Universidad de Córdoba

Máster en Sistemas Inteligentes

Trabajo de Fin de Máster

Minería de Reglas de Asociación en GPU

Córdoba, Julio de 2013

Autor:
Alberto Cano Rojas
Director:
Dr. Sebastián Ventura Soto

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Computación Evolutiva

La computación evolutiva engloba 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. 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 como solución para un problema, surge la necesidad de establecer criterios de calidad y de selección y también la idea de combinar características de las 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.

Bajo el paraguas de los algoritmos evolutivos exiten diferentes paradigmas tales como los algoritmos genéticos, los algoritmos meméticos, la evolución diferencial o la programación genética, que se originaron con distintas motivaciones y enfoques. Sin embargo, actualmente los algoritmos tienden a combinar características de otros paradigmas y campos de estudio en la búsqueda de una hibridación que mejore la solución al problema.

1.2. Programación Genética

La programación genética (PG) es un paradigma de computación evolutiva, orientada a encontrar programas de ordenador que realicen una tarea definida por el usuario. Se trata de una especialización de los algoritmos genéticos donde cada in-

dividuo es un programa de ordenador. Por tanto, puede considerarse una técnica de aprendizaje automático usada para optimizar una población de programas de ordenador según una heurística definida en función de la capacidad del programa para realizar una determinada tarea computacional, definida por el usuario.

Los primeros resultados de la aplicación de la PG fueron reportados por Stephen F. Smith [21] y Nichael L. Cramer [5]. Sin embargo, John R. Koza [10] es considerado el padre de este paradigma, siendo quien lo aplicó a la resolución de varios problemas complejos.

La PG es un paradigma con grandes exigencias desde el punto de vista computacional. Por esta razón, en los años 90 sólo se aplicó a problemas relativamente sencillos. En la actualidad, las mejoras en el hardware y en la propia PG están permitiendo resolver problemas en áreas tales como la computación cuántica, diseño electrónico, programación de juegos, búsqueda y otros. Su flexibilidad en la representación de soluciones le permite abordar múltiples tipos de problemas de optimización y aprendizaje. La Figura 1.1 representa la estructura de un árbol de expresión. La evaluación de un árbol de expresión requiere del uso de una pila donde almacenar los resultados temporales de la evaluación de los nodos y la expresión se puede representar mediante notación postfija o notación polaca inversa (RPN). La notación polaca inversa no necesita usar paréntesis para indicar el orden de las operaciones mientras la aridad del operador sea fija.

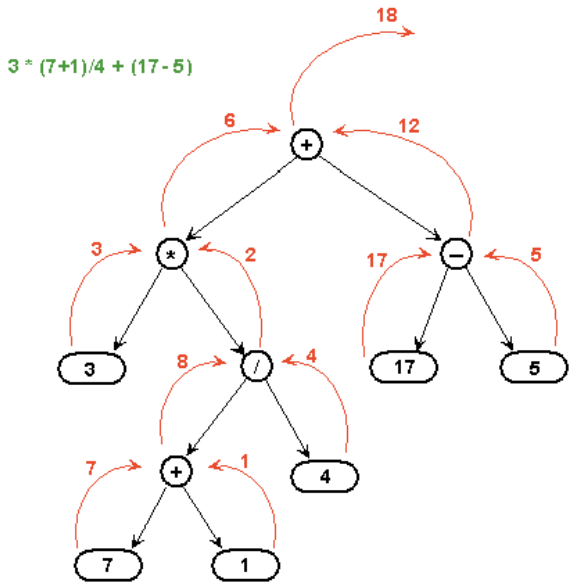

Figura 1.1: Árbol de expresión y su evaluación

1.3. Minería de datos

La minería de datos [4, 7, 8] 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 día a día se multiplica la cantidad de datos almacenados, 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 conocimiento de los datos en forma de modelos descriptivos (clustering y segmentación), modelos predictivos (regresión y clasificación), reglas de asociación, etc.

La minería de datos constituye la fase central del proceso de extracción de conocimiento (Knowledge Discovery), representado en la Figura 1.2. En este sentido la minería de datos es un punto de encuentro de diferentes disciplinas: la estadística, el aprendizaje automático, las técnicas de bases de datos y los sistemas para la toma de decisiones que, conjuntamente, permiten afrontar problemas actuales del tratamiento de la información.

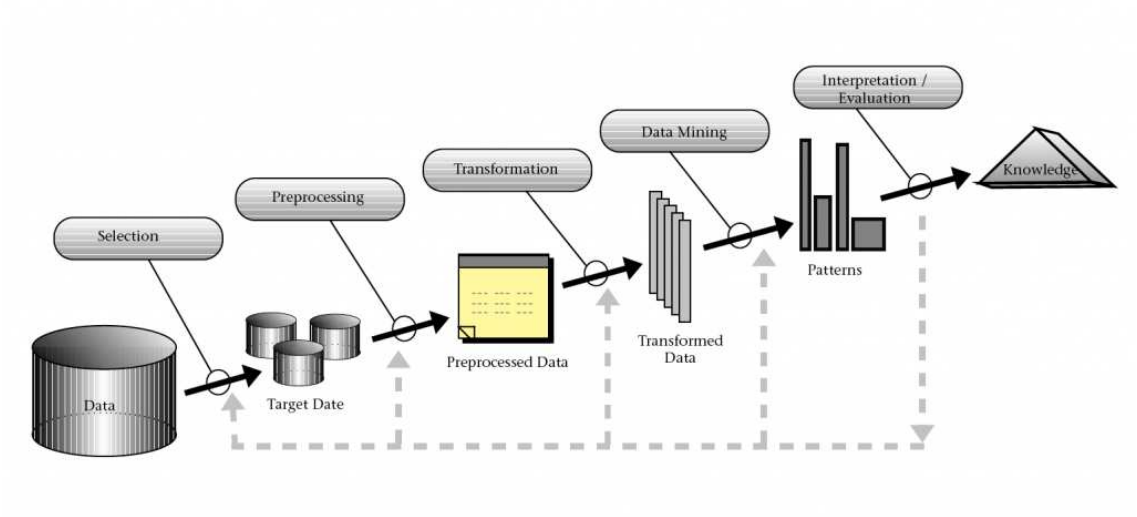

Figura 1.2: Proceso de extracción de conocimiento

1.4. Reglas de Asociación

La minería de reglas de asociación [1, 20] se marca como objetivo encontrar relaciones fuertes o de interés entre elementos o *items* en grandes bases de datos. Una regla de asociación se define como una implicación de la forma $A \rightarrow C$, donde A y C son conjuntos de *items* que no poseen atributos en común, es decir, $A \cap C = \emptyset$. En esta regla presentada, A es el antecedente y C el consecuente de la regla. El significado de una regla de asociación es que si todos los *items* de A están presentes en una transacción, entonces es bastante probable que todos los items de C estén también en dicha transacción.

El origen de las reglas de asociación se basa en la idea del análisis de las bolsas de la compra donde existen compras conjuntas de varios artículos. Este análisis ayuda a organizar y llevar a cabo ciertas estrategias de marketing, promoción de artículos, etc. Hoy en día, las reglas de asociación son aplicadas en muchísimos campos, como es el caso de la biología [6], medicina [17], etc, para descubrir relaciones entre los datos que aporten nuevo conocimiento previamente desconocido y útil para el usuario.

2. MOTIVACIÓN DEL TRABAJO

Este trabajo resume la labor de investigación del autor a lo largo del último año, apoyándose en los conocimientos adquiridos en las asignaturas del Máster en Sistemas Inteligentes, incluyendo algoritmos evolutivos, minería de datos, sistemas basados en reglas, minería de reglas de asociación, validación de resultados, etc.

Este trabajo aborda el problema de la escalabilidad en algoritmos evolutivos para minería de reglas de asociación, centrándose en sistemas basados en programación genética. Las tarjetas gráficas (GPUs) aportan un modelo de programación masivamente paralelo abierto a acelerar y solucionar múltiples problemas en aprendizaje automático. Por lo tanto, se propone un modelo de evaluación de reglas de asociación mediante GPU y se realiza un estudio experimental para analizar sus ventajas y medir su rendimiento.

El conocimiento adquirido a través del estudio de distintos problemas abiertos en minería de datos ha permitido al autor formarse en este campo sobre el que ya tenemos pendiente considerable trabajo futuro con el objetivo de continuar la tesis doctoral.

3. EL PROBLEMA DE LA ESCALABILIDAD

El tiempo requerido por los algoritmos de minería de reglas de asociación aumenta conforme crece el número de instancias y el número de atributos del conjunto de datos. El grado de complejidad computacional y la dependencia del número de instancias o del número atributos depende de cada algoritmo.

Los algoritmos de aprendizaje de reglas de asociación evalúan la calidad de las reglas aprendidas sometiéndolas al conjunto de datos y midiendo la calidad de su respuesta. Por lo tanto, es lógico que conforme mayor sea el número de datos, mayor sea el tiempo requerido en evaluar la calidad de las reglas. Por otro lado, el número de atributos también puede influir en el coste computacional del algoritmo, y dependerá de la filosofía de construcción de las reglas que tenga el algoritmo. Por ejemplo, los algoritmos de colonias de hormigas son muy sensibles al número de atributos ya que deben guardar un espacio de estados de la exploración de los diferentes atributos.

Centrándonos en algoritmos evolutivos de minería de reglas de asociación, y más concretamente de programación genética, nos encontramos con que cada individuo debe ser evaluado frente a todo el conjunto de datos. Los individuos del algoritmo de programación genética para asociación son expresiones compuestas por nodos que conforman reglas para indicar relaciones entre atributos de un conjunto de datos. El uso de programación genética para tareas de asociación ha demostrado ser una técnica que obtiene buenas soluciones [11, 12, 13, 14, 16, 18, 19]. Un base de reglas de asociación puede expresarse como un conjunto de reglas de tipo *Si–Entonces*, en las que el antecedente y el consecuente de cada regla está formado por una serie de condiciones que debe cumplir una instancia para que se cumpla la regla. La gran diferencia con las reglas de clasificación es que el consecuente, en lugar de ser una etiqueta clase, es a su vez, otro conjunto de relaciones entre atributos.

La fase de evaluación de los algoritmos evolutivos consiste en tomar cada uno de los individuos de la población y evaluar su aptitud en la resolución del problema. Por lo tanto, el coste computacional debido al número de instancias hay que multiplicarlo por el número de individuos de la población, lo que hace que el aumento de cualquiera de los dos parámetros afecte en gran medida el tiempo de ejecución. Para evaluar

la población, se interpreta el genotipo del individuo para obtener el fenotipo (regla o conjunto de reglas), que se evalúan sobre cada instancia del conjunto de datos, obteniendo el número de veces que se cumple el antecedente y el consecuente sobre dicho conjunto de instancias. Con estos datos se construye la matriz de contingencia y empleando distintas métricas para cada algoritmo, se obtiene la aptitud (*fitness*) del individuo. La Tabla 3.1 muestra la construcción de la matriz de contingencia en función de los casos en los que se cumple el antecedente (A) o el consecuente (C) sobre el número de instancias (N). Por lo tanto podemos dividir la fase de evaluación en dos pasos: evaluación de las instancias y cálculo del *fitness* usando los datos de la matriz de contingencia.

Tabla 3.1: Matriz de contingencia

	A	$\neg A$	
C	$n(AC)$	$n(\neg AC)$	$n(C)$
$\neg C$	$n(A\neg C)$	$n(\neg A\neg C)$	$n(\neg C)$
	$n(A)$	$n(\neg A)$	N

Tradicionalmente el proceso de evaluación ha sido implementado de forma secuencial, tal y como se representa en el Algoritmo 1, por lo que su tiempo de ejecución aumenta conforme el número de instancias o el número de individuos se incrementa.

A continuación analizaremos las opciones de paralelización y optimización de la función de evaluación con el objetivo de reducir su coste computacional. El primer paso de la evaluación es por definición paralelo, la evaluación de cada regla sobre cada instancia es un proceso completamente independiente, por lo que éstas se pueden paralelizar en hilos independientes sin ninguna restricción. Su función es interpretar las expresiones del antecedente y del consecuente. El resultado de esta comparación se almacena para cada instancia y cada regla en modo de cubre o no cubre. Este modelo de evaluación se representa en la Figura 3.1. El número de hilos es igual al número de evaluaciones en un determinado momento, es decir, el producto del número de individuos por el número de instancias, por lo que el número total de hilos puede ascender desde miles hasta millones de ellos.

El segundo paso de la evaluación es una operación de reducción [9] y exige el recuento del número de veces que se cubre el antecedente y consecuente de cada regla y para todas las reglas. El recuento es un proceso nativamente secuencial, pero

Algorithm 1 Evaluador de reglas secuencial

Require: population_size, number_instances

```
1: for each individual within the population do
2:     n(AC) ← 0, n(A¬C) ← 0, n(¬AC) ← 0, n(¬A¬C) ← 0
3:     for each instance from the dataset do
4:         if individual's antecedent covers actual instance then
5:             if individual's consequent covers actual instance then
6:                 n(AC)++
7:             else
8:                 n(A¬C)++
9:             end if
10:        else
11:            if individual's consequent covers actual instance then
12:                n(¬AC)++
13:            else
14:                n(¬A¬C)++
15:            end if
16:        end if
17:    end for
18:    individual fitness ← ContingencyTable(n(AC), n(A¬C), n(¬AC), n(¬A¬C));
19: end for
```

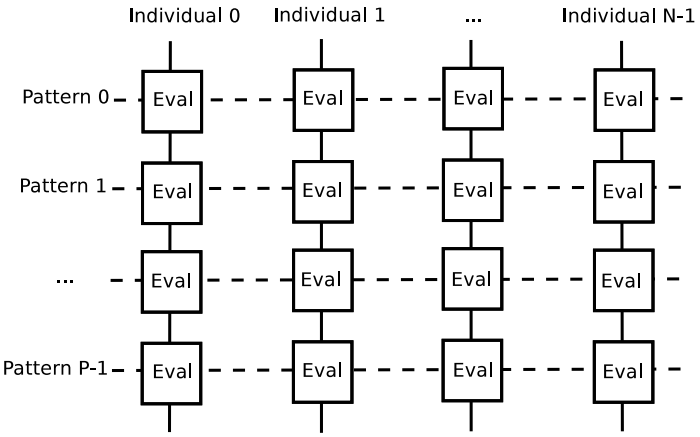

Figura 3.1: Modelo de evaluación paralelo

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 cubrimientos para cada instancia y regla. El segundo paso debe contarlos de manera eficiente y para ello, cada regla emplea múltiples hilos que realizan el recuento de los resultados,

obteniendo semisumas parciales. Posteriormente, se realiza la suma total de dichas 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 todo caso, la evaluación de las condiciones del antecedente y del consecuente pueden ser a su vez interpretadas en paralelo, por lo que el grado de paralelización que permite este problema es máximo.

En la próxima sección se presenta el modelo de programación CUDA, mientras que el desarrollo del modelo propuesto de evaluador en GPU se muestra en la sección 5.

4. 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 en el rendimiento del sistema.

4.1. 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 4.1 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.

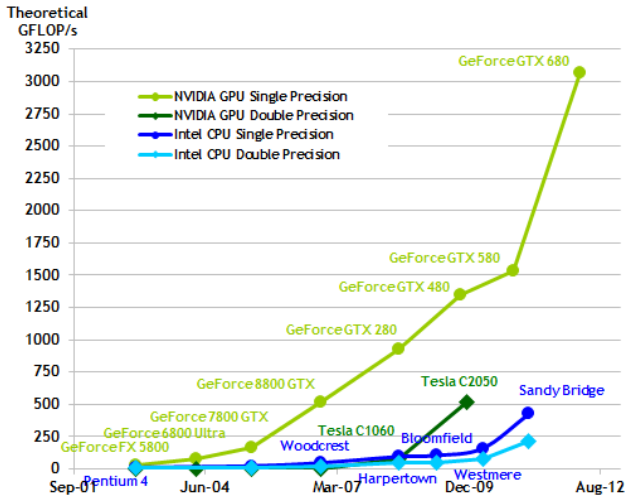

Figura 4.1: Evolución de las FLOPs de las CPUs y las GPUs

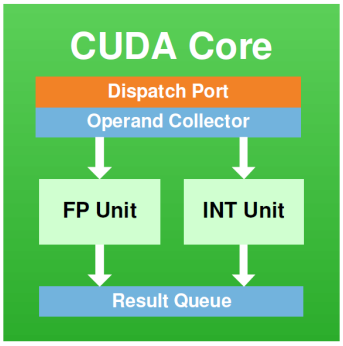

Figura 4.2: Shader o núcleo

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 4.2, se agrupan en una unidad conocida como multiprocesador (Streaming Multiprocessor, SM), ilustrado en la Figura 4.3, 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 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.

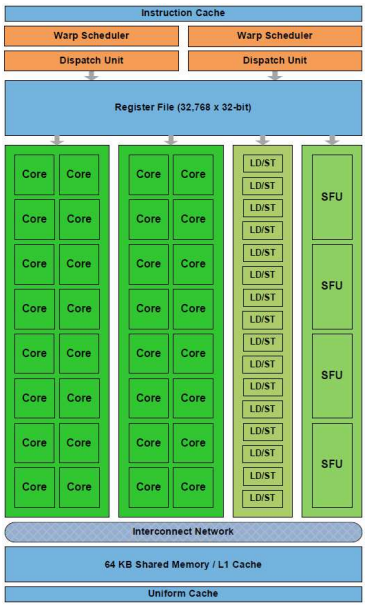

Figura 4.3: Streaming Multiprocessor

4.2. 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 }
```

4.3. 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 1024 hilos, es decir $Dx * Dy * Dz \leq 1024$.

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 4.4.

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*.

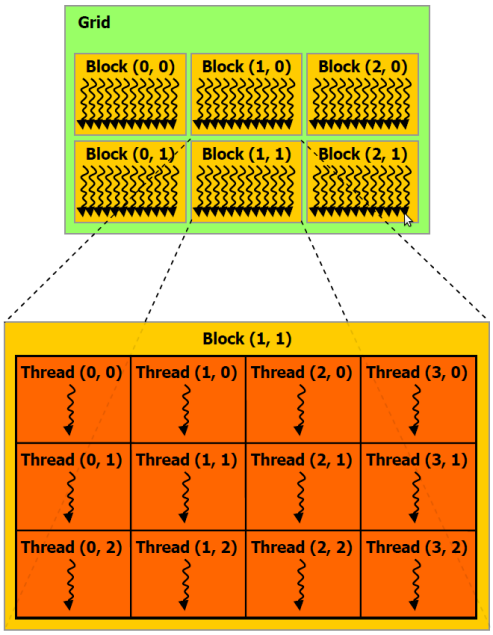

Figura 4.4: Grid de bloques de hilos

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 4.5 .

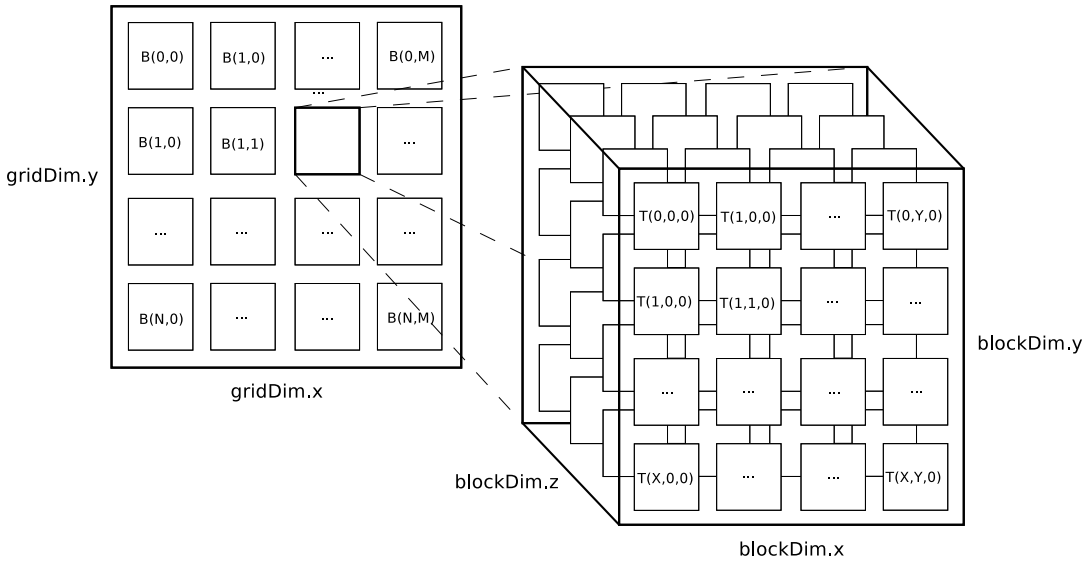

Figura 4.5: Jerarquía de hilos y bloques

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.

4.4. Jerarquía de Memoria

La memoria de la GPU necesita ser rápida y suficientemente amplia para procesar millones de polígonos y texturas. En la Figura 4.6 se representa la evolución del ancho de banda de la memoria principal en las CPUs y GPUs a lo largo de los últimos años.

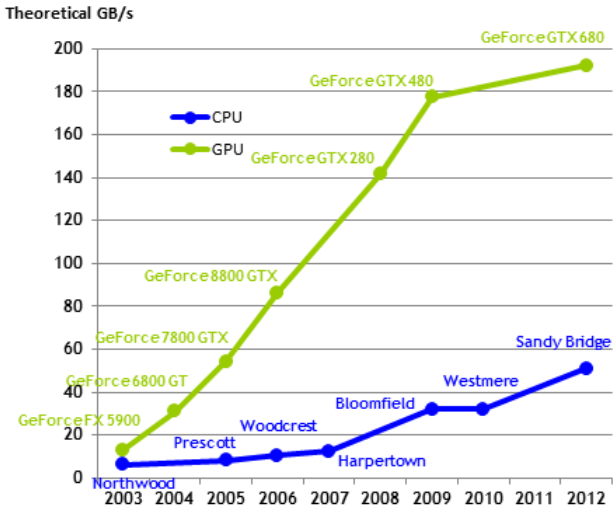

Figura 4.6: Evolución del ancho de banda de la memoria de las CPUs y las GPUs

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 variable 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 con una GPU con de 32768 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 5 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 o 48 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 coalescientes 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. 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.

5. EVALUACIÓN DE REGLAS EN GPU

5.1. Modelo propuesto

En la sección 3 se detalló la función de evaluación 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 del cubrimiento del antecedente y consecuente para cada instancia y regla. El segundo paso debe contarlos de manera eficiente y para ello cada regla emplea un número determinado de hilos en donde cada uno de los hilos realiza el conteo de una parte de los resultados. Posteriormente, se realiza la suma total de las semisumas parciales.

Las decisiones de diseño del evaluador en GPU han seguido las recomendaciones de la guía de programación de NVIDIA CUDA y del manual de buenas prácticas [23]. 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 128, 256 o 512 hilos, que es el tamaño habitualmente usado. Empleando los resultados de nuestra experimentación, hemos concluido que el número óptimo de hilos para nuestro problema es de 256, 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 una instancia es inmediata. Cada hilo representa la evaluación de un individuo sobre una instancia. Los hilos se agrupan en conjuntos de 256 para formar un bloque de hilos, requiriendo por lo tanto un número de bloques de (Nº instancias / 256) 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 instancias / 256) x Nº individuos, representada en la Figura 5.1. 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.

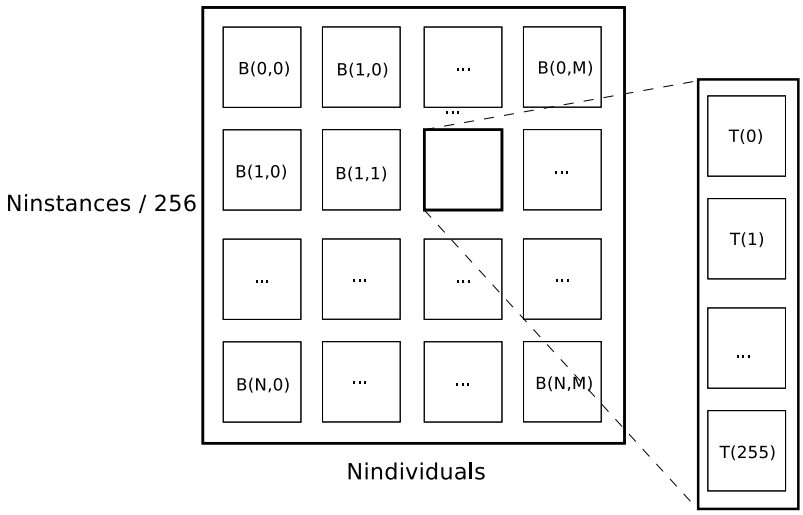

Figura 5.1: Matriz de bloques de hilos de ejecución

Una primera aproximación al recuento paralelo de los resultados de todas las evaluaciones sería que el primer hilo sumase los (Nº instancias / 256) primeros valores, el segundo los (Nº instancias / 256) siguientes, etc. Sin embargo, esta forma es ineficiente en el sentido de que no favorece la coalescencia en los accesos a memoria. Las posiciones de memoria a las que acceden los hilos del warp en ejecución se encuentran distanciadas (Nº instancias / 256) 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, 256, 512... el segundo hilo las posiciones 1, 257, 513... hasta llegar al último hilo que sumaría las posiciones 255, 511, 767... De esta forma tenemos igualmente 256 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 y es más eficiente al poder disponer de muchos más datos en un único acceso a memoria, evitando latencias innecesarias. 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 5.2.

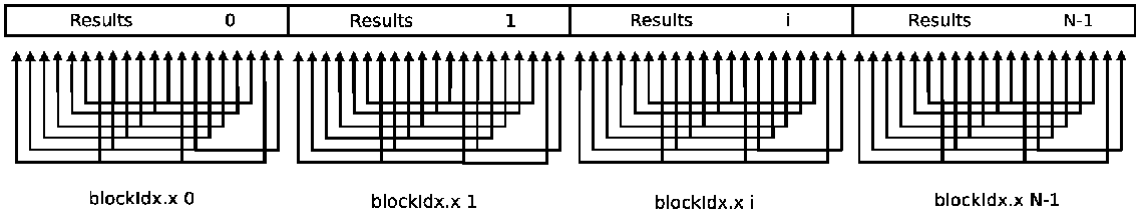

Figura 5.2: Modelo de reducción paralelo

5.1.1. Ejecución con kernels concurrentes

En los últimos años, CUDA ha introducido los kernels concurrentes, permitiendo la ejecución simultánea de varios kernels mientras que existan recursos suficiences disponibles en los multiprocesdores y no existan dependencias entre los kernels. Los kernels concurrentes se lanzan mediante streams de CUDA, que también permiten la transferencia de datos asíncrona entre la memoria principal y la de la GPU, solapando las transferencias de datos y la ejecución de los kernels. El kernel que comprueba el cubrimiento del antecedente y del consecuente puede ejecutarse concurrentemente para ambas partes, puesto que no existen dependencias de datos entre ellos. La operación de reducción sobre los resultados del cubrimiento del antecedente y consecuente también puede realizarse de forma concurrente.

El uso de kernels concurrentes permite el lanzamiento de transferencias de datos asíncronas solapando transferencias y cómputo. No existe la necesidad de esperar la terminación del kernel para copiar datos entre la memoria principal y la de la GPU. De esta forma, la ejecución del kernel sobre el antecedente puede realizarse mientras se copian los datos del consecuente. La Figura 5.3 muestra los beneficios de la ejecución de kernels concurrentes y transferencias de memoria asíncronas, en comparación con el modelo clásico serializado de copia y ejecución. Las dependencias entre kernels son inevitablemente serializadas, pero kernels independientes pueden ejecutarse de forma solapada, que junto con la transferencia de datos asíncrona, nos permite el ahorro de un tiempo considerable.

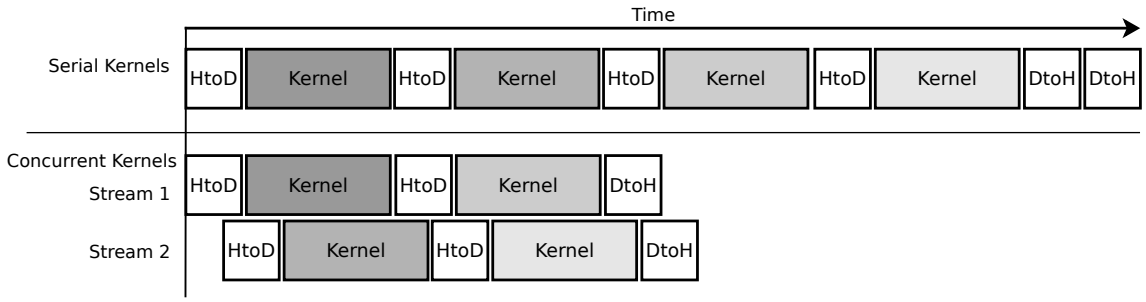

Figura 5.3: Línea temporal usando kernels serializados y concurrentes con transferencias asíncronas

5.2. Estudio experimental del modelo GPU

En esta sección se detalla la configuración de los experimentos, los conjuntos de datos, el algoritmos paralelizado y los parámetros de ejecución.

Para mostrar la flexibilidad y la aplicabilidad de nuestro modelo, el algoritmo G3PARM [11] de programación genética se paralelizará con nuestra propuesta, de la misma forma que se podría aplicar a otros algoritmos o paradigmas. Este algoritmo está implementado en Java dentro del software de computación evolutiva JCLEC [22].

5.2.1. Conjuntos de datos

Para evaluar el desempeño del modelo de evaluación de reglas propuesto, hemos seleccionado varios conjuntos de datos del repositorio de UCI [15] y del sitio web de la herramienta de software KEEL [3] y de su repositorio de conjuntos de datos [2]. Estos conjuntos de datos son muy variados teniendo en cuenta diferentes grados de complejidad. El número de instancias varía desde el más simple que contiene 569 instancias, hasta los más complejos que contienen hasta un millón de instancias. Esta información se resume en la Tabla 5.1. La gran variedad de conjuntos de datos considerados nos permite evaluar los resultados del modelo bajo diferentes complejidades del problema. Es interesante mencionar que algunos de estos conjuntos de datos tales como KDDcup o Poker no han sido comúnmente tratados hasta la fecha, porque no son habitualmente tratables por los modelos tradicionales.

Tabla 5.1: Información de los conjuntos de datos

Dataset	#Instancias	#Atributos
WDBC	569	31
Abalone	4174	9
Spambase	4597	58
Page-blocks	5472	11
Thyroid	7200	22
Magic	19020	11
House_16H	22784	17
Shuttle	58000	10
KDDcup	494020	42
Poker	1025010	11

5.2.2. Configuración de la experimentación

El código de evaluación de la GPU se compila en una biblioteca compartida y se cargan en el software JCLEC [22] utilizando JNI. Utilizando dicha biblioteca, nuestro modelo puede ser fácilmente exportado y empleado en cualquier otro sistema de aprendizaje de reglas.

Los experimentos se llevaron a cabo en un PC equipado con un procesador Intel Core i7 920 de cuatro núcleos a 2.66GHz y 12 GB de memoria principal DDR3-1600. El equipo cuenta con dos GPUs NVIDIA GeForce GTX 480 con 1,5 GB de RAM GDDR5. El sistema operativo fue Ubuntu 11.10 de 64 bits, junto con la runtime de CUDA 4.2.

5.3. Resultados del modelo GPU

En esta sección se analizan los resultados de los experimentos, cuyo propósito es analizar el efecto de la complejidad del algoritmo y de los conjuntos de datos sobre el desempeño del modelo de evaluación GPU y la escalabilidad de la propuesta. El algoritmo se ejecuta en todos los conjuntos de datos mediante un enfoque secuencial, un enfoque paralelo en CPU, y un enfoque GPU masivamente paralelo. El primer experimento evalúa el rendimiento del intérprete de reglas, en términos del número de símbolos por segundo que puede analizar. El segundo experimento evalúa la aceleración de las versiones paralelas en CPU y GPU con respecto a la implementación original secuencial.

5.3.1. Rendimiento del intérprete de reglas

El componente más importante del evaluador es el intérprete de las reglas, que evalúa las condiciones de las reglas. En el primer experimento se mide el rendimiento del intérprete para diferentes números de instancias y de reglas, que determinan la complejidad computacional del evaluador. La métrica para medir el rendimiento es el número de símbolos por segundo que puede evaluar. Específicamente, en el contexto de la programación genética se denominan GPops/s, es decir, el número de operaciones de programación genética por segundo.

La Tabla 5.2 muestra los tiempos del intérprete de reglas y el rendimiento en términos de GPops/s. Los resultados se han obtenido de la media de 10 ejecuciones. El rendimiento del modelo en GPU usando una y dos GPUs GTX 480 se compara con el obtenido para CPU con un hilo y múltiples hilos. Cada fila representa el caso del intérprete con un número diferente de instancias y reglas. Centrándonos en el número de operaciones de programación genética a evaluar, su valor depende del número de instancias, el número de reglas y el número de condiciones de las reglas.

Cuanto mayor es el número de instancias y de reglas a evaluar, mayor es el número de operaciones de GP a computar, y por lo tanto, mayor es el tiempo de computación requerido. Con respecto al rendimiento en CPU, la implementación en un hilo logra un rendimiento lineal y permanece constante en torno a 10 millones de GPops/s, independientemente del número de instancias y de reglas. La implementación multi-hilo incrementa el rendimiento hasta alrededor de 35 millones de GPops/s empleando los 4 núcleos de la CPU.

La implementación en GPU logra un gran rendimiento independientemente del número de reglas a interpretar y el número de instancias. Cuanto mayor es el número de reglas e instancias, mayor es el número de hilos y bloques de hilos a computar, y consecuentemente, se logra una mayor ocupación de los multiprocesadores de la GPU. Empleando una GPU, el límite de rendimiento se alcanza en 33 mil millones de GPops/s. Por otro lado, cuando se emplean dos GPUs, el límite se eleva a 67 mil millones de GPops/s, demostrando la gran escalabilidad del modelo en cuanto al número de GPUs y la complejidad del problema. De esta forma, es incuestionalbe el gran rendimiento obtenido, que sólo toma 0.172 segundos en interpretar 200 reglas sobre un millón de instancias. Los resultados se resumen en la Figura 5.4, que muestra el rendimiento del intérprete con dos GPUs en términos del número de GPops/s con respecto al número de instancias y de reglas.

Tabla 5.2: Rendimiento del intérprete de reglas

			Tiempo (ms)				GPops/s (M)			
Instancias	Reglas	GPops	CPU	4 CPUs	1 GPU	2 GPUs	CPU	4 CPUs	1 GPU	2 GPUs
2.5×10^2	25	3.62×10^5	34	14	0.053	0.044	10.66	25.89	6.84	8.20
2.5×10^2	50	7.25×10^5	68	25	0.059	0.047	10.66	29.00	12.26	15.43
2.5×10^2	100	1.45×10^6	153	68	0.073	0.057	9.48	21.32	19.77	25.41
2.5×10^2	200	2.90×10^6	296	93	0.145	0.077	9.80	31.18	20.03	37.53
5.0×10^2	25	7.25×10^5	70	23	0.050	0.044	10.36	31.52	14.41	16.57
5.0×10^2	50	1.45×10^6	137	46	0.074	0.058	10.58	31.52	19.50	24.82
5.0×10^2	100	2.90×10^6	307	111	0.110	0.077	9.45	26.13	26.33	37.62
5.0×10^2	200	5.80×10^6	544	182	0.225	0.124	10.66	31.87	25.73	46.93
1.0×10^3	25	1.45×10^6	137	44	0.068	0.057	10.58	32.95	21.45	25.36
1.0×10^3	50	2.90×10^6	309	89	0.119	0.076	9.39	32.58	24.32	38.05
1.0×10^3	100	5.80×10^6	565	158	0.207	0.122	10.27	36.71	28.06	47.46
1.0×10^3	200	1.16×10^7	1,076	390	0.409	0.207	10.78	29.74	28.39	56.04
2.5×10^3	25	3.62×10^6	359	130	0.135	0.088	10.10	27.88	26.91	40.97
2.5×10^3	50	7.25×10^6	730	295	0.253	0.141	9.93	24.58	28.65	51.53
2.5×10^3	100	1.45×10^7	1,371	375	0.461	0.254	10.58	38.67	31.48	57.10
2.5×10^3	200	2.90×10^7	2,632	1,127	0.938	0.471	11.02	25.73	30.93	61.57
5.0×10^3	25	7.25×10^6	839	243	0.241	0.140	8.64	29.84	30.14	51.90
5.0×10^3	50	1.45×10^7	1,385	496	0.461	0.252	10.47	29.23	31.45	57.55
5.0×10^3	100	2.90×10^7	2,602	921	1.102	0.476	11.15	31.49	26.30	60.86
5.0×10^3	200	5.80×10^7	5,699	2,214	1.824	0.912	10.18	26.20	31.79	63.57
1.0×10^4	25	1.45×10^7	1,442	484	0.459	0.249	10.06	29.96	31.59	58.30
1.0×10^4	50	2.90×10^7	2,689	1,010	0.901	0.471	10.78	28.71	32.20	61.56
1.0×10^4	100	5.80×10^7	5,392	1,651	1.791	0.912	10.76	35.13	32.39	63.59
1.0×10^4	200	1.16×10^8	11,010	3,971	3.580	1.799	10.54	29.21	32.40	64.48
2.5×10^4	25	3.62×10^7	3,528	1,318	1.099	0.581	10.27	27.50	32.97	62.38
2.5×10^4	50	7.25×10^7	6,535	2,419	2.180	1.100	11.09	29.97	33.25	65.91
2.5×10^4	100	1.45×10^8	14,086	4,421	4.350	2.306	10.29	32.80	33.34	62.88
2.5×10^4	200	2.90×10^8	28,006	8,983	8.699	4.367	10.35	32.28	33.34	66.40
5.0×10^4	25	7.25×10^7	7,902	2,450	2.177	1.141	9.17	29.59	33.30	63.56
5.0×10^4	50	1.45×10^8	14,677	4,694	4.341	2.177	9.88	30.89	33.41	66.61
5.0×10^4	100	2.90×10^8	28,688	11,411	8.679	4.350	10.11	25.41	33.41	66.67
5.0×10^4	200	5.80×10^8	55,528	20,762	17.470	8.688	10.45	27.94	33.20	66.76
1.0×10^5	25	1.45×10^8	13,749	4,889	4.382	2.257	10.55	29.66	33.09	64.26
1.0×10^5	50	2.90×10^8	28,609	9,662	8.757	4.422	10.14	30.01	33.11	65.58
1.0×10^5	100	5.80×10^8	56,188	17,125	17.402	8.740	10.32	33.87	33.33	66.36
1.0×10^5	200	1.16×10^9	110,195	31,418	35.100	17.462	10.53	36.92	33.05	66.43
2.5×10^5	25	3.62×10^8	36,032	10,682	10.766	5.607	10.06	33.94	33.67	64.65
2.5×10^5	50	7.25×10^8	69,527	21,163	21.990	10.911	10.43	34.26	32.97	66.45
2.5×10^5	100	1.45×10^9	133,125	42,765	43.573	22.036	10.89	33.91	33.28	65.80
2.5×10^5	200	2.90×10^9	269,071	84,591	86.747	43.604	10.78	34.28	33.43	66.51
5.0×10^5	25	7.25×10^8	67,989	20,980	21.512	11.194	10.66	34.56	33.70	64.77
5.0×10^5	50	1.45×10^9	137,671	40,351	43.057	21.537	10.53	35.93	33.68	67.33
5.0×10^5	100	2.90×10^9	291,238	81,028	86.703	43.147	9.96	35.79	33.45	67.21
5.0×10^5	200	5.80×10^9	602,719	170,074	172.970	86.480	9.62	34.10	33.53	67.07
1.0×10^6	25	1.45×10^9	144,987	43,357	43.001	22.396	10.00	33.44	33.72	64.74
1.0×10^6	50	2.90×10^9	285,639	84,678	86.248	43.009	10.15	34.25	33.62	67.43
1.0×10^6	100	5.80×10^9	560,459	170,977	172.258	86.129	10.35	33.92	33.67	67.34
1.0×10^6	200	1.16×10^{10}	1,097,670	332,387	344.908	172.681	10.57	34.90	33.63	67.18

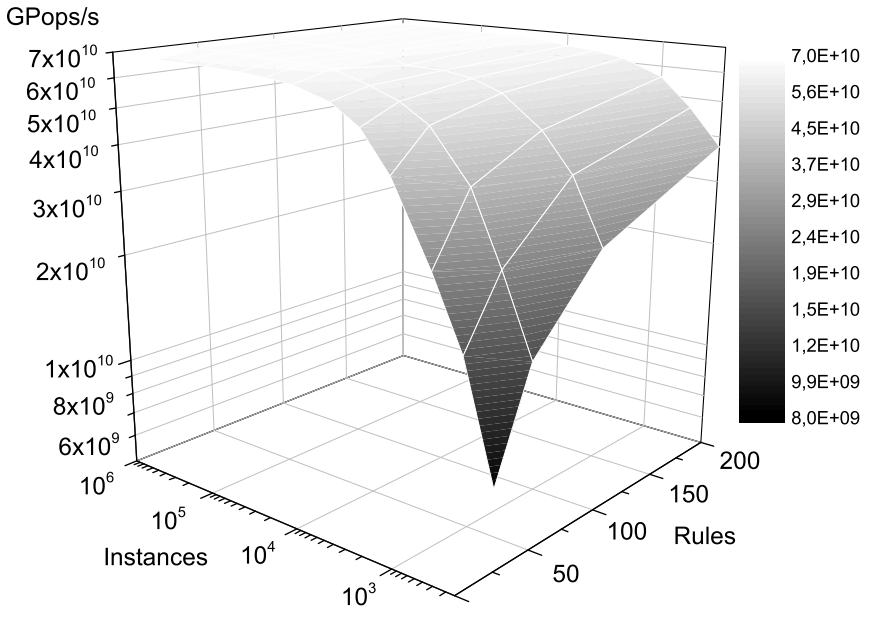

Figura 5.4: Rendimiento del intérprete con dos GPUs

5.3.2. Rendimiento del modelo de evaluación de reglas

La Tabla 5.3 muestra los tiempos de ejecución y la aceleración (speedup) obtenidos por el modelo sobre 10 conjuntos de datos del repositorio UCI. Los resultados muestran la media de 10 ejecuciones. Cada fila representa el tiempo de ejecución y la aceleración de la evaluación de una serie de reglas sobre un grupo de conjuntos de datos con diferente número de instancias y atributos.

En cuanto al rendimiento del modelo en CPU, se muestra que el tiempo de evaluación se incrementa de forma lineal conforme aumenta el número de instancias y de reglas a evaluar. Por lo tanto, la evaluación de reglas de asociación sobre conjuntos de datos de alta dimensionalidad se convierte en un problema intratable, especialmente si el proceso se encuentra embebido dentro de un algoritmo evolutivo, puesto que el proceso de evaluación deberá repetirse en todas las generaciones.

Con respecto al modelo de evaluación en GPU, los tiempos de evaluación presentan pocas variaciones conforme aumenta el número de reglas e instancias. Empleando dos GPUs 480, el tiempo es menor de un segundo en la mayoría de los casos evaluados. Para casos extremos, el modelo de evaluación requiere únicamente tres segundos para evaluar 200 reglas sobre conjuntos de datos de más de un millón de instancias.

Tabla 5.3: Rendimiento del evaluador en conjuntos de datos UCI

		Tiempo evaluación (ms)				Speedup vs CPU			Speedup vs 4 CPUs	
Conjunto de datos	Reglas	CPU	4 CPUs	1 GPU	2 GPUs	4 CPUs	1 GPU	2 GPUs	1 GPU	2 GPUs
WDBC	25	111	39	5	3	2.85	22.20	37.00	7.80	13.00
Instancias: 569	50	251	86	6	4	2.92	41.83	62.75	14.33	21.50
Atributos: 31	100	516	184	11	7	2.80	46.91	73.71	16.73	26.29
	200	785	406	13	7	1.93	60.38	112.14	31.23	58.00
Abalone	25	556	233	5	4	2.39	111.20	139.00	46.60	58.25
Instancias: 4,174	50	1,158	450	11	9	2.57	105.27	128.67	40.91	50.00
Atributos: 9	100	2,250	748	21	14	3.01	107.14	160.71	35.62	53.43
	200	4,083	1,676	30	19	2.44	136.10	214.89	55.87	88.21
Spambase	25	796	282	9	4	2.82	88.44	199.00	31.33	70.50
Instancias: 4,597	50	1,460	558	14	8	2.62	104.29	182.50	39.86	69.75
Atributos: 58	100	3,120	1,146	21	15	2.72	148.57	208.00	54.57	76.40
	200	6,326	2,140	35	20	2.96	180.74	316.30	61.14	107.00
Page-blocks	25	718	243	9	6	2.95	79.78	119.67	27.00	40.50
Instancias: 5,472	50	1,715	524	17	15	3.27	100.88	114.33	30.82	34.93
Atributos: 11	100	3,354	1,068	22	20	3.14	152.45	167.70	48.55	53.40
	200	7,127	2,098	47	26	3.40	151.64	274.12	44.64	80.69
Thyroid	25	1,066	349	9	5	3.05	118.44	213.20	38.78	69.80
Instancias: 7,200	50	2,348	710	16	10	3.31	146.75	234.80	44.38	71.00
Atributos: 22	100	4,386	1,493	28	18	2.94	156.64	243.67	53.32	82.94
	200	9,269	3,451	51	26	2.69	181.75	356.50	67.67	132.73
Magic	25	2,812	1,045	16	17	2.69	175.75	165.41	65.31	61.47
Instancias: 19,020	50	5,593	2,044	31	26	2.74	180.42	215.12	65.94	78.62
Atributos: 11	100	11,675	4,301	61	36	2.71	191.39	324.31	70.51	119.47
	200	24,968	9,452	115	73	2.64	217.11	342.03	82.19	129.48
House_16H	25	3,323	1,115	33	21	2.98	100.70	158.24	33.79	53.10
Instancias: 22,784	50	7,390	2,119	59	34	3.49	125.25	217.35	35.92	62.32
Atributos: 17	100	13,318	4,234	100	60	3.15	133.18	221.97	42.34	70.57
	200	27,595	9,296	182	100	2.97	151.62	275.95	51.08	92.96
Shuttle	25	9,519	3,151	51	28	3.02	186.65	339.96	61.78	112.54
Instancias: 58,000	50	19,051	6,308	96	56	3.02	198.45	340.20	65.71	112.64
Atributos: 10	100	38,535	11,822	154	91	3.26	250.23	423.46	76.77	129.91
	200	78,674	23,916	301	173	3.29	261.38	454.76	79.46	138.24
Kddcup	25	75,835	18,452	368	281	4.11	206.07	269.88	50.14	65.67
Instancias: 494,020	50	142,892	36,280	684	446	3.94	208.91	320.39	53.04	81.35
Atributos: 42	100	310,216	73,164	1,502	931	4.24	206.54	333.21	48.71	78.59
	200	622,942	152,385	2,815	1,550	4.09	221.29	401.90	54.13	98.31
Poker	25	154,548	41,700	691	436	3.71	223.66	354.47	60.35	95.64
Instancias: 1,025,010	50	337,268	100,278	1,504	915	3.36	224.25	368.60	66.67	109.59
Atributos: 11	100	735,638	219,419	3,137	1,974	3.35	234.50	372.66	69.95	111.15
	200	1,465,010	414,267	6,180	3,397	3.54	237.06	431.27	67.03	121.95

Finalmente, centrándonos en la aceleración lograda con una GPU, se muestra que la aceleración obtenida alcanza un límite de 200. Esta aceleración máxima es obtenida en conjuntos de datos de más de 50 mil instancias. Por otro lado, empleando dos GPUs, la aceleración máxima alcanza valores en torno a 400, doblando el rendimiento de una GPU. La aceleración máxima se obtiene cuando todos los multiprocesadores de la GPU están plenamente ocupados con carga de trabajo, por lo que emplear múltiples GPUs permite abordar conjuntos de datos con un mayor número de instancias. Los resultados se resumen en la Figura 5.5, que muestra los resultados de aceleración de la evaluación empleando dos GPUs GTX 480, en función del número de instancias del conjunto de datos y del número de reglas a evaluar.

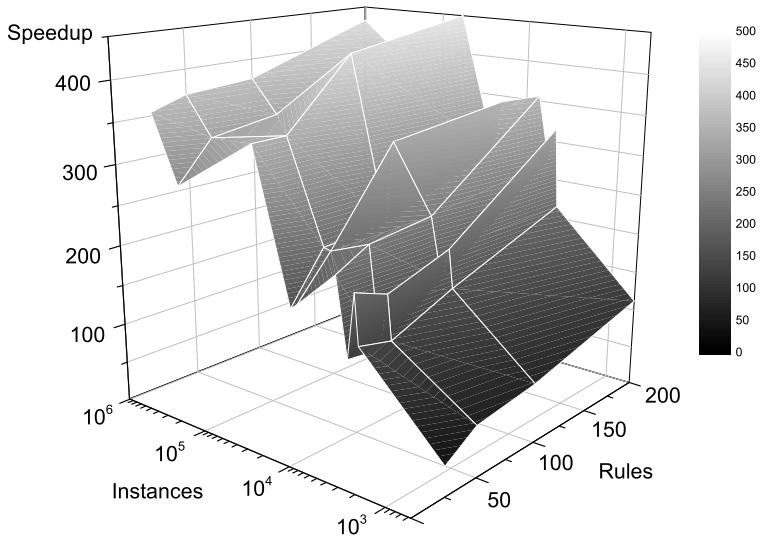

Figura 5.5: Aceleración con dos GPUS GTX 480

5.3.3. Kernels serial vs concurrentes

La independencia entre los kernels sobre el antecedente y el consecuente de las reglas permite que su ejecución esté solapada. La Figura 5.6 muestra la línea temporal de la computación en GPU para kernels serial y concurrentes. Esta figura se ha obtenido mediante el software de NVIDIA Visual Profiler. Se muestra claramente el ahorro de tiempo debido a la ejecución concurrente y las transferencias asíncronas. En primer lugar se observan las transferencias de memoria desde memoria principal a la memoria de la GPU, que copian los antecedentes y consecuentes de las reglas. Estas transferencias se realizan de forma asíncrona y simultánea en varios streams. En segundo lugar, se observa la ejecución de los kernels de cubrimiento y reducción de forma concurrente para el antecedente y el consecuente. Finalmente, se ejecuta el kernel del cómputo del fitness. Por último, se realizan las transferencias de resultados desde la memoria de la GPU a memoria principal. Estas transferencias también se ejecutan de forma asíncrona y concurrente con la computación de los kernels.

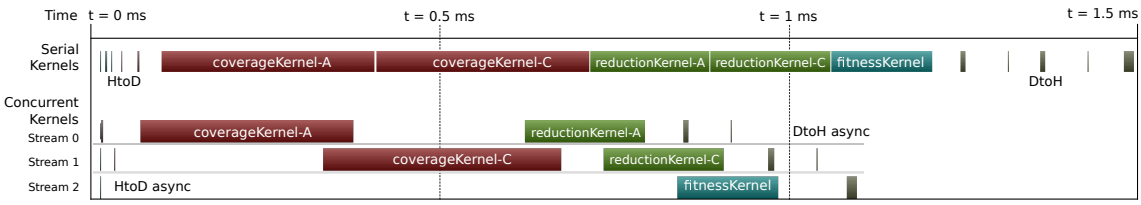

Figura 5.6: Línea temporal de la GPU

6. CONCLUSIONES

La tarea de minería de reglas de asociación ha demostrado ser un verdadero reto con problemas abiertos a nuevas investigaciones para mejorar la eficiencia, rendimiento y calidad de los resultados mediante nuevos modelos.

En este trabajo, hemos abordado el problema del tiempo de ejecución para algoritmos basados en reglas mediante algoritmos evolutivos y hemos aportado un modelo paralelo de evaluación de las reglas mediante GPUs. Los resultados obtenidos con este modelo, aplicado sobre múltiples conjuntos de datos, aportan aceleraciones de hasta 454 veces empleando una pequeña parte del coste económico de un sistema de alto rendimiento tradicional. El modelo ha demostrado una buena escalabilidad frente al número de instancias y permite el uso de múltiples GPUs, aumentando los límites del problema abordables por el modelo. El uso de la GPU permite abordar a bajo coste nuevos retos en minería de reglas de asociación ya que hace viables algunos problemas previamente dados por imposible debido a su elevado coste computacional.

El trabajo realizado y el bagaje adquirido a lo largo del último año en el máster ha abierto nuevos problemas y oportunidades para abordar con nuevos modelos de asociación con objetivo de continuar la tesis doctoral.

El uso de GPUs para acelerar el desempeño de modelos de aprendizaje automático ha demostrado ser muy eficaz, prueba de ello es la tendencia de artículos que abordan dicha temática. En los próximos años las GPUs se aplicarán para resolver de forma más eficiente todo tipo de problemas y además permitirá abordar nuevos problemas. La escalabidad de los modelos con GPUs será clave para ello, por lo tanto será necesario tener en cuenta el diseño eficiente de sistemas de alto rendimiento con múltiples GPUs distribuidas en múltiples máquinas interconectadas. De esta forma se podrán resolver de una manera más intuitiva y eficiente nuevos problemas, como los algoritmos evolutivos con múltiples poblaciones distribuídas en islas, que evolucionan independientemente y cada cierto tiempo migran individuos entre ellas.

BIBLIOGRAFÍA

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[2] J. Alcalá-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. García, L. Sánchez, and F. Herrera. KEEL Data-Mining Software Tool: Data Set Repository, Integration of Algorithms and Experimental Analysis Framework. *Analysis Framework. Journal of Multiple-Valued Logic and Soft Computing*, 17:255–287, 2011.

[3] J. Alcalá-Fdez, L. Sánchez, S. García, M. del Jesus, S. Ventura, J. Garrell, J. Otero, C. Romero, J. Bacardit, V. Rivas, J. Fernández, and F. Herrera. KEEL: A Software Tool to Assess Evolutionary Algorithms for Data Mining Problems. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13:307–318, 2009.

[4] M. S. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.

[5] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, 1985.

[6] J. H. Eom and B. T. Zhang. Prediction of protein interaction with neural network-based feature association rule mining. *Lecture Notes in Computer Science*, 4234:30–39, 2006.

[7] U. Fayyad, G. Piatetsky, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.

[8] W. J. Frawley, G. Piatetsky, and C. J. Matheus. Knowledge discovery in databases: An Overview. *AI Magazine*, 13, 1992.

[9] D. Kirk, W. W. Hwu, and J. Stratton. Reductions and their implementation. Technical report, University of Illinois, Urbana-Champaign, 2009.

[10] J. R. Koza. *Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems.* Stanford University, Stanford, CA, USA, 1990.

[11] J. M. Luna, J. R. Romero, and S. Ventura. Analysis fo the Effectiveness of G3PARM Algorithm. In *Proceedings of the 5th International Conference on Hybrid Artificial Intelligent Systems, HAIS'10*, pages 27–34, 2010.

[12] J. M. Luna, J. R. Romero, and S. Ventura. G3PARM: A Grammar Guided Genetic Programming Algorithm for Mining Association Rules. In *Proceedings of the IEEE Congress on Evolutionary Computation, IEEE CEC 2010*, pages 2586–2593, 2011.

[13] J. M. Luna, J. R. Romero, and S. Ventura. Mining and Representing Rare Association Rules through the Use of Genetic Programming. In *Proceedings of the 3rd World Congress on Nature and Biologically Inspired Computing, NaBIC 2011*, pages 86–91, 2011.

[14] J. M. Luna, J. R. Romero, and S. Ventura. Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. *Knowledge and Information Systems*, 32(1):53–76, 2012.

[15] D. J. Newman and A. Asuncion. UCI machine learning repository, 2007.

[16] J. L. Olmo, J. M. Luna, J. R. Romero, and S. Ventura. Association Rule Mining using a Multi-Objective Grammar-Based Ant Programming Algorithm. In *Proceedings of the 11th International Conference on Intelligent Systems Design and Applications, ISDA 2011*, pages 971–977, 2011.

[17] C. Ordonez, N. Ezquerra, and C.A. Santana. Constraining and summarizing association rules in medical data. *Knowledge and Information Systems*, 9(3):259–283, 2006.

[18] C. Romero, J. M. Luna, J. R. Romero, and S. Ventura. Mining Rare Association Rules from e-Learning Data. In *Proceedings of the 3rd International Conference on Educational Data Mining, EDM 2010*, pages 171–180, 2010.

[19] C. Romero, A. Zafra, J. M. Luna, and S. Ventura. Association rule mining using genetic programming to provide feedback to instructors from multiple-choice quiz data. *Expert Systems*, In press, 2012.

[20] A. Schuster, R. Wolff, and D. Trock. A high-performance distributed algorithm for mining association rules. *Knowledge and Information Systems*, 7(4):458–475, 2004.

[21] S. F. Smith. *A learning system based on genetic adaptive algorithms.* PhD thesis, Pittsburgh, PA, USA, 1980.

[22] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás. JCLEC: a Java framework for evolutionary computation. *Soft Computing*, 12:381–392, 2007.

[23] NVIDIA CUDA Zone. NVIDIA Programming and Best Practices Guide 5.0, http://www.nvidia.com/cuda.

J Supercomput
DOI 10.1007/s11227-013-0937-4

High performance evaluation of evolutionary-mined association rules on GPUs

Alberto Cano · José María Luna · Sebastián Ventura

Abstract Association rule mining is a well-known data mining task, but it requires much computational time and memory when mining large scale data sets of high dimensionality. This is mainly due to the evaluation process, where the antecedent and consequent in each rule mined are evaluated for each record. This paper presents a novel methodology for evaluating association rules on graphics processing units (GPUs). The evaluation model may be applied to any association rule mining algorithm. The use of GPUs and the compute unified device architecture (CUDA) programming model enables the rules mined to be evaluated in a massively parallel way, thus reducing the computational time required. This proposal takes advantage of concurrent kernels execution and asynchronous data transfers, which improves the efficiency of the model. In an experimental study, we evaluate interpreter performance and compare the execution time of the proposed model with regard to single-threaded, multi-threaded, and graphics processing unit implementation. The results obtained show an interpreter performance above 67 billion giga operations per second, and speed-up by a factor of up to 454 over the single-threaded CPU model, when using two NVIDIA 480 GTX GPUs. The evaluation model demonstrates its efficiency and scalability according to the problem complexity, number of instances, rules, and GPU devices.

Keywords Performance evaluation · Association rules · Parallel computing · GPU

A. Cano · J.M. Luna · S. Ventura (✉)
Department of Computer Science and Numerical Analysis, University of Cordoba, Cordoba 14071, Spain
e-mail: sventura@uco.es

A. Cano
e-mail: acano@uco.es

J.M. Luna
e-mail: jmluna@uco.es

Published online: 26 April 2013

1 Introduction

Association rule mining (ARM) is a widely known data mining technique which was first introduced by Agrawal et al. [3] in the early 1990s. ARM was conceived as an unsupervised learning task for finding close relationships between items in large data sets. Let $\mathcal{I} = \{i_1, i_2, i_3, \ldots, i_n\}$ be the set of items and let $\mathcal{D} = \{t_1, t_2, t_3, \ldots, t_n\}$ be the set of all transactions in a relation. An association rule is an implication of the form $X \rightarrow Y$ where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The meaning of an association rule is that if antecedent X is satisfied, then it is highly likely that consequent Y will also be satisfied. ARM was originally designed for market basket analysis to obtain relations between products like *diapers* $\rightarrow$ *beer* that describes the high probability of someone buying diapers also buying beer. It would allow shop-keepers to exploit this relationship by moving the products closer together on the shelves. ARM tasks have also been applied in a wide range of domains, such as customer behavior [10], bank transactions [44], and medical diseases [38], where it is particularly important to discover, for instance, relationships between a specific disease and specific life habits.

The first algorithms in this field were based on an exhaustive search methodology, extracting all those rules which satisfy minimum quality thresholds [9, 22]. These approaches divide the ARM problem into two sub-problems: mining items and itemsets whose frequency of occurrence is greater than a specific threshold, and discovering strong relationships between these frequent itemsets. The first step is computationally expensive since it analyzes every item within the data set. The second step requires large amounts of memory because it holds every rule generated in memory.

As interest in storing information grows, data sets are increasing in the number of transactions or of items, thus prompting researchers to study the computational and memory requirements concerning ARM algorithms [22] in depth. Besides, real-world data sets usually contain numerical items, so a pre-processing step is required for mining rules by using exhaustive search approaches. The problem with using numerical attributes is the huge number of distinct values they might take on, making their search spaces much bigger, enlarging the computational time and memory requirements and therefore hampering the mining process.

For the sake of overcoming such drawbacks, researchers have achieved some promising results when using evolutionary algorithms [36, 40], which is becoming the most widely employed means to this end. Whereas exhaustive search algorithms for mining association rules first mine frequent items, evolutionary algorithms extract association rules directly, thus not requiring a prior step for mining frequent items. Grammar-guided genetic programming (G3P) [24] has also been applied to the extraction of association rules [37], where the use of a grammar allows for defining syntax constraints, restricting the search space, and obtaining expressive solutions in different attribute domains. Nevertheless, the capabilities of data collection in real-world application domains are still growing, hindering the mining process even when evolutionary algorithms are used. Therefore, it is becoming essential to design algorithms capable of handling very large data collections [5] in a reasonable time.

Parallel frequent pattern mining is emerging as an interesting research area, where many parallel and distributed methods have been proposed to reduce computational

time [19, 25, 48]. To this end, hierarchical parallel environments with multi-core processors [2] and graphic processing units (GPUs) [6, 14] help to speed up the process. The use of GPUs has already been studied in machine learning [13, 21, 39], and specifically for speeding up algorithms within the framework of evolutionary computation [15, 23, 33] and data mining [28, 35]. Specifically, GPUs have also been successfully applied to speeding up the evaluation process of classification rules [12, 16].

In this paper, a new parallel methodology for evaluating association rules on GPUs is described. This methodology could be implemented on any ARM approach, regardless of the design and methodology of the algorithm. This issue is of special interest for mining association rules by means of evolutionary algorithms, and more specifically for G3P algorithms, which allow for mining rules on any domain. Evolutionary computation algorithms devote much time to the evaluation of the population. Therefore, it is especially interesting to speed up the evaluation phase in this kind of algorithms. This synergy prompts developing a high performance model, which demonstrates good performance in the experimental study presented herein.

This paper is organized as follows. In Sects. 2 and 3, association rule mining algorithms and the way of evaluating them are presented, respectively. Section 4 presents some details concerning the CUDA programming model. In Sect. 5, the evaluation methodology on GPUs is proposed. Section 6 describes the experimental study. The results obtained are discussed in Sect. 7. Finally, Sect. 8 presents some concluding remarks.

2 Association rules mining algorithms

Most existing proposals for mining association rules are based on Agrawal's proposal in [3], which is widely known as *Apriori*. In the *Apriori* algorithm, authors divided the ARM problem into two steps. First, those patterns that frequently appear in a data set are extracted. Second, using the whole set of previously mined patterns, the algorithm seeks association rules. This exhaustive search problem becomes impracticable with increasing information storage in a data set, so the anti-monotone property was introduced: if a length-k itemset is not frequent in a data set, its length-$(k+1)$ super-itemsets cannot be frequent in the data set. *Apriori* has four major drawbacks. First, the higher the number of items in a data set, the more tedious the mining of frequent patterns becomes. Secondly, the higher the number of frequent patterns, the more tedious becomes the discovery of association rules. Besides, *Apriori* works in two steps, increasing the computational time required. Finally, real-world data sets usually comprise numerical items which are hard to be mined using *Apriori*, so a pre-processing step is used. Therefore, the number of steps required by this algorithm is even greater, thus hampering the mining process and significantly increasing execution time.

Many researchers have focused their research on *Apriori*-like algorithms with the goal of overcoming these problems. One of the most relevant algorithms to this end was FP-Growth [22]. In this algorithm, Han et al. proposed a novel way of storing the frequent patterns mined, putting them in a frequent pattern tree (FP-tree) structure. In such a way, the algorithm works on the FP-tree structure instead of the whole set

Table 1 Runtime of evolutionary ARM phases [37]

Initialization	Selection	Crossover	Mutation	Evaluation	Replacement	Total
4.47 %	0.01 %	0.01 %	0.03 %	95.45 %	0.03 %	100 %

of frequent patterns, employing a divide-and-conquer method to reduce the computational time. Recently, an improved version of FP-Growth was presented in [29]. This proposal, called IFP-Growth (Improved FP-Growth), uses a novel structure, called FP-tree+, and an address table to decrease the complexity of building the entire FP-tree. Both the FP-tree+ and the address table require less memory but provide a better performance than the algorithm originally proposed by Han et al.

With the growing interest in evolutionary algorithms (EA), more and more researchers have focused on the evolutionary perspective of data mining tasks [4, 40], especially in ARM, where an exhaustive search could not be an option using huge data sets. In such a way, most existing EAs for mining association rules overcome the *Apriori*-like algorithms' problems. First, the mere fact of using an evolutionary perspective enables the computational and memory requirements to be tackled. Furthermore, most EAs for mining association rules are based on genetic algorithms (GA), which discover rules directly instead of requiring two steps. Nevertheless, some GAs in the ARM field still require a pre-processing step for dealing with numerical items.

Recently, a genetic programming (GP) approach was introduced for mining association rules [37]. The main difference between GAs and that proposal is their representation. Whereas GAs use a fixed-length chromosome which is not very flexible, the proposal presented by Luna et al. represents individuals by using a variable-length hierarchical structure, where the shape, size, and structural complexity of the solutions are not constrained a priori. Furthermore, this algorithm uses a grammar to encode individuals, allowing for enforcing syntactic and semantic constraints and also dealing with any item regardless of its domain.

Despite the fact that the use of EAs in the field of ARM has overcome most of the drawbacks, evaluating the rules mined is still hard work when the data set is large. The higher the number of records to be checked, the higher is the evaluation time. Table 1 shows the runtime of the different phases of the evolutionary association rule mining algorithm [37]. The evaluation phase is clearly noted to take about 95 % of the algorithm's runtime. Therefore, it is desirable to solve this problem and GPUs are currently being presented as efficient and high-performance platforms for this goal.

3 Evaluation of association rules

ARM algorithms usually discover an extremely large number of association rules. Hence, it becomes impossible for the end user to understand such a large set of rules. Nowadays, the challenge for ARM is to reduce the number of association rules discovered, focusing on the extraction of only those rules that have real interest for the user or satisfy certain quality criteria. Therefore, evaluating the rules discovered in

Table 2 Contingency table for a sample rule

	X	$\neg X$	
Y	$n(XY)$	$n(\neg XY)$	$n(Y)$
$\neg Y$	$n(X\neg Y)$	$n(\neg X\neg Y)$	$n(\neg Y)$
	$n(X)$	$n(\neg X)$	N

the mining process has been studied in depth by different researchers and, many measures introduced to evaluate the interest of the rules [43]. These measures allow for 'common' rules to be filtered out from 'quality' or interesting ones.

In ARM, measures are usually calculated by means of frequency counts, so the study of each rule by using a contingency table helps to analyze the relationships between the patterns. Given a sample association rule $X \rightarrow Y$, its contingency table is defined as in Table 2.

On the basis of this contingency table, one can note the following.

- X states the antecedent of the rule.
- Y defines the consequent of the rule.
- $\neg X$ describes the fact of not satisfying the antecedent of the rule.
- $\neg Y$ defines the fact of not satisfying the consequent of the rule.
- $n(AC)$ is the number of records satisfying both A and C. Notice that A could be either X or $\neg X$, and C might be either Y or $\neg Y$.
- N is the number of records in the data set studied.

Once the contingency table has been obtained, any association rule measure can be calculated. Some measures focus not only on absolute frequencies but also on relative frequencies, which are denoted $p(A)$ instead of $n(A)$ when relative frequencies are borne in mind—a relative frequency $p(A)$ being defined as $n(A)/N$.

The two most frequently used measures in ARM are support and confidence. Support, also known as frequency or coverage, calculates the percentage of instances covering the antecedent X and the consequent Y in a data set. Support is defined by Eq. (1) as a relative frequency. Rules having a low support are often misleading since they do not describe a significant number of records. We have

$$Support(X \rightarrow Y) = p(XY) = n(XY)/N \tag{1}$$

Confidence measures the reliability of the association rule. This measure is defined by Eq. (2) as the proportion of the number of transactions which include X and Y among all the transactions that include X. We have

$$Confidence(X \rightarrow Y) = p(XY)/p(X) = n(XY)/n(X) \tag{2}$$

Most ARM proposals base their extraction process on the use of a support–confidence framework [9, 22, 47], attempting to discover rules whose support and confidence values are greater than certain minimum thresholds. However, the mere fact of having rules that exceed these thresholds is no guarantee that the rules will be of any interest, as noted by [8]. Other widely used measures to evaluate the interest of the extracted rules are lift [45], Piatetsky–Shapiro's measure [41] and conviction [11].

Lift or interest measure (see Eq. (3)) calculates how many times more often the antecedent and consequent are related in a data set than would be expected if they were statistically independent. It could also be calculated as the ratio between the confidence of the rule and the support of its consequent:

$$Lift(X \to Y) = p(XY)/p(X)p(Y) = n(XY)N/n(X)n(Y) \tag{3}$$

The leverage measure (see Eq. (4)) was proposed by Piatetsky-Shapiro. This measure calculates the difference between X and Y appearing together in the data set from what would be expected if they were statistically independent:

$$Leverage(X \to Y) = p(XY) - p(X)p(Y) \tag{4}$$

Conviction is another well-known measure in the ARM field (see Eq. (5)). It was developed as an alternative to confidence, which was found not to adequately capture the direction of associations. Conviction compares the probability that X would appear without Y if they were dependent with the actual frequency of the appearance of X without Y. We have

$$Conviction(X \to Y) = p(X)p(\neg Y)/p(X\neg Y) \tag{5}$$

These measures are those commonly used in ARM problems for evaluating the interest of the extracted association rules. However, there is a large variety of other measures which could also be applied, such as interest strength [20], Klosgen's measure [30], etc. Any measure for evaluating association rules is calculated by using the contingency table, so the mere fact of calculating this table allows for any association rule to be evaluated. Nevertheless, it is also interesting not only to obtain the contingency table but also to calculate the coverage of each condition within the rule.

4 The CUDA programming model

Computer unified device architecture (CUDA) [1] is a parallel computing architecture developed by the NVIDIA corporation that allows programmers to take advantage of the parallel computing capacity of NVIDIA GPUs in a general purpose manner. The CUDA programming model executes kernels as batches of parallel threads. These kernels comprise thousands to millions of lightweight GPU threads per invocation.

CUDA's threads are organized as a two-level hierarchy: at the higher one, all the threads in a data-parallel execution phase form a 3D grid. Each call to a kernel execution initiates a grid composed of many thread groupings, called thread blocks. Thread blocks are executed in streaming multiprocessors, as shown in Fig. 1. A stream multiprocessor can perform zero overhead scheduling to interleave warps (a warp is a group of threads that execute together) and hide the overhead of long-latency arithmetic and memory operations. Current GPU architectures may execute up to 16 kernels concurrently as long as there are multiprocessors available. Moreover, asynchronous data transfers can be performed concurrently with the kernel executions. These two features allow for speedier execution compared to a sequential kernel pipeline and synchronous data transfers in previous GPU architectures.

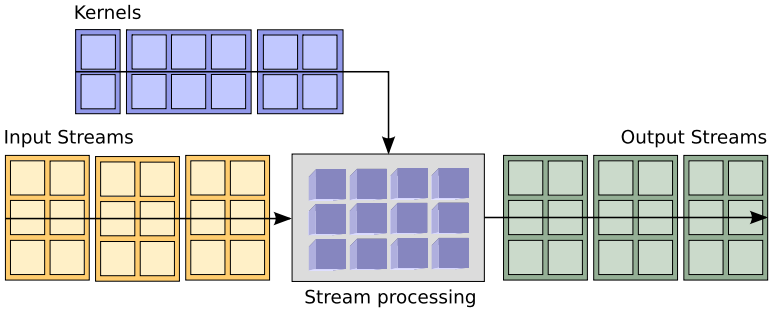

Fig. 1 GPU streaming processing paradigm

There are four different main memory spaces: global, constant, shared, and local. These GPU memories are specialized and have different access times, lifetimes, and output limitations.

- Global memory is a large long-latency memory which exists physically as an off-chip dynamic device memory. Threads can read and write global memory to share data and must write the kernel's output to be readable after the kernel terminates. However, a better way to share data and improve performance is to take advantage of shared memory.
- Shared memory is a small low-latency memory which exists physically as on-chip registers. Its contents are only maintained during thread block execution, and are discarded when the thread block completes. Kernels which read or write a known range of global memory with spatial or temporal locality can employ shared memory as a software-managed cache. Such caching potentially reduces global memory bandwidth demands and improves overall performance.
- Local memory is where each thread also has its own local memory space for registers, so the number of registers a thread uses determines the number of concurrent threads executed in the multiprocessor, which is called the multiprocessor occupancy. To avoid wasting hundreds of cycles while a thread waits for a long-latency global-memory load or store to complete, a common technique is to execute batches of global accesses, one per thread, exploiting the hardware's warp scheduling to overlap the threads' access latencies.
- Constant memory is specialized for situations in which many threads will read the same data simultaneously. This type of memory stores data written by the host thread, is accessed constantly, and does not change during the execution of the kernel. A value read from the constant cache is broadcast to all threads in a warp, effectively serving all loads from memory with a single-cache access. This enables a fast, single-ported cache to feed multiple simultaneous memory accesses.

Recommendations exist for improving performance on a GPU [18]. Memory accesses must be coalesced, as with accesses to global memory. Global memory resides in the device memory and is accessed via 32-, 64-, or 128-byte segment memory transactions. It is recommended to perform fewer but larger memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the

memory accesses of the threads within the warp into one or more of these memory transactions, depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, thus reducing the instruction throughput.

To maximize global memory throughput, it is therefore important to maximize coalescing by following optimal access patterns, using data types that meet size and alignment requirements, or padding data in some cases, for example, when accessing a two-dimensional array. For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size.

5 GPU evaluation model

This section presents the GPU model designed to evaluate association rules on the GPU. First, parallelization opportunities are analyzed to find the best way to parallelize the computation. Then, the kernels and data structures are given in detail.

The evaluation process of association rules has been described in Sect. 3. Its purpose is to check the coverage of the conditions of the rule, hence obtaining the coverage of the antecedent and consequent over the instances of the data set allowing the fitness measures to be computed.

Computing the evaluation of an association rule is independent from the evaluation of another rule, because there are no internal dependencies. Thus, the rules could be evaluated concurrently. Furthermore, the coverage process of the antecedent and the consequent of a rule over the instances is also independent. Hence, both the coverage of the antecedents and consequents from each rule could also be computed concurrently. Moreover, the coverage of a single instance is also an independent operation. Therefore, the coverage of the antecedents and consequents is independent for each rule and instance, which provides a high degree of parallelism. Computing the support of the conditions of the antecedent and the consequent is interesting when evaluating association rules and it could also be performed concurrently for each condition and rule. Nevertheless, calculating the support of the conditions and building the contingency table for computing the fitness measures calls for a reduction operation [26, 46] of the coverage results.

The GPU model designed to evaluate association rules comprises three kernels for computations of the coverage, reduction, and fitness. Figure 2 shows the computation flow. First, the rule, antecedent and consequent, are copied to the GPU memory. Specifically, they are copied to the constant memory, which provides broadcast to the GPU threads, taking advantage of the memory properties described in Sect. 4. Second, the coverage kernel is executed both on the antecedent and the consequent, using the instances of the data set. Third, the reduction kernel is executed and the support of the conditions is calculated. Fourthly, the fitness computation kernel is executed, providing the fitness measures described in Sect. 3. Finally, the results are copied back to the host memory. This figure represents the evaluation process of an association rule using serial kernels (one after the other). However, as mentioned before, there are some independencies which allow us to overlap their execution.

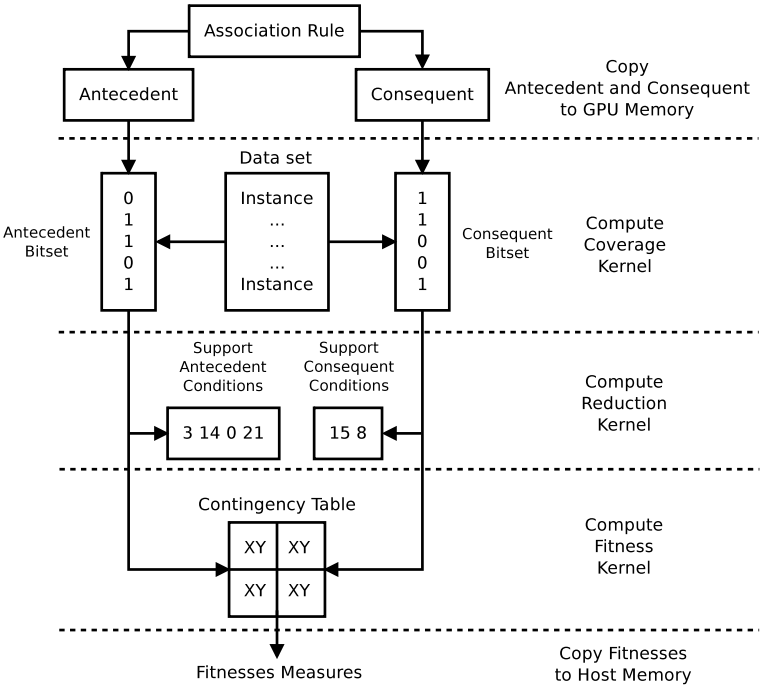

Fig. 2 GPU kernel model

The CUDA 2.0 architecture introduced concurrent kernels, allowing the kernel executions to be overlapped as long as there are sufficient available resources on the multiprocessors and no dependencies between kernels. Concurrent kernels are issued by means of CUDA streams, which also allow asynchronous data transfers between host and device memory, overlapping data transfers and kernel executions. The coverage kernel may overlap its execution when concurrently evaluating the antecedent and the consequent, since they operate with different data and they have no dependencies. After the coverage kernel is completed, the reduction kernel may also overlap its execution with the antecedent and consequent bitsets. Finally, the fitness kernel has no dependencies with the reduction kernel. Therefore, their execution may also overlap.

Concurrent kernels enable asynchronous data transfers, which overlap memory transactions and kernel computations. There is no need to wait for kernel completion to copy data between host and device memories. In such a way, the execution of the coverage kernel over the antecedent could be concurrent to copying the consequent to the GPU memory. Similarly, copying the support of the conditions back to host memory could be concurrent to the computation of the fitness kernel. Figure 3 illustrates the benefits of concurrent kernels and asynchronous transfers in the 2.0 architecture versus the old-style serial execution pipeline from previous GPU architectures. Kernel dependencies should still be serialized within the stream, but those kernels with independent computations may overlap their execution using multiple streams, along with asynchronous data transfers, thus saving time.

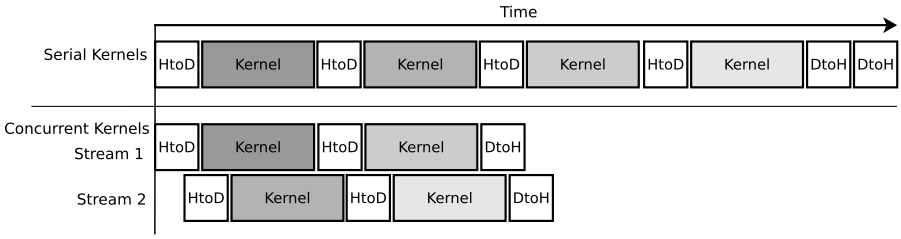

Fig. 3 GPU timeline using serial and concurrent kernels with asynchronous transfers

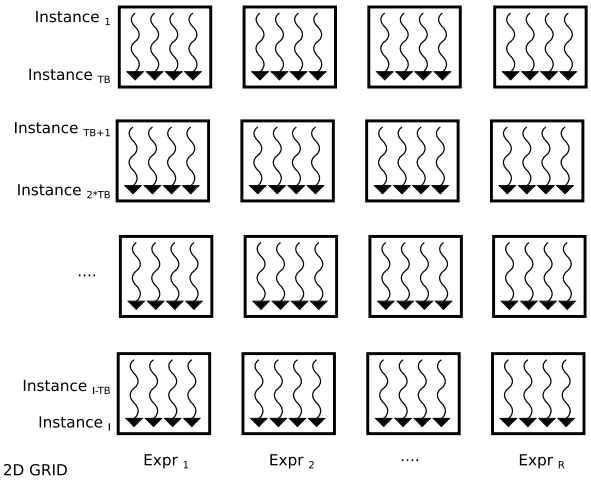

Fig. 4 Two dimensional grid of thread blocks for the coverage kernel

In the following subsections, the kernels and data structures are given in detail, considering the best way to facilitate coalescing, occupancy, and maximizing instruction throughput.

5.1 Coverage kernel

The coverage kernel interprets expressions (antecedents or consequents), which are expressed in reverse Polish notation (RPN), over the instances of the data set. The expressions interpreter is stack-based, i.e., operands are pushed onto the stack, and when an operation is performed, its operands are popped from the stack and its result pushed back on.

The kernel checks the coverage of the expressions and their conditions over the instances and stores the results in a bitset array. Each thread is responsible for the coverage of an expression over a single instance. Threads are grouped into a two dimensional grid of thread blocks, as illustrated in Fig. 4, whose size depends on the number of expressions (width) and instances (height). The number of vertical blocks depends on the number of instances and the number of threads per block, which is recommended to be a multiple of the warp size, usually being 128, 256 or 512 threads per block.

Listing 1 Coverage kernel

```
__global__ void coverageKernel(unsigned char* coverage, unsigned char* bitset) {
  int instance = blockDim.y * blockIdx.y + threadIdx.y;
  coverage[blockIdx.x * numberInstances + instance] =
          covers(expression[blockIdx.x], instance, bitset);
}
```

Table 3 Threads per block and multiprocessor occupancy

Threads per block	128	256	512
Active threads per multiprocessor	1024	1536	1536
Active warps per multiprocessor	32	48	48
Active thread blocks per multiprocessor	8	6	3
Occupancy of each multiprocessor	67 %	100 %	100 %

The selection of the optimal number of threads per block is critical in CUDA since it limits the number of active blocks and warps in a multiprocessor depending on the register pressure of the kernels. Table 3 shows the multiprocessor occupancy, which is desired to be maximized, for different block sizes over the coverage kernel. The NVIDIA CUDA compiler (nvcc) reports that the coverage kernel requires 20 registers per thread and the number of registers available per multiprocessor is limited to 32768. Therefore, 256 and 512 threads per block configurations achieve 100 % occupancy with 48 active warps and 1536 threads per multiprocessor. However, we consider that the best option is to employ 256 threads per block, since it provides more active threads blocks per multiprocessor to hide the latency arising from register dependencies and, therefore, a wider range of possibilities for the dispatcher to issue concurrent blocks to the execution units. Moreover, it provides better scalability to future GPUs with more multiprocessors capable of handling more active blocks.

To maximize global memory throughput, it is important to maximize the coalescing, by following the most optimal access patterns, using data types that meet the size and alignment requirements or padding the data arrays. For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size. Therefore, the bitset array employs intra-array padding to align the memory addresses to the memory transfer segment sizes [12, 27, 42].

The kernel must be called for by the antecedents and consequents and their executions can be overlapped and executed concurrently because they are independent. Moreover, the copy of the expressions to the GPU memory is asynchronous, i.e., the data from one stream can be copied while a kernel from another stream is executing. The code for the coverage kernel is shown in Listing 1. The *covers()* method performs the push and pop operations to interpret the expression.

5.2 Reduction kernel

The reduction kernel performs a reduction operation on the coverage bitsets of the expressions (antecedents and consequents) to compute the absolute support of the conditions. The naïve reduction operation is conceived as an iterative and sequential

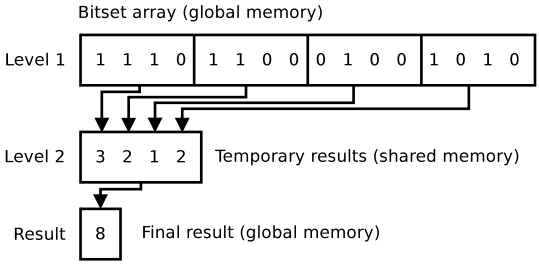

Fig. 5 2-level parallel reduction

Listing 2 Reduction kernel

```
__global__ void reductionKernel(unsigned char* bitset) {
  __shared__ int counts[128];
  counts[threadIdx.y] = 0;
  int base = blockIdx.x*gridDim.y*numberInstances +
             blockIdx.y*numberInstances + threadIdx.y;
  int top =  numberInstances - threadIdx.y;

  // Performs the first level reduction of the thread corresponding values
  for(int i = 0; i < top; i+=128)
    counts[threadIdx.y] += bitset[base + i];

  __syncthreads();

  // Performs the second level reduction of the temporary results
  if(threadIdx.y == 0) {
    for(int i = 1; i < 128; i++)
      counts[0] += counts[i];

    supportCondition[blockIdx.x*gridDim.y + blockIdx.y] = counts[0];
  }
}
```

process. However, there are many ways to perform this operation in parallel. In fact, NVIDIA provides six different ways of optimizing parallel reduction in CUDA. The one which performed best for reducing the bitsets of the expression are the 2-level parallel reduction (see Fig. 5), which illustrates a 4-threaded reduction using shared memory to compute the condition's support.

Threads within a block cooperate to reduce the bitset. Specifically, each thread is responsible for reducing a subset of the bitset, and stores its temporary result in its corresponding position of the shared memory. Finally, only one thread performs the last reduction of the temporary results from the different threads, thus obtaining the final result. The number of reduction levels and the number of threads per block determine the efficiency and cost of the reduction. More reduction levels imply more steps but reduce fewer values, and owing to the fact that the shared memory size is limited, the number of threads per block should not be too high. Thus, the implementation considers a two-level reduction using 128 threads per block, which is a trade-off solution that obtained the best experimental results in previous studies [12].

The threads are grouped into a two dimensional grid of thread blocks, whose size depends on the number of expressions (width) and number of conditions (height). Notice that the kernel must be called for by the antecedent and consequent conditions and their executions can be overlapped and executed concurrently because they are

independent. Moreover, the copying of supports back to the host memory is asynchronous, i.e., the supports for the conditions of the antecedent can be copied while the kernel is executing on the conditions of the consequent in a different stream. The reduction kernel can only be called after the completion of the coverage kernel. The code for the reduction kernel is shown in Listing 2.

5.3 Fitness kernel

The fitness kernel reduces both the bitsets of the antecedents and the consequents to calculate the contingency table. The reduction process is similar to the one from the reduction kernel, but it stores the temporary results of the contingency table in shared memory. Finally, one thread collects the temporary results of the contingency table and computes the final contingency table for the rule. The contingency table allows for computing the support, confidence, lift, leverage, conviction, and any other measure for evaluating association rules.

The fitness measures are all stored in a unique float array to perform a single but larger memory transaction back to host memory, avoiding multiple and smaller transactions, which is known to produce inefficiencies and low memory throughput due to the high latency of the peripheral component interconnect express bus (PCI-e).

The kernel is only called once and the threads are grouped into a grid of thread blocks, whose size depends on the number of rules. The fitness kernel has no dependencies with the reduction kernel and they can be executed concurrently. The code for the fitness kernel is shown in Listing 3.

6 Experimental Setup

This section presents the hardware configuration and the different experiments used to evaluate the performance of the proposed model.

6.1 Hardware configuration

The experiments were run on a machine equipped with an Intel Core i7 quad-core processor running at 3.0 GHz and 12 GB of DDR3-1600 host memory. It featured two NVIDIA GeForce 480 GTX video cards equipped with 1.5 GB of GDDR5 video RAM. The 480 GTX GPU comprised 15 multiprocessors and 480 CUDA cores clocked at 1.4 GHz. The host operating system was GNU/Linux Ubuntu 11.10 64 bit along with CUDA runtime 4.2, NVIDIA drivers 302.07, Eclipse integrated development environment 3.7.0, Java OpenJDK runtime environment 1.6-23 64 bit, and GCC compiler 4.6.3 (O2 optimization level).

6.2 Experiments

Three different experiments were carried out. Firstly, the performance of the rule interpreter was evaluated. Secondly, the efficiency of the evaluation model was analyzed on a series of real-world data sets. Finally, the performance of the serial vs. concurrent kernels model was compared. In order to make a fair comparison, both codes on CPUs and GPUs were computed by using single precision floating points.

Listing 3 Fitness kernel

```
__global__ void fitnessKernel(unsigned char* antecedentsBitset,
                              unsigned char* consequentsBitset, float* fitness) {

  __shared__ int contigencyTable[512];
  int base = blockIdx.x*numberInstances + threadIdx.y;
  int top =  numberInstances - threadIdx.y;

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

  // Performs the first level reduction of the thread corresponding values
  for(int i = 0; i < top; i+=128)
    contigencyTable[threadIdx.y*4 +
                    antecedentsBitset[base + i]*2 +
                    consequentsBitset[base + i]]++;

  __syncthreads();

  // Performs the second level reduction of the temporary results
  if(threadIdx.y < 4) {
    for(int i = 4; i < 512; i+=4) {
      contigencyTable[0] += contigencyTable[i];
      contigencyTable[1] += contigencyTable[i+1];
      contigencyTable[2] += contigencyTable[i+2];
      contigencyTable[3] += contigencyTable[i+3];
    }

    if(threadIdx.y == 0) {
      int NOTcoversAntecedentANDNOTcoversConsequent = contigencyTable[0];
      int NOTcoversAntecedentANDcoversConsequent = contigencyTable[1];
      int coversAntecedentANDNOTcoversConsequent = contigencyTable[2];
      int coversAntecedentANDcoversConsequent = contigencyTable[3];

      float A = (coversAntecedentANDcoversConsequent +
                 coversAntecedentANDNOTcoversConsequent) / numberInstances;
      float C = (coversAntecedentANDcoversConsequent +
                 NOTcoversAntecedentANDcoversConsequent) / numberInstances;

// Support
fitness[NUM_FITNESS*blockIdx.x] = coversAntecedentANDcoversConsequent /
                                  numberInstances;
// Confidence
fitness[NUM_FITNESS*blockIdx.x+1] = coversAntecedentANDcoversConsequent /
                                    (coversAntecedentANDcoversConsequent +
                                    coversAntecedentANDNOTcoversConsequent);
// Lift
fitness[NUM_FITNESS*blockIdx.x+2] = fitness[NUM_FITNESS * blockIdx.x] /
                                    (A * C);
// Leverage
fitness[NUM_FITNESS*blockIdx.x+3] = fitness[NUM_FITNESS * blockIdx.x] -
                                    (A * C);

// Conviction
fitness[NUM_FITNESS*blockIdx.x+4] = (A - A*C) /
                                    (A - fitness[NUM_FITNESS * blockIdx.x]);
    }
  }
}
```

6.2.1 Rule interpreter performance

The efficiency of rule interpreters is often reported using the number of primitives interpreted by the system per second, similarly to GP interpreters, which determine the

number of GP operations per second (GPops/s) [7, 31, 32, 34]. In GP, interpreters evaluate expression trees, which represent solutions to performing a user-defined task.

In this experimental stage, the efficiency and performance of the interpreter is evaluated by running it on different numbers of instances and rules. Hence, a sensitivity analysis of the effect of these parameters on the speed of the interpreter is achieved.

6.2.2 Evaluation model performance

A different experiment was carried out to study the performance of the complete parallelized evaluation model, including all kernel execution times and data transfer times. In this second experiment, the number of rules to be evaluated and the number of instances were increased, to analyze the scalability of the evaluation model regarding the values of the parameters mentioned above and to study its behavior when using more than one GPU. To this end, a series of data sets from the University of California Irvine (UCI) repository [17] were used. The data sets chosen for the experiment comprised a wide range in both the number of attributes and instances.

6.2.3 Serial vs. concurrent kernels

The new capabilities of modern GPUs to execute concurrent kernels is a major improvement for the efficient computation of independent kernels. Therefore, a profiling analysis was carried out of the serial and concurrent kernel execution on the GPUs. This analysis was obtained from the NVIDIA Visual Profiler tool.

7 Results

This section describes and discusses the experimental results obtained.

7.1 Rule interpreter performance

Table 4 shows interpreter execution times and performance in terms of GPops/s. The results depicted are the average results obtained over ten different executions. Here, the performance of the GPU model using one and two 480 GPUs is compared to the single-threaded and multi-threaded CPU implementation. Each row represents the case of interpreter performance when a different number of instances and rules is used. Focusing on the number of GP operations interpreted (GPops), its value depends on the number of instances, the number of rules, and the number of conditions included in each rule.

The higher the number of instances and rules to be evaluated, the larger the number of GP operations to be interpreted and, in consequence, the more execution time is required. With regard to CPU performance, the single-threaded interpreter achieves a linear performance and remains constant around at 10 million GPops/s (GP operations per second), regardless of the number of instances and rules. The multi-threaded interpreter increases performance by around 35 million GPops/s when using the four CPU cores available in the Intel i7 processor.

Table 4 RPN interpreter performance

Instances	Rules	GPops	Time (ms)				GPops/s (M)		GPops/s (B)	
			CPU	4 CPUs	1 GPU	2 GPUs	CPU	4 CPUs	1 GPU	2 GPUs
2.5×10^2	25	3.62×10^5	34	14	0.053	0.044	10.66	25.89	6.84	8.20
2.5×10^2	50	7.25×10^5	68	25	0.059	0.047	10.66	29.00	12.26	15.43
2.5×10^2	100	1.45×10^6	153	68	0.073	0.057	9.48	21.32	19.77	25.41
2.5×10^2	200	2.90×10^6	296	93	0.145	0.077	9.80	31.18	20.03	37.53
5.0×10^2	25	7.25×10^5	70	23	0.050	0.044	10.36	31.52	14.41	16.57
5.0×10^2	50	1.45×10^6	137	46	0.074	0.058	10.58	31.52	19.50	24.82
5.0×10^2	100	2.90×10^6	307	111	0.110	0.077	9.45	26.13	26.33	37.62
5.0×10^2	200	5.80×10^6	544	182	0.225	0.124	10.66	31.87	25.73	46.93
1.0×10^3	25	1.45×10^6	137	44	0.068	0.057	10.58	32.95	21.45	25.36
1.0×10^3	50	2.90×10^6	309	89	0.119	0.076	9.39	32.58	24.32	38.05
1.0×10^3	100	5.80×10^6	565	158	0.207	0.122	10.27	36.71	28.06	47.46
1.0×10^3	200	1.16×10^7	1,076	390	0.409	0.207	10.78	29.74	28.39	56.04
2.5×10^3	25	3.62×10^6	359	130	0.135	0.088	10.10	27.88	26.91	40.97
2.5×10^3	50	7.25×10^6	730	295	0.253	0.141	9.93	24.58	28.65	51.53
2.5×10^3	100	1.45×10^7	1,371	375	0.461	0.254	10.58	38.67	31.48	57.10
2.5×10^3	200	2.90×10^7	2,632	1,127	0.938	0.471	11.02	25.73	30.93	61.57
5.0×10^3	25	7.25×10^6	839	243	0.241	0.140	8.64	29.84	30.14	51.90
5.0×10^3	50	1.45×10^7	1,385	496	0.461	0.252	10.47	29.23	31.45	57.55
5.0×10^3	100	2.90×10^7	2,602	921	1.102	0.476	11.15	31.49	26.30	60.86
5.0×10^3	200	5.80×10^7	5,699	2,214	1.824	0.912	10.18	26.20	31.79	63.57
1.0×10^4	25	1.45×10^7	1,442	484	0.459	0.249	10.06	29.96	31.59	58.30
1.0×10^4	50	2.90×10^7	2,689	1,010	0.901	0.471	10.78	28.71	32.20	61.56
1.0×10^4	100	5.80×10^7	5,392	1,651	1.791	0.912	10.76	35.13	32.39	63.59
1.0×10^4	200	1.16×10^8	11,010	3,971	3.580	1.799	10.54	29.21	32.40	64.48
2.5×10^4	25	3.62×10^7	3,528	1,318	1.099	0.581	10.27	27.50	32.97	62.38
2.5×10^4	50	7.25×10^7	6,535	2,419	2.180	1.100	11.09	29.97	33.25	65.91
2.5×10^4	100	1.45×10^8	14,086	4,421	4.350	2.306	10.29	32.80	33.34	62.88
2.5×10^4	200	2.90×10^8	28,006	8,983	8.699	4.367	10.35	32.28	33.34	66.40
5.0×10^4	25	7.25×10^7	7,902	2,450	2.177	1.141	9.17	29.59	33.30	63.56
5.0×10^4	50	1.45×10^8	14,677	4,694	4.341	2.177	9.88	30.89	33.41	66.61
5.0×10^4	100	2.90×10^8	28,688	11,411	8.679	4.350	10.11	25.41	33.41	66.67
5.0×10^4	200	5.80×10^8	55,528	20,762	17.470	8.688	10.45	27.94	33.20	66.76
1.0×10^5	25	1.45×10^8	13,749	4,889	4.382	2.257	10.55	29.66	33.09	64.26
1.0×10^5	50	2.90×10^8	28,609	9,662	8.757	4.422	10.14	30.01	33.11	65.58
1.0×10^5	100	5.80×10^8	56,188	17,125	17.402	8.740	10.32	33.87	33.33	66.36
1.0×10^5	200	1.16×10^9	110,195	31,418	35.100	17.462	10.53	36.92	33.05	66.43
2.5×10^5	25	3.62×10^8	36,032	10,682	10.766	5.607	10.06	33.94	33.67	64.65
2.5×10^5	50	7.25×10^8	69,527	21,163	21.990	10.911	10.43	34.26	32.97	66.45
2.5×10^5	100	1.45×10^9	133,125	42,765	43.573	22.036	10.89	33.91	33.28	65.80
2.5×10^5	200	2.90×10^9	269,071	84,591	86.747	43.604	10.78	34.28	33.43	66.51
5.0×10^5	25	7.25×10^8	67,989	20,980	21.512	11.194	10.66	34.56	33.70	64.77
5.0×10^5	50	1.45×10^9	137,671	40,351	43.057	21.537	10.53	35.93	33.68	67.33
5.0×10^5	100	2.90×10^9	291,238	81,028	86.703	43.147	9.96	35.79	33.45	67.21
5.0×10^5	200	5.80×10^9	602,719	170,074	172.970	86.480	9.62	34.10	33.53	67.07
1.0×10^6	25	1.45×10^9	144,987	43,357	43.001	22.396	10.00	33.44	33.72	64.74
1.0×10^6	50	2.90×10^9	285,639	84,678	86.248	43.009	10.15	34.25	33.62	67.43
1.0×10^6	100	5.80×10^9	560,459	170,977	172.258	86.129	10.35	33.92	33.67	67.34
1.0×10^6	200	1.16×10^{10}	1,097,670	332,387	344.908	172.681	10.57	34.90	33.63	67.18

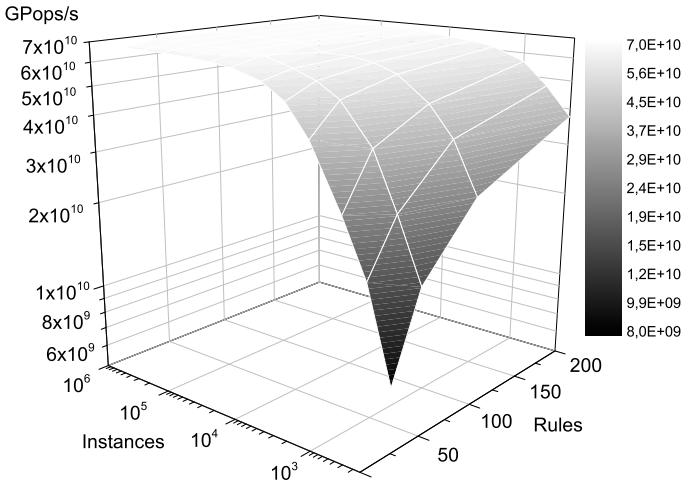

Fig. 6 RPN interpreter performance using 2 × 480 GPUs

The GPU implementation achieves high performance regardless of the number of rules and instances to be evaluated. The larger the number of rules and instances, the larger the number of thread blocks to compute consequently, the higher occupancy of the GPU multiprocessors. Using one 480 GPU, the limit is reached at 33 billion GPops/s. On the other hand, working with two 480 GPUs, the limit is reached at 67 billion GPops/s, demonstrating the great scalability of the model in the number of GPU devices and the complexity of the problem. In this way, unquestionable performance is obtained, which takes only 0.172 seconds to interpret 200 rules over one million instances. In other words, the GPU implementation using two 480 GPUs requires less than two deciseconds to carry out more than 11 billion GP operations. All the results are summarized in Fig. 6, which depicts the RPN performance of two 480 GPUs in terms of GPops/s with regard to the number of instances and rules.

7.2 Evaluation model performance

Table 5 shows execution times and the speed-up obtained for ten different real-world data sets. The results depicted are the average results obtained over ten different executions. Each row represents the execution time and the speed-up achieved when evaluating a series of rules over a group of data sets having different number of instances and attributes.

As far as the CPU evaluation model is concerned, it is shown that execution time increases linearly with the number of instances and rules to be evaluated (as is theoretically to be expected). Therefore, evaluating association rules over data sets having many instances becomes unmanageable, especially if the evaluation is embedded within an evolutionary algorithm, since it has to evaluate the population of each generation.

With regard to the GPU evaluation model, the execution times present few variations as the number of instances and rules increases. When using two 480 GPUs,

Table 5 UCI data sets evaluation performance

Data set	Rules	Evaluation time (ms)				Speedup vs. CPU			Speedup vs. 4 CPUs	
		CPU	4 CPUs	1 GPU	2 GPUs	4 CPUs	1 GPU	2 GPUs	1 GPU	2 GPUs
WDBC	25	111	39	5	3	2.85	22.20	37.00	7.80	13.00
Instances: 569	50	251	86	6	4	2.92	41.83	62.75	14.33	21.50
Attributes: 31	100	516	184	11	7	2.80	46.91	73.71	16.73	26.29
	200	785	406	13	7	1.93	60.38	112.14	31.23	58.00
Abalone	25	556	233	5	4	2.39	111.20	139.00	46.60	58.25
Instances: 4,174	50	1,158	450	11	9	2.57	105.27	128.67	40.91	50.00
Attributes: 9	100	2,250	748	21	14	3.01	107.14	160.71	35.62	53.43
	200	4,083	1,676	30	19	2.44	136.10	214.89	55.87	88.21
Spambase	25	796	282	9	4	2.82	88.44	199.00	31.33	70.50
Instances: 4,597	50	1,460	558	14	8	2.62	104.29	182.50	39.86	69.75
Attributes: 58	100	3,120	1,146	21	15	2.72	148.57	208.00	54.57	76.40
	200	6,326	2,140	35	20	2.96	180.74	316.30	61.14	107.00
Page-blocks	25	718	243	9	6	2.95	79.78	119.67	27.00	40.50
Instances: 5,472	50	1,715	524	17	15	3.27	100.88	114.33	30.82	34.93
Attributes: 11	100	3,354	1,068	22	20	3.14	152.45	167.70	48.55	53.40
	200	7,127	2,098	47	26	3.40	151.64	274.12	44.64	80.69
Thyroid	25	1,066	349	9	5	3.05	118.44	213.20	38.78	69.80
Instances: 7,200	50	2,348	710	16	10	3.31	146.75	234.80	44.38	71.00
Attributes: 22	100	4,386	1,493	28	18	2.94	156.64	243.67	53.32	82.94
	200	9,269	3,451	51	26	2.69	181.75	356.50	67.67	132.73

Table 5 (*Continued*)

Data set	Rules	Evaluation time (ms)				Speedup vs. CPU			Speedup vs. 4 CPUs	
		CPU	4 CPUs	1 GPU	2 GPUs	4 CPUs	1 GPU	2 GPUs	1 GPU	2 GPUs
Magic	25	2,812	1,045	16	17	2.69	175.75	165.41	65.31	61.47
Instances: 19,020	50	5,593	2,044	31	26	2.74	180.42	215.12	65.94	78.62
Attributes: 11	100	11,675	4,301	61	36	2.71	191.39	324.31	70.51	119.47
	200	24,968	9,452	115	73	2.64	217.11	342.03	82.19	129.48
House_16H	25	3,323	1,115	33	21	2.98	100.70	158.24	33.79	53.10
Instances: 22,784	50	7,390	2,119	59	34	3.49	125.25	217.35	35.92	62.32
Attributes: 17	100	13,318	4,234	100	60	3.15	133.18	221.97	42.34	70.57
	200	27,595	9,296	182	100	2.97	151.62	275.95	51.08	92.96
Shuttle	25	9,519	3,151	51	28	3.02	186.65	339.96	61.78	112.54
Instances: 58,000	50	19,051	6,308	96	56	3.02	198.45	340.20	65.71	112.64
Attributes: 10	100	38,535	11,822	154	91	3.26	250.23	423.46	76.77	129.91
	200	78,674	23,916	301	173	3.29	261.38	454.76	79.46	138.24
Kddcup	25	75,835	18,452	368	281	4.11	206.07	269.88	50.14	65.67
Instances: 494,020	50	142,892	36,280	684	446	3.94	208.91	320.39	53.04	81.35
Attributes: 42	100	310,216	73,164	1,502	931	4.24	206.54	333.21	48.71	78.59
	200	622,942	152,385	2,815	1,550	4.09	221.29	401.90	54.13	98.31
Poker	25	154,548	41,700	691	436	3.71	223.66	354.47	60.35	95.64
Instances: 1,025,010	50	337,268	100,278	1,504	915	3.36	224.25	368.60	66.67	109.59
Attributes: 11	100	735,638	219,419	3,137	1,974	3.35	234.50	372.66	69.95	111.15
	200	1,465,010	414,267	6,180	3,397	3.54	237.06	431.27	67.03	121.95

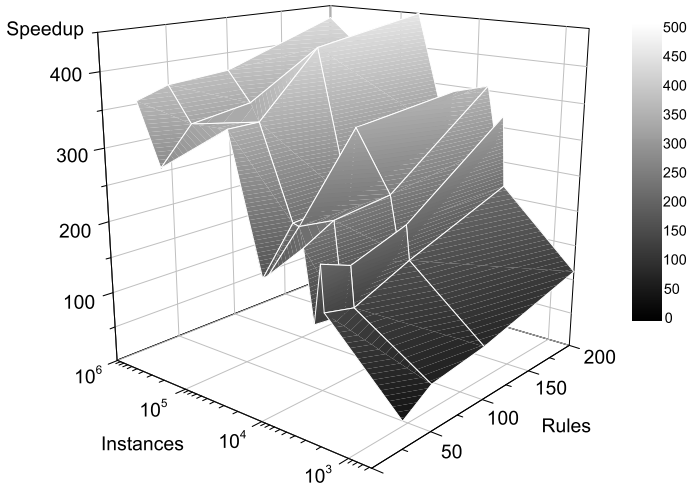

Fig. 7 GPU evaluation speed-up using 2 × 480 GPUs

execution time is less than one second in most of the evaluation cases. In extreme cases, the evaluation model requires only three seconds to evaluate 200 rules over a data set comprising more than one million instances. The scalability of the GPU model in the problem complexity and in the number of GPU devices is maintained.

Finally, focusing on the speed-up achieved when working with one 480 GPU (see Table 5), it is shown that the speed-up reaches its maximum at values of around 200. This maximum speed-up is obtained for data sets having more than 50 thousand instances. On the other hand, working with two 480 GPUs, the evaluation model achieves its maximum speed-up at values around 400×, doubling the performance of a single 480 GPU. The maximum speed-up is obtained when all the GPU multiprocessors are fully occupied, so using multiple GPUs allows for working on larger data sets with a higher number of instances and better speed-ups to be achieved. All the results are summarized in Fig. 7, depicting the speed-ups obtained by two 480 GPUs as a function of the number of rules and instances of the data sets.

NVIDIA Parallel Nsight software has been used to debug and profile the GPU implementation, allowing for the CPU/GPU execution timeline to be traced and analyzed. The results reported by the software demonstrate the efficiency of the implementation using shared memory effectively and avoiding bank conflicts, while providing fully coalesced memory accesses. The software also enables the concurrent execution of the kernels and asynchronous data transfers to be monitored. The throughput of data transfer between host and device memory has been reported to achieve 6 GB/s, which is the maximum speed of the PCI-e bus 2.0.

7.3 Serial vs. concurrent kernels

As mentioned, there are some independencies among the kernels that enables their execution to be efficiently overlapped. Concurrent kernels are issued by means of CUDA streams, which also allow for asynchronous data transfers. Figure 8 shows

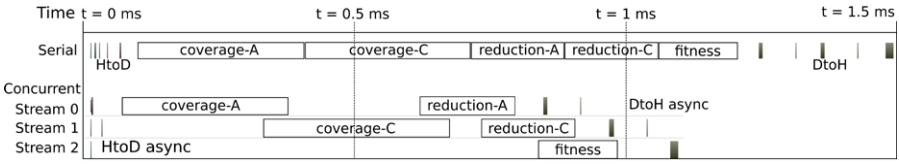

Fig. 8 GPU timeline from NVIDIA Visual Profiler developer tool

Table 6 Runtime of kernels and data transfers

HtoD mem	Coverage	Reduction	Fitness	DtoH mem	Total
11.456 µs	0.303 ms (×2)	0.169 ms (×2)	0.143 ms	56.906 µs	1.49 ms (serial) 1.08 ms (concurrent)

the performance profiling of the serial and concurrent kernel execution on the GPUs obtained from the NVIDIA Visual Profiler developer tool. The timeline clearly shows the runtime saved due to concurrent execution and data transfers. First, there are host to device (HtoD) memory transfers, which copy the antecedent and the consequent of the rules to the GPU memory. These transfers can be issued in different streams concurrently in the case of asynchronous transfers (HtoD async). Second, the coverage kernel is executed on both the antecedent and the consequent. Their execution is overlapped as soon as there are GPU resources available. Third, the reduction kernel is executed, respectively. Four, the fitness kernel is executed and overlapped with the reduction kernel, since it only depends on the coverage kernel being completed. Finally, memory transfers copy back results and fitness measures to the host memory (DtoH). In the case of streams, these transactions are concurrent with kernel executions. Table 6 shows the runtime for each of the kernels and for data transfers. The concurrent kernels and asynchronous transfers model reduces runtime significantly from 1.49 ms to 1.08 ms, which is about 27 % faster.

8 Concluding remarks

Association rule mining was conceived as an unsupervised learning task for finding close relationships between items in large data sets. However, with growing interest in the storage of information, high dimensionality data sets have been appearing, giving rise to high computational and memory requirements. This computational cost problem is mainly due to the evaluation process, where each condition within each rule has to be evaluated for each instance.

This paper describes the use of GPUs for evaluating association rules employing a novel and efficient model which may be applied to any association rule mining approach. It allows for the mined rules to be evaluated in parallel, thus reducing computational time. The proposal takes advantage of the concurrent execution of kernels and asynchronous data transfers to improve the efficiency and occupancy of the GPU.

The results of our experiments demonstrate the performance of the interpreter and the efficiency of the evaluation model by comparing the runtimes of the single-threaded and multi-threaded CPU with those of the GPU. The results of the GPU model have shown an interpreter performance above 67 billion giga genetic programming operations per second. As far as on the speed-up achieved over the CPU evaluation model, a value of 454× was obtained when using two NVIDIA 480 GTX GPUs. The experimental study demonstrated the efficiency of this evaluation model and its scalability in problem complexity, number of instances, rules, and GPU devices.

Acknowledgements This work was supported by the Regional Government of Andalusia and the Ministry of Science and Technology, projects P08-TIC-3720 and TIN-2011-22408, and FEDER funds. This research was also supported by the Spanish Ministry of Education under FPU grants AP2010-0042 and AP2010-0041.

References

1. NVIDIA CUDA Programming, Best practices guide, http://www.nvidia.com/cuda (2013)
2. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969
3. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of 20th international conference on very large data bases, pp 487–499
4. Alatas B, Akin E (2006) An efficient genetic algorithm for automated mining of both positive and negative quantitative association rules. Soft Comput 10:230–237
5. Alba E, Tomassini M (2002) Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 6(5):443–462
6. Bai H, Ouyang D, He L (2009) GPU-based frequent pattern mining algorithm. Chin J Sci Instrum 30(10):2082–2087
7. Banzhaf W, Harding S, Langdon WB, Wilson G (2009) Accelerating genetic programming through graphics processing units. In: Genetic programming theory and practice. VI. Genetic and evolutionary computation, pp 1–19
8. Berzal F, Blanco I, Sánchez D, Vila M (2002) Measuring the accuracy and interest of association rules: a new framework. Intell Data Anal 6(3):221–235
9. Borgelt C (2003) Efficient implementations of Apriori and Eclat. In: Proceedings of the 1st workshop on frequent itemset mining implementations
10. Brin S, Motwani R, Silverstein C (1997) In: Beyond market baskets: generalizing association rules to correlations (SIGMOD '97), pp 265–276
11. Brin S, Motwani R, Ullman J, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 255–264
12. Cano A, Zafra A, Ventura S (2012) Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Comput 16:187–202
13. Cecilia JM, Nisbet A, Amos M, García JM, Ujaldón M (2013) Enhancing GPU parallelism in nature-inspired algorithms. J Supercomput 63(1–17):773–789
14. Cui Q, Guo X (2013) Research on parallel association rules mining on GPU. In: Proceedings of the international conference on green communications and networks. Lecture notes in electrical engineering, vol 224, pp 215–222
15. Fok KL, Wong TT, Wong ML (2007) Evolutionary computing on consumer graphics hardware. IEEE Intell Syst 22(2):69–78
16. Franco MA, Krasnogor N, Bacardit J (2010) Speeding up the evaluation of evolutionary learning systems using GPGPUs. In: Proceedings of the genetic and evolutionary computation conference, pp 1039–1046
17. Frank A, Asuncion A (2010) In: UCI machine learning repository
18. Garland M, Le Grand S, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V (2008) Parallel computing experiences with CUDA. IEEE MICRO 28(4):13–27

19. George T, Nathan M, Wagner M, Renato F (2010) Tree projection-based frequent itemset mining on multi-core CPUs and GPUs. In: Proceedings of the 22nd international symposium on computer architecture and high performance computing, pp 47–54
20. Gray B, Orlowska M (1998) CCAIIA: clustering categorical attributes into interesting association rules. In: Research and development in knowledge discovery and data mining. Lecture notes in computer science, vol 1394, pp 132–143
21. Green RC II, Wang L, Alam M, Formato RA (2012) Central force optimization on a GPU: a case study in high performance metaheuristics. J Supercomput 62(1):378–398
22. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min Knowl Discov 8:53–87
23. Harding S, Banzhaf W (2007) Fast genetic programming on GPUs. Lect Notes Comput Sci 4445:90–101
24. Hoai RI, Whigham NX, Shan PA, O'Neill Y, McKay M (2010) Grammar-based genetic programming: a survey. Genet Program Evol Mach 11(3–4):365–396
25. Hu J, Yang-Li X (2008) A fast parallel association rules mining algorithm based on FP-forest. In: Proceedings of the 5th international symposium on neural networks: advances in neural networks, part II. Lecture notes in computer science, vol 5264, pp 40–49
26. Hwu WW (2009) Illinois ECE 498AL: Programming massively parallel processors, Lecture 13: Reductions and their implementation
27. Jang B, Schaa D, Mistry P, Kaeli D (2011) Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans Parallel Distrib Syst 23(1):105–118
28. Jian L, Wang C, Liu Y, Liang S, Yi W, Shi Y (2011) Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA). The Journal of Supercomputing, 1–26
29. Lin K-C, Liao I-E, Chen Z-S (2011) An improved frequent pattern growth method for mining association rules. Expert Syst Appl 38:5154–5161
30. Klosgen W (1996) Explora: a multipattern and multistrategy discovery assistant. In: Advances in knowledge discovery and data mining, pp 249–271
31. Langdon WB (2010) A many threaded CUDA interpreter for genetic programming. Lect Notes Comput Sci 6021:146–158
32. Langdon WB (2011) Graphics processing units and genetic programming: an overview. Soft Comput 15(8):1657–1669
33. Langdon WB (2011) Performing with CUDA. In: Proceedings of the genetic and evolutionary computation conference, pp 423–430
34. Langdon WB, Banzhaf W (2008) A SIMD interpreter for genetic programming on GPU graphics cards. Lect Notes Comput Sci 4971:73–85
35. Langdon WB, Harrison AP (2008) GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Comput 12(12):1169–1183
36. Luna JM, Romero JR, Ventura S (2010) G3PARM: a grammar guided genetic programming algorithm for mining association rules. In: Proceedings of the 2010 IEEE world Congress on computational intelligence, pp 2586–2593
37. Luna JM, Romero JR, Ventura S (2012) Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. Knowl Inf Syst 32(1):53–76
38. Ordoñez C, Ezquerra N, Santana C (2006) Constraining and summarizing association rules in medical data. Knowl Inf Syst 9(3):259–283
39. Pallipuram VK, Bhuiyan M, Smith MC (2012) A comparative study of GPU programming models and architectures using neural networks. J Supercomput 61(3):618–673
40. Papè NF, Alcalá-Fernandez J, Bonarini A, Herrera F (2009) Evolutionary extraction of association rules: a preliminary study on their effectiveness. Lect Notes Comput Sci 5572:646–653
41. Piatetsky-Shapiro G (1991) Discovery, analysis, and presentation of strong rules
42. Rivera G, Tseng CW (1998) Data transformations for eliminating conflict misses. ACM SIGPLAN Not 33(5):38–49
43. Romero C, Luna JM, Romero JR, Ventura S (2011) RM-tool: a framework for discovering and evaluating association rules. Adv Eng Softw 42(8):566–576
44. Sánchez D, Serrano JM, Cerda L, Vila MA (2008) Association rules applied to credit card fraud detection. Expert Syst Appl 36(2):3630–3640
45. Silverstein C, Brin S, Motwani R (1998) Beyond market baskets: generalizing association rules to dependence rules. Data Min Knowl Discov 2(1):39–68

46. Wu XL, Obeid N, Hwu WM (2010) Exploiting more parallelism from applications having generalized reductions on GPU architectures. In: Proceedings of the 10th IEEE international conference on computer and information technology, pp 1175–1180
47. Zhang C, Zhang S (2002) Association rules mining: models and algorithms. Lecture notes in computer science, vol 2307. Springer, Berlin
48. Zhou J, Yu KM, Wu BC (2010) Parallel frequent patters mining algorithm on GPU. In: Proceedings of the IEEE international conference on systems, man and cybernetics, pp 435–440