Programación Automática con Colonias de Hormigas Multi-Objetivo en GPUs

Alberto Cano, Juan Luis Olmo, y Sebastián Ventura

Departamento de Informática y Análisis Numérico,
Universidad de Córdoba, España
{acano,jlolmo,sventura}@uco.es

Abstract. La tarea de clasificación ha sido abordada recientemente con éxito mediante el paradigma de programación automática con hormigas. Esta tarea de minería de datos demanda grandes recursos computacionales, especialmente cuando se abordan conjuntos de elevada dimensionalidad. Este trabajo presenta un modelo de paralelización de un algoritmo de programación automática con hormigas multi-objetivo para clasificación, usando tarjetas gráficas (GPUs) y el modelo de programación NVIDIA CUDA. El modelo es escalable a múltiples GPUs y a conjuntos de gran tamaño. Se realiza un estudio experimental donde se evalúa el rendimiento y la eficiencia del modelo comparando los tiempos de ejecución de la implementación en CPU y en GPU. Los resultados experimentales nos permiten analizar la escalabilidad y eficiencia del modelo con respecto a la complejidad del conjunto de datos y el tamaño de la población del algoritmo. El modelo en GPU demuestra un gran rendimiento y escalabilidad en conjuntos de datos con un elevado número de instancias, logrando una aceleración de alrededor de 250 $\times$ cuando se comparan los tiempos de ejecución del algoritmo en 4 GPUs con la versión multi-hilo en CPU. El gran rendimiento obtenido en GPU nos permite ampliar la aplicabilidad del algoritmo a conjuntos de datos sensiblemente más complejos, que previamente eran inmanejables en un tiempo razonable.

Palabras clave: Minería de datos, programación automática con colonias de hormigas, paralelización en GPU

1 Introducción

La tarea de clasificación es una tarea de aprendizaje automático supervisado consistente en predecir la clase de ejemplos no etiquetados, cuya clase es desconocida, utilizando las propiedades de los ejemplos de entrenamiento empleados para aprender previamente un modelo, y cuya etiqueta de clase era conocida.

El problema de la dimensionalidad es uno de los mayores retos en la aplicación de algoritmos basados en optimización mediante colonias de hormigas (ACO) [1] debido al elevado tiempo computacional que requieren. Además, centrándonos en la tarea de clasificación, la inducción de clasificadores resulta cada vez más complicada, debido al hecho de que las capacidades de generación y recolección de datos en dominios reales crecen a un ritmo exponencial. De hecho, puede que no sea posible ejecutar ciertos algoritmos secuenciales sobre determinados conjuntos de datos. Por este motivo, resulta

crucial diseñar algoritmos paralelos capaces de manejar estas ingentes cantidades de datos [2, 3]. Recientemente se ha proporcionado una taxonomía de las estrategias de paralelización de los algoritmos de ACO, que distingue entre modelos maestro-esclavo, celular, de ejecuciones independientes paralelas, multicolonia e híbrido [4]. Por otro lado, en lo que respecta a las arquitecturas que se han utilizado hasta la fecha para paralelizar algoritmos de ACO, destacan las plataformas de clúster como opción más empleada, seguidas por los multiprocesadores y los computadores masivamente paralelas [4]. Recientemente, otras opciones van ganando atención, como la computación en grid, los servidores multinúcleo y las unidades de procesamiento gráfico (GPUs) [5, 6].

Las GPUs son dispositivos con unidades de procesamiento masivamente paralelo, que proporcionan hardware paralelo muy rápido por un precio mucho más económico que los sistemas paralelos tradicionales. De hecho, desde la introducción de la arquitectura CUDA en 2007, multitud de investigadores han aprovechado el poder de la GPU para computación de propósito general (GPGPU) [7, 8]. Concretamente, se ha estudiado la aplicación de GPGPU para acelerar algoritmos de computación evolutiva [9, 10] y metaheurísticas [11]. Debido a las ventajas que ofrece la GPGPU, en este trabajo pretendemos explorar el rendimiento de un algoritmo multiobjetivo de clasificación basado en programación con hormigas, denominado MOGBAP [12], paralelizado mediante GPUs [13], comparándolo con su versión paralela multihilo. El estudio experimental analiza el rendimiento y escalabilidad del algoritmo desde problemas más simples a más complejos, incrementando el número de instancias y atributos. Los resultados demuestran que su paralelización permite extender la aplicación de MOGBAP a dominios que presentan conjuntos de datos con enormes cantidades de instancias y atributos, donde antes era extremadamente complicado ejecutar el algoritmo.

El artículo se organiza como sigue. En la siguiente sección se introducirá el algoritmo multiobjetivo para clasificación en que se centra este trabajo, para pasar a discutir la paralelización del mismo y finalizar presentando la implementación GPU. La Sección 3 describe el estudio experimental. Los resultados obtenidos se presentan en la Sección 4. Por último, en la Sección 5, se presentan las conclusiones del trabajo.

2 Algoritmo GPU-MOGBAP (GPU Multi-Objective Grammar Based Ant Programming)

El algoritmo MOGBAP, acrónimo de *Multi-Objective Grammar-Based Ant Programming*, es un algoritmo de programación automática multiobjetivo para el problema de clasificación multiclase [12]. Como entrada, recibe un conjunto de datos de entrenamiento, y construye un clasificador compuesto de reglas *IF-THEN* y que actúa como lista de decisión donde las reglas descubiertas se ordenan de forma descendente en base a su *fitness*. Así, a la hora de predecir la clase para una nueva instancia del conjunto de test, la clase asignada corresponderá con el consecuente de la primera regla del clasificador que se dispare, que será aquella cuyo antecedente cubra la nueva instancia.

El algoritmo MOGBAP se basa en el uso de una gramática de contexto libre que restringe el espacio de búsqueda y que asegura que cualquier individuo generado sea sintácticamente válido [14]. En lo concerniente a la codificación de los individuos, cada hormiga representará una regla de clasificación. La creación de una nueva hormiga

seguirá la aplicación de la regla de transición desde el estado inicial del entorno hasta alcanzar un estado final, entorno que adopta la forma de un árbol de derivación.

El algoritmo MOGBAP presenta también una estrategia multiobjetivo novedosa para la tarea de clasificación en el sentido de que descubre un frente de Pareto separado para cada clase del conjunto de datos, presuponiendo que determinadas clases van a ser más difíciles de predecir que otras. De hecho, si en lugar de separar los individuos por clase, se les asignase un rango a todos juntos de acuerdo a la dominancia de Pareto, es muy posible que ocurriera solapamiento. La estrategia se puede resumir como sigue. Una vez que los individuos de la generación actual se han creado y evaluado por cada objetivo considerado, se dividen en k grupos, uno por cada clase del conjunto de entrenamiento. Entonces, cada grupo de individuos se combina con las soluciones almacenadas previamente en el frente de Pareto correspondiente a esa clase en la generación anterior, para ordenarlos de acuerdo a la dominancia de Pareto, encontrando así un nuevo frente de Pareto para cada clase. Por tanto, existirán k frentes de Pareto, y sólo las soluciones no dominadas participarán en el refuerzo de feromonas sobre el entorno. Por último, el clasificador final se compondrá a partir de los individuos no dominados que se han almacenado en cada uno de los k frentes de Pareto descubiertos en la última generación del algoritmo. Para seleccionar las reglas del clasificador apropiadamente, se lleva a cabo sobre cada frente un enfoque de nichos para seleccionar los individuos del mismo que se incluirán en el clasificador.

2.1 Versión paralelizada con multihilo

En esta sección comentaremos las fases del algoritmo MOGBAP que se han paralelizado utilizando multihilo. Esta versión paralela en CPU del algoritmo se ha diseñado para ejecutar los hilos utilizando el *Java ExecutorService*, que define un *pool* de hilos de forma dinámica teniendo en cuenta el número de procesadores disponibles. Así, la escalabilidad del algoritmo con respecto al número de procesadores se lleva a cabo automáticamente sin supervisión por parte del usuario.

El esquema de esta versión paralela de MOGBAP se ajusta a la estructura jerárquica maestro-esclavo definida por Pedemonte et al. [4], dado que un proceso maestro gestiona las estructuras de información globales y delega tareas en los procesos esclavos. Dichas tareas son la generación de individuos, su evaluación, el descubrimiento de los frentes de Pareto y el procedimiento de nichos, comunicando los resultados de vuelta al proceso maestro. A continuación se justifica la paralelización de cada fase.

La fase de inicialización de MOGBAP se encarga de inicializar la gramática a utilizar a partir de los metadatos existentes en el conjunto de entrenamiento, inicializando, asimismo, la estructura de datos que albergará el espacio de estados. Dicha estructura sigue un enfoque de construcción incremental, de manera que cada hormiga que se cree en cualquier iteración almacenará sus estados visitados en dicha estructura sólo en caso de que no hayan sido insertados previamente. La fase de creación de hormigas se puede paralelizar mediante multihilo, ya que las hormigas se pueden crear simultáneamente, al no influir el camino seguido por una hormiga en el que vaya a seguir otra en la misma generación. Sin embargo, hay que tener en cuenta que el almacenamiento de los estados visitados en la estructura de datos del espacio de estados no se puede delegar en los hilos directamente, al tratarse de una estructura común que no se puede modificar

concurrentemente. Por este motivo, la actualización de dicha estructura no se puede hacer en paralelo y debe ser efectuada por el proceso maestro, que recibirá los caminos seguidos por cada hormiga generada.

La fase de evaluación comprueba las instancias que cubre cada hormiga y calcula los valores de *fitness* de cada objetivo basándose en la matriz de confusión. La implementación paralela con multihilo es elemental.

En cuanto a la estrategia multiobjetivo, el descubrimiento de cada frente de Pareto a partir de un conjunto de individuos no se puede paralelizar, ya que es necesario seguir un procedimiento secuencial para descubrir las relaciones de dominancia entre los mismos. No obstante, ya que la dominancia de Pareto se evalúa por grupos de individuos, uno por clase, es posible utilizar hilos, de forma que se emplee un hilo para calcular la dominancia en un grupo de individuos determinado.

Las actividades de refuerzo, evaporación y normalización las realiza el proceso maestro, al requerir el acceso a la estructura de datos del espacio de estados y, por tanto, no se puede paralelizar (los accesos para escritura no pueden ser concurrentes).

El enfoque de nichos que se lleva a cabo sobre los individuos de cada frente de Pareto se puede paralelizar utilizando múltiples hilos, y los individuos resultantes se devuelven al proceso maestro, que los ordena para construir el clasificador.

2.2 Versión paralelizada con GPU

En esta sección se presenta la implementación en GPU del algoritmo MOGBAP. Antes de centrarnos en la paralelización mediante GPUs de la fase de evaluación, pasamos a justificar el motivo por el cual el resto de fases no se paralelizan también en GPU.

La inicialización de la gramática y el espacio de estados no se paralelizó utilizando multihilo y, del mismo modo, no es posible hacerlo con GPU. La paralelización de la fase de creación de los individuos tampoco se efectuó en GPU, manteniéndola en multihilo, ya que la transferencia de la estructura de datos utilizada por el espacio de estados a la GPU no es apropiada. De hecho, no es posible conseguir una paralelización interna más allá de la obtenida mediante la creación de cada hormiga en su hilo correspondiente, ya que no consiste más que en seleccionar una serie de transiciones desde el estado inicial a uno final, lo cual se hace secuencialmente. Así pues, los costes de transferencia sobrepasarían claramente los beneficios de implementar esta fase en GPU, además de la complejidad inherente a dicha implementación, que está controlada por una gramática y por el número de derivaciones disponibles en cada momento. La implementación paralela en CPU utilizando multihilo es, en cambio, simple y eficiente.

La estrategia multiobjetivo se paralelizó utilizando multihilo, como se explicó en la Sección 2.1. No es posible paralelizar este paso más allá, debido al hecho de que localizar las relaciones de dominancia de Pareto en un conjunto de individuos es un proceso secuencial. Para el procedimiento de nichos se puede argumentar la misma justificación, ya que internamente procede ordenando los individuos de acuerdo a su *fitness* y efectuando una secuencia serial de operaciones siguiendo dicho orden. Por último, el proceso maestro es el encargado de llevar a cabo las fases de refuerzo, evaporación, normalización y la construcción del clasificador, como se explicó con anterioridad, por lo que no se pueden paralelizar.

Por otro lado, se ha demostrado que la paralelización de la fase de evaluación en algoritmos bioinspirados mediante GPUs obtiene un gran rendimiento [15, 16]. En nuestro caso, la evaluación de una hormiga no depende de la evaluación del resto de hormigas. Es más, el proceso de interpretar la regla codificada por una hormiga y comprobar si cubre una determinada instancia o no es también independente de las reglas y las instancias. Por todo ello, cada hilo del modelo en GPU se encarga de interpretar la regla de una hormiga sobre una instancia. La Figura 1 muestra el diagrama de flujo del modelo de GPU donde la fase de evaluación en la GPU se lleva a cabo empleando dos funciones de núcleo o *kernel*.

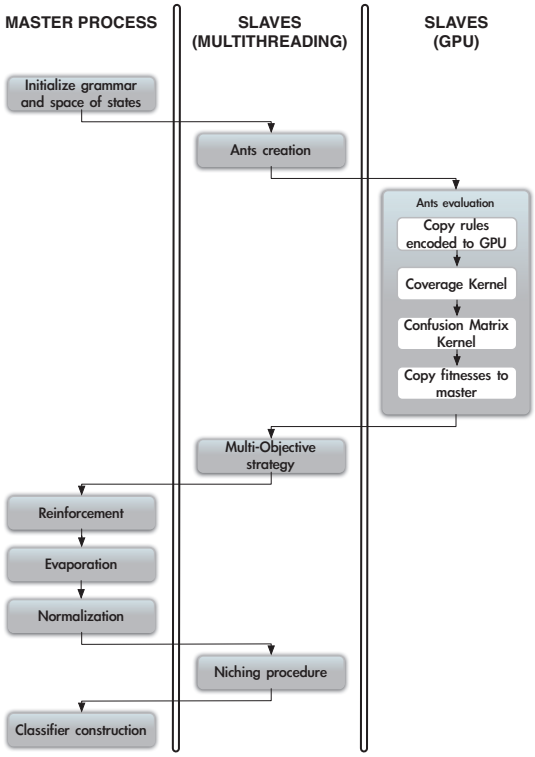

Figura 1. Computational flow chart of the GPU version

***Kernel* de cubrimiento** El *kernel* de cubrimiento interpreta las reglas, que se expresan en notación polaca inversa, sobre las instancias del conjunto de datos. El *kernel* se muestra en la parte superior de la Figura 2 y se ejecuta utilizando un grid de bloques de hilos en 2D. La longitud de la primera dimensión viene dada por el número de reglas. La longitud de la segunda depende del número de instancias y del número de hilos por bloque. Por tanto, el total de hilos en el grid será el producto de ambas dimensiones.

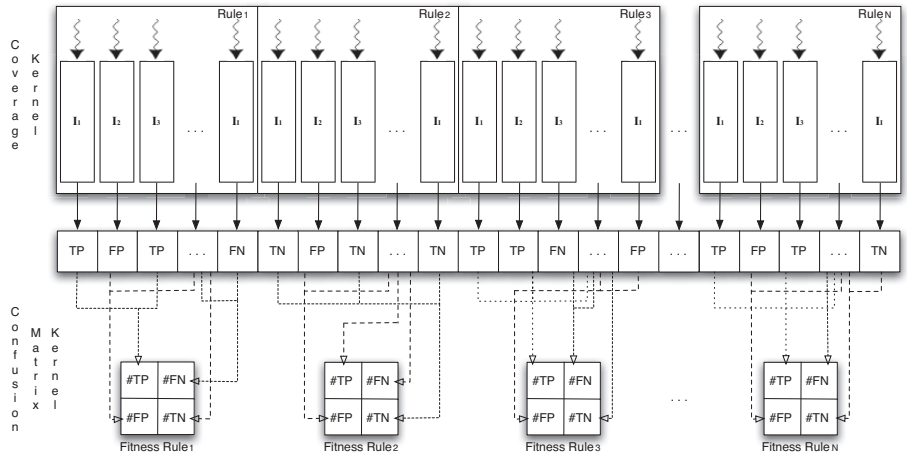

Figura 2. *Kernels* en la GPU

Los kernels se diseñan de modo que los hilos acceden a posiciones de memoria global alineadas para alcanzar un rendimiento óptimo. Así, los hilos solicitan direcciones de memoria consecutivas que se pueden servir en un menor número de transacciones de memoria. Las reglas se alojan en la memoria de tipo constante a modo de multidifusión para todos los hilos en un grupo. Todos los hilos del grupo evalúan la misma regla, pero sobre instancias diferentes. Por este motivo se sigue el nivel de paralelismo de una instrucción, múltiples datos, que se consigue cuando cada procesador realiza la misma tarea sobre instancias diferentes en datos distribuidos [17].

La interpretación de las reglas se efectúa utilizando una pila, insertando operandos en ella y, cuando una operación se realiza, extrayendo los operandos de la pila e insertando de vuelta el resultado de operar con ellos. El intérprete comprueba si la instancia es cubierta por la regla, teniendo en cuenta la clase predicha y la real de la instancia, pudiendo producirse uno de los siguientes valores: verdadero positivo (T_P), falso positivo (F_P), verdadero negativo (T_N), y falso negativo (F_N).

Kernel de la matriz de confusión El *kernel* de la matriz de confusión, que se muestra en la parte inferior de la Figura 2, realiza el conteo del número de T_P, F_P, T_N, y F_N que han sido previamente calculados para cada regla. Estos valores se utilizan para construir la matriz de confusión que nos permite calcular los valores de *fitness*. El proceso de sumar todos los valores de un array se conoce como reducción [18].

Los hilos se organizan para realizar una reducción en dos niveles y proporcionar accesos de memoria alineados que puedan servirse en menos transacciones de memoria, evitando así conflictos de bancos en memoria compartida. Cada hilo lleva a cabo la reducción parcial de los resultados de cubrimiento de una hormiga, almacenando la contabilización parcial en la memoria compartida. Una vez todos los elementos han sido contabilizados, se efectúa una llamada a una barrera de sincronización. La barrera

de sincronización se usa para coordinar la comunicación entre los hilos de un mismo bloque. Cuando algunos hilos dentro de un bloque acceden las mismas direcciones en memoria global o compartida, existen riesgos potenciales de lectura-tras-escritura, escritura-tras-lectura y escritura-tras-escritura para algunos accesos a memoria.

A continuación, sólo cuatro hilos realizan la suma final de las cuatro posiciones de la matriz de confusión. En este punto, no es necesario llamar a la barrera de sincronización dado que los hilos se encuentran en el mismo grupo y, ya que un grupo ejecuta una instrucción común a la vez, los hilos dentro de un grupo están sincronizados implícitamente.

Por último, se calcula los valores de *fitness* para los diferentes objetivos y se escriben de vuelta en la memoria global. Todos los valores de *fitness* se devuelven en un único array para ahorrar múltiples transacciones de memoria segmentadas.

3 Configuración del estudio experimental

En esta sección se presenta la configuración del estudio experimental. Los experimentos se ejecutaron en un PC equipado con un procesador de cuatro núcleos Intel Core i7 a 2.66 GHz, 12 GB de DDR3-1600 y 2 tarjetas gráficas NVIDIA GTX 690 de doble GPU (en total existen 4 GPUs operativas). El sistema operativo fue GNU/Linux Ubuntu 12.04 64 bit, junto con CUDA runtime 5.0, controladores NVIDIA 310.40, Eclipse 3.7.0, Java OpenJDK 1.6-23 y compilador GCC 4.6.4 (nivel de optimización O3).

En cuanto a la parametrización de la experimentación hay que definir por un lado los parámetros de los kernels, y por otro lado los parámetros del algoritmo MOGBAP. La configuración de los kernels requiere definir el número de hilos por bloque, habitualmente siendo 128, 256 o 512 hilos por bloque. De acuerdo a la hoja de cálculo de NVIDIA de utilización de la GPU, el número óptimo de hilos por bloque para nuestro problema es de 256 hilos por bloque. Este número garantiza la máxima utilización de los recursos de la GPU, mantiendo la carga de trabajo de la GPU lo más ocupada posible, a la vez que proporciona un gran número de bloques en el grid que permitan facilitar la escalabilidad a futuras GPUs con un mayor número de procesadores. Por otro lado, el algoritmo MOGBAP se ejecutó usando los parámetros por defecto recomendados por los autores del algoritmo en [12].

4 Resultados

En esta sección se presentan y analizan los resultados experimentales procedentes de la ejecución del modelo sobre 11 conjuntos de datos del repositorio UCI [19]. Los conjuntos de datos poseen una gran variedad en su grado de complejidad. El número de clases varía entre 2 y 17, el número de atributos varía entre 4 y 44, y el número de instancias varía entre 150 y 1 millón. Los experimentos evalúan diferentes tamaños de población para analizar la escalabilidad del modelo, especialmente sobre conjuntos de datos completos con un elevado número de instancias. Los resultados experimentales mostrados representan la media del tiempo de ejecución y la aceleración sobre 100 ejecuciones independientes.

Tabla 1. Resultados en conjuntos de datos UCI

					Tiempo de ejecución (ms)				Aceleración vs 4 CPUs		
Conjunto de datos	Instancias	A	C	R	4 CPUs	1 690	2 690	4 690	1 690	2 690	4 690
iris	150	4	3	10	62.80	35.01	20.91	15.14	1.79	3.00	4.15
				20	90.70	37.44	25.25	16.45	2.42	3.59	5.51
				50	156.00	49.31	29.34	21.47	3.16	5.32	7.27
				100	235.30	57.92	35.09	25.78	4.06	6.71	9.13
				200	434.30	81.41	58.75	37.64	5.33	7.39	11.54
ionosphere	351	33	2	10	282.70	38.21	23.30	12.62	7.40	12.13	22.40
				20	418.70	42.46	25.75	15.23	9.86	16.26	27.49
				50	770.00	53.02	38.04	29.83	14.52	20.24	25.81
				100	1,411.90	85.07	51.85	34.62	16.60	27.23	40.78
				200	2,691.50	122.21	77.74	58.34	22.02	34.62	46.13
australian	690	14	2	10	208.50	33.56	23.01	17.58	6.21	9.06	11.86
				20	315.30	41.94	30.49	19.66	7.52	10.34	16.04
				50	429.37	51.57	30.72	23.51	8.33	13.98	18.26
				100	847.62	70.94	45.08	28.86	11.95	18.80	29.37
				200	1,732.65	73.02	45.76	37.37	23.73	37.86	46.36
vowel	990	13	11	10	308.00	54.90	35.43	28.18	5.61	8.69	10.93
				20	487.40	68.70	43.51	30.26	7.09	11.20	16.11
				50	761.05	82.46	56.34	38.47	9.23	13.51	19.78
				100	1,458.00	100.86	63.84	41.58	14.46	22.84	35.06
				200	3,262.87	154.51	91.48	62.96	21.12	35.67	51.82
segment	2,310	19	7	10	722.40	67.45	49.38	41.12	10.71	14.63	17.57
				20	1,215.70	76.33	53.46	44.26	15.93	22.74	27.47
				50	2,117.47	113.36	71.90	45.29	18.68	29.45	46.75
				100	3,962.32	159.06	103.02	80.06	24.91	38.46	49.49
				200	7,630.10	222.55	137.10	98.81	34.28	55.65	77.22
mushroom	8,124	22	2	10	2,074.40	94.46	57.98	49.20	21.96	35.78	42.16
				20	3,723.90	117.48	70.86	52.89	31.70	52.55	70.41
				50	7,141.74	152.41	95.64	55.00	46.86	74.67	129.85
				100	15,581.70	245.83	145.94	75.02	63.38	106.77	207.70
				200	29,596.69	424.43	231.88	142.33	69.73	127.64	207.94
kr-vs-kp	28,056	6	17	10	1,320.30	70.28	46.91	34.55	18.79	28.15	38.21
				20	2,394.70	71.24	49.63	42.15	33.61	48.25	56.81
				50	4,179.39	97.42	69.59	52.96	42.90	60.06	78.92
				100	8,058.31	159.90	80.38	57.69	50.40	100.25	139.68
				200	15,899.56	185.08	123.26	82.04	85.91	128.99	193.80
connect-4	67,557	42	3	10	24,665.15	794.75	636.26	473.05	31.04	38.77	52.14
				20	42,737.95	886.92	652.52	524.94	48.19	65.50	81.41
				50	110,240.25	1,351.64	1,086.06	836.53	81.56	101.50	131.78
				100	212,262.88	2,027.63	1,559.34	1,201.56	104.69	136.12	176.66
				200	412,462.96	3,191.51	2,485.45	1,843.21	129.24	165.95	223.77
fars	100,968	29	8	10	28,265.81	954.79	584.11	417.38	29.60	48.39	67.72
				20	51,754.57	1,110.29	657.12	452.73	46.61	78.76	114.32
				50	121,558.74	1,880.54	998.31	638.77	64.64	121.76	190.30
				100	251,519.89	2,727.52	1,332.41	994.23	92.22	188.77	252.98
				200	491,768.62	4,902.58	2,574.66	1,833.58	100.31	191.00	268.20
kddcup	494,020	41	10	10	162,285.80	7,547.98	4,042.97	2,357.78	21.50	40.14	68.83
				20	323,639.32	8,217.94	5,433.55	3,727.78	39.38	59.56	86.82
				50	775,597.05	12,802.72	7,942.96	4,816.93	60.58	97.65	161.01
				100	1,667,554.00	19,620.57	14,207.00	9,116.12	84.99	117.38	182.92
				200	3,283,987.91	29,580.09	17,324.18	13,272.88	111.02	189.56	247.42
poker	1,025,010	10	10	10	118,820.52	5,389.48	2,968.95	1,627.00	22.05	40.02	73.03
				20	245,588.40	7,994.84	5,250.21	3,291.00	30.72	46.78	74.62
				50	591,440.37	12,464.92	6,815.25	5,157.01	47.45	86.78	114.69
				100	1,219,990.65	20,725.03	10,011.94	6,737.62	58.87	121.85	181.07
				200	2,508,965.42	28,032.03	16,520.27	10,429.78	89.50	151.87	240.56

La Tabla 1 muestra los conjuntos de datos (instancias, attributos (A), clases (C)), los tiempos de ejecución y la aceleración obtenida. El modelo demuestra alcanzar un gran rendimiento y eficiencia, que se incrementa conforme el número de instancias y el tamaño de la población (número de reglas (R)) se incrementan. Concretamente, la aceleración máxima se alcanza cuando se emplean 4 GPUs sobre los tres conjuntos de datos con mayor número de instancias y con los mayores tamaños de población. Por otro lado, resultados menos significativos se obtienen en los conjuntos más pequeños. No obstante, pese a su menor número de instancias, se obtiene en todo caso aceleraciones superiores a 1, es decir, se demuestra que independientemente del tamaño de los datos y del número de reglas, siempre es recomendable realizar la evaluación en GPU. Además, se observa una buena escalabilidad del modelo respecto al aumento del número de GPUs, tamaño de la población y del tamaño de los datos. La Figura 3 muestra la tendencia de la aceleración con 1, 2 y 4 GPUs con respecto al número de instancias.

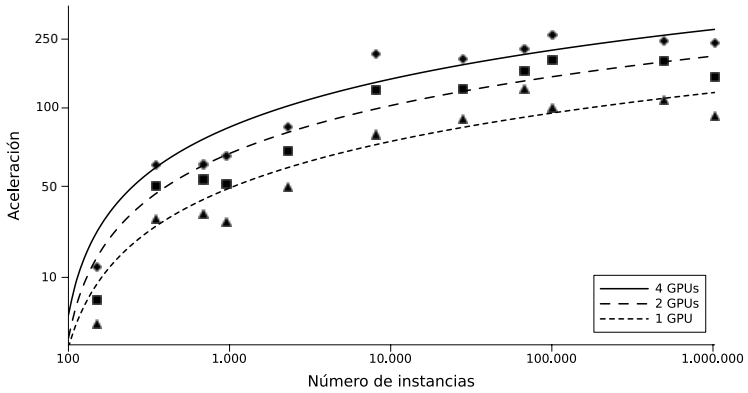

Figura 3. Aceleración con respecto al número de instancias

5 Conclusiones

En este trabajo se ha presentado una implementación paralela en GPU del algoritmo MOGBAP. El proceso de creación de hormigas, la estrategia multiobjetivo y el enfoque de nichos para seleccionar las reglas del clasificador se han paralelizado usando multihilo, debido a sus requerimientos secuenciales internos. Por otro lado, la evaluación de las hormigas, que es el proceso que demanda más tiempo, se realiza en GPU, alcanzando una gran aceleración especialmente sobre conjuntos de datos de alta dimensionalidad. El estudio experimental ha analizado el rendimiento y eficiencia del modelo atendiendo a diferentes tamaños de población y conjuntos de datos con variado número de instancias y atributos. El modelo ha demostrado una buena escalabilidad a conjuntos de datos de gran tamaño y a múltiples GPUs. Los resultados experimentales demuestran la gran aceleración que supone la implementación del modelo de evaluación en GPU,

alcanzando aceleraciones de alrededor de 250 $\times$ cuando se compara el tiempo de evaluación en 4 GPUs con el multi-hilo en CPU. Además, se nos permite abordar conjuntos de datos mucho más grandes que anteriormente no era posible en un tiempo razonable.

Referencias

1. Michelakos, I., Mallios, N., Papageorgiou, E., Vassilakopoulos, M.: Ant colony optimization and data mining. Studies in Computational Intelligence **352** (2011) 31–60
2. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation **6**(5) (2002) 443–462
3. Luque, G., Alba, E.: Parallel genetic algorithms: Theory and realworld applications. Studies in Computational Intelligence **367** (2011) 1–183
4. Pedemonte, M., Nesmachnow, S., Cancela, H.: A survey on parallel ant colony optimization. Applied Soft Computing **11** (2011) 5181–5197
5. Cecilia, J.M., García, J.M., Nisbet, A., Amos, M., Ujaldón, M.: Enhancing data parallelism for ant colony optimization on GPUs. Journal of Parallel and Distributed Computing **73**(1) (2013) 42–51
6. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing **73**(1) (2013) 52–61
7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing **68**(10) (2008) 1370–1380
8. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1) (2007) 80–113
9. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems **22**(2) (2007) 69–78
10. Vidal, P., Luna, F., Alba, E.: Systolic neighborhood search on graphics processing units. Soft Computing **In press** (2013)
11. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: Recent advances and new trends. International Transactions in Operational Research **20**(1) (2013) 1–48
12. Olmo, J.L., Romero, J.R., Ventura, S.: Classification rule mining using ant programming guided by grammar with multiple Pareto fronts. Soft Computing **16** (2012) 2143–2163
13. Cano, A., Olmo, J., Ventura, S.: Parallel multi-objective ant programming for classification using gpus. Journal of Parallel and Distributed Computing **73**(6) (2013) 713–728
14. Wong, M.L., Leung, K.S.: Data Mining Using Grammar-Based Genetic Programming and Applications. Kluwer Academic Publishers, Norwell, MA, USA (2000)
15. Cano, A., Zafra, A., Ventura, S.: Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Computing **16**(2) (2012) 187–202
16. Franco, M.A., Krasnogor, N., Bacardit, J.: Speeding up the evaluation of evolutionary learning systems using GPGPUs. In: Genetic and Evolutionary Computation Conference (GECCO) (2010). (2010) 1039–1046
17. Leist, A., Playne, D., Hawick, K.: Exploiting graphical processing units for data-parallel scientific applications. Concurrency Computation Practice and Experience **21**(18) (2009) 2400–2437
18. Hwu, W.: Illinois ECE 498AL: Programming Massively Parallel Processors, Lecture 13: Reductions and their Implementation (2009)
19. Newman, D.J., A. Asuncion: UCI machine learning repository (2007)