Solving k-Nearest Neighbor Problem on Multiple Graphics Processors

Kimikazu Kato and Tikara Hosino

Nihon Unisys, Ltd.

Abstract—A recommendation system is a software system to predict customers' unknown preferences from known preferences. In a recommendation system, customers' preferences are encoded into vectors, and finding the nearest vectors to each vector is an essential part. This vector-searching part of the problem is called a k-nearest neighbor problem. We give an effective algorithm to solve this problem on multiple graphics processor units (GPUs).

Our algorithm consists of two parts: an N-body problem and a partial sort. For a algorithm of the N-body problem, we applied the idea of a known algorithm for the N-body problem in physics, although another trick is need to overcome the problem of small sized shared memory. For the partial sort, we give a novel GPU algorithm which is effective for small k. In our partial sort algorithm, a heap is accessed in parallel by threads with a low cost of synchronization. Both of these two parts of our algorithm utilize maximal power of coalesced memory access, so that a full bandwidth is achieved.

By an experiment, we show that when the size of the problem is large, an implementation of the algorithm on two GPUs runs more than 330 times faster than a single core implementation on a latest CPU. We also show that our algorithm scales well with respect to the number of GPUs.

I. INTRODUCTION

A recommendation system is a software system which utilizes known customers' preferences to predict unknown preferences. It is widely used in Internet-based retail shops and other service providers, such as Amazon.com [1] for example. In a recommendation system, customers' preferences or buying patterns for items are encoded into vectors and finding nearest vectors is an essential part of its computation. This vector-finding part is called a k-nearest neighbor problem. We give an effective GPU algorithm to solve this problem.

Generally a recommendation system deals with large samples and large dimensions, as person×item for example. In such a case, the dimensionality reduction method such as singular value decomposition or latent Dirichlet allocation has been widely used [2], [3], [4]. As the result of the reduction, the problem becomes the k-nearest neighbor search for a moderate dimension. However, the effect of the sample size n is $O(n^2)$ and it is a computational burden. Therefore some approximation has been considered to be necessary [5]. This paper indicates that strict computation in practical time is possible. Our target size for n is $\sim 10^6$ to $\sim 10^8$, for the dimension after reduction is $\sim 10^2$ to $\sim 10^3$.

The k-nearest neighbor problem is defined as follows: when a set of vectors $v_1 \dots v_n \in \mathbb{R}^d$, distance function δ and an integer k is given, find k nearest vectors to each v_i. We propose an effective and scalable algorithm to solve it on

multiple Graphics Processor Units (GPUs). Our algorithm is implemented in CUDA [6], which is extension of C language provided by NVIDIA.

A GPU is a powerful commodity processor. Although a GPU is originally designed for processing of graphics, the movement of the GPGPU (General Purpose computing on GPU) has arisen as an expected breakthrough for a large scale numerical computation. The typical characteristic of the GPGPU is highly massive parallelism. A GPU has hundres of cores, and to extract its power, it is necessary to run tens of thousand of threads per unit. Because of that property, a GPU consumes large energy as a unit, but it is energy effective per FLOPS.

The algorithm of the k-nearest neighbor problem is fundamentally a combination of N-body problem and partial sorting. Nyland et al. [7] showed an effective algorithm for N-body problem on CUDA. Because dealing with high dimensional vectors, we give some trick in addition to the known N-body algorithm. About sorting, [8] showed an effective algorithm, but we have employed another algorithm because we have to sort many arrays at once and we only need to have top k element not fully sorted data.

Garcia et al. [9] showed a GPU algorithm to compute the k-nearest neighbor problem with respect to Kullback-Leibler divergence. Their algorithm mainly uses a texture memory, which in effect, works as a cache memory. Its performance largely depends on the cache-hit ratio, and for a large data, it is likely that a cache miss occurs frequently. On the other hand, our algorithm utilizes maximal power of coalesced memory access, so that such loss as a cache miss never happens. Moreover, our algorithm is effective even for a symmetric distance function and for multiple GPUs.

The rest of this paper is organized as follows. In Sect. II, outline of CUDA's programming model is explained. In Sect. III, we define the problem formally. We give overview of the algorithm in Sect. IV. In Sect. V and VI, we explain the detail of each step of the algorithm. In Sect. VII, we show the result of experiment. We conclude in Sect. VIII.

II. PROGRAMMING MODEL OF CUDA

In this section, programming model of CUDA is briefly explained. For more details of CUDA, refer to [10].

a) Thread model.: NVIDIA's recent graphics processor contains hundreds of *stream processors* (SPs). An SP is like a core in a CPU; it can compute simultaneously. For example, GTX280 has 240 SPs. With such many SPs and very low cost

of context switch, a GPU performs well for tens of thousands of threads. Threads are divided into thread blocks. Each thread block can contain at most 1024 threads. A function to synchronize threads in a block is provided, while there is no such function to synchronize thread blocks. The only way to synchronize thread blocks is to bring back the control to the CPU.

b) Hierarchal memories.: Before running a GPU, the CPU must explicitly copy data to the GPU's memory. The memory in GPU to share the data with CPU is called *global memory*. A thread block is also a unit to share data. Each thread block has a memory to share only in the thread block. It is called *shared memory*. The access to the global memory is relatively slow, and usually copying necessary data to shared memory is better for performance. Although global memory has some gigabytes, shared memory has only 16KB for each thread block. Each thread also has a local memory which is called a *register*. The access to a register is fast, but its size is also limited.

c) Coalesced memory access.: In CUDA, for example, if successive 16 threads are accessing the successive 128 bytes in global memory at the same time, the memory access is coalesced. When a memory access is coalesced, it is done in only one fetch while otherwise access by 16 threads takes 16 fetches. Hence, effective utilization of coalesced memory access affects very much the total performance of an application. The detailed condition about when memory access can be coalesced is explained in [10].

III. DESCRIPTION OF THE PROBLEM

The k-nearest neighbor problem is described as follows.

Suppose that a set of vectors $v_1, \cdots, v_n \in \mathbb{R}^d$ and distance function $\delta : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is given. Then output the k nearest vectors to each v_i.

In other words, for each i, find a subset of indices $\{j_{i1}, \ldots, j_{ik}\} \subset \{1, 2, \ldots, n\}$ such that

$$\delta(v_i, v_{j_{i1}}) \leq \delta(v_i, v_{j_{i2}}) \leq \cdots \leq \delta(v_i, v_{j_{ik}})$$

and

$$\delta(v_i, v_{j_{ik}}) \leq \delta(v_i, v_j) \text{ for all } j \notin \{j_{i1}, \ldots, j_{ik}\}$$

The distance function δ is arbitrary. Although we use the word "distance", it does not necessarily need to satisfy the axiom of distance. We assume that δ is *cumulatively computable*. It means δ can be computed step by step by referring to each coordinate values. In other words, it is computed with a function $\bar{\delta} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and some initial value a_1 by $a_i = \bar{\delta}(u_{i-1}, v_{i-1}, a_{i-1})$ and $\delta(u, v) = a_{n+1}$.

In this paper, we only discuss the case when δ is symmetric: i.e. $\delta(u, v) = \delta(v, u)$. In the symmetric case, we can omit the half of distance calculations, and consequently, balancing of the workload becomes more difficult. The algorithm explained in this paper is easily modified for non-symmetric distance function.
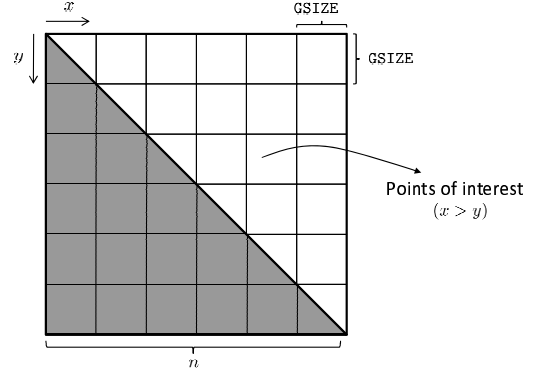

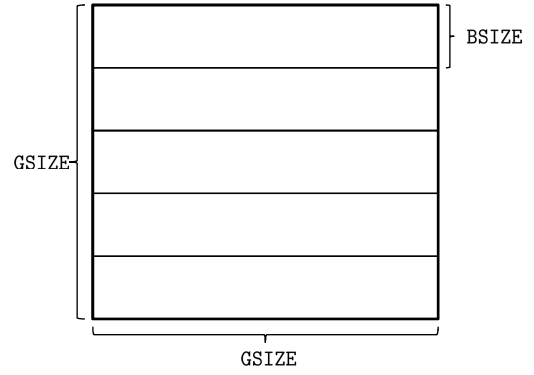Fig. 1.   First level division of the problem

Fig. 2.   Devision of a grid

IV. OVERVIEW OF THE ALGORITHM

Since we have assumed that δ is symmetric, we only compute $\delta(v_x, v_y)$ for $x > y$. For an explanation, we depict the whole problem as a square where the point (x, y) stands for the computation of $\delta(v_x, v_y)$. The distances to compute is represented by upper right triangle of the square.

Because of the limitation of the number of threads which can be run at once on GPU, the problem is divided into a first level blocks. We call each of them a *grid* (Fig. 1). Each grid is processed in a GPU at once. A grid can be divided row-wisely into *blocks*, each of which is computed in a thread block. We denote the size of each side of a grid by GSIZE. It means the grid (X, Y) stands for the region $\texttt{GSIZE} \cdot X \leq x < \texttt{GSIZE} \cdot (X+1)$, $\texttt{GSIZE} \cdot Y \leq y < \texttt{GSIZE} \cdot (Y+1)$. Similarly we denote the size of a block (i.e. the number of rows in a block) by BSIZE (Fig. 2). GSIZE is determined depending on n so that the problem can be devided effectively, while BSIZE is fixed according to the capability of CUDA.

To balance the workload, we assign GPUs as in Fig. 3. In other words, the i-th row of grids is assigned to j-th GPU when $i \bmod 2 \cdot \texttt{nDevices} = j$ or $i \bmod 2 \cdot \texttt{nDevices} = 2 \cdot \texttt{nDevices} - j - 1$, where nDevices means the number of GPUs available. Here note that although it is enough to compute the upper-right part of the problem, each GPU virtually compute the mirror side of the assigned part (see also Fig. 4)
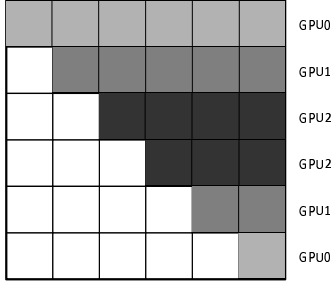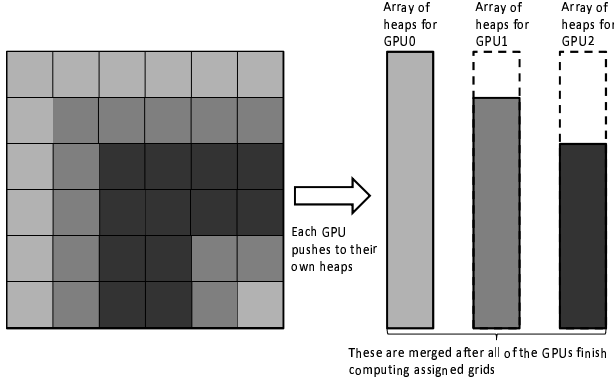
Fig. 3. Assignment for GPUs

Fig. 4. Heaps for GPUs

To keep the k-nearest vectors, we use a heap structure. The heap has at most k elements and is in descending order, so that the k-th smallest element can be found in $O(1)$. Moreover, each GPU keeps their own heaps to avoid a costly synchronization (Fig. 4). It means each GPU has n heaps which stores the k nearest elements computed by itself. At the last phase, the heaps of different GPUs are merged in CPU.

Thus the outline of the algorithm is shown in Fig. 5. In this algorithm, the calclation of distances is explained in Sect. V, and how to push the distances to the heaps is described in Sect. VI.

V. PHASE 1: CALCULATION OF DISTANCES

Basically the framework of the process to compute the distances of vectors is the same as the algorithm of N-body problem written in [7]. A grid is row-wisely devided into blocks, and each block is assigned to a thread block. Each thread corresponds to a row. A block first copies a fixed number (which we denote by C1) of columns to the shared memory. Then compute the distances.

However, in our problem, since the dimension d is large, it is not possible to copy all the coordinate data to the shared memory even for a small C1. Hence, a thread iteratively reads a fixed number C2 of coordinate values of corresponding vectors. In other words, if v_i is expressed as $(v_i^{(0)}, \ldots, v_i^{(d-1)})$, then $v_i^{(j \cdot \text{C2})}, \ldots, v_i^{(j \cdot (\text{C2}+1)-1)}$ are read in j-th iteration (Fig. 6). If a vector is expressed by single precision numbers, C2 must be a multiple of 32 to utilize full power of coalesced memory accesses.

procedure ThreadMain(n,d,$\{v_i\}$)
 nGrids $\leftarrow \lfloor (n-1)/\text{GSIZE} \rfloor + 1$
 Prepare the heaps $\{h_i\}_{i=0}^{n-1}$
 for $Y := 0$ **to** nGrids $- 1$ **do**
   for $X := 0$ **to** nGrids-1 **do**
     if $i \bmod 2 \cdot \text{nDevices} = j$
       or $i \bmod 2 \cdot \text{nDevices}$
       $= 2 \cdot \text{nDevices} - j - 1$ **then**
     Calculate the distances for the grid (X, Y)
     Push the i-th row of distances
       to h_i for the grids (X, Y) and (Y, X)
     end if
   end for
 end for
end procedure

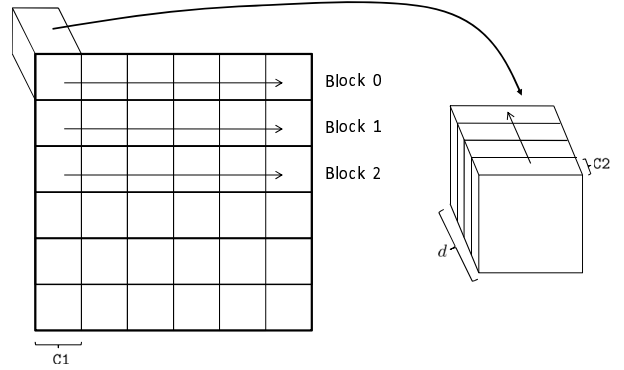Fig. 5. Overall algorithm: each GPU is assigned to CPU thread and its thread id is given by tid

Fig. 6. Illustration of the algorithm to compute the distaces of d dimensional vectors

The algorithm to calculate the distaces for a given grid is shown in Fig. 7. Here, the arguments n_1, $\{v_{1i}\}_{n=0}^{n_1-1}$, n_2, and $\{v_{2i}\}_{i=0}^{n_2-1}$ are given as re-indexed $\{v_i\}$ so that this procedure can calculate for the assigned grid. The index for the block is expressed by bid, and each block has BSIZE $\times$ C1 threads. Each thread is indexed by (tx, ty)

VI. PHASE 2: TAKING k SMALLEST ELEMENTS

In the second phase, each thread block is assigned to each row. The smallest k distances are computed by parallel processing of threads in the block. If the number of thread in a block is denoted by nThreads, each thread read distances skipping nThreads, so that memory access is coalesced. A thread check if the element is smaller than current k-th largest element in the heap, and store it in the local buffer if so. This is because k is relatively small than n and it is likely that only a few elements is stored in the local buffer. Because of this mechanism, the waiting time is shortened even though when pushing to the heap, the threads must be synchronized.

The algorithm is shown in Fig. 8. Here, the index for the block and thread is denoted by bid and tid respectively, and

procedure CalcDistances($d,n_1,\{v_{1i}\}_{i=0}^{n_1-1}, n_2, \{v_{2i}\}_{i=0}^{n_2-1}$)
  $bx \leftarrow 0$
  Prepare the shared memory to store the distances
  while $bx \cdot \texttt{C1} < n_1$ **do**
    $l \leftarrow 0$
    while $l < d$
      Copy $v_{1i}^{(k)}, v_{2j}^{(k)}$
        ($bx \leq i < bx + \texttt{C2}$,
        $bid \cdot \texttt{BSIZE} \leq j < (bid+1) \cdot \texttt{BSIZE}$,
        $l \leq k < l + \texttt{C2}$) to the shared memory
      Calculate cumulatively all the combinations of
        v_{1i} and v_{2i} which are in the shared memory
        and store it in a local resister $dist$
      $l \leftarrow l + \texttt{C2}$
    end while
    Store the resulting distance $dist$ in the global memory
    $bx \leftarrow bx + \texttt{C1}$
  end while
end procedure

Fig. 7. Algorithm for calculation of distances: for simplicity, it is assumed that n_1 is multiple of $\texttt{C1}$ and d is multiple of $\texttt{C2}$

$\texttt{buffer}$ is thread-local array and its size is $\texttt{bufsize}$.

procedure KSmallest ($k,n,m,\{h_i\}$: heaps,
                  $\{a_{ij}\}_{0 \leq i < m-1, 0 \leq j < n-1}$)
  for $i := tid$ **to** $n-1$ **step** $\texttt{nThreads} \cdot \texttt{bufsize}$ **do**
    for $j := 0$ **to** $\texttt{nThreads} \cdot \texttt{bufsize}$ **step** $\texttt{bufsize}$ **do**
      for $l := 0$ **to** $\texttt{bufsize}$ **do**
        $\nu \leftarrow i \cdot \texttt{nthreads} \cdot \texttt{bufsize} + j + l$
        if $a_{bid,\nu}$ is smaller than top of the heap h_{bid} **then**
          Store $a_{bid,\nu}$ to $\texttt{buffer}$
        end if
      end for
      Push elements of $\texttt{buffer}$
        to h_{bid} (blocking other threads)
    end for
  end for
end procedure

Fig. 8. Algorithm to get k-smallest numbers from multiple arrays

VII. EXPERIMENT

We experimented our algorithm on two GTX280's and one GTX280. For a comparison, we also implemented CPU version and experimented it on Intel i7 920 (2.67GHz). GTX280 is one of the latest NVIDIA's graphics chips. The algorithm experimented on the CPU is a simple one: it calculates each $\delta(v_x, v_y)$ $(x > y)$ and pushes it to the corresponding heaps. Note that although Intel i7 has four cores with hyperthreading capability, we only worked on serial algorithm, i.e. it only uses one core.

The distance employed here is Hellinger distance, which often used in the context of statistics. Hellinger distance for

TABLE I
ELAPSE TIME FOR k-NEAREST NEIGHBOR PROBLEM (SEC)

n	10000	20000	40000	80000
2× GTX280 (a)	1.8	5.7	17.7	68.6
1× GTX280 (b)	2.7	8.6	34.1	131.8
i7 920 (CPU) (c)	354.2	1419.0	5680.7	22756.9
(c)/(a)	196.7	248.9	320.9	331.7
(c)/(b)	131.1	173.3	166.5	172.6
(b)/(a)	1.50	1.51	1.92	1.92

two vectors u and v is defined as:

$$\sum_i \left(\sqrt{u^{(i)}} - \sqrt{v^{(i)}} \right) \quad (1)$$

The result of the experiment for various n is shown in Table I. The other parameters are set as $k = 100$ and $d = 256$; and the data is generated randomly. It shows that for a large problem, our algorithm work well from the viewpoint of parallelism of GPUs. Moreover, it also tells the GPUs substantially outperforms the CPU; for a large problem, two GPU implementation is more than 330 times faster than the CPU.

VIII. CONCLUSION

We introduced an effective algorithm for k-nearest neighbor problem which works on multiple GPUs. By an experiment, we have shown that it runs more than 330 times faster than an implementation on a single core of an up-to-date CPU. We have also shown that the algorithm is effective from the viewpoint of parallelism of GPUs. That is because 1) there is no synchronization between GPUs until the very end of the process and 2) the workload is well balanced.

Our algorithm includes simultaneous partial sort of multiple arrays. It minimizes the inter-thread synchronization utilizing the fact that if $k \ll n$, most of the data are discarded. About this part of algorithm, we have achieved a good performance but still there is a room for improvement because it uses arrays in a local scope which are stored in a slow global memory in effect. To improve the performance of the simultaneous partial sort is our ongoing work, and we believe this problem alone is also important because it can be applied to other problems.

REFERENCES

[1] Amazon.com, http://www.amazon.com.
[2] M. Brand, "Fast online SVD revisions for lightweight recommender systems," in *In SIAM International Conference on Data Mining*, 2003.
[3] D. M. Blei, A. Y. Ng, M. I. Jordan, and J. Lafferty, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, p. 2003, 2003.
[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW '07: Proceedings of the 16th international conference on World Wide Web.* New York, NY, USA: ACM, 2007, pp. 271–280.

[5] P. Indyk and W. Miller, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the 1998 Symposyum on Theory of Computing*, 1998.

[6] NVIDIA, "CUDA Zone," http://www.nvidia.com/object/cuda_home.html.

[7] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with CUDA," in *GPU Gems III*. NVIDIA, 2007, pp. 677–695.

[8] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 246–258.

[9] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *In Proceedings of Conference on Computer Vision and Pattern Recognition Workshop (CVPRW)*, no. 23–28, 2008, pp. 1–6.

[10] NVIDIA, "CUDA 2.1 programming guide," http://www.nvidia.com/object/cuda_develop.html, 2008.