Parallel Search of k-Nearest Neighbors with Synchronous Operations

Nikos Sismanis and Nikos Pitsianis
Department of Electrical and Computer Engineering
Aristotle University, Thessaloniki, Greece

Xiaobai Sun
Department of Computer Science
Duke University, NC, USA

***Abstract*—We present a new study of parallel algorithms for locating k-nearest neighbors (kNN) of each single query in a high dimensional (feature) space on a many-core processor or accelerator that favors synchronous operations, such as on a graphics processing unit. Exploiting the intimate relationships between two primitive operations, select and sort, we introduce a cohort of truncated sort algorithms for parallel kNN search. The truncated bitonic sort (TBiS) in particular has desirable data locality, synchronous concurrency and simple data and program structures. Its implementation on a graphics processing unit outperforms the other existing implementations for kNN search based on either sort or select operations. We provide algorithm analysis and experimental results.**

I. Introduction

A. Parallel kNN Search

The search for nearest neighbors to a given query point in a metric space [1], such as an image query in an image feature corpus, with a particular similarity measure, has been a primitive operation in statistical learning, interpolation, estimation, recognition or classification. It becomes prevalent with rapid and broad growth in volume or video data acquisition and analysis, in both conventional and new fields of quantitative studies. Design and development of algorithms for searching k nearest neighbors (kNN) remain active because its performance is critical to timely or real-time data processing and analysis, such as segmentation or retrieval of video images [2], [3], collaborative filtering [4], GIS-moving objects in road networks [5], network intrusion detection [6] and text categorization [7], to name a few. The desire and effort to accelerate data processing are undoubtedly encouraged by the radical shifts in computer engineering and architectures to available and affordable commercial parallel processors.

We introduce in this paper a new study of parallel algorithms for kNN search on a parallel processor or accelerator that favors synchronous operations and has high synchronization cost. We examine the intimate relationship between two related basic operations, sort and select, and introduce a cohort of truncated sort algorithms for kNN search. We use a graphics processing unit (GPU) by NVIDIA, instead of a customized accelerator, as the experiment platform, because the GPU resides in nearly every desktop or laptop computer and NVIDIA provides a convenient software programming environment and application interface[1]. The truncated bitonic sort (TBiS), in particular, renders outstanding performance on a GPU, which would be unexpected had one looked only at the conventional complexity in terms of logical comparisons taken by the full bitonic sort. We show that TBiS is advantageous in all other performance factors, it has desirable data locality, synchronous concurrency and regular data and program structures for efficient kNN search.

B. Associated and Related Work

There has been substantial work on accelerating kNN search, with various algorithmic or/and architectural means. For kNN search over a large data corpus, it is important to locate a near neighborhood first, utilizing both distance and population information. To this end, many algorithms use space partitions with tree-type data structures, and use in addition preprocessed reference data points. Tree-type data structures are typically associated with regular space partitions [8], [9], [10]. For a high dimensional space, however, algorithms based on regular space partitions suffer from high overhead costs [11] as the data distributions tend to be non-uniform and relatively sparse. Irregular spatial partitions, such as triangulation, seem exclusively limited to a two or three dimensional space [12]. Another popular approach is to use probabilistically approximate algorithms when certain approximation is acceptable and beneficial [13]. Approximate algorithms cluster or project data points of the high dimensional corpus to bins and then search the bins that are most likely to contain the nearest neighbors. The search time is sublinear in corpus data size. At the core of each main approach, however, is the exact kNN search over a smaller corpus, which is still large enough to take a significant or even dominant portion of the total kNN search time [14]. With these associated outer-layer methods in mind, we focus on the core case, the exact kNN search, for the rest of the paper.

In utilizing architectural means, many efforts have used GPUs (especially, those by NVIDIA) as many-core parallel processing units, available and affordable, with increased and increasing support of application interfaces [15], [16], [17],

[1] http://developer.nvidia.com/cuda-downloads

978-1-4673-1576-0/12/$31.00 ©2012 IEEE

[18], [19]. Most of the GPU implementations are based on certain sort algorithms, with some modification or customization. We give a few examples. Garcia *et al.* used the insertion sort and a parallel version of the comb sort [16]. Kuang and Zhao opted to sort the entire list [6] by using the radix sort. Kato and Hosino modified the heap sort with atomic operations for inter-thread synchronization [17]. They also mentioned the option of sorting the entire list by using the quick sort. Some explore other approaches. Barrientos *et al.* used the range search as the basic algorithmic step, with the help of priority queues [15]. Pan *et al.* introduced an approximate kNN algorithm, using the locality sensitive hashing technique, and implemented it on GPUs [20].

There are two common phenomena in previous, although diverse, efforts on parallel kNN search: the kNN search is taken as a single invocation of a stand-alone module procedure; each module procedure is singularly designed and isolatedly tuned. In the next section, we present a unified approach to truncating sort algorithms for kNN search. In the subsequent sections, we elaborate a particular one with surprisingly outstanding performance, and we extend our study to explore additional concurrency patterns as well as data reuse for more efficient kNN search in practical applications.

II. Truncated Sort Algorithms

In this section, we introduce truncated sort algorithms. Assume that we are provided with a corpus Ω of n feature vectors and a distance function or a similarity score function ϕ. The feature vectors may be, for example, the SIFT feature vectors [21] for the images in BelgaLogos data set [22]. For any test image, a region of interest (a subimage for logo identification for instance) may be submitted as a query to have its feature vector $\mathbf{y}$ extracted and k nearest neighbors located,

$$\max_{1\le i\le k} \phi(\mathbf{y},\mathbf{x}_i) \le \phi(\mathbf{y},\mathbf{z}), \quad \forall \mathbf{z}\in\Omega-\{\mathbf{x}_i, 1\le i\le k\}. \quad (1)$$

where k is a modest integer, substantially smaller than the corpus size n.

The computation of (1) can be done by an ascending sort of the n scalar elements $\phi(\mathbf{y},\mathbf{z})$, $\mathbf{z}\in\Omega$, followed by a truncation of the last $n-k$ elements. High-performance implementations of sort algorithms are made available on many computation platforms. But this approach of sort-and-truncate is expected to be outperformed by any kNN search algorithm [2]. We introduce a more efficient way to utilizing the sort algorithms: truncation *during* the sorting process.

Table I summarizes the truncated versions, in terms of the conventional time complexity, corresponding to a few distinguished sort algorithms. Each recovers the respective complete sort when $k=n$. We describe briefly for each how to truncate and reduce the data during the sort and how to parallelize the operations.

[2]We were surprised to find that some existing kNN algorithms fail to outperform the approach of truncation after a complete sort.

TABLE I: Truncated Sort Algorithms

Algorithm	**Serial**	**Parallel** (in length)	**Truncation Method**
BubbleSort	nk	$k(\log n-\log k+1)$	k reversal passes
MergeSort	$n\log k$	$k(\log n-\log k+1)$	eliminate half
InsertionSort	nk	$k(\log n-\log k+1)$	length-k array
HeapSort	$n\log k$	$k(\log n-\log k+1)$	max-heap of size k
QuickSort	nk	$k(\log n-\log k+1)$	eliminate half
RadixSort	nb	b	reverse radix (MSB)
BitonicSort	$n\log^2 k$	$\log k\log n$	length-k bitonic

- ≻ The truncated bubble sort takes k passes and n comparisons in each pass. It can be efficiently mapped into any synchronous parallel architecture as n/k systolic priority queues [23] that are pairwise merged/eliminated.
- ≻ The truncated merge sort starts with sublists of length one and merges them pairwise, in parallel, until the sublist length reaches k. Subsequent merges purge the upper half of each resulting list until there is only one list of length k left.
- ≻ The truncated insertion sort uses a k-length sorted list to hold the k smallest elements found so far. When an element smaller than the last on the list is found, the former is inserted at the right place by a scan operation and the latter is removed. The operations are parallelized by partitioning the corpus into multiple subcorpora, each subcorpus maintains a private k-length sorted list. These k-length lists are then merged pairwise, followed by a purge of the larger k elements. As the insertion into each list is data dependent, the parallel insertions at multiple lists are unnecessarily synchronous. In a similar truncation fashion, the truncated heap sort uses a k-node max-heap instead of a sequential list.
- ≻ The truncation in QuickSort is to prune off the branch that does not contain the smallest k elements. So is in RadixSort, which utilizes the finite range of the elements in b bits or digits in other radix base. The savings by truncation is data dependent, it is by half in average for any k, regardless its value.

Two of the algorithms have data-independent synchronous operations: BubbleSort and BitonicSort, the former has much higher complexity. We note that the comparisons in RadixSort can be synchronous, but the branching at every recursion level (and hence the pruning) is data dependent, strictly speaking.

The above puts kNN search over the sorting spectrum. At the extreme case $k=1$, one selects the first only. [3] This brings us to consider an alternative path for kNN search that is built upon two basic operations *select* and *prefix-scan*. One first locates the k-th element as a threshold, then use a scan to find all the elements smaller than the threshold, followed by another scan to finish the k-list with elements equal to the threshold. The time complexity of parallel scan is $O(\log n)$. There are many algorithms for select, such as QuickSelect and

[3]We shall comment on the existence, in theory, of an algorithm with k consecutive minimum searches. Each min search takes O($\log(\log n)$) steps on $n/2$ processors in synchronous comparisons [24], assuming a CRCW PRAM with the COMMON policy in concurrent writes, which has not materialized yet in operational systems.

RadixSelect [25], [26]. In particular, RadixSelect, which starts with the most significant bit, has the same time complexity in parallel as the RadixSort. For very large n and k, k-NN search using RadixSelect and scans is effective and has efficient GPU implementations [4] [27]. Also like RadixSort, the branching and data access pattern of this algorithm are data dependent, dynamic and weaken the performance in the practical range of k and n at the core level of kNN search. Such performance penalty is not reflected in the complexity model that is based on idealistic PRAM model.

We provide a detailed discussion on the truncated bitonic sort (TBiS) in the next section.

III. Truncated Bitonic Sort (TBiS)

Bitonic sort is known for being well suited for parallel computers or networks with synchronous operations because its sequence of reads, comparisons, exchanges and writes for sorting is data independent [28], despite the fact that it takes more logical comparisons than some other sort algorithms, as seen in Table I.

A. The Basic Algorithm

The basic algorithm of truncated bitonic sort consists of two functions, it recovers the complete sort algorithm when $k = n$, assuming that n and k are powers of 2.

Function $\mathbf{b}$ = `TBiSort(`$\mathbf{a}$, n, k, dir`)`

if n == *1* **then**
  $\mathbf{b} = \mathbf{a}$;
else
  $\mathbf{h}_1$ = TBiSort($\mathbf{a}(1:\frac{n}{2})$, $\frac{n}{2}$, k, dir);
  $\mathbf{h}_2$ = TBiSort($\mathbf{a}(\frac{n}{2}+1:n)$, $\frac{n}{2}$, k, -dir);
  $\mathbf{b}$ = TBiMerge([$\mathbf{h}_1$, $\mathbf{h}_2$], n, k, dir)

Function $\mathbf{b}$ = `TBiMerge(` $\mathbf{hh}$, n, k, dir`)`

$[\mathbf{h}_{\min}, \mathbf{h}_{\max}] = \text{minmax}(\ \mathbf{hh}(1:\frac{n}{2}), \mathbf{hh}(1+\frac{n}{2}:n))$;
$\mathbf{b}_{\min}$ = TBiMerge($\mathbf{h}_{\min}$, $\frac{n}{2}$, k, dir);
if n == *2k* **then**
  $\mathbf{b} = \mathbf{b}_{\min}$;
else
  $\mathbf{b}_{\max}$ = TBiMerge($\mathbf{h}_{\max}$, $\frac{n}{2}$, k, dir);
  if dir == 'up' **then**
    $\mathbf{b} = [\mathbf{b}_{\min}, \mathbf{b}_{\max}]$;
  else
    $\mathbf{b} = [\mathbf{b}_{\max}, \mathbf{b}_{\min}]$;

For conceptual clarity we present the algorithm in recursion fashion. We comment on the truncation. The TBiSort recursion goes downward first portioning the initial data list $\mathbf{a}$ all the way to the base level (sublist length 1), then goes upward with monotonic merges. In TBiMerge, the minmax function renders the minimal and maximal between each element pair on the two input lists. The truncation begins at, and continues from, the point where the monotonic sublists reach in size to k. At each merge step, $\mathbf{b}_{\max}$, the upper part of the merged list is purged, and the total data is reduced by half. By TBiMerge, the merged sublists never exceed k in size. There are at most $\log k$ TBiSort steps, each of which invokes at most $\log(k)$ TBiMerge steps, involving all n elements. The total number of pairwise comparisons is $n\log^2(k)/4$. In kNN search applications, the numerical range of $\log(k)$ is modest (less than 10 mostly). This makes TBiS even competitive in the conventional complexity with other truncated sort algorithms, see Table I.

In parallel implementation, the recursion is unfolded. The concurrency degree, i.e., the number of simultaneous operations, is at the maximal in the first $\log(k)$ TBiSort steps. More critically important to performance, the concurrent operations are inherently synchronous or systolic because they are data independent. The algorithm is therefore free of hashing or conditional branching (except the comparison between k and n). Data arrays are the natural choice of data structures, and data accesses are in regular strides, with locality no greater than k. In fact, the data access pattern for the full bitonic sort can be supported the best by the shuffle network, the same as used for the fast Fourier transform [29]. TBiS for a modest k simplifies the shuffle network substantially. After $\log k$ TBiSort steps, work volume per step is reduced as the data are purged, which we will address shortly.

B. Scoring and Sorting Interleaved

Upon any query, evaluating its similarity scores or distances takes a significant portion of kNN search time in a high dimensional search space. For example, the dimension of a SIFT feature space may be 128 or higher [21].

When the Euclidean metric is used for scoring, the calculation of the squared distances can be cast as a low-rank matrix multiplication with a BLAS routine [16], [30], which is broadly supported on many computer architectures, including CUBLAS on GPUs. This can be extended straightforwardly to multiple queries as well. However, a couple of other issues may arise : (i) the bandwidth in data transportation between the host processor and the GPU may be limited in comparison to the speed in parallel low-rank matrix multiplication; (ii) the data reuse is poor between scoring and sorting. A simple solution is to interleave the scoring and sorting operations. These operations can be interleaved by elements or by segments of certain size.

C. Corpus Partition and Streaming

We have at least two supporting arguments for corpus partition and streaming between subcorpora. First, the corpus may be too large, especially in high dimensions, to be accommodated all at once at the memory of an accelerator, such as a GPU. Secondly, the workload by TBiS is reduced every step after the first $\log(k)$ TBiSort steps and the GPU is under utilized. A simple solution is to partition the corpus at the external or host memory into subcorpora, according to

[4] `http://www.moderngpu.com`

the size of the GPU memory. Apply TBiS for kNN search on each subcorpus, successively, followed by a merge-and-purge step to update the historical kNN list with the kNN list for the current subcorpus. We then explore and exploit two kinds of overlaps to reduce latency. One is the overlap between data communication (load data from the external memory) and computation. The other is the overlap between the terminal stage of processing the previous corpus and the initial stage of processing the current corpus. Most of the time, there are three subcorpora on GPU, one is in the read buffer and two are in the processing buffers. The output takes place only at the end of processing the last subcorpus.

IV. Experimental Studies

In experimental study of parallel kNN search algorithms on computing architectures that favor synchronous operations, we use graphics processing units (GPUs) by NVIDIA. In particular, we used the GTX480 graphics card (Fermi) with 480 CUDA cores in 15 streaming multiprocessors, with 48KB of shared memory per streaming multiprocessor, operating at a core clock of 1.401GHz and 1.5GB of on board memory. The programs were written in CUDA version 4.0. We report a few sets of experiments.

A. TBiS vs. Sort and Select

The experiments presented in this section assume that the corpus data are on GPU memory and there is a single query. The test corpus is composed of 128 dimensional SIFT vectors extracted from the BelgaLogos dataset [22]. We let the corpus size to vary from $1,024$ to $1,048,576$ in terms of feature vectors.

Figure 1 shows the ratio, in kNN search time, of the approach with truncation after a full sort and that with truncation during the sort process. For the former, we use the radix sort algorithm provided by `thrust::sort` [31]. For the latter, we use TBiS. Without the distance calculation, TBiS is 2 to 16 times faster. With the distance calculation (using CUBLAS in both), TBiS is 1.3 to 4 times faster. We found that some other existing GPU implementations for kNN search are actually slower than the radix sort followed by truncation.

Figure 2 shows the comparison between TBiS and the use of the MGPU radix select for kNN search [5]. Radix select locates the k^{th} smallest element and scans the corpus again in order to find all elements smaller than the k^{th}. The truncated bitonic is faster than the `thrust::sort`, up to 16 times, for corpus size up to 2^{17}, depending on the number k of neighbors. Over the same parameter ranges, the MGPU select is at best 3 times faster than `thrust::sort`. We address next a few other performance factors.

Figure 3 shows the effect of interleaving the distance calculation with the operations for locating the nearest neighbors. The performance is improved by up to 2.4 times.

[5] `http://www.moderngpu.com`

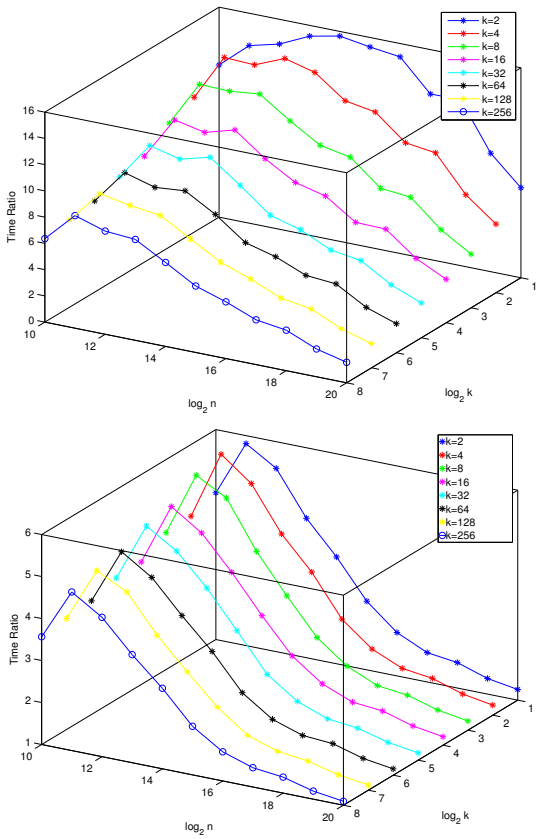

Fig. 1: Comparison between two approaches : truncation after sort and truncation during sort, without distance evaluation (top) or with distance evaluation (bottom) but without interleaved with sorting.

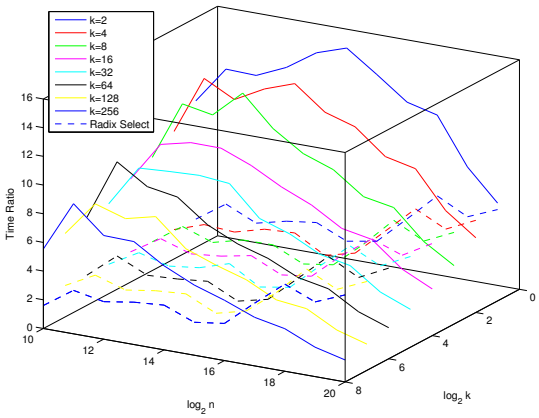

Fig. 2: Comparison of TBiS and the MGPU radix-select for kNN search.

B. TBiS in Stream Processing

This set of experiments is concerned with the impact of data transportation. The bar chart in Figure 4 is a performance profile among distance calculation (in red), neighbor search (green) and data transfer (blue) from the external memory to GPU memory.

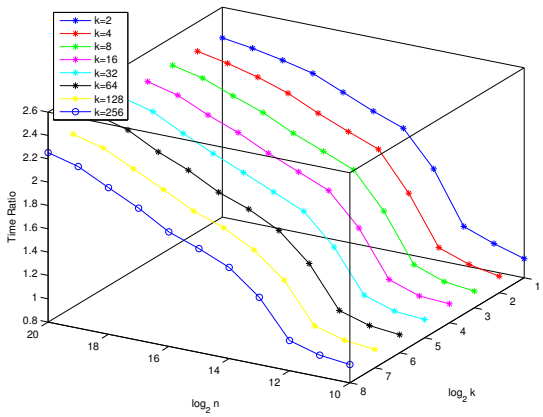

Fig. 3: Improved kNN search by TBiS with the distance calculation and truncated sorting interleaved.

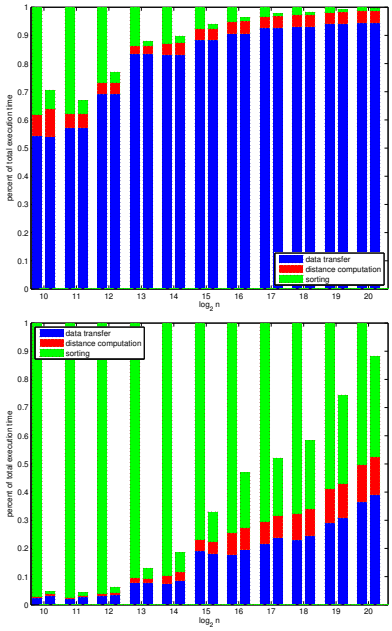

Fig. 4: Performance profile in percentile for a single query (top) and 128 queries (bottom), respectively, data transportation in blue, distance calculation in red and neighbor location in green.

The top bar chart shows the experimental results for a single query. For each experiment with a different corpus size, the green bar to the left is by `thrust::sort`, and the green bar to the right is by TBiS. The bottom chart shows the experimental results for 128 queries per experiment. In many image data processing applications, multiple subimages or multiple image frames in a time sequence are submitted as multiple queries for search of k nearest neighbors each. Both charts show that the kNN performance evaluation for timely or real-time processing must not ignore the limitation in communication bandwidth and that one must find every way to reduce the ratio of communication over computation.

The bottom chart illustrates the benefit of grouping multiple queries in a neighborhood. It indicates also the opportunity to overlap the communication and computation by partitioning the corpus and then pipeline processing the subcorpora. There are also situations where the corpus is large and high dimensional, the data volume is too large to be accommodated on GPU memory all at once. The corpus partition becomes necessary.

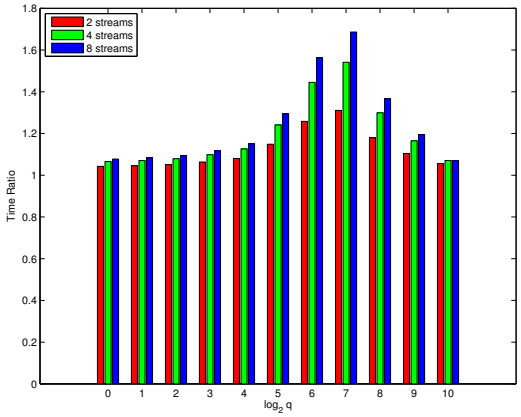

Fig. 5: Further improvement with 2, 4 and 8 streams over a single stream when processing large and high-dimensional corpus ($16,777,216$ vectors of 128 dimensions).

Figure 5 shows the effect of the streaming approach. In a stream, we overlap the computational processing over a subcorpus with data transferring of another stream. Only the output data of kNN search are transferred back to the external memory. We also overlap the computation between two successive subcorpora, by careful design of the initial corpus partition, according to the available resources. We use 2 (red), 4 (green) and 8 (blue) GPU streams, pipelining the processing of respective subcorpora. All bars show the speed-up from using a single GPU stream, without pipelining data transfer and computations. The further improvement, up to another 1.75 times, in performance by corpus partition and stream processing the subcorpora, is shown. The corpus is composed of $16,777,216$ SIFT vectors of 128 dimensions, extracted from approximately $10,000$ images in the BelgaLogos data set [22].

C. Application of TBiS to Feature Matching

TBiS on GPU is applied to image retrieval via SIFT vector matching [21]. Figure 6 presents the comparison in performance for feature matching to the kNN search procedure in VLFeat, which is an open source computer vision library [32]. The VLFeat procedure for kNN search is based on an approximate algorithm with space partition and tree-type data structure. VLFeat executes serially on one CPU core. The CPU used in these experiments is an AMD Opteron 6168 processor. The speedup of TBiS on GPU over the VLFeat kNN search routine on the CPU is remarkable, from 180 to 250 times, depending on the values of k, for $1,024$ queries into $65,536$ corpus feature vectors of 128 dimensions. The chosen

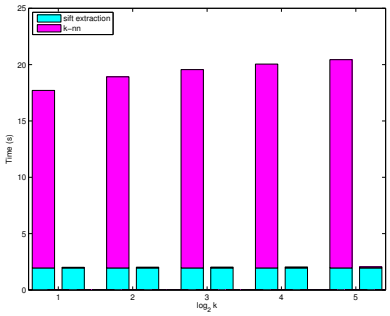

Fig. 6: Speedup in SIFT feature matching by the parallel execution of TBiS on the GPU (the nearly invisible magenta bar to the right) over the serial execution of the kNN search procedure in VLFeat on a single-core CPU (the magenta bar to the left), for $1,024$ queries into $65,536$ corpus feature vectors of 128 dimensions and 32 neighbors. The light blue bars are the time for feature extraction on the single-core CPU.

number of queries is close to the average number of query regions in one or more image frames in certain image retrieval applications [3]. The algorithm for SIFT feature extraction is provided in VLFeat also. It has parallel implementation on GPUs [33].

ACKNOWLEDGMENT

The authors acknowledge the support of a Marie Curie International Reintegration Grant and a CCF grant by National Science Foundation. We thank the anonymous referee for her/his comments on *select*.

REFERENCES

[1] T. M. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[2] M. Cooper, T. Lui, and E. Rieffel, "Video Segmentation via Temporal Pattern Classification," *IEEE Transactions on Multimedia*, vol. 9, no. 3, pp. 610–618, 2007.

[3] A. Joly and O. Buisson, "Logo Retrieval with a Contrario Visual Query Expansion," in *Proceedings of 17th ACM international conference on multimedia*, ser. MM '09. ACM, 2009, pp. 581–584.

[4] X. Luo, Y. Ouyang, and Z. Xiong, "Improving K-nearest neighborhood based Collaborative Filtering via Similarity Support," *International Journal of Digital Content Technology and its Applications*, vol. 5, pp. 248–256, 2011.

[5] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Proceedings of the 10th European Conference on Computer Vision: Part I*, ser. ECCV '08. Springer-Verlag, Oct 2008, pp. 304–317.

[6] L. Kuang and M. Zulkernine, "An anomaly intrusion detection method using CSI-KNN algorithm," in *Proceeding of ACM symposium on Applied computing*, ser. SEC '08, 2008, pp. 921–926.

[7] S. Manne, S. K. Kotha, and S. S. Fratima, "A Query based Text Categorization using K-nearest neighbor Approach," *International Journal of Computer Applications*, vol. 32, no. 7, pp. 16–21, 2011.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *Journal of ACM Transactions on Mathematical Software*, vol. 3, no. 2, pp. 209–226, 1977.

[9] S. M. Omohundro, "Five Balltree Construction Algorithms," International Computer Science Institute, Berkeley, Tech. Rep., 1989.

[10] J. Uhlmann, "Satisfying General Proximity/Similarity Queries with Metric Trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991.

[11] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," in *Proceeding of the 24th International Conference on Very Large Data Bases*, ser. VLDB '98. Morgan Kaufmann Publishers Inc., 1998, pp. 194–205.

[12] J. Z. C. Lai, Y.-C. Liaw, and J. Liu, "Fast k-nearest-neighbor search based on projection and triangular inequality," *Journal of Pattern Recognition*, vol. 40, no. 2, pp. 351–359, 2007.

[13] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ser. STOC '98. ACM, 1998, pp. 604–613.

[14] L. Paulevé, H. Jegou, and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognition Letters*, vol. 31, pp. 1348–1358, 2010.

[15] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matías, and M. Marín, "kNN Query Processing in Metric Spaces using GPUs," in *Proceedings of the 17th international European Conference on Parallel and Distributed Computing*, ser. Euro-Par '11, 2011, pp. 380–392.

[16] V. Garcia, Éric Debreuve, F. Nielsen, and M. Barlaud, "K-Nearest Neighbor Search: Fast GPU-Based Implementation and Application to High-Dimensional Feature Matching," in *Proceedings of IEEE 17th International Conference on Image Processing*, ser. ICIP '10. IEEE Computer Society, 2010, pp. 3757–3760.

[17] K. Kato and T. Hosino, "Solving k-Nearest Neighbor Problem on Multiple Graphics Processors," in *Proceedings of 10th International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. IEEE Computer Society, 2010, pp. 769–773.

[18] Q. Kuang and L. Zhao, "A Practical GPU Based kNN Algorithm," in *Proceeding of the 2nd International Symposium on Computer Science and Computational Technology*, ser. ISCSCT '09, vol. 7, no. 3, 2009, pp. 151–155.

[19] S. Liang, Y. Lui, C. Wang, and L. Jian, "Design and Evaluation of a Parallel K-Nearest Neighbor Algorithm on CUDA-enabled GPU," in *Proceedings of the 2nd IEEE Symposium of web Society*, ser. SWS '10. IEEE Computer Society, 2010, pp. 53–60.

[20] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computations," in *Proceeding of the 19th International Conference on Advances in Geographic Information Systems*, ser. GIS '11, 2011, pp. 211–220.

[21] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[22] "BelgaLogos images," 2007, http://www.belga.be/.

[23] C. E. Leiserson, "Systolic Priority Queues," Department of Computer Science, Carnegie Mellon University, Tech. Rep., 1979.

[24] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," University of California at Berkeley, Tech. Rep. CSD-88-408, 1988.

[25] C. A. R. Hoare, "Algorithm 65: find," *Commun. ACM*, vol. 4, no. 7, 1961.

[26] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1997, vol. 3.

[27] T. Alabi, J. D. Blanchard, B. Gordin, and R. Steinbach, "Fast K-selection Algorithm for Graphics Processing Units," 2012, to appear in ACM Journal of Experimental Algorithms.

[28] K. E. Batcher, "Sorting Networks and their Applications," in *Proceedings of the spring joint computer conference*, ser. AFIPS '68 (Spring). ACM, 1968, pp. 307–314.

[29] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE transactions on computers*, vol. c-20, no. 2, pp. 153–161, 1971.

[30] *NVIDIA CUDA CUBLAS Library*, 4th ed., 2011, http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[31] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[32] A. Vedaldi and B. Fulkerson, "VLFeat: An Open and Portable Library of Computer Vision Algorithms," 2008, http://www.vlfeat.org/.

[33] C. Wu, "SiftGPU: A GPU implementation of scale invariant feature transform (SIFT)," University of North Carolina at Chapel Hill, 2012, http://cs.unc.edu/ ccwu/siftgpu/.