Parallelization of Error Weighted Hashing for Approximate k-Nearest neighbour search

B M Pavan Kumar (SR: 06-02-01-10-51-12-1-09240)

SuperComputer Education and Research Centre, IISc-Bangalore

***Abstract*—Error Weighted Hashing (EWH) is an efficient algorithm for Approximate k-Nearest neighbour search in Hamming space. It is more efficient than Locality sensitive Hashing algorithm (LSH) as it generates shorter list of strings for finding exact distance from the querry. Parallelization strategies for EWH algorithm using CUDA are discussed. The results are compared with distributed LSH algorithm proposed in [2].**

***Index Terms*—Hamming space, Nearest neighbour search**

I. Introduction

Nearest neighbour search in Hamming space is the problem of finding closest matches of a given querry string, given a set of reference strings. Given a b-bit binary string (i.e, contains 0 or 1 in all bits), the problem is to find stings in the database which match with it closely in the database. A string is said to be matched closely with other string if it has the same values (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. The aim is to find approximate matches very fast. As the database becomes larger the searching could become a bottleneck. Parallel implementations would make the search fast.

Normal searching methods like linear search or binary search do not work here due to huge size of data. Hence hashing is generally used for this problem. LSH is a popular algorithm for nearest neighbour search. Recently, EWH algorithm was proposed in [1] which modifies the process of retrieval by giving scores to strings in the database and is better in terms of generating the shortlist of strings which will be checked for exact mathces. 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 CPU. The key idea is to use the parallel computing power of the GPU to achieve significant speedups. Hashing is a problem where the hash value of a string doesn't depend on the other. So, the massely large amount of threads available on a GPU could be exploited to obtain good performance.

In the next section LSH, EWH and distributed LSH proposed in [2] will be briefly described. In section III, the strategies used for parallelising EWH are discussed. In section IV, experiment details and results are shown. The report is concluded in section V.

II. Related Work

Hashing strings to hash table is done in the following way. If we consider b-bit binary vectors, 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 is formed from the given string. Its value in decimal representation will become the hash value. For a h-bit hash function there will be 2^h hash buckets. Thus for n-hash functions, total hash table consists of 2^h rows and n colums. As the number of hash functions increase, we are considering many bits and hence the result will be more accurate. But as the number of hash functions increase the time complexity increases. A string which goes into different buckets with respect to different hash functions as shown in Figure 1. Given a reference string f and considering hash functions k_1,k_2,k_3....k_n with h=4 bits, the following figure shows how the hashing is done for that string.

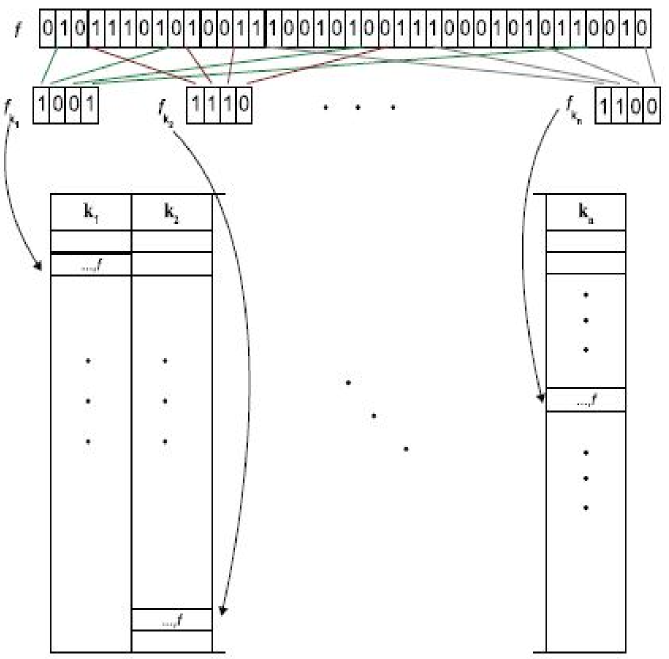

Figure 1. Hash table construction with reference string f and h=4 bits and n hash functions

Mani et al. [1] proposed the Error weighted hashing algorithm which basically assigns scores to the strings depending on how close they are from the querry string. Given a querry string, the algorithm considers exact matching (i.e, 0 bit difference), 1-bit difference and 2-bit difference strings with respect to a hash function. It adds scores to all the strings which have the corresponding hash value (i.e, corresponding bucket in hash table). It adds a high score to exact matching strings and a low score to 2-bit difference strings. The next step is to retrieve all the strings which have score greater than a particular threshold. The EWH algorithm differs from LSH in the way it retrieves the data. LSH doesn't have this score concept and ends up computing exact distances for all the strings in all the buckets. The number of strings which LSH checks for exact match is higher than the number which EWH does. Since this is the bottleneck, LSH ends up in high execution times. Though there is an additional overhead in computing the scores for each string in the database, the number of strings it checks for computing exact distance is less and hence is expected to perform well.

Smita and Pawan [2] proposed an algorithm for parallelising LSH using distributed systems. Basically the task of constructing the hash tables is distributed among the processors. Each processor constructs a hash table independently. Given a querry string, it is broadcasted to all the processors. Now, with respect to this hash table is searched and a shortlist is retrieved. Then, the k-best among them are selected. This method of having part of hash tables in different processors would lead to load imbalance because the size of shortlist on different processors would be different.

EWH algorithm has the advantage of generating a smaller shortlist and hence is better than LSH. Parallelising EWH would improve the performance even better. CUDA has shared memory architecture and hence a single hash table can be constructed. The objective of this project is to devise parallelising strategies for EWH.

III. Methodology

EWH algorithm can be divided into two stages. One is hashing and the other is retrieval. For hashing the data has to be moved into the global memory. To construct the hash table, the number of strings that would go into each bucket are not known apriori. An array to store the starting indices for each bucket in the hash table is maintained. The data is scanned to find the number of strings which would go into each bucket. Each thread acts on a string and finds out the bucket to which it goes. Once we get these numbers, a prefix scan would give the starting indices of each each bucket with respect to a hash function. For different hash functions this can be done parallelly. Once these pointers for each bucket are known, the strings are actually hashed into the respective buckets by using offset values for each bucket i.e, when a string is to be placed into a bucket, its index is calculated by adding the starting index of the bucket and the offset in that bucket. Offset determines the number of strings already inserted into the bucket. One more issue here is that two threads acting on different strings that would go into same bucket could access the offset variable at same time leading to inconsistencies. Hence this offset increment is made atomic using atomic increment function. Since each thread acts on consecutive strings, the acceses can be coalesced.

Given a querry, the retrieval step can be further divided into four stages, one to find out which buckets to search(which buckets have 0,1 or 2-bit difference with querry), second updating the scores for strings, third is computing exact distance for strings which have score greater than a threshold value and the last one to find the top k nearest strings among them. These four steps are executed on four different kernels.

The firts step, i.e to find which buckets are at 0,1 or 2-bit difference solely depends on querry and is a light weight task. This could be done on cpu and hence can be considered for cpu-gpu overlap which will be discussed later. h-bit hash functions will have one h-bit vector which matches with it exactly, h different vectors which match with 1-bit difference and h_{C_2} vectors which are 2-bit different. For doing this, n blocks are launced for n- different hash functions. The first thread computes the bucket number which matches exactly for that particular block. The next h threads compute bucket numbers which are 1-bit different from querry. The next h_{C_2} threads compute the bucket numbers which are 2-bit different. All these numbers are stored in the global memory and the next kernel updates scores for all the fingerprints in these corresponding buckets.

The next step is to update scores of all the strings in the buckets computed in the previous kernel. Each string occurs exactly once in a particular hash table corresponding to a single 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. Other strategy is to use atomic functions and update them parallelly. If the number of hash functions used are high in number, the first strategy will take more time. In this project, I have used the second strategy which is working fine. The number of blocks to be assigned to a bucket depends on the data size. On an average the number of strings in each bucket is equal to $\frac{datasize}{no.of buckets}$. From this, the number of blocks to be assigned to each bucket is computed. Since we have stored the indices of strings belonging to a bucket in consecutive locations in an array, the thread accesses will be coalesced.

Once the 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 querry. But this will lead to severe load imbalance because many threads would just check for the condition but because the score is below threshold will be idle. Hence a group of strings are assigned to each thread. One way to do this is assign consecutive strings to each thread. The other is to assign consecutive strings to consecutive threads just like round-robin strategy. This 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 corresponding string index to shared memory of the block. The next step is to compute the exact distance from querry which is to compare bit by bit. One thread could do this for one string. But the total no.of strings selected by a block could be less than the number of threads. A better strategy is to use b threads (b is the dimension of the vector) to compare and then do the reduce operation to compute the distance. This computation is a bottleneck and using a good strategy like this should give better performance. For all the strings in shared memory this computation is performed and exact distance is stored in shared memory.

Next we need to find the global top k strings which are close to the querry. Sorting of all these selected strings is not necessary since the value of k is very small compared to the data size. Hence we use a method which basically computes the rank of the string in the array. To do this each thread scans through the array and finds how many strings are having distance less than it. Once the count reaches k it breaks from the search. 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 thread all the threads will be accessing consecutive locations and hence the accesses will be coalesced. In all the kernels discussed above the computation done in kernel is used by next kernel i.e, the next will not compute it again. Incase of multiple querries, while the gpu is computing the nearest neighbours for a querry the cpu can be given the light weight task as discussed earlier. The task of finding which buckets to search i.e, buckets that are at 0,1 and 2-bit difference from the querry can be done on cpu improving the total execution time.

IV. Experiments and Results

The experiments are done with 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. The experiments are repeated with number of strings varying from 50,000 to 2,50,000 in intervals of 50,000.

The results were compared with the distributed LSH algorithm proposed in [2] which uses n processors for n hash functions. Here, since I am using 4 hash functions the number of processors used in 4. Speedup obtained are close 11 over the sequential version while the speedup is close to 2 incase of distributed LSH. The speedup obtained for hashing for gpu version are close to 25 while it almost constant at 3 for the mpi version. This might be because hashing is highly data parallel. The mpi programs are on 4 processors (since no.of hash functions taken are 4) where each processor builds its own hash table and hence the speed ups are close to 3.

V. Conclusions and Future Work

EWH algorithm is parallelised using CUDA. Different startegies for parallelising the steps of the algorithm are discussed. Distributed LSH algorithm proposed in [2] is implementd in MPI. The results obtained are compared with the distributed LSH algorithm. For multiple querries better methods using cpu-gpu overlap to effectively use the resources

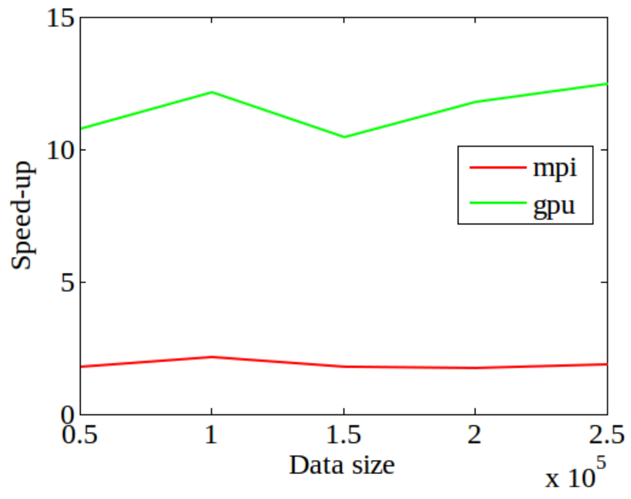

Figure 2. Speedup obtained for retrieval time per querry for EWH gpu version compared against distributed LSH version

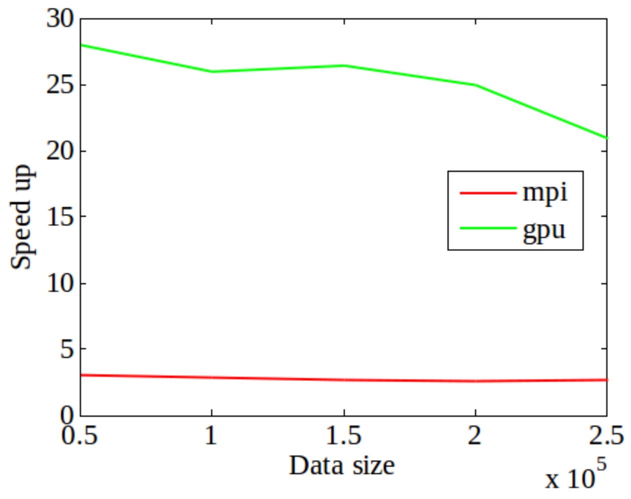

Figure 3. Hashing speedup obtained EWH gpu version compared against distributed LSH version

are discussed. If the data size is too large that it can't fit in gpu memory, the possibility of using multiple gpus could be explored in future. The performance of the algorithm using real databases could be tested. Other LSH based algorithms implements in gpu can be used to evaluate the efficiency of parallelisation.

References

1) "A Fast Approximate Nearest Neighbor Search Algorithm in the Hamming Space" Mani Malek Esmaeili and IEEE, Rabab Kreidieh Ward, Fellow, IEEE, and Mehrdad Fatourechi, Member, IEEE.
2) "Distributed Locality Sensitivity Hashing." Smita Wadhwa and Pawan Gupta. IEEE CCNC 2010 proceedings.