las propiedades estadísticas de una secuencia de números verdaderamente aleatorios, sin embargo, es generada por un algoritmo determinista. Una secuencia cuasialeatoria de puntos n-dimensionales es determinada por un algoritmo determinista diseñado para llenar el espacio n-dimensional.
4. Error: too many resources requested for launch.
   Este error apareció al incrementar las operaciones que realiza cada kernel, ya que al incrementar las operaciones también se incrementó las variables que necesita para la ejecución. El problema proviene que el número de registros de memoria usados en cada bloque se excedió de los límites de la tarjeta. Este se produjo porque se usaban los 512 hilos de un bloque entonces a cada hilo le pertenece (8192/512) = 16 registros, o lo que es lo mismo 16K (dado en las propiedades de la memoria compartida Shared Per Block: 16 KiB) y en la ejecución de cada kernel el tamaño de la memoria necesitada por hilo es mayor. Por esto se incluye en los parámetros de entrada a la aplicación el divisor de bloques, ya que dependiendo de cada tarjeta el divisor puede ser distinto.
5. Manejo de fechas.
   Como ya se ha comentado en el punto 3 de este apartado CUDA no puede ejecutar funciones de C, por ello ha sido necesario sustituir las funciones mktime y difftime por el manejo de fechas en su día juliano.
6. Error al ejecutar un kernel con un bucle interno: error code the launch timed out and was terminated.

Este problema se produce en los kernels generarEspecimen y mutarEspecimen, al contener un bucle do/while para la generación de forma correcta del espécimen. El problema radica en que la tarjeta gráfica es la misma que la usada para la interfaz del sistema operativo y al entrar en un bucle el código se apodera de la gráfica pero el sistema operativo la requiere para actualizar la interfaz, entonces el sistema operativo detiene la ejecución del programa.

7. División en varios kernels.
   El problema que surgió al escribir un kernels con todas las operaciones que se deben realizar con los especímenes, es que el espacio en memoria para la cantidad de instrucciones era menor al necesitado por restricciones físicas de la tarjeta gráfica. En concreto el error que apareció fue error: uses too much local data. La solución fue dividir cada método en kernel.

Interfaz gráfica

En lo que respecta a la interfaz gráfica, se han creado dos partes, la primera usada para la creación de los ficheros de los precios de cada producto y la segunda que es la parte que interconecta al usuario con el núcleo de la aplicación.

Para la primera parte se tiene un libro en Excel donde cada hoja corresponde a cada mes y dentro de esa hoja están los precios de todos los productos, en la Figura 27 se puede ver un fragmento de la hoja que corresponde al mes de enero.

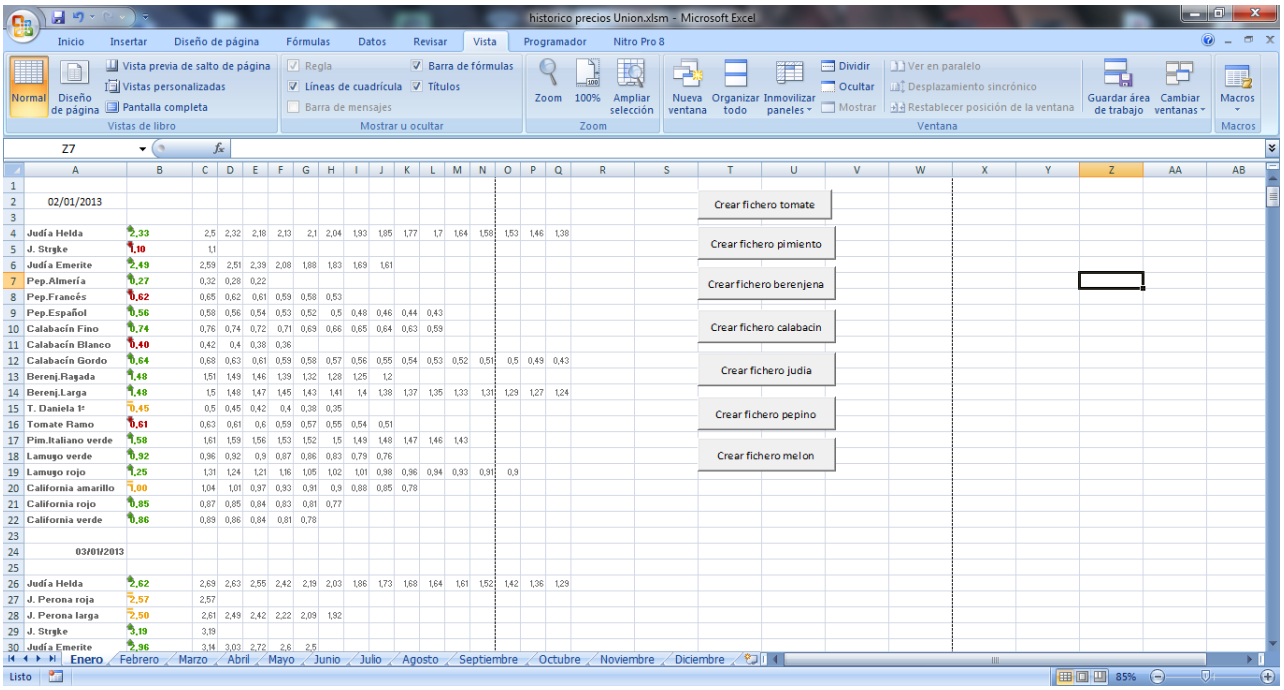

FIGURA 27: INTERFAZ DE CREACIÓN DE DATOS

En esta imagen anterior también se puede ver los botones que crean cada uno de los ficheros de datos de precios de cada planta.

El código que se ejecuta al pulsar cada botón es el mismo para todos ellos, la única diferencia que existe es el tipo de planta que busca. El código para generar el fichero de precios de tomate Daniela es el siguiente,

```
Private Sub CommandButton1_Click()
 Dim strNombreArchivo, strRuta, strArchivoTexto As String
 Dim f As Integer

 'nombre y ruta del archivo de texto
 'lectura del fichero

 'abrimos el archivo para escribir
 f = FreeFile
 Open strArchivoTexto For Output As #f

 Dim salida As String
 Dim mySheet As Worksheet
 Dim valorCelda As String

 For Each mySheet In Worksheets
```

```
 Print #f, mySheet.Name
 For a_counter = 1 To 1000
 Dim celda As String
 celda = "A" & a_counter
 valorCelda = ""
 valorCelda = Worksheets(mySheet.Name).Range(celda)

 Dim comp As Integer
 comp = StrComp(valorCelda, "")
 If comp <> 0 Then 'existe algo en la celda
  Dim encontrado As Integer
  encontrado = InStr(valorCelda, "/")
  If encontrado <> 0 Then 'es una fecha
   salida = Left(valorCelda, encontrado - 1)
   salida = salida & ";"
   salida = salida & Mid(valorCelda, encontrado + 1, 2)
   salida = salida & ";"
  End If

  encontrado = StrComp(valorCelda, "T. Daniela 1ª")
  If encontrado = 0 Then 'es tomte. almacenar precio
   Dim precio As String
   celda = "B" & a_counter
   precio = Worksheets(mySheet.Name).Range(celda)
   precio = Replace(precio, ",", ".")
   salida = salida & precio
   Print #f, salida
  End If

 End If
 Next a_counter
 Next mySheet

 'cerramos el archivo de texto
 Close f
 i = MsgBox("Fichero datos Tomate creado con exito", vbOKOnly,
"Correcto")
End Sub
```

Las primeras líneas de este método pertenecen a la creación del fichero de salida, donde se define el nombre del fichero y la ruta donde se guarda. La mayor parte del método representa a los dos bucles anidados que existen encargados de recorrer todas las celdas de cada hoja del libro. El primero de ellos `For Each mySheet In Worksheets` se usa para recorrer todas las hojas del libro y el segundo para recorrer la primera columna de cada hoja `For a_counter = 1 To 1000.` Como puede verse en la Figura 27, la columna A es la que almacena las fechas y las plantas, así, durante todo este bucle se busca una fecha y se interpreta para almacenarla, después en iteraciones posteriores se busca el producto correspondiente y se obtiene la celda contigua en la columna B.

Como ya se ha dicho el código es el mismo para el resto de productos, salvo por el nombre del fichero que se crea `strNombreArchivo`, y el valor que se busca en cada iteración `StrComp(valorCelda, "T. Daniela 1ª").`

La otra parte de la que se compone la interfaz es la que interacción directamente con el núcleo creado para la estrategia evolutiva, esta se divide en dos partes, la que permite la inserción de datos y la que muestra los resultados obtenidos por la EE.

La parte que inserta los datos y lanza la EE es la que se puede ver en la Figura 28.

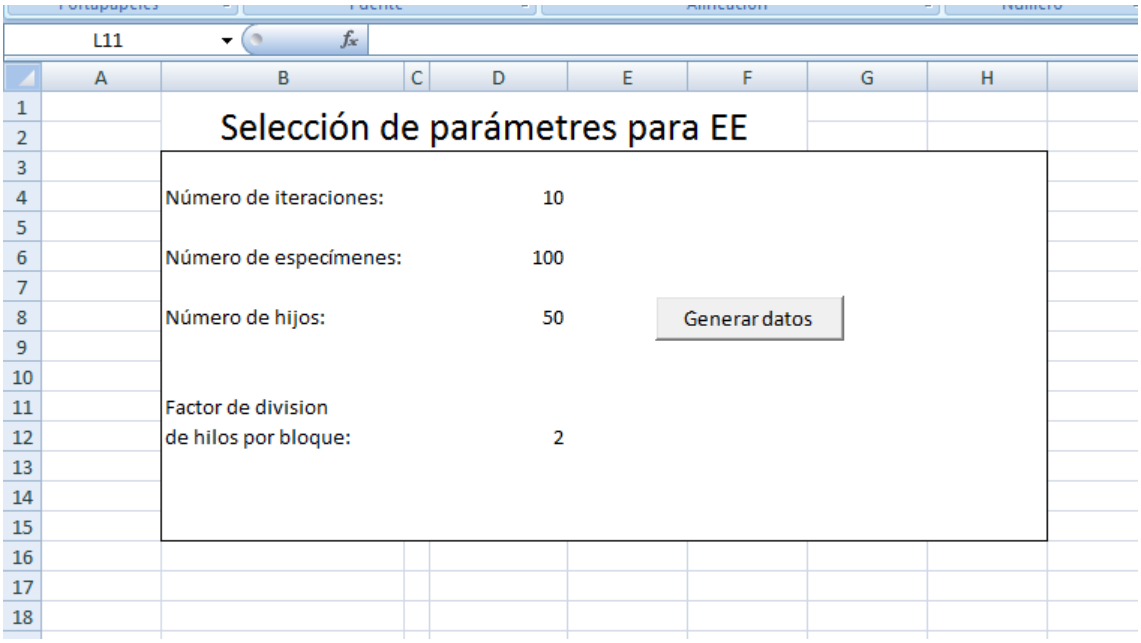

FIGURA 28: INTERFAZ DE INSERCCIÓN DE DATOS

En esta podemos indicar el número de iteraciones que repetirá la EE, el número de especímenes iníciales, conocidos como padres o factor μ, el número de especímenes hijos o facto λ y por último el factor de división, que se usa para dividir el máximo número de thread que un bloque puede lanzar entre ese divisor.

Una vez establecidos estos datos se pulsa sobre el botón generar datos y se ejecuta el código que crea el fichero con la configuración (input.dat) que lee el núcleo de la aplicación y lanza esta.

La parte que muestra los datos obtenidos se encuentra en la hoja 2, esta se muestra vacía hasta que se pulsa el botón de Ver, el cual en su código tiene las instrucciones para leer el fichero que devuelve el núcleo de la aplicación (output.dat) dentro de este se encuentra la evolución que han seguido los especímenes, más concretamente los cinco especímenes ganadores de cada iteración. Esta información se almacena en el fichero de la siguiente forma Iteración | ordenespécimenenIteracion | fitness, a continuación se muestra un fragmento de ese fichero,

```
1|1|24427204.000000
1|2|23329984.000000
1|3|21961096.000000
1|4|20859464.000000
1|5|20463960.000000
2|1|24427204.000000
2|2|23762948.000000
```

```
2|3|23329984.000000
2|4|22989196.000000
2|5|21961096.000000
3|1|27906944.000000
3|2|25270204.000000
3|3|24427204.000000
3|4|23762948.000000
3|5|23329984.000000
4|1|27906944.000000
4|2|27644150.000000
4|3|25270204.000000
4|4|24427204.000000
4|5|23762948.000000
5|1|27906944.000000
5|2|27644150.000000
5|3|26419818.000000
5|4|25270204.000000
5|5|24427204.000000
6|1|27906944.000000
6|2|27644150.000000
6|3|26419818.000000
```

Una vez leídos los datos, estos se almacenan en forma de tabla en la segunda hoja de libro y por último se muestran los resultados en una gráfica. El resultado final se puede ver en la Figura 29.

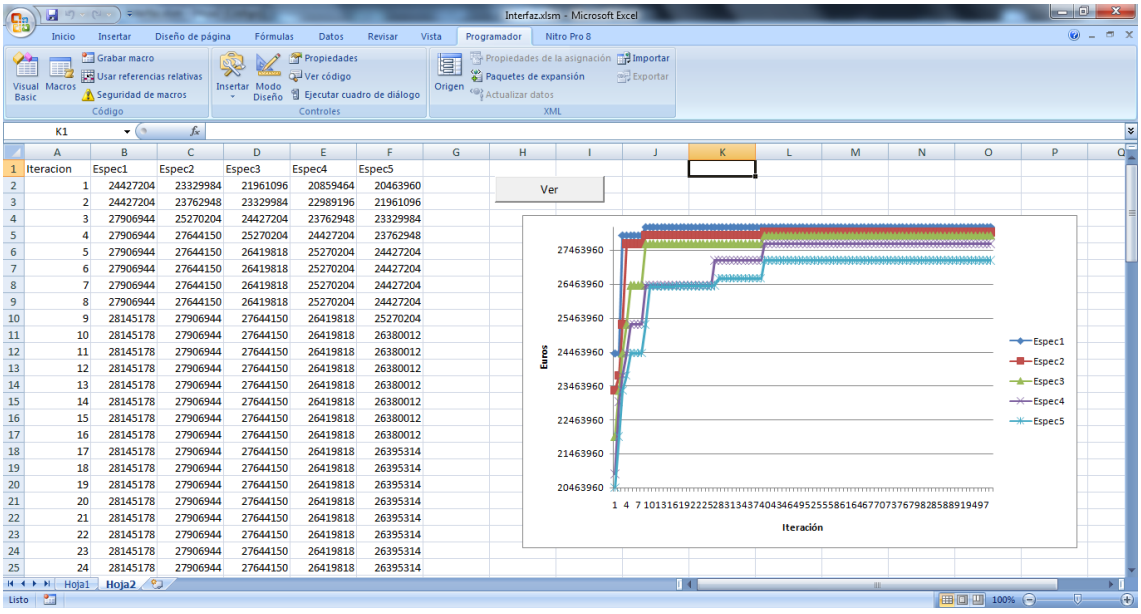

FIGURA 29: INTERFAZ DE DATOS DE SALIDA

El código que realiza todas estas operaciones se muestra a continuación:

```
Private Sub btnVer_Click()
' lectura datos fichero
Dim f, j, fila, cont As Integer
Dim strTexto, col As String
```

```
Dim supercadena() As String

'nombre y ruta del archivo de texto
'lectura del fichero de entrada

'eliminar datos antiguos

'estableces propiedades del botón

'introducir encabezado para la tabla
Cells(1, "A") = "Iteracion"
Cells(1, "B") = "Espec1"
Cells(1, "C") = "Espec2"
Cells(1, "D") = "Espec3"
Cells(1, "E") = "Espec4"
Cells(1, "F") = "Espec5"

'lectura del fichero
cont = 0
f = FreeFile
Open strArchivoTexto For Input As #f

fila = 2
col = "B"
While Not EOF(f)
 Line Input #f, strTexto
 supercadena = Split(strTexto, "|")
 fila = supercadena(0)
 fila = fila + 1
 j = supercadena(1)
 Select Case j
    Case 1  col = "B"
    Case 2  col = "C"
    Case 3  col = "D"
    Case 4  col = "E"
    Case 5  col = "F"
    Case Else  LRegionName = "J"
 End Select
 Cells(fila, col) = supercadena(2)
 cont = cont + 1
Wend
Close f

cont = cont / 5
'colocar el identificador de la iteración
For j = 2 To cont + 1
 Cells(j, "A") = j - 1
Next j

'representación datos en la gráfica
'myChtObj representa el objeto gráfica
Dim myChtObj As ChartObject
Set myChtObj = ActiveSheet.ChartObjects.Add(Left:=450, Width:=500,
Top:=50, Height:=300)

With myChtObj.Chart.Axes(xlValue)
 .HasTitle = True
 .AxisTitle.Text = "Euros"
 .MinimumScale = ActiveSheet.Range("f2")
 .MaximumScale = ActiveSheet.Range("b" & cont)
End With
```

```
'etiqueta eje x
With myChtObj.Chart.Axes(xlCategory)
 .HasTitle = True
 .AxisTitle.Text = "Iteración"
End With

'cada una de las 5 series que representa a los 5 primeros especímenes
With myChtObj.Chart.SeriesCollection.NewSeries
 .Name = ActiveSheet.Range("b1")
 .Values = ActiveSheet.Range("b2:b" & cont)
 .XValues = ActiveSheet.Range("A2:A" & cont)
 End With
With myChtObj.Chart.SeriesCollection.NewSeries
 .Name = ActiveSheet.Range("c1")
 .Values = ActiveSheet.Range("c2:c" & cont)
 .XValues = ActiveSheet.Range("A2:A" & cont)
End With
With myChtObj.Chart.SeriesCollection.NewSeries
 .Name = ActiveSheet.Range("d1")
 .Values = ActiveSheet.Range("d2:d" & cont)
 .XValues = ActiveSheet.Range("A2:A" & cont)
 End With
With myChtObj.Chart.SeriesCollection.NewSeries
 .Name = ActiveSheet.Range("e1")
 .Values = ActiveSheet.Range("e2:e" & cont)
 .XValues = ActiveSheet.Range("A2:A" & cont)
End With
With myChtObj.Chart.SeriesCollection.NewSeries
 .Name = ActiveSheet.Range("f1")
 .Values = ActiveSheet.Range("f2:f" & cont)
 .XValues = ActiveSheet.Range("A2:A" & cont)
End With

myChtObj.Chart.ChartType = xlLineMarkers
End Sub
```

Comparativas

En este apartado se muestran las comparativas que se han realizado de la versión secuencial frente a la versión paralela, ambas ejecutadas en el equipo 2. Además, se realiza una comparativa de las ejecuciones realizadas entre las dos tarjetas gráficas (equipo 1 y equipo 2) enunciadas en el apartado Hardware.

Versión secuencial frente paralela

Para esta prueba se han recopilado diversos datos, los mostrados en la Tabla 2, además es necesario recordar que es el divisor de bloques. Este divisor es un entero pasado como parámetro que devide el resultado devuelto por la variable `maxThreadsPerBlock` obtenida de forma dinámica de las características de la tarjeta gráfica en particular, para este caso el valor devuelto por esta variable es 1024, por tanto el tamaño de los bloques que maneja esta tarjeta gráfica es de 1024 threads por bloque.

#	Divisor de bloque	Num. Threads por bloque	Num. Iteraciones	Población inicial	Num Hijos	Tiempo Paralelo (seg)	Tiempo Secuencial (seg)
1	2	512	100	500	250	17	2
2	4	256	10	500	250	1	0
3	4	256	100	500	250	17	2
4	4	256	500	50	25	47	1
5	4	256	500	500	250	86	8
6	8	128	100	2500	2000	22	21
7	8	128	100	5000	5000	30	80
8	16	64	100	500	250	16	2
9	16	64	100	1000	500	18	3
10	16	64	200	1500	1000	41	17

TABLA 2: COMPARATIVA SECUENCIAL FRENTE PARALELO

Los datos anteriores se muestran en la siguiente gráfica (ver Figura 30).

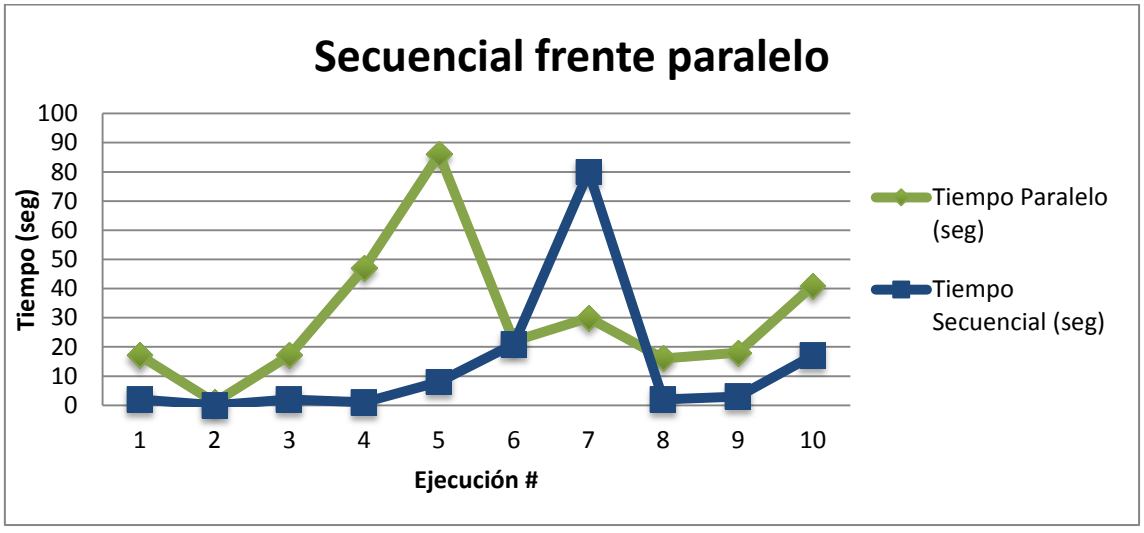

FIGURA 30: GRÁFICA SECUENCIAL FRENTE PARALELO

Con estos datos se puede afirmar que:

- La versión paralela se muestra más lenta cuanto menor es el divisor de bloque y por tanto mayor número de threads se ejecutan por bloque. Esto es debido a que la carga que soporta cada SM es mayor, así como la carga con el trabajo de memorias, lo que conlleva unos tiempos de ejecución mayores. Este efecto se puede ver entre las ejecuciones 3 y 8, en ambas el número de iteraciones, población inicial e hijos es el mismo salvo el tamaño del bloque donde en la 3 es de 256 y en la 8 de 64, esta diferencia hace que la ejecución número 8 termine un segundo antes que la tercera ejecución.
- La versión secuencial se muestra en todas las ejecuciones menos en una (ejecución 7) más rápida que la versión paralela debido a que las operaciones con las que trabaja el procesador del host son básicas a lo cual se le suma un mayor número de operaciones por segundo procesadas y una mayor velocidad de ejecución. Los tiempos de espera que necesita el código en la versión paralela son mayores por los procesos de sincronización necesarios entre los distintos kernels, lo que aumenta el tiempo de ejecución.
- En la ejecución 7 la versión paralela se ha mostrado 2,5X veces más rápida que la versión secuencial. Esto es debido a que la cantidad de operaciones que necesita realizar el procesador del host sobrepasan la cantidad de registros de los que dispone el procesador, así como, la memoria cache viéndose sobrecargadas y llegando a colapsar el procesador. En cambio, esto en la versión paralela no sucede, ya que aun disponiendo de menos memoria y menor velocidad de procesamiento, toda la carga de procesamiento es repartido entre todos los SM lo que permite una menor carga en los procesadores de la tarjeta gráfica.

GeForce 9500 GT frente GeForce GTX 470

En este último apartado de comparativas se enfrentan las ejecuciones antes realizadas lanzadas en las dos tarjetas gráficas de las que se dispone.

Los datos obtenidos son:

#	Divisor de bloque	Num. Threads por bloque	Num. Iteraciones	Población inicial	Num Hijos	GeForce 9500 GT	GeForce GTX 470
1	2	512	100	500	250	-	17
2	4	256	10	500	250	5	1
3	4	256	100	500	250	56	17
4	4	256	500	50	25	157	47
5	4	256	500	500	250	279	86
6	8	128	100	2500	2000	-	22
7	8	128	100	5000	5000	-	30
8	16	64	100	500	250	93	16
9	16	64	100	1000	500	153	18
10	16	64	200	1500	1000	623	41

La representación gráfica de estos datos puede verse en la Figura 31.

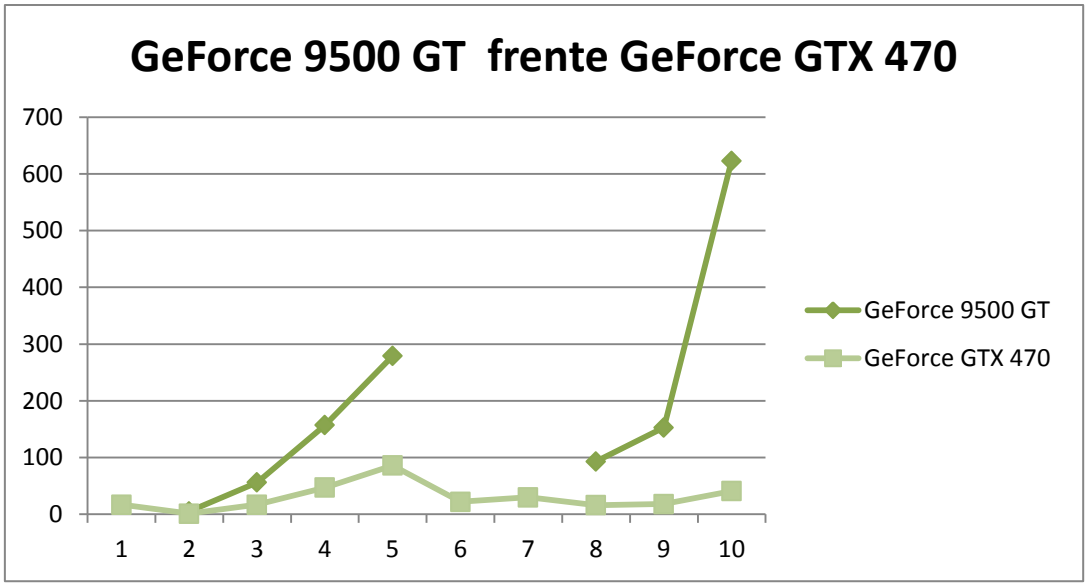

FIGURA 31: COMPARATIVA GEFORCE 9500 GT FRENTE GEFORCE GTX 470

Como se puede ver los resultados obtenidos en la tarjeta GTX 470 son mejores, llegando incluso a no obtener resultados en algunas ejecuciones en la tarjeta 9500GT. Las ejecuciones que han sido fallidas y sus errores en la 9500GT son:

- Ejecución 1: error encontrado numerado como 4 en el apartado Problemas encontrados durante la implementación. Esto es debido a que el tamaño necesario para la ejecución de un kernels por bloques de threads es inferior a la disponible físicamente.
- Ejecución 6: error encontrado numerado como 6 en el apartado Problemas encontrados durante la implementación. Como ya se explicó este error viene dado por el uso del sistema operativo de la tarjeta gráfica y esta encontrarse ocupada por el kernel lanzado por nuestra aplicación.
- Ejecución 7: error encontrado numerado como 6 en el apartado Problemas encontrados durante la implementación.

Conclusiones y trabajo futuros

Conclusiones

Tras el primer estudio, el diseño de la estrategia evolutiva, su implementación y posteriormente comparación, se puede afirmar que el diseño de una estrategia evolutiva y su implementación puede llevarse a cabo en la arquitectura de procesamiento paralelo CUDA, así como que una estrategia evolutiva puede ser beneficiosa para la ayuda a la toma de decisiones en la agricultura.

También se puede afirmar que la paralelización de la EE tiene sentido cuando la cantidad de especímenes con la que se trabaja es superior a 2.500 tanto en la población inicial como en las sucesiones futuras de especímenes hijos, de no ser así, no es necesaria la paralelización de la EE.

Otro dato que se ha corroborado es que la tarjeta gráfica GeForce 9500 GT de menores prestaciones ha obtenido mayores tiempos de ejecución frente a la tarjeta GeForce GTX 470.

Trabajos futuros

Como trabajo futuro se dejan algunos puntos para refinar la estrategia evolutiva y sobre cuestiones de implementación en CUDA. Estas son las siguientes:

- Introducir en los protocolos de las plantas las temperaturas idóneas e inapropiadas para la siembra, maduración y recolecta e ir modificando los kilos de acuerdo a las temperaturas recogidas en la provincia. De igual forma se pueden insertar modificaciones de acuerdo a la orientación de la plantación respecto de la trayectoria del sol o alteraciones en las cosechas producidas por virus o cualquier agente externo de forma más específica, es decir, que cada plantación solo tenga alteraciones por los virus que le afecten.
- Cambiar los especímenes a multiinvernadero, es decir, ahora cada espécimen representa a un invernadero, la idea es que un espécimen represente a varios dos o tres, como ocurre en la realidad. Ya que por normal general cada agricultor dispone de dos o tres invernaderos para cosechar. Con esto también habrá que cambiar la función de fitness puesto que si todos los invernaderos obtienen las mismas plantas que sembrar el precio de los productos finales de estas se ve afectado de forma negativa.
- Integrar todos los kernels que se han creado en uno solo lo que aceleraría el proceso al eliminar las sincronizaciones que se realizan en el código secuencial.

Otros campos de aplicación

Otros sectores donde es aplicable el simulador que se desarrollará son aquellos relacionados con la compra y venta de productos de temporada (calzado, textil, etc.)

También se puede aplicar al campo de la mercadotecnia donde se pueden experimentar con los distintos parámetros de una campaña de publicidad a la hora de ver el momento en el que se lanza, público al que va dirigido o el tiempo que tiene que estar disponible la publicidad.

Anexos

Método para convertir tm a día juliano

Este método se ha creado a partir de la explicación de la web marcada como (26) en la bibliografía.

```
int tmToJuliano(struct tm fecha){
	long JD=0;
	int day = fecha.tm_mday;
	int month = fecha.tm_mon+1;
	int year = fecha.tm_year+1900;

	double x =(14 - month) / 12;
	double a = (int)(x);
	double y = year + 4800 - a;
	double m = month + 12*a - 3;
	JD = day + (int)( (153*m + 2)/5) + 365*y + (int)(y/4) - (int)(y/100) +
(int)(y/400) - 32045;
	return JD;
}
```

Método para convertir de día juliano a tm

Este método se ha creado a partir de la explicación de la web marcada como (27) en la bibliografía.

```
tm julianoToTm(double fecha){
	tm fechaFinal;
	fecha = fecha + 0.5;
	int Z = (int)fecha; //parte entera del día juliano
	double F = fecha - Z; // parte decimal del día juliano
	int A;
	int beta = (int) ((Z - 1867216.25) / 36524.25);
	if(Z < 2299161)
		A = Z;
	else
		A = Z + 1 + beta - (int)(beta/4);
	int B = A + 1524;
	int C = (int) ((B - 122.1) / 365.25);
	int D = (int) (365.25*C);
	int E = (int) ((B - D) / 30.6001);
	int dia = B - D - (int)(30.6001 * E) + F;
	int mes;
	if( E< 14 )
		mes = E-1;
	if (E==14 || E==15)
		mes = E -13;
	int anyo1;
	if (E ==1 || E == 2)
		anyo1 = C - 4715;
	if (E>2)
		anyo1 = C - 4716;
	fechaFinal.tm_mday=dia;
	fechaFinal.tm_mon = mes - 1;
	fechaFinal.tm_year = anyo1-1900+1;
	return fechaFinal;
}
```

Bibliografía

1. Proveedor de invenaderos inteligentes para el control y protecion de sus cutivos. [En línea] http://www.invernaderosinteligentes.com/.

2. **Ferguso, Massey.** Soluciones tecnológicas. [En línea] http://www.masseyferguson.es/documents/tractors/MF_Technology_solutions_ES.pdf.

3. **Ministrerio de agricultura, alimentación y medio ambiente.** Revista Ambienta. *Uso de vehículos aéreos no tripulados (UAV) para la evaluación de la producción agraria.* [En línea] http://www.revistaambienta.es/WebAmbienta/marm/Dinamicas/secciones/articulos/UAV.htm.

4. **Nates, Javier.** La tecnología punta de la agricultura. *Natural.* 2013.

5. **Ferre, Francisco Camacho.** *Técnicas de producción en cultivos protegidos.* s.l. : Cajamar.

6. **Google.** Google Earth engine. [En línea] https://earthengine.google.org/#intro/Amazon.

7. **Cámara de Comercio de, Almería.** Informe económico provincial "Almería en Cifras 2012". [En línea] http://www.camaradealmeria.es/innovacion/jornadas/item/880-informe-econ%C3%B3mico-provincial-almer%C3%ADa-en-cifras-2012.html.

8. **Fundación, Cajamar.** *Análisis de la campaña hortofrutícola de Almería.* 2012.

9. **Universidad de Urugay, Facultad de Ingeniería.** Inteligencia Artificial. *Técnicas de computación evolutiva - Capítulo 2.*

10. **NVIDIA.** CUDA C Programming Guide. [En línea] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

11. **UNED.** *Introducción a los Computadores Paralelos - Capítulo 4.*

12. **NVIDIA.** CUDA Toolkit Documentation. [En línea] http://docs.nvidia.com/cuda/index.html.

13. —. Ejemplo suma de vectores en CUDA. [En línea] http://docs.nvidia.com/cuda/cuda-samples/index.html#vector-addition.

14. **Microsoft.** Introducción a VBA en Office 2010. [En línea] http://msdn.microsoft.com/es-es/library/office/ee814735(v=office.14).aspx.

15. **Sanders, Jason.** *CUDA by Example: An Introduction to General-Purpose GPU Programming.*

16. **Wilt, Nicholas.** *CUDA Handbook: A Comprehensive Guide to GPU Programming.*

17. **Cajamar, Instituto.** *Técnicas de producción en cultivos protegidos.*

18. **fhalmeria.** Pizarra de precios, Histórico. [En línea] http://www.fhalmeria.com/historico_mercados.aspx.

19. **Colegio Oficial de Ingenieros Tecnicos Agricolas de, Almeria.** *Estudio de prevención de riesgos laborales en invernaderos.* 2005.

20. **Hung, Yukai.** CUDA Advanced Memory Usage and Optimization. *Department of Mathematics - Universidad de Taiwan.* [En línea] http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf.

21. **Ansorge, Richard.** CUDA Textures & Image Registration . *University of Cambridge.* [En línea] http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/rea/CUDA_May_09_REA_L6.pdf.

22. **NVIDIA.** Documentación librería Thrust. [En línea] http://docs.nvidia.com/cuda/thrust/index.html.

23. **Nvidia.** Documentación algoritmo de ordenación librería Thrust. [En línea] http://docs.nvidia.com/cuda/thrust/#sorting.

24. **NVIDIA.** Documentación CUDA, memoria de texturas. [En línea] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-memory.

25. —. Documentación de la librecia cuRAND. [En línea] http://docs.nvidia.com/cuda/curand/.

26. **Departamento de, informática.** Universidad de Texas. *Cálculo de día juliano.* [En línea] http://www.cs.utsa.edu/~cs1063/projects/Spring2011/Project1/jdn-explanation.html.

27. **Departamento de, Informática.** Universidad de Texas. *Cálculo día juliano a fecha gregoriana .* [En línea] http://quasar.as.utexas.edu/BillInfo/JulianDatesG.html.

28. **Esteban, Enrique Vicente Bonet.** *Lenguaje C.*