Fast GPU Based k-NN Algorithm

Ishbir Singh

Computer Science Extended Essay

Candidate Number: 001424-0121
Examination Session: May 2014

School: Pathways World School, Aravali
Supervisor: Ms. Usha Kasana

Word Count: 3967

Abstract

k-nearest neighbour or k-NN is a classical computer science problem in the field of machine learning and pattern recognition with a polynomial time complexity. GPGPU or General Purpose computing on Graphics Processing Units is an upcoming technology that allows massive parallelization because of the presence of hundreds of independant cores. This extended essay explores existing work in this nascent field of artificial intelligence algorithms on graphics card and discusses an algorithm for massively parallelizing k nearest neighbour calculations using CUDA (a technology for GPU computing by NVIDIA) so that data sets with a large number of data points, test points and dimensions (limited by GPU memory) can be efficiently processed. Thus the research question for this extended essay is: how can k nearest neighbours algorithm be optimized to run faster using CUDA GPGPU technology?

The researcher reviewed and analyzed past work in the same field to come up with a simple but efficient algorithm which gave a maximum 74.5x performance increase with a mid-range graphics card (NVIDIA GT 650M) over a serial CPU version of the same algorithm. Testing on two different systems showed that although the algorithm proposed is brute-force, it has tremendous capacity for being parallelized because it was found that with an increase in the number of test and training points, the GPU version was even faster.

In spite of such an increase in performance, it was found that the proposed solution had its limitations, such as excessive consumption of memory. The algorithm could still be optimized further using yet unknown techniques or better mathematical insight to further boost speed and bring down the memory consumption so that even more number of test points, training points and dimensions can be processed using the same amount of memory.

Word Count: 289

Table of Contents

Ishbir Singh
001424-0121

1 Introduction

Research on this topic began with Dr. Suely Oliveira at the University of Iowa, USA[1]. However, since the results were a little inconclusive and the need for improvement in technique and more optimization was felt, the same research was continued and is now presented in the form of an extended essay.

The k-nearest neighbour or k-NN problem is a method for classifying unknown objects based on the closest training samples in feature space. It is a type of supervised learning and is among the simplest of all machine-learning algorithms (Peterson 2009). The objects are assumed to be vectors of numbers x_i, where $i = 1, 2, \ldots, n$ in n-dimensional space. Each number x_i is a feature of the data object. Euclidean distances between a test sample and specified training samples are commonly used distances (Peterson 2009); however, they can be generalized to be a Minkowski metric of the form:

$$\sqrt[p]{\sum_{i=1}^{n}(|x_i - y_i|)^p}$$

where p is a real number (1 for Manhattan or 2 for Euclidean distances) and n refers to the number of features or dimensions of the point (Andoni 2009; Nene and Nayar 1997).

k-NN has various applications in the fields of data processing and analysis. Some other applications include information retrieval, searching image databases (Andoni 2009), medical analysis involving detection of QRS complexes in ECG (electrocardiogram) (Saini, Singh, and Khosla 2013) and pattern recognition of antibody results (Binder et al. 2005).

In the past decade, GPUs have become commonplace and GPGPU technologies such as OpenCL and CUDA facilitate their use for not just graphics processing but also general purpose computing. They have hundreds of cores that can process data simultaneously with high precision and performance that exceed that of the CPUs. The specialized nature of GPUs makes it easier to use additional transistors for computation. Moreover, the multi-billion dollar game market, which brings more graphic intensive games every year, is a key factor driving the innovation behind each generation of GPUs (Harris 2004).

Thus the research question for this extended essay is: how can k nearest neighbours algorithm be optimized to run faster using CUDA GPGPU technology?

2 Background

GPGPU technology is being used for a variety of purposes such as advanced rendering, computational geometry, computer vision and scientific computing. However, algorithms designed

[1] Abstract of the previous research can be found on page 22 of http://www2.education.uiowa.edu/belinblank/students/summer/pdf/sstp_abstracts.pdf

for CPUs cannot be simply 'ported' over to GPUs because of the complexity of the GPU programming model. CUDA is a parallel computing architecture developed by NVIDIA for their graphic cards. The strength of CUDA lies in the fact that it is massively scalable to use all the available resources of the GPUs, provided that the code is written keeping in mind the architecture of a GPU.

In CUDA architecture, the device is the GPU that has many multi-processors each containing multiple stream processors capable of processing one thread at a time. All the multi-processors share the device-wide memory known as global memory. Each global memory transaction is of 128 bytes so if consecutive threads access consecutive chunks of global memory, the reads/writes are clubbed into one. This is known as coalescing. Also shared with all the multi-processors is the read-only constant memory. All the stream processors in a multi-processor share the memory known as shared memory, which is typically in the range of 16 KB - 48 KB. Each stream processor has its own memory known as the register memory. Each multi-processor processes a number of threads and these threads are collectively known as a thread block (*CUDA C Programming Guide* 2013).

3 Previous Work

3.1 Distance Computation

The first stage of solving the k-nearest neighbours problem is calculating the distance between the test points and the query point. High dimensional input data is common for many real-world problems and brute-force k-NN, due to its high running complexity of $O(mnd)$, where m is the number of test points, n is the number of training points and d is the number of dimensions of data, is impractical for running on a CPU. Various techniques have been proposed by researchers to bring down its polynomial time complexity.

Nene and Nayar (1997) proposed a simple algorithm for nearest neighbour search in high dimensions. Their algorithm, however, did not exactly look for k-nearest neighbours but neighbours within a specified distance e. The complexity of their algorithm is $O(ne+n\left(\frac{1-e^d}{1-e}\right))$ and for small e, it grows very slowly with d. It relies on dynamic space partitioning by searching for the points in a hypercube of side $2e$ centered at a query point Q. The closest point is then found by exhaustive search on these candidate points, the cost of which is negligible since the number of query points is typically small. The disadvantage of this approach, however, is that it does not solve the k-nearest neighbour problem but tackles a different, albeit similar problem. Another issue is that an appropriate value of e is hard to determine because the distribution of data may not be known beforehand.

Arya et al. (1998) presented an algorithm for approximate nearest neighbour search which brought the complexity down to $O(dn \log n)$ for pre-processing, $O(c_{d,\varepsilon} \log n)$ for computing the approximate nearest neighbour of a query point q (where $\varepsilon > 0$ and $c_{d,\varepsilon} \leq d[1 + 6d/\varepsilon]^d$)

and $O(kd \log n)$ for computing the approximate k-nearest neighbours. The algorithm relied on hierarchical de-composition of space called a balanced box-decomposition (BDD) tree. The tree was divided such that each small hyper-rectangle (cell) had one associated point. The algorithm locates the test point in the BDD tree in $O(\log n)$ time and enumerates over the cells nearest to it in the increasing order of distance from the test point. As soon as $\text{dist}(p, q)/(1 + \varepsilon)$ is greater than the lowest distance seen so far, the loop terminates, reporting the approximate nearest neighbour. This step is repeated k times to get the k approximate nearest neighbours. The downsides of this were that the algorithm is inefficient when $d > 20$ due to an increase in the average error as well as the running time.

Another way to speed up the computation in spite of having the same time complexity is by utilizing the parallel power of graphics cards. Kuang and Zhao (2009) proposed a practical GPU based KNN algorithm implemented in CUDA that used data segmentation to increase performance compared to ordinary CPU brute-force algorithms. They use a segmentation strategy that splits the matrix containing the results of the calculation into a large number of tiles with width T. Each thread block containing $T \times T$ threads takes charge of a tile in the result matrix. Each thread in the block processes one element of the result set. Test and train matrices are also split into tiles and each thread calculates the partial distances of T points using d/T dimensions. This approach yields the researchers a speed-up of 34.91x from a GPU over a CPU. However, it is important to note that the CPU was a Pentium D processor (quite old) and the graphics card 9600GT (not as old as Pentium D). Moreover, the data segmentation strategy is complex and the researcher believes that such complexity is not required.

Garcia, Debreuve, and Barlaud (2008) presented an algorithm for fast k-NN search using GPUs. Their approach was to process a pair of test and training points on each thread and calculate the distance between them. They use global memory for the test points (coalesced data[2]) and texture memory for the training points (non-coalesced data). However, their data partitioning and work group assignment techniques are not mentioned in the paper.

3.2 Sorting

Sorting the distance calculations is not required in some of the previously mentioned algorithms (Nene and Nayar 1997; Arya et al. 1998) because of the way in which they process points. However, for brute force approach, there exist numerous sorting algorithms.

Radix sort involves doing a stable distribution sort on the digit-places from least significant to most significant, partitioning the keys (positional representation) into r distinct buckets where $r = 2^b$ and b is number of bits in a digit.

Bucket sort is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then filled with the elements of the specified range and those elements are further sorted by using bucket sort recursively or some other sorting algorithm.

[2] Coalesced data refers to data arranged in memory such that coalescing is possible

Insertion sort is a simple sorting algorithm that has an average complexity of $O(n^2)$. However, this complexity is too high if the number of training points is large.

Merge sort is divide-and-conquer sorting algorithm with an average complexity of $O(n \log n)$. It works by splitting the array to be sorted recursively such that each resulting array has only 1 element. Then, it goes back up, merging the lists and placing elements in the desired order.

Cederman and Tsigas (2010) proposed a practical quicksort algorithm for GPUs. Quicksort is an algorithm that works by selecting a random pivot (in randomized quicksort or the first element in the original algorithm proposed by Hoare 1962) from the array of elements to be sorted. It then loops through the array and places all the elements smaller than the pivot to the left of it and the bigger elements to its right. This operation is recursively repeated on the left and right lists until the entire array is sorted. Quicksort has long been considered to be one of the fastest sorting algorithms for single processor systems but it has not been an efficient sorting solution for GPUs. It's actual time complexity is better than radix sort when $n < 2^{32}$. Their parallel implementation runs in 3 distinct phases with thread as well as thread-block synchronization.

Sintorn and Assarsson (2008) developed an algorithm for fast parallel GPU sorting using a hybrid algorithm which relied on multiple sorting strategies to bring the running time for GPU sorting down to $O(n \log n)$ which was previously unheard of for GPU sorting. The first step is to partition into L sub-lists using either quicksort or bucket sort. The second step is to run a merge sort and get the final sorted list.

Merrill and Grimshaw (2011) proposed an algorithm for parallel radix sort. Their implementation of parallel radix sort uses a composite bitwise-parallel scan resulting in a sorting speed of up to a billion keys per second. It is now the standard algorithm distributed with Thrust library bundled with CUDA 4.2 onwards.

Helluy (2011) described a portable implementation of radix sort algorithm on OpenCL designed to run on CPUs as well as GPUs. Its speedup is directly proportional to the number of elements suggesting that it faces an overhead while reading data from host. The researcher has also created a C++ library for public use.

Garcia, Debreuve, and Barlaud (2008) used an insertion sort variant that outputs only the k smallest elements, having considered comb sort. However, the time taken by insertion sort increased linearly with k and it is faster than comb sort only up to around $k = 120$. This is one of the major drawbacks of this approach, since it is not feasible to use their insertion sort variant when k is huge (1001 or 10001).

Kuang and Zhao (2009) used radix sorting proposed by Satish, Harris, and Garland (2009) on the GPU for sorting the distances between test and training points, having considered using GPU based bitonic sort. They write that sorting became the bottleneck in the performance of the whole application, suggesting that further innovations were needed.

However, sorting leads to wastage of clock cycles since the distances for k-NN need not be sorted. Even for weighted k-NN, it would be much more efficient to take the k smallest distances and then sort just those.

3.3 Selection

Instead of sorting, selection algorithms could be used to find the k^{th} smallest distance between the test point and the training points. After that, a simple loop could be run to get all the elements that have a value of less than that of the k^{th} element.

Various selection algorithms exist, one of the simplest being Find by Hoare (1961). Commonly known as quick select, it relies on partitioning the array into lists containing elements that are less than or equal to the pivot and greater than the pivot. The value of k decides whether the element lies in former or the latter list. If the element lies in either of the list, the process is recursively repeated only for the list where the element will be. Quick select has an average running time of $O(n)$.

Akl (1984) described an optimal algorithm for parallel selection. His algorithm assumes the existence of n^{1-x} processors operating in parallel where $0 < x < 1$ and n is the number of elements to be sorted. The array is divided into n^{1-x} sub-arrays of n^x elements and each processor finds the median of its associated sub-array. Then, the median of all the medians is found and used to divide the array into 3 sub-arrays S_1, S_2 and S_3 of elements smaller than, equal to and larger than the median of medians, respectively. This procedure is recursively repeated until $\mid S_1 \mid + \mid S_2 \mid \geq k$. The median of medians is the final answer. The parallel running time of this algorithm is $O(n^x)$.

Bader (2004) presented an improved randomized algorithm for parallel selection on CPU using MPI (Message Passing Interface) that can be used to achieve multi-core, multi-processor or even multi-machine level parallelization. His algorithm works by choosing two random splitters k_1 and k_2 that partition into 3 groups G_0, G_1 and G_2 iteratively such that $\forall x \in G_0, x < k_1$ and $\forall x \in G_1, x \in [k_1, k_2]$ and $\forall x \in G_2, x > k_2$. The aim is to have the middle group G_1 much smaller than the other groups with the condition that it contains the required selection index. Once G_1 is small enough, the rest of the calculations are performed sequentially. However, the downsides of MPI are that there is increased latency with multiple machines and CPUs can only go so fast. GPUs are much faster than CPU in all of the parallel problems discussed here.

Monroe, Wendelberger, and Michalak (2011) described an algorithm for randomized selection on the GPU that works using iterative probabilistic guess-and-check process on pivots for a three-way partition. Their basic algorithm is similar to Bader (2004) but they use different probabilistic calculations and do GPU specific optimizations so because as k increases, the timing goes up.

Alabi et al. (2012) presented fast k-selection algorithms for GPUs. They reviewed and improved upon two k-selection algorithms namely radix select and bucket select. Their implementation of bucket select has lesser mean running time than randomized select proposed by Monroe, Wendelberger, and Michalak (2011).

However, the problem with existing k selection algorithms is that they can only work on one huge array at a time instead of numerous smaller arrays.

3.4 Classification

Once the list of sorted elements has been obtained, the classification of the test point is decided on the basis of maximum number of objects/points of a specific category in the k-nearest neighbours result set. If the need calls for weighted k-NN, the closeness of training point to the test point influences the classification of the test point. The farther points are given lesser preference while the nearer points are given higher preference.

This step is not computation intensive since only the mode of the training points needs to be calculated for getting a classification match.

4 Proposed Algorithm

4.1 Distance Calculation

The brute-force approach to distance calculation was chosen because it is the most parallelizable on the GPU architecture due to a high number of small, yet independant calculations between each pair of test and training points which can be handled by each thread. A work-group is assigned one test point and the threads within that work-group calculate the distances between that test and all the training points. The data containing information about the test and the training points is stored in the global memory of the GPU. The CUDA kernel[3] code is as follows:

```
__global__ void distances_computation(float* test_g, float* train_g, float *output, int dims) { // dims is dimensions of data
        /**
        * test_g: Array of test points in global memory
        * train_g: Array of training points in global memory
        * output: Array of output distance calculations in global memory
        * dims: Number of dimensions in the incoming data
        */

        float res = 0; // Stores the final result
        int global_id_0 = blockIdx.x * blockDim.x + threadIdx.x; // ID of test point
        int global_id_1 = blockIdx.y * blockDim.y + threadIdx.y; // ID of training point
        int global_size_0 = gridDim.x * blockDim.x; // Number of test points
        int global_size_1 = gridDim.y * blockDim.y; // Number of training points

        extern __shared__ float test[];
        if (threadIdx.y < dims) { // first 'dims' threads copy each dimension float to local memory
```

[3]A CUDA kernel is a function designed to run on the GPU

```
17                 test[threadIdx.y] = test_g[dims*global_id_0 + threadIdx.y];
18         }
19         __syncthreads(); // wait for copy operation
20
21         for (int i=0; i < dims; i++) { // loop over!
22                 res += pow((train_g[global_size_1*i+global_id_1] - test[i]), 2); // find the right train point to use
23         }
24
25         int id = global_id_0*global_size_1 + global_id_1; // ID of test point*Number of training points + Training point ID
26
27         // Thus, the corresponding distances between one test point and all training points are stored in a contiguous location
28         // This approach is very useful for segmented sorting.
29
30         output[id] = res;
31 }
```

The kernel begins with getting the data of the test point being handled by the work group into the shared memory (lines 15-19). Multiple shared memory accesses to the same data are broadcast to all the threads in a warp, eliminating any memory conflicts. For global memory requests to be coalesced, the format of the array containing the training points was:

$$[x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n]$$

where x, y, z are 3 training points and n refers to nth dimension of the data. Consecutive threads would thus access consecutive blocks in memory, bringing down access time and number of memory requests. The test points are stored in the form of:

$$[x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n, \ldots]$$

Each thread copies one float x_i from the global memory to the shared memory, where i is the thread ID and x is the test point assigned to the workgroup (line 17). Thus one of the constraints is that the number of dimensions (d) has to be less than or equal to the thread per work group count.

The number of test and training points is only limited by the memory and the upper limit of workgroups which is 65535×65535 in each dimension (x and y). Thus, if each thread block processes 256 training points, there are 65535 testing points possible and around 256 possible training points. But if the number of test points is reduced, the number of training points can be increased. This can be expressed in the form of a ratio:

$$\text{Number of test points} \times \text{Number of training points} = k$$

where k is constant. However, it is important to note that this limitation only exists in graphics cards with Compute capability 2.x or below. On the newer models, the limit of workgroups has been extended to $2^{32} - 1$ in x dimension but remains the same in y (*CUDA C Programming Guide* 2013).

The storage of the points also happens such that memory coalescing is possible (lines 25-30). Each consecutive thread stores points in consecutive memory locations. The format of this array is:

$$[x1_{y1}, x1_{y2}, x1_{y3}, \ldots, x1_{yn}, x2_{y1}, x2_{y2}, \ldots]$$

where $xn_{yn'}$ refers to Euclidean distance between the nth test point and n'th training point. This format also makes it easier for the next stage (sorting/selection) to easily happen since distances from one test point and all training points are stored together, thus preventing random memory accesses.

4.2 Selection/Sorting Algorithm

Since the sorting phase has often been called a bottleneck in kNN calculations (Kuang and Zhao 2009), the researcher intended to find a parallel k-selection algorithm which could work on segmented data (since k-selection needs to be run on many relatively small arrays) in parallel. However, such an algorithm either has not been made or is not publicly available. Due to lack of requisite skills for making such an algorithm, a pre-made segmented sorting algorithm named Segmented Sort Pairs which forms a part of ModernGPU library by NVIDIA Research (Baxter 2013) was thus used.

This sorting algorithm is a high performance variant of merge sort which operates on non-uniform random data. It sorts pair-wise by the distance between each test and training point, preserving the index of the training point after sorting. Although this solves the problem, it results in double the memory consumption than required (the distances array and the indices array which are stored separately).

4.3 Classification

For the purposes of this paper, the last stage, which is classification, was not implemented becuase it involves simple matching against an array of pairs of start and end indices to find out which category the top training points belong to. Once this is known, finding the mode of the cateogry will give us the category of the testing point thus completing the k-NN algorithm. This step by its very nature is not compute intensive at all and can easily be done on the CPU (Kuang and Zhao 2009).

5 Performance and Evaluation

For testing the speed of the algorithm on the GPU, the timer function available with ModernGPU was used. The *chrono* header file from C++ standard library was used for measuring times of the functions intended to run on CPUs. The releases were compiled for x64 target architecture on Release configuration using Visual Studio 2012 with all optimizations for speed. Two test systems were used. The first system ran on Windows 8 64-bit with 6GB DDR3 RAM, Intel Core i7 2670QM CPU at 2.20GHz and NVIDIA GT 520MX GPU (1GB DDR3 memory). The second system ran Windows 7 64-bit and had 8GB DDR3 RAM, Intel Core i7 3720QM CPU at 2.6GHz and NVIDIA GT 650M GPU (1GB GDDR5 memory). All the tests were repeated 5 times to eliminate any random variations in the times. The dimensional data came from a psuedorandom number generator *random.random* from the Python standard library and had a range of $[0, 1]$. This data was saved into files using a Python helper script (Appendix B.1) and then read and processed by the CUDA program (Appendix B.3) .

The Raw Time Measurements appendix contains detailed data on each of the 9 test cases which were used. Each case used different number of training and test points and varying dimensions. Maximum speed up of 74.5x was obtained using a mid-range graphics card (system 2) and around 43.8x using a low-range graphics card (system 1). It was also observed that as the size and dimensionality of the data set increased, the speedup factor also increased implying that a GPU is better at handling more data than the CPU. However, moving from case 8 to 9, a drop in performance was seen in both the systems. This is probably because of increasing the number of threads from 256 to 512 implying that the former number of simultaneous threads gives better performance although it limits the number of dimensions. This is a necessary trade-off. Moreover, from case 7 to 8, the CPU performance sharply dipped in both systems. The reason for this could not be found by the researcher and needs more thorough investigation not central to this paper.

The researcher could not use higher number of test and training points due to memory constraints (1GB memory in the tested cards). The main consumer of memory was the large array containing the distances between each test and training point and another array containing the corresponding indices of the training points so that a paired sort can be done. Without a paired sort, the indices of the training points would get lost and it would not be possible to know which points have the smallest Euclidean distances. Moreover, the researcher had initially tried for even more test points which technically should have fit into the GPU memory. However, the sort operation was repeatedly failing. On investigation of the source, it was found that the GPU sort is not an in-place sort but creates two new temporary arrays for key-value pairs, thus consuming twice the memory. But a fast in-place sort on the GPU is out of the question because each memory transaction is of 128 bytes (*CUDA C Programming Guide* 2013) and in-place sorting would result in 32 times slower sorting (each float is of 4 bytes). An alternative technique which does not need to store the array of indices would allow 2 times more points to be processed.

Moreover, it is important to note the CPU algorithm is a very crude algorithm which could definitely be sped up using threads, OpenMP (a parallelization library) or hand-optimization using assembly language instructions sets like MMX and SSE. But since the comparison was essentially between a single core and a many core system, there was no need for such extreme optimizations as modern compilers usually optimize code well.

6 Conclusion

This paper explored what the k-nearest neighbours algorithm is and how it can be parallelized on a GPU so that the massively parallel data processing capabilities of this device can be fully exploited for artificial intelligence tasks. Sorting did not prove to be a bottleneck as the timing charts in Appendix A a show. Nevertheless, the increase in sorting time was not linear with the increase in number of elements to be sorted and the speedup of GPU over CPU was lesser.

The distance algorithm could still be optimized further using yet unknown techniques or better mathematical insight. This could include finding a way such that no sorting is necessary, for example the use of n dimensional trees (as done by Arya et al. 1998, but decreasing the error in higher dimensions). This would also greatly reduce memory consumption since it was found that sorting is the procedure consuming the maximum memory. However, if such a technique is difficult to implement on a GPU, the existing algorithm could be modified such that multiple test and training points could be processed in the same workgroup as done by Kuang and Zhao (2009). Although this would increase the complexity and might not have any effect on the speed (merely a speculation), more test and training points could be processed because of the work group size limitations (which were not reached in any of the test cases). But note that this still requires more graphics memory.

Even if the sorting operation has to be done, it could certainly be optimized to use less memory. This could be done by changing the sorting algorithm so that instead of returning the values in the ascending order, it returns only the indexes to those values. Or it would be even better if a segmented k-selection algorithm for the GPU could be made. Unfortunately, this could not be done because of the limited knowledge of the researcher.

References

Akl, Selim G. (1984). "An optimal algorithm for parallel selection". In: *Information Processing Letters* 19.1, pp. 47–50. ISSN: 0020-0190. DOI: 10.1016/0020-0190(84)90128-5. URL: http://www.sciencedirect.com/science/article/pii/0020019084901285.

Alabi, Tolu et al. (Oct. 2012). "Fast k-selection algorithms for graphics processing units". In: *J. Exp. Algorithmics* 17, 4.2:4.1–4.2:4.29. ISSN: 1084-6654. DOI: 10.1145/2133803.2345676. URL: http://doi.acm.org/10.1145/2133803.2345676.

Andoni, Alexandr (2009). "Nearest Neighbor Search: the Old, the New, and the Impossible". PhD thesis. Massachusetts Institute of Technology.

Arya, Sunil et al. (Nov. 1998). "An optimal algorithm for approximate nearest neighbor searching fixed dimensions". In: *J. ACM* 45.6, pp. 891–923. ISSN: 0004-5411. DOI: 10.1145/293347.293348. URL: http://doi.acm.org/10.1145/293347.293348.

Bader, David A. (2004). "An improved, randomized algorithm for parallel selection with an experimental study". In: *Journal of Parallel and Distributed Computing* 64.9, pp. 1051–1059. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2004.06.010. URL: http://www.sciencedirect.com/science/article/pii/S0743731504001169.

Baxter, Sean (2013). *Modern GPU*. NVIDIA Research. URL: http://nvlabs.github.io/moderngpu/ (visited on 12/09/2013).

Binder, Steven R. et al. (2005). "Computer-Assisted Pattern Recognition of Autoantibody Results". In: *Clinical and Diagnostic Laboratory Immunology* 12.12, pp. 1353–1357. DOI: 10.1128/CDLI.12.12.1353-1357.2005. eprint: http://cvi.asm.org/content/12/12/1353.full.pdf+html. URL: http://cvi.asm.org/content/12/12/1353.abstract.

Cederman, Daniel and Philippas Tsigas (Jan. 2010). "GPU-Quicksort: A practical Quicksort algorithm for graphics processors". In: *J. Exp. Algorithmics* 14, 4:1.4–4:1.24. ISSN: 1084-6654. DOI: 10.1145/1498698.1564500. URL: http://doi.acm.org/10.1145/1498698.1564500.

CUDA C Programming Guide (2013). NVIDIA. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 12/12/2013).

Garcia, V., E. Debreuve, and M. Barlaud (2008). "Fast k nearest neighbor search using GPU". In: *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pp. 1–6. DOI: 10.1109/CVPRW.2008.4563100.

Harris, Mark (2004). "GPGPU: General-purpose computation on GPUs". In: *Tutorial Course 5, EuroGraphics04*.

Helluy, P (2011). "A portable implementation of the radix sort algorithm in OpenCL".

Hoare, C. A. R. (July 1961). "Algorithm 64: Quicksort". In: *Commun. ACM* 4.7, pp. 321–. ISSN: 0001-0782. DOI: 10.1145/366622.366644. URL: http://doi.acm.org/10.1145/366622.366644.

Hoare, C. A. R. (1962). "Quicksort". In: *The Computer Journal* 5.1, pp. 10–16. DOI: 10.1093/comjnl/5.1.10. eprint: http://comjnl.oxfordjournals.org/content/5/1/10.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/5/1/10.abstract.

Kuang, Quansheng and Lei Zhao (2009). "A practical GPU based kNN algorithm". In: *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pp. 151–155.

Merrill, Duane and Andrew Grimshaw (2011). "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing". In: *Parallel Processing Letters* 21.02, pp. 245–272. DOI: 10.1142/S0129626411000187. eprint: http://www.worldscientific.com/doi/pdf/10.1142/S0129626411000187. URL: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000187.

Monroe, Laura, Joanne Wendelberger, and Sarah Michalak (2011). "Randomized selection on the GPU". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, pp. 89–98. ISBN: 978-1-4503-0896-0. DOI: 10.1145/2018323.2018338. URL: http://doi.acm.org/10.1145/2018323.2018338.

Nene, S.A. and S.K. Nayar (1997). "A simple algorithm for nearest neighbor search in high dimensions". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19.9, pp. 989–1003. ISSN: 0162-8828. DOI: 10.1109/34.615448.

Peterson, L. E. (2009). "K-nearest neighbor". In: *Scholarpedia* 4.2, p. 1883.

Saini, Indu, Dilbag Singh, and Arun Khosla (2013). "QRS detection using K-Nearest Neighbor algorithm (KNN) and evaluation on standard ECG databases". In: *Journal of Advanced Research* 4.4, pp. 331–344. ISSN: 2090-1232. DOI: 10.1016/j.jare.2012.05.007. URL: http://www.sciencedirect.com/science/article/pii/S209012321200046X.

Satish, Nadathur, Mark Harris, and Michael Garland (2009). "Designing efficient sorting algorithms for manycore GPUs". In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IPDPS '09. Washington, DC, USA: IEEE Computer Society, pp. 1–10. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161005. URL: http://dx.doi.org/10.1109/IPDPS.2009.5161005.

Sintorn, Erik and Ulf Assarsson (2008). "Fast parallel GPU-sorting using a hybrid algorithm". In: *Journal of Parallel and Distributed Computing* 68.10, pp. 1381–1388. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2008.05.012. URL: http://www.sciencedirect.com/science/article/pii/S0743731508001196.

A Raw Time Measurements

The code (in Appendix B.3) was compiled in Visual Studio 2012 on the Release x64 environment. All debugging code output was disabled and compiler optimizations were done for ensuring maximum speed. The tests were repeated 5 times and the average was taken. The average total time of CPU was divided by the average total time of GPU to get the speedup in number of times.

A.1 Test System 1

Specifications: Windows 8 64-bit, 6GB DDR3 RAM, Intel Core i7 2670QM CPU at 2.20GHz and NVIDIA GT 520MX GPU (1GB DDR3 memory)

A.1.1 Case 1

$$\text{Test Count} = 1024$$
$$\text{Train Count} = 512$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.00626672	0.00827923	0.014546	0.094052	0.028017	0.122069
2	0.00625661	0.00827194	0.0145285	0.094048	0.028017	0.122065
3	0.00627974	0.00829027	0.01457	0.090037	0.029017	0.119054
4	0.00625533	0.00815242	0.0144077	0.090055	0.027016	0.117071
5	0.00625786	0.00833962	0.0145975	0.090055	0.026016	0.116071
Average	0.00626325	0.00826670	0.014530	0.091649	0.027617	0.119266

$$\text{Speedup} = \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 8.21\text{x}$$

A.1.2 Case 2

$$\text{Test Count} = 2048$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0251949	0.0310295	0.0562244	0.372232	0.115071	0.487303
2	0.0251612	0.0310594	0.0562206	0.368228	0.119074	0.487302
3	0.0251622	0.0309711	0.0561333	0.378235	0.119072	0.497307
4	0.025184	0.0310445	0.0562284	0.379234	0.119073	0.498307
5	0.0251689	0.0310748	0.0562436	0.375232	0.11407	0.489302
Average	0.0251742	0.0310359	0.0562101	0.374632	0.117272	0.491904

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 8.75\text{x}$$

A.1.3 Case 3

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0502904	0.0604823	0.110773	0.78147	0.24315	1.02462
2	0.0502541	0.0601704	0.110425	0.751466	0.238146	0.989612
3	0.0502917	0.0604044	0.110696	0.738463	0.231137	0.9696
4	0.0502729	0.0604098	0.110683	0.771475	0.242149	1.01362
5	0.0502715	0.0603323	0.110604	0.746464	0.232143	0.978607
Average	0.0502761	0.0603598	0.110636	0.75787	0.23735	0.99521

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 9.00\text{x}$$

A.1.4 Case 4

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 2048$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.101081	0.129163	0.230244	1.52294	0.522324	2.04527
2	0.101136	0.12958	0.230716	1.76308	0.523323	2.28641
3	0.101089	0.129287	0.230376	1.62	0.527326	2.14733
4	0.101202	0.12955	0.230752	1.56797	0.529328	2.0973
5	0.101121	0.129454	0.230575	1.66603	0.709439	2.37547
Average	0.101126	0.129407	0.230533	1.62800	0.562348	2.19036

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 9.50\text{x}$$

A.1.5 Case 5

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 128$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.102825	0.0607072	0.163532	2.27942	0.312194	2.59161
2	0.107014	0.0608962	0.16791	2.19737	0.278171	2.47554
3	0.111446	0.060802	0.172248	1.70406	0.239147	1.94321
4	0.111488	0.0606537	0.172142	2.18435	0.309191	2.49355
5	0.111927	0.0606709	0.172598	2.20437	0.308192	2.51256
Average	0.108940	0.0607460	0.169686	2.11391	0.289379	2.40329

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 14.16\text{x}$$

A.1.6 Case 6

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.212993	0.0610609	0.274054	4.5188	0.317196	4.836
2	0.212703	0.0606853	0.273388	4.36671	0.308191	4.6749
3	0.213086	0.0605786	0.273664	4.32268	0.309192	4.63187
4	0.212915	0.0605678	0.273482	3.8734	0.245153	4.11856
5	0.212874	0.0606156	0.27349	4.23363	0.308191	4.54182
Average	0.212914	0.0607016	0.273616	4.2630	0.297585	4.561

Speedup= $\dfrac{\text{Total CPU Time}}{\text{Total GPU Time}} = 16.67\text{x}$

A.1.7 Case 7

$$\text{Test Count} = 8192$$
$$\text{Train Count} = 2048$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.823771	0.259007	1.08278	18.9157	1.04465	19.9604
2	0.82391	0.260843	1.08475	18.3557	1.36689	19.7226
3	0.812734	0.259508	1.07224	17.3313	1.36689	18.6982
4	0.823145	0.257485	1.08063	19.8189	1.08471	20.9036
5	0.823148	0.258584	1.08173	20.1802	1.03568	21.2158
Average	0.821342	0.259085	1.08043	18.9204	1.17976	20.1001

Speedup= $\dfrac{\text{Total CPU Time}}{\text{Total GPU Time}} = 18.60\text{x}$

A.1.8 Case 8

$$\text{Test Count} = 8192$$
$$\text{Train Count} = 4096$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	1.67809	0.57073	2.24882	96.9833	3.01096	99.9943
2	1.67641	0.567334	2.24375	94.6523	2.97396	97.6263
3	1.67695	0.570159	2.24711	99.3715	3.01199	102.383
4	1.67822	0.565061	2.24328	92.8634	3.012	95.8754
5	1.68864	0.565648	2.25429	93.2767	3.01206	96.2887
Average	1.67966	0.56779	2.24745	95.4294	3.00419	98.4335

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 43.80\text{x}$$

A.1.9 Case 9

$$\text{Test Count} = 16384$$
$$\text{Train Count} = 3072$$
$$\text{Dimensions} = 512$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	8.31662	0.814474	9.13109	116.994	4.35898	121.353
2	8.31672	0.83142	9.14814	135.515	5.36961	140.885
3	8.31632	0.829826	9.14614	141.605	5.2645	146.869
4	8.31666	0.817691	9.13435	132.293	4.34389	136.637
5	8.31628	0.811922	9.1282	115.323	4.36291	119.686
Average	8.31652	0.821067	9.13758	128.346	4.73998	133.086

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 14.56\text{x}$$

A.2 Test System 2

Specifications: Windows 7 64-bit, 8GB DDR3 RAM, Intel Core i7 3720QM CPU at 2.6GHz and NVIDIA GT 650M GPU (1GB GDDR5 memory)

A.2.1 Case 1

$$\text{Test Count} = 1024$$
$$\text{Train Count} = 512$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.00273517	0.00457142	0.00730659	0.078	0.0156	0.0936
2	0.00274637	0.00368224	0.00642861	0.0624	0.0156	0.078
3	0.00274426	0.00341846	0.00616272	0.078	0.0156	0.0936
4	0.00274323	0.00353757	0.0062808	0.078	0.0312	0.1092
5	0.00273843	0.00358922	0.00632765	0.078	0.0156	0.0936
Average	0.00274149	0.00375978	0.00650127	0.075	0.0187	0.0936

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 14.40\text{x}$$

A.2.2 Case 2

$$\text{Test Count} = 2048$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0107162	0.0115028	0.022219	0.312	0.0936	0.4056
2	0.0107217	0.0124233	0.0231451	0.312	0.0936	0.4056
3	0.0107263	0.012358	0.0230843	0.3276	0.0936	0.4212
4	0.0107382	0.0114077	0.022146	0.2964	0.0936	0.39
5	0.0107178	0.0124285	0.0231463	0.2964	0.0936	0.39
Average	0.0107240	0.0120241	0.022748	0.309	0.0936	0.4025

$$\text{Speedup}=\frac{\text{Total CPU Time}}{\text{Total GPU Time}}=17.69\text{x}$$

A.2.3 Case 3

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0213395	0.0220546	0.0433941	0.608401	0.1872	0.795601
2	0.0216219	0.0221313	0.0437532	0.608401	0.2028	0.811201
3	0.0213215	0.0217304	0.0430519	0.639601	0.1872	0.826801
4	0.0213533	0.0217957	0.0431491	0.608401	0.1872	0.795601
5	0.0213565	0.0220736	0.0434301	0.624001	0.1872	0.811201
Average	0.0213985	0.0219571	0.0433557	0.617761	0.1903	0.808081

$$\text{Speedup}=\frac{\text{Total CPU Time}}{\text{Total GPU Time}}=18.64\text{x}$$

A.2.4 Case 4

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 2048$$
$$\text{Dimensions} = 64$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0428435	0.0476087	0.0904522	1.1856	0.4368	1.6224
2	0.0428354	0.0480581	0.0908935	1.1856	0.4212	1.6068
3	0.0425371	0.046056	0.0885931	1.2012	0.4212	1.6224
4	0.0425585	0.0473332	0.0898918	1.1856	0.4212	1.6068
5	0.0425606	0.0469811	0.0895417	1.2012	0.4212	1.6224
Average	0.0426670	0.0472074	0.0898745	1.1918	0.4243	1.6162

$$\text{Speedup}=\frac{\text{Total CPU Time}}{\text{Total GPU Time}}=17.98\text{x}$$

A.2.5 Case 5

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 128$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.04279	0.022589	0.0653789	1.3416	0.1872	1.5288
2	0.0427892	0.0216988	0.0644879	1.404	0.1872	1.5912
3	0.0430812	0.022398	0.0654792	1.3416	0.1872	1.5288
4	0.0427858	0.0225653	0.0653511	1.3884	0.1872	1.5756
5	0.0430575	0.0223927	0.0654502	1.3728	0.1872	1.56
Average	0.04290	0.022329	0.0652295	1.3697	0.1872	1.5569

$$\text{Speedup=} \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 23.87\text{x}$$

A.2.6 Case 6

$$\text{Test Count} = 4096$$
$$\text{Train Count} = 1024$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.0855503	0.0226527	0.108203	2.6364	0.1872	2.8236
2	0.0855485	0.0225285	0.108077	2.7768	0.2028	2.9796
3	0.0762698	0.0196984	0.0959682	2.652	0.1872	2.8392
4	0.0762714	0.0202253	0.0964968	2.6364	0.1872	2.8236
5	0.0762684	0.0198918	0.0961602	2.7612	0.1872	2.9484
Average	0.0799817	0.0209993	0.100981	2.6926	0.1903	2.8829

$$\text{Speedup=} \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 28.55\text{x}$$

A.2.7 Case 7

$$\text{Test Count} = 8192$$
$$\text{Train Count} = 2048$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.304428	0.0828186	0.387246	10.7172	0.842401	11.5596
2	0.323697	0.0817387	0.405435	10.7796	0.842401	11.622
3	0.303893	0.0817805	0.385673	10.7328	0.842401	11.5752
4	0.322288	0.0827449	0.405033	10.6392	0.842401	11.4816
5	0.303631	0.0827345	0.386366	10.6236	0.826801	11.4504
Average	0.311587	0.0823634	0.393951	10.6985	0.839281	11.5378

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 29.29\text{x}$$

A.2.8 Case 8

$$\text{Test Count} = 8192$$
$$\text{Train Count} = 4096$$
$$\text{Dimensions} = 256$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	0.631164	0.178724	0.809887	61.1521	1.8564	63.0085
2	0.632803	0.17719	0.809993	56.4721	1.8564	58.3285
3	0.632591	0.177067	0.809658	59.4361	1.8408	61.2769
4	0.623135	0.178502	0.801637	57.5017	1.8564	59.3581
5	0.635041	0.176679	0.811719	57.2833	1.8564	59.1397
Average	0.630947	0.177632	0.808579	58.3691	1.8533	60.2223

$$\text{Speedup}= \frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 74.48\text{x}$$

A.2.9 Case 9

$$\text{Test Count} = 16384$$
$$\text{Train Count} = 3072$$
$$\text{Dimensions} = 512$$

	GPU			CPU		
Trial no.	Distance	Sorting	Total Time	Distance	Sorting	Total Time
1	1.85124	0.259527	2.11077	86.1278	2.6832	88.811
2	1.84638	0.256011	2.10239	83.3665	2.6832	86.0498
3	1.85291	0.256721	2.10963	80.9641	2.6832	83.6473
4	1.86197	0.256333	2.1183	82.8829	2.6832	85.5661
5	1.85899	0.255952	2.11495	84.9421	2.6832	87.6254
Average	1.85430	0.256909	2.11121	83.6567	2.6832	86.340

$$\text{Speedup}=\frac{\text{Total CPU Time}}{\text{Total GPU Time}} = 40.90\text{x}$$

B Reproduced Code

B.1 dataGenerator.py

This is a helper file intended for generating random data in the required format and putting it into files.

```python
import random

def generate_data(test_points, train_points, train_cats, dimensions, testfile, trainfile):
    f = open(trainfile, "w")
    for a in range(train_cats):
        for b in range(train_points):
            line = []
            for c in range(dimensions):
                line.append("%.3f" % random.random())
            f.write(",".join(line) + "\n")
        if a != train_cats - 1: # last point in a category, leave a line
            f.write("\n")
    f.seek(f.tell()-2) # used to eliminate the last \n
    f.truncate()
    f.close()

    f = open(testfile, "w")
    for a in range(test_points):
        line = []
        for c in range(dimensions):
            line.append("%.3f" % random.random())
        f.write(",".join(line) + "\n")

    f.seek(f.tell()-2) # used to elimininate the last \n
    f.truncate()
    f.close()

testCases = [
    [1024, 128, 4, 64, "case1test.txt", "case1train.txt"],
    [2048, 256, 4, 64, "case2test.txt", "case2train.txt"],
    [4096, 256, 4, 64, "case3test.txt", "case3train.txt"],
    [4096, 512, 4, 64, "case4test.txt", "case4train.txt"],
    [4096, 256, 4, 128, "case5test.txt", "case5train.txt"],
    [4096, 256, 4, 256, "case6test.txt", "case6train.txt"],
    [8192, 512, 4, 256, "case7test.txt", "case7train.txt"],
    [8192, 1024, 4, 256, "case8test.txt", "case8train.txt"],
    [16384, 768, 4, 512, "case9test.txt", "case9train.txt"]
]

for case in testCases:
    generate_data(*case)

print("Done")
```

B.2 Definitions.cuh

This file contains a small function helpful in reading and converting data from text files.

```
//
//  Definitions.cuh
//
//  Created by Ishbir Singh on 24/07/12.
//  Copyright (c) 2012-2014 webmaster@ishbir.com. All rights reserved.
//

#ifndef _Definitions_h
#define _Definitions_h

#include <sstream>
struct bad_conversion { };
/*
 * Convert from string to any type via streaming operations.
 */
template <class T>
void from_string(T& t,
                 const std::string& s,
                 std::ios_base& (*f)(std::ios_base&))
{
    std::istringstream iss(s);
    if((iss >> f >> t).fail())
        throw bad_conversion();
}
#endif
```

B.3 CUDA-KNN.cu

This is the main file constituting the program. It contains the data reading and processing code as well as all the algorithms and timers.

```
//
//  CUDA-KNN.cu
//
//  Created by Ishbir Singh on 22/06/12.
//  Copyright (c) 2012-2014 webmaster@ishbir.com. All rights reserved.
//

#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <iterator>
#include <algorithm>
#include <vector>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <chrono>

#include "Definitions.cuh"
```

```
#include "kernels/segmentedsort.cuh"

using namespace mgpu;
using namespace std;

__global__ void populate_keys_with_training_ids(int* keys) {
        /**
        * Populates a keys array with numbers from 0 to train_count
        */
        int global_id_0 = blockIdx.x * blockDim.x + threadIdx.x; // ID of test point
        int global_id_1 = blockIdx.y * blockDim.y + threadIdx.y; // ID of training point
        int global_size_0 = gridDim.x * blockDim.x; // Number of test points
        int global_size_1 = gridDim.y * blockDim.y; // Number of training points

        int id = global_id_0*global_size_1 + global_id_1;
        keys[id] = global_id_1;
}

__global__ void populate_segments(int* segments, int train_count) {
        /**
        * Populates a segments array with indices of where each new test point begins (segments in sorting)
        */
        int global_id_0 = blockIdx.x * blockDim.x + threadIdx.x; // ID of test point
        segments[global_id_0] = global_id_0*train_count;
}

__global__ void distances_computation(float* test_g, float* train_g, float *output, int dims) { // dims is dimensions of data
        /**
        * test_g: Array of test points in global memory
        * train_g: Array of training points in global memory
        * output: Array of output distance calculations in global memory
        * dims: Number of dimensions in the incoming data
        */

        float res = 0; // Stores the final result
        int global_id_0 = blockIdx.x * blockDim.x + threadIdx.x; // ID of test point
        int global_id_1 = blockIdx.y * blockDim.y + threadIdx.y; // ID of training point
        int global_size_0 = gridDim.x * blockDim.x; // Number of test points
        int global_size_1 = gridDim.y * blockDim.y; // Number of training points

        extern __shared__ float test[];
        if (threadIdx.y < dims) { // first 'dims' threads copy each dimension float to local memory
                test[threadIdx.y] = test_g[dims*global_id_0 + threadIdx.y];
        }
        __syncthreads(); // wait for copy operation

```

```
        for (int i=0; i < dims; i++) { // loop over!
                res += pow((train_g[global_size_1*i+global_id_1] - test[i]), 2); // find the right train point to use
        }

        int id = global_id_0*global_size_1 + global_id_1; // ID of test point*Number of training points + Training point ID

        // Thus, the corresponding distances between one test point and all training points are stored in a contiguous location
        // This approach is very useful for segmented sorting.

        output[id] = res;
}

///
/// This is the entry point of the application
/// Accepted filetypes: CSV
///
int main(int argc, char* argv[]) {
        if (argc != 4) // File name is not present or malformed arguments,
        {
                cerr << "Invalid arguments. Arguments are: test_data_file training_data_file num_types_train" << endl;
                return 1;
        }

        /**
        Read the test data from specified file

        Test Data Format:
        _________________


        x1,x2,x3,x4
        x1,x2,x3,x4
        x1,x2,x3,x4
        x1,x2,x3,x4
        y1,y2,y3,y4
        y1,y2,y3,y4
        y1,y2,y3,y4
        z1,z2,z3,z4
        z1,z2,z3,z4
        **/
        int test_count, col_count;

        ifstream infile(argv[1]); // Open file
        test_count = count(istreambuf_iterator<char>(infile), istreambuf_iterator<char>(), '\n')+1; // count number of points
        infile.seekg(0); // seek back

        string line; // temp var for one line
        getline(infile, line); // read one line

        stringstream firstStream(line, stringstream::in | stringstream::out); // make a stream
        col_count = count(istreambuf_iterator<char>(firstStream), istreambuf_iterator<char>(), ',')+1; // count number of dimensions [2 commas mean 3 dims]
```

```

        float* test_points = new float[test_count*col_count];

        for(unsigned int i=0; i < test_count/* && !infile.eof() && infile.good()*/; i++) { // Keep reading and storing
                stringstream lineStream(line, stringstream::in | stringstream::out); // make a stream

                string cell;
                float val;

                for(unsigned int j=0; j < col_count; j++) { // we don't want to store the last thing which gives class
                        getline(lineStream, cell, ',');
                        if (cell == "") // empty
                                continue;
                        from_string<float>(val, cell, std::dec);
                        test_points[i*col_count+j] = val; // Convert to float and store
                }
                getline(infile, line); // read one line
        }

        // Cleanup
        infile.close(); // We are done

        /** Done with reading test data; Start reading training data **/
        /**
        Train Data Format:
        _________________


        x1,x2,x3,x4
        x1,x2,x3,x4
        x1,x2,x3,x4
        x1,x2,x3,x4

        y1,y2,y3,y4
        y1,y2,y3,y4
        y1,y2,y3,y4

        z1,z2,z3,z4
        z1,z2,z3,z4

        This tells us that the test data has 3 classifications.

        Train Data Array Format:
        ___________________________________


        [x1,x2,x3,x4, ... y1,y2,y3,y4, ... z1,z2,z3,z4]

        Train Data Classification Vector Format: [k is index]
        __________________________________________________


        ( index: [classifcation_start_index, classification_end_index] )

        k0: [0, 19],
        k1: [20, 39],
        k2: [40, 59]
```

```

    k0, k1 and k2 give the various classes of the test data to compare to.

    Both these arrays have been kept separate to reduce complexity of code and to maintain a 2 x 2 matrix in both test and train set as specified in paper.
    **/

    ifstream trainfile(argv[2]);

    int train_count, train_line_count;

    // Count the lines and the types
    train_line_count = count(istreambuf_iterator<char>(trainfile), istreambuf_iterator<char>(), '\n')+1; // count last line also
    trainfile.seekg(0);
    trainfile.clear(); // clear EOF bit

    int num_types; // num of types of training points
    from_string<int>(num_types, (string)argv[3], std::dec);

    train_count = train_line_count-num_types+1; // +1 is because of the fact that if there are 2 types, there will be only 1 "\n\n"

    float* train_points = new float[train_count*col_count];

    vector<vector<int>> train_points_classes(num_types, vector<int>(2, 0)); // Init vector for keeping track of classes

    unsigned int type_count = 0; // Keep track of the type id

    train_points_classes[0][0] = 0; // Starting point is 0

    for (int i=0; i < train_count /*&& !trainfile.eof() && trainfile.good()*/; i++) { // Keep reading and storing
        string line;
        getline(trainfile, line);

        if (line == "" && type_count < num_types) { // classification boundary
            train_points_classes[type_count][1] = i-1;

            if (type_count == num_types-1) // last type so set its ending beforehand
                train_points_classes[type_count][1] = train_count*col_count - 1; // common sense
            else
                train_points_classes[type_count][0] = i; // set beginning of next type
            type_count += 1; // increment
            i--; // compensation necessary
        }
        else {
            stringstream lineStream(line, stringstream::in | stringstream::out); // make a stream

```

```
			string cell;
			float val;

			for(int j=0; j < col_count; j++) {
				getline(lineStream, cell, ',');
				if (cell == "") // empty
					continue;
				from_string<float>(val, cell, std::dec);
				train_points[j*train_count + i] = val; // Convert to float and store
			}
		}
	}
	trainfile.close();

	/** Done reading all the data **/
	cout << "Test Count: " << test_count<< "\n";
	cout << "Train Count: " << train_count << "\n";
	cout << "Dimensions: " << col_count << "\n\n";

	// Main stuff comes here
	ContextPtr context = CreateCudaDevice(0);

	int work_items_per_group = col_count > 256 ? 512 : 256; // Max dimensions 512 for our experiments, but more efficiency at 256
	int k = 5; // get k smallest element

	// allocate memory and copy data
	MGPU_MEM(float) devPtrTest = context->Malloc<float>(test_points, test_count*col_count);
	MGPU_MEM(float) devPtrTrain = context->Malloc<float>(train_points, train_count*col_count);
	MGPU_MEM(float) devPtrOutput = context->Malloc<float>(train_count*test_count);

	// create two dimensional blocks
	dim3 block_size;
	block_size.x = 1;
	block_size.y = work_items_per_group;

	// configure a two dimensional grid as well
	dim3 grid_size;
	grid_size.x = test_count / block_size.x;
	grid_size.y = train_count / block_size.y;

	int temp_mem = sizeof(float) * col_count; // allocate enough for one training point

	double GPUDistanceTime = 0;

	context->Start();
	distances_computation <<< grid_size, block_size, temp_mem >>>(devPtrTest->get(),
		devPtrTrain->get(),
		devPtrOutput->get(),
		col_count
		);
	GPUDistanceTime = context->Split();
```

```

        MGPU_SYNC_CHECK("distances_computation");

        cout << "GPU Distance Computation Time: " << GPUDistanceTime << "\n";

        // don't need these 2 anymore
        devPtrTest.release();
        devPtrTrain.release();
        // Keep the test and train points so that CPU processing can also happen

        // STAGE 2 BEGIN
        double GPUSortTime = 0;

        // Allocate memory for sorting stage
        MGPU_MEM(int) keys = context->Malloc<int>(train_count*test_count);
        MGPU_MEM(int) segments = context->Malloc<int>(test_count);

        // fill in keys here
        context->Start();
        populate_keys_with_training_ids<<< grid_size, block_size >>>(keys->get());
        GPUSortTime = context->Split();

        MGPU_SYNC_CHECK("populate_keys_with_training_ids");

        // fill in segments here

        // create two dimensional blocks
        block_size.x = work_items_per_group;
        block_size.y = 1;

        // configure a two dimensional grid as well
        grid_size.x = test_count / block_size.x;
        grid_size.y = 1;

        context->Start();
        populate_segments<<< block_size, grid_size >>>(segments->get(), train_count);
        GPUSortTime += context->Split();

        context->Start();
        SegSortPairsFromIndices<float, int>(devPtrOutput->get(), keys->get(), train_count*test_count,
                segments->get(), test_count, *context);
        GPUSortTime += context->Split();

        cout << "GPU Sorting Time: " << GPUSortTime << "\n";

        cout << "GPU Time Taken: " << GPUSortTime+GPUDistanceTime << "\n\n";

        int *kSmallestIndicesGPU = new int[test_count*k];

        int offset = 0;

        for(int i=0; i < test_count; i++) {
```

```
			keys->ToHost(offset, sizeof(int)*k, &kSmallestIndicesGPU[k*i]);
			offset += sizeof(int)*train_count; // increment to next test point
	}

	// Release data on GPU
	keys.release();
	segments.release();
	devPtrOutput.release();
	// don't need these 2 anymore
	devPtrTest.release();
	devPtrTrain.release();
	// Stage 3 NOT IMPLEMENTED

	// CPU Processing BEGIN

	// Allocate memory for outputpu
	vector<pair<float, int>> CPUOutput(test_count*train_count); // create vector so that it can be easily sorted later

	double CPUDistanceTime = 0;

	auto start_time = chrono::steady_clock::now();

	// Stage 1
	for (int i=0; i < test_count; i++) {
		for (int j=0; j < train_count; j++) {
			float res = 0;
			for (int p=0; p < col_count; p++) {
				res += pow(test_points[i*col_count+p]-train_points[p*train_count+j], 2);
			}

			CPUOutput[i*train_count+j] = pair<float, int>(res, j);
		}
	}

	CPUDistanceTime = chrono::duration_cast<chrono::microseconds>(chrono::steady_clock::now() - start_time).count() / 1000000.0;
	cout << "CPU Distance Computation Time: " << CPUDistanceTime << "\n";

	// Stage 2
	// Sort each list
	double CPUSortTime = 0;
	start_time = chrono::steady_clock::now();

	for (int i=0; i < test_count; i++) {
		sort(CPUOutput.begin() + i*train_count, CPUOutput.begin() + (i+1)*train_count);
	}
	CPUSortTime = chrono::duration_cast<chrono::microseconds>(chrono::steady_clock::now() - start_time).count() / 1000000.0;
	cout << "CPU Sorting Time: " << CPUSortTime << "\n";
	cout << "CPU Time Taken: " << CPUDistanceTime+CPUSortTime << "\n";

```

```
        // Create a separate small array for the smallest indices
        int* kSmallestIndicesCPU = new int[k*test_count];
        offset = 0;

        start_time = chrono::steady_clock::now();

        for(int i=0; i < test_count; i++) {
                for (int j=0; j < k; j++) {
                        kSmallestIndicesCPU[k*i+j] = CPUOutput[i*train_count+j].second;
                }
        }

        // Stage 3 NOT IMPLEMENTED

        // Clear up
        free(test_points);
        free(train_points);

        int rtrn = 0;

        // Selection successful, now check answers against GPU
        for (int i=0; i < test_count*k; i++) {
                if (kSmallestIndicesCPU[i] != kSmallestIndicesGPU[i]) {
                        cerr << "ERROR, mismatch at: " << i << "\n";
                        rtrn = 1;
                }
        }

        return rtrn; // exit code
}
```