UNIVERSIDAD CATÓLICA DEL MAULE

Facultad de Ciencias de la Ingeniería

Escuela de Ingeniería Civil Informática

Profesor Guía

Nombre_profesor_Guía

TÍTULO DE LA TESIS

NOMBRE(S) ALUMNO(S) TESISTA(S)

Tesis para optar al

Título Profesional de Ingeniero Civil Informático

Talca, MES 20XX

UNIVERSIDAD CATÓLICA DEL MAULE

FACULTAD DE xCIENCIAS DE LA INGENIERÍA

ESCUELA DE INGENIERÍA CIVIL INFORMÁTICA

TESIS PARA OPTAR AL

TÍTULO PROFESIONAL DE INGENIERO CIVIL INFORMÁTICO

"TÍTULO DE LA TESIS"

NOMBRE(S) ALUMNO(S) TESISTA(S)

COMISIÓN EXAMINADORA	**FIRMA**
PROFESOR GUÍA NOMBRE_PROFESOR_GUÍA	_____
PROFESOR COMISIÓN NOMBRE_PROFESOR_COMISIÓN	_____
PROFESOR COMISIÓN NOMBRE_PROFESOR_COMISIÓN	_____
NOTA FINAL EXAMEN DE TÍTULO	_____

TALCA, MES DE 20XX

Sumario

resumen trabajo.... (máximo 3 páginas)

A continuación algunos ejemplos de figuras, referencia, cita, tabla y algoritmo:

Una referencia a la Figura 1 y a las subfiguras 2.4(a) y 2.4(b). Una cita a libro de Pthreads [NBF96] y OpenMP ([CJvdP07]).

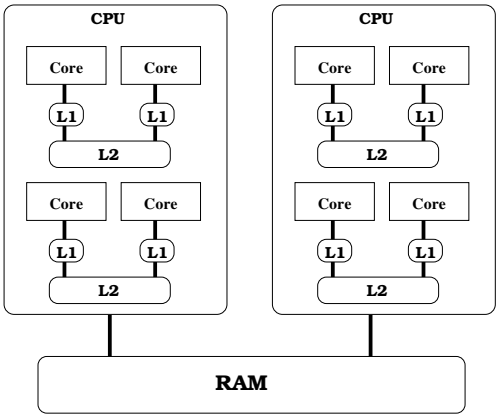

Figura 1: Plataforma multi-core.

```
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) 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) Subfigura 2

Figura 2: Ejemplos para ilustrar la sincronización de los threads de un *warp*.

Algoritmo 1 *EGNAT*: búsqueda por rango r para la consulta q.

busquedarango(Nodo *P*, Consulta *q*, Rango *r*)

1: {Sea R el conjunto resultado}
2: $R \leftarrow \emptyset$
3: $d \leftarrow dist(p_0, q)$
4: **if** $d <= r$ **then**
5: se reporta p_0
6: **end if**
7: $range(p_0, q) \leftarrow [d - r, d + r]$
8: **for all** $x \in P$ **do**
9: **if** $range(p_0, q) \cap range(p_0, D_{p_x}) \neq \emptyset$ **then**
10: se agrega x a R
11: **if** $dist(x, q) <= r$ **then**
12: se reporta x
13: **end if**
14: **end if**
15: **end for**
16: **for all** $p_i \in R$ **do**
17: busquedarango(D_{p_i},q,r)
18: **end for**

Tabla 1: Características Generales

Processor	2xIntel Quad-Xeon (2.66 GHz)
L1 Cache	8x32KB + 8x32KB (inst.+data) 8-way associative, 64byte per line
L2 Unifed Cache	4x4MB (4MB shared per 2 procs) 16-way associative, 64 byte per line
Memory	16GBytes (4x4GB) 667MHz DIMM memory 1333 MHz system bus
Operating System	GNU Debian System Linux kernel 2.6.22-SMP for 64 bits

Índice general

Índice de figuras

Índice de tablas

Índice de Algoritmos

Capítulo 1

Introducción

texto...

1.1. Objetivos

El objetivo general de esta tesis es ....

Los objetivos específicos son los siguientes:

- objetivo-especifico-1
- objetivo-especifico-2
- objetivo-especifico-N

Capítulo 2

Marco Teórico

2.1. Sistemas Multi-core y OpenMP

En este capitulo se describen los conocimientos basicos sobre programación multi-nucleos, programación en coprocesadores tales como progración en Intel Xeon Phi, programación en GPU y trabajos relacionados que dan base a esta tesis.

Actualmente las arquitecturas de los dispositivos personales estan 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 nucleos 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.

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.

Exiten diferentes modelos y estandares que permiten la programación de múltiples hilos, tales como Pthreads [NBF96], TBB [Rei07] u OpenMP [CJvdP07], en particular en esta tesis se selecciono el uso de la biblioteca OpenMP que fue desarrollada en 1997. Además es importante mencionar el hecho de que actualmente los compiladores en esta linea son bastante robustos, como tambien el hecho de que el equipo que gestiona OpenMP esta conformado por diferentes

compañias (AMD, Intel, Sun Mycrosystem y otros) y no siendo gestionada por una sola, esto nos proporciona confianza.

2.2. Coprocesadores

Un Coprocesador es utilizado para desligar funciones de la CPU, en esta linea existen desde coprocesadores matematicos hasta coprocesadores graficos, estos poseen caracteristicas y especificaciones a ciertos tipos de operaciones, por ejemplo un coprocesador matematico solo esta desarrollado para revolver problemas aritmeticos o afines al área esto quiere decir que pueden realizar operaciones matematicas a una velocidad mayor que el procesador principal (CPU), a su vez un coprocesador gráfico esta diseñado en otra área. Estos proporcionan al hardware una acelaración en la operación solo cuando se ejecuta una tárea o software que se haya diseñado para aprovechar el coprocesador, las intrucciones para un coprocesador son diferentes que las intrucciones de una CPU de modo que existe un programa que detecta la existencia de un coprocesador y entonces se ejecutan las intrucciones escritas explicitamente para aquel coprocesador, por tanto existen diferentes lineas de coprocesadores en el mercado pero en esta tesis solo consideramos Intel Xeon Phi y GPU NVIDIA.

2.2.1. Intel Xeon Phi

El coprocesador Intel Xeon Phi [xeo] [Wan14] [LA17] consta de hasta 72 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 intrucciones 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 intrucciones 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 alcanzan una velocidad de hasta 5.5GB/sec.

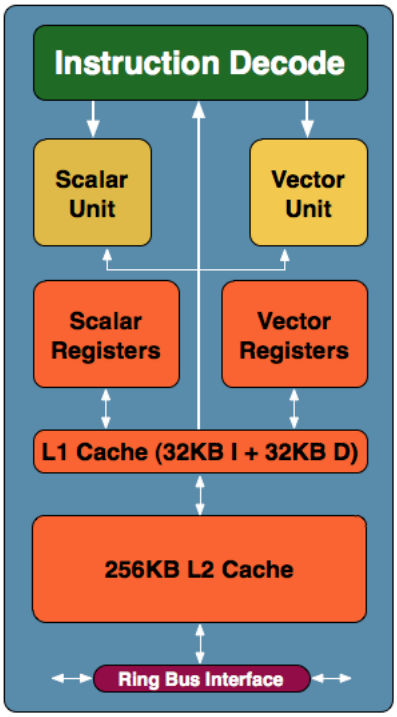

Figura 2.1: Diagrama de una arquitectura MIC de un nucleo de un coprocesador Xeon Phi

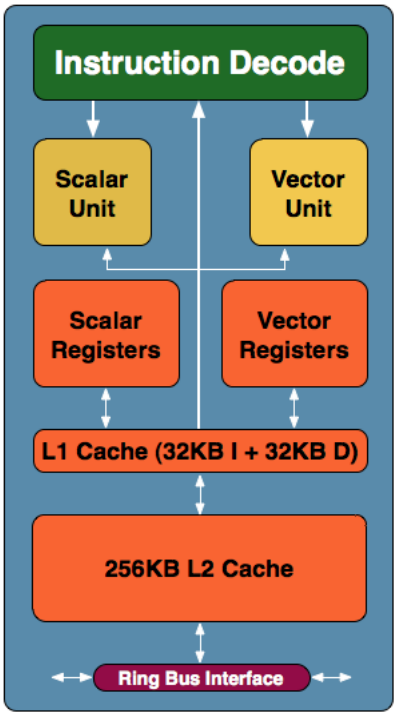

Figura 2.2: Diagrama de una arquitectura MIC de un nucleo de un coprocesador Xeon Phi

2.2.2. NVIDIA GPU

Todas las implementaciones sobre GPU se realizaron usando el modelo de programación CUDA [cud] 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 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 **??** 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 **??** 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*.
- **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 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.

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 dividen 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 2.3 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

```
#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 2.3: 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 2.4: Ejemplos para ilustrar la sincronización de los threads de un *warp*.

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. Un ejemplo de esto lo muestra la figura 2.4, donde en el código 2.4(a) se deben realizar instrucciones de sincronización para garantizar el correcto valor del arreglo result, mientras que el código 2.4(b) es realizado solamente por los threads de un warp, y debido a esto no es necesario utilizar sincronización.

Capítulo 3

Estado del Arte

La regla del vecino más cercano fue originalmente propuesta por Cover y Hart [Har66], [CH67a] 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, en base a este cálculo se analiza la similud entre todos los puntos existentes. En un artículo posterior, Hart [HIL68] 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 "$Iris$" [Fre69].

El algoritmo k-NN [AKA91] [CH67b] 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 [BF79]. Hay una gran cantidad de trabajo sobre las adaptaciones de la estructura básica kd-tree para el problema k-NN [BPS+08] [PN09]. También se han propuesto estructuras no-tree que dividen eficientemente el espacio de búsqueda [CN00] [BFPR06] .Las implementaciones basadas en MPI que utilizan estas estructuras también aparecen con mucha frecuencia en la literatura [PK07].
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, k-NN 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. [BDHK06] 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 [GDB08], [KZ09]. Vecente Garcia et al. [GDB08] propuso un orden de inserción paralelo modificado con el fin de obtener sólo los k elementos más cercanos, mientras que Kuang et al. [KZ09] propuso un mejor Radix-sort para realizar la clasificación final.

Además existe una versión de GPU de Quicksort que fue propuesta por Daniel Cederman y Philippas Tsigas [CT09], 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 [BGTP10]

Capítulo 4

Desarrollo

4.1. Introducción

texto...

Capítulo 5

Experimentos

texto...

Capítulo 6

Conclusiones

texto...

6.1. Trabajos Futuros

- uno...
- dos...
- N...

6.2. Contribuciones de la Tesis

- uno...
- dos...
- N...

Bibliografía

[AKA91] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[BDHK06] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *International Conference on Computational Science*, pages 196–199. Springer, 2006.

[BF79] Jon Louis Bentley and Jerome H Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[BFPR06] Nieves R Brisaboa, Antonio Farina, Oscar Pedreira, and Nora Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 881–888. IEEE, 2006.

[BGTP10] RJ Barrientos, JI Gómez, C Tenllado, and M Prieto. Heap based k-nearest neighbor search on gpus. In *Congreso Espanol de Informática (CEDI)*, pages 559–566, 2010.

[BPS+08] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. 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, 2008.

[CH67a] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[CH67b] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[CN00] Edgar Chávez and Gonzalo Navarro. 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, 2000.

[CT09] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4, 2009.

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

[Fre69] JJ Freeman. Experiments in discrimination and classification. *Pattern Recognition*, 1(3):207–218, 1969.

[GDB08] Vincent Garcia, Eric Debreuve, and Michel Barlaud. 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, 2008.

[Har66] Peter E Hart. An asymptotic analysis of the nearest-neighbor decision rule. Technical report, DTIC Document, 1966.

[HIL68] CG HILBORN. Dg lainiotis. *IEEE transactions on information theory*, 1968.

[KZ09] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. In *International symposium on computer science and computational technology (ISCSCT)*, pages 151–155. Citeseer, 2009.

[LA17] Intel LA. Procesadores intel® xeon phi™. urlhttps://www.intel.la/content/www/xl/es/products/processors/xeon-phi/xeon-phi-processors.html,, jan 2017.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[PK07] Erion Plaku and Lydia E Kavraki. Distributed computation of the knn graph for large high-dimensional point sets. *Journal of parallel and distributed computing*, 67(3):346–359, 2007.

[PN09] Roberto Uribe Paredes and Gonzalo Navarro. 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, 2009.

[Rei07] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[Wan14] Q. Shen B. Zhang G. Lu X. Wu Q. Wang Y. Wang, E. Zhang. *High-Performance Computing on the Intel Xeon Phi(TM): How to Fully Exploit MIC Architectures*. Springer, New York, 2014.

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