

Machine Learning: KNN Algorithm with GPU

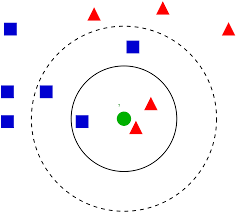
Quincy Wu

University of Washington

Winter EE 590 A

3/14/17

# Executive Summary

This project is designed to implement the k nearest neighbor algorithm using OpenCL kernel. OpenCL is a low level, cross platform environment to execute program with CPU or GPU. This project focus on the GPU, which is powerful for computing highly parallelable code.

KNN algorithm is one of the most widely used machine learning algorithm. Utilizing the parallel component of the KNN algorithm, this project aimed to speed up the computation time via parallelizable part comparing to sequential code.

In this project, KNN were implement using Bitonic sorting algorithm. The Bitonic sort is created for parallel computing. And it successfully speeds up sorting computation time of KNN algorithm. The result discuss in more detail in this report. The speed up using parallel computing resulted in a speed up of 3 to 20 times comparing to sequential code.

Table of Contents

[Executive Summary 1](#_Toc477552861)

[OpenCL 3](#_Toc477552862)

[KNN Algorithm 3](#_Toc477552863)

[Weighting Scheme 4](#_Toc477552864)

[Sorting (Bitonic) 4](#_Toc477552865)

[Complexity and Estimated speedup 5](#_Toc477552866)

[Code Explanation 5](#_Toc477552867)

[Result 9](#_Toc477552868)

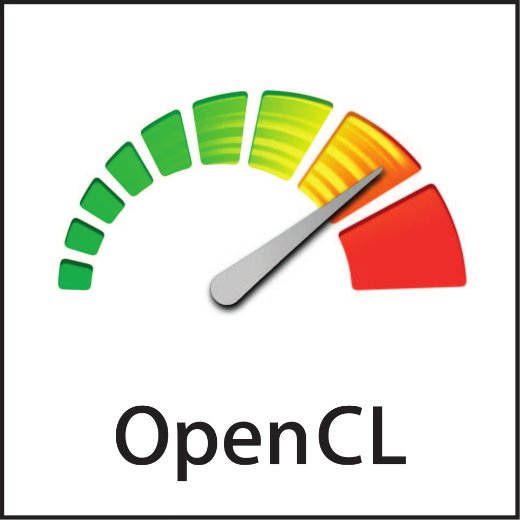
[Optimization 10](#_Toc477552869)

[Process & What I learned 10](#_Toc477552870)

[Conclusion 11](#_Toc477552871)

[Reference 11](#_Toc477552872)

# OpenCL

OpenCL is a framework developed for computing in both CPU and GPU. It specifies based on C99 for control the platform between CPU and GPU. It provide a detail insight for program execute in kernels. It allows programmer to control and understand the memory utilization. Unlike CUDA (another GPU computing programming language specify for NVidia GPU), OpenCL is extremely portable. It allows programmer to natively program on large range of device. Although it does not optimize for speed, OpenCL is portable between many devices.

Ever since OpenCL is started in 2009, many mathematical problems were simulated using OpenCL kernel. Heavy computing mathematical model which are easily parallelable, are best for using kernel computing. Image Ray-tracing is a good example for computing using OpenCL kernel.

# KNN Algorithm



K nearest neighbor’s (KNN) algorithm is one of the simplest and widely used machine learning algorithm. Machine learning focus on training the algorithm to do either classification or regression. In this project, a classification technique is studied. KNN is a type of supervised learning. It takes input of large set of training example of known output to analyses and predict the next output of testing data. Providing the function with increasing amount of data would eventually build a best-fit model for all incoming testing data. KNN algorithm consist of 3 parts, distance matrix calculation, sorting, and majority. For each of the testing data, KNN algorithm calculate the distance between every reference point and testing data to build a distance matrix. Then based on the distance matrix, the k nearest reference data is used to calculate the testing data’s class, based on the majority of class of the k nearest data. KNN algorithm is sensitive to local structure of the reference data to provide the best result.

Euclidean

For multiclass KNN algorithm (k greater than 1), KNN algorithm is guaranteed to yield an error rate of no worse than twice the Bayes error rate (minimum error rate of distribution of the data). A special case of the KNN algorithm is 1-nearest neighbor classifier. And in this case the nearest neighbor is used to define the class of testing data.

The process of choosing the k value also directly related to the accuracy of KNN algorithm. For a theoretical properties, **k < sqrt(n).** And in this project, small value of k were used. But for theory, k can be a larger number which is less than square root of n.

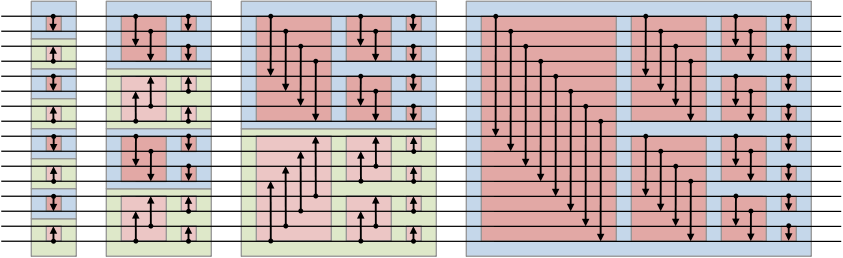
# Weighting Scheme

KNN algorithm sensitive to noise. For clustered data and a large number of k (k > 10), a weighting scheme is comely used to reduce the error rate. The weighting scheme generally multiply the k nearest neighbor’s class by 1 over the distance, to increase the importance of closer reference data.

Weighted Euclidean

# Sorting (Bitonic)

Many sorting algorithm is studied for faster computing speed. This project focus of Bitonic merge sort. Although Bitonic sorting algorithm is a O(n log2 n) and is much slower than radix sort, it is significantly faster than O(n2) bubble sort. “Bitonic sort used by GPUSort does O(n log2 n) work … (compare) to O(n) radix sort.” (Satish) Bitonic sort is designed for parallel computing unlike many other sorting algorithm. It make use of building a sorting network by merging result. An image below provide a good understanding about the connection of Bitonic sort network. Each vertical box’s content can be done in parallels, and its halt and wait for each event in vertical box to finish before moving to the next stage. The Bitonic sort does not really make sense to sort in sequential, so a Batcher’s odd-even merge sort is used in host side for sequential reference.



Sorting (Batcher\_odd–even\_mergesort, size , and depth

# Complexity and Estimated speedup

AI = computational work / communication = W(work) / Q(memory traffic)

|  |  |  |
| --- | --- | --- |
|  | Complexity | AI |
| Distance matrix calculation | O(n) | , d=2  (for each test point) |
| Sorting | , non-parallel  (Best, Worst, Average) Performance in parallel = | (for each test point) |

Speed up = Sp = , where T1 latency of program with one worker, and Tp is the latency of program with P workers

Amdahl’s Law = Sp , Wpar in this case is parallel portion of the W, which is all the comparisons.

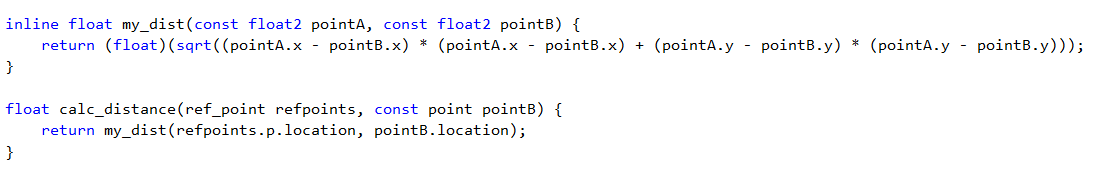
# Code Explanation

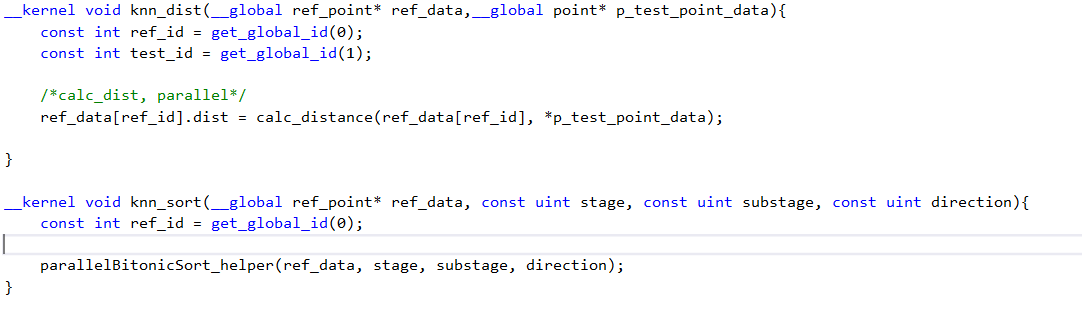
Machine used in this project is the following

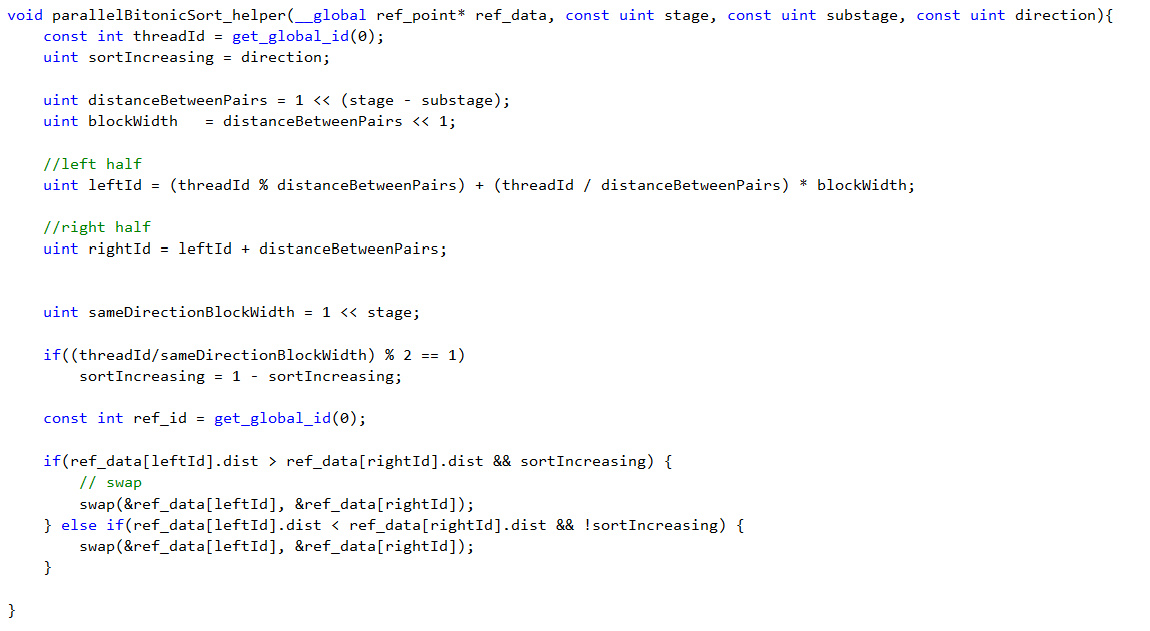
GPU: Intel® HD Graphics 530

CPU: Intel Core™ i5-6500T CPU @2.50 GHz

Kernel (device side) code:



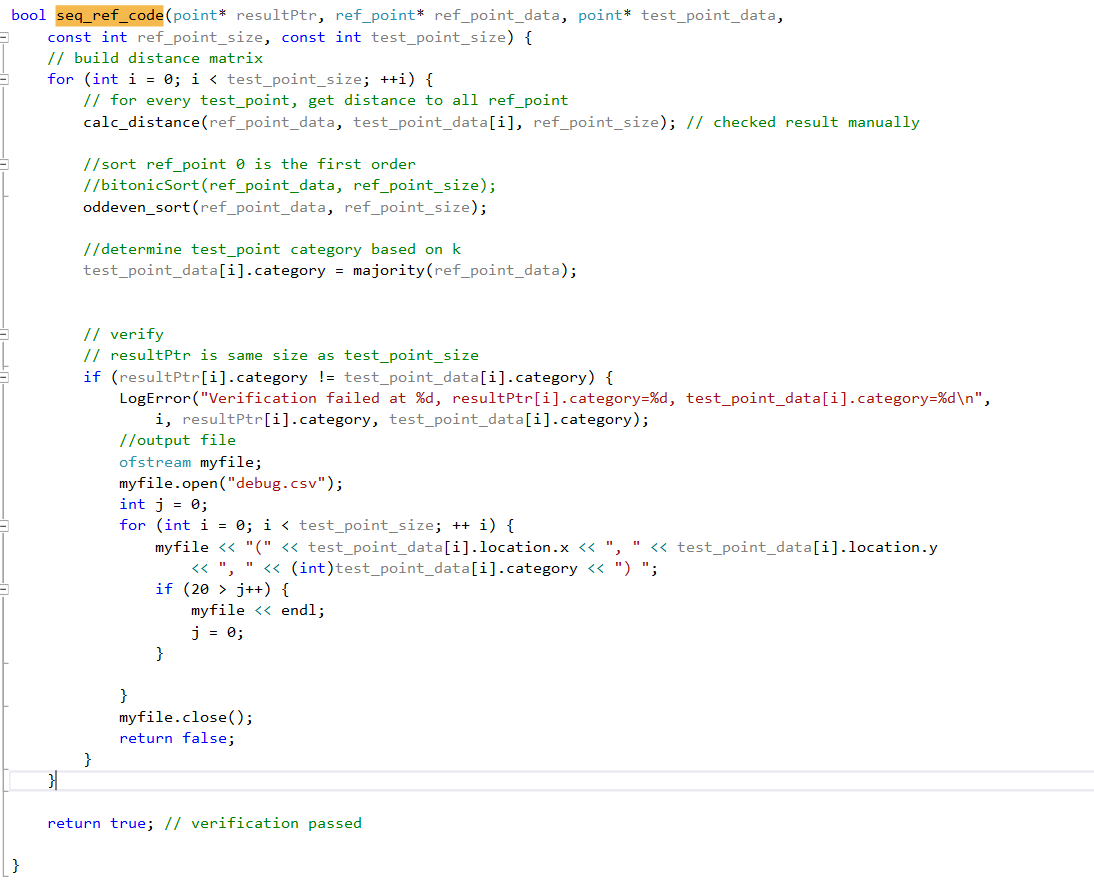


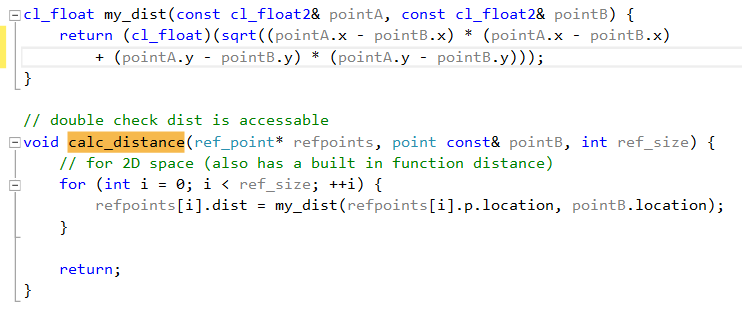


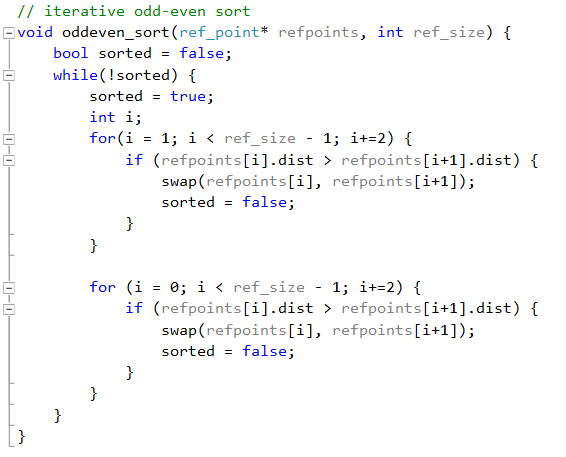
In this project, the calc\_distance and my\_dist is a helper function for knn\_dist kernel. Knn\_dist is the first part of the knn algorithm which calculate the distance matrix. The knn\_dist kernel would calculate the distance between all reference point and one testing point. The main purpose of helper function in here is for code cleanliness. The parallel BitonicSort\_helper function is call by knn\_sort, the second part of KNN algorithm. Both the first part and second part is parallelize using kernel. Although an attempt were made to parallelize the third part of KNN algorithm, it was not successful and sequential code were used instead. In calculation of performance timing, the third part of the KNN algorithm is ignored, as it is a very simple and quick computation, and negligible compare to the distance matrix calculation and sorting algorithm.

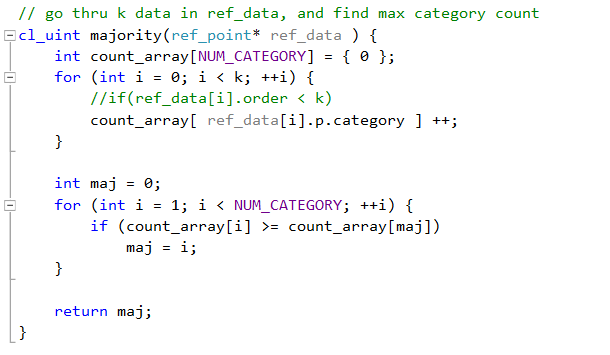
Host side code:

The hose side sequential reference code is used for verifying the result of parallel computed result on the device side and also the comparing the profiling timing. The host code was able to group into one main function, *seq\_ref\_code­.* The sequential host code still divide the KNN problem into 3 parts and call related function.









The output result would output failed verifications to debug.csv file for detail inspection. Verification passed on all case except for an unknown classification when k nearest neighbor has two equal number of category reference code. An example would be for k = 6:

In the example shown above, two equal number of category is within k nearest neighbor. When this case happened, a random category between the two is choose to assign to the test point. Therefore the verification simply did not provide a good enough test.

## Result

GPU kernel

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time | Reference size | Test size | k | Category | Local size |
| 164.45 ms | 256 | 8 | 10 | 50 | 4 |
| 256.16 ms | 1024 | 8 | 10 | 50 | 4 |
| 484.41 ms | 1024\*16 | 8 | 10 | 50 | 64 |
| 3547.74 ms | 1024\*16 | 64 | 10 | 50 | 64 |
| 14401.12 ms | 1024\*128 | 128 | 10 | 50 | 64 |
| 34223.5 ms | 1024\*512 | 128 | 10 | 50 | 64 |
| 44548064 ms | 1024\*1024 | 16 | 5 | 30 | 64 |

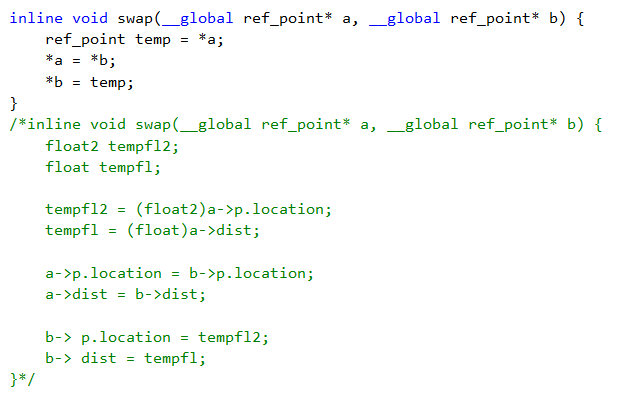
Host

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time | Reference size | Test size | k | Category | Local size |
| 3.44 ms | 256 | 8 | 10 | 50 | - |
| 38.05 ms | 1024 | 8 | 10 | 50 | - |
| 10125.99 ms | 1024\*16 | 8 | 10 | 50 | - |
| 22791.06 ms | 1024\*16 | 64 | 10 | 50 | - |
| 64446.18 ms | 1024\*128 | 128 | 10 | 50 | - |
| Over 10 mins | 1024\*512 | 128 | 10 | 50 | - |
| Over 50 mins | 1024\*1024 | 16 | 5 | 30 | - |

Results shown in the above table has shown that on a small data set. Sequential would provide a better result. As the overhead of moving and accessing the global host memory excessed the optimization of paralleled part. Host side sequential code does not need to stop and wait for each thread to finish to start the next set of execution. For smaller data set (reference\_size < 1024, test size = 8), host side sequential is faster than GPU kernel side. However, for reference size > 1024 \* 16, GPU kernel is significantly faster ~ 20 times faster. For reference size > 1024 \* 1024, the result of host side sequential code does not even provide a result after a wait time of over 50 minutes.

The result has shown that a successful result. Computing the KNN algorithm with GPU achieved a better result after initial set up overhead.

## Optimization



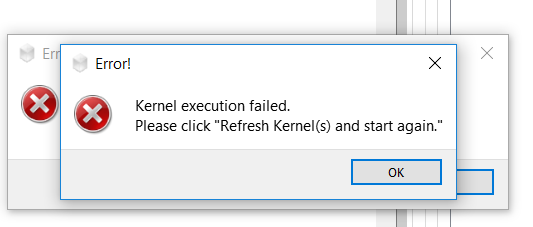
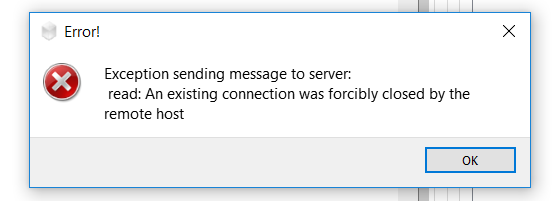
Very few optimization of the kernel side code is made in this project. Mainly due to the fact that OpenCL in Visual Studio does not support struct. Manual optimization on local size is used in this project, which increase the processing time for each execution.

On kernel side code, accessing memory to the global memory was the most time consuming part of the algorithm. An optimization would be bring a portion of the global memory to local memory to compute. However this is not implement due to the time constrain of this project.

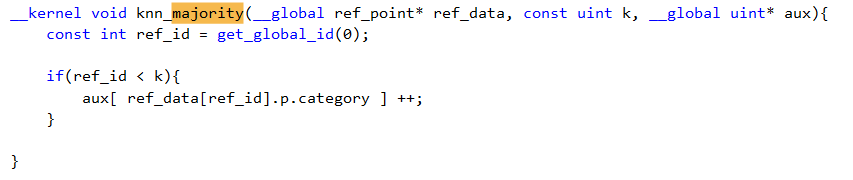
VTunes and KDF were not used in this project also due to the functionality of user define struct is not implemented in Visual Studio.

# Process & What I learned

In this project, I had a lot of trouble using OpenCL to pass information between host and device and optimize the kernel. I used a struct to represent each point and reference point. Struct provide a clean and simple way for my solution to easily increase number of properties for each data point. Switching from 2d to 3d, or adding 10 more properties for each reference point would be easier to implement using struct. However, visual studio OpenCL does not support struct for kernel side Code Builder. Therefore, I had an attempt to use the Intel OpenCL Code Builder, which support user defined struct. However, the OpenCL Code Builder by Intel is deprecated and it does not provide any result.



The most difficult part of this project is to pass piece of memory from host side to kernel side. Although it seems to contain very simple of allocating space, create buffer, and set kernel argument, after receiving the data from host side I had trouble writing global address space variable from kernel back to host side. Variable updated in kernel did not successfully update in host side. Another problem that I had was counting in kernel side. After some research, a global address qualifier is not enough to use as a counter, rather a volatile keyword need to be used. A racing condition were also studied during this project. During the third part of KNN algorithm, the problem I encounter was during the finding of majority category count and update to the host. For example,



If 2 thread received in kernel both tried to update the same location, only one of them would be successfully increase the aux array for the example above. The only solution I found is to move the majority part to host side to compute it sequentially.

# Conclusion

In this project, a lot of effort were made to contain everything in one single kernel. However, the way that I implement the Bitonic kernel, does not allow it to contain everything in kernel. Although KNN algorithm can be implemented without using struct, I am interested in learning that component of kernel computing. Passing struct as a kernel variable from host to device did not discuss in details in class. If I would have implement the kernel without a struct, many steps would get simplify and I will also be able to use VTunes and KDF. However, the portability of the code would decrease.

Due to the time limitation of the project, the kernel did not get fully optimize. For the future, optimization such as avoid accessing the global memory every time the kernel is use by utilizing the local memory, using another more efficient sorting algorithm can be done to the kernel to achieve a faster KNN algorithm. Many other parallel sorting algorithm were also studied in this project. It provides me a better understanding of the difference in parallel computing versus sequential.

# Reference

[Cover TM](https://en.wikipedia.org/wiki/Thomas_M._Cover), [Hart PE](https://en.wikipedia.org/wiki/Peter_E._Hart) (1967). "Nearest neighbor pattern classification". *IEEE Transactions on Information Theory*. **13** (1): 21–27. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1109/TIT.1967.1053964](https://dx.doi.org/10.1109%2FTIT.1967.1053964).

Hai Peng, Letian Huang(2016). “An Efficient FPGA Implementation for odd-even sort based KNN algorithm using OpenCL”

Jan Masek, Radim Burget (2016), “Multi-GPU Implementation of Machine Learning Algorithm using CUDA and OpenCL”

Kuang, Quansheng, and Lei Zhao. "A practical GPU based kNN algorithm." International symposium on computer science and computational technology (ISCSCT). 2009.

“OpenCL Parallel Programming Development Cookbook” by Raymond Tay (2013)

Nadathur Satish (2008), “Designing Efficient Sorting Algorithms for Manycore GPUs”

<http://www.bealto.com/gpu-sorting_parallel-bitonic-1.html>

<http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm>

<https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm>

<https://en.wikipedia.org/wiki/Bitonic_sorter>