Universidad de córdoba
Escuela Politécnica Superior

PROYECTO FIN DE CARRERA

INGENIERÍA EN INFORMÁTICA

Aceleración de problemas de clasificación con algoritmos de programación genética en GPU

CÓRDOBA, JUNIO 2010

Directores:
Prof. Dra. Amelia Zafra Gómez
Prof. Dr. Sebastián Ventura Soto

Autor:
Alberto Cano Rojas

Prof. Dr. D. Sebastián Ventura Soto

Profesor Titular de la Universidad de Córdoba.

Prof. Dra. Dña. Amelia Zafra Gómez

Profesora Titular de la Universidad de Córdoba.

Informan:

Que el presente Proyecto Fin de Carrera titulado "Aceleración de problemas de clasificación con algoritmos de programación genética en GPU", que constituye la Memoria presentada por Alberto Cano Rojas para aspirar al título de Ingeniero en Informática, ha sido realizado bajo nuestra dirección, reuniendo, a nuestro juicio, las condiciones necesarias exigidas para este tipo de trabajos.

Y para que conste, se expide y firma el presente informe en Córdoba, a Junio de 2010.

Dr. D. Sebastián Ventura Soto **Dra. Dña. Amelia Zafra Gómez**

Agradecimientos

A mis padres por enseñarme todo lo que sé y soy.

A mis compañeros que me han apoyado y con los que he vivido grandes momentos en los últimos 5 años.

A Sebastián y Amelia por su orientación y disponibilidad en todo momento, pero especialmente por haber depositado su confianza en mí para la realización de este proyecto.

"En los momentos de crisis, sólo la imaginación es más importante que el conocimiento".

Albert Einstein

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

La computación evolutiva se engloba dentro de un amplio conjunto de técnicas de resolución de problemas basados en la emulación de los procesos naturales de evolución. La principal aportación de la computación evolutiva a la metodología de resolución de problemas consiste en el uso de mecanismos de selección de soluciones potenciales y de construcción de nuevos candidatos por recombinación de características de otras ya presentes, de modo parecido a como ocurre con la evolución de los organismos ya presentes. No se trata tanto de reproducir ciertos fenómenos que se suceden en la naturaleza sino de aprovechar las ideas genéricas que hay detrás de ellos. En el momento en que se tienen varios candidatos a solución para un problema, surge la necesidad de establecer criterios de calidad y de selección y también la idea de combinar características de buenas soluciones para obtener otras mejores. Dado que fue en el mundo natural donde primeramente se han planteado problemas de este tipo, no tiene nada de extraño el que al aplicar tales ideas en la resolución de problemas científicos y técnicos se obtengan procedimientos bastante parecidos a los que ya se encuentran por la naturaleza tras un largo periodo de adaptación.

Dentro de la computación evolutiva se encuentra la programación genética, en la que los individuos que evolucionan en el sistema son programas informáticos que representan total o parcialmente la solución al problema planteado. Es por tanto una técnica de aprendizaje automático utilizada para optimizar una población de programas de acuerdo a una función de ajuste que evalúa la capacidad de cada programa para llevar a cabo una tarea en cuestión. Sobre cada uno de los individuos se pueden realizar una serie de modificaciones utilizando los operadores genéticos de modo parecido a como ocurre en los organismos naturales. La formulación más conocida de la programación genética es debida a John Koza [12], que representa a los individuos como árboles de instrucciones.

Los algoritmos de programación genética se han aplicado a la resolución de numerosos problemas en minería de datos. La minería de datos [27],[29] se define como la extracción no trivial de información implícita, previamente desconocida y potencialmente útil, a partir de datos. En la actual sociedad de la información, donde cada día a día se multiplica la cantidad de datos almacenados casi de forma exponencial,

la minería de datos es una herramienta fundamental para analizarlos y explotarlos de forma eficaz. Las técnicas de minería de datos permiten obtener conocimiento a partir de las relaciones de los datos y proporcionan a los investigadores y usuarios reglas de clasificación, asociación [31] y predicción. El uso de la programación genética en la resolución de problemas de clasificación es relativamente frecuente [1],[11],[14],[21]. En este caso, se trata de algoritmos de aprendizaje supervisado donde los individuos de la población representan total o parcialmente un clasificador, y la evaluación de estos mide la capacidad de clasificar correctamente a un conjunto de datos (dataset) que ha sido evaluado externamente. El objetivo es que el clasificador generado pueda ser utilizado con éxito en la clasificación de patrones desconocidos. En minería de datos y aprendizaje automático no supervisado, las reglas de asociación [4] se utilizan para descubrir hechos que ocurren en común dentro de un determinado conjunto de datos. Se han investigado ampliamente diversos métodos para aprendizaje de reglas de asociación que han resultado ser muy interesantes para descubrir relaciones entre variables en grandes conjuntos de datos.

En la resolución de este tipo de problemas se debe intentar minimizar una serie de inconvenientes como pueden ser el coste computacional alto que posee, el gran número de datos necesarios para evaluar a los individuos, etc. debido a que su convergencia a la solución puede ser muy lenta en problemas complejos o de grandes dimensiones. Para acelerar su desempeño, su paralelización ha sido objeto de estudio desde múltiples perspectivas aprovechando el desarrollo de nuevo hardware paralelo y de las diferentes características de los dominios del problema en particular. La mayoría de los algoritmos paralelos desarrollados en las últimas dos décadas se centran en la implementación en clusters o arquitecturas masivamente paralelas. Más recientemente, los trabajos sobre paralelización se centran en el uso de unidades de procesamiento gráfico (GPU) que proporcionan un hardware muy rápido y nativamente paralelo por una fracción del coste de un sistema paralelo tradicional. Este paradigma es conocido como computación GPGPU (General Purpose Graphic Processing Unit) [8]. Concretamente, el uso de las GPUs para la resolución de problemas de programación genética se denomina GPGPGPU [9].

El objetivo de este proyecto fin de carrera es analizar el funcionamiento y el coste computacional de los algoritmos de programación genética aplicados a problemas de clasificación con el propósito de acelerar su ejecución en dominios de cualquier complejidad bajo el modelo de la programación masivamente paralela que proporcionan las GPUs. Para ello se realizará un estudio de las fases del algoritmo evolutivo con

la finalidad de comprender su alto coste computacional y un análisis del modelo de programación en GPU que nos permita adquirir el conocimiento suficiente para realizar un diseño eficiente del algoritmo. En la experimentación se buscará aquella configuración de los parámetros del algoritmo que maximicen la calidad de las soluciones generadas minimizando el tiempo de ejecución. Los resultados de la realización de una batería de pruebas con múltiples conjuntos de datos comparan el tiempo de ejecución de los algoritmos implementados con respecto a nuestra propuesta en multihilo y en GPU. Por último, se extraen las conclusiones del proyecto y se enuncian las vías de trabajo futuro.

2. DEFINICIÓN DEL PROBLEMA

En este capítulo se tratará de mostrar detalladamente el problema al que se pretende dar solución con la realización de este proyecto. Para definir el problema, se distingue entre el problema real, que se refiere a la visión que ofrece el usuario, y el problema técnico, que define el problema desde el punto de vista de la ingeniería.

2.1. Definición del problema real

Los algoritmos evolutivos y en particular la programación genética han demostrado ser una buena técnica para resolver problemas de clasificación de conjuntos de datos pero con un elevado coste computacional. La finalidad que se pretende lograr con el presente proyecto fin de carrera es acelerar la ejecución de los algoritmos de clasificación mediante la propuesta de un modelo general de ejecución en GPU de la función de ajuste de los individuos de la población.

La realización de este proyecto implica el estudio de los algoritmos evolutivos y de programación genética en particular, el estudio de los tiempos de cómputo de cada una de la fases del proceso evolutivo, el estudio del funcionamiento del modelo de programación paralelo en GPU con su respectiva jerarquía de hilos y de memoria, el análisis y diseño de las posibles optimizaciones para acelerar al máximo posible la ejecución del algoritmo, la implementación del código para su ejecución en la GPU, la evaluación y comparativa del desempeño de los algoritmos con respecto a diferentes conjuntos de datos y la extracción de conclusiones que permitan abordar nuevos trabajos de manera eficiente.

2.2. Definición del problema técnico

Para llevar a cabo la identificación del problema técnico se empleará una técnica de ingeniería denominada PDS (Product Design Specification). Esta técnica permite realizar un análisis de los principales condicionantes técnicos del problema mediante la respuesta de una serie de cuestiones básicas.

2.2.1. Funcionamiento

Los algoritmos de programación genética para el aprendizaje de reglas de clasificación deben extraer conocimiento de un conjunto de datos (dataset) y proponer una serie de reglas que ayuden a clasificar nuevos datos. Para ello, se evoluciona una población de individuos de reglas de clasificación que se evaluará en la GPU para maximizar la paralelización del proceso.

2.2.2. Entorno

El entorno se define como el conjunto de aspectos o propiedades que rodean al problema y que aun presentándose de manera externa al mismo, pueden incluir en el planteamiento de una solución, puesto que pueden afectar al sistema software desarrollado para tal fin.

En el análisis del entorno del proyecto que se pretende desarrollar se tendrán en cuenta los siguientes puntos de vista: entorno de programación, entorno software, entorno hardware y entorno de usuario.

- Entorno de programación: la presente aplicación será desarrollada en el lenguaje de programación Java y C, haciendo uso del entorno de desarrollo Eclipse que incluye el J2SDK 1.6 (Java 2 Standard Edition Development Kit) y del framework NVIDIA CUDA 3.0.
- Entorno software: para el correcto funcionamiento de la aplicación serán necesarios los siguientes componentes software:
  - Eclipse Integrated Development Environment.
  - JCLEC 4: software de computación evolutiva desarrollado con fines docentes e investigadores por los miembros del Grupo de Investigación KDIS.
  - NVIDIA CUDA SDK 2.3 o posterior.
  - NVIDIA CUDA Toolkit 2.3 o posterior.
  - NVIDIA CUDA Driver 195.36.18 o posterior.
- Entorno hardware: también conocido como entorno físico o de trabajo, hace referencia a las características del sistema informático en el que se ejecutará la

aplicación, así como el ambiente que lo rodea. El sistema a desarrollar se instalará en un ordenador personal, requiriendo de una tarjeta gráfica NVIDIA serie 8000 o posterior.

- Entorno de usuario: el sistema software a desarrollar deberá ser lo más intuitivo posible para permitir a los usuarios una manipulación simple de los datos. Los usuarios finales deberán tener conocimientos básicos sobre el tema para poner en marcha el programa y comprender la información y los resultados que se obtendrán.

2.2.3. Vida esperada

La vida esperada de un producto software puede definirse como el tiempo estimado durante el cual puede realizarse una aplicación útil del mismo. Esta estimación es difícil de realizar al influir en la misma numerosos factores. Al tratarse de un producto basado en estudios de investigación, se estima que la vida esperada de este producto no sea muy larga. Sin embargo, los algoritmos que se implementan en este proyecto pueden ser objeto de renovación o actualización. La implementación propuesta garantiza la escalabilidad del rendimiento de los algoritmos en un plazo no inferior a 5 años si la tendencia actual del incremento del número de procesadores de las GPUs se mantiene. Por tanto, es lógico pensar que el presente proyecto pueda ser utilizado como punto de partida para trabajos posteriores.

2.2.4. Ciclo de mantenimiento

El ciclo de mantenimiento se identifica con el conjunto de modificaciones que resulta necesario realizar para que una determinada aplicación pueda hacer frente a diferentes circunstancias que puedan surgir, o a nuevas exigencias procedentes del usuario final o del propio sistema.

El mantenimiento de la aplicación deberán llevarlo a cabo programadores informáticos atendiendo a las necesidades generadas, pudiendo ser éstas de tres tipos:

- Perfeccionamiento: conjunto de funciones cuyo objetivo consiste en una mejora de la funcionalidad de la aplicación o agregación de funcionalidades requeridas para la evolución de las redes de distribución. Estas mejoras pueden consistir en aumentar la eficiencia de la aplicación, de alguna de las funciones que realiza,

la facilidad de mantenimiento de la aplicación para posibles cambios futuros, etc.

- Adaptación: conjunto de actividades realizadas para adaptar la aplicación al entorno tecnológico, pudiéndose producir cambios, por ejemplo, en el formato de los datos o en los ficheros que utiliza la aplicación para alguno de sus fines.
- Corrección: conjunto de actividades necesarias para llevar a cabo las correcciones de errores no detectados en la aplicación hasta el momento.

2.2.5. Competencia

Pese a que este producto tiene únicamente fines docentes e investigadores y por lo tanto, no entrará en un mercado comercial, se puede tener en cuenta algunas aplicaciones de Data Mining relacionadas:

- Weka: software para aprendizaje automático y minería de datos escrito en Java y desarrollado en la Universidad de Waikato.
- Keel: herramienta de software para evaluar algoritmos evolutivos para problemas tales como regresión, clasificación, clustering, etc. Keel posee una gran colección de algoritmos clásicos de extracción de conocimiento, técnicas de preprocesamiento, algoritmos de aprendizaje basados en inteligencia computacional, etc.

2.2.6. Aspecto externo

La apariencia externa de una aplicación hace referencia no solamente al aspecto visual que de la misma tiene el usuario durante su ejecución, sino que también es necesario considerar la presentación física del sistema, los mecanismos de instalación proporcionados junto al mismo y los manuales que pueden acompañarlo.

Se van a considerar los siguientes tres aspectos:

- Interfaz de usuario: la interfaz de usuario de la aplicación que se pretende desarrollar, debe facilitar el uso de la aplicación por parte del usuario, siendo, para ello, lo más intuitiva posible. Hay que destacar que la aplicación a desarrollar, al encontrarse en el ámbito de la investigación, no tendrá una interfaz clara, sino que ésta será la del framework JCLEC.

- Formato de almacenamiento: el dispositivo de almacenamiento en que será entregada es un CD-ROM debido a la portabilidad, bajo coste, capacidad de almacenamiento, resistencia, seguridad y su uso generalizado entre los usuarios. En el CD-ROM también se incluirá parte del software necesario para que la aplicación funcione correctamente, así como los manuales generados en la realización de la misma.

- Documentación: la documentación que se incluirá en el CD-ROM de la aplicación, incluirá el Manual Técnico, en el que se engloba la información general disponible acerca del proceso de análisis, diseño, implementación y prueba del sistema; y el Manual de Código, en el que se incluirá información detallada acerca de la implementación del producto final obtenido.

2.2.7. Estandarización

La aplicación se desarrollará utilizando el lenguaje de programación Java y haciendo uso del framework JCLEC. Todas las librerías que incluye el framework JCLEC están bastante extendidas y no representan un problema en el aspecto de la estandarización.

Cuando se hace uso del lenguaje de programación Java, es recomendable emplear una serie de normas relativas al estilo de codificación. A pesar de que las reglas proporcionadas por el lenguaje son muy amplias y otorgan al programador una gran libertad, resulta habitual respetar ciertas consideraciones que facilitan la lectura y el mantenimiento de los programas desarrollados. Se trata de un estándar no oficial usado de manera habitual en todos los componentes de los módulos desarrollados por Sun MicroSystems.

Las principales instrucciones que se emplearán en el desarrollo de la presente aplicación son las siguientes:

- Los nombres de los paquetes deberán ser lo más breve posible.

- Los nombre de las clases e interfaces comenzará siempre por una letra mayúscula. En el caso de las interfaces, deberán incluir antes del nombre la letra “I”.

- Los nombres de objetos, métodos y variables miembro, así como los nombres de las variables locales de los métodos, comenzarán siempre por una letra minúscula.

- Cuando un nombre conste de varias palabras, se colocará una a continuación de la otra, sin ningún carácter separador entre ellas. La primera letra de cada palabra, excepto la primera que dependerá del elemento al que corresponda, se escribirá en mayúscula.
- El código fuente deberá estar correctamente documentado de forma que cualquier programador informático pueda entenderlo y utilizarlo de cara a futuras mejoras.

2.2.8. Calidad y fiabilidad

La calidad y la fiabilidad son dos factores muy importantes que hay que tener en cuenta en el desarrollo de cualquier aplicación, puesto que es necesario proporcionar al usuario final garantías que le permitan depositar su confianza en el producto, ya que, en caso contrario, podrían surgir reacciones adversas a su utilización.

La calidad de cualquier aplicación se asocia al hecho de que durante su ejecución no se produzcan errores que induzcan a su terminación irregular. La fiabilidad, por su parte, hace referencia a la capacidad del sistema para proporcionar datos reales, asegurando que las acciones realizadas durante el procesamiento resulten correctas y se lleven a cabo de manera óptima.

Tanto en la aplicación a desarrollar como en cualquier sistema software, la fiabilidad será un factor importante, ya que los datos o informes que obtiene el usuario deberán ser totalmente correctos, por lo que se realizarán pruebas que intenten minimizar el número de errores producidos. Destacar que, a priori, los únicos errores que debería cometer el sistema serán aquellos derivados de un uso incorrecto del mismo.

2.2.9. Programación de tareas

El programa de tareas se define como el conjunto de etapas y actividades que constituyen el proceso de desarrollo de una aplicación. A continuación, se describirán las diferentes etapas en las que se puede organizar el proceso de desarrollo de esta herramienta software:

1. Estudio teórico de diversos conceptos y conocimientos generales sobre Algoritmos Evolutivos y Programación Genética.

2. Estudio teórico del modelo de programación CUDA.
3. Estudio de antecedentes sobre paralelización de Algoritmos Evolutivos y trabajos en GPU.
4. Estudio de los mecanismos necesarios para implementación de algoritmos en la librería JCLEC.
5. Implementación de nuevos métodos para acelerar los algoritmos de clasificación basados en Programación Genética en JCLEC.
6. Realización de pruebas de la aceleración de los nuevos métodos sobre bases datos específicas de clasificación.
7. Análisis de los resultados de estas pruebas.
8. Obtención de conclusiones.
9. Redacción de la documentación del proyecto.

2.2.10. Pruebas

Las pruebas se definen como el conjunto de acciones y datos que son utilizados para poder depurar la aplicación desarrollada, además del medio para demostrar la funcionalidad de la misma y su utilidad. Durante la fase de implementación, el software será sometido a diversas pruebas para garantizar la corrección del software. En la fase de experimentación y resultados se someterá a los algoritmos a pruebas de ejecución para estudiar su funcionamiento de acuerdo a los objetivos del proyecto.

2.2.11. Seguridad

La seguridad de la herramienta que se pretende desarrollar deberá consistir en garantizar que durante su ejecución, no se realice ninguna actividad incorrecta ajena a su propia funcionalidad. Además, será necesario asegurar la protección de los dispositivos de almacenamiento propios del usuario, aunque para controlar este aspecto no será necesario tomar ninguna medida adicional al considerarse suficientes los mecanismos de seguridad proporcionados por el lenguaje Java.

Por otro lado, destacar que no será necesario establecer ninguna medida de seguridad orientada a proteger la confidencialidad de los datos, ya que toda la información

manejada por la aplicación será de carácter público. La información con la que se trabaja y los resultados generados por el sistema carecen de privacidad. Por último, decir que no será necesaria ninguna protección contra copia.

2.2.12. Licencia

Como parte del desarrollo de JCLEC, el proyecto se encuentra disponible bajo la licencia GNU General Public License (GPL), por lo que puede ser usado, modificado y distribuido libremente por cualquier usuario.

3. OBJETIVOS

En el presente capítulo se expondrán todos los objetivos funcionales que se pretenden alcanzar con el desarrollo de este proyecto.

El propósito de este proyecto es acelerar la ejecución de los modelos de programación genética para resolver problemas de clasificación mediante la implementación en GPU de aquellas partes del algoritmo evolutivo que sean costosas y puedan ser objeto de paralelización. Habiéndose demostrado que la fase de evaluación es la que requiere la mayor parte del tiempo de computación, se propone paralelizar esta fase para ser utilizada por los diferentes algoritmos. Por lo tanto, el objeto de este proyecto es diseñar un evaluador paralelo de reglas utilizando la arquitectura de las GPUs para acelerar la ejecución de un clasificador basado en programación genética.

El proyecto se marca los siguientes objetivos:

1. Profundizar en el conocimiento de la arquitectura de las unidades de procesamiento gráfico.

2. Profundizar en el conocimiento de los algoritmos evolutivos, concretamente en los de programación genética.

3. Paralelizar aquellas partes del algoritmo de programación genética que resulten más beneficiosas.

4. Acelerar al máximo la ejecución del algoritmo, minimizando su tiempo de ejecución.

5. Aumentar el límite del tamaño de los conjuntos de datos sobre los que extraer información y clasificar sus patrones.

6. Realizar un estudio comparativo entre diferentes algoritmos y conjuntos de datos de variado tamaño.

7. Abrir las puertas al desarrollo de nuevos algoritmos nativamente paralelos en GPU.

4. ANTECEDENTES

En este capítulo mencionaremos toda aquella información previa que sirve como base para la ejecución del proyecto. Es por ello, que en el presente capítulo se concretará el punto de partida del proyecto.

4.1. JCLEC

El uso de algoritmos de computación evolutiva para la resolución de problemas es una práctica cada vez más extendida. Pese a la potencia de dichos algoritmos, el desarrollo de una aplicación basada en computación evolutiva para la resolución de un problema concreto requiere una cierta experiencia en programación, así como un tiempo y esfuerzo considerables. Este tedioso trabajo necesita ser realizado antes de que los usuarios puedan comenzar la propia resolución del problema planteado. Una forma de acometer esta tarea es el uso de un sistema software de computación evolutiva. Estos sistemas, de propósito general, disponen de una multitud de componentes que pueden ser utilizados por el usuario para construir sus propios algoritmos, acelerando considerablemente el desarrollo de aplicaciones. Muchos disponen de bibliotecas de algoritmos ya implementados que pueden ser directamente utilizados por el usuario y de sistemas de visualización y monitorización que le permiten evaluar la calidad de los mismos para resolver un problema determinado.

JCLEC [24] (Java Class Library for Evolutionary Computation) es un sistema software de computación evolutiva desarrollado por el grupo de investigación Knowledge Discovery and Intelligent Systems (KDIS) de la Universidad de Córdoba. Este sistema forma parte del Proyecto KEEL, en el que participan universidades de toda España en un intento de ofrecer una alternativa factible a Weka. Entre las características principales de JCLEC podemos citar su facilidad de extensión, la gran variedad de algoritmos y paradigmas de cómputo que contiene, la posibilidad de configurar desde un fichero las ejecuciones y de personalizar las salidas obtenidas y la realización de baterías de ejecuciones.

JCLEC es un sistema software para el desarrollo de aplicaciones de computación evolutiva. Este sistema, desarrollado en lenguaje Java, aprovecha las ventajas de la programación orientada a objetos para confeccionar un conjunto de clases que permitan, mediante el desarrollo de una cantidad mínima de código, resolver problemas de muy distinta naturaleza mediante algoritmos evolutivos.

El sistema puede considerarse dividido en dos módulos: el núcleo, que contiene las definiciones de tipos abstractos y sus implementaciones; y el runtime environment, una aplicación de consola que permite la ejecución de baterías de algoritmos en cola.

En la Figura 4.1 se resume la arquitectura que presenta el sistema JCLEC.

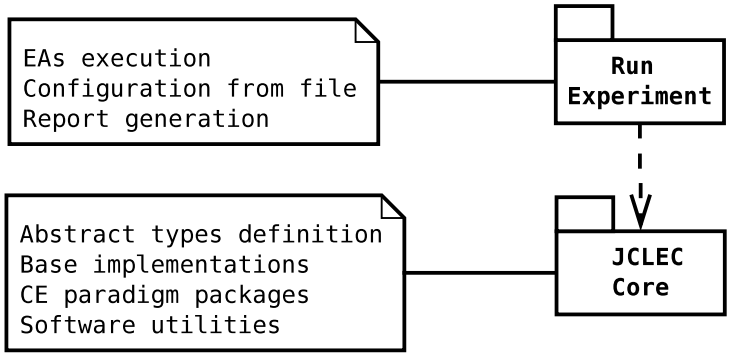

Figura 4.1: Arquitectura de JCLEC.

En primer lugar tenemos el núcleo del sistema, en el que se definen los tipos abstractos, sus implementaciones base y una serie de módulos software que proporcionan toda la funcionalidad del núcleo de la aplicación.

Sobre esta primera capa, se define el sistema de ejecución de trabajos. En este sistema, un trabajo es una secuencia de ejecuciones de algoritmos evolutivos, definidos mediante un fichero de configuración. El sistema recibe como entrada este fichero, y devuelve como resultado uno o varios informes de resultados de las ejecuciones realizadas.

El sistema también puede emplearse para llevar a cabo la definición de secuencias de ejecuciones, que serán ejecutadas por el módulo de ejecución de trabajos, e incluir código desarrollado por el usuario, siempre que éste haya sido desarrollado atendiendo a la jerarquía definida en el núcleo del sistema.

El núcleo del sistema JCLEC define los tipos de datos básicos que definen la funcionalidad del framework que se muestran en la Figura 4.2.

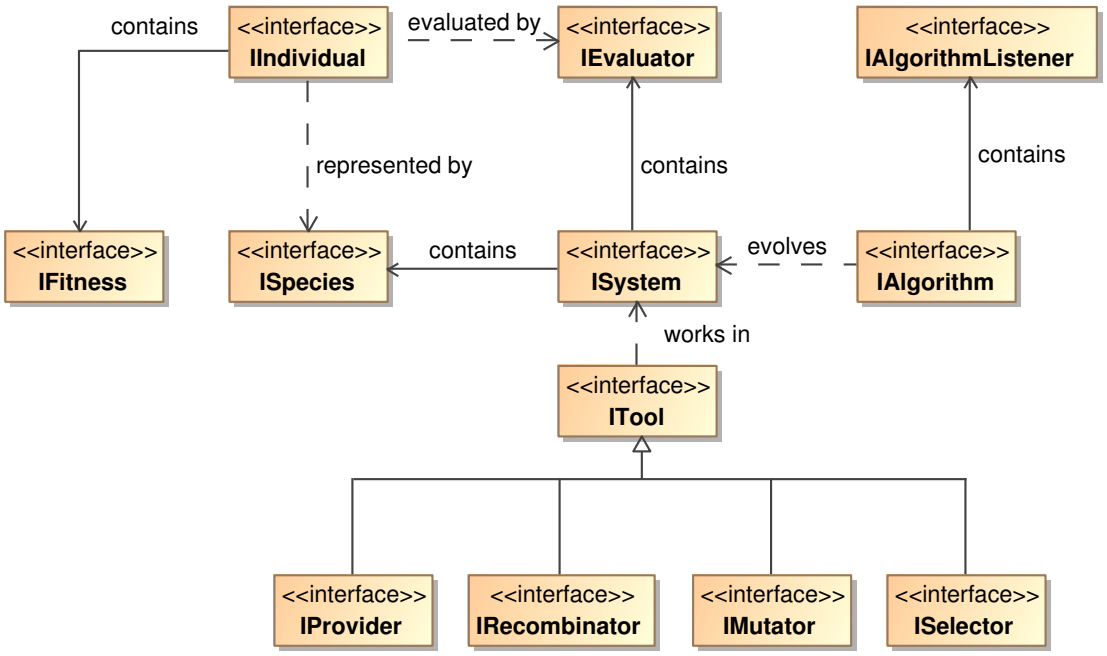

Figura 4.2: Jerarquía de clases de JCLEC.

Como puede comprobarse, existen tres tipos de interfaces bien diferenciados:

- Los relacionados con el sistema que está en evolución (interfaces IIndividual, IFitness, ISpecies, IEvaluator e ISystem),
- Los que se corresponden con acciones realizadas durante la evolución (interfaces IProvider, ISelector, IRecombinator e IMutator pertenecientes a ITool)
- El propio algoritmo evolutivo (interfaz IAlgorithm e IAlgorithmListener).

En el capítulo de diseño del sistema se detallarán las interfaces y clases empleadas para el desarrollo de este proyecto.

4.2. Modelo de programación CUDA

CUDA (Compute Unified Device Architecture) es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU para proporcionar un incremento extraordinario del rendimiento del sistema.

4.2.1. Arquitectura de la GPU

La GPU es un procesador dedicado que tradicionalmente se ha dedicado exclusivamente al renderizado de gráficos en videojuegos o aplicaciones 3D interactivas.

Hoy en día superan en gran medida el rendimiento de una CPU en operaciones aritméticas y en ancho de banda en transferencias de memoria. La Figura 4.3 representa la evolución del potencial de cálculo en FLOPs (Floating-Point Operations per Second) de las CPUs y las GPUs a lo largo de los últimos años. Su gran potencia se debe a la alta especialización en operaciones de cálculo con valores en punto flotante, predominantes en los gráficos 3D. Conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo completamente independientes y dedicadas al procesamiento de los vértices y píxeles de las imágenes.

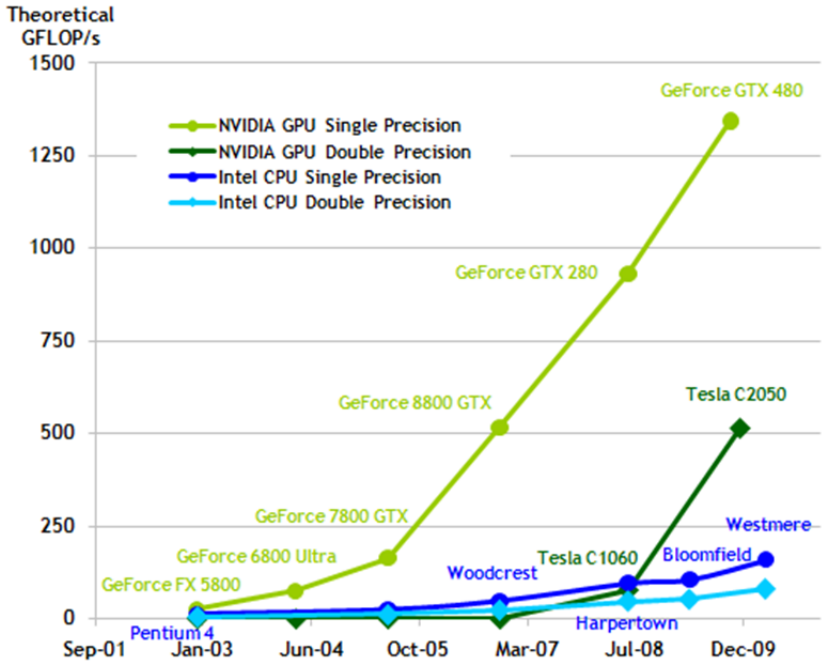

Figura 4.3: Evolución de las FLOPs de las CPUs y las GPUs.

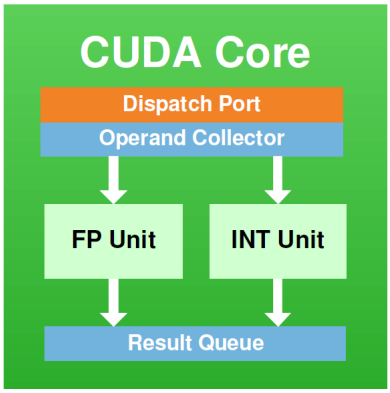

Figura 4.4: Shader o núcleo.

Desde 2006, las arquitecturas unifican el procesamiento en unidades versátiles denominadas shaders que poseen una unidad de enteros y otra de punto flotante. Estos shaders o núcleos, representados en la Figura 4.4, se agrupan en una unidad conocida como multiprocesador (Streaming Multiprocessor, SM), ilustrado en la Figura 4.5, que además contiene algo de memoria compartida entre los núcleos y que gestiona la planificación y ejecución de los hilos en los núcleos de su multiprocesador. Una GPU como muestra la Figura 4.6 se compone de varios grupos de multiprocesadores interconectados a una memoria global GDDR (Graphics Double Data Rate). El número de núcleos por multiprocesador depende de la generación de la gráfica, y el número de multiprocesadores en la GPU determina su potencia máxima y su modelo dentro de la generación.

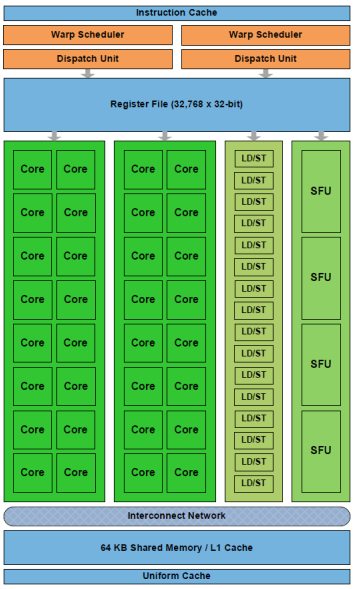

Figura 4.5: Streaming Multiprocessor.

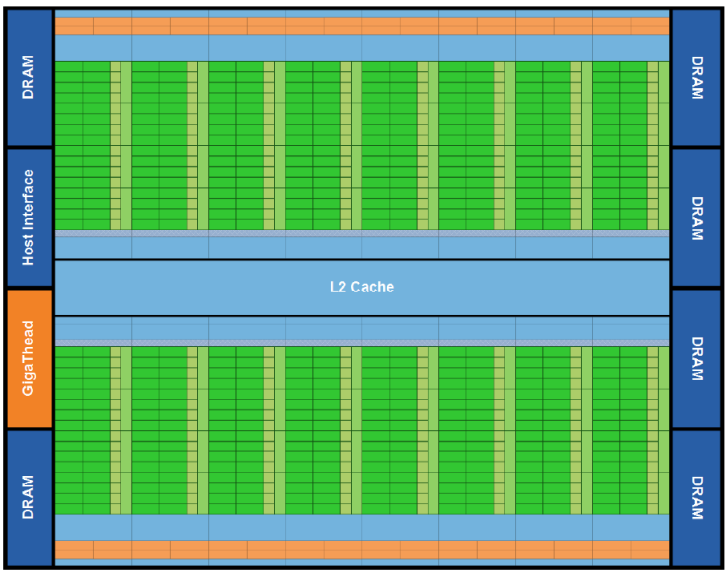

Figura 4.6: Arquitectura de GPU unificada.

4.2.2. Kernels

CUDA es un entorno de desarrollo de algoritmos en GPU que extiende el lenguaje de programación C. Las funciones a ejecutar se denominan kernels. Cuando se realizan llamadas a funciones kernel, éstas se ejecutan N veces en paralelo en N hilos CUDA diferentes, a diferencia de las funciones tradicionales del lenguaje C.

Un kernel se define usando el prefijo *__global__* en su declaración y el número de hilos que ejecuta se determina en cada llamada al kernel. Cada hilo del kernel se identifica mediante un threadID que es accesible dentro del kernel mediante las variables $threadIdx$.

Como ejemplo inicial, el siguiente código realiza la suma de dos vectores A y B de tamaño N y almacena el resultado en el vector C.

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C) {
3    int i = threadIdx.x;
4    C[i] = A[i] + B[i];
5  }
6
7  int main() {
8    // Kernel invocation with N threads
9    VecAdd<<<1, N>>>(A, B, C);
10 }
```

4.2.3. Jerarquía de Hilos

Los hilos del kernel se pueden identificar haciendo uso de la variable $threadIdx$, un vector de 3 componentes que permite establecer configuraciones unidimensionales, bidimensionales o tridimensionales de hilos. Estos hilos se agrupan en un bloque por lo que se proporciona una computación versátil para dominios de vectores, matrices o volúmenes.

El índice de un hilo y su threadID están relacionados de la siguiente manera:

- Bloque unidimensional: el threadID se corresponde con el índice x para un bloque de dimensión Dx.
- Bloque bidimensional: para un bloque de dimensiones (Dx, Dy) el threadID de un hilo con índice (x, y) es $(Dx * y + x)$.
- Bloque tridimensional: para un bloque de dimensiones (Dx, Dy, Dz) el threadID de un hilo con índice (x, y, z) es $(Dx * Dy * z + Dx * y + x)$.

Como ejemplo, el siguiente código realiza la suma de dos matrices A y B de tamaño NxN y almacena el resultado en la matriz C.

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
3  {
4    int i = threadIdx.x;
5    int j = threadIdx.y;
6    C[i][j] = A[i][j] + B[i][j];
7  }
8
9  int main()
10 {
11   ...
12   // Kernel invocation with one block of N * N * 1 threads
13   int numBlocks = 1;
14   dim3 threadsPerBlock(N, N);
15   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
16 }
```

Existe un límite del número de hilos por bloque, puesto que todos los hilos de un bloque se ejecutan en un multiprocesador y deben compartir los recursos de memoria

de dicho multiprocesador. En las GPUs actuales un bloque de puede contener hasta 512 hilos, es decir $Dx * Dy * Dz \leq 512$.

Sin embargo, un kernel puede ejecutar concurrentemente múltiples bloques de hilos, por lo que el número total de hilos es igual al producto del número de bloques por el número de hilos por bloque. Los bloques se organizan a su vez en configuraciones unidimensionales o bidimensionales como ilustra la Figura 4.7.

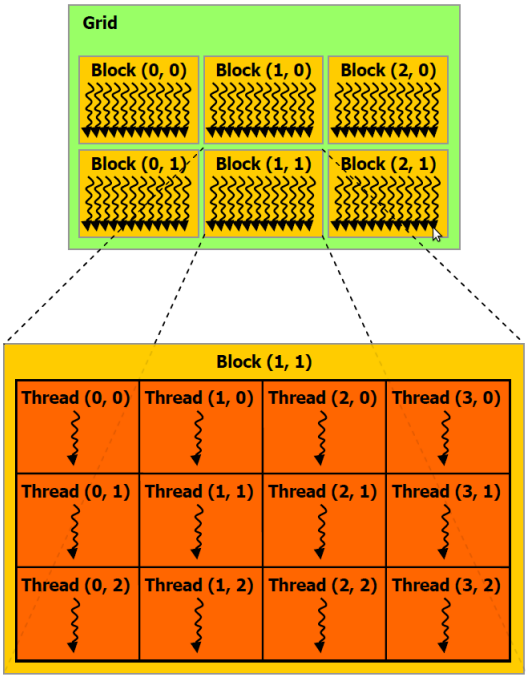

Figura 4.7: Grid de bloques de hilos.

El número de hilos por bloque y el número de bloques en el grid (matriz de bloques de hilos) se especifican como parámetros en la llamada a ejecución del kernel.

Cada bloque del grid se identifica mediante una variable *blockIdx* accesible dentro del kernel. La dimensión del bloque también se puede obtener mediante la variable *blockDim*.

Extendiendo el código del ejemplo anterior de $MatAdd()$ para manejar múltiples bloques y por lo tanto, matrices de mayores dimensiones, quedaría:

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    int j = blockIdx.y * blockDim.y + threadIdx.y;
5
6    if (i < N && j < N)
7      C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main() {
11   ...
12   // Kernel invocation
13   dim3 threadsPerBlock(16, 16);
14   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
15   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
16 }
```

El bloque de hilos tiene un tamaño de 16x16 (256 hilos). El grid se compone de la cantidad suficiente de bloques como para computar la suma sobre todos los elementos de las matrices.

El modelo de programación CUDA mediante la jerarquía de hilos y bloques permite por lo tanto la identificación unívoca de cualquier hilo en un espacio de 5 dimensiones representado en la Figura 4.8 .

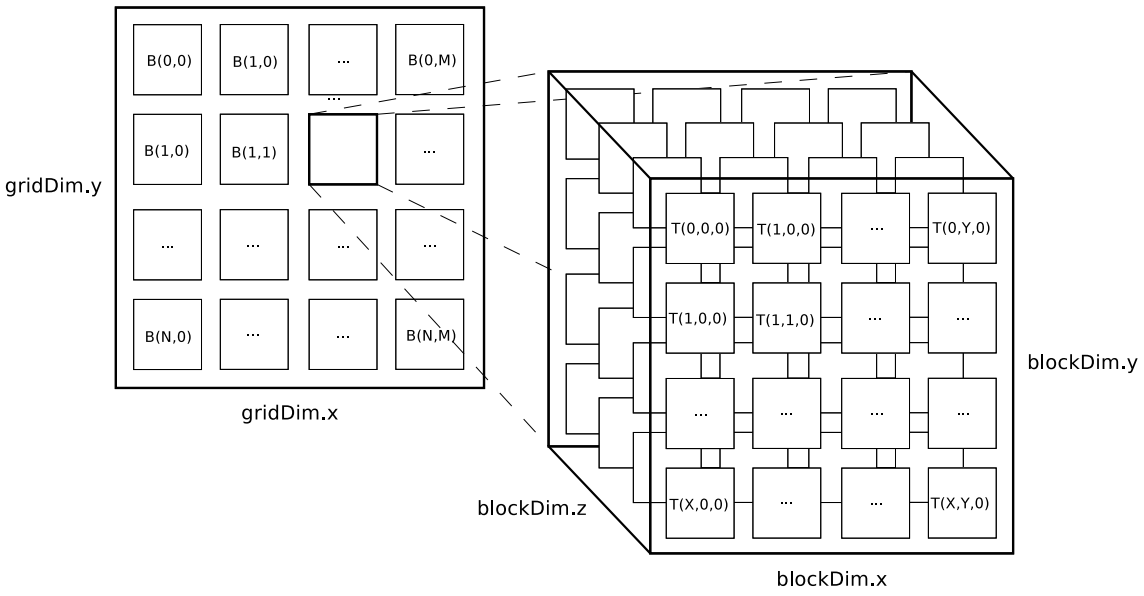

Figura 4.8: Jerarquía de hilos y bloques.

Los bloques se ejecutan independientemente en cada multiprocesador. Debe ser posible ejecutarlos en cualquier orden, en paralelo o en serie. Esta independencia permite que los bloques sean planificados en cualquier orden y en cualquier multiprocesador, facilitando a los desarrolladores la programación de código que escale con el número de procesadores.

Los hilos dentro de un bloque pueden cooperar compartiendo datos mediante la memoria compartida y sincronizando su ejecución para coordinar los accesos a memoria. La sincronización de hilos dentro de un bloque se lleva a cabo mediante la llamada a *__syncthreads*() que actúa como barrera en la que los hilos se bloquean hasta que todos los hilos del bloque alcancen la barrera.

4.2.4. Jerarquía de Memoria

La memoria de la GPU necesita ser rápida y suficientemente amplia para procesar millones de polígonos y texturas. En la Figura 4.9 se representa la evolución del ancho de banda de la memoria principal en las CPUs y GPUs a lo largo de los últimos años.

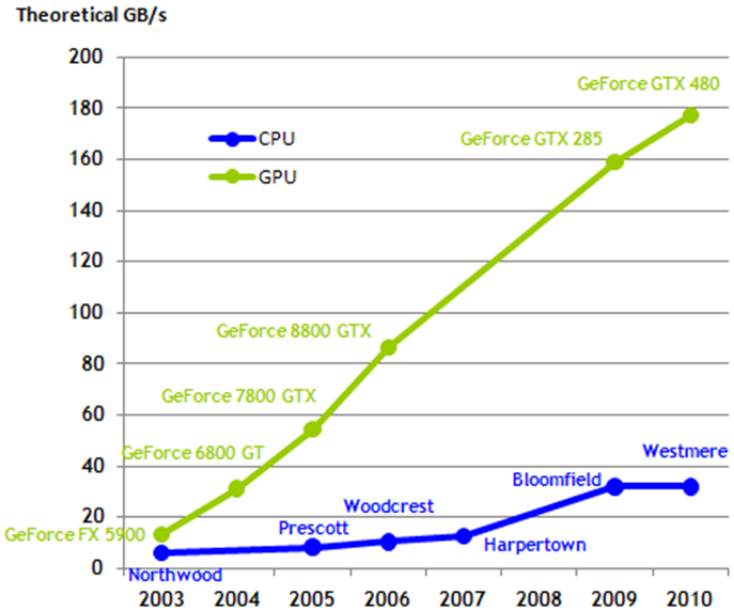

Figura 4.9: Evolución del ancho de banda de la memoria de las CPUs y las GPUs.

Existen cuatro grandes espacios de memoria diferenciados: local, global, de constantes y compartida. Cada uno de dichos espacios de memoria están especializados para unas determinadas funciones. Se diferencian en su tiempo y modo de acceso y

en el tiempo de vida de los datos que contienen.

Cada hilo tiene un espacio de memoria local en forma de registros del multiprocesador para almacenar variables locales al hilo. El número de registros por multiprocesador es de 16384 o 32768, dependiendo de la generación de la GPU, este factor limitará el número de bloques que se pueden ejecutar concurrentemente en un multiprocesador, es decir, el grado de ocupación del multiprocesador. Por ejemplo, un kernel que consuma 24 registros, ejecutado con 256 hilos por bloque, nos dará que un bloque necesita para su ejecución 6144 registros. Con tal demanda de registros, podremos servir concurrentemente un máximo de 2 o 4 bloques por multiprocesador. Es interesante por lo tanto, minimizar el consumo de registros para maximizar la ocupación del multiprocesador.

La memoria global en forma de chips GDDR proporciona un espacio de memoria muy amplio de hasta varios GB que comparten todos los hilos de todos los bloques. Sin embargo, al encontrarse fuera del chip de la GPU, sufre de una alta latencia en su acceso de alrededor de 400-800 ciclos. Todos los hilos pueden leer y escribir en la memoria global, donde deberán almacenar el resultado de la ejecución del kernel.

La memoria de constantes es una zona especializada de la memoria global en la que muchos hilos pueden leer el mismo dato simultáneamente. Esta memoria se encuentra limitada a 64 KB por multiprocesador. Un valor que se lea de la memoria de constantes se sirve a todos los hilos del warp (grupo 32 hilos que entran en ejecución), resultando en el servicio de 32 lecturas de memoria en un único acceso. Esto proporciona una caché muy rápida que sirva a múltiples accesos simultáneos a memoria.

La memoria compartida es una zona de memoria construida en el multiprocesador que se comparte entre todos los hilos de un bloque. Su tamaño es muy reducido, apenas de 16 KB. Sin embargo, al encontrarse dentro del multiprocesador proporciona un acceso con mínima latencia y su contenido sólo se mantiene durante la ejecución de un bloque, por lo que cuando éste termina su ejecución el contenido de la memoria compartida se desecha. Los kernels que leen o escriben un rango conocido de memoria con localidad espacial o temporal pueden emplear la memoria compartida como una caché administrada vía software donde cooperar. La memoria compartida proporciona un método natural de cooperación entre los hilos con mínima latencia, reduciendo los accesos a memoria global y mejorando la velocidad de ejecución. De nuevo, el uso que se haga de memoria compartida determinará el máximo número

de bloques que se podrán ejecutar concurrentemente en el multiprocesador.

Para evitar desperdiciar cientos de ciclos esperando que sean servidos los accesos de lectura o escritura en memoria global, estos se suelen agrupar en accesos colaescientes aprovechando la planificación del warp para solapar las latencias de acceso.

Se dice que los accesos son coalescientes si los hilos en el warp acceden a cualquier palabra en cualquier orden y se emite una única instrucción de acceso a memoria para el acceso al segmento direccionado. Una buena forma de lograrlo es hacer que el hilo i-ésimo acceda a la posición iésima de memoria, así cada hilo accederá a su dirección efectiva pero el conjunto del warp se servirá con un único acceso a memoria global. Las Figuras 4.10, 4.11 y 4.12 reflejan las situaciones posibles del alineamiento, acceso y transacciones de memoria. Garantizar la coalescencia en los accesos a memoria es una de los criterios de mayor prioridad a la hora de optimizar la ejecución de los kernels en la GPU.

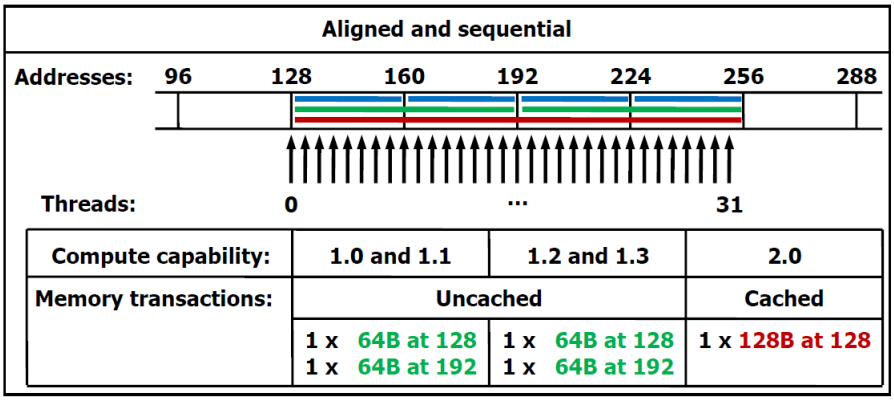

Figura 4.10: Acceso alineado y secuencial.

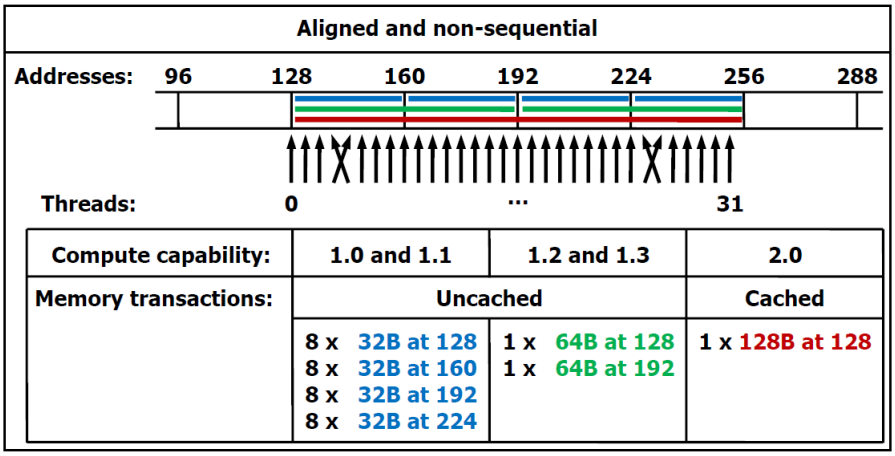

Figura 4.11: Acceso alineado y no secuencial.

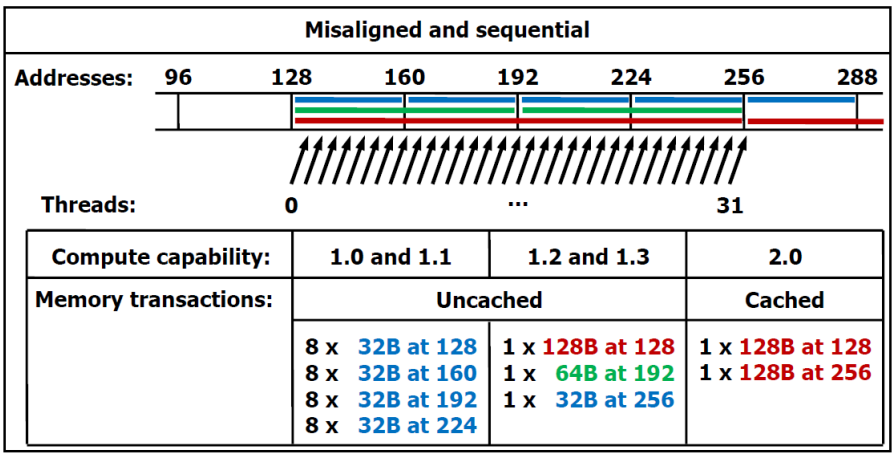

Figura 4.12: Acceso desalineado y secuencial.

4.3. Estudios relacionados de programación genética en GPU

Se ha encontrado numerosos trabajos y publicaciones acerca del uso de las GPUs y arquitecturas masivamente paralelas en el marco de la computación evolutiva. Concretamente, nos centraremos en estudios de algoritmos de programación genética en GPUs [9].

D. Chitty [5] plantea una técnica de computación de propósito general empleando tarjetas gráficas y cómo extender dicha técnica a la programación genética. Demuestra la mejora en el rendimiento del tiempo de ejecución en la resolución de problemas de programación genética sobre arquitecturas monoprocesador. S. Harding [22] centra esta visión en cómo acelerar específicamente la evaluación de los individuos en programación genética.

Estudios previos de trabajos en GPU demuestran la viabilidad de la aceleración de la evaluación para grandes conjuntos de datos. Además D. Robilliard [7] propone un esquema de paralelización para aprovechar el potencial de cálculo de la GPU sobre conjuntos de datos de menor tamaño. Para conseguir una optimización con bases de datos menores, en lugar de evaluar secuencialmente los individuos, paralelizando la evaluación de los patrones de entrenamiento, se comparte la paralelización de la GPU también entre los individuos. Por lo tanto, se evalúan los diferentes programas en paralelo y se asigna cada uno a una unidad de ejecución en la que ejecutar los patrones de entrenamiento en paralelo.

Una técnica similar pero empleando una implementación basada en un modelo Single Program Multiple Data (SPMD) es propuesta por W. Landgdon y A. Harrison [30]. El uso de un modelo SPMD en lugar de uno Single Instruction Multiple Data (SIMD) ofrece la oportunidad de alcanzar mayores aceleraciones, puesto que por ejemplo, una unidad de ejecución puede realizar la rama *if* de un condicional mientras que otra unidad puede ejecutar independientemente la rama *else*. Realizar la ejecución de ambas ramas dentro de una misma unidad de ejecución también es posible, pero las dos ramas se procesarían secuencialmente conforme al modelo SIMD, esto se denomina divergencia y por lo tanto es menos eficiente.

Estos estudios nos han ayudado a diseñar y optimizar nuestra propuesta para alcanzar una máxima aceleración de la ejecución sobre conjuntos de datos y poblaciones de diferentes tamaños.

5. RESTRICCIONES

En este capítulo se expondrán todas las restricciones, o factores limitativos, existentes en el ámbito del diseño y que condicionan la elección de una u otra alternativa. Los factores limitativos pueden estructurarse en dos grupos:

- Factores dato: son aquellos que no pueden ser modificados durante el transcurso del proyecto, como puede ser el presupuesto económico asignado al proyecto o la duración estimada del mismo.
- Factores estratégicos: representan variables de diseño que permiten la elección entre diferentes alternativas por parte del ingeniero. En función de la opción escogida, podrá alterarse el proceso de desarrollo y el propio producto final obtenido, con lo que resultará necesario analizar las posibilidades existentes en las primeras etapas del proceso.

5.1. Factores dato

En el desarrollo de este proyecto se van a considerar los siguientes factores dato impuestos por el tipo de proyecto y por el hardware utilizado:

- Los lenguajes de programación a utilizar serán Java y C haciendo uso de los entornos J2SDK y NVIDIA CUDA y el entorno de desarrollo Eclipse, ya que éste presenta numerosas ventajas derivadas de su naturaleza multiplataforma y orientada a objetos que lo caracteriza.
- En la medida de lo posible, los recursos software utilizados durante el desarrollo del proyecto deberán ser libres, como consecuencia de las restricciones económicas existentes en el mismo, al tratarse de un Proyecto Fin de Carrera. En caso de no resultar posible emplear software libre, se intentará hacer uso de los recursos de los que disponga el autor del proyecto, así como de aquellos que pueda proporcionar el Departamento de Informática y Análisis Numérico de la Universidad de Córdoba.

5.2. Factores estratégicos

Los principales factores estratégicos que afectan al presente proyecto, así como los diferentes motivos que justifican la elección realizada en cada restricción, son los que se muestran a continuación:

- La aplicación que se desea desarrollar deberá ser modular, de manera que permita de manera fácil y con la mayor agilidad posible, la realización de futuras modificaciones y ampliaciones que puedan considerarse necesarias.
- La escalabilidad de la solución a nuevo hardware es imprescindible dada la naturaleza del proyecto.
- La aplicación se desarrollará en el entorno de desarrollo Eclipse, ya que el framework JCLEC sobre el que se basa el presente proyecto hace uso de dicho entorno de desarrollo.
- Para la elaboración de la documentación LaTeXse utilizará el editor Kile, ya que es de libre distribución.
- Para la elaboración de diagramas se hará uso de la herramienta Dia, disponible en libre distribución tanto para Windows como Linux.

6. RECURSOS

En este capítulo se expondrán de forma clara y concisa los recursos humanos y materiales necesarios para este proyecto. Los recursos se definen como aquellos medios de los que se dispone para abordar el proceso de desarrollo del proyecto. El análisis de los recursos existentes se realizará atendiendo a una doble perspectiva:

- Recursos humanos: son aquellos que están constituidos por toda persona que intervenga en el proceso de desarrollo del sistema.

- Recursos materiales: son aquellos que pueden definirse como el conjunto de todas las entidades no animadas que permiten realizar el proceso de desarrollo de la aplicación, así como la generación de la documentación relativa a la misma.

6.1. Recursos humanos

El conjunto de personas que intervendrán durante el proceso de desarrollo del presente proyecto se enumeran a continuación:

- **Directores**: Prof. Dr. Sebastián Ventura Soto.

  Prof. Dra. Amelia Zafra Gómez.

  Los directores se encargarán de supervisar las tareas de desarrollo para comprobar que los resultados obtenidos se corresponden con los requisitos planteados. Además, facilitarán aquellos recursos materiales que resulten necesarios para abordar el proceso de desarrollo exitosamente.

- **Alumno**: Alberto Cano Rojas.

6.2. Recursos materiales

El conjunto de herramientas utilizadas en el proceso de desarrollo del proyecto pueden organizarse en dos categorías principales:

- Recursos software: conjunto de programas informáticos necesarios durante el proceso de desarrollo del proyecto, cuya ejecución se realiza mediante los recursos hardware disponibles. Destacar dos tipos de recursos software: para el desarrollo y para la implantación.
- Recursos hardware: conjunto de equipos informáticos utilizados durante el proceso de realización del proyecto.

6.2.1. Recursos software

Los recursos software empleados para el desarrollo de la aplicación y para la elaboración de la documentación asociada serán:

- Sistemas Operativos Ubuntu 9.10 64, Windows 7 64, Mac OS X 10.6.3
- Compilador de C/C++ GNU 4.3.4
- JAVA SDK & RE 1.6.0_15
- NVIDIA CUDA SDK 3.0
- Eclipse 3.5.1
- JCLEC 4
- Editor de Latex Kile
- Hoja de cálculo Gnumeric
- Hoja de cálculo de OpenOffice.org
- Editor de diagramas Dia
- Editor de imágenes Gimp

6.2.2. Recursos hardware

Una estación de trabajo con las siguientes características:

- Procesador: Intel Core i7 920 @ 2.6 GHz
- Placa Base: Asus P6T Deluxe V2
- Memoria: 12 GB RAM DDR3-1333
- Tarjetas Gráficas: 2 GPUs NVIDIA GTX 285 2GB GDDR3 con las especificaciones de la Tabla 6.1 e ilustradas en la Figura 6.1.
- HDD: 1 TB RAID0 Seagate

NVIDIA GTX 285	
Núcleos de procesamiento	240
Reloj de gráficos (MHz)	648
Reloj de procesador (MHz)	1475
Reloj de la memoria (MHz)	1242
Config. de memoria	2048 MB GDDR3
Interfaz de memoria	512-bit
Ancho de banda de memoria (GB/s.)	159.0
Potencia máxima en la tarjeta gráfica (W)	204 W

Tabla 6.1: Especificaciones NVIDIA GTX 285.

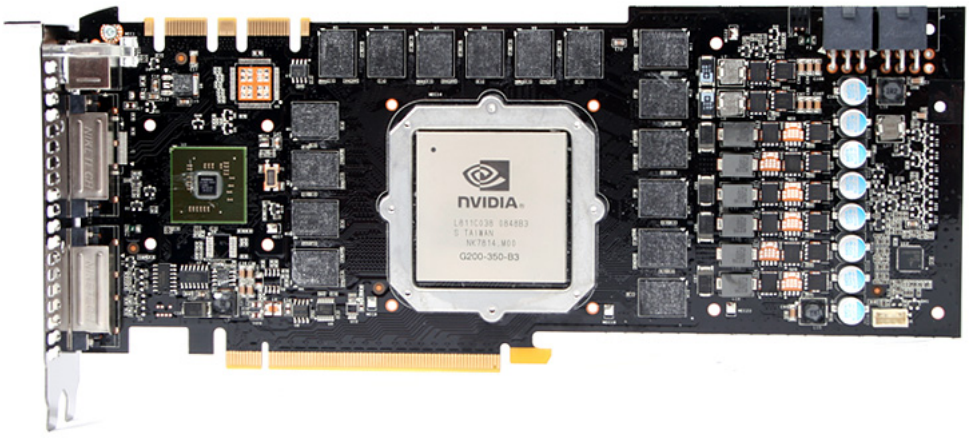

Figura 6.1: NVIDIA GTX 285 PCB.

7. ESPECIFICACIÓN DEL SISTEMA

En los capítulos anteriores se ha proporcionado una visión general del problema a resolver. En este capítulo, dedicado a la especificación se va a detallar el funcionamiento de los algoritmos de programación genética para clasificación, se realizará un análisis del coste computacional de sus fases atentiendo a posibles optimizaciones y se establecerán las bases de nuestro sistema, realizándose una descripción detallada de la aplicación que se pretende desarrollar, prestando especial atención tanto al tipo de información que debe manejar el sistema como a la funcionalidad que el mismo debe aportar.

7.1. Especificación de los algoritmos de clasificación

La programación genética introducida por Koza [12], es una metodología basada en algoritmos evolutivos [25] e inspirada en la evolución natural para encontrar programas de ordenador que realicen una tarea definida por el usuario. Se trata de una especialización de los algoritmos genéticos donde cada invididuo es un programa de ordenador. Por lo tanto, es una técnica de aprendizaje automático que se usa para optimizar una población de programas de acuerdo a una función de ajuste que determina la habilidad del programa para cumplir una tarea.

En nuestro caso, los individuos del algoritmo de programación genética para clasificación son expresiones compuestas por nodos que conforman reglas para resolver la correcta clasificación de un conjunto de instancias. El uso de programación genética para tareas de clasificación ha demostrado ser una técnica que obtiene buenas soluciones [2],[26]. Un clasificador puede expresarse como un conjunto de reglas de tipo Si-Entonces, en las que el antecedente de cada regla está formado por una serie de condiciones que debe cumplir un objeto para que se considere que pertenece a la clase indicada en el consecuente.

La generación de reglas se basa en la utilización de gramáticas para especificar un lenguaje al cual se atendrán los individuos de la población [18], lo cual da lugar a la programación genética gramatical [19]. El uso de una gramática debe ir acompañado de unos operadores de mutación y cruce que se apoyen en la misma, de manera que se garantice que todo individuo de la población, tanto los que forman la población inicial como los que se van generando a lo largo del proceso evolutivo, son legales con respecto a la gramática. Además de servir para solucionar el problema de la clausura (una función u operador debería ser capaz de aceptar como entrada cualquier salida producida por cualquier función u operador del conjunto de no terminales), la utilización de una gramática permite sesgar el proceso evolutivo, de manera que los individuos tengan ciertas características. También puede ser beneficioso desde el punto de vista de la eficiencia, al restringir el espacio de búsqueda.

En la Figura 7.1 se describe la gramática que se emplea para representar las reglas de clasificación de este proyecto.

<ReglaClasificación> ::= "**SI**" (<antecedente>)
"**ENTONCES**" <consecuente>
<antecedente> ::= <condición> |
<condición> "Y" <antecedente> |
<condición> "O" <antecedente>
<consecuente> ::= "**PERTENECE A**" *etiqueta clase*
<condición> ::= <atributo> <operador> <valor>
<atributo> ::= *Cada uno de los atributos del conjunto*
<valor> ::= *Valor del dominio correspondiente*
<operador> ::= "=" | "≠" | "<" | ">" | "≤" | "≥"

Figura 7.1: Formato genérico de una regla de clasificación.

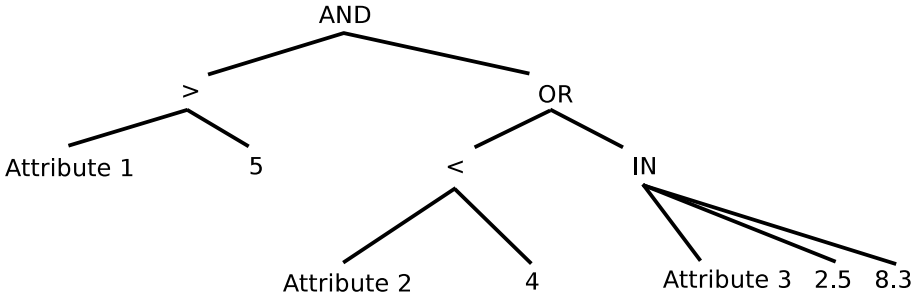

Figura 7.2: Árbol de expresión.

Un programa (individuo de la población) se suele representar como un árbol en la Figura 7.2, las hojas corresponden con los símbolos terminales (variables y constantes), mientras que los nodos internos se corresponden con los no terminales (operadores y funciones).

El proceso de evolución de los algoritmos de programación genética [15], similar al de otros algoritmos evolutivos, se representa en la Figura 7.3 y consta de los siguientes pasos:

1. Se genera una población de individuos inicial de acuerdo a un procedimiento de inicialización.

2. Se evalúa para obtener la calidad de sus individuos, es decir, se comprueba la calidad de las reglas en cuanto a su capacidad de clasificar un conjunto de datos correctamente.

3. En cada generación, el algoritmo selecciona una parte de la población como padres para procrear. El procedimiento de selección suele tomar los mejores individuos como padres para asegurarse la supervivencia de la mejor genética.

4. Este subconjunto de individuos se cruza aplicando los diferentes operadores genéticos de cruce, obteniendo una descendencia.

5. Esta descendencia puede mutarse aplicando los diferentes operadores genéticos de mutación.

6. Estos nuevos individuos deben evaluarse usando la función de ajuste para obtener su índice de calidad.

7. Se pueden aplicar diferentes estrategias de reemplazo de los individuos de la población y de los padres por la descendencia para garantizar que el tamaño de la población en la siguiente generación permanezca constante y mantenga los mejores individuos.

8. El algoritmo realiza una fase de control en la que determina si debe finalizar su ejecución por haber encontrado soluciones de una calidad aceptable o bien por haber alcanzado un número límite de generaciones, si no vuelve al paso 3 y efectúa una nueva iteración. Los procesos encadenados de selección de padres, cruce, mutación, evaluación, reemplazo y control constituyen una generación del algoritmo.

El pseudocódigo del algoritmo generacional simple sería el siguiente:

Algorithm 1 Algoritmo generacional simple

Require: max_generaciones

```
1: Inicializar (P)
2: Evaluar (P)
3: num_generaciones ← 0
4: while num_generaciones < max_generaciones do
5:     P' ← Seleccionar padres (P)
6:     Cruce (P')
7:     Mutación (P')
8:     Evaluar (P')
9:     P ← Reemplazar (P' ∪ P)
10:    num_generaciones++
11: end while
```

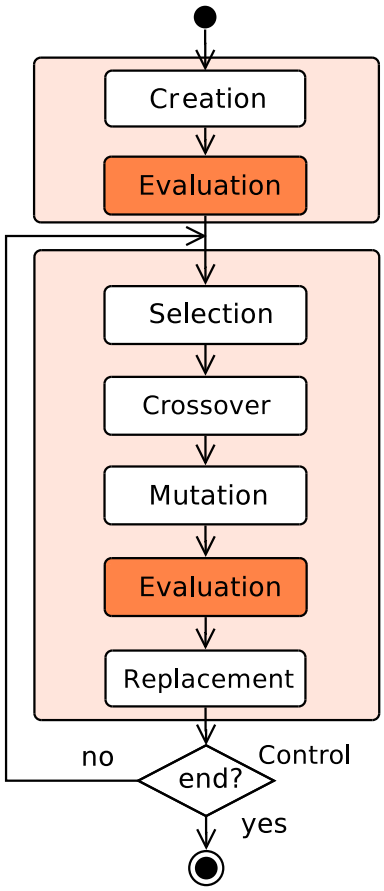

Figura 7.3: Modelo de evolución del algoritmo generacional simple.

7.1.1. Algoritmo de Falco

Falco, Della y Tarantino [11] proponen un método para obtener el fitness del individuo evaluando el antecedente de la regla sobre todos los patrones del conjunto de datos. La gramática de Falco emplea los operadores lógicos AND, OR, NOT los operadores relaciones $=, <, >, \leq, \geq$ y dos operadores de rango numérico IN y OUT. Además el proceso de evolución se repite tantas veces como clases posea el conjunto de datos, de forma que en cada iteración se trata de obtener la mejor regla para cada clase.

En el operador de cruce se seleccionan dos individuos padre, en cada uno de ellos se elegirá un nodo del árbol que representa la regla que implementa el individuo, y se intentarán intercambiar los dos subárboles. Las restricciones que puede presentar el intercambio de subárboles viene dado por dos factores: que los nodos sean compatibles y que la profundidad de los nuevos árboles generados no supere un umbral establecido.

El operador de mutación puede ser aplicado bien a un nodo terminal, o bien a una función que compone el genotipo del individuo. Este operador trabaja de la siguiente forma: se selecciona un nodo aleatoriamente, si es un nodo terminal, se reemplaza por otro nodo terminal compatible seleccionado aleatoriamente; en el caso de que el nodo seleccionado fuese una función, ésta será reemplazada por otra elegida aleatoriamente cumpliéndose que la misma presente la misma compatibilidad y aridad que la seleccionada en el individuo en el punto de mutación.

La función de ajuste calcula la diferencia entre el número de instancias donde la regla predice correctamente la pertenencia o no a la clase y el número de instancias donde ocurre lo contrario, es decir la predicción falla. Finalmente la función de ajuste se define como:

$$fitness = numInstances - (hits - fails) + \alpha * N \tag{7.1}$$

donde α es un valor entre 0 y 1 y N es el número de nodos de la regla. Lo más cercano que α esté a 1 implica que mayor importancia se le otroga a la simplicidad de la regla. El objetivo de la evolución de la población es encontrar reglas que minimicen dicha función de ajuste. El modelo de evolución concuerda con el modelo genérico generacional simple de la Figura 7.3.

7.1.2. Algoritmo de Tan

Tan, Tay, Lee y Heng [14] proponen una versión modificada del algoritmo de estado estacionario [28] en la que se incorpora una población externa y elitismo para asegurar que algunos de los mejores individuos de la generación actual sobrevivirán en la siguiente generación.

Concretamente, la función de ajuste obtiene cuatro valores: los verdaderos positivos (t_p) son el número de ejemplos positivos clasificados como positivos, los falsos negativos (f_n) son los ejemplos positivos que han sido clasificados como negativos, los falsos positivos (f_p) son los ejemplos negativos que han sido clasificados como positivos, y los verdaderos negativos (t_n) son los ejemplos negativos que han sido clasificados como negativos. Varias métricas de rendimiento pueden definirse en base a estas cuatro cantidades.

En el operador de cruce se seleccionan dos individuos padre, en cada uno de ellos se elegirá un nodo del árbol que representa la regla que implementa el individuo, y se intentarán intercambiar los dos subárboles. Las restricciones que puede presentar el intercambio de subárboles viene dado por dos factores, que los nodos sean compatibles y que la profundidad de los nuevos árboles generados no supere un umbral establecido.

El operador de mutación puede ser aplicado bien a un nodo terminal, o bien a una función que compone el genotipo del individuo. Este operador trabaja de la siguiente forma: se selecciona un nodo aleatoriamente, si es un nodo terminal, se reemplaza por otro nodo terminal compatible seleccionado aleatoriamente; en el caso de que el nodo seleccionado fuese una función, ésta será reemplazada por otra elegida aleatoriamente cumpliéndose que la misma presente la misma compatibilidad y aridad que la seleccionada en el individuo en el punto de mutación.

La función de ajuste combina dos indicadores habituales, la sensibilidad (Se) y la especificidad (Sp) que se definen como:

$$Se = \frac{t_p}{t_p + w1 * f_n} \qquad Sp = \frac{t_n}{t_n + w2 * f_p} \tag{7.2}$$

Los parámetros $w1$ y $w2$ se utilizan para ponderar la influencia de los falsos negativos y de los falsos positivos en la calidad del invididuo, esto es muy importante ya

que estos valores son críticos en determinados problemas como diagnóstico. Decrementando $w1$ o incrementando $w2$ mejorará por lo general la calidad de la predicción pero favorece la tendencia a aumentar el número de reglas. El rango [0.2-1] para $w1$ y [1-20] para $w2$ suele ser razonable para la mayoría de los casos. Finalmente, el fitness se obtiene como el producto de dichos dos indicadores:

$$fitness = Se * Sp \tag{7.3}$$

La gramática de Tan es similar a la de Falco pero sin emplear el operador OR ya que mediante combinaciones de AND y NOT se puede generar todas las reglas necesarias, por lo que la simplicidad de las reglas de esta gramática será considerablemente menor. Tan además introduce la técnica de token competition propuesta por Wond y Leung [16] y se emplea como una aproximación alternativa a nichos para promocionar la diversidad en la evolución de múltiples reglas. Múltiples reglas que cubren una misma instancia de entrenamiento incrementan la tendencia de convergencia prematura. La mayoría del tiempo, sólo unas pocas reglas son útiles y cubren la mayoría de las instancias mientras que el resto suelen ser redundantes. La técnica de token competition supone una manera efectiva de eliminar reglas redundantes. El modelo de evolución de Tan con la población auxiliar y la técnica de token competition se representa en la Figura 7.4.

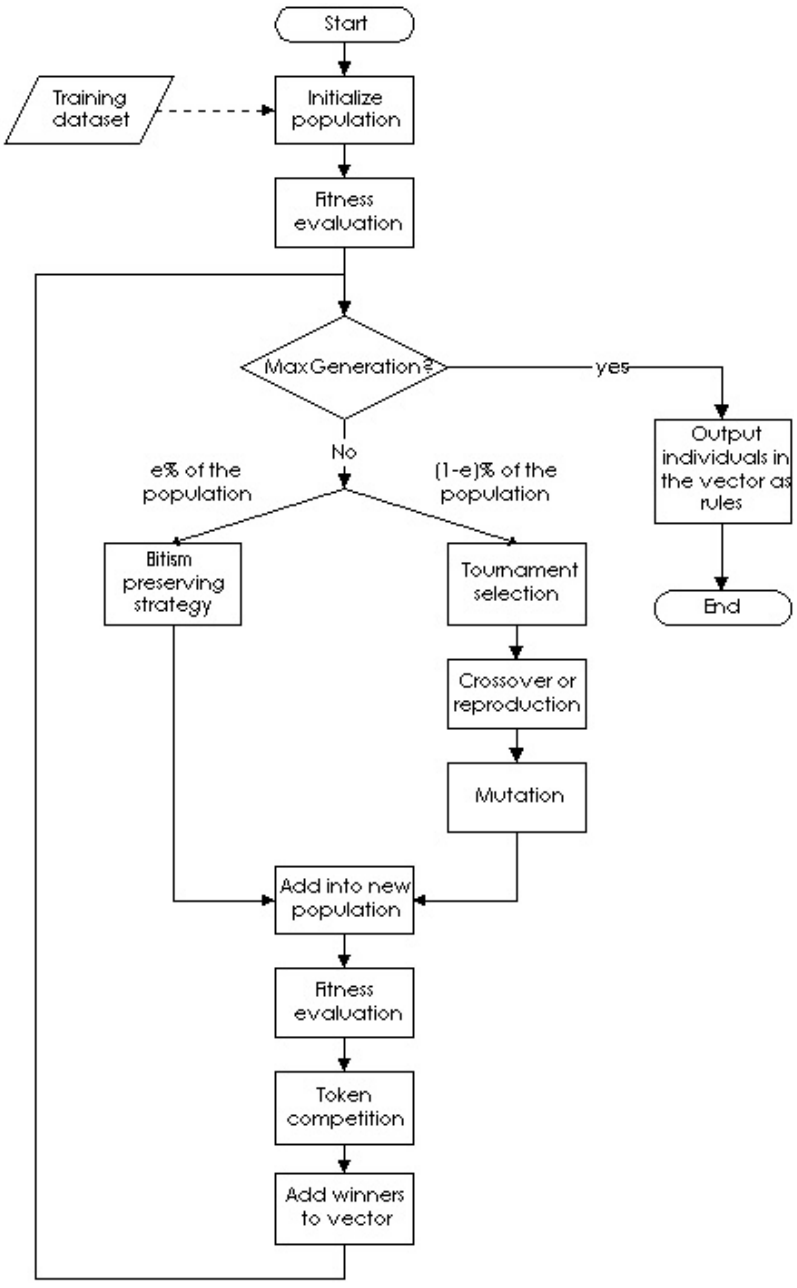

Figura 7.4: Modelo de evolución del algoritmo de Tan.

7.1.3. Algoritmo de Freitas

La propuesta de Bojarczuk, Lopes y Freitas [3] presentan un método en el que cada regla se evalúa para todas las clases simultáneamente para un determinado patrón. El clasificador se forma tomando el mejor individuo para cada clase generado durante el proceso de evolución.

En el operador de cruce se seleccionan dos individuos padre, en cada uno de ellos se elegirá un nodo del árbol que representa la regla que implementa el individuo, y se intentarán intercambiar los dos subárboles. Las restricciones que puede presentar el intercambio de subárboles viene dado por dos factores, que los nodos sean compatibles y que la profundidad de los nuevos árboles generados no supere un umbral establecido. Sin embargo, la versión original del algoritmo carece de operador de mutación.

La gramática de Freitas emplea los operadores lógicos AND, OR, NOT, pese a que con AND y NOT sería suficiente, de esta forma se permite reducir el tamaño de las reglas generadas. La programación genética gramatical no genera soluciones simples. La comprensibilidad de una regla es inversamente proporcional a su tamaño. Por lo tanto, Freitas define la simplicidad (Sy) y por lo tanto la calidad de una regla como:

$$Sy = \frac{maxnodes - 0{,}5 * numnodes - 0{,}5}{maxnodes - 1} \tag{7.4}$$

$$fitness = Se * Sp * Sy \tag{7.5}$$

donde *maxnodes* representa la profundidad máxima de la regla, *numnodes* el número de nodos de la regla actual y *Se* y *Sp* la sensibilidad y la especificidad definidas en el algoritmo de Tan con los parámetros $w1$ y $w2$ igual a 1. El modelo de evolución de Freitas es similar al genérico del generacional simple representado en la Figura 7.3 pero su versión original carece de operador de mutación.

7.2. Análisis del coste computacional

En este apartado se analiza el funcionamiento y el coste computacional de cada una de las fases de los algoritmo evolutivos para identificar aquellas partes cuya paralelización maximicen el rendimiento y minimicen el tiempo de ejecución. A continuación se muestran los tiempos de ejecución expresados en milisegundos de cada una de las fases de los tres algoritmos de programación genética gramatical propuestos. Las pruebas han sido realizadas con 200 individuos y 10 generaciones.

Fase	Tiempo (ms)	Porcentaje
Inicialización	52192	38.28 %
Creación	931	0,68 %
Evaluación	51261	37,60 %
Generación	84129	61,71 %
Selección	11	0,01 %
Cruce	13	0,01 %
Mutación	26	0,02 %
Evaluación	78618	57,67 %
Reemplazo	26	0,02 %
Control	5435	3,99 %
Total	136321	100 %

Tabla 7.1: Tiempos de ejecución del algoritmo de Falco.

Fase	Tiempo (ms)	Porcentaje
Inicialización	56940	11,29 %
Creación	299	0,06 %
Evaluación	56641	11,23 %
Generación	447381	88,70 %
Selección	15	0,00 %
Cruce	37	0,01 %
Mutación	26	0,01 %
Evaluación	302265	59,94 %
Reemplazo	102118	20,25 %
Control	42920	8,51 %
Total	504321	100 %

Tabla 7.2: Tiempos de ejecución del algoritmo de Tan.

Fase	Tiempo (ms)	Porcentaje
Inicialización	9712	12,68 %
Creación	115	0,15 %
Evaluación	9597	12,53 %
Generación	66869	87,31 %
Selección	3	0,00 %
Cruce	15	0,02 %
Evaluación	66270	86,54 %
Reemplazo	17	0,02 %
Control	564	0,74 %
Total	76581	100 %

Tabla 7.3: Tiempos de ejecución del algoritmo de Freitas.

Las pruebas realizadas con el objeto de determinar el coste computacional de las diferentes fases de los algoritmos de Falco y Freitas representados en las Tablas 7.1 y 7.3 muestran que como media, el 96 % del tiempo de ejecución se emplea en la fase de evaluación. En el caso del algoritmo de Tan reflejado en la Tabla 7.2, se emplea el 71 % del tiempo, ya que la técnica del token competition también consume un tiempo considerable en la fase de reemplazo. Este porcentaje está principalmente relacionado con el tamaño de población y el número de patrones, incrementándose hasta a un 98 % en problemas de grandes dimensiones. Por lo tanto, la mayor mejora del tiempo de ejecución se podrá obtener paralelizando la fase de evaluación de los individuos de la población.

La fase de evaluación representa un alto coste computacional debido a que todos los individuos deben ser evaluados sobre todos y cada uno de los patrones del conjunto de datos. Para cada individuo, se debe interpretar su expresión y traducirla a un formato ejecutable para ser evaluada sobre cada patrón. Los resultados de la evaluación del individuo sobre todos los patrones se emplean para contruir la matriz de confusión. La matriz de confusión nos permite aplicar diferentes métricas de calidad para obtener el fitness.

El proceso de evaluación de los individuos consiste en dos bucles, donde cada individuo se evalúa sobre cada patrón. Estos dos bucles determinan el alto coste computacional de los algoritmos de clasificación conforme el número de individuos de la población o el número de patrones se incrementa. Por lo tanto, nuestra prioridad se centra en el estudio del evaluador y el análisis de las posibles propuestas para su aceleración.

7.3. Especificación de la fase de evaluación

La fase de evaluación consiste en tomar cada uno de los individuos de la población y enfrentarlos a los patrones de un conjunto de datos para obtener el índice de calidad del individuo. Para ello se interpreta la regla de clasificación que aporta cada individuo y se evalúa sobre cada instancia del conjunto de datos, obteniendo el número de aciertos y fallos de la regla sobre dicho conjunto de patrones. Concretamente, se obtienen el número de verdaderos positivos (t_p), falsos positivos (f_p), verdaderos negativos (t_n) y falsos negativos (f_n). Con estos datos se construye la matriz de confusión, que empleando la métrica particular de cada algoritmo, obtendrá el índice de calidad de la regla, es decir, el fitness del individuo. Por lo tanto podemos dividir la fase de evaluación en dos pasos: evaluación sobre los patrones y cálculo del fitness.

Tradicionalmente el proceso de evaluación ha sido implementado de forma secuencial, por lo que su tiempo de ejecución aumenta conforme el número de patrones o el número de individuos se incrementa.

Algorithm 2 Evaluador genérico secuencial

Require: population_size, number_patterns

```
1: for each individual within the population do
2:     HITs ← 0, FAILs ← 0
3:     for each pattern from the dataset do
4:         if individual's rule covers actual pattern then
5:             if individual's consequent matches predicted class then
6:                 HITs++
7:             else
8:                 FAILs++
9:             end if
10:        else
11:            if individual's consequent doest not match predicted class then
12:                HITs++
13:            else
14:                FAILs++
15:            end if
16:        end if
17:    end for
18:    individual.Fitness ← ConfusionMatrix(HITs,FAILs);
19: end for
```

El primer paso de la evaluación es por definición paralelo, la evaluación de cada regla sobre cada patrón es un proceso completamente independiente, por lo que éstas se pueden paralelizar en hilos independientes sin ninguna restricción. Su función es interpretar la expresión y comprobar si el resultado coincide con el valor predicho (aprendizaje supervisado). El resultado de esta comparación se almacena para cada patrón y cada regla en modo de acierto o fallo. Este modelo de evaluación propuesto se representa en la Figura 7.5 .

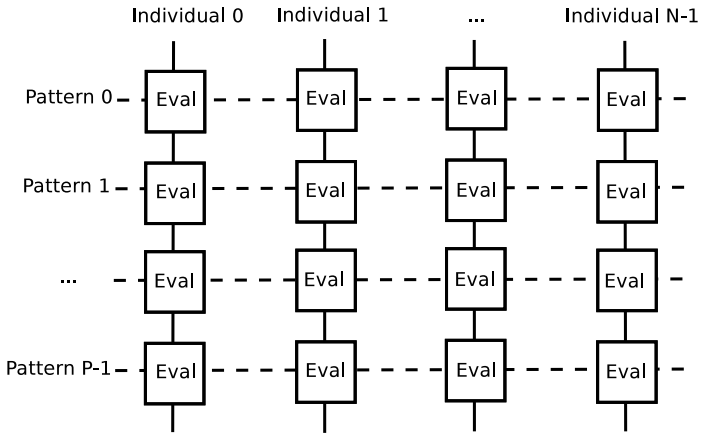

Figura 7.5: Modelo de evaluación paralelo

El segundo paso de la evaluación es una operación de reducción [6] y exige el recuento del número de aciertos y de fallos de cada regla y para todas las reglas. El recuento es un proceso nativamente secuencial, pero puede ser paralelizado mediante la suma de semisumas realizadas en paralelo para cada regla. Además, los diferentes procesos de recuento para cada una de las reglas pueden ser paralelizados sin ninguna interferencia.

El primer paso de la evaluación finaliza con el almacenamiento de los aciertos y fallos para cada patrón y regla. El segundo paso debe contarlos de manera eficiente y para ello, cada regla emplea N hilos que realizan el recuento de la (Nº patrones / Nº hilos) parte de los resultados. Posteriormente, se realiza la suma total de estas N semisumas. Un diseño eficiente de este segundo paso requiere tener consideraciones importantes acerca del hardware y de cómo se almacenan los resultados en memoria.

En el capítulo 9 se detalla la implementación de los evaluadores de los algoritmos en la GPU atendiendo a la optimización del código a la arquitectura concreta.

7.4. Descripción funcional del sistema

En esta sección el sistema será descrito con detalle, haciendo uso de técnicas de modelado que mostrarán de un modo inequívoco cómo interviene el usuario en el sistema, cuáles son los ítems que componen el sistema, y qué operaciones soportan estos ítems. Para ello se empleará la metodología UML (Lenguaje Unificado de Modelado). UML es un lenguaje de modelado que mediante un determinado vocabulario y un conjunto de reglas es capaz de representar conceptual y físicamente un sistema. El motivo de escoger esta metodología es que incorpora técnicas sencillas que dan una visión más cercana de lo que el usuario espera obtener.

7.4.1. Valores para los diferentes parámetros que utiliza el sistema

Con respecto al tamaño de la población, probabilidad de cruce, probabilidad de mutación, número máximo de generaciones, etc. serán valores que se decidirán más adelante, durante las experimentación o en cuanto se pueda evaluar el funcionamiento del módulo, de manera que se consiga una mejor solución. A continuación se describen los parámetros más comunes a definir por el usuario para cada algoritmo:

Algoritmo De Falco:

- *Número de Generaciones:* número máximo de generaciones del algoritmo.
- *Tamaño de la población:* número de individuos que componen la población.
- *Alpha:* factor de penalización para los individuos con mayor número de nodos.
- *Probabilidad de copia:* probabilidad de que un individuo de la población sea copiado directamente a la siguiente generación sin seguir aplicarse operador de reproducción.
- *Probabilidad de cruce:* probabilidad de que un individuo de la población sea seleccionado para aplicársele el operador de cruce o recombinación.
- *Probabilidad de mutación:* probabilidad de que un individuo de la población sea seleccionado para aplicársele el operador de mutación.

Algoritmo Tan:

- *Número de Generaciones:* número máximo de generaciones del algoritmo.
- *Tamaño de la población:* número de individuos que componen la población.
- *Probabilidad de copia:* probabilidad de que un individuo de la población sea copiado directamente a la siguiente generación sin seguir aplicarse operador de reproducción.
- *Probabilidad de cruce:* probabilidad de que un individuo de la población sea seleccionado para aplicársele el operador de cruce o recombinación.
- *Probabilidad de mutación:* probabilidad de que un individuo de la población sea seleccionado para aplicársele el operador de mutación.
- *elitism:* porcentaje de individuos elitistas se desea que estén presentes en la siguiente generación.
- *suport:* parámetro que establece el limite de individuos por instancia dentro de la competición por token (Token Competition).
- *W1:* parámetro que establece el factor de penalización para los f_n.
- *W2:* parámetro que establece el factor de penalización para los f_p.

Algoritmo Freitas:

- *Número de Generaciones:* número máximo de generaciones del algoritmo.
- *Tamaño de la población:* número de individuos que componen la población.
- *Probabilidad de copia:* probabilidad de que un individuo de la población sea copiado directamente a la siguiente generación sin seguir aplicarse operador de reproducción.
- *Probabilidad de cruce:* probabilidad de que un individuo de la población sea seleccionado para aplicársele el operador de cruce o recombinación.

7.4.2. Diagramas de Casos de Uso

Para obtener el modelo de objetos del software a desarrollar, se partirá de un conjunto posible de casos de uso que afectarán al mismo, pudiendo obtener de esta forma todos los requisitos necesarios y los objetos que conformarán el sistema. Un caso de uso especifica el comportamiento de un sistema o de una parte del mismo, y es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor [10]. Los casos de uso se emplean para capturar el comportamiento deseado del sistema en desarrollo, sin tener que especificar cómo se implementa ese comportamiento, además de proporcionar un medio para que los desarrolladores, los usuarios finales del sistema y los expertos del dominio lleguen a una comprensión común del sistema.

Perfiles de usuario

En el sistema de resolución de problemas de modelado se pueden distinguir distintos tipos de actores, un actor representa un conjunto coherente de roles que juegan los usuarios de los casos de uso al interactuar con él. Los actores pueden ser personas o pueden ser sistemas mecánicos estableciendo la siguiente clasificación:

- *Principales:* personas que usan el sistema, serán los usuarios del sistema. El presente trabajo no se basa en una interacción con el usuario del sistema. En este caso, el usuario pretende ejecutar y comparar los resultados ontenidos de diferentes algoritmos de clasificación mediante programación genética, para alcanzar dicho fin, es necesario que el usuario introduzca unos parámetros de entrada, es decir se debe configurar los parámetros referentes a la población de individuos definidos para cada algoritmo en el apartado 7.4.1 . Una vez que ya indica los parámetros, debe esperar a que el programa lleve a cabo su desempeño y recoger las salidas. Las personas que usen el sistema serán aquellas que tengan conocimientos acerca de computación evolutiva.

- *Secundarios:* personas que mantienen o administran el sistema. El mantenimiento de esta aplicación va a ser llevada por otras personas que realicen mejoras, o también actualizaciones debido a evoluciones que sufra la librería JCLEC.

- *Material externo:* dispositivos materiales imprescindibles que forman parte del

ámbito de la aplicación y deben ser utilizados.

- *Otros sistemas:* sistemas con los que el sistema interactúa. Sistema Operativo: indica el sistema sobre el cual trabajará nuestra aplicación, que dado a que Java es un lenguaje multiplataforma podrá ser cualquiera de las arquitecturas soportadas.

Casos de Uso

El modelado de Casos de Uso es la técnica más efectiva y a la vez la más simple para modelar los requisitos del sistema desde la perspectiva del usuario. Los Casos de Uso se utilizan para modelar cómo un sistema funciona actualmente o cómo los usuarios desean que funcione. No es realmente una aproximación a la orientación a objetos; es realmente una forma de modelar procesos. Es, sin embargo, una manera muy eficaz de dirigirse hacia el análisis de sistemas orientados a objetos. Los casos de uso son generalmente el punto de partida del análisis orientado a objetos con UML. La Figura 7.6 y la Tabla 7.4 representan el caso de uso correspondiente al contexto del sistema.

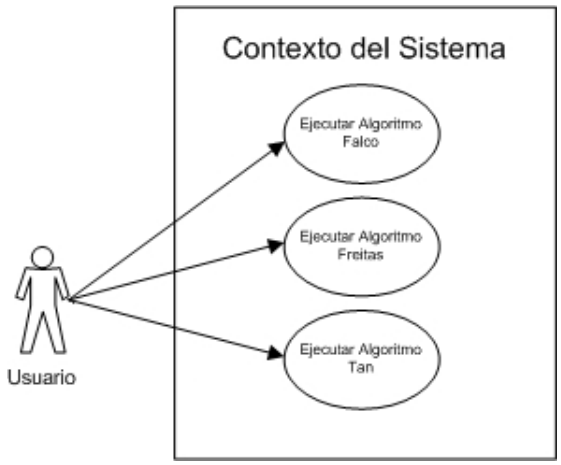

Figura 7.6: Diagrama de Contexto del Sistema.

Nombre	Contexto del Sistema.
Actores	Usuario.
Descripción	El sistema proporciona al usuario cuatro opciones en la ejecución, esto es, realizando las pruebas con el algoritmo Falco, Tan o Freitas.
Casos de Uso	CU1. Algoritmo de Falco. Si el usuario quiere ejecutar el sistema con el algoritmo Falco, tendrá inicialmente que seleccionar dicho algoritmo dentro del conjunto de algoritmos que engloba el sistema JCLEC. CU2. Algoritmo de Tan. Si el usuario quiere ejecutar el sistema con el algoritmo Tan, tendrá inicialmente que seleccionar dicho algoritmo dentro del conjunto de algoritmos que engloba el sistema JCLEC. CU3. Algoritmo de Freitas. Si el usuario quiere ejecutar el sistema con el algoritmo Freitas, tendrá inicialmente que seleccionar dicho algoritmo dentro del conjunto de algoritmos que engloba el sistema JCLEC.
Flujo Principal de Eventos	El usuario comenzará estableciendo el algoritmo que desea ejecutar, de los que podrá elegir Falco, Tan o Freitas dentro del conjunto de algoritmos que incluye el sistema JCLEC.

Tabla 7.4: Diagrama de caso de uso CU0: Contexto del Sistema.

La Figura 7.7 y la Tabla 7.5 representan el caso de uso 1 correspondiente a la ejecución del algoritmo de Falco.

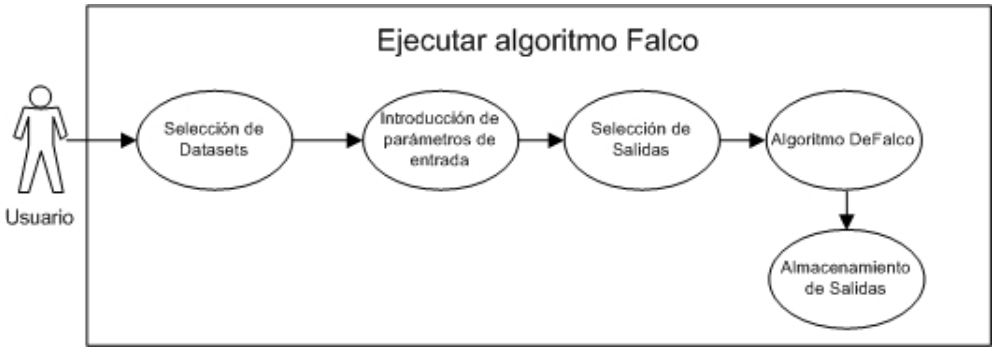

Figura 7.7: Diagrama de caso de uso CU1: Ejecutar algoritmo de Falco.

Nombre	Ejecutar algoritmo de Falco.
Actores	Usuario.
Descripción	Se ejecuta el algoritmo De Falco explicado en el apartado 7.1.1.
Casos de Uso	Selección de Datasets: consiste en la selección de uno o varios de datasets de entrada dentro del conjunto de datasets de que dispone el sistema JCLEC. Introducción de parámetros de entrada: consiste en establecer los datos generales para el sistema de programación genética, así como los particulares de este algoritmo. Algoritmo de Falco: se realizará la evolución a través del algoritmo de Falco, obteniendose una salida que pasará a almacenamiento de salidas.
Flujo Principal de Eventos	El usuario proporciona la información de entrada necesaria para que el sistema inicie su proceso, esta información hace referencia a todos los parámetros y características a la programación genética. El primer paso que efectuará el sistema será la inicialización de la población de individuos, con lo cual podrá comenzar el proceso de evolución. Este proceso consiste en realizar sucesivos ciclos de evolución en los cuales los individuos que conforman la población irán mejorando paulatinamente en aptitud, la cual vendrá determinada por el evaluador de individuos. Para la evaluación de los individuos será necesario el acceso a cada uno de los datasets seleccionados. Produciéndose varias ejecuciones del algoritmo, una por cada conjunto de datos de cada dataset. Esta mejora de aptitud se llevará a cabo con los operadores genéticos especificados por el usuario sobre los individuos de la población. El criterio de terminación del algoritmo es comprobado por el mismo proceso de evolución. Los operadores genéticos, ciclos evolutivos y evaluadores para resolver los problemas de clasificación se incluyen en el framework JCLEC.

Tabla 7.5: CU1: Ejecutar algoritmo de Falco.

La Figura 7.8 y la Tabla 7.6 representan el caso de uso 2 correspondiente a la ejecución del algoritmo de Tan.

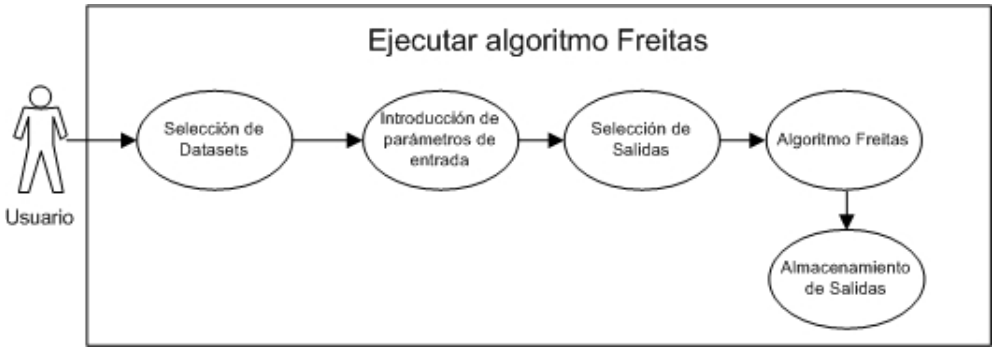

Figura 7.8: Diagrama de caso de uso CU2: Ejecutar algoritmo de Tan.

La Figura 7.9 y la Tabla 7.7 representan el caso de uso 3 correspondiente a la ejecución del algoritmo de Freitas.

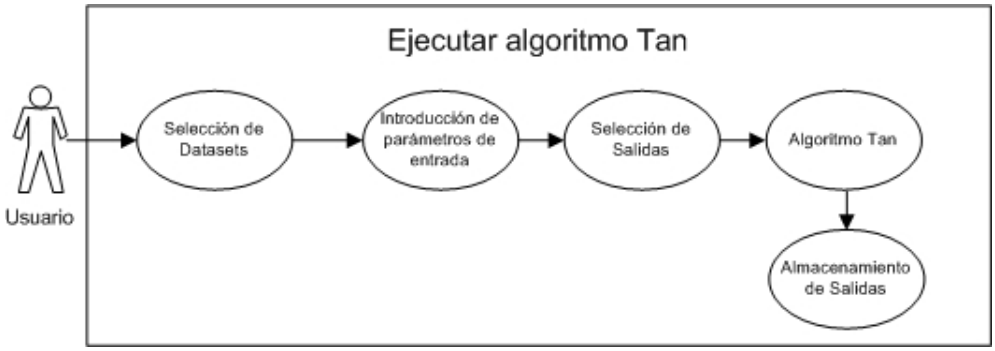

Figura 7.9: Diagrama de caso de uso CU3: Ejecutar algoritmo de Freitas.

Nombre	Ejecutar algoritmo de Tan.
Actores	Usuario.
Descripción	Se ejecuta el algoritmo De Tan explicado en el apartado 7.1.2.
Casos de Uso	Selección de Datasets: consiste en la selección de uno o varios de datasets de entrada dentro del conjunto de datasets de que dispone el sistema JCLEC. Introducción de parámetros de entrada: consiste en establecer los datos generales para el sistema de programación genética, así como los particulares de este algoritmo. Algoritmo de Tan: se realizará la evolución a través del algoritmo de Tan, obteniendose una salida que pasará a almacenamiento de salidas.
Flujo Principal de Eventos	El usuario proporciona la información de entrada necesaria para que el sistema inicie su proceso, esta información hace referencia a todos los parámetros y características a la programación genética. El primer paso que efectuará el sistema será la inicialización de la población de individuos, con lo cual podrá comenzar el proceso de evolución. Este proceso consiste en realizar sucesivos ciclos de evolución en los cuales los individuos que conforman la población irán mejorando paulatinamente en aptitud, la cual vendrá determinada por el evaluador de individuos. Para la evaluación de los individuos será necesario el acceso a cada uno de los datasets seleccionados. Produciéndose varias ejecuciones del algoritmo, una por cada conjunto de datos de cada dataset. Esta mejora de aptitud se llevará a cabo con los operadores genéticos especificados por el usuario sobre los individuos de la población. El criterio de terminación del algoritmo es comprobado por el mismo proceso de evolución. Los operadores genéticos, ciclos evolutivos y evaluadores para resolver los problemas de clasificación se incluyen en el framework JCLEC.

Tabla 7.6: CU2: Ejecutar algoritmo de Tan.

Nombre	Ejecutar algoritmo de Freitas.
Actores	Usuario.
Descripción	Se ejecuta el algoritmo De Freitas explicado en el apartado 7.1.3.
Casos de Uso	Selección de Datasets: consiste en la selección de uno o varios de datasets de entrada dentro del conjunto de datasets de que dispone el sistema JCLEC. Introducción de parámetros de entrada: consiste en establecer los datos generales para el sistema de programación genética, así como los particulares de este algoritmo. Algoritmo de Freitas: se realizará la evolución a través del algoritmo de Freitas, obteniendose una salida que pasará a almacenamiento de salidas.
Flujo Principal de Eventos	El usuario proporciona la información de entrada necesaria para que el sistema inicie su proceso, esta información hace referencia a todos los parámetros y características a la programación genética. El primer paso que efectuará el sistema será la inicialización de la población de individuos, con lo cual podrá comenzar el proceso de evolución. Este proceso consiste en realizar sucesivos ciclos de evolución en los cuales los individuos que conforman la población irán mejorando paulatinamente en aptitud, la cual vendrá determinada por el evaluador de individuos. Para la evaluación de los individuos será necesario el acceso a cada uno de los datasets seleccionados. Produciéndose varias ejecuciones del algoritmo, una por cada conjunto de datos de cada dataset. Esta mejora de aptitud se llevará a cabo con los operadores genéticos especificados por el usuario sobre los individuos de la población. El criterio de terminación del algoritmo es comprobado por el mismo proceso de evolución. Los operadores genéticos, ciclos evolutivos y evaluadores para resolver los problemas de clasificación se incluyen en el framework JCLEC.

Tabla 7.7: CU2: Ejecutar algoritmo de Freitas.

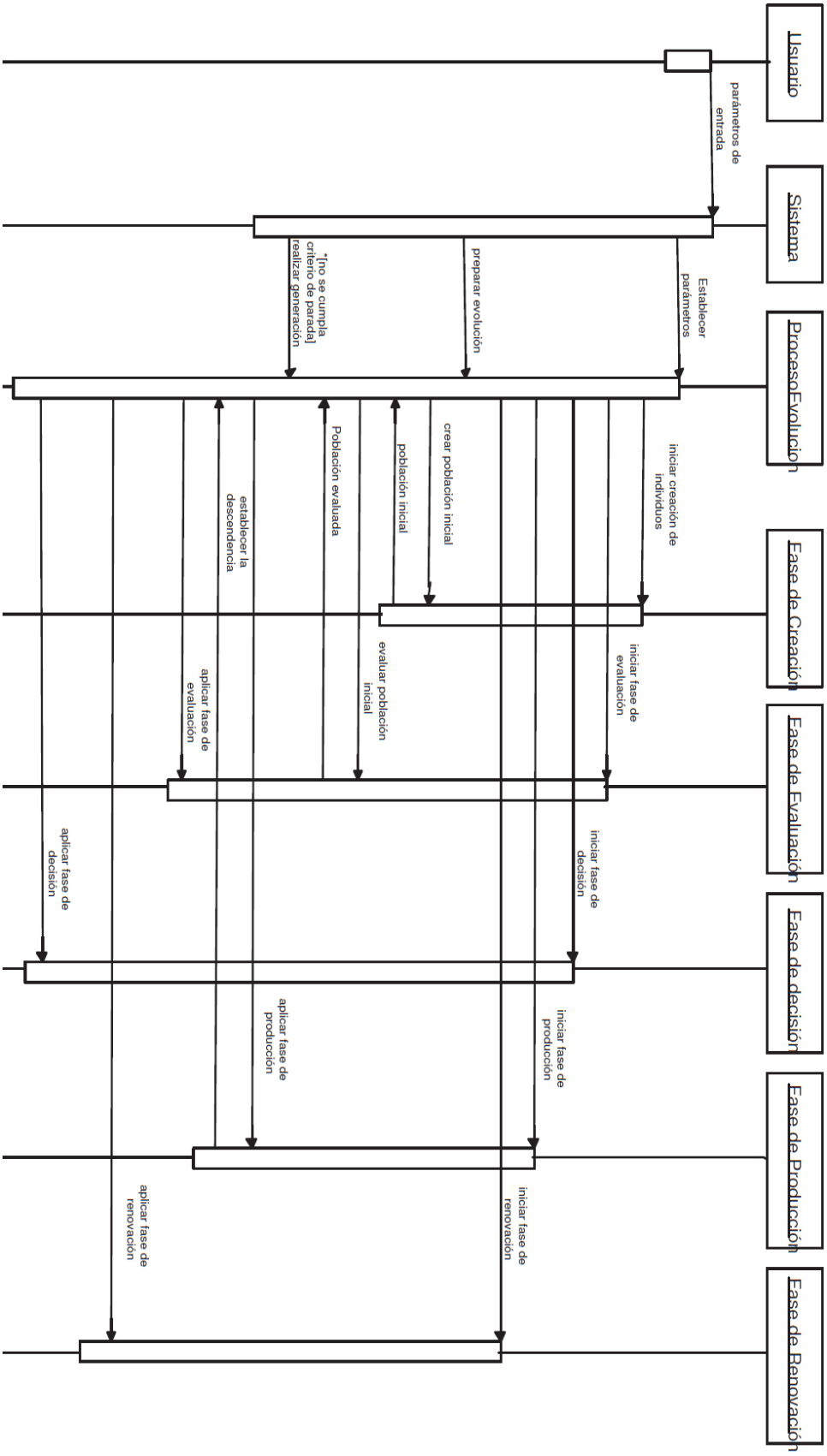

Figura 7.10: Diagrama de secuencia del sistema.

8. DISEÑO DEL SISTEMA

En este capítulo se detallarán las clases que será necesario utilizar e implementar. Para cada clase se especificará el nombre de la clase, una descripción general de la clase, las variables miembro que utiliza y los métodos que utiliza. Estas clases se extraen tanto de los diagramas de casos de uso y los diagramas de secuencia como de la propia definición del problema. Todas se tendrán en cuenta en el modelo de objetos. En cada una de las clases, que se describirán a continuación, se distinguirán principalmente los siguientes apartados:

- Nombre de la clase: nombre que se asignará a la clase para su identificación. Este nombre deberá ser descriptivo, de manera que proporcione una idea acerca del funcionamiento general del elemento al cual pertenece.
- Descripción general de la clase: indicará cuál es el objetivo principal de la clase. Se explicará de forma breve el objetivo fundamental de la misma, así como toda aquella información que pudiera resultar de interés en esta etapa.
- Variables miembro de la clase: en este apartado se analizarán todos aquellos atributos que deberá definir la clase para almacenar la información que ésta necesita para desarrollar la funcionalidad que se le ha asignado de manera correcta. Para cada una de estas variables se indicará su tipo y las principales características a destacar.
- Métodos de la clase: se analizarán todos aquellos métodos definidos por la clase, es decir, aquellas funciones en las que puede estructurarse la funcionalidad desarrollada por la misma.

8.1. Algoritmo de Falco

En este apartado se detallarán las clases que formarán parte del algoritmo de Falco. La Figura 8.1 resume las clases del algoritmo.

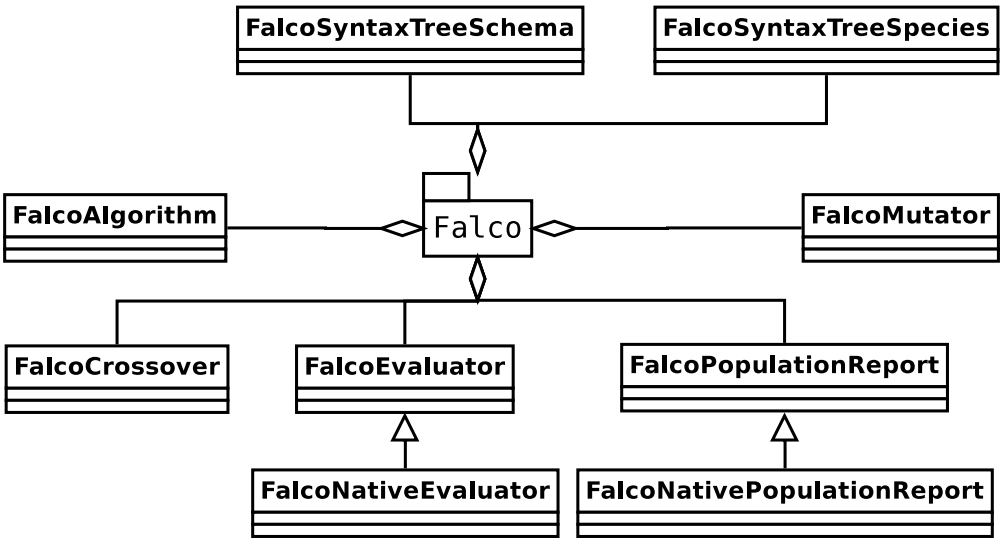

Figura 8.1: Estructura de clases del paquete del algoritmo de Falco.

FalcoAlgorithm

- Descripción: esta clase representa el propio algoritmo que define Falco.
- Variables miembro:
  - parentsSelector: selector de padres para reproducción.
  - recombinator: operador de recombinación utilizado por el algoritmo.
  - mutator: operador de mutación utilizado por el algoritmo.
  - classifier: clasificador generado por el algoritmo.
  - copyProp: probabilidad de que un individuo sea copiado a la siguiente fase.
  - classesNumber: número de clases del dataset al que se le aplica el algoritmo.
  - execution: número de clase considerada en esta ejecución.
  - randGen: generador de números aleatorios.
  - bettersSelector: selector de los mejores individuos.

- trainSet: conjunto de datos de entrenamiento.
- testSet: conjunto de datos de test.
- trainMetadata: metadata del conjunto de datos de entrenamiento.
- testMetadata: metadata del conjunto de datos de test.

- Métodos:
  - setParentsSelector: establece el selector de padres.
  - getTrainMetadata: obtiene el metadata del conjunto de datos de entrenamiento.
  - setTrainSet: establece el conjunto de datos de entrenamiento.
  - getTestSet: obtiene el conjunto de datos de test.
  - setTestSet: establece el conjunto de datos de test.
  - getTestMetadata: obtiene el metadata del conjunto de datos de test.
  - setTestMetadata: establece el metadata del conjunto de datos de test.
  - setRecombinationProb: establece la probabilidad de cruce.
  - setMutationProb: establece la probabilidad de mutación.
  - getRecombinator: devuelve el operador de cruce utilizado por el algoritmo.
  - setRecombinator: establece el operador de cruce utilizado por el algoritmo.
  - getMutator: devuelve el operador de mutación utilizado por el algoritmo.
  - setMutator: establece el operador de mutación utilizado por el algoritmo.
  - getClassifier: devuelve el clasificador resultado del algoritmo.
  - setClassifier: establece el clasificador resultado del algoritmo.
  - getCopyProb: devuelve la probabilidad de copia de un individuo a la generación siguiente.
  - setCopyProb: establece la probabilidad de copia de un individuo a la generación siguiente.
  - configure: configura los parámetros iniciales del algoritmo.
  - equals: define si dos algoritmos son iguales o no.

- doSelection: realiza la selección de los padres a los que se les aplicarán los operadores de reproducción.
- doGeneration: realiza una generación completa.
- doReplacement: realiza el reemplazo de los individuos de la antigua generación por los nuevos.
- doUpdate: actualiza las poblaciones y prepara para la generación siguiente.
- doControl: evalúa si el algoritmo ha llegado a su fin para cada clase.

FalcoEvaluator

- Descripción: esta clase representa el evaluador específico a aplicar al algoritmo de Falco para evaluar a los individuos.
- Variables miembro:
  - rule: regla a evaluar en cada momento.
  - dataset: conjunto de datos de entrenamiento para la evaluación del individuo.
  - testDataset: conjunto de datos de test para la evaluación del individuo.
  - maximize: bandera para decidir si se maximiza la función de fitness del evaluador.
  - randgen: generador de númeos aleatorios.
  - maxDerivSize: tamaño máximo de derivación de las reglas.
  - alpha: porcentaje de simplicidad buscado en las reglas para evaluación.
  - classifiedClass: clase sobre la que se está evaluando en cada momento.
  - comparator: comparador de fitness.
- Métodos:
  - getDataset: devuelve el dataset utilizado para evaluar en entrenamiento.
  - setDataset: establece el dataset utilizado para evaluar en entrenamiento.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.

- getMaxDerivSize: devuelve el número máximo de derivaciones permitidas.
- setMaxDerivSize: establece el número máximo de derivaciones permitidas.
- getAlpha: devuelve el valor de Alpha.
- setAlpha: establece el valor de Alpha.
- getClassifiedClass: devuelve la clase que se está clasificando.
- setClassifiedClass: establece la clase que se está clasificando.
- configure: método para configurar los parámetros del evaluador.
- getComparator: devuelve el comparador de fitness que se utiliza.
- evaluate: método para evaluar a cada individuo.

FalcoNativeEvaluator

- Descripción: esta clase representa el evaluador nativo específico a aplicar al algoritmo de Falco para evaluar a los individuos en GPU o en multihilo.
- Variables miembro:
  - indsToEvaluate: lista de individuos a evaluar.
- Métodos:
  - configure: configuración del evaluador nativo. Reserva la memoria dinámica necesaria y lanza los hilos de ejecución del evaluador.
  - getExprTree: obtiene la regla del individuo en un formato ejecutable.
  - setFitness: establece el fitness al individuo especificado.
  - evaluate: evalúa los individuos indicados en la variable indsToEvaluate;

FalcoCrossover

- Descripción: esta clase representa el operador de cruce específico a aplicar al algoritmo de Falco. A partir de dos padres, generará dos nuevos hijos, siempre que no se violen las restricciones de tamaño máximo de derivación para las reglas.

- Variables miembro: ninguna.
- Métodos:
  - compare: devuelve si dos símbolos son compatibles para el cruce.
  - recombine: dados dos padres y dos hijos resultado, genera en los dos hijos el resultado de aplicar el operador de cruce a los padres.
  - searchSymbolIn: realiza la búsqueda de un símbolo determinado dentro del árbol sintáctico que forma el genotipo del individuo.
  - selectSymbol: selecciona un símbolo al azar dentro del árbol sintáctico que representa el genotipo del individuo.
  - endOfBranch: dado un árbol (genotipo), devuelve el punto final de la producción que se inicia en el nodo indicado como argumento.
  - areCompatible: dados el nombre identificativo de dos símbolos o bloques, devuelve como resultado si ambos son compatibles gramaticalmente.

FalcoMutator

- Descripción: esta clase representa el operador de mutación específico a aplicar al algoritmo de Falco. A partir de un padre, generará un nuevo hijo, siempre cuidando que el nodo seleccionado para mutación y el que se desea insertar sean compatibles y presenten la misma aridad.
- Variables miembro: ninguna.
- Métodos:
  - mutateSyntaxTree: dado el genotipo de un padre y el esquema sintáctico de los árboles de la especie, devolverá un nuevo genotipo hijo ya mutado.
  - selectSymbol: selecciona un símbolo al azar dentro del árbol sintáctico que representa el genotipo del individuo.
  - endOfBranch: dado un árbol (genotipo), devuelve el punto final de la producción que se inicia en el nodo indicado como argumento.
  - selectOtherTerminalSymbol: elige un símbolo terminal, y se selecciona otro diferente compatible con el primero

FalcoPopulationReport

- Descripción: esta clase representa el reporter del algoritmo de Falco que genera los ficheros de información al usuario.
- Variables miembro:
  - reportDirName: nombre del directorio que se creará para guardar los reports.
  - globalReportName: nombre global del directorio de los reports.
  - reportFrequency: frecuencia de generación de informes.
  - initTime: marca de tiempo de inicio del algoritmo.
  - endTime: marca de tiempo de finalización del algoritmo.
  - especificSymbol: establece el formato de los símbolos decimales.
  - absolutelyBest: almacena el mejor individuo absoluto.
  - reportDirectory: directorio de los reports.
- Métodos:
  - getReportDirName: obtiene el nombre del directorio de los reports.
  - setReportDirName: establece el nombre del directorio de los reports.
  - getGlobalReportName: obtiene el nombre del directorio global de los reports.
  - setGlobalReportName: establece el nombre del directorio global de los reports.
  - getReportFrequency: obtiene la frecuencia de generación de informes.
  - setReportFrequency: establece la frecuencia de generación de informes.
  - algorithmStarted: evento que se dispara cuando el algoritmo se inicia.
  - algorithmFinished: evento que se dispara cuando el algoritmo finaliza.
  - iterationCompleted: evento que se dispara cuando se completa una iteración del algoritmo.
  - configure: método de configuración del report del algoritmo.
  - equals: compara si dos directorios son iguales.

- doIterationReport: genera el informe de la iteración actual.
- doClassificationReport: genera el informe con el clasificador obtenido.

FalcoNativePopulationReport

- Descripción: esta clase representa el reporter nativo del algoritmo de Falco que genera los ficheros de información al usuario.
- Variables miembro: ninguna.
- Métodos:
  - algorithmFinished: evento que se dispara cuando el algoritmo finaliza. Libera la memoria nativa reservada dinámicamente.

FalcoSyntaxTreeSchema

- Descripción: esta clase representa el esquema específico a seguir por la especie de individuos que definirá el algoritmo de Falco para que todos los individuos tengan características comunes.
- Variables miembro:
  - nonTerminalSymbolMap: mapa de símbolos no terminales.
  - moreAnd: determina si otra producción AND de la gramática podrá ser seleccionada.
- Métodos:
  - fillSyntaxBranch: completa una rama de producción sintáctica.

FalcoSyntaxTreeSpecies

- Descripción: esta clase representa la especie específica de los individuos que definirá el algoritmo de Falco para que todos los individuos tengan características comunes.
- Variables miembro:
  - metadata: especificación del dataset.

- existCategoricalAttributes: determina si existen atributos categóricos.
- existNumericalAttributes: determina si existen atributos numéricos.

- Métodos:
  - getMetadata: devuelve la especificación del dataset utilizado.
  - setMetadata: establece la especificación del dataset utilizado.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.
  - setFalcoGrammar: crea la gramática especifica con la que se va a trabajar.
  - getMaxDerivSize: obtiene el tamaño máximo de derivación.
  - setMaxDerivSize: Establece el tamaño máximo de derivación.
  - setTerminalSymbols: establece los símbolos terminales de la gramática.
  - setNonTerminalSymbols: establece los símbolos no terminales de la gramática.
  - setTerminalNodes: establece los símbolos terminales de la gramática.
  - setNonTerminalNodes: establece los símbolos no terminales de la gramática.
  - createIndividual: crea un individuo de tipo FalcoIndividual.
  - configure: establece los parámetros de configuración de la especie.
  - decode: obtiene la expresión sintáctica como regla de clasificación a partir del genotipo.

8.2. Algoritmo de Tan

En este apartado se detallarán las clases que formarán parte del algoritmo de Tan. La Figura 8.2 resume las clases del algoritmo.

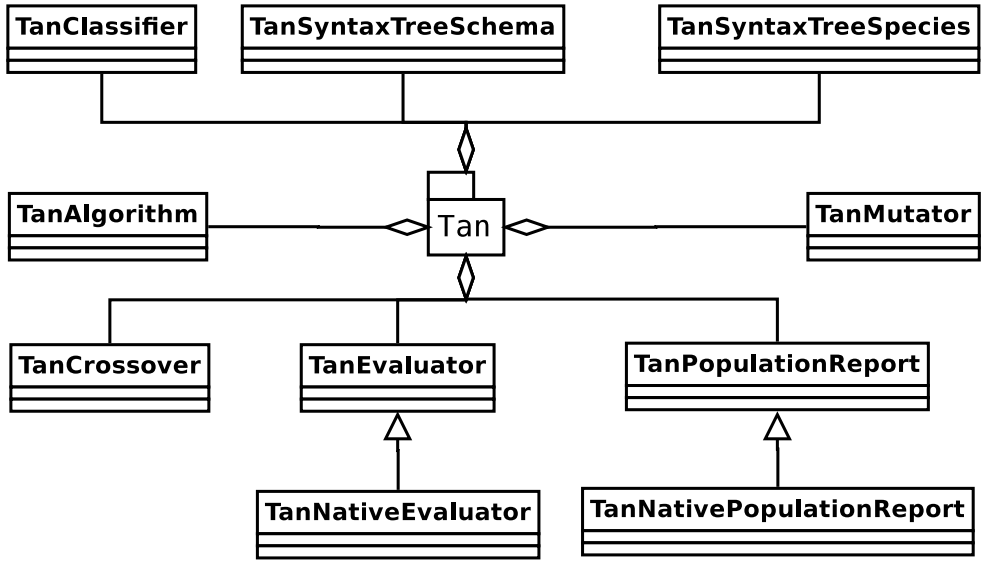

Figura 8.2: Estructura de clases del paquete del algoritmo de Tan.

TanClassifier

- Descripción: esta clase representa el clasificador específico del algoritmo de Tan.
- Variables miembro:
  - array: vector para la ordenación.
- Métodos:
  - sortClassifier: ordena las reglas de clasificación en función de su calidad.
  - setSorter: establece el conjunto de reglas a ordenar.
  - quicksort: ordena el conjunto de reglas.

TanAlgorithm

- Descripción: esta clase representa el propio algoritmo que define Tan.

- Variables miembro:
  - parentsSelector: selector de padres para reproducción.
  - recombinator: operador de recombinación utilizado por el algoritmo.
  - mutator: operador de mutación utilizado por el algoritmo.
  - classifier: clasificador generado por el algoritmo.
  - copyProp: probabilidad de que un individuo sea copiado a la siguiente fase.
  - elitistProb: probabilidad para el porcentaje de individuos elitistas.
  - support: umbral de margen para realizar la competición por token.
  - eset: población auxiliar.
  - classesNumber: número de clases del dataset al que se le aplica el algoritmo.
  - execution: número de clase considerada en esta ejecución.
  - randGen: generador de números aleatorios.
  - bettersSelector: selector de los mejores individuos.
  - trainSet: conjunto de datos de entrenamiento.
  - testSet: conjunto de datos de test.
  - trainMetadata: metadata del conjunto de datos de entrenamiento.
  - testMetadata: metadata del conjunto de datos de test.
- Métodos:
  - setParentsSelector: establece el selector de padres.
  - getTrainMetadata: obtiene el metadata del conjunto de datos de entrenamiento.
  - setTrainSet: establece el conjunto de datos de entrenamiento.
  - getTestSet: obtiene el conjunto de datos de test.
  - setTestSet: establece el conjunto de datos de test.
  - getTestMetadata: obtiene el metadata del conjunto de datos de test.
  - setTestMetadata: establece el metadata del conjunto de datos de test.
  - setRecombinationProb: establece la probabilidad de cruce.

- setMutationProb: establece la probabilidad de mutación.
- getRecombinator: devuelve el operador de cruce utilizado por el algoritmo.
- setRecombinator: establece el operador de cruce utilizado por el algoritmo.
- getMutator: devuelve el operador de mutación utilizado por el algoritmo.
- setMutator: establece el operador de mutación utilizado por el algoritmo.
- getClassifier: devuelve el clasificador resultado del algoritmo.
- setClassifier: establece el clasificador resultado del algoritmo.
- getCopyProb: devuelve la probabilidad de copia de un individuo a la generación siguiente.
- setCopyProb: establece la probabilidad de copia de un individuo a la generación siguiente.
- getElitistProb: obtiene la probabilidad para el porcentaje de individuos elitistas.
- setElitistProb: establece la probabilidad para el porcentaje de individuos elitistas.
- getSupport: obtiene el umbral de margen para realizar la competición por token.
- setSupport: establece el umbral de margen para realizar la competición por token.
- configure: configura los parámetros iniciales del algoritmo.
- equals: define si dos algoritmos son iguales o no.
- doSelection: realiza la selección de los padres a los que se les aplicarán los operadores de reproducción.
- doGeneration: realiza una generación completa.
- doReplacement: realiza el reemplazo de los individuos de la antigua generación por los nuevos.
- doUpdate: actualiza las poblaciones y prepara para la generación siguiente con los individuos del token competition.
- doControl: evalúa si el algoritmo ha llegado a su fin para cada clase.

- doTokenCompetition: realiza la competición por token.

TanEvaluator

- Descripción: esta clase representa el evaluador específico a aplicar al algoritmo de Tan para evaluar a los individuos.
- Variables miembro:
  - rule: regla a evaluar en cada momento.
  - dataset: conjunto de datos de entrenamiento para la evaluación del individuo.
  - testDataset: conjunto de datos de test para la evaluación del individuo.
  - maximize: bandera para decidir si se maximiza la función de fitness del evaluador.
  - randgen: generador de númeos aleatorios.
  - maxDerivSize: tamaño máximo de derivación de las reglas.
  - w1: double parámetro que establece el factor de penalización para los fn para la primera fase.
  - w2: double parámetro que establece el factor de penalización para los fp para la primera fase.
  - classifiedClass: clase sobre la que se está evaluando en cada momento.
  - comparator: comparador de fitness.
- Métodos:
  - getDataset: devuelve el dataset utilizado para evaluar en entrenamiento.
  - setDataset: establece el dataset utilizado para evaluar en entrenamiento.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.
  - getMaxDerivSize: devuelve el número máximo de derivaciones permitidas.
  - setMaxDerivSize: establece el número máximo de derivaciones permitidas.

- getW1: devuelve parámetro con el factor de penalización para los fn.
- setW1: establece devuelve parámetro con el factor de penalización para los fn.
- getW2: devuelve parámetro con el factor de penalización para los fp.
- setW2: establece parámetro con el factor de penalización para los fp.
- getClassifiedClass: devuelve la clase que se está clasificando.
- setClassifiedClass: establece la clase que se está clasificando.
- configure: método para configurar los parámetros del evaluador.
- getComparator: devuelve el comparador de fitness que se utiliza.
- evaluate: método para evaluar a cada individuo.

TanNativeEvaluator

- Descripción: esta clase representa el evaluador nativo específico a aplicar al algoritmo de Tan para evaluar a los individuos en GPU o en multihilo.
- Variables miembro:
  - indsToEvaluate: lista de individuos a evaluar.
- Métodos:
  - configure: configuración del evaluador nativo. Reserva la memoria dinámica necesaria y lanza los hilos de ejecución del evaluador.
  - getExprTree: obtiene la regla del individuo en un formato ejecutable.
  - setFitness: establece el fitness al individuo especificado.
  - evaluate: evalúa los individuos indicados en la variable indsToEvaluate;

TanCrossover

- Descripción: esta clase representa el operador de cruce específico a aplicar al algoritmo de Tan. A partir de dos padres, generará dos nuevos hijos, siempre que no se violen las restricciones de tamaño máximo de derivación para las reglas.
- Variables miembro: ninguna.

- Métodos:
  - compare: devuelve si dos símbolos son compatibles para el cruce.
  - recombine: dados dos padres y dos hijos resultado, genera en los dos hijos el resultado de aplicar el operador de cruce a los padres.
  - searchSymbolIn: realiza la búsqueda de un símbolo determinado dentro del árbol sintáctico que forma el genotipo del individuo.
  - selectSymbol: selecciona un símbolo al azar dentro del árbol sintáctico que representa el genotipo del individuo.
  - endOfBranch: dado un árbol (genotipo), devuelve el punto final de la producción que se inicia en el nodo indicado como argumento.
  - areCompatible: dados el nombre identificativo de dos símbolos o bloques, devuelve como resultado si ambos son compatibles gramaticalmente.

TanMutator

- Descripción: esta clase representa el operador de mutación específico a aplicar al algoritmo de Tan. A partir de un padre, generará un nuevo hijo, siempre cuidando que el nodo seleccionado para mutación y el que se desea insertar sean compatibles y presenten la misma aridad.
- Variables miembro: ninguna.
- Métodos:
  - mutateSyntaxTree: dado el genotipo de un padre y el esquema sintáctico de los árboles de la especie, devolverá un nuevo genotipo hijo ya mutado.
  - selectSymbol: selecciona un símbolo al azar dentro del árbol sintáctico que representa el genotipo del individuo.
  - endOfBranch: dado un árbol (genotipo), devuelve el punto final de la producción que se inicia en el nodo indicado como argumento.
  - selectOtherTerminalSymbol: elige un símbolo terminal, y se selecciona otro diferente compatible con el primero

TanPopulationReport

- Descripción: esta clase representa el reporter del algoritmo de Tan que genera los ficheros de información al usuario.
- Variables miembro:
  - reportDirName: nombre del directorio que se creará para guardar los reports.
  - globalReportName: nombre global del directorio de los reports.
  - reportFrequency: frecuencia de generación de informes.
  - initTime: marca de tiempo de inicio del algoritmo.
  - endTime: marca de tiempo de finalización del algoritmo.
  - especificSymbol: establece el formato de los símbolos decimales.
  - absolutelyBest: almacena el mejor individuo absoluto.
  - reportDirectory: directorio de los reports.
- Métodos:
  - getReportDirName: obtiene el nombre del directorio de los reports.
  - setReportDirName: establece el nombre del directorio de los reports.
  - getGlobalReportName: obtiene el nombre del directorio global de los reports.
  - setGlobalReportName: establece el nombre del directorio global de los reports.
  - getReportFrequency: obtiene la frecuencia de generación de informes.
  - setReportFrequency: establece la frecuencia de generación de informes.
  - algorithmStarted: evento que se dispara cuando el algoritmo se inicia.
  - algorithmFinished: evento que se dispara cuando el algoritmo finaliza.
  - iterationCompleted: evento que se dispara cuando se completa una iteración del algoritmo.
  - configure: método de configuración del report del algoritmo.
  - equals: compara si dos directorios son iguales.

- doIterationReport: genera el informe de la iteración actual.
- doClassificationReport: genera el informe con el clasificador obtenido.

TanNativePopulationReport

- Descripción: esta clase representa el reporter nativo del algoritmo de Tan que genera los ficheros de información al usuario.
- Variables miembro: ninguna.
- Métodos:
  - algorithmFinished: evento que se dispara cuando el algoritmo finaliza. Libera la memoria nativa reservada dinámicamente.

TanSyntaxTreeSchema

- Descripción: esta clase representa el esquema específico a seguir por la especie de individuos que definirá el algoritmo de Tan para que todos los individuos tengan características comunes.
- Variables miembro:
  - nonTerminalSymbolMap: mapa de símbolos no terminales.
  - moreAnd: determina si otra producción AND de la gramática podrá ser seleccionada.
- Métodos:
  - fillSyntaxBranch: completa una rama de producción sintáctica.

TanSyntaxTreeSpecies

- Descripción: esta clase representa la especie específica de los individuos que definirá el algoritmo de Tan para que todos los individuos tengan características comunes.
- Variables miembro:
  - metadata: especificación del dataset.

- existCategoricalAttributes: determina si existen atributos categóricos.
- existNumericalAttributes: determina si existen atributos numéricos.

- Métodos:
  - getMetadata: devuelve la especificación del dataset utilizado.
  - setMetadata: establece la especificación del dataset utilizado.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.
  - setTanGrammar: crea la gramática especifica con la que se va a trabajar.
  - getMaxDerivSize: obtiene el tamaño máximo de derivación.
  - setMaxDerivSize: Establece el tamaño máximo de derivación.
  - setTerminalSymbols: establece los símbolos terminales de la gramática.
  - setNonTerminalSymbols: establece los símbolos no terminales de la gramática.
  - setTerminalNodes: establece los símbolos terminales de la gramática.
  - setNonTerminalNodes: establece los símbolos no terminales de la gramática.
  - createIndividual: crea un individuo de tipo TanIndividual.
  - configure: establece los parámetros de configuración de la especie.
  - decode: obtiene la expresión sintáctica como regla de clasificación a partir del genotipo.

8.3. Algoritmo de Freitas

En este apartado se detallarán las clases que formarán parte del algoritmo de Freitas. La Figura 8.3 resume las clases del algoritmo.

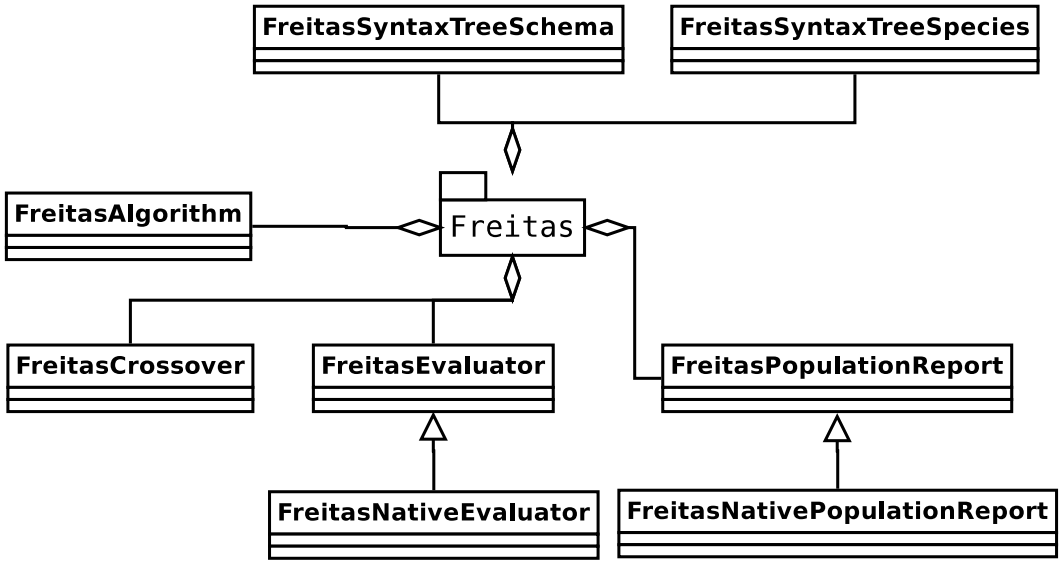

Figura 8.3: Estructura de clases del paquete del algoritmo de Freitas.

FreitasAlgorithm

- Descripción: esta clase representa el propio algoritmo que define Freitas.
- Variables miembro:
  - parentsSelector: selector de padres para reproducción.
  - recombinator: operador de recombinación utilizado por el algoritmo.
  - mutator: operador de mutación utilizado por el algoritmo.
  - classifier: clasificador generado por el algoritmo.
  - copyProp: probabilidad de que un individuo sea copiado a la siguiente fase.
  - classesNumber: número de clases del dataset al que se le aplica el algoritmo.
  - execution: número de clase considerada en esta ejecución.
  - randGen: generador de números aleatorios.

- bettersSelector: selector de los mejores individuos.
- trainSet: conjunto de datos de entrenamiento.
- testSet: conjunto de datos de test.
- trainMetadata: metadata del conjunto de datos de entrenamiento.
- testMetadata: metadata del conjunto de datos de test.

- Métodos:
  - setParentsSelector: establece el selector de padres.
  - getTrainMetadata: obtiene el metadata del conjunto de datos de entrenamiento.
  - setTrainSet: establece el conjunto de datos de entrenamiento.
  - getTestSet: obtiene el conjunto de datos de test.
  - setTestSet: establece el conjunto de datos de test.
  - getTestMetadata: obtiene el metadata del conjunto de datos de test.
  - setTestMetadata: establece el metadata del conjunto de datos de test.
  - setRecombinationProb: establece la probabilidad de cruce.
  - setMutationProb: establece la probabilidad de mutación.
  - getRecombinator: devuelve el operador de cruce utilizado por el algoritmo.
  - setRecombinator: establece el operador de cruce utilizado por el algoritmo.
  - getMutator: devuelve el operador de mutación utilizado por el algoritmo.
  - setMutator: establece el operador de mutación utilizado por el algoritmo.
  - getClassifier: devuelve el clasificador resultado del algoritmo.
  - setClassifier: establece el clasificador resultado del algoritmo.
  - getCopyProb: devuelve la probabilidad de copia de un individuo a la generación siguiente.
  - setCopyProb: establece la probabilidad de copia de un individuo a la generación siguiente.
  - configure: configura los parámetros iniciales del algoritmo.

- equals: define si dos algoritmos son iguales o no.
- doSelection: realiza la selección de los padres a los que se les aplicarán los operadores de reproducción.
- doGeneration: realiza una generación completa.
- doReplacement: realiza el reemplazo de los individuos de la antigua generación por los nuevos.
- doUpdate: actualiza las poblaciones y prepara para la generación siguiente.

FreitasEvaluator

- Descripción: esta clase representa el evaluador específico a aplicar al algoritmo de Freitas para evaluar a los individuos.
- Variables miembro:
  - rule: regla a evaluar en cada momento.
  - dataset: conjunto de datos de entrenamiento para la evaluación del individuo.
  - testDataset: conjunto de datos de test para la evaluación del individuo.
  - maximize: bandera para decidir si se maximiza la función de fitness del evaluador.
  - randgen: generador de númeos aleatorios.
  - maxDerivSize: tamaño máximo de derivación de las reglas.
  - alpha: porcentaje de simplicidad buscado en las reglas para evaluación.
  - classifiedClass: clase sobre la que se está evaluando en cada momento.
  - comparator: comparador de fitness.
- Métodos:
  - getDataset: devuelve el dataset utilizado para evaluar en entrenamiento.
  - setDataset: establece el dataset utilizado para evaluar en entrenamiento.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.

- getMaxDerivSize: devuelve el número máximo de derivaciones permitidas.
- setMaxDerivSize: establece el número máximo de derivaciones permitidas.
- getAlpha: devuelve el valor de Alpha.
- setAlpha: establece el valor de Alpha.
- getClassifiedClass: devuelve la clase que se está clasificando.
- setClassifiedClass: establece la clase que se está clasificando.
- configure: método para configurar los parámetros del evaluador.
- getComparator: devuelve el comparador de fitness que se utiliza.
- evaluate: método para evaluar a cada individuo.

FreitasNativeEvaluator

- Descripción: esta clase representa el evaluador nativo específico a aplicar al algoritmo de Freitas para evaluar a los individuos en GPU o en multihilo.
- Variables miembro:
  - indsToEvaluate: lista de individuos a evaluar.
- Métodos:
  - configure: configuración del evaluador nativo. Reserva la memoria dinámica necesaria y lanza los hilos de ejecución del evaluador.
  - getExprTree: obtiene la regla del individuo en un formato ejecutable.
  - setFitness: establece el fitness al individuo especificado.
  - evaluate: evalúa los individuos indicados en la variable indsToEvaluate;

FreitasCrossover

- Descripción: esta clase representa el operador de cruce específico a aplicar al algoritmo de Freitas. A partir de dos padres, generará dos nuevos hijos, siempre que no se violen las restricciones de tamaño máximo de derivación para las reglas.

- Variables miembro: ninguna.
- Métodos:
  - compare: devuelve si dos símbolos son compatibles para el cruce.
  - recombine: dados dos padres y dos hijos resultado, genera en los dos hijos el resultado de aplicar el operador de cruce a los padres.
  - searchSymbolIn: realiza la búsqueda de un símbolo determinado dentro del árbol sintáctico que forma el genotipo del individuo.
  - selectSymbol: selecciona un símbolo al azar dentro del árbol sintáctico que representa el genotipo del individuo.
  - endOfBranch: dado un árbol (genotipo), devuelve el punto final de la producción que se inicia en el nodo indicado como argumento.
  - areCompatible: dados el nombre identificativo de dos símbolos o bloques, devuelve como resultado si ambos son compatibles gramaticalmente.

FreitasPopulationReport

- Descripción: esta clase representa el reporter del algoritmo de Freitas que genera los ficheros de información al usuario.
- Variables miembro:
  - reportDirName: nombre del directorio que se creará para guardar los reports.
  - globalReportName: nombre global del directorio de los reports.
  - reportFrequency: frecuencia de generación de informes.
  - initTime: marca de tiempo de inicio del algoritmo.
  - endTime: marca de tiempo de finalización del algoritmo.
  - especificSymbol: establece el formato de los símbolos decimales.
  - absolutelyBest: almacena el mejor individuo absoluto.
  - reportDirectory: directorio de los reports.
- Métodos:
  - getReportDirName: obtiene el nombre del directorio de los reports.

- setReportDirName: establece el nombre del directorio de los reports.
- getGlobalReportName: obtiene el nombre del directorio global de los reports.
- setGlobalReportName: establece el nombre del directorio global de los reports.
- getReportFrequency: obtiene la frecuencia de generación de informes.
- setReportFrequency: establece la frecuencia de generación de informes.
- algorithmStarted: evento que se dispara cuando el algoritmo se inicia.
- algorithmFinished: evento que se dispara cuando el algoritmo finaliza.
- iterationCompleted: evento que se dispara cuando se completa una iteración del algoritmo.
- configure: método de configuración del report del algoritmo.
- equals: compara si dos directorios son iguales.
- doIterationReport: genera el informe de la iteración actual.
- doClassificationReport: genera el informe con el clasificador obtenido.

FreitasNativePopulationReport

- Descripción: esta clase representa el reporter nativo del algoritmo de Freitas que genera los ficheros de información al usuario.
- Variables miembro: ninguna.
- Métodos:
  - algorithmFinished: evento que se dispara cuando el algoritmo finaliza. Libera la memoria nativa reservada dinámicamente.

FreitasSyntaxTreeSchema

- Descripción: esta clase representa el esquema específico a seguir por la especie de individuos que definirá el algoritmo de Freitas para que todos los individuos tengan características comunes.
- Variables miembro:

  - nonTerminalSymbolMap: mapa de símbolos no terminales.
  - moreAnd: determina si otra producción AND de la gramática podrá ser seleccionada.

- Métodos:
  - fillSyntaxBranch: completa una rama de producción sintáctica.

FreitasSyntaxTreeSpecies

- Descripción: esta clase representa la especie específica de los individuos que definirá el algoritmo de Freitas para que todos los individuos tengan características comunes.
- Variables miembro:
  - metadata: especificación del dataset.
  - existCategoricalAttributes: determina si existen atributos categóricos.
  - existNumericalAttributes: determina si existen atributos numéricos.
- Métodos:
  - getMetadata: devuelve la especificación del dataset utilizado.
  - setMetadata: establece la especificación del dataset utilizado.
  - getRandGen: devuelve el generador de números aleatorios.
  - setRandGen: establece el generador de números aleatorios.
  - setFreitasGrammar: crea la gramática especifica con la que se va a trabajar.
  - getMaxDerivSize: obtiene el tamaño máximo de derivación.
  - setMaxDerivSize: Establece el tamaño máximo de derivación.
  - setTerminalSymbols: establece los símbolos terminales de la gramática.
  - setNonTerminalSymbols: establece los símbolos no terminales de la gramática.
  - setTerminalNodes: establece los símbolos terminales de la gramática.
  - setNonTerminalNodes: establece los símbolos no terminales de la gramática.

- createIndividual: crea un individuo de tipo FreitasIndividual.
- configure: establece los parámetros de configuración de la especie.
- decode: obtiene la expresión sintáctica como regla de clasificación a partir del genotipo.

9. IMPLEMENTACIÓN

En este capítulo se detallarán aquellas particularidades para la implementación real de los tres algoritmos de clasificación y sus evaluadores nativos. El diseño y la implementación de los kernels de ejecución están ligados por naturaleza con el objetivo de conseguir una buena optimización de los recursos de la arquitectura de la GPU que permita minimizar el tiempo de ejecución. Se han seguido las recomendaciones de la guía de programación de NVIDIA CUDA y del manual de buenas prácticas [17], así como de las aportaciones de otros autores [23].

9.1. Java Native Interface

Java Native Interface (JNI) es un framework de programación que permite que un programa escrito en Java ejecutado en la máquina virtual java pueda interactuar con programas escritos en otros lenguajes como C y C++. El framework JNI permite a un método nativo utilizar los objetos Java y sus métodos de la misma forma que en el propio código en Java.

También se usa para modificar programas existentes escritos en algún otro lenguaje, permitiéndoles ser accesibles desde aplicaciones Java. Muchas de las clases de la API estándar de Java dependen del JNI para proporcionar funcionalidad al desarrollador y al usuario, por ejemplo las funcionalidades de sonido o lectura/escritura de ficheros. El desarrollador debe asegurarse que la API estándar de Java no proporciona una determinada funcionalidad antes de recurrir al JNI, ya que la primera ofrece una implementación segura e independiente de la plataforma.

Dado que puede ser usado para interactuar con código escrito en otros lenguajes como C++, también se usa para operaciones y cálculos de alta complejidad temporal, porque el código nativo es por lo general más rápido que el que se ejecuta en una máquina virtual. Por ello, externalizaremos la función de ajuste en un método nativo implementado en C que se ejecutará en multihilo en la CPU o implementado en CUDA para su ejecución en la GPU. La Figura 9.1 muestra el diagrama de relaciones entre el código Java y nativo en el proceso de compilación.

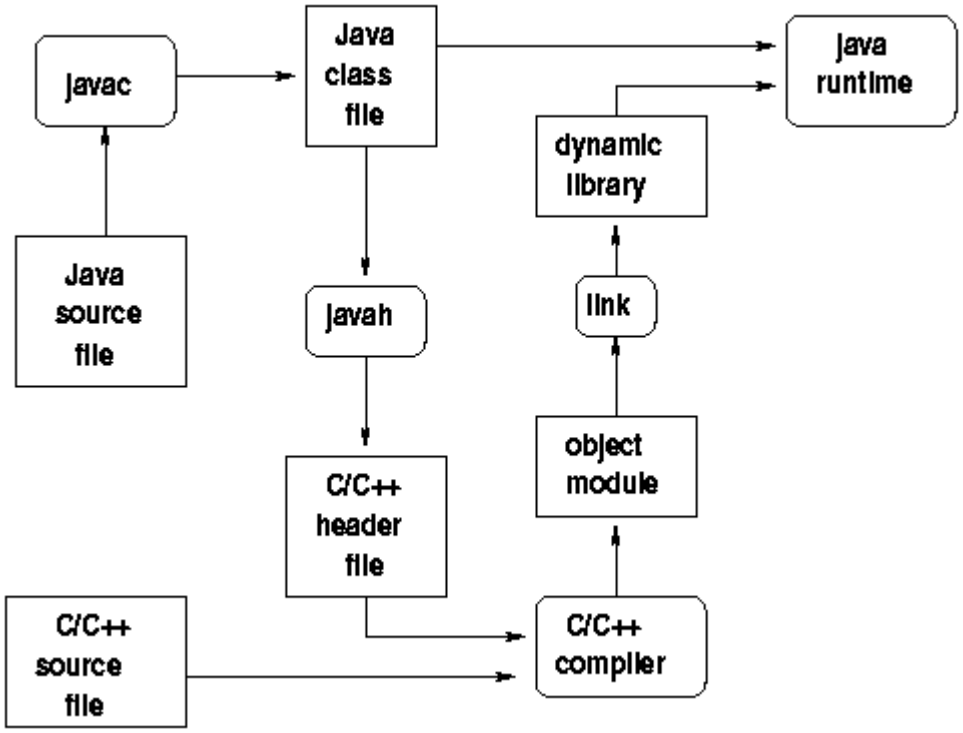

Figura 9.1: Java Native Interface.

9.2. Implementación del evaluador nativo

En la sección 7.3 se detalló la especificación del evaluador y se observó que el proceso de evaluación puede dividirse en dos fases completamente paralelizables. El primer paso de la evaluación finaliza con el almacenamiento de los aciertos y fallos para cada patrón y regla. El segundo paso debe contarlos de manera eficiente y para ello cada regla emplea un número determinado de hilos que realizan el recuento de la (Nº patrones / Nº hilos) parte de los resultados. Posteriormente, se realiza la suma total de estas semisumas.

La guía de programación CUDA recomienda un número de hilos que sea múltiplo del tamaño del warp (conjunto de hilos que entran en ejecución por el despachador) que en las arquitecturas actuales es de 32 hilos. Una buena aproximación sería bloques de hilos de 32, 64, 96, 128, ... hilos. En la sección de experimentación se ha obtenido que el número óptimo de hilos para nuestro problema es de 128, por lo que procederemos a realizar la explicación con este número, aunque podría variar en futuras arquitecturas.

La paralelización de la primera fase de la evaluación en la que se determina si una regla cubre o no a un patrón es inmediata. Cada hilo representa la evaluación de un individuo sobre un patrón. Los hilos se agrupan en conjuntos de 128 para formar un bloque de hilos, requiriendo por lo tanto un número de bloques de (Nº patrones / 128) para la evaluación de un individuo. A su vez, esto debe extenderse para todos los individuos configurando así una matriz bidimensional de bloques de hilos de tamaño (Nº patrones / 128) x Nº individuos, representada en la Figura 9.2. Es interesante destacar que en conjuntos de datos grandes con una población numerosa, el número total de hilos en la GPU alcanzará una cifra de millones de hilos de ejecución.

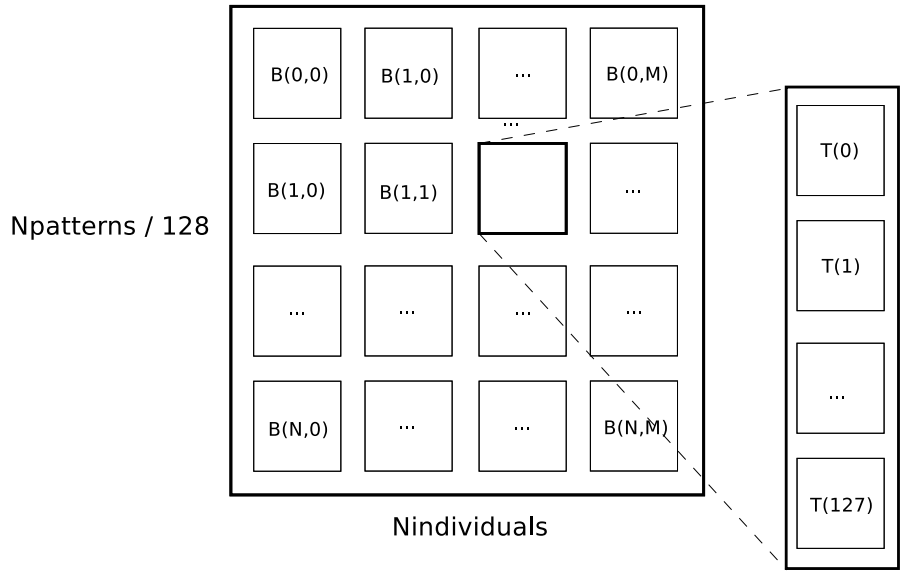

Figura 9.2: Matriz de bloques de hilos de ejecución.

Una primera aproximación al recuento paralelo de los resultados de todas las evaluaciones sería que el primer hilo sumase los (Nº patrones / 128) primeros valores, el segundo los (Nº patrones / 128) siguientes, etc. Sin embargo, esta forma es ineficiente en el sentido de que no favorece la coalescencia en los accesos a memoria (ver sección 4.2.4). Las posiciones de memoria a las que acceden los hilos del warp en ejecución se encuentran distanciadas (Nº patrones / 128) bytes, por lo que para servir dichos accesos el controlador de memoria las serializará resultando en múltiples accesos a memoria, uno para cada valor.

Una segunda aproximación sería que el primer hilo sumase los resultados de los valores en las posiciones de memoria 0, 128, 256... el segundo hilo las posiciones 1, 129, 257... hasta llegar al último hilo que sumaría las posiciones 127, 255, 383... De esta forma tenemos igualmente 128 hilos realizando semisumas en paralelo, pero

además los hilos del warp acceden a posiciones de memorias consecutivas de memoria por lo que empleando una única transferencia de memoria de 32 bytes es suficiente para traernos todos los datos de memoria necesarios. Se dice que este tipo de acceso es coalesciente.

La paralelización para cada individuo se realiza mediante la asignación a cada individuo de un bloque de hilos independiente. Este esquema con la segunda aproximación se refleja en la Figura 9.3.

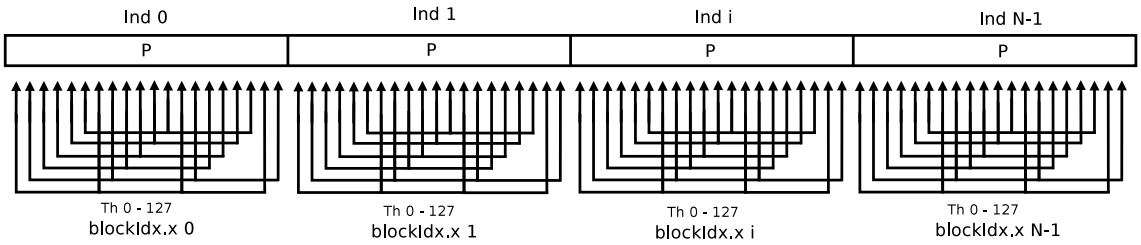

Figura 9.3: Modelo de reducción paralelo

A continuación procedemos a describir la implementación del evaluador para los tres algoritmos de clasificación propuestos en el capítulo 7.

9.2.1. Evaluador de Falco

La implementación nativa del evaluador de Falco requiere externalizar el método *evaluate()*. Mediante JNI se implementa la función nativa que deberá copiar los individuos a la memoria de la GPU, ejecutar los kernels de evaluación y reducción, copiar los resultados a memoria principal y por último, asignar el fitness a los individuos. La estructura de la secuencia de llamadas del evaluador nativo de Falco se representa en la Figura 9.4.

En el evaluador de Falco, si el antecedente de la regla se cumple se comprueba si la clase predicha coincide con el consecuente, en ese caso se considera un acierto si no, un fallo. Si el antecedente no se cumple y la clase predicha no coincide con el consecuente, se considera un acierto ya que dicha instancia no pertenece a la clase y así se predijo, en otro caso es un fallo.

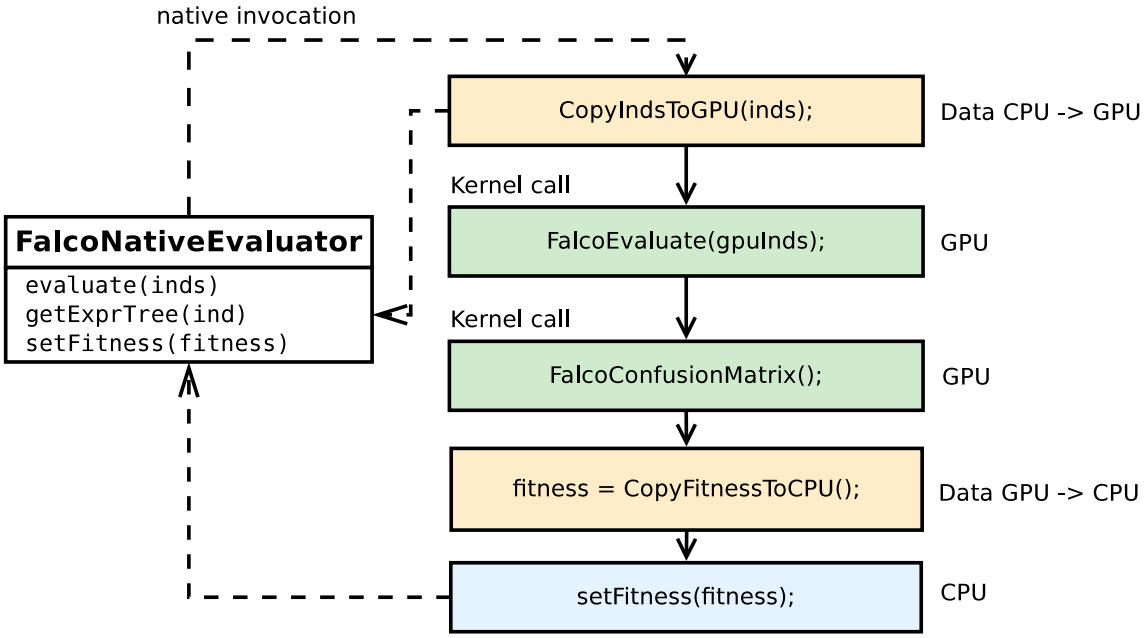

Figura 9.4: Estructura de las llamadas del evaluador de Falco.

```
1  __global__ void FalcoEvaluate(individuals) {
2    if(covers(rule,instance)) {
3        if( classifiedClass  == instancesClass[instance])
4            result[individual][instance] = HIT;
5        else
6            result[individual][instance] = FAIL;
7    }
8    else {
9        if( classifiedClass  != instancesClass[instance])
10           result[individual][instance] = HIT;
11       else
12           result[individual][instance] = FAIL;
13   }
14 }
```

El cálculo del fitness se realiza contando el número de aciertos y fallos en relación al número de instancias. La operación de reducción emplea 128 hilos para realizar las semisumas y posteriormente se realiza la suma total y se asigna el fitness.

```
__global__ void FalcoConfusionMatrix()
{
    // Performs the reduction of the thread corresponding values
    for(int instance = 0; instance < topInstance; i+=128)
    {
        MC[thread] += result[individual][instance];
    }

    // Calculates the final amount
    if(thread == 0)
    {
        int fails = 0;
        for(int i = 0; i < 128; i++)
        {
            fails += MC[i];
        }

        // Set the fitness to the individual
        fitness[individual] = 2*fails;
    }
}
```

9.2.2. Evaluador de Tan

La implementación nativa del evaluador de Tan requiere externalizar el método *evaluate()*. Mediante JNI se implementa la función nativa que deberá copiar los individuos a la memoria de la GPU, ejecutar los kernels de evaluación y reducción, copiar los resultados a memoria principal y por último, asignar el fitness a los individuos. La estructura de la secuencia de llamadas del evaluador nativo de Tan se representa en la Figura 9.5.

En el evaluador de Tan en vez de atender a aciertos y fallos en la predicción, se obtiene verdadero positivo en el caso de que el antecedente se cumpla y el consecuente coincida con la clase predicha, falso positivo en el caso de que el antecedente se cumpla pero el consecuente no coincida con la clase predicha, verdadero negativo en el caso de que el antecedente no se cumpla y el consecuente no coincida con la clase precicha y falso negativo en el caso de que el antecedente no se cumpla y el consecuente coincida con la clase predicha.

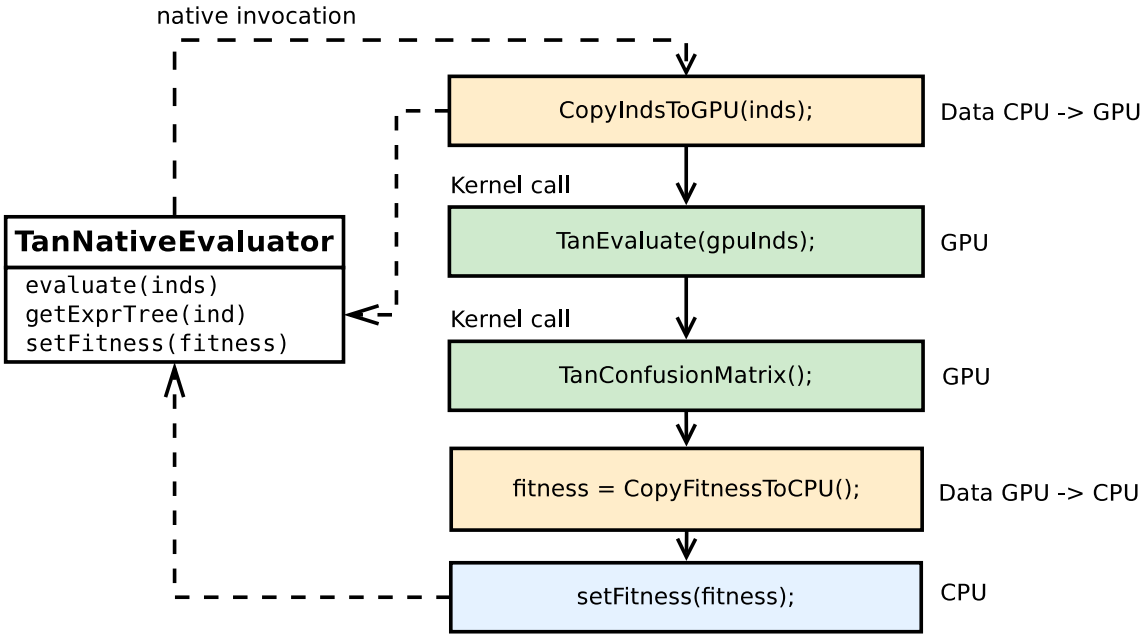

Figura 9.5: Estructura de las llamadas del evaluador de Tan.

```
__global__ void TanEvaluate(individuals) {
  if(covers(rule,instance))
  {
      if( classifiedClass  == instancesClass[instance])
          result[individual][instance] = TRUE_POSITIVE;
      else
          result[individual][instance] = FALSE_POSITIVE;
  }
  else
  {
      if( classifiedClass  != instancesClass[instance])
          result[individual][instance] = TRUE_NEGATIVE;
      else
          result[individual][instance] = FALSE_NEGATIVE;
  }
}
```

El cálculo del fitness se realiza contando el número de verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos. La operación de reducción emplea 128 hilos para realizar las semisumas y posteriormente se realiza la suma total y se asigna el fitness atendiendo a los parámetros $w1$ y $w2$ que ponderan el peso de la sensibilidad y la especificidad.

```
__global__ void TanConfusionMatrix()
{
    // Performs the reduction of the thread corresponding values
    for(int instance = 0; instance < topInstance; i+=128)
    {
        MC[thread + result[individual][instance]]++;
    }

    // Calculates the final amount
    if(thread < 4) {
        for(int i = 4; i < 512; i+=4) {
            MC[0] += MC[i];    // Number of true positives
            MC[1] += MC[i+1]; // Number of true negatives
            MC[2] += MC[i+2]; // Number of false positives
            MC[3] += MC[i+3]; // Number of false negatives
        }
    }
    if(thread == 0) {
        int tp = MC[0], tn = MC[1], fp = MC[2], fn = MC[3];

        // Set the fitness to the individual
        fitness[individual] = (tp/(tp+w1*fn)) * (tn/(tn+w2*fp));
    }
}
```

9.2.3. Evaluador de Freitas

La implementación nativa del evaluador de Freitas requiere externalizar el método *evaluate()*. Mediante JNI se implementa la función nativa que deberá copiar los individuos a la memoria de la GPU, ejecutar los kernels de evaluación y reducción, copiar los resultados a memoria principal y por último, asignar el fitness a los individuos. La estructura de la secuencia de llamadas del evaluador nativo de Freitas se representa en la Figura 9.6.

En el evaluador de Freitas se dispone de tantos vectores de resultados como número de clases tiene el conjunto de datos. Si el antecedente se cumple se asigna como verdadero positivo el resultado en el vector de resultados de la clase de la instancia y además se asigna como falso positivo en los vectores de resultados de todas las demás clases. Si el antecedente no se cumple se asigna como verdadero

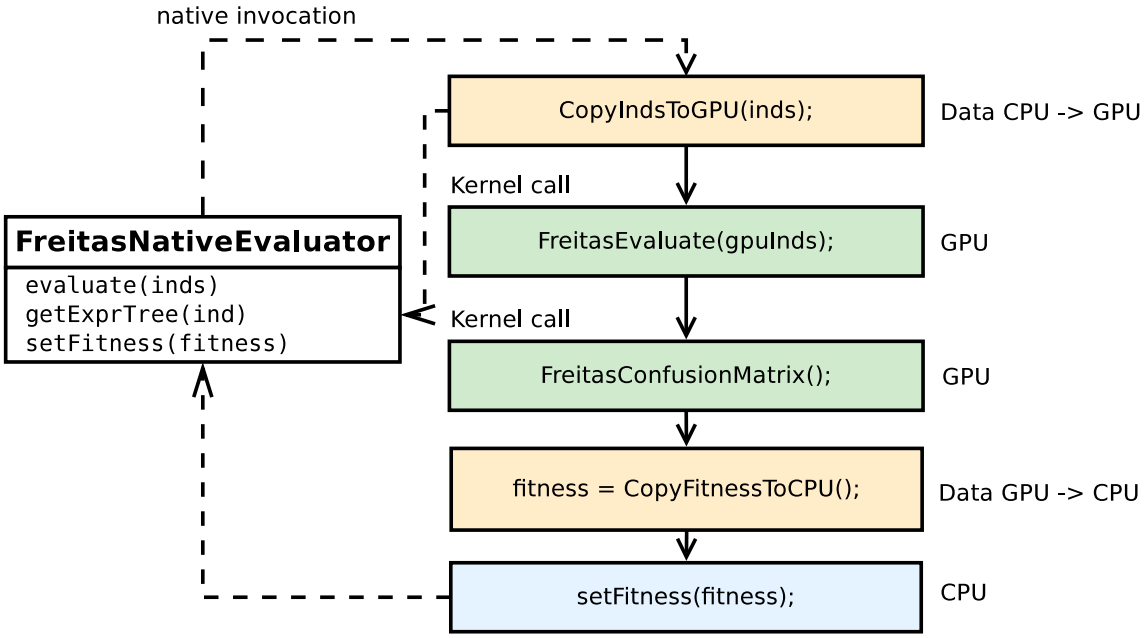

Figura 9.6: Estructura de las llamadas del evaluador de Freitas.

negativo el resultado en el vector de resultados de la clase de la instancia y además se asigna como falso negativo en los vectores de resultados de todas las demás clases.

```
1  __global__ void FreitasEvaluate(individuals)
2  {
3    if(covers(rule,instance))
4    {
5        for(int i = 0; i < numClasses; i++)
6            result[i][individual][instance] = FALSE_POSITIVE;
7
8        result[instancesClass[instance]][individual][instance] = TRUE_POSITIVE;
9    }
10   else
11   {
12       for(int i = 0; i < numClasses; i++)
13           result[i][individual][instance] = TRUE_NEGATIVE;
14
15       result[instancesClass[instance]][individual][instance] = FALSE_NEGATIVE;
16   }
17 }
```

El cálculo del fitness se realiza contando el número de verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos para cada clase. La operación de reducción emplea 128 hilos para realizar las semisumas y posteriormente se realiza

la suma total. Con la suma de estos valores se calcula la sensibilidad, especificidad y la mejor clase que predice cada regla. Este proceso debe repetirse para cada clase.

```
__global__ void FreitasConfusionMatrix()
{
    if (threadIdx.y == 0) {
        se[individual] = -1;
        sp[individual] = 1;
    }

    // Checks the best class for the individual
    for (int j = 0; j < numClasses; j++) {

        // Performs the reduction of the thread corresponding values
        for (int instance = 0; instance < topInstance; i+=128)
            MC[thread + result[j][individual][instance]]++;

        if (thread < 4) {
            for (int i = 4; i < 512; i+=4)
            {
                MC[0] += MC[i];    // Number of true positives
                MC[1] += MC[i+1]; // Number of true negatives
                MC[2] += MC[i+2]; // Number of false positives
                MC[3] += MC[i+3]; // Number of false negatives
            }
        }

        if (thread == 0) {
            int tp = MC[0], tn = MC[1], fp = MC[2], fn = MC[3];

            if (tp + fn == 0)
                seAux = 1;
            else
                seAux = tp / (tp + fn);

            if (tn + fp == 0)
                spAux = 1;
            else
                spAux = tn / (tn + fp);

            if (seAux*spAux == se[individual]*sp[individual])
                bestClass[individual] = j;

```

```
41            // If the actual class is best for the individual we keep it
42            if(seAux*spAux > se[individual]*sp[individual]) {
43               se[individual] = seAux;
44               sp[individual] = spAux;
45               bestClass[individual] = j;
46            }
47         }
48      }
49   }
```

10. EXPERIMENTACIÓN

El capítulo de experimentación tiene como finalidad dar una idea de las estrategias seguidas en el desarrollo del software y de los resultados obtenidos en las mismas. El proceso de experimentación consiste en someter al software a una evaluación para comprobar que se comporta de acuerdo a las especificaciones y poder obtener conclusiones de su ejecución y resultados. La información obtenida en la experimentación nos ayudará a definir los parámetros de los algoritmos para mejorar su ejecución.

10.1. Configuración software y hardware

En esta sección se detalla la configuración software y hardware de la máquina donde se ejecutará la experimentación.

Software

- Sistema Operativo GNU/Linux Ubuntu 9.10 kernel 2.6.33 x86_64
- Compilador de C/C++ GNU gcc 4.3.4
- JAVA SDK & RE 1.6.0_18 64
- NVIDIA CUDA SDK 3.0 64
- Eclipse 3.5.1
- JCLEC 4

Hardware

- Microprocesador Intel Core i7 920 2.6 GHz, 8 MB L3
- 12 GB DRAM 1333 MHz
- 2 GPUs NVIDIA GTX 285 2GB

Se emplea una tarjeta gráfica adicional NVIDIA 8400 GS para conectarla al monitor como señal de vídeo y permitir que las dos GTX 285 se dediquen exclusivamente a computación. Además se realizará una prueba inicial para determinar el Sistema Operativo en el que se ejecuta más rápido el algoritmo, que dependerá de la calidad de la implementación del gestor de memoria y de las comunicaciones con las GPUs. El tiempo de ejecución medio de los tres algoritmos con los parámetros por defecto se muestra en la Figura 10.1.

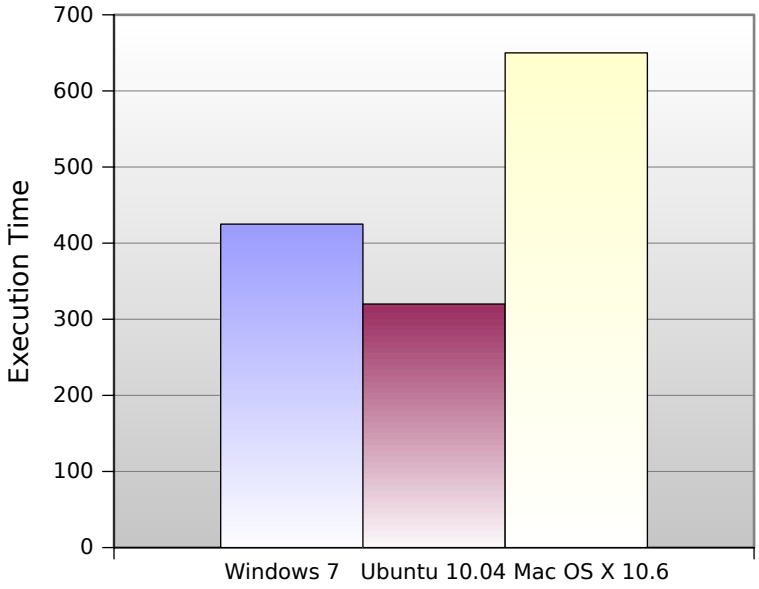

Figura 10.1: Comparación de una ejecución en los tres sistemas operativos.

El sistema operativo GNU/Linux Ubuntu 10.04 se muestra más rápido en la ejecución comparativa, por lo que centraremos la ejecución de la experimentación sobre esta plataforma.

10.2. Conjutos de datos

Se ha llevado a cabo experimentos sobre 10 conjuntos de datos de dominio público, procedentes del repositorio para aprendizaje automático de la UCI [20]. Estos conjuntos de datos presentan una buena diversidad respecto a distintas características, como número de ejemplos, número de clases, número de atributos y tipos de datos. El objetivo es analizar la influencia del tamaño del conjunto de datos sobre el rendimiento de la ejecución del algoritmo. A continuación se detallan las características de los conjuntos de datos empleados que se resumen en la Tabla 10.1.

Nombre	Nº patrones	Nº atributos	Nº clases
Adult	32561	14	2
Connect	67557	42	3
Contraceptive	1325	9	3
Iris	135	4	3
New-thyroid	193	5	3
Penbased	7494	16	10
Poker	25010	10	10
Shuttle	43500	9	7
Thyroid	3772	21	3
Poker-I	1000000	10	10

Tabla 10.1: Conjuntos de datos usados.

- Adult: conjunto de datos creado por Barry Becker en base a la extracción de registros del censo de EEUU en 1994. Su predicción consiste en determinar si una persona puede ganar más de 50.000 dólares al año. Se basa en un conjunto de 14 atributos categóricos (sexo, raza, estado civil, educación, etc.) y enteros (edad, horas de trabajo semanales, etc.).

- Connect: esta base de datos contiene todas las posiciones válidas en el juego de conecta-4 en una posición en la que no ha ganado ninguno de los jugadores y en el que el siguiente movimiento no sea forzoso. Sus 42 atributos se corresponden con cada una de las casillas del tablero, que pueden ser tomadas por cada uno de los jugadores o encontrarse en blanco. Se debe obtener cuál será el resultado de la partida.

- Contraceptive: el problema de este conjunto de datos es predecir el mejor método anticonceptivo para una mujer en base a sus características socioeconómicas y su entorno demográfico.

- Iris: ésta es quizás la base de datos más conocida en el reconocimiento de patrones. Consta de 3 clases de 50 instancias cada una que representan un tipo de planta iris. El clasificador debe predecir el tipo de la planta en base cuatro atributos: longitud y ancho del sépalo y del pétalo.

- New-thyroid: conjunto reducido de clasificación del funcionamiento normal, hipoactivo o hiperactivo de la glándula tiroides en base a 5 atributos.

- Penbased: conjunto de reconocimiento de la escritura a mano.

- Poker: conjunto de datos donde cada registro es una mano con cinco cartas de una baraja. El clasificador debe obtener la relación entre las cartas que determinan su valor en la jugada (pareja, trío, escalera, etc.).
- Shuttle: conjunto de datos procedente de mediciones del transbordador espacial. Se debe decicidir la ejecución de una acción de alguno de los instrumentos.
- Thyroid: conjunto mayor a new-thyroid donde se determina si un paciente clínico es hipertiroideo en base a 21 atributos.
- Poker-I: base de datos obtenida del conjunto Poker, invirtiendo los patrones de entrenamiento y test para obtener un número de instancias de aprendizaje superior al millón de patrones.

El aprendizaje a partir de datos no balanceados puede resultar espacialmente dificultoso. Se dice que un conjunto de datos está desbalanceado cuando presenta una desigualdad marcada en la distribución de las clases, es decir, que hay diferencias acusadas entre los porcentajes de ejemplos de las distintas clases. El problema surge porque, normalmente, los algoritmos de aprendizaje que se usan para construir los clasificadores tienden a centrarse en las clases mayoritarias y a pasar por alto las minoritarias. La consecuencia de este comportamiento es que el clasificadorno será capaz de clasificar correctamente los ejemplos correspondientes a las clases menos frecuentes.

La estrategia de aprendizaje no es la única cuestión que hay que afrontar cuando se trabaja con datos no balanceados. Otro punto fundamental es el de cómo medir la calidad de los clasificadores obtenidos. En clasificación, el rendimiento suele medirse en términos de la tasa de aciertos, es decir, la proporción de ejemplos correctamente clasificados. No obstante, es bien sabido que esta medida no es adecuada para juzgar el rendimiento del clasificador cuando los datos están desbalanceados. Si tuviésemos un problema de clasificación con dos clases que presentasen una proporciónn de 99/1 en el número de ejemplos, obtendríamos un porcentaje de aciertos del 99 % con un clasificador que se limitase a asignar todos los ejemplos a la clase mayoritaria. En este caso tendríamos un clasificador con una tasa de aciertos muy elevada pero que no clasificaría correctamente ninguno de los ejemplos de la clase minoritaria.

El único conjunto de datos no balanceado que disponemos es *Shuttle* donde el 80 % de las instancias pertenecen a la primera clase. La NASA, proveedora de la base de datos, exige que el clasificador obtenga una tasa de aciertos superior al

99 %, no sólo por el balanceo de datos sino por la naturaleza del conjunto de datos, la decisión de qué acción debe tomar el transbordador espacial ante los datos que le aportan sus instrumentos, resulta crítica para la vida de los astronautas.

El tamaño del conjunto de datos es hasta ahora el factor más relevante que limita la resolución de problemas de clasificación mediante métodos evolutivos debido a que el tiempo computacional requerido para clasificar grandes conjuntos de datos es desmesurado.

10.3. Parametrización del algoritmo

En esta sección se realizará un estudio de la influencia de los diferentes parámetros del algorimo evolutivo en el tiempo de ejecución y en la calidad de las soluciones.

10.3.1. Número de hilos por bloque

La ejecución de un kernel en la GPU requiere de la disposición de los hilos en bloques y de la configuración del tamaño de la matriz de bloques. El número máximo de hilos por bloque es de 512 y NVIDIA en su guía de programación recomienda tamaños de bloque que sean múltiplo del tamaño del warp para maximizar la ocupación de los multiprocesadores. Las GPUs disponibles para la experimentación tienen un tamaño del warp de 32 hilos por los que el número de hilos por bloque deberá ser múltiplo de 32.

En nuestro caso, el número de registros no es un factor que limite la ocupación del multiprocesador pues los kernels emplean como máximo 14 registros, es decir, aunque se usase 512 hilos por bloque y como el máximo de hilos por multiprocesador es 1024, podría ejecutarse concurrentemente 2 bloques de 512 hilos manteniendo una ocupación del multiprocesador máxima. 1024 hilos por 12 registros cada hilo resulta en una necesidad de registros inferior al número de registros disponibles. Sin embargo, es interesante reducir el número de hilos por bloque, siempre que se mantenga un grado de ocupación máximo, para aumentar el número de bloques que se pueden ejecutar concurrentemente en cada multiprocesador. Un mayor número de bloques concurrentes permitirá al despachador ocultar latencias de acceso a memoria de un bloque ejecutando código aritmético de otro bloque mientras se sirve el acceso a memoria.

La Figura 10.2 representa el grado de ocupación para los valores posibles de hilos por bloque. Concretamente un número de hilos por bloque de 128 resulta ideal para nuestro problema, garantizando un máximo de 8 bloques concurrentes por multiprocessador y un grado de ocupación máximo, pudiendo ejecutar 32 warps de 32 hilos, los 1024 hilos por multiprocesador máximos que permite la arquitectura. La ocupación desciende en algunos puntos de la gráfica debido a que no se pueden crear warps suficientes que ocupen todo el multiprocesador con el número de hilos por bloque dado. Por ejemplo, para 350 hilos por bloque hasta un máximo de 1024 hilos se podrá ejecutar 2 bloques por multiprocesador, es decir, los 700 hilos conforman 22 warps de 32 hilos de los 32 posibles, luego el grado de ocupación desciende.

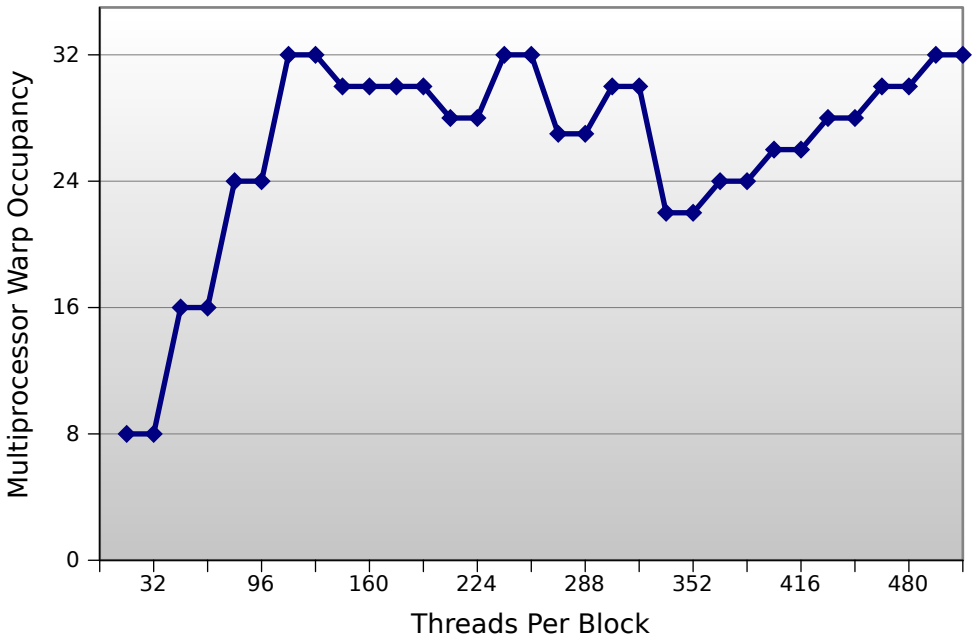

Figura 10.2: Ocupación del multiprocesador en base al número de hilos por bloque.

10.3.2. Tamaño de la población

El tamaño de la población es un factor clave en la generación de soluciones de calidad. Una mayor cantidad de individuos permitirá ampliar el espacio de búsqueda favoreciendo la diversidad y obteniendo un mayor número de buenas soluciones. Sin embargo, es el parámetro de configuración que más afecta al tiempo de ejecución en cada generación del algoritmo.

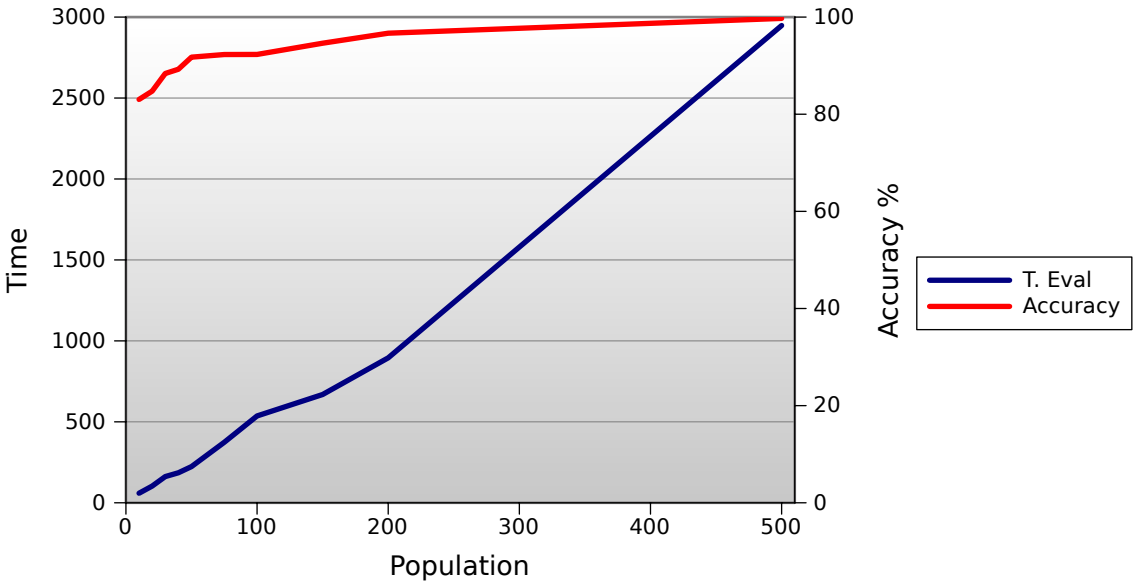

Figura 10.3: Índice de aciertos en función del tamaño de población.

Como se observa en la Figura 10.3 el tiempo de ejecución es una funcion lineal respecto al tamaño de población y para obtener soluciones de calidad se requiere un tamaño de población grande. Habitualmente, un tamaño de población de 200 puede ser suficiente, pero para la clasificación de grandes conjuntos de datos podemos aumentar el tamaño hasta 500 individuos para explorar mayores áreas del espacio de búsqueda si fuese necesario. Un valor de compromiso aceptable se sitúa en torno a los 250 individuos, éste será el tamaño de población para la ejecución de las pruebas.

10.3.3. Número de generaciones

Los algoritmos evolutivos mejoran sus individuos en cada generación aplicando operadores de cruce, mutación y selección para que los mejores individuos pasen a la siguiente generación. Por regla general, un mayor número de generaciones debería ofrecer individuos de mayor calidad.

La Figura 10.4 representa que el tiempo de ejecución es una funcion lineal respecto al número de generaciones. Sin embargo, un número de generaciones excesivamente alto no garantiza la convergencia a una solución excelente. Habitualmente, con un número de generaciones superior a 100 se alcanzan buenas soluciones, por lo que éste será el valor del parámetro en la ejecución de las pruebas.

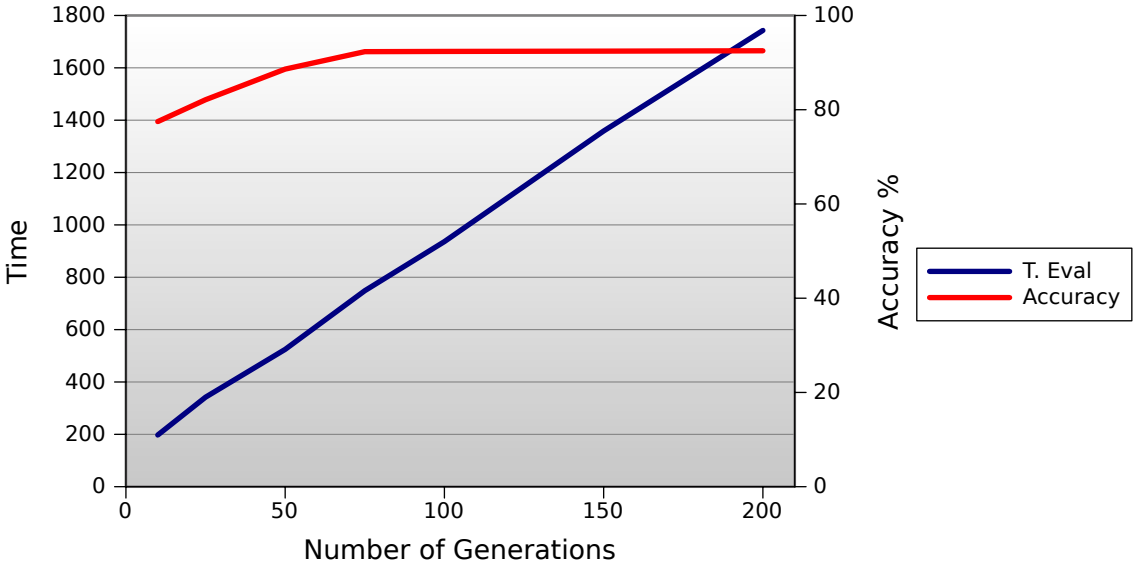

Figura 10.4: Índice de aciertos en función del número de generaciones.

Para criterios prácticos, es muy útil la definición de convergencia. Si el algoritmo genético ha sido correctamente implementado, la población evolucionará a lo largo de las generaciones sucesivas de tal manera que la adaptación media extendida a todos los individuos de la población, así como la adaptación del mejor individuo se irán incrementando hacia el óptimo global. El concepto de convergencia está relacionado con la progresión hacia la uniformidad: un gen ha convergido cuando al menos el 95 % de los individuos de la población comparten el mismo valor para dicho gen. Se dice que la población converge cuando todos los genes han convergido. Se puede generalizar dicha definición al caso en que al menos unos pocos de los individuos de la población hayan convergido. Uno de los mayores enemigos de los algoritmos genéticos es la convergencia prematura ya que si no se converge a la solución, el algoritmo se cierra a explorar otras zonas del espacio de búsqueda donde pueda encontrar soluciones mejores.

Se puede concluir con la experimentación de los dos últimos parámetros que determinan la calidad de las soluciones, que para obtener un alta tasa de acierto en grandes conjuntos de datos se requerirá de una población de gran tamaño que evolucione en un elevado número de generaciones y por lo tanto el tiempo de ejecución será determinante. Ésta es la oportunidad de las GPUs para mostrar su potencial y acelerar el proceso en problemas de compleja dimensionalidad.

10.3.4. Probabilidad de cruce

En este apartado analizaremos cómo influye la probabilidad de cruce en los datos obtenidos por los algoritmos. Interesa una probabilidad de cruce relativamente alta que favorezca la transmisión de suficiente información genética a la descendencia.

Probabilidad	Falco	Tan	Freitas
0,6	93 %	93 %	53 %
0,65	93 %	100 %	73 %
0,7	93 %	100 %	80 %
0,75	93 %	93 %	73 %
0,8	93 %	67 %	67 %

Tabla 10.2: Aciertos del clasificador en función de la probabilidad de cruce.

La prueba se realiza variando la probabilidad de cruce y manteniendo constantes el resto de parámetros. Los resultados de la experimentación reflejados en la Tabla 10.2 aportan que una probabilidad de cruce de 0,7 resulta la más adecuada.

10.3.5. Probabilidad de mutación

En este apartado analizaremos cómo influye la probabilidad de mutación en los datos obtenidos por los algoritmos de Falco y Tan, ya que Freitas carece de operador de mutación. Cuando incrementamos la probabilidad de mutación estamos introduciendo diversidad en la población y por tanto estaremos abriendo el espacio de búsqueda, aunque un valor excesivamente alto provocaría la destrucción de la información genética y pasaría a ser una aleatorización.

Probabilidad	Falco	Tan
0,1	93 %	93 %
0,15	93 %	93 %
0,2	93 %	93 %
0,25	93 %	100 %
0,3	93 %	93 %

Tabla 10.3: Aciertos del clasificador en función de la probabilidad de mutación.

La prueba se realiza variando la probabilidad de mutación y manteniendo constantes el resto de parámetros. Los resultados de la experimentación reflejados en

la Tabla 10.3 aportan que una probabilidad de mutación de 0,25 resulta la más adecuada.

10.3.6. Conclusiones de la experimentación

En este capítulo se ha experimentado con los parámetros que influyen en la calidad de las soluciones del clasificador y en la velocidad de ejecución del algoritmo. Los resultados indican que para la generación de individuos aceptables, el proceso de evolución requiere de un tamaño de población considerable y de un número de generaciones suficiente. Es interesante contar con una probabilidad de cruce alta y una probabilidad de mutación baja de los operadores genéticos. El número de hilos por bloque óptimo para nuestros kernels resulta de 128 individuos por bloque, aunque se espera que este valor aumente en arquitecturas futuras con un mayor grado de paralelismo y un número de unidades de ejecución. No obstante, la implementación está preparada para escalar a un mayor número de hilos por bloque. Las baterías de pruebas para estudiar la aceleración de los algoritmos se realizarán con los parámetros que maximicen la calidad de las soluciones buscando minimizar el tiempo de ejecución.

11. RESULTADOS

En este capítulo se detallan los resultados de las ejecuciones de los algoritmos con el objetivo de analizar los resultados, su aceleración y poder obtener conclusiones que apoyen los objetivos del proyecto. Los tres algoritmos se ejecutan en una batería de pruebas con los diferentes conjuntos de datos empleando los parámetros estudiados en la experimentación.

11.1. Algoritmo de Falco

El algoritmo de Falco ha sido testado con todos los conjuntos de datos y se ha obtenido los siguientes tiempos de la fase de evaluación, expresados en milisegundos por columnas en la Tabla 11.1 y en la Figura 11.1, para un una ejecución con un tamaño de población igual a 250 que garantice buenas soluciones. Cada base de datos se prueba con diferentes configuraciones: la primera columa representa la versión iterativa original en Java, la segunda externaliza la fase de evaluación en un método iterativo pero nativo escrito en C, en la tercera columna la evaluación se divide en dos hilos nativos en el que cada uno evalúa la mitad de la población, la cuarta se corresponde con la evaluación nativa empleando 4 hilos para su ejecución en los 4 núcleos del microprocesador, las dos siguientes reflejan los tiempos de ejecución externalizando la evaluación en una y dos tarjetas gráficas respectivamente.

Los tiempos de ejecución se relacionan en la Tabla 11.2 que representa la aceleración de nuestra propuesta con respecto al código Java y también se ilustran en la gráfica de la Figura 11.2.

Dataset	Java	C	2C	4C	1 GPU	2 GPUs
Adult	63254	59569	30648	16319	639	309
Connect	211073	204508	104306	53869	1665	889
Contraceptive	2900	3003	1537	823	85	45
Iris	316	693	356	202	71	38
New-thyroid	471	1003	520	289	74	48
Penbased	77706	87946	45039	23485	922	516
Poker	211755	216976	111163	58562	2043	1093
Shuttle	220632	251739	130428	69910	2411	1328
Thyroid	9057	14863	7697	4029	240	129
Poker-I	8682402	8947525	4612126	2427434	77221	41268

Tabla 11.1: Tiempos de ejecución del algoritmo de Falco.

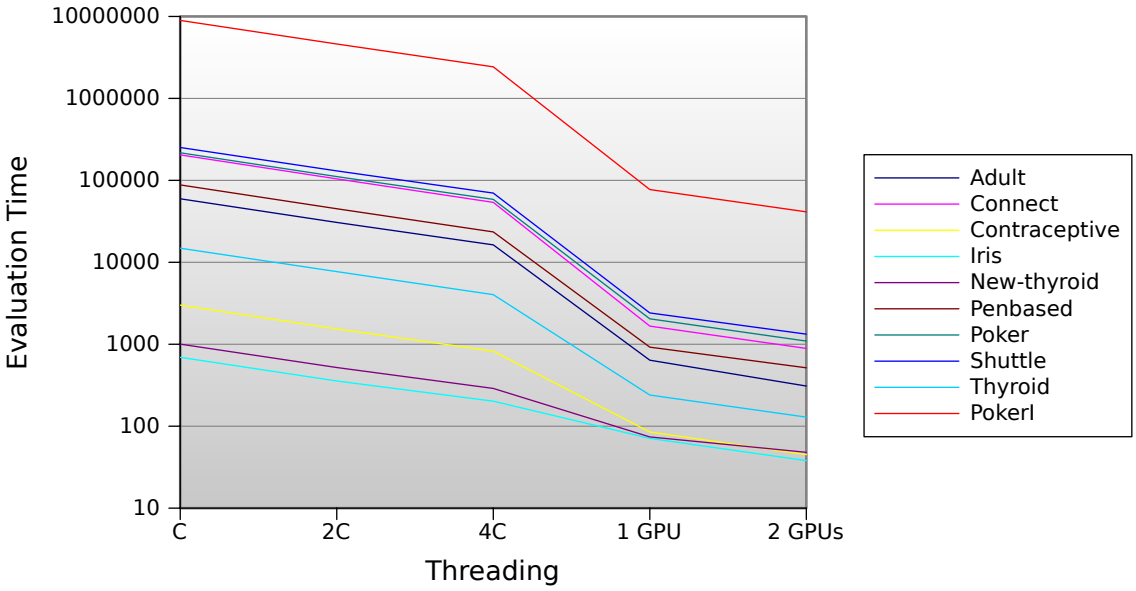

Figura 11.1: Tiempos de ejecución del algoritmo de Falco.

En algunos casos, la evaluación externa en código nativo resulta ineficiente para ciertos conjuntos de datos con el algoritmo de Falco. Esto es debido a que puede suponer mayor coste el traslado de los datos desde la memoria de la máquina virtual Java hasta el espacio de memoria nativo que realizar la evaluación desde la propia máquina virtual. Sin embargo, en todos los casos independientemente del tamaño del conjunto de datos, la evaluación nativa en GPU siempre resulta considerablemente más rápida que en Java.

Dataset	C	2C	4C	1 GPU	2 GPUs
Adult	1.06	2.06	3.88	98.99	204.71
Connect	1.03	2.02	3.92	126.77	237.43
Contraceptive	0.97	1.89	3.52	34.12	64.44
Iris	0.46	0.89	1.56	4.45	8.32
New-thyroid	0.47	0.91	1.63	6.36	9.81
Penbased	0.88	1.73	3.31	84.28	150.59
Poker	0.98	1.90	3.62	103.65	193.74
Shuttle	0.88	1.69	3.16	91.51	166.14
Thyroid	0.61	1.18	2.25	37.74	70.21
Poker-I	0.97	1.88	3.58	112.44	210.39

Tabla 11.2: Aceleración del algoritmo de Falco.

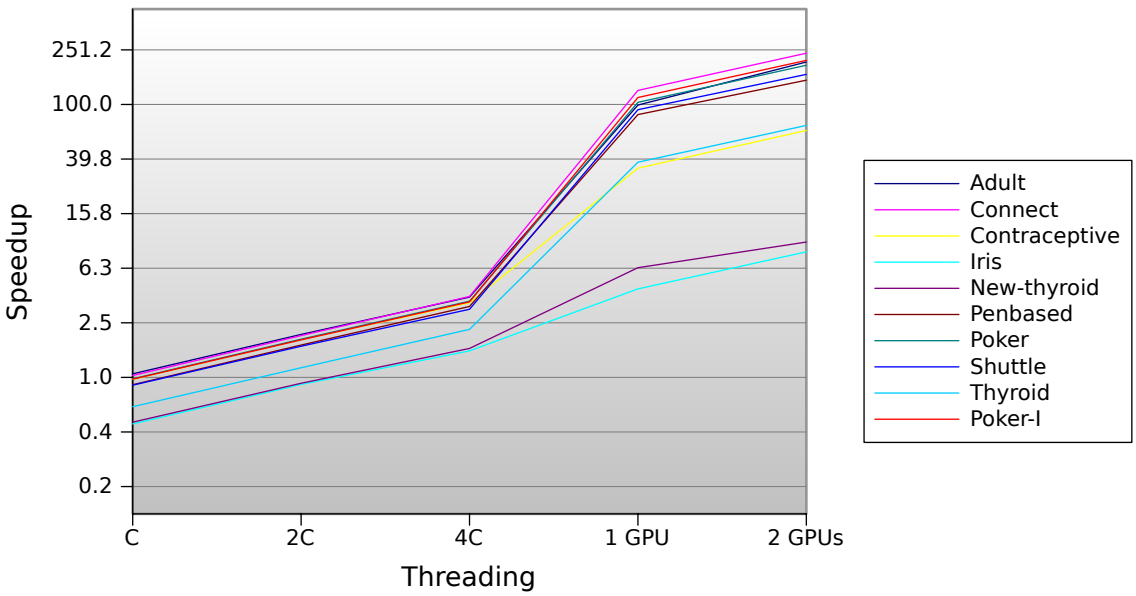

Figura 11.2: Aceleración del algoritmo de Falco.

Concretamente, la aceleración es mayor en los grandes conjuntos de datos alcanzando cotas de hasta 237 veces más rápido, lo que sin duda es muy buena noticia. La escalabilidad de la propuesta es idónea, ya que duplicar el número de hilos o de tarjetas gráficas supone prácticamente reducir a la mitad el tiempo de ejecución.

11.2. Algoritmo de Tan

El algoritmo de Tan ha sido testado con todos los conjuntos de datos y se ha obtenido los siguientes tiempos de la fase de evaluación, expresados en milisegundos en la Tabla 11.3 y en la Figura 11.3, para un una ejecución con un tamaño de población igual a 250 que garantice buenas soluciones. Cada base de datos se prueba con diferentes configuraciones: la primera columa representa la versión iterativa original en Java, la segunda externaliza la fase de evaluación en un método iterativo pero nativo escrito en C, en la tercera columna la evaluación se divide en dos hilos nativos en el que cada uno evalúa la mitad de la población, la cuarta se corresponde con la evaluación nativa empleando 4 hilos para su ejecución en los 4 núcleos del microprocesador, las dos siguientes reflejan los tiempos de ejecución externalizando la evaluación en una y dos tarjetas gráficas respectivamente.

Dataset	Java	C	2C	4C	1 GPU	2 GPUs
Adult	291800	207230	108645	58110	2169	1066
Connect	926957	496148	260776	139241	4835	2471
Contraceptive	16231	14259	7299	3849	309	169
Iris	1191	2412	1338	733	248	126
New-thyroid	2651	5793	3015	1596	288	167
Penbased	368035	310001	160422	83509	3360	1737
Poker	1266653	923497	477094	249826	8512	4335
Shuttle	1391059	1554389	803297	418066	12948	6570
Thyroid	60470	96521	49515	25649	1174	629
Poker-I	I50004099	39435765	20223469	10814689	344887	175962

Tabla 11.3: Tiempos de ejecución del algoritmo de Tan.

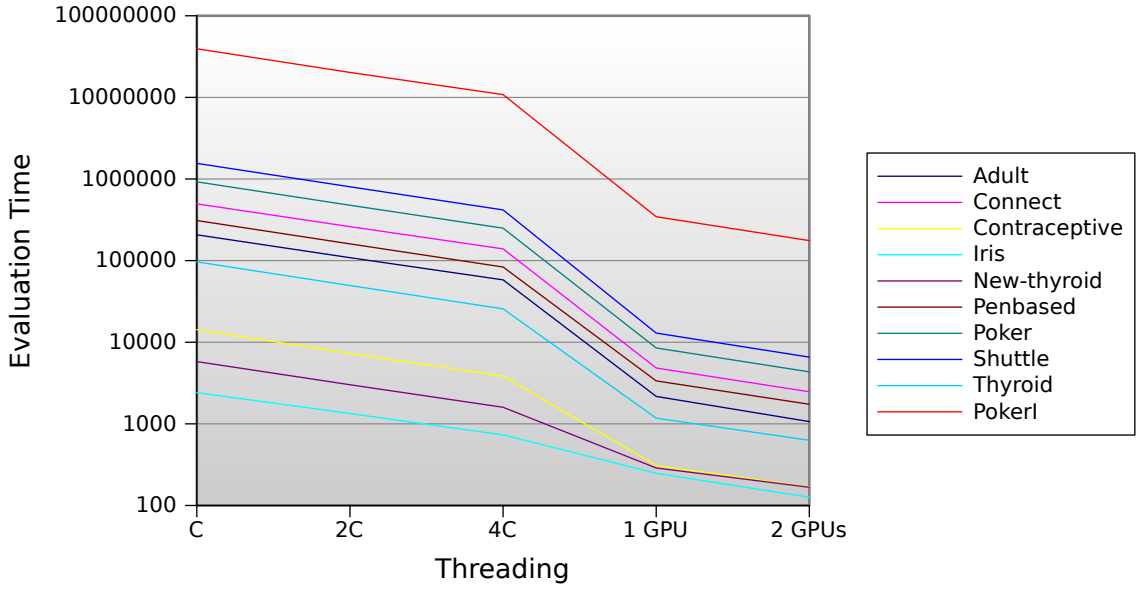

Figura 11.3: Tiempos de ejecución del algoritmo de Tan.

Los tiempos de ejecución se relacionan en la Tabla 11.4 que representa la aceleración de nuestra propuesta con respecto al código Java y también se ilustran en la gráfica de la Figura 11.4.

Dataset	C	2C	4C	1 GPU	2 GPUs
Adult	1.41	2.69	5.02	134.53	273.73
Connect	1.87	3.55	6.66	191.72	375.13
Contraceptive	1.14	2.22	4.22	52.53	96.04
Iris	0.49	0.89	1.62	4.80	9.45
New-thyroid	0.46	0.88	1.66	9.20	15.87
Penbased	1.19	2.29	4.41	109.53	211.88
Poker	1.37	2.65	5.07	148.81	292.19
Shuttle	0.89	1.73	3.33	107.43	211.73
Thyroid	0.63	1.22	2.36	51.51	96.14
Poker-I	1.27	2.47	4.62	144.99	284.18

Tabla 11.4: Aceleración del algoritmo de Tan.

En algunos casos, la evaluación externa en código nativo también resulta ineficiente para ciertos conjuntos de datos, pero se reduce el número de casos en el que esto sucede. En todos los casos independientemente del tamaño del conjunto de datos, la evaluación nativa en GPU siempre resulta considerablemente más rápida que en Java. Concretamente, la aceleración es mayor en los grandes conjuntos de

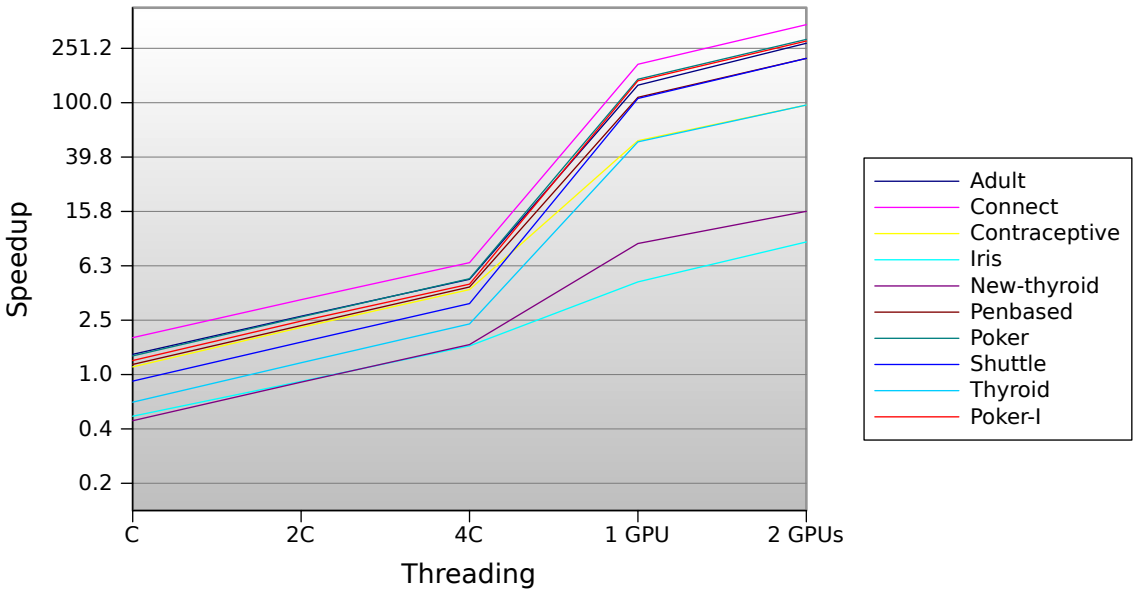

Figura 11.4: Aceleración del algoritmo de Tan.

datos alcanzando cotas de hasta 375 veces más rápido. La escalabilidad de la propuesta es idónea, ya que duplicar el número de hilos o de tarjetas gráficas supone prácticamente reducir a la mitad el tiempo de ejecución.

11.3. Algoritmo de Freitas

El algoritmo de Freitas ha sido testado con todos los conjuntos de datos y se ha obtenido los siguientes tiempos de la fase de evaluación, expresados en milisegundos en la Tabla 11.5 y en la Figura 11.5, para un una ejecución con un tamaño de población igual a 250 que garantice buenas soluciones. Cada base de datos se prueba con diferentes configuraciones: la primera columa representa la versión iterativa original en Java, la segunda externaliza la fase de evaluación en un método iterativo pero nativo escrito en C, en la tercera columna la evaluación se divide en dos hilos nativos en el que cada uno evalúa la mitad de la población, la cuarta se corresponde con la evaluación nativa empleando 4 hilos para su ejecución en los 4 núcleos del microprocesador, las dos siguientes reflejan los tiempos de ejecución externalizando la evaluación en una y dos tarjetas gráficas respectivamente.

Los tiempos de ejecución se relacionan en la Tabla 11.6 que representa la aceleración de nuestra propuesta con respecto al código Java y también se ilustran en la gráfica de la Figura 11.6.

Dataset	Java	C	2C	4C	1 GPU	2 GPUs
Adult	20476	14695	7478	3915	315	139
Connect	32242	32954	17076	8945	585	302
Contraceptive	564	536	271	145	34	16
Iris	58	110	58	32	28	12
New-thyroid	80	153	80	43	27	12
Penbased	3216	3244	1691	884	170	86
Poker	11127	10358	5265	2793	455	226
Shuttle	19706	19000	9772	5165	562	287
Thyroid	1852	2980	1539	808	72	37
Poker-I	444249	445752	229769	138036	17873	9118

Tabla 11.5: Tiempos de ejecución del algoritmo de Freitas.

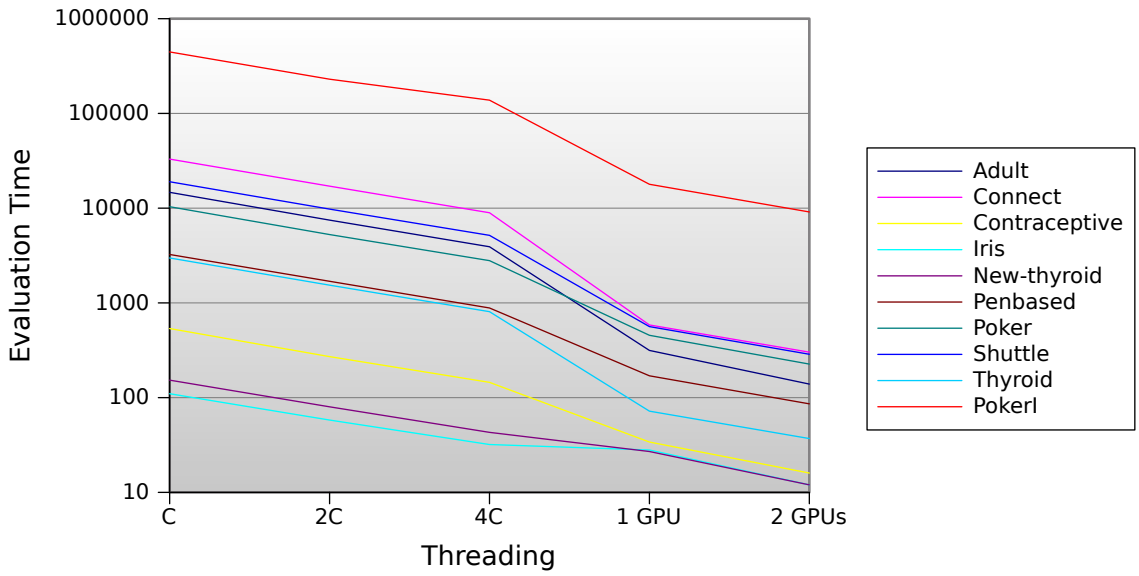

Figura 11.5: Tiempos de ejecución del algoritmo de Freitas.

En algunos casos, la evaluación externa en código nativo también resulta ineficiente para ciertos conjuntos de datos, pero también se reduce el número de casos en el que esto sucede. En todos los casos independientemente del tamaño del conjunto de datos, la evaluación nativa en GPU siempre resulta considerablemente más rápida que en Java. Concretamente, la aceleración es mayor en los grandes conjuntos de datos alcanzando cotas de hasta 106 veces más rápido. Sin embargo son menores que las de los algoritmos de Falco y Tan debido a la menor complejidad de la función de evaluación. La escalabilidad de la propuesta es idónea, ya que duplicar el número de hilos o de tarjetas gráficas supone reducir a la mitad el tiempo de ejecución.

Dataset	C	2C	4C	1 GPU	2 GPUs
Adult	1.39	2.74	5.23	65.00	147.31
Connect	0.98	1.89	3.60	55.11	106.76
Contraceptive	1.05	2.08	3.89	16.59	35.25
Iris	0.53	1.00	1.81	2.07	4.83
New-thyroid	0.52	1.00	1.86	2.96	6.67
Penbased	0.99	1.90	3.64	18.92	37.40
Poker	1.07	2.11	3.98	24.45	49.23
Shuttle	1.04	2.02	3.82	35.06	68.66
Thyroid	0.62	1.20	2.29	25.72	50.05
Poker-I	1.00	1.93	3.22	24.86	48.72

Tabla 11.6: Aceleración del algoritmo de Freitas.

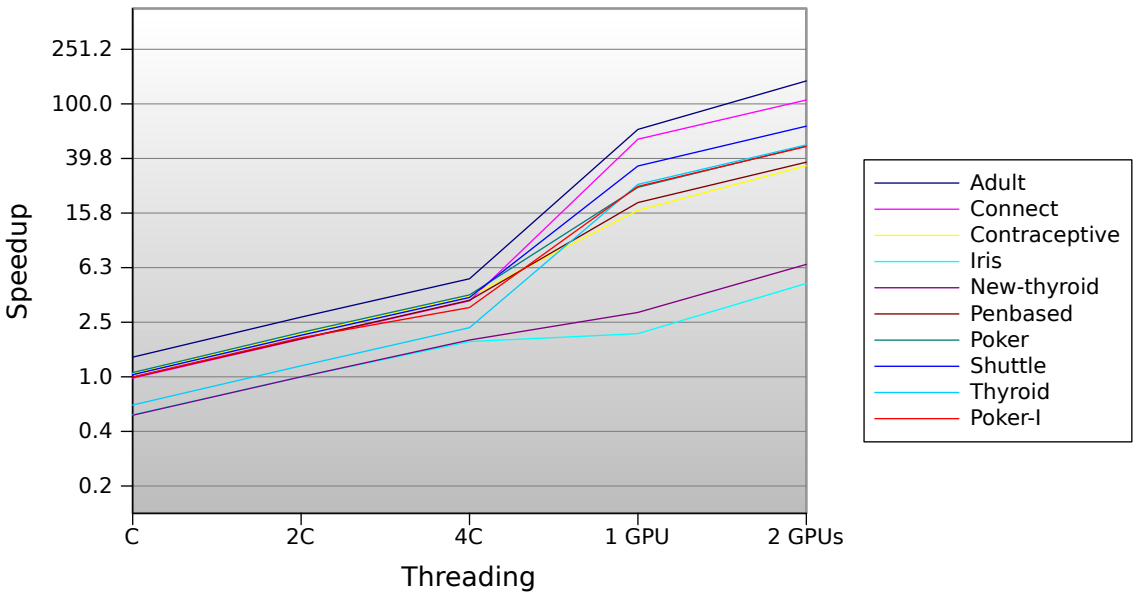

Figura 11.6: Aceleración del algoritmo de Freitas.

11.4. Comparativa de los algoritmos

La comparativa de los tiempos de ejecución y de la aceleración para cada conjunto de datos y algoritmo empleando 2 GPUs se ilustran en las Tablas 11.7 y 11.8 respectivamente. Las tarjetas gráficas se muestran más eficientes en los grandes conjuntos de datos donde pueden demostrar su potencial. La función de evaluación del algoritmo de Tan es más compleja que la del resto, por lo que se requiere la ejecución de un mayor número de instrucciones y accesos a memoria. Son estos casos donde las GPUs maximizan su potencial de ejecución paralela.

Dataset	Falco	Tan	Freitas
Adult	309	1066	139
Connect	889	2471	302
Contraceptive	45	169	16
Iris	38	126	12
New-thyroid	48	167	12
Penbased	516	1737	86
Poker	1093	4335	226
Shuttle	1328	6570	287
Thyroid	129	629	37
Poker-I	41268	175962	9118

Tabla 11.7: Comparativa del tiempo de ejecución obtenido en cada algoritmo.

Dataset	Falco	Tan	Freitas
Adult	204.706	273.734	147.309
Connect	237.427	375.134	106.762
Contraceptive	64.4444	96.0414	35.25
Iris	8.31579	9.45238	4.83333
New-thyroid	9.8125	15.8743	6.66667
Penbased	150.593	211.880	37.3953
Poker	193.737	292.192	49.2345
Shuttle	166.139	211.729	68.6620
Thyroid	70.2093	96.1367	50.0541
Poker-I	210.391	284.176	48.7222

Tabla 11.8: Comparativa de la aceleración obtenida en cada algoritmo.

En cuanto a la calidad de las soluciones generadas por los algoritmos, donde se muestra el porcentaje de predicciones correctas en la Tabla 11.9, se observa que el algoritmo de Tan genera en la mayoría de los casos mejores reglas que clasifiquen los conjuntos de datos. Sin embargo, aunque el algoritmo de Tan genera mejores individuos no hay que olvidar que también requiere un mayor tiempo de cómputo, pero la calidad de los resultados lo merece. Hay que tener en cuenta que los resultados también dependen de una semilla de inicialización, por lo que para una experimentación más completa habría que realizar al menos 10 ejecuciones de cada caso con distintas semillas para poder obtener conclusiones determinantes sobre la calidad del algoritmo. No obstante, sí nos sirve para justificar que el mejor algoritmo de los propuestos es el que más tiempo de cómputo requiere y por lo tanto es donde más nos interesa acelerar su ejecución.

Dataset	Falco	Tan	Freitas
Adult	81.13 %	71.00 %	45.97 %
Connect	12.28 %	65.83 %	18.20 %
Contraceptive	39.19 %	37.84 %	46.62 %
Iris	93.33 %	100.00 %	93.33 %
New-thyroid	86.36 %	95.45 %	95.45 %
Penbased	71.87 %	79.30 %	41.25 %
Poker	50.49 %	15.90 %	16.41 %
Shuttle	99.61 %	94.91 %	51.84 %
Thyroid	91.72 %	96.24 %	41.48 %

Tabla 11.9: Comparativa de la calidad de las soluciones de los algoritmos.

12. CONCLUSIONES Y TRABAJO FUTURO

Una vez concluido el desarrollo y la experimentación del proyecto, se realizará una exposición de las conclusiones que se han extraído. Estas conclusiones irán en relación con los objetivos planteados al comienzo del desarrollo descrito en el presente documento y con los resultados obtenidos durante la fase de experimentación.

12.1. Conclusiones

La clasificación de grandes conjuntos de datos es una operación que demanda mucho tiempo y recursos computacionales conforme la dimensión del problema aumenta. La paralelización de la fase de evaluación de la población consigue acelerar la ejecución del algoritmo. El uso de hilos permite dividir los individuos a evaluar en subconjuntos que pueden ejecutarse en paralelo en cada uno de los núcleos del microprocesador. Las CPUs actuales disponen de hasta 4 núcleos, por lo que no se puede acelerar más allá. Sin embargo, las GPUs han evolucionado a una arquitectura con un número masivo de núcleos que bajo un modelo SIMD, pueden ejecutar millones de hilos de forma concurrente.

El modelo de ejecución de las GPUs, junto con las necesidades computacionales de la evaluación de los individuos, crea un entorno de ejecución ideal donde las GPUs pueden mostrar todo su potencial. Especialmente, su desempeño es el más indicado en casos en los que se trate con problemas de gran dimensionalidad y bases de datos con un elevado número de patrones, que hasta ahora se podría tardar semanas e incluso meses en resolver. Nuestra propuesta con GPUs permite reducir dicho tiempo a unas pocas horas bajo un modelo de paralelización masiva que aprovecha las ventajas de la paralelización de grano fino y de grano grueso para alcanzar una buena escalabilidad conforme se aumenta el número de GPUs.

La Figura 12.1 resume los resultados experimentales generando un mapa de superficie de la aceleración obtenida en función del número de individuos de la población y del tamaño del conjunto de datos.

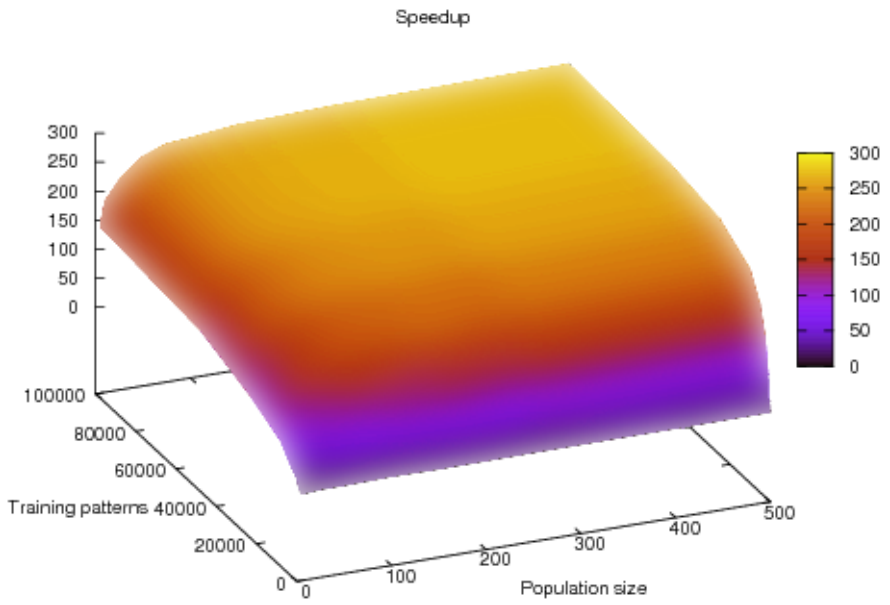

Figura 12.1: Aceleración en función del tamaño de la población y del dataset.

12.2. Trabajo futuro

Los algoritmos genéticos son interesantes de paralelizar desde la perspectiva de evolucionar en paralelo una población de individuos. Sin embargo, la visión clásica del algoritmo genético no es completamente paralelizable debido a la necesidad de poseer un control global en ciertas etapas como la selección y el cruce que requieren de la serialización de la ejecución de un trozo de código y forzar interacciones entre procesadores remotos, lo que origina un alto costo de comunicaciones.

Se han presentado varias propuestas para superar estas limitaciones alcanzando excelentes resultados. Los diferentes modelos empleados para distribuir la computación y facilitar la paralelización se centran en dos aproximaciones [13]:

- El modelo de islas donde varias subpoblaciones aisladas evolucionan en paralelo y periódicamente intercambian sus mejores individuos entre las islas vecinas.
- El modelo de vecindad donde se evoluciona una única población y cada individuo se coloca en una celda de una matriz. La selección y el cruce se aplican solamente entre individuos vecinos de la matriz de acuerdo a un criterio predefinido de vecindad.

Estos dos modelos se ofrecen a ser utilizados en arquitecturas paralelas MIMD en el caso del modelo de islas y SIMD en el de vecindad. La GPU resuelve por naturaleza el modelo SIMD y la escalabilidad en número de GPUs otorga el modelo MIMD. Por lo tanto, podemos combinar ambas perspectivas para poder desarrollar múltiples modelos de algoritmos paralelos y distribuidos donde se aproveche la paralelización de hilos en GPU, el uso de varias GPUs y a su vez la distribución del cómputo en múltiples máquinas conectadas en red con dichas GPUs. Esto abre la puerta de un buen nicho de investigación para el trabajo futuro.

BIBLIOGRAFÍA

[1] A. A. Freitas, "Data Mining and Knowledge Discovery with Evolutionary Algorithms", Springer-Verlag, 2002.

[2] A. Tsakonas, "A comparison of classification accuracy of four Genetic Programming-evolved intelligent structures", *Information Sciences*, 176 (6), 2006, pp. 691-724.

[3] C. C. Bojarczuk, H. S. Lopes, A. A. Freitas, and E. L. Michalkiewicz, "A constrained-syntax Genetic Programming system for discovering classification rules: application to medical data sets", *Artificial Intelligence in Medicine*, 30 (1), 2004, pp. 27-48.

[4] C. Romero, J. R. Romero, J. M. Luna, S. Ventura, "Mining Rare Association Rules from e-Learning Data", *Educational Data Mining 2010*, 2010.

[5] D. Chitty, 2007. "A data parallel approach to Genetic Programming using programmable graphics hardware", *GECCO'07: Proceedings of the Conference on Genetic and Evolutionary Computing*, 2007, pp. 1566-1573.

[6] D. Kirk, Wen-mei W. Hwu, John Stratton, "Reductions and Their Implementation", University of Illinois, Urbana-Champaign, 2009.

[7] D. Robilliard, V. Marion-Poty, C. Fonlupt, "Genetic programming on graphics processing units", *Genetic Programming and Evolvable Machines*, 10 (4), 2009, pp. 447-471.

[8] General-Purpose Computation on Graphics Hardware, *GPGPU*, http://www.gpgpu.org . Última visita: Junio 2010.

[9] Genetic Programming On General Purpose Graphics Processing Units, *GPGPGPU*, http://www.gpgpgpu.com . Última visita: Junio 2010.

[10] G. Booch, J. Rumbaugh, I. Jacobson, "El Lenguaje Unificado de Modelado", Ed. Addison Wesley Iberoamericana. Madrid. 1999.

[11] I. De Falco, A. Della Cioppa, and E. Tarantino, "Discovering interesting classification rules with Genetic Programming", *Applied Soft Computing Journal*, 1 (4), 2002, pp. 257-269.

[12] J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992.

[13] J. Stender, "Parallel Genetic Algorithms: Theory and Applications", IOS Press, 1993, pp. 10-25.

[14] K. C. Tan, A. Tay, T. H. Lee and C. M. Heng, "Mining multiple comprehensible classification rules using Genetic Programming", *CEC '02: Proceedings of the Evolutionary Computation on 2002*, 2002, pp. 1302-1307.

[15] K. Deb, "A population-based algorithm-generator for real-parameter optimization", *Soft Computing*, 9 (4), 2005, pp. 236-253.

[16] M. L. Wong and K. S. Leung, "Data Mining Using Grammar Based Genetic Programming and Applications", London: Kluwer Academic Publishers, 2000.

[17] NVIDIA Programming and Best Practices Guide 3.0, *NVIDIA CUDA Zone*, http://www.nvidia.com/object/cuda_home.html

[18] P. A. Whigham, "Gramatical Bias for Evolutionary Learning", Tesis doctoral, School of Computer Science - University College - University of New South Wales - Australian Defence Force Academy, 1996.

[19] P. González, C. Romero, C. Hervás and S. Ventura, "Programación Genética Gramatical para el Descubrimiento de Reglas de Clasificación", *Cuarto Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB'05)*, Granada, 2005, pp. 653-661.

[20] P. M. Murphy and D. W. Aha, "UCI Repository of Machine Learning Databases", Department of Information and Computer Science, University of California, Irvine, California, 1994. [http://www.ics.uci.isu/ mlearn/MLRepository.html]

[21] P. Ngan, M. Wong, W. Lam, K. Leung, J. Cheng. "Medical data mining using evolutionary computation", *Artificial Intelligence in Medicine*, 16, 1999, pp. 73-96.

[22] S. Harding, W. Banzhaf "Fast Genetic Programming and artificial developmental systems on gpus", *HPCS'07: Proceedings of the Conference on High Performance Computing and Simulation*, 2007.

[23] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk and W. Hwu "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73-82.

[24] S. Ventura, C. Romero, A. Zafra, J. A. Delgado and C. Hervás. "JCLEC: A Java framework for evolutionary computation", *Soft Computing*, 12 (4), 2007, pp. 381-392.

[25] T. Back, D. Fogel, Z. Michalewicz, "Handbook of Evolutionary Computation", *Oxford University Press*, Oxford, UK, 1997.

[26] T. Lensberg, A. Eilifsen, T. E. McKee, "Bankruptcy theory development and classification via Genetic Programming", *European Journal of Operational Research*, 169 (2), 2006, pp. 677-697.

[27] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, "From data mining to knowledge discovery in databases", *American Association for Artificial Intelligence*, 17, 1996, pp. 37-54.

[28] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, "Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and its Applications", *Morgan Kaufmann*, San Francisco, 1998.

[29] W. Frawley, G. Piatetsky-Shapiro, C. Matheus, "Knowledge Discovery in Databases: An Overview", *Knowledge Discovery in Databases*, 1991, pp. 1-30.

[30] W. Langdon, A. Harrison, 2008. "GP on SPMD parallel graphics hardware for mega bioinformatics data mining", *Soft Computing. A Fusion of Foundations, Methodologies and Applications 12*, 12 (12), 2008, pp. 1169-1183.

[31] W. Langdon, B. Buxton, "Genetic programming for combining classifiers", *Proceedings of the Genetic and Evolutionary Computation Conference*, 2001, pp. 66-73.

A. MANUAL DE USUARIO

Este apéndice constituye el manual de usuario del producto software desarrollado por el presente proyecto fin de carrera e incluye los requisitos mínimos y la información necesaria para su ejecución.

A.1. Requisitos del Sistema

A.1.1. Requisitos Hardware

La ejecución nativa requiere disponer de una tarjeta gráfica NVIDIA compatible con CUDA.

- Requisitos mínimos: tarjeta gráfica NVIDIA 8400, 9400, 210 con 512 MB GDDR.
- Requisitos recomendados: tarjeta gráfica NVIDIA GTX 260, 275, 285, 465, 470, 480 con 1 GB de GDDR.

A.1.2. Requisitos Software

Los requisitos software mínimos para la ejecución nativa son:

- Sistemas Operativos GNU/Linux, Windows XP, Vista, 7 o Mac OS X 10.6.3.
- Compilador de C/C++ GNU 4.3.4.
- JAVA SDK & RE 1.6.
- NVIDIA CUDA SDK y Toolkit 2.3.
- JCLEC 4.
- NVIDIA driver con soporte CUDA.

A.2. Instalación y desinstalación

Cree un nuevo proyecto Java en Eclipse e incluya los ficheros que se aportan en la documentación. Resultando una estructura de directorios y paquetes similar a la de la Figura A.1.

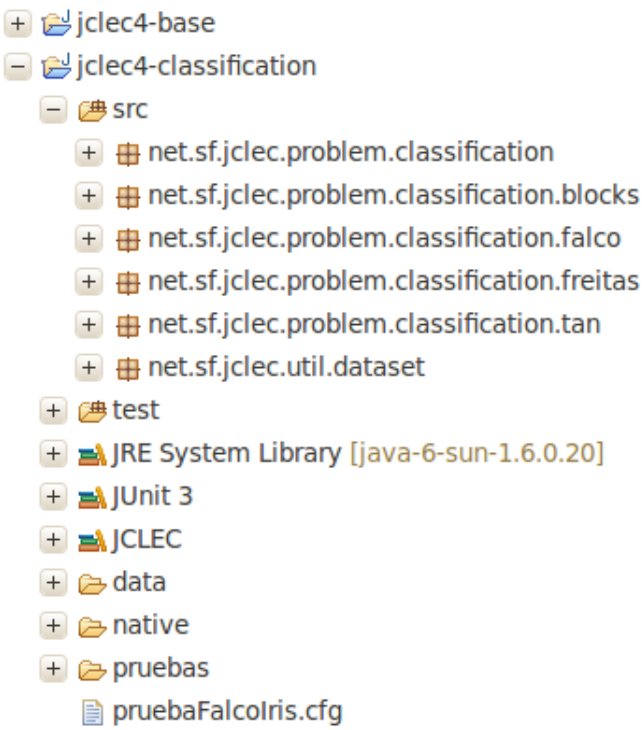

Figura A.1: Estructura de paquetes y directorios en Eclipse.

Para poder ejecutar código nativo, se necesita compilarlo empleando el *Makefile* que se encuentra en el directorio *native*. Compruebe el contenido del *Makefile* para verificar las rutas al directorio de instalación de Java *JAVA_DIR*. Ajuste la arquitectura del sitema operativo *ARCH = -m64* para sistemas de 64 bits o *ARCH = -m32* para sistemas de 32 bits.

En caso de compilación para GPU, revise la ruta del SDK de NVIDIA *NVIDIA_DIR* y la arquitectura de su GPU *GPU_ARCH*. Podrá encontrar la versión de su arquitectura en la página web de NVIDIA.

Ejecute el *Makefile* con el comando *make* para compilar los algoritmos para ejecución nativa en CPU y GPU o *make gpu=0* para compilarlos exclusivamente para CPU.

A.3. Uso de la Aplicación

El funcionamiento es similar al de cualquier algoritmo que se ejecute en JCLEC. Se emplea un fichero de configuración con etiquetas en las que se define el algoritmo a ejecutar, los operadores genéticos, el tamaño de población, el número de generaciones, etc. Para más información puede consultar el tutorial de JCLEC disponible en la página web del framework. A continuación se muestra un fichero de configuración de ejemplo:

```
<experiment>
  <process algorithm-type="net.sf.jclec.problem.classification.tan.TanAlgorithm">
    <rand-gen-factory type="net.sf.jclec.util.random.RanecuFactory" seed="123456789"/>
    <population-size>50</population-size>
    <max-of-generations>50</max-of-generations>
    <max-deriv-size>20</max-deriv-size>
    <dataset type="net.sf.jclec.util.dataset.KeelDataSet">
      <train-data>
        <file-name>data/Iris/Iris-10-1tra.dat</file-name>
      </train-data>
      <test-data>
        <file-name>data/Iris/Iris-10-1tst.dat</file-name>
      </test-data>
      <default-class>0</default-class>
      <attribute-class-name>class</attribute-class-name>
    </dataset>
    <species type="net.sf.jclec.problem.classification.tan.TanSyntaxTreeSpecies"/>
    <evaluator type="net.sf.jclec.problem.classification.tan.TanNativeEvaluator">
      <w1>0.7</w1>
      <w2>0.8</w2>
      <population-size>50</population-size>
      <native-dataset>data/Iris/Iris.dat</native-dataset>
      <use-gpu>true</use-gpu>
      <number-threads>1</number-threads>
    </evaluator>
    <provider type="net.sf.jclec.syntaxtree.SyntaxTreeCreator"/>
    <parents-selector type="net.sf.jclec.selector.RouletteSelector"/>
    <recombinator type="net.sf.jclec.syntaxtree.SyntaxTreeRecombinator" rec-prob="0.8">
      <base-op type="net.sf.jclec.problem.classification.tan.TanCrossover"/>
    </recombinator>
    <mutator type="net.sf.jclec.syntaxtree.SyntaxTreeMutator" mut-prob="0.1">
      <base-op type="net.sf.jclec.problem.classification.tan.TanMutator"/>
```

```
33        </mutator>
34        <copy-prob>0.01</copy-prob>
35        <elitist-prob>0.06</elitist-prob>
36        <support>0.03</support>
37        <listener type="net.sf.jclec.problem.classification.tan.TanNativePopulationReport">
38            <report-dir-name>pruebas/reportTan</report-dir-name>
39            <global-report-name>resumenTan</global-report-name>
40            <report-frequency>50</report-frequency>
41        </listener>
42    </process>
43 </experiment>
```

Para ejecutarlo se debe invocar a la clase *net.sf.jclec.RunExperiment* y el nombre del fichero de configuración como argumento. Si la ejecución es nativa, se debe incluir la referencia al directorio de la librería mediante la variable de entorno *LD_LIBRARY_PATH* apuntanto al directorio *native*. Además, si se emplea GPU, hay que añadir la referencia a las librerías de NVIDIA resultado la variable de entorno *LD_LIBRARY_PATH = native:/usr/local/cuda/lib64:/usr/local/cuda/lib*. La Figura A.2 muestra la configuración de la ventana de lanzamiento de Eclipse.

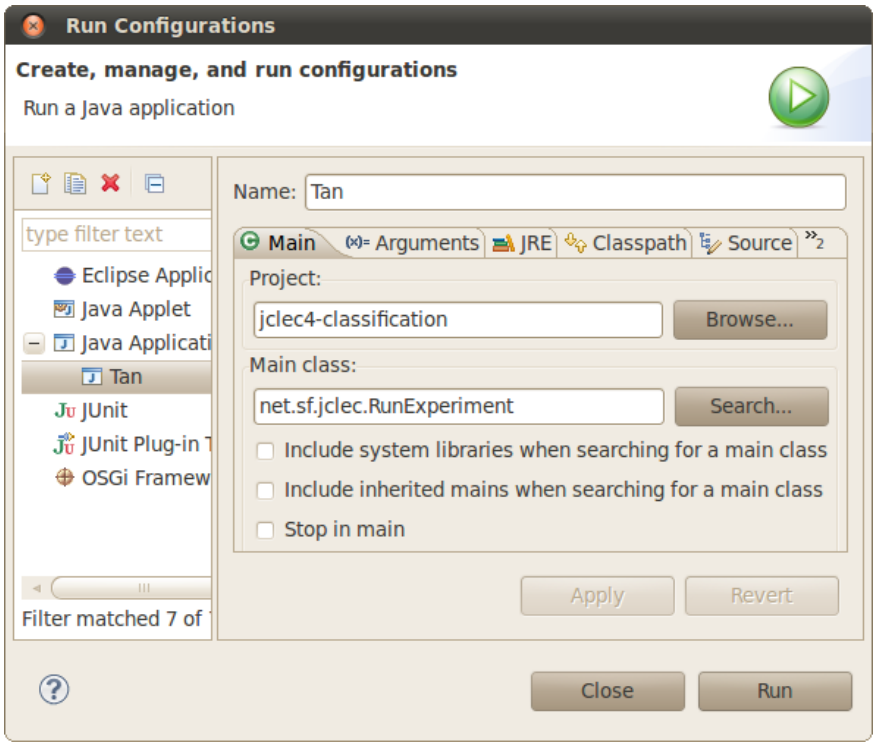

Figura A.2: Configuración de la ejecución en Eclipse.

B. MANUAL DE CÓDIGO

Este apéndice constituye el manual de código del proyecto fin de carrera. Se dispone de los siguientes ficheros:

- jclec-native.cu: definición de las variables y funciones comunes.
- falco.cu: fichero con las funciones del algoritmo de falco.
- tan.cu: fichero con las funciones del algoritmo de tan.
- freitas.cu: fichero con las funciones del algoritmo de freitas.
- functions.cu: fichero con las funciones para la interpretación de las expresiones.
- parameters.h: fichero con los parámetros de configuración de la ejecución.
- multithreading.cpp: fichero principal de la librería de hilos multiplataforma.
- multithreading.h: fichero de cabeceras de la librería de hilos multiplataforma.
- Makefile: ficheros de texto que utiliza make para llevar la gestión de la compilación de la librería.
- FalcoNativeEvaluator.java: evaluador de Falco de JCLEC con la llamada a la función de evaluación nativa.
- TanNativeEvaluator.java: evaluador de Tan de JCLEC con la llamada a la función de evaluación nativa.
- FreitasNativeEvaluator.java: evaluador de Freitas de JCLEC con la llamada a la función de evaluación nativa.

B.1. jclec-native.cu

```
#include "parameters.h"
#include "functions.h"

/**
 * Global vars required by algorithms
 */

SEMAPHORE wait_sem[MAX_THREADS],post_sem[MAX_THREADS];
CUTThread threadID[MAX_THREADS];
Plan plan[MAX_THREADS];

int numClasses, numberAttributes, numberInstances, numberInstances_A;
int classifiedClass, numThreads, populationSize;
jfloat *fitnessResult, *seResult, *spResult;
jint* bestClassResult;
bool evaluate = true;
float *h_instancesData;
int *h_instancesClass;
jobject indsList;

#ifdef __USE_GPU__
__constant__ char EXPR[256*MAX_EXPR_LEN];
#endif

/**
 * Free the dynamic memory space
 */
void nativeFree(JNIEnv *env, jobject obj)
{
   evaluate = false;

    // Wake up threads to finish them
   for(int i = 0; i < numThreads; i++)
   SEM_POST (&wait_sem[i]);

   free(h_instancesData);
   free(h_instancesClass);

   cutWaitForThreads(threadID, numThreads);

   #if _WIN32
   for(int i = 0; i < numThreads; i++)
   {
      CloseHandle(wait_sem[i]);
      CloseHandle(post_sem[i]);
   }
   #endif
}

/**
 * Gets the VM running at the host
```

```
 */
static void Get_VM(JavaVM** jvm_p, JNIEnv** env_p) {

   JavaVM jvmBuffer;
   JavaVM* vmBuf = &jvmBuffer;
   jsize jvmTotalNumberFound = 0;
   jint resCheckVM = JNI_GetCreatedJavaVMs(&vmBuf, 1, &jvmTotalNumberFound);

   if (jvmTotalNumberFound < 1)
   {
      fprintf(stderr, "No JVM found\n");
      exit(0);
   }
   *jvm_p = vmBuf;

   (*jvm_p)->AttachCurrentThread((void**)env_p, NULL);
}

#include "falco.cu"
#include "tan.cu"
#include "freitas.cu"
```

B.2. falco.cu

```
#ifdef __USE_GPU__

/**
 * Evaluation GPU kernel for Falco Algorithm
 *
 * param The result array, the instances array, the classes array, the number of attributes,
 *  the number of instances, the aligned number of instances and the actual class to be classified
*/
__global__ void kernelFalco(unsigned char* result, float* instancesData, int* instancesClass,
          int numberAttributes, int numberInstances, int numberInstances_A, int classifiedClass)
{
   int instance = blockDim.y * blockIdx.y + threadIdx.y;

   if(instance < numberInstances)
   {
      int resultMemPosition = blockIdx.x * numberInstances_A + instance;

      if(covers(instance, &EXPR[MAX_EXPR_LEN * blockIdx.x], instancesData, numberInstances_A))
      {
         if(classifiedClass == instancesClass[instance])
            result[resultMemPosition] = 0; // HIT
         else
            result[resultMemPosition] = 1; // FAIL
      }
      else
      {
         if(classifiedClass != instancesClass[instance])
```

```
            result[resultMemPosition] = 0; // HIT
         else
            result[resultMemPosition] = 1; // FAIL
      }
   }
}

/**
 * Reduction GPU Confusion Matrix kernel for Falco Algorithm
 *
 * param The result array, the fitness array, the number of instances and the aligned number of instances
*/
__global__ void MC_kernelFalco(unsigned char* result, jfloat* fitness, int numberInstances,
          int numberInstances_A)
{
   __shared__ int MC[128];

   MC[threadIdx.y] = 0;

   int base = blockIdx.x*numberInstances_A + threadIdx.y;
   int top = blockIdx.x*numberInstances_A + numberInstances - base;

    // Performs the reduction of the thread corresponding values
   for(int i = 0; i < top; i+=128)
   {
      MC[threadIdx.y] += result[base + i];
   }

   __syncthreads();

    // Calculates the final amount
   if(threadIdx.y == 0)
   {
      int fails = 0;
      for(int i = 0; i < 128; i++)
      {
         fails += MC[i];
      }
        // Set the fitness to the individual
      fitness[blockIdx.x] = 2*fails;
   }
}

/**
 * GPU device thread that performs the evaluation of a portion of the population
 *
 * param The job plan for the thread
*/
CUT_THREADPROC gpuThreadFalco(Plan *plan)
{
    // Set the GPU device number, each thread on a different GPU
   cudaSetDevice(plan->thread);
```

```
float* d_instancesData;
int *d_instancesClass;
int threadPopulationSize;
unsigned char* d_result;
jfloat* d_fitness;
JNIEnv* env;
JavaVM* jvm;

// GPU dynamic memory allocation
cudaMalloc((void**) &d_fitness, BLOCK_SIZE_FALCO*sizeof(jfloat));
cudaMalloc((void**) &d_instancesData, numberAttributes*numberInstances_A*sizeof(float));
cudaMalloc((void**) &d_instancesClass, numberInstances*sizeof(int));
cudaMalloc((void**) &d_result, BLOCK_SIZE_FALCO * numberInstances_A * sizeof(unsigned char));

// Copy instances data and classes to the GPU
cudaMemcpy(d_instancesData, h_instancesData, numberAttributes*numberInstances_A*sizeof(float),
        cudaMemcpyHostToDevice );
cudaMemcpy(d_instancesClass, h_instancesClass, numberInstances*sizeof(int),
        cudaMemcpyHostToDevice );

// Signal: thread is ready to evaluate
SEM_POST(&post_sem[plan->thread]);

Get_VM(&jvm, &env);

dim3 threads_evaluate(1, THREADS_EVAL_BLOCK);
dim3 threads_mc(1,128);

do
{
   // Wait until evaluation is required
   SEM_WAIT (&wait_sem[plan->thread]);

   if(evaluate)
   {
      // Calculate the thread population size
      threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

      // If population overflow, recalculate the thread actual population size
      if((plan->thread + 1) * threadPopulationSize > populationSize)
      {
         if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
            threadPopulationSize = 0;
      }
      if(threadPopulationSize > plan->size)
      {
         printf("Error. Population size overflow\n");
         exit(-1);
      }
      if(threadPopulationSize > 0)
      {
         // Calculate the base index of the individual for this thread
         int base = plan->thread * (int)ceil(populationSize/(float)numThreads);
```

```
    // Get the methods from Java
  jclass cls = env->GetObjectClass(indsList);
  jmethodID midR = env->GetMethodID(cls, "getReverseExpression", "(I)Ljava/lang/String;");
  jmethodID midW = env->GetMethodID(cls, "setFitness", "(IF)V");

  int blockIdxSize = BLOCK_SIZE_FALCO;                                    140

  // Population is evaluated using blocks of BLOCK_SIZE_FALCO individuals
  for(int j = 0; j < threadPopulationSize; j += BLOCK_SIZE_FALCO)
  {
     // Copy the rule from each individual in the block from the thread population to the GPU
    for(int i = 0; i < BLOCK_SIZE_FALCO && j+i < threadPopulationSize; i++)
    {
      cudaMemcpyToSymbol("EXPR",
      env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i),0),
      strlen(env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i), 150
      0))*sizeof(char),i*MAX_EXPR_LEN,cudaMemcpyHostToDevice);
    }

       // If the last block size is smaller, fix the block size to the number of the rest of individuals
    if(j+BLOCK_SIZE_FALCO > threadPopulationSize)
      blockIdxSize = threadPopulationSize - j;

    // Setup evaluation grid size
    dim3 grid_evaluate(blockIdxSize, (int)ceil(numberInstances/(float)THREADS_EVAL_BLOCK));
                                                                          160
    // Evaluation kernel call
    kernelFalco <<< grid_evaluate, threads_evaluate >>> (d_result, d_instancesData,
      d_instancesClass, numberAttributes, numberInstances, numberInstances_A, classifiedClass);

    // Setup reduction grid size
    dim3 grid_mc(blockIdxSize, 1);

    // Wait until evaluation kernel finishes
    cudaThreadSynchronize();
                                                                          170
    cutilCheckMsg("kernelFalco() execution failed.\n");

    // Reduction kernel call
    MC_kernelFalco <<< grid_mc, threads_mc >>> (d_result, d_fitness, numberInstances,
      numberInstances_A);

    // Copy the fitness values from the GPU to Host memory and set them to the individuals
    cudaMemcpy(fitnessResult + base + j, d_fitness, blockIdxSize*sizeof(jfloat),
      cudaMemcpyDeviceToHost );
                                                                          180
    for(int i = 0; i < blockIdxSize; i++)
    {
       env->CallVoidMethod(indsList, midW, base + j + i, fitnessResult[base+j+i]);
    }
  }
}
```

```
        }
        else
        {
            // Algorithm finished, Free dynamic memory                    190
            cudaFree(d_instancesData);
            cudaFree(d_instancesClass);
            cudaFree(d_result);
            cudaFree(d_fitness);
        }

        // Iteration finished
        SEM_POST(&post_sem[plan->thread]);

    }while(evaluate);                                                     200

    jvm->DetachCurrentThread();
    CUT_THREADEND;
}

#endif

/**
 * Function executed when nativeFree() call from Java
 */                                                                       210
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_falco_FalcoNativeEvaluator_nativeFree(JNIEnv *env, jobject obj)
{
    nativeFree(env,obj);
    delete [] fitnessResult;
}

/**
 * Evaluation Host function for Falco Algorithm
 *                                                                        220
 * param The result array, the rules expression, the instances array, the classes array, the number of attributes,
    the number of instances, the thread population size and the actual class to be classified
*/
void kernelFalcoCPU(unsigned char* result, char** expr, float* instancesData, int* instancesClass,
                    int numberAttributes, int numberInstances, int threadPopulationSize, int classifiedClass)
{
    for(int i = 0; i < threadPopulationSize; i++)
    for(int j = 0; j < numberInstances; j++)
    {
        if(coversCPU(j, expr[i], instancesData, numberInstances_A))       230
        {
            if(classifiedClass == instancesClass[j])
                result[i*numberInstances_A+j] = 0; // HIT
            else
                result[i*numberInstances_A+j] = 1; // FAIL
        }
        else
        {
            if(classifiedClass != instancesClass[j])
```

```
            result[i*numberInstances_A+j] = 0; // HIT
        else
            result[i*numberInstances_A+j] = 1; // FAIL
    }
  }
}

/**
 * Reduction Confusion Matrix Host function for Falco Algorithm
 *
 * param The result array, the fitness array, the number of instances and the thread population size
*/
void MC_kernelFalcoCPU(unsigned char* result, jfloat* fitness, int numberInstances, int threadPopulationSize)
{
   // For each individual
  for(int i = 0; i < threadPopulationSize; i++)
  {
    int fails = 0;
    // Calculate the number of fails
    for(int j = 0; j < numberInstances; j++)
    {
      fails += result[i*numberInstances_A + j];
    }
    // Set the fitness to the individual
    fitness[i] = 2*fails;
  }
}

/**
 * Host thread that performs the evaluation of a portion of the population
 *
 * param The job plan for the thread
*/
CUT_THREADPROC cpuThreadFalco(Plan *plan)
{
  char **h_array_list;
  unsigned char* h_result;
  int threadPopulationSize;
  JNIEnv* env;
  JavaVM* jvm;

  // Host dynamic memory allocation
  h_array_list = (char**)malloc(plan->size * sizeof(char*));

  for (int i = 0; i < plan->size; i++)
    h_array_list[i] = (char*)malloc(MAX_EXPR_LEN * sizeof(char));

  h_result = (unsigned char*)malloc(plan->size * numberInstances_A * sizeof(unsigned char));

  // Signal: thread is ready to evaluate
  SEM_POST(&post_sem[plan->thread]);

  Get_VM(&jvm, &env);
```

```
do
{
   // Wait until evaluation is required
   SEM_WAIT (&wait_sem[plan->thread]);

   if(evaluate)
   {                                                                    300
      // Calculate the thread population size
      threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

      // If population overflow, recalculate the thread actual population size
      if((plan->thread + 1) * threadPopulationSize > populationSize)
      {
         if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
            threadPopulationSize = 0;
      }
      if(threadPopulationSize > plan->size)                             310
      {
         printf("Error. Population size overflow\n");
         exit(-1);
      }
      if(threadPopulationSize > 0)
      {
         // Calculate the base index of the individual for this thread
         int base = plan->thread * (int)ceil(populationSize/(float)numThreads);

            // Get the methods from Java                                320
         jclass cls = env->GetObjectClass(indsList);
         jmethodID midR = env->GetMethodID(cls, "getReverseExpression", "(I)Ljava/lang/String;");
         jmethodID midW = env->GetMethodID(cls, "setFitness", "(IF)V");

         // Copy the rule from each individual in the block from Java to native memory
         for(int i = 0; i < threadPopulationSize; i++)
         {
            strcpy(h_array_list[i],
            env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+i),0));
         }                                                              330

         // Evaluation Host function call
         kernelFalcoCPU(h_result, h_array_list, h_instancesData, h_instancesClass, numberAttributes,
           numberInstances, threadPopulationSize, classifiedClass);

         // Reduction Host function call
         MC_kernelFalcoCPU(h_result, fitnessResult + base, numberInstances, threadPopulationSize);

         // Set the fitness to the individuals
         for(int i = 0; i < threadPopulationSize; i++)                  340
         {
            env->CallVoidMethod(indsList, midW, base + i, fitnessResult[base+i]);
         }
      }
   }
```

```
        else
        {
            // Algorithm finished, Free dynamic memory
            for(int i = 0; i < plan->size; i++)
            free(h_array_list[i]);                                          350
            free(h_array_list);
            free(h_result);
        }

        // Iteration finished
        SEM_POST(&post_sem[plan->thread]);

    }while(evaluate);

    jvm->DetachCurrentThread();                                             360
    CUT_THREADEND;
}

/**
 * Function executed when nativeMalloc() call from Java
 */
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_falco_FalcoNativeEvaluator_nativeMalloc(JNIEnv *env, jobject obj,
  jint popSize, jstring dataset, jboolean useGPU, jint jnumThreads)
{                                                                           370
    // Set the number of threads
    numThreads = jnumThreads;

    // Set up semaphores
    for(int i = 0; i < numThreads; i++)
    {
        SEM_INIT (&wait_sem[i], 0);
        SEM_INIT (&post_sem[i], 0);
    }
                                                                            380
    // Open native dataset
    FILE *f;
    if ((f = fopen(env->GetStringUTFChars(dataset,0),"r")) == NULL) exit(0);

    // Read configuration parameters
    fscanf(f,"%d%d%d",&numberAttributes,&numberInstances, &numClasses);

    // Calculate the aligned number of instances
    numberInstances_A = ceil(numberInstances/16.0)*16;
                                                                            390
    // Dynamic host memory allocation
    h_instancesData = (float*)malloc(numberAttributes*numberInstances_A*sizeof(float));
    h_instancesClass = (int*)malloc(numberInstances*sizeof(int));

    // Read dataset from file
    for(int i=0; i < numberInstances; i++)
    {
        for(int j=0; j < numberAttributes; j++)
```

```
      fscanf(f,"%f",&h_instancesData[j*numberInstances_A+i]);
      fscanf(f,"%d",&h_instancesClass[i]);
   }

    // Set up threads plans
   for(int i = 0; i < numThreads; i++)
   {
      plan[i].thread = i;
      plan[i].size = (int)ceil(popSize/(float)numThreads);
   }

   // If GPU usage launch GPU devices control threads
   if(useGPU)
   {
      #ifdef __USE_GPU__
      int deviceCount;
      cudaGetDeviceCount(&deviceCount);

      if(numThreads > deviceCount)
      {
         fprintf(stderr, "Can't use %d threads. CUDA devices (non-display) count is %d\n",
           numThreads, deviceCount);
         exit(0);
      }

      for(int i = 0; i < numThreads; i++)
          threadID[i] = cutStartThread((CUT_THREADROUTINE)gpuThreadFalco, (void *)&plan[i]);
      #endif

      #ifndef __USE_GPU__
      fprintf(stderr, "This code has not been compiled for GPU usage");
      exit(0);
      #endif
   }
   else
   {
       // Launch Host threads
      for(int i = 0; i < numThreads; i++)
          threadID[i] = cutStartThread((CUT_THREADROUTINE)cpuThreadFalco, (void *)&plan[i]);
   }

   fclose(f);

   fitnessResult = new jfloat[popSize];

   // SIGNAL: threads ready to evaluate
   for(int i = 0; i < numThreads; i++)
   SEM_WAIT (&post_sem[i]);
}

/**
 * Function executed when nativeEvaluate() call from Java
```

```
 *
 * param The number of individuals, the actual class to classify and the list of individuals
*/
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_falco_FalcoNativeEvaluator_nativeEvaluate(JNIEnv *env, jobject obj,
  jint size, jint classifiedClassarg, jobject listaArg)
{
   evaluate = true;
                                                                                                    460
   indsList = listaArg;
   populationSize = size;
   classifiedClass = classifiedClassarg;

   // SIGNAL: wake up threads to evaluate
   for(int i = 0; i < numThreads && i < size; i++)
   SEM_POST (&wait_sem[i]);

   // Wait until threads finish
   for(int i = 0; i < numThreads && i < size; i++)                                                  470
   SEM_WAIT (&post_sem[i]);
}
```

B.3. tan.cu

```
float w1, w2;

#ifdef __USE_GPU__

/**
 * Evaluation GPU kernel for Falco Algorithm
 *
 * param The result array, the instances array, the classes array, the number of attributes,
    the number of instances, the aligned number of instances and the actual class to be classified
*/                                                                                                  10
__global__ void kernelTan(unsigned char* result, float* instancesData, int* instancesClass,
          int numberAttributes, int numberInstances, int numberInstances_A, int classifiedClass)
{
   int instance = blockDim.y * blockIdx.y + threadIdx.y;

   if(instance < numberInstances)
   {
      int resultMemPosition = blockIdx.x * numberInstances_A + instance;

      if(covers(instance, &EXPR[MAX_EXPR_LEN * blockIdx.x], instancesData, numberInstances_A))  20
      {
         if(classifiedClass == instancesClass[instance])
            result[resultMemPosition] = 0; // TRUE POSITIVE
         else
            result[resultMemPosition] = 2; // FALSE POSITIVE
      }
      else
```

```
        {
            if(classifiedClass != instancesClass[instance])
                result[resultMemPosition] = 1; // TRUE NEGATIVE
            else
                result[resultMemPosition] = 3; // FALSE NEGATIVE
        }
    }
}

/**
 * Reduction GPU Confusion Matrix kernel for Tan Algorithm
 *
 * param The result array, the fitness array, the number of instances,
    the aligned number of instances and the algorithm float values w1 and w2
*/
__global__ void MC_kernelTan(unsigned char* result, jfloat* fitness, int numberInstances,
          int numberInstances_A, float w1, float w2)
{
    __shared__ int MC[512];

    MC[threadIdx.y] = 0;
    MC[threadIdx.y+128] = 0;
    MC[threadIdx.y+256] = 0;
    MC[threadIdx.y+384] = 0;

    int base = blockIdx.x*numberInstances_A + threadIdx.y;
    int top = blockIdx.x*numberInstances_A + numberInstances - base;

    // Performs the reduction of the thread corresponding values
    for(int i = 0; i < top; i+=128)
    {
        MC[threadIdx.y*4 + result[base + i]]++;
    }

    __syncthreads();

     // Calculates the final amount
    if(threadIdx.y < 4)
    {
        for(int i = 4; i < 512; i+=4)
        {
            MC[0] += MC[i]; // Number of true positives
            MC[1] += MC[i+1]; // Number of true negatives
            MC[2] += MC[i+2]; // Number of false positives
            MC[3] += MC[i+3]; // Number of false negatives
        }
    }

    if(threadIdx.y == 0)
    {
        int tp = MC[0], tn = MC[1], fp = MC[2], fn = MC[3];
        // Set the fitness to the individual
        fitness[blockIdx.x] = (tp/(tp+w1*fn)) * (tn/(tn+w2*fp));
```

```
    }
}

/**
 * GPU device thread that performs the evaluation of a portion of the population
 */
CUT_THREADPROC gpuThreadTan(Plan *plan)
{
    // Set the GPU device number, each thread on a different GPU
    cudaSetDevice(plan->thread);

    float* d_instancesData;
    int *d_instancesClass;
    int threadPopulationSize;
    unsigned char* d_result;
    jfloat* d_fitness;
    JNIEnv* env;
    JavaVM* jvm;

    // GPU dynamic memory allocation
    cudaMalloc((void**) &d_fitness, BLOCK_SIZE_TAN*sizeof(jfloat));
    cudaMalloc((void**) &d_instancesData, numberAttributes*numberInstances_A*sizeof(float));
    cudaMalloc((void**) &d_instancesClass, numberInstances*sizeof(int));
    cudaMalloc((void**) &d_result, BLOCK_SIZE_TAN * numberInstances_A * sizeof(unsigned char));

    // Copy instances data and classes to the GPU
    cudaMemcpy(d_instancesData, h_instancesData, numberAttributes*numberInstances_A*sizeof(float),
      cudaMemcpyHostToDevice );
    cudaMemcpy(d_instancesClass, h_instancesClass, numberInstances*sizeof(int),
      cudaMemcpyHostToDevice );

    // Signal: thread is ready to evaluate
    SEM_POST(&post_sem[plan->thread]);

    Get_VM(&jvm, &env);

    dim3 threads_evaluate(1, THREADS_EVAL_BLOCK);
    dim3 threads_mc(1,128);

    do
    {
        // Wait until evaluation is required
      SEM_WAIT (&wait_sem[plan->thread]);

      if(evaluate)
      {
        // Calculate the thread population size
        threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

        // If population overflow, recalculate the thread actual population size
        if((plan->thread + 1) * threadPopulationSize > populationSize)
        {
          if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
```

```
        threadPopulationSize = 0;
    }
    if(threadPopulationSize > plan->size)
    {
       printf("Error. Population size overflow\n");
       exit(-1);
    }                                                                    140
    if(threadPopulationSize > 0)
    {
        // Calculate the base index of the individual for this thread
       int base = plan->thread * (int)ceil(populationSize/(float)numThreads);

       // Get the methods from Java
       jclass cls = env->GetObjectClass(indsList);
       jmethodID midR = env->GetMethodID(cls, "getReverseExpresion", "(I)Ljava/lang/String;");
       jmethodID midW = env->GetMethodID(cls, "setFitness", "(IF)V");
                                                                         150
       int blockIdxSize = BLOCK_SIZE_TAN;

       // Population is evaluated using blocks of BLOCK_SIZE_TAN individuals
       for(int j = 0; j < threadPopulationSize; j += BLOCK_SIZE_TAN)
       {
          // Copy the rule from each individual in the block from the thread population to the GPU
          for(int i = 0; i < BLOCK_SIZE_TAN && j+i < threadPopulationSize; i++)
          {
             cudaMemcpyToSymbol("EXPR",
             env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i),0),  160
             strlen(env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i),
             0))*sizeof(char),i*MAX_EXPR_LEN,cudaMemcpyHostToDevice);
          }

          // If the last block size is smaller, fix the block size to the number of the rest of individuals
          if(j+BLOCK_SIZE_TAN > threadPopulationSize)
             blockIdxSize = threadPopulationSize - j;

          // Setup evaluation grid size
          dim3 grid_evaluate(blockIdxSize, (int)ceil(numberInstances/(float)THREADS_EVAL_BLOCK));  170

          // Evaluation kernel call
          kernelTan <<< grid_evaluate, threads_evaluate >>> (d_result, d_instancesData, d_instancesClass,
            numberAttributes, numberInstances, numberInstances_A, classifiedClass);

          // Setup reduction grid size
          dim3 grid_mc(blockIdxSize, 1);

          // Wait until evaluation kernel finishes
          cudaThreadSynchronize();                                       180

          cutilCheckMsg("kernelTan() execution failed.\n");

          // Reduction kernel call
          MC_kernelTan <<< grid_mc, threads_mc >>> (d_result, d_fitness, numberInstances,
            numberInstances_A, w1, w2);
```

```
                // Copy the fitness values from the GPU to Host memory and set them to the individuals
                cudaMemcpy(fitnessResult + base + j, d_fitness, blockIdxSize*sizeof(jfloat),
                  cudaMemcpyDeviceToHost );

                for(int i = 0; i < blockIdxSize; i++)
                {
                    env->CallVoidMethod(indsList, midW, base + j + i, fitnessResult[base+j+i]);
                }
            }
        }
    }
    else
    {
        // Algorithm finished, Free dynamic memory
        cudaFree(d_instancesData);
        cudaFree(d_instancesClass);
        cudaFree(d_result);
        cudaFree(d_fitness);
    }

    // Iteration finished
    SEM_POST(&post_sem[plan->thread]);

}while(evaluate);

jvm->DetachCurrentThread();
CUT_THREADEND;
}

#endif

/**
 * Function executed when nativeFree() call from Java
 */
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_tan_TanNativeEvaluator_nativeFree(JNIEnv *env, jobject obj)
{
    nativeFree(env,obj);
    delete [] fitnessResult;
}

/**
 * Evaluation Host function for Tan Algorithm
 *
 * param The result array, the rules expression, the instances array, the classes array, the number of attributes,
    the number of instances, the thread population size and the actual class to be classified
*/
void kernelTanCPU(unsigned char* result, char** expr, float* instancesData, int* instancesClass,
  int numberAttributes, int numberInstances, int threadPopulationSize, int classifiedClass)
{
    for(int i = 0; i < threadPopulationSize; i++)
    for(int j = 0; j < numberInstances; j++)
```

```
    {
        if(coversCPU(j, expr[i], instancesData, numberInstances_A))
        {
            if(classifiedClass == instancesClass[j])
                result[i*numberInstances_A+j] = 0; // TRUE POSITIVE
            else
                result[i*numberInstances_A+j] = 2; // FALSE POSITIVE
        }
        else
        {
            if(classifiedClass != instancesClass[j])
                result[i*numberInstances_A+j] = 1; // TRUE NEGATIVE
            else
                result[i*numberInstances_A+j] = 3; // FALSE NEGATIVE
        }
    }
}

/**
 * Reduction Confusion Matrix Host function for Tan Algorithm
 *
 * param The result array, the fitness array, the number of instances, the thread population size and
    the algorithm float values w1 and w2
*/
void MC_kernelTanCPU(unsigned char* result, jfloat* fitness, int numberInstances, int threadPopulationSize,
  float w1, float w2)
{
    // For each individual
    for(int i = 0; i < threadPopulationSize; i++)
    {
        int MC[4] = {0,0,0,0};
        // Performs the reduction of the thread corresponding values
        for(int j = 0; j < numberInstances; j++)
        {
            MC[result[i*numberInstances_A + j]]++;
        }
        // Set the fitness to the individual
        fitness[i] = (MC[0]/(MC[0]+w1*MC[3])) * (MC[1]/(MC[1]+w2*MC[2]));
    }
}

/**
 * Host thread that performs the evaluation of a portion of the population
 *
 * param The job plan for the thread
*/
CUT_THREADPROC cpuThreadTan(Plan *plan)
{
    char **h_array_list;
    unsigned char* h_result;
    int threadPopulationSize;
    JNIEnv* env;
    JavaVM* jvm;
```

```
// Host dynamic memory allocation
h_array_list = (char**)malloc(plan->size * sizeof(char*));

for (int i = 0; i < plan->size; i++)
   h_array_list[i] = (char*)malloc(MAX_EXPR_LEN * sizeof(char));

h_result = (unsigned char*)malloc(plan->size * numberInstances_A * sizeof(unsigned char));

// Signal: thread is ready to evaluate
SEM_POST(&post_sem[plan->thread]);

Get_VM(&jvm, &env);

do
{
    // Wait until evaluation is required
   SEM_WAIT (&wait_sem[plan->thread]);

   if(evaluate)
   {
      // Calculate the thread population size
      threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

      // If population overflow, recalculate the thread actual population size
      if((plan->thread + 1) * threadPopulationSize > populationSize)
      {
         if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
            threadPopulationSize = 0;
      }
      if(threadPopulationSize > plan->size)
      {
         printf("Error. Population size overflow\n");
         exit(-1);
      }
      if(threadPopulationSize > 0)
      {
         // Calculate the base index of the individual for this thread
         int base = plan->thread * (int)ceil(populationSize/(float)numThreads);

         // Get the methods from Java
         jclass cls = env->GetObjectClass(indsList);
         jmethodID midR = env->GetMethodID(cls, "getReverseExpresion", "(I)Ljava/lang/String;");
         jmethodID midW = env->GetMethodID(cls, "setFitness", "(IF)V");

         // Copy the rule from each individual in the block from Java to native memory
         for(int i = 0; i < threadPopulationSize; i++)
         {
            strcpy(h_array_list[i],
            env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+i),0));
         }

         // Evaluation Host function call
```

```
            kernelTanCPU(h_result, h_array_list, h_instancesData, h_instancesClass, numberAttributes,
              numberInstances, threadPopulationSize, classifiedClass);

            // Reduction Host function call
            MC_kernelTanCPU(h_result, fitnessResult + base, numberInstances, threadPopulationSize, w1, w2);

            // Set the fitness to the individuals
            for(int i = 0; i < threadPopulationSize; i++)
            {
                env->CallVoidMethod(indsList, midW, base + i, fitnessResult[base+i]);
            }
        }
    }
    else
    {
        // Algorithm finished, Free dynamic memory
        for(int i = 0; i < plan->size; i++)
        free(h_array_list[i]);
        free(h_array_list);
        free(h_result);
    }

    // Iteration finished
    SEM_POST(&post_sem[plan->thread]);

}while(evaluate);

jvm->DetachCurrentThread();
CUT_THREADEND;
}

/**
 * Function executed when nativeMalloc() call from Java
 */
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_tan_TanNativeEvaluator_nativeMalloc(JNIEnv *env, jobject obj,
  jint popSize, jstring dataset, jboolean useGPU, jint jnumThreads)
{
    // Set the number of threads
  numThreads = jnumThreads;

  // Set up semaphores
  for(int i = 0; i < numThreads; i++)
  {
    SEM_INIT (&wait_sem[i], 0);
    SEM_INIT (&post_sem[i], 0);
  }

    // Open native dataset
  FILE *f;
  if ((f = fopen(env->GetStringUTFChars(dataset,0),"r")) == NULL) exit(0);

    // Read configuration parameters
```

```
fscanf(f,"%d%d%d",&numberAttributes,&numberInstances, &numClasses);
                                                                    400
// Calculate the aligned number of instances
numberInstances_A = ceil(numberInstances/16.0)*16;

// Dynamic host memory allocation
h_instancesData = (float*)malloc(numberAttributes*numberInstances_A*sizeof(float));
h_instancesClass = (int*)malloc(numberInstances*sizeof(int));

 // Read dataset from file
for(int i=0; i < numberInstances; i++)
{                                                                   410
   for(int j=0; j < numberAttributes; j++)
   fscanf(f,"%f",&h_instancesData[j*numberInstances_A+i]);

   fscanf(f,"%d",&h_instancesClass[i]);
}

 // Set up threads plans
for(int i = 0; i < numThreads; i++)
{
   plan[i].thread = i;                                              420
   plan[i].size = (int)ceil(popSize/(float)numThreads);
}

// If GPU usage launch GPU devices control threads
if(useGPU)
{
   #ifdef __USE_GPU__
   int deviceCount;
   cudaGetDeviceCount(&deviceCount);
                                                                    430
   if(numThreads > deviceCount)
   {
      fprintf(stderr, "Can't use%d threads. CUDA devices (non-display) count is%d\n",
        numThreads, deviceCount);
      exit(0);
   }

   for(int i = 0; i < numThreads; i++)
       threadID[i] = cutStartThread((CUT_THREADROUTINE)gpuThreadTan, (void *)&plan[i]);
   #endif                                                           440

   #ifndef __USE_GPU__
   fprintf(stderr, "This code has not been compiled for GPU usage");
   exit(0);
   #endif
}
else
{
    // Launch Host threads
   for(int i = 0; i < numThreads; i++)                              450
       threadID[i] = cutStartThread((CUT_THREADROUTINE)cpuThreadTan, (void *)&plan[i]);
```

```
    }

    fclose(f);

    fitnessResult = new jfloat[popSize];

    // SIGNAL: threads ready to evaluate
    for(int i = 0; i < numThreads; i++)
    SEM_WAIT (&post_sem[i]);                                                    460
}

/**
 * Function executed when nativeEvaluate() call from Java
 *
 * param The number of individuals, the actual class to classify and the list of individuals
*/
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_tan_TanNativeEvaluator_nativeEvaluate(JNIEnv *env, jobject obj,
  jint size, jint classifiedClassarg,     jfloat w1arg, jfloat w2arg, jobject listaArg)          470
{
    evaluate = true;

    indsList = listaArg;
    populationSize = size;
    classifiedClass = classifiedClassarg;
    w1 = w1arg;
    w2 = w2arg;

    // SIGNAL: wake up threads to evaluate                                      480
    for(int i = 0; i < numThreads && i < size; i++)
    SEM_POST (&wait_sem[i]);

    // Wait until threads finish
    for(int i = 0; i < numThreads && i < size; i++)
    SEM_WAIT (&post_sem[i]);
}
```

B.4. freitas.cu

```
#ifdef __USE_GPU__

/**
 * Evaluation GPU kernel for Falco Algorithm
 *
 * param The result array, the instances array, the classes array, the number of attributes,
    the number of instances, the aligned number of instances and the number of classes of the dataset
*/
__global__ void kernelFreitas(unsigned char** result, float* instancesData, int* instancesClass,
          int numberAttributes, int numberInstances, int numberInstances_A, int numClasses)
{
   int instance = blockDim.y * blockIdx.y + threadIdx.y;

   if(instance < numberInstances)
   {
      int resultMemPosition = blockIdx.x * numberInstances_A + instance;

      if(covers(instance, &EXPR[MAX_EXPR_LEN * blockIdx.x], instancesData, numberInstances_A))
      {
         for(int i = 0; i < numClasses; i++)
               result[i][resultMemPosition] = 2; // FALSE POSITIVE
         result[instancesClass[instance]][resultMemPosition] = 0; // TRUE POSITIVE
      }
      else
      {
         for(int i = 0; i < numClasses; i++)
               result[i][resultMemPosition] = 1; // TRUE NEGATIVE
         result[instancesClass[instance]][resultMemPosition] = 3; // FALSE NEGATIVE
      }
   }
}

/**
 * Reduction GPU Confusion Matrix kernel for Freitas Algorithm
 *
 * param The result array, the number of instances, the se array, the sp array, the best class array,
    the aligned number of instances and the number of classes
*/
__global__ void MC_kernelFreitas(unsigned char** result, int numberInstances, jfloat* se, jfloat* sp,
          jint* bestClass, int numberInstances_A, int numClasses)
{
   __shared__ int MC[512];

   float seAux, spAux;

   int base = blockIdx.x*numberInstances_A + threadIdx.y;
   int top = blockIdx.x*numberInstances_A + numberInstances - base;

   if(threadIdx.y == 0)
   {
      se[blockIdx.x] = -1;
```

```
    sp[blockIdx.x] = 1;
}

// Checks the best class for the individual
for(int j = 0; j < numClasses; j++)
{
    MC[threadIdx.y] = 0;
    MC[threadIdx.y+128] = 0;
    MC[threadIdx.y+256] = 0;
    MC[threadIdx.y+384] = 0;

    // Performs the reduction of the thread corresponding values
    for(int i = 0; i < top; i+=128)
    {
        MC[threadIdx.y*4 + result[j][base + i]]++;
    }

    __syncthreads();

    if(threadIdx.y < 4)
    {
        for(int i = 4; i < 512; i+=4)
        {
            MC[0] += MC[i]; // Number of true positives
            MC[1] += MC[i+1]; // Number of true negatives
            MC[2] += MC[i+2]; // Number of false positives
            MC[3] += MC[i+3]; // Number of false negatives
        }
    }

    if(threadIdx.y == 0)
    {
        int tp = MC[0], tn = MC[1], fp = MC[2], fn = MC[3];

        if(tp + fn == 0)
            seAux = 1;
        else
            seAux = (float) tp / (tp + fn);

        if(tn + fp == 0)
            spAux = 1;
        else
            spAux = (float) tn / (tn + fp);

        if(seAux*spAux == se[blockIdx.x]*sp[blockIdx.x])
        {
            bestClass[blockIdx.x] = j;
        }
        // If the actual class is best for the individual we keep it
        if(seAux*spAux > se[blockIdx.x]*sp[blockIdx.x])
        {
            se[blockIdx.x] = seAux;
            sp[blockIdx.x] = spAux;
```

```
            bestClass[blockIdx.x] = j;
          }
        }

        __syncthreads();
    }
}

/**
 * GPU device thread that performs the evaluation of a portion of the population
 */
CUT_THREADPROC gpuThreadFreitas(Plan *plan)
{
    // Set the GPU device number, each thread on a different GPU
    cudaSetDevice(plan->thread);

    float* d_instancesData;
    int *d_instancesClass;
    int threadPopulationSize;
    jfloat *se, *sp, *d_se, *d_sp;
    jint *bestClass, *d_bestClass;
    unsigned char** d_result, **h_result;
    JNIEnv* env;
    JavaVM* jvm;

    // Host dynamic memory allocation
    se = new jfloat[BLOCK_SIZE_FREITAS];
    sp = new jfloat[BLOCK_SIZE_FREITAS];
    bestClass = new jint[BLOCK_SIZE_FREITAS];

    h_result = (unsigned char**)malloc(numClasses * sizeof(unsigned char*));

    // GPU dynamic memory allocation
    cudaMalloc((void**) &d_se, BLOCK_SIZE_FREITAS*sizeof(jfloat));
    cudaMalloc((void**) &d_sp, BLOCK_SIZE_FREITAS*sizeof(jfloat));
    cudaMalloc((void**) &d_bestClass, BLOCK_SIZE_FREITAS*sizeof(jint));

    cudaMalloc((void**) &d_instancesData, numberAttributes*numberInstances_A*sizeof(float));
    cudaMalloc((void**) &d_instancesClass, numberInstances*sizeof(int));
    cudaMalloc((void**) &d_result, numClasses * sizeof(unsigned char*));
    for(int i = 0; i < numClasses; i++)
        cudaMalloc((void**) &h_result[i], BLOCK_SIZE_FREITAS * numberInstances_A * sizeof(unsigned char)

    // Copy instances data and classes to the GPU
    cudaMemcpy(d_result, h_result, numClasses * sizeof(float*), cudaMemcpyHostToDevice);
    cudaMemcpy(d_instancesData, h_instancesData, numberAttributes*numberInstances_A*sizeof(float),
      cudaMemcpyHostToDevice );
    cudaMemcpy(d_instancesClass, h_instancesClass, numberInstances*sizeof(int),
      cudaMemcpyHostToDevice );

    // Signal: thread is ready to evaluate
    SEM_POST(&post_sem[plan->thread]);
```

```
Get_VM(&jvm, &env);

dim3 threads_evaluate(1, THREADS_EVAL_BLOCK);
dim3 threads_mc(1,128);

do
{
   // Wait until evaluation is required
   SEM_WAIT (&wait_sem[plan->thread]);

   if(evaluate)
   {
      // Calculate the thread population size
      threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

      // If population overflow, recalculate the thread actual population size
      if((plan->thread + 1) * threadPopulationSize > populationSize)
      {
         if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
            threadPopulationSize = 0;
      }
      if(threadPopulationSize > plan->size)
      {
         printf("Error. Population size overflow\n");
         exit(-1);
      }
      if(threadPopulationSize > 0)
      {
          // Calculate the base index of the individual for this thread
         int base = plan->thread * (int)ceil(populationSize/(float)numThreads);

         // Get the methods from Java
         jclass cls = env->GetObjectClass(indsList);
         jmethodID midR = env->GetMethodID(cls, "getReverseExpresion", "(I)Ljava/lang/String;");
         jmethodID midW = env->GetMethodID(cls, "setFitness", "(IFFI)V");

         int blockIdxSize = BLOCK_SIZE_FREITAS;

         // Population is evaluated using blocks of BLOCK_SIZE_FREITAS individuals
         for(int j = 0; j < threadPopulationSize; j += BLOCK_SIZE_FREITAS)
         {
            // Copy the rule from each individual in the block from the thread population to the GPU
            for(int i = 0; i < BLOCK_SIZE_FREITAS && j+i < threadPopulationSize; i++)
            {
               cudaMemcpyToSymbol("EXPR",
               env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i),0),
               strlen(env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+j+i),
               0))*sizeof(char),i*MAX_EXPR_LEN,cudaMemcpyHostToDevice);
            }

            // If the last block size is smaller, fix the block size to the number of the rest of individuals
            if(j+BLOCK_SIZE_FREITAS > threadPopulationSize)
               blockIdxSize = threadPopulationSize - j;
```

```
            // Setup evaluation grid size
            dim3 grid_evaluate(blockIdxSize, (int)ceil(numberInstances/(float)THREADS_EVAL_BLOCK));

            // Evaluation kernel call
            kernelFreitas <<< grid_evaluate, threads_evaluate >>> (d_result, d_instancesData,
              d_instancesClass, numberAttributes, numberInstances, numberInstances_A,numClasses);

            // Setup reduction grid size
            dim3 grid_mc(blockIdxSize, 1);                                                    220

            // Wait until evaluation kernel finishes
            cudaThreadSynchronize();

            cutilCheckMsg("kernelFreitas() execution failed.\n");

            // Reduction kernel call
            MC_kernelFreitas <<< grid_mc, threads_mc >>> (d_result, numberInstances, d_se, d_sp,
              d_bestClass, numberInstances_A,numClasses);
                                                                                              230
            // Copy the se, sp and bastClass values for each individual and pass them to Java method
            cudaMemcpy(se, d_se, blockIdxSize*sizeof(jfloat), cudaMemcpyDeviceToHost );
            cudaMemcpy(sp, d_sp, blockIdxSize*sizeof(jfloat), cudaMemcpyDeviceToHost );
            cudaMemcpy(bestClass, d_bestClass, blockIdxSize*sizeof(jint), cudaMemcpyDeviceToHost );

            for(int i = 0; i < blockIdxSize; i++)
            {
               env->CallVoidMethod(indsList, midW, base + j + i, se[i],sp[i],bestClass[i]);
            }
         }                                                                                    240
      }
   }
   else
   {
       // Algorithm finished, Free dynamic memory
      cudaFree(d_instancesData);
      cudaFree(d_instancesClass);
      cudaFree(d_se);
      cudaFree(d_sp);
      cudaFree(d_bestClass);                                                                  250
      for(int i = 0; i < numClasses; i++)
      cudaFree(h_result[i]);
      cudaFree(d_result);

      free(h_result);

      delete [] se;
      delete [] sp;
      delete [] bestClass;
   }                                                                                          260

   // Iteration finished
   SEM_POST(&post_sem[plan->thread]);
```

```
    }while(evaluate);

    jvm->DetachCurrentThread();
    CUT_THREADEND;
}
                                                                                    270
#endif

/**
 * Function executed when nativeFree() call from Java
 */
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_freitas_FreitasNativeEvaluator_nativeFree(JNIEnv *env, jobject obj)
{
    nativeFree(env,obj);
}                                                                                   280

/**
 * Evaluation Host function for Freitas Algorithm
 *
 * param The result array, the rules expression, the instances array, the classes array, the number of attributes,
     the number of instances, the thread population size and the number of classes
*/
void kernelFreitasCPU(unsigned char** result, char** expr, float* instancesData, int* instancesClass,
  int numberAttributes, int numberInstances, int threadPopulationSize, int numClasses)
{                                                                                   290
    for(int i = 0; i < threadPopulationSize; i++)
    for(int j = 0; j < numberInstances; j++)
    {
        if(coversCPU(j, expr[i], instancesData, numberInstances_A))
        {
            for(int k = 0; k < numClasses; k++)
                result[k][i*numberInstances_A+j] = 2; // FALSE POSITIVE

                result[instancesClass[j]][i*numberInstances_A+j] = 0; // TRUE POSITIVE
        }                                                                           300
        else
        {
            for(int k = 0; k < numClasses; k++)
                result[k][i*numberInstances_A+j] = 1; // TRUE NEGATIVE

                result[instancesClass[j]][i*numberInstances_A+j] = 3; // FALSE NEGATIVE
        }
    }
}
                                                                                    310
/**
 * Reduction Confusion Matrix Host function for Freitas Algorithm
 *
 * param The result array, the number of instances, the se array, the sp array, the best class array,
     the thread population size and the number of classes
*/
```

```
void MC_kernelFreitasCPU(unsigned char** result, int numberInstances, jfloat* se, jfloat* sp, jint* bestClass,
  int threadPopulationSize, int numClasses)
{
   int*** MC;                                                                    320
   MC = (int***)malloc(numClasses * sizeof(int**));

   // Alloc dynamic memory
   for(int j = 0; j < numClasses; j++)
   {
      MC[j] = (int**)malloc(threadPopulationSize * sizeof(int*));
      for(int i = 0; i < threadPopulationSize; i++)
      {
         MC[j][i] = (int*)calloc(4,sizeof(int));
      }                                                                          330
   }

    // For each individual
   for(int k = 0; k < threadPopulationSize; k++)
   {
       // For each class
      for(int j = 0; j < numClasses; j++)
      {
          // Performs the reduction of the thread corresponding values
         for(int i = 0; i < numberInstances ; i++)                              340
         {
            MC[j][k][result[j][numberInstances_A*k + i]]++;
         }
      }
   }

   // For each individual
   for(int k = 0; k < threadPopulationSize; k++)
   {
      float seAux, spAux;                                                       350
      se[k] = -1;
      sp[k] = 1;

      // Checks the best class for the individual
      for(int i = 0; i < numClasses; i++)
      {
         if(MC[i][k][0] + MC[i][k][3] == 0)
            seAux = 1;
         else
            seAux = (float) MC[i][k][0]/(MC[i][k][0]+MC[i][k][3]);              360

         if(MC[i][k][1]+MC[i][k][2] == 0)
            spAux = 1;
         else
            spAux = (float) MC[i][k][1]/(MC[i][k][1]+MC[i][k][2]);

         if(seAux*spAux == se[k]*sp[k])
         {
            bestClass[k] = i;
```

```
            }
            // If the actual class is best for the individual we keep it
            if(seAux*spAux > se[k]*sp[k])
            {
               se[k] = seAux;
               sp[k] = spAux;
               bestClass[k] = i;
            }
         }
      }

      // Free dynamic memory
      for(int j = 0; j < numClasses; j++)
      {
         for(int i = 0; i < threadPopulationSize; i++)
            free(MC[j][i]);
         free(MC[j]);
      }
      free(MC);
}

/**
 * Host thread that performs the evaluation of a portion of the population
 *
 * param The job plan for the thread
*/
CUT_THREADPROC cpuThreadFreitas(Plan *plan)
{
   char **h_array_list;
   jfloat *se, *sp;
   jint* bestClass;
   unsigned char** h_result;
   int threadPopulationSize;
   JNIEnv* env;
   JavaVM* jvm;

   // Host dynamic memory allocation
   se = new jfloat[plan->size];
   sp = new jfloat[plan->size];
   bestClass = new jint[plan->size];

   h_array_list = (char**)malloc(plan->size * sizeof(char*));
   for (int i = 0; i < plan->size; i++)
      h_array_list[i] = (char*)malloc(MAX_EXPR_LEN * sizeof(char));

   h_result = (unsigned char**)malloc(numClasses * sizeof(unsigned char*));
   for(int i = 0; i < numClasses; i++)
      h_result[i] = (unsigned char*)malloc(plan->size * numberInstances_A * sizeof(unsigned char));

   // Signal: thread is ready to evaluate
   SEM_POST(&post_sem[plan->thread]);

   Get_VM(&jvm, &env);
```

```
do
{
    // Wait until evaluation is required
   SEM_WAIT (&wait_sem[plan->thread]);

   if(evaluate)
   {                                                                    430
      // Calculate the thread population size
      threadPopulationSize = (int)ceil(populationSize/(float)numThreads);

      // If population overflow, recalculate the thread actual population size
      if((plan->thread + 1) * threadPopulationSize > populationSize)
      {
         if((threadPopulationSize = populationSize - threadPopulationSize * plan->thread) < 0)
            threadPopulationSize = 0;
      }
      if(threadPopulationSize > plan->size)                             440
      {
         printf("Error. Population size overflow\n");
         exit(-1);
      }
      if(threadPopulationSize > 0)
      {
         // Calculate the base index of the individual for this thread
         int base = plan->thread * (int)ceil(populationSize/(float)numThreads);

         // Get the methods from Java                                   450
         jclass cls = env->GetObjectClass(indsList);
         jmethodID midR = env->GetMethodID(cls, "getReverseExpresion", "(I)Ljava/lang/String;");
         jmethodID midW = env->GetMethodID(cls, "setFitness", "(IFFI)V");

         // Copy the rule from each individual in the block from Java to native memory
         for(int i = 0; i < threadPopulationSize; i++)
         {
            strcpy(h_array_list[i],
            env->GetStringUTFChars((jstring)env->CallObjectMethod(indsList, midR, base+i),0));
         }                                                              460

         // Evaluation Host function call
         kernelFreitasCPU (h_result, h_array_list, h_instancesData, h_instancesClass, numberAttributes,
           numberInstances, threadPopulationSize,numClasses);

         // Reduction Host function call
         MC_kernelFreitasCPU (h_result, numberInstances, se, sp, bestClass,
           threadPopulationSize,numClasses);

         // Set the fitness to the individuals                          470
         for(int i = 0; i < threadPopulationSize; i++)
         {
             env->CallVoidMethod(indsList, midW, base + i, se[i],sp[i],bestClass[i]);
         }
      }
```

```
      }
      else
      {
         // Algorithm finished, Free dynamic memory
         for(int i = 0; i < plan->size; i++)                    480
         free(h_array_list[i]);
         free(h_array_list);
         for(int i = 0; i < numClasses; i++)
         free(h_result[i]);
         free(h_result);

         delete [] se;
         delete [] sp;
         delete [] bestClass;
      }                                                         490

      // Iteration finished
      SEM_POST(&post_sem[plan->thread]);

   }while(evaluate);

   jvm->DetachCurrentThread();
   CUT_THREADEND;
}
                                                              500
/**
 * Function executed when nativeMalloc() call from Java
 */
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_freitas_FreitasNativeEvaluator_nativeMalloc(JNIEnv *env, jobject obj,
  jint popSize, jstring dataset, jboolean useGPU, jint jnumThreads)
{
    // Set the number of threads
   numThreads = jnumThreads;
                                                              510
   // Set up semaphores
   for(int i = 0; i < numThreads; i++)
   {
      SEM_INIT (&wait_sem[i], 0);
      SEM_INIT (&post_sem[i], 0);
   }

    // Open native dataset
   FILE *f;
   if ((f = fopen(env->GetStringUTFChars(dataset,0),"r")) == NULL) exit(0);          520

    // Read configuration parameters
   fscanf(f,"%d%d%d",&numberAttributes,&numberInstances, &numClasses);

   // Calculate the aligned number of instances
   numberInstances_A = ceil(numberInstances/16.0)*16;

   // Dynamic host memory allocation
```

```
h_instancesData = (float*)malloc(numberAttributes*numberInstances_A*sizeof(float));
h_instancesClass = (int*)malloc(numberInstances*sizeof(int));

 // Read dataset from file
for(int i=0; i < numberInstances; i++)
{
   for(int j=0; j < numberAttributes; j++)
   fscanf(f,"%f",&h_instancesData[j*numberInstances_A+i]);

   fscanf(f,"%d",&h_instancesClass[i]);
}

 // Set up threads plans
for(int i = 0; i < numThreads; i++)
{
   plan[i].thread = i;
   plan[i].size = (int)ceil(popSize/(float)numThreads);
}

// If GPU usage launch GPU devices control threads
if(useGPU)
{
   #ifdef __USE_GPU__
   int deviceCount;
   cudaGetDeviceCount(&deviceCount);

   if(numThreads > deviceCount)
   {
      fprintf(stderr, "Can't use %d threads. CUDA devices (non-display) count is %d\n",
        numThreads, deviceCount);
      exit(0);
   }

   for(int i = 0; i < numThreads; i++)
       threadID[i] = cutStartThread((CUT_THREADROUTINE)gpuThreadFreitas, (void *)&plan[i]);
   #endif

   #ifndef __USE_GPU__
   fprintf(stderr, "This code has not been compiled for GPU usage");
   exit(0);
   #endif
}
else
{
    // Launch Host threads
   for(int i = 0; i < numThreads; i++)
       threadID[i] = cutStartThread((CUT_THREADROUTINE)cpuThreadFreitas, (void *)&plan[i]);
}

fclose(f);

// SIGNAL: threads ready to evaluate
for(int i = 0; i < numThreads; i++)
```

```
    SEM_WAIT (&post_sem[i]);
}

/**
 * Function executed when nativeEvaluate() call from Java
 *
 * param The number of individuals, the actual class to classify and the list of individuals
*/
JNIEXPORT void JNICALL
Java_net_sf_jclec_problem_classification_freitas_FreitasNativeEvaluator_nativeEvaluate(JNIEnv *env, jobject obj,
  jint size, jobject indsListArg)
{
    evaluate = true;

    indsList = indsListArg;
    populationSize = size;

    // SIGNAL: wake up threads to evaluate
    for(int i = 0; i < numThreads && i < size; i++)
    SEM_POST (&wait_sem[i]);

    // Wait until threads finish
    for(int i = 0; i < numThreads && i < size; i++)
    SEM_WAIT (&post_sem[i]);
}
```

B.5. functions.cu

```
#ifdef __USE_GPU__

/**
 * Checks if a character is a digit
 *
 * param The character
 * returns True or False
 */
__device__ bool isdigit_gpu(char c)
{
    if ((int)c >= 48 && (int)c <= 57)
        return true;
    return false;
}

/**
 * Returns the float value of a character array
 *
 * param Pointer to the char array
 * returns The float value
 */
__device__ float atof_gpu(const char *str)
{
```

```
float number = 0.;
int exponent = 0;
int negative = 0;
char *p = (char *) str;
float p10;
int n;
int num_digits = 0;
int num_decimals = 0;

// Checks if the number is negative
if(*p == '-')
{
   negative = 1;
   p++;
}

// Gets the integer portion of the number
while (isdigit_gpu(*p))
{
  number = number * 10. + (*p - '0');
  p++;
  num_digits++;
}

// If the number is float gets the float portion of the number
if (*p == '.')
{
  p++;

  while (isdigit_gpu(*p))
  {
    number = number * 10. + (*p - '0');
    p++;
    num_digits++;
    num_decimals++;
  }

  exponent -= num_decimals;
}

if (negative) number = -number;
negative = 0;

// If exponential portion gets it
if (*p == 'e' || *p == 'E')
{
  switch(*++p)
  {
    case '-': negative = 1;
    case '+': p++;
  }
  n = 0;
```

```
    while (isdigit_gpu(*p))
    {
      n = n * 10 + (*p - '0');
      p++;
    }

    if (negative)
      exponent -= n;
    else
      exponent += n;
  }

  // Calculate number value
  p10 = 10.;
  n = exponent;
  if (n < 0) n = -n;
  while (n)
  {
    if (n & 1)
    {
      if (exponent < 0)
        number /= p10;
      else
        number *= p10;
    }
    n >>= 1;
    p10 *= p10;
  }

  return number;
}

/**
 * Returns the integer value of a character array
 *
 * param Pointer to the char array
 * returns The integer value
 */
__device__ int atoi_gpu(const char *s)
{
        int a = 0, c;
        while ((c = *s++) != '\0' && isdigit_gpu(c)) {
                a = a*10 + (c - '0');
        }
        return a;
}

/**
 * Pushes the value into the stack and increments the stack depth pointer
 *
 * param The value, the stack and the stack depth pointer
*/
__device__ void push(float f, float pila[], int* sp)
```

```
{
   pila[(*sp)++] = f;
}

/**
 * Pops the float value from the stack and decrements the pointer
 *
 * param The stack and the stack depth pointer
 * returns The value
 */
__device__ float pop(float pila[],int* sp)
{
   return pila[--(*sp)];
}

/**
 * Gets the character located at the pointer from the buffer
 *
 * param The stack and the stack depth pointer
 * returns The character
 */
__device__ int getchar(int* bufp, char buf[])
{
   return buf[(*bufp)++];
}

/**
 * Reads the following rule's operator that copies into the buffer and returns the operator's type
 *
 * param The rule characters array, the rule pointer and the buffer
 * returns The operator's type
 */
__device__ int getnode(char s[],int* bufp, char buf[])
{
  int i = 1,c;
  bool number = false;
  s[0] = '0';
  while ((s[1] = c = getchar(bufp,buf)) == ' ');

  if (!isdigit_gpu(c))
  {
    // Checks the first character of the operator
     switch(c)
     {
      case('A'):
        (*bufp)+=2;
         return AND;
    case('O'):
       if((c = getchar(bufp,buf)) == 'R')
           return OR;
       else
       {
          (*bufp)++;
```

```
            return _OUT;
        }
    case('N'):
         (*bufp)+=2;
          return NOT;
    case('I'):
         (*bufp)++;
        return _IN;                                                    190
    case('<'):
       if(buf[*bufp] == '=')
       {
          (*bufp)++;
          return LESS_EQ;
       }
       else   return '<';
    case('>'):
       if(buf[*bufp] == '=')
       {                                                               200
          (*bufp)++;
          return GREATER_EQ;
       }
       else   return '>';
    case('-'):
       s[0] = '-';
       i--;
       break;
    default:
       return c;                                                       210
     }
  }

  while ((s[++i] = c = getchar(bufp,buf)) != ' ')
  if (c == '.') number = true;

  s[i] = 0;
  if(number) return NUMBER;
  else       return VAR;
}                                                                      220

/**
 * Checks if the rule covers the instance
 *
 * param The number of the instance, the rule, the instances data matrix and the number of instances aligned
 * returns True or False
 */
__device__ bool covers(int instance, char* expr, float* instancesData, int numberInstances_A)
{
      int sp = 0, bufp = 1;                                            230
      float stack[MAX_STACK];
      char s[MAX_OP];
      float op1, op2, arg, min, max;

        // While there are more nodes to parse
```

```
  while(1)
{
    // Get the next node and perform action
   switch(getnode(s, &bufp, expr))
   {                                                                    240
     case NUMBER:
       push(atof_gpu(s), stack, &sp);
       break;
     case VAR:
       push(instancesData[instance + numberInstances_A*atoi_gpu(s)], stack, &sp);
       break;
     case AND:
       op1 = pop(stack, &sp);
       op2 = pop(stack, &sp);
       if (op1 * op2 == 1)          push(1, stack, &sp);                 250
       else                       push(0, stack, &sp);
       break;
     case OR:
       op1 = pop(stack, &sp);
       op2 = pop(stack, &sp);
       if (op1 == 1 || op2 == 1)   push(1, stack, &sp);
       else                        push(0, stack, &sp);
       break;
     case NOT:
       op1 = pop(stack, &sp);                                           260
       if(op1 == 0)              push(1, stack, &sp);
       if(op1 == 1)              push(0, stack, &sp);
       break;
     case '>':
       op1 = pop(stack, &sp);
       if (op1 > pop(stack, &sp))   push(1, stack, &sp);
       else                       push(0, stack, &sp);
       break;
     case '<':
       op1 = pop(stack, &sp);                                           270
       if (op1 < pop(stack, &sp))   push(1, stack, &sp);
       else                       push(0, stack, &sp);
       break;
     case GREATER_EQ:
       op1 = pop(stack, &sp);
       op2 = pop(stack, &sp);
       if (op1 >= op2)             push(1, stack, &sp);
       else                       push(0, stack, &sp);
       break;
     case LESS_EQ:                                                      280
       op1 = pop(stack, &sp);
       op2 = pop(stack, &sp);
       if (op1 <= op2)             push(1, stack, &sp);
       else                       push(0, stack, &sp);
       break;
     case _OUT:
       arg = pop(stack, &sp);
       min = pop(stack, &sp);
```

```
            max = pop(stack, &sp);
            if(min > max){
               float aux = min;
               min = max;
               max = aux;
            }
            if (arg <= min || arg >= max) push(1, stack, &sp);
            else                          push(0, stack, &sp);
            break;
         case _IN:
            arg = pop(stack, &sp);
            min = pop(stack, &sp);
            max = pop(stack, &sp);
            if(min > max){
               float aux = min;
               min = max;
               max = aux;
            }
            if (arg > min && arg < max) push(1, stack, &sp);
            else                        push(0, stack, &sp);
            break;
         case '=':
            if (pop(stack, &sp) == pop(stack, &sp)) push(1, stack, &sp);
            else                                    push(0, stack, &sp);
            break;
         case ')':
            if(pop(stack, &sp) == 1)
                return true;
            else
                return false;
         default:
            break;
      }
   }
}

#endif

/**
 * Pushes the value into the stack and increments the stack depth pointer
 *
 * param The value, the stack and the stack depth pointer
*/
void pushCPU(float f, float pila[], int* sp)
{
   pila[(*sp)++] = f;
}

/**
 * Pops the float value from the stack and decrements the pointer
 *
 * param The stack and the stack depth pointer
 * returns The value
```

```
 */
float popCPU(float pila[],int* sp)
{
   return pila[--(*sp)];
}

/**
 * Gets the character located at the pointer from the buffer
 *
 * param The stack and the stack depth pointer
 * returns The character
 */
int getcharCPU(int* bufp, char buf[])
{
   return buf[(*bufp)++];
}

/**
 * Reads the following rule's operator that copies into the buffer and returns the operator's type
 *
 * param The rule characters array, the rule pointer and the buffer
 * returns The operator's type
 */
int getnodeCPU(char s[],int* bufp, char buf[])
{
  int i = 1,c;
  bool number = false;
  s[0] = '0';
  while ((s[1] = c = getcharCPU(bufp,buf)) == ' ');

  if (!isdigit(c))
  {
    // Checks the first character of the operator
     switch(c)
     {
      case('A'):
        (*bufp)+=2;
         return AND;
    case('O'):
       if((c = getcharCPU(bufp,buf)) == 'R')
           return OR;
       else
       {
          (*bufp)++;
          return _OUT;
       }
    case('N'):
         (*bufp)+=2;
          return NOT;
    case('I'):
         (*bufp)++;
        return _IN;
    case('<'):
```

```
        if(buf[*bufp] == '=')
        {
           (*bufp)++;
           return LESS_EQ;
        }
        else   return '<';
     case('>'):
        if(buf[*bufp] == '=')
        {
           (*bufp)++;
           return GREATER_EQ;
        }
        else   return '>';
     case('-'):
        s[0] = '-';
        i--;
        break;
     default:
        return c;
      }
  }

  while ((s[++i] = c = getcharCPU(bufp,buf)) != ' ')
  if (c == '.') number = true;

  s[i] = 0;
  if(number)  return NUMBER;
  else        return VAR;
}

/**
 * Checks if the rule covers the instance
 *
 * param The number of the instance, the rule, the instances data matrix and the number of instances aligned
 * returns True or False
 */
bool coversCPU(int instance, char* expr, float* instancesData, int numberInstances_A)
{
      int sp = 0, bufp = 1;
      float stack[MAX_STACK];
      char s[MAX_OP];
      float op1, op2, arg, min, max;

        // While there are more nodes to parse
        while(1)
      {
            // Get the next node and perform action
         switch(getnodeCPU(s, &bufp, expr))
         {
            case NUMBER:
               pushCPU(atof(s), stack, &sp);
               break;
            case VAR:
```

```
    pushCPU(instancesData[instance + numberInstances_A*atoi(s)], stack, &sp);
    break;
  case AND:                                                              450
    op1 = popCPU(stack, &sp);
    op2 = popCPU(stack, &sp);
    if (op1 * op2 == 1)          pushCPU(1, stack, &sp);
    else                        pushCPU(0, stack, &sp);
    break;
  case OR:
    op1 = popCPU(stack, &sp);
    op2 = popCPU(stack, &sp);
    if (op1 == 1 || op2 == 1)  pushCPU(1, stack, &sp);
    else                        pushCPU(0, stack, &sp);                  460
    break;
  case NOT:
    op1 = popCPU(stack, &sp);
    if(op1 == 0)                pushCPU(1, stack, &sp);
    if(op1 == 1)                pushCPU(0, stack, &sp);
    break;
  case '>':
    op1 = popCPU(stack, &sp);
    if (op1 > popCPU(stack, &sp)) pushCPU(1, stack, &sp);
    else                        pushCPU(0, stack, &sp);                  470
    break;
  case '<':
    op1 = popCPU(stack, &sp);
    if (op1 < popCPU(stack, &sp)) pushCPU(1, stack, &sp);
    else                        pushCPU(0, stack, &sp);
    break;
  case GREATER_EQ:
    op1 = popCPU(stack, &sp);
    op2 = popCPU(stack, &sp);
    if (op1 >= op2)              pushCPU(1, stack, &sp);                 480
    else                        pushCPU(0, stack, &sp);
    break;
  case LESS_EQ:
    op1 = popCPU(stack, &sp);
    op2 = popCPU(stack, &sp);
    if (op1 <= op2)              pushCPU(1, stack, &sp);
    else                        pushCPU(0, stack, &sp);
    break;
  case _OUT:
    arg = popCPU(stack, &sp);                                            490
    min = popCPU(stack, &sp);
    max = popCPU(stack, &sp);
    if(min > max){
      float aux = min;
      min = max;
      max = aux;
    }
    if (arg <= min || arg >= max) pushCPU(1, stack, &sp);
    else                          pushCPU(0, stack, &sp);
    break;                                                               500
```

```
            case _IN:
               arg = popCPU(stack, &sp);
               min = popCPU(stack, &sp);
               max = popCPU(stack, &sp);
               if(min > max){
                  float aux = min;
                  min = max;
                  max = aux;
               }
               if (arg > min && arg < max) pushCPU(1, stack, &sp);
               else                        pushCPU(0, stack, &sp);
               break;
            case '=':
               if (popCPU(stack, &sp) == popCPU(stack, &sp)) pushCPU(1, stack, &sp);
               else                                          pushCPU(0, stack, &sp);
               break;
            case ')':
               if(popCPU(stack, &sp) == 1)
                  return true;
               else
                  return false;
            default:
               break;
         }
      }
}
```

B.6. parameters.h

```
#ifndef _PARAM_H_
#define _PARAM_H_

// Required includes

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <jni.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#if _WIN32
#include "multithreading.cpp"
#include <windows.h>
#else
#include "multithreading.h"
#endif

// Include CUDA libs if GPU compilation
#ifdef __USE_GPU__
#include <cuda.h>
```

```
#include <cutil_inline.h>
#include <cuda_runtime_api.h>
#endif

// Include JNI interfaces

#include "jni/net_sf_jclec_problem_classification_freitas_FreitasNativeEvaluator.h"
#include "jni/net_sf_jclec_problem_classification_falco_FalcoNativeEvaluator.h"
#include "jni/net_sf_jclec_problem_classification_tan_TanNativeEvaluator.h"

using namespace std;

// Maximum number of CPU threads or GPU devices [0-16]
#define MAX_THREADS 16
// Number of threads per block at evaluation kernels
#define THREADS_EVAL_BLOCK 128

// Maximum number of individuals evaluated concurrently [0-256] Recommended values 32*i
#define BLOCK_SIZE_FALCO 256
#define BLOCK_SIZE_FREITAS 128
#define BLOCK_SIZE_TAN 256

// Plan structure to let tasks know its thread number and population size
typedef struct {
    int thread;
    int size;
} Plan;

// Maximum stack depth
#define MAX_STACK 16
// Maximum node characters length
#define MAX_OP 32
// Maximum rule characters length
#define MAX_EXPR_LEN 256

#define NUMBER 0
#define AND 1
#define OR 2
#define NOT 3
#define _OUT 4
#define _IN 5
#define GREATER_EQ 6
#define LESS_EQ 7
#define VAR 8

#endif
```

B.7. multithreading.cpp

```
#include "multithreading.h"
```

```
#if _WIN32
    //Create thread
    CUTThread cutStartThread(CUT_THREADROUTINE func, void *data){
        return CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)func, data, 0, NULL);
    }

    //Wait for thread to finish
    void cutEndThread(CUTThread thread){
        WaitForSingleObject(thread, INFINITE);
        CloseHandle(thread);
    }

    //Destroy thread
    void cutDestroyThread(CUTThread thread){
        TerminateThread(thread, 0);
        CloseHandle(thread);
    }

    //Wait for multiple threads
    void cutWaitForThreads(const CUTThread * threads, int num){
        WaitForMultipleObjects(num, threads, true, INFINITE);

        for(int i = 0; i < num; i++)
            CloseHandle(threads[i]);
    }

#else
    //Create thread
    CUTThread cutStartThread(CUT_THREADROUTINE func, void * data){
        pthread_t thread;
        pthread_create(&thread, NULL, func, data);
        return thread;
    }

    //Wait for thread to finish
    void cutEndThread(CUTThread thread){
        pthread_join(thread, NULL);
    }

    //Destroy thread
    void cutDestroyThread(CUTThread thread){
        pthread_cancel(thread);
    }

    //Wait for multiple threads
    void cutWaitForThreads(const CUTThread * threads, int num){
        for(int i = 0; i < num; i++)
            cutEndThread(threads[i]);
    }

#endif
```

B.8. multithreading.h

```
#ifndef MULTITHREADING_H
#define MULTITHREADING_H

//Simple portable thread library.

#if _WIN32
   //Windows threads.
   #include <windows.h>

   typedef HANDLE CUTThread;
   typedef unsigned (WINAPI *CUT_THREADROUTINE)(void *);

   #define CUT_THREADPROC unsigned WINAPI
   #define CUT_THREADEND return 0
   #define SEMAPHORE HANDLE
   #define SEM_INIT(pobject,pattr) (*pobject=CreateSemaphore(NULL,0,1,NULL))
   #define SEM_WAIT(pobject) WaitForSingleObject(*pobject,INFINITE)
   #define SEM_POST(pobject) ReleaseSemaphore(*pobject,1,NULL)

#else
   //POSIX threads.
   #include <pthread.h>
   #include <semaphore.h>

   typedef pthread_t CUTThread;
   typedef void *(*CUT_THREADROUTINE)(void *);

   #define CUT_THREADPROC void
   #define CUT_THREADEND pthread_exit(NULL)
   #define SEMAPHORE sem_t
   #define SEM_INIT(pobject,pattr) sem_init(pobject,0,0)
   #define SEM_WAIT(pobject) sem_wait(pobject)
   #define SEM_POST(pobject) sem_post(pobject)

#endif

#ifdef __cplusplus
   extern "C" {
#endif

//Create thread.
CUTThread cutStartThread(CUT_THREADROUTINE, void *data);

//Wait for thread to finish.
void cutEndThread(CUTThread thread);

//Destroy thread.
void cutDestroyThread(CUTThread thread);

//Wait for multiple threads.
void cutWaitForThreads(const CUTThread *threads, int num);
```

```
#ifdef __cplusplus
} //extern "C"
#endif

#endif //MULTITHREADING_H
```

B.9. Makefile

```
##############################################################################################
# JCLEC - NATIVE
# Makefile Version 1.04
# May 20 2010
##############################################################################################
#
# Usage: make             CPU and GPU code compilation
#            make gpu=0   only for CPU code
#
#               PLEASE MAKE SURE TO UPDATE DIR PATHS BEFORE MAKE    10
#
##############################################################################################

# Machine architecture -m32 -m64
ARCH = -m64
# Java path
JAVA_DIR = /usr/lib/jvm/java-6-sun

##############################################################################################
# GPU SETTINGS (ONLY IF GPU COMPILATION                                                    20

# GPU architecture -arch=sm_10 -arch=sm_11 -arch=sm_12 -arch=sm_13 -arch=sm_20
GPU_ARCH = -arch=sm_13
# NVIDIA CUDA SDK path
NVIDIA_DIR = /home/acano/NVIDIA_GPU_Computing_SDK
# NVIDIA CUDA Toolkit path
CUDA_DIR = /usr/local/cuda
# /lib64 for 64 bit Systems /lib for 32 bit Systems
CUDA_LIB_DIR = $(CUDA_DIR)/lib64
                                                                                         30
# DO NOT EDIT BELOW THIS LINE
##############################################################################################

ifeq ($(gpu), 0)
  SRC_FILE := jclec-native.c
  CUDA_INC :=
  CUDA_LIB :=
  COMPILER := g++ -fPIC
  FLAGS :=
else                                                                                     40
  SRC_FILE := jclec-native.cu
  CUDA_INC := -I$(CUDA_DIR)/include -I$(NVIDIA_DIR)/C/common/inc
```

```
  CUDA_LIB := -L$(CUDA_LIB_DIR) -L$(NVIDIA_DIR)/C/lib -L$(NVIDIA_DIR)/C/common/lib/linux
  COMPILER := nvcc --ptxas-options=-v $(GPU_ARCH) -Xcompiler -fPIC $(CUDA_INC) $(CUDA_LIB)
  FLAGS := -lcuda -lcudart -D__USE_GPU__
endif

todo: javah multithreading native lib
javah:
	javah -classpath ../bin -jni -d ./jni/ net.sf.jclec.problem.classification.freitas.FreitasNativeEvaluator
	javah -classpath ../bin -jni -d ./jni/ net.sf.jclec.problem.classification.falco.FalcoNativeEvaluator
	javah -classpath ../bin -jni -d ./jni/ net.sf.jclec.problem.classification.tan.TanNativeEvaluator
multithreading: multithreading.cpp
	g++ $(ARCH) -O3 -fPIC -lpthread -c -o multithreading.o multithreading.cpp -I. $(CUDA_INC)
native:
	$(COMPILER) -O3 $(ARCH) -shared -I. -I$(JAVA_DIR)/include/linux -I$(JAVA_DIR)/include
	-c $(SRC_FILE) -o jclec-native.o $(FLAGS) -lpthread -lm
ptx:
	nvcc $(GPU_ARCH) -ptx jclec-native.cu $(CUDA_INC) $(CUDA_LIB) -O3 $(ARCH)
	-I$(JAVA_DIR)/include/linux -I$(JAVA_DIR)/include $(FLAGS)
lib: jclec-native.o
	g++ $(ARCH) -O3 -fPIC -shared $(CUDA_INC) -L/usr/lib $(CUDA_LIB) -Wl,-soname,jclec_native
	multithreading.ojclec-native.o -o libjclec_native.so -lc -lpthread $(FLAGS)
clean:
	rm multithreading.o jclec-native.o libjclec_native.so
```

B.10. FalcoNativeEvaluator.java

```
package net.sf.jclec.problem.classification.falco;

import java.io.File;
import java.util.List;

import net.sf.jclec.IConfigure;
import net.sf.jclec.IIndividual;
import net.sf.jclec.fitness.SimpleValueFitness;
import net.sf.jclec.problem.classification.Individual;
import net.sf.jclec.problem.classification.SyntaxTreeClassificationRule;

import org.apache.commons.configuration.Configuration;

/**
 * Native Evaluator for Falco et al.
 *
 */

public class FalcoNativeEvaluator extends FalcoEvaluator implements IConfigure
{
	/** Native functions */
	public native void nativeEvaluate(int numInds, int classifiedClass, FalcoNativeEvaluator object);
	public native void nativeMalloc(int popSize, String dataset, boolean useGPU, int numThreads);
	public native void nativeFree();
```

```
/** Individuals list to evaluate */
public List<IIndividual> indsToEvaluate;

public FalcoNativeEvaluator()
{
        super();
}

////////////////////////////////////////////////////////////////
// ———————— Overwriting FreitasEvaluator methods
////////////////////////////////////////////////////////////////

/**
 * Configuration method.
 *
 */

public void configure(Configuration settings) {

        double alpha = settings.getDouble("alpha");
        setAlpha(alpha);

        if(settings.getString("native-dataset") == null || settings.getString("use-gpu") == null ||
           settings.getString("number-threads") == null || settings.getString("population-size") == nu
        {
                System.out.println(" native-dataset | use-gpu | number-threads | population-size ta
                System.exit(0);
        }

        File f = new File(settings.getString("native-dataset"));
        if(!f.exists())
        {
                System.out.println("No such native-dataset file");
                System.exit(0);
        }

        try {
                System.loadLibrary("jclec_native");
        } catch (Exception e) {
                System.out.println("Can't load jclec_native library. Please make sure to include na
                System.exit(0);
        }

        nativeMalloc(Integer.parseInt(settings.getString("population-size")),settings.getString("native-da
                        Boolean.parseBoolean(settings.getString("use-gpu")),Integer.parseInt(settings.ge
}

/**
 * Get and return the index individual's antecedent as a String
 *
 * param the index from the individual at indsToEvaluate
 * return the antecedent
 */
```

```
        public String getReverseExpression(int index) {                                    80
                return ((SyntaxTreeClassificationRule) ((Individual) indsToEvaluate.get(index)).getPhenotype()).ge
        }

        /**
         * Set the fitness to index individual
         *
         * param the index from the individual at indsToEvaluate and its fitness
*/

        public void setFitness(int index, float fails)                                     90
        {
                Individual ind = (Individual) indsToEvaluate.get(index);

                int numnodes = ind.getGenotype().derivSize();

                ind.setFitness(new SimpleValueFitness(fails + getAlpha()*numnodes));
        }

        /**
         * Apply function over phenotype, then create a instance of the                     100
         * SimpleValueFitness class with its value set to function value
         * and assigns it to the individual and assigns the consequent.
         *
         * param ind Individual to evaluate
*/

        public void evaluate(List<IIndividual> inds)
        {
                indsToEvaluate = inds;
                                                                                           110
                if(inds.size() > 0)
                {
                        for (IIndividual ind : inds) {
                                if(ind.getFitness() == null)
                                        numberOfEvaluations++;
                        }

                        nativeEvaluate(inds.size(), classifiedClass, this);
                }
        }                                                                                  120
}
```

B.11. TanNativeEvaluator.java

```
package net.sf.jclec.problem.classification.tan;

import java.io.File;
import java.util.List;
```

```
import net.sf.jclec.IConfigure;
import net.sf.jclec.IIndividual;
import net.sf.jclec.fitness.SimpleValueFitness;
import net.sf.jclec.problem.classification.Individual;
import net.sf.jclec.problem.classification.SyntaxTreeClassificationRule;

import org.apache.commons.configuration.Configuration;

/**
 * Native Evaluator for Tan et al. 2002
 *
 */

public class TanNativeEvaluator extends TanEvaluator implements IConfigure
{
	/** Native functions */
	public native void nativeEvaluate(int numInds, int classifiedClass, float w1, float w2, TanNativeEvalua
	public native void nativeMalloc(int popSize, String dataset, boolean useGPU, int numThreads);
	public native void nativeFree();

	/** Individuals list to evaluate */
	public List<IIndividual> indsToEvaluate;

	public TanNativeEvaluator()
	{
		super();
	}

	////////////////////////////////////////////////////////////////
	// ------------ Overwriting FreitasEvaluator methods
	////////////////////////////////////////////////////////////////

	/**
	 * Configuration method.
	 *
	 */

	public void configure(Configuration settings) {

		float w1 = settings.getFloat("w1");
		setW1(w1);

		float w2 = settings.getFloat("w2");
		setW2(w2);

		if(settings.getString("native-dataset") == null || settings.getString("use-gpu") == null ||
		   settings.getString("number-threads") == null || settings.getString("population-size") == nu
		{
			System.out.println(" native-dataset | use-gpu | number-threads | population-size ta
			System.exit(0);
		}

		File f = new File(settings.getString("native-dataset"));
```

```
        if(!f.exists())
        {
                System.out.println("No such native-dataset file");
                System.exit(0);
        }

        try {
                System.loadLibrary("jclec_native");
        } catch (Exception e) {
                System.out.println("Can't load jclec_native library. Please make sure to include na
                System.exit(0);
        }

        nativeMalloc(Integer.parseInt(settings.getString("population-size")),settings.getString("native-da
                        Boolean.parseBoolean(settings.getString("use-gpu")),Integer.parseInt(settings.ge
}

/**
 * Get and return the index individual's antecedent as a String
 *
 * param the index from the individual at indsToEvaluate
 * return the antecedent
 */

public String getReverseExpression(int index) {
        return ((SyntaxTreeClassificationRule) ((Individual) indsToEvaluate.get(index)).getPhenotype()).ge
}

/**
 * Set the fitness to index individual
 *
 * param the index from the individual at indsToEvaluate and its fitness
*/

public void setFitness(int index, float fitness)
{
        Individual ind = (Individual) indsToEvaluate.get(index);

        ind.setFitness(new SimpleValueFitness(fitness));
}

/**
 * Apply function over phenotype, then create a instance of the
 * SimpleValueFitness class with its value set to function value
 * and assigns it to the individual and assigns the consequent.
 *
 * param ind Individual to evaluate
*/

public void evaluate(List<IIndividual> inds)
{
        indsToEvaluate = inds;
```

```
                if(inds.size() > 0)
                {
                        for (IIndividual ind : inds) {
                                if(ind.getFitness() == null)
                                        numberOfEvaluations++;
                        }

                        nativeEvaluate(inds.size(), classifiedClass, w1, w2, this);
                }
        }
}
```

B.12. FreitasNativeEvaluator.java

```
package net.sf.jclec.problem.classification.freitas;

import java.io.File;
import java.util.List;

import net.sf.jclec.IConfigure;
import net.sf.jclec.IIndividual;
import net.sf.jclec.fitness.SimpleValueFitness;
import net.sf.jclec.problem.classification.Individual;
import net.sf.jclec.problem.classification.SyntaxTreeClassificationRule;

import org.apache.commons.configuration.Configuration;

/**
 * Native Evaluator for Bojarczuk et al.
 *
 */

public class FreitasNativeEvaluator extends FreitasEvaluator implements IConfigure
{
        /** Native functions */
        public native void nativeEvaluate(int numInds, FreitasNativeEvaluator object);
        public native void nativeMalloc(int popSize, String dataset, boolean useGPU, int numThreads);
        public native void nativeFree();

        /** Individuals list to evaluate */
        public List<IIndividual> indsToEvaluate;

        public FreitasNativeEvaluator()
        {
                super();
        }

        ////////////////////////////////////////////////////////////
        // ---------- Overwriting FreitasEvaluator methods
        ////////////////////////////////////////////////////////////
```

```
/**
 * Configuration method.
 *
 */

public void configure(Configuration settings) {

        if(settings.getString("native-dataset") == null || settings.getString("use-gpu") == null ||
           settings.getString("number-threads") == null || settings.getString("population-size") == nu
        {
                System.out.println(" native-dataset | use-gpu | number-threads | population-size ta
                System.exit(0);
        }

        File f = new File(settings.getString("native-dataset"));
        if(!f.exists())
        {
                System.out.println("No such native-dataset file");
                System.exit(0);
        }

        try {
                System.loadLibrary("jclec_native");
        } catch (Exception e) {
                System.out.println("Can't load jclec_native library. Please make sure to include na
                System.exit(0);
        }

        nativeMalloc(Integer.parseInt(settings.getString("population-size")),settings.getString("native-da
                        Boolean.parseBoolean(settings.getString("use-gpu")),Integer.parseInt(settings.ge
}

/**
 * Get and return the index individual's antecedent as a String
 *
 * param the index from the individual at indList
 * return the antecedent
 */

public String getReverseExpression(int index) {
        return ((SyntaxTreeClassificationRule) ((Individual) indsToEvaluate.get(index)).getPhenotype()).ge
}

/**
 * Set the fitness to index individual
 *
 * param the index from the individual at indList and its se, sp and bestClass values
*/

public void setFitness(int index, float se, float sp, int bestClass)
{
        Individual ind = (Individual) indsToEvaluate.get(index);
```

```
                rule = (SyntaxTreeClassificationRule) ind.getPhenotype();

                // Assign as consequent the class that reports the best fitness
                rule.setConsequent(bestClass);

                double numnodes = ind.getGenotype().derivSize();

                double sy = (getMaxDerivSize() - 0.5*numnodes - 0.5)/(getMaxDerivSize()-1);

                ind.setFitness(new SimpleValueFitness(se*sp*sy));                    100
        }

        /**
         * Apply function over phenotype, then create a instance of the
         * SimpleValueFitness class with its value set to function value
         * and assigns it to the individual and assigns the consequent.
         *
         * param ind Individual to evaluate
*/
                                                                                     110
        public void evaluate(List<IIndividual> inds)
        {
                indsToEvaluate = inds;

                if(inds.size() > 0)
                {
                        for (IIndividual ind : inds) {
                                if(ind.getFitness() == null)
                                        numberOfEvaluations++;
                        }                                                            120

                        nativeEvaluate(inds.size(), this);
                }
        }
}
```