GPU-SME-kNN: kNN escalable y eficiente en memoria utilizando GPU

Pablo D. Gutiérrez[1], Miguel Lastra[2], Jaume Bacardit[3], José M. Benítez[1], and Francisco Herrera[1]

[1] Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada, Granada, España,
[2] Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada, Granada, España
[3] Interdisciplinary Computing and Complex BioSystems (ICOS) Research Group, School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
pdgp@decsai.ugr.es, mlastral@ugr.es, jaume.bacardit@newcastle.ac.uk, J.M.Benitez@decsai.ugr.es, herrera@decsai.ugr.es

Resumen El clasificador de los k vecinos más cercanos (kNN) es una de las técnicas más usadas en minería de datos por su simplicidad y bajo error de identificación. No obstante, el esfuerzo de su cálculo está directamente relacionado con el tamaño de los conjuntos de datos, lo que resulta en un rendimiento bajo cuando el tamaño de éstos aumenta. Los procesadores gráficos han demostrado que pueden mejorar el rendimiento del clasificador kNN, pero las propuestas actuales presentan limitaciones. En este trabajo proponemos un nuevo diseño escalable y eficiente en memoria para el clasificador kNN utilizando procesadores gráficos, llamado GPU-SME-kNN. GPU-SME-kNN elimina la relación entre el tamaño del conjunto de datos y la cantidad de memoria necesaria para su cálculo al mismo tiempo que mantiene un gran rendimiento. El estudio experimental que se presenta confirma este hecho y muestra un consumo de recursos aceptable para la mayor parte de los procesadores gráficos comerciales.

Keywords: kNN, GPU, CUDA

1. Introducción

El clasificador basado en la regla de los k vecinos más cercanos (k nearest neighbor o kNN) [1] [2] es una de las técnicas más utilizadas en minería de datos y reconocimiento de patrones. Su sencillez y alto rendimiento en clasificación la convierten en la técnica de referencia con la que probar clasificadores y conjuntos de datos [3]. Esta técnica está considerada entre las 10 mejores del mundo [4].

El clasificador kNN se basa en la idea de que una instancia desconocida será similar a otras instancias que se encuentren cerca de ella en el espacio de características. Si bien la idea es sencilla, el coste computacional que requiere es elevado y se incrementa cuando aumentan tanto el número de atributos como el de instancias en el conjunto de datos. En conjuntos suficientemente grandes resulta imposible utilizar el clasificador kNN debido al tiempo que requiere.

En la actualidad encontramos aplicaciones que producen rutinariamente cantidades masivas de datos que introducen retos de escalabilidad para las técnicas de minería de datos [5]. Resulta imprescindible abordar los problemas de escalabilidad inherentes al clasificador kNN para poder aplicarla en estos casos.

Los procesadores gráficos (Graphics Processing Units, GPU), a través de su paralelismo masivo, han demostrado su capacidad para manejar grandes cantidades de datos de forma eficiente en diversas situaciones como la identificación mediante huellas dactilares [6] o la minería de datos [7]. El clasificador kNN ha sido adaptada con éxito para su ejecución en estos dispositivos mejorando su rendimiento [8] [9] [10]. Sin embargo, estas propuestas aún presentan limitaciones, introducidas por la limitada cantidad de memoria de que dispone la GPU.

En este trabajo, proponemos un diseño del clasificador kNN, denominado GPU-SME-kNN por sus siglas en ingles: GPU-based scalable and memory efficient kNN, que aborda los problemas mencionados. En esta propuesta, introducimos un cálculo incremental del vecindario que elimina las dependencias entre el tamaño del conjunto de datos y la cantidad de memoria necesaria para los cálculos. Esta estrategia se combina con una selección de vecinos eficiente basada en QuickSort que proporciona una solución escalable y de alto rendimiento.

GPU-SME-kNN ha sido probado con diferentes conjuntos de datos del repositorio UCI [11] incrementando el número de instancias hasta los 4,5 millones y con diferentes valores del parámetro k para estudiar detalladamente su comportamiento en términos de escalabilidad. Los resultados de centran en la evaluación del rendimiento y de la memoria requerida por la propuesta. GPU-SME-kNN se compara con otras propuestas para el clasificador kNN sobre GPU bien conocidas en la literatura, mostrando un buen rendimiento.

El resto del trabajo se organiza como sigue: la Sección 2 presenta los antecedentes sobre los que se fundamenta este trabajo, la Sección 3 describe GPU-SME-kNN, la Sección 4 muestra los resultados de nuestra propuesta y, por último, la Sección 5 resume las conclusiones de este trabajo.

2. El clasificador de los k vecinos más cercanos y los procesadores gráficos

Esta sección resume los principales aspectos del clasificador de los k vecinos más cercanos, así como de los procesadores gráficos y de las propuestas para el clasificador kNN haciendo uso de estos dispositivos.

2.1. El clasificador de los k vecinos más cercanos

El clasificador los k vecinos más cercanos (kNN) [1] predice la clase de una instancia de test como la clase mayoritaria entre las k instancias de entrenamiento más cercanas a la instancia de test. De este modo, cada instancia de test es comparada con todas las instancias de entrenamiento, midiéndose la distancia entre ambas. Las instancias correspondientes a los k valores menores se utilizan para predecir la clase de la instancia de test.

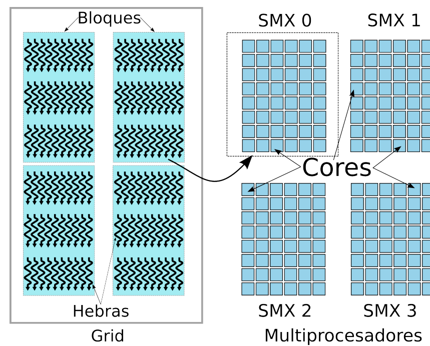


Figura 1. Grid, bloques, hebras, multiprocesadores y cores en la GPU.

El clasificador kNN se aplica habitualmente a conjuntos de instancias de test. Si M es el número de instancias de test y N el número de instancias de entrenamiento el algoritmo requiere $M \times N$ operaciones de distancia, así como M selecciones de k instancias entre un array de N elementos. El incremento del número de operaciones, cuando los conjuntos de datos crecen, dificulta el uso de esta regla en conjuntos de datos grandes.

2.2. Procesadores gráficos y NVIDIA CUDA

Los procesadores gráficos (GPU) se crearon originalmente para descargar al procesador principal (CPU) de cálculos relacionados con el proceso de gráficos en tres dimensiones, utilizando una arquitectura SIMD (Single Instruction Multiple Data). NVIDIA CUDA es una plataforma hardware/software que permite utilizar GPU de NVIDIA para aplicaciones de propósito general, presentándola como un coprocesador paralelo con sus propias memorias, cachés y registros. Para sacar partido de estos dispositivos es necesario rediseñar los algoritmos, identificando que partes del algoritmo se adaptan mejor a la CPU y cuales a la GPU, intentando minimizar las transferencias de memoria entre las mismas.

Las funciones que se ejecutan en una GPU (denominadas kernel) utilizan un conjunto de hebras que ejecutan el mismo código sobre distintos datos, siguiendo la arquitectura SIMD, llamado grid. Las hebras de un grid se distribuyen en bloques, identificados por un índice tridimensional. Del mismo modo, las hebras de un bloque se identifican mediante otro índice tridimensional.

A un nivel más bajo, una GPU tiene un conjunto de cores de cálculo agrupados dentro de multiprocesadores (SMX). Cuando se ejecuta un grid cada bloque de hebras se asigna a un SMX que lo ejecuta de forma independiente a los demás. Por este motivo es posible sincronizar las hebras dentro de un bloque pero no es posible sincronizar todas las hebras pertenecientes al grid.

Dentro del SMX cada bloque se divide en grupos de 32 hebras, denominados warp, que se ejecutan de forma paralela. Todas las hebras de un warp ejecutan la misma instrucción. En el caso de que aparezcan divergencias en el código,

por ejemplo, con una instrucción condicional evaluada de forma distinta en cada hebra, la ejecución se serializa, penalizando el rendimiento de la aplicación.

En la llamada a un kernel el programador tiene que indicar el número de hebras por bloque y el número de bloques que tiene el grid. En los casos en los que el número de bloques es elevado pero el número de hebras de cada uno de ellos es bajo aparecen limitaciones en el rendimiento ya que existe una limitación máxima de bloques que se pueden ejecutar en un SMX simultáneamente.

Otro aspecto que se debe indicar en la llamada al kernel es la cantidad de memoria compartida que cada bloque utiliza. Los SMX tienen una memoria caché programable muy rápida que permite compartir datos entre hebras de un bloque. Esta memoria es limitada por lo que el número de bloques que se pueden ejecutar en el SMX también se ve afectado por su uso.

Finalmente, la memoria global de la GPU, a través de la cual se realiza la transferencia de datos de entrada y salida con la CPU, está optimizada para realizar accesos de forma coalescente. Esto es que hebras con índices consecutivos dentro de un bloque acceden a posiciones de memoria también consecutivas. Cuando los accesos no son de este tipo el rendimiento se ve penalizado.

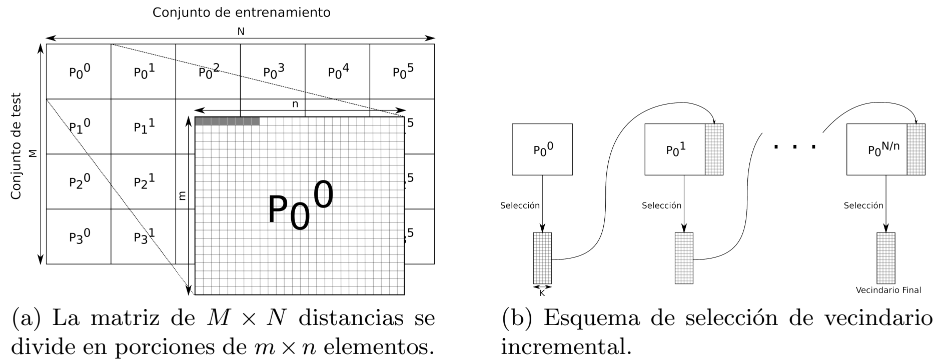
2.3. Propuestas basadas en GPU para el clasificador *k*NN en la literatura

Existen varias propuestas de diseño para el clasificador *k*NN basadas en GPU [12] [8] [13] [9] [14] [10]. Todas estas propuestas dividen el cálculo del clasificador *k*NN en dos partes, el cálculo de distancias (agrupando el cálculo de todas las distancias entre instancias de entrenamiento y test en una matriz) y la selección de los vecinos más cercanos (realizada en paralelo sobre las filas de la matriz). Las distintas propuestas varían en la forma en la que se abordan estos dos pasos.

Para comparar los resultados hemos seleccionado tres propuestas. Garcia et al. [8] que es la propuesta de referencia con la que se compara en la literatura. GPU-FS-*k*NN, presentado por Arefin et al. [9], que introduce un esquema en el que se divide la matriz en partes para su procesamiento. Y, finalmente, Komarov et al. [10] que utiliza un método de selección basado en QuickSort [15].

Todas las propuestas, salvo la propuesta de Arefin et al., asumen que la matriz de distancias puede almacenarse de forma completa en la memoria de la GPU. Sin embargo esto no es posible para conjuntos de datos grandes como por ejemplo el conjunto KDDCup 1999. La solución propuesta por García et al. consiste en limitar el número de instancias de test y realizar los cálculos de forma iterativa hasta cubrir todas las instancias. Sin embargo no considera el uso de copias asíncronas por lo que el rendimiento se ve afectado notablemente. Además esta propuesta tiene impacto en otras partes del algoritmo, al reducir el número de instancias de test por iteración.

GPU-FS-*k*NN es la excepción, al operar con trozos de la matriz de distancias. Sin embargo, los propios autores admiten que su propuesta aún tiene limitaciones de memoria al requerir que los conjuntos de entrenamiento y test se copien íntegramente a memoria GPU. Nuestra propuesta aborda estos problemas produciendo una solución que no afecte al rendimiento del algoritmo.



(a) La matriz de $M \times N$ distancias se divide en porciones de $m \times n$ elementos.

(b) Esquema de selección de vecindario incremental.

Figura 2. Representación de las etapas del cálculo de la matriz y el vecindario.

3. GPU-SME-*k*NN: Cálculo del clasificador *k*NN basado en GPU escalable y eficiente en memoria

En esta sección se explica en detalle el diseño seguido en la construcción de nuestra propuesta, GPU-SME-*k*NN. La Sección 3.1 describe las interacciones entre CPU y GPU, vitales para el buen rendimiento de la propuesta. La Sección 3.2 presenta el diseño de los kernel en aras de obtener el mejor rendimiento.

3.1. Modelo de interacción CPU-GPU

La cantidad de memoria disponible en una GPU es, en general, inferior a la que tendríamos disponible en una CPU. Algunas estructuras de datos, como la matriz de distancias podrían tener un tamaño mayor a la memoria disponible. Por este motivo dividimos el cálculo en etapas que además harán uso de transferencias de memoria asíncronas para mantener el rendimiento.

Siendo M el número de instancias del conjunto de test y N el número de instancias del conjunto de entrenamiento, este esquema, que hemos denominado calculo incremental de vecindario, divide la matriz de distancias, de dimensión $M \times N$, en trozos de tamaño $m \times n$. Estos trozos tienen un tamaño suficientemente grande para proveer a los kernel de suficiente carga de cálculo a la vez que son lo suficientemente pequeños para no ocupar en exceso la memoria de la GPU.

Una vez calculada la porción de la matriz de distancias se calcula el vecindario de esa porción y dicho vecindario se incorpora al trozo siguiente de matriz y se tiene en cuenta en el siguiente cálculo de vecindario. De esta manera cuando se calcula el vecindario del último trozo de una fila de la matriz el vecindario obtenido es el vecindario final de las instancias. La Figura 2 muestra como se divide la matriz y como se aplica este esquema de cálculo de vecindario.

Dos kernel se ocupan de cada una de las partes de la regla, el primero calculando la porción de la matriz de distancias y el segundo la selección del vecindario. Esto permite realizar las transferencias de memoria de forma asíncrona de forma sencilla ya que los datos que se utilizan en el cálculo de la matriz, partes

```
1  CopiaAsincronaPorcionTest_1
2  CopiaAsincronaPorcionEntrenamiento_1
3  for i ← 1 to M/m do
4  |  ComprobacionCopiaPorcionTest_i
5  |  for j ← 1 to N/n do
6  |  |  ComprobacionCopiaPorcionEntrenamiento_j
7  |  |  CalculoMatrizDistancias_{i,j}
8  |  |  if j = N/n then
9  |  |  |  CopiaAsincronaPorcionTest_{i+1}
10 |  |  |  CopiaAsincronaPorcionEntrenamiento_1
11 |  |  else
12 |  |  |  CopiaAsincronaPorcionEntrenamiento_{j+1}
13 |  |  end
14 |  |  CalculoSeleccion_j
15 |  end
16 |  ComprobacionCopiaPorcionVecindario_i
17 |  CopiaSincronaVecindario_i
18 |  CopiaAsincronaPorcionVecindario_i
19 end
```

Figura 3. Pseudocódigo del algoritmo propuesto.

del conjunto de entrenamiento y test, no son necesarios en la selección de distancias y se pueden copiar asíncronamente mientras se realiza esta parte del cálculo. De esta manera se puede mantener el rendimiento del algoritmo sin tener que copiar los conjuntos de entrenamiento y test completos a la memoria de la GPU, lo que podría penalizar el rendimiento al ocupar una mayor cantidad de memoria en algunos casos. La copia del vecindario se hace en dos partes, una primera, síncrona, dentro de la GPU para asegurar que los datos no se sobrescriben y una segunda, asíncrona, mientras se calculan las siguientes porciones de la matriz. La figura 3 presenta el pseudocódigo del algoritmo, en el que se pueden apreciar todas las partes y copias necesarias. Al tratarse de copias asíncronas, en dicha figura se incluyen también las comprobaciones de que la copia ha terminado antes de continuar el proceso.

3.2. Diseño de kernels para el clasificador *k*NN

El cálculo del clasificador *k*NN requiere de dos kernel: uno para calcular una porción de la matriz de distancias y otro que realice el proceso de selección.

El cálculo de la porción de la matriz de distancias es similar al presentado por Arefin et al. [9] aunque presenta algunas diferencias. Arefin divide la matriz en porciones cuadradas y calcula una distancia por cada hebra siendo un bloque cada fila de la porción. En GPU-SME-*k*NN el tamaño de la porción de la matriz es arbitrario determinado por el usuario a través de los parámetros m y n. El bloque sigue considerándose una fila pero en lugar de utilizar n hebras se utiliza un número predefinido d de hebras, menor que n de forma que se calculan varias

distancias por hebra de forma consecutiva. En la figura 2(a) la zona marcada en gris representa las d hebras del primer bloque calculando la primera distancia del mismo.

Posteriormente, el segundo kernel realiza la selección del vecindario basándose en el algoritmo QuickSort [15]. Este método de selección trabaja recursivamente sobre los elementos menores o mayores que el pivote hasta encontrar los k vecinos más cercanos. Para adaptarlo a GPU es necesario transformarlo en un método iterativo. Esto se consigue intercambiando los papeles del array dividido y la parte sobre la que se ejecutaría la llamada recursiva.

El diseño de este kernel de selección presenta dos problemas más. El primero es que las escrituras en memoria global resultantes de dividir el array en dos partes no son coalescentes. La solución a este problema se solventa realizando las escrituras en dos etapas, una primera no coalescente en memoria compartida y una segunda coalescente. El segundo problema es determinar la posición en la que se debe escribir en memoria compartida. CUDA proporciona funciones que resuelven este problema para un warp, sin embargo el uso de bloques de este tamaño no resultaría eficiente. Este problema se puede resolver de dos formas: incrementando el número de hebras e introduciendo mecanismos de sincronización, la solución adoptada por Komarov et al. [10], o incrementando el número de filas que se seleccionan en un bloque calculando cada una de ellas en un warp, solución que presenta nuestra propuesta.

4. Estudio Experimental

En esta sección se detallan los experimentos (Sección 4.1), resultados (Sección 4.2) y el análisis (Sección 4.3 de la evaluación de GPU-SME-kNN frente a otras propuestas de la literatura.

4.1. Experimentos

Aunque el diseño del algoritmo difiere considerablemente según se vaya a ejecutar en CPU o en GPU, las operaciones que se realizan son las mismas y, por tanto, los resultados obtenidos a nivel de acierto en clasificación son los mismos. Teniendo esto en cuenta los experimentos se han diseñado para evaluar el rendimiento en tiempo de las distintas propuestas.

Se han utilizado dos conjuntos de datos grandes del repositorio UCI [11] para realizar los experimentos: Poker, con 1 025 009 instancias, 10 atributos y 10 clases, y KDDCup 1999, con 4 898 431 instancias, 41 atributos y 5 clases.

Estos conjuntos de datos han sido submuestreados a distintos tamaños, para comprobar cómo evolucionan los algoritmos, sobre cada uno de estos submuestreos se ha aplicado un esquema de validación cruzada con cinco partes.

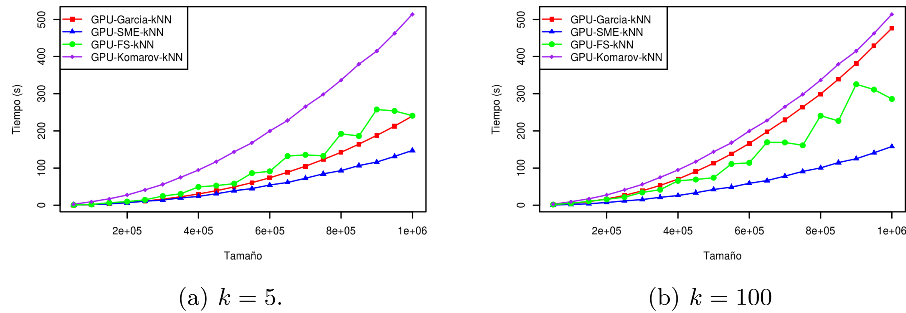
Así mismo se han utilizado dos valores de k: 5 y 100. El valor de $k = 100$ puede ser poco significativo para los resultados a nivel de acierto, pero como ya hemos comentado el objetivo de los experimentos es evaluar la escalabilidad de la propuesta.

GPU-SME-kNN se ha comparado con tres propuestas disponibles en la literatura: Garcia et al. [8], denominado GPU-Garcia-kNN, Arefin et al. [9],denominado GPU-FS-kNN y Komarov et al. [10], denominado GPU-Komarov-kNN. En este último, se ha reemplazado el cálculo de la matriz original basado en GPU-Garcia-kNN por nuestro cálculo incremental del vecindario, aunque los parámetros se ajustan para ofrecer un resultado equivalente a la propuesta original.

Todos los experimentos se han llevado a cabo en una GPU NVIDIA Tesla K20m con 5GB de memoria RAM y 2496 CUDA cores. No obstante, GPU-SME-kNN puede modificar sus parámetros para adaptarse a equipos con especificaciones menores. Los parámetros de GPU-SME-kNN en todas las ejecuciones han sido $m = 16384$, $n = 2048$ y $d = 256$.

4.2. Resultados

La figura 4 presenta los resultados del conjunto de datos Poker. La figura 5 presenta los resultados del conjunto de datos KDDCup 1999. Cada parte de estas figuras corresponde a uno de los valores de k seleccionados.

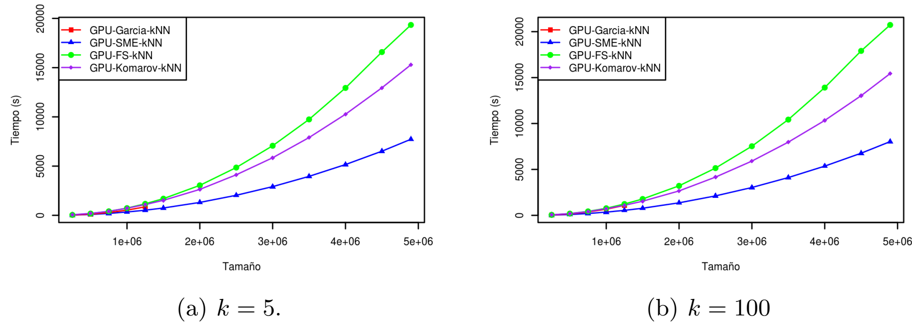


(a) $k = 5$. (b) $k = 100$

Figura 4. Resultados del conjunto de datos Poker.

4.3. Análisis de los resultados

Como se puede observar en las figuras anteriores, GPU-SME-kNN mejora el rendimiento del resto de propuestas vistas. La forma de gestionar la matriz de distancias introduce la primera gran diferencia en el rendimiento. GPU-Garcia-kNN solo es capaz de completar su ejecución hasta conjuntos de datos con un tamaño de 1,250 millones de instancias al necesitar almacenar tiras de la matriz completas en memoria. El método de selección también influye en los resultados de GPU-Garcia-kNN ya que el método de ordenación por inserción utilizado es susceptible a provocar divergencias en la ejecución de los warp, algo que sucede con mayor frecuencia al elevar el valor de k.



(a) $k = 5$. (b) $k = 100$

Figura 5. Resultados del conjunto de datos KDDCup 1999.

Las divergencias en la ejecución también afectan a GPU-FS-kNN en su método basado en la selección por inserción. Su esquema de división de la matriz de distancias le permite completar todos los experimentos si bien el rendimiento se ve afectado por el esquema de selección y por no hacer uso de transferencias asíncronas. Este hecho introduce tiempos de espera en los que no se está realizando cálculo reduciendo así el rendimiento de la propuesta.

En el caso de GPU-Komarov-kNN, las diferencias radican sólo en el método de selección. Tanto esta propuesta como nuestra GPU-SME-kNN utilizan una selección basada en QuickSort. La principal diferencia entre ambos métodos es que GPU-Komarov-kNN utiliza un bloque grande para calcular cada selección, esto obliga a introducir puntos de sincronización dentro del kernel para que los distintos warp intercambien las posiciones en las que deben escribir los resultados en cada iteración del algoritmo de selección. GPU-SME-kNN utiliza un solo warp, por lo que no requiere de sincronizaciones. Por último, comentar que GPU-Komarov-kNN se comportaría de la misma manera que GPU-Garcia-kNN si no hubiéramos utilizado el cálculo incremental de vecindario.

El consumo de memoria GPU en los experimentos en el caso de Poker fue de 1158 MB y 1247 MB para los respectivos valores de $k = 5$ y $k = 10$. En el caso de KDDCup 1999 el consumo fue de 1162 MB y 1251 MB respectivamente. Esto se debe a que por el diseño del algoritmo la cantidad de memoria necesaria sólo depende de los parámetros m, n y k (fijos en todas las ejecuciones) y del número de atributos del conjunto de datos. Además los resultados obtenidos muestran unos valores asumibles para la mayoría de procesadores gráficos recientes.

5. Conclusiones

Se ha presentado una nueva propuesta de diseño para el clasificador kNN basada en GPU que mejora el rendimiento de las propuestas disponibles en la literatura y que proporciona una gran escalabilidad en términos de tamaño del conjunto de datos y número de vecinos. GPU-SME-kNN mantiene estable el uso

de memoria para los cálculos independientemente del tamaño de los conjuntos de datos, algo que no ocurría en las propuestas anteriores para el clasificador kNN basadas en GPU. Además como la cantidad de memoria utilizada puede ser definida por el usuario y no se utilizan características presentes solo en los procesadores gráficos más modernos nuestra propuesta puede ser utilizada de forma eficiente en una gran variedad de modelos de GPU.

Agradecimientos

Este trabajo ha sido financiado por los proyectos TIN2014-57251-P, TIN2013-4720-P, P10-TIC-06858 y P12-TIC-2958. P.D. Gutiérez disfruta de una beca FPI del Ministerio de Economía y Competitividad (BES-2012-060450).

Referencias

1. Cover, T., Hart, P.: Nearest neighbor pattern classification. Information Theory, IEEE Transactions on **13**(1) (January 1967) 21–27
2. Shakhnarovich, G., Darrell, T., Indyk, P.: Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing). MIT Press (2006)
3. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification (2Nd Edition). Wiley-Interscience (2000)
4. Wu, X., Kumar, V.: The top ten algorithms in data mining. CRC Press (2010)
5. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. Cambridge University Press (2011)
6. Lastra, M., Carabaño, J., Gutiérrez, P.D., Benítez, J.M., Herrera, F.: Fast fingerprint identification using gpus. Information Sciences **301**(0) (2015) 195 – 214
7. Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. (2008) 104–111
8. Garcia, V., Debreuve, E., Nielsen, F., Barlaud, M.: K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In: Proceedings - International Conference on Image Processing, ICIP. (2010) 3757–3760
9. Arefin, A.S., Riveros, C., Berretta, R., Moscato, P.: GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs. PLoS ONE **7**(8) (2012)
10. Komarov, I., Dashti, A., D'Souza, R.M.: Fast k-NNG Construction with GPU-Based Quick Multi-Select. PLoS ONE **9**(5) (2014)
11. Bache, K., Lichman, M.: UCI machine learning repository (2013)
12. Kuang, Q., Zhao, L.: A practical GPU based KNN algorithm. In: In Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCSCT '09). (December 2009)
13. Kato, K., Hosino, T.: Multi-GPU algorithm for k-nearest neighbor problem. Concurrency and Computation: Practice and Experience **24**(1) (2012) 45–53
14. Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., Shi, Y.: Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). Journal of Supercomputing **64**(3) (2013) 942–967
15. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7) (July 1961) 321–322