SCALABLE DYNAMIC LOCALITY-SENSITIVE HASHING FOR STRUCTURED DATASET ON MAIN MEMORY AND GPGPU MEMORY

Toan Nguyen Mau and Yasushi Inoguchi

Inoguchi Laboratory, School of Information Science, JAIST

ABSTRACT

Locality-sensitive hashing(LSH) is a significant algorithm for big-data hashing. The original LSH uses a static hash-table as a reduce mapping for the data. Which make LSH challenging to apply on real-time information retrieval system. The database of a realtime system needs to be scalably updated over time. In this research, we concentrate on increasing the accuracy, searching speed and throughput of the nearest neighbor searching problem on big dynamic database. The dynamic Locality-sensitive hashing(DLSH) is proposed for facing the static problem of original LSH. DLSH is targeted for deploying on main memory or GPGPU's global memory, which can increase the throughput searching by parallel processing on multiple cores. We analyzed the efficiency of DLSH by building the big dataset of structured audio fingerprint and comparing the performance with original LSH. To achieve the dynamics, DLSH requires more memory space and takes slightly slower than the LSH. With DLSH's advantages, it can be improved and fully applied in practice in a real-life information retrieval system.

KEYWORDS

Locality-sensitive hashing, Structured dataset, GPGPU Memory, Similarity Searching, Parallel Processing

1. INTRODUCTION

Big-data with high dimensions is becoming a severe problem in analyzing and processing. A high dimensional dataset like audio fingerprint, images featuring or text requires a lot of storage space and takes long time to retrieval. With the optimized storing data and supporting for searching the similarity data, people can deploy many recommender systems likes recommendations of music/video/new, providing the advertising with context or detecting the illegal/similar digital content on the Internet [1].

Because the requirements of fast searching of an information retrieval system, many algorithms are proposed for indexing the big data such as Dimensionality reduction, clustering, classification or hashing algorithm [2{4]. With the principle of the hashing algorithm, the data can be easily divided by the similarity factors. the data that similar to each other by these factors can be

Natarajan Meghanathan et al. (Eds) : SPPR, SCAI, CSIA, WiMoA, ICCSEA, InWeS, NECO, GridCom - 2018
pp. 167–180, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81715

grouped and store as same location or device [2]. The hashing algorithm can deploy on multiple levels or multiple hash processes that can overlap with each other [5].

Locality-sensitive hashing is a popular hashing algorithm that can group the data into multiple "buckets", the same bucket value will index the similar data. For this characteristic, we can simpler find the nearest neighbors of a query by this hashing value [6].

LSH is an efficient algorithm for approximate nearest neighbor searching. However, the hash-table of LSH is only desirable for the static dataset. Make it not yet widely used for the real-time dynamic database.

It is necessary to improve the structure of LSH to meet the requirements of new generated big dataset system. In this paper, we proposed and analyzed the new approach of storing hash-table targeting for main memory and GPGPU's memory. At this time, DLSH can show the significant result on parallel dynamic and factor on a single memory device. For the further purpose, DLSH can be optimized for use on distributed GPPGU memory.

2. RESEARCH BACKGROUND

2.1 Locality-sensitive hashing (LSH)

Neighborhood-based methods are the method that sorts the similar items/data by its characteristics [7]. LSH algorithm can compute the similarity weights and compare the differences of data/items for detecting the nearest neighbor of an item [8]. Nearest Neighbor Search(NNS) is a typical problem that searches the most similar item for the query [7]. With NNS, there is a difficulty for the guaranty of the output without comparing the output will all feasible data n the dataset. The ε-Nearest Neighbor Search (ε-NNS) is an is a variation of NNS that approximate the nearest neighbor by a threshold function.

Theorem 1 (Nearest Neighbor Search (NNS)). *Given a set P of objects represented as points in a normed space* l_p^d, *preprocess P so as to efficiently answer queries by finding the point in P closest to query point p [9-11].*

NNS is a famous problem in many fields of science and engineering. There are many algorithms already proposed to handle the NNS for every species of the database. However, the complexity of algorithms grows exponentially with the dimensions (curse of dimension), which is a significant difficulty for the real-time system with high dimensions. By a simple trade-off, we can deal with the curse of dimension by using a technique for approximating the NNS [12].

Theorem 2 (ε-Nearest Neighbor Search (ε-NNS)) *Given a set P of objects that are represented as points in a normed space* l_p^d, *pre-process P in order to return efficiently a point* $p \in P$ *for any given query point q in such a way that* $d(q,p) \leq (1+\varepsilon)d(q, P)$ *where d(q, P) is the distance from q to its closest point in P [9,13].*

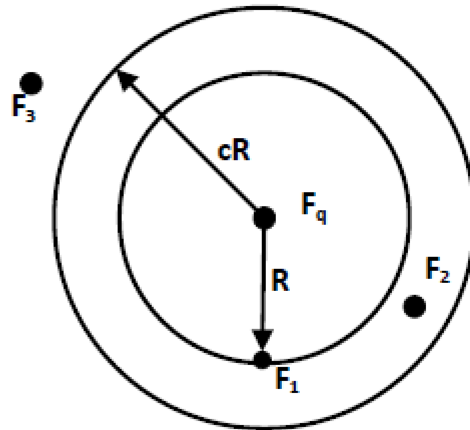

Fig. 1. An illustration of nearest neighbor and approximate nearest neighbor [9, p.2]

In Fig. 1, we can see there are three points in the database F_1, F_2, F_3 and a query point F_q. For the nearest neighbor problem, F_1 should be the chosen one. However, when we consider approximate nearest neighbor problem on this database, suppose that the distance between F_q and F_1 is R and the approximate factor is c, there are two points F_1, F_2 will meet the requirement $|F_q - F_i| \leq cR$. In this case, we can also return F_1 or F_2 both of which are fine.

LSH is a well-known algorithm to handle the ϵ-NNS problem which uses approximate nearest neighbor. LSH divides the data into multiple buckets, the number of hash function in hash family function will indicate the number of buckets in system. Vectors/points in the same buckets will be similar to each other because of the continuity of the selection of hash functions. Therefore, instead of computing the similarity of the input vector with all of the vectors in database, we need to compare the query with the vectors in several buckets.

With LSH, hash functions will be used for choosing l subnets $I_1, I_2,..I_l$ of database vectors. Let p_I be the projection of vector F_j on the coordinate positions. Then denoting $g_j(p) = P_{I,}$ we store every $F_j \in F$ in the bucket $g_j(F_p)$. Since the number of buckets is probably large or the numbers of points in every bucket differ considerably, so another table is also needed to save the map of buckets.

As to the searching problem within LSH, the same hash functions are also used for each query. As for the query F_q, we define all $g1(q), g2(q),\ldots gl(q),$ and then let $F1, F2,\ldots Ft$ be the points in bucket on the current process. We have to calculate the distance $l_1(Fp, Fq)$ for each point in this bucket. As for KNN problem, we do not stop until we reach K points in the same or different buckets. However, with the audio fingerprint, it can be returned at the first F_p having the $l_1(Fp, Fq) < P_1$ to attain a good result, along with a better performance. Note that in case the two points are close to each other, P1 is a threshold of the maximum distance.

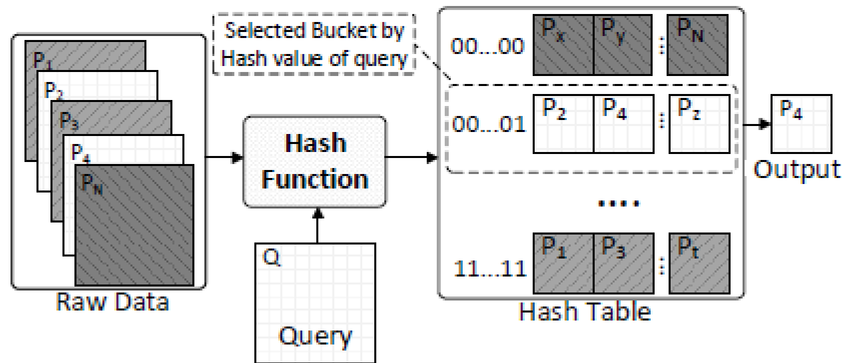

Fig. 2. An illustration of locality-sensitive hashing

In Fig. 2, LSH chooses a family of hash function for handling database and query also. In the preprocessing stage, hash function is used for dividing the database into buckets. There are three buckets with the different colors in the figure. Each bucket will have its hash value by the principle of hash function. In term of the Searching stage, the query also needs calculating the hash value by the previous hash function. This value of hash function will indicate to a bucket that holds the similar points to the query (light box). Next, there is another step for comparing the distance from query to all points in the purple buckets and returning the closest one.

3. DYNAMIC LOCALITY-SENSITIVE HASHING (DLSH)

In this paper, we proposed the DLSH that have the dynamic structure for storing the dataset on heap-memory or GPGPU's global memory.

3.1 Memory Pools Allocation

The memory manager of main memory and GPGPU's memory is very similar by using the address pointer. However, the pointer in CUDA has the restriction for dynamic allocation on the kernel. We choose to use the memory pools allocation for both kind of memory for reducing the number of fragment memory and also the reusable of memory pools.

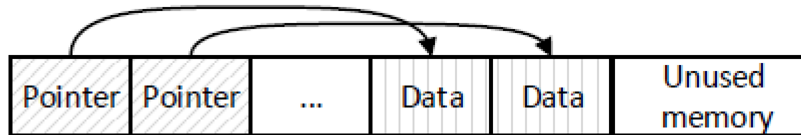

Fig. 3. An example of Memory pools allocation

In Fig. 3, the allocated array have the fixed size and can hold multiple user-defined pointers and data on it. In the last part, there an unused memory for storing the new dynamic data/item.

It is clear that the hash table DLSH will take advantages of memory pools when storing both mapping key and value on the same array.

3.2 Linked-list of bucket

Data will be stored as a sequence that indicated by an hash-value on LSH. We proposed to used linked-list structure to track all the data in a bucket. An important part of hashing table of DLSH is the pointers of all available buckets. To handle this issues, in the first part of hash-table array, we store the static pointers for every bucket by the hash values.

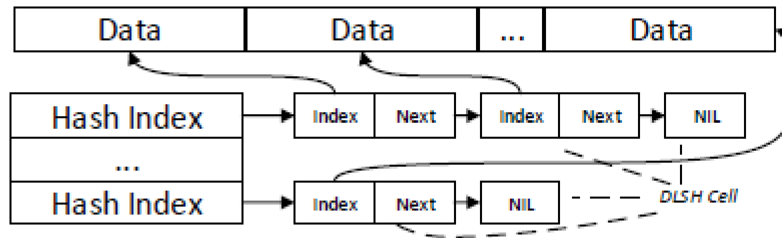

Fig. 4. Linked-list bucket structure of DLSH

Fig. 4 shows the static Hashing Index part correspondings with hashing values of family of hash function. The value of each Hash Index will point to its first index of data. the cell of bucket linked-list contains two important values the address of the data/item and the pointer of the continues cell. The NIL cell is a special cell that denotes the ending of a bucket. In case of a bucket is empty the pointer to the first cell will be set to NIL.

By using the linked-list, the hash-table is compacted and stored on a single array. This help the system can easily allocate the array for both main memory and GPGPU's global memory. Besides that, the linked-list has the significant advantage in modifying the hash-table. However, compare to LSH, each cell of the list required one more addressing space for index the next cell on the list, which make the size of DLSH bigger the size of LSH's hash-table.

3.3 Remove item from DLSH's hash-table

For the real-time information retrieval system, the system needs to support to remove items from the hash-table, which is an essential factor that makes the DLSH become the dynamic hashing system. To delete an item from hash-table, we need to find the previous cell that point directly to the deleting item, then the next pointer of the previous cell will point to the address of next-point of deleting time.

In Fig. 5, the red link will be changed to the blue link, and the deleting item will be unreached by using hash-table's linked-list. That is the reason we proposed to use another pointer-list that point to the deleted cell to keep them on track.

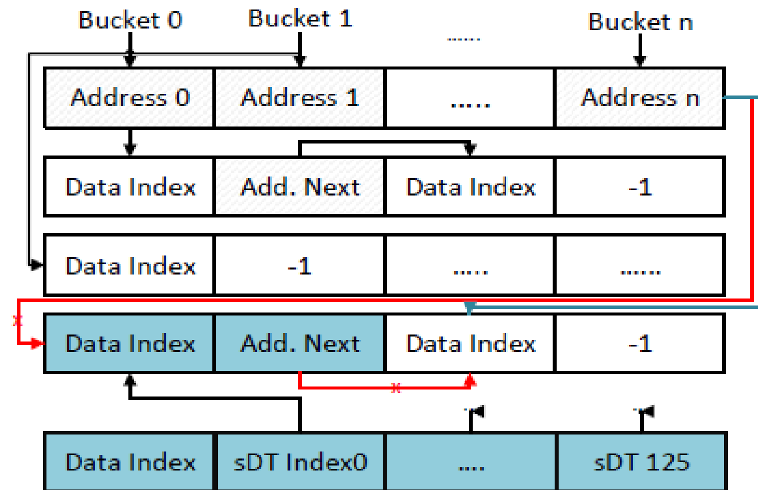

Fig. 5. Scenario of deleting item from DLSH's hash-table

In Fig. 1, because there is no data addressing pointer in static pointer for buckets, the system has to check the first element of linked-list. The previous-pointers of these specific cells are assigned directly to its memory location.

3.4 Add item to DLSH's hash-table

We highly recommend that the allocation of the extra empty space for the hash-table of DLSH. This unused space can be used when the hash-table is full and need more space for the new cell for the new item. In Fig .6, the empty space is used for creating new linked-list cell, the data index points to the address of new item/data on the data array. In this example, we need to find the last cell of the list that indicated by new hash value and assign the next pointer to the newly created cell. However, it will be faster when we assign the new cell as the first element of its linked-list. In this case, the next-pointer of new cell will point to the ex-first cell of its bucket.

Algorithm 1 Removing item/data from DLSH's hash-table

1: $hash \leftarrow \textsc{HashFunction}(q)$
2: $hash_{pre} \leftarrow hash$
3: $hash \leftarrow HT[hash]$
4: $is_first_element \leftarrow$ TRUE
5: **while** $hash \neq$ NIL **do**
6:     **if** $\textsc{Match}(q)$ and $\textsc{GetData}(hash)$ **then**
7:         **if** $is_first_element$ **then**
8:             $HT[hash_{pre}] \leftarrow HT[hash+1]$
9:         **else**
10:             $HT[hash_{pre}+1] \leftarrow HT[hash+1]$
11:         **end if**
12:     **end if**
13:     $hash_{pre} \leftarrow hash$
14:     $hash \leftarrow HT[hash+1]$
15:     $is_first_element \leftarrow$ FALSE
16: **end while**

Algorithm 2 Adding item/data to DLSH's hash-table

1: $hash \leftarrow$ HASHFUNCTION(q)
2: $hash \leftarrow HT[hash]$; $hash_{pre} \leftarrow$ NIL
3: **if** Exist empty-cell in HT **then**
4: $\quad hash_{restored} \leftarrow$ FINDEMPTYPOINTER
5: **else**
6: $\quad hash_{restored} \leftarrow$ EXTEND(HT)
7: **end if**
8: **while** $hash \neq$ NIL **do**
9: $\quad hash_{pre} \leftarrow hash$
10: $\quad hash \leftarrow HT[hash + 1]$
11: **end while**
12: **if** $hash_{pre} \equiv$ NIL **then**
13: $\quad hash_{pre} \leftarrow hash \leftarrow$ HASHFUNCTION(q) -1
14: **end if**
15: $HT[hash_{pre} + 1] \leftarrow hash_{restored}$
16: $HT[hash_{restored} + 1] \leftarrow$ NIL

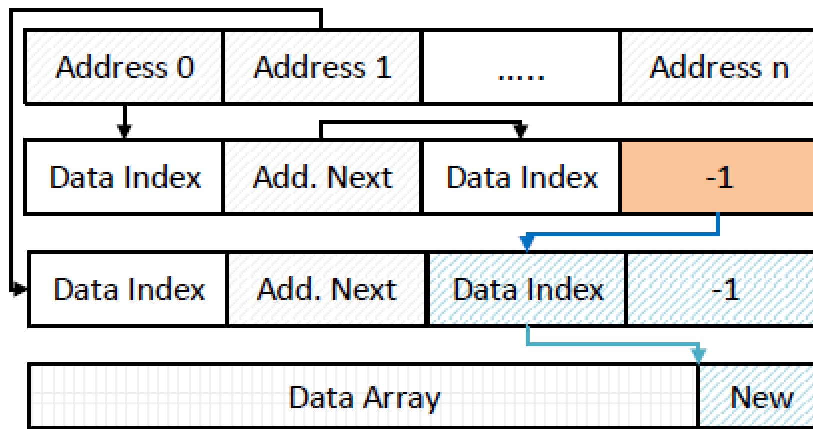

Fig. 6. Scenario of adding item of DLSH's hash-table when the hash-table is full

In Algorithm 2, If the DLSH's hash-table is full, there is required to extend the size of array of HT. We can use the unused space when allocated the memory pool for the HT array. Otherwise, there is an empty linked-list cell, we have to reuse it to reduce the memory size and increase the performance. In Fig. 7, the restored cell (yellow) has been reused, that become the last element in the linked-list of first bucket. The memory space of data/item also can be reused in this figure.

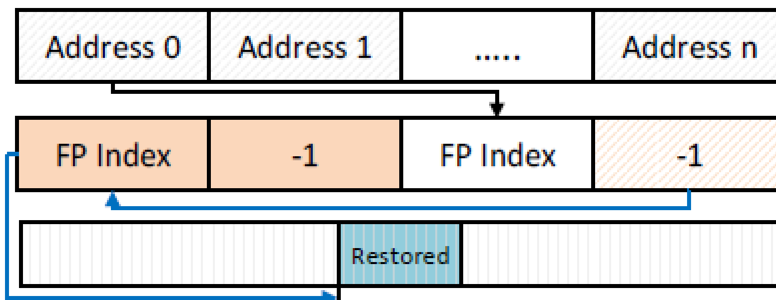

Fig. 7. Scenario of adding item with empty-cell on hash-table

4. EXPERIMENTAL SETUP

In this research, we are targeting on using DLSH for handling the database with millions of structured audio fingerprint for the information retrieval system. For Collaborative Filtering (CF) problem, we experimented ε-NNS problem for the dataset of audio fingerprint to find the approximate most similar song on the database with the queries.

4.1 HiFP2.0: An Binary Sequence Audio Fingerprint

An audio fingerprint is a digital vector that extracted from the audio/song waveform and able to standardize the content of audio/song source. The audio fingerprint can easily help for comparing the similarities and differences of songs. In addition, using audio fingerprint for storing can reduce the size of original audio/song with the standard structure. In our system's database, we store the audio fingerprints and its meta information for every songs/track we considered instead of storage the real waveform of the song [14, 15].

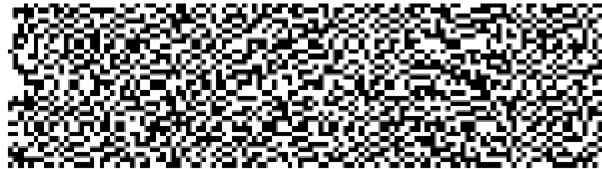

Fig. 8. A example of 4096-bits fingerprint extracted by HiFP2.0 audio fingerprint extraction algorithm

In Fig. 8, there is an audio fingerprint that is represented by a binary sequence extracted by HiFP2.0 fingerprint extraction algorithms. This audio fingerprint is extracted from first 2.97 seconds of a song, and that can reduce the size of the song by 512 times [16].

For description the whole content of an audio input, with the difference in audio length, we will have different size of audio fingerprint [17]. With the different length of audio fingerprint, there is some problem with storage for fast searching [18].

Using audio waveform for comparing have many difficulties because that is un-normalized, so we favor to using an audio fingerprint for storing and processing in our system.

4.2 Dataset and Computer Memory Architecture

We aimed to examine our proposed system on larger memory with the enormous number of data on the database. With the typical size of an HiFP2.0 feature is 512 bytes, we generated a set of 10 million HiFP2.0 with the size of 5 GB for testing. To analyze the accuracy of both LSH and DLSH system, we created numerous of testing queries with different distortion from the dataset and examined with different number of hash function on hash function family.

The Staged-LSH was proposed by [16], that can significantly increase the accuracy of ε-NNS for similarity song searching and that was verified in [16]. Staged-LSH group the data/item into multiple sub-sequences and computes the hash value for each sub-sequence. With the size of HiFP2.0, authors recommended dividing the audio fingerprint to 126 sub-fingerprints, which make a single audio fingerprint can have up to 126 different hash values and indexed by 126 different buckets.

Table 1. The dataset's and hash-table's size for the database of 1 million HiFP2.0 features

Database size (MB)	Num Hash function	Staged-LSH HT Size(MB)	Staged-DLSH HT Size(MB)
488.28	17	480.78	961.43
488.28	18	480.90	961.55
488.28	19	481.15	961.80
488.28	20	481.65	962.30
488.28	21	482.65	963.30
488.28	22	484.65	965.30
488.28	23	488.65	969.30

Table 1 shows the actual memory size for Staged-LSH and Staged-DLSH. Because Staged-LSH increases the number of indexing-cell to 126 times, that make the size of hash-table nearly equal to the size of the dataset. The DLSH take more memory space than LSH by the linked-list pointer, which makes the serious problem of DLSH when trade-off with the dynamic structure.

The specifications of the testing computer are shown in Table 2. We examined and compared the performance of LSH and DLSH using the same computer and same conditions.

5. RESULTS AND COMPARISON

The dynamic feature is the most important factor of DLSH, the dynamic changes of DLSH hash-table (adding/removing) are mainly analyzed. We also compare the efficiencies of DLSH to this main competitor LSH to very the feasibility of DLSH. The principle of DLSH is inherited from LSH, which make the accuracy of NNS of both methods are similar. Therefore, we only

the data size is shown in Fig. 9. We deployed tests on various dataset size with 19 hash functions. The number of hash-table lookup of deleting and adding commands is depended to the average number of cell in each bucket. When continues deleting data/item from the dataset, the number of hash-table lookups will be reduced. Adding the data/item at the head of linked-list is recommended for getting performance. However, this approach is easier to make the database fragment.

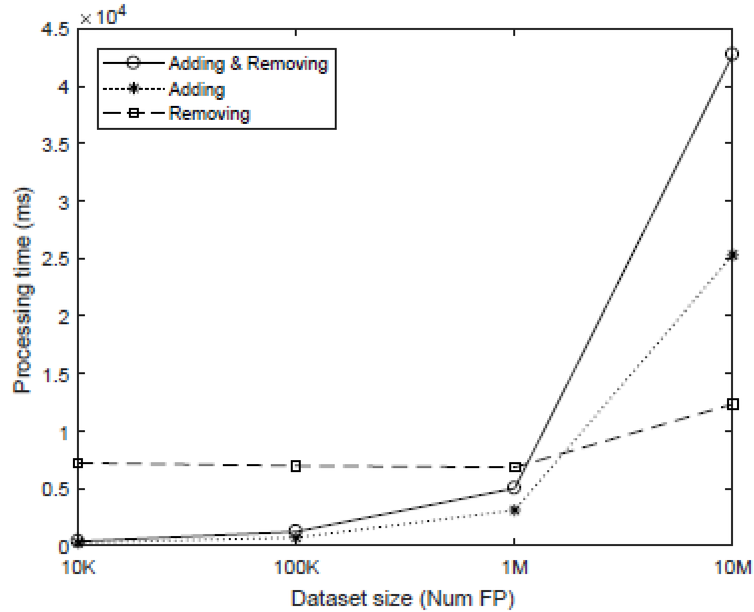

Fig. 10. Processing time of adding/removing 10,000 queries over different dataset size with 19 hashing functions

In Fig. 10, when the dataset size is small, the adding data/item tasks takes less time than removing by using the empty slots in DLSH's hash-table. With 10 million audio fingerprint dataset, the adding/removing tasks requires more to find the suitable position for deleting or adding.

For the scalability factor, we conducted experiments on various dataset size by turning the number of audio fingerprint on the dataset. Fig. 11 compares the searching time of the original LSH with our proposed Dynamic LSH. With numerous additional pointers, our method has an overhead around 10% of the original method. The DLSH's hash-table structure is well developed for supporting parallel processing, especially for multiple core processor by using fix-size pointers of buckets on same 'page-locked' memory.

In Fig. 12, with different number of hash function, the searching of our method have slightly higher than original LSH on finding the approximate nearest neighbor of 10,000 queries. Choosing hashing number is very sensitive, with the low number of hash function we can quickly achieve highly accurate but the high density of buckets will reduce the searching time. Using GPGPU can increase the speed of searching into nearly 2 times than using CPU. In other hand, the performance of GPGPU can get better when we testing on higher throughput of queries.

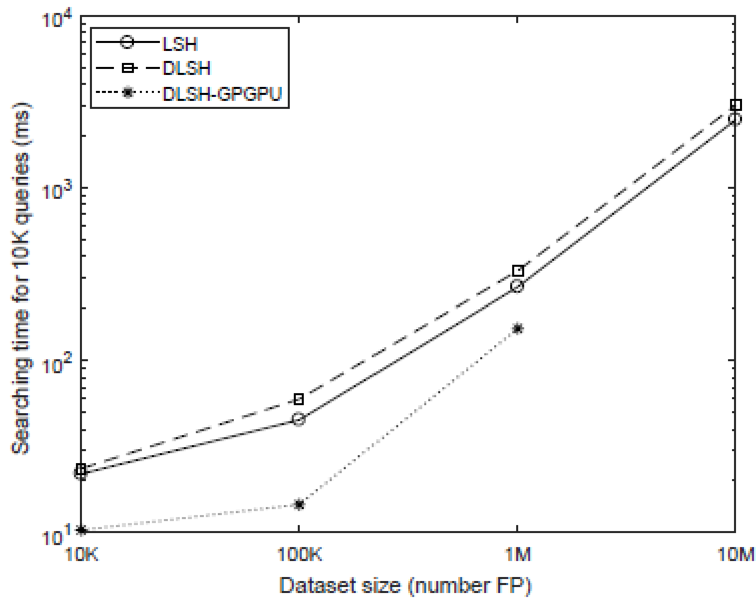

Fig. 11. The ε-NNS processing time comparing between LSH and DLSH on 10,000 queries with 10% distortion queries. Both methods use 19 hash functions on hash function family.

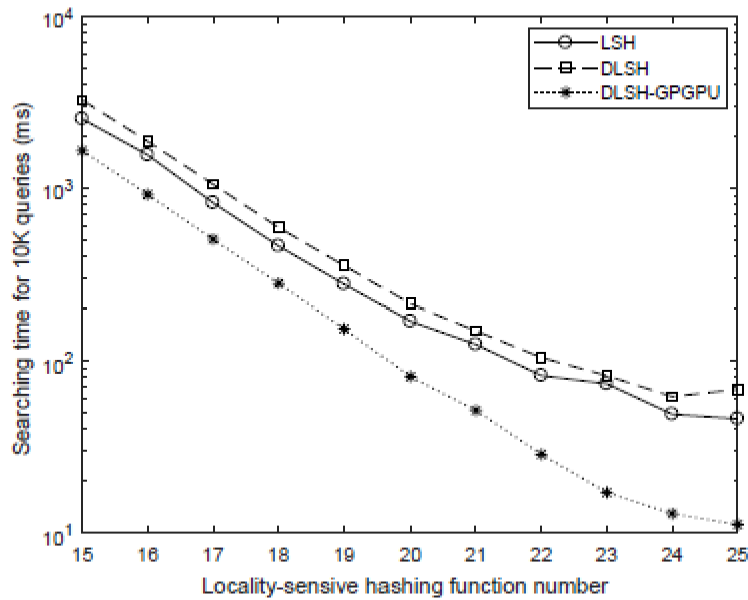

Fig. 12. The ε-NNS processing time comparing between LSH and DLSH on 10,000 queries with 10% distortion queries on dataset with 1 million HiFP2.0 features.

In Fig. 13, when there is no distortion on testing queries, the DLSH of takes more memory accessing by the static pointer. Besides that, the LSH can directly point to the first data of the buckets, which make the searching time of DLSH nearly double the LSH's.

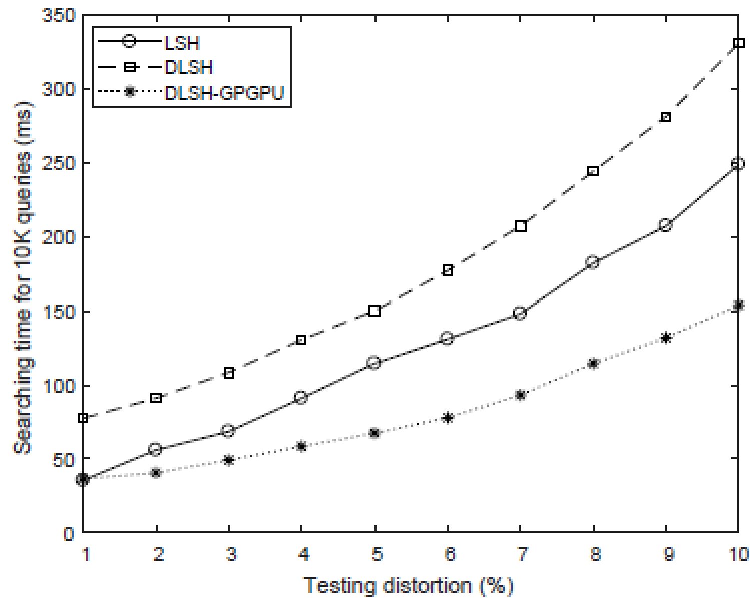

Fig. 13. Comparing performance between LSH and DLSH with different testing distortion

6. CONCLUSION

In this paper, we proposed DLSH a new approach for dynamic hashing dataset that inherited the advantages of locality-sensitive hashing. The DLSH aimed for handling the big real-time dataset for information retrieval system which requires quick response time and high throughput.

The dynamic structure needs extra memory space for holding the pointers of hash-table's cell and addressing to the dataset. However, the performance of DLSH is acceptable for a dynamic system comparing to the performance of the original LSH. The DLSH can reuse the space in main memory or GPGPU memory. Although it still takes time to find the exact memory address.

For the future work, we concentrate on optimizing the DLSH for getting better performance and stabilization. The memory size of main memory and GPGPU is limited, we also target to extend the DLSH for scaling on distributed GPGPU computer system.

REFERENCES

[1] Gema Bello-Orgaz, Jason J Jung, and David Camacho. Social big data: Recent achievements and new challenges. Information Fusion, 28:45-59, 2016.

[2] Jong Soo Park, Ming-Syan Chen, and Philip S Yu. An effective hash-based algorithm for mining association rules, volume 24. ACM, 1995.

[3] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. Mobile networks and applications,19(2):171-209, 2014.

[4] William Bruce Frakes and Ricardo Baeza-Yates. Information retrieval: Data structures & algorithms, volume 331. prentice Hall Englewood Cliffs, NJ, 1992.

[5] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pages 37-48. ACM, 2011.

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In Proceedings of the twentieth annual symposium on Computational geometry, pages 253-262. ACM, 2004.

[7] Christian Desrosiers and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In Recommender systems handbook, pages 107{144. Springer, 2011.

[8] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems, pages 211{220. ACM, 2011.

[9] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on, pages 459{468. IEEE, 2006.

[10] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604-613. ACM, 1998.

[11] Wei Cen and Kehua Miao. An improved algorithm for locality-sensitive hashing. In Computer Science & Education (ICCSE), 2015 10th International Conference on, pages 61-64. IEEE, 2015.

[12] Anirban Dasgupta, Ravi Kumar, and Tamas Sarlos. Fast locality-sensitive hashing. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1073-1081. ACM, 2011.

[13] Edward Y Chang. Approximate high-dimensional indexing with kernel. In Foundations of Large-Scale Multimedia Information Management and Retrieval, pages 231-258. Springer, 2011.

[14] Jaap Haitsma and Ton Kalker. A highly robust audio fingerprinting system. In Ismir, volume 2002, pages 107-115, 2002.

[15] Kouichi Araki, Yukinori Sato, Vijay K Jain, and Yasushi Inoguchi. Performance evaluation of audio fingerprint generation using haar wavelet transform. In Proceedings of the International Workshop on Nonlinear Circuits, Communications and Signal Processing, Tianjin, China, pages 380-383, 2011.

[16] Fan Yang, Yukinori Sato, Yiyu Tan, and Yasushi Inoguchi. Searching acceleration for audio fingerprinting system. In Joint Conference of Hokuriku Chapters of Electrical Societies, 2012.

[17] Toan Nguyen Mau and Yasushi Inoguchi. Audio fingerprint hierarchy searching strategies on gpgpu massively parallel computer. Journal of Information and Telecommunication, pages 1-26, 2018.

[18] Toan Nguyen Mau and Yasushi Inoguchi. Audio fingerprint hierarchy searching on massively parallel with multi-gpgpus using k-modes and lsh. In Eighth International Conference on Knowledge and Systems Engineering (KSE), pages 49-54. IEEE, 2016.

AUTHORS

Toan Nguyen Mau received the BS degree (Honors Program) from University of Science - Ho Chi Minh Nation University (VNU) , Vietnam, in 2013 and the MS degree in information science from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Ishikawa, in 2016. He is currently a phD student in Inoguchi laboratory at the Graduate School of Information Science at JAIST. His research interests are parallel processing using GPGPU, massively parallel process and parallel audio fingerprint extraction algorithms.

Yasushi Inoguchi received the BE degree from the Department of Mechanical Engineering, Tohoku University in 1991, and received the MS and PhD degrees from Japan Advanced Institute of Science and Technology (JAIST) in 1994 and 1997, respectively. He is currently a professor of RCAsCI at JAIST. He was a research fellow of the Japan Society for the promotion of Science from 1994 to 1997. He is also a researcher of PRESTO program of Japan Science and Technology Agency from 2002 to 2006. His research interest has been mainly concerned with parallel computer architecture, interconnection networks, GRID architecture, and high-performance computing on parallel machines. Dr Inoguchi is a member of IEEE and IPS of Japan.