ISSN 1870-4069

Implementación de *kNN* sobre un *GPU* para predicción de la velocidad del viento

Hector Rodriguez Rangel[1], Glenn Della Rocca[1], Juan J. Flores[2], Luis A. Morales Rosales[3], Nora E. Cancela García[1]

[1]Instituto Tecnológico de Culiacán,
División de Estudios de Posgrado e Investigación,
Mexico

[2] Universidad Michoacana de San Nicolás de Hidalgo,
Facultad de Ingeniería Eléctrica,
División de Estudios de Postgrado, Michoacán,
Mexico

[3] CONACYT-Universidad Michoacana de San Nicolás de Hidalgo,
Facultad de Ingeniería Civil, Morelia, Michoacán,
Mexico

{hrodriguez, glenndellarocca}@itculiacan.edu.mx, juanf@umich.mx,
amorales@conacyt.mx, noracancela@gmail.com

Resumen. El algoritmo *kNN* se encuentra entre los primeros 10 mejores algoritmos de minería de datos. Puede ser utilizado en muy diversas áreas como regresor o pronosticador. Entre las desventajas que posee el algoritmo se encuentran los tiempos de ejecución largos y la dependencia a un valor óptimo de *k*. En este trabajo se propone la implementación del algoritmo *kNN* sobre un *GPU*, para dar solución a los problemas inherentes al mismo. Se hace uso del lenguaje de programación *Python* en conjunto con la librerías *PyCUDA* y *NumPy*, además del framework *CUDA* de *Nvidia* para la programación paralela. Se utilizó *kNN* como pronosticador y se evaluó con series de tiempo de la velocidad del viento. Los resultados experimentales demuestran grandes mejoras en los tiempos de ejecución de la implementación paralela con respecto a la versión secuencial, manteniendo la misma calidad en los resultados finales (i.e., error de predicción).

Palabras clave: Predicción, velocidad del viento, series de tiempo, KNN, GPU, paralelización.

Parallel Implementation of *kNN* on a *GPU* for Wind Speed Forecasting

Abstract. The *kNN* algorithm is amongst the top ten algorithms in data mining. It can be used on a wide range of areas for classification

or forecasting. Some of the disadvantages the algorithm has are the long execution times and the need for an optimum k value. This research proposes an implementation of the algorithm running on a graphics card (GPU), to fix the algorithm's problems. The implementation uses the *Python* programming language, the libraries *PyCUDA* and *NumPy* and the parallel part of the code is programmed with *Nvidia's CUDA* framework. The algorithm was used as a forecaster and evaluated with wind speed time series. The experimental results shows big improvements on execution times for the parallel implementation in contrast with the sequential one, keeping up the quality of the final results (i.e., error prediction).

Keywords: Forecasting, wind speed, time series, KNN, GPU, parallel.

1. Introducción

Los pronósticos son estimaciones sobre eventos futuros. La acción de pronosticar conlleva una predicción sobre algo que aún no existe. Son utilizados en diferentes campos y/o áreas (procesos de planeación, facturación, mantenimiento, etc.). Esto hace que sean útiles en campos como los negocios, la industria, la economía, las ciencias medio ambientales, ciencias sociales, política, medicina y finanzas, entre otros.

Existen dos enfoques para los métodos de pronóstico, el enfoque cualitativo y el enfoque cuantitativo [6]. El enfoque cualitativo hace uso de técnicas no estadísticas como la intuición y la experiencia del usuario para realizar estimaciones. Por otro lado, los métodos cuantitativos utilizan técnicas estadísticas y métodos de inteligencia artificial (IA), entre otros.

Dentro de los métodos de IA utilizados para hacer pronósticos se encuentran las redes neuronales artificiales (RNA), los k vecinos más cercanos (*kNN*, por sus siglas en inglés), máquinas de soporte vectorial (SVM en inglés), etc.

El algoritmo de k vecinos más cercanos se encuentra entre los diez mejores algoritmos de minería de datos de acuerdo con un estudio hecho por la IEEE en 2006 dentro de la Conferencia Internacional de Minería de Datos (ICDM) [9]. Pero este es poco utilizado para tareas de pronóstico, siendo descartado por algoritmos más complejos debido a problemas inherentes al mismo.

Entre los problemas que presenta *kNN* está la dependencia de un valor óptimo de k, la influencia nociva de atributos irrelevantes, el problema del ruido en los datos y los tiempos de respuesta largos.

En este trabajo se propone un algoritmo que realiza de manera paralela el método de vecinos más cercanos sobre una tarjeta aceleradora gráfica, aplicado al área de predicción. La implementación del algoritmo está basado en el framework *CUDA* de *Nvidia*.

El trabajo está estructurado de la siguiente manera: en la Sección 2 se encuentra un breve repaso de los pronósticos, las técnicas modernas de pronóstico, y trabajos que hacen uso de computación paralela. En la Sección 3 tenemos la descripción y funcionamiento del algoritmo *kNN* como pronosticador.

El modelo de cómputo distribuido de *Nvidia* (*CUDA*) se encuentra en la Sección 4. La descripción de la implementación del algoritmo de manera distribuida está en la Sección 5. Por último, en las Secciones 6 y 7 tenemos los resultados de los experimentos y conclusiones, respectivamente.

2. Trabajos relacionados

Pronosticar es una actividad que se realiza desde la antigüedad. Pero fue poco antes del siglo *XX* que se inició el análisis estadístico de las series de tiempo [11]. La teoría y métodos del análisis de series de tiempo son importantes bases y herramientas para los pronósticos. Ruey S. Tsay en [8] dice que los pronósticos son la razón por la que existen las series de tiempo y el análisis de éstas.

En el área de la industria y con mayor énfasis en el área de la energía, se han realizado trabajos de pronóstico desde hace años. Se utilizan los pronósticos para la planeación de la cantidad de materia prima disponible para las operaciones de las centrales eólicas, para la transmisión de la electricidad, la estabilidad y confiabilidad, entre muchos otros fines [10].

Las técnicas de pronóstico utilizadas para la predicción de la velocidad del viento son muy variadas. Dentro de las utilizadas, de acuerdo al trabajo de Zhao *et al.* [10], se encuentran: predicción numérica del clima (*NWP*, en inglés), métodos estadísticos, métodos de inteligencia artificial (IA), y métodos híbridos.

Entre los métodos de IA más utilizados en la literatura para el pronóstico del viento se encuentran las redes neuronales (*ANN*, en inglés), máquinas de soporte vectorial (*SVM*, en inglés), redes neuronales recurrentes (*RNN* en inglés), búsqueda de vecinos cercanos (*kNN*, en inglés), entre otros [12].

Se puede decir que el algoritmo de *kNN* es ampliamente utilizado para tareas de clasificación y regresión. Sin embargo este algoritmo cuenta con ciertos inconvenientes; uno de sus más grandes problemas es sus largos tiempos de ejecución.

En la literatura se encuentran trabajos que enfrentan este problema mediante la distribución de procesos del algoritmo. Entre otros, encontramos los trabajos de Garcia V. *et al.* [1], Kuang Q. y Zhao L. [2] y Liang S. *et al.* [3], que hacen uso de versiones tempranas del framework *CUDA*. Sin embargo, estas implementaciones tienen como limitantes principales el uso de funciones recursivas y la creación dinámica de hilos en el *GPU*, creando sobrecarga en las soluciones para algoritmos complejos.

El framework *CUDA* de la corporación *Nvidia* [7] es su propuesta para el desarrollo de aplicaciones de cómputo general sobre tarjetas aceleradoras gráficas (*GPU*, en inglés). Los *GPUs* son dispositivos masivamente paralelos, poseen miles de unidades de procesamiento aritmético-lógicas y grandes buses de datos para transferencias rápidas de información.

En la Figura 1 se muestra un diagrama que representa a grandes rasgos las diferencias entre las arquitecturas del *CPU* versus *GPU*. Se aprecia que la arquitectura de un *GPU* se enfoca enteramente en el procesamiento de grandes cantidades de operaciones.

Esto se debe a la gran cantidad de unidades de procesamiento Aritmético-lógicas (*ALU*, por sus siglas en inglés) que posee un solo chip gráfico.

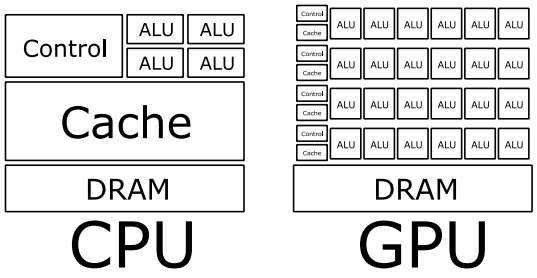

Fig. 1. Arquitectura de un *CPU* común en comparación con un *GPU*. Se observa que el *CPU* posee pocas unidades ALU, mientras que la arquitectura del *GPU* es diseñada pensando en el procesamiento de muchas operaciones simultáneas, con un pequeño *control* y *cache* a diferencia del *CPU* [7].

Teniendo en cuenta las complicaciones para la programación paralela, *Nvidia* desarrolla desde el año 2007 su framework *CUDA* (*Compute Unified Device Architecture*, en inglés). El framework es una extensión del lenguaje de programación C, que permite la utilización de los *GPUs* de la marca para tareas de cómputo general [4].

CUDA esta basado en la arquitectura paralela *SIMD* (*Single Instruction Multiple Data)*, pero haciendo uso de hilos es que la renombra como *SIMT* (*Single Instruction Multiple Threads*) [4].

Estos tipos de arquitectura hardware basan su funcionamiento en la ejecución de una única instrucción sobre un conjunto considerablemente grande de datos. Por lo que con una única unidad de *control* administra una cantidad grande de unidades de procesamiento (*ALUs*). Cada una de estas *ALUs* realiza de manera independiente operaciones en su porción específica de datos, completando la tarea de manera paralela.

3. K vecinos más cercanos

K Vecinos más Cercanos es un algoritmo de aprendizaje máquina (*ML* por sus siglas en inglés) de tipo no paramétrico y de aprendizaje perezoso (*Lazy Learning*). Lo que significa que no hace suposiciones sobre la distribución del conjunto de datos y no realiza ningún trabajo de pre-procesamiento del tipo *entrenamiento*.

Es un algoritmo de clasificación de objetos desconocidos. Utiliza un conjunto de datos etiquetados, que poseen atributos cuantificables. La clasificación se realiza midiendo la semejanza de un objeto a clasificar con el conjunto completo de datos, obteniendo los k objetos más similares, para definir una etiqueta a partir de estos.

En la Figura 2 se muestra un ejemplo de clasificación, en donde se aprecia que existen dos tipos de clase (Pentágonos y Cuadrados). Con un número k arbitrario se clasifica el objeto desconocido o hipótesis de acuerdo a los individuos más cercanos. Se puede observar si se utiliza la Moda, para $k = 3$ el objeto se clasificaría como pentágono y para $k = 7$ sería cuadrado. Mostrando la dependencia del algoritmo hacia un valor óptimo de k.

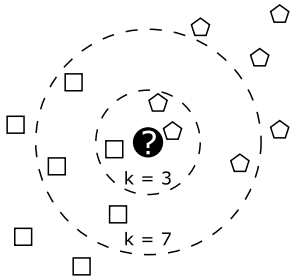

Fig. 2. Algoritmo de clasificación *kNN*. Se observa en el diagrama la dependencia hacia un valor óptimo de k del algoritmo. En este ejemplo en concreto, siendo $k = 3$ el objeto a clasificar sería un pentágono, mientras que con $k = 7$ sería un cuadrado.

La simplicidad del algoritmo permite su utilización para realizar regresión (la tarea de pronóstico se puede plantear como un problema de regresión). Además, esta simplicidad es la que hace que sea utilizado en un amplio rango de campos que van desde: visión computacional, geometría computacional, grafos, entre muchos otros.

El algoritmo asume que los datos se encuentran en un espacio de características y que los puntos de datos se pueden ubicar en un espacio métrico. Los datos pueden ser escalares o vectores multidimensionales, pero deben tener una noción de *distancia*; la métrica de la distancia *Euclidiana* es la más comúnmente utilizada.

Usando la regresión con *kNN*, el pronosticador de series de tiempo puede resolverse de manera similar a la clasificación. Al pronosticar se hace uso de valores continuos, por lo tanto, todos los valores calculados poseen un margen de error que es medible por distintas funciones.

Para utilizar *kNN* como pronosticador, en este trabajo se crea una base de datos. Esta base de datos (*BD*) generada en el ciclo de las Líneas 1 a 3 del Algoritmo 1 se hace a partir del recorrido de una ventana de w observaciones de la serie de tiempo. Cada una de estas ventanas genera una instancia nueva para la *BD*.

El siguiente paso en el pronóstico es el cálculo de las distancias. En el ciclo de las Líneas 4 a 6 del algoritmo se observa que el cálculo de la distancia se realiza entre la hipótesis y cada una de las instancias de la *BD*.

Como tercer paso, el pronosticador realiza el ordenamiento de las distancias, para la obtención de las k instancias más cercanas (Líneas 7 y 8 del algoritmo

respectivamente). Una vez con las distancias más cortas, se realiza un promedio de las etiquetas de las instancias seleccionadas, como se aprecia en la Línea 9. Este promedio es el resultado del algoritmo.

Algoritmo 1 pronosticador_kNN(serie_tiempo, hipotesis)

```
1: para cada w_observaciones de serie_tiempo hacer
2:     BD ← instancia //Creación de nueva instancia
3: fin para
4: para cada instancia de la BD hacer
5:     distancias ← calculo_distancia(instancia, hipotesis)
6: fin para
7: ordenamiento(distancias) //Llamada a función de ordenamiento
8: instancias_seleccionadas ← k_menores(distancias)
9: pronostico ← promedio(instancias_seleccionadas)
10: regresar pronostico
```

Entre los problemas del algoritmo se encuentran los tiempos largos de ejecución para conjuntos de datos de grandes dimensiones, su dependencia hacia un valor óptimo de k y problemas inherentes a las series de tiempo como los *outliers* o los valores nulos.

Los tiempos de ejecución largos del algoritmo al utilizar conjuntos grandes de datos hacen que el proceso no sea apto para soluciones en tiempo real. Esto es causado por la particularidad del aprendizaje perezoso del algoritmo, ya que no realiza trabajo alguno de pre-procesamiento o modelado de datos. Por lo que para cada pronóstico o clasificación es necesario realizar una gran cantidad de operaciones.

La selección de un valor óptimo de k es otra de las desventajas del método. Un número adecuado de k cambia el resultado obtenido por el algoritmo. Por lo general se selecciona un número arbitrario. Aunque también se puede obtener mediante técnicas de optimización, causando una sobrecarga aún mayor en los tiempos de ejecución.

4. Modelo *cuda* de *nvidia*

La programación paralela no es un invento reciente. Tiene ya bastantes años, incluso desde el surgimiento de los primeros computadores se crearon arquitecturas y modelos paralelos.

Pero fue el auge de la industria de los videojuegos que hizo que las tarjetas aceleradoras gráficas se convirtieran en más que un dispositivo de salida de vídeo. La necesidad de mayor poder de cómputo que requirieron los videojuegos impulsó a la fabricación de *GPUs* cada vez más potentes, abaratando los costos de las tarjetas y haciendo asequible utilizarlas como co-procesador del *CPU*.

Nvidia Corp., la más grande compañía productora de *GPUs*, decide comenzar el desarrollo de una arquitectura unificada para la programación de aplicaciones

de cómputo general [4]; con la generación G80 (o Tesla por su nombre comercial), *Nvidia* crea la primera versión de su framework *CUDA*.

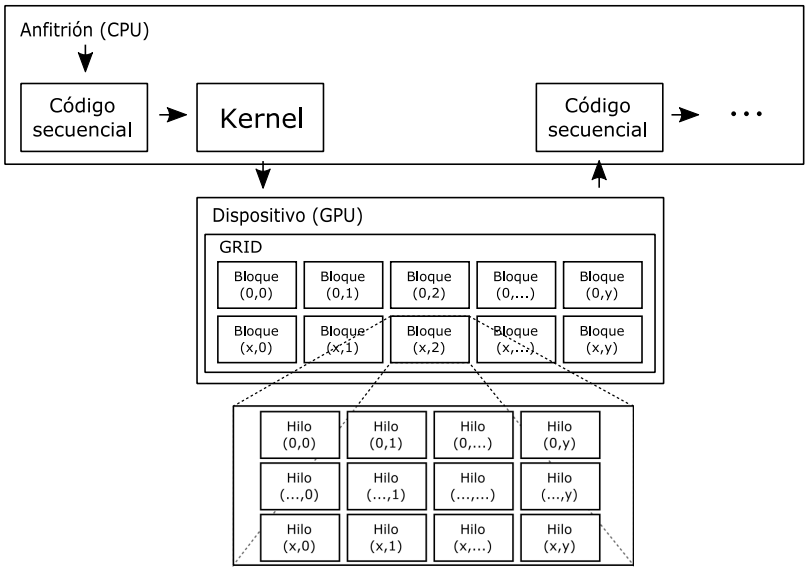

Fig. 3. Proceso de ejecución paralela sobre un *GPU*. El anfitrión es el encargado de las transferencias del código *Kernel* y de los datos hacia la memoria del dispositivo gráfico. Hecho esto, el control pasa al *GPU*, que realiza las operaciones indicadas en la función *Kernel* y regresa el control al *CPU*. Como último paso el anfitrión copia a la memoria principal los resultados dentro de la memoria del *GPU*.

El modelo *CUDA* de *Nvidia* es una colección de hilos ejecutando instrucciones en paralelo. La ejecución de estos hilos es completamente planificada por las unidades de control dentro del *GPU Nvidia*. El proceso de ejecución paralela se basa en la ejecución de las llamadas funciones *Kernel* que no son más que funciones ejecutadas paralelamente por conjuntos de hilos dentro del *GPU*. Estos *Kernels* son programados con el lenguaje de programación C y un conjunto de directivas especificadas por *CUDA*.

En la Figura 3 se aprecia el proceso de ejecución de un *Kernel* sobre un *GPU*. En el diagrama se muestra el modelo paralelo *CUDA*. A nivel software se hace referencia a una cuadrícula o *GRID*, la cual se dispone de acuerdo a las configuraciones del usuario. Se genera una serie de bloques de hilos (*Thread Blocks*, en inglés) y una configuración de bloque (*Block Dimension*, en inglés). Estos dos componentes constituyen la configuración del *Kernel* y son los que determinan el número de hilos a utilizar en la ejecución del código paralelo.

El proceso de ejecución paralela sobre un *GPU* es el siguiente: las funciones *Kernel* llamadas por el anfitrión (*CPU*) y los datos a utilizar son transferidos a través del bus *PCI-Express* hacia la tarjeta gráfica. El planificador de la tarjeta

administra y despacha la ejecución de hilos y datos para la ejecución de manera no secuencial. Los resultados obtenidos de los *Kernels* residen en la memoria del dispositivo y son transferidos como último paso hacia la memoria principal.

5. K vecinos cercanos paralelo

A partir de la programación del algoritmo secuencial se detectaron los cuellos de botella en su ejecución. Con esto realizamos la selección de la función del cálculo de la distancia y del ordenamiento, para su implementación sobre el *GPU*. En esta sección se describirá la implementación distribuida y las principales diferencias con la versión secuencial del algoritmo.

La comparación entre el elemento a clasificar o pronosticar con uno de los elementos del conjunto de *entrenamiento* es una tarea sencilla. Sin embargo con grandes conjuntos de datos el algoritmo realiza una gran cantidad de cálculos, requiriendo más poder computacional y tiempo de ejecución a medida que el número de datos crece o la dimensionalidad de los objetos aumenta.

En las siguientes subsecciones se definirá la arquitectura propuesta para el cálculo de las distancias, así como también para la función de obtención de resultados y de ordenamiento.

5.1. Función de cálculo de la distancia en paralelo

El cálculo de la distancia es la base central del algoritmo *kNN* puesto que evalúa de manera individual cada uno de los objetos dentro del conjunto de *entrenamiento* con cada uno de los objetos a clasificar o hipótesis a predecir.

Para el cálculo de las distancias se puede realizar de distintas maneras. En este trabajo se utilizó la distancia Euclidiana como medida de semejanza entre objetos. Esta distancia se encuentra definida en la ecuación (1), donde o es una instancia particular y h, la hipótesis a comparar:

$$D(o,h) = \sqrt{\sum_{j=1}^{v}(o_j - h_j)^2}, \tag{1}$$

donde v es el tamaño de los objetos. Para el cálculo de la distancia distribuida se generó, como se muestra en la Figura 4, una cantidad de hilos (T_1 a T_n) igual a la cantidad de instancias dentro del conjunto de entrenamiento. Cada hilo calculó la distancia Euclidiana de su respectiva instancia con cada una de las hipótesis (h_1 a h_m) de validación.

Los resultados de estos cálculos son almacenados en la memoria del dispositivo y utilizados como entrada para la función de ordenamiento.

El Algoritmo 2 muestra el proceso del cómputo de la distancia. Las entradas del algoritmo son n (Número de instancias), v (Número de atributos de las instancias), m (Número de hipótesis), $*E$ (Apuntador al vector de entrenamiento), $*V$ (Apuntador al vector de validación), $*D$ (Apuntador al vector de distancias).

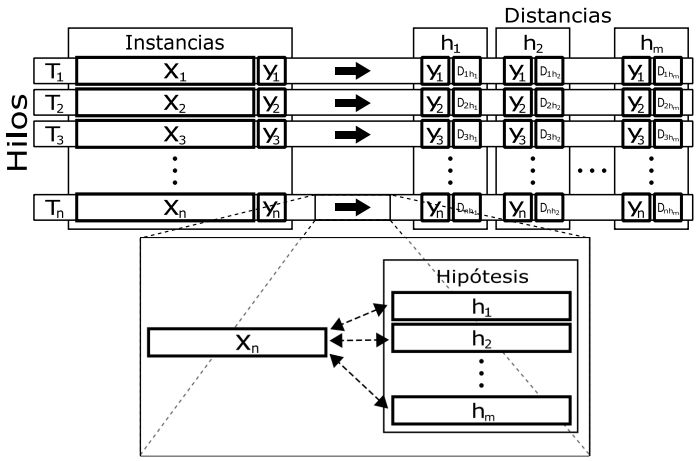

Fig. 4. Cálculo distancia distribuida. El diagrama muestra que se generó una cantidad de hilos igual a la cantidad de instancias en el conjunto de entrenamiento. Cada hilo computó la distancia entre su instancia y todas las hipótesis. Como resultado del cálculo se obtiene una matriz en memoria del *GPU*, de tuplas etiqueta-distancia.

Algoritmo 2 distancia_euclidiana_distribuida(n, v, m, *E, *V, *D)

1: $indice\_hilo \leftarrow (id\_bloque * tamaño\_bloque) + id\_hilo$ //Definición de ID global
2: //Ciclo GRID-STRIDE
3: **para** $indice\_hilo$ hasta n **hacer**
4: $fila\_entrenamiento \leftarrow v * indice\_hilo$
5: $fila\_resultado \leftarrow (m * 2) * indice\_hilo$
6: $etiqueta \leftarrow E[fila\_entrenamiento + v - 1]$
7: //Ciclo de cálculo de la distancia
8: **para** $i \leftarrow 0$ hasta m **hacer**
9: $suma \leftarrow 0.0$
10: $fila\_validacion \leftarrow v * i$
11: //Cálculo de la distancia euclidiana
12: **para** $j \leftarrow 0$ hasta $v - 1$ **hacer**
13: $suma \leftarrow (E[fila\_entrenamiento + j] - V[fila\_validacion + j])^2$
14: **fin para**
15: $indice\_resultado \leftarrow fila\_resultado + (i * 2)$
16: $D[indice\_resultado] \leftarrow etiqueta$
17: $D[indice\_resultado + 1] \leftarrow \sqrt{suma}$
18: **fin para**
19: **fin para**

La Línea 1 calcula el índice global de hilo. Este sirve para determinar la sección de datos específica con la que trabajará el hilo.

El ciclo externo de la función (Línea 3) sirve para trabajar con cualquier número de hilos. Es llamado ciclo *GRID-STRIDE* y permite utilizar cantidades de datos de un tamaño mayor al número total de hilos del *GPU* utilizado.

Cuando el hilo comienza su trabajo se realiza el cálculo de la posición de la fila de entrenamiento y de la de resultado (Líneas 4 y 5). Esto debido a que se utilizó un enfoque de vectores lineales.

En la Línea 6 se realiza la copia de la etiqueta de la instancia utilizada por el hilo. Con los *índices* de posición calculados, el algoritmo recorre todos y cada uno de los objetos del conjunto de validación (Ciclo en Líneas 8 a 18).

En la Línea 10 se calcula la posición del objeto de validación o hipótesis. Un último ciclo (Línea 12) obtiene la distancia euclidiana.

Para guardar los resultados se calcula la posición en el vector de distancias (Línea 15). Como paso final se escriben en el vector de distancias la etiqueta y distancia calculada (Líneas 16 y 17, respectivamente).

5.2. Función de obtención de resultados

Una vez concluido el cálculo de todas las distancias, se procede a obtener los resultados. El proceso de obtención de resultados se realiza a través de dos funciones *Kernel*. La función de obtención de resultados y la función de ordenamiento.

Algoritmo 3 obtener_resultados_distribuido(n, k, l, r, *D, *R)

1: $indice\_hilo \leftarrow (id\_bloque * tamaño\_bloque) + id\_hilo$ //Definición de ID global
2: //Ciclo GRID-STRIDE
3: **para** $indice\_hilo$ hasta n **hacer**
4: $columna \leftarrow indice\_hilo * 2$
5: $ancho\_matriz \leftarrow n * 2$
6: //Cálculo de la posición del pivote y ordenamiento parcial
7: $posicion\_pivote \leftarrow particion(l, r, c, w, D)$
8: $quicksort\_parcial\_distribuido(l, posicion\_pivote - 1, c, w, k, *D)$
9: $resultado \leftarrow 0.0$
10: //Ciclo de promedio
11: **para** $i \leftarrow 0$ hasta k **hacer**
12: $resultado \leftarrow D[(ancho\_matriz * i) + columna]$
13: **fin para**
14: $R[indice\_hilo] \leftarrow resultado/k$
15: **fin para**

La obtención de resultados como se aprecia en la Línea 1 del Algoritmo 3, inicia con el cálculo del índice global, explicado en la sección anterior, para generar un ciclo *GRID-STRIDE* como se detalló anteriormente.

El cálculo de la columna a ordenar por el hilo se realiza en la Línea 4 y el ancho de la matriz para la ubicación de la columna por parte de la función de ordenamiento se realiza en la Línea 5. Se procede a continuación al ordenamiento por medio de la llamada a la selección del elemento pivote de la columna seleccionada (Línea 7).

En la Línea 8 se observa la llamada recursiva hacia la parte izquierda o inferior del vector a ordenar. Esta función será descrita en la sección siguiente. Con el trabajo de ordenamiento concluido de la función *Quick Sort Parcial* [5], el hilo calcula la etiqueta a través del promedio de las etiquetas de los k objetos más cercanos (Línea 11).

Y por último el hilo en ejecución guarda en la posición adecuada del vector de resultados la etiqueta generada (Línea 14 del algoritmo).

5.3. Función de ordenamiento

Para la función de ordenamiento se utilizó la algoritmo *Quick Sort*, un algoritmo rápido y bastante utilizado en la literatura para operaciones de ordenamiento. Pero no se utilizó el algoritmo original, se utilizó una variante llamada *Quick Sort Parcial* [5].

El algoritmo *Quick Sort Parcial* al igual que su contraparte secuencial hace uso de la premisa, divide y conquista. Se seleccionó la variante parcial porque solo son necesarios los primeros k resultados ordenados correctamente. Con esto se evitan movimientos y comparaciones innecesarias, reduciendo aún más el tiempo de ejecución. Aún y cuando el orden de magnitud no cambia ($O(n\ log\ n)$ en el peor de los casos), el tiempo de ordenamiento se reduce en el caso promedio.

Como se aprecia en la Figura 5, para el modelo distribuido del ordenamiento se crearon una cantidad de hilos igual al número de hipótesis del conjunto de *validación*. Esto para el ordenamiento en paralelo de cada una de las columnas de tuplas etiqueta-distancia resultado del *Kernel* de la distancia distribuida.

En versiones iniciales, *CUDA* no permitía la ejecución de funciones recursivas, ni la invocación de hilos hijo que realizaran trabajo de manera más eficiente. Para realizar trabajos complejos era necesario regresar el control al *CPU* frecuentemente, con la consecuente sobrecarga en llamadas y manejo de memoria. Así, en operaciones complejas era difícil hacer un uso óptimo del paralelismo del *GPU*.

El *Kernel* de ordenamiento se encuentra definido en el Algoritmo 4 y como entradas del mismo tenemos: l (Posición izquierda del vector), r (Posición derecha del vector), c (Índice de la columna de datos a ordenar),w (Ancho de la matriz), k (Tamaño de k), $*D$ (Apuntador al vector de distancias).

El algoritmo utiliza ordenamiento por selección cuando el tamaño de datos a ordenar cae por debajo de un límite arbitrario de 32 datos (Línea 2). Esto para evitar una recursión más profunda y la consecuente sobrecarga que esto conlleva.

Si se utiliza *Quick Sort* (Línea 5), el algoritmo hace uso de invocaciones a *Kernels* hijo. La función de partición divide a la mitad el vector y utiliza de pivote el elemento central. Realiza el intercambio de valores de un extremo a otro y regresa la posición del elemento central, todo esto en la línea 7 del algoritmo con la invocación a la función de partición.

Una vez terminado este proceso, se procede recursivamente a la mitad más pequeña de la columna a ordenar (Línea 10). Se continua así hasta que el pivote divido entre 2 sea menor que k (Línea 9). Esto nos permite ordenar correctamente los primeros k elementos y obtener un resultado correcto.

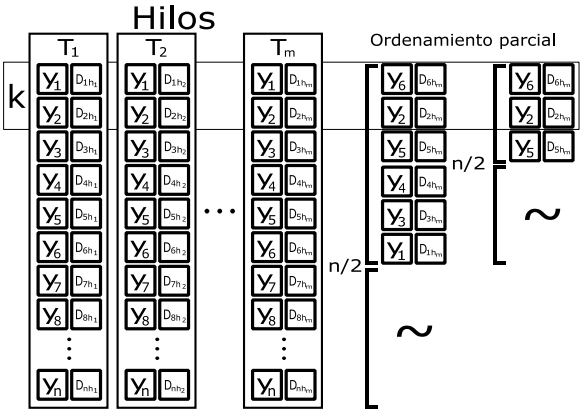

Fig. 5. *Quick Sort Parcial*. En el diagrama se muestra una cantidad de hilos generados igual a la cantidad de columnas dentro de la matriz de tuplas del paso anterior. Cada hilo ejecuta el algoritmo de *Quick Sort Parcial* para el ordenamiento de los datos, resultando en un ordenamiento óptimo de las k instancias más cercanas de cada una de las hipótesis. $\sim$ significa que los datos en ese segmento son mayores que los k menores, por consiguiente ya nos nos interesa ordenarlos.

Algoritmo 4 quicksort_parcial(l, r, c, w, k, *D)

```
1: //Comprobación del tamaño mínimo para el ordenamiento por selección
2: si (r − l) <= 32 entonces
3:    ordenamiento_seleccion(l, r, c, w, D)
4:    regresar
5: fin si
6: si l < r entonces
7:    posicion_pivote ← particion(l, r, c, w, D) //Cálculo del pivote y ordenamiento
8:    //Llamadas recursivas a las particiones del vector
9:    si r/2 > k entonces
10:      quicksort_parcial(l, posicion_pivote − 1, c, w, k, ∗D)
11:   si no
12:      quicksort_parcial(l, posicion_pivote − 1, c, w, k, ∗D)
13:      quicksort_parcial(posicion_pivote + 1, r, c, w, k, ∗D)
14:   fin si
15: fin si
```

6. Resultados

Los datos utilizados durante la experimentación fueron obtenidos de distintos puntos del estado de Michoacán, México. Estos se recopilaron en estaciones meteorológicas con el uso de dispositivos llamados anemómetros. Los cuales son dispositivos que sirven para la medición de la velocidad del viento en intervalos.

Se utilizaron 4 series de tiempo de diferentes tamaños: 8 mil, 22 mil, 32 mil y 43 mil datos (El Fresno, Malpaís, Melchor Ocampo y Markazuza, respectiva-

mente). Para probar la eficiencia, y velocidad de la propuesta presentada en este documento, 210 diferentes configuraciones por serie de tiempo fueron utilizadas para cada serie de tiempo descrita anteriormente.

Las configuraciones utilizadas fueron las siguientes, cabe destacar que fueron tomadas de manera arbitraria, en base a experiencia propia de los autores:

- Para el tamaño del vector de atributos w (ventana de predicción) se utilizaron los valores de : [2, 3, 5, 10, 15, 20, 25]
- Para la división entre el conjunto de entrenamiento y de validación, se utilizó : [70-30 %, 80-20 %, 90-10 %, 95-5 %, 99-1 %]
- Para el número vecinos k se utilizó : [1, 3, 5, 10, 15, 20]

Los resultados de los experimentos realizados se observa en la Tabla 1. La cual muestra el promedio de la aptitud (MSE) de cada serie de tiempo descrita anteriormente. Las diferencias menores entre el *knn* sobre *CPU* vs *GPU* observados en los resultados de la Tabla 1 se deben la diferencia de arquitectura hardware que poseen el *CPU* y el *GPU*. *Nvidia* fabricante de la tarjeta define una velocidad para cálculos de 32 bits y otra velocidad para el cálculo de valores de 64 bits, siendo este último significativamente más lento que el primero. Y aunque el *GPU* sea capaz de procesar información de 64 bits, se prefirió utilizar direcciones de memoria de 32 bits para no mermar el rendimiento de la tarjeta.

Tabla 1. Comparación entre *CPU* y *GPU*. Diferencias en la precisión entre la arquitectura secuencial y paralela.

MSE			
Serie	**Tamaño**	**CPU**	**GPU**
El Fresno	8 mil datos	**0.00636108**	0.007125687
Malpais	22 mil datos	**0.003438114**	0.003823412
Melchor Ocampo	32 mil datos	**0.002532585**	0.002834963
Markazuza	43 mil datos	**0.00024559**	0.000282856

Por otro lado en cuestión de tiempo de ejecución, la Tabla 2 muestra los tiempos promedio de ejecución de los 210 diferentes experimentos de cada serie de tiempo. De la Tabla 2 se observa que a medida que el número de datos aumenta, el tiempo de ejecución tanto en la implementación secuencial como en la paralela aumenta, aunque el incremento de la implementación secuencial (*CPU*) es considerablemente mayor. Además, se observa que los experimentos ejecutados sobre el *GPU* realiza de manera casi instantánea; para el experimento de Markazuza sobre *CPU* el tiempo de ejecución es de 45 minutos aproximandamente.

Tabla 2. Comparación entre *CPU* y *GPU*. Tiempo en segundos de la arquitectura secuencial y paralela.

Tiempos			
Serie	**Tamaño**	**CPU**	**GPU**
El Fresno	8 mil datos	130.614 s	**0.150 s**
Malpais	22 mil datos	601.143 s	**0.478 s**
Melchor Ocampo	32 mil datos	1510.716 s	**1.550 s**
Markazuza	43 mil datos	2732.124 s	**1.922 s**

Al comparar ambas implementaciones de kNN (*CPU* vs *GPU*), se observa que se reduce considerablemente el tiempo de ejecución, pero se tiene perdida en la precisión del modelo esto debido a las características del hardware.

Finalmente cabe señalar que los experimentos secuenciales fueron ejecutados sobre un servidor HP con un procesador Intel Xeon E5-2603V4 a 1.7GHz con 8 GB DDR4. Mientras que para la parte no secuencial se utilizó sobre el mismo servidor una tarjeta gráfica EVGA GTX 1080 TI.

7. Conclusiones

La utilización de valores de 32 bits con respecto a los de 64 bits no altera significativamente los resultados obtenidos, como se aprecia en la Tabla 1. Por otro lado, el cambio más significativo se dio en los tiempos de ejecución del algoritmo. En la Tabla 2 se aprecia claramente que en todos los casos la implementación sobre el *GPU* venció con un gran margen a la implementación secuencial.

Otro resultado a tener en cuenta es la reducción del error con el aumento de datos registrado con las distintas series de tiempo. Esto se demuestra que a mayor número de datos de entrenamiento, la precisión de los resultados mejoran con creces. Pero cabe mencionar que el tiempo de ejecución de ambas arquitecturas aumenta.

Las configuraciones del modelo influyen en la precisión de éste. Pero sin importar el modelo, con un mayor número de datos es posible obtener mejores resultados. La mejor opción entonces, es la arquitectura paralela debido a que con los tiempos de ejecución reducidos es posible utilizar una mayor cantidad de datos.

Los resultados demuestran que el algoritmo es capaz de obtener excelentes resultados en la predicción de variables del viento. No por ello limitado al pronóstico del viento, claro está. Sin embargo, se tiene que mencionar que la sobrecarga en tiempo de desarrollo del algoritmo sobre la arquitectura *CUDA*,

es solo un pequeño precio a pagar con respecto a los grandes beneficios que aportan en rendimiento los *GPUs*.

Referencias

1. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on. pp. 1–6. IEEE (2008)
2. Kuang, Q., Zhao, L.: A practical gpu based knn algorithm. In: International symposium on computer science and computational technology (ISCSCT). pp. 151–155. Citeseer (2009)
3. Liang, S., Wang, C., Liu, Y., Jian, L.: Cuknn: A parallel implementation of k-nearest neighbor on cuda-enabled gpu. In: Information, Computing and Telecommunication, 2009. YC-ICT'09. IEEE Youth Conference on. pp. 415–418. IEEE (2009)
4. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: A unified graphics and computing architecture. IEEE micro 28(2) (2008)
5. Martınez, C.: Partial quicksort. In: Proc. 6th ACMSIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics. pp. 224–228 (2004)
6. Montgomery, D.C., Jennings, C.L., Kulahci, M.: Introduction to time series analysis and forecasting. John Wiley & Sons, New Jersey, USA, second edn. (2015)
7. Nvidia: Cuda 9.0 compute unified device architecture programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, last accessed: 2018-02-15
8. Tsay, R.S.: Time series and forecasting: Brief history and future research. Journal of the American Statistical Association 95(450), 638–643 (2000)
9. Wu, X., Kumar, V., Quinlan, J.R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Philip, S.Y., et al.: Top 10 algorithms in data mining. Knowledge and information systems 14(1), 1–37 (2008)
10. Wu, Y.K., Hong, J.S.: A literature review of wind forecasting technology in the world. In: Power Tech, 2007 IEEE Lausanne. pp. 504–509. IEEE (2007)
11. Yule, G.U., et al.: Vii. on a method of investigating periodicities disturbed series, with special reference to wolfer's sunspot numbers. Phil. Trans. R. Soc. Lond. A 226(636-646), 267–298 (1927)
12. Zhao, X., Wang, S., Li, T.: Review of evaluation criteria and main methods of wind power forecasting. Energy Procedia 12, 761–769 (2011)