

Contents

Abstract	3
Objectives	3
Technical Details/ Background	4
Nearest neighbor search in high dimensional spaces	4
LSH algorithm	4
C++	5
Multithread	5
Cluster Computing	5
Message Passing Interface	6
Spark	6
In-Memory storage	7
Memcached	7
Redis	7
Python Wrapper-SWIG	7
Graphics Processing Units	8
General-purpose computing on graphics processing units (GPGPU)	8
SIMD (single instruction, multiple data)	9
Scalability	9
Methodology	12
FastLSH-ESS	13
FastLSH-Lib	13
FastLSH-Python	14
FastLSH-Spark	14
FastLSH-MPI	15
FastLSH-GPU	15
FastLSH-UI	16
FastLSH-VR	16
Discussion and Analysis of results	17
Benchmarking Hardware	17
Experimental Setup	18
Basic performance test	18
Unit Test	18
Cluster & Multithread Test	19

Result & Analysis	20
Basic Performance Test	20
Unit Test	20
Cluster & Multithread Test	22
FastLSH UI & FastLSH VR	23
FastLSH UI	23
FastLSH VR	25
Conclusion	27
Future Work / Development	27
System Core with lower level control	27
Stronger cross platform ability	27
Fault tolerance	28
Batching system	28
Real time visualization with VR	28
Reference	29

Abstract

Similarity Search is an important component in Artificial Intelligence field. While existing solutions are usually problem specific, we proposed a flexible and high performance similarity Search Engine on High Dimensional Big data. The system can complete a query with one million base set in less than a hundred milliseconds. The system aims at providing a fast, flexible, scalable and extensible solution for similarity search by its eight different packages. By providing a wide range of interfaces, user may pick the one best suits their own needs. While a VR implementation has also been included, to serve as promotional and demo purpose.

Objectives

Similarity Search is an important component of Artificial Intelligence field. Imagine that you need to find similar pictures from a huge picture pool, how to realize it in both fast and accurate manner is a huge challenge.

Performing operation on dataset lying on high dimensional space has another obstacle which is referred as "Curse of dimensionality"[23]. As different points' relative contrast of distances will decrease as dimension increases. This makes similarity search problem on high dimensional space even harder.

Many researchers have brought effective algorithms to tackle with the problem. Locality Sensitive Hashing (LSH) is one of them which brings approximate result for similarity search. However, like most of the laboratory researches, the available systems are specifically tuned for certain machines to deal with specific problems. There is no general implementation that is extensible and flexible for various problems. Furthermore, such implementations are often designed to run on high-end clusters, which make researchers and students harder to make use of existing solution.

Our goal is to build a similarity search engine which can easily fit into different models and boost the performance of the whole process. Fast, Flexible, Scalable and Extensible is the four main features of the system.

On the other hand, visualization of big data is also another objective of our project. Our frontend provides an extensive control of the similarity search system while allowing professional user to visualize their data set before running similarity search. Meanwhile it serves as a unique feature to assist data analysis. This would greatly simplify researchers work flow.

In addition, another common phenomenal that there is often a huge gap between Computer Science researcher and normal non-technical people. Our system also provides another portal with Virtual Reality technology to visualize high dimensional data and demonstrate the algorithm in an interesting and attractive manner. With such approach, we narrow the technical gap and attract non-technical people to have a glimpse of one of the most important algorithm in big data field.

Technical Details/ Background

Nearest neighbor search in high dimensional spaces

Data spreading out in High-Dimensional space can easily and often be encountered in a lot of areas, i.e. feature extraction from a text documents in [17], as well as in medical or biological research. Doing nearest neighbor search on high dimensional space requires a massive amount of computational time. While the "curse of dimensionality"[23] has not been completely solved even after decades of effort since discovery of the problem in 1960s by [24, pp. 222-225]. Besides in [19], result with sets of experiments, has shown that different points' relative contrast of distances will decreases as dimension increases. This exploration of the effect of dimensionality on nearest neighbor problem has been a topic in the field. While studies [20], [21] also show that the traditional indexing method of nearest neighbor search in high dimensional space has been failed and can easily be beaten by sequential scan [22].

LSH algorithm

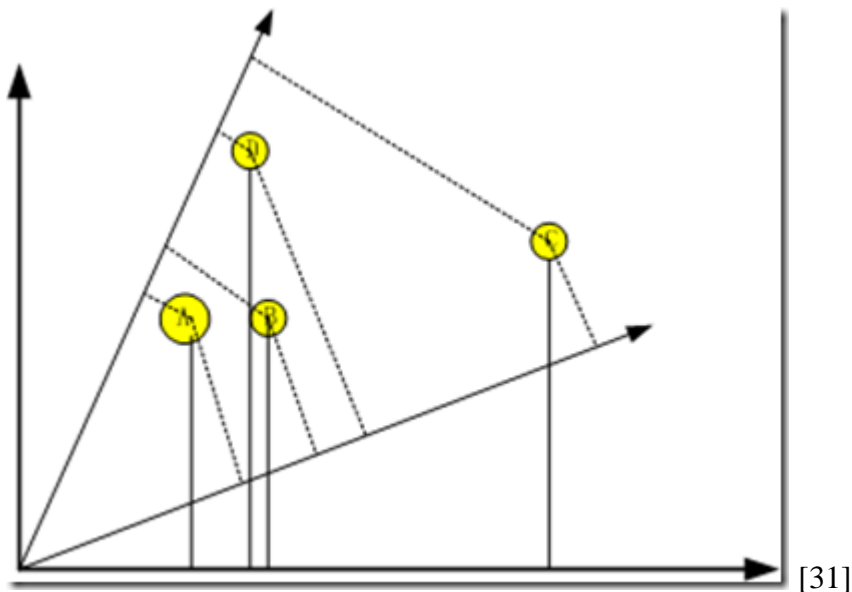
While performing nearest neighbor search can be computational intensive and time consuming especially in high dimensional spaces. Numerous of methods of approximate nearest neighbor search have been proposed aiming to eliminate the curse of dimensionality, for example Locality sensitive hashing, Best bin first, etc. Locality sensitive hashing, commonly known as LSH, was introduced in 1998. [25] It is a commonly used approximate nearest neighbor search algorithm as it proves query in sub-linear time complexity. There are a lot of methods being proposed to perform LSH, i.e. MinHash, one of the famous solution, SimHash, based on random projection [26]. While the implementation of this project is based on a method of the LSH family, Collision Counting LSH, C2LSH, that is published by Gan et al. in 2012 [10]. This scheme construct a hash function that has dynamic compound instead of the traditional approach of generating a static one. It implements the idea of defining a pre-specified collision threshold to evaluate if a data object (or sometimes being referred as vector) is a candidate. [10] This proposed scheme has been found to be suitable in perform an estimation on the similarity between two points as well as providing range search for a query. [17] This scheme has also proved to be guaranteeing on the query quality especially in high dimensional space.

LSH Definition *A locality sensitive hash (LSH) is a hash family where similar items are more likely to collide. Formally, a hash family $\mathcal{H} = \{h : \mathcal{U} \rightarrow S\}$ is called (r_1, r_2, p_1, p_2) -locally sensitive if for all points $p, p' \in \mathcal{U}$,*

1. *if $d(p, p') \leq r_1$, then $\mathbb{P}[h(p) = h(p')] \geq p_1$.*
2. *if $d(p, p') > r_2$, then $\mathbb{P}[h(p) = h(p')] \leq p_2$.*

Note that this definition only makes sense if $r_1 < r_2$ and $p_1 > p_2$.

[31]



C++

C++ is chosen to be as our implementation language. C++ is one of the fastest languages, several magnitudes faster than popular scientific computing language Python, and it provides us low level control of the memory. Both the features allow us make the engine efficient.

Multithread

Nowadays, nearly all central processing unit (CPU) equipped are multi-core processor. It enables multiple processes or threads to run concurrently. However, without explicitly tuning the implementation, the operation will be processed by a single core. When the large data set is being used, the system will occupy large portion of memory while most of the computing power remain idle. For example, on commodity machine equipped with an eight-core CPU, very often, only one eighth of the computing power is used if the program is written in single thread computation style.

FastLSH make use of three technology namely OpenMP, stdThread and Pthread to empower the system with multithreaded computation ability. While their performances are similar, certain difference still exists. Their features will be discussed below.

Cluster Computing

While the multi thread technology enables the algorithm to run in parallel style making full usage of the computing power on a single machine. Memory limitation remains a problem when large data set comes. Intuitively, putting more memory into a single machine to "scale up" the machine is the way to

solve the problem. However, when the memory on a single machine stack up, the cost will become the dominant issue and it is not an ideal strategy for an ordinary experiment setting. Another strategy is to "scale out", linking multiple machines together and use the network communication to make use of all their memories. In addition, their computing power can also be used in the "bigger system". The distribution of work among cluster is basically similar to multi thread approach with additional communication management.

FastLSH uses Message Passing Interface and the distributed engine Spark to realize cluster computing versions. Together with the multi thread technology which is deployed on each node, cluster computing assemble the most powerful version of FastLSH.

Message Passing Interface

Message Passing Interface (MPI) is a standardized and portable message passing system designed for parallel computing architecture. It provides a communication protocol for parallel computing. By using this system, FastLSH realized the specially tuned cluster solution. To use the MPI, a shared directory needs to be allocated among the cluster and the program needs to be run inside that directory. Each node will run exactly the same program. During the run time, different nodes will be assigned a "world rank". The program will run different portion of code depends on the specific world rank assigned. Because of that, different nodes are able to have different behaviors in the cluster. In the LSH algorithm, some parts must be computed in one node while some can be paralleled. So, we used a master-slave structure. Master node will do the non-parallel task, which includes broadcasting public variables, sending out portion of data and gathering the result, and also get involved in computing the parallel task. Each slave node will only perform the parallel task with the data portion assigned by the master node. At first glance, there seems to be unbalance of computing load between master and slave node. However, all the heavy load tasks are parallel tasks while non-parallel task's computing load is trivial. The unbalance would not reduce the cluster computation performance.

Spark

Spark is a lightning-fast cluster computing engine. Spark is a handy framework for distributed system, which results in user not having explicitly control on network communication. Although it is fast and has been optimized in various aspects, the Spark implementation performance is not as good as the MPI version. After analysis, there are several reasons causing that. Firstly, making use of the resilient distributed dataset (RDD), the operation has to be written in mapReduce programming model. The FastLSH can be transformed into mapreduce

version while maintain the accurate result. However, not all of the operation needs to be conducted in a distributed style, some only needed to be conducted in the single node, so Spark implementation may cause an additional time cost.

In-Memory storage

Memcached

Memcached is a general-purpose memory caching system. It is an open-source software. Memcached is relatively light weight and it supports simple key value style storage as well as storage across a cluster of machines. It provides a comprehensive C++ API, as long as the machine address and port are provided, the system can access the memcached.

Redis

The Redis is also an open source data structure store. While providing similar functionality as Memcached, the Redis is slightly heavier while providing more features. For example, it provides more data structures like strings, lists and sets. It also has persistence storage. It will automatically keep a copy of value in the disk on a regular basis. When the system gets reboot, it will load the backup into memory. So, the in-memory persistence storage is realized. Also, the C++ API of Redis is used in the system to provides access.

Python Wrapper-SWIG

While the fast language C++ is used in the core parts which largely improves the performance over several orders of magnitude over script language, C++ is not an ideal language for rapid development and interactive programming which is heavily practiced in data science research field. C++ interface is reserve for data engineers, allowing them to extend the system or plug in the system into an existing one. Meanwhile, the details are not needed to be exposed to data scientists who only needs to know the higher-level interface of the system. Python is a widely used high level programming language for scientific programming. It is used extensively in data science field for its widely supported library, high readability and simple syntax. It is ideal to build the higher-level interface. However, Python, as a script language, is several orders of magnitude slower than low level language like C++. To maintain the high performance achieved by C++, python will only be the wrapper of the core functionality. This is the time SWIG comes into stage, Swig stands for Simplified Wrapper and Interface Generator. It is an open-source software tool used to connect C language to scripting languages and many others. While there are a lot of needs to be considered when configuring the wrapping files, the whole process is intuitively simple. After writing the interface, the SWIG can generate the wrapper scripts. With the wrapper, the engine can be imported into any python program.

Graphics Processing Units

General-purpose computing on graphics processing units (GPGPU)

Graphics Processing Units was originally designed to process high-speed image rendering tasks, the current trend of scientific calculation on GPU was only introduced in the first few years of the century. The background information on GPGPU will be discussed in this section.

General-purpose computing on graphics processing units, GPGPU, is a chapter of parallel computing. As GPU's architecture and its computation pipeline were found to fit scientific calculation involving continuous looping task. The first implementation of the idea of GPGPU was the matrix multiplication task in 2001[9]. However, it was not productively applicable due to its calculation preciseness as floating-point support was only introduced after two years in 2003[13]. Then in 2005, the LU factorization was one of the first common implementation of GPGPU application that has a faster performance comparing with an optimized CPU implementation. [15] Since then migrating computation process from CPU to GPU has become a trend because of the performance boost shown in previously published researches and implementations. However, quite a lot of efforts were required to run any kernel program on GPUs before the introduction of high level programming model and computation platform including CUDA and OpenCL. The NVIDIA proprietary programming model, Compute Unified Device Architecture, CUDA, was launched in 2006. It was the pilot of the field. This programming model allows programmers and researchers to use C or C++ language as an interface to interact with NVIDIA GPU. With CUDA, program can be written relatively easy to make use of GPU's computing power and parallelism without knowing the details within the GPU hardware.

For the GPGPU pipeline in traditional CUDA programming implementation, in high level, has a computation cycle that includes two sets of code and the programming flow generally can be divided into five steps.

Two sets of code include, the host code which execute on CPU and the kernel/device code which is usually executed by different trends simultaneously in GPUs.

The five steps include,

- 1) Memory declaration and allocation on host and device (host code)
- 2) initialization of data on host (host code)
- 3) Data transfer from host's memory to device's memory (host code)
- 4) Execution of task (kernel/device code)

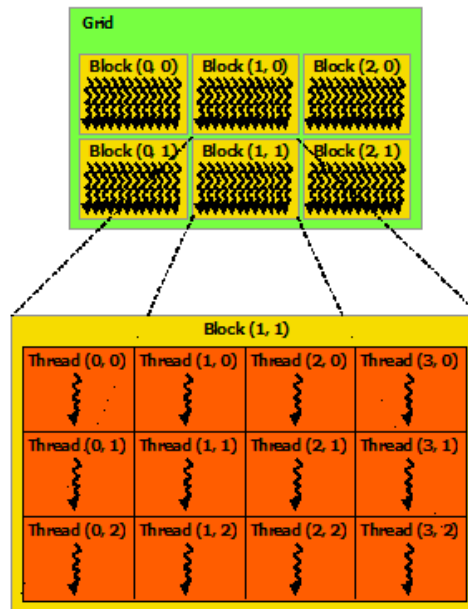
5) Result being copied to host from device (host code)

SIMD (single instruction, multiple data)

Single instruction, multiple data (SIMD) is how GPU processor operate. This is the main reason why GPU is suitable for our FastLSH CUDA implementation as well as why we get the performance boost in our implementation. There are many SIMD multiprocessors (known as Streaming Multiprocessor on our testing device) within a GPU. SIMD is referring to those processors (CUDA cores) within the multiprocessor that is executing the single instruction with different data at any given computation cycle. Even on commodity GPUs, there are over hundreds of CUDA core in one single card, this makes tasks that can be done in parallel way faster than it is being compute on CPU. However, SIMD also comes with other constraints in dealing with other daily operations within programs, such as the entire unit will be stalled by a memory access of a single thread. So, it is important to foresee and cope with such problem during the program design stage. Due to this design, as suggested by [1], some LSH algorithm such as the LSH-forest and PCA, are both not suitable for GPU implementation. Besides, it is also proven that the multi-probe technique is not suitable for GPU due to its imbalance of size in the intermediate data structure, as it will stall the overall performance of the computation.

Scalability

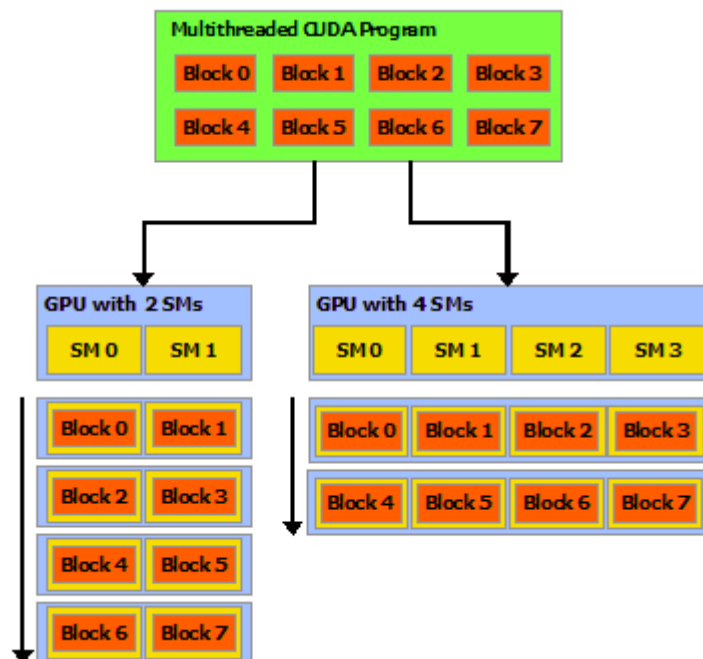
Automated scalability is one of the advantages of CUDA, the computation platform we used. This scalable programming model allows us to scale the implementation accordingly simply by initiating the number of threads in each thread block and number of thread blocks in the Grid of Thread Blocks. The below figure provided by NVIDIA has illustrated the idea of threads, thread blocks and Grid of Thread Blocks clearly.



(pic. 1) Grid of thread blocks

Using compute capability 6.1 hardware as an example [16]

On each GPU hardware, the number of threads allowed in each thread block is 1024, while the maximum number of blocks allowed (allows 3-dimensional structure, illustrated as x, y and z) is $2147483647(x) * 65535(y) * 65535(z) = 9.2230906e+18$, which is hardly reachable in one single kernel call, as the global memory on the device would become the constraint. The below figure provided by NVIDIA can illustrate the automated scalability on different GPU hardware. [16]

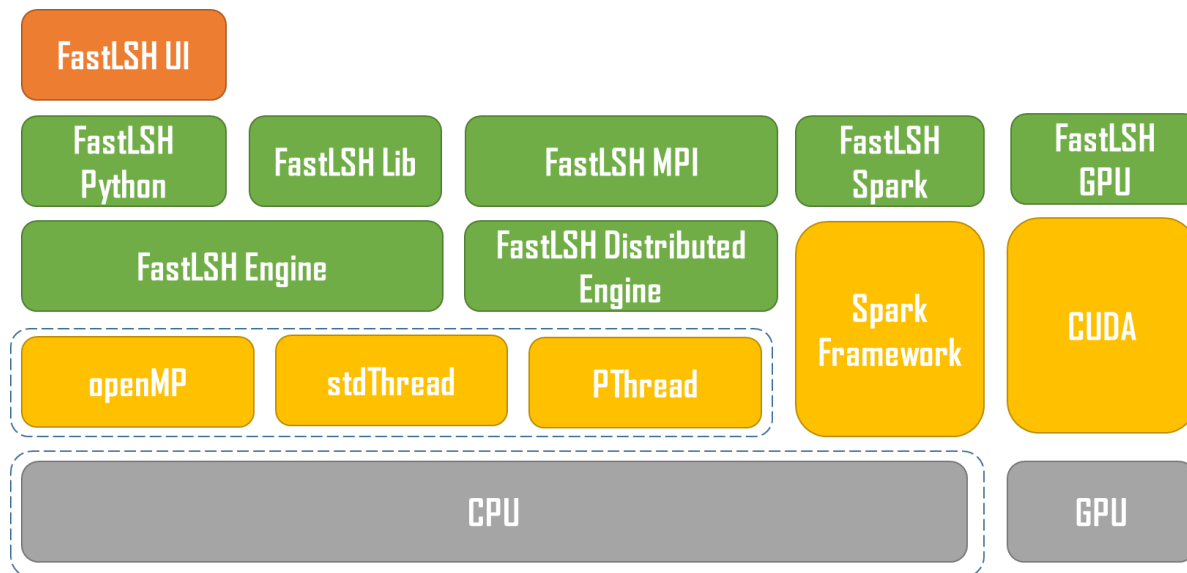


(pic. 2) Automatic Scalability

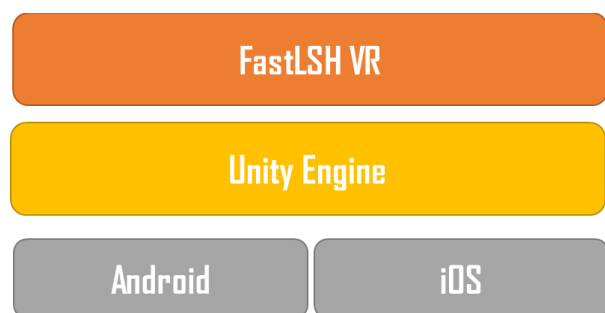
Methodology

Our big data similarity search explorer allows user to perform similarity search in many ways, including accessing it with web UI (FastLSH UI), using it as a C++/Python library (FastLSH Lib /FastLSH Python), entering Scala command (FastLSH Spark) as well as running it as a program on cluster (FastLSH MPI).

The graph below shows the stack of our system:



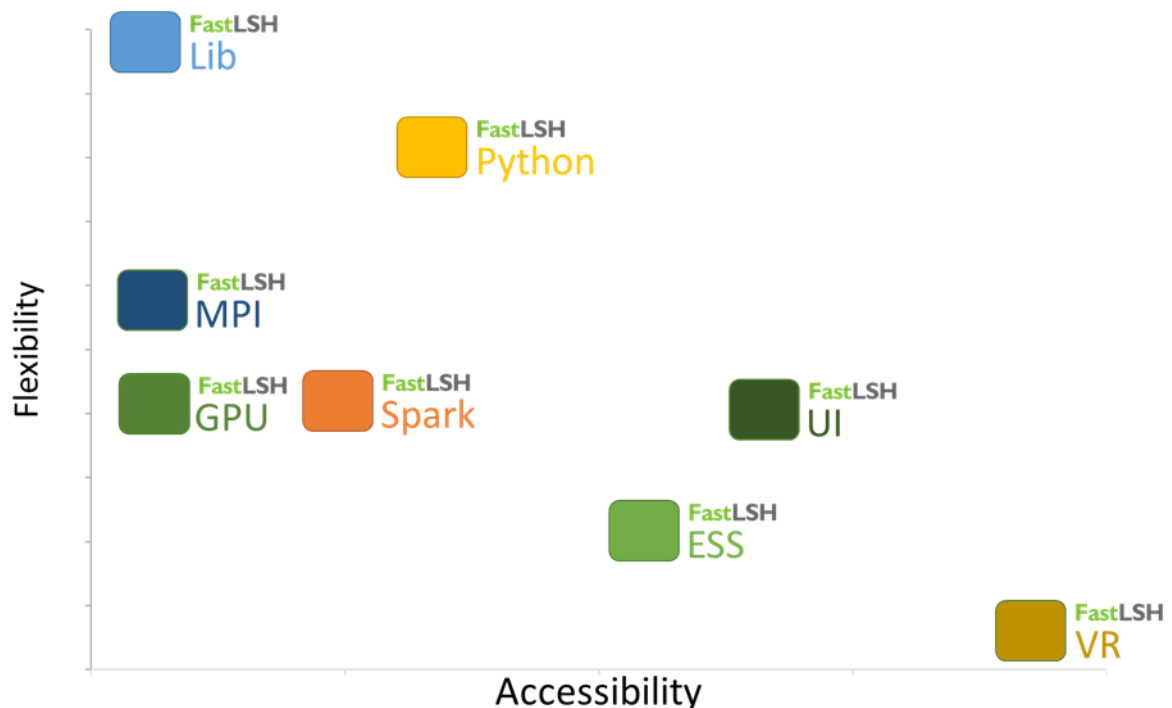
Algorithms are hard to imagine and math formulas are obscure. Professionals always find it hard to explain them. This gap also stands in the way of stopping young fellows to get in touch with technical detail. Our similarity search explorer provides a virtual reality implementation allowing user to experience how does our FastLSH engine operate. The graph below is our VR implementation stack:



As indicated, the FastLSH UI and FastLSH VR provides a front end for user to access our system with user-friendly interfaces. While the other part of the system provides a professional fast, flexible, scalable and extensible solution for professional user. Besides, our implementations and user guideline are well documented. Users may easily gain hands-on experience with any package provided in the system.

The following section, each package in our system will be listed and each of their accessibility level and flexibility level will be discussed.

The following graph is the overview of all packages:



FastLSH-ESS

Accessibility: ****

Flexibility: *

FastLSH-ESS is the basic interface of FastLSH. ESS stands for essential. It is an instant application.

With a simple click, the whole process can be executed.

However, FastLSH-ESS is not as flexible as another interface. The file can only be read from and write to the disk with a settled execution flow. It is also not easy to directly plug into large project since the input and output are going through the disk which is not ideal for high performance computing. As the name, essential indicates, it only performs the essential workflow of FastLSH to help user to locate highly similar points.

FastLSH-Lib

Accessibility: *

Flexibility: *****

FastLSH-Lib is a C++ library. It is importable to other C++ program. Among all the interfaces, the FastLSH-Lib provides the highest flexibility. User can include the library in their projects, and use all the public functions of FastLSH. It makes the FastLSH engine deeply and seamlessly integrated into the existing project. Moreover, since all the source code is open to the user, user can dive into the FastLSH engine, make changes per their needs or even tear it apart and redesign the workflow and the algorithm. However, high flexibility comes with low accessibility, the technical bar is relatively high. To use the C++ library, the user needs to be able to program in C++. To use the public functions, the user needs to read through the documentation. Although the code follows high software engineering standard and has been well documented, to modify the system or to redesign the algorithm, user needs to have deep understanding of them. It requires no little time to learn. FastLSH-Lib is made for development providing highest flexibility to let user benefit most from the engine.

FastLSH-Python

Accessibility: ***

Flexibility: *****

FastLSH-Python is a Python module. It can be imported to python script. Being a high-level language, it has the feature of both high accessibility and flexibility. The core of the module remains the C++ implementation. Python is only the wrapper of the system. It serves as a bridge between lower level and higher level language. All the public functions in FastLSH-Lib have corresponding bridging method in python module, so the python module have full access to the core engine. Benefitted from the wide usage of python in scientific computing world, the FastLSH can be easily plug into tons of existing projects. Though python is a relative easy language, the user still needs certain level of programming skill and understanding the behavior of FastLSH public functions. The source code of the engine is not open to the user, so user is not able to change the workflow or the specific implementation. FastLSH-Python intends to provide high level interface to serve more data science projects.

FastLSH-Spark

Accessibility: **

Flexibility: **

FastLSH-Spark is a Spark implementation of the LSH. It is a standalone program. Scala is used to implement the program. Functional programming style is adopted so the code is concise and clean. When use the FastLSH-Spark, user needs to have the Spark ready in the cluster. The access to Spark makes the FastLSH easy scalable. Certain programming skill is still required to change parameter setting. Since the source code is open, user can change and modify the workflow or the algorithm. It can also be plugged into the existing spark project. However, the understanding of scala, Spark and the LSH is still required if user want to make deep change. The technical bar is quite high which reduces the accessibility. FastLSH-Spark intends to make

FastLSH accessible to popular distributed data processing engine to increase its scalability.

FastLSH-MPI

Accessibility: *

Flexibility: ***

FastLSH-MPI is a cluster computing version of FastLSH. It uses Message Passing Interface(MPI) to realize the communication among the cluster. Because the cluster version needs to break up the algorithm into separate pieces so that the communication can happen between the nodes, the core engine is extensively changed. So it is a standalone package. FastLSH-MPI makes the algorithm scalable among cluster with minimal communication cost. However, before using the FastLSH-MPI, the user needs to build the MPI cluster and make all the configuration ready. To change the behavior of the program, the user needs to have the C++ and MPI programming capability which raise the technical bar. So the accessibility is low. Though all the source code is open to user, if the existing project is not in MPI style, it will be a bit hard to bridge it to the existing project. So it is not very flexible. FastLSH-MPI intends to make the algorithm run in a cluster to provide high scalability and performance.

FastLSH-GPU

Accessibility: *

Flexibility: **

FastLSH-GPU is a CUDA implementation of FastLSH. It is empowered by the SIMD feature of GPU. With hundreds or even thousands of cores in each graphics card, the calculation is being run by thousands of threads and be processed simultaneously. As the CUDA implementation requires allocating different module of the FastLSH into kernel and host code, this implementation is also a standalone program. Its high performance in terms of speed is boosted by the nature of GPU computation, threads generated for processing the problem is decided to grow as the size of the input dataset grows, such that better performance can be achieved

As the code is open for user to edit, user can modify the program to fit their computation needs with prior knowledge of CUDA programming background. As certain programming background is needed for user to modify the program to fit their own needs, the accessibility is low. Meanwhile, the implementation is less flexible than other implementation due to the nature of GPU memory size is lower than host memory which may limit the size of the input data set. However, this can be solved by batching the query set in exchange of performance.

FastLSH-UI

Accessibility: *****

Flexibility: **

FastLSH-UI is a web interface of FastLSH. It provides a graphic interface for the core engine. Django framework is used for the web application. User can use the high-performance engine by the well organized and user friendly web interface. No programming skill is needed for the whole process and user can concentrate on designing their model without being stopped by technical bar. Data visualisation is also provided as a unique feature to assist data analysis for FastLSH. While the high accessibility is provided and user can change all the parameter setting in the UI, it is not easy to bridge it to any existing projects and user is not able to change the Implementation details and the workflow of the algorithm. So, the flexibility is not high. FastLSH-UI intends to provide a user-friendly interface with low technical bar for easy data analytic access.

FastLSH-VR

Accessibility: *****

Flexibility: -

FastLSH-VR is a Virtual Reality demonstration project for promotion and education purpose. It does not run the LSH algorithm but demonstrate the LSH procedure in the 3D space. Pseudo data points will be placed in a 3D coordination system; users may explore the virtual reality world and have an understanding of how the algorithm works. User can move in the virtual environment by simply pressing the button and looking at the direction user wants to move. Animation simulated algorithm will be represented in virtual space to illustrate each step performed by the FastLSH package. Users will feel that they are immersing inside the data. The demonstration is implemented with Unity Engine. The virtual reality is achieved by Google Cardboard, a commodity VR viewer. Users can easily download and install the "Unity Engine" app in their smartphone, the app supports both Android and iOS. By placing their phone inside the cardboard, they can enjoy the LSH algorithm demonstration without hassles. It requires no technical skill or any background knowledge. However, the demonstration is a settled process which cannot be changed. As discussed before, FastLSH-VR does not do actual computing, it is intended to promote and educate people.

Discussion and Analysis of results

In total, three sets of experiments have been take place and each of them are repeated 10 times in order to ensure the reliability of the measurement.

Four sections, namely “Benchmarking Hardware”, “Experimental Setup”, “Results & Analysis” and “FastLSH-UI & FastLSH-VR“, will be discussed in the section. In Benchmarking Hardware section, we will introduce the hardware setups being used for measurement. Secondly, in Experimental Setup section, the experiment design will be listed in details, while reasoning of the design will also be discussed. Meanwhile, in the Results & Analysis section, the figure of the experiment result will be displayed in both text and graphical format, while analysis will also be given. Lastly, in “FastLSH-UI & FastLSH-VR“ section

Benchmarking Hardware

Hardware used for benchmarking can simply divided into two,

The first type is being used in running FastLSH-ESS, FastLSH-Lib and FastLSH Python as well as FastLSH-MPI and FastLSH-Spark

Specification:

CPU: i7 4770 3.40 GHz

Ram: 8GB DDR3

System: Ubuntu 16.04LTS

While FastLSH-MPI and FastLSH-Spark requiring a cluster to operate, we connected them with the MPI implementation, OpenMPI, for FastLSH-MPI and installing Spark framework on each of them for FastLSH-Spark. Meanwhile they are connected with a 100Mbs switch by CAT-5e cable on their gigabyte Ethernet port.

The second type of hardware is used in running the FastLSH-GPU

CPU: i7 6700k 4.00 GHz

Graphic Card: NVIDIA GeForce GTX 1080

Ram: 64GB DDR4

System: Ubuntu 16.04LTS

CUDA toolkit version: 8.0

Remarks: In eightfold volume of ram has tested not having any interference with the experimental result, meanwhile the CPU difference has minimal influence on the result as well.

Experimental Setup

Four sets of experiments have been done, namely, Basic performance test, unit test, memory test and cluster test.

Basic performance test

In the basic performance test, the default set of data with 57 dimensions, while 1,000,000 points base set and 100 points query set are being used. While other values remain as default, number of group hash: 200, number of hash functions in each group hash: 1 and Bucket width: 1.2

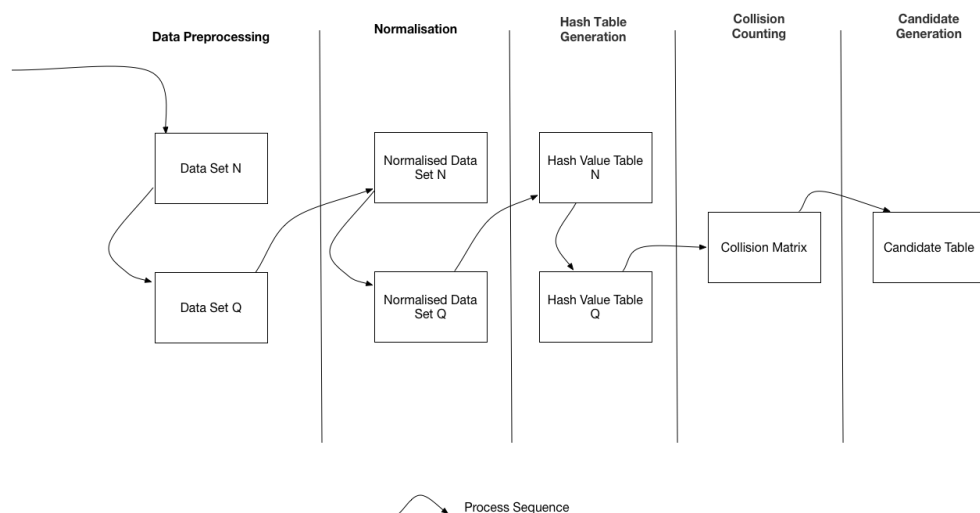
Packages: FastLSH-ESS, FastLSH-Lib, FastLSH-Python, FastLSH-GPU, FastLSH-MPI (3 nodes) and FastLSH-Spark

Unit Test

In the unit test, value and data set used are same as what have been used in the Basic performance test. With dataset with 57 dimensions, 1,000,000 points base set and 100 points query. While the system configurations are: number of group hash: 200, number of hash functions in each group hash: 1 and Bucket width: 1.2.

The unit test is aim to figure out the time needed to run on each unit of the FastLSH package. This could help us calculate the exact query time needed for any input set in the future. Meanwhile, helping us to identify the system performance on each unit, which may need to performance boost in future development.

The following graph is the overview of the system and the unit test will be performed based on these units.



Packages: FastLSH-ESS, FastLSH-Lib, FastLSH-Python, FastLSH-GPU, FastLSH-MPI and FastLSH-Spark

Cluster & Multithread Test







In the Cluster & Multithread Test, value and data set used are same as what have been used in the Basic performance test. With dataset with 57 dimensions, 1,000,000 points base set and 100 points query. While the system configurations are: number of group hash: 200, number of hash functions in each group hash: 1 and Bucket width: 1.2.

Cluster test will be performed by increasing number of threads as well as the number of node. This test can allow us to figure out the performance (how many times speed up can be achieved) when increasing nodes or threads.

Packages: FastLSH-ESS (1-8 threads), and FastLSH-MPI (1-6 nodes)

Result & Analysis

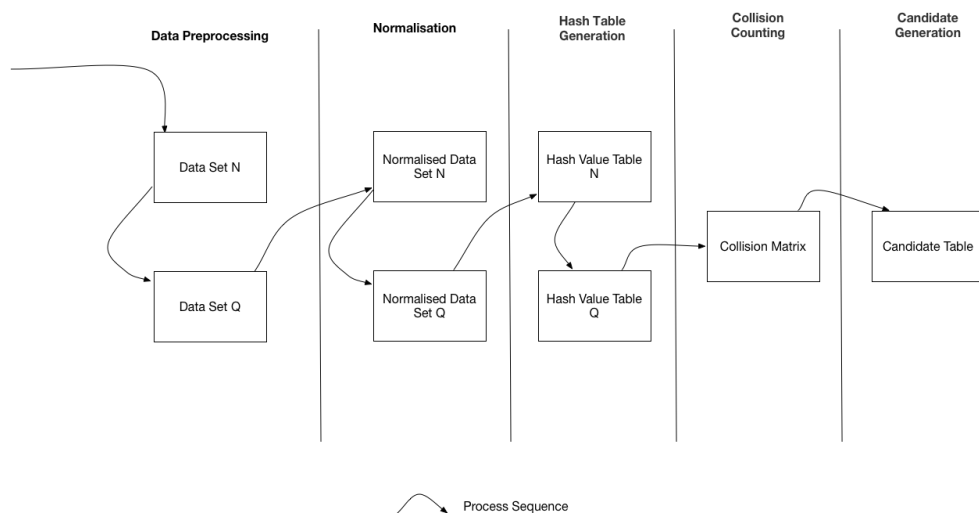
Basic Performance Test







	 FastLSH ESS	 FastLSH Lib	 FastLSH Python	 FastLSH GPU	 FastLSH MPI	 FastLSH Spark
Time(s)	84			38	60	198

In the basic performance test, the result is as shown above. The FastLSH-GPU has the best performance, while FastLSH-MPI is second faster in the test with 3 nodes simultaneously computing. While the FastLSH-ESS, Fast Lib and FastLSH-Python has the exact same result as they are sharing the core engine. It is worth noticing that, user using FastLSH-UI to run similarity search will get the same performance as these three interfaces, since the FastLSH-UI is interacting with the FastLSH-Python interface. Even though the result took more than two times of the FastLSH-GPU package, but by looking at the benchmarking hardware, these three interfaces are operating on a commodity hardware, while the FastLSH-GPU packages is running on a rather high-end hardware. Last but not least, the FastLSH-Spark runs, as expected, as the slowest interface. Its speed can be explained as it is running on top of the Spark framework while only single thread is used for the computation process.

Unit Test

Full process

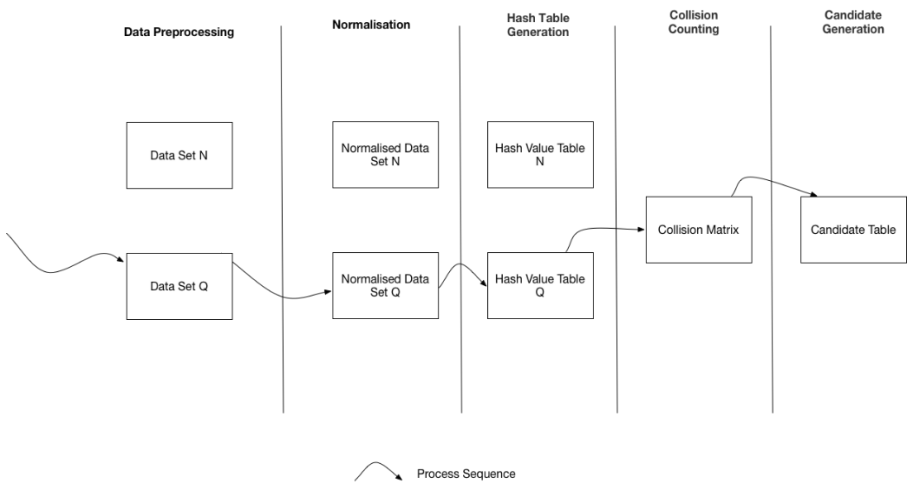








	 FastLSH ESS	 FastLSH Lib	 FastLSH Python	 FastLSH GPU	 FastLSH MPI	 FastLSH Spark
Data Preprocessing & Normalization Time(s)	20			12	23	24
Hash Table Generalization Time(s)	47			19	27	119
Collision Counting Time(s)	17			7	11	54
Candidate Generation Time(s)	0.1			0.1	0.2	0.6

The above table illustrated the time required for our FastLSH system to complete each unit. We can see that among all packages the Hash Table Generation take up the longest time while the time used in candidate generation is usually ignorable. While as the system designed, the data preprocessing & Normalization run on the CPU in single thread even the FastLSH-GPU is operating similarly. So a similar time is being recorded.

The test below shows when the system is under operating state. The base set is being processed and stored in memory. So only query has to go through data preprocessing, normalization and Hash Table Generation unit. Then the processed query can interact with the pre-processed base set N in the collision counting and the candidate generation unit.

Query process



	 FastLSH ESS	 FastLSH Lib	 FastLSH Python	 FastLSH GPU	 FastLSH MPI	 FastLSH Spark
Query Time(s)	18			7	12	46
Query Time per single query	180ms			70ms	120ms	460ms

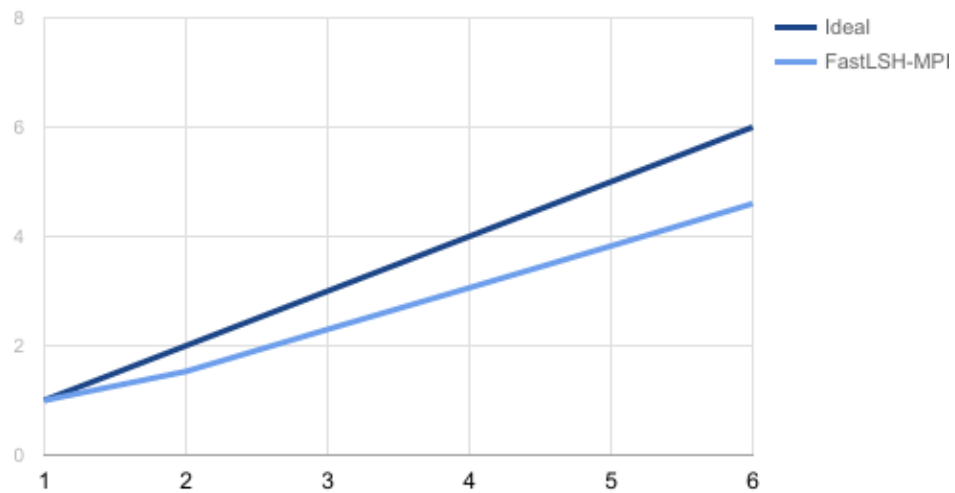
As shown above each query can be completed in as fast as 70ms, even the slowest implementation can be completed in 460ms.

Cluster & Multithread Test

In Cluster & Multithread test, we perform the test on the FastLSH-MPI and FastLSH-ESS. We gradually increase the threads used in the FastLSH-ESS and the nodes used in FastLSH-MPI. The y-axis represents the times of performance it improved and the x-axis represented the number of nodes in the FastLSH-MPI and the number of thread in the FastLSH-ESS

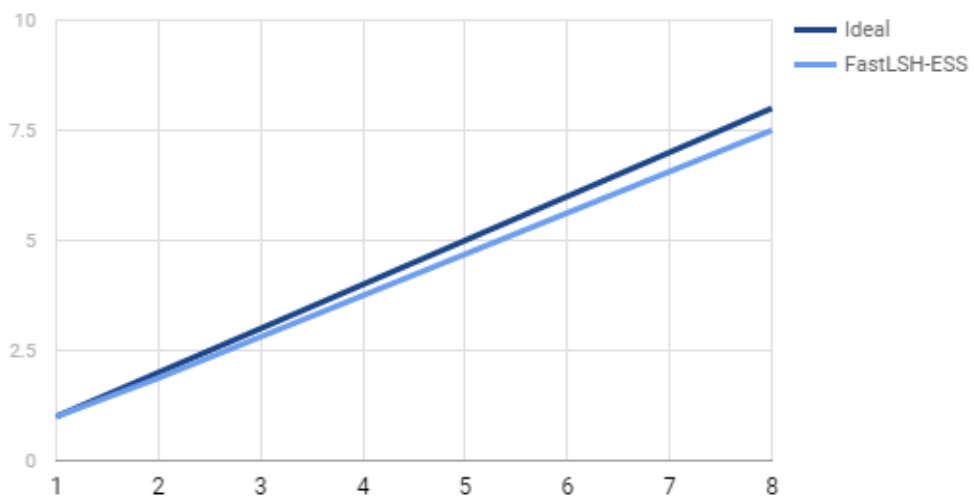
FastLSH-MPI

Cluster Speedup



FastLSH-ESS

MultiThread Speedup



As indicated by the ideal performance line, the overhead is larger in the FastLSH-MPI solution while the FastLSH-ESS line stay closer to its ideal line. This is due to the communication cost is expensive among nodes in the FastLSH-MPI solution. So as the scale of the MPI cluster expand, the line will finally become horizontal as the overhead is higher than the performance gain.

FastLSH UI & FastLSH VR

FastLSH UI

User needs to input the parameters and related information before using the FastLSH engine. The FastLSH-UI provides a user-friendly interface for user to set the parameters and interact with this High Dimensional Big Data Similarity Search Engine.

The screenshot shows the 'Parameter Setting' window with a progress bar at the top indicating four steps: Step 1 (Basic Information), Step 2 (Parameters), Step 3 (Execution Mode), and Step 4 (File Paths). Step 1 is currently active. Below the progress bar, there is a text input field labeled 'Run Name *'. At the bottom right, there are three buttons: 'Finish', 'Next', and 'Previous'.

Parameter Setting

Please input parameter setting here

1 Step 1 Step 1 Basic Information 2 Step 2 Step 2 Parameters 3 Step 3 Step 3 Execution Mode 4 Step 4 Step 4 File Paths

Run Name *

Finish Next Previous

Parameter Setting Step 1 Basic Information

The screenshot shows the 'Parameter Setting' window with the progress bar indicating Step 2 (Parameters) is active. Below the progress bar, there are seven parameter input fields with their respective values: N (1000), Q (1000), D (56), L (200), K (1), W (1.2), and T (100).

Parameter Setting

Please input parameter setting here

1 Step 1 Step 1 Basic Information 2 Step 2 Step 2 Parameters 3 Step 3 Step 3 Execution Mode 4 Step 4 Step 4 File Paths

N: number of points(lines,rows) in the dataset

Q: number of points(lines,rows) in the queryset

D: number of dimensions (columns)

L: number of group hash

K: number of hash functions in each group hash

W: bucket width

T: threshold

Parameter Setting Step 1 Parameters

Parameter Setting

Please input parameter setting here

1

2

3

4

Step 1

Step 2

Step 3

Step 4

Step 1 Basic Information

Step 2 Parameters

Step 3 Execution Mode

Step 4 File Pathes

Step 3 Content

Compute Mode

0 - normal

Thread Mode

0 - singleThread

Finish

Next

Previous

Parameter Setting Step 3 Execution Mode

Parameter Setting

Please input parameter setting here

1

2

3

4

Step 1

Step 2

Step 3

Step 4

Step 1 Basic Information

Step 2 Parameters

Step 3 Execution Mode

Step 4 File Pathes

Input Path for datasetN:

./dataset1000NoIndex.csv

Input Path for datasetQ:

./dataset1000NoIndex.csv

Output Path:

candidate.csv

Make sure your input is right. Click Finish for next step.

Finish

Next

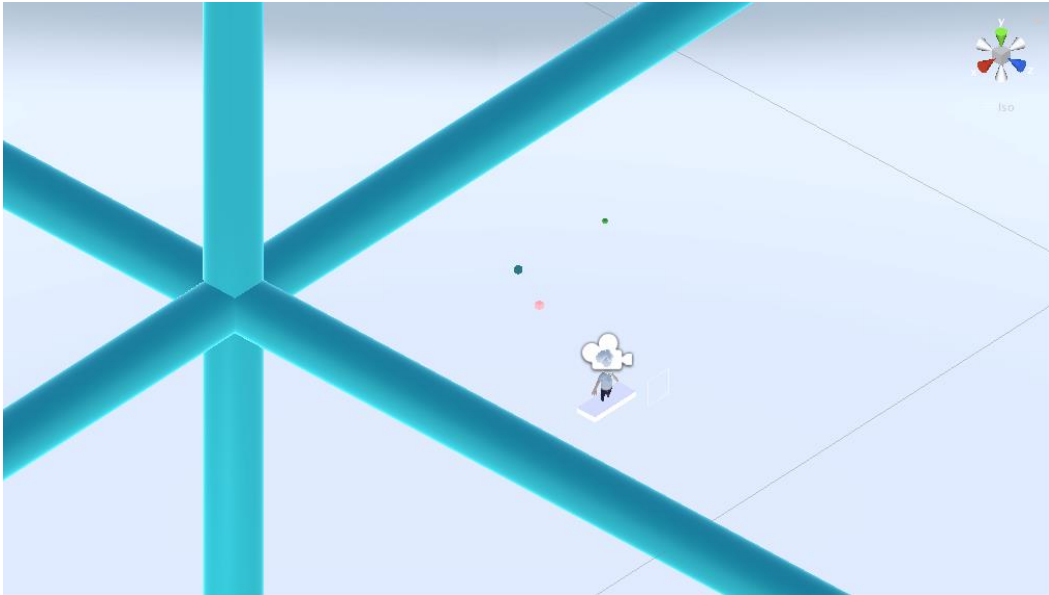
Previous

Parameter Setting Step 4 File Paths

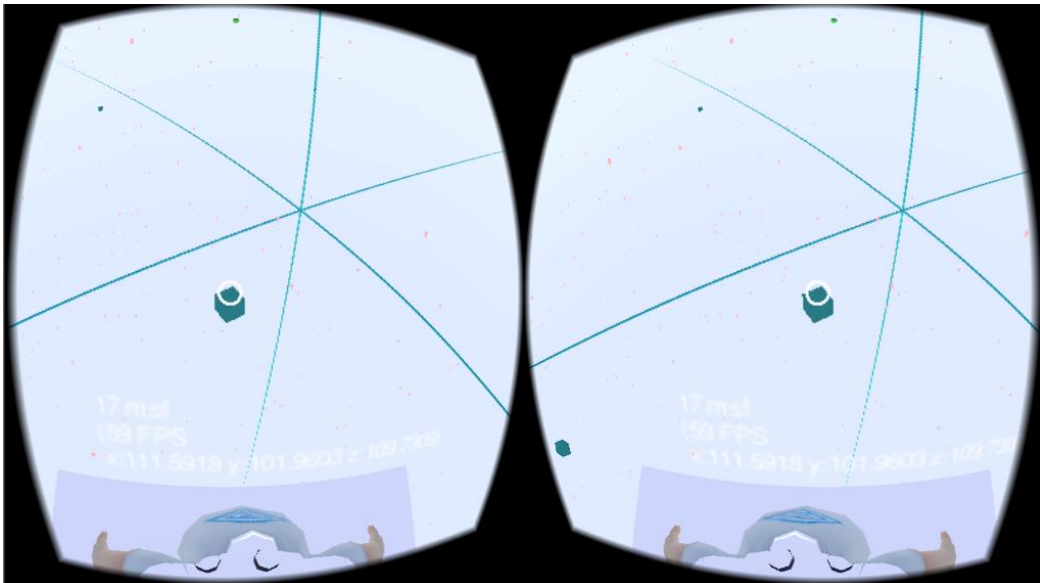
FastLSH VR

The FastLSH-VR has been successfully built by the unity engine. It can be easily installed in common iOS and Android phone. With the help of Google Cardboard, user can have a quick and easy VR experience.

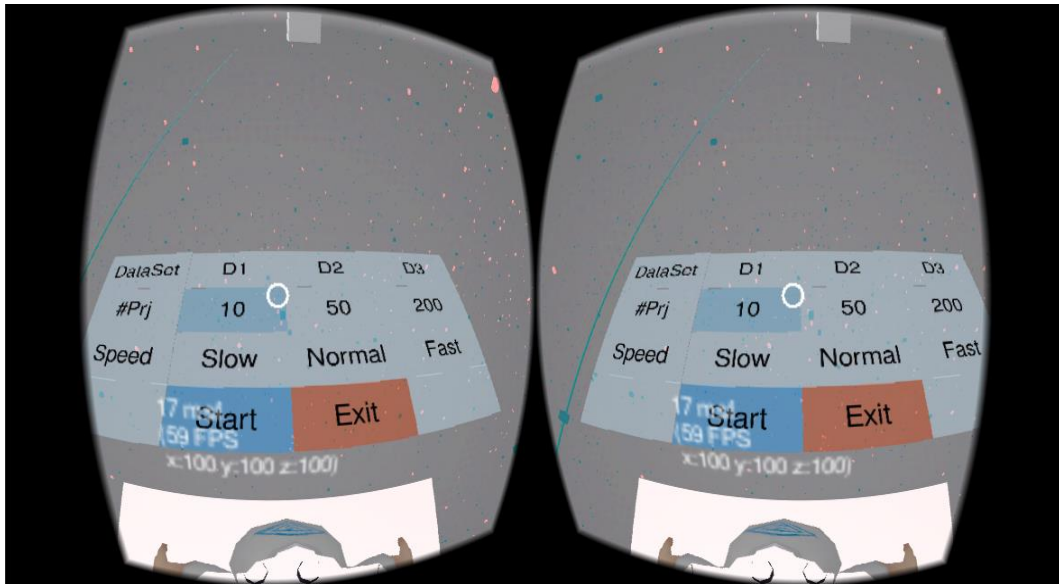
The following screenshot shows the rendering view of the VR.



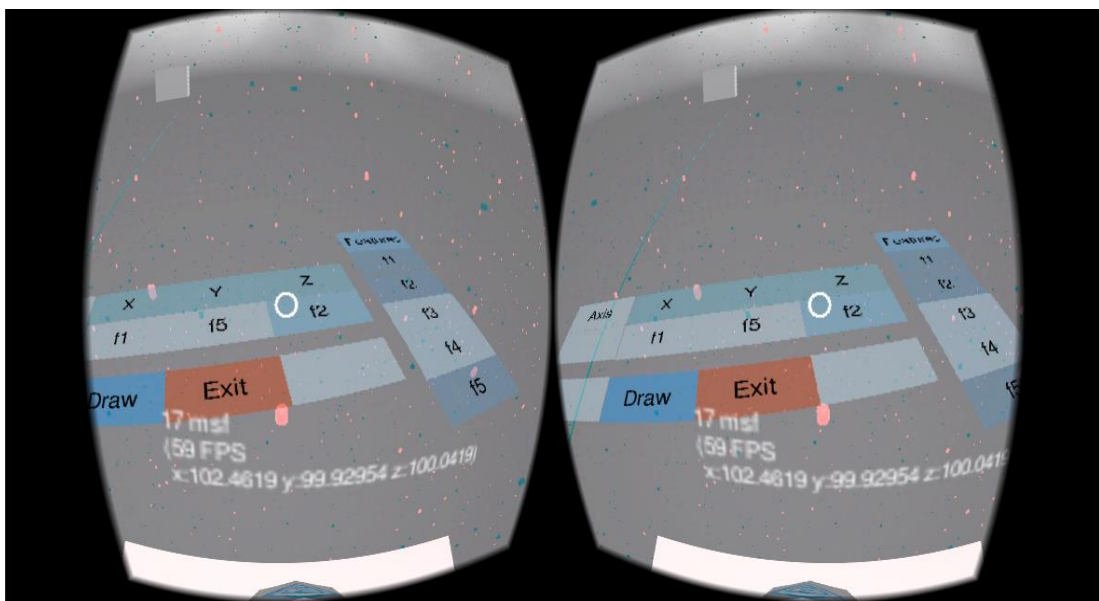
User is traveling through 3D space on a hoverboard.



User see his/her body and the 3d axis



In algorithm demonstration mode user, can set different opinions by looking at the control panel and clicking.



In high dimension data observation mode, user can choose the specific dimensions to show by the control panel.

Conclusion

This report introduced the system FastLSH which gives a fast-deployable parallel and cluster solution as a High dimensional big data similarity search engine. Multiple interfaces and packages are provided for different levels of users by the system. Extensive technologies have been applied to system as well. Moreover, various experiments have been conducted which proves the rightness and high performance of the system.

Future Work / Development

Although the system already has high performance and certain level of software engineering consideration. The ultimate goal of this system is to provide a solution to bridge the research-to-production gap. The system should not only be flexible and performing good enough for research purpose but also stable and accessible for production. There is still space for improvement. They will be illustrated one by one below.

System Core with lower level control

The core of the system is implemented by C++ for its high performance which have been achieved. Thanks to C++, we can explicitly control the memory allocation to optimize the performance. However, there is space to make even lower control. For instance, the system heavily use the STL Vector to store the matrix of the pipeline. The STL vector itself have wrapped up memory control for the ease of programming, for example, it will twice its memory consumption if the number of objects hit its boundary. However, lots of this action can be specially tuned to suit our algorithm for even higher performance, that's to say we can make our own Vector for our system. It requires no less effort and higher programming skill, but it can push the performance to even closer to the ideal line.

Stronger cross platform ability

Although various interfaces for different levels of users have been made. The system still requires certain dependency and configuration to ensure its successful execution. There are two solutions that will be enable us to tackle the issue.

Firstly, one of the under-development package, FastLSH-Docker. By implementing the Docker version, the entire system is running on top of the Docker, such that the system in some level will be a portable solution, as long as Docker is well installed on the target system.

Secondly, wrapping the entire project with OS dependent installation package, then user may install the package on their computer with simply a click. This implementation will enable to check if dependency has been installed on the target system, this can ensure the smooth user experience. However, in return, giving out some of the flexibility.

Fault tolerance

The system currently does not provide any error tolerance among cluster. A dead node or the shortest offline incident is enough to kill the running process. Popular cluster solution like Hadoop, duplicates each record to achieve high error tolerance. With no doubt there is another tradeoff between performance and stability.

The current FastLSH build is not aiming for hundreds of nodes like Hadoop, so single node error is of low probability and error tolerance function is not of sever need. However, to make the system even stronger, the error tolerance cannot be avoided in the future. This is another future development we are excitingly anticipating.

Batching system

The data size is still bounded by accumulate memory size among clusters. With batching technology implementation, eliminate the memory limit is possible, though there will be a huge trade-off in performance. However, the system may still include batch mode as another execution mode in the future development.

Real time visualization with VR

The FastLSH-VR currently have rare flexibility. By implementing communication channel between VR device and execution machine, it is possible for the FastLSH-VR implementation to display real-time visualization of user defined input. Then the FastLSH-VR will no longer just be useful for education and demonstration process but a professional data visualization tool.

Reference

- [1] J. Pan and D. Manocha, "Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation," pp. 211–220, 2011.
- [2] J. Pan and D. Manocha, "Bi-level Locality Sensitive Hashing for K-Nearest Neighbor Computation," 2010.
- [3] T. Karnagel, R. Mueller, and G. M. Lohman, "Optimizing GPU-accelerated Group-By and Aggregation," *Sixth Int. Work. Accel. Data Manag. Syst. Using Mod. Process. Storage Archit.*, pp. 13--24, 2015.
- [4] B. He *et al.*, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 1–39, 2009.
- [5] D. a. Alcantara *et al.*, "Real-time parallel hashing on the GPU," *ACM Trans. Graph.*, vol. 28, no. 5, p. 1, 2009.
- [6] K. Kato and T. Hosino, "Solving k-Nearest Vector Problem on Multiple Graphics Processors," *0906.0231*, 2009.
- [7] J. Pan, C. Lauterbach, and D. Manocha, "Efficient nearest-neighbor computation for GPU-based motion planning," *IEEE/RSJ 2010 Int. Conf. Intell. Robot. Syst. IROS 2010 - Conf. Proc.*, pp. 2243–2248, 2010.
- [8] Y.-F. Zhang, X.-Q. Yu, X.-D. An, and W.-G. Wan, "A Locality Sensitive Hashing Algorithm Based on CUDA," *J. Appl. Sci. — Electron. Inf. Eng.*, vol. 33, no. 5, 2015.
- [9] E. S. Larsen and D. McAllister, "Fast Matrix Multiplies using Graphics Hardware," *SC '01 Proc. 2001 ACM/IEEE Conf. Supercomput.*, p. 55, 2001.
- [10] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," *Proc. 2012 Int. Conf. Manag. Data - SIGMOD '12*, pp. 541–552, 2012.
- [11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," *Proc. Twent. Annu. Symp. Comput. Geom.*, pp. 253–262, 2004.

- [12] Harris, Mark. "Unified Memory In CUDA 6". Nvidia Developer Blog. N.p., 2017. Web. 1 Feb. 2017.
- [13] "NVIDIA Gpudirect". NVIDIA Developer. N.p., 2017. Web. 1 Feb. 2017.
- [14] Ádám Moravánszky, Dense matrix algebra on the GPU, 2003. .
- [15] N. Galoppo, N.K. Govindaraju, M. Henson, D. Manocha, LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware, in: Proceedings of Supercomputing 2005, Seattle, WA, 2005.
- [16] "Programming Guide :: CUDA Toolkit Documentation", *Docs.nvidia.com*, 2017. [Online]. Available: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [17] S. Nutanong, C. Yu, R. Sarwar, P. Xu and D. Chow, "A Scalable Framework for Stylometric Analysis Query Processing," 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, 2016, pp. 1125-1130.
- [18] Kriegel, H. P.; Kröger, P.; Zimek, A. (2009). "Clustering high-dimensional data". *ACM Transactions on Knowledge Discovery from Data*. **3**: 1.
- [19] Beyer K., Goldstein J., Ramakrishnan R., Shaft U.: When is Nearest Neighbors Meaningful?, Proc. of the Int. Conf. Database Theorie, 1999, pp.217-235.
- [20] Berchtold S., Keim D. A., Kriegel H.-P.: The XTree: An Index Structure for High-Dimensional Data, Proc. Int. Conf. on Very Large Databases (VLDB'96), Bombay, India, 1996, pp. 28-39.
- [21] Weber R., Schek H.-J., Blott S.: A Quantitative Analysis and Performance Study for Similarity Search Methods in High-Dimensional Spaces, Proc. of 24rd Int. Conf. on Very Large Data Bases (VLDB'98), New York, 1998, pp. 194-205.
- [22] Shaft U., Goldstein J., Beyer K.: Nearest Neighbor Query Performance for Unstable Distributions, Technical Report TR 1388, Department of Computer Science, University of Wisconsin at Madison
- [23] K. Clarkson. An algorithm for approximate closest-point queries. In: Proceedings of the Tenth Annual ACM Symposium on Computational Geometry, 1994, pp. 160-164.

- [24] M. Minsky and S. Papert. Perceptrons. MIT Press, Cambridge, MA, 1969.
- [25] P. Indyk and R. Motwani. Approximate NearestNeighbor: Towards Removing the Curse of Dimensionality. In Proceedings of the 30th Symposium on Theory of Computing, 1998, pp. 604-613.
- [26] Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In Proceedings of the thirty-fourth annual ACM symposium on Theory of computing (STOC '02), pp. 380–388. ACM.
- [27] Jacobsen DA, Thibault JC, Senocak I (2010) An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In: Proceedings of the 48th AIAA aerospace sciences meeting
- [28] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. 2011. Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 23-29.
- [29] N. Karunadasa and D. D. N. Ranasinghe "Accelerating high performance applications with CUDA and MPI " in Proceedings of the Fourth International Conference on Industrial and Information Systems (ICIIS 2009) Sri Lanka 28-31 December 2009.
- [30] Ashwin M. Aji , Lokendra S. Panwar , Feng Ji , Milind Chabbi , Karthik Murthy , Pavan Balaji , Keith R. Bisset , James Dinan , Wu-chun Feng , John Mellor-Crummey , Xiaosong Ma , Rajeev Thakur, On the efficacy of GPU-integrated MPI for scientific applications, Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, June 17-21, 2013, New York, New York, USA
- [31] A. Moitra, "Advanced Algorithms - Lecture 6," MIT.