Processing the Signature Quadratic Form Distance on Many-Core GPU Architectures

Martin Kruliš ° Jakub Lokoč ° Christian Beecks • Tomáš Skopal ° Thomas Seidl •
°SIRET Research Group, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic
°{krulis,lokoc,skopal}@ksi.mff.cuni.cz
•Data Management and Data Exploration Group, RWTH Aachen University, Germany
•{beecks,seidl}@cs.rwth-aachen.de

ABSTRACT

The Signature Quadratic Form Distance on feature signatures represents a flexible distance-based similarity model for effective content-based multimedia retrieval. Although metric indexing approaches are able to speed up query processing by two orders of magnitude, their applicability to large-scale multimedia databases containing billions of images is still a challenging issue. In this paper, we propose the utilization of GPUs for efficient query processing with the Signature Quadratic Form Distance. We show how to process multiple distance computations in parallel and demonstrate efficient query processing by comparing many-core GPU with multi-core CPU implementations.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Performance, Design, Algorithm

Keywords

GPU, many-core, quadratic form distance, similarity search

1. INTRODUCTION

Multimedia retrieval systems frequently store billions of images and provide users with different ways of searching and browsing (e.g., catalog-based or keyword-based search). However, effective yet efficient techniques for content-based similarity search are still a big issue. To this end, multimedia retrieval systems are designed based on advanced similarity models consisting of image representations and similarity/distance measures. A flexible way to represent the content of an image is by means of *feature signatures* [7].

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

In general, a feature signature of an image is a set consisting of multiple local image features, where the length of a feature signature is not fixed (to distinguish more or less complex images). However, the comparison of feature signatures requires more sophisticated and computationally expensive adaptive distance measures [2], such as the Earth Mover's Distance (EMD) [7] or the Signature Quadratic Form Distance (SQFD) [3]. In this paper, we focus on the latter, as the SQFD shows high retrieval quality and lower time complexity compared to the EMD ($O(n^2)$ vs. $O(n^4)$).

In order to reduce the computational effort, *metric indexing* [8] approaches have been applied to the SQFD. It has been shown, that pivot tables [1] and ptolemaic indexing [5] reach a speed-up of over two orders of magnitude with respect to the sequential scan. Nevertheless, by using metric indexing approaches, this speed-up is generally limited due to the intrinsic dimensionality [8]. Thus, in order to use the SQFD for large-scale image retrieval, we propose to parallelize the SQFD query processing.

In this paper, we consider many-core GPU devices for parallel SQFD query processing. While parallel multi-core CPU processing is straightforward and supported by many development tools, designing efficient algorithms for GPUs is a challenging task for content-based retrieval purposes. Although GPUs generally contain more cores than CPUs, they suffer from slow data transfer rates and code execution restrictions. We discuss GPU processing limitations and introduce a new schema for efficient SQFD query processing utilizing the combination of metric indexing approaches and the computational power of GPUs.

The paper is organized as follows. Section 2 describes the SQFD. Section 3 discusses the most important aspects of GPU architectures, while Section 4 describes our SQFD implementation for GPUs. Section 5 presents the experimental results, and Section 6 concludes this paper.

2. SIGNATURE QUADRATIC FORM DIST.

The Signature Quadratic Form Distance (SQFD) [3] is an adaptive distance-based similarity measure for feature signatures. It is defined as follows.

Definition 1 (SQFD). *Given two feature signatures* $S^q = \{\langle r_i^q, w_i^q \rangle\}_{i=1}^n$ *and* $S^o = \{\langle r_i^o, w_i^o \rangle\}_{i=1}^m$ *and a similarity function* $f_s : \mathbb{F} \times \mathbb{F} \to \mathbb{R}$ *over a feature space* $\mathbb{F}$*, the signature quadratic form distance* SQFD_{f_s} *between* S^q *and* S^o *is defined as:*

$$\mathrm{SQFD}_{f_s}(S^q, S^o) = \sqrt{(w_q \mid -w_o) \cdot A_{f_s} \cdot (w_q \mid -w_o)^T},$$

where $A_{f_s} \in \mathbb{R}^{(n+m)\times(n+m)}$ is the similarity matrix arising from applying the similarity function f_s to the corresponding feature representatives, i.e., $a_{ij} = f_s(r_i, r_j)$. Furthermore, $w_q = (w_1^q, \ldots, w_n^q)$ and $w_o = (w_1^o, \ldots, w_m^o)$ form weight vectors, and $(w_q \mid -w_o) = (w_1^q, \ldots, w_n^q, -w_1^o, \ldots, -w_m^o)$ denotes the concatenation of weights w_q and $-w_o$.

The similarity function f_s is used to determine similarity values between all pairs of representatives from the feature signatures. In our implementation we use the similarity function $f_s(r_i, r_j) = e^{-\alpha L_2(r_i, r_j)^2}$, where α is a constant for controlling the precision-indexability tradeoff, as investigated in our previous works [1, 5], and L_2 denotes the Euclidean distance.

3. GPU FUNDAMENTALS

In general, GPU architectures [6] differ from CPU architectures in multiple ways. In the remainder of this section, we describe two major differences, the *thread execution* and *memory organization*, which have direct impact on the design of our SQFD implementation.

3.1 Thread Execution

The first difference is the specific program execution. Portions of code, which are executed on the GPU, are called *kernels*. A kernel is a function that is invoked multiple times simultaneously, so that all spawned threads execute the same code. Each spawned thread gets the same set of calling arguments and a unique identifier, which is used to select proper parts of the parallel work.

The thread managing and context switching capabilities of the GPU are very advanced. Thus, it is usually better to create a multitude of threads, even if they execute only a few instructions each, in order to optimize the load balancing. In addition, fast context switching capabilities of the GPU are used to hide latency of global memory transactions.

Threads are aggregated into small bundles called groups. A group usually contains tens to hundreds of threads which execute the kernel in SIMT[1] or virtual SIMT fashion. Every thread executes the same instruction, but it has its own set of registers, thus working on different portions of the data. The SIMT execution suffers from branching problems when different threads in the group choose different branches, for instance when executing 'if' statements. To execute conditional code properly, all branches must be executed by all threads and each thread masks instruction execution according to local result of the condition. On the other hand, the SIMT approach eases synchronization within the group and allows threads to communicate and collaborate through shared local memory.

3.2 Memory Organization

The second difference is the memory organization, which is divided into the following memory address spaces:

- host memory,
- global memory,
- local memory,
- and private memory.

In fact, the *host memory* is the operational memory of the computer. It is directly accessible by the CPU, but it cannot be accessed by any peripheral device such as the GPU. Input data needs to be transferred from the host memory to the graphic device memory (global memory), and the results need to be transferred back when the kernel execution finishes. This transfer uses the PCI-Express bus, which is rather slow in comparison with the internal memory bus.

The *global memory* is directly accessible from GPU cores. Input data and computed results of a kernel are stored here. The global memory bus shows both high latency and high bandwidth. In order to access the global memory optimally, threads in one group are encouraged to use *coalesced loads*. A coalesced load is performed when all threads of a group load or store a contiguous memory area, so that each thread transfers a single 4-byte word of this block.

The *local memory* is shared among threads within one group. It is very small (tens of kB) but almost as fast as GPU registers. The local memory can play role of program-managed cache for global memory, or the threads may share intermediate results in here while they cooperate on a task.

Finally, the *private memory* exclusively belongs to a single thread and corresponds to the GPU core registers. Private memory size is very limited (tens to hundreds of words), therefore it is suitable just for a few local variables.

4. QUERYING BY SQFD ON A GPU

In our approach, we consider both the parallel execution of multiple SQFD computations during the query evaluation as well as the parallel computation of a single SQFD between two feature signatures.

4.1 Multiple SQFD Computations

Since the SQFD is computed between the query signature and each database signature, it would be ineffective to execute each computation separately on the GPU due to the latencies caused by data transfer and kernel executions. Therefore, we perform a block-wise computation of multiple SQFDs in parallel. Each block contains $N + 1$ feature signatures. The first feature signature is the query signature and remaining N feature signatures are database signatures, thus each block yields a vector of N distances as a result. The choice of N is essential for good performance. In general, a large number of N performs better. However, at least two blocks of feature signatures must fit each GPU device in order to process the first block and transfer the data of the other one.

Each SQFD is computed by a group of 256 threads, thus $256 \times N$ threads are spawned for one block. The constant 256 was selected based on current hardware capabilities. We have assigned one thread group to compute a single SQFD in a block, because these threads benefit from shared local memory, as the group does cache the input data from global memory and keeps intermediate results. Using multiple groups to compute one SQFD would be problematic as the groups do not have any effective means of communication. The opposite approach (using one group to compute multiple SQFDs) is feasible. However, the parallelism would not be exploited any further and many technical complications would arise due to the limited size of local memory.

4.2 Computing The Distance in Parallel

The SQFD between two feature signatures has been defined in Definition 1. For the sake of parallelism, we compute the items of the similarity matrix A_{f_s}, and multiply them directly with the corresponding weights of $(w_q \mid -w_o)$. Fi-

[1]Single Instruction Multiple Threads

nally, we compute a sum of every item in the matrix and we find its square root. Our algorithm has the following phases:

1. Load feature signatures into local memory
2. Compute the similarity matrix A_{f_s} and multiply its entries by corresponding entries in the weight vectors
3. Sum up items in the matrix and yield the square root

Data are loaded into local memory, since they are required multiple times and it would be ineffective to load them from global memory each time. Furthermore, the loading is more efficient when all threads cooperate in coalesced loads.

The similarity matrix has $(m+n)\times(m+n)$ entries, where m and n are the numbers of feature representatives in S^q and S^o, respectively. Since $m+n$ is usually smaller than 256 and varies for each feature signature, we use an irregular mapping of similarity matrix items to threads.

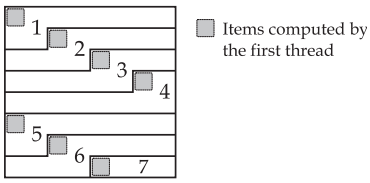

Figure 1: Matrix work decomposition

Figure 1 depicts the mapping scheme, where each area represents items being computed simultaneously. The numbers indicate in which step the items will be processed. In the last step, the remaining part of the similarity matrix may be smaller than total number of threads. In such case, some threads will remain idle in this step.

The entire similarity matrix is not stored in memory since only a sum of its items is required. When a thread computes a new item in the similarity matrix, its value is added to a partial sum and the item itself is discarded. Even though this method requires significantly less memory, it creates a synchronization problem as multiple items are computed and added to the partial sum concurrently. To avoid explicit synchronization, every thread is provided with its own instance of partial sum.

When the second phase terminates, the total sum of the partial sums of each thread is computed. This total sum is only computed by the first thread, which is also responsible for finding the square root and for writing the computed distance into the global memory. The total sum can also be computed cooperatively by all threads in logarithmic time; however, such improvement has no measurable impact on the performance as the time required by the second phase dominates significantly over the time required by the final summation.

4.3 Integration to Query Evaluation System

We have described how to compute distances between a query signature and a block of database signatures on the GPU. How to integrate this functionality into a system that computes range and kNN queries is explained in the remainder of this section.

The rest of the query processor treats our SQFD implementation as an asynchronous operation that does not block the CPU when started, and the system can wait for its termination. The system may start as many operations as required, and the operations are queued and distributed over available GPU devices equally. The architecture is flexible, thus it can be easily used with any type of distance-based index.

4.3.1 Range Query

The sequential range query algorithm is easy to implement. The database is divided into blocks of appropriate size[2] and all blocks are enqueued for GPU processing. The system waits for all SQFD computations to complete, and the computed distances are filtered on the CPU to exclude objects outside of the query range from the result.

This method can be improved easily with pivot table indexing (either metric or ptolemaic variant). First, distances from the query to all pivot objects are computed on the GPU. Afterwards, the CPU prefilters database objects using the pivot table. The objects which are not excluded in the prefiltering step are formed into blocks. A block is passed to the GPU as soon as it has a sufficient number of objects (we use $8,000$, observed empirically as optimal).

4.3.2 kNN Query

The kNN query evaluation is slightly more complicated. When no indexing is used, it works very much like sequential range query. When the pivot table indexing is employed, some additional modifications are required. The problem is that the kNN query has no fixed query range for the pivot table prefiltering, as this range is dynamically refined during the kNN query processing using heuristics.

In order to adapt to the heuristics, we have limited the block size to a value of 64. Also, there are at most $2g$ blocks pending where g is the number of GPU devices available. These constants have been chosen empirically[3]. When the limit of pending blocks is reached, the system waits for the first enqueued block to finish, its results are taken, and the query range is refined. This way a pipeline effect is achieved, so that the CPU prefilters the database objects and refines the kNN set while the GPU computes the SQFD.

5. EXPERIMENTS

We conduct GPU and CPU tests with up to 12 cores on a desktop PC based on six-core Intel Core i7 870 CPU with hyper-threading clocked at 2.93 GHz. The machine was equipped with 16 GB of RAM and two NVIDIA GTX 580 GPU cards with 512 CUDA cores and 1.5 GB of RAM each. More extensive multi-core CPU tests for 48 cores were executed on a Dell M910 server with four six-core Intel Xeon E7540 processors with hyper-threading (i.e., 48 logical cores) clocked at 2.0 GHz. The server was equipped with 128 GB of RAM organized as 4-node NUMA. A RedHat Enterprise Linux 6 was used as operating system on both machines.

Our approach was tested on the ALOI database [4] which consists of $72,000$ images. Although the selected database was not very large, its size was sufficient to demonstrate the speed-up of our method. To perform test using Pivot tables, we have randomly selected 32 pivots from the database. The queries were selected as 100 random objects from the database (and excluded from indexing). All tests were performed for $\alpha \in \{3, 0.2, 0.01\}$, where $\alpha = 3$ exhibits the best

[2] We use as large blocks as possible that fit the GPU memory.
[3] Actually, these constants are suitable only for $\alpha = 0.2$ and $\alpha = 0.01$. The value $\alpha = 3$ requires the largest possible blocks since it does not benefit much from indexing.

precision, $\alpha = 0.01$ offers the best indexability, and $\alpha = 0.2$ is the best compromise between precision and indexability.

5.1 Simple Sequential Scan

The first test demonstrates the behavior of the sequential scan, i.e. either range or kNN query, without pivot table indexing. The measured values represent real times required for computing a vector of distances between the query object and all database objects. Each test was performed for 32 randomly chosen objects and the presented values are arithmetic averages of the measured times. Each test is denoted as follows. The n CPU stands for SQFD running on n processor cores and m GPU means that the test was using m GPU devices (512 cores each). The data type in brackets indicates the floating point precision used for intermediate results.

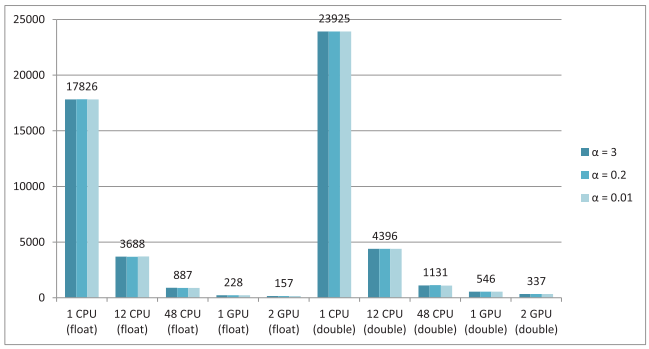

Figure 2: Times[ms] of sequential scan

Figure **??** clearly indicates that the GPU approach is faster by two orders of magnitude than sequential version of the algorithm running on CPU. Furthermore, the GPU is significantly faster (5.6× in float and 3.4× in double) than 4-CPU NUMA server with 48 logical cores (which is financially an order of magnitude more expensive than a desktop PC with GPU cards). We have also observed that different values of α have no effect on the evaluation speed.

5.2 SQFD with Ptolemaic Indexing

In the second experiment, we have tested the GPU acceleration for kNN queries, where objects are prefiltered using Ptolemaic indexing on CPU [5]. The results are denoted as in the previous test, except for only float precision was used for intermediate results.

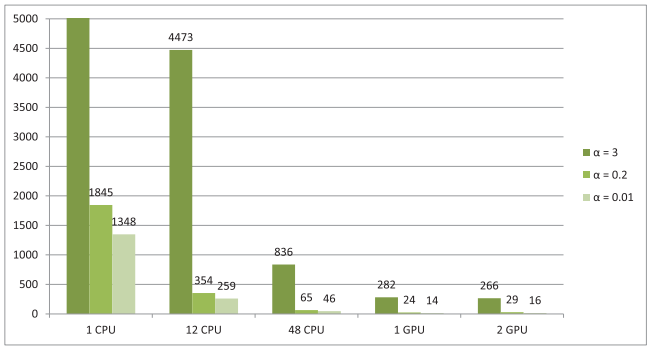

Figure 3: Times[ms] of 10NN with ptolemaic index

Figure **??** verifies that 10NN queries with indexing can be accelerated by computing SQFD on GPU, although the speed-up is slightly less significant than in the previous test. The average speed-up of a GPU w.r.t. 48 CPU cores (2.98×) proves that GPU can accelerate also indexing methods significantly. Despite our efforts, the 2-GPU version was slower than one-GPU for $\alpha \in \{0.2, 0.01\}$. We belive that this was the result of fragile balance between CPU and GPU workload at the moment the CPU prefilters the database objects. We are planning to study this phenomenon further and improve our method by moving the index prefiltering to GPU as well.

6. CONCLUSIONS

In this paper, we have shown how to exploit the computational power of GPUs for an efficient similarity query processing for multimedia retrieval models utilizing the Signature Quadratic Form Distance. We have experimentally proved that the speedup is considerable even in comparison with expensive NUMA server.

In the future, we plan to employ GPUs in other operations (such as index filtering) and to create a sophisticated large-scale multimedia retrieval system utilizing SQFD on feature signatures as similarity model.

7. ACKNOWLEDGMENTS

This research has been supported by Czech Science Foundation (GAČR) projects 201/09/0683 and 202/11/0968, by the grant agency of Charles University (GAUK) project no. 277911, and by the Deutsche Forschungsgemeinschaft within the Collaborative Research Center SFB 686.

8. REFERENCES

[1] C. Beecks, J. Lokoč, T. Seidl, and T. Skopal. Indexing the signature quadratic form distance for efficient content-based multimedia retrieval. In *Proc. ACM Int. Conf. on Multimedia Retrieval*, pages 24:1–24:8, 2011.

[2] C. Beecks, M. S. Uysal, and T. Seidl. A comparative study of similarity measures for content-based multimedia retrieval. In *Proc. IEEE ICME*, pages 1552–1557, 2010.

[3] C. Beecks, M. S. Uysal, and T. Seidl. Signature quadratic form distance. In *Proc. ACM CIVR*, pages 438–445, 2010.

[4] J.-M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The Amsterdam Library of Object Images. *IJCV*, 61(1):103–112, 2005.

[5] J. Lokoč, M. Hetland, T. Skopal, and C. Beecks. Ptolemaic indexing of the signature quadratic form distance. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, pages 9–16. ACM, 2011.

[6] NVIDIA. Fermi GPU Architecture. http://www.nvidia.com/object/fermi_architecture.html.

[7] Y. Rubner, C. Tomasi, and L. J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *IJCV*, 40(2):99–121, 2000.

[8] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.