Clasificación kNN de documentos usando GPU

Document kNN Clasification using GPU

Rubén Bresler Camps[1] y Reynaldo Gil García[2]
[1]Empresa de Desarrollo de Aplicaciones, Tecnologías y Sistemas,
Santiago de Cuba, Cuba
ruben.bressler@cerpamid.co.cu
[2]Centro de Reconocimiento de Patrones y Minería de Datos,
Santiago de Cuba, Cuba
gil@cerpamid.o.cu

Artículo recibido el 12 de febrero de 2011; aceptado el 30 junio de 2011

Resumen. *La búsqueda de los k vecinos más cercanos, ha sido aplicada a una amplia variedad de aplicaciones en el campo de la Minería de Textos y la Recuperación de Información por su simplicidad y precisión. Sin embargo, estas áreas del conocimiento en general manipulan objetos con altas dimensiones de rasgos que hacen que el proceso de encontrar los k objetos más similares a uno dado tenga una intensidad computacional elevada, debido a la gran cantidad de operaciones que se realizan para calcular la semejanza entre todos los objetos implicados. En este trabajo se proponen dos métodos de multiplicación paralela de matrices dispersas usando una GPU, que minimizan el tiempo empleado en el cálculo de semejanzas entre objetos del algoritmo kNN para clasificar documentos.*
Palabras clave. *GPGPU, clasificación de documentos y multiplicación de matrices dispersas.*

Abstract. *The search for the k nearest neighbors, has been applied to a wide variety of applications in the field of Text Mining and Information Retrieval for its simplicity and accuracy. However, these general areas of knowledge in handling high-dimensional objects with features that make the process of finding the k most similar objects to a given computer has a high intensity, due to the large number of operations performed to calculate the similarity between all the objects involved. In this paper we propose two methods for parallel sparse matrix multiplication using a GPU, which minimize the time spent in the calculation of similarities between objects in the kNN algorithm to classify documents.*
Keywords. *GPGPU, document classification and sparse matrix multiplication.*

1 Introducción

Probablemente, el tema más común en el análisis de documentos complejos es la clasificación o categorización de textos. De forma general, la tarea consiste en clasificar un documento de texto en un conjunto de categorías preestablecidas. Dado por un conjunto de categorías (materias, temas) y una colección de documentos de texto, el proceso consiste en encontrar el tema (o temas) correcto para cada documento.

La clasificación de textos es un componente importante en muchos sistemas de administración de información como el filtrado de correos spam, enrutamiento y diseminación de documentos, identificación de tópicos, clasificación de páginas Web, etc.

En general, la categorización de textos puede definirse formalmente como la tarea de aproximar la función de asignación de categorías $F: D \times C \rightarrow (0,1)$, donde D es el conjunto de todos los documentos posibles y C es el conjunto de categorías predefinidas. El valor de $F(d,c)$ es 1 si el documento d pertenece a la categoría C ó 0 en caso contrario. La aproximación de la función $M: D \times C \rightarrow (0,1)$ se llama clasificador. La tarea consiste en construir un clasificador que produzca resultados próximos a la verdadera función de asignación de categorías F [4].

Existen varios tipos de clasificadores, entre ellos se encuentran los clasificadores basados en ejemplos. Estos clasificadores no construyen representaciones declarativas explícitas de las categorías, sino que dependen directamente del cálculo de la similitud entre el documento que se clasifica y los documentos de entrenamiento. El conjunto de entrenamiento, para los clasificadores basados en ejemplos, consiste en almacenar las

representaciones de los documentos junto con sus etiquetas de categoría.

El ejemplo más prominente de un clasificador basado en ejemplos es el k NN (k -nearest neighbor, por su nombre en inglés) [4]. Para decidir si un documento d pertenece a la categoría C, k NN comprueba si los k documentos de entrenamiento más similares a d pertenecen a C. Si la respuesta es positiva para una proporción suficientemente grande de ellos, se etiqueta el documento con esa categoría, de lo contrario, la decisión es negativa. La distancia de la versión ponderada de k NN es una variación que pesa lacontribución de cada vecino por su semejanza con el documento de prueba.

Afortunadamente, el algoritmo *k*NN presenta un paralelismo de datos suficiente para permitir implementaciones en varias plataformas paralelas como por ejemplo las GPU. El uso de dispositivos gráficos para potenciar la aceleración de algoritmos, ha mostrado un inusitado interés en muchas comunidades científicas. Actualmente es común encontrar información sobre implementaciones de esta índole para resolver problemas paralelos con gran intensidad aritmética. Con los métodos propuestos en este trabajo se obtuvieron reducciones del tiempo de ejecución entre 65 % y el 85 %, comparados contra los mejores algoritmos para CPU.

El resto de este artículo está organizado de la siguiente forma: la sección 2 se describen los trabajos relacionados y la importancia de nuestras propuestas, luego se describe brevemente algunas características de la programación con GPU específicamente con CUDA en la sección 3, en las siguientes secciones 4 y 5 se describen las bases teóricas y la descripción de nuestras propuestas respectivamente y por último en la sección 6 se describen los resultados experimentales alcanzados.

2 Trabajo relacionado

El algoritmo de clasificación kNN ha sido ampliamente usado en el Reconocimiento de Patrones y en la Minería de Datos y es uno de los que mejores resultados obtiene cuando se trabaja con textos. Es robusto, en el sentido de no exigir a las categorías ser linealmente separadas. Su único inconveniente es el costo computacional relativamente alto de la clasificación, es decir, para cada documento de prueba, debe ser calculada su semejanza con todos los documentos de entrenamiento.

En grandes bases de datos de entrenamiento la búsqueda por fuerza bruta no es una opción acertada. Muchas variantes de algoritmos *k*NN han sido propuestas para reducir el tiempo de cómputo. Ellas, generalmente se orientan a reducir el número de semejanzas calculadas [11, 9]. En el caso de la clasificación de documentos de texto, los mejores resultados se han obtenido a costa de reducir la calidad de la clasificación y algunos casos haciendo uso de estructuras de datos muy complejas.

Con el advenimiento de la computación paralela utilizando dispositivos gráficos, se han publicado algunos trabajos interesantes sobre optimizaciones de métodos para acelerar el algebra matricial dispersa. En particular algunos como [3, 2, 12] han marcado un punto de comparación en este sentido. Se han revisado publicaciones de este tipo en el área de la Minería de Datos, en general orientados al problema del ordenamiento de las semejanzas entre objetos que determina los k objetos de entrenamiento utilizados en el algoritmo para dar un resultado [1].

Especificamente sobre kNN se conocen algunos trabajos entre los que se destacan [7, 6], los que en su mayoría aplican varias optimizaciones en los pasos del algoritmo pero siempre aplicados a datos densos y de relativamente poca dimensionalidad como es el caso de imágenes. No se conoce hasta el momento trabajos que procesen documentos textuales, los que constituyen un reto en cualquier arquitectura por su alta dimensionalidad y dispersidad.

En este trabajo se proponen dos métodos diseñados tomando en cuenta las peculiaridades de la representación computacional de los documentos de texto, que calculan en paralelo la semejanza entre los objetos de entrenamiento y los objetos a clasificar. Uno de los métodos calcula la semejanza entre la matriz de aprendizaje y un documento de prueba y el segundo entre un conjunto de documentos de prueba. Ambos métodos se utilizaron en la implementación de un proceso de clasificación kNN.

3 CUDA

Los chips gráficos de computadoras son hoy posiblemente, el hardware computacional más potente por unidad de dólar. Estos chips conocidos como Graphic Processing Unit (GPU, por sus siglas en inglés), han transitado desde periféricos hasta procesadores modernos, potentes y programables, por derecho propio.

Medidos con los indicadores tradicionales de rendimiento gráfico, la tasa de crecimiento de las GPU supera la muy citada Ley de Moore que se aplica a los microprocesadores tradicionales; en comparación con una tasa anual de aproximadamente 1.4x de rendimiento de CPU como se muestra en la Figura 1[1], el rendimiento del hardware gráfico practicamente se duplica cada 6 meses.

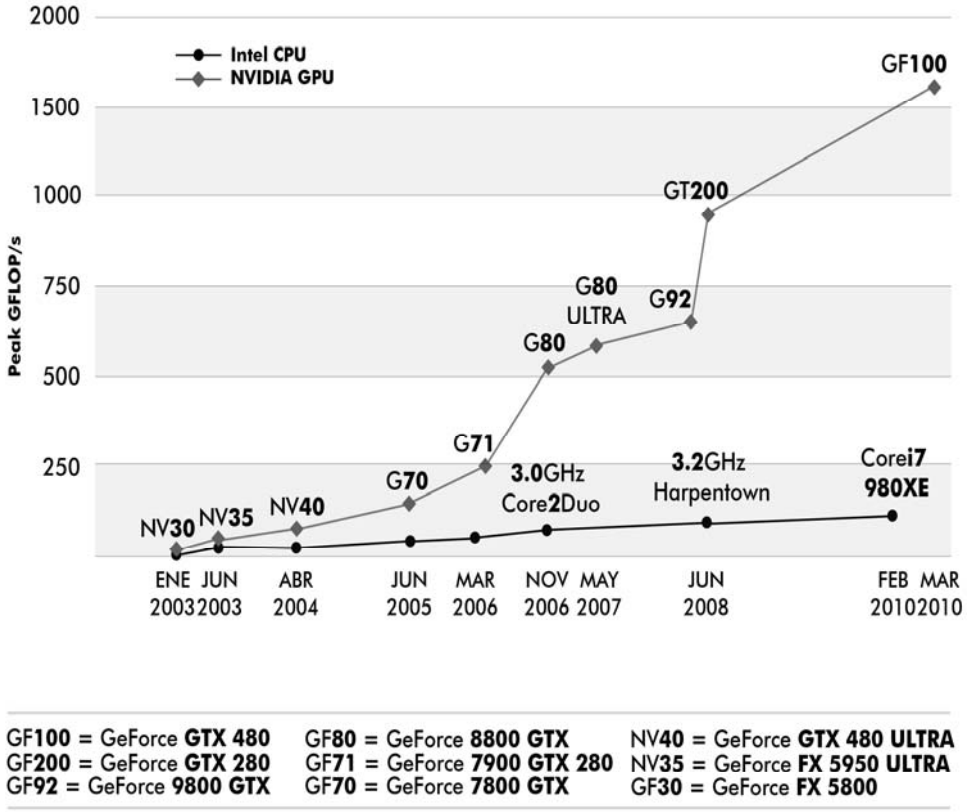


Fig. 1. Comparación del aumento del rendimiento entre GPU y CPU

[1]Basado en la diapositiva 7 de S. Green, "GPU Physics," SIGGRAPH 2007 GPGPU Course.
http://www.gpgpu.org/static/s2007/slides/15-GPGPU-physics.pdf

Este poder computacional está disponible y es barato. Una tarjeta de última generación se puede encontrar en los mercados informáticos a precios que oscilan entre $400 y $500 y el precio cae rápidamente a medida que se libera un nuevo hardware en el mercado.

En noviembre de 2006, NVIDIA introdujo CUDA, una arquitectura de cómputo paralelo de propósito general, con un nuevo modelo de programación y un conjunto de instrucciones, que le permite al motor paralelo de las GPU de NVIDIA resolver muchos problemas computacionales complejos, de forma más eficiente que en una CPU.

CUDA, Arquitectura de Cálculo Unificada (Compute Unified Device Architecture, en inglés), cuenta con un entorno de desarrollo de software que permite a los programadores, utilizar C como un lenguaje de programación de alto nivel.

El modelo de programación de CUDA, está diseñado para desarrollar aplicaciones de software que escalen su paralelismo transparentemente y aprovechen el creciente número de núcleos de procesador, mientras mantiene una curva de aprendizaje baja para programadores familiarizados con la programación en lenguajes estándar como el C.

CUDA tiene tres abstracciones, una jerarquía de grupos de hilos, memorias compartidas y barreras de sincronización, que son expuestas a un programador como un conjunto mínimo de extensiones del lenguaje. En este modelo una GPU ejecuta código implementado en una extensión de C que permite la transferencia de datos entre la CPU y la GPU.

El código paralelo, denominado "Kernel" es asignado al dispositivo como una matriz de bloques de hilos como se muestra en la Figura 2. Los bloques de hilos conteniendo cientos de hilos se despachan a un Stream Multiprocessor (SM) para su ejecución. Los hilos de un bloque se agrupan en conjuntos de 32 hilos denominados Warp, estos avanzan bajo un modelo de ejecución SIMT (Single Instruction, Multiple Threads) [10].

En cuanto a la memoria, los Registros son la estructura más rápida pero solo pueden ser accedidos por hilos. Cada SM posee una Memoria Compartida de 16KB, que puede ser utilizada por los bloques de hilos. La Memoria Global es la memoria principal en el dispositivo gráfico. La Memoria Constante y de Textura tienen la misma velocidad de la Memoria Global, pero son de solo lectura además de estar cacheadas en el SM, lo que las hace mucho más eficientes en determinadas situaciones. En la Figura 3 se muestra gráficamente la distribución de la jerarquía de memoria en CUDA.

Un kernel ejecuta un mismo código en muchos hilos organizados en bloques. Los hilos de un bloque cooperan entre sí mediante una memoria compartida.

A la ejecución de un kernel se especifican la cantidad de bloques e hilos por bloques a ejecutar. La cantidad en cada una de estas dimensiones está determinada por el problema específico que se intente resolver, aunque por problemas prácticos las implementaciones conocidas tienen límites máximos definidos.

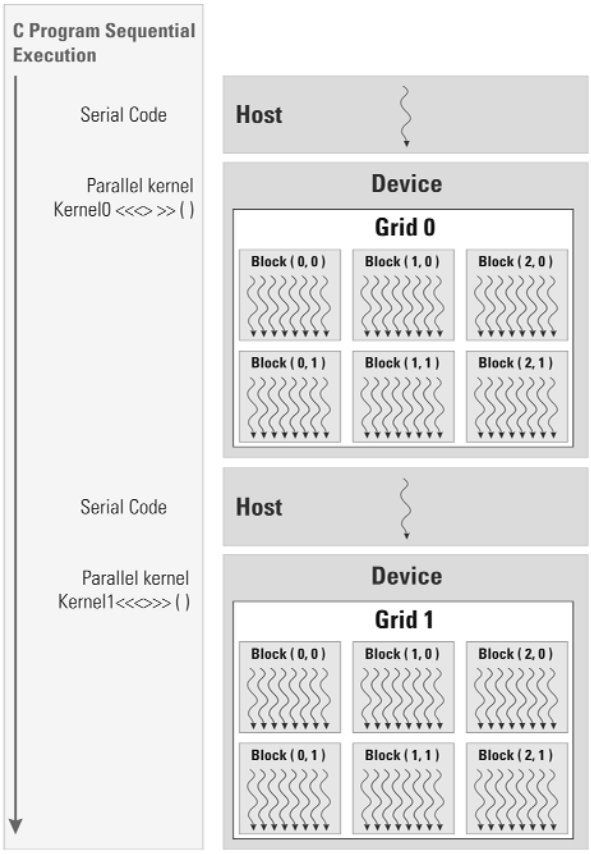


Fig. 2. Jerarquía de hilos en CUDA

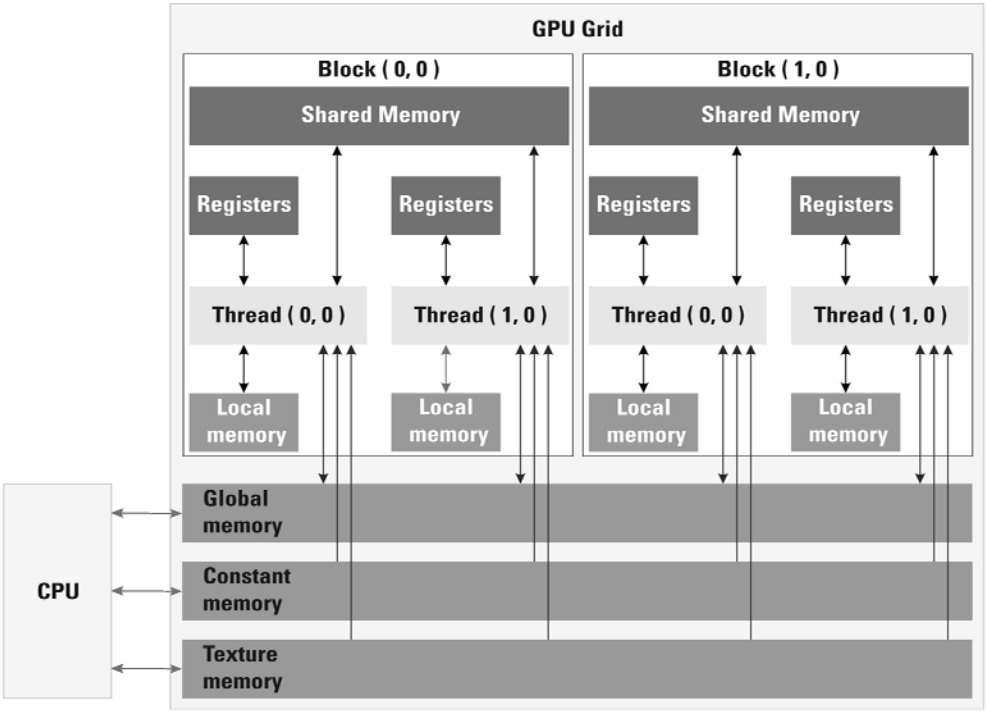


Fig. 3. Jerarquía de memoria de CUDA

Los hilos son creados en hardware y se planifican hasta que todos terminan su ejecución. En los kernel se puede acceder a un conjunto de variables que identifican la posición dentro de la matriz de hilos de ejecución, con ellas se puede de forma muy sencilla implementar un mecanismo de ejecución paralela tipo SIMD (Single Instruction Multiple Data, en inglés) mediante el cual se garantiza aplicar una misma operación a diferentes datos simultaneamente.

En el caso particular de CUDA, la ejecución simultánea se realiza dentro de un warp, que no es más que un conjunto de 32 hilos de bloque que ejecutan atómicamente una operación de forma paralela. El tamaño de un warp es fijo en la arquitectura y en [10] se exponen algunas explicaciones al respecto. El modelo de ejecución permite la ejecución simultánea de los 32 hilos a nivel de hardware y la duración de la operación se determina por el último hilo en terminar.

4 Semejanza entre documentos

Como se menciona anteriormente, la etapa que más tiempo consume en los clasificadores NN, es el cálculo de la semejanza entre el documento de prueba y los documentos del conjunto de entrenamiento. Los clasificadores comunes y los algoritmos de aprendizaje, no pueden procesar directamente los documentos de texto en su forma original. Por lo tanto, en un paso de preprocesamiento, los documentos se convierten en una representación vectorial para el cálculo.

Normalmente, los documentos se representan por vectores de características o rasgos. Un rasgo es simplemente una dimensión en el espacio de características. El documento es representado como un vector en este espacio, es decir, una serie de características y sus ponderaciones o valores de relevancia.

El modelo más común utiliza todas las palabras en el documento como rasgos, y por lo tanto la

dimensión del espacio de características es igual al número de palabras diferentes entre todos los documentos. Los métodos para asignar peso a los rasgos pueden variar. En sistemas más complejos, es posible que tengan en cuenta la frecuencia de la palabra en el documento, en la categoría, y en toda la colección. El esquema más común, TF-IDF, le da a la palabra w en el documento d el peso:

$$W(w,d) = TF(w,d) \cdot \log\big(N/DF(w)\big) \quad (1)$$

donde $\mathrm{TF}(w,d)$ es la frecuencia de la palabra en el documento, N es la cantidad de documentos de la colección, y $\mathrm{DF}(w)$ es el número de los documentos que contienen la palabra w.

De forma general un documento puede tener por sí solo una dimensión relativamente elevada, de forma que la dimensionalidad de una colección se convierte en un serio problema. Esto hace que el proceso de calcular la semejanza entre documentos sea extremadamente costoso.

Para el cálculo de la semejanza en documentos, la métrica más popular es la semejanza Coseno. El cálculo de la semejanza Coseno entre dos documentos está definido por la siguiente fórmula [5]:

$$Sem(d_i, d_j) = (w_i \cdot w_j) = \sum_{k=1}^{m} (w_{ik} \cdot w_{jk}) \quad (2)$$

Donde w_k es el vector normalizado

$$w_k = \frac{d_k}{\| d_k \|} \quad (3)$$

Como se puede apreciar, la fórmula 2 es una suma de productos de dos vectores de términos. Además, un conjunto de n documentos en un espacio de dimensión m, no es más que una matriz M de n filas y m columnas, donde cada columna corresponde a un término en particular y el valor $w_{ij} \in M$ es el peso del término j en el documento i. La matriz M es, de forma general, una estructura dispersa.

Si tenemos en cuenta que la dimensión de un conjunto de documentos es muy elevada, es lógico que cada documento contendrá sólo un número de términos relativamente menor a la dimensión del conjunto, provocando que por cada fila de la matriz M exista gran cantidad de ceros. Por tanto, es posible afirmar que calcular las semejanzas entre un documento de prueba y los documentos de entrenamiento, puede ser reducido a multiplicar una matriz por un vector, además tanto la matriz como el vector son dispersos, porque ambos se definen sobre el mismo espacio de características.

Si tenemos en cuenta que la dimensión de un conjunto de documentos es muy elevada, es lógico que cada documento contendrá sólo un número de términos relativamente menor a la dimensión del conjunto, provocando que por cada fila de la matriz M exista gran cantidad de ceros. Por tanto, es posible afirmar que calcular las semejanzas entre un documento de prueba y los documentos de entrenamiento, puede ser reducido a multiplicar una matriz por un vector, además tanto la matriz como el vector son dispersos, porque ambos se definen sobre el mismo espacio de características.

5 Propuesta

La paralelización de la multiplicación dispersa de matrices se ha realizado siguiendo varias estrategias y con varias herramientas tanto de software como de hardware. Este trabajo, concentró su esfuerzo en realizar el proceso de cálculo sobre la arquitectura gráfica de una GPU con soporte para NVIDIA CUDA.

Para una mejor comprensión de los términos expuestos en esta sección, nos referiremos con MA a la matriz de documentos de aprendizaje y con MP a la matriz de documentos de prueba.

Algoritmo 1: *Clasificación kNN multiplicando matrices dispersas*

1. Copiar MA al dispositivo GPU
2. $nd \leftarrow$ Cantidad de documentos en MP
3. ***for*** $i = 0$ ***to*** nd ***do***
4. Copiar documento $d_i \in MP$ al dispositivo GPU
5. Obtener vector de semejanzas $y_i = MA * d_i$
6. Copiar y_i a CPU
7. Aplicar regla kNN
8. Clasificar d_i
9. ***endfor***

La implementación propuesta en este trabajo, consiste en un proceso de clasificación de documentos que realiza el cálculo de semejanzas entre documentos mediante una función de

multiplicación de matrices dispersas, donde a partir de ellas se clasifica atendiendo a los pasos del algoritmo kNN La descripción de la implementación se muestra en el Algoritmo 1.

Utilizando este esquema se proponen dos algoritmos para multiplicar matrices dispersas con los que se resuelve de forma paralela el cálculo de las semejanzas entre los documentos de prueba y los documentos de la matriz de aprendizaje.

Las implementaciones de algoritmos sobre matrices dispersas apoyan gran parte de su estrategia en una estructura de datos convenientemente seleccionada. En la literatura existen varias representaciones como se muestra en [3] y se destaca por su generalidad la representación dispersa CSR (Compressed Sparse Row, en inglés).

La representación CSR la componen tres vectores que almacenan los índices de inicio y fin de cada fila, los índices de las columnas y los valores distintos de cero existentes en la matriz. La figura 4 muestra una ejemplo de matriz dispersa que ilustra la estructura CSR; el vector denotado por data almacena los valores distintos de cero que contiene la matriz, el vector row almacena los índices dentro de data donde comienza cada fila y cols el índice de la columna de la matriz A donde se encuentra cada uno de los valores de data respectivamente.

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$data = \{1,7,2,8,5,3,9,6,4\};$$

$$row = \{0,2,4,7,9\};$$

$$cols = \{0,1,1,2,0,2,3,1,3\};$$

Fig. 4. Ejemplo de una matriz dispersa CSR

Para establecer un punto de comparación se utilizó un resultado obtenido en una publicación técnica de NVIDIA [3], donde se propuso un algoritmo de multiplicación de una matriz dispersa por un vector denso. Este algoritmo propone una implementación de multiplicación de una matriz dispersa CSR por un vector denso con la que se obtuvieron reducciones de tiempo de ejecución de hasta 69% comparado con el tiempo secuencial de CPU. Los resultados de este algoritmo fueron obtenidos con las mismas configuraciones y datos que se utilizaron en la evaluación de los algoritmos propuestos.

5.1 Multiplicación usando un vector disperso y memoria compartida

Resolver el acceso a los datos en la programación de algoritmos adaptados para GPU, es uno de los puntos que más incidencia tiene en el resultado final. La memoria global es la de mayor latencia de toda la jerarquía de memoria de un dispositivo gráfico y su máximo aprovechamiento está condicionado al patrón de acceso de lectura o escritura [10]. Por ejemplo en el diseño G80, cada instrucción toma 4 ciclos de reloj en procesarse y una operación de lectura en la memoria global toma 200 ciclos. Por tal motivo es recomendable garantizar el acceso combinado de los hilos de un warp a la memoria, de otra manera, todas las operaciones de memoria se serializarán influyendo directamente en el rendimiento de la aplicación.

Paralelizar la multiplicación dispersa no es una tarea fácil para arquitecturas gráficas. En nuestro caso, los patrones de acceso a la memoria afectan con mucha influencia el rendimiento general del kernel. En general, es casi imposible para la multiplicación dispersa tratar de combinar las operaciones de lectura en la memoria global, por tanto, una solución es usar alguna de las memorias internas del dispositivo para realizar operaciones desde ellas. En este caso utilizamos la memoria compartida.

La memoria compartida tiene una latencia mucho menor que la memoria global, de modo que los problemas ocasionados por los patrones de acceso se ocultan, permitiendo alcanzar un buen rendimiento.

Por lo general en este tipo de problema se multiplica una matriz dispersa por un vector denso para reducir la complejidad del algoritmo desde el punto de vista de la programación. En el caso particular de las representaciones de documentos este vector presentaba un alto grado de dispersidad, por lo que nuestra propuesta se concentra además en la multiplicación de una matriz por un vector ambos dispersos.

En el Algoritmo 2, se muestran los pasos de la nueva implementación. En éste, cada fila de la matriz MA, será procesada por un hilo de la GPU. Es sencillo notar que las filas de MA se recorrerán

una sola vez por un único hilo, mientras que el vector a multiplicar será recorrido por todos los hilos planificados. Además, cada hilo seguirá un camino distinto en dependencia de los datos utilizados.

Algoritmo 2: Obtención del vector de semejanzas

Requiere: $data,\ col,\ ptr,\ num\_cols$ representación CSR de MA
Requiere: $vdata,\ vidx$ representación dispersa de d_i
1. Copiar $vdata$ y $vidx$ a memoria compartida
2. $row \leftarrow blockDim * blockIdx + threadIdx$
3. $sum \leftarrow 0$
4. $row\_start \leftarrow ptr[row]$
5. $row\_end \leftarrow ptr[row + 1]$
6. $lidx \leftarrow row\_start$
7. $ridx \leftarrow 0$
8. ***while*** $lidx < row\_end$ ***and*** $ridx < num\_cols$ ***do***
9. ***if*** $col[lidx] = vidx[ridx]$ ***then***
10. $sum \leftarrow sum + data[lidx] * vdata[ridx]$
11. $lidx \leftarrow lidx + 1$
12. $ridx \leftarrow ridx + 1$
13. ***elseif*** $col[lidx] < vidx[ridx]$ ***then***
14. $lidx \leftarrow lidx + 1$
15. ***else***
16. $ridx \leftarrow ridx + 1$
17. ***endif***
18. $y[row] \leftarrow sum$
19. ***endwhile***

De esta forma, inicialmente colocamos la representación dispersa del vector a multiplicar $\mathrm{d_i}$ en la memoria compartida, garantizando un acceso de muy baja latencia en un patrón de acceso no combinado. Luego, dado que la representación del $\mathrm{d_i}$ es dispersa, entonces se debe garantizar que se multipliquen sólo los elementos que se encuentran en el mismo índice de fila respecto a la matriz o al vector real, según sea el caso.

La implementación se realizó mediante un mecanismo de pivotes, en el que se fija el mayor entre los índices de la fila de MA y el $\mathrm{d_i}$ y se va incrementando el menor, de tal forma que cuando ambos sean iguales se acumule la multiplicación de los valores de cada uno. Si al incrementarse el menor de los índices supera al pivote, estos se intercambian y se repiten los pasos anteriores de la misma forma. El proceso de nivel superior que hace uso de este kernel es similar al descrito en el Algoritmo 1.

Todos estos detalles de implementación permitieron que aumentara el rendimiento del kernel y que en general se hiciera uso de un mayor ancho de banda de memoria, además la calidad de la clasificación se mantuvo igual. Este algoritmo se ajusta al caso en que no se conozcan a priori la cantidad de documentos de prueba a clasificar por lo que el cálculo de las semejanzas se realizaría secuencialmente entre los documentos de prueba.

5.2 Multiplicación de matrices dispersas usando caché de texturas

Los dispositivos gráficos están diseñados para soportar un alto grado de paralelismo, representado a nivel de hardware en varios Stream Processors (SP, por sus siglas en inglés), cada uno con la capacidad de ejecutar una gran cantidad de hilos en paralelo. De forma general estos dispositivos organizan este conjunto de hilos en estructuras jerárquicas que por lo general pueden tener hasta tres dimensiones.

En particular, los productos de NVIDIA se organizan en una jerarquía de tres niveles, donde el nivel superior llamado Grid, consiste en una matriz de bloques de hilos de dos dimensiones; cada bloque de hilos, a su vez, es una matriz de hilos de tres dimensiones.

Esta jerarquía permite distribuir el trabajo, dependiendo del algoritmo o la capacidad de particionar de manera equitativa los datos sobre los que actuará un problema determinado. En el algortimo propuesto anterior, se describió una solución al problema de la multiplicación de una matriz dispersa con un vector disperso. En ella se utilizó un bloque de hilos de una sola dimensión y se distribuyeron los datos de forma que en cada hilo se calcula la multiplicación del vector disperso y una fila de la matriz, de modo que, el procesamiento de la colección de documentos se realiza secuencialmente de uno en uno.

Este método realiza los cálculos entre varios vectores de documentos de prueba y la matriz de aprendizaje, lo que se traduce en la multiplicación de dos matrices dispersas, la matriz de aprendizaje MA y una matriz de vectores de prueba dispersos, denominada en lo adelante MV, donde $\mathrm{MV} \subseteq \mathrm{MP}$.

De esta forma, se disminuye en cierta medida la latencia de las transferencias de datos entre el dispositivo y la CPU, ya que en cada transferencia se mueven más vectores dispersos a procesar. Por

otro lado el kernel implementado utiliza un bloque de hilos de dos dimensiones con el que se emplea una mayor cantidad de hilos en ejecución en la GPU, aumentando el ancho de banda de memoria, y por consiguiente, el rendimiento del proceso completo.

Una particularidad añadida por el hecho de multiplicar varios vectores dispersos en paralelo, es que en cada uno de ellos se accederá a las mismas filas de la matriz de aprendizaje ya que cada uno debe calcular su semejanza con cada documento de ésta. De forma general, no se puede garantizar un patrón de acceso eficiente con esta implementación, ya que estos patrones están diseñados para optimizar los accesos dentro de un bloque de hilos y en especial entre hilos de un mismo warp.

Para optimizar este acceso a la memoria hemos utilizado la memoria de textura. Esta memoria, está optimizada para reducir la demanda de ancho banda cuando los datos se encuentran espacialmente cercanos, actuando como un caché.

La esencia del método es similar al expuesto en el Algoritmo 2. Las diferencias más importantes se encuentran en el uso de la memoria de textura, cuando se referencian los datos de las filas de la matriz de aprendizaje MA y en el direccionamiento de los índices de los hilos de ejecución, ya que los bloques de hilos empleados son de dos dimensiones y los índices globales de cada hilo se tienen que calcular tomando en cuenta esta estructura.

La nueva implementación se muestra en el Algoritmo 3, en ella se utilizan bloques de hilos de dos dimensiones que se representan por las variables internas de CUDA $\mathrm{blockIdx.x}$ y $\mathrm{blockIdx.y}$, con las que se identifica la fila de la matriz de documentos de prueba (ver línea 1), que se va a utilizar para calcular la multiplicación con cada fila de MA, referenciada por la segunda variable respectivamente (ver línea 4).

Los pasos a seguir en cada uno de los documentos de MV, es similar al descrito en el Algoritmo 2 además, la recuperación de los datos de MA, se realiza a través del acceso a la memoria de textura, como se muestra en las líneas 11 y 13 mediante la función fetch2. Finalmente, el resultado de cada cálculo se almacena en el vector y de acuerdo a la posición global de las filas de MV y MA multiplicadas, como se muestra en la línea 22.

Este algortimo es más eficiente cuando se conocen a priori la cantidad de documentos de prueba, por lo que al utilizar una mayor cantidad de hilos de forma simultánea el rendimiento general aumenta porque existe más trabajo y disminuye la cantidad de hilos inactivos. Los resultados de la clasificación fueron los mismos que en el algoritmo anterior.

Algoritmo 3: Obtención del vector de semejanzas

Requiere: $MAdata$, $MAcol$, $MAptr$, $MAnum\_rows$ representación CSR de MA
Requiere: $MVdata$, $MVcol$, $MVptr$ representación CSR de MV
1. $i \leftarrow blockIdx.x$ { Seleccionar el vector a multiplicar en el grid i}
2. $num\_cols \leftarrow MVptr[i+1] - MVptr[i]$
3. Copiar datos de d_i a memoria compartida
4. $row \leftarrow blockDim.x * blockIdx.y + threadIdx.x$
5. $sum \leftarrow 0$
6. $row\_start \leftarrow MAptr[row]$
7. $row\_end \leftarrow MAptr[row+1]$
8. $lidx \leftarrow row\_start$
9. $ridx \leftarrow 0$
10. ***while*** $lidx < row\_end$ ***and*** $ridx < num\_cols$ ***do***
11. $col \leftarrow fetch(MAcol, lidx)$
12. ***if*** $col = MVcol[ridx]$ ***then***
13. $sum \leftarrow sum + fetch(MAdata, lidx) * MVdata[ridx]$
14. $lidx \leftarrow lidx + 1$
15. $ridx \leftarrow ridx + 1$
16. ***elseif*** $col < MVcol[ridx]$ ***then***
17. $lidx \leftarrow lidx + 1$
18. ***else***
19. $ridx \leftarrow ridx + 1$
20. ***endif***
21. $y[row] \leftarrow sum$
22. ***endwhile***

6 Resultados experimentales

Cada uno de los algoritmos propuestos, alcanza en mayor o menor grado una ventaja respecto al proceso de clasificación en una CPU. En esta sección se mostrará como la implementación en GPU de los algoritmos propuestos, mejora el rendimiento de la clasificación k NN en documentos, a través de los resultados

[2]Para facilitar la comprensión, esta función describe el proceso de recuperar un dato desde la memoria de textura

experimentales de cada una de las implementaciones y una evaluación de los mismos.

6.1 Configuración experimental

Para los experimentos se utilizaron dos configuraciones de hardware, la primera una computadora personal y la segunda en un servidor profesional. Éstas se utilizaron para mostrar la diferencia en el rendimiento alcanzado teniendo en cuenta la potencia de cálculo en diferentes procesadores gráficos.

El sistema usado para la evaluación del rendimiento en la primera configuración (referenciada como Configuración #1 en lo adelante) consiste en, un procesador Intel Core2 Duo E7300 con una frecuencia de reloj de 2.66 GHz, 2 GB de memoria RAM y una tarjeta NVIDIA GeForce 9800 GT con 112 núcleos y 512 MB de memoria dedicada. La aplicación se generó con el compilador GNU C++ Compiler versión 4.3.4 y NVIDIA CUDA Compiler Driver versión 2.3. La segunda configuración (referenciada como Configuración #2 en lo adelante) consta de dos procesadores Intel Xeon E5405 con una frecuencia de reloj de 2 GHz para un total de 8 núcleos de CPU, 7 GB de memoria RAM y cuatro tarjetas NVIDIA Tesla C870 con 128 núcleos y 1.5 GB de memoria dedicada; la aplicación se generó con el compilador GNU C++ Compiler versión 4.1.2 y NVIDIA CUDA Compiler Driver versión 2.2.

Todos los resultados en la GPU presentados a continuación incluyen el tiempo requerido de transferir los datos desde la memoria principal a la memoria de la GPU y de recuperar los resultados desde el dispositivo. Los kernel operan en simple precisión para datos reales.

Para las pruebas se utilizó la colección estándar Reuters 2000 RCV1 [8]. Esta colección está dividida en conjuntos de archivos de entrenamiento y prueba, cada uno con la distribución de documentos por categoría. El conjunto de entrenamiento contiene 23149 documentos y 47152 términos diferentes, distribuidos en 101 categorías. La matriz de documentos resultantes contiene 1091521648 elementos y su representación CSR contiene 1757801 elementos distintos de cero, lo que representa un 0.16% del total de elementos.

El conjunto de pruebas se dividió en seis subcolecciones para probar la escalabilidad y el rendimiento de la implementación, usando conjuntos de datos de diferentes tamaños. El tamaño total del conjunto es de 781265 documentos, divididos en subconjuntos de 10, 100, 1000, 10000, 100000, y 781265 respectivamente.

6.2 Evaluación del rendimiento

En este estudio, no se muestran los resultados de la calidad de la clasificación kNN, porque en cada prueba, se observó que en ambas arquitecturas fueron los mismos. Por tanto, nos limitamos sólo a evaluar el tiempo de ejecución en segundos para el proceso de categorización completo.

Cada prueba ejecutada con un juego de parámetros, se realizó un total de 10 veces y los resultados finales mostrados son el promedio de todas ellas. Los resultados obtenidos se muestran divididos en tres secciones, cada una representando las tres variantes de algoritmos descritas anteriormente. De cada una de estas variantes, se selecciona la de mejor resultado y se compara con la próxima solución, siempre tomando como referencia en cada una de ellas la ejecución en CPU.

Las pruebas se etiquetaron de la siguiente forma: BaseCSR corresponde a las pruebas del algoritmo secuencial en CPU; las ejecuciones usando el algoritmo propuesto por [3] se muestran con la etiqueta Base y como sufijo la cantidad de hilos por bloque utilizados; las pruebas correspondientes al algoritmo propuesto en la sección 5.1 se etiquetaron como Shared y como sufijo la cantidad de hilos por bloque utilizados; por último el algoritmo propuesto en la Sección 5.2 se muestra como Grid y como sufijo el par que especifica la cantidad de documentos y la cantidad de hilos por bloque utilizados.

Las gráficas de rendimiento están en función de la cantidad de documentos de prueba y en cada uno de los algoritmos se muestra una tabla con los tiempos obtenidos en segundos y gráficamente la aceleración alcanzada por cada prueba, definida como la división entre el tiempo del algoritmo en CPU y el obtenido experimentalmente en la prueba. En cada discusión de resultados se menciona el valor porcentual que representa el tiempo obtenido

por la prueba del tiempo secuencial en CPU, lo que referencia como porciento de reducción.

Los resultados de las pruebas de referencia con el algoritmo propuesto en [3] se muestran en la Tabla 1. En ellas se representan los tiempos de ejecución del algoritmo en CPU y las corridas en GPU para valores de cantidad de hilos por bloque de 32, 64, 128 y 256 hilos respectivamente, en cada una de las configuraciones de hardware especificadas en la sección 6.1. Además se muestra la relación de los tiempos de ejecución respecto a la ejecución del algoritmo secuencial en CPU en la Figura 5.

Tabla 1. Tiempo de ejecución en segundos para el algoritmo propuesto en [3]

	10	100	1000	10000	100000	781265
BaseCSR	0.09	0.7	6.8	67.6	674.67	5283.84
Base32	0.09	0.67	6.4	63.55	632.57	4873.35
Base64	**0.08**	**0.63**	**6.25**	**62.1**	620.72	4944.77
Base128	0.09	0.65	**6.25**	62.11	**620.5**	**4857.4**
Base256	**0.08**	0.64	**6.25**	62.11	620.78	4858.86

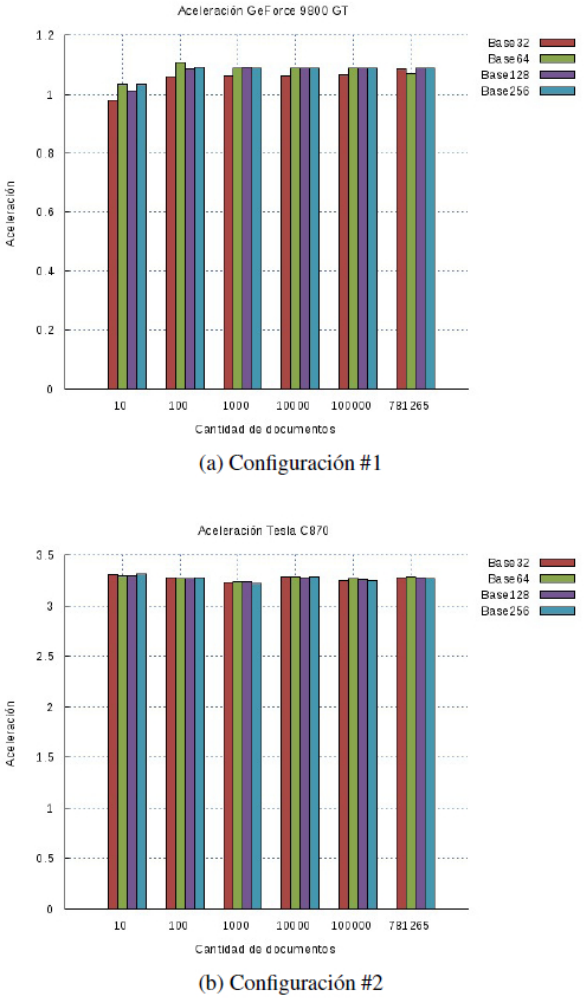
(a) Configuración #1

	10	100	1000	10000	100000	781265
BaseCSR	0.18	1.71	17	171.22	1701.19	13303.8
Base32	**0.05**	**0.52**	5.27	52.16	523.85	4064.09
Base64	0.09	**0.52**	**5.24**	52.18	**521.3**	**4051.86**
Base128	**0.05**	**0.52**	5.25	52.21	521.63	4065.94
Base256	**0.05**	**0.52**	5.28	**52.13**	523.55	4068.07

(b) Configuración #2

En esta implementación, se muestra una ganancia en la aceleración para la Configuración #1, que supera ligeramente los resultados de la CPU, obteniendo como promedio una reducción del tiempo del 7% respecto al tiempo secuencial en CPU. Por otro lado, en la Configuración #2 se nota que se logran alcanzar resultados superiores a 3x, que representan una reducción del tiempo de ejecución de 69.4% como promedio. Los resultados de esta propuesta en general son similares en cuanto a la aceleración alcanzada para ambas configuraciones.

La propuesta definida en el Algoritmo 2 obtuvo mejores resultados, para las dos configuraciones de hardware. En la Tabla 2 se muestran los tiempos de la ejecución para distintos tamaños de bloques de hilos, representados con los valores de 32, 64, 128 y 256, respectivamente. En la Figura 6 se muestran los resultados comparativos de las ejecuciones en CPU y los de la ejecución de mejor rendimiento promedio para cada configuración con el algoritmo propuesto en [3].



(a) Configuración #1

(b) Configuración #2

Fig. 5. Aceleración obtenida por el algoritmo propuesto por [3]

Se puede observar en las Figuras 6a y 6b un aumento en la aceleración respecto al algoritmo en CPU, alcanzando valores promedio de 1.3x que representan una disminución del 28% del tiempo de ejecución para la Configuración #1 y de 4.3x para la Configuración #2 que representa un y 78% de reducción. Estos resultados demuestran la eficacia de utilizar memoria compartida, en casos donde el patrón de acceso no combinado a la memoria afecta el rendimiento de un algoritmo en GPU. Los mejores resultados promedio se obtuvieron con bloques de 64 hilos para la Configuración #1 y de 128 para la Configuración #2.

Tabla 2. Tiempo de ejecución en segundos para el Algoritmo 2

	10	100	1000	10000	100000	781265
Shared32	0.08	0.47	4.93	46.14	458.84	3564.81
Shared64	0.07	0.48	5.12	47.8	475.89	3685.09
Shared128	0.08	0.5	5.34	50.04	488.69	3788.59
Shared256	0.07	0.5	5.33	50.09	497.66	3878.7

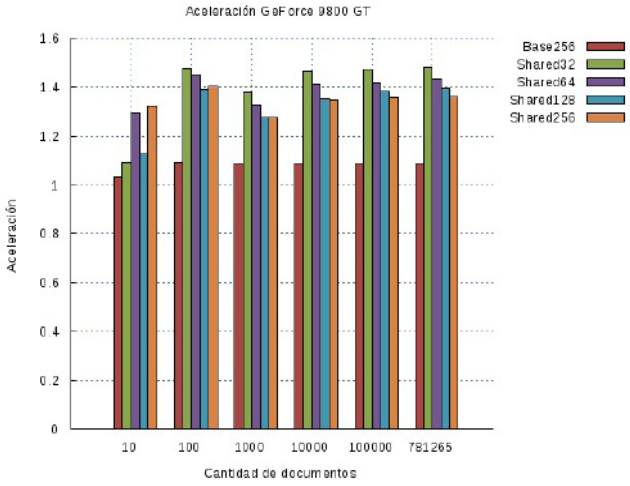
(a) Configuración #1

	10	100	1000	10000	100000	781265
Shared32	0.04	0.4	4.34	40.85	407.37	3154.76
Shared64	0.04	0.4	4.22	39.7	395.44	3071.21
Shared128	0.04	0.39	4.23	39.77	396.16	2144.49
Shared256	0.04	0.39	4.29	40.22	400.05	2793.32

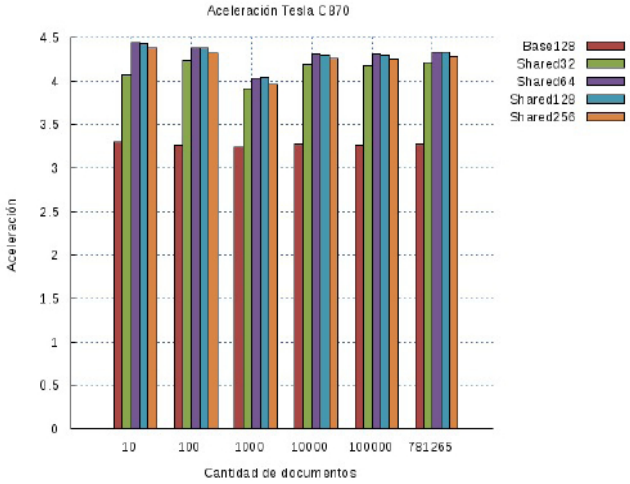
(b) Configuración #2

Para el Algoritmo 3 se obtuvieron los mejores tiempos como se muestran en la Tabla 3 y en la Figura 7 se puede comparar gráficamente estos resultados con la ejecución en CPU y los mejores valores de las pruebas anteriores.

Como se explicó en la Sección 5.2, este algoritmo trabaja con dos dimensiones de hilos en el dispositivo gráfico. Para las pruebas se realizaron ejecuciones fijando la cantidad de documentos en paralelo a comparar (tamaño del grid) en 32, 64, 128 y 256 respectivamente. Por cada uno de los valores de grid se fijó el tamaño de los bloques de hilos en 32, 64, 128, y 256, respectivamente. De este modo las pruebas de este algoritmo equivalen a un total de 16, que por simplificar se muestran los valores de la mejor ejecución por tamaño de grid.



(a) Configuración #1



(b) Configuración #2

Fig. 6. Aceleración obtenida para el Algoritmo 2

Los resultados de esta implementación superan los resultados de la ejecución secuencial en una CPU para ambas configuraciones. En la Figura 7, se muestran los resultados obtenidos, donde se alcanzan valores promedio de aceleración para la

Configuración #1 de 3.6x y para la Configuración #2 de 9x, que representan una disminución del tiempo de ejecución respecto al algoritmo secuencial en CPU del 69% y 89% respectivamente.

Tabla 3. Tiempo de ejecución en segundos para el Algoritmo 3

	10	100	1000	10000	100000	781265
Grid32x128	0.05	0.22	2.1	19.19	192.15	1487.4
Grid64x128	0.05	**0.21**	2.07	**16.1**	**161.02**	**1120.23**
Grid128x128	0.05	0.26	2.01	17.55	176.03	1238.33
Grid256x128	**0.04**	0.26	**1.99**	17.38	176.28	1372.42

(a) Configuración #1

	10	100	1000	10000	100000	781265
Grid32x64	**0.02**	**0.21**	2.18	19.66	197.61	1528.6
Grid64x64	**0.02**	0.24	2.17	19.15	193.75	1501.3
Grid128x128	**0.02**	0.24	1.98	17.74	177.88	1372.05
Grid256x128	**0.02**	0.24	**1.92**	**16.82**	**169.58**	**1304.09**

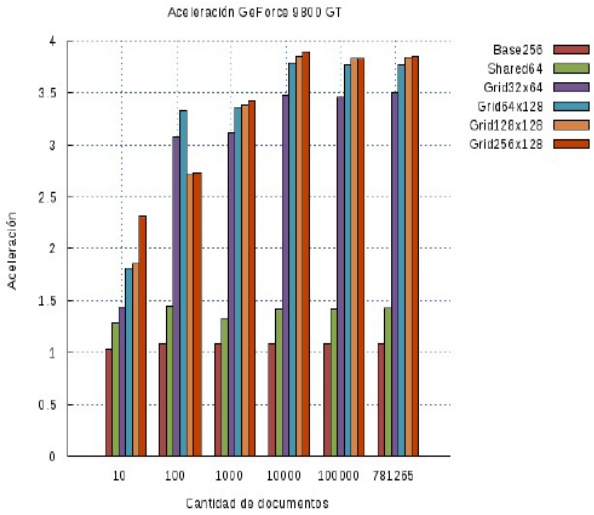
(b) Configuración #2

La comparación de todos los tiempos de ejecución obtenidos con el hardware gráfico en las dos configuraciones, muestra mejores resultados para la Configuración #2, debido a que el dispositivo usado en ella cuenta con una mayor cantidad de núcleos y de memoria. Además este tipo de hardware está dedicado al uso en GPGPU, por ese motivo no cuenta con la salida de video tradicional en estos dispositivos, dedicando toda su potencia al cálculo aritmético.

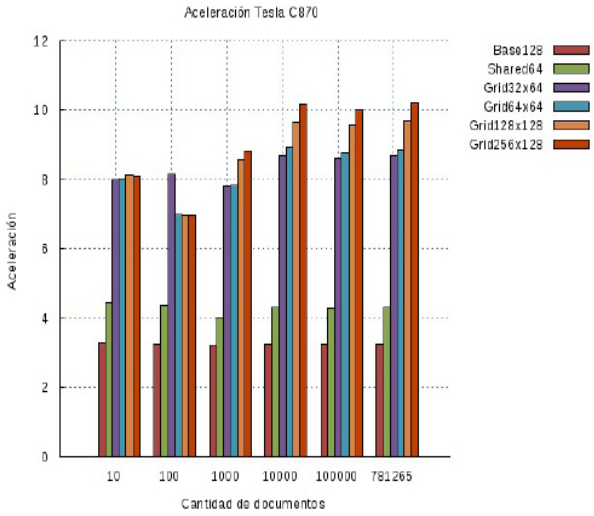
7 Conclusiones

A partir de la investigación realizada, se obtuvieron los siguientes resultados.

Se propuso un algoritmo para multiplicar una matriz por un vector, ambos dispersos, que se puede aplicar a procesos donde se desconozca de antemano la cantidad de vectores a multiplicar. Con este algoritmo se implementó un proceso de clasificación *k*NN con el que se alcanzan reducciones promedio en el tiempo de ejecución, para el hardware utilizado, de 28% y 79% respectivamente respecto a una implementación en CPU.



(a) Configuración #1



(b) Configuración #2

Fig. 7. Aceleración obtenida para el Algoritmo 3

Se implementó un algoritmo para multiplicar dos matrices dispersas, que se puede aplicar además en procesos donde se deseen multiplicar una matriz dispersa y un conjunto de vectores de forma paralela. Con este algoritmo se implementó un

proceso de clasificación k NN, con el que se alcanzan reducciones promedio en el tiempo de ejecución, para el hardware utilizado, de 69% y 88% respectivamente respecto a una implementación en CPU.

Los resultados obtenidos en la investigación, constituyen un aporte para tareas de clasificación de documentos y ofrecen al programador una biblioteca de algoritmos de multiplicación de matrices dispersas para GPU, que queda disponible para futuras investigaciones y para utilización en proyectos de software.

Como trabajo futuro se propone un estudio de otras estructuras de datos para la representación de matrices dispersas, que permitan optimizar el uso de la memoria y el acceso coordinado a la misma, en dispositivos gráficos, para obtener mejores aceleraciones en los procesos implementados. Además se han reportado varios trabajos [7, 6] que utilizan el poder de la GPU para realizar el ordenamiento de las semejanzas luego del computo de las mismas, proponemos la inclusión de una optimización al proceso k NN similar para su utilización en aplicaciones reales que utilicen este algoritmo.

Los algoritmos propuestos en este trabajo se diseñaron especialmente para el uso en arquitecturas de hardware gráfico, y se utilizaron en un proceso de clasificación estático como es el kNN, donde la matriz de aprendizaje se mantiene sin cambios en la memoria gráfica. De modo que se propone el diseño de nuevos algoritmos que administren dinámica y eficientemente la memoria gráfica de las GPU, y adicionen filas a la matriz de aprendizaje. Esto permitirá emplear estos algoritmos en otras tareas de la Minería de Textos como el agrupamiento de documentos.

Referencias

1. **Barrientos, R. J., Gómez, J. I., Tenllado, C. & Prieto M. (2010).** Heap Based k-Nearest Neighbor Search on GPUs. *XXI Jornadas de Paralelismo*, Valencia, España, 559-566.
2. **Baskaran, M.M. & Bordawekar, R. (2009).** *Optimizing Sparse Matrix-Vector Multiplication on GPUs* (IBM Technical Report RC24704). USA: IBM Research Division.
3. **Bell, N. & Garland, M. (2008).** *Efficient Sparse Matrix-Vector Multiplication on CUDA* (NVIDIA Technical ReportNVR-2008-004). USA: NVIDIA Corporation.
4. **Feldman, R. & Sanger, J. (2006).** *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge; New York: Cambridge University Press.
5. **Frakes, W. B. & Baeza-Yates, R. (1992).** *Information Retrieval, Data Structure and Algorithms.* Englewood Cliffs, N.J.: Prentice Hall.
6. **Garcia, V., Debreuve, E., Nielsen, F. & Barlaud, M. (2010).** K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. 17th *IEEE International Conference on Image Processing.* Hong Kong, China, 3757-3760.
7. **Kuang, Q. & Zhao, L. (2009).** A Practical GPU Based KNN Algorithm. *Second Symposium International Computer Science and Computational Technology, Huangshan, China,*151-155.
8. **Lewis, D. D., Yang, Y., Rose, T. G. & Li, F. (2004).** RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research*, 5(2004), 361-397.
9. **Moreno-Seco, F., Micó, L. & Oncina, J. (2003).** Approximate Nearest Neighbour Search with the Fukunaga and Narendra Algorithm and Its Application to Chromosome Classification. *Progress in Pattern Recognition, Speech and Image Analysis. Lecture Notes in Computer Science*, 2905, 322-328.
10. *NVIDIA CUDA™ 2.3 Programming Guide, Version 2.3.1*, 2009
11. **Hernández-Rodríguez, S., Carrasco-Ochoa, J. A & Martínez-Trinidad, J. F. (2007).** Fast k Most Similar Neighbor Classifier for Mixed Data Based on a Tree Structure and Approximating-Eliminating. *Progress in Pattern Recognition, Image Analysis and Applications. Lecture Notes in Computer Science*, 5197, 364-371.
12. **Wang, Z., Xu, X., Zhao, W., Zhang, Y. & He, S. (2010).** Optimizing sparse matrix-vector multiplication on CUDA. *2nd International Conference on Education Technology and Computer (ICETC)*, 109-113.

Rubén Bresler Camps

Es graduado de Ciencia de la Computación en 2005 en la Universidad de Oriente, Cuba. Obtuvo el título de M.S en Ciencia de la Computación en la propia Universidad de Oriente en el año 2010. Profesionalmente comenzó en la empresa DATYS dedicada al desarrollo de software como programador, actualmente es Jefe de Proyecto en la misma

empresa y sus áreas de intereses son el Reconocimiento de Patrones, la Minería de Datos y la Paralelización y Distribución de aplicaciones. Los trabajos en los que ha participado han tenido el asesoramiento del Centro de Reconocimiento de Patrones y Minería de Datos (CERPAMID) asociado a la Universidad de Oriente.

Reynaldo Gil García

Es Profesor Titular del Departamento de Ciencia de la Computación en la Universidad de Oriente, Cuba. Figura como parte de los autores en varios artículos científicos presentados en eventos de importancia internacional, la mayor parte se encuentran publicados en LNCS. Sus áreas de interes son los Algoritmos de Agrupamiento y Clasificación de documentos y la Paralelización de algoritmos.