Accelerating Exact Similarity Search on CPU-GPU Systems

Takazumi Matsumoto and Man Lung Yiu
Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong
{cstakazumi,csmlyu}@comp.polyu.edu.hk

***Abstract*—In recent years, the use of Graphics Processing Units (GPUs) for data mining tasks has become popular. With modern processors integrating both CPUs and GPUs, it is also important to consider what tasks benefit from GPU processing and which do not, and apply a heterogeneous processing approach to improve the efficiency where applicable.**

Similarity search, also known as k-nearest neighbor search, is a key part of data mining applications and is used also extensively in applications such as multimedia search, where only a small subset of possible results are used. Our contribution is a new exact kNN algorithm with a compressed partial heapsort that outperforms other state-of-the-art exact kNN algorithms by leveraging both the GPU and CPU.

I. Introduction

k-nearest neighbor search, also known as similarity search, involves finding the top k results (e.g., top 10 most similar) to each query. It is a key component of data mining applications such as kNN join [1], [2], which uses two datasets rather than a dataset and a query set. kNN is also a key part of real world applications such as multimedia search and classification of images, music and video [3]. In a server setting, there are also typically many simultaneous queries that need to be satisfied in a timely fashion.

Formally, similarity search can be defined as:

Given a set $\mathcal{D}$ of points in a r-dimensional space and a query point q, find the k points in $\mathcal{D}$ with the smallest distances $dist(q, p)$.

There are two major approaches to similarity search — (i) exact methods, which return the exact set of k-nearest neighbors, and (ii) approximate methods, which may not return the exact set of neighbors.

Several efficient kNN algorithms have been proposed in the literature, designed to reduce the number of high dimensional distance calculations that bottlenecks performance on traditional CPUs. These approaches include exact kNN with indexing [4], [5] as well as approximate kNN methods [6], [7]. However, since an exact kNN process is required at some stage in any kNN operation, even in approximate kNN search [8], there is a lot of interest in optimizing exact kNN.

Exact similarity search can be decomposed into two fundamental operations: distance computation between the data points and the query points, followed by sorting the results to find the smallest distances (i.e., the points that are most similar to the query).

Our proposed algorithm[1] is in the exact kNN family of algorithms and utilizes threshold pruning compression with a heap based partial sort that is faster than competing state-of-the-art kNN algorithms with parallel partial sorting. This is achieved using a heterogeneous computing platform combining a GPU for distance computation and a CPU for sorting.

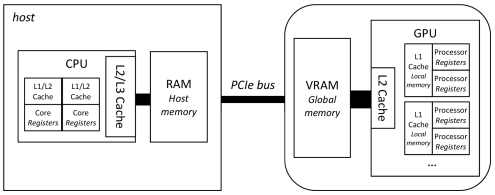


Fig. 1: CPU-GPU system

Fig. 1 shows a typical system containing a CPU and a GPU. There are four key points to such a system:

- the potential computing performance of a GPU (thousands of GFLOPS) is much greater than CPUs (hundreds of GFLOPS), if fully utilized
- the memory bandwidth available to a GPU (hundreds of GB/s) is much greater than that of CPUs (tens of GB/s)
- the host memory connected to a CPU (dozens of GB) is much larger than the video memory connected to a GPU (a few GB)
- the GPU must copy data to host memory into video memory and vice versa over the PCIe interconnect (approx. 32GB/s)

Heterogeneous processing, which uses a combination of processors (in this case a CPU and GPU) on a single task, is an area of growing interest as processors with powerful integrated GPUs become commonplace. These technologies are commonly seen in mobile System-on-a-Chip (SoC) processors that are required to fit in the limited power and thermal envelope of a mobile device (e.g., smartphones), however most mainstream CPUs now integrate GPUs as well.

Distance computation has been established as a highly parallelizable task well suited to GPUs. In our proposed algorithm, distance computation occurs twice: once on a small sample to quickly determine a threshold value for the k-nearest neighbors, and then on the full dataset.

[1]The source code for the implementation used in this paper is available at https://bitbucket.org/cstakazumi/knn-gpu/src

While distance computation is readily parallelized, sorting is also a large cost to consider. Some well known CPU efficient algorithms such as quicksort are challenging for GPUs as diverging code branches reduce parallel efficiency [3], [9]. Existing parallel exact similarity search has focused on using the fastest GPU sorting algorithms, such as radix sort [10], [11].

However, such sorting algorithms do not exploit the result size k to further reduce the sorting cost. The solution to this is a partial sort such as the heapsort, which involves maintaining only the set of nearest k neighbors in memory. A state-of-the-art GPU partial sorting method, based on a parallel bitonic-merge sort, has been proposed in [8]. This method first splits the input dataset recursively until each partition is of length 1, then proceeds to merge them bottom-up in sorted order. Unlike in a full bitonic sort, once a sorted sublist reaches length $2k$, the upper half is discarded.

Bitonic sort is well known as a parallel sorting algorithm as its operations are data parallel (i.e., independent of each other), however partial bitonic sort has a number of issues. Firstly, it scales poorly with k — it is only competitive at small values of k compared to other partial sorting algorithms. Secondly, the sort is not parallel efficient on GPUs — as the sorted sublists are merged, half are discarded, shrinking the number of active threads and not utilizing the GPU computing performance and bandwidth fully.

In our proposed algorithm, rather than interleaving the GPU efficient distance computation with GPU inefficient sorting, we use the GPU for primarily distance computation and the host CPU for sorting. The 'freed' GPU cycles are used for additional distance computation on a small sample of the dataset in order to prune and compress the main dataset to speed up sorting. On a CPU, it is feasible to maintain parallel efficiency and also use an algorithm that can scale with a much larger range of k. Our main contributions in this work can be summarized as follows:

- a fast sampling based pruning method to compress distance matrices
- a parallel implementation of partial minmax heapsort with query batching

In our testing on publicly available image datasets, our proposed TH-heap algorithm outperforms the competing bitonic sort-based kNN algorithm by a factor of 10 or more. Table II lists the symbols that are used in this paper.

II. Related Work

Although the straightforward solution to kNN search is obvious (i.e., the brute force method of measuring all distances and sorting them into order), it is computationally expensive and inefficient with large, high dimensional datasets. In the literature, many approaches were proposed to improve the efficiency of brute force searching, such as VA-file [4] and iDistance [5], which use an index based approach. However, a key issue with index based methods is that their efficiency degrades to brute force with high dimensionality, and that indexes have to be rebuilt if the data changes [3], meaning that with high dimensionality any advantage of index-based methods are lost.

Many data mining applications are good candidates for parallelization [12] as they are typically data parallel tasks. This also makes data mining a suitable problem for GPUs [13], as GPUs are massively parallel processors typically with high bandwidth dedicated RAM. A subroutine, called a *kernel* in GPU programming, can be executed in multiple instances, called *threads*. For GPUs, threads are arranged in blocks also known as *workgroups*, which is assigned to 'Compute Units' as shown in Fig. 2. Since each GPU processor is simple, it shares resources such as cache and scheduling in these units.

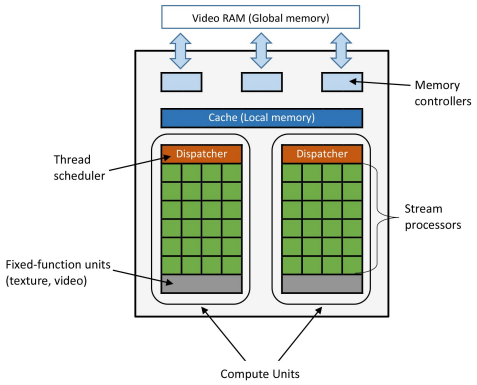


Fig. 2: GPU block diagram

Compared to a multi-core CPU that may run tens of threads simultaneously, a GPU may run hundreds or thousands of threads simultaneously. This is advantageous in data parallel problems where a long loop over the entire dataset can instead of divided up into many threads with shorter loops. Fig. 3 demonstrates a simple operation on six data elements. Fig. 3(a) shows the serial loop given in Algorithm 1 and Fig. 3(b) shows the same task executed in parallel (Algorithm 2). Since each iteration is independent of each other, it can be unrolled completely.

Algorithm 1 Serial loop

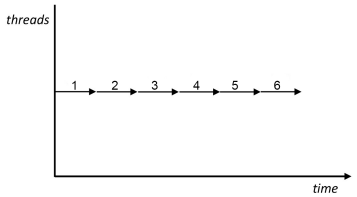
```
for i := 1 to 6 do
    out[i] := i
```

Algorithm 2 Parallel execution

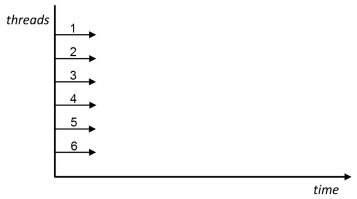
```
parfor each i do
    out[i] := i
end parfor
```

In this paper, the *parfor* keyword to describe the parallel nature of kernels in pseudo-code. A *parfor* block represents a for-loop where all iterations are executed by corresponding threads in parallel. To represent the thread-workgroup hierarchy, a nested *parfor* block is used (see Algorithm 3), with the outer *parfor* indicating workgroup creation and the inner *parfor* indicating thread creation.

In GPU programming, memory has a hierarchy and is logically divided up by thread and workgroup as well. A single thread has access to *private* memory for storing variables, which are typically very high speed registers on a GPU. To share data between threads, there is *local* (a.k.a. shared)



(a) Serial loop



(b) Parallel execution

Fig. 3: Examples of execution on 6 data elements

memory and *global* memory. Local memory is a high speed cached memory that is accessible between threads in the same workgroup. However there is only limited amount of local memory available, usually 32 to 48kB, making it useful for frequently accessed data. Global memory is the slowest and largest amount of memory available, accessible by all threads in all workgroups. Efficient memory access to global memory is key in achieving good parallel performance.

TABLE I: Overview of exact kNN algorithms using GPU acceleration

Algorithm	Sort	Distance	Data structure
1NN [14]	N/A	L_1 norm	r 2D textures
insertion [9]	Partial	L_2, KLD	r-dim. array
BF-reduce [11]	Full	L_2 norm	r-dim. array
BF-radix [11]	Full	L_2 norm	r-dim. array
CU-heap [8]	Partial	L_2 norm	r-dim. array
CU-bitonic [8]	Partial	L_2 norm	r-dim. array

Table I shows several GPU accelerated kNN algorithms, noting their data structures and distance measures. GPU algorithms often tend to focus on exact brute force methods [9], [14], [15]. The primary reason for this is that the large number of simple processors in a GPU are well suited to the large number of simple calculations used in brute force algorithms. On the other hand, algorithms leveraging data structures such as trees and hash tables or complex heuristics limit the parallel potential on GPUs.

Two popular programming frameworks for general purpose computing on GPUs (GPGPU) are (i) CUDA, a proprietary solution developed by NVIDIA [16], and (ii) OpenCL, an open standard managed by The Khronos Group [17] and backed by multiple companies including Intel, AMD, NVIDIA and Apple. The work in [14] predates the introduction of the CUDA framework for GPGPU, and thus uses graphical operations for processing. In [14] data is mapped to a 2D texture, with each texture mapping to a single dimension and several fragment programs (also known as a shader) to calculate distance, eventually reducing the answer to a single nearest neighbor. The authors in [9] implement their brute force kNN algorithm using CUDA, with a parallel insertion sort. An improved version of this brute force algorithm is presented in [18], however in both cases the implementation is restricted in the number of objects it can process due to use of limited shared memory.

The sort algorithm is the key point of difference between these methods. Many GPU algorithms utilize a full sort such as radix sorting. A partial sorting approach has some key advantages however, as only a short list of sorted elements has to be maintained so it can be stored in fast cache.

While parallel heapsort algorithms have been proposed previously [19], [20], a GPU implementation of these algorithms is problematic due to the highly branching nature of the algorithm. The authors in [3] propose partitioning the data and running multiple partial sorts, with the result being gathered from the partitions at the end. In their evaluation, they demonstrate that this heapsort method is as fast or faster than a competing full sort on a GPU where $k < 100$. Another state-of-the-art method discussed earlier is a partial version of the bitonic sort [8]. We compare our proposed algorithm to the work in [8], using the source code provided by the authors.

In contrast to exact methods, approximate distance methods such as Locality Sensitive Hashing (LSH) [6] reduce the k-nearest neighbor problem into an approximate nearest neighbor search with a reduced search space.

The advantage of LSH is that the number of distance calculations necessary to obtain the k nearest neighbors is reduced significantly, as all similar objects should already be in the same hash bucket. The authors in [6] also provide a theoretical bound on the error created by the approximation, something that is not available for heuristic approximation methods such as clustering. There are several implementations of LSH, including a GPU variant in [11].

The primary drawbacks of this LSH implementation are that it is memory intensive, complicated to implement efficiently on GPUs, and depending on the size and distribution of the dataset, the performance can degrade to linear scan in the worst case [21]. On the other hand, exhaustive methods are predictable, have no approximation factor, and are competitive with approximate methods in terms of performance in some circumstances. In this paper, we utilize a fast sampling and threshold compression that maintains exact results while reducing the distance sorting workload significantly.

III. TH-HEAP ALGORITHM

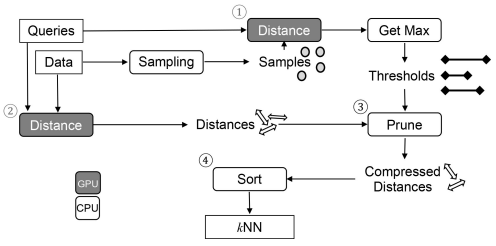


Fig. 4: Flowchart of TH-heap

There are four main stages of TH-heap: sampling, distance computation, pruning and sorting. The overall algorithm for the proposed threshold-heap (TH-heap) algorithm is shown in Fig. 4, with the key operations numbered in order. Distance

TABLE II: Definition of symbols

Notation	Meaning
$\mathcal{D}$	Dataset
$\mathcal{Q}$	Set of queries
$\mathcal{M}$	Set of distances between points in $\mathcal{D}$ and $\mathcal{Q}$
$\mathcal{H}$	Heap used for sorting
$\mathcal{I}$	Set of sorted distances
$\mathcal{S}$	Set of samples from dataset $\mathcal{D}$
$\mathcal{T}$	Set of threshold values
$n_{\mathcal{D}}$	Number of data points in $\mathcal{D}$
$n_{\mathcal{Q}}$	Number of query points in $\mathcal{Q}$
$n_{\mathcal{S}}$	Number of samples
r	Number of dimensions of points in $\mathcal{D}$ and $\mathcal{Q}$
k	Number of nearest neighbors
$size_{qbatch}$	Size of each query batch used in compression

computation is discussed on in Section IV, and while there are two distance operations numbered sequentially, they can be run concurrently. Pruning to compress the distance matrix is discussed in Section V-A and our parallel sorting strategy is discussed in Section V-C.

Initially, $n_{\mathcal{S}}$ samples are chosen at random from the dataset $\mathcal{D}$. On a small sample, initial distance computation with $\mathcal{Q}$ can be done quickly, with the k-th nearest neighbor $d_{\mathcal{T}}$ extracted as the threshold value. This threshold value is used to prune the distance array — objects that have a distance from the query greater than the threshold can be discarded without compromising the k-nearest neighbors in the full dataset. To balance parallelism and compression, queries are arranged into smaller batches of size $size_{qbatch}$.

IV. Background: Parallel Distance Computation

As discussed previously, distance computation using GPUs has been well explored in the literature. In this work we use a standard matrix based approach to compute Euclidean ($L2$) distance between the data points and the query points. This follows from the definition of $L2$ distance:

$$\begin{aligned}\text{distance}_{L2}(q,o) &= \sqrt{\sum^{r}(q_i - o_i)^2} \\ &= \sqrt{\sum^{r}(q_i^2 + o_i^2 - 2q_i o_i)}\end{aligned}$$

As such, it is possible to leverage highly hardware optimized routines for matrix operations and summation. *Reduction* is a common GPU operation that is designed to maximize throughput on mathematical operations such as summation [22]. High performance implementations of the Basic Linear Algebra Subprograms (BLAS) [23] has also been created for GPUs (e.g., cuBLAS[2], clBLAS[3]).

The distance computation algorithm is given in Algorithm 3. Note that as part of the BLAS specification, the general matrix-matrix function (GEMM) takes the following form, where α and β are scalar factors, and $\mathcal{A}$ $\mathcal{B}$ and $\mathcal{C}$ are matrices.

[2]https://developer.nvidia.com/cuBLAS

[3]https://github.com/clMathLibraries/clBLAS

$$\mathcal{C} = \alpha\mathcal{A}\mathcal{B} + \beta\mathcal{C} \tag{1}$$

However in Algorithm 3, α is always left as 1 and β is always left as 0. Equation 1 can then be treated as a simple function with two parameters as in Equation 2.

$$\mathcal{C} = \mathcal{A}\mathcal{B} \tag{2}$$

Algorithm 3 DISTANCES (Dataset $\mathcal{D}$, Queries $\mathcal{Q}$, Distances $\mathcal{M}$)

```
1: Let Dv[1, n_D] be a 1D array (global memory)
2: Let Qv[1, n_Q] be a 1D array (global memory)
3: Let squared[1, r] be a 1D array (local memory)
4: parfor each query q_i do                          ▷ Compute Qv
5:     parfor each dimension r_k do
6:         squared[r_k] := Q[q_i][r_k]^2
7:     end parfor
8:     Reduce squared into sum
9:     Qv[q_i] := sum
10: end parfor
11: parfor each data point o_j do                    ▷ Compute Dv
12:     parfor each dimension r_k do
13:         squared[r_k] := D[o_j][r_k]^2
14:     end parfor
15:     Reduce squared into sum
16:     Dv[o_j] := sum
17: end parfor
18: M := GEMM(D, Q^T)
19: parfor each query q_i do                         ▷ Compute distances M
20:     parfor each data point o_j do
21:         M[q_i][o_j] := SQRT(Dv[o_j] − 2 × M[q_i][o_j] + Qv[q_i])
22:     end parfor
23: end parfor
```

Example 1: Consider a simple example with $n_{\mathcal{D}} = 8, r = 2, n_{\mathcal{Q}} = 2, k = 3, n_{\mathcal{S}} = 4, size_{qbatch} = 2$. Let the dataset $\mathcal{D}$ and query set $\mathcal{Q}$ be defined as follows.

$$\mathcal{D} = \begin{pmatrix} 0.4 & 0.0 \\ 0.7 & 0.1 \\ 1.0 & 0.6 \\ 0.2 & 0.7 \\ 0.8 & 0.5 \\ 0.3 & 0.2 \\ 0.0 & 1.0 \\ 0.9 & 0.5 \end{pmatrix} \quad \mathcal{Q} = \begin{pmatrix} 0.7 & 0.4 \\ 0.1 & 0.5 \end{pmatrix}$$

From $\mathcal{D}$ we take a set of four samples.

$$\mathcal{S} = \begin{pmatrix} 0.4 & 0.0 \\ 0.2 & 0.7 \\ 0.8 & 0.5 \\ 0.3 & 0.2 \end{pmatrix}$$

To compute the distance matrix as in Algorithm 3, we first compute vectors $\mathbf{Dv}'$ and $\mathbf{Qv}$. Note that when using the sample set $\mathcal{S}$ instead of data set $\mathcal{D}$, the vectors are marked as prime ($'$).

$$\mathbf{Dv}' = \begin{pmatrix} 0.16 \\ 0.53 \\ 0.89 \\ 0.13 \end{pmatrix} \quad \mathbf{Qv} = \begin{pmatrix} 0.65 \\ 0.26 \end{pmatrix}$$

$\mathcal{S} \times \mathcal{Q}^T$ is also required for computing $\mathcal{M}$:

$$\mathcal{S} \times \mathcal{Q}^T = \begin{pmatrix} 0.28 & 0.04 \\ 0.42 & 0.37 \\ 0.76 & 0.33 \\ 0.23 & 0.13 \end{pmatrix}$$

Finally, each element in the distance matrix can be determined, for e.g. $d_0^{'0} = \sqrt{0.16 - (2 \times 0.28) + 0.65} = 0.50$

The completed distance matrix is as follows:

$$\mathcal{M}' = \begin{pmatrix} 0.50 & 0.58 \\ 0.58 & 0.22 \\ 0.14 & 0.70 \\ 0.45 & 0.36 \end{pmatrix}$$

After sorting, we can see the k-th largest distance for each query is the threshold value. Sorting is demonstrated in Example 3.

$$\mathcal{T} = \begin{pmatrix} 0.50 & 0.58 \end{pmatrix}$$

V. Parallel Partial Sort

Our work employs a heapsort with a fixed size minmax heap data structure, primarily targeting the multi-core CPU in a CPU-GPU system for sorting. The reason for this is that although written in OpenCL and capable of running on GPUs, partial sorting has not been established to be an optimal problem for current GPU architectures. Traditionally, GPU sort implementations have used full sorts complexity of $\mathbf{O}(n)$ (radix sort [11]) or $\mathbf{O}(n^2)$ (insertion sort [9], [11]). Although the full sort can be parallelized readily, n is the major factor particularly as $k \ll n$. We compare our approach to the state-of-the-art GPU partial bitonic sort, and find that our method can outperform it significantly, even with modest CPU power.

A. Compression

By sampling the dataset and saving the k-th distance for each query, it is possible to discard any distance values above that threshold without compromising the set of exact k nearest neighbors. By changing the number of samples, it is possible to balance the time taken for calculating the threshold vs. how close the threshold is to the actual k-th distance of the whole dataset. The closer the threshold to the actual k-distance, the smaller the dataset being sorted is. We measure the ratio of the threshold value to the actual k-distance as:

$$\text{threshold ratio} = \frac{\sum^{n_\mathcal{Q}} d_k^i / d_\mathcal{T}^i}{n_\mathcal{Q}} \tag{3}$$

In Section VI-A we explore the effect of sample size on performance. Another metric for measuring compression efficiency is compression ratio. While the exact compression ratio will depend on the dataset and query set, it can be estimated as:

$$\frac{n_\mathcal{D}}{\text{expected rank of } k\text{-th best sample}} = \frac{n_\mathcal{D}}{k n_\mathcal{D} / n_\mathcal{S}} = \frac{n_\mathcal{S}}{k}$$

Thus setting the sample size as a multiple of k gives a stable compression ratio. Empirically, even a small sample with negligible sampling time will reduce sort time significantly. Based on this, we find a value between $n_\mathcal{S} = 8k$ and $n_\mathcal{S} = 32k$ to be appropriate.

Since queries are batched for parallelism and to manage buffer sizes, the actual compression ratio for each query batch as shown in Equation 4 is bounded by the maximum size of each row of the distance matrix $\mathcal{M}_q$ after pruning with the threshold value $d_\mathcal{T}^q$.

$$\text{compression ratio} = \frac{n_\mathcal{D}}{\max\{\text{count}(d_i^q < d_\mathcal{T}^q), i = [1, n_\mathcal{D}], \mathcal{M}^q\}} \tag{4}$$

Example 2: Continuing from the previous Example 1, the distance matrix $\mathcal{M}$ is as follows:

$$\mathcal{M} = \begin{pmatrix} 0.50 & 0.58 \\ 0.30 & 0.72 \\ 0.36 & 0.91 \\ 0.58 & 0.22 \\ 0.14 & 0.70 \\ 0.45 & 0.36 \\ 0.92 & 0.51 \\ 0.22 & 0.80 \end{pmatrix}$$

Deleting distance values that are above the threshold gives the following sparse matrix.

$$\mathcal{M}' = \begin{pmatrix} 0.50 & 0.58 \\ 0.30 & \\ 0.36 & \\ & 0.22 \\ 0.14 & \\ 0.45 & 0.36 \\ & 0.51 \\ 0.22 & \end{pmatrix}$$

It is apparent that the first query has a compressed size of 6, while the second query has a compressed size of 4. However, since $size_{qbatch} = 2$ in this example, the larger of the compressed sizes is used. Empty spaces are padded with the max float value. In addition, prior to compressing the sparse matrix the original indices should be stored so they can be referenced correctly after sorting. The final compressed matrix is thus as follows.

$$\mathcal{M}' = \begin{pmatrix} 0.50 & 0.58 \\ 0.30 & 0.22 \\ 0.36 & 0.36 \\ 0.14 & 0.51 \\ 0.45 & \\ 0.22 & \end{pmatrix} \quad \mathcal{M}'_{indices} = \begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 6 \\ 5 & 7 \\ 6 & \\ 8 & \end{pmatrix}$$

By Equation 4, the compression ratio in this example is thus $8/6 = 1.33$.

B. Background: Parallel minmax heap

Initial heap creation time from an initial unsorted array is $\mathbf{O}(k)$. A minmax heap is flexible in that both minimum and maximum elements can be accessed in $\mathbf{O}(1)$ time, similar to a sorted array and faster on average than a standard min or max heap. For maintaining a k nearest neighbor structure, this

means that insertion of new objects onto the heap is on the same order as for a max heap, while generating the kNN list in order is like that of a min heap. Also importantly for a fixed size heap (of size k), insertion has a time complexity of $\mathbf{O}(\log k)$ as opposed to $\mathbf{O}(k)$ (sorted array) or $\mathbf{O}(k + \log k)$ (min or max heap).

There are two key operations for minmax heapsort [24], trickle and bubble. These operations involve moving values either down or up the heap, and varies depending on whether the operation is on a min level or a max level. They are used for updating the heap after every insertion or deletion.

For this implementation there are several changes made from the original algorithm. The heap is a fixed size and starts fully populated, thus the bubble operation is not used as elements are never appended to the bottom of the heap. Instead the heap is initialized from the first k items in the dataset, and subsequent additions always either substitute the root (min) or one of the root node's children (max). Since the heap is not done in-place, it can benefit from locality of reference (CPU) or being stored in local memory (GPU).

Note that to work around some limitations in OpenCL, recursive functions have been re-written to be iterative. In addition, arbitrary loop termination in an inner loop is not well supported, so some loops have been flattened.

Algorithm 4 PTRICKLEMINONCE (Heap $\mathcal{H}$, current root node $root$)

if $root$ has child nodes **then**
  Let $mchild$ be the smaller of $root$'s child nodes
  if $root$ has grandchild nodes **then**
    Let $mgchild$ be the smaller of itself and $root$'s grandchild nodes
if $mchild \leq mgchild$ and $mchild < root$ **then**
  Swap $mchild$ and $root$ in $\mathcal{H}$
else if $mgchild < root$ **then**
  Swap $mingchild$ and $root$ in $\mathcal{H}$
  if $mgchild >$ parent of $mgchild$ **then**
    Swap $mgchild$ and parent of $mgchild$ in $\mathcal{H}$
  return $mgchild$
return none

Algorithm 4 shows the operation trickle min, adapted from [24]. The companion operation trickle max (PTRICKLEMAXONCE) is identical with the inequality signs reversed. The trickle operations maintain the minmax structure by swapping nodes in the heap and is used when initializing the heap, inserting new values and extracting values.

Algorithm 5 PINSERT (Heap $\mathcal{H}$, new node new)

if $heapmax > new$ **then**
  Replace $heapmax$ with new
  if $new < heapmin$ **then**
    Swap new and $heapmin$ in $\mathcal{H}$
  return PTRICKLEMINONCE($\mathcal{H}$, $heapmax$)
else
  return none

Algorithm 5 differs from the original insertion method in two ways. Firstly, as the heap is initially already populated, there are two checks – one against the smallest object in the heap (always the root) one against the largest object in the heap (one of the root's children). Once inserted, the heap must be updated from the point where the new object was inserted.

The operation for heap delete is similar to heap insert, but for delete the root node is overwritten with the last node in the heap, and the heap is then updated again.

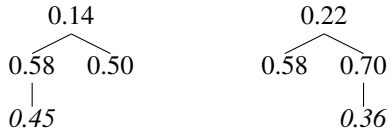
Algorithm 6 PHEAPIFYMINMAX (Heap $\mathcal{H}$)

Let $level$ and $update$ be temporary variables
$level := \lfloor \text{size}(\mathcal{H}) - 2/2 \rfloor$
$update := level$ ▷ Stores the current node being updated
while $level \geq 0$ **do**
  if $update$ is a min level **then**
    $update :=$ PTRICKLEMINONCE($\mathcal{H}$, $update$)
  else
    $update :=$ PTRICKLEMAXONCE($\mathcal{H}$, $update$)
  if $update$ is none **then**
    Decrement $level$
    $update := level$

Algorithm 6 shows the initialization process of the heap, given an initial array. This approach is faster than the original repeated insertion into an empty heap given in [24].

Example 3: Following from Example 2, the two initial heaps are constructed as follows:

Then, the fourth sample is compared to the max value of the heap, which in a minmax heap is always one of the child nodes of the root node. If the new value is smaller than the max value in the heap, the new value replaces it.



The tree is then updated to ensure the root node is still the smallest value and one of its children the largest value.

For computing threshold, the max value can be found immediately, which as shown in Example 2 is 0.50 and 0.58.

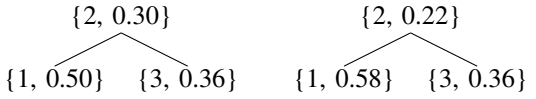
C. Sorting

Algorithm 7 shows the overall sort algorithm. Example 4 illustrates its operation.

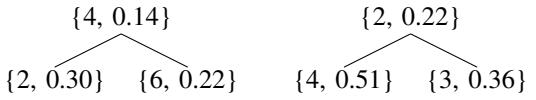
Example 4: In contrast to Example 3, for the sort on the compressed distance matrix both the values and the indices are used, represented as a {key, value} pair. The initial heap is constructed as follows.

Algorithm 7 MINMAXPARTIALHEAPSORT (Heap $\mathcal{H}$, Compressed distances $\mathcal{M}$, Sorted index $\mathcal{I}$, number of neighbors k)

```
Let current and update be temporary variables
parfor each query q ∈ qbatch do
    Copy first k distances into H
    Call PHEAPIFYMINMAX(H)
    for each distance d ∈ M do
        current := k
        update := none
        while current < |M| do
            if update = none then
                update := PINSERT(H, current)
            if update ≠ none then
                update := PTRICKLEMAXONCE(H, current)
            if update = none then
                Increment current
    Call PHEAPIFYMINMAX(Full heap H)
    current := 0
    update := none
    while current < k do
        if update = none then
            Add the root of H to I
            update := PDELETE(H)
        if update ≠ none then
            update := PTRICKLEMINONCE(H, k − current)
        if update = none then
            Increment current
end parfor
```



Following the same procedure as in Example 3, the final heap is:



To retrieve the k-nearest neighbors in order, the root node is deleted and replaced with the largest node. The heap is then updated.

Using the indices in Example 2, we can determine that the k nearest neighbors for each query are, in order: 5, 8, 2 and 4, 6, 7 respectively.

VI. EXPERIMENTAL EVALUATION

This section is broken down into two parts. Section VI-A examines the per-query performance of the threshold based compression with varying parameters. Section VI-B compares the per-query performance of our proposed TH-heap algorithm with competitor methods CU-bitonic and CU-radix [8]. For fair comparison, we obtained and compiled the code for CU-bitonic[4] and CU-heap[4] on for same platform as TH-heap. Since the difference between the algorithms lies in the sort operation, we focus on sort performance.

Table III describes the hardware used in this paper, running Microsoft Windows 8.1 and Visual Studio 2013 Update 4. Note that SP GFLOPS refers to the number of single precision floating point operations per second. The AMD A10-7850K processor has an integrated GPU, however it is not used in these tests as competing methods require the use of the NVIDIA GPU. Also for fair comparison, tasks are not distributed to multiple processors simultaneously.

TABLE III: Platforms used in testing

	CPU	GPU
Name	AMD A10-7850K	NVIDIA GTX 780Ti
Cores/Processors	4	2880
Theoretical performance (SP GFLOPS)	118.4	5040
Threads per workgroup	1024	1024
Simultaneous workgroups	4	15
Physical memory (GB)	16	3
Theoretical bandwidth (GB/s)	30.7	336

Two well known public image datasets, NUS-WIDE [25] and ImageNet [26], along with a sampled ImageNet are used to evaluate performance. The parameters of the datasets are as shown in Table IV. The datasets are normalized to the range $[0.0, 1.0]$, and query points are generated synthetically in that range. Table V gives the range of parameters used in testing, with the default value in **bold**.

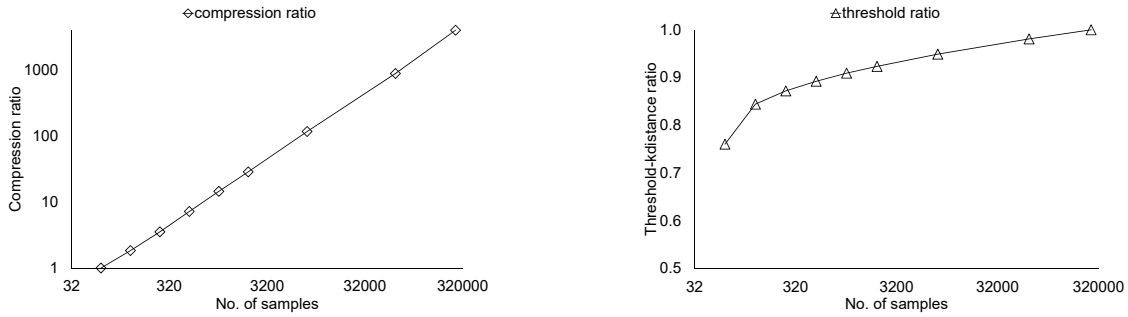
TABLE IV: Parameters of real datasets used

Dataset	$n_{\mathcal{D}}$	r
NUS-WIDE	269648	500
ImageNet	2213937	150
Sampled ImageNet	[250000, 2213937]	150

A. Compression

There are three parameters to sample based threshold compression, with the primary parameter being the number of samples $n_{\mathcal{S}}$. The number of neighbors k and the size of the query batch also affect compression. We measure compression efficiency using two metrics: 1) compression ratio, which is the ratio of the uncompressed distance matrix to the compressed distance matrix (Equation 4) and 2) threshold to k-distance ratio (Equation 3), which measures how close the threshold value was to the actual k-th distance. In addition, the execution times of the sampling and sorting kernels are measured.

[4] http://autogpu.ee.auth.gr/lib/exe/fetch.php?id=cuknns\%3Agpu_accelerated_k-nearest_neighbor_library&cache=cache&media=cuknns:cuknn-toolbox-v2.1.0.tar



(a) Compression ratio

(b) Threshold-k-distance ratio

Fig. 5: $n_{\mathcal{S}}$ affecting compression of NUS-WIDE

TABLE V: Parameter ranges used in testing

Parameter	Description	Range
k	No. of nearest neighbors	[32...**64**...512]
$n_{\mathcal{Q}}$	No. of simultaneous queries	[**512**, 8192]
$n_{\mathcal{S}}$	No. of samples	[64...**512**...8192]
$size_{qbatch}$	Size of each query batch	[1...**4**...16]

Fig. 5(a) shows the number of samples to have a strong impact on the compression ratio. At $n_{\mathcal{S}} = k$ compression is minimal, and at $n_{\mathcal{S}} = n_{\mathcal{D}}$ compression is maximal, as it scans the whole dataset. Threshold-k-distance, as shown in Fig. 5(b), shows the threshold value computed from only k samples to be off from the actual k-distance by approximately 25%.

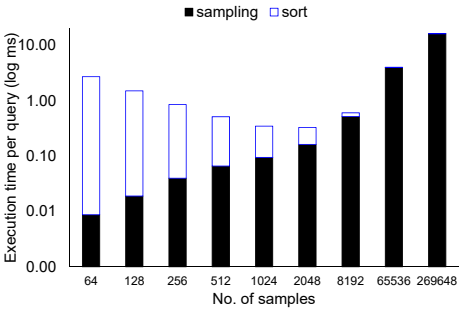


Fig. 6: $n_{\mathcal{S}}$ affecting execution time of sampling and sorting of NUS-WIDE

However it can be observed from Fig. 6 that a relatively small number of samples (less than 1% of $n_{\mathcal{D}}$) can reduce sort time significantly, at minimal sampling cost. Note that for Figs. 5 and 6 the axes are log scaled for clarity.

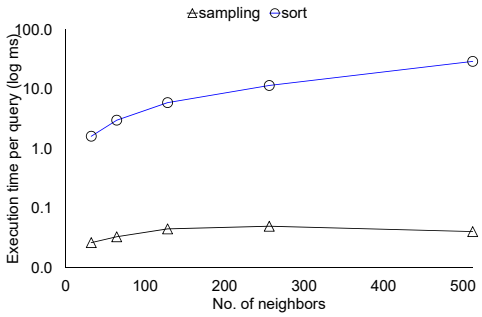


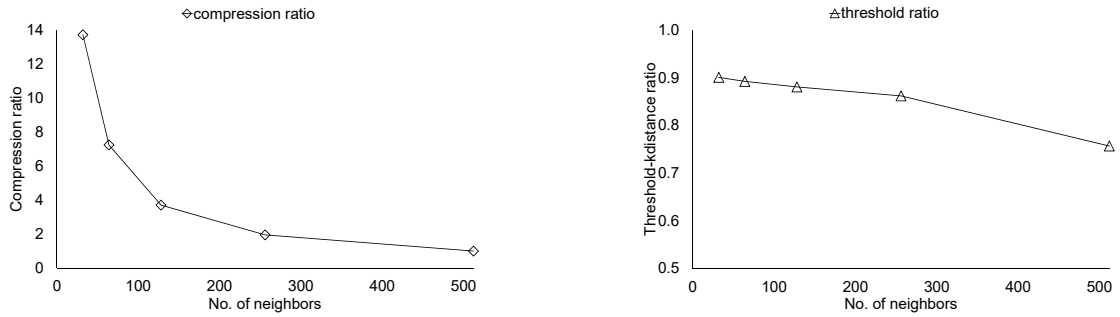
Fig. 8: k affecting execution time of sampling and sorting of ImageNet

Figs. 7 and 8 shows the inverse scenario of Figs. 5 and 6, with a larger k causing a decline in compression efficiency and a decrease in performance. However, since the ratio of k and $n_{\mathcal{S}}$ affects sort time more than sampling time, we can set $n_{\mathcal{S}}$ as a multiple of k to stabilize sort time. For the rest of this work we set $n_{\mathcal{S}} = 8k$.

As described previously, queries are batched when sorting in order to take advantage of parallelism. However, as shown in Fig. 9 since the size of the buffers is bounded by the query with the largest compressed size, batching more queries together reduces the compression ratio somewhat. Note that the threshold-k-distance ratio is not affected by query batching. The performance gain from batching is significant up to the available number of processor cores. Launching more threads than the number of cores available has negligible performance change, so the query batch size should set according to the processor.

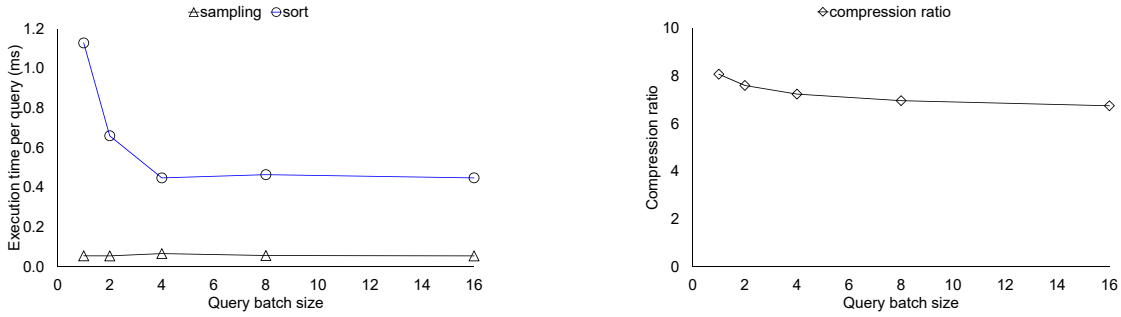
B. Performance comparison

This section compares the proposed TH-heap algorithm, with the state-of-the-art partial bitonic sorting algorithm (CU-bitonic), as well as a competing heapsort CU-heap using the code provided by the authors [8].



(a) Compression ratio (b) Threshold-k-distance ratio

Fig. 7: k affecting compression of ImageNet



(a) Execution time (b) Compression ratio

Fig. 9: $size_{qbatch}$ affecting compression of NUS-WIDE

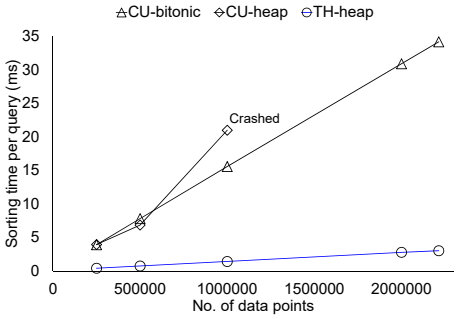


Fig. 10: Varying $n_{\mathcal{D}}$ with CU-bitonic, CU-heap, and TH-heap on sampled ImageNet

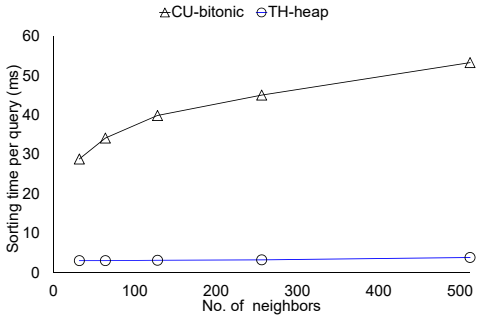


Fig. 11: Varying k with CU-bitonic and TH-heap on ImageNet

Fig. 10 shows the proposed TH-heap algorithm outperforming the competing algorithms by a significant margin. Unfortunately due to programming issues, CU-heap failed to complete most tests. As a result, CU-heap is excluded from the remaining tests.

Compared to the earlier Fig. 7, by using a variable sample size, Fig. 11 shows TH-heap is unaffected by changes in k compared to CU-bitonic.

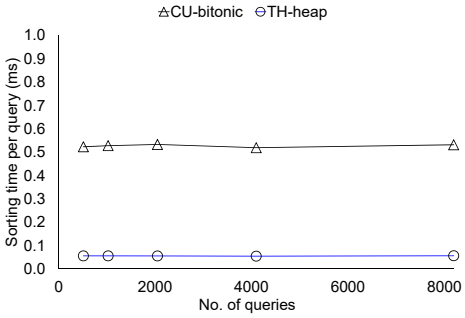


Fig. 12: Varying n_Q with CU-bitonic and TH-heap on NUS-WIDE

Fig. 12 shows that even with a relatively large number of simultaneous queries, TH-heap can maintain a significant performance lead over CU-bitonic, despite the latter running with many more threads on the GPU.

In summary, our proposed TH-heap algorithm offers significantly improved performance over the existing state-of-the-art CU-bitonic algorithm.

VII. Conclusion

In this paper we proposed the TH-heap algorithm for similarity search designed for heterogeneous processing, applying the strengths of both CPU and GPU architectures. We have shown that our proposed TH-heap algorithm offers $10\times$ or more per-query sorting performance for kNN over existing state-of-the-art partial sorting algorithms for GPUs. This is achieved with a minimal overhead threshold and compression approach.

In future work, we would like to leverage the portability of the OpenCL framework to load balance the work across multiple parallel processors, which would allow for more work to be overlapped during transfer and other overhead. This could also be expanded to distributed systems where there are multiple CPU-GPU nodes.

Acknowledgment

This work was supported by grant GRF 152201/14E from the Hong Kong RGC.

References

[1] C. Bohm and F. Krebs, "The k-nearest neighbor join: Turbo charging the KDD process," *Journal Knowledge and Information Systems*, vol. 6, no. 6, pp. 728–749, 2005.

[2] S. C. B. C. O. Wei Lu, Yanyan Shen, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[3] G. Teodoro, E. Valle, N. Mariano, R. Torres, W. M. Jr, and J. H. Saltz, "Approximate similarity search for online multimedia services on distributed CPU–GPU platforms," *The VLDB Journal*, vol. 23, pp. 427–448, 2014.

[4] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. of the 24th VLDB Conference*, 1998, pp. 194–205.

[5] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 364–397, 2005.

[6] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proc. of the 13th ACM STOC*, 1998, pp. 604–613.

[7] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold, "Clustering for approximate similarity search for high-dimensional spaces," *IEEE TKDE*, vol. 14, no. 4, pp. 792–807, 2002.

[8] N. Sismanis, N. Pitsianis, and X. Sun, "Parallel search of k-nearest neighbors with synchronous operations," in *Proc. of the IEEE Conference on High Performance Extreme Computing*, 2012.

[9] V. Garcia and E. Debreuve, "Fast k nearest neighbor search using GPU," in *IEEE CVPR*, 2008, pp. 1–6.

[10] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. of the 2009 IEEE IPDPS*, 2009, pp. 1–10.

[11] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation," in *Proc. of the 19th ACM SIGSPATIAL GIS*, 2011, pp. 211–220.

[12] E. Hung and D. W. Cheung, "Parallel mining of outliers in large database," *DAPD*, vol. 12, no. 1, pp. 5–26, 2002.

[13] M. Alshawabkeh, B. Jang, and D. Kaeli, "Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems," in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 104–110.

[14] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," *Computational Science ICCS*, vol. LNCS 3994, pp. 196–199, 2006.

[15] R. Uribe-Paredes, P. Valero-Lara, E. Arias, J. L. Sanchez, and D. Cazorla, "Similarity search implementations for multi-core and many-core processors," in *Proc. of the International Conference on High Performance Computing and Simulation*, 2011, pp. 656–663.

[16] NVIDIA Corporation. (2011, 07) CUDA. http://www.nvidia.com/object/cuda_home_new.html. Accessed Oct/2012.

[17] Khronos Group. (2011) OpenCL. http://www.khronos.org/opencl/. Accessed Oct/2012.

[18] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching," in *Proc. of the IEEE ICIP*, 2010.

[19] S. K. Prasad and S. I. Sawant, "Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors," in *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995, pp. 328–335.

[20] M. A. A. Alhija, A. Zabian, S. Qawasmeh, and O. H. A. Alhaija, "A heapify based parallel sorting algorithm," *Journal of Computer Science*, vol. 4, no. 11, pp. 897–902, 2008.

[21] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, "DSH: Data sensitive hashing for high-dimensional k-nn search," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2014, pp. 1127–1138.

[22] M. Harris. Optimizing parallel reduction in CUDA. NVIDIA Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[23] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[24] M. D. Atkinson, J. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Communications of the ACM*, vol. 29, no. 10, pp. 996–1000, 1986.

[25] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y.-T. Zheng. NUS-WIDE image dataset. http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm. Accessed Mar/2014.

[26] L. Fei-Fei, K. Li, O. Russakovsky, J. Krause, J. Deng, and A. Berg. ImageNet image database. http://www.image-net.org/. Accessed May/2014.