UNIVERSIDAD CATÓLICA DEL MAULE

Facultad de Ciencias de la Ingeniería

Escuela de Ingeniería Civil Informática

PROFESOR GUÍA

Dr. Ricardo Barrientos

Aplicación gráfica de procesamiento de consultas kNN sobre diferentes plataformas paralelas

Cristofher Andrés Rojas Rojas

Tesis para optar al Título Profesional de Ingeniero Civil Informático

Talca, Junio, 2017

UNIVERSIDAD CATÓLICA DEL MAULE
FACULTAD DE CIENCIAS DE LA INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL INFORMÁTICA

TESIS PARA OPTAR AL
TÍTULO PROFESIONAL DE INGENIERO CIVIL INFORMÁTICO

Interfaz gráfica para usuarios para resolver consultas Knn en diferentes plataformas paralelas

Cristofher Andrés Rojas Rojas

COMISIÓN EXAMINADORA	**FIRMA**
Profesor Guía Dr. Ricardo Barrientos Universidad Católica del Maule	_____
Profesor Evaluador Universidad Católica del Maule	_____
Profesor Evaluador Universidad Católica del Maule	_____
NOTA FINAL EXAMEN	_____

Talca, Junio, 2017

Agradecimientos

Cristofher Rojas

Primeramente agradecer a Dios por permitirme tener una grata experiencia en mi universidad, gracias a mi universidad por darme las herramientas para poder formarme como profesional en un área que me apasiona, agradecer a cada profesor por su proceso integral en mi formación como profesional, de esta manera agradezco los conocimientos obtenidos y que me ayudaron directa o indirectamente en la realización de esta tesis.

De manera especial, dedico palabras de agradecimiento para mi asesor de Tesis, Dr. Ricardo Barrientos, tanto por su esfuerzo y dedicación. Destacando además el apoyo que me brindo con sus conocimientos, sus orientaciones, su persistencia y su motivación en el proceso de desarrollo de la tesis. A su manera fue capaz de ganar mi admiración, respeto y lealtad, sintiéndome en deuda con el por su apoyo en esta tesis.

Para finalizar mi agradecimiento no puedo no incluir a mi familia ya que son un pilar fundamental en mi vida y en todo el proceso que estuve en la universidad, ya que gracias a ellos he logrado concluir mi carrera, en especial a mis padres Héctor Y Eliana por su apoyo y consejos para ayudarme a ser una mejor persona, a mi hermana Constanza por llenar con su alegría mi vida luego de su llegada, a Katheryn por sus palabras de apoyo y consejos. Además de todas aquellas personas que estimo y forman parte directa o indirectamente de mi vida, apoyándome en este proceso.

Talca , 2017

Resumen

Hoy en día, es indudable la utilización de un computador para realizar análisis más profundos, dado que de otro modo el cálculo se vuelve complejo, por esto es fundamental contar con algoritmos y/o software que nos permitan mejorar las respuestas de tiempo al realizar dichos análisis.
En el presente trabajo se desarrolla un software, el cual tiene el objetivo de ser utilizado por usuarios que no cuenten con conocimientos en el área de programación, por tanto no puedan programar el algoritmo que resuelve o procesa consultas kNN ni secuencial ni paralelamente. Este software está desarrollado de manera que pueda ser intuitivo en su utilización, donde solo se deben cargar los datos, tanto la base de datos como las consultas. Éstas deben estar en un archivo de texto plano y para esto posee botones de examinar y localizar el archivo en cuestión. No obstante, puede ser que el usuario no sepa bien el nombre de archivo, o no seleccione el archivo correcto (tanto la base de datos como la consulta), de manera que el software cuente con una vista previa de dichos archivos, de tal manera que el usuario pueda verificar los datos antes de realizar algún procedimiento.

Dado que el software cuenta con el algoritmo secuencial kNN, algoritmo paralelo multi-hilo kNN (utilizando la librería OpenMP), algoritmo paralelo Intel Xeon Phi (utilizando la librería OpenMP), algoritmo paralelo GPU (utilizando la extensión de C denominada CUDA).
Así como el algoritmo secuencial utiliza como parámetros de entrada la base de datos y las consultas, también el usuario debe indicar la dimensión de los objetos de la base de datos y la cantidad de elementos más cercanos deseados (K). Los algoritmos paralelos (multi-hilo y Intel Xeon Phi) utilizan también una bases de datos y las consultas como parámetros de entrada y además el usuario debe indicar la dimensión de los objetos de la base de datos y la cantidad de elementos más cercanos deseados (K), pero estos algoritmo cuentan con una diferencia debido a que se debe indicar el número de hilos

con los que el usuario desee ejecutar el proceso. Por defecto, el software reconoce la cantidad de núcleos que tiene el computador, y los establece como la cantidad de hilos a utilizar. Este dato es una sugerencia de manera que el usuario puede indicar el número que estime conveniente. A su vez para la GPU se estableció un valor por defecto, que se obtiene de las características del hardware del modelo de GPU instalado.

Cabe mencionar que los resultados obtenidos mediante nuestro software, pueden ser exportados a los siguientes formatos XLS (archivos de Excel), DOC (archivos word), TXT (archivos de texto plano), PDF (formato de documento portátil), de forma que el usuario pueda elegir el formato que más le acomode. Para aquellos que si poseen conocimientos de programación, pueden añadir sus métodos utilizando las entradas establecidas para cada caso. Dado que el software cuenta con un menú para añadir nuevos métodos del algoritmo, el usuario en cuestión debe seleccionar a que tipo de este (secuencial, multi-hilo, Xeon Phi, GPU). Con esto el usuario debe seleccionar el archivo fuente (respetando ciertas características establecidas previamente). De manera similar al caso de las bases de datos y consultas, el software cuenta con una vista previa para el chequeo correcto de el archivo cargado.

Índice general

Índice de figuras

Índice de tablas

Índice de Algoritmos

Capítulo 1

Introducción

En esta tesis se propone un software pensado en usuarios que poseen poco o nulo conocimientos de programación, de modo que puedan utilizar este software para resolver consultas kNN. Para esto, se han seleccionado de distintas versiones de algoritmos que resuelven consultas kNN sobre distintas plataformas paralelas propuestos en la literatura. En este software, se presentan cuatro algoritmos diferentes: algoritmo secuencial, algoritmo multi-hilo, algoritmo sobre una plataforma Intel Xeon Phi, algoritmo sobre una plataforma GPU. Se utilizo JAVA 8 [Oracle, 2017] para programar la interfaz gráfica del software debido a que este lenguaje es soportado en los distintos sistemas operativos: Windows en sus versiones 10, 8.x (Escritorio), 7 SP1, Vista SP2, Windows Server 2008 R2 SP1 (64 bits), Windows Server 2012 y 2012 R2 (64 bits) y los recursos necesarios son 128 MB de RAM, un espacio minino de 124 MB, procesador pentium 2 a 266 MHz. También en MAC OSx equipos Mac con un procesador Intel que ejecuta Mac OS X 10.8.3+, 10.9+, donde la instalación se debe realizar en modo administrador. En linux en cambio está disponible para Oracle Linux desde la versión 5.5+ o en versiones superiores, en Red Hat Enterprise Linux desde la versión 5.5+ o en versiones superiores, Suse Linux Enterprise Server desde 10 SP2+ o versiones superiores y en Ubuntu Linux en desde versiones 12.04 LTS o en versiones superiores.

Para la programación de los algoritmos se empleó el lenguaje de programación C [Kernighan and Ritchie, 2006], dado que este lenguaje puede ser utilizado en diversos sistemas operativos, además de poder utilizar librerías que permiten el uso del paralelismo en distintos tipos de plataformas paralelas, como son multi-núcleo, Intel Xeon Phi y GPU.

Las plataformas paralelas que se emplean en esta tesis son las mencionadas anteriormente, tanto para multi-núcleo e Intel Xeon Phi se utiliza la librería OpenMP [Chapman et al., 2007][Nichols et al., 1996], Esta librería está pensada para los lenguajes C, C++ y Fortran, disponible además para diversos tipos de arquitecturas, incluidas algunas plataformas como Windows y Unix. A su vez para GPU se cuenta con librerías propuestas por NVIDIA y CUDA [cud, 2017] la cual permite trabajar las concurrencias en GPU, poder utilizar los hilos y multiprocesadores que propone NVIDIA.

kNN *(K nearest neighbors)* es un tipo de consulta de minería de datos más utilizado, este es un clasificador basado en instancias (aprendizaje por analogía). Se buscan los casos más parecidos y la clasificación se realiza en función de la clase analizada, siendo conocido por ser un paradigma perezoso debido a ser un análisis exhaustivo con la consulta y todos lo elementos del conjunto.

Si bien el algoritmo kNN implica alto cómputo, al ser exhaustivo (evaluar cada elemento del conjunto) es exacto con una complejidad del tipo O[k*N] donde N es el número de tuplas y k el número de elementos más cercanos, este tipo de consultas suele ser utilizada en muchos campos, como lo son minería de datos usado para estimar una función de densidad $\mathbf{F(x}/C_j\mathbf{)}$ donde predice el valor x para la clase C_j. Este algoritmo además es muy utilizado en el reconocimiento de patrones, por ejemplo imaginemos que dado el caso de una base de datos con rostros, se desea identificar los k rostros más similares a un rostro consulta. En este caso con las imágenes de los rostros se crean descriptores de la imágenes tanto para la base de datos como para la consulta en cuestión; este descriptor está compuesto por un vector lineal, el cual con

el cálculo de la distancia entre dos vectores (descriptor) se obtiene la similitud entre ambos. Se debe considerar ciertos aspectos para establecer una variable K, dado que éste depende de los datos, de modo que si se selecciona un valor muy pequeño estos pueden ser ruidos (ser elementos de otra clase) y si se selecciona un K muy grande puede pasar exactamente lo mismo (abarcar elementos de otra clase). De tal manera que la elección del K debe ser elegida de acuerdo a la cantidad de datos.

Este algoritmo puede llevarse a un sinfín de otros usos, asociados a la clasificación y reconocimiento de patrones. En este punto es importante añadir que existen diferentes formas de emplear y aplicar el algoritmo de modo que en algunas situaciones es utilizado como un clasificador previo al análisis más potentes.

1.1. Objetivos

El objetivo general de esta tesis se centra en la selección de algoritmos que resuelven la consulta kNN (k-nearest neighbor) encontrados en la literatura, estos proponen una solución más óptima en tiempos de ejecución, la selección de los algoritmos fue dada por aquellos que han sido propuestos en las lineas de programación paralela. Como en la literatura se aprecian diversas soluciones, las más óptimas fue donde se emplean heap para la resolución de este algoritmo.

Para poder realizar la unión de los diferentes algoritmos paralelos, se propone un software el cual permite la agrupación de algoritmos de diferentes plataformas paralelas, además de añadir el algoritmo secuencial base como un agregado. En base a lo anteriormente mencionado se pretende que el software sea diseñado en consideración a usuarios los cuales no estén familiarizados con la programación de estos. Por tanto el usuario solo se preocupa de la utilización del software, y no vea complicaciones a sus conocimientos en áreas de programación.

A continuación se mencionan los objetivos específicos que se han propuesto:

- Realizar la interfaz gráfica del software

Esta se realiza en JAVA debido a su capacidad de ser interpretada tanto en Windows, Linux y OSx

- Añadir los algoritmos kNN al software

  kNN multi-núcleo

  kNN Intel Xeon Phi

  kNN GPU

- Añadir menú para agregar nuevos métodos

- Permitir al usuario exportación de los resultados en los siguientes formatos

  Excel

  Word

  Archivo de texto plano (TXT)

  Documento portátil PDF

Capítulo 2

Marco Teórico

2.1. Sistemas Multi-core y OpenMP

En este capitulo se describen los conocimientos básicos sobre programación multi-núcleo, programación en coprocesadores tales como programación en Intel Xeon Phi, programación en GPU y trabajos relacionados que dan base a esta tesis.
Actualmente las arquitecturas de los dispositivos personales están diseñadas con multiprocesadores o coprocesadores, desde un equipo de bolsillo como un celular (smartphone) hasta un computador personal (notebook), esto quiere decir que los dispositivos cuentan con varios núcleos en una sola CPU (unidad central de procesamiento) en el caso de multiprocesadores y con más de una CPU en el caso de un coprocesador. Imagen de referencia 2.1

2.1.1. Threads

Existen varias bibliotecas que permiten la programación en multi-hilos, con esto somos capaces de ejecutar hilos en paralelo, ejecutándose en un núcleo diferente.
Existen diferentes modelos y estándares que permiten la programación de múltiples hilos, tales como Pthreads [Nichols et al., 1996], TBB [Reinders, 2007] u OpenMP [Chapman et al., 2007], en particular en esta tesis se selecciono el uso de la biblioteca

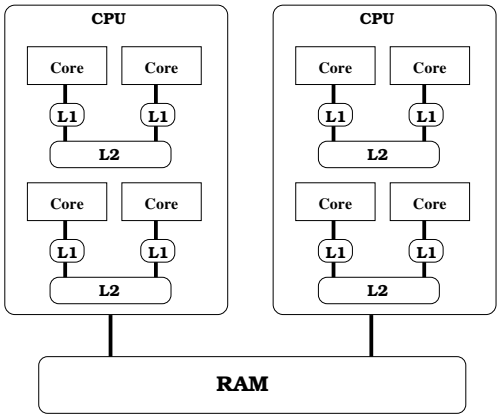


Figura 2.1: Plataforma multi-core.

OpenMP que fue desarrollada en 1997. También es importante mencionar el hecho de que actualmente los compiladores en esta linea son bastante robustos y el equipo que gestiona OpenMP está conformado por diferentes compañías (AMD, Intel, Sun Mycrosystem y otros), al no ser gestionada por una sola nos proporciona confianza.

2.2. Coprocesadores

Un coprocesador es utilizado para desligar funciones de la CPU, en esta línea existen desde coprocesadores matemáticos hasta coprocesadores gráficos, estos poseen características y especificaciones a ciertos tipos de operaciones, por ejemplo un coprocesador matemático solo está desarrollado para revolver problemas aritméticos o afines al área, esto quiere decir que pueden realizar operaciones matemáticas a una velocidad mayor que el procesador principal (CPU), a su vez un coprocesador gráfico está diseñado en otra área. Estos proporcionan al hardware una aceleración en la operación sólo cuando se ejecuta una tarea o software que se haya diseñado para aprovechar el coprocesador, las instrucciones para un coprocesador son diferentes que las instrucciones de una CPU de modo que existe un programa que detecta la existencia de un coprocesador y entonces se ejecutan las instrucciones escritas explícitamente para

aquel coprocesador, por tanto existen diferentes líneas de coprocesadores en el mercado pero en esta tesis sólo consideramos Intel Xeon Phi y GPU NVIDIA.

2.2.1. Intel Xeon Phi

El coprocesador Intel Xeon Phi [xeo, 2017] [Wang, 2014] [LA, 2017] consta de hasta 72 núcleos conectados en un anillo bidireccional de alto rendimiento en el chip. El coprocesador ejecuta un sistema operativo Linux y es compatible con todas las principales herramientas de desarrollo de Intel como C / C++, Fortran, MPI y OpenMP. El coprocesador se conecta a un procesador Intel Xeon (el host) a través del bus PCI Express (PCIe). Las propiedades más importantes de la arquitectura MIC se muestran en figura 2.2.1.

Core: El núcleo de la intel Xeon phi (scalar unit en la figura 2.2.1) es una arquitectura de orden (basada en la familia de procesadores Intel Pentium). Implementa instrucciones de recuperación y decodificación para 4 hilos (hardware) por núcleo. Las instrucciones vectoriales que posee el coprocesador Intel Xeon Phi, utiliza una unidad de punto flotante (VPU) dedicada, con un ancho de 512-bit, la que está disponible en cada núcleo. Un núcleo puede ejecutar dos instrucciones por ciclo de reloj, una en U-pipe y otra en V-pipe (no todos los tipos de instrucciones pueden ser ejecutadas por el V-pipe). Cada núcleo compone una interconexión en anillo mediante la CRI (Core Ring Interface).

Vector Processing Unit (VPU): La VPU incluye la EMU (Extended Math Unit), y es capaz de realizar 16 operaciones en punto flotante con precisión simple por ciclo, 16 operaciones de enteros de 32-bit por ciclo u 8 operaciones en punto flotante con precisión doble por ciclo.

L1 Cache: Tiene una caché L1 de 32KB para instrucciones y de 32KB para datos,

asociativa de 8-vías, con tamaño de línea de caché de 64 bytes. Tiene una latencia de cargado de 1 ciclo, que significa que un valor entero cargado desde L1 puede ser usado en el siguiente ciclo por una instrucción entera (instrucciones vectoriales tienen diferente latencia que las enteras).

L2 Cache: Cada núcleo posee 512KB de caché L2. Si ningún núcleo comparte algún dato o código, entonces el tamaño total de la caché L2 es de 31 MB. Por otro lado, si todos los núcleos comparten exactamente el mismo código y datos en perfecta sincronía, el tamaño total de la caché L2 sería solo de 512KB. Su latencia es de 11 ciclos de reloj.

Ring: Incluye interfaces y componentes, ring stops, ring turns, direccionamiento y control de flujo. Un coprocesador Xeon Phi tiene 2 de estos anillos, uno viajando en cada dirección (Figure 2.2.1).

La Figura 2.2.1 ilustra una visión general de la arquitectura, donde se muestra como los núcleos están conectados con cachés coherentes mediante un anillo bidireccional de 115GB/sec. La figura también muestra una memoria GDDR5 con 16 canales de memoria que alcanza una velocidad de hasta 5.5GB/sec.

2.2.2. NVIDIA GPU

Todas las implementaciones sobre GPU se realizaron usando el modelo de programación CUDA [cud, 2017] de NVIDIA. Este modelo hace una distinción entre el código ejecutado en la CPU (host) con su propia DRAM (host memory) y el ejecutado en GPU (device) sobre su DRAM (device memory). La GPU se representa como un coprocesador capaz de ejecutar funciones denominadas kernels, y provee extensiones para el lenguaje C que permiten alojar datos en la memoria de la GPU y transferir datos entre GPU y CPU. Por lo tanto, la GPU sólo puede ejecutar las funciones

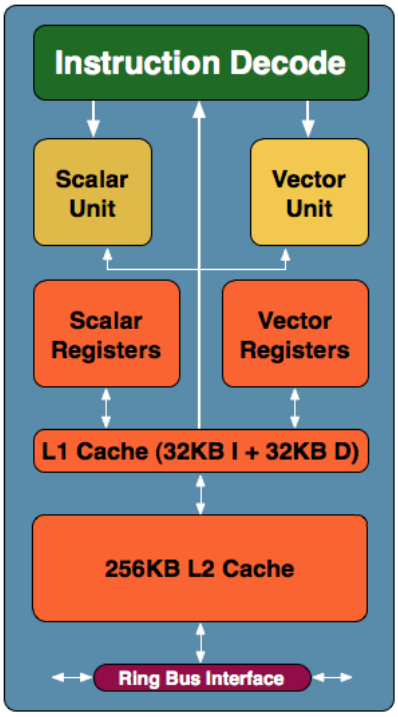


Figura 2.2: Diagrama de una arquitectura MIC de un núcleo de un coprocesador Xeon Phi

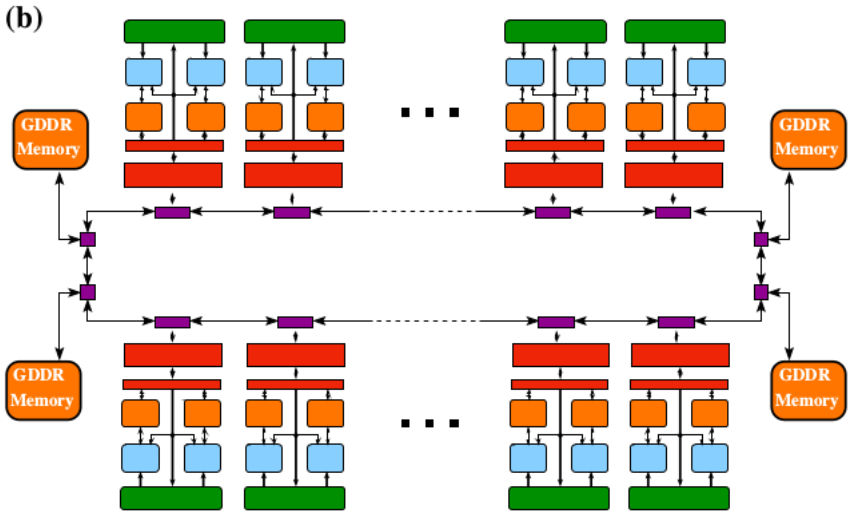


Figura 2.3: Diagrama de una arquitectura MIC interconectada

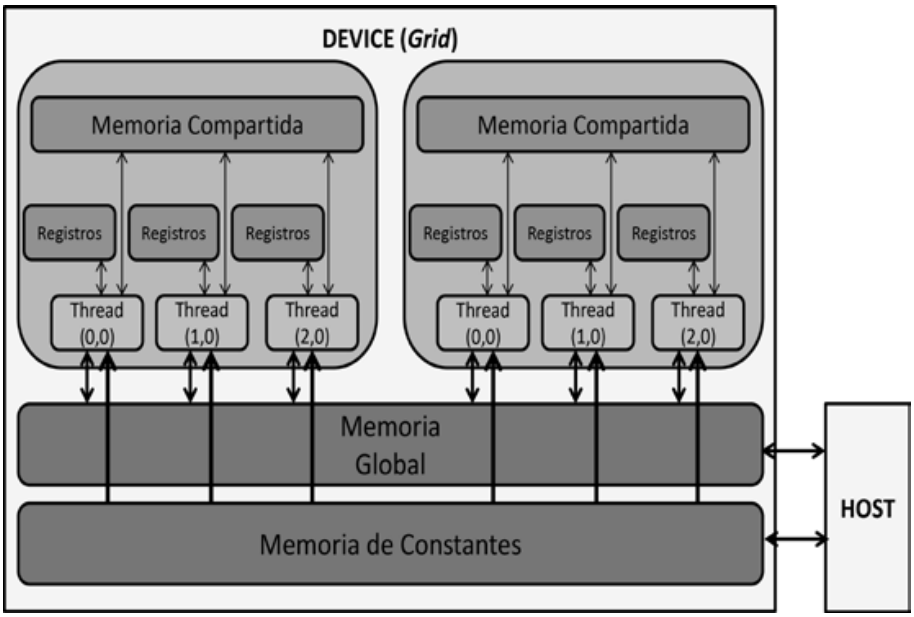


Figura 2.4: Composición del *device*

declaradas como kernels dentro del programa, y sólo puede usar variables alojadas en device memory, lo cual significa que todos los datos necesarios deberán ser movidos a esta memoria de forma previa a la ejecución la figura 2.6 muestra un esquema de la diferencia entre una CPU y una GPU. Esta última está especializada para cómputo intensivo, y por lo tanto más transistores están dedicados para el procesamiento de datos en vez de caching de datos y control de flujo.

Esto implica que la GPU es capaz de realizar un mayor número de operaciones aritméticas en punto flotante que una CPU. El correcto aprovechamiento del sistema de memoria en la GPU es primordial. La figura 2.4 muestra un esquema del hardware de una GPU, la que consta de las siguientes características:

- Una GPU está constituida por una serie de *multiprocesadores*, donde cada uno de éstos cuenta con varios núcleos.
- Cada núcleo cuenta con sus propios registros y todos los núcleos del mismo multiprocesador comparten una *shared memory*.

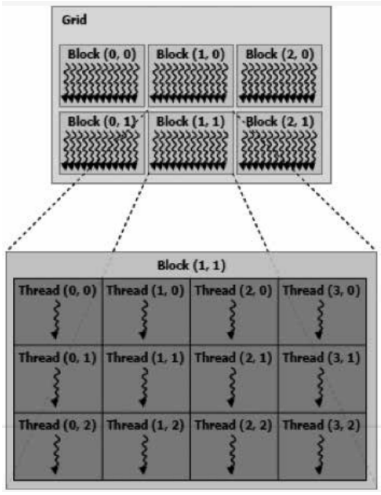


Figura 2.5: Esquema de división de threads.

- **Shared memory** es una memoria que se encuentra próxima a los núcleos del multiprocesador, y por lo tanto es de muy baja latencia.
- Cada multiprocesador cuenta con un par de memorias caché compartidas por todos los núcleos, *constant cache* y *texture cache*.
- **Constant cache** es una caché de datos de la constant memory, que es una memoria de sólo lectura de reducido tamaño alojada en *device memory*.
- **Texture cache** es una caché de la texture memory, que es una memoria optimizada para localidad espacial 2D, y está alojada en *device memory*.
- **Device memory** es una memoria compartida por todos los multiprocesadores, y es de gran tamaño.

Organización y ejecución de threads

Los threads son organizados en *CUDA Blocks*, y cada uno de estos se ejecuta completamente en un único multiprocesador. Esto permite que todos los threads de un mismo CUDA Block compartan la misma *shared memory*. Un *kernel* puede ser

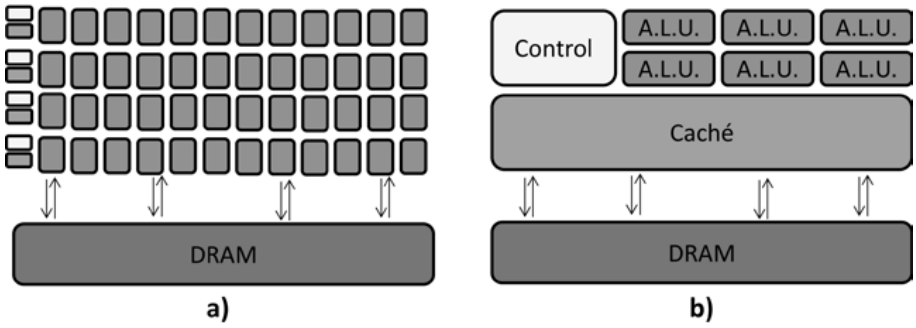


Figura 2.6: Composición de la CPU a) vs. GPU b).

ejecutado por un número limitado de CUDA Blocks, y cada CUDA Block por un número acotado de threads. Sólo los threads que pertenecen al mismo CUDA Block pueden ser sincronizados. Es decir, hilos que pertenezcan a CUDA Blocks distintos sólo pueden ser sincronizados realizando un nuevo lanzamiento de kernel.

Para soportar la gran cantidad de threads en ejecución se emplea una arquitectura SIMT (Single-Instruction, Multiple-Thread). La unidad de ejecución no es un thread, sino un *warp*, que es un conjunto de threads, y un *half-warp* corresponde a la primera o segunda mitad del conjunto de threads de un *warp*. La forma en que un CUDA Block es dividido en *warps* es siempre la misma, el primer conjunto de threads representan el primer *warp*, el segundo conjunto consecutivo de threads el segundo *warp* y así sucesivamente.

En la figura 2.5 se aprecia como una jerarquía, los threads se agrupan en estructuras de 1, 2 ó 3 dimensiones denominados bloques, los bloques a su vez, se agrupan en un grid de bloques que también puede ser de 1, 2 ó 3 dimensiones.
En la figura 2.6 se aprecia la comparación de una CPU (a) y una GPU (b), estas poseen diferencias técnicas, como se aprecia en la siguiente tabla 2.1.

Tabla 2.1: Tabla comparativa CPU vs GPU

CPU	**GPU**
Procesador escalar	Procesador vectorial
Acceso aleatorio a memoria	Acceso secuencial a memoria
Modelo de programación muy conocido	Modelo de programación poco conocido
20 GB/s Ancho de banda	100 -150 GB/s Ancho de banda
Consumo de 1 Gflop/watt	Consumo de 10 Gflop/watt

Reducción de lecturas a memoria

Los accesos a memoria de un *half-warp* a device memory son fusionados en una sola transacción de memoria si las direcciones accedidas caben en el mismo segmento de memoria. Por lo tanto, si los threads de un *half-warp* acceden a n diferentes segmentos, entonces se ejecutarán n transacciones de memoria. Si es posible reducir el tamaño de la transacción, entonces se lee sólo la mitad de ésta.

Funciones atómicas

Existe un grupo de funciones atómicas que realizan operaciones de tipo *read-modify-write* sobre palabras de residentes en *global memory o shared memory*. Estas funciones garantizan que serán realizadas completamente sin interferencia de los demás threads. Cuando un thread ejecuta una función atómica, ésta bloquea el acceso a la dirección de memoria donde reside la palabra involucrada hasta que se complete la operación. Las funciones atómicas se divide en funciones aritméticas y funciones a nivel de bits, a continuación se muestran algunas de las funciones aritméticas relevantes para el presente trabajo:

- atomicAdd int atomicAdd(int *address, int val)

  Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor *(old+val)*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- atomicSub int atomicSub(int *address, int val)

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor *(old-val)*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- atomicMin int atomicMin(int *address, int val)

  Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor mínimo entre *(old-val)*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- atomicMax int atomicMax(int *address, int val)

  Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor máximo entre *(old-val)*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

Ejemplos

La figura 2.7 muestra un código que suma 2 vectores utilizando la GPU. Para esto primero se asigna memoria a los arreglos en device memory, luego se copian los valores necesarios a estas variables, y posteriormente se invoca al kernel con 1 CUDA Block (dado por la variable N_BLOQUES) y 200 threads por cada CUDA Block (dado por la variable N).

Cada thread obtiene su identificador (variable tid) usando variables predefinidas por la GPU, ID_{Thread} (que retorna el ID del thread en el CUDA Block), T_{Block} (que retorna la cantidad de threads de un CUDA Block) y ID_{Block} (que retorna el identificador del CUDA Block). Cada thread realiza la suma del elemento tid-ésimo de ambos arreglos (dev_A y dev_B). Y ya una vez terminado el kernel se copia el arreglo resultado a la memoria de CPU para poder imprimir los resultados, pues no se puede invocar una función para imprimir dentro del kernel en la GPU utilizada.

Los threads de un warp siempre están sincronizados, es decir, ningún thread se puede adelantar a otro que pertenezca al mismo warp en la ejecución de instrucciones.

```
#include <stdio.h>
#include <cuda.h>
#define N 200
#define N_BLOQUES 1

/* Se define un kernel llamado 'function' */
__global__ void function(float *A, float *B, float *result)
{
    /* Se obtiene el identificador del thread */
    int tid = ID_Thread + ( T_Block *ID_Block );
    /* Cada thread realiza la suma de un elemento distinto */
    result [tid] = A[tid] + B[tid];
    return;
}

main()
{
   float host_A[N], host_B[N], host_result[N];
   float *dev_A, *dev_B, *dev_result;

  /* Asignamos memoria en device memory */
   cudaMalloc((void **)&dev_A, sizeof(float)*N);
   cudaMalloc((void **)&dev_B, sizeof(float)*N);
   cudaMalloc((void **)&dev_result, sizeof(float)*N);

   inicializar_valores(host_A, host_B, host_result);
   /* Se copian los valores a las variables alojadas en device memory */
   cudaMemcpy(dev_A, host_A, sizeof(float)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(dev_B, host_B, sizeof(float)*N, cudaMemcpyHostToDevice);

   /*Invocamos al kernel con 'N_BLOQUES' bloques y 'N' threads por bloque
     con las variables alojadas en device memory */
   function<<<N_BLOQUES, N>>>(dev_A, dev_B, dev_result);

   /* Se copian los resultados a memoria de la CPU */
   cudaMemcpy(host_result, dev_result, sizeof(float)*N, cudaMemcpyDeviceToHost);

   imprimir_resultados(host_result);
    cudaThreadExit();
    return 0;
}
```

Figura 2.7: Ejemplo de suma de vectores con CUDA.

```
int tid = ID_Thread
...
int ref1 = myArray[tid] * 1;
__syncthreads();
myArray[tid + 1] = 2;
__syncthreads();
int ref2 = myArray[tid] * 1;
result[tid] = ref1 * ref2;
...
```

(a) Caption Subfigura 1

```
int tid = ID_Thread
...
if (tid < warpSize) {
int ref1 = myArray[tid] * 1;
myArray[tid + 1] = 2;
int ref2 = myArray[tid] * 1;
result[tid] = ref1 * ref2;
}
...
```

(b) Caption Subfigura 2

Figura 2.8: Ejemplos para ilustrar la sincronizaci'on de los threads de un *warp*.

Un ejemplo de esto lo muestra la figura 2.8, donde en el código 2.8(a) se deben realizar instrucciones de sincronización para garantizar el correcto valor del arreglo result, mientras que el código 2.8(b) es realizado solamente por los threads de un warp, y debido a esto no es necesario utilizar sincronización.

2.3. Plataforma JAVA

Una plataforma es el entorno de hardware o software en el cual se ejecuta un programa. A diferencia de las plataformas tradicionales como Linux, Windows y Solaris, la plataforma Java está basada sólo en software que se ejecuta sobre otras plataformas basadas en hardware. Por esto la plataforma Java y su software pueden ser utilizadas sobre variados sistemas operativos y hardware.
La plataforma Java está constituida de tres componentes: el lenguaje, la máquina virtual y las bibliotecas. [Nuñez, 2003, pág. 5]

2.3.1. Lenguaje JAVA

El lenguaje de programación Java es de propósito general, de alto nivel que utiliza el paradigma de orientación a objetos. Su sintaxis y tipos están basados principalmente en C++, sin embargo, las diferencias principales con éste es la administración de memoria,

siendo ejecutada por la máquina virtual automáticamente y no por el código de cada programa, y el soporte de procesos livianos o threads a nivel del lenguaje que ayuda a controlar la sincronización de procesos paralelos. Estas características dan al lenguaje Java las propiedades de robustez y seguridad, evitando por ejemplo problemas de buffer overflow utilizados en ataques a sistemas. [Nuñez, 2003, pág. 5]

2.3.2. La maquina virtual

Los programas escritos en Java son compilados como archivos ejecutables de una máquina virtual llamada Java Virtual Machine (JVM). Existen implementaciones de esta máquina para múltiples plataformas, permitiendo ejecutar en diferentes arquitecturas el mismo programa ya compilado. La característica de independencia de la plataforma hace posible el libre intercambio de software desarrollado en Java sin necesidad de modificaciones, lo que ha sido llamado "Write once, Run anywhere". [Deepak et al., 2001] Java es un lenguaje compilado e interpretado a la vez. Compilado ya que previo a su ejecución un programa debe ser transformado a un lenguaje intermedio, llamado Java bytecodes. Interpretado porque cada programa luego debe ser procesado y ejecutado por alguna implementación de la JVM específica a la plataforma. [Nuñez, 2003, pág. 6]

2.3.3. Bibliotecas

El conjunto de bibliotecas del lenguaje es conocido como la Java Application Programming Interface (Java API) que es un gran conjunto de componentes que proporcionan diferentes herramientas para el desarrollo de programas Java. La API de Java está agrupada en conjuntos de bibliotecas relacionadas conocidas como paquetes, que contienen grupos de elementos básicos de Java, llamados clases e interfaces. [Nuñez, 2003, pág. 6]

2.4. Metodología

La metodología utilizada es el desarrollo incremental (iterativo y creciente) Defino por José Joskowicz de la siguiente manera:

"*El modelo incremental consiste en un desarrollo inicial de la arquitectura completa del sistema, seguido de sucesivos incrementos funcionales. Cada incremento tiene su propio ciclo de vida y se basa en el anterior, sin cambiar su funcionalidad ni sus interfaces. Una vez entregado un incremento, no se realizan cambios sobre el mismo, sino únicamente corrección de errores. Dado que la arquitectura completa se desarrolla en la etapa inicial, es necesario, al igual que en el modelo en cascada, conocer los requerimientos completos al comienzo del desarrollo. Respecto al modelo en cascada, el incremental tiene la ventaja de entregar una funcionalidad inicial en menor tiempo.*" [Joskowicz, 2008, pág. 6 - 7]

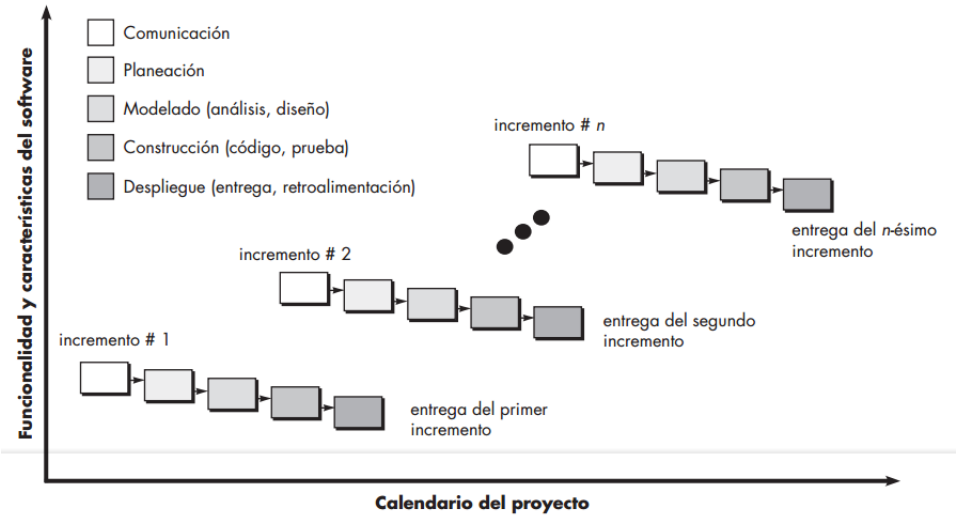


Figura 2.9: Modelo incremental

En relación con la figura 2.9, el modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce "incrementos" de software susceptibles de entregarse [McDermid, 1993] de manera parecida a los incrementos producidos en un flujo de proceso evolutivo.

[Pressman and Troya, 1988, pág. 35]

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande tal vez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante. [Pressman and Troya, 1988, pág. 36]

Se implemento esta metodología debido a la característica propia del software desarrollado, dado que una etapa del software no estaba ligada a otra, de manera que era posible agregar un algoritmo a medida que su desarrollo se finalizaba correctamente y luego se añadían los siguientes al software.

Ventajas del modelo

El modelo iterativo y creciente presenta excelentes ventajas para ser aprovechadas en el desarrollo del software propuesto, tales como:

- No se debe esperar a que el desarrollo del sistema (software) este completo para poder ser utilizado por algún usuarios. Tomando en cuenta que el primer incremento es el más importante y da paso a que los usuarios interesados ya puedan utilizar el sistema (software).
- Los usuarios pueden utilizar incrementos iniciales como el prototipo, esto genera una experiencia real sobre el cumplimiento de los requisitos del sistema (software).

- Como el sistema (software) se va probando gradualmente existe menos probabilidad de errores en la etapa final.
- Se entregan primero los requisitos principales, de modo que al integrar nuevas características estas puede ser agregadas más fácilmente.
- Al ser un proceso iterativo da una mejor retroalimentación del sistema (software) final.

Sobre la base de las ventajas mencionadas anteriormente y las características del software, se optó por emplear esta metodología.

- Su primer incremento correspondió a la creación de la interfaz gráfica, ya que esta es la base de todo el software que se desarrolló y luego correspondió la comunicación entre la interfaz gráfica en JAVA y el algoritmo kNN secuencial en el lenguaje de programación C.

A partir de las ventajas del modelo, el primer incremento cumple con el punto anterior de entregar los requisitos principales y luego integrar nuevas características.

2.5. Selección de algoritmos

La selección de algoritmos se realizó mediante la búsqueda en la literatura, con el objetivo principal de responder de manera rápida a la consulta kNN, los algoritmos que presentaron mejores resultados en la literatura fueron aquellos que utilizan Heap.

Estructura Heap

En programación un heap es una estructura de datos del tipo "árbol" con información de un conjunto ordenado de datos. Se utiliza para implementar colas con prioridad. En este tipo de colas, el elemento a ser eliminado o borrado es el cual tiene la mayor o menor prioridad según sea el caso. A su vez en cualquier momento se pueden

insertar elementos con una prioridad arbitraria.

Existen tres tipos de heap, de acuerdo a sus características de orden

- Max heap: Es un árbol en el cual el valor del nodo padre es mayor a los valores de los nodos de sus hijos (en caso de poseer algún hijo), además de ser un árbol binario completo. Imagen de referencia 2.10.

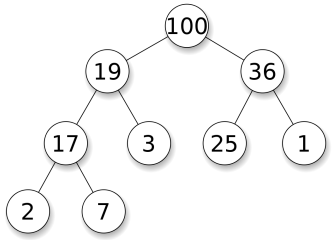


Figura 2.10: Ejemplo de Max Heap

- Min heap: Es un árbol en el cual el valor del nodo padre es menor a los valores de los nodos de sus hijos (en caso de poseer algún hijo), además de ser un árbol binario completo. Imagen de referencia 2.11.

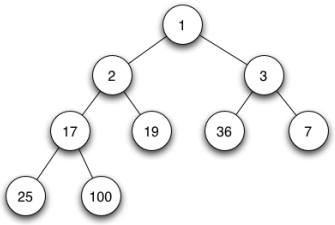


Figura 2.11: Ejemplo de Min Heap

- Min-Max heap: Es un árbol binario, que se construye a partir de un orden min o max de acuerdo a los valores de los nodos, el valor del nodo padre debe ser menor que el valor del nodo hijo (En caso de poseer algún hijo), si posee uno o dos hijos estos tiene un valor superior tanto al nodo padre como también a los sus correspondientes nodos hijos, este es el procedimiento secuencialmente hasta el ultimo nodo del árbol. Imagen de referencia 2.12

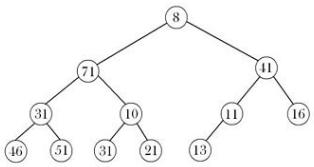


Figura 2.12: Ejemplo de Min Max Heap

Las operaciones básicas del un heap son las siguientes:

1 Creación de un heap vacío.

  Si no existe un heap se crea la estructura.

2 Inserción de un nuevo elemento en la estructura heap.

  Si existe un heap se inserta el elemento aplicando criterios de ordenación, de forma que se respeta el orden de los tipos de heap vistos anteriormente, el árbol se ordena de manera que siempre cumpla con el orden binario.

  Si no existe un heap se crea uno y se añade como el nodo padre.

3 Eliminación del elemento más grande del heap.

  Si existe un heap se puede eliminar el elemento raíz (nodo padre) del heap y el heap debe respetar el orden binario de modo que se reordena el árbol y uno de sus nodos hijos debe ocupar la raíz vacía, luego de haber ocupado ese nodo se debe comparar con los hijos si otro puede ocupar el lugar de nodo padre de acuerdo con las características del tipo de heap.

En esta tesis se han utilizado para la implementación de los algoritmos el heap de tipo Max Heap, esto debido a que el algoritmo busca los K vecinos más cercanos. Se debe realizar el cálculo exhaustivo de la distancia entre dos elementos vectorizados, de una base de datos con un número variado de elementos para comparar, de modo que en un comienzo se debe ir llenado el heap de tamaño K, con los primeros K elementos analizados, el heap realiza su organización de árbol binario y el nodo padre siempre será el elemento de más distancia al elemento comparado (consulta). De esta manera si se descubre una nueva distancia más cercana al valor del nodo padre (raíz del árbol),

se procede a eliminar el valor del nodo actual y se realiza la inserción del nuevo valor y se reordena el árbol nuevamente. Con esto solamente se debe comparar el nodo padre (raíz) con cada una de las distancias que se van obteniendo y no con todos los valores dentro del heap, siendo muy útil esta estructura para realizar las consultas kNN debido a reducir tiempos en la comparación de elementos.

2.6. Selección de lenguajes

Para la selección de un lenguaje de programación óptimo para el algoritmo que procesa una consulta kNN, se debió considerar ciertas condiciones

- Tipo de lenguaje
- Tiempos de ejecución de instrucciones
- Tiempos de ejecución de estructura heap
- Posibilidad de utilizar librerías para trabajos en paralelismo

Tipo de lenguaje

En esta área se establecen muchas diferencias en los lenguajes de programación, como se sabe un computador sólo reconoce el código binario y sus instrucciones están en ordenes de 0's y 1's.
A partir de esto existen lenguajes de programación de acuerdo a la proximidad de la arquitectura de un hardware, existiendo las siguientes categorías.

- Lenguajes de bajo nivel:
  Estos lenguajes son totalmente dependientes de la máquina para aprovechar al máximo las características de ésta, dentro de esta categoría se consideran lenguajes como:

- Lenguaje de maquina
- Lenguaje ensamblador

- Lenguajes de alto nivel:

  Estos lenguajes son los más lejanos a la arquitectura de un computador, de modo que son totalmente independientes de la máquina y muy cercanos al lenguaje humano, dentro de esta categoría se consideran lenguajes como:

  - Python
  - Fortran
  - Etc

- Lenguajes de medio nivel:

  Estos lenguajes se encuentran en un punto medio de entre los dos anteriores, debido a que su sintaxis se asimila al lenguaje humano pero se puede acceder a registros del sistema o direcciones de memoria, estas características son de lenguajes de bajo nivel, dentro de esta categoría se consideran lenguajes como:

  - C

Además un lenguaje se puede agrupar en dos categorías de acuerdo con la naturaleza del lenguaje, estas categorías son:

- Lenguaje interpretado :

  Estos lenguajes se caracterizan porque sus instrucciones o su código fuente, escritos mayoritariamente en lenguajes de altos nivel, estos son traducidos por un interprete (o máquina virtual según sea el caso) a un lenguaje que sea comprensible por la máquina, instrucción por instrucción. Este proceso se repite cada vez que se ejecuta un código de este tipo.

  La principal ventaja es una gran independencia de la plataforma donde se ejecutan los códigos o instrucciones anteriormente mencionados.

  Su principal desventaja es el tiempo que necesite para ser interpretados, generalmente son más lentos en tiempo de ejecución de un lenguaje compilado.

- Lenguaje compilado:

  Estos lenguajes se caracterizan porque su código fuente escrito en alto nivel o medio nivel (C), es traducido por un compilador a un archivo ejecutable para una máquina en específico, en otras palabras cada máquina debe compilar sus códigos fuentes. De manera que se pueden ejecutar las veces que sea necesario sin la necesidad de repetir el proceso de compilación.

Tiempos de ejecución de instrucciones

Según [spe, 2011] realizó la comparación en términos de velocidad de los lenguajes de programación, obteniendo los resultados como se aprecian en la siguiente figura 2.13

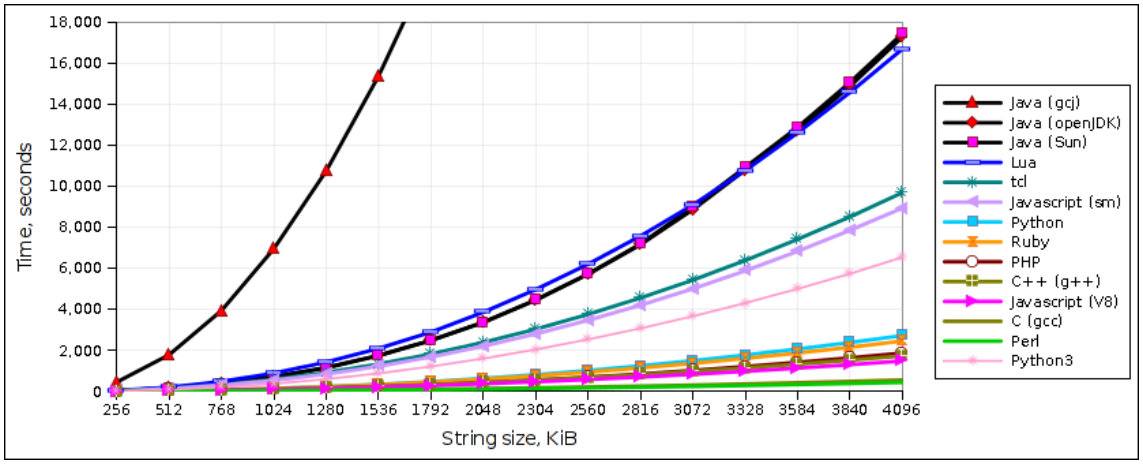
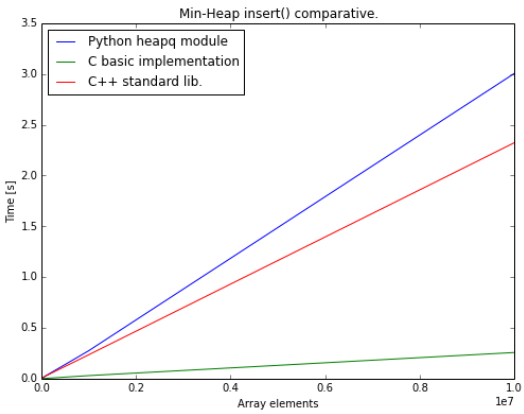


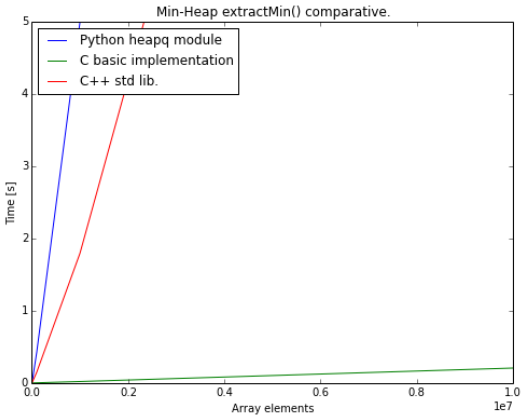
Figura 2.13: Gráfica de tiempo de velocidad

Tiempos de ejecución de estructuras Heap

Una métrica importante para seleccionar el lenguaje más acorde a las necesidades que requiere el algoritmo que procesa una consulta kNN, recordando que el tiempo de ejecución de este es importante, según [Bernabeu, 2013] realizo una comparación de tiempos en diversos lenguajes de programación, como C, C++ y Python, los resultados obtenidos se aprecian en la siguiente figura 2.14



(a) Ejemplo Min Heap insert



(b) Ejemplo Min Heap ExtractMin

Figura 2.14: Gráfica de tiempo de velocidad

Como se aprecia en la figura 2.14 el tiempo de ejecución de un heap es menor en C vs C++ / Python, considerando los tiempos de ejecución el lenguaje de programación C es superior a partir de estas características.

Posibilidad de utilizar librerías para trabajos en paralelismo

Basado en la guía de programación en paralelo [Acosta et al., 2012], se establecen los conceptos básicos de la programación en paralelo basado tres en diferentes modelos como son:

- **MPM** - *Message Passing Model* mediante la librería **MPI**
  Este modelo aprovecha los múltiples *cores* de trabajo que están instalados en los equipos de nueva tecnología. Se asume el *hardware* como reunión de diferentes procesos a través de una red de interconexión donde cada uno tiene acceso directo y exclusivo a la información almacenada en su propia memoria local.

  MPM maneja un tipo de acceso a memoria denominado **NUMA** *(Nom-Uniform Memory Access)*. En el que se describe que los procesos que se comunican por la red escalable de interconexión.

El hardware a su vez, cada memoria está conectada directamente a una CPU, en vez de estar controlado a una memoria (modelo UMA). Las CPUs están conectadas con un I/O hub que permite el ordenamiento de los datos y reduciendo los problemas de tráfico.
Para el uso de un programa en MPM se debe especificar el número de procesos que cuenta la aplicación, cada proceso posee un ID o Rank.

Los mensajes de MPM generalmente contienen:

- La variable en la que reposan los datos que se envían.
- La cantidad de datos que se envían.
- El proceso receptor, el que recibe el mensaje.
- Los datos que se esperan recibir por parte del receptor.
- El tipo de dato que se envía.
- El proceso emisor del mensaje.
- La ubicación o espacio en memoria (puntero en C) donde se almacenarán los datos.

- Librería MPI

  La primera versión de esta librería en 1993 utilizo PVM V3 *(Parallel Virtual machine)*, luego en 1994 se ajustaron complementos a la librería de PVM y nace la primera versión de MPI 1.0 *(Message Passing Interface)*
  Una característica llamativa de MPI es que permite trabajar con grupos de procesadores definidos según el programador lo disponga mediante objetos virtuales denominados comunicadores que permiten distribuir los recursos según la tarea a realizar. Con la librería MPI se debe tener claro que sólo se puede declarar una única vez el ambiente en paralelo (sección comprendida entre las

funciones MPI_Init y MPI_Finalize) y que todo el código que este dentro de la zona se ejecutará en simultáneo por todos los procesos.
Dentro de los grupos de funciones para MPI se destacan algunas como:

• Funciones de control de flujo: permiten crear y establecer parámetros de la sección en paralelo como número de procesos a usar, el comunicador, los ID de los procesos de la aplicación, etc.

• Funciones para el manejo de grupos y comunicadores: facilitan la conformación de los grupos de procesadores.

• Funciones de administración de recurso.

• Funciones de comunicación: permiten la interacción (enviar y recibir información) entre diferentes procesos. Según el número de procesos presentes en la comunicación ésta se clasifica en punto a punto y multipunto.

• Funciones para comunicación punto a punto: implican la interacción de dos procesos exclusivamente (maestro y esclavo) que según el tipo de petición para establecer la conexión se dividen en método bloqueante y no bloqueante.

• Funciones para comunicación multipunto: interactúan con múltiples procesos simultáneamente, el uso de ellas requiere que el desarrollador tenga claro el recurso con el que cuenta.

- **SMM** - *Shared Memory Model* mediante la librería **OpenMP**

  El modelo SMM es una abstracción del modelo de multiprocesamiento centralizado. El método de acceso a memoria es llamado UMA (*Uniform Memory Access*), también conocido como SMP (*Symmetric Multi- Processing*).

  El hardware en SMM está basado en FSB (front-side bus) que a su vez es el modelo acceso a memoria usado en UMA (Uniform Memory Access). Consta de un controlador de memoria (MCH) al que se conecta a la memoria general,

las CPU interactúan con el MCH cada vez que necesitan acceder a memoria. También, cuenta con un controlador de I/O que se conecta al MCH.

SMM está fundamentado en un modelo de ejecución denominado *fork/join* que básicamente describe la posibilidad de pasar de una zona secuencial ejecutada por un único hilo maestro (*master thread*) a una zona paralela ejecutada por varios hilos esclavos (*fork*), posteriormente, cuando finalice la ejecución en paralelo, la información y resultados se agrupan de nuevo mediante un proceso de escritura en memoria en un proceso llamado *join*.

- Librería OpenMP

  La gran portabilidad es una característica de MPI debido a que está soportado para C, C++ y Fortran, disponible para sistemas operativos como Solaris, AIX, HP-UX, GNU/Linux, MAC OS, y Windows.

  Además, OpenMP soporta la interacción con el modelo de paso por mensajes (MPM) permitiendo la integración de la librería MPI en las aplicaciones, lo que amplía aún más las alternativas de programación.

  Una particularidad de OpenMP es que permite administrar el recurso en rutinas que contienen cálculos iterativos, mediante la distribución de iteraciones de los ciclos en los diferentes threads (hilos) que maneja la aplicación mediante funciones especiales denominadas constructores.

- **CUDA** - *(Compute Unified Device Architecture)*

  La Gpu *(Graphic Processing Unit)* se caracteriza por tener funciones de punto flotante, en sus inicios nació como alternativas para grandes volúmenes de información, aplicables en el desarrollo de videojuegos, reproductores de vídeo y simulaciones complejas. Posteriormente se el modelo de programación extendida a C genero lo que hoy se conoce como CUDA.

  El hardware de CUDA se compone por la CPU de un equipo principal o host y

una GPU dispuesta en un dispositivo externo o device.

Las ventajas de programación en GPU son:

- Código compacto: una instrucción define N operaciones.
- Reduce la frecuencia de los saltos en el código.
- No se requiere de hardware adicional para detectar el paralelismo.
- Permite la ejecución en paralelo asumiendo N flujos de datos paralelos.
- No maneja dependencias.
- Usa patrones de acceso a memoria continua.

Ademas cuenta con varias memorias tales como: *Memoria compartida de lecto/escritura.- Memoria de constantes - Memoria de texturas*

A partir de todos los puntos mencionados y dadas cada una de las características se optó por el *Lenguaje de programación C* debido a ser un lenguaje de nivel medio que permite acceder a direcciones de memoria como un lenguaje de bajo nivel y además de su sintaxis ser conocida como de alto nivel, esto lo hace un lenguaje más potente en esta linea, también cabe destacar que el lenguaje de programación C es un lenguaje compilado, esto permite que no se necesita utilizar de una maquina virtual o un interprete para procesar sus ejecuciones o el código fuente como lo realizan otros lenguajes, con esto solo se debe compilar una vez y ejecutar el código iterativamente. En términos de tiempos de ejecución de instrucciones como se aprecia en la gráfica de la figura 2.13, se aprecia que el lenguaje de programación C es más veloz que muchos otros lenguajes, debido a las características ya mencionadas.
Ya acotando el rango de lenguajes entre C, C++, Python como se aprecia en la gráfica 2.14, el lenguaje de programación C es considerablemente mas veloz para procesar la estructura Heap y a esto se le añade la característica de ser un lenguaje que soporta paralelismo en todas las plataformas y cuenta con diversas librerías como las ya

mencionadas, siendo utilizadas en esta tesis OpenMP y CUDA las cuales son soportadas en el lenguaje de programación C.

Capítulo 3

Estado del Arte

La regla del vecino más cercano fue originalmente propuesta por Cover y Hart [Hart, 1966], [Cover and Hart, 1967] y a lo largo de las investigaciones está siendo utilizada por varios investigadores. Una razón para el uso de esta regla es su simplicidad conceptual que conduce a la programación directa, si no necesariamente la más eficiente. Este algoritmo utiliza la idea central de la distancia euclidiana entre dos puntos en el espacio, a partir de este cálculo se analiza la similitud entre todos los puntos existentes. En un artículo posterior, Hart [HILBORN, 1968] sugirió un medio de disminución de la memoria y los requisitos de computación. Este artículo introduce una técnica, la regla del vecino más cercano reducido que puede conducir a más ahorros. Los resultados de esta regla se demuestran aplicándola a los datos de "$Iris$" [Freeman, 1969].

El algoritmo kNN [Aha et al., 1991] [Cover and Hart, 1967] tiene grandes requisitos de almacenamiento, pero pueden reducir significativamente el trabajo de aprendizaje de otro tipo de técnicas y aumentar la precisión en la clasificación. Uno de los problemas más grandes que presenta este algoritmo es el alto costo de procesamiento y memoria para revolver las consultas.

Se aplica ampliamente en el reconocimiento de patrones y en clasificación para minería de datos, debido a su simplicidad y baja tasa de error. Antes de la aparición masiva de plataformas paralelas, la realización por fuerza bruta

(exhaustivamente) no se consideraba como una opción válida, especialmente para grandes bases de datos de entrenamiento y espacios de alta densidad. Para reducir el espacio de búsqueda y evitar tantos cálculos de distancia como sea posible, se han propuesto muchos enfoques de indexación. La mayoría de los métodos de recuperación se basan en kd-trees [Bentley and Friedman, 1979]. Hay una gran cantidad de trabajo sobre las adaptaciones de la estructura básica kd-tree para el problema kNN [Brisaboa et al., 2008] [Paredes and Navarro, 2009]. También se han propuesto estructuras no-tree que dividen eficientemente el espacio de búsqueda [Chávez and Navarro, 2000] [Brisaboa et al., 2006] .Las implementaciones basadas en MPI que utilizan estas estructuras también aparecen con mucha frecuencia en la literatura [Plaku and Kavraki, 2007].
La principal desventaja de estos métodos es que necesitan construir y mantener estructuras de datos complejas para el conjunto de datos de referencia. De manera que, kNN se implementa típicamente utilizando métodos de fuerza bruta.

Por lo tanto, las soluciones actuales de GPU han sido de alguna manera más simples y menos eficientes desde el punto de vista del algoritmo. Por ejemplo, Benjamin Bustos et al. [Bustos et al., 2006] propuso una implementación no-CUDA explotar memorias de textura GPU. Su implementación sólo busca el elemento mínimo, lo que simplifica significativamente el problema.

El cálculo de la distancia exhaustiva en conjunción con la clasificación en paralelo se propuso en [Garcia et al., 2008], [Kuang and Zhao, 2009]. Vecente Garcia et al. [Garcia et al., 2008] propuso un orden de inserción paralelo modificado con el fin de obtener sólo los K elementos más cercanos, mientras que Kuang et al. [Kuang and Zhao, 2009] propuso un mejor Radix-sort para realizar la clasificación final.

Existe una versión de GPU de Quicksort que fue propuesta por Daniel Cederman y Philippas Tsigas [Cederman and Tsigas, 2009] que mostró ser más rápido que los algoritmos de clasificación anteriores. Este trabajo se basa en la propuesta que también calcula exhaustivamente todas las distancias, pero utiliza una metodología específica basada en heap para encontrar los K elementos más cercanos, propuesta por Ricardo

Barrientos y José Gómez [Barrientos et al., 2010]

Capítulo 4

Desarrollo

4.1. Introducción

En este capítulo se describirá el proceso de creación del Software ya mencionado anteriormente en capítulos previos, en los cuales se indica que el software contiene una recopilación de diversos algoritmos kNN en diferentes plataformas paralelas y además del kNN secuencial para equipos que no posea las características necesarias para utilizar este software, de este modo se presentara desde la metodología utilizada hasta su desarrollo final con el software en su primera versión terminada. Para esto será necesario describir procesos y etapas que se dividirían en este capítulo.

4.2. Desarrollo interfaz

En esta sección se aborda como se desarrolló la interfaz gráfica, en un principio se estableció *JAVA* como el lenguaje de programación y se utilizo el IDE *netbeans* [and/or its affiliates, 2017], a continuación se explican cada uno de los pasos tomados en consideración al realizar la interfaz gráfica.

En un principio se estableció *JAVA* debido a su capacidad de ser soportado por diversos sistemas operativos, como son Windows, OSx, Linux. Dado que JAVA es un lenguaje interpretado y no es compilado, el interpretador (máquina virtual) de Java

es soportado por todos los sistemas antes mencionados, además el uso de *netbeans* permite la fácil creación de una interfaz gráfica, esto debido a poseer *drag and drop* para sus componentes gráficos, en otras palabras sólo se necesita arrastrar una componente y situarla en la posición necesaria, esta ventaja dada por *netbeans* es útil, debido al menor tiempo que se ejecuta para crear la interfaz y poder otorgar el tiempo a las funciones correspondientes.

Otra característica importante de *JAVA* se debe a que se puede realizar la ejecución de los ejecutables creados luego de la compilación de los algoritmos kNN y retornar los valores del ejecutable a la interfaz, de esto se logra poder potencial de mejor manera el software. De modo que la interfaz gráfica sólo realiza funciones de selección de bases de datos y campos de formularios, en otras palabras la interfaz gráfica sólo es una fachada con la finalidad de no realizar la ejecución de los algoritmos desde la terminal.

La figura 4.1 muestra la interfaz de usuario cuando el software inicia, la cual se compone de la barra superior la cual posee dos menús "File" y "Menus", una sección de *input* que corresponde a los datos de entrada para el algoritmo kNN, una sección de *Result* donde en el primer recuadro de la izquierda se muestra la vista previa de las primeras 1000 lineas del archivo de base de datos, en el recuadro central se muestra la vista previa de las primeras 1000 lineas del archivo de consultas, en el cuadro de la derecha muestra los resultados del proceso, muestra si la carga de los datos fue exitosa, el tiempo de ejecución o algún error que pudiese haber ocurrido, como el fallo en la carga de los datos. En la parte inferior del software indica la cantidad de núcleos que posee la máquina donde el software está instalado.

Menús del software

El software posee dos menús como se menciono anteriormente, estos corresponden a "$File$" y "$Menus$", en el primer menú $File$ posee dos sub-menús New y $Exit$, El

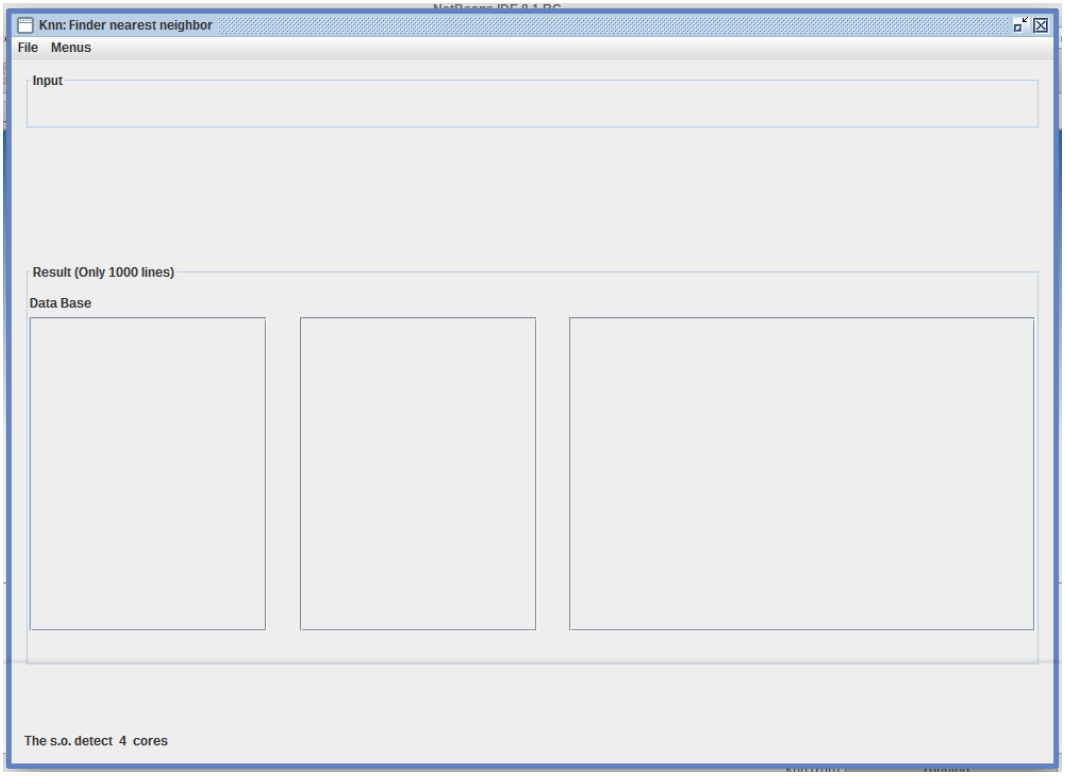


Figura 4.1: Interfaz gráfica inicial del software

primero posee sub-menús que corresponde a los algoritmos que posee el software, estos son:

- *Secuencial*
- *Multihilos*
- *Xeon Phi*
- *GPU*

Cada uno de estos sub-menús, posee al menos un item que corresponde al algoritmo kNN programado de acuerdo a la lógica que su nombre indica, esto quiere decir que el sub-menú multihilos cuenta con a lo menos un item que permite ejecutar el algoritmo kNN multi-hilo, donde este utiliza *OpenMP* para utilizar de mejor manera los recursos de la CPU.

Al seleccionar un item del sub-menú *Multihilos* la interfaz gráfica tiene las siguientes características como se aprecia en la figura 4.2, en la sección *input* se visualizan los campos de entrada que posee el software, en primer caso es la base de datos, esta se carga mediante el botón examinar, para cargar los datos de consulta se utiliza el botón examinar, *sizeobj* es un valor numérico que corresponde a la magnitud del vector ingresado en la base de datos y base de consultas, K es la cantidad de vecinos más cercanos que se desea obtener, $Threads$ por defecto se utiliza la cantidad cantidad de núcleos que posee la máquina donde esta instalado el software, este valor puede ser modificado por la cantidad de hilos que desea el usuario.
Además se cuenta con el botón *Start* para comenzar a ejecutar el las consultas kNN, se añadió bajo el botón *Start* un *check box* que permite ejecutar un $Profiler$ que utiliza la librería PapiC [IT, 2017] que indica estadísticas sobre el uso de los hilos. En la sección *Result* no varia en relación al item seleccionado.

Al seleccionar un item del sub-menú *Secuencial* la interfaz gráfica tiene las siguientes características como se aprecia en la figura 4.3, en la sección *input* se

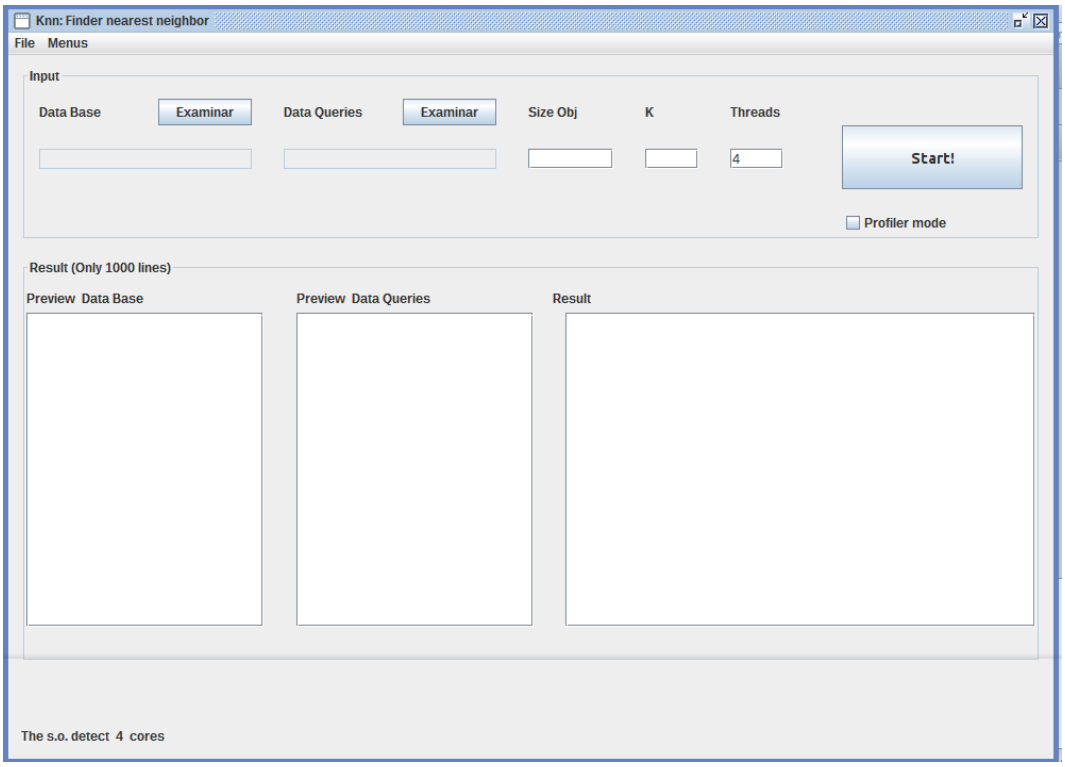


Figura 4.2: Interfaz gráfica item multihilos del software

visualizan los campos de entrada que posee el software, en primer caso es la base de datos, esta se carga mediante el botón examinar, para cargar los datos de consulta se utiliza el botón examinar, *sizeobj* es un valor numérico que corresponde a la magnitud del vector ingresado en la base de datos y base de consultas, K es la cantidad de vecinos más cercanos que se desea obtener.

La selección de los otros item en los sub-menús son similares a los antes mencionados, de modo que solamente varia la cantidad de datos de entrada que se ingresan dado las condiciones de programación de cada algoritmo.

Además el software cuenta con mensajes dirigidos al usuario, estos son dividen en mensajes de alerta y mensajes de éxito. El primero se aprecia en la figura 4.4(a), este mensaje aparece una vez que se presiona el botón *Start* y si algún campo está vacío, indica que campos debe completar para poder ejecutar correctamente el proceso.

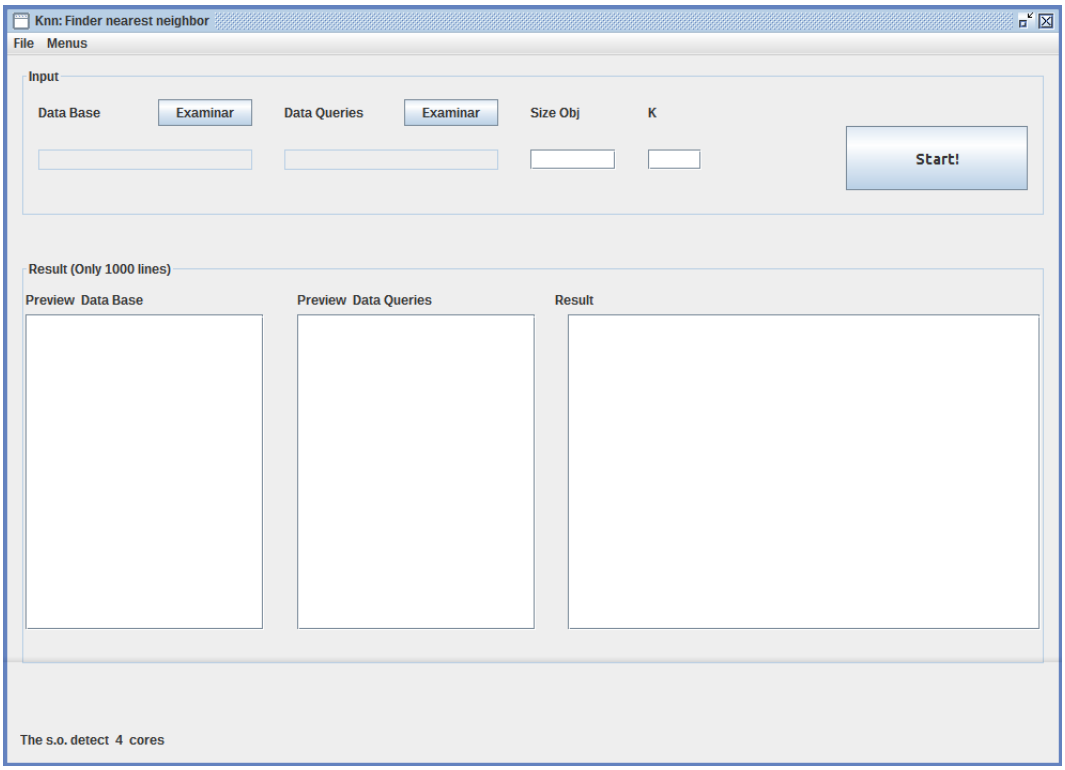


Figura 4.3: Interfaz gráfica item secuencial del software

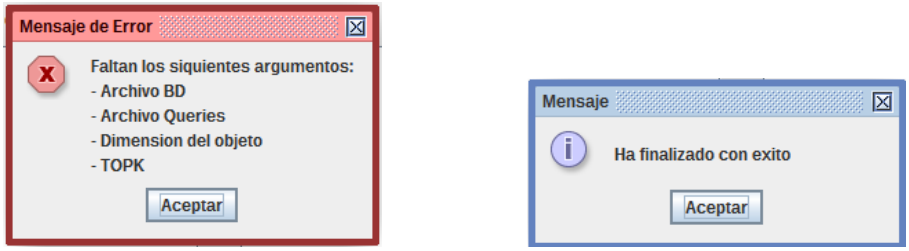
Si no presenta errores el proceso parte correctamente, si los archivos de bases de datos y consultas son archivos correspondientes, el proceso muestra un mensaje de éxito, como se aprecia en la figura 4.4(b), al presionar el botón aceptar se muestra la ventana de los resultados de la consulta kNN, este paso corresponde a la sección 4.7 *Métodos de exportación de resultados.*

4.3. Integración de algoritmo kNN Secuencial

Si bien este algoritmo, no estaba considerado en un comienzo este se añadió para la posibilidad de poder ejecutar el software en equipos que no cuenten con procesadores con multi-núcleos y necesiten realizar consultas kNN. Este algoritmo se realizado con la estructura heap 2.5.

Para implementar el algoritmo kNN secuencial, se definió la estructura heap, la cual se aprecia en el segmento de código en el algoritmo 1, este almacena en *doble dist*

Figura 4.4: Mensajes al usuario



(a) Mensaje 1 - Error

(b) Mensaje 2 - Éxito

Algoritmo 1 Estructura para realizar los *Heap*.

```
1 struct _Elem {
2     double dist;
3     int ind;
4 };
5 typedef struct _Elem Elem;
```

la distancia del elemento consultado, ésta variable es de tipo *double* debido a que las distancias son clasificadas como números reales y es posible obtener tener valores muy altos, a su vez *intind* indica la posición del vector dentro de la base de consultas. Cabe recalcar que solamente se almacenan K resultados en esta estructura, además de sólo almacenar los valores cuya distancia *dist* sean las K menores.

Con el lenguaje de programación C, es posible utilizar la cantidad de memoria justa gracias a la instrucción *malloc* que permite asignar la cantidad de memoria que corresponde al tamaño de la base de datos y la base de consulta, tal como se aprecia en el algoritmo 2. Donde *consultas* se crea una matriz de $N\_queriesxDIM$ donde $N\_queries$ es el número tuplas de vectores contenidas en la base de consulta y DIM es el tamaño del vector. De manera análoga se establece la matriz DB de N_DBxDIM donde N_DB es el número tuplas de vectores contenidas en la base de datos y DIM es el tamaño del vector. A su vez *answer* corresponde a la asignación de memoria para las $N\_queries$ con K vecinos más cercanos donde se almacenan todas las respuestas a las consultas para ser finalmente mostrados. *Heap* es la estructura a utilizar con cada

Algoritmo 2 Utilización de *malloc*.

```
1   Consultas = (double **) malloc(sizeof (double *)*N_QUERIES);
2   for (i = 0; i < N_QUERIES; i++)
3       Consultas[i] = (double *) malloc(sizeof (double)*DIM);
4
5   DB = (double **) malloc(sizeof (double *)*N_DB);
6   for (i = 0; i < N_DB; i++)
7       DB[i] = (double *) malloc(sizeof (double)*DIM);
8
9   answer = (Elem *)malloc(sizeof(Elem)*N_QUERIES*TOPK);
10
11  heap = (Elem *) malloc(sizeof (Elem) * TOPK);
```

consulta, esta corresponde a un heap del tipo heap max, donde el nodo padre es el mayor valor de la estructura.

El desarrollo central del algoritmo para obtener los K vecinos más cercanos está dado por el algoritmo 3, este algoritmo indica las siguientes acciones, se establece un contador $n\_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como comparador debido a que se deben ingresar K elementos al heap, mientras n_elem sea menor que el número k se ingresaran los datos al heap en caso contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor se extrae la raíz y se añade el elemento actual y se reordena el heap. Este proceso itera exhaustivamente hasta que no queden elementos en la base de datos, cuando el heap está finalizado se almacenan sus resultados en la matriz *answer* y se debe continuar con la siguiente consulta hasta que se realicen todas las consultas del base de consultas.

4.3.1. Funciones claves del algoritmo secuencial

Distancia : Algoritmo 4 es la función de cálculo de la distancia entre vectores, tanto del vector de la base de datos y el vector de la base de consultas, este cálculo se realiza con la fórmula (4.1).

Dados $A = (X_1, Y_1)$ y $B = (X_2, Y_2)$

$$distancia = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2} \tag{4.1}$$

Algoritmo 3 Proceso iterativo de una consulta kNN.

```
for (i = 0; i < N_QUERIES; i++) {
        n_elem = 0;
        for (j = 0; j < N_DB; j++) {
                d = distancia(Consultas[i], DB[j]);
                if(n_elem<TOPK){
                        e_temp.dist = d;
                        e_temp.ind = j;
                        inserta2(heap, &e_temp, &n_elem);
                }
                else{
                        if (d < topH(heap, &n_elem)) {
                                e_temp.dist = d;
                e_temp.ind = j;
                                popush2(heap, &n_elem, &e_temp);
                }}
        }
        for (int k = 0; k < TOPK; ++k){
                extrae2(heap, &n_elem, &e_temp);
                answer[i*TOPK+k].ind = e_temp.ind;
                answer[i*TOPK+k].dist = e_temp.dist;
        }
}
```

Algoritmo 4 Cálculo de distancias entre vectores.

```
double distancia(double *p1, double *p2) {
    int i = 0;
    double suma = 0;

    for (i = 0; i < DIM; i++)
        suma += ((p1[i] - p2[i])*(p1[i] - p2[i]));
    return sqrt(suma);
}
```

Esta ecuación puede ser llevada a vectores de tamaño n.

Sumado a esta función existen las funciones para manipular un heap, estas corresponden a obtener el valor nodo padre (Algoritmo 5), realizar una inserción de un elemento, extraer un elemento y una añadida para facilitar el procedimiento del algoritmo cómo es el caso de una extracción e inserción que se utiliza cuando se debe cambiar un nodo dentro del heap y realizar el reordenamiento del heap.
El algoritmo 5 retorna el valor de la raíz (nodo padre) del heap si no está vacío, en caso contrario retorna el valor máximo de un *double*.

El algoritmo 6 realiza la inserción de un elemento, guardando la distancia y la posición del elemento, el contador n_elem incrementa en 1 luego de la inserción, y se realiza el ordenamiento del heap. Esta función es la que se utiliza en el algoritmo 1

Algoritmo 5 Valor de la raíz del heap.

```
1  double topH(Elem *heap, int *n_elem) {
2      if (*n_elem == 0)
3          return DBL_MAX;
4      return heap[0].dist;
5  }
```

Algoritmo 6 Realiza la inserción de un elemento al heap.

```
1  void inserta2(Elem *heap, Elem *elem, int *n_elem) {
2      int i;
3      Elem temp;
4
5      heap[*n_elem].dist = elem->dist;
6      heap[*n_elem].ind = elem->ind;
7      (*n_elem)++;
8      for (i = *n_elem; i > 1 && heap[i - 1].dist > heap[(i / 2) - 1].dist; i = i / 2) {
9          temp = heap[i - 1];
10         heap[i - 1] = heap[(i / 2) - 1];
11         heap[(i / 2) - 1] = temp;
12     }
13 }
```

hasta llenar el heap.

El algoritmo 7 realiza la extracción de un elemento, guardando la distancia y la posición del elemento, el contador *n_elem* decrementando en 1 luego de la extracción, y se realiza el ordenamiento del heap. Esta función es la que se utiliza para completar la matriz *answer*.

El algoritmo 8 realiza la extracción del elemento raíz y la posterior inserción del elemento cuya distancia era menor que la distancia del elemento raíz, esta función guarda la distancia y la posición del elemento, con esto luego se realiza el ordenamiento del heap. Esta función es la que se utiliza en el algoritmo 1 cuando el heap está lleno y se obtiene un elemento que cuya distancia es menor a la distancia de la raíz.

Para realizar la integración de este algoritmo en la interfaz gráfica, se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec*(). La implementación de esta rutina permite ejecutar otro programa o ejecutable y luego obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 9 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

Algoritmo 7 Realiza la extracción de un elemento al heap.

```
1  void extrae2(Elem *heap, int *n_elem, Elem *elem_extraido) {
2      int i, k;
3      Elem temp;
4
5      (*elem_extraido).dist = heap[0].dist;
6      (*elem_extraido).ind = heap[0].ind;
7
8      heap[0] = heap[(*n_elem) - 1]; // Movemos el ultimo a la raiz y achicamos el heap
9      (*n_elem)--;
10     i = 1;
11     while (2 * i <= *n_elem) // mientras tenga algun hijo
12     {
13         k = 2 * i; //el hijo izquierdo
14         if (k + 1 <= *n_elem && heap[(k + 1) - 1].dist > heap[k - 1].dist)
15             k = k + 1; //el hijo derecho es el mayor
16         if (heap[i - 1].dist > heap[k - 1].dist)
17             break; //es mayor que ambos hijos
18
19         temp = heap[i - 1];
20         heap[i - 1] = heap[k - 1];
21         heap[k - 1] = temp;
22         i = k; //lo intercambiamos con el mayor hijo
23     }
24     return;
25 }
```

Algoritmo 8 Realiza la extracción y la inserción de un elemento al heap.

```
1
2  void popush2(Elem *heap, int *n_elem, Elem *elem) {
3      int i, k;
4      Elem temp;
5
6      heap[0].dist = elem->dist;
7      heap[0].ind = elem->ind;
8
9      i = 1;
10     while (2 * i <= *n_elem) // mientras tenga algun hijo
11     {
12         k = 2 * i; //el hijo izquierdo
13         if (k + 1 <= *n_elem && heap[(k + 1) - 1].dist > heap[k - 1].dist)
14             k = k + 1; //el hijo derecho es el mayor
15         if (heap[i - 1].dist > heap[k - 1].dist)
16             break; //es mayor que ambos hijos
17
18         temp = heap[i - 1];
19         heap[i - 1] = heap[k - 1];
20         heap[k - 1] = temp;
21         i = k; //lo intercambiamos con el mayor hijo
22     }
23     return;
24 }
```

Algoritmo 9 Ejecutar un programa o ejecutable desde *Java*.

```
1
2  import java.io.*;
3
4  public class llamarruntime
5  {
6     public static void main(String[] args)
7     {
8        try{
9           Process theProcess =
10                 Runtime.getRuntime().exec(Ejecutable);
11       }
12       catch(IOException e){
13          System.err.println("Error en el metodo exec()");
14          e.printStackTrace();
15       }
16    }
17 }
```

4.4. Integración de algoritmo kNN Paralelo Multi-hilos

Este algoritmo a posee ciertas semejanzas y diferencias con respecto a la versión secuencial, éste utiliza *malloc* como se indica en el algoritmo 2, estableciendo los mismos atributos y designando la memoria de la misma manera que la versión secuencial, léase 4.3.

La librería *OpenMP* establece variables de tipo compartida *Shared* y de tipo privadas *Private*, donde con una variable del tipo compartida se puede obtener su valor o realizar alguna modificación sin restricciones desde cualquier hilo, a su vez una variable del tipo privada solamente puede ser accesible por un hilo y se crea una copia de la variable para tantos hilos creados existan. De esta manera se han establecido sólo variables del tipo *Shared* para este algoritmo.

El desarrollo central del algoritmo para obtener los K vecinos más cercanos está dado por el algoritmo 10, este algoritmo indica las siguientes acciones, se establece un contador $n\_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como comparador debido a que se debe ingresar K elementos al heap, mientras n_elem sea menor que el número K se ingresaran los datos al heap en caso

Algoritmo 10 Proceso iterativo de una consulta kNN multi-core.

```
#pragma omp master
for (i = tid; i < N_QUERIES; i += procs) {
        n_elem = 0;
    for (j = 0; j < N_DB; j++) {
        d = distancia(Consultas[i], DB[j]);
        if(n_elem<TOPK){
                e_temp.dist = d;
                e_temp.ind = j;
                inserta2(heap, &e_temp, &n_elem);
        }
        else{
                if (d < topH(heap, &n_elem)) {
                e_temp.dist = d;
                e_temp.ind = j;
            }}
        }
        for (j = 0; j < TOPK ; j++) {
                extrae2(heap, &n_elem, &e_temp);
                answer[i*TOPK+j].ind = e_temp.ind;
                answer[i*TOPK+j].dist = e_temp.dist;
        }
}

#pragma omp barrier
```

contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor, se extrae la raíz, se añade el elemento actual y se reordena el heap. Este proceso itera paralelamente hasta que no queden elementos en la base de datos, cuando el heap está finalizado se almacenan sus resultados en la matriz *answer* y se debe continuar con la siguiente consulta hasta que se realicen todas las consultas de la base de datos de consultas.

A diferencia del método secuencial, el método multi-hilo utiliza la librería OpenMP, por tanto como se muestra en el algoritmo 10 *#pragma omp master* indica el trabajo en la zona paralela, de manera que el ciclo for realiza un paso cíclico con una cantidad determinada de hilos dada por *procs*, donde *tid* es el identificador de cada hilo, esto quiere decir que si utilizo 2 hilos el hilo 0 con $tid| = 0$ realiza la consulta 0 y el hilo 1 con $tid = 1$ realiza la consulta 1 simultáneamente finalizado el trabajo de al menos uno de éstos, el hilo con su actividad finalizada toma el valor correspondiente, en caso del hilo 0 con $tid = 0$ le corresponde tomar la consulta 2 y al hilo 1 con $tid = 1$ le corresponde el hilo 3. Este proceso se realiza hasta llegar al final de la base de datos de consultas $N\_queries$.

4.4.1. Funciones claves del algoritmo multi-hilos

Estas funciones son las mismas que el método secuencial, donde se consideran *topH* para saber el valor del nodo raíz, *inserta*2 para insertar un nodo en un heap con el número de elementos menor al de K elementos, *extrae*2 extrae el nodo raíz del heap, *popush*2 realiza la extracción del nodo raíz e inserción de un nuevo nodo de menor valor que la raíz extraída, léase 4.3.1
Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec*(). La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 9 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.5. Integración de algoritmo kNN Paralelo Xeon Phi

A diferencia de los métodos anteriores acá se utiliza un coprocesador, de modo que se debe realizar la comunicación entre la CPU y el Coprocesador (*Intel Xeon Phi*). A continuación se describe en detalle las funciones de este método.

Con el lenguaje de programación C, es posible utilizar la cantidad de memoria justa gracias a la instrucción *malloc* que permite asignar la cantidad de memoria que corresponde al tamaño de la base de datos y la base de consulta, tal como se aprecia en el algoritmo 11. Donde *queries* se crea una matriz de *num_queriesxdimaux* donde *num_queries* es el número tuplas de vectores contenidas en la base de consulta y *dimaux* es el tamaño del vector. De manera análoga se establece la matriz *db* de *num_dbxdimaux* donde *num_db* es el número tuplas de vectores contenidas en la base de datos y *dimaux* es el tamaño del vector. A su vez *answer* corresponde a la asignación de memoria para las *num_queries* con K vecinos más cercanos donde se

almacenan todas las respuestas a las consultas para ser finalmente mostrados. *Heap* es la estructura a utilizar con cada consulta, ésta corresponde a un heap del tipo heap max, donde el nodo padre es el mayor valor de la estructura. La Xeon phi permite sólo permite realizar el trabajo con vectores de modo que las matrices deben ser trabajadas como vectores y además los vectores deben ser múltiplos de 16, de modo que para solucionar este problema y poder trabajar con la medida de cualquier vector de entrada se parametriza a su vector multiplo de 16 superior, en otras palabras si ingresa un vector de tamaño 8 se parametriza a 16, si ingresa un vector 17 se parametriza a 32 y los valores restantes son establecidos como 0, para no interferir en el cálculo de la distancia.

El desarrollo central del algoritmo para obtener los K vecinos más cercanos está dado por el algoritmo 12, este algoritmo indica las siguientes acciones, las operaciones de este algoritmo están dentro de la región paralela de la *Xeon phi* (*#pragma*) de manera que se debe enviar desde la CPU a la *Xeon phi* todas las variables o constantes a utilizar, donde se deben cumplir ciertas restricciones como vectorizar la matriz (explicado anteriormente), además se especifican claramente cuales son datos exclusivamente de entrada *In* y exclusivamente datos de salida *Out* o datos que pueden ser de entrada y salida *(Inout)*. La *Xeon phi* a través de la librería *OpenMP* utiliza además variables de tipo compartida *Shared* y variables del tipo privadas *Private*, donde una variable del tipo compartida puede obtener su valor o realizar alguna modificación sin restricciones, a su vez una variable del tipo privada solamente puede ser accesible por un hilo y se crea una copia de la variable para tantos hilos creados existan. A diferencia del método anterior que sólo utilizaba variables tipo *Shared*, en este método las variables del tipo *Private* se han establecido las siguientes variables *i, j, thread_num* de modo que cada hilo puede realizar independientemente sus iteraciones en los ciclos sin ser interrumpido por otro, esto debido a que se utilizan vectores y se debe emular la lectura de la matriz a través de los vectores. Dentro de la región paralela *#pragma omp parallel* se crea un heap de tamaño K, luego se establece un contador $n\_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como

Algoritmo 11 Utilización de *malloc*.

```
1   db= (double **)malloc(sizeof(double *)*num_db);
2   for (i=0; i<num_db; i++)
3      db[i] = (double *)malloc(sizeof(double)*dimaux);
4   queries = (double **)malloc(sizeof(double *)*num_queries);
5   for (i=0; i<num_queries; i++)
6      queries[i] = (double *)malloc(sizeof(double)*dimaux);
7   answer = (Elem *)malloc(sizeof(Elem)*num_queries*k);
8
9   //Se transfieren datos de una matriz a un vector
10  db_vector = (double *)_mm_malloc(sizeof(double)*dimaux*num_db, 64);
11  for (i=0; i < dimaux*num_db; i++)
12      db_vector[i] = 0.0;
13  if (sizeof(double)*dimaux*num_queries < 64)
14  {
15        queries_vector = (double *)_mm_malloc(sizeof(double)*16, 64);
16        for (i=0; i < 16; i++)
17            queries_vector[i] = 0.0;
18  }
19  else
20  {
21        queries_vector = (double *)_mm_malloc(sizeof(double)*dimaux*num_queries, 64);
22        for (i=0; i < dimaux*num_queries; i++)
23            queries_vector[i] = 0.0;
24  }
```

comparador debido a que se debe ingresar K elementos al heap, mientras *n_elem* sea menor que el número *k* se ingresaran los datos al heap en caso contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor se extrae la raíz y se añade el elemento actual y se reordena el heap. Este proceso itera paralelamente hasta que no queden elementos en la base de datos, cuando el heap está finalizado se almacenan sus resultados en la matriz *answer*. Este proceso dentro de la región paralela *#pragma omp parallel* realiza cada uno de los pasos mencionados por cada hilo utilizado en el proceso

4.5.1. Funciones claves del algoritmo Xeon phi

A diferencia de los métodos anteriores donde solamente se utiliza una *CPU*, aca se emplea tanto la *CPU* como un *Coprocesador* el cual sólo realiza tareas específicas (léase 2.2). A continuación se detallan las principales funciones que se implementaron en *Xeon phi*.

Algoritmo 12 Proceso iterativo de una consulta kNN Xeon phi.

```
1  #pragma offload target(mic:0) in(dim) in(db_vector:length(num_db*dimaux))
       in(queries_vector:length(num_queries*dimaux)) out(answer:length(k*num_queries))
2  {
3     #pragma omp parallel private(i, j, thread_num) shared(db_vector, num_db,
          queries_vector, num_queries, dimaux, k, answer, num_threads)
4     {
5
6        Elem *heap;
7        heap = (Elem *)malloc(sizeof(Elem)*k);
8        #pragma omp master
9        {
10          num_threads = omp_get_num_threads();
11          printf("run with %d threads\n", num_threads);
12       }
13       #pragma omp barrier
14       thread_num = omp_get_thread_num();
15       int n_elem;
16       Elem e_temp;
17       double d;
18
19
20       for(i=thread_num*dimaux; i<num_queries*dimaux; i+=num_threads*dimaux){
21          n_elem = 0;
22          for(j=0; j<k; j++){
23             e_temp.dist = distancia(&(queries_vector[i]), &(db_vector[j*dimaux]),
                   dimaux);
24             e_temp.ind = j;
25             inserta2(heap, &e_temp, &n_elem);
26          }
27
28          for(j=k; j<num_db; j++){
29             d = distancia(&(queries_vector[i]), &(db_vector[j*dimaux]), dimaux);
30             if(d < topH(heap, &n_elem))
31                {
32                   e_temp.dist = d;
33                   e_temp.ind = j;
34                   popush2(heap, &n_elem, &e_temp);
35                }
36          }
37
38          for(j=0; j<k; j++){
39             extrae2(heap, &n_elem, &e_temp);
40             printf("%d ind = %d :: dist = %f posicion:: %d \n", j, e_temp.ind,
                   e_temp.dist,(i/dimaux)*k+j);
41             answer[(i/dimaux)*k+j].ind = e_temp.ind;
42             answer[(i/dimaux)*k+j].dist = e_temp.dist;
43          }
44
45
46       }
47       free(heap);
48    }
49 }
```

Algoritmo 13 Función distancia *Xeon phi*.

```
__attribute__((target(mic))) double distancia(double *p1, double *p2, int DIM){
    int i=0;
    double suma=0.0;
   __assume_aligned(p1, 64);
   __assume_aligned(p2, 64);
    #pragma vector aligned
    #pragma ivdep
    #pragma simd
    for (i=0; i < DIM; i++){
        suma += (p1[i]-p2[i])*(p1[i]-p2[i]);
    }
    return sqrt(suma);
}
```

Algoritmo 14 Pasa una matriz a vector.

```
void matrixToVector(double **matrix, int num_cols, int num_rows, double *vector){
    int i,j;
    for(i=0; i<num_rows; i++)
        for(j=0; j<num_cols; j++)
            vector[(i*num_cols)+j] = matrix[i][j];
}
```

Distancia : Algoritmo 14 es la función de cálculo de la distancia entre vectores, tanto del vector de la base de datos y el vector de la base de consultas, este cálculo se realiza con la fórmula (4.1). Esta función se realiza en la *Xeon phi* para indicar que se realizara en el coprocesador se utiliza la sintaxis ***__atribute__((target(mic)))***. Las sintaxis ***__assume_aligned(), #pragma vector aligned, #pragma ivdep, #pragma simd*** alinea los vectores y permite que el cálculo de la distancia se realice correctamente.

Debido a que la *Xeon phi* no permite la utilización de matrices se implementa la función *matrixToVector* (Algoritmo 14) para traspasar la matriz a vector, esta función corresponde a la CPU no a la *Xeon phi* de manera que no utiliza sintaxis especiales.

De manera similar a los métodos anteriores (Secuencial y Multi-núcleo) se emplean las mismas funciones, pero no se ejecutan en la CPU, estas se ejecutan en el *Coprocesador* para esto se añade al principio de cada función la sintaxis ***__atribute__((target(mic)))*** por ejemplo el algoritmo 15 que devuelve la raíz del

Algoritmo 15 Valor de la raíz del heap.

```
1 __attribute__((target(mic))) double topH(Elem *heap, int *n_elem)
2 {
3     if ((*n_elem) == 0)
4         return MAXDOUBLE;
5     return heap[0].dist;
6 }
```

heap, solamente varia en la sintaxis indicada anteriormente. A su vez tanto *inserta*2, *extrae*2, *popush*2, se deben ejecutar en la *Xeon phi* de modo que se debe añadir al principio de cada función la sintaxis ***__atribute__((target(mic)))***.

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec*(). La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 9 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.6. Integración de algoritmo kNN Paralelo GPU

Para GPU es necesario utilizar $CUDA$ que es una arquitectura de cálculo paralelo de $NVIDIA$ que aprovecha la potencia de la GPU, si bien la lógica central del algoritmo kNN no varia a los métodos presentados anteriormente, sólo difiere en estructuras y/o sintaxis específicas de $CUDA$. De esta manera nos enfocaremos solamente en las funciones específicas utilizadas en este método. La arquitectura CUDA como se menciona en el capítulo 2.2.2, necesita una librería en el lenguaje C, para poder contar con todas las capacidades de CUDA de este modo con la instrucción *#include cuda.h* .

Con el lenguaje de programación C, es posible utilizar la cantidad de memoria justa gracias a la instrucción *malloc* que permite asignar la cantidad de memoria

Algoritmo 16 Proceso iterativo de una consulta kNN en GPU.

```
1
2   while(contQ < N_QUERIES){
3     contQ += Q;
4     if (contQ > N_QUERIES)
5     N_BLOQUES = N_QUERIES - (contQ-Q);
6     printf("\nN_BLOQUES = %d :: T_per_BLOCK = %d\n", N_BLOQUES, T_per_BLOCK);
7
8     Batch_Heap_Reduction<<<N_BLOQUES, T_per_BLOCK>>> (Elems, (int)pitch, HEAPS_dev,
          (int)pitch_H, QUERY_dev, (int)pitch_Q, arr_Dist, (int)pitch_Dist, Q*cont,
          res_final);
9
10    if (cudaSuccess != cudaMemcpy((double *)res_final_H, (double *)res_final,
          sizeof(double)*Q*TOPK, cudaMemcpyDeviceToHost)){
11      printf("\nERROR 41 :: cudaMemcpy :: iteraH\n");
12      cudaThreadExit();
13      return 0;
14    }
15    cont++;
16  }
```

que corresponde al tamaño de la base de datos y la base de consulta, tal como se aprecia en el algoritmo 22. Donde *Consultas* se crea una matriz de $N\_Queries$ *x dimension* donde $N\_QUERIES$ es el número tuplas de vectores contenidas en la base de consulta y *dimension* es el tamaño del vector. De manera análoga se establece la matriz *vectores* de N_ELEM *x dimension* donde N_ELEM es el número tuplas de vectores contenidas en la base de datos y *dimension* es el tamaño del vector. A su vez res_final_H corresponde a la asignación de memoria para las $N\_Queries$ con K vecinos más cercanos donde se almacenan todas las respuestas a las consultas para ser finalmente mostrados. $Linea\_temp$ es la estructura a utilizar con cada consulta, esta corresponde a un heap del tipo heap max, donde el nodo padre es el mayor valor de la estructura, considerar que la lectura de as consultas se realiza de modo inverso a la lectura de las bases de datos.

Como se aprecia en el algoritmo 16, se utiliza la función $Batch\_Heap\_Reduction$ con $N\_BLOQUES$ que indica la cantidad de bloques a utilizar y T_per_BLOCK que indica la cantidad de hilos por bloques.

El algoritmo **??**lst:batch) realiza todo el proceso de cada una de las consultas, donde cada bloque realiza una consulta con los T_per_BLOCK hilos por bloque. De manera

Algoritmo 17 Batch Heap Reduction.

```
1  __global__ void Batch_Heap_Reduction(double *DB_dev, int pitch_DB, Elem *heap, int
       pitch_H, double *QUERY_dev, int pitch_QUERY, Elem *arr_Dist, int pitch_Dist, int
       beginQ, double *res_final){
2    int i, j, n_elem=0, n_elemWarp=0,id;
3    Elem eresult;
4    __shared__ Elem matrizWarp[TOPK][TAM_WARP];
5    __shared__ Elem heapfin[TOPK][1];
6    __shared__ double query[DIM];
7    id = threadIdx.x + (blockDim.x * blockIdx.x);
8    for (i=threadIdx.x; i < DIM; i += blockDim.x)
9      query[i] = ((double *)((char *)QUERY_dev + ((blockIdx.x + beginQ) *
           (int)pitch_QUERY)))[i];
10   __syncthreads();
11   for (i=threadIdx.x; i < NE; i += blockDim.x){
12     ((Elem *)((char *)arr_Dist + (blockIdx.x*pitch_Dist)))[i].dist =
           distancia_trans(DB_dev, pitch_DB, i, query);
13     ((Elem *)((char *)arr_Dist + (blockIdx.x*pitch_Dist)))[i].ind = i;
14   }
15   for(i=threadIdx.x; i < NE; i += blockDim.x){
16     if (n_elem >= TOPK){
17       if (topH(heap, id) > ((Elem *)((char *)arr_Dist +
             (blockIdx.x*pitch_Dist)))[i].dist)
18         popush(heap, &(((Elem *)((char *)arr_Dist + (blockIdx.x*pitch_Dist)))[i]),
               &n_elem, pitch_H, id);
19       }
20     else
21       insertaH(heap, &(((Elem *)((char *)arr_Dist + (blockIdx.x*pitch_Dist)))[i]),
             &n_elem, pitch_H, id);
22   }
23   __syncthreads();
24   if (threadIdx.x < TAM_WARP){
25     for(j=id; j < blockDim.x*(blockIdx.x+1); j += TAM_WARP){
26       n_elem = TOPK;
27       for(i=0; i < TOPK; i++){
28         extraeH(heap, &n_elem, pitch_H, j, &eresult);
29         if (n_elemWarp < TOPK)
30           insertaH(&(matrizWarp[0][0]), &eresult, &n_elemWarp, sizeof(Elem)*TAM_WARP,
                 threadIdx.x);
31         else
32           if (topH(&(matrizWarp[0][0]), threadIdx.x) > eresult.dist)
33             popush(&(matrizWarp[0][0]), &eresult, &n_elemWarp, sizeof(Elem)*TAM_WARP,
                   threadIdx.x);
34       }
35     }
36   }
37   __syncthreads();
38   if (threadIdx.x == 0){
39     n_elem = 0;
40     for(j=0; j < TAM_WARP; j++){
41       for(i=0; i < TOPK; i++)
42         if (n_elem < TOPK)
43           insertaH((Elem *)heapfin, &(matrizWarp[i][j]), &n_elem, sizeof(Elem), 0);
44         else
45           if (topH((Elem *)heapfin, 0) > matrizWarp[i][j].dist)
46             popush((Elem *)heapfin, &(matrizWarp[i][j]), &n_elem, sizeof(Elem), 0);
47     }
48     for (i=TOPK*blockIdx.x; i < (TOPK*blockIdx.x)+TOPK; i++){
49       extraeH(&(heapfin[0][0]), &n_elem, sizeof(Elem), 0, &eresult);
50       res_final[i] = eresult.dist;
51     }
52   }
53   return;
54 }
```

Algoritmo 18 Utilización de *malloc*.

```
1  res_final_H = (double *)malloc(sizeof(double)*Q*TOPK);
2
3  Elem *linea_temp = (Elem *)malloc(sizeof(Elem)*Q*T_per_BLOCK);
4
5
6  consultas =(double **)malloc(sizeof(double *)*N_QUERIES);
7  for (i=0; i<N_QUERIES; i++)
8    consultas[i] = (double *)malloc(sizeof(double)*dimension);
9
10 vectores =(double **)malloc(sizeof(double *)*dimension);
11 for (i=0; i<dimension; i++)
12   vectores[i] = (double *)malloc(sizeof(double)*N_ELEM);
```

que realiza la evaluación de la consulta y se almacenan en el Heap las primeras K, luego de esto se realiza la evaluación con la raiz (nodo padre) del heap, y se pregunta si el elemento actual es menor que el elemento del nodo se reemplaza sino se mantiene el heap de la misma manera.

4.6.1. Funciones claves del algoritmo GPU NVIDIA

De manera diferente de los métodos secuenciales y multi-hilos donde solamente se utiliza una *CPU*, acá se emplea tanto la *CPU* como un *Coprocesador* el cual sólo realiza tareas específicas (léase 2.2). A continuación se detallan las principales funciones que se implementaron en *GPU NVDIA*.

Si bien la lógica principal del algoritmo kNN no varia, si lo hacen las instrucciones para ser implementadas como se a observado en los distintos métodos, de manera que el caso de la GPU no es diferente, ya que esta cuenta con su propia librería de CUDA y difiere de acuerdo con la arquitectura propia que está diseñada la GPU. Cada función que es implementada en GPU tiene como inicio de la declaración de la función *__device__*

A continuación se describen las principales funciones en GPU, donde el algoritmo 19 se encarga de realizar la inserción de los elementos en un heap, este a diferencia de los anteriores difiere en la forma de su sintaxis dado particularmente en la forma de

Algoritmo 19 Inserta un elemento en el heap.

```
1
2  __device__ void insertaH(Elem *heap, Elem *elem, int *n_elem, int pitch, int id){
3    int i;
4    Elem temp;
5
6    ((Elem *)((char *)heap + (*n_elem)*pitch))[id].dist = elem->dist;
7    ((Elem *)((char *)heap + (*n_elem)*pitch))[id].ind = elem->ind;
8    (*n_elem)++;
9
10   for (i = *n_elem; i>1 && ((Elem *)((char *)heap + (i-1)*pitch))[id].dist > ((Elem
         *)((char *)heap + ((i/2)-1)*pitch))[id].dist; i=i/2){
11     //Intercambiamos con el padre
12     temp.dist = ((Elem *)((char *)heap + (i-1)*pitch))[id].dist;
13     temp.ind = ((Elem *)((char *)heap + (i-1)*pitch))[id].ind;
14     ((Elem *)((char *)heap + (i-1)*pitch))[id].dist = ((Elem *)((char *)heap +
           ((i/2)-1)*pitch))[id].dist;
15     ((Elem *)((char *)heap + (i-1)*pitch))[id].ind = ((Elem *)((char *)heap +
           ((i/2)-1)*pitch))[id].ind;
16     ((Elem *)((char *)heap + ((i/2)-1)*pitch))[id].dist = temp.dist;
17     ((Elem *)((char *)heap + ((i/2)-1)*pitch))[id].ind = temp.ind;
18   }
19   return;
20 }
```

acceder a cierta posición del heap, esto es debido a la forma en que Nvidia realiza el acceso a este tipo de estructuras. El algoritmo 20 realiza la extracción de la raíz del heap y lo reordena como corresponde. a su vez en algoritmo **??** si el heap está lleno realiza la extracción de la raíz del heap e inserta el nuevo elemento y reorganiza el heap. A su vez el algoritmo **??** calcula la distancia euclidiana pero con la matriz de consultas transversa debido que el acceso de esta manera en GPU permite ser aun más rápido en ejecución.

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec*(). La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 9 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

Algoritmo 20 Realiza la extracción de un elemento del *heap*

```
1
2  __device__ void extraeH(Elem *heap, int *n_elem, int pitch, int id, Elem *eresult){
3
4    int i, k;
5    Elem temp;
6    eresult->dist = ((Elem *)((char *)heap+0))[id].dist; //Se guarda el maximo
7    eresult->ind = ((Elem *)((char *)heap+0))[id].ind;
8
9    ((Elem *)((char *)heap+0))[id].dist = ((Elem *)((char *)heap +
         ((*n_elem)-1)*pitch))[id].dist;// Movemos el ultimo a la raiz y achicamos el
         heap
10   ((Elem *)((char *)heap+0))[id].ind = ((Elem *)((char *)heap +
         ((*n_elem)-1)*pitch))[id].ind;
11   (*n_elem)--;
12   i = 1;
13   while(2*i <= *n_elem)
14   {
15     k = 2*i; //el hijo izquierdo
16     if(k+1 <= *n_elem && ((Elem *)((char *)heap + ((k+1)-1)*pitch))[id].dist > ((Elem
           *)((char *)heap + (k-1)*pitch))[id].dist)
17       k = k+1;
18
19     if(((Elem *)((char *)heap + (i-1)*pitch))[id].dist > ((Elem *)((char *)heap +
           (k-1)*pitch))[id].dist)
20       break;
21
22     temp.dist = ((Elem *)((char *)heap + (i-1)*pitch))[id].dist;
23     temp.ind = ((Elem *)((char *)heap + (i-1)*pitch))[id].ind;
24     ((Elem *)((char *)heap + (i-1)*pitch))[id].dist = ((Elem *)((char *)heap +
           (k-1)*pitch))[id].dist;
25     ((Elem *)((char *)heap + (i-1)*pitch))[id].ind = ((Elem *)((char *)heap +
           (k-1)*pitch))[id].ind;
26     ((Elem *)((char *)heap + (k-1)*pitch))[id].dist = temp.dist;
27     ((Elem *)((char *)heap + (k-1)*pitch))[id].ind = temp.ind;
28     i = k;    //lo intercambiamos con el mayor hijo
29   }
30   return;
31 }
```

Algoritmo 21 Obtener el valor de la raíz del *heap*

```
1
2  __device__ double topH(Elem *heap, int id){
3    return ((Elem *)((char *)heap + 0))[id].dist;
4  }
```

Algoritmo 22 Realiza la extracción y la inserción de un elemento al *heap*

```
1
2  __device__ void popush(Elem *heap, Elem *elem, int *n_elem, int pitch, int id){
3
4    int i, k;
5    Elem temp;
6
7    ((Elem *)((char *)heap+0))[id].dist = elem->dist;
8    ((Elem *)((char *)heap+0))[id].ind  = elem->ind;
9
10   i = 1;
11   while(2*i <= *n_elem) // mientras tenga algun hijo
12   {
13     k = 2*i; //el hijo izquierdo
14     if(k+1 <= *n_elem && ((Elem *)((char *)heap + ((k+1)-1)*pitch))[id].dist > ((Elem
         *)((char *)heap + (k-1)*pitch))[id].dist)
15       k = k+1;   //el hijo derecho es el mayor
16
17     if(((Elem *)((char *)heap + (i-1)*pitch))[id].dist > ((Elem *)((char *)heap +
         (k-1)*pitch))[id].dist)
18       break;   //es mayor que ambos hijos
19
20     temp.dist = ((Elem *)((char *)heap + (i-1)*pitch))[id].dist;
21     temp.ind = ((Elem *)((char *)heap + (i-1)*pitch))[id].ind;
22     ((Elem *)((char *)heap + (i-1)*pitch))[id].dist = ((Elem *)((char *)heap +
         (k-1)*pitch))[id].dist;
23     ((Elem *)((char *)heap + (i-1)*pitch))[id].ind = ((Elem *)((char *)heap +
         (k-1)*pitch))[id].ind;
24     ((Elem *)((char *)heap + (k-1)*pitch))[id].dist = temp.dist;
25     ((Elem *)((char *)heap + (k-1)*pitch))[id].ind = temp.ind;
26     i = k;    //lo intercambiamos con el mayor hijo
27   }
28   return;
29 }
```

Algoritmo 23 Cálculo de distancia transpuesta

```
1
2  __device__ double distancia_trans(double *p1, int pitch_p1, int col_1, double *q){
3
4     int i=0;
5     double suma=0;
6
7     for (i=0; i < DIM; i++)
8        suma += (((double *)((char *)p1 + (i*pitch_p1)))[col_1] - q[i]) *
9                (((double *)((char *)p1 + (i*pitch_p1)))[col_1] - q[i]);
10
11    return sqrtf(suma);
12 }
```

4.7. Métodos de exportación de resultados

La exportación de los resultados obtenidos se realiza en consideración de los formatos más utilizados comúnmente, como son archivos de texto plano (.txt), formato de documento portable (.pdf), archivos Word (.doc), archivos de hojas de calculo (.xls), la figura 4.7 muestra como es la interfaz gráfica de esta parte del software.

Como se menciono en la sección anterior 4.2, cuando se ejecuta exitosamente un algoritmo kNN se muestra la ventana gráfica de los resultados obtenidos, en la parte central de la figura 4.7 se aprecio el recuadro donde se muestran todos los resultados de los K vecinos más cercanos de cada uno de todos las consultas realizadas.

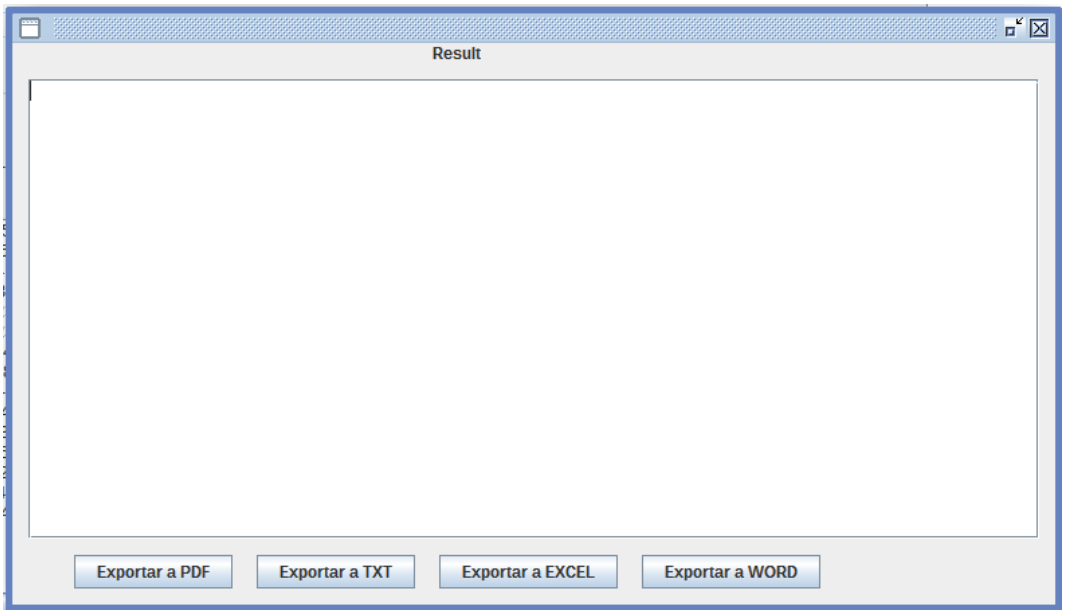


Figura 4.5: Interfaz gráfica de ventana de exportación de resultados

En la figura 4.6 muestra la gráfica de la opción de guardar, cuando se selecciona cualquiera de los tipos de exportación, la gráfica es la misma y es intuitiva de acuerdo al común de las gráficas de exportación de diversos software.

A continuación se muestra extracto de los códigos de exportación de acuerdo con sus

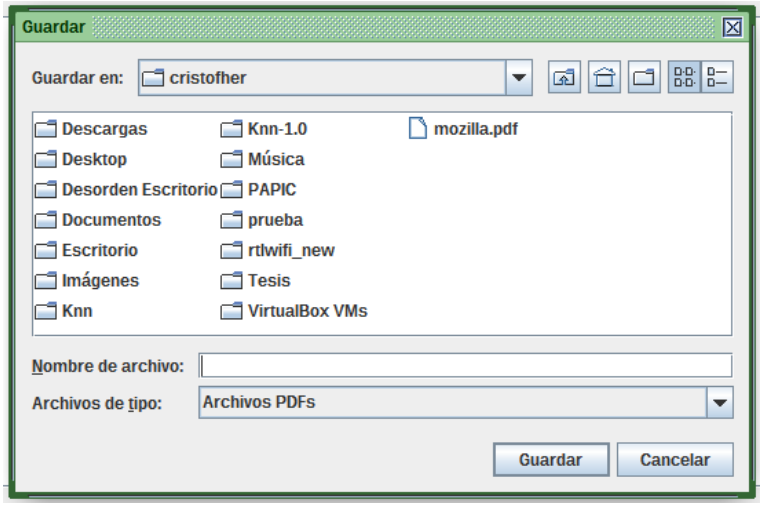


Figura 4.6: Interfaz gráfica de ventana de guardar archivo

formatos, el primer método (Algoritmo 24) corresponde al método de exportación de resultados a Excel en formato *.xls*. Para implementar tanto la exportación a Word y a Excel fue necesario incluir una API llamada *POI* en su *versión 3.16* [fundation, 2017], esta API permite que desde una aplicación desarrollada en *Java* se pueda exportar a diversos formatos como *Word, Hojas de cálculo, Presentaciones, etc* siendo los dos primeros considerados en esta tesis.

El segundo método (Algoritmo 25) es el caso de la exportación a Formato de documento portátil (*PDF*) fue necesario la importación de *iText* en su *versión 5* [**?**], esta biblioteca es de código libre (Open Source) desarrollada por iText Group. Ésta está disponible para *Java* y *C#*. Esta API permite crear y manipular archivos *PDF* - *RTF* - *HTML* en java, siendo el primero considerado para esta tesis.

El tercer método (Algoritmo 26) es el caso de la exportación a archivo de texto plano (*TXT*), para este tipo de documentos no se utilizo librerías externas debido a que *Java* posee métodos para la creación y manipulación de estos archivos.

Los métodos mencionados anteriormente cuentan con una función *obtenerRutaArchivo*, esta permite guardar el archivo con el nombre que le da el usuario y la extensión correspondiente al documento que desea exportar. En caso que el archivo que desea crear ya existe pide la confirmación al usuario si se debe

Algoritmo 24 Método de exportación de resultados a *Excel*.

```
1  public void generarExcel() throws IOException {
2      String rutaArchivo = obtenerRutaArchivo("xls", "Archivos Excel");
3      if (rutaArchivo != null && jTextArea1.getText().length() != 0) {
4          File archivoXLS = new File(rutaArchivo);
5          if (archivoXLS.exists()) {
6              archivoXLS.delete();
7          }
8          archivoXLS.createNewFile();
9          Workbook libro = new HSSFWorkbook();
10         FileOutputStream archivo = new FileOutputStream(archivoXLS);
11         Sheet hoja = (Sheet) libro.createSheet("Resultados Knn");
12         String texto = jTextArea1.getText();
13         String[] lineas = texto.split("\n");
14         for (int i = 0; i < lineas.length; i++) {
15             Row fila = hoja.createRow(i);
16             String[] sublineas = lineas[i].split(" ");
17             for (int j = 0; j < sublineas.length; j++) {
18                 Cell celda = fila.createCell(j);
19                 celda.setCellValue(sublineas[j]);
20             }
21         }
22         libro.write(archivo);
23         archivo.close();
24         }
25 }
```

reemplazar el archivo.

4.8. Nuevo modulo Añadir menú

Para añadir un nuevo método al software además de los que ya cuenta por defecto, el usuario cuenta con un menú de opciones para añadir un método creado por él, pero debe cumplir con ciertos requisitos estipulados de acuerdo con cada uno de los tipos de algoritmo ya sea secuencial o paralelo según la arquitectura.

Cabe mencionar que la realización de este menú, fue realizado a través de una interfaz gráfica en JAVA y su lógica en el lenguaje C, de manera que se dividen las tareas en base al principio divide y vencerás. La interfaz gráfica como se aprecia en la figura 4.7, donde se aprecia en la parte izquierda de la interfaz los 4 tipos de algoritmos contenidos en el software de modo que si desea ingresar un nuevo algoritmo, se debe indicar a cual de ellos pertenece, de manera similar a la carga de bases de datos y consultas se debe

Algoritmo 25 Método de exportación de resultados a *PDF*.

```
public void generarPDF() throws IOException, DocumentException {
        String rutaArchivo = obtenerRutaArchivo("pdf", "Archivos PDFs");
        if (rutaArchivo != null) {
            File archivoPDF = new File(rutaArchivo);
            if (archivoPDF.exists()) {
                archivoPDF.delete();
            }
            archivoPDF.createNewFile();
            FileOutputStream archivo = new FileOutputStream(archivoPDF);
            Document documento = new Document();
            PdfWriter.getInstance(documento, archivo);
            documento.open();
            documento.add(new Paragraph("Resultados Knn \n"));
            String texto = jTextArea1.getText();
            String[] lineas = texto.split("\n");
            for (String linea : lineas) {
                documento.add(new Paragraph(linea));
            }
            documento.close();
            JOptionPane.showMessageDialog(null,
                    "El archivo se a guardado Exitosamente",
                    "Informacion", JOptionPane.INFORMATION_MESSAGE);
        }
    }
```

Algoritmo 26 Método de exportación de resultados a *TXT*.

```
    public void generarTXT() throws IOException {
        String rutaArchivo = obtenerRutaArchivo("txt", "Archivo de texto plano TXT");
        try {
            if (rutaArchivo != null) {
                File archivoTXT = new File(rutaArchivo);
                if (archivoTXT.exists()) {
                    archivoTXT.delete();
                }
                try (FileWriter save = new FileWriter(archivoTXT)) {
                    save.write(jTextArea1.getText());
                }
                JOptionPane.showMessageDialog(null,
                        "El archivo se a guardado Exitosamente",
                        "Informacion", JOptionPane.INFORMATION_MESSAGE);
                Desktop.getDesktop().open(archivoTXT);
            }
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(null,
                    "Su archivo no se ha guardado",
                    "Advertencia", JOptionPane.WARNING_MESSAGE);
        }

    }
```

cargar el archivo fuente, este será visto previamente en el recuadro inferior izquierdo. Además del paso anterior debe especificar el nombre del menú el cual aparecerá en la lista de los menús, para finalizar y lograr añadir correctamente un nuevo método debe presionar el botón *Compile & add Menu* y el recuadro inferior derecho mostrara información detallada respecto del proceso de compilación, de manera que indica si el proceso fue exitoso o no.

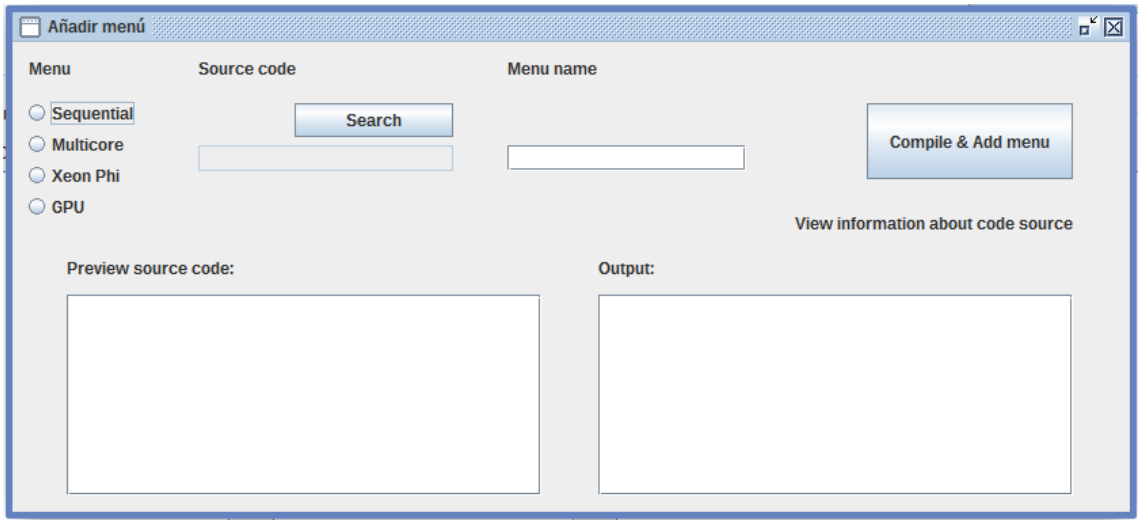


Figura 4.7: Interfaz gráfica de ventana Agregar menú

Capítulo 5

Conclusión

5.1. Conclusiones

En este trabajo se han buscado e implementados algoritmos presentes en la literatura para la realización de consultas kNN en un software diseñado y pensando en usuarios, los cuales no tengan conocimientos previos en programación, éste software cuenta con algoritmos para procesar consultas kNN en tres tipos diferentes de plataformas paralelas: procesador multi-núcleo, coprocesador Xeon Phi y coprocesador GPU.

Los algoritmos seleccionados se basan en la utilización de heap y búsquedas exhaustivas, estos presentan buenos resultados en relación al tiempo, estos fueron implementados exitosamente en un software que junta mundos completamente opuestos, tales como son Xeon Phi (Intel) y GPU (NVIDIA). Cabe destacar que el software no afecta el funcionamiento del algoritmo en términos de tiempo, debido a la capacidad de JAVA de ejecutar la shell (Consola) a través de una rutina simple, de este modo es posible ejecutar los algoritmos programados en C, los cuales son compilados previamente en la instalación y luego solo son llamados para ser ejecutados. Esta idea es la que permite la unión de éstos mundos opuestos dentro de un sólo software.

Este trabajo potencia la idea de juntar arquitecturas totalmente opuestas, si bien no es necesario realizar la interconectar entre ambas, es posible ampliar esta idea y realizar lo visto en esta tesis, donde un usuario puede disponer de diversas soluciones a un problema y no esta ligado a una arquitectura especifica. Si no se cuenta con alguno de éstos dos coprocesadores el usuario no está forzado a adquirir uno de éstos, puede realizar las consultas desde su propio equipo utilizando tanto el modo secuencial como paralelo pero utilizando las características propias del equipo.

5.2. Trabajos futuros

El presente trabajo es la primera versión de éste software, el cual en un futuro puede contar con mejoras, tales como:

- Llevar el software a otro sistema operativo cómo Windows, debido a ser el sistema operativo más utilizado. Además de intentar llevarlo a OSx, de manera que el software sea multi-plataforma y multi-copresadores. De esta manera se generaría un software más potente.

- Intercomunicación entre el paralelismo multi-núcleo con el paralelismo en Xeon phi

  Este trabajo futuro puede ser implementado de variados modos, de manera que se debe investigar y realizar la implementación de esta mejora.

- Intercomunicación entre el paralelismo multi-núcleo con el paralelismo en GPU

  Este trabajo futuro puede ser implementado de variados modos, de manera que se debe investigar y realizar la implementación de esta mejora.

- Añadir nuevos métodos de exportación a otros tipos de archivo que sean requeridos según ciertos criterios.

- Añadir aparte de los 4 métodos principales, nuevos métodos los cuales utilicen otras propuestas para el algoritmo.

- No solo se podrían añadir algoritmos kNN sino también otros asociados a métodos de clasificación, o bien el software puede contener una recopilación de algoritmos en diversas áreas.

Bibliografía

[spe, 2011] (2011). Perl, python, ruby, php, c, c++, lua, tcl, javascript and java comparison.

[cud, 2017] (2017). Cuda: Compute unified device architecture. In *©2007 NVIDIA Corporation*.

[xeo, 2017] (2017). *PRACE (Partnership for Advanced Computing in Europe). Best Practice Guide - Intel Xeon Phi*.

[Acosta et al., 2012] Acosta, F. A., Segura, O. M., and Ospina, A. E. (2012). Guía y fundamentos de la programación en paralelo. *Revista en telecomunicaciones e informática*, 2(4):81–97.

[Aha et al., 1991] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6(1):37–66.

[and/or its affiliates, 2017] and/or its affiliates, O. C. (2017). Netbeans ide fits the pieces together.

[Barrientos et al., 2010] Barrientos, R., Gómez, J., Tenllado, C., and Prieto, M. (2010). Heap based k-nearest neighbor search on gpus. In *Congreso Espanol de Informática (CEDI)*, pages 559–566.

[Bentley and Friedman, 1979] Bentley, J. L. and Friedman, J. H. (1979). Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409.

[Bernabeu, 2013] Bernabeu, S. R. (2013). Max/min heap, priority queues... ¿qué son y cómo lo hago? comparativa python, c/c++.

[Brisaboa et al., 2008] Brisaboa, N., Pedreira, O., Seco, D., Solar, R., and Uribe, R. (2008). Clustering-based similarity search in metric spaces with sparse spatial centers. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 186–197. Springer.

[Brisaboa et al., 2006] Brisaboa, N. R., Farina, A., Pedreira, O., and Reyes, N. (2006). Similarity search using sparse pivots for efficient multimedia information retrieval. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 881–888. IEEE.

[Bustos et al., 2006] Bustos, B., Deussen, O., Hiller, S., and Keim, D. (2006). A graphics hardware accelerated algorithm for nearest neighbor search. In *International Conference on Computational Science*, pages 196–199. Springer.

[Cederman and Tsigas, 2009] Cederman, D. and Tsigas, P. (2009). Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4.

[Chapman et al., 2007] Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.

[Chávez and Navarro, 2000] Chávez, E. and Navarro, G. (2000). An effective clustering algorithm to index high dimensional metric spaces. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 75–86. IEEE.

[Cover and Hart, 1967] Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

[Deepak et al., 2001] Deepak, A., Crupi, J., and Malks, D. (2001). Core j2ee patterns: Best practices and design strategies. *Sun Microsystems. Palo Alto*.

[Freeman, 1969] Freeman, J. (1969). Experiments in discrimination and classification. *Pattern Recognition*, 1(3):207–218.

[fundation, 2017] fundation, A. (2017). Apache poi - the java api for microsoft documents.

[Garcia et al., 2008] Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE.

[Hart, 1966] Hart, P. E. (1966). An asymptotic analysis of the nearest-neighbor decision rule. Technical report, DTIC Document.

[HILBORN, 1968] HILBORN, C. (1968). Dg lainiotis. *IEEE transactions on information theory*.

[IT, 2017] IT, I. (2017). Papi - papic:profiling.

[Joskowicz, 2008] Joskowicz, J. (2008). Reglas y prácticas en extreme programming. *Universidad de Vigo*, page 22.

[Kernighan and Ritchie, 2006] Kernighan, B. W. and Ritchie, D. M. (2006). *The C programming language*.

[Kuang and Zhao, 2009] Kuang, Q. and Zhao, L. (2009). A practical gpu based knn algorithm. In *International symposium on computer science and computational technology (ISCSCT)*, pages 151–155. Citeseer.

[LA, 2017] LA, I. (2017). Procesadores intel® xeon phi™. urlhttps://www.intel.la/content/www/xl/es/products/processors/xeon-phi/xeon-phi-processors.html,.

[McDermid, 1993] McDermid, J. y. P. R. (1993). *"Software Development Process Models", en Software Engineer's Reference Book*. CRC Press. pp. 15/26-15/28.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly.

[Nuñez, 2003] Nuñez, J. M. B. (2003). Investigación de la plataforma j2ee y su aplicacion practica. *Chile: Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación*.

[Oracle, 2017] Oracle (2017). ¿cuáles son los requisitos del sistema para java? urlhttps://www.java.com/es/download/help/sysreq.xml,.

[Paredes and Navarro, 2009] Paredes, R. U. and Navarro, G. (2009). Egnat: A fully dynamic metric access method for secondary memory. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 57–64. IEEE Computer Society.

[Plaku and Kavraki, 2007] Plaku, E. and Kavraki, L. E. (2007). Distributed computation of the knn graph for large high-dimensional point sets. *Journal of parallel and distributed computing*, 67(3):346–359.

[Pressman and Troya, 1988] Pressman, R. S. and Troya, J. M. (1988). Ingeniería del software.

[Reinders, 2007] Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly.

[Wang, 2014] Wang, E. Zhang, Q. S. B. Z. G. L. X. W. Q. W. Y. (2014). *High-Performance Computing on the Intel Xeon Phi(TM): How to Fully Exploit MIC Architectures*. Springer, New York.