2016 IEEE 2nd International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, IEEE International Conference on Intelligent Data and Security

# Parallelization of Error Weighted Hashing for Approximate k-Nearest neighbour search on GPU-CPU hybrid

Siddharth Bhatia
*Birla Institute of Technology and Science, Pilani*
*siddharthbhatia2003@gmail.com*

Mohan Pavan Kumar Badarla
*SERC, Indian Institute of Science*
*pavanbadarla@gmail.com*

***Abstract*—Error Weighted Hashing (EWH) is a fast algorithm for Approximate k-Nearest neighbour search in Hamming space. It is more efficient than traditional Locality Sensitive Hashing algorithm (LSH) since it generates shorter list of strings for finding the exact distance from the query. We have parallelized the EWH algorithm using Cuda and OpenMP. Speedup of 44 times on a 16 core GPU and 16 core CPU machine was achieved in case for hashing and 24 times for retrieval.**

***Keywords*-Approximate nearest neighbour search; Hamming space; Locality Sensitive Hashing**

## I. Introduction

Nearest neighbour search in Hamming space is the problem of finding close matches of a query string from a set of strings in reference database. Given a $b$-bit binary string (i.e, contains 0 or 1 in all the $b$ bits), the problem is to find $k$ strings that match with the query string approximately. A string is closely matched with other string if it has the same values (i.e, 0 or 1) in many bits. For example [0010] matches more closely with [1010] (1-bit difference, only fourth bit) than [0100] (2-bit difference, second and third bits). Hamming distance between two strings is defined as the number of bits that are different.

Many signal processing applications like matching fingerprints, identifying existing copies of multimedia and image search involve nearest neighbour search. Also, it is required to find the matches very fast. As the database becomes larger, the searching could become a bottleneck. Parallel implementations would be very helpful in such scenarios. Searching methods like linear search or binary search do not work here due to the huge size of data. Hence, hashing is used in general for such applications. LSH is a popular algorithm for nearest neighbour search. EWH algorithm was proposed recently [1] which modifies the process of retrieval of strings from the database. It generates a smaller shortlist of strings which has to be checked for exact matches and hence leads to a better performance.

General-purpose computing on graphics processing units (GPGPU) is the technique of using a Graphics processing unit (GPU) to perform the computations usually handled by the central processing unit (CPU). The key idea is to use the huge parallel computing power of GPU to achieve good speed-up. Hashing is a problem where the hash value of one string doesn't depend on the other. So, the massively large amount of threads available on a GPU could be exploited to obtain good performance. The aim of this work is to devise and implement the stategies for EWH algorithm on GPGPUs.

Rest of the paper is organised as follows. In the next section LSH, EWH and distributed LSH [2] will be briefly described. Section III describes the strategies used for parallelizing EWH. Section IV contains the details of experiments and results. The paper is concluded in section V.

## II. Related Work

LSH is a prominent algorithm in Nearest neighbor search literature [1]. LSH and EWH algorithms use similar strategy for hashing. They differ only in the retrieval process. LSH is built on a simple idea: Assume two binary strings that are close in the Hamming space, i.e., they differ only in few bits. If a sub-string is extracted from each of them in the same order, then the probability of these sub-strings being identical will be high. By the same order, we mean that, for example, if the first bit of one sub-string corresponds to the fifth of one original string, then the first bit of the second sub-string should correspond to the fifth bit of the second string, and so on.

Hashing of strings is done in the following way. Let us consider that our target database consists of b-bit binary strings and hash functions are characterised by h random bits from 0 to $(b-1)$. Given a string, the hash function looks only into these $h$ bits of the total $b$ bits. The $h$-bit binary vector of a hash function $k_i$ is formed from the query and let us call it as the hash vector of the query string with respect to $k_i$. Its value in decimal representation is taken to be the hash value. A $h$-bit hash function will have $2^h$ hash buckets in total. Thus for $n$-hash functions, the hash table consists of $2^h$ rows and $n$ colums. As the number of hash

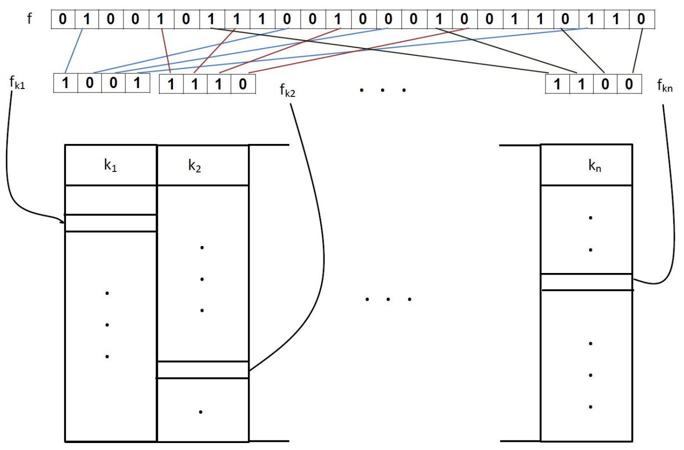

Figure 1. Hashing of a string $f$ with functions $k_1$, $k_2$, ... $k_n$

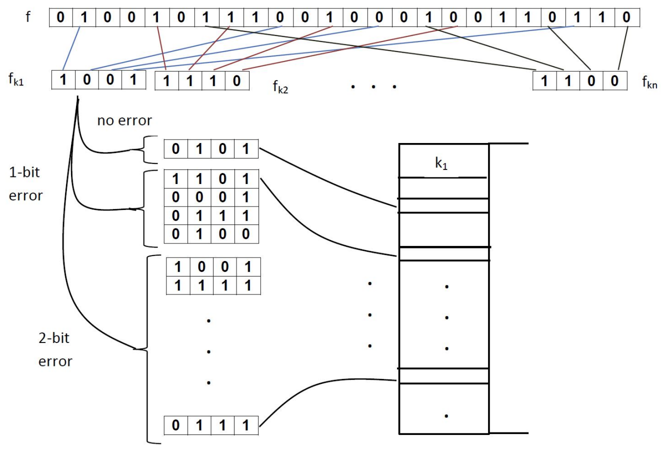

Figure 2. EWH illustration for string $f$ with hash function $k_1$

functions increase, we would be considering many bits of the string and hence the result will be more accurate. But as the number of hash functions increase, the time complexity increases. A string goes into its respective buckets for different hash functions $k_1$ ,$k_2$ ,$k_3$ ....$k_n$ with $h = 4$ bits as shown in Figure 1.

Mani et al. [1] proposed the Error weighted hashing algorithm which has better detection accuracy and less retrieval time compared to Locality sensitive hashing. LSH and similar algorithms fail if there happens to be errors in many hash vectors of the query. Also, LSH fails to include the nearest neighbour in candidate list if none of the hash functions match exactly with it. EWH resolves this problem by considering erroneous hash vectors and using them to generate candidate list. Given a query string, the algorithm considers buckets where hash vectors match exactly (i.e., no-error) or with 1-bit error or with 2-bit-error and so up to $e$-bit-error. $e$ is assumed to be 2 in current work. with respect to a hash function. Then, scores are added to strings which have the corresponding hash value (i.e, corresponding bucket in hash table). It adds a high score to strings present in $no - error$ buckets and a relatively low score to 2-bit-error buckets. A string with less error with respect to many hash vectors (out of $n$ functions) will obtain high score. Next step is to retrieve all the strings which have score greater than a particular threshold and this is the step where EWH algorithm differs from LSH. LSH doesn't have this score concept and ends up computing exact distances for all the strings in a bucket. The number of strings which LSH checks for exact match is claimed to be higher than the number which EWH does [1]. Since the exact match is the bottleneck of performance, LSH ends up in high execution times. Though there is an additional overhead in computing the scores for each string in database, the number of strings EWH checks for computing exact distance is less and hence is expected to perform well.

Smita and Pawan [2] proposed an algorithm for parallelizing LSH using distributed systems. The hash table consists of buckets corresponding to $n$ functions. The task of constructing the hash tables is distributed among the processors. Each processor constructs a hash table for one hash function independently. During the retrieval, the query string is broadcasted to all the processors. Now, with the local hash table the query is searched and a shortlist (i.e., strings in a bucket where hash vector matches exactly) is constructed. Then, the $k$-best among them are selected. The $k$-best from each processor is gathered on one processor and the final $k$-nearest neighbours are computed. This method of having part of hash tables in different processors may lead to load imbalance because the size of shortlist on different processors would be different.

EWH algorithm has better performance in terms of both execution time and detection accuracy. Parallelizing EWH would improve the execution time even better. The problem is inherently parallel as the work that needs to be done with respect to one hash function is independent of other. Also, the task of hashing and operations like updating scores of strings in a large hash bucket are data-parallel. GPUs have shared-memory many-core architecture which is suitable for this application. The objective of this work is to devise parallelizing strategies for EWH on GPUs.

## III. Methodology

### A. *Hashtable construction*

Algorithm 1 describes the method for constructing the complete hash table. For $n$ hash functions with each function characterized by $h$ random bits and hence hash table will have $2^h$ rows and $n$ columns. The size of hash table would be $2^h$ $X$ $n$ buckets. The sum of elements in buckets of each column will be equal to total number of strings in the database. Every string needs to be stored once in each column. We use dictionary compression to save on memory. Each bucket of hash table contains the identifiers (index of string in the database) of strings whose hash value maps to its bucket number.

**Algorithm 1** Construction of hash table

INPUT: Database of $b$ bit strings: $F$, Number of hash functions: $n$, key length: $h$

OUTPUT: Hash table of $2^h$ rows and $n$ columns: $T$

```
1) for $i = 1$ to $n$ do
    a) Generate random keys $k_i = z_1 z_2 ... z_h$ with $z_j \epsilon 1, 2, .., b$ for $1 \leq j \leq h$
    b) for all $f \epsilon F$ do
        i) Add $f$ to row $H(f_{k_i})$ and column $i$ of $T$
    c) end for
2) end for
```

The data-structure we use for hash table is a simple array of ($n\ X\ number\ of\ strings$) integers where $n$ is the number of hash functions used. As the task of hashing is highly data parallel, the massively large amount of cores of GPUs can be used to maximum extent. A string may go into different buckets for different hash functions. While hashing, we need to maintain an index which would be the position of next string to be inserted into the bucket. Initially, we find how many strings would go into a bucket. This has to be done before the actual hashing happens. Since the threads of different blocks of GPU cannot be synchronized, we launch two kernels for the hash table construction. One kernel to find out the start indices of each bucket and the other to actually hash the strings to buckets. In the first kernel, when a thread encounters a string that would go into a bucket, it increments the bucket size atomically. Same has to be followed during actual hashing also to avoid synchronization issues. After finding out the number of elements in each bucket, a prefix scan is done to get the start index of each bucket. Algorithm 2 describes the parallel algorithm for hashing.

### B. *Retrieval of k-Nearest neighbours of a query*

The retrieval process for Sequential EWH is described in Algorithm-3. Scores of all the strings in the database is initialized to zero. Retrieval process can be divided into four steps. The score of each string is initialized to zero.

1) Find the buckets to search (buckets that have 0,1 or 2-bit error difference with query)
2) Update score of strings present in these buckets

**Algorithm 2** Parallel implementation

INPUT: Database of $b$ bit strings: $F$, Number of hash functions: $n$, key length: $h$

OUTPUT: Hash table of $2^h$ rows and $n$ columns: $T$

```
1) for each string $f$ in the database do
   (This $for$ loop is executed by each thread in parallel)
    a) for $j = 1$ to $n$ do
        i) Get the hash value(bucket number) of $f$ for function $j$
        ii) Atomically increment size of corresponding bucket
        iii) Store the index of string in database in bucket
    b) end for
2) end for
```

**Algorithm 3** Sequential EWH retieval

INPUT: Query fingerprint $q$, fingerprint(binary strings) database $F$, hash table $T$, a set of weights $\alpha_r$ $(0 \leq r)$, the similarity threshold $s_0$

OUTPUT: Nearest neighbor(s) of $q$

```
1) Initialize the score to 0 for all the fingerprints in the database
2) for $i = 1$ to $n$ do
    a) Find $h_0$, the hash value for $q_{k_i}$
    b) Add $\alpha_0$ to the score of each fingerprint in column $i$ and row $h_0$
    c) for $r = 1$ to $e$ do
        i) Find $h_r$, the set of hash values for all binary vectors that have $r$-bit difference with $q_{k_i}$
        ii) Add $\alpha_r$ to score of all fingerprints in buckets in column $i$ corresponding to rows $h_r$
    d) end for
3) end for
```

3) Compute exact distance from query for strings which have score greater than a threshold
4) Find the top $k$ nearest among these strings

*1) **Find buckets to be searched:*** The first step is to find the buckets of hash table which have 0,1 or 2-bit error with the given query. This task solely depends on the query and is a light weight task. $h$-bit hash vector (corresponding to one hash function) of query will have one $h$-bit vector which matches with it exactly, $h$ vectors which match with 1-bit error and $h_{C_2}$ vectors which match with 2-bit error. Total number of buckets to be searched will be $(1 + h + h_{C_2})$ for each hash function. This step can be considered for CPU-GPU overlap i.e., when GPU is busy doing the other three steps of retrieval, CPU can compute the first step of the next query.

*2) **Update scores of strings:*** Next step is to update scores of all strings in the buckets computed. The motivation for this can be explained as follows. Lets us say a string $s_1$ matches with 0-bit error for one hash function and doesn't match at all for the rest. String $s_2$ matches with 1-bit error for two of the hash functions. i.e., $s_1$ matches for $h$ bits with query while $s_2$ matches $2 * (h - 1)$ bits. The probability of $s_2$ being nearest neighbour is higher than $s_1$ and should be considered for exact matching in the next step.

Each string occurs exactly once in the column of hash table corresponding to a hash function. If we try to update strings from all hash functions simultaneously, it will lead to inconsistencies. This is because the string could be present in selected hash buckets of different hash functions. One simple strategy is to update scores for each hash function one after the other. Another would be to use $atomic$ functions and update them in parallel. If the number of hash functions used are high in number, the first strategy will take more time. In this work, we have used the second strategy. A certain number of blocks of the kernel would be assigned to update scores of strings in one bucket. Threads in a single block access consecutive locations of hash bucket. Since we have stored the indices of strings belonging to a bucket in consecutive locations in an array, the thread accesses will be coalesced.

*3) **Computation of exact distance from query for shortlisted strings:*** Once scores have been updated, the strings which are greater than threshold have to be shortlisted for further search. One simple strategy to do this is to assign one string per thread to check whether its score is above threshold and then calculate the exact distance from the query. But this will lead to severe load imbalance because many threads would just check for the condition and when the score is below threshold would be idle. Hence a group of strings are assigned to each thread. One way to do this is to assign consecutive strings to a thread. Another way is to assign consecutive strings to threads with consecutive thread-ids just like round-robin strategy. The second method should work well because it will lead to memory accesses which are coalesced. So, each thread checks whether the score is above threshold and brings the index of corresponding string in database to shared memory of the block.

The next step is to compute the exact distance from query i.e., to compare bit-by-bit. A single thread could be assigned to do this for one string. But the total no.of strings shortlisted in a block of threads could be less than the total number of threads leading to some threads being idle. Better strategy is to use $b$ threads (where $b$ is the dimension of strings in database) to compare and then do a reduce operation to compute the exact distance. This computation is the bottleneck of EWH algorithm and using an efficient strategy should boost overall performance of algorithm. The result is stored in the shared memory and will be used in the next step.

*4) **Top $k$-nearest neighbours among shortlisted strings:*** We need to find the top $k$ strings which are close to the query. Sorting of all these selected strings is not necessary since the value of $k$ is expected to be small compared to the data size. Hence, we use a method which computes the rank of strings in the array. The set of strings positioned at $thread - id$ multiples are assigned to a thread. For each string, a scan through the array will give the number of strings having $hamming\ distance$ less than itself. The scan can be stopped once the count reaches $k$. Since we have already stored the distance of each string in the shared memory, we first compute $k$ best among the strings in shared memory. Then copy them to global memory and compute the global rank in another kernel. Another kernel has to be launched becaue we need synchronization among blocks. While finding the top $k$ in an array stored in global memory, each thread could start its scan from starting of the array. But a better thing to do is each thread starts the scan from its $global - id$ position so that thread all the threads will be accessing consecutive locations and hence the accesses would be coalesced.

## C. *Overlap of CPU and GPU*

The system to which GPU is connected would typically be a multi-core system. We can load the cores on CPU with some work to completely use the hardware available. Work distribution between CPU and GPU can be a tricky task and often depends on the relative speeds. It varies from system to system and needs to be determined experimentally. In present work, for Hashing (builing of hash table) we did some trial and error to divide the work between CPU and GPU. We found that about 6% of work done on CPU is optimal for the system we worked on.

## IV. Experiments and Results

The experiments are done with a database of 4-million 512-bit strings. The number of hash tables considered are four. Each of the hash function contains 4 bits and hence the hash table for each function contains 16 buckets. Total size of hash table is $64$ $(4X16)$. Experiments were performed for 5 cases i.e., Sequential LSH(baseline), MPI LSH (as in [2]), Sequential EWH, CUDA EWH and CUDA+OpenMP EWH.

## A. *System configuration*

*1) CPU:*

Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
32 cores with hyperthreading [16 real cores]

Cache size : 20480 KB
MemTotal : 132264384 kB

*2) GPU:* Tesla K20m

## B. Results

Results obtained for hashing and retrieval of one query are tabulated in the Tables below. The results were compared with the distributed LSH algorithm proposed in [2] which uses $n$ processors for $n$ hash functions. Since we use 4 hash functions, the number of CPUs used for MPI in 4. Each processor builds its own hash table for a hash function. Speed-up is calculated w.r.t sequential LSH. For hashing, speed-up obtained is much higher because it is highly data parallel. LSH and EWH follow the same strategy for hashing and hence we see that the time taken is almost same.

| Experiment | Hashing (s) | Speed-up |
|---|---|---|
| Seq. LSH | 26.6 | - |
| MPI LSH | 10.5 | 2.5 |
| Seq. EWH | 26.9 | - |
| CUDA EWH | 0.7 | 38 |
| CUDA+OpenMP EWH | 0.6 | 44 |

| Experiment | Retrieval (s) | Speed-up |
|---|---|---|
| Seq. LSH | 19.3 | - |
| MPI LSH | 6.4 | 3 |
| Seq. EWH | 7.6 | 2.5 |
| CUDA EWH | 0.8 | 24 |
| CUDA+OpenMP EWH | 0.8 | 24 |

## V. Conclusion

Results show-case the speed-up that can be obtained by using EWH on GPUs for Approximate k-Nearest neighbor search. We Distributed 6% of GPUs independent workload to CPU for better resource utilization. Speedup of 44 times on a 16 core GPU and 16 core CPU machine was achieved in case for hashing and 24 times for retrieval.

The possibility of using multiple GPUs to further accelerate the execution time could be explored in future. We also plan to devise parallelization strategies for the cases where database doesn't fit in GPU memory.

## Acknowledgment

The authors would like to thank Prof. Sathish Vadhiyar, Super-computer Education and Research Centre(SERC), Indian Institute of Science who guided us during the project. We would also like to thank fellow students of MARS lab, SERC.

## References

[1] Mani Malek Esmaeili, Rabab Kreidieh Ward, and Mehrdad Fatourechi. A fast approximate nearest neighbor search algorithm in the hamming space. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(12):2481–2488, 2012.

[2] Smita Wadhwa and Pawan Gupta. Distributed locality sensitivity hashing. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–4. IEEE, 2010.

[3] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. Efficient distributed locality sensitive hashing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2174–2178. ACM, 2012.

[4] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[5] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832, 2004.

[6] 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.