UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient Implementation of Hierarchical Navigable Small World Similarity Matching Algorithm

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Yu Zhong

Committee in charge:

Professor Truong Nguyen, Chair
Professor Chung-Kuan Cheng, Co-Chair
Professor Patrick Pannuto

2021

The Thesis of Yu Zhong is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures v

List of Tables vi

Abstract of the Thesis vii

Introduction 1

Chapter 1 Detailed Overview 2
1.1 Quantization Method 2
1.2 Hashing Algorithm 2
1.3 Graph Based Method 5
1.4 Thesis Objective 12

Chapter 2 Experimental Design and Preliminary Studies 13
2.1 Dataset Introduction and Study 13
2.2 Hyperparameters Pre-tuning 13
2.2.1 Purpose of Tuning 13
2.2.2 Experiment Results 15

Chapter 3 Parallel Computation Improvement 18
3.1 Overall Introduction 18
3.2 General Improvement Overview 20
3.3 Details on Three Proposed Improvements 22
3.4 Results 25

Chapter 4 Conclusions 30

LIST OF FIGURES

Figure 1.1. Locality Sensitivity Hashing Representation [1] 3

Figure 1.2. Demonstration of High Dimensional Vectors [2] 6

Figure 1.3. Approximate Nearest Neighbors Oh Yeah(ANNOY) Illustration [3] 7

Figure 1.4. Small World Graph [4] 9

Figure 1.5. HNSW Multiple Layer Structure [5] 10

Figure 1.6. ANN Benchmark Results [6] 11

Figure 2.1. Hyperparamters Results Under 100 ef. Top - precision/recall curve. Bottom: Time against Number of Nearest Neighbors in Query 15

Figure 2.2. Hyperparamters Results Under 150 ef. Top - precision/recall curve. Bottom: Time against Number of Nearest Neighbors in Query 16

Figure 3.1. CPU vs GPU 19

Figure 3.2. CUDA Hardware and Software [7] 21

Figure 3.3. New Tensor Cores For FP16 [8] 24

Figure 3.4. CUDA Inter Layer Calculation Results: : Time (seconds) against ef_{SEARCH} 26

Figure 3.5. CUDA Batch Query Results: Time (seconds) against ef_{SEARCH} 28

LIST OF TABLES

Table 3.1. Relationship Between Number of Calculation and Node Number 22

Table 3.2. CPU Memory Consumption 26

Table 3.3. Batch Query Results, Measured in Multiplication of Size = 1 28

Table 3.4. FP 16 Comparison using 150 ef for search on 48-66 preset 29

Table 3.5. GPU Memory Consumption on Different Number of Vectors and Hyperparameters 29

ABSTRACT OF THE THESIS

Efficient Implementation of Hierarchical Navigable Small World Similarity Matching Algorithm

by

Yu Zhong

Master of Science in Computer Science

University of California San Diego, 2021

Professor Truong Nguyen, Chair
Professor Chung-Kuan Cheng, Co-Chair

This project focuses primarily on improving the implementation of the Hierarchical Navigable Small World Algorithm by maintaining the hierarchical structures and certain path of query utilizing the GPU for better performance. Specifically, multiple queries could be now used and the overall time is reduced to around 83%. Such goal is achieved by investigation of parallel computation within layer nodes, batch query, and using half-precision representation which could increase query speed without affecting the actual precision and recall of the algorithm. By incorporating these modifications, we are able to improve the algorithm in for a query by a factor of 2% - 5%.

Introduction

As applications of artificial intelligence and machine learning thrive, speed of accumulating data today is increasing and there is an urging need for faster and better matching algorithm, especially in tasks such as detection, recognition, and identification. Naive approach such as brute force comparison suffers from the larger scale and high dimensions, in which dampens the possibilities of using simple approaches for high dimensional data [9]. Therefore, as higher and higher dimensional data becomes essential in daily application such as protein deconstruction and matching [10], word2vec [11, 12] embedding matching, researchers realize that exact results based on exhaustive searching is no longer an achievable goal, and consequently approximation search becomes the mainstream similarity searching approach, corresponding to methods that only search portions of the entire data to achieve efficient results. However, balancing between partial searches and overall speed of the searches introduces a new challenge. For instance, as it is commonly known that brute force refers to thorough comparison between the query data and all data, if one reduces the number of comparison by certain percentage, then the speed of one query will benefit from such reduction. However, as the number of comparison diminishes, the recall and precision would probably suffer, resulting in a fast but inaccurate matching result. In general, trade-off between speed and accuracy underlies every matching algorithm.

Chapter 1

Detailed Overview

1.1 Quantization Method

Facebook research group proposes the quantization method [13] to greatly compress the storage and minimize memory usage during query phase [14]. Meanwhile, by exploiting parallelization computation of graphics processing unit (GPU), the brute force search algorithm can in fact be efficiently implemented and used with parallelization of the hardware design. As proposed in [6], multi-stage quantization significantly reduces the storage space and memory consumption, but to some extent, also adds into inaccuracy representation of the original higher dimension data. Therefore, a trade-off is necessary for using compression technique such as quantization. Nevertheless, quantization proposes a new way of approaching highly complicated and massive amount of data due to the advantages of highly-parallel computation on graphic processing unit, in which it provides possibility of high throughput of data [15, 16]. Therefore, despite the disadvantages of losing original features through compression, quantization still remains a competitive algorithm due to its capabilities to fully utilize hardware to reach unprecedented speed of query, especially for large database [6].

1.2 Hashing Algorithm

In compression of original data features, hashing algorithm represents another approach, which enhances the ability of storing more data on lower level hardware, therefore increasing

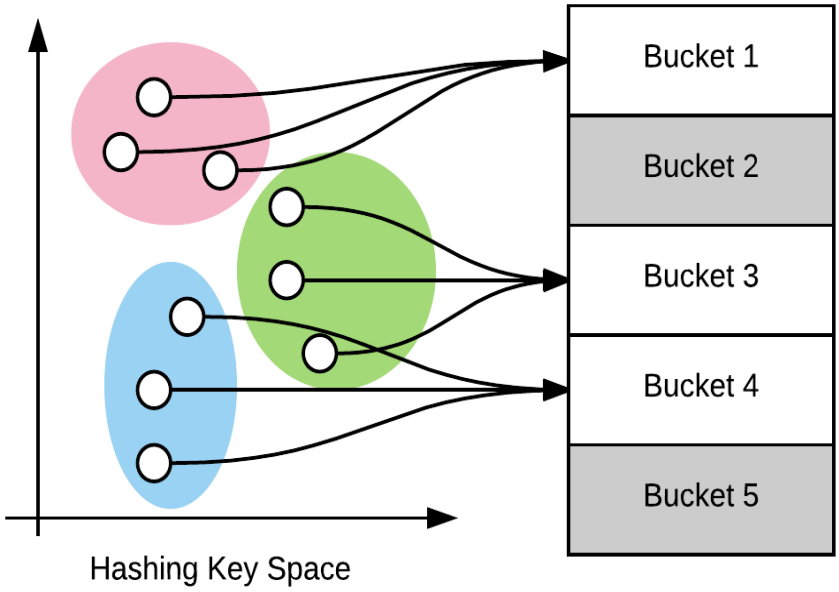

Figure 1.1. Locality Sensitivity Hashing Representation [1]

the speed of calculation by exploiting hardware, such as memory and registers, instead of hard disks. Locality Sensitivity Hashing (LSH) [17], represents another method of compressing original features of the data to preserve accuracy and precision but still occupies fewer memory spaces and storage spaces. Instead of seeking minimal hash collision, hashing, as a technique applied in similarity matching, aims for maximization of hash collisions. Therefore, similar data points are able to categorized locally, and through this locality sensitive philosophy, it is easier to approximate a nearest neighbor in high dimension without involving all dimensions of the original features.

The essential concept of the implementation of LSH algorithm is to assume that high-dimensional data tend to have similar hashing values if they appear close to each other locally (Figure 1.1). On the other hand, once two data points are far away from each other in terms of dimensional spaces, it is unlikely for two points to have similar hash values, therefore, collisions, resulting the following categorization is not favorable. Based on aforementioned philosophy,

it is advisable to choose distance function and hashing function wisely to achieve such goal. For instance, whether the distance function is metric or not would contribute greatly to the final performance of the LSH algorithm, due to the reason that metric function is directly connected to hashing collision control of all data points. Hence, multiple novel concepts of metric function are in fact playing critical roles in LSH algorithm [18, 19]. With suitable metric function, the other core concept should affect the choice of the hashing function. For instance, the initial proposal of metric function, which is the common Euclidean distance, has no match for a perfect hashing function and therefore such metric cannot be used in high dimensional data. To better accommodate the hashing requirement, Hamming distance and cosine distance are two possible candidates. In terms of the process of the entire similarity matching mechanism, LSH algorithm shares the same procedures as most of the other matching algorithms:

1. Build off-line index for database using a pre-determined algorithm, such as the LSH algorithm: the first step is to choose metric function and corresponding hashing function that satisfy the requirement.

2. Adjust the hyperparameters such as number of hash tables, and key length. With all parameters tuned to ensure high collision between similar data points, index can now be constructed with the selected hash function arrays and tables.

3. During the search phase, the query vector is mapped through hashing function to achieve a unique coding and such coding leads to specific bucket of the hashing tables. Within the same bucket, candidates are chosen based on one of the several methods, can be simple as first ten candidates, or more complex method that involves distance calculation. Among all candidates, an exhaustive approach is used to select the final k nearest neighbor [19, 20].

Based on the development of LSH, certain implementation are widely accepted and used. For instance, the very first application of sensitivity hashing: it utilizes projection based on stable distribution, in which users have to choose two parameters before the construction of

the indices, and the hash values are represented by integers instead of bits. However, the first application requires users to choose two parameters before usages, causing extra amount of tuning experiments before applying. Aside from choosing hyperparamters, the hash values are in forms of integers, also causes prolonged period of transforming original data and consumes great amount of storage space. Moses Charikar, one of the major contributors in development of the LSH algorithm implementation, proposed of a novel method of hashing all vectors under metric of cosine distances [19]. Hyperplanes are used in his methodology and some of the advantages of such approach includes

1. No hyperparameters needed before hashing.

2. Hash values are inherently bits representation and therefore avoid excessive transformation before matching.

Soon Moses Charikar had perfected the hyperplane idea and therefore SimHash was first brought to the LSH application: this new implementation circumvents the extra cost for unnecessary projections onto hyperplanes, and was designed for token-like features [20]. Later development of LSH branched out into different implementations due to the introduction of kernel functions into the metric calculation between vectors. Special situation includes the uneven distribution of hash key values, where the users may need to zoom into or out of certain areas to discern areas with various densities. Moreover, with the introduction of kernel-based metric function, supervised learning is also incorporated in certain advanced new implementations, and therefore it might become more user-friendly to those who apply LSH in similarity searching, especially for natural language processing area [21, 22].

1.3 Graph Based Method

The last area of similarity searching roots in one of the most common searching algorithm– tree-based search methodology. One of the most common and popular philosophy is to dissect the

original dataset into smaller datasets and therefore it is faster to skim through smaller portions of data instead of the entire dataset. Tree-based method is the easiest way to implement divide-and-conquer philosophy. For instance, using binary search tree can easily sort the entire dataset and limit the maximum number of searches by half. However, similarity matching has different goals compared to sorting and therefore tree-based algorithm had been considered unfavorable due to its lack of compression of original dataset and extremely challenging toward high-dimensional data. A typical application of tree-based algorithm is the KD-tree method [23], which is indeed less efficient for classifying high dimensional data. KD-tree determines the dimension that has the largest square difference among all data and starts to divide data into halves and then proceeds to the next candidate of dimension. Consequently, when dimension increases, the construction time of the tree increases exponentially, resulting in low efficiency [24], such dilemma is also known for the name "Curse of Dimensionality" [1.2]. As Figure 1.2 shows, it is easy to divide and conquer in 1 dimension, slightly harder in 2 dimensions, and still doable in 3 dimensions. However, if the number of dimension increases to 100, the division will take much longer time.

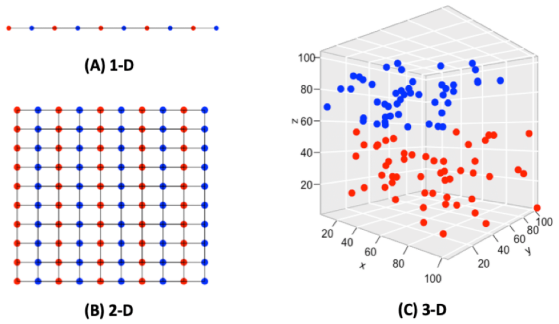

Figure 1.2. Demonstration of High Dimensional Vectors [2]

Approximate Nearest Neighbors Oh Yeah (abbreviated as ANNOY) is a famous implementation of tree-based algorithm for faster searching. Before construction of trees [3, 25, 26, 27], ANNOY first categorizes all data points into two clusters where the centers of mass are calculated for two clusters and a line is drawn to connect two centers. Then the normal plane of this straight line divides all data points into two separate regions and such plane is recorded as the root node

of a tree. Consequently, more and more normal planes are calculated and recorded during the construction of the indices. Eventually multiples binary search trees are established as index of each data point is assigned to every other data point. The final effects of such index construction process can be seen in a colorful representation in Figure 1.3. During the query phase, based on the normal plane information stored in trees, the query phase can bypass unnecessary plane calculation due to the fact that every normal plane separates clusters of data points. With different results from multiple trees, the return of the final results must be processed again to eliminate repeated elements and therefore union find (a common algorithm to traverse and search) is applied to all candidate nodes. However, clusters related algorithm is presented as heuristic results of unknown parameter tuning, in which the question on how to balance centroids distribution is not fully understood. In addition, due to the hyperplanes' calculation and corresponding storage space after the construction of indices, ANNOY sometimes suffers from excessive construction time, unlike aforementioned quantization method [9].

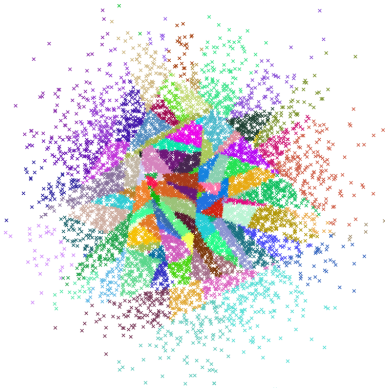

Figure 1.3. Approximate Nearest Neighbors Oh Yeah(ANNOY) Illustration [3]

Navigable Small World (NSW) [4, 5] method is a new approach to the similarity matching problem, where it has tree-like structures but does not precisely requires divide-and-conquer mechanism during implementation. Similar to the core idea of skipping unnecessary comparison in binary search tree, the NSW algorithm applies ideas of basic data structure such as skip list, which resembles linked list but there are shortcuts to skip through intermediate nodes to achieve

faster traversal of the entire data structure. With such structure, it is natural to assign all data points onto the corresponding nodes and then traversal and distance calculation can be highly optimized due to lower volume of comparison needed. As Figure 1.4 shows, one can traverse the entire graph using larger nodes, which can be the nearest neighbor node during query or index construction. For instance, if I traverse the graph on orange nodes, then large portion of blue nodes remain intact and therefore calculations are not needed. In general, NSW algorithm has the following steps:

1. Assign a random probability of an initial incoming data point.
2. Use greedy method to search for its nearest neighbors
3. Record such neighbor, use it as the next starting point to approach the next layer until the ground layer.
4. Store intermediate nearest neighbor information within the node at the mean time.

Similarly, during the query phase, the query node goes through the same process so that the final nearest neighbors can be returned instead of being stored. Within four years, Yuri Malkov, who originally proposed the algorithm of NSW, published new researches into an even more efficient data structure called Hierarchical Navigable Small World (HNSW) [5]. Hierarchical structure introduces a novel idea of classification of data points into level design, in which a pyramid data structure is created eventually, with the highest level storing least number of nodes, while the number of nodes increases exponentially as one traverses down the hierarchical structure, for the bottom layer encapsulates all data points, shown in Figure 1.5: red node can be query node or new node waiting for index construction. Blue node in layer 2 is the entry node. By calculating the distances between entry node and its nearest neighbor, one can access the other nodes in the same layer and acquire nearest neighbor in that specific layer. The process is repeated in every layer until layer 0, while the green node is actually the nearest neighbor of the red node. The probability of assigning each node into different level is completely decided by

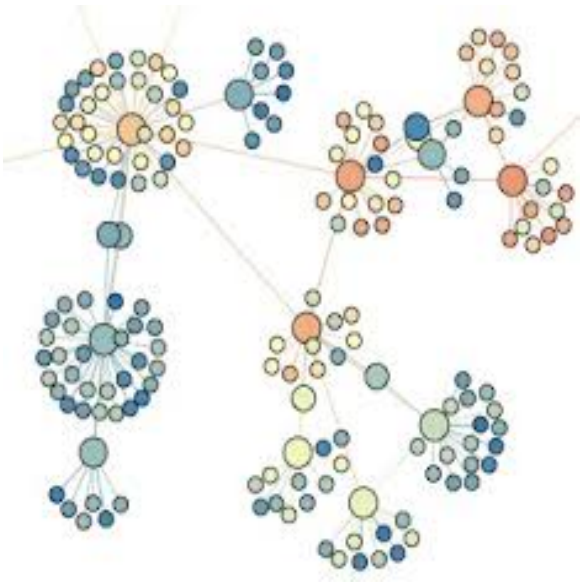

Figure 1.4. Small World Graph [4]

random outcomes of exponential function. As a result, every node will have equal probability of being assigned into different level and such heuristic process varies with each new construction even with the same data. One of the advantages of hierarchical structure is to further reduce unnecessary comparison calculation due to greedy approach applied only to nodes that are in the same level. By reducing necessary inter-layer and intra-layer comparison, researchers are able to reduce the number of comparison exponentially in Table 3.3 and empirically less than 2% of the total number of nodes [6].

One of the distinctive disadvantages of tree-based method is the inability of parallel computation, which is in fact one of the most noticeable advantages of quantization method since quantization refers to separate calculation of each nodes. Consequently, one calculation is independent from other calculation and hence all calculations are able to be distributed to different hardware, considering those hardware must record common information of node characteristics such as indices in order not to lose ordinal information.

Graphics processing unit is one of those special hardware made for parallel computation. For instance, Nvidia graphic cards provide specific hardware such as CUDA cores, which are designed for massive parallel computation. In addition to specifically designed hardware, Nvidia

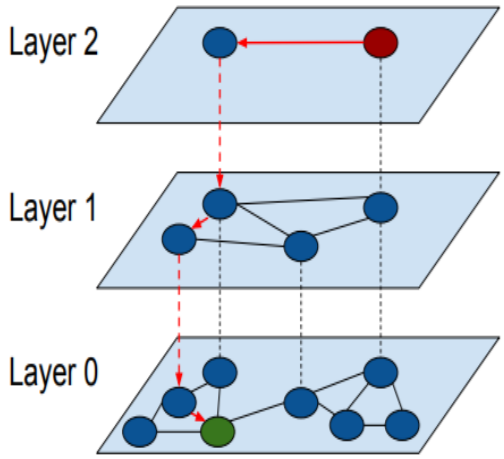

Figure 1.5. HNSW Multiple Layer Structure [5]

also provided corresponding software API, exclusively in C and C++ API, to enable developers to exploit the computation capability directly. C API provides developers with complete control of multi-level hardware, from the most basic unit thread, to larger integrated unit such as block and grid [8]. Such mutli-layer structure can be seen in Figure 3.2, in terms of software terminologies, grid consists of blocks, and block consists of threads. In terms of hardware, threads refer to CUDA cores or tensor cores. Likewise, besides computation units, the memory units are directly programmable given several levels of different priorities, such as register, L1 and L2 cache between blocks and grids which contain multiple basic hardware unit – thread. Such complete control of the finest unit of hardware becomes convenient for algorithm researchers to distribute certain calculations onto different units.

The easiest example that one can think of is the distance calculation during the index construction phase and the query phase. This idea seems very promising but there is one underlying issue of feasibility of massive parallel computation on similarity matching - all calculation should be independent from other calculation and otherwise the order of the results does not affect the algorithm itself. The naive way of brute force calculation is a perfect match due to its inherent independence of all distance calculation and it is indeed a beneficiary of

parallel computation. A carefully-implemented brute force algorithm can provide performance boost of more than 10 times than CPU only computation [14]. Quantization is another algorithm that be enhanced through parallel computation since the process of compressing original feature vectors can be highly parallel, and the compressed states can be sometimes accommodated by large GPU memory, at the very least benefited from the increasing GPU on-board memory for recent high demand for machine learning [14].

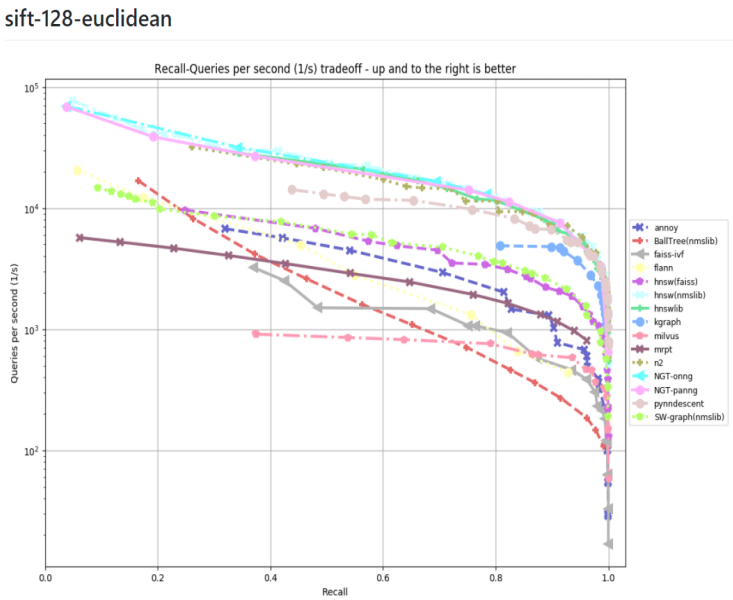

Figure 1.6. ANN Benchmark Results [6]

From the perspective of tree-based method, it is difficult for developers to apply parallel computation. The computation of distances in tree-based method is sequentially connected, because each calculation of distances has to depend on previous calculation results and will impact future calculation. For instance, when traversing a tree-based structure, the distance between one query node and the root node must precede calculation of its left or right node, because the result of such calculation determines the flow of traversal and will save unnecessary calculation from the other branch of nodes. In Figure 1.6, as one can see, the HNSW method lies on the top right corner of the speed-recall graph, showing great advantages over other algorithms (The horizontal axis refers to recall and the vertical axis refers to query per second. In general,

"higher is better" applies to the graph and top right corner indicates that the algorithm is fast and accurate). Consequently, despite the fact that tree-based method usually provides the best recall and accuracy [6] rate among all other similarity algorithms, the efficiency of the algorithm is one of its major disadvantages because only CPU and memories can be utilized to implement the algorithm, instead of GPU that was built for massive parallel computation. The Navigable Small World algorithm, despite achieving significant success of reducing distance calculation through both index construction and query phase, still suffers from lack of general information within one layer. The consequence of such situation is that when data reaches the scale of millions or billions, even intra-layer greedy search can be very slow that overall performance is weakened.

1.4 Thesis Objective

The purpose of this study is to exploit GPU for Hierarchical Navigable Small World Algorithm and enable certain parallel computation during index construction and query phase, despite the fact that tree-based structures are not able to be altered into parallel structures that can perform multiple independent calculations at the same time. However, we plan to calculate extra information within each layer and aims to gain performance improvement by exploiting the GPU. To achieve the goal, parallel computation is exploited through GPU and the intra-layer computation should improve greedy performance in searching the nearest neighbor. Additionally, the multiple query can be realized after intra-layer computation is done so that multiple query search can be enabled based on users' choices. Furthermore, we delve into the latest floating point (16 bit) feature of newer generation of graphic cards, planning to discover further performance boost.

Chapter 2

Experimental Design and Preliminary Studies

2.1 Dataset Introduction and Study

To further refine the general study, the dataset that are used in this research consists of mainly 256 or 512 dimension features of de-sensitized human faces, in which all values have been normalized between 0 and 1. One of the distinctive features of such dataset is that the closest neighbor of one single vector originates from the feature vector of the faces from the same person. Therefore, it is relatively convenient to construct ground truth for every single feature vector. Furthermore, the high dimension ensures that features of each vector is evenly distributed in all directions, avoiding clusters of feature vectors to form and therefore might introduce bias to the implementation of the algorithm. The number of the entire data range from 97,765 feature vectors to more than 550,000 feature vectors. Experiments are conducted on different subsets of the larger dataset due to time consumption consideration.

2.2 Hyperparameters Pre-tuning

2.2.1 Purpose of Tuning

To better achieve the optimal states of speed comparison, the hyperparameters of the HNSW algorithm must be adjusted to reflect the most saturated navigable small world structure.

Specifically, several hyperparameters in the algorithm application are crucial to the overall performance of the algorithm. For instance, there are mainly two important parameters in the algorithm, ef and M and the search phase and query phase follow the same philosophy. The prior parameter controls the number of candidates that have the closest distance to the incoming new element, in which this can be a new element waiting to be inserted or a query vector. Upon the traversal of the entire data structure: a higher ef corresponds to a larger list of possible candidates and therefore traversing all candidates will consume more time compared to that of a shorter list. The latter parameter M refers to the number of nearest neighbor stored in every node once the construction of index is finished for the entire database. This parameter is straightforward for understand since traversal has to be completed after searching every bi-directional link stored in every node.

Due to the inherent design of the structure, merely increasing ef will not boost performance significantly, because recall and precision have a upper bound of 100%. Similarly, if M is set to be higher, the construction process takes longer time and the level of connection between nodes become more complicated. Researches have found that lower M setting can be sufficient enough for lower-dimension data but with higher dimensional datasets, higher M is more preferable [5, 6]. Both hyperparameters contribute to the quality of indices of the entire datasets and will negatively affect the speed of query if both were set too high. Balancing the speed and accuracy of query is always the ultimate goal of any similarity matching algorithm: refined searches or construction will typically leads to more accurate results with higher recall but the time consumption can be intolerable. By analyzing the features of the datasets, researchers can ensure the most efficient combination of the hyperparameter set and such set will fix a performance baseline for further improvement that will be conducted later in the research.

To generate the closest neighbors of all input feature vector, researchers have utilized brute-force methodology to generate database format with 10 nearest neighbors of all feature vectors, in both reduced dataset and comprehensive datasets. In the tuning phase, the 2 hyperparameters are fixed in turns during index construction phase and the following query phase as

well. The entire construction phase and query phase are both timed and the results are collected afterwards. Meanwhile, the recall and precision are calculated and compiled into the following table and data graphs.

2.2.2 Experiment Results

As one can see from multiple results calculated from different set of parameters (Figure 2.1, 2.2), the precision and recall will actually increase while pairing with higher number of M and ef_construction. However, as the set of hyperparameters increases, the time consumption of entire query phase is prolonged but performance is saturated. Therefore, for further studies and statistics, only 2 presets of hyperparameters are picked to save time in experiments.

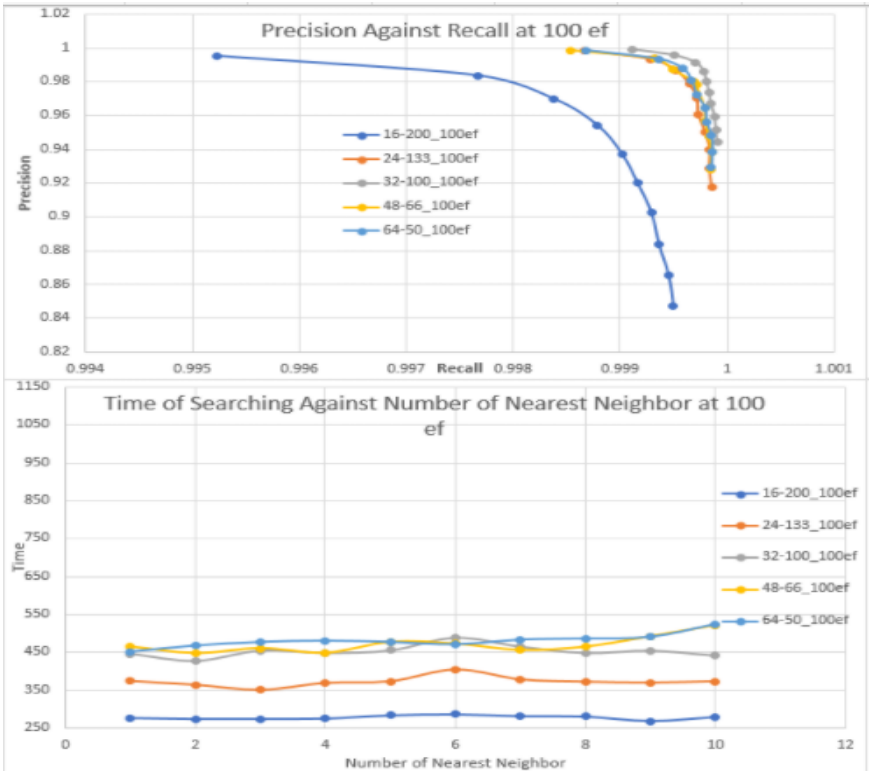

Figure 2.1. Hyperparamters Results Under 100 ef. Top - precision/recall curve. Bottom: Time against Number of Nearest Neighbors in Query

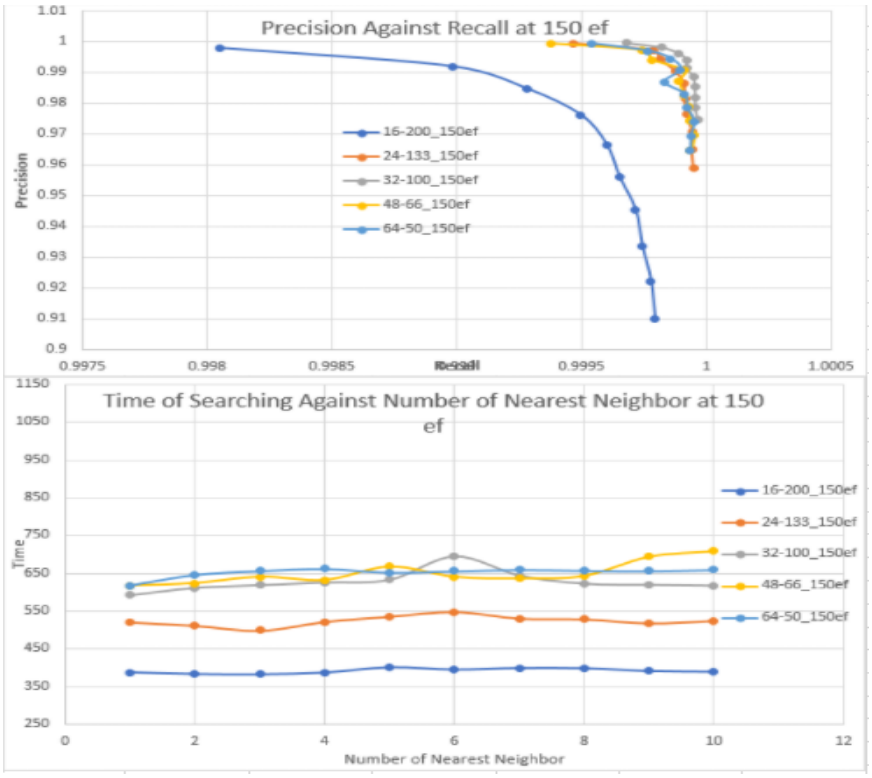

Figure 2.2. Hyperparamters Results Under 150 ef. Top - precision/recall curve. Bottom: Time against Number of Nearest Neighbors in Query

Generalization of Hyperparameter Tuning

From the summary graph and different parameters from above data, the corresponding M should be set between 48 and 64 and M_construction between 50 and 66 to achieve a reasonable recall and precision while not unnecessarily extending the construction process to an unbearable level. Setting M to a reasonable range has guaranteed the experimental datasets a decent quality of indices and the looming potential of undue parameters that can affect future implementation has been reduced to minimum. Upon query phase, a careful setting of the ef value will not only reduce the time consumption of a single query but also guarantee the overall performance of the algorithm. Too low of accuracy and precision will dampen the performance boost gained from improvement of algorithm implementation. From the data shown above, the best combination for a high dimensional data is M = [48.66],[64,50], the ef_search should be set around 150.

Consequently, the following studies are conducted based on these sets of parameters.

Chapter 3

Parallel Computation Improvement

3.1 Overall Introduction

One of the disadvantages of any tree-based data structure is to suffer from sequential calculation due to traversal of the data structure. Such traversal pattern must be completed from start to end, in which the entire process cannot be paused or distributed onto different hardware. Hence, the load of traversal is dependent only on Central Processing Unit (CPU) and different levels of memory, including L1, L2, L3 cache inside CPU and memory. Based on the Von Neumann architecture [28], I/O speed of different levels of memory decreases as storage increases, and will be further away from the instructions center - the CPU. Such design guarantees registers that store mostly instructions and address will be the fastest hardware component and closest to the CPU cores. L1, L2, and L3 cache will then be the next level high-speed storage for very minimum space due to its relatively high speed. Considering the largest server process only consists of 30 to 100 MB of L3 cache, most data storage, upon dealing with millions of feature vectors must be stored inside memory. Memory, on the next level of storage option, provides reasonable amount of size to store certain data including process and thread information, such as stack and heap information, allocated memory and so on. Most personal computers now are equipped with 4 to 16 GB of memory on the motherboard and typical DDR4 memory speed throughput is around 21000 MB/s [29] with very low latency. Even with decent speed of memory, the storage can be a problem upon processing millions of feature vectors. Hence,

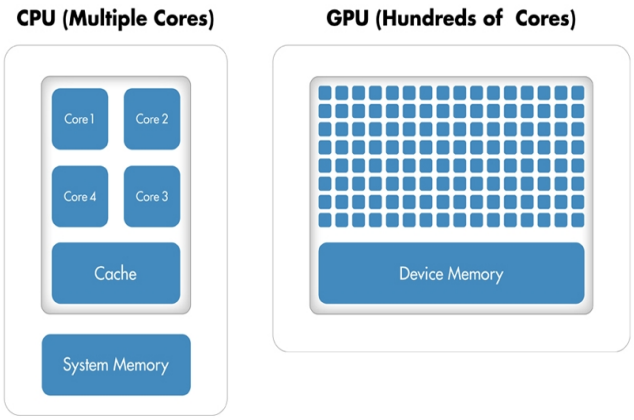

Figure 3.1. CPU vs GPU

early implementations of matching algorithm were heavily based on CPU and memory storage, if encountered cache miss in memory, the page swapping between memory and hard drive can be extremely time-consuming to running the algorithm.

Due to such restriction, compressing features have become one method to increase performance as it is mentioned before. For instance, quantization of the feature vectors compresses the size of vectors to enable greater storage inside memory. Most calculation can now happen inside high speed memory instead of swapping to a much slower hard disk. While compression alleviates the storage pressure on computer system, modification on calculation process can benefit matching algorithm in another aspect. CPU is designed to process fast and sudden instructions in daily routine. With such purpose, tolerance to high latency is low since dispatching instructions and finishing tasks that in most cases are small and rapid response is the uttermost important goal. The purpose of the CPU leads to hardware design that the corresponding storage counterpart must be responsive in short time but can be relaxed on extremely high throughput. This is how memory is designed, produced and iterated through generations [30].

The graphic cards unit, despite being separated from the core system and sometimes be integrated within CPU, deal with distinctive scenarios otherwise. Vast data coming from processing pixels of information requires high throughput but can be tolerant towards latency.

Accordingly, the storage counterpart of Graphic Processing Unit(GPU) is the memory on graphic card. Although in principle the graphic memory is Random Access Memory, same as what memory on motherboard refers to, the graphic memory is designed to have higher in frequency and larger in bandwidth to accommodate the need of high throughput in graphic processing unit. Considering what the graphic memory can provide, some data can be transferred from CPU and memory combination to GPU/Graphic memory combination under certain circumstances, one of which is parallel computation. FAISS method [14], exploits both the advantages of parallel computation and high throughput of graphic memory transfer speed, through quantization, the data are compressed and stored into graphic memory and the high degree of parallel computation boosts the throughput between GPU and graphic memory. Therefore, FAISS is one of the major algorithm that pushes the utilization of GPU upon matching to an unprecedented level and can manage more than billions of feature vectors [14]. Given the advantage of high recall rates of HNSW but lower throughput via CPU and memory, improvements of the algorithm implementation can be via the following approaches.

3.2 General Improvement Overview

The following three steps are proposed to improve HNSW computation using parallel computation on GPU:

1. Increase throughput via certain pre-calculation within layers.
2. Implement batch query via multi-thread.
3. Apply less accurate floating point 16 to compress data storage and speed up GPU calculation.

The first step applies basic parallel computation dealing with inter layer distance calculation. Consider the core algorithm of HSNW, consecutive calculations of the distances between nodes in different layers can be precalculated after layers are assigned. After all data points are

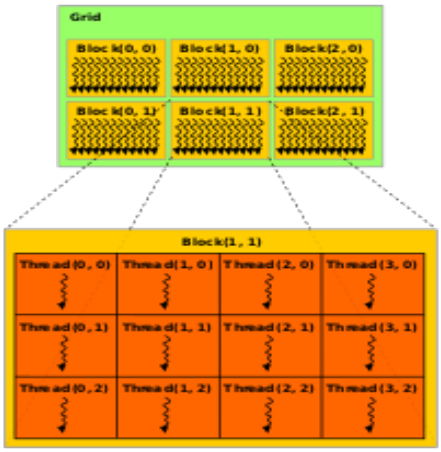

Figure 3.2. CUDA Hardware and Software [7]

categorized into different layers stated in original implementation, we apply parallel computation starting from layer 1 to (n-1) layer. The reasoning behind this range is listed as following:

- Layer 0 contains all nodes, parallel computation in this layer will be too time-consuming (actually equivalent to brute force method).
- Layer n contains the entry node only, no need to do any calculation.
- Layer 1 to n-1 contain partial nodes and can be accelerated through parallel computation. Normally, this method can save number of comparison roughly 10% of the number of nodes in Table 3.3.

Consequently, in the query phase, distances within layer 1 to n-1 can be directly called from memory instead of calculation by CPU, thereby notably accelerating the overall query speed.

The second step concentrates multiple queries into one function call, so the parallel computation can be exploited. Overall speed on average can increase compared to multiple single queries because of the pre-calculated distances aforementioned.

The last step further takes advantages of the latest Nvidia graphic card features by compressing data vectors into smaller representation in memory, thereby increasing volume of

parallel computation that are only available in the latest generation of NVidia GPU. The Pascal, Turing, and Ampere architectures of Nvidia GPU have enhanced its ability in 16-bit floating point calculation and therefore vastly increases its ability in parallel computation.

3.3 Details on Three Proposed Improvements

The HNSW algorithm applies the data structure of skip list, in which the tree structure can be traversed in very quick fashion due to it layered structure, however, inside each layer, the naive L2 distances are calculated to retrieve the closest node inside each layer. According to the exponential mechanism of determination of the depth of the layer, the size of nodes within each layer increases from top to bottom. The top layer is defined to be the entrance layer, in which it stores the entrance node for the first comparison of incoming new node. The bottom layer contains all the feature vectors by design. The calculation within each layer is actually directly proportional to the number of calculation happened during the query phase, as it is shown in Table 3.3. Table 3.3 shows the node distributions per layer for a database of 23,326 vectors. Clearly, the number of nodes increases exponentially as one traverse from top layer to layer 0. A straightforward way to compress the time for single calculation between two nodes is to pre-calculate the distances within layers that have most nodes except the bottom layer, due to its capacity of holding all feature vectors within the tree. By storing the calculation results inside GPU memory, the actual memory caused by such additional step is minimized and the computational power of GPU is fully utilized with simple kernels [24].

Table 3.1. Relationship Between Number of Calculation and Node Number

Number of Calculation Per Layer	Layer Node Count
4	6
45	456
735	23326

Batch query is hard to implement due to the limited computation power of CPU. Even

Algorithm 1. Parallel Computation Improvement

```
1:  procedure INDEX CONSTRUCTION(incoming feature vectors)
2:      while remaining nodes > 0 do                          ▷ Index Construction
3:          a = incoming new node
4:          Exponential Random Function Determines Level
5:          Start With Entry Node, N
6:          Closest Distance d = ∞
7:          while Nearest Neighbor not Found || ef < ef_set do
8:              n = N's Nearest Neighbor
9:              Calculate Distance dist between n Nearest Neighbor and a
10:             if dist ≥ d then
11:                 n = N's next nearest neighbor
12:                 continue
13:             else
14:                 dist ← d
15:         Go to Next Level with Nearest Neighbor Node
16:     Start With Entry Node N_query                         ▷ Start Query
17:     while Layer ≠ 0 do
18:         if Layer > 1 then
19:             Start Mass Computation of All Nodes in the Layer
20:             Generalize the results
21:             Find Nearest Neighbor b
22:             N_query ← b
23:     Greedy Search For Nearest Neighbor at Level 0
24:     Return Candidates
```

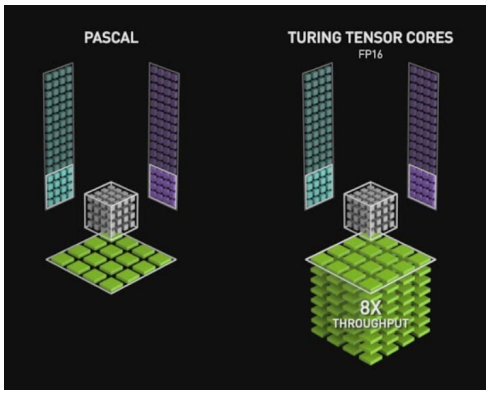

Figure 3.3. New Tensor Cores For FP16 [8]

though different queries can be distributed into different threads, the actual computational power is limited by the number of physical threads within the hardware system. The latest tenth generation of Intel Core family of GPU provides up to 20 threads with 10 physical cores, but in most personal computers, such high number of physical cores and threads is not easily available. With high number of physical tensors inside GPU, however, the batch query can be distributed into thousands of threads and blocks inside GPU and the entire layer of closest nodes can be calculated and returned instantly. Given such advantages and previous mentioned distances stored within each nodes, the batch query can be now realized without overwhelming the CPU.

The new release of the Pascal architecture of Nvidia GPU not only greatly enhanced its ability in machine learning, but it is the first time that native half precision is supported within tensor cores (also known as CUDA cores). Before this new generation, the half precision calculation is significantly slower than double precision due to the fact that the atomic structure of graphic processing unit is designed to be 32 bit instead of 16 bit. Therefore, compressing feature vectors into 16 bits is deemed to save storage space. But in fact the speed of half-precision calculation is also cut down to half in all existing previous generations [31]. Accordingly the application of half-precision cannot be popularized since the loss in speed outweighs the need for reduction of storage space. Until new generation of CUDA introduces the fusion calculation inside tensors cores, by concatenating matrices of half-precision into double-precision, the speed of calculation finally reach to the ideal level of cutting down half of the data. This fusion of

calculation process is illustrated in Figure 3.3. According to official release of the hardware information, the FP16 can be at most twice as fast as the FP32 in different application. Hence, applying such hardware will continue to benefit overall performance of using HNSW algorithm in general.

3.4 Results

Testing Hardware Environment

1. **CPU**: Intel Core i7-8750H, 6 cores 12 threads, 3.0 - 4.0 GHz
2. **Memory**: SK Hynix 16 GB DDR4, 2666 MT/s
3. **GPU**: NVidia RTX 2080, with 8GB GDDR5 Memory, TDP 80-90 Watt
4. **Hard Drive**: Plextor M9Pe SSD, 1TB

Testing Software Environment

1. **Operating System**: Ubuntu 18.04 LTS
2. **C++ Standard**: C++11
3. **GCC Version**: 7.5.0
4. **CUDA Version**: 10.1.243
5. **NVidia Driver Version**: 450.80

Based on the inter-layer results tested from 50,000 human faces data, the intra-layer calculation has certain impact on index construction phase, however, the faster query phase has compensated the disadvantages brought by prolonged construction phase. The memory consumption remains unchanged since most calculation are transferred from CPU to GPU, and therefore the data calculation and occupied memory are mainly stored in the GPU memory.

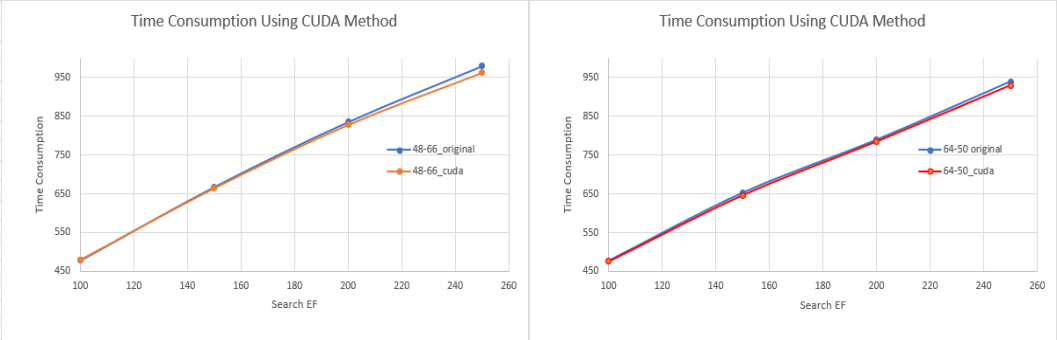

Figure 3.4. CUDA Inter Layer Calculation Results: : Time (seconds) against ef_{SEARCH}

Figre 3.4 shows that the relationship between time and search ef parameters in two different hyperparamters. The search ef is fixed to set of [100, 150, 200, 250], and the two hyperparameters sets are [48,66] and [64,50]. As shown in Figure 3.4, the search time increases as the search ef increases. This is reasonable since more refined searches requires more comparison during the execution. However, the CUDA implementation will then save unnecessary computations in intermediates layers, so the total time of query phase has overall 2% to 7% increase in speed while the CPU memory roughly stays the same in Table 3.2. Table 3.2 refers to memory usages during the original method and CUDA method. Looking at the same hyperparameters settings, the memory consumption is within error range and therefore no evidence of increase in memory consumption is shown.

Table 3.2. CPU Memory Consumption

Hyperparameter Settings	Original (MB)	CUDA (MB)
24-133	432	431
32-100	445	445
48-66	480	481
64-50	487	489

Reasons to such improvement is that 1. extra computation is transferred to GPU without loading extra tasks onto the CPU, and 2. the results of the calculated distances are stored only in GPU memory without consuming memory. By saving calculations both on construction and query phase, the GPU is now functional and can be helpful to store intermediate results within

specific layers. By not storing the original feature vector information, GPU only saves index information and distance information inside a node, which is determined by hyperparameter M. As long as the M is set within a reasonable range, for example the pre-set number 48 or 64 tested in the tuning session, the GPU memory consumption is limited, and usually consumes less storage compared to memory, as Table 3.5 shows. Therefore, it is safe to suggest users that have available GPU unit with appropriate driver and CUDA version to try this feature. Docker images provided by Nvidia can usually help users build up the CUDA environment needed for this implementation.

With the parallel computation enabled, the batch query is realized as following: between layer 1 and n-1, the greedy tasks are transferred into a N to M computation of nearest neighbor that can be completely finished on GPU. N refers to the size of multiple incoming query vectors while M refers to number of node in a specific layer. When all incoming nodes have found their nearest neighbor in layer 1, those nodes are the entry nodes to ground layer. Next, the simple multi-thread initiation will do the greedy part in ground 0. Since all queries are independent with each other and the process does not required writing to a common data structure, there is no race condition. Such performance gain comes directly from the rid of intermediate level greedy computations, the GPU manages all intermediate layer calculation. The benefits can be seen when batch size exceeds 10. The increase in speed outweighs the extra time of data transfer between CPU and GPU. As Figure 3.5 shows the relationship between time and search ef parameters in two different hyperparamters. The search ef is fixed to set of [100, 150, 200, 250], and the two hyperparameters sets are [48,66] and [64,50]. Batch size is fixed as 20 for illustration purposes, other batch size results can be seen in Table 3.3. One can see that as the number of batch size increases, the time saving gain increases, but the increase is always less than the multiplications of the batch size, indicating a success in batch query implementation with parallel computation. The batch query will generally saves 15% to 20% percent of total time consumption without significant impact on algorithm accuracy as Table 3.3 has shown.

To make a even further attempt, we only calculate the nearest neighbor in layer 1 and

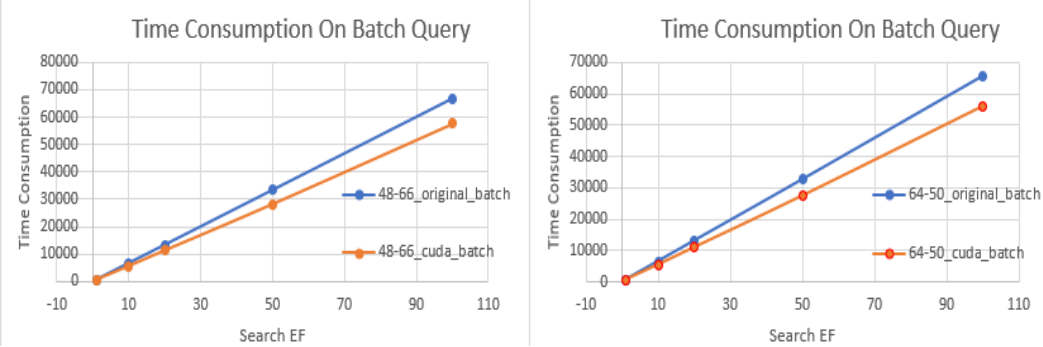

Figure 3.5. CUDA Batch Query Results: Time (seconds) against ef_{SEARCH}

Table 3.3. Batch Query Results, Measured in Multiplication of Size = 1

Batch Size (in multiplication of size = 1)	Time Consumption in Total (in multiplication of size = 1)
10 ×	8.45 ×
20 ×	17.02 ×
50 ×	42.63 ×
100 ×	86.75 ×

proceed to layer 0 using the answer as entry node. Skipping previous layers' calculations will not yield observable difference if the datasets are small, due to the fact that layer 1 to n-1 are filled with less than thousands of nodes. Because queries on large dataset are time-consuming, we only tested on batch size 25 on 500,000 vectors and about 2.45% of time is saved, when we skipped the previous layers' calculation. In terms of logic, skipping the previous layers but still finding the nearest neighbor in layer 1 is almost equivalent to the original greedy method, but sometimes the new attempt can have fluctuations in results because original greedy method does not guarantee nearest neighbor. The precision on the same run shows on average a ± 0.65 % of fluctuation, which in many cases acceptable. Even without this further attempt, users of HNSW now has certain degrees of freedom to choose whether to process query one by one instantly, or accumulating queries into batch and then exploit the power of GPU.

Compared to previous generation of graphic cards, RTX 2080 has shown a significant boost in FP 16 calculation in official documents provided by Nvidia. This claim is consistent

with results recorded in Table 3.4. Parameters in the table include $ef_{\_search}$ = 150 as the searching parameter while the construction parameters are M=48 and ef = 64. The results were recorded on multiple datasets but for simplicity issue, one of the comparison results is shown in Table 3.4. In most cases, on a Nvidia graphic cards that are after Turing architecture, the time consumption gained by compressing the original data into FP16 inside CUDA calculation will generate around 34% deficit of total time while basically maintaining same accuracy and precision, because most feature data would not require very high precision that extends to usage of all 32 bits. At least in most human face feature vectors, such strategy can benefit users of matching feature vectors.

Table 3.4. FP 16 Comparison using 150 ef for search on 48-66 preset

Hyperparameter Settings	FP32 (s)	FP16 (s)
48-66	477.34	311.12
64-50	476.23	320.34

Table 3.5. GPU Memory Consumption on Different Number of Vectors and Hyperparameters

Hyperparameter Settings	23,326 vectors	500,000 vectors
48-66	34.32 MB	650.65 MB
64-50	37.89 MB	709.94 MB

Chapter 4

Conclusions

This research focuses mainly on one of the best similarity matching algorithm – Hierarchical Navigable Small World algorithm, stemmed from tree-based algorithm navigable small world algorithm. Due to its skip list like structure, traversal of the entire graph is limited to sequential traversal between each layer of the entire structure. The interwoven relationship when every new node is inserted into the graph upon index construction phase is recorded inside each node. The nearest neighbor information has notably reduced the distance calculation between every existing node. As a consequence of its design, the number of total calculation is limited to a very small amount that the query phase can be finished within a thousandth of the number of total datum point stored, but at the meantime, precision and recall rates are not impaired. With simple implementation exploiting the built-in instruction set from Intel, the CPU based HNSW can handle up to millions of data point, therefore popularizing the algorithm.

Consistent improvement of a popular and outstanding algorithm is one benign process of scientific research. Researchers have spent efforts in determining a suitable hyperparameters on the face data. Starting from the appropriate set of parameters, the parallel computation enhancement can be further tested at optimal condition. Based on high throughput provided by the graphic processing units with its designed purpose, the intra-layer distances can be calculated through thousands of tensor cores after index construction is finished, thereby accelerating the query phase calculations of distance leading to the identification of the nearest nodes within

the same layer. In addition, the unimplemented batch query is now realized through GPU acceleration in that multiple-core calculation can be distributed to different tensor cores inside GPU, hence greatly increasing the overall speed of multiple queries by at least 2% to 7%.

Given the results of all those possible enhancement on matching algorithm on GPU, the GPU load distribution, multiple query implementation becomes more available for users that possess extra hardware such as Nvidia graphic cards. The multiple queries even gain extra benefits from the parallel computation by 15% to 20%. One additional study includes exploitation of FP16 in the matching algorithm so that data can be compressed in GPU with faster calculation. Consequently, latest generations of graphic cards can now utilize new hardware and software features that will further improve algorithm performance by at least 30%.

Future improvement on HNSW algorithm has already generated interests from other researchers groups. Some of the researchers have already shown greater improvement on the original simple implementation. For example, researchers have expanded hierarchical structures to pyramid like multi-trees to reduce even more unnecessary calculation and distribute such calculation in different hardware [32]. As the era of big data rises, more exploration would be soon made in the near future.

Bibliography

[1] J. BRADLEY, Y. NI, and K. CHU, "Detecting abuse at scale:Locality sensitive hashing at uber engineering," *databricks. com/blog/2017/05/09/detecting-abuse-scalelocalitysensitive-hashing-uber-engineering. html.[Online*, 2017.

[2] T. A. Cui, "The curse of dimensionality," *Cofactor Genomics.https://cofactorgenomics.com/curse-of-dimensionality-wk-16/.[Online*, 2017.

[3] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[4] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[5] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[6] M. Aumuller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool¨ for approximate nearest neighbor algorithms," in *International Conference on Similarity Search and Applications*, pp. 34–49, Springer, 2017.

[7] "Thread block (cuda programming)," *Wikiwand. wiki-wand.com/en/Thread block (CUDA programming)*, 2016.

[8] R. Farber, *CUDA application design and development*. Elsevier, 2011.

[9] P.-C. Lin and W.-L. Zhao, "A comparative study on hierarchical navigable small world graphs," *arXiv preprint arXiv:1904.02077*, 2019.

[10] G. H. Gonnet, M. A. Cohen, and S. A. Benner, "Exhaustive matching of the entire protein sequence database," *Science*, vol. 256, no. 5062, pp. 1443–1445, 1992.

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, pp. 3111–3119, 2013.

[13] A. Gersho and V. Cuperman, "Vector quantization: A pattern-matching technique for speech coding," *IEEE Communications Magazine*, vol. 21, no. 9, pp. 15–21, 1983.

[14] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with gpus,"′ *arXiv preprint arXiv:1702.08734*, 2017.

[15] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[16] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[17] P. M. Gionis, A.; Indyk, "Similarity search in high dimensions via hashing," *Proceedings of the 25th Very Large Database (VLDB) Conference.*, 1999.

[18] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pp. 7–13, IEEE, 2013.

[19] M. Charikar and P. Siminelakis, "Hashing-based-estimators for kernel density in high dimensions," in *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1032–1043, IEEE, 2017.

[20] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate nearest neighbor search in high dimensions," *arXiv preprint arXiv:1806.09823*, vol. 7, 2018.

[21] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu, "Fast knn graph construction with locality sensitive hashing," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 660–674, Springer, 2013.

[22] K. Brinker, F. Moerchen, B. Glomann, and C. Neubauer, "Document clustering that applies a locality sensitive hashing function to a feature vector to obtain a limited set of candidate clusters," Sept. 14 2010. US Patent 7,797,265.

[23] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, 1975.

[24] S. Har-Peled, P. Indyk, and R. Motwani, "Approximate nearest neighbor: Towards removing the curse of dimensionality," *Theory of computing*, vol. 8, no. 1, pp. 321–350, 2012.

[25] P. Cunningham and S. J. Delany, "k-nearest neighbour classifiers," *arXiv preprint arXiv:2004.04523*, 2020.

[26] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[27] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration.," *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.

[28] J. Von Neumann, “First draft of a report on the edvac,” *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[29] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, *DRAM circuit design: fundamental and high-speed topics*, vol. 13. John Wiley & Sons, 2007.

[30] N. L. L. Julia, *The essentials of computer organization and architecture*, vol. 36. Jones &Bartlett Learning, 2010.

[31] E. K. Wittenbrink, Craig M. and A. Prabhu, “Fermi gf100 gpu architecture,” *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[32] K. K. J. C. C. J. Deng S, Yan X, “Pyramid: A general framework for distributed similarity search on large-scale datasets,” *In2019 IEEE International Conference on Big Data*, pp. 1066–1071, 2019.