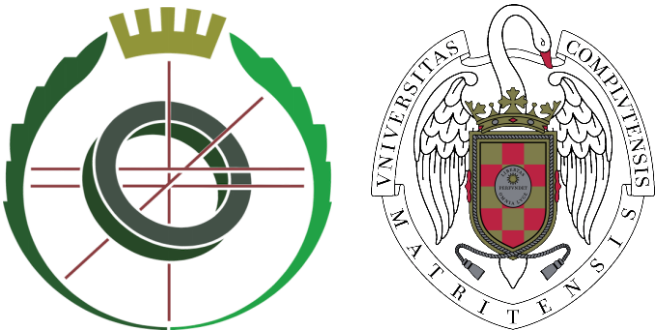
Trabajo de fin de Grado

Grado en Ingeniería Informática

Implementación de Gramáticas Evolutivas sobre GPUs

Facultad de Informática



Universidad Complutense de Madrid

Director: José Ignacio Hidalgo Pérez

Autor: Ismael Gonjal Montero

Agradecimientos

Este trabajo se realizó bajo la supervisión de José Ignacio Hidalgo Pérez. A él me gustaría mostrar mi total agradecimiento por la cantidad de cosas que me ha hecho ver que soy capaz de hacer, a veces a regañadientes. Por sus despistes y los míos, y por ver cada aspecto que podría resultar en algo mejor.

A mis padres, Julio y Gloria, por aguantar todo ese mal humor que he derrochado cuando las cosas no salían bien, por todas las risas de estos años y por todos los llantos. A mi hermana Gloria, por las veces que no te atendí al estar trabajando, por los momentos de buen humor tanto como por las peleas. A Paula, por darme ánimos siempre, por mantenerme en este barco y ayudarme a no escorar. A vosotros cuatro porque cuando no he sido el mejor hijo, hermano o novio, no sólo no lo habéis dicho, sino que ni siquiera lo habéis pensado.

A mi amigo David, por tomarse la vida con filosofía. A mi amigo Jonás, por sembrar su carácter en tierra de vikingos. A vosotros, por mantenernos tan cerca estando tan lejos.

A Kata, porque siempre tiene lo que necesitas, o su ayuda y una pieza clave para empezar a construirlo.

A Zeki, porque sus ideas en conversaciones informales han ayudado a moldear este texto más de lo que él piensa.

A la asociación ASCII, por darme una familia en la universidad, una mesa en la cafetería y unos pocos quebraderos de cabeza.

Índice

Resumen

Resumen

En el presente trabajo se ha realizado la implementación y análisis de resultados de la ejecución de un algoritmo que utiliza gramáticas evolutivas sobre una tarjeta de procesamiento gráfico o GPU, y su comparación con una implementación equivalente sobre CPU. El problema sobre el que se ha aplicado es el conocido como de regresión simbólica, que consiste en obtener una expresión matemática de la forma y=f(x) a partir de un conjunto de pares x/y. Adicionalmente se ha diseñado una implementación de una gramática para resolver un problema de identificación de perfiles de glucosa en pacientes diabéticos para mostrar una posible aplicación.

Abstract

This work presents the implementation of an algorithm based on Grammatical Evolution under a GPU architecture. We made an analysis of the results in terms of execution time and quality. We compare to an equivalent implementation on a CPU. The benchmark of application is the symbolic regression problem, which consists on obtaining a mathematical expression in the form y=f(x) from a set of pairs x/y. We also present an implementation of a grammar for the identification of glucose profiles in diabetic patients as an application case.

Etiquetas de búsqueda

Gramáticas Evolutivas, Evolución Gramatical, GE, GPGPU, CUDA, Algoritmos Evolutivos NVidia, SIMD.

Search Tags

Grammatical Evolution, CPU, GPGPU, CUDA, Evolutionary Algorithms, NVidia, SIMD.

1.Introducción

Objetivo

El objetivo de este trabajo es la implementación eficiente de algoritmos evolutivos conocidos como de evolución gramatical (GE de sus siglas en inglés) sobre unidades de procesamiento gráfico de propósito general (GPGPU). El tipo de problemas sobre los que se va a trabajar son los de regresión simbólica, que consisten encontrar una ecuación $y = f(x)$ dada la regresión de x sobre y en un número discreto de puntos.

Introducción

En la actualidad uno de los aspectos que aparecen de forma reiterada, independientemente del área de aplicación, es el tratamiento de datos y de la información, incluyendo la identificación de patrones y el modelado mediante expresiones matemáticas. Los algoritmos evolutivos son herramientas de optimización y búsqueda basadas en la selección y evolución natural que han demostrado su utilidad en numerosos campos de aplicación en ciencia e ingeniería [1] [2] [3] [4].

Los algoritmos evolutivos (AEs) trabajan con un conjunto de soluciones en un proceso iterativo aplicando un conjunto de operadores que simulan los procesos que la naturaleza utiliza para evolucionar. Existen varios tipos de AEs, entre ellos encontramos los algoritmos de evolución gramatical (GE), o gramáticas evolutivas. Las GE tienen la particularidad de utilizar una gramática para decodificar las soluciones al problema, permitiendo acotar los espacios de búsqueda e incorporar conocimiento al proceso de búsqueda. Por ello, resultan de especial utilidad en problemas de identificación y modelado.

Uno de los problemas que puede aparecer cuando se atacan problemas reales mediante AEs, es el incremento en el tiempo de ejecución, dado que hay que realizar la evaluación de un elevado número de individuos. Además, en las GE, este tiempo

puede ser incluso mayor, dado que hay un proceso adicional de traducción mediante la gramática.

En los últimos años han aparecido distintas opciones para acelerar la ejecución de estos algoritmos [5]. Entre éstas, se encuentran algunos métodos como el uso de unidades de procesamiento gráfico de propósito general (GPGPUs [6]). El uso de unidades de procesamiento gráfico en este campo se debe a su capacidad de trabajar con múltiples datos en una sola instrucción, un tipo de computación ampliamente utilizada en el tratamiento de entornos bidimensionales o tridimensionales. En este trabajo se propone una implementación de GE sobre GPU en la que se realiza una implementación específica para cada una de las gramáticas utilizadas, aprovechando la gran cantidad de multiprocesadores de este tipo de hardware. Esta solución reduce la versatilidad, pero permite optimizar la implementación para un problema específico en términos de tiempo de computación.

Para comprobar la efectividad de esta propuesta se ha realizado un conjunto de pruebas sobre problemas de regresión simbólica utilizados habitualmente en la literatura [7]. Se han realizado medidas de la calidad de las soluciones obtenidas y de los tiempos de ejecución.

Los resultados demuestran que el tiempo de ejecución sobre GPU se reduce de media en un 966.94% con respecto a la implementación CPU, con un promedio de 0.18 segundos menos por generación (correlación de Pearson=0,6 en los tiempos de CPU y GPU). La calidad de las soluciones obtenidas se ve aumentada un 11.87% de media con una correlación de 0.99, un índice muy elevado que indica una correspondencia muy alta entre los resultados.

Adicionalmente se ha diseñado una implementación de una gramática para resolver un problema de identificación de perfiles de glucosa en pacientes diabéticos para mostrar una posible aplicación.

Adicionalmente se ha diseñado una implementación de una gramática para resolver un problema de identificación de perfiles de glucosa en pacientes diabéticos para mostrar una posible aplicación.

El resto del documento, después de la breve introducción a los objetivos y complejidades a las que se enfrenta el presente proyecto presentada en esta introducción está organizado como sigue. En el capítulo 2 se repasa la base teórica sobre la que se apoya la implementación ofreciendo, para comenzar, una descripción del funcionamiento de los algoritmos evolutivos y continuando con un esquema del funcionamiento de la GPU y los motivos de su idoneidad para nuestro problema.

En el capítulo 3 Tras esta base teórica continuaremos con un punto dedicado a detalles relevantes sobre las diferentes implementaciones hechas y las diferencias que hay entre ellas para pasar a un punto que hablará de los resultados obtenidos en la ejecución y la comparación de distintas implementaciones y gramáticas. Para finalizar, se describe un ejemplo de aplicación del algoritmo. Terminamos el documento con la descripción de las conclusiones y trabajo futuro.

2. Base Teórica

a. Algoritmos evolutivos

Introducción

Bien conocida es la tendencia humana de utilizar la naturaleza como inspiración a sus creaciones. En este sentido, la informática también suele tomar ideas del entorno para desarrollar programas más eficientes e imaginativos, siendo un buen ejemplo el algoritmo para recorrido de grafos basado en feromonas de hormigas [8] o el algoritmo de enfriamiento simulado [9]. Es importante para el avance de la ciencia la interdisciplinariedad y, por ello, combinando el concepto supervivencia del más fuerte de Darwin [10] y los relativamente recientes descubrimientos hechos en biología, bioquímica y, más concretamente en el campo de los genes por mentes preclaras como Gregor Mendel, se idean los algoritmos genéticos. Éstos mezclan conceptos sobre la teoría de la evolución, el intercambio y la mejora progresiva de la información presente en la carga genética de todas las células. Así, tomando la replicación del ADN como ejemplo, se desarrolló un innovador y potente paradigma de programación, la programación evolutiva.

Este tipo de programación recibe su nombre como consecuencia del gran parecido que comparte con la evolución del ADN. Los algoritmos genéticos usan una simulación de la mutación y cruce de cromosomas en la meiosis celular para hacer evolucionar poco a poco un conjunto de soluciones que harán las veces de ancestros hasta encontrar una población que cada vez se aproxime más al resultado deseado. Un dato importante sobre los algoritmos evolutivos es que, si bien la evolución será guiada hacia un resultado favorable, este tipo de programación no ofrece ninguna garantía de que se llegue a encontrar un óptimo global. Para buscar la solución a un problema se comienza con otras soluciones, inicialmente aleatorias, que serán consideradas los progenitores. Éstos procrearán siguiendo las leyes de la evolución hasta crear un resultado que se adapte a nuestras necesidades. En nuestro caso, sólo nos importará el resultado último y en ningún caso el proceso para conseguirlo.

Terminología

En este punto es necesario definir una serie de conceptos necesarios para poder entender los términos con los cuales nos referiremos a determinados aspectos de los algoritmos genéticos.

- Cromosoma. Array de números enteros que codifica un resultado del problema.
- Gen. Cada una de las posiciones del array que representa el cromosoma, es decir, la mínima unidad de información genética, representada por un número entero.
- Población. Consiste en una serie de cromosomas. La población inicial es el conjunto de cromosomas habitualmente generados de forma aleatoria y las sucesivas serán el resultado de la evolución. En nuestra implementación se representa mediante un array que contiene cromosomas.
- Generación. Fracción de la población originada tras el mismo número de iteraciones en el algoritmo genético, por lo que están en el mismo nivel del árbol de soluciones.
- Fitness. La función de fitness o función de idoneidad indica la calidad de cada individuo. En nuestro caso el mejor fitness será 0, lo que significa que el individuo poseedor del mismo proporciona exactamente la salida esperada. Sin embargo, no siempre es conocida la solución óptima al problema que se pretende resolver.

Diagrama de flujo de un algoritmo genético

El paradigma de programación de algoritmos evolutivos consta de una serie de pasos, en gran parte inspirados en la evolución y cruce de cromosomas en la naturaleza. El primer paso consiste en generar la población inicial, que tradicionalmente se crea de forma aleatoria y constituye el conjunto de ancestros originales a partir de los cuales comenzará la evolución. Tras haber creado una generación inicial, las siguientes acciones que se lleven a cabo se repetirán en ciclo, como representa la Figura 1.

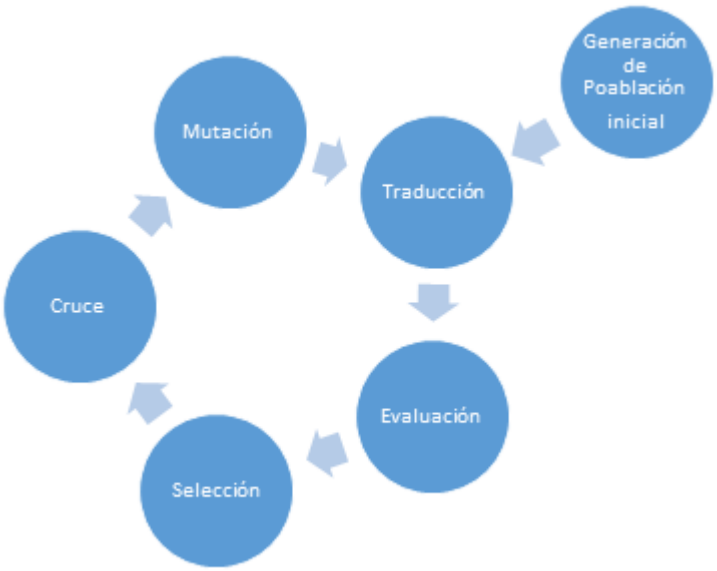


Figura 1. Esquema de una iteración en la evolución.

Llegados a este punto, tendremos una población de cromosomas más o menos válidos. Para poder continuar evaluando su validez y así elegir los resultados más prometedores, se les asignará una calificación con una función, a la que llamaremos de idoneidad o fitness. Ésta evaluará la calidad del resultado que representa ese cromosoma, tras lo cual se realizará la selección del conjunto de ancestros para la próxima generación. Esta selección se llevará a cabo mediante métodos que se detallan en apartados posteriores, escogiendo finalmente un grupo de individuos que deberá incluir mayoritariamente buenas soluciones sin hacer desaparecer soluciones en apariencia menos favorables, para que haya una buena variabilidad genética. Cuando los progenitores se han elegido y convertido en la población actual, comenzará el intercambio genético entre cromosomas.

El cruce consiste en intercambio de parte del material genético, y podrá contar con uno o más puntos de cruce. La zona de cruce puede ser aleatoria, aunque dependiendo del problema podría determinarse mediante otra heurística. La cantidad de puntos en que se realice el cruce también es variable. Una vez seleccionada la zona o zonas de cruce se intercambian regiones completas de los cromosomas, produciéndose el cruce (Figura 2).

Cuando los dos nuevos cromosomas hayan intercambiado material genético, (o no, pues depende de una cierta probabilidad de cruce) cada uno de los genes tendrá cierta posibilidad de mutar, es decir recibir un cambio aleatorio que tenga

probabilidades de producirse en uno o varios de los genes al azar y que modifique de forma espontánea la solución.

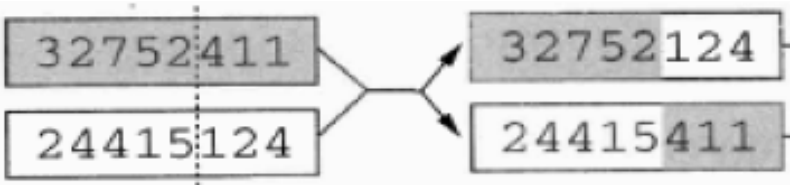


Figura 2. Esquema que describe el cruce de dos cromosomas.

El ciclo evolutivo se detendrá cuando se alcance el máximo de generaciones estipulado o cuando una o más de las soluciones superen el umbral mínimo de idoneidad, un valor que indica el mínimo fitness que debe alcanzar una solución para ser considerada como correcta. En caso de este programa el fitness mínimo será 0, haciendo necesario para la finalización del proceso de búsqueda que la población converja a un óptimo global o consuma el máximo número de generaciones permitido.

Selección

El proceso de selección es realmente importante en los algoritmos evolutivos [11], pues determina en gran medida qué individuos sobreviven y cuáles se extinguen. Para ello es clave la función de fitness, que señalará los cromosomas más adecuados para su supervivencia. En una primera aproximación general se suelen utilizar dos métodos para el cruce: la ruleta y el torneo.

En el método de la ruleta se toman todas las evaluaciones y a cada cromosoma se le asigna una probabilidad de ser seleccionado acorde a su calidad. Así, se asigna a cada uno un porcentaje de selección, haciendo que entre todos los porcentajes sumen 100. Tras esto, se distribuyen todos los porcentajes en un gráfico circular. Como si de una ruleta de juegos de azar se tratase, se obtienen tantos números aleatorios módulo 100 como individuos queramos en la nueva generación y las soluciones obtenidas serán todas aquellas seleccionadas por los números aleatorios en la ruleta. Evidentemente, las mejores soluciones ocupan un mayor espacio en la ruleta, haciendo que sea más probable que resulten seleccionadas en varias ocasiones.

En el método del torneo cada cromosoma se enfrentará a otro u otros cualesquiera de la población y el/los que ofrezcan una mejor solución serán seleccionados. Esta forma de selección ofrece varias alternativas, como variar la cantidad de cromosomas comparados y obtenidos. Por ejemplo, en una población de 9, podríamos elegir los 3 mejores de 4 cromosomas aleatorios y realizar este proceso 3 veces. En el presente trabajo se ha utilizado el método de selección por torneo ya que es el más apropiado para una implementación eficiente de AEs sobre la GPU. En concreto, cada cromosoma se compara con otro aleatorio produciendo un único vencedor. Otra modificación al funcionamiento normal de la selección es añadir una inestabilidad poblacional, como ocurriría en la naturaleza, añadiendo una pequeña probabilidad de supervivencia de cromosomas más débiles, ya que podrían tener potencial de cruce aun no teniendo potencial de ser una solución positiva. De esta manera imitamos el fenómeno que se da en la naturaleza cuando un individuo mal adaptado al entorno, pero poseedor de rasgos útiles, se reproduce, haciendo que esos rasgos se hereden.

Elitismo

La evolución en nuestro algoritmo podría hacer que se perdiesen los mejores individuos por cruce, mutación o selección y eso no resulta conveniente, pues retrasaría la aparición del óptimo global. Por este motivo se implementa la técnica del elitismo, en la que se conserva en las primeras posiciones a los individuos con mejores resultados en la función de adecuación (usualmente entre 1 y 4 cromosomas) y se les impide ser modificados de cualquier modo. De este modo en el cruce no tomarán carga genética de otros cromosomas, se evitará que muten y se asegurará su selección. Es fácil ver que cuando aparezca un individuo más adecuado para su supervivencia, debe viajar a la parte superior de su generación sustituyendo al cromosoma elitista que tenga un fitness menor.

Consideraciones en el diseño de algoritmos genéticos

La consideración más importante a tener en cuenta al diseñar un algoritmo genético es la selección de operaciones que codifican las operaciones de la solución. Llamaremos a esto set de funciones y para ello debe tenerse en cuenta un dato imprescindible: se debe cumplir la propiedad de la suficiencia, es decir, el conjunto de

operaciones que forma el set de funciones debe ser suficiente para poder completar la solución que trata de hallar el algoritmo. Parece un lema simple, pero para operaciones complejas, no es trivial demostrar o hallar a simple vista un set de funciones que tenga todo lo requerido para lograr una solución que alcance el óptimo global. Además de esto, el conjunto de operaciones no tiene por qué garantizar una solución única. Adicionalmente, el set de funciones no debe ser excesivamente grande, porque puede afectar severamente al rendimiento al causar que el espacio de búsqueda aumente de forma exponencial.

Otra importante propiedad que debe tener en cuenta un algoritmo genético es la completabilidad (llamada en inglés 'Closure'). Esta propiedad enuncia que los operadores del set de funciones deben ser capaces de manejar todos los posibles valores de entrada que pueda recibir el algoritmo. Las funciones resultantes de mutaciones o cruces también deben cumplir esta propiedad. Dicho de otra forma, para que un algoritmo cumpla la propiedad de la completabilidad, si recibe una entrada correcta no debe producir en ninguna de sus generaciones una salida inesperada por el manejo de funciones.

El siguiente punto en que se hará un inciso es la homogeneización de poblaciones. Si, fruto del azar o de una mala población inicial, a partir de determinado punto las generaciones se han vuelto muy homogéneas y alcanzan un fenómeno conocido en biología como cuello de botella genético [12] [13], las siguientes generaciones potencialmente seguirán siendo muy homogéneas, lo que conducirá a la necesidad de contar con un gran número de generaciones al no haber una convergencia apreciable. Por tanto, se producirá un gran consumo de recursos sin que tenga lugar una aproximación significativa a la solución. Si se detecta este fenómeno es conveniente intentar mitigar los efectos negativos con técnicas como la modificación del umbral de mutación a un nivel más alto, haciendo que se produzcan muchos cambios espontáneos en los individuos para volver a aumentar la diferencia entre soluciones. Otra opción, la tomada en la implementación presente, es la producción de cierta cantidad de individuos de refresco cada determinado número de generaciones, es decir, intercambiar algunos de los individuos actuales por nuevos individuos aleatorios, permitiendo de este modo que exista variabilidad en los resultados y que, por tanto, se produzcan cruces significativos en generaciones

sucesivas, lo que simulará, por ejemplo, la marcha o muerte espontánea de varios especímenes y la llegada de otros nuevos ajenos al ecosistema. Este método se conoce como operador de regeneración [14].

Por último, vamos a considerar la generación inicial de especies, que puede ser aleatoria o puede generarse mediante estadística para obtener soluciones muy distintas entre sí que produzcan más diversidad genética. Esto hará que los cromosomas contengan información muy diferente entre sí, haciendo aparecer una mezcla menos uniforme entre los hijos y evitando los ya nombrados cuellos de botella genéticos. Si bien este método tiene la gran ventaja de una gran potencia inicial para crear una cantidad elevada de soluciones poco parecidas entre sí, el consumo de recursos aumentará en la creación de la población inicial, además de la alta complejidad que entraña diseñar una heurística que produzca una población con una distribución suficientemente variable.

b. Gramatical Evolutivas (GE)

Descripción de las GE

Las GE (grammatical evolution) [15], son una forma relativamente nueva de tratar uno de los paradigmas de la computación genética: los algoritmos evolutivos (Aes). Su diferencia principal con éstos es la forma de codificar los resultados, ya que, en los AEs tradicionales, el cromosoma codifica una solución directa en forma de árbol. En la programación mediante evolución gramatical se añade un nuevo nivel de abstracción, haciendo que cada cromosoma codifique las producciones que se aplicarán en una gramática. Esto resulta en la adición de una fase más dentro de la traducción, a la que llamaremos transcripción, y que se enmarca entre la mutación y la traducción (figura 3).

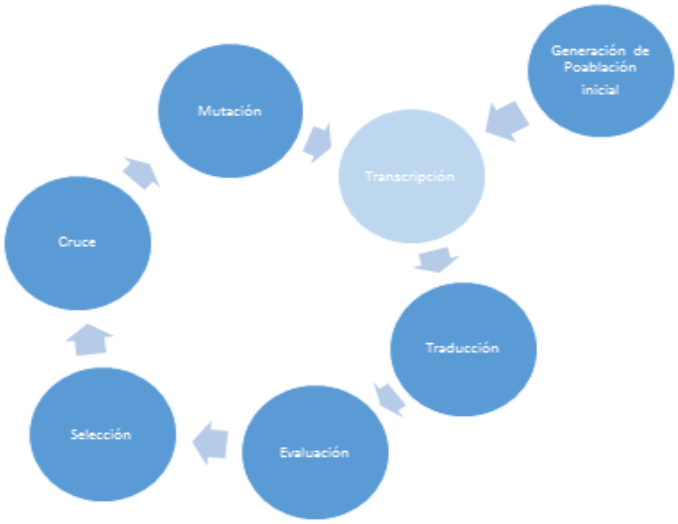


Figura 3. Esquema de una gramática evolutiva. En azul claro se observa el nuevo paso introducido por este tipo de programación.

La transcripción se caracteriza por la inclusión de una gramática como la Gramática 1, que guía la búsqueda de resultados. Una particularidad de estos programas es que varios genes distintos en un cromosoma pueden representar la misma solución. No ocurre a la inversa, ya que dos representaciones idénticas deben siempre producir el mismo resultado. La forma usada para representar una solución de este tipo utiliza los módulos de los números que contiene cada gen. Imaginemos, por ejemplo, que tenemos la Gramática 1, presente más adelante para un cromosoma como el siguiente.

Cromosoma → 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

El algoritmo comenzará evaluando mediante el módulo de 4, al ser ese el número de expresiones. En nuestro caso, la producción inicial (<s>::= <expr>) no consumirá un gen del cromosoma.

- $\mathbf{12\ mod\ 4\ =\ 0}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

Obtenemos así que la expresión 0 representaba **<expr> <op> <expr>**. En nuestro algoritmo, los operadores <op> y <unop>, son genes almacenados a continuación de la expresión. Por esto, la siguiente posición marcará el operador de esa producción.

- $\mathbf{28\ mod\ 4\ =\ 0}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

Con este dato sabemos que la producción actual está descifrada hasta este punto: **<expr> * <expr>**. El siguiente número en nuestro cromosoma indica la siguiente expresión. Por lo tanto, el módulo será 4, ya que hay 4 expresiones.

- $\mathbf{13\ mod\ 4\ =\ 1}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

Este resultado marca que hay una constante a continuación. Quedando la producción como **<const> * <exp>**, por lo que el siguiente punto a descifrar será la constante y el módulo será 3 al haber 3 de estas producciones.

- $\mathbf{126\ mod\ 3\ =\ 2}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

En este punto se define la constante quedando fijada la producción a **10.0 * <expr>**. La siguiente producción es una expresión, por lo que, se usará módulo 4.

- $\mathbf{231\ mod\ 3\ =\ 3}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

Nuestra producción ha alcanzado **10.0 * <unop> <expr>**. Como se ha comentado anteriormente, los operadores unarios y binarios van siempre a continuación, por lo tanto, el siguiente operador sería con módulo 3 al haber esa cantidad de <unop>.

- $\mathbf{1\ mod\ 3\ =\ 1}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

En este momento nos hallamos ante **10.0 * cos <expr>**, con lo que volvemos a utilizar el módulo de 4 al tener que descifrase una producción de tipo <expr>.

- $\mathbf{74\ mod\ 4\ =\ 2}$ — 12 | 28 | 13 | 126 | 231 | 1 | 74 | 66

Aplicamos de nuevo una sustitución en nuestra producción contando actualmente con **10.0 * cos <var>**. En este instante, el siguiente será nuestro último paso incluso si el cromosoma continuase, al estar abierta sólo una producción y ser esta un terminal. Finalmente aplicaremos módulo 1 al haber sólo una variable.

- $\boldsymbol{66\ mod\ 1\ =\ 0}$

12	28	13	126	231	1	74	66

Con esto, nuestra gramática habría utilizado su última producción y nos encontraríamos con la función.

- $\boldsymbol{f(x)\ =10.0\ \times\ cos(x)}$

12	28	13	126	231	1	74	66

Puede observarse que, si en lugar del 1 presente en nuestro cromosoma hubiera cualquier otro número congruente con 1 módulo 3 y los resultados representarían la misma solución, siendo cromosomas distintos. Sin embargo, si en lugar de ese número, fuese cualquier otro valor no congruente, aplicaríamos una producción distinta y todos los genes del cromosoma posteriores a este podrían indicar operaciones distintas a las aplicadas dependiendo del caso.

```
<s>::= <expr>

<expr> ::=
            <expr> <op> <expr>      (0)
            | <const>               (1)
            | <var>                 (2)
            | <unop> <expr>         (3)

<op> ::=
               *                    (0)
            |  +                    (1)
            |  -                    (2)
            |  /                    (3)

 <const> ::=
              0.1                   (0)
            | 1.0                   (1)
            | 10.0                  (2)

 <var> ::=
              x                     (0)

 <unop> ::=
              sen                   (0)
            | cos                   (1)
            | -                     (2)
```

Gramática 1. Ejemplo de una gramática para GE.

Desbordamiento de cromosoma

Un problema que existe en los algoritmos genéticos es la necesidad asegurar que una expresión sintácticamente correcta resulte de la evolución y, sin embargo, en la evolución gramatical esto no ocurre. Al componerse todas las expresiones mediante el uso de gramáticas, se producirá casi siempre una expresión sintácticamente correcta. La excepción a la corrección sintáctica ocurre cuando se alcanza el final del cromosoma sin haber hallado terminales suficientes para que se cierre la expresión.

Este tipo de desbordamiento puede suceder cuando el tamaño máximo del cromosoma es muy corto, cuando la expresión que se trata de hallar es muy larga o, en raras ocasiones fruto del azar. Para paliar este problema existen varias técnicas, como es el caso del “wrapping”, es decir, convertir el cromosoma en circular, de forma que cuando se termina de evaluar, si no se ha cerrado la expresión, se continúe iterando desde el principio del cromosoma. Esta técnica debe aplicarse de forma cuidadosa ya que, si accidentalmente las expresiones convirtiesen la cadena en un anillo, se correría el riesgo de ejecutar un bucle infinito y/o desbordar la pila de memoria destinada al programa, por lo que habría que limitar la cantidad de veces que una cadena continúa su traducción en el inicio. Otra manera que tenemos de paliar los efectos de un posible desbordamiento, la que se ha llevado a cabo en el código desarrollado, implica el marcaje de la cadena en la decodificación si no se ha alcanzado un cierre satisfactorio de la expresión al llegar el final del cromosoma. De esta forma, se indicará con un fitness elevado que el individuo no produce resultado.

c. Unidades de Procesamiento Gráfico. GPUs

Introducción a las GPUs

GPU o unidad de procesamiento gráfico es un dispositivo hardware presente en las tarjetas gráficas actuales. Está optimizado para una serie muy concreta de operaciones, para las cuales ofrece un rendimiento mejor que las CPU de propósito general. En principio fue planteado para la ejecución de aplicaciones gráficas y se utilizaba exclusivamente para la rasterización rápida de primitivas gráficas con simulaciones de espacios 3D interactivos, quitando carga de trabajo a la CPU.
En los últimos años y con la popularización de tarjetas gráficas en ordenadores de sobremesa en muchos ámbitos, entre ellos el científico, se ha tendido a trasladar determinados tipos de operaciones a las GPU por su mejor desempeño. En la actualidad las unidades de procesamiento gráfico están optimizadas para el cálculo con valores en coma flotante, ya que la representación de espacios 3D utiliza entornos matemáticos en R^3: $(R \times R \times R)$, lo cual conlleva muchas operaciones de este tipo. Simétricamente, este tipo de aplicaciones suelen traer consigo un alto grado de paralelismo, por lo que cada GPU está diseñada de forma que trabajen en paralelo múltiples (es habitual que se cuenten por centenares) unidades de procesamiento que permitan el cálculo simultáneo.

Arquitectura de las GPU

Mientras que en una CPU se suelen utilizar pocos núcleos de alto rendimiento, el paradigma de las GPU es totalmente distinto, como se muestra en la figura 4, y conlleva la inclusión de una elevada cantidad de unidades funcionales. Por esto, la gran diferencia entre CPU y GPU es que, mientras la primera usa una arquitectura Von Neumann [16], las GPU's utilizan un diseño que apuesta por la paralelización. Este tipo de arquitectura presenta varios multiprocesadores de tipo SIMD (Single Instruction Multiple Data), de modo que todos los núcleos ejecutan a la vez una misma instrucción en paralelo. Por ello, la decodificación de cada instrucción sólo se realiza una vez. Estos procesadores simples (CUDA cores) comparten planificador, por lo que son incapaces de ejecutar códigos distintos en paralelo, pero pueden sincronizarse a gran velocidad para mantener la consistencia. A pesar de esta

velocidad, la GPU se conecta al procesador del equipo anfitrión por un bus PCI-Express (figura 5), el cual, comparado con la memoria GPU es muy lento y no puede tener acceso directo a memoria principal. Por tanto, cada petición a ésta que se realice se lanzará a la CPU, lo que conlleva copias explícitas entre CPU y GPU. Para finalizar con la arquitectura, cada multiprocesador SIMD dispone de una memoria caché L1 y registros exclusivos para cada núcleo. Adicionalmente, habrá una memoria de tipo DRAM compartida entre todos los procesos.

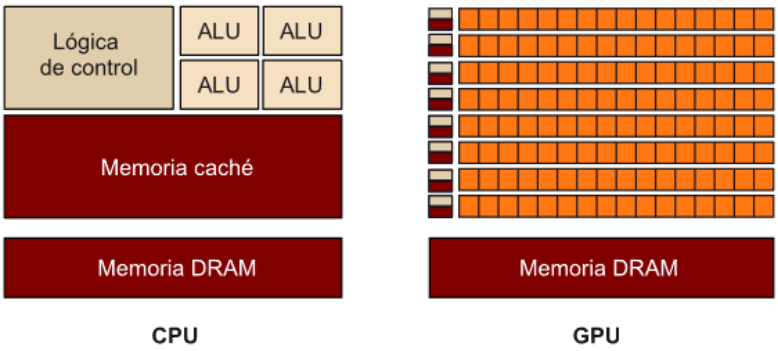


Figura 4. Diferencias entre GPU y CPU.

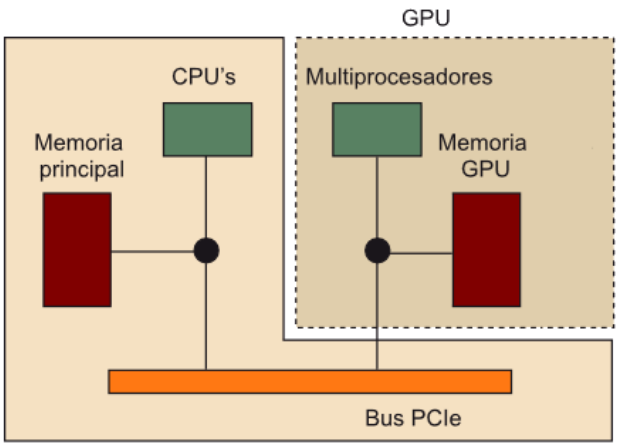


Figura 5. Conexión mediante PCI-E

La programación y 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 [17] CUDA incluye las especificaciones de la arquitectura y un modelo de programación asociado [18].

Además de la propia arquitectura, de la cual puede verse un esquema en la figura 6, CUDA también se refiere al nombre del compilador y del lenguaje. La gran ventaja que ofrece es el control simplificado de múltiples hilos, pudiendo tratar con cientos o incluso miles de threads en una sola instrucción. En realidad, no se ejecutarán todos los procesos de forma simultánea, aunque existen múltiples mecanismos que ocultan la latencia. Es el caso de accesos a memoria mediante ingeniosos métodos, como cambios de contexto de forma inmediata, conmutando de uno a otro de forma gratuita en tiempo y ocupando todos los procesadores. Esto evita en muchos casos esperas innecesarias. El lenguaje de programación CUDA es una leve modificación a C estándar, siendo todo el código CUDA que se ejecuta en el host (CPU) realmente código C [19]. El lenguaje de NVidia ofrece algunas herramientas en su semántica para notificar dónde se ejecutará cada fragmento de código, si en el Hardware CUDA o en el microprocesador. Concretamente, añadir __host__ antes de la declaración de una función, garantiza que es una función de CPU, y añadir __global__ indica que esa función realizará el lanzamiento de un kernel. Esa función se ejecutará en la GPU y sólo podrá ser lanzada desde una función __host__ y, por último, añadir __device__ asegura que esa función sólo puede ser ejecutada desde una función __global__ o __device__ y que se ejecutará en un multiprocesador CUDA de la tarjeta gráfica al igual que una función __global__. Es importante notificar que al lanzar un kernel, deberán indicarse dos atributos, la cantidad de bloques y los threads que tendrá cada bloque. Cada Hardware CUDA puede ejecutar un máximo de hilos diferente, especificados en la hoja técnica de cada dispositivo.

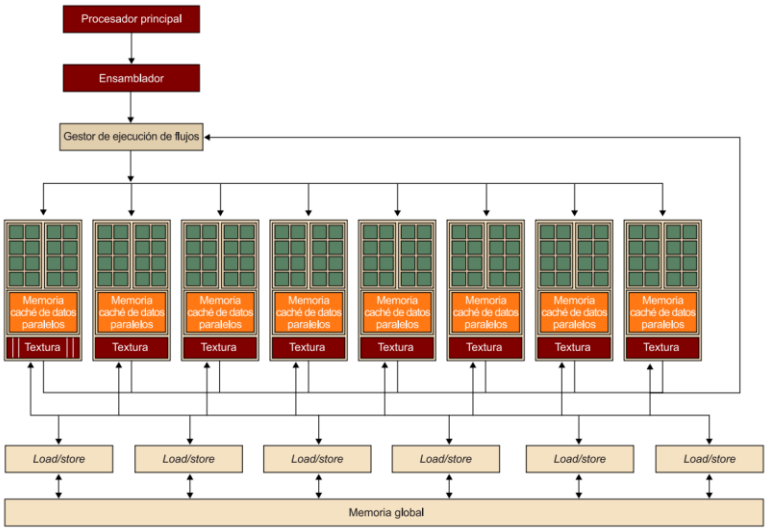


Figura 6. Arquitecturas Cuda-compatibles.

Otro detalle interesante de CUDA es que el tiempo máximo que un programa puede permanecer en ejecución en una GPU que utilice un sistema Windows y esté conectada a un monitor está alrededor de 5 segundos [20], dado que el administrador del adaptador gráfico primario limita ese tiempo. En esta implementación se realizan múltiples sincronizaciones con la CPU en cada generación, evitando este problema que podría producirse si se obrara de otro modo. También es importante indicar que la ejecución en GPU es asíncrona, es decir, una vez se lanza un kernel desde la CPU, el flujo vuelve automáticamente a la CPU, haciendo que se trabaje en paralelo entre los dos dispositivos. Si se quiere lanzar un nuevo kernel, el flujo se detendrá hasta que el dispositivo esté libre y, si se quisiera hacer una transferencia de datos entre la memoria de la tarjeta gráfica y la memoria del ordenador, el flujo también se detendría hasta que los datos estén disponibles. Aun así, en la API de CUDA existen métodos disponibles para sincronizar de forma explícita la CPU con el final de la ejecución de la GPU, o para copiar memoria de forma asíncrona. Asimismo, hay una diferencia importante en el lanzamiento de hilos con CUDA y con C: mientras en C el coste de lanzar y planificar un hilo es elevado, en CUDA consume sólo unos pocos ciclos [18].

3. Implementación de GE sobre GPUs

Implementación gramatical

Sobre el diseño de las gramáticas hay una serie de puntos interesantes que deben tenerse en cuenta, en especial la forma que se ha utilizado para representar una evolución gramatical. En la implementación a la que el presente documento se refiere, puede comprobarse que la gramática no sigue un patrón convencional, sino que utiliza una implementación recursiva propia en los lenguajes C y CUDA, según sea el código de la CPU o GPU, respectivamente. Si bien, una notación BNF sería más legible, perdería eficiencia de forma notoria.

La primera aproximación se realizó en Haskell, un lenguaje interpretado, utilizando librerías para la ejecución de este tipo de lenguaje. Sin embargo, la ineficiencia provocada por tener que interpretar la gramática de cada individuo en cada medición era evidente, por lo que se decidió trasladar la ejecución al lenguaje nativo de la aplicación: la versión CUDA de C. En este lenguaje existen muchos intérpretes de notación BNF que leen directamente de un archivo la gramática requerida, lo cual exige múltiples lecturas de fichero. Esto dañaría gravemente el tiempo de ejecución, y nos encontraríamos con un problema de lectura concurrente por miles de hilos, haciendo que en arquitecturas antiguas pudiera haber problemas con la cantidad de controladores de apertura de ficheros.

Finalmente, la aproximación que se llevó a cabo es la que se encuentra en el programa final, una gramática incrustada en el proyecto. Este tipo de implementación tiene dos inconvenientes fácilmente visibles. El primer problema que encontramos es la legibilidad. Evidentemente, una gramática en notación BNF es mucho más accesible para su lectura que un programa recursivo que utiliza numerosas constantes globales y punteros a memoria. El segundo punto débil de esta implementación está directamente heredado de la legibilidad, y es el problema de la modularidad. Si se desea cambiar la gramática, es mucho más complejo tener que modificar el código y se hace necesario recompilar el proyecto. Una solución propuesta para una futura actualización es la inclusión de varios archivos con

gramáticas precompiladas que ofrezcan la posibilidad de seleccionar una u otra en función de lo necesario para la resolución del problema al que se enfrente el algoritmo. Este problema también podría ser resuelto suministrando la gramática como un programa por separado que interactúe con el resto de la aplicación, de forma que, si se pretende modificar la gramática, sólo sea necesario recompilar ese archivo.

Teniendo dos problemas como los anteriormente citados en el diseño de esta gramática sería lógico cuestionar el motivo de utilizar este código. Sin embargo, este modelo tiene una característica principal que justifica su uso: el código puede ejecutarse en paralelo sobre una GPU. De este modo, cuando se debe evaluar un número alto de mediciones para un solo cromosoma, se lanzan tantos hilos como mediciones son necesarias y todas las traducciones y transcripciones se llevan a cabo de forma simultánea. Este es el único caso del algoritmo descrito en que la GPU lanza tantos hilos como mediciones y no tantos hilos como individuos.

Otro dato importante sobre la codificación de la gramática hace referencia a la traducción y la transcripción, pasos clave en la programación de gramáticas evolutivas. Ambos pasos se ejecutan juntos y de forma casi simultánea, es decir, al ser una representación recursiva, en cada paso de la recursión se desgranan parte de ambos problemas y al llegar a un caso base, se devuelve el valor final de la gramática para el terminal correspondiente. Por este motivo, al ser la misma cadena de llamadas recursivas con distintos datos, puede aprovecharse para lanzar tantos hilos en paralelo como datos tenemos, correspondiéndose con la constante $numMediciones$, y permitiendo a su vez, una ocupación del 100% de los multiprocesadores CUDA, al ejecutarse el mismo código para todos ellos.

Ordenación de la Población

Uno de los detalles más importantes y estudiados en la informática es la ordenación. En general, cuando se trabaja sobre el procesador, las mejores ordenaciones consumirían un tiempo $O(n \cdot log\,(n))$ [21], siendo en este caso n el número de individuos de la población. Debe tenerse en cuenta que el tiempo consumido será levemente mayor al citado, ya que en este caso estamos ordenando cromosomas por su fitness, de forma que habrá que conmutar los contenidos de cromosomas

atendiendo a la longitud de éstos. Sin embargo, el problema afrontado tiene varias particularidades, una de ellas relacionada de forma importante con la ordenación. Si bien es necesario mantener una cantidad de cromosomas elitistas ordenados, la cantidad de elitistas es realmente baja, así que no necesitaremos una ordenación *per se*, sino que tan sólo necesitaremos hallar los elitistas. Así, dependiendo de la arquitectura, nos convendrá hacer una cosa u otra.

Ordenación en CPU

Al principio del diseño se realizó una aproximación a la ordenación mediante el algoritmo quicksort, que además de trabajar con los fitness, trabajase también con los cromosomas, moviéndolos con respecto al valor del propio fitness. Sin embargo, como hemos mencionado, nuestro problema genético tiene particularidades dependiendo del dispositivo en el que lo estemos ejecutando. Concretamente, para la ejecución en procesador, es importante destacar que en principio no sólo no necesitamos mantener ordenados los datos de todos los individuos, sino que además será contraproducente al necesitar barajarlos más tarde. Por ello, bastará con tener localizados y ordenados sólo unos pocos individuos, los mejores, para que el elitismo se pueda llevar a cabo. Para este fin, siempre ubicaremos los mejores resultados en la parte superior de la matriz de la generación. En la implementación para CPU, este problema está resuelto mediante una iteración simple que busca los mejores individuos, convirtiéndolos en los individuos elitistas y ordenándolos en la parte superior de la matriz. Esto consumirá un tiempo cercano a $O(n + a * m)$, siendo a una constante que indica la cantidad de cromosomas elitistas, y m un valor que representa el tiempo de ordenación y que, a efectos prácticos, al no superarse nunca los 8 cromosomas elitistas, podríamos tratarlo como constante. De este modo, reducimos de forma importante el consumo de tiempo para este caso, siendo $O(n)$ en lugar de $O(n \cdot log(n))$.

Ordenación en GPU

Tras trasladar el problema del elitismo a la GPU, si anteriormente no resultaba necesario ordenar todos los individuos, ahora se generan otras particularidades. La primera es que recorrer la población buscando el mejor individuo tiene un coste $O(n)$, que en un lenguaje como CUDA tendría lugar mucho más despacio que en el procesador. Sin embargo, existen ordenaciones en tiempo $O(log(n))$, por lo que

resulta rentable utilizar un algoritmo de ordenación bitónica (figura 7) escrito para el problema.

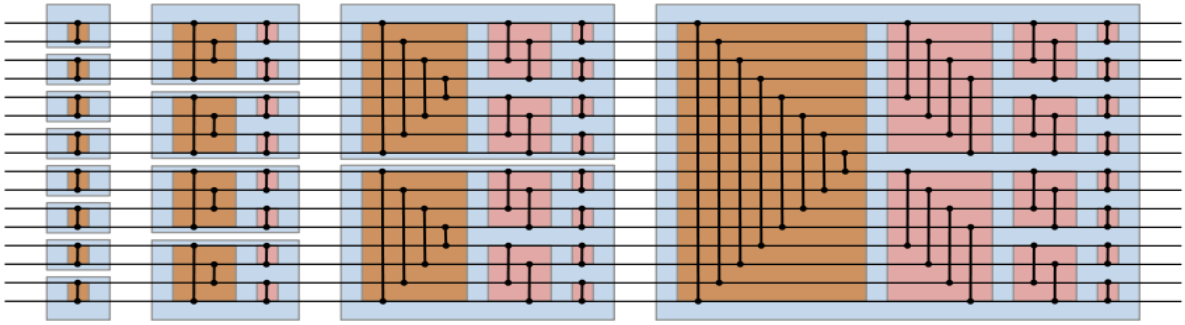


Figura 7. Esquema de ordenación bitónica [28]

El problema a la hora de paralelizar este tipo de soluciones es que hay tantos hilos como cromosomas y en cada momento se deben realizar la mitad de comparaciones. Es decir, debemos comparar al cromosoma X con el cromosoma Y, pero no debemos comparar de nuevo el cromosoma Y con el cromosoma X, ya que estaríamos trabajando de forma concurrente con los mismos datos. Por ello, se realizan la mitad de operaciones comprobando el índice de los hilos y reduciendo en gran medida la ocupación de los procesadores. Trabajando de esta forma, es decir, tomando $numCromosomas = 512$ y trabajando en contra del tiempo ($O(n)$) del algoritmo de CPU, si aplicamos estas técnicas lograremos que sólo se realicen $\sum_{i=2}^{(log_2\ numCromosomas)}(\sum_{j=i}^{(log_2\ numCromosomas)} 1) = 55$ comparaciones, resultando en un tiempo cercano al logarítmico para la ordenación, ya que se harán $55 * numCromosomas$ comparaciones. Obviando el resto de operaciones, al realizarse en paralelo con un número de hilos igual a $numCromosomas$, a efectos prácticos se traduce en 55 operaciones para esta cantidad de datos. Para evitar un intercambio masivo de cromosomas, lo que aun realizándose en paralelo consumiría mucho tiempo, la ordenación se hace mediante marcaje, es decir, se indica qué cromosomas se cambiarán y a qué posición se trasladarán. Tras la ordenación, se mueven en paralelo todos los cromosomas marcados una única vez a su posición de destino.

Paradójicamente, tras la ordenación, tenemos la obligación de desordenar la generación de individuos, ya que una población sin entropía producirá efectos negativos en el cruce y la selección. Por ello, después de realizar una ordenación, se barajan todos los cromosomas exceptuando los elitistas mediante el algoritmo de

barajado tipo faro o perfect shuffle, haciendo que la población vuelva a tener un buen nivel de entropía.

Selección

En la selección se ejecuta un hilo por cada individuo. Al ser un algoritmo genético elitista, hay cierta cantidad de individuos (por defecto 4) que, al ser los mejores, resultan seleccionados automáticamente y permanecen inmutables en todos los procesos evolutivos. Tras eso, para el resto de cromosomas, se toma en paralelo todo individuo i de la población para una generación g: $P_{i,g}$ y se selecciona un contrincante j aleatorio, siendo $i \neq j$. Se obtiene a continuación otro número aleatorio en el rango [1,100]. En caso de ser mayor que un parámetro inicial (por defecto 2), la forma de seleccionar será $p_{i,g+1} = min(p_{i,g}, p_{i,j})$. En este caso, se habrá seleccionado el mejor individuo. Si el valor aleatorio del comienzo fuese menor que 2, el azar habría dictaminado que el individuo mejor debe desaparecer. Esta forma de proceder simula muertes espontáneas en la naturaleza y otorga una baja cantidad de ruido, recomendado en algoritmos genéticos. El ganador suplanta al individuo actual en la próxima generación y se continúa con el proceso genético.

Cruce

La forma de realizar el cruce implementado es idéntica para CPU y GPU. Dados N (tamaño de la población) y x (número aleatorio en cada generación menor o igual que N/2), se crean y bloques, donde $y = 2 \times trunc(N \div 2x)$. El motivo de crear bloques pares es enfrentar los bloques de forma que el bloque 0 se encuentre con el 1, el 2 con el 3, etc. Como la cantidad de bloques no siempre cubre el total de individuos, si en la parte inferior de la población quedan individuos sin emparejamiento, se parte la sección en 2 y se emparejan con el individuo correspondiente. Un esquema sobre este procedimiento se puede ver en la figura 8.

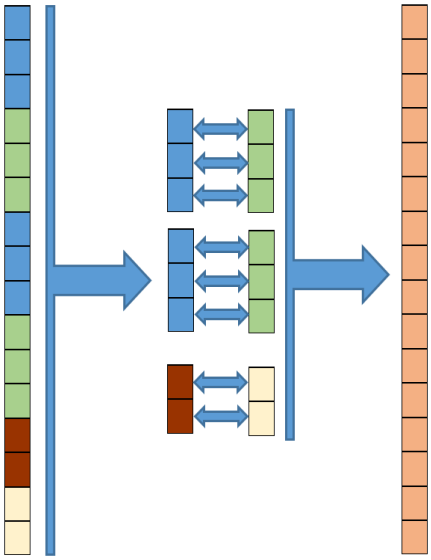


Figura 8. Esquema que resume el cruce implementado

Compilación

Si bien las implementaciones sobre GPU y CPU son casi idénticas, exceptuando las diferencias ya anotadas anteriormente, podríamos generar variaciones de la forma de ejecución o del código ensamblador que se ejecutará finalmente durante la compilación ya que, según algunas referencias [22], en numerosos lenguajes de programación la compilación y sus opciones pueden modificar el rendimiento de una aplicación. Por este motivo, se ha optado por utilizar código C para las implementaciones de CPU. Sin embargo, este código se compila bajo la extensión de CUDA (.cu y .cuh para reemplazar a .c y .h), las mismas extensiones que utilizan las implementaciones para GPU. A su vez, ambos han sido compilados utilizando nvcc, es decir, el compilador por defecto de CUDA para Windows. De esta forma se obtiene la máxima igualdad de condiciones posible. Las opciones utilizadas para la compilación han sido *para ambos problemas:*

- -dlink
- -o Release\<GPU_Genetico/CPU>.device-link.obj
- -Xcompiler "/EHsc /nologo /Zi "

Nota sobre los parámetros de ejecución

CUDA limita el máximo número de hilos lanzados por cada bloque lógico y ofrecería la posibilidad de lanzar varios bloques para superar el máximo de hilos. Esta cantidad no depende de la implementación del lenguaje, sino del hardware sobre el que se

ejecute. De este modo, un código que compila correctamente y se ejecuta de forma satisfactoria en una GPU, podría fallar en otra GPU si el número de hilos ejecutados excede la cantidad máxima máximo de esta unidad gráfica. La presente implementación ejecuta en cada kernel un único bloque, corriendo la totalidad de código de forma simultánea. El programa ejecutará tantos hilos como puntos haya en la regresión, en el caso de la traducción y transcripción, y tantos hilos como individuos en el resto de momentos. Al estar el lanzamiento de hilos en CUDA y el número máximo soportado de éstos supeditado a la arquitectura hardware, la cantidad máxima de individuos y la cantidad máxima de mediciones depende del hardware, siendo 1024 para ambos valores en la tarjeta en que se han realizado las pruebas (NVidia GTX 960). Un detalle a tener en cuenta es que, en la CPU, cuanto mayor sea la cantidad de individuos, mayor será el tiempo consumido. Sin embargo, en GPU esta cifra no aumentaría de forma tan significativa.

Sobre la implementación en GPU

Es importante notar que el esquema de construcción del algoritmo en GPU ha seguido un esquema similar al propuesto por el artículo Aceleration of Grammatical Evolution Using Graphics Processing Units [6], que incluye una pequeña etapa para control de la CPU entre algunos de sus pasos. Podemos ver en la figura 9 un diagrama indicando el funcionamiento en este dispositivo. El mencionado documento también propone la transferencia de la población completa a memoria principal tras cada generación. Sin embargo, se ha decidido realizar una transferencia mucho menor, que contenga únicamente el mejor individuo, para minimizar el uso del bus PCI-Express, que produciría un cuello de botella en la transferencia de datos GPU-CPU. Una opción que se baraja al inicio del mencionado documento es realizar únicamente la evaluación en la GPU. Sin embargo, nuestra transferencia mediante el bus es mínima, por lo tanto, deberíamos añadir múltiples operaciones de este tipo haciendo que el rendimiento cayera. Por este motivo, se eligió una implementación de la práctica totalidad del programa en la GPU. Otra diferencia con el artículo citado es que, mientras en su implementación se ejecutan en todo momento tantos bloques como cromosomas, aquí esa aproximación se ha tomado ejecutando hilos. Con este enfoque se salvarían los cambios de contexto de bloques, que tendrían un reducido coste, en beneficio de un cambio de contexto por hilos, que son conmutados

automáticamente. Por último, hay otra diferencia con el artículo y es el uso de una gramática monolítica incrustada en el proyecto.

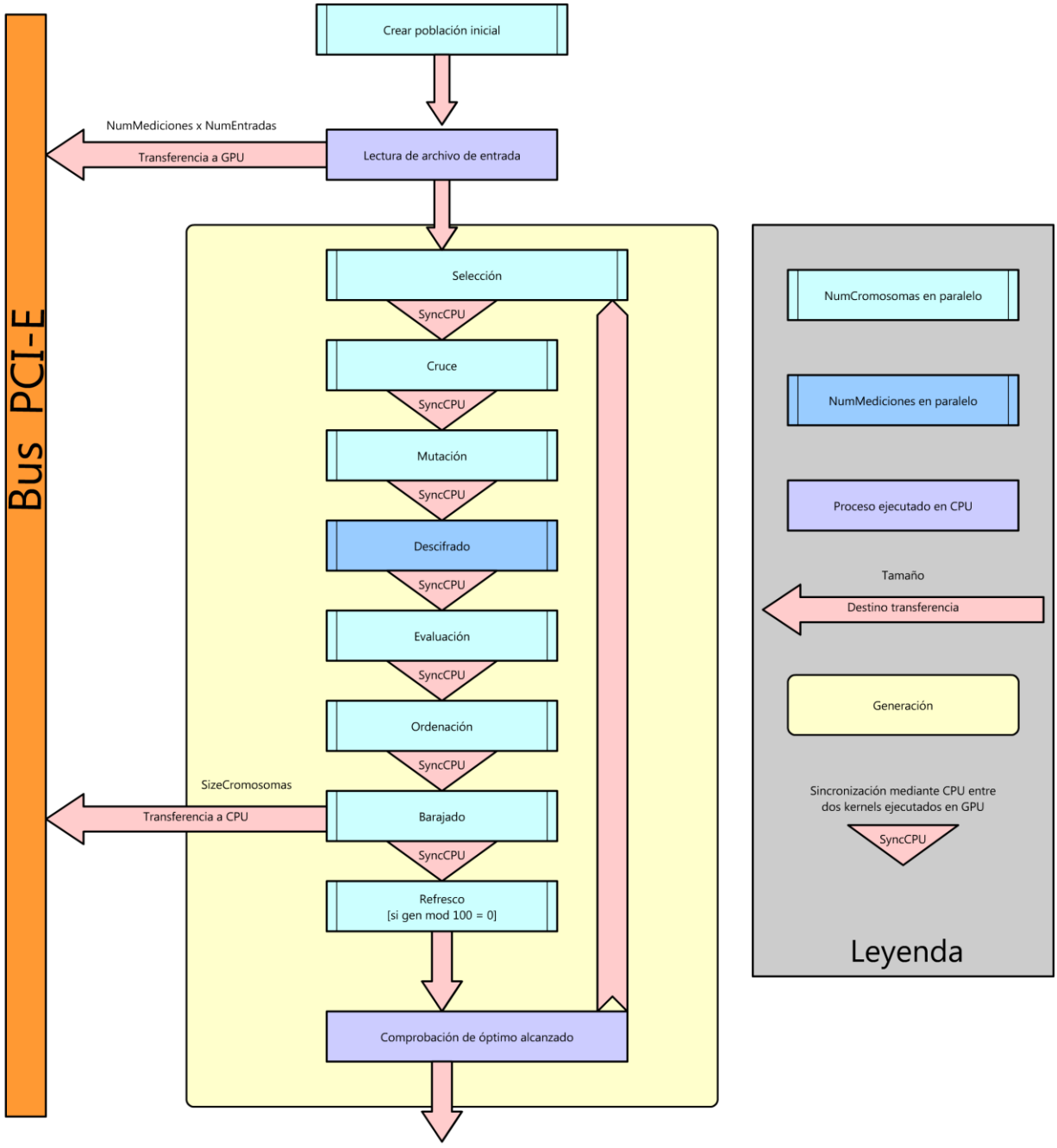


Figura 9. Esquema de funcionamiento del algoritmo diseñado sobre GPU

4. Resultados Experimentales

a. Experimental Set-up

La ejecución de las pruebas se ha llevado a cabo en un equipo que cuenta con el siguiente hardware y software:

- Procesador Intel Core i5-2500K
- GPU NVidia GTX 960, ensamblador ASUS, serie StriX
- Placa base Asrock P67 pro 3
- 8GB RAM DDR3
- Windows 7 Ultimate 64 bits
- Microsoft Visual Studio 2013
- Cuda Compilation Tools v7.5.17

i. Metodología, constantes y funciones de prueba

Metodología

Los resultados aquí presentados corresponden a la ejecución de las distintas implmentaciones de las gramáticas evolutivas sobre GPU y CPU, para 8 problemas distintos de regresión simbólica univariada. Se han realizado 10 ejecuciones de cada una de las instancias de los problemas con cada una de las configuraciones. Las tablas de datos resumen la información relevante para el análisis:

- Número de generaciones en que se obtiene un fitness óptimo
- Media aritmética del fitness obtenido en las 10 ejecuciones.
- Mejor fitness, peor, valor medio, desviación estándar.
- Tiempo medio de ejecución.
- Convergencia y adecuación al problema de las distintas gramáticas.

Parámetros

En las pruebas, la población contenía 512 individuos, los cuales contaban con 256 genes, siendo el máximo valor de cada gen 255. En total, se trató de encontrar una ecuación dados 288 puntos. La cantidad de generaciones que se ejecutó el programa en cada ejemplo dependía del caso enfrentado, utilizándose un número para el que la mayoría de las ejecuciones hubieran convergido a un fitness aceptable (es decir, un error absoluto medio menor al 10%), o se hubiera alcanzado una cifra muy alta de generaciones (estas cifras se indican más adelante). La probabilidad de que dos cromosomas emparejados para el cruce realicen el intercambio genético fue del 65%, y la probabilidad de mutación del 5% a nivel de población, pero alterando 7 genes de cada cromosoma que mutaba. Esta cifra se debe a que la longitud de un cromosoma es 255, por ello, para algunas soluciones cortas, como el caso de $\cos(2x)$, que puede contar con hasta 7 genes, una mutación tiene muy baja probabilidad de alcanzar un gen perteneciente a la solución. Para continuar, en la selección por torneo, un individuo con un fitness peor que su oponente tenía un 2% de probabilidades de supervivencia y, por último, el refresco de cromosomas se produjo con una probabilidad del 10% cada 100 generaciones.

Funciones y gramáticas de prueba

Se han probado 8 funciones usando 3 gramáticas. Que sean distintas tiene dos objetivos: el primero mostrar la importancia de la gramática en la delimitación del espacio de búsqueda, el segundo es hacer posible la comparación con los experimentos de otros artículos. Estas dos gramáticas han sido implementadas en CPU y GPU. Los ocho problemas a los que las distintas gramáticas se han enfrentado, ligados a la máxima cantidad de generaciones, son los siguientes.

1. $cos(2x)$……..1000 generaciones
2. $\int cos(x) + x + 1\ dx$ …….....15000 generaciones
3. $sin(x + 2)$……..…1000 generaciones
4. $x^2 + x - 3$……..…1000 generaciones
5. $x^3 + 2x^2 - x - 2$……7000 generaciones
6. $x^4 + x^3 + 2x^2 + x$……7000 generaciones
7. $x^2 + x$……..1000 generaciones
8. $x + 7$…….....1000 generaciones

Las gramáticas, por su parte, difieren levemente y, en la zona de resultados, puede apreciarse que no todas obtienen la misma cantidad de óptimos para todos los problemas. Si un problema converge al óptimo global antes de esas generaciones, el algoritmo se detendrá. En estas gramáticas se diferencia principalmente la generación de variables y los operadores unarios. Esperamos encontrar que gramáticas más complejas pueden resultar contraproducentes y a su vez que gramáticas más sencillas pueden no encontrar óptimo global si no disponen de una buena generación de constantes.

1. Gramática 1 (G1)

```
<program> ::=      <expr>

<expr> ::=         <expr> <op> <expr>
       |    <unop> <expr>
       |    <const>
       |    <var>

<op> ::=     +
       |    -
       |    *
       |    /

<unop> ::=         sin
       |    cos
       |    -

<const> ::=         1
       |    0.1
       |    10.0

<var> ::=          x
```

Gramática 2. Primera gramática utilizada en el problema. Su nomenclatura es G1.

2. Gramática 2 (G2)

```
<program> ::= <expr>

<expr> ::=     <expr> <op> <expr>
       |    <unop> <expr>
       |    <var>

<op> ::=    +
       |    -
       |    *

<unop> ::=    sin
       |    cos
       |    exp
       |    log

<var> ::=    x
```

Gramática 3. Segunda gramática utilizada en el problema. Su nomenclatura es G2.

3. Gramática 3 (G3)

```
<program> ::= <expr>

<expr> ::=     <expr> <op> <expr>
       |    <unop> <expr>
       |    <const>
       |    <var>

<op> ::=    +
       |    -
       |    *
       |    /

<unop> ::=    sin
       |    cos
       |    -

<const> ::= 1.0

<var> ::=    x
```

Gramática 4. Tercera gramática utilizada en el problema. Su nomenclatura es G3.

Podemos comprobar en la tabla 1 las diferencias entre las distintas gramáticas.

	Operaciones binarias a ○ b	**Operaciones unarias ○(a)**	**Constantes**
G1	+ - * /	sin, cos, -	1.0, 0.1, 10.0
G2	+ - *	sin, cos, exp, log	No
G3	+ - * /	sin, cos, -	1.0

Tabla 1. Relación de operaciones de las gramáticas.

ii. Medidas de error (fitness)

Todas las medidas del fitness se han tomado mediante el error medio. Es decir, siendo N el número de puntos que se comparan, f(x) la función que tenemos y e(x) la función esperada, la medida de fitness se ha tomado utilizando la función $fitness = \sum_{i=1}^{N}(|f(i) - e(i)|)$.

b. Resultados

Las tablas 2 a 15 muestran los resultados experimentales con los resultados medios obtenidos de 10 ejecuciones de cada problema en cada gramática y arquitectura. Las tablas contienen 8 columnas donde:

- Problem identifica la instancia acorde a la lista anterior
- Best: El mejor fitness obtenido en las 10 ejecuciones.
- Avg: El fitness medio obtenido en las 10 ejecuciones.
- Worst: El peor fitness obtenido en las 10 ejecuciones.
- AvgTime: El tiempo medio de 10 ejecuciones.
- #NumGen: La cantidad media de ejecuciones realizada.
- #NumOpt: La cantidad de veces que se halló el óptimo global.
- Desv: La desviación estándar sobre el fitness
- Gram: La gramática utilizada

GPU/G1							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desv**
P1	0	0	0	0,472	120,7	10	0
P2	82,03	378,97	1524,54	68	15000	0	513,5471
P3	0,0007	0,03	0,2	3,908	1000	0	0,062
P4	0,06	16,327	140,5	3,152	1000	0	43,6661
P5	0	36,3	890,19	36,3	6896	1	275,4861
P6	0	31,195	275,04	27,885	5840,5	3	85,8272
P7	0	5,57	55,7	1,201	338	8	17,61
P8	0,004	6,263	2,65	0,9844	1000	0	0,894

Tabla 2. Resultados de ejecución.

CPU/G1							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desviacion**
1	0	7,3E-06	0,000073	55,651	319,3	9	0,000023
2	76,33	126,105	271,19	2748,991	15000	0	55,3189
3	0,005	0,0654	0,2	164,259	1000	0	0,0554
4	0,64	161,279	3	161,279	1000	0	0,8854
5	0	72,88	369	1304,442	6443,2	2	128,8588
6	0	0,137	1,37	429,629	2350,4	9	0,4332
7	0	0	0	13,288	105,8	10	0
8	0	0,8744	3	108,9967	911,9	1	1,1703

Tabla 3. Resultados de ejecución.

GPU/G2							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desviacion**
1	0	0	0	0,418	40,4	10	0
2	211,02	2557,6	4791,16	226,156	15000	0	1687,1364
3	0,05	0,198	0,26	14,756	1000	0	0,0808
4	0,82	3,583	12,64	7,586	1000	0	3,2813
5	1,88	24,375	145,49	69,097	7000	0	44,3206
6	0	9,26	52,67	67,348	6390,6	4	16,55
7	0	0	0	0,425	35,7	10	0
8	0,03	3,159	6,18	3,159	1000	0	2,01

Tabla 4. Resultados de ejecución.

CPU/G2							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desviacion**
1	0	0,047	0,47	29,578	149,1	9	0,1486
2	91,77	3105,356	4416,64	2875,341	15000	0	1365,2586
3	0	0,18	0,26	225,781	998,5	1	0,0983
4	0,28	2,584	3	238,56	1000	0	0,9001
5	0,44	34,624	145,5	1643,342	7000	0	53,39
6	0	0,896	8,94	862,537	3670,7	7	2,8263
7	0	0	0	3,527	18,6	10	0
8	0,69	2,51	4,73	226,304	1000	0	1,2907

Tabla 5. Resultados de ejecución.

GPU/G3							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desviacion**
1	0,0005	0,093	0,27	4,232	1000	0	0,0842
2	91,76	146,663	355,66	83,624	15000	0	76,3056
3	0	0,036	0,09	2,09	616,7	6	0,0464
4	1,24	16,378	140,54	2,933	1000	0	43,6305
5	0	237,8	237,8	25,957	6556,3	2	86,8194
6	0	7,638	57,94	22,384	4886,2	5	18,1994
7	0	0	0	0,8228	280,9	10	0
8	0,02	2,518	5,32	3,516	1000	0	2,3086

Tabla 6. Resultados de ejecución.

CPU/G3							
Problem	**Best**	**Avg**	**Worst**	**AvgTime**	**#NumGen**	**#NumOpt**	**Desviacion**
1	0,01	0,171	0,34	182,867	1000	0	0,129
2	83,23	147,299	446,91	2567,519	15000	0	115,0694
3	0	0,027	0,09	70,598	446	7	0,0434
4	0	1,46	3,15	168,255	922,9	2	1,2473
5	0	289,79	1541,53	629,7225	4991,3	4	612,414
6	0	6,708	54,25	831,719	4163,6	7	17,1834
7	0	0	0	9,531	63,3	10	0
8	0	0,5	5	106,666	738,9	8	1,5763

Tabla 7. Resultados de ejecución.

A continuación, pueden observarse las tablas 8 a 15, que incluyen los datos de ejecución ya mostrados agrupados por problema.

Problem 1							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0	0	0	0,472	120,7	10	0
GPU2	0	0	0	0,418	40,4	10	0
GPU3	0,0005	0,093	0,27	4,232	1000	0	0,0842
CPU1	0	7,3E-06	0,000073	55,651	319,3	9	0,000023
CPU2	0	0,047	0,47	29,578	149,1	9	0,1486
CPU3	0,01	0,171	0,34	182,867	1000	0	0,129

Tabla 8. Resultados de ejecución agrupados por problema.

Problem 2							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	82,03	378,97	1524,54	68	15000	0	513,5471
GPU2	211,02	2557,6	4791,16	226,156	15000	0	1687,1364
GPU3	91,76	146,663	355,66	83,624	15000	0	76,3056
CPU1	76,33	126,105	271,19	2748,991	15000	0	55,3189
CPU2	91,77	3105,356	4416,64	2875,341	15000	0	1365,2586
CPU3	83,23	147,299	446,91	2567,519	15000	0	115,0694

Tabla 9. Resultados de ejecución agrupados por problema.

Problem 3							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0,0007	0,03	0,2	3,908	1000	0	0,062
GPU2	0,05	0,198	0,26	14,756	1000	0	0,0808
GPU3	0	0,036	0,09	2,09	616,7	6	0,0464
CPU1	0,005	0,0654	0,2	164,259	1000	0	0,0554
CPU2	0	0,18	0,26	225,781	998,5	1	0,0983
CPU3	0	0,027	0,09	70,598	446	7	0,0434

Tabla 10. Resultados de ejecución agrupados por problema.

Problem 4							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0,06	16,327	140,5	3,152	1000	0	43,6661
GPU2	0,82	3,583	12,64	7,586	1000	0	3,2813
GPU3	1,24	16,378	140,54	2,933	1000	0	43,6305
CPU1	0,64	161,279	3	161,279	1000	0	0,8854
CPU2	0,28	2,584	3	238,56	1000	0	0,9001
CPU3	0	1,46	3,15	168,255	922,9	2	1,2473

Tabla 11. Resultados de ejecución agrupados por problema.

Problem 5							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0	36,3	890,19	36,3	6896	1	275,4861
GPU2	1,88	24,375	145,49	69,097	7000	0	44,3206
GPU3	0	237,8	237,8	25,957	6556,3	2	86,8194
CPU1	0	72,88	369	1304,442	6443,2	2	128,8588
CPU2	0,44	34,624	145,5	1643,342	7000	0	53,39
CPU3	0	289,79	1541,53	629,7225	4991,3	4	612,414

Tabla 12. Resultados de ejecución agrupados por problema.

Problem 6							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0	31,195	275,04	27,885	5840,5	3	85,8272
GPU2	0	9,26	52,67	67,348	6390,6	4	16,55
GPU3	0	7,638	57,94	22,384	4886,2	5	18,1994
CPU1	0	0,137	1,37	429,629	2350,4	9	0,4332
CPU2	0	0,896	8,94	862,537	3670,7	7	2,8263
CPU3	0	6,708	54,25	831,719	4163,6	7	17,1834

Tabla 13. Resultados de ejecución agrupados por problema.

Problem 7							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0	5,57	55,7	1,201	338	8	17,61
GPU2	0	0	0	0,425	35,7	10	0
GPU3	0	0	0	0,8228	280,9	10	0
CPU1	0	0	0	13,288	105,8	10	0
CPU2	0	0	0	3,527	18,6	10	0
CPU3	0	0	0	9,531	63,3	10	0

Tabla 14. Resultados de ejecución agrupados por problema.

Problem 8							
Gram	**best**	**avg**	**worst**	**avgTime**	**#Gen**	**#Opt**	**Desv**
GPU1	0,004	6,263	2,65	0,9844	1000	0	0,894
GPU2	0,03	3,159	6,18	3,159	1000	0	2,01
GPU3	0,02	2,518	5,32	3,516	1000	0	2,3086
CPU1	0	0,8744	3	108,9967	911,9	1	1,1703
CPU2	0,69	2,51	4,73	226,304	1000	0	1,2907
CPU3	0	0,5	5	106,666	738,9	8	1,5763

Tabla 15. Resultados de ejecución agrupados por problema.

c. Discusión

Con los resultados de las 480 ejecuciones se obtuvieron los datos de las tablas 2 a 15. Sin embargo, esto no es muy útil a la hora de comprobar la validez de ejecuciones, por lo que a continuación haremos un estudio más detallado sobre los tiempos de ejecución, la convergencia a óptimos y adecuación al problema, y el tiempo medio por generación. Por último, se compararán los resultados de ejecución con los obtenidos en el ya mencionado artículo [15] sobre las GE. De esta manera, tendremos una visión mucho más global del alcance del algoritmo en sus dos implementaciones, CPU y GPU. Hay que hacer notar que el tiempo de ejecución ha limitado la cantidad de ejecuciones que se han realizado, dada la gran cantidad de tiempo y recursos que consumen. Por ejemplo, en el Problema 2 para CPU en la gramática 2, una sola ejecución alcanza de media 2875 segundos.

Tiempo total de ejecución

Estos procesos se han ejecutado en un ordenador que funcionaba de forma nativa sobre Windows 7. Su prioridad de ejecución ha sido la estándar y no se ha ejecutado ningún otro programa de forma simultánea. Para comenzar, conviene hacer un inciso en los tiempos de ejecución. A continuación, se podrá comprobar el resultado de las tres ejecuciones de las tres gramáticas, cada una resolviendo los 8 problemas mencionados en el punto 5.i. Metodología, constantes y funciones de prueba, en una comparativa entre GPU y CPU separados por gramáticas.

En la Figura 10 se muestran los resultados comparados de los tiempos de ejecución de los algoritmos correspondientes a la gramática G1 sobre las CPU y la GPU para los distintos problemas. Como se puede apreciar, el tiempo de ejecución es mucho más bajo en GPU (barras azules) que en CPU (barras naranjas), lo que significa que, efectivamente, el paralelismo resulta útil con la presente implementación.

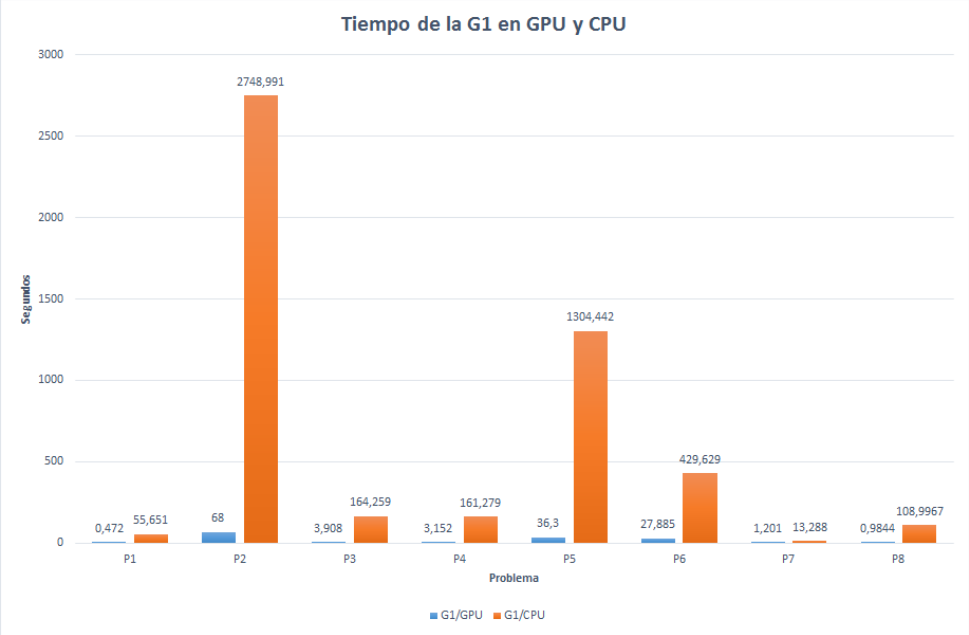


Figura 10. Comparativa de tiempos entre GPU y CPU bajo la G1.

Un detalle importante es que las ejecuciones que más generaciones tenían fueron las que sufrieron una diferencia más notoria. El problema 2, en concreto, es el que ejecutó mayor número de generaciones (15.000), al ser un problema muy complejo como es una integral. Además, dado que no se produjo ninguna convergencia a un óptimo local, el resultado fue que en todas las ejecuciones se recorrió el total de generaciones, haciéndola una de las ejecuciones más lentas. En el caso más destacado en CPU tenemos un tiempo de 2749, que se aproxima a los 46 minutos de ejecución, mientras que el mismo problema en la implementación de GPU tan solo ocupará el hardware durante 68 segundos alcanzando en este problema un speedup de 39,42. Si solo viésemos el gráfico, para problemas con una diferencia menos destacada como, por ejemplo, en el problema 7, donde se da el caso de una convergencia temprana alcanzando el óptimo global en todos los casos, tendríamos diferencias menores. En este caso sería de 1,2 segundos para GPU y 13,29 en CPU, concluyendo con un speedup de 11,06.

En la figura 11, un gráfico sobre el tiempo, volvemos a apreciar que se conserva la tendencia en las diferencias de tiempos de ejecución. Podemos observar un aumento de tiempo en algunos problemas, como el 2 o el 3, y una reducción en otros, como el 7 y el 4. El motivo se analizará más adelante.

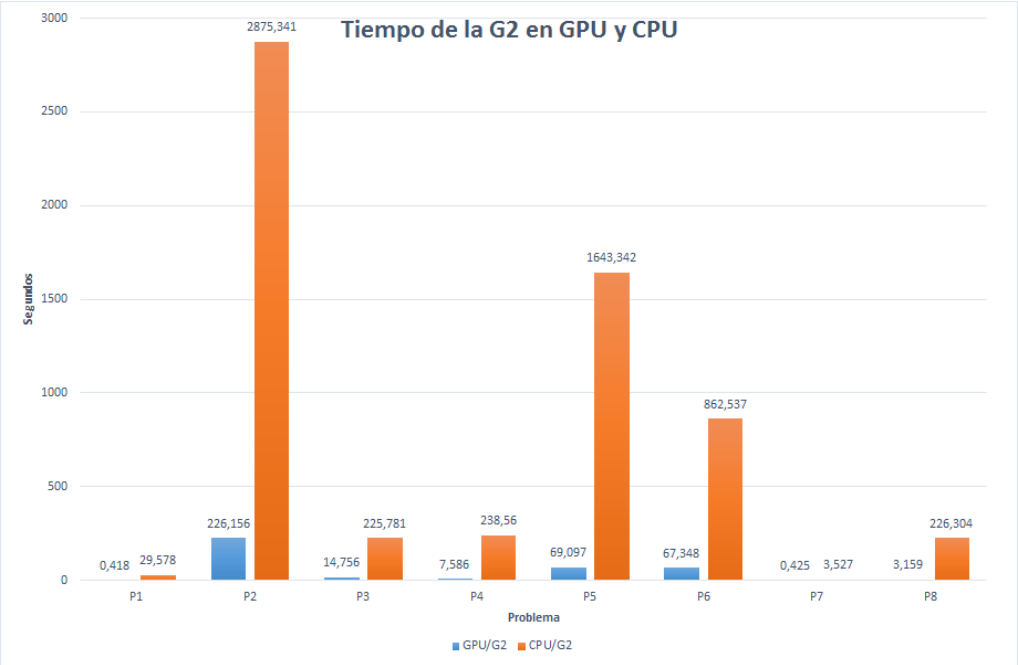


Figura 11. Comparativa de tiempos entre GPU y CPU bajo la G2.

En la figura 12 también podemos apreciar la diferencia destacada entre ejecución para CPU y GPU, siendo el color naranja de la CPU el predominante en la gráfica, como muestra de un tiempo de ejecución superior. Las figuras 9, 10 y 11 señalan que la implementación de gramáticas evolutivas en GPU resulta un éxito si el diseño se centra en explotar de forma masiva el paralelismo entre hilos, de forma que la ocupación de los núcleos CUDA ronde el 100% la mayor cantidad de tiempo posible.

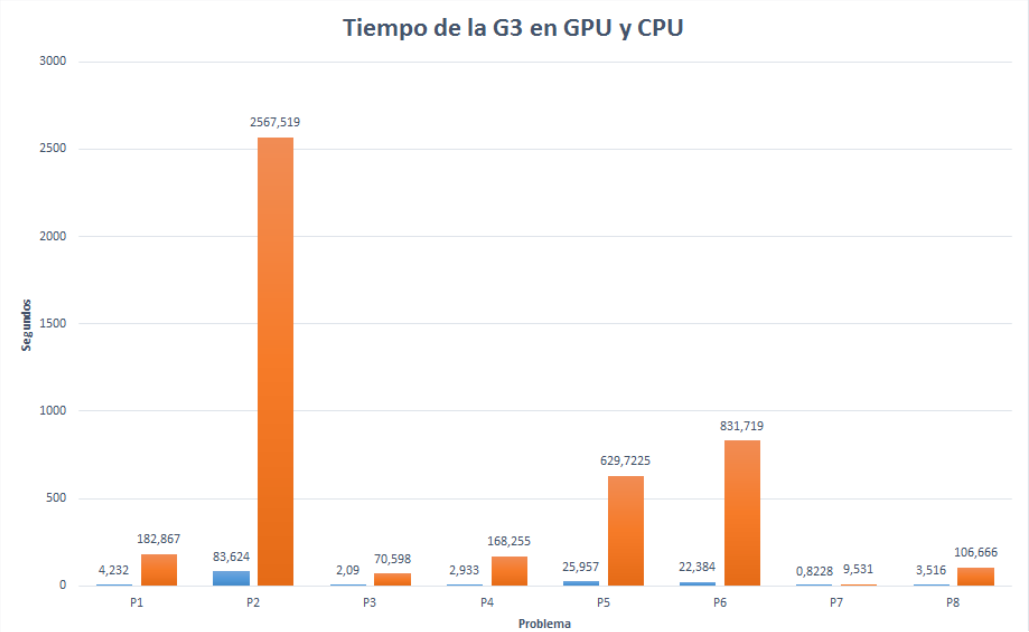


Figura 12. Comparativa de tiempos entre GPU y CPU bajo la G3.

Tiempo medio por generación

Las medias de tiempo de la tabla 16 muestran unas desviaciones estándar moderadamente pequeñas, con unos coeficientes de variación cercanos 26% para CPU y 6% para GPU. Por esto, podemos extrapolar unos tiempos hipotéticos de ejecución para varias cantidades de generaciones.

Tiempo por generación		
	Media (240 ejecuciones)	**Desviación**
CPU	0,182056808	0,03485742
GPU	0,059492194	0,00391476

Tabla 16. Tiempo medio de cada generación.

En la tabla 16 observábamos que, si el número de generaciones aumenta, el tiempo de una ejecución de CPU crece en mayor medida que la de GPU. Para apreciarlo de una forma más visual, se ha añadido la figura 13. Podemos comprobar que la estimación es bastante acertada: para una ejecución de 10.000 generaciones el tiempo consumido en CPU ronda los 2.000, mientras que, en nuestro programa, 15.000 ejecuciones tardan cerca de 2.500 segundos, una cifra que encajaría con

nuestro gráfico, teniendo en cuenta el coeficiente de variación. Hay que tener en cuenta que la cifra de tiempo por número de generaciones se ha obtenido con la media de todas las ejecuciones en todas las gramáticas y para todos los problamas.Además se ha realizado con un tamaño de población de 512. Es de esperar que, si la población fuese mayor y se usara una sola gramática con muchas operaciones de punto flotante, la diferencia aumentaría aún más.

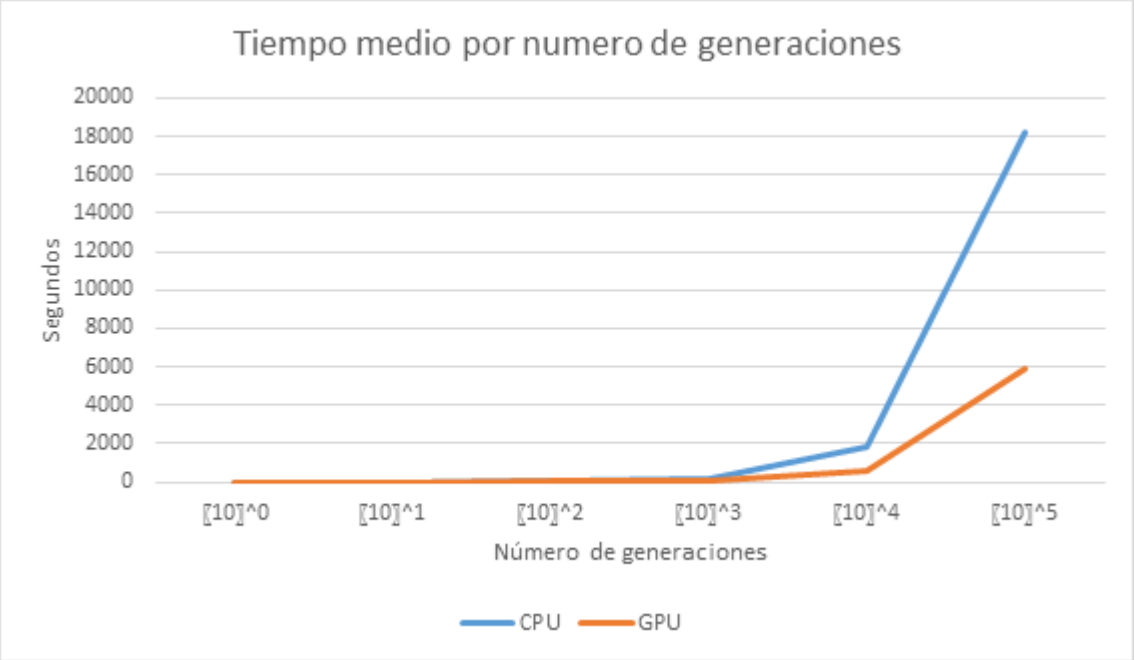


Figura 13. Tiempo medio por número de generaciones

Para concluir con el análisis del tiempo medio consumido por generación, podemos observar en la tabla 17 un análisis estadístico de la relación entre el tiempo de ejecución de cada problema en sus implementaciones de GPU y CPU (izquierda y derecha, respectivamente). Comprobamos que su coeficiente de correlación es de 0.6, por lo que habría correlación entre las dos variables. Podríamos asimismo observar la diferencia entre ellas, produciéndose una mejora del 30.25% de tiempo en las ejecuciones señaladas.

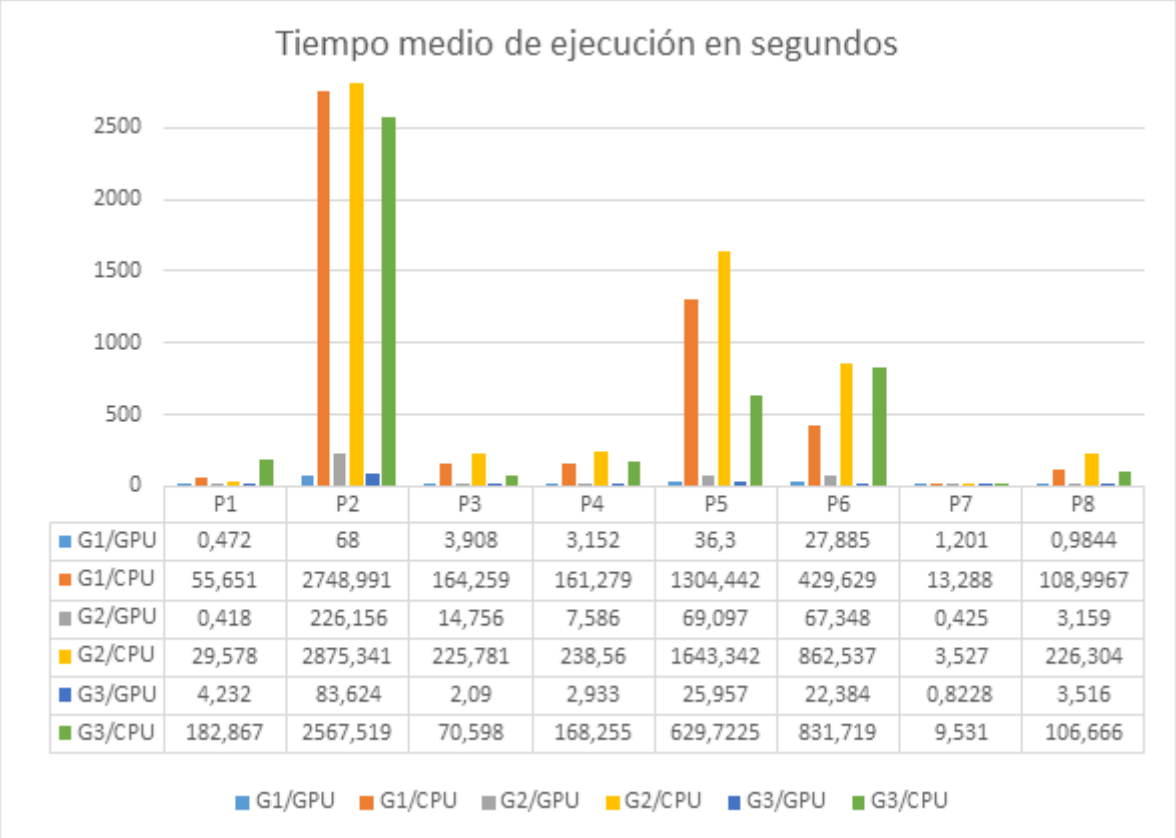
	Tiempo GPU	Tiempo CPU
Media	0,006018045	0,18205681
Varianza	1,53253E-05	0,00121504
Observaciones	24	24
Coeficiente de correlación de Pearson	0,601387131	
Diferencia hipotética de las medias	0	
Grados de libertad	23	
Estadístico t	-26,41113892	
P(T<=t) una cola	5,23052E-19	
Valor crítico de t (una cola)	1,713871528	
P(T<=t) dos colas	1,0461E-18	
Valor crítico de t (dos colas)	2,06865761	

Tabla 17. Comparación entre ejecuciones de GPU y CPU

Convergencia y adecuación de la gramática al problema

A continuación, presentamos un estudio sobre la convergencia de las gramáticas para cada uno de los problemas. En este caso, al ser prácticamente el mismo código fuente exceptuando las pequeñas variaciones de la gramática (que pueden observarse en el archivo Gramatica.cu del proyecto), podemos observar en la figura 14 que los tiempos de ejecución medios de todas las funciones comparadas comparten correlación entre GPU y CPU, como se explicará en la tabla 18. Es decir, cuando el algoritmo de GPU de una determinada gramática tarda más que sus homólogos en otras gramáticas, el

algoritmo de CPU de ésta sigue ese patrón, por lo que podemos convenir que la gramática utilizada es realmente relevante y la convergencia o no del algoritmo depende en gran parte de ella. De esta forma, si añadiéramos muchas operaciones superfluas, repetidas o equivalentes, aumentaríamos de forma innecesaria el tiempo de ejecución.

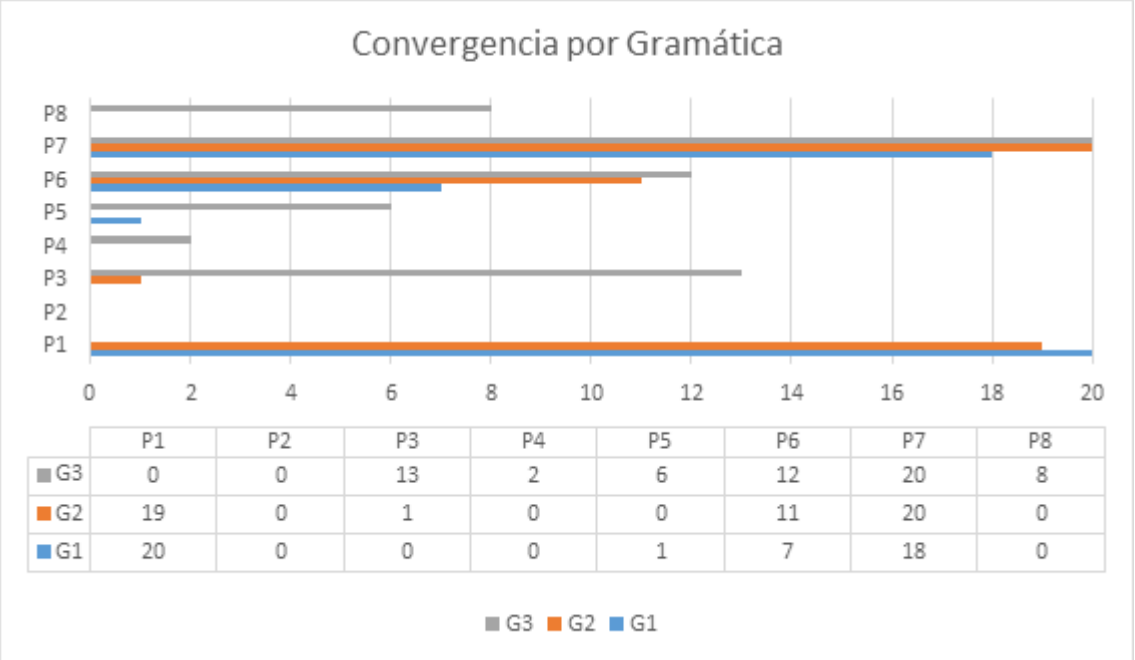


	P1	P2	P3	P4	P5	P6	P7	P8
G1/GPU	0,472	68	3,908	3,152	36,3	27,885	1,201	0,9844
G1/CPU	55,651	2748,991	164,259	161,279	1304,442	429,629	13,288	108,9967
G2/GPU	0,418	226,156	14,756	7,586	69,097	67,348	0,425	3,159
G2/CPU	29,578	2875,341	225,781	238,56	1643,342	862,537	3,527	226,304
G3/GPU	4,232	83,624	2,09	2,933	25,957	22,384	0,8228	3,516
G3/CPU	182,867	2567,519	70,598	168,255	629,7225	831,719	9,531	106,666

Figura 14. Comparativa de tiempos entre GPU y CPU en todas las gramáticas.

En la figura 15 se puede observar la cantidad de veces que converge cada gramática en la horizontal, para cada problema en la vertical. La cifra se obtiene haciendo una suma de la convergencia de esa misma gramática para ese problema en su modalidad de GPU y CPU. Podemos observar que, para problemas muy complejos, como por ejemplo P2, no se produjo convergencia en ningún caso. Sin embargo, para otros muy sencillos, como P1 y P7 se produjo una gran cantidad de convergencias. Finalmente, para otros problemas existe una mayor variedad. Veamos el ejemplo del problema P5. En este caso la gramática 1 no produjo ninguna convergencia a óptimo global, lo que indicaría que esta gramática no tiene la capacidad de hallar el óptimo global, o que le resulta muy complicado hacerlo. La gramática 2, por su parte, converge 1 vez, con lo que podemos afirmar que estamos ante un caso parecido al de la primera

gramática. Sin embargo, la tercera gramática produjo una mayor tasa de convergencia, alcanzando 6 veces la solución óptima, con lo que podríamos suponer que su adecuación para el problema es mayor que en los casos anteriores.



	P1	P2	P3	P4	P5	P6	P7	P8
G3	0	0	13	2	6	12	20	8
G2	19	0	1	0	0	11	20	0
G1	20	0	0	0	1	7	18	0

Figura 15. Convergencia total por gramática

En la tabla 18 se han comparado los valores de fitness de problemas en GPU y CPU, haciendo un promedio de éstos. Se puede observar que el valor medio del fitness en GPU es levemente superior (un 11,87% de mejora), con un coeficiente de correlación de Pearson de 0.99, lo que indica que los resultados están muy relacionados entre sí. Esto tiene sentido, dado que los datos comparados son las soluciones a los mismos problemas, por lo que sería sorprendente que en este caso los resultados obtenidos tuvieran variaciones. Aunque resulte favorable al estudio en GPU, no deja de ser extraño que las implementaciones similares de dos problemas idénticos reciban fitness distintos. Recordemos que la única variación entre los problemas consiste en que en GPU se realiza una ordenación y luego una ronda del algoritmo “perfect shuffle”, haciendo que los resultados se enfrenten a la evolución siempre con una disposición local en memoria similar y con una entropía bastante estable, mientras que en CPU sólo se reubican unos pocos cromosomas y el resto permanecen situados en memoria sin un orden determinado. Esto podría influir positivamente en la evolución de los individuos ya que, aunque pseudo-ordenado en memoria, la

mezcla de individuos con un buen y mal fitness sería muy uniforme, haciendo que el cruce o la selección pudieran resultar más provechosos para el problema.

	Fitness GPU	***Fitness CPU***
Media	145,16496	164,72887
Varianza	272213,55	397658,37
Observaciones	24	24
Coeficiente de correlación de Pearson	0,9928171	
Diferencia hipotética de las medias	0	
Grados de libertad	23	
Estadístico t	-0,7444003	
P(T<=t) una cola	0,2320864	
Valor crítico de t (una cola)	1,7138715	
P(T<=t) dos colas	0,4641727	
Valor crítico de t (dos colas)	2,0686576	

Tabla 18. Comparativa de fitness en las ejecuciones de CPU y GPU.

Una pequeña comparativa

Además de estas mediciones, se ha querido comparar nuestros resultados con los del mencionado artículo de O'Neill *et al.* [6]. En él, la gramática utilizada es más sencilla y por ello no podemos comparar el tiempo que tomaría obtener una convergencia al óptimo global, pues el resultado no sería justo. Por este motivo tomaremos el tiempo que toma ejecutar 100 generaciones (la cantidad empleada en el artículo). Otro punto a tener en cuenta es que los hardware sobre los que se ejecutó el programa no son idénticos y tomaremos como referencia de rendimiento la cifra passmark rating, obtenida de la media de performance de miles de usuarios mediante el software PassMark performance test, para corregir estas diferencias.

En la implementación para GPU de O'Neill *et al.*, la ejecución tardaría 0,5 segundos de media, mientras que nuestro algoritmo consumirá 0,3, contando con un speedup de 0,66. La tarjeta gráfica usada para las pruebas del artículo de O'neill (Nvidia GTX 480), y la de este estudio (Nvidia GTX 960), cuentan una con un rendimiento de 4324 y 5913 respectivamente [23]. Podemos observar que nuestra tarjeta de pruebas es un 36% más rápida, por lo que corregimos el speedup original, obteniendo 0,42.

Tomando la media de GEVA para el mismo artículo, su tiempo medio es de 175,2 segundos, sobre el que obtendríamos un speedup de 583. No se realizarán correcciones sobre esta cifra al ejecutarse en hardware diferentes.

Por último, al ejecutar O'Neill su algoritmo de CPU obtiene una medida de 9,3 segundos, siendo en nuestro caso 15,81. El hardware utilizado es un procesador Intel i7 sin especificar modelo. Extrapolando la fecha del artículo y del resto de componentes se ha estimado similar al Intel i7-980x [24]. Este modelo y el utilizado (i5-2500k) tienen unos rendimientos de 8961 y 6440 respectivamente [25], siendo nuestro hardware cerca de un 30% más lento. Acelerando nuestro algoritmo en esta medida obtenemos 11,1 segundos, una cifra aún por debajo del 9,3 indicado por O'Neill. El motivo de un tiempo superior en CPU es la complejidad de las operaciones de la gramática, incluyendo en nuestro caso senos y cosenos, operaciones en coma flotante que consumen mucho más tiempo de ejecución que una operación primitiva. En GPU, esta cifra no resultará destacada por ser más rápidos este tipo de cálculos.

5. Un caso de aplicación: modelos de predicción de glucosa en sangre

La Diabetes Mellitus es una enfermedad que en la actualidad afecta a millones de personas en el mundo [26]. En la mayoría de las ocasiones, los pacientes dependen de unas pocas mediciones de glucosa al día y un registro discreto de funciones corporales (que deberían ser continuas), para llevar a cabo el control de la enfermedad. Afortunadamente, en los últimos tiempos la tecnología está ofreciendo métodos cada vez mejores para el seguimiento en tiempo real de todo tipo de eventos. Entre estos métodos se encuentra la medición continua de glucosa, que reduce la mayoría de los molestos pinchazos de los glucómetros.

Mediante la instalación de un medidor de glucosa que funcione de forma ininterrumpida, podríamos evitar realizar 5 mediciones al día, teniendo a la vez un mejor control sobre los ciclos de glucemia. Es más, contando con mediciones tan fiables, precisas y numerosas, tenemos capacidad para hacer mucho más. Por ello, podríamos tratar de conseguir una ecuación que se corresponda con el nivel de glucosa que el cuerpo de una persona va experimentar a lo largo de un día dados parámetros concretos del momento actual. Es decir, un algoritmo que calcule funciones de la glucemia con una serie de parámetros de entrada que sirvan para entrenar el algoritmo.

Propuesta

Una función de este tipo, para algo tan complejo como la glucosa, también podría requerir previsiblemente el acceso a glucemias anteriores en el tiempo, y basar su cálculo en ello. Por este motivo, la gramática debería tener la posibilidad de aplicar recursión, un caso que no hemos contemplado hasta el momento. Una aproximación de gramática que queda propuesta en este apartado es la siguiente:

Como se puede observar en la gramática 5, el punto de inicio utiliza un cálculo anterior en el tiempo en glucosa, haciendo la gramática recursiva. El motivo por el que no se ha realizado esta implementación directamente en el estudio es la propia recursión, que forzaría la eliminación de determinada cantidad de paralelismo, ralentizando el algoritmo. El motivo de esto es que resulta imposible paralelizar datos si es necesario hallar parte de los mismos a la vez. Al ser la velocidad en la ejecución un objetivo clave del trabajo, esta implementación se deja diseñada para futuras actualizaciones.

```
<expr> ::=          <expr> <op> <expr>
      |             <unop> <expr>
      |             <const>
      |             <var>
      |             <gluc>
      |             <carb>

<gluc> ::=          glucosa(x - <constInt>)
<carb> ::=          carb(x)

<op> ::=            +
      |             -
      |             *
      |             /

<unop> ::=          sin
       |            cos
       |            exp
       |            log
       |            -

<const> ::=         <constInt> <constDec>
       |            <constInt>

<constInt> ::=      <constInt> <num>
       |            <num>

<constDec> ::=      . <constInt>
<num> :: =          1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
<var> ::=          x
```

Gramática 5. Propuesta de gramática con recursión aplicada a la glicemia.

Al utilizarse en el programa varias columnas de información pertenecientes a un archivo para formar el conjunto de mediciones que analizará nuestra evolución gramática, podemos obrar una modificación clave que podría evitar la recursión. Mediante la adición de un preprocesamiento, sería útil tratar los datos de forma que tuviésemos en la misma medición datos anteriores de glucemia, añadiendo por ejemplo 4 columnas con la media de glucosa obtenida en fracciones de 30 minutos anteriores. Con esto, la recursión se salvaría, haciendo que la ocupación en los multiprocesadores no se viese reducida.

6. Conclusiones

En este trabajo se ha realizado una propuesta novedosa para la implementación de gramáticas evolutivas sobre GPUs, con el objetivo de reducir el tiempo de ejecución de una manera eficiente. Las conclusiones obtenidas son:

- Es posible y útil el diseño y desarrollo de una GE aprovechando el multiparalelismo masivo de los SIMD y la potencia del lenguaje y entorno de CUDA.
- Con la presente implementación, cuanto mayor sea el número de generaciones en ejecución, más crecerá la diferencia de rendimiento entre GPU y CPU.
- La ordenación de todos los elementos puede ralentizar la ejecución, por lo que es importante tomar consciencia de la arquitectura y circunstancias del uso de cada función.
- El número de individuos de una población, así como la cantidad de mediciones, están limitados por el hardware. En el caso del hardware utilizado, el límite de ambos parámetros será 1024 elementos.
- La implementación de GPU obtiene mejores fitness debido a la ordenación y barajado mediante el algoritmo de barajado perfect shuffle.
- Para una evaluación más certera de la validez de las gramáticas, la cantidad de generaciones debería ser la misma para todos los problemas.

Conclusions

In this work a new proposal has been made in order to implement grammatical evolution on GPUs. The aim of this approach was to decrease the execution time on an effective way. The conclusions obtained are:

- It is possible and useful designing and developing a GE taking advantage of the massive multiparalelism of the SIMD and the power provided by the CUDA architecture.
- With the implementation presented here, a greater number of generations executed will increase the differences on performance between GPU and CPU.
- Sorting every element could result into slowing the program, so it's important to be aware of the architecture and circumstances of every function.
- The number of individuals in a population, likewise the number of solutions, are limited by hardware. In concrete, with the hardware used on testing, the limit of both parameters will be 1024 elements.
- The GPU implementation gets generally better fitness values on its executions because of the sorting and the application of perfect shuffle algorithm.
- In order to have a more accurate evaluation of the grammars, the number of generations should be the same for every problem.

7. Bibliografía

[1] Hidalgo, J. I., Fernández, R., Colmenar, J. M., Cioffi, F., Risco-Martín, J. L., & González-Doncel, G. (2016). Using evolutionary algorithms to determine the residual stress profile across welds of age-hardenable aluminum alloys. Applied Soft Computing, 40, 429-438.

[2] Cortés, J. C., Colmenar, J. M., Hidalgo, J. I., Sánchez-Sánchez, A., Santonja, F. J., & Villanueva, R. J. (2016). Modeling and predicting the Spanish Bachillerato academic results over the next few years using a random network model. Physica A: Statistical Mechanics and its Applications, 442, 36-49.

[3] Cuesta, D., Risco-Martín, J. L., Ayala, J. L., & Hidalgo, J. I. (2015). Thermal-aware floorplanner for 3D IC, including TSVs, liquid microchannels and thermal domains optimization. Applied Soft Computing, 34, 164-177.

[4] Hidalgo, J. I., Colmenar, J. M., Risco-Martin, J. L., Cuesta-Infante, A., Maqueda, E., Botella, M., & Rubio, J. A. (2014). Modeling glycemia in humans by means of Grammatical Evolution. Applied Soft Computing, 20, 40-53.

[5] Colmenar, J. M., Hidalgo, J. I., Lanchares, J., Garnica, O., Risco, J. L., Contreras, I., & Velasco, J. M. (2016, March). Compilable Phenotypes: Speeding-Up the Evaluation of Glucose Models in Grammatical Evolution. In European Conference on the Applications of Evolutionary Computation (pp. 118-133). Springer International Publishing.

[6] Pospichal, P., Murphy, E., O'Neill, M., Schwarz, J., & Jaros, J. (2011, July). Acceleration of grammatical evolution using graphics processing units: computational intelligence on consumer games and graphics hardware. In Proceedings of the 13th annual conference companion on Genetic and evolutionary computation (pp. 431-438). ACM.

[7] Ryan, C., & O'Neill, M. (1998). Grammatical evolution: A steady state approach. Late Breaking Papers, Genetic Programming, 1998, 180-185.

[8] Coloni, A., Dorigo, M., & Maniezzo, V. (1996). Ant system: optimization by a colony of cooperating agent. IEEE Trans on Systems, Man and Cybernetics-Part B: Cybernetics, 26(1), 29-41.

[9] Meisel, J. D., & Prado, L. K. (2010). Un algoritmo genético híbrido y un enfriamiento simulado para solucionar el problema de programación de pedidos job shop. Revista EIA, (13), 31-51.

[10] C. Darwin, On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life, John Murray, 1859.

[11] Noraini, M. R., & Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP.

[12] U. N. C. f. S. E. University of California Museum of Paleontology, «Understanding Evolution,» [En línea]. Available: http://evolution.berkeley.edu/evolibrary/article/0_0_0/bottlenecks_01. [Último acceso: 3 07 2016].

[13] M. M. T. C. R. Nei, «The Bottleneck Effect and Genetic Variability in Populations,» *International Journal of Organic Evolution,* vol. 29, nº 1-10, 1975.

[14] J. I. Hidalgo, «Evolutionary Algorithms for Solving the Partitioning and Placement Problems in Multi-FPGA Systems.,» de *Genetic and Evolutionary Computation Conference; Gecco 2000, Workshop Program.*, Las Vegas, 2000.

[15] M. O' Neill,. C. Ryan. Grammatical evolution: Evolutionary automatic programming in an Arbitrary language, Kluwer Academic Publishers, 2003.

[16] Godfrey, M. D., & Hendry, D. F. (1993). The computer as von Neumann planned it. IEEE Annals of the History of Computing, 15(1), 11-21.

[17] Nvidia, «Nvidia,» [En línea]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html. [Último acceso: 4 12 2015].

[18] F. R. Guim, «Arquitecturas basadas en computación gráfica,» 2014.

[19] M. F. Piccoli, Computación de alto desempeño en GPU, Editorial de la Universidad Nacional de La Plata (EDULP), 2011.

[20] NVidia, «Developer Nvidia,» [En línea]. Available: https://developer.nvidia.com/cuda-faq. [Último acceso: 31 8 2016].

[21] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, 1984.

[22] IBM, «IBM Knowledge center,» [En línea]. Available: http://www.ibm.com/support/knowledgecenter/es/ssw_i5_54/rzaha/tranperf.htm. [Último acceso: 14 7 2016].

[23] passMark software, «Pass Mark's Videocard Benchmark,» [En línea]. Available: http://www.videocardbenchmark.net/compare.php?cmp%5B%5D=3114&cmp%5B%5D=100. [Último acceso: 4 9 2016].

[24] intel .inc, «Intel ark,» [En línea]. Available: http://ark.intel.com/es-es/products/47932/Intel-Core-i7-980X-Processor-Extreme-Edition-12M-Cache-3_33-GHz-6_40-GTs-Intel-QPI. [Último acceso: 4 9 2016].

[25] passmark software, «Passmark software's CPU benchmark,» [En línea]. Available: http://www.cpubenchmark.net/compare.php?cmp%5B%5D=804&cmp%5B%5D=866. [Último acceso: 4 9 2016].

[26] Escolar-Pujolar A, Mayoral-Sánchez E, Corral-San Laureano F, Fernández-Fernández C I, Ruiz-Ramos M, «La diabetes mellitus en España: mortalidad, prevalencia, incidencia, costes económicos y desigualdades» *Gaceta Sanitaria,* nº 20, pp. 15-24, 2006.

[27] Free Software Foundation, Inc. IBM, «sourceware,» 2016. [En línea]. Available: https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEAD#l281. [Último acceso: 2016 8 25].

[28] Wikimedia foundation, «Wikipedia,» 1 6 2011. [En línea]. Available: https://es.wikipedia.org/wiki/Ordenaci%C3%B3n_bit%C3%B3nica#/media/File:BitonicSort.svg. [Último acceso: 13 11 2015].

AUTORIZACIÓN PARA LA DIFUSIÓN DEL TRABAJO FIN DE GRADO Y SU DEPÓSITO EN EL REPOSITORIO INSTITUCIONAL E-PRINTS COMPLUTENSE

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en .....Ingeniería.informática.....de la Facultad de .....Informática....., autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Grado (TF) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Periodo de embargo (opcional):

- ☐ 6 meses
- ☒ 12meses

TÍTULO del TFG: Implementación.de.Gramáticas.Evolutivas.sobre.GPUs

Curso académico: 2015 / 2016

Nombre del Alumno/s:
Ismael..Gonjal..Montero

Tutor/es del TFG y departamento al que pertenece:
José.Ignacio..Hidalgo Pérez
Arquitectura.de.Computadores.y.Automática

Firma del alumno/s

Firma del tutor/es