Heap Based k-Nearest Neighbor Search on GPUs

R.J. Barrientos J.I. Gómez
C. Tenllado M. Prieto
ArTeCS Group, UCM Madrid
corresponding author: ribarrie@fdi.ucm.es

Abstract

The k-nearest neighbor algorithm finds, for a given query, the k most similar samples from a reference set. It has been successfully applied in a broad range of applications in the field of information retrieval due to its simplicity and accuracy. However, specially when dealing with high-dimensional feature spaces, its computational load is very high. In this paper we propose a parallel GPU implementation that outperforms state-of-the-art sort-based solutions. Instead of sorting the vector of query distances, we build in parallel a *heap* with the current k smaller elements. Our strategy consistently outperforms a parallel quick-sort based implementation for all tested input datasets.

1 Introduction

The k-nearest neighbor (k-NN) classifier [2, 9] is a classic machine learning algorithm which classifies new examples based on its reliance on the original training examples. The k-NN algorithm is instance-based and it is used in many applications in the field of statistical pattern recognition, data mining, image processing and many others.

This algorithm first compares a query object q_i with all the objects of a reference database, whose elements have been previously classified. If the majority of k most similar samples to the query point q_i belong to a certain category, then we may conclude that the object falls into that category. Similarity can be measured by any metric distance in the feature space.

Recently, content-based retrieval of similar objects is attracting increasing attention in many fields such as image-based web searching. For traditional k-NN applications, a point in the feature space ($\mathbb{R}^n$) may be represented with a vector of low dimensionality (usually n in the order of tens). However, multimedia objects usually reside in high-dimensional feature spaces, with n being higher than a few hundreds.

As a lazy learner (no energy is spent on creating an abstract model), k-NN is simple but computationally intensive, since every classification involves the comparison of the target query to all the elements of the reference database. When the size of this database and/or many queries need to be solved, the execution time may become unacceptably high, specially for high-dimensional feature spaces.

Fortunately, k-NN algorithms have usually enough data parallelism to allow efficient implementations on most parallel computing platforms. Furthermore, in many applications, such as web searching, it becomes necessary to solve many queries in parallel and it is even possible to exploit inter-query parallelism. Exhaustive distance computation is not a major problem, since this operation scales linearly and the main bottleneck becomes the method for finding the closest k-elements. In this paper we have presented an alternative method for this stage and compared it with previous proposals based on state-of-the-art parallel sorting methods. As computational platform we have chosen an heterogeneous system based on GPUs, since these accelerators allows us for

an efficient intra-query parallelization.

The rest of this paper is organized as follows. Section 2 presents the related work and highlights our main contributions. In Section 3 we review GPU technology and the CUDA programming model. Section 4 provides a formal description of the k-NN algorithm while Section 5 describes the GPU mappings that evaluated in this paper. In Section 6 we report performance measurements and discuss their significance. Section 7 summarizes our findings.

2 Related Work

The k-NN algorithm [2, 9] is widely applied in pattern recognition and data mining for classification, given its simplicity and low error rate. Before the advent of massively parallel platforms, brute force exploration was usually not considered as a valid option, specially for large training databases and high-dimensional spaces. To prune the search space and avoid as many distance computations as possible, many indexing approaches have been proposed. Most retrieval methods are based on kd-trees [3]. There is a large amount of work on adaptations of the basic kd-tree structure to the k-NN problem ([5, 21]). Non-tree structures that efficiently split the search space have been also proposed ([8, 4]. Coarse-grained MPI based implementations using these structures are also appear very frequently in literature [17].

The main disadvantage of these methods is that they need to build and maintain complex data structures for the reference data set. Moreover, CUDA imposes many restrictions that make it hard to efficiently deal with such structures. Thus, k-NN is typically implemented using brute force methods on GPU. Therefore, current GPU solutions have been somehow simpler and less efficient from an algorithm standpoint. For instance, Benjamin Bustos et al. [6] proposed a non-CUDA implementation exploiting GPU texture memories. Their implementation only looks for the minimum element, which significantly simplifies the problem. Exhaustive distance computation in conjunction with parallel sorting was proposed in [11, 15]. Vecente García et al. [11] proposed a modified parallel insertion sort in order to just get the k closer elements while Kuang et al. [15] proposed an improved Radix-sort to perform the final sort. Recently, a GPU version of Quicksort was proposed by Daniel Cederman and Philippas Tsigas [7]), which showed to be faster than previous sorting algorithms. Therefore, in this paper we use this Quicksort implementation as baseline for performance evaluation. Our proposal also exhaustively computes all the distances, but uses a specific heap-based methodology to find the k closer elements. We will show that this methodology outperforms previous GPU brute-force proposals.

3 Graphics processing units

All our GPU implementations have been developped using the NVIDIA's CUDA programming model [10]. This model represents the GPU as a coprocessor that can run data-parallel kernels and provides extensions to the C language to (1) allocate data on the GPU, (2) transfer data between the GPU and the CPU and (3) launch kernels.

A CUDA kernel executes a sequential code on a large number of threads in parallel. Those threads are organized into a grid of *CUDA blocks*. Threads within a block – up to 512 threads – can cooperate with each other by (1) efficiently sharing data through a shared low latency local memory and (2) synchronizing their execution via barriers. In contrast, threads from different blocks in the same grid can only coordinate their execution via accesses to a high latency global memory (the graphic's board memory). Within certain limits, the programmer specifies how many blocks and how many threads per block are assigned to the execution of a given kernel. When a kernel is launched, threads are created by hardware and dispatched to the GPU cores until all the required threads are completed.

Figure 1 gives an schematic view of the actual hardware of modern NVIDIA's GPU. The GPU cores (processors) are organized into sev-

eral *multiprocessors*. Each of these cores integrates its own functional units and a large register file that accommodates the execution of hundreds of concurrent threads – to tolerate the long latency associated with the accesses to the graphic's board memory (the *device memory*) –. The multiprocessors integrate a single instruction unit and a local shared memory [1]. The memory hierarchy also includes read-only cache memories to speed up access to textures and constants, which are shared by pairs of multiprocessors. The CUDA block abstraction is closely related to this organization: each CUDA block is executed by one multiprocessor, which depending on the resource availability can accommodate multiple blocks concurrently – the actual number of concurrent threads per multiprocessor is limited by the allocated resources –.

As mentioned above, thread creation and scheduling is performed entirely in hardware. The scheduling unit is not the individual thread but a group of threads called *warp*. The threads of a block are assigned to warps by their thread ID. Every cycle, the scheduler in each multiprocessor chooses the next warp to execute, i.e. no individual threads but warps are swapped in and out. If threads in a warp execute different code paths, only threads on the same path can be executed simultaneously and a penalty is incurred.

According to NVIDIA specifications, one of the main concerns for GPU programmers should be occupancy maximization. Current NVIDIA GPUs support a maximum of 1024 concurrent threads per multiprocessor distributed across equally sized blocks. Other than unveiling parallelism, resource distribution must be considered to achieve an acceptable occupancy.

The other most significant factor affecting performance is the memory usage. Despite the GPU takes advantage of multithreading to hide memory access latencies, having hundreds of threads simultaneously accessing the global memory introduces a high pressure on the memory bus bandwidth. Reducing memory accesses by using local shared memory to exploit inter thread locality and data reuse largely improves kernel execution time. In addition, memory access patterns are relevant to allow coalescing of warp loads and to avoid bank conflicts on shared memory accesses.

[1] On the GT200 series, there is also a single double-precision unit per multiprocessor

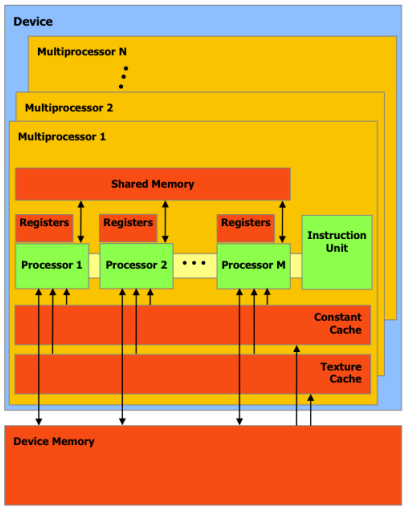

Figure 1: The CUDA programming model is designed for compute. It represents the GPU as a coprocessor that integrates several multiprocessors and a complex memory hierarchy.

4 k Nearest Neighbours

This section gives a formal description of the k-Nearest Neighbours (k-NN) algorithm. Let $\mathcal{S} = r_1, r_2, ..., r_m$ be a set of m elements, with each $r_i \in \mathbb{R}^n$, let $q_i \in \mathbb{R}^n$ be a query, and δ be the function of distance between two elements in $\mathbb{R}^n$. The k-NN search problem consists in searching the k elements in $\mathcal{S}$ with the smallest distance to the query q_i using δ. These elements are the k nearest neighbors. Figure 2 graphically illustrates the k-NN problem using $k = 5$. Without loss of generality in this paper we use the Euclidean distance as δ. Other common metrics are the Manhattan distance, the Chebychev norm and the Mahalanobis distance.

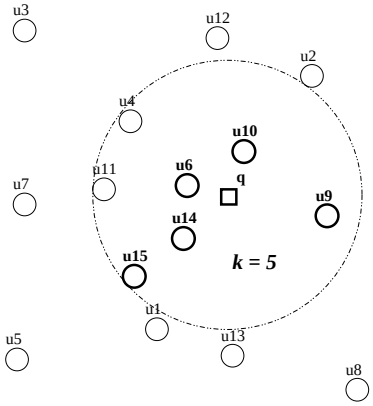

Figure 2: Example of a k-NN query with k=5.

The straight forward sequential solution to the k-NN problem is to compute the distance between the query and all elements in $\mathcal{S}$, sort the elements of $\mathcal{S}$ according to their distance to the query, in increasing order, and finally select the first k elements.

5 Mapping k-NN on GPUs

This section starts with a description of some common strategies proposed in literature to solve the kNN problem on the GPU. All of them fall into a common category, which we have denoted as *Sorting-based Reduction* methods. Then we introduce our alternative approach, which we have denoted as *Heap-based Reduction*, and show its main benefits over the previous apparoches.

All methods start by computing the distance between the query and all the elements in $\mathcal{S}$, and only differ in how the k elements with the smallest distances are selected. Therefore their GPU implementations share a common first kernel that computes a vector $D : D_i = \delta(q, r_i)$, i.e. its i-th component is the distance between the query and the i-th element in $\mathcal{S}$. As its computation is fully parallel, the kernel is launched on a grid of 1D threadblocks with 128 threads per block, where each thread computes one of the elements in D.

5.1 Sorting-based Reduction

One way to select the k-NN elements of given query is to sort all the elements in $\mathcal{S}$ according to D and then select the first k elements. Effective implementations of this strategy on the GPU need an efficient parallel sort algorithm. Cederman and Philips [7] proposed a GPU-Quicksort implementation that showed to be more efficient than all previous GPU sorting methodologies presented so far [13, 18, 19].

It is divided in two phases: Partitioning and Sorting. The former consists in splitting the original vector into P partitions, that can then be sorted independently. This process is recursively repeated until there are enough partitions so that the GPU can be filled with a grid of blocks, each block being responsible of the sorting of one partition.

The Partitioning process starts by selecting a global pivot; all the elements greater than this pivot are stored in one partition (hi-partition) while the others are stored in the low-partition. Essentially, the original sequence is divided into M subsequences that will be processed by different threadblocks. The implementation exploits the *atomicAdd* instruction, available in CUDA from computing capability 1.3, to make these threadblocks cooperate in finding the correct positions where they must store their respetive elements, lower and greater than the global pivot, found in the subsequence managed by the threadblock.

This *atomicAdd* instruction allows us to perform the partition of one sequence in a single kernel. For older devices, where no *atomicAdd* is available, this process requires three kernels, which introduces three synchronization barriers that limit the scalability of the implementation.

To obtain P partitions, at least $P - 1$ pivots must be selected and the Partition kernel (or kernels in case of no *atomicAdd*) must be executed $P - 1$ times, which involves $P - 1$ synchronization barriers.

When the Partitioning phase finishes, a grid of 1D blocks executes the Sorting kernel. Each threadblock is in charge of sorting one partition. Sorting is performed in parallel by

the threads in the threadblock using the same parallel-quicksort strategy, handling the recursion with an explicit stack implemented on shared memory. When the size of the subsequences generated by quicksort go bellow a given threshold, a *bitonic sort* [12] is used to complete the sorting.

Apart from the synchronizations needed by this algorithm, one main problem is the high probability to have unbalanced workloads for the different threadblocks in the second phase. The size of the partitions depends on the pivots selected and cannot be controlled, unbalancing the work of the threadblocks that are assigned partitions with different sizes.

5.2 Heap-based Reduction

One of the disadvantages of the sorting reduction strategies is that the whole set $\mathcal{S}$ needs to be sorted, while we are interested only in the k elements with smallest distance to the query. Our alternative approach, which we have denoted as *Heap-based Reduction*, is able to outperform previous proposals since essentially, it just finds these k elements. Furthermore, we have also explored a more aggressive scheme, which we have denoted as *Bulk Heap-based Reduction*, which exploits inter-query parallelism from several concurrent queries. As we mentioned above, coarse-grain inter-query parallelism is available in many applications such as image searching algorithms for the Web.

The search of the k-NN elements of a single query is performed by the Heap Reduction kernel, described in Algorithm 1. The set $\mathcal{S}$ is distributed among the threads in the threadblock. The kernel is structured in three phases separated by intra-block synchronization barriers. In the first phase each thread t_i finds the k-NN elements of its assigned subset $S_i \in \mathcal{S}$ using an auxiliary *max-heap* [14] of size k. Elements in S_i are processed sequentially, using the max-heap for keeping track of the k elements of S_i with the minimum distance that have been found so far. When the distance of a new element is smaller than the root counterpart, the root is popped off the heap and the new element is pushed in to its corresponding position using a *popush(x)* method. Let N be the size of the threadblock, the *heaps* of the N threads are stored as columns of a $k \times N$ array to maximize the coalesced accesses to the heaps and reduce bank conflicts if the heap fits in the shared memory, which depends on the values of k and N.

After the first phase, the thread block has build $N \times k$ *heaps*. The second phase, executed only by the first *warp* in the block, stores these elements in 32 new *heaps* (one per thread in the *warp*) following the same strategy as before. We end up having the $(32 \times k)$-NN to the query. In this phase for most k the 32 heaps can be stored in shared memory.

The last phase is executed only by the first thread in the block, that sequentially selects the k-NN by using a single *heap* in shared memory. We have made public this algorithm in the webpage [1].

From the description above we see that parallelism is reduced from N to 32 in the second phase, and to 1 in the third phase. However, for large values of N, the heaviest phase is the first one. In the worst case, when all the elements are inserted in the *heap* replacing the root, it has a complexity of $O\left(\frac{N}{Size_{Block}} log(K)\right)$. On the other hand, in the best case, when the first k elements are the smallest and the rest can be discarded, the complexity is $O\left(\frac{N}{Size_{Block}}\right)$.

This strategy exploits parallelism for a single query. In the next section we will show that for small values of k (relative to N), even this single query Heap-Reduction method is able to outperforms the state-of-the-art Sorting-Reduction method based on quicksort. But since GPUs allows us to launch a large number of thread blocks in parallel, this scheme can be easily extended to process multiple queries simultaneously.

6 Experimental Results

In this section we have evaluated the performance of our Heap Reduction method and compared it over a state-of-art quicksort-based algorithm. As experimental platform we have used a NVIDIA GeForce GTX 280, which has

Algorithm 1 *Heap Reduction* kernel.

```
Heap_Reduction()
  {D = Array of distances to be reduced.}
  {DH_i = Heap of thread i stored in Device Memory}
  {SH_i = Heap of thread i stored in Shared Memory}
  {Size_warp = Size of a warp (actually this is 32)}
  {tid = Unique ID for the thread among all the threads of all the blocks.}
  {TxBlock = Threads per block of the kernel.}

  {Each thread stores the elements in one heap}
  for i = tid; i < D.size(); i+=TxBlock do
    if DH_tid.size() < K then
      DH_tid.push(D[i])
    else
      if DH_tid.top() > D[i] then
        DH_tid.popush(D[i])
      end if
    end if
  end for

  __syncthreads()

  {One warp stores the elements of the previous heaps in Size_warp shared memory heaps}
  if tid < Size_warp then
    for i = tid; i < TxBlock; i+=Size_warp do
      x = DH_i.pop()
      if SH_tid.size() < K then
        SH_tid.push(x)
      else
        if SH_tid.top() > x then
          SH_tid.popush(x)
        end if
      end if
    end for
  end if

  __syncthreads()

  {One thread visit exhaustively SH and selects the first K}
  if tid == 0 then
    get_first_K(SH)
  end if
```

30 multiprocessors, which 8-core per multiprocessor and 16K of shared memory. The device global memory is 1GB.

For our experiments we use a database of face images, obtained from the Face Recognition Grand Challenge [16]. We apply the *Eigen Faces* method [20] to obtain a projection matrix, that can be used to generate a feature vector from any face image. The distance between two face images is then computed as the euclidean distance between their corresponding feature vectors. Although there are different alternatives to extract feature vectors from the images in the database, the performance of the search methods evaluated in this paper does not depend on the feature extraction process selected.

Figures 3 and 4 show the execution time needed to process 8000 queries, for different DB sizes and k values (8, 16 and 32). Three algorithms are shown:

- Sorting: GPU-Quicksort based sorting reduction. The 8000 queries are sequentially processed.
- Heap: heap reduction kernel with only one block to process every single query. The 8000 queries are sequentially processed.
- BulkH: heap reduction kernel on a grid of B threadblocks, where each block processes a different query. The kernel must be executed $8000/B$ times to process the 8000 queries. The size of the grid (B) is constrained by the memory needed to store the heaps of all the associated threadblocks.

As we see from Figure 3 (notice that the time is normalized to the slowest algorithm), in all cases the BulkH performs better than the sorting strategy. As BulkH execution time increases with k, the speedup slightly decreases with increasing k values. We should highlight the good scalability respect to k exhibited by the BulkH algorithm. In addition, we see that for small ratios between k and the DB size, even the Heap version is faster than the sorting one.

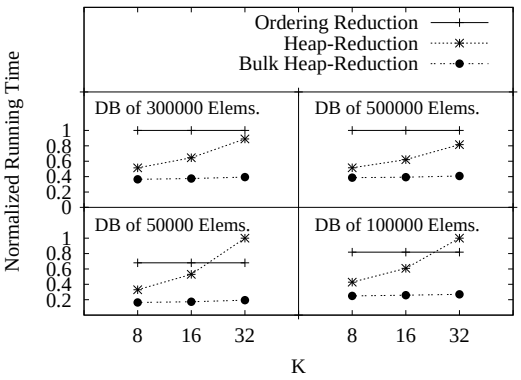

Figure 3: Normalized Running time for different size of DB.

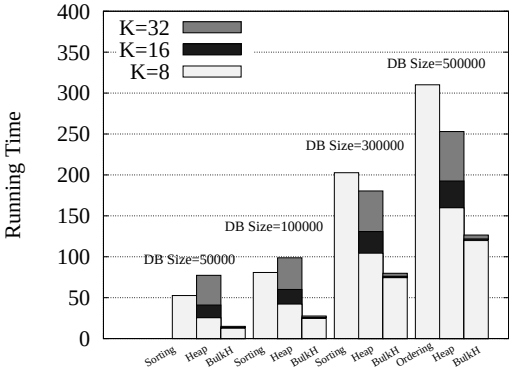

Figure 4: Running time for different size of DB using K=8,16,32.

Figure 4 shows the same data but represented in a different way, more convenient to analyze the scalability of the different algorithms as a function of the DB size. The speedup of the BulkH algorithm over the Sorting one gets reduced with increasing DB sizes. However, this reduction gets less significant as the DB size increases, which seems to indicate that it tends to a constant value for very large DBs.

7 Conclusions

In this paper we have presented a new GPU mapping of the k-NN algorithm. Our implementation exhaustively measures the distance between the query and all the reference objects. In a second step, we find the k smaller distances maintaining in parallel a *heap* structure. This new implementation clearly outperforms state-of-the-art GPU mappings, which traditionally (partially) sort the vector of distances. Moreover our *bulk-heap* version get higher scalability than sort-based implementations as the size of the database increases.

References

[1] Batch heap reduction method.

[2] David W. Aha and Dennis Kibler. Instance-based learning algorithms. In *Machine Learning*, pages 37–66, 1991.

[3] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[4] Nieves R. Brisaboa, Antonio Fariña, Oscar Pedreira, and Nora Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *ISM*, pages 881–888, 2006.

[5] Nieves R. Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *SOFSEM*, pages 186–197, 2008.

[6] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science (ICCS)*, volume 3994, pages 196–199. Springer, 2006.

[7] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24, 2009.

[8] E. Chav'ez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *The 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 75–86. IEEE CS Press, 2000.

[9] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[10] CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation.

[11] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. *Computer Vision and Pattern Recognition Workshop*, 0:1–6, 2008.

[12] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[13] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[14] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[15] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. pages 151–155, Huangshan, China, 2009.

[16] P.J. Phillips, P.J. Flynn, T. Scruggs, K.W. Bowyer, Jin Chang, K. Hoffman, J. Marques, Jaesik Min, and W. Worek. Overview of the face recognition grand challenge. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 947–954 vol. 1, June 2005.

[17] Erion Plaku and Lydia E. Kavraki. Distributed computation of the k nn graph for large high-dimensional point sets. *Journal of Parallel and Distributed Computing*, 67(3):346–359, March 2007.

[18] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[19] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[20] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[21] R. Uribe and G. Navarro. Egnat: A fully dynamic metric access method for secondary memory. In *Proc. 2nd International Workshop on Similarity Search and Applications (SISAP)*, pages 57–64. IEEE CS Press, 2009.