HPCLatAm 2013, pp. 101-112 (full paper)
Session: GPU Architecture and Applications
C. Garcia Garino and M. Printista (Eds.)
Mendoza, Argentina, July 29-30, 2013

Permutation Index and GPU to Solve efficiently Many Queries

Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, Nora Reyes

LIDIC. Universidad Nacional de San Luis,
Ejército de los Andes 950 - 5700 - San Luis - Argentina
{omlopres, ncmiran, mpiccoli, nreyes}@ unsl.edu.ar

Abstract. Similarity search is a fundamental operation for applications that deal with multimedia data. For a query in a multimedia database it is meaningless to look for elements exactly equal to a given one as query. Instead, we need to measure the similarity (or dissimilarity) between the query object and each object of the database. The similarity search problem can be formally defined through the concept of metric space, which provides a formal framework that is independent of the application domain. In a metric database, the objects from a metric space can be stored and similarity queries about them can be efficiently answered. In general, the search efficiency is understood as minimizing the number of distance calculations required to answer them. Therefore, the goal is to preprocess the dataset by building an index, such that queries can be answered with as few distance computations as possible. However, with very large metric databases is not enough to preprocess the dataset by building an index, it is also necessary to speed up the queries by using high performance computing, as GPU. In this work we show an implementation of a pure GPU architecture to build the *Pemutation Index*, used for approximate similarity search on databases of different data nature. Our proposal is able to solve many queries at the same time.

1 Introduction

Due to an increasing interest in manipulating and retrieving multimedia data, nowadays the problem of similarity searching receives much attention. The metric space model is a paradigm that allows to model all the similarity search problems. A metric space (X, d) is composed of a universe of valid objects X and a distance function $d : X \times X \rightarrow R^+$ defined among them. The distance function determines the similarity (or dissimilarity) between two given objects and satisfies several properties which make it a metric. Given a dataset of $| U |= n$ objects, a query can be trivially answered by performing n distance evaluations, but sequential scan does not scale for large problems. The reduction of number of distance evaluations is important to achieve better results. Therefore, in many cases preprocessing the dataset is a good option to solve queries with as few distance computations as is possible. An index helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluations during searches [1]. One of these indices is the *Permutation Index* [2].

Moreover, for very large metric database is not enough to preprocess the dataset by building an index, it is also necessary to speed up the queries by using high performance computing (HPC). In order to employ HPC to speedup the preprocess of the dataset to obtain an index, and to answer posed queries, the Graphics Processing Unit (GPU) represents a good alternative. The GPU is attractive in many application areas for its characteristics, especially because of its parallel execution capabilities and fast memory access. They promise more than an order of magnitude speedup over conventional processors for some non-graphics computations.

A GPU computing system consists of two basic components, the traditional CPU and one or more GPUs (Streaming Processor Array). The GPU can be considered as a manycores coprocessor able to support fine grain parallelism (a lot of threads run in parallel, all of them collaborate in the solution of the same problem) [3, 4]. GPU is different than other parallel architectures because it shows flexibility in the local resources allocation to the threads.There are many tools to program the GPU, CUDA is one of them. CUDA is a standard C/C++ extended by several keywords and constructs. Its programming model is SPMD (Single Process-Multiple Data) with two main characteristics: the parallel work through concurrent threads and the memory hierarchy.

In metric spaces, the indexing and query resolution are the most common operations. They have several aspects that accept optimizations through the application of HPC techniques. There are many parallel solutions for some metric space operations implemented to GPU. Querying by k-Nearest Neighbors (k-NN) has concentrated the greatest attention of researchers in this area, so there are many solutions that consider GPU. In [5–9] differents proposal are made, all of them are improvements to brute force algorithm (sequential scan) to find the k-NN of a query object.

The paper is organized as follows: Section 2 describes all the previous concepts necessary to understand our work (it is a magister thesis in progress) and the state of art, Section 3 introduces the sequential version of *Permutation Index*, Sections 4, 5, and 6 sketch the characteristics of our proposal and its empirical performance. Finally, the conclusions and future works are exposed.

2 Previous Concepts

In this section, we explain the main concepts to develop this work.

2.1 Metric Space, Queries and Index

A metric space (X, d) is composed of a universe of valid objects X and a distance function $d : X \times X \to R^+$ defined among them. The distance function determines the similarity (or dissimilarity) between two given objects and satisfies several properties such as strict positiveness (except $d(x, x) = 0$, which must always hold), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). The finite subset $U \subseteq X$ with size $n = |U|$, is called the *database* and represents the set of objects of the search space. The distance is assumed to be expensive to compute, hence it is customary to define the search complexity as the number of distance evaluations performed, disregarding other components. There are two main queries of interest [1, 10]: Range Searching and the k-NN. The goal of a range search $(q, r)_d$ is to retrieve all the objects $x \in U$ within the radius r of the query q (i.e. $(q, r)_d = \{x \in U/d(q, x) \leq r\}$). In k-NN queries, the objective is to retrieve the set k-NN(q)$\subseteq U$ such that $\mid k\text{-NN}(q) \mid= k$ and $\forall x \in k\text{-NN}(q), v \in U \wedge v \notin k\text{-NN}(q), d(q, x) \leq d(q, v)$.

When an index is defined, it helps to retrieve the objects from U that are relevant to the query by making much less than n distance evaluations during searches. The saved information in the index can vary, some indices store a subset of distances between objects, others maintain just a range of distance values. In general, there is a tradeoff between the quantity of information maintained in the index and the query cost it achieves. As more information an index stores (more memory it uses), lower query cost it obtains. However, there are some indices that use memory better than others. Therefore in a database of n objects, the most information an index could store is the $n(n - 1)/2$ distances among all element pairs from the database. This is usually avoided because $O(n^2)$ space is unacceptable for realistic applications [11].

Proximity searching in metric spaces usually are solved in two stages: preprocessing and query time. During the preprocessing stage an index is built and it is used during query time to avoid some distance computations. Basically the state of the art in this area can be divided in two families [1]: *pivot based algorithms* and *compact partition based algorithms*. There is an alternative to "exact" similarity searching called approximate similarity searching [12], where accuracy or determinism is traded for faster searches [1, 10], and encompasses *approximate* and *probabilistic algorithms*. The goal of approximate similarity search is to reduce *significantly* search times by allowing some errors in the query output. In these algorithms one usually has a threshold ϵ as parameter, so that the retrieved elements are guaranteed to have a distance to the query q at most $(1 + \epsilon)$ times of what was asked for [13]. Probabilistic algorithms on the other hand state that the answer is correct with high probability. Some examples are [14, 15]. In the next section we detail a probabilistic method: *Permutation Index* [2].

2.2 GPGPU

Mapping general-purpose computation onto GPU implies to use the graphics hardware to solve any applications, not necessarily of graphic nature. This is called GPGPU (General-Purpose GPU), GPU computational power is used to solve general-purpose problems [3, 4]. The parallel programming over GPUs has many differences from parallel programming in typical parallel computer, the most relevant are: *The number of processing units*, *CPU-GPU memory structure* and *Number of parallel threads*.

Every GPGPU program has many basic steps, first the input data transfers to the graphics card. Once the data are in place on the card, many threads can be started (with little overhead). Each thread works over its data and, at the end of the computation, the results should be copied back to the host main memory. Not all kind of problem can be solved in the GPU architecture, the most suitable problems are those that can be implemented with stream processing and using limited memory, i.e. applications with abundant parallelism.

The Compute Unified Device Architecture (CUDA), supported from the NVIDIA Geforce 8 Series, enables to use GPU as a highly parallel computer for non-graphics applications [3, 16]. CUDA provides an essential high-level development environment with standard C/C++ language. It defines the GPU architecture as a programmable graphic unit which acts as a coprocessor for CPU. It has multiple streaming multiprocessors (SMs), each of them contains several (eight, thirty-two or forty-eight, depending GPU architecture) scalar processors (SPs). The CUDA programming model has two main characteristics: the parallel work through concurrent threads and the memory hierarchy. The user supplies a single source program encompassing both host (CPU) and *kernel* (GPU) code. Each CUDA program consists of multiple phases that are executed on either CPU or GPU. All phases that exhibit little or no data parallelism are implemented in CPU. Contrary, if the phases present much data parallelism, they are coded as *kernel* functions in GPU. A *kernel* function defines the code to be executed by each thread launched in a parallel phase.

3 Sequential Permutation Index

Let $\mathcal{P}$ be a subset of the database U, $\mathcal{P} = \{p_1, p_2, \ldots, p_m\} \subseteq U$, that is called the permutants set. Every element x of the database sorts all the permutants according to the distances to them, thus forming a permutation of $\mathcal{P}$: $\Pi_x = \langle p_{i_1}, p_{i_2}, \ldots p_{i_m} \rangle$. More formally, for an element $x \in U$, its permutation Π_x of $\mathcal{P}$ satisfies $d(x, \Pi_x(i)) \leq d(x, \Pi_x(i+1))$, where the elements at the same distance are taken in arbitrary, but consistent, order. We use $\Pi_x^{-1}(p_{i_j})$ for the *rank* of an element p_{i_j} in the permutation Π_x. If two elements are similar, they will have a similar permutation [2].

Basically, the permutation based algorithm is an example of probabilistic algorithm, it is used to predict proximity between elements, by using their permutations. The algorithm is very simple: In the offline preprocessing stage it is computed the permutation for each element in the database. All these permutations are stored and they form the index. When a query q arrives, its permutation Π_q is computed. Then, the elements in the database are sorted in increasing order of a similarity measurement between permutations, and next they are compared against the query q following this order, until some stopping criterion is achieved. The similarity between two permutations can be measured, for example, by *Kendall Tau*, *Spearman Rho*, or *Spearman Footrule* metrics [17]. All of them are metrics, because they satisfy the aforementioned properties. We use the Spearman Rho metric because it is not expensive to compute and according to the authors in [2] it has a good performance to predict proximity between elements. The square of the Spearman Rho S_ρ metric is defined as $S_\rho(x, q) = S_\rho(\Pi_x, \Pi_q) = \sum_{i=1}^{m} |\Pi_x^{-1}(p_i) - \Pi_q^{-1}(p_i)|^2$.

At query time we first compute the real distances $d(q, p_i)$ for every $p_i \in \mathcal{P}$, then we obtain the permutation Π_q, and next we sort the elements $x \in U$ into increasing order according to $S_\rho(\Pi_x, \Pi_q)$ (the sorting can be done incrementally,

```
**RangeQuery**(element $q$, radius $r$, fraction $f$)
1.  Let $A[1,n]$ be an array of tuples and $U = \{x_1, \ldots, x_n\}$
2.  Compute $\Pi_q^{-1}$
3.  For $i \leftarrow 1$ to $n$ do    $A[i] \leftarrow \langle x_i, S_\rho(\Pi_{x_i}, \Pi_q)\rangle$
4.  **SortIncreasing**($A$) /* by second component of tuples */
5.  For $i \leftarrow 1$ to $fn$ do
6.      $\langle x, s\rangle \leftarrow A[i]$
7.      If $d(q,x) \leq r$ Then Report $x$
```

Algorithm 1: Range query of q with radius r in a permutation index, f DB fraction.

because only some of the first elements are actually needed). Then U is traversed in that sorted order, evaluating the distance $d(q,x)$ for each $x \in U$. For range queries, with radius r, each x that satisfies $d(q,x) \leq r$ is reported, and for k-NN queries the set of the k smallest distances so far, and the corresponding elements, are maintained. Algorithm 1 shows the process for a range query. The efficiency and the quality of the answer obviously depend on f. In [2], the authors discuss a way to obtain good values for f.

4 GPU-CUDA Permutation Index

The Figure 1 shows the GPU-CUDA system to work with a permutation index: the processes of indexing and querying. The Indexing process has two stages and the Querying process four steps. In this last process, we pay special attention to one step: the sorting. The next sections detail the characteristics of each process, their steps and peculiarities.

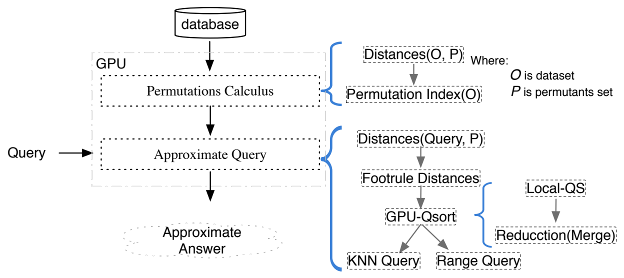


Fig. 1. Indexing and Querying in GPU-CUDA permutation index.

4.1 Building the Permutation Index

Building a permutation index in GPU involves at least two steps. The first step calculates the distance among every object in database and the permutants. The second one sets up the signatures of all objects in database, i.e. all object permutations. The process input is the database and the permutants. At process end, the index is ready to be queried. The idea is to divide the work in threads blocks, each thread calculates the object permutation according to a global permutants set. In the first task ($Distances(O, P)$), the number of blocks will be defined according of the size of the database and the number of threads per block which depends of the quantity of resources required by each block. At the step end, each threads block save in device memory its calculated distances. This stage requires a structure of size $m \times n$ (m: permutants number and n: database size) and an auxiliar structure of fixed size defined in the shared memory of block (It stores the permutants, if the permutants size is greater than auxiliar structure size, the process is repeated until all distances to permutants are calculated). The second step ($Permutation\ Index(O)$) takes all calculated distances in the previous step and determines the permutations of each object in database: its signature. To stablish the object permutation, each thread considers an object in database and sorts the permutants according to their distance. The output of second step is the *Permutation Index*, which is saved in device memory. Its size is $n \times m$.

4.2 Solving Approximate Queries

The pemutation index allows to answer to all kinds of queries in approximated manner. Queries can be *"by range"* or "k-NN". This process implies four steps. In the first, the permutation of query object is computed. This task is carried out by so many threads as permutants exist. The next step is to contrast all permutations in the index with query permutation. Comparison is done through the $Footrule$ distance, one thread by object in database. In the third step, it sorts the calculated $Footrule$ distances. As sorting methodology, we implement the Quick-sort in the GPU, its characteristics are explained bellow. Finally, depending of query kind, the selected objects have to be evaluated. In this evaluation, the *Euclidean distanc*e between query object and each candidate element is calculated again. Only a database percentage is considered for this step, for example the 10% (it can be a parameter). If the query is by range, the elements in the answer will be those that their distances are less than reference range. If it is k-NN query, once each thread computes the *Euclidean distance*, all distances are sorted (using GPU-Qsort) and the results are the first k elements of sorted list.

Considering the sorting algorithm, we describe a parallel Quicksort algorithm for GPU, called GPU-Qsort. The designed algorithm takes into account the highly parallel nature of graphics processors (GPUs) and the CUDA capabilities 1.2 or higher. GPU-Qsort carries out the task into two stages: *Local-Qsort* and *Merge-Reduction.* The first stage, **Local-Qsort**, has a data sequence as input and its output are n sorted subsequences. Each subsequence is ordered by a threads block according to iterative quicksort. Therefore, there are n threads blocks, where the number of threads by block is fix and is determined in relation to the required resources by block. Each block chooses a local pivot (it has to belong to input data list of block) and divides the data sequence in two subsequences: one has the elements smaller than pivot and another has the elements greater or equal than pivot. The pivot is the median among three elements of data subsequence: the first, middle and last element [18]. Each block works independently of other blocks eliminating the need of synchronization among threads of different blocks. In base to the selected pivot, all elements lower than the pivot are moved to a position to the pivot's left, and the greater or equal are shifted to the pivot's right. The task is made by using shared memory and each thread can determine itself the position for its element in shared structure (using CUDA atomic functions).

The process is applied iteratively over two subsequences. It is possible if it uses an stack. The stack saves all subsequences that still remain to be sorted. When there are two ready subsequences to work, one is selected and the another is pushed in the stack. If one subsequence is sorted, the subsequence in the top of stack is selected to work. The iterative process ends when the stack is empty and the list is sorted. When the number of elements in the sequence is lower than eight, it is sorted in sequential manner, because the process overhead is too large compared to sequence size. At the end of stage, each one of n blocks copies its sorted subsequence to device memory. The output is n sorted subsequence. For second stage of GPU-Qsort, **Merge-Reduction**, its input is n sorted list and the output is whole sorted sequence. This phase makes a reduction, the reduction operation is a merge of sorted lists. A block merges two list at a time. In consequence, $log_2 n$ iterations are necessary to find the final result. This stage requires $\lceil \frac{n}{2} \rceil$ blocks with thirty two threads per each and an auxiliary structure in device memory. In both stages, different techniques are used to optimize the performance, they are the use of shared memory, anticipatory copies and coalesced access to global memory.

5 Solving Parallely Many Queries

In large-scale systems such as Web Search Engines indexing multimedia content, it is critical to deal efficiently with streams of queries rather than with single queries. Therefore, it is not enough to speed up the time to answer only one query, but it is necessary to leverage the capabilities of the GPU to parallely answer several queries. So we have to show how to achieve efficient and scalable performance in this context. We need to devise algorithms and optimizations specially tailored to support high-performance parallel query processing in GPU. GPU has characteristics of software and hardware which allow us to think in to solve many approximated queries in parallel. The represented system in Figure 1 is modified and it is shown in Figure 2. In this, it can be observed that the permutation index is built once and then is used to answer many queries.

In order to answer parallely many approximate queries, GPU receives the queries set and it has to solve all of them. Each query, in parallel, applies the process explained in 4.2, therefore the number of needed resources for this is equal to the resources amount to compute one query multiplied the number of queries solved in parallel. The number of queries to solve in parallel is determined according to the GPU resources mainly its memory. If q are parallel queries, m the needed memory quantity per query and i the needed memory by permutation index, $q * m + i$ is the total required memory to solve q queries in parallel.

Once the q parallel queries are solved, the results are sent from the GPU to the CPU through a single transfer via PCI-Express.

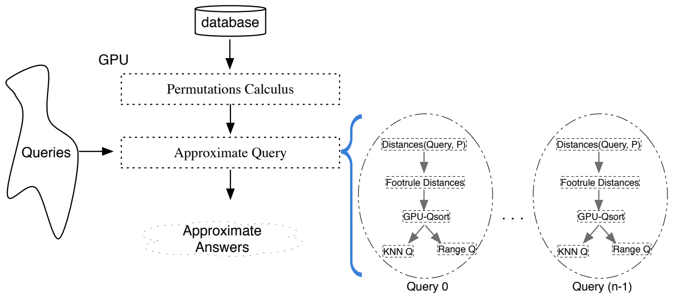


Fig. 2. Solving q queries in GPU-CUDA permutation index.

Solving many queries in parallel involves carefully manage the blocks and their threads. At the same time, blocks of different queries are accessed in parallel. Hence it is important a good administration of threads: which query it is solved and which database element it is responsible. The task is possible by establishing a relationship among *Thread Id*, *Block Id*, *Query Id*, and *Database Element*.

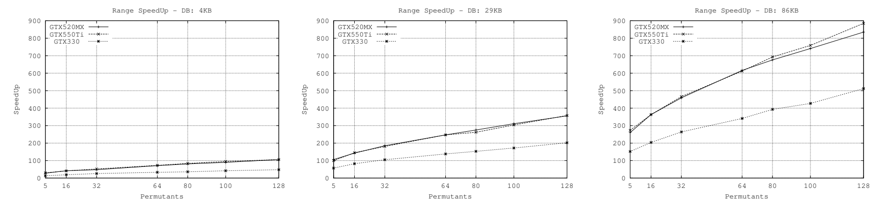
6 Experimental Results

Our experiments considered different database sizes: 4KB, 29KB, and 84KB, on a metric database consisting of English words and using the *Levenshtein* distance, also called *edit* distance (that is, the minimum number of character insertions, deletions, and substitutions needed to make two strings equal). The analysis was made for three GeForce GPU whose characteristics (Global Memory, SM, SP, Clock rate, Compute Capability) are GTX330: (512MB, 6, 48, 1.04GHz, 1.2), GTX520MX: (1024MB, 1, 48, 1.8GHz, 2.1) and GTX550Ti: (1024MB, 4, 192, 1.96GHz, 2.1). The CPU is an Intel core i3, 2.13 GHz and 3 GB of memory. The results are expressed in Speed up ($Sp = \frac{Time_{CPUSec}}{Time_{GPUPar}}$). The three GPUs are used to analyse the behavior of our proposal over different number of resources.

In this paper, we do not display the speed up of construction of *Permutation Index*. These results are illustrated in [19].

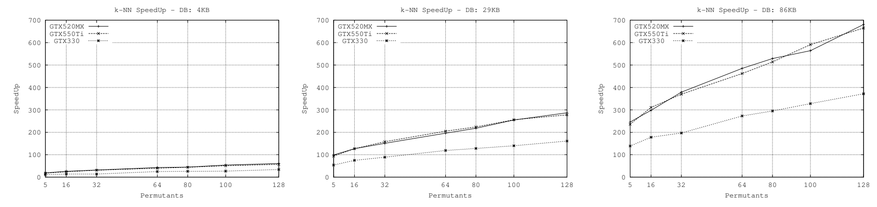
Figures 3 and 4 show the obtained acceleration in range queries (3) and k-NN (4) queries for three database sizes and different number of permutants. In these results, 80 queries are solved in parallel. As it can be noticed *Range queries* show improvements respect to k-NN queries, but in both cases the achieved speed up is very good. In all cases, it is clear the influence of database size, but evenly we accomplish good performance. The best case is for largest database and maximum number of permutants.

Tables I shows the obtained throughput (number of queries by second) by our implementation. The results clearly show the benefits for all used architectures of GPU. In every case and query kind, the number of queries by second is high. Also we can observe that the number of permutants is not so important, for each



(a) 4KB (b) 29KB (c) 84KB

Fig. 3. Speedup of Range search Queries on three different GPUs.



(a) 4KB (b) 29KB (c) 84KB

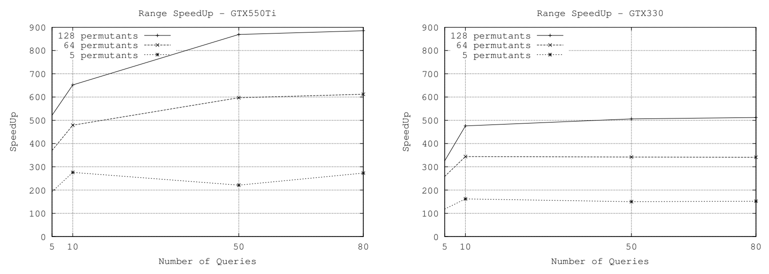
Fig. 4. Speedup of k-NN search Queries on three different GPUs.

GPU architecture and for all number of permutants, the throughput is similar, quasi constant.

Table 1. Range and *k*-NN search Throughput.

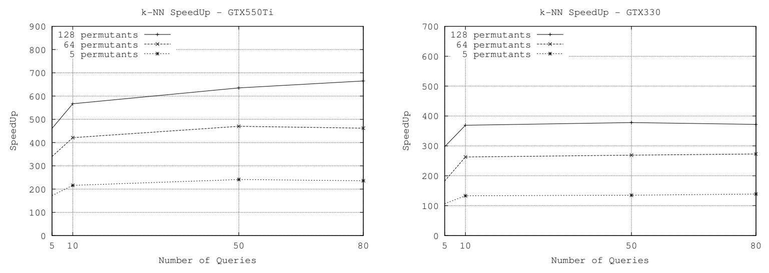
# Permut.	GTX520MX	GTX550Ti	GTX330	GTX520MX	GTX550Ti	GTX330
128	27639,72	29310,63	16973,21	19824,25	19377,68	10850,85
100	28188,86	28907,35	16279,76	18816,38	19696,23	10956,71
80	28921,82	29621,19	16828,75	19771,86	19203,07	11051,72
64	29539,57	29362,77	16379,24	19797,83	18857,32	11137,65
32	29144,71	29582,42	16774,19	19774,12	19289,62	10263,80
16	29255,39	29248,81	16464,29	18785,79	19645,99	11237,29
5	28197,27	29604,32	16474,46	19906,59	19121,16	11262,48

Figures 5 and 6 resume the behavior of two operations: Range Query (Figure 5) and *k*-NN Query (Figure 6), for biggest database and three numbers of permutants (5, 64, 128) when we vary the number of parallel queries in tow GPUs. It can be seem that the best speed up was obtained when the number of queries is equal to 80 and the number of permutants is the maximum. Also it is clear the influence of GPU architecture, when it has more resources, better speed up are achieved. Figures 5(a) and 6(a) depict these results.



(a) GTX550Ti (b) GTX330

Fig. 5. Speedup of Range Search Queries for different number of parallel queries.



(a) GTX550Ti (b) GTX330

Fig. 6. Speedup of k-NN Search Queries for different number of parallel queries.

6.1 GPU-Qsort with Other Solution

There are many quick sort library, one of them is *Thrust*. It is part of CUDA repositories. *Thrust* library provides a collection of fundamental parallel algorithms such as *scan*, *sort* and *reduction*. It solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With this library, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation [20]. A disadvantage of *Thrust* is that it can isolate the developer from the hardware and expose only a subset of the hardware capabilities. In some circumstances, the C++ interface can become too awkward or verbose [21].

We compared our implementation with a solution based in *Thrust* library. We used the *Thrust* as a black box. Figure 7 shows the comparison considering four frame sizes. The results are the average of one hundred executions. In the Figure 7, we can observe that our implementation obtains better speed up than the solution using Thrust library. Besides it is important to notice the independence of GPU-Qsort from GPU characteristics, it works fine in all GPU.

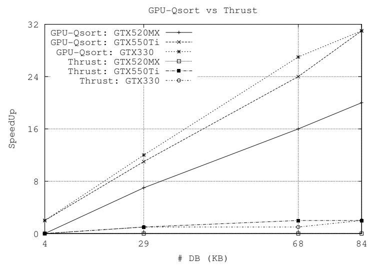


Fig. 7. Speedup of GPU-Qsort and Thrust on three different GPUs.

7 Conclusions

As it is mentioned before, in large-scale systems such as Web Search Engines indexing multimedia content, it is critical to deal efficiently with streams of queries rather than with single queries. Therefore, it is not enough to speed up the time to answer only one query, but it is necessary to solve several queries at the same time. In this work we present a solution to solve many queries in parallel taking advantage of GPU architecture: it is a massively parallel architecture, it has a high throughput because its capacity of parallel processing for thousands of threads.

In this work we show an implementation that uses a *Pemutation Index* to solve approximate similarity search on a database of English words. However, it is possible to easily extend our proposal to other metric databases of different data nature, such as vectors, documents, DNA sequences, images, music, among others. The empirical results have shown improvements in every different considered architecture of GPU. Both obtained speed up and throughput are very good, showing better performance when the load work is hard.

In the future, we plan to make an exhaustive experimental evaluation considering others kinds of database, comparing with other solutions that apply GPU in the scenario of metric space approximate searches. We need also to evaluate retrieval effectiveness of the answer of the *Permutation Index*, as the number of objects directly compared with the query grows, by using *Recall* and *Precission* measures.

8 Acknowledgements

We wish to thank to the UNSL for allowing us the access to their computational resources. This research has been partially supported by Project UNSL-PROICO-30310 and Project UNSL-PROICO-330303.

References

1. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

2. E. Chávez, K. Figueroa, and G. Navarro, "Proximity searching in high dimensional spaces with a proximity preserving order," in *Proc. 4th Mexican International Conference on Artificial Intelligence (MICAI)*, ser. LNAI 3789, 2005, pp. 405–414.
3. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, A Hands on Approach*. Elsevier, Morgan Kaufmann, 2010.
4. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," in *IEEE*, vol. 96, no. 5, 2008, pp. 879 – 899.
5. R. J. Barrientos, J. Gomez, C. Tenllado, M. Prieto, and M. Marin, "kNN Query Processing in Metric Spaces using GPUs," vol. 6852, 2011, pp. 380–392.
6. V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "k-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature matching," in *IEEE Intern. Conf. on Image Processing*, Hong Kong, Sept. 2010.
7. K. Kato and T. Hosino, "Solving k-nearest neighbor problem on multiple graphics processors," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID*, ACM, Ed., 2010, pp. 769–773.
8. S. Liang, Y. Liu, C. Wang, and L. Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," in *IEEE 2nd Symposium on Web Society (SWS)*, 2010, pp. 53 – 60.
9. R. Uribe, P. Valero, E. Arias, J. L. Sánchez, and D. Cazorla, "A GPU-Based Implementation for Range Queries on Spaghettis Data Structure," in *ICCSA (1)*, ser. Lecture Notes in Computer Science, vol. 6782. Springer, 2011, pp. 615–629.
10. P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search: The Metric Space Approach*, ser. Advances in Database Systems, vol.32. Springer, 2006.
11. K. Figueroa, E. Chávez, G. Navarro, and R. Paredes, "Speeding up spatial approximation search in metric spaces," *ACM Journal of Experimental Algorithmics*, vol. 14, p. article 3.6, 2009.
12. P. Ciaccia and M. Patella, "Approximate and probabilistic methods," *SIGSPATIAL Special*, vol. 2, no. 2, pp. 16–19, Jul. 2010.
13. B. Benjamin and G. Navarro, "Probabilistic proximity searching algorithms based on compact partitions," *Discrete Algorithms*, vol. 2, no. 1, pp. 115–134, Mar. 2004.
14. A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *The 12th intern. conf. on Information and knowledge management*, ser. CIKM '03. New York, NY, USA: ACM, 2003, pp. 91–98.
15. F. Moreno, L. Mic, and J. Oncina, "A modification of the laesa algorithm for approximated k-nn classification," *Pattern Recognition Letters*, vol. 24, no. 13, pp. 47 – 53, 2003.
16. NVIDIA, "Nvidia cuda compute unified device architecture, programming guide version 4.2." in *NVIDIA*, 2012.
17. R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *Proc. of the 40th annual ACM-SIAM symposium on Discrete algorithms, SODA '03*. Philadelphia, USA: Society for Industrial and Applied Mathematics, 2003, pp. 28–36.
18. R. Singleton, "Algorithm 347: an efficient algorithm for sorting with minimal storage [m1]," *Commun. ACM*, vol. 12, no. 3, pp. 185–186, Mar. 1969.
19. M. Lopresti, N. Miranda, F. Piccoli, and N. Reyes, "Efficient similarity search on multimedia databases," in *XVIII Congreso Argentino de Ciencias de la Computacin, CACIC 2012*, 2012, pp. 1079–1088.
20. J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010.
21. R. Farber, *CUDA Application Design and Development*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.