Human Re-identification System On Highly Parallel GPU and CPU Architectures

Slawomir Bak, Krzysztof Kurowski, Krystyna Napierala

Human Re-identification System On Highly Parallel GPU and CPU Architectures

Sławomir Bąk[1], Krzysztof Kurowski[2], and Krystyna Napierała[3]

[1]INRIA Sophia Antipolis, PULSAR group, France
{slawomir.bak@inria.fr}
[2]Poznań Supercomputing and Networking Center, Poland
{krzysztof.kurowski@man.poznan.pl}
[3]Institute of Computing Science, Poznań University of Technology, Poland,
{krysia@man.poznan.pl}

Abstract. The paper presents a new approach to the human reindetification problem using covariance features. In many cases, a distance operator between signatures, based on generalized eigenvalues, has to be computed efficiently, especially once the real-time response time is expected from the system. This is a challenging problem as many procedures are in fact computationally intensive tasks and must be repeated constantly. To deal with this problem we have successfully designed and then tested a new video surveillance system. To obtain the required high efficiency we took the advantage of highly parallel computing architectures such as FPGA, GPU and CPU units to perform calculations. However, we had to propose a new GPU-based implementation of the distance operator for querying the example database. Thus, in this paper we present experimental evaluation of the proposed solution in the light of the database response time depending on its size.

Keywords: Re-identification, Covariance Matrix, Generalized Eigenvalues, High Performance Computing, GPU

1 Introduction

Human re-identification is one of the most challenging and important problems in computer vision and pattern recognition. The re-identification problem can be defined as a determination whether a given person of interest has already been observed over a network of cameras. This issue (also called the *person re-identification problem*) can be considered on different levels depending on information cues currently available in the system. Biometrics such as *face*, *iris* or *gait* can be used to recognize identities. Nevertheless, in most video surveillance scenarios such detailed information is not available due to a low video resolution or a difficult segmentation (crowded environments, *e.g.* airports, metro stations). Therefore a robust modeling of a global appearance of an individual is necessary to re-identify a given person of interest. In these identification techniques (named *appearance-based approaches*) clothing is the most reliable information about an identity of an individual (there is an assumption that individuals wear the same clothes between different sightings). A model of appearance has to handle differences in illumination, pose and camera parameters to allow matching

appearances of the same individual observed in different cameras. High accuracy of re-idenfication approaches can only be achieved using an appearance representation based on descriptors which are invariant across different camera views. Recently, a covariance descriptor [9] has proved its effectiveness in recognition [1] and classification approaches [8]. It has been shown that the performance of the covariance features is superior to other methods as rotation and illumination changes are absorbed by the covariance matrix [1]. Moreover, *integral images* [10] used for fast covariance computation make this descriptor very efficient concerning extraction of the covariance.

However, as covariance matrices does not lay on Euclidean space, there is necessary to apply complex differential geometry to compute a distance between two covariances. The distance operator is computationally heavy as it requires solving *the generalized eigenvalues problem*. As a consequence, matching of the covariance descriptors is slower than matching of other computer vision descriptors, which are usually represented by vectors laying on Euclidean space. This often makes covariance-based approaches difficult to apply in real-time systems in spite of their effectiveness. Moreover, in the person re-identication problem there is usually a large number of candidate matches which makes the issue much more challenging (concerning the matching accuracy as well as the response-time of a database of human appearances). Hence, we propose a new hybrid architecture based on Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) accelerators to take advantage of high performance computing and make covariance-based approaches applicable to large-scale databases. This paper makes the following contributions:

- We describe a new GPU- and FPGA-based architecture for the person re-identification problem. This architecture can be easily adjusted to more general video surveillance problems (such as object recognition or object classification)(Section 3).
- We propose an implementation for finding generalized eigenvalues and eigenvectors for distance operator, using NVIDIA GPU architecture (Section 5).

We evaluate our approach in Section 6 before concluding the paper.

2 Motivations and Related Work

There is an increasing demand for effective surveillance systems, *i.e.* systems that can perform low-cost, low-power and high-speed operations. Recently, recognition problems became one of the most important tasks in video surveillance. As recognition is an extremely difficult task, the existing approaches are computationally heavy. Thus, a new high performance architectures are necessary to apply these approaches to real-time systems. In [7] biologically-inspired algorithms are adjusted to GPU to perform large-scale object recognition. Similarly, in [6] GPU-based neural network is presented to recognize human faces.

We offer a surveillance system based on FPGA and GPU architectures giving much better computing facilities in comparison with traditional CPU-based systems. The most demanding part of the system – the generalized eigenvalues

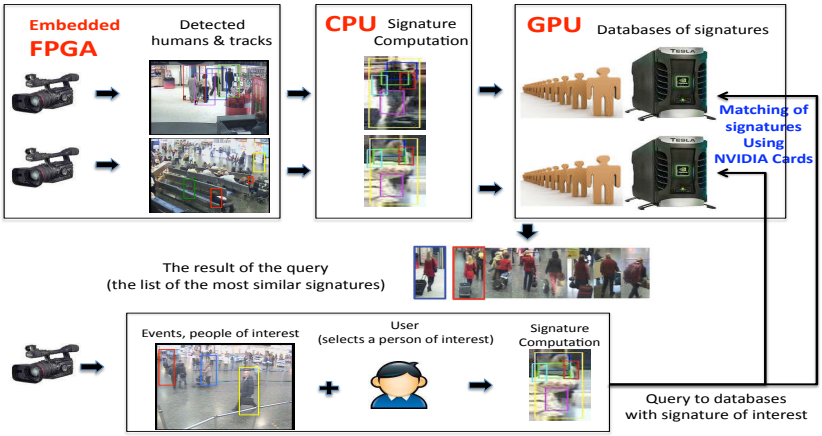


Fig. 1. The GPU-based architecture for the person re-identification.

calculations – is performed using the GPU architecture. There are already a few approaches of finding eigenvalues of symmetric matrices using GPUs [5], but these implementations are focused on computation of the eigenvalues of large matrices (the implementations are optimized for matrices larger than 1024). In contrary to these approaches, we concern a domain-specific problem where it is necessary to solve the generalized eigenvalues problem for a large number of small matrices (the covariance descriptor is mostly represented by square matrix of size between 8 and 16). To the best of our knowledge, we are the first to propose an implementation for finding the generalized eigenvalues and eigenvectors of a large amount of small matrices using NVIDIA GPU cards.

3 System Architecture

Our GPU- and FPGA-based surveillance system for the person re-identification problem (Fig. 1) is designed to assign the tasks to the most suitable architectures. The system consists of a network of cameras with embedded FPGA units, which preprocess a video stream (image denoising, object detection and classification). FPGA units are well suited for video processing tasks usually exposing a high level of parallelism. Therefore, only the necessary information (*blobs* of interest) is sent in the network, without the need of transfering the whole video stream to the central unit. The partially transformed data is collected on a central unit with a CPU and GPU processor. In our approach *a human appearance* is represented by a set of covariance matrices extracted on different resolutions from detected body parts [1]. In total, the human signature is represented by 26 covariance matrices of the size 11. Covariance signature can be computed on a CPU as there exist an efficient way to extract covariances using *integral images*. Signature is then stored on a GPU unit which serves as a database of signatures. The advantage of such a solution is that, first, CPU unit is offloaded from storing this information, and second, when the distance is calculated, the data is already stored on a proper unit. We use a Tesla S1070 with four GPU units, on which there is 4GB of available global memory for each unit. Taking into account

the free space needed for calculations for a query to a database, about 200,000 signatures can be stored in the database on one unit, which is sufficient for the purposes of our system. The user (administrator of a surveillance system) can select an object of interest and query the database with a new signature. The new signature is compared using the distance operator to all signatures in the database. Distance operator is calculated in parallel directly on the GPU unit. The result is a list of the most similar signatures.

The most important part in the system is the database stored on GPU and the calculation of distance on this architecture. The preprocessing on FPGA is not that crucial for making the system real-time, therefore we show only how to calculate the distance on GPU (Sec. 5), and we evaluate experimentally the time of the database response depending on its size (Sec. 6).

4 Covariance Descriptor

In [9] the covariance of d-features has been proposed to characterize a region of interest. Now, we introduce *the geodesic distance definition* proposed by [3] as its computation is the main topic of our work. The distance between two covariance matrices is defined as

$$\rho(C_i, C_j) = \sqrt{\sum_{k=1}^{d} \ln^2 \lambda_k(C_i, C_j)} \tag{1}$$

where $\lambda_k(C_i, C_j)_{k=1\ldots d}$ are the generalized eigenvalues of C_i and C_j, determined by $\lambda_k C_i x_k - C_j x_k = 0,\ k = 1 \ldots d$ and $x_k \neq 0$ are the generalized eigenvectors. The complexity of finding generalized eigenvalues increases rapidly with the size of d. The efficient methods work for dimensions below 5. Often, a dimension of covariance matrix has to be decreased to conform requirements of a real-time systems at the expense of the accuracy. Hence, as the distance computation is the main bottleneck in covariance-based approaches we decided to take advantage of GPU to speed up the computation of the generalized eigenvalues.

5 GPU Implementation

5.1 GPU Architecture

GPU is a high performance computing unit in which the emphasis is put on the computing units instead of data caching and control flow units in contrast with CPU. The memory is not cached, so to obtain the maximum performance it is crucial that the programmer ensures the coalesced memory accesses. In our setup we use the NVIDIA Tesla 1070S with 4 graphic cards, each consiting of 30 SMs (Streaming Multiprocessors) with 8 scalar processors. The total number of available processors is therefore equal to 240x4 GPU. The threads work in a SIMD model (Single Instruction Multiple Data) and are grouped into blocks. All threads of a block reside on the same processor core. Threads within one block are split into basic scheduling units called warps, consisting of 32 threads. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. On GPU there are

different types of memory. The two most important are *shared memory* and *global memory*. Shared memory is allocated per block and shared among all threads of a block. It is organized in banks, and if the data is accessed so that each thread of a block accesses a different bank, shared memory is as fast as registers. Global memory is a slow off-chip memory, which can be accessed by both CPU and GPU. It is also used to synchronize data between threads in different blocks. In order to have a low latency, it should be accessed in a coalesced fashion.

5.2 Generalized Eigenvalues Problem

The most computationally heavy component of calculating the distance between the signatures is the calculation of generalized eigenvalues of positive definite symmetric matrices A and B (two corresponding covariance matrices from the compared signatures). It is defined by the equation $\mathbf{A}x = \lambda\mathbf{B}x$, where λ denotes a vector of eigenvalues and x is the eigenvector. This equation can be decomposed to the equation

$$(\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})\mathbf{L}^{T}x = \lambda(\mathbf{L}^{T}x) \tag{2}$$

where $\mathbf{L}$ is the upper triangular matrix calculated as $\mathbf{B} = \mathbf{L}\mathbf{L}^{T}$. We can notice that the decomposed Eq. (2) already corresponds to original eigenvalues problem. We solve the generalized eigenvalues problem in 4 steps:

1. Calculate the Cholesky Decomposition to solve the equation $\mathbf{B} = \mathbf{L}\mathbf{L}^{T}$
2. Calculate twice the Forward Substitution to solve $\mathbf{C} = (\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})\mathbf{L}^{T}x$
3. Tridiagonalize the symmetric matrix $\mathbf{C}$ to prepare for solving the eigenvalues problem
4. Use the Bisection Algorithm to find the eigenvalues of a symmetric tridiagonal matrix $\mathbf{C}$

Let us note that there are many methods to calculate the eigenvalues of a symmetric matrix. On the CPU we use a Jacobi algorithm, which does not need to perform the tridiagonalization first, however this algorithm is not easy to parallelize. For the GPU implementation we first tridiagonalize the matrix and then we use the bisection algorithm, which can be parallelized much more efficiently.

5.3 Porting the Algorithm to the GPU Architecture

Two sources of parallelism can be used in the algorithm. The first one is obvious – a comparison of two signatures involves comparing n pairs of covariance matrices, which can be naturally processed in parallel. The second one involves the parallelism extracted from each step of the equation performed on a given pair of covariance matrices. We will now briefly describe how we use the parallelism of each of the procedures to efficiently port it to the GPU architecture.
In **Cholesky Decomposition of n matrices**, the formula to calculate each element of the $\mathbf{L}$ upper triangular matrix is given as

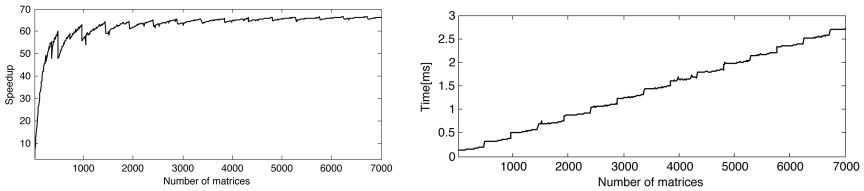
$$L_{j,j} = \sqrt{B_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^{2}}, \quad L_{i,j} = \frac{1}{L_{j,j}}(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}) \quad for \;\; i > j \tag{3}$$

The matrix B is loaded from global memory to shared memory in a coalesced way. After the calculations, the matrix $\mathbf{L}$ is loaded back to global memory, also assuring coalescence. For the calculations we use the Cholesky-Crout algorithm, which starts from the upper left corner of the matrix $\mathbf{L}$ and proceeds to calculate the matrix column by column, as processing data in column order ensures the memory coalescence. To calculate an element of a column, only the elements from the columns to the left are needed. So, all the elements in one column can be processed in parallel. For our matrices of size 11, we can therefore use 11 threads to calculate each column in parallel, using 11 iterations to calculate the whole matrix. The natural decomposition of a problem would be to assign one matrix to a block. However, 11 threads is much less than the size of a warp, which is the smallest scheduling unit. Assigning two matrices to a block enables to process two matrices without the increase of processing time, as 22 threads still form only one warp scheduled in one operation. More matrices however will lead to the creation of more warps, which in our specific situation (calculation of each matrix is independent) makes it similar to the creation of more blocks – it does not matter if in the next step another warp or another block is scheduled. So in our implementation there are $n/2$ blocks, each block consists of 22 threads and calculates two matrices. Our preliminary experiments showed that indeed, inreasing size of a block from 11 to 22 decreased the time of calculating n matrices twice, while futher increase of the size of a block (up to 110 with 10 matrices assigned) did not bring further improvement.

Forward Substitution solves the equation $\mathbf{Lx} = \mathbf{A}$. Two forward substitutions solve the equation $\mathbf{C} = (\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T})\mathbf{L}^{T}\mathbf{X}$. We process them together to avoid copying the intermediate data to the global memory. An element $x_{m,k}$ of a matrix $\mathbf{x}$ is calculated as $x_{m,k} = \frac{A_m - \sum_{i=1}^{m-1} L_{m,i} x_{i,k}}{L_{m,m}}$. All the columns of $\mathbf{x}$ are calculated independently. To calculate an element in the column, one could calculate the sum in the equation using map-reduce model, however for small matrices of size 11 it does not save much calculations so we calculate them sequentially. As a result, for one matrix we need 11 threads, and following our observations from Cholesky Distance procedure, we assign 22 threads to one block.

Tridiagonalization is the part of the generalized eigenvalues algorithm which has the lowest level of paralellism. To tridiagonalize the symmetric matrix we use the Householder transformation [4]. In this algorithm, $n - 2$ iterations need to be performed sequentially; in each iteration the appropriate elements in the *k-th* row and column are zeroed. In one iteration some of the computations, such as matrix multiplication and vector-vector multiplications can be parallelised. To do this, we use $n \times n$ threads for each matrix. We also tested the version in which there are only 11 threads, each calculating one column of the multiplied matrices, but it was less efficient. As 121 threads is more than a size of a warp, we can assign one matrix per block without loosing the efficiency.

Bisection Algorithm is used to calculate the eigenvalues of a symmetric tridiagonal matrix $\mathbf{C}$. A detailed description of the bisection algorithm can be found *e.g.* in [2]. This algorithm finds all the eigenvalues of a matrix with a given approximation. The core function of this algorithm is the `Count()` procedure



(a) Speedup - CPU vs GPU (b) Time (ms) of Bisection algorithm

Fig. 2. Speedup for finding generalized eigenvalues and time of Bisection algorithm.

returning the number of eigenvalues present in a given interval. The algorithm starts with the initial interval constructed using Gerschgorin's theorem. Then, it is divided in two and `Count()` procedure returns a number of eigenvalues in each subset. If it uquals zero, the node is abandoned, otherwise it is further subdivided into two subsets, until the size of a subset is not bigger than the assumed approximation. For our purposes it is enough to use the approximation equal to 10e-6. The main source of parallelisation comes from the fact that the `Count()` function in each node can be calculated independently. As only 11 eigenvalues can be found, on each level of the binary tree there will be only up to 11 nodes containing at least one eigenvalue. We therefore use 11 threads to calculate one matrix. There are also some other sources of parallelism in the calculation of `Count()` function itself, but we will not present it here. Again, we assign 22 threads to a block, calculating 2 matrices. Gerschgorin's procedure cannot be paralellized, so we process it sequentially, but a parallelism is obtained by calculating all the procedures for different matrices in parallel.

6 Experimental results

In our experimental setup, we calculated the performance for matrices number ranging from 10 to 3000. As we described in Section 3, we assume that the database of signatures is stored directly on the GPU. In our time estimation, we do not take into account the time of the data transfer, because the reference signatures already reside in the device memory, and the time of transferring the query signature is negligible.

Below we present the speedup obtained with comparison to the optimal version on the CPU – that is, a version with Jacobi algorithm (implementation from LTI library), as on CPU this version is faster than calculation of tridiagonalization and then bisection algorithm. The results are presented in Fig. 2(a). One can easily note that the speedup grows with the number of matrices, and reaches its maximum (66) from about 1500 matrices (corresponding to about 50 signatures). Distributing the database of signatures equally between the GPU cards, and performing the calculations for a query signature on all the GPU cards in parallel would result in further speedup improvements. In these tests we used oneTesla node consisting of 4 GPUs. Table 1 presents the time of the component procedures for 5 numbers of matrices. The Cholesky and Forward Substitution times are the lowest, while the biggest impact on the total time comes from the

tridiagonalization procedure, as it has the lowest degree of parallelism. Nevertheless, for example for the size of 3000, the total time on the GPU is equal to 3.76ms, while the time on the CPU is equal to 240ms.

N	Cholesky	Forward.S	Tridiag.	Bisect.	Total.GPU	Total.CPU
200	0.037	0.035	0.140	0.147	0.359	16
400	0.049	0.071	0.272	0.187	0.579	32
600	0.082	0.078	0.389	0.325	0.874	48
800	0.093	0.112	0.522	0.352	1.08	64
1000	0.126	0.120	0.657	0.505	1.41	80

Table 1. Time[ms] of component procedures

In Fig. 2(a) one can notice the *"stairs"* (decrease of speedup) which occur in regular intervals every 480 matrices. This is a result of a similar effect observed in component functions. Let us analyse it on the time performance of the Bisection Algorithm (Fig. 2(b)). One can clearly see that the time rises suddenly every 480 matrices. This is correlated to the number of multiprocessors on the GPU cards. The architecture is the most efficiently used when all the processors (on Tesla S1070 – 240 processors) have some blocks assigned. As in our model each block calculates two matrices, the architecture is used most efficiently when k*480 matrices are processed. Otherwise some processors remain idle. The worst case is when k*480+1 matrices are processed – then, the time is almost equal to processing (k+1)*480 matrices, which results in sudden decrease of speedup in these points.

7 Conclusions

In this paper we demonstrate that it is possible to improve significantly the performance of example video surveillance procedures, in particular the distance operator for querying the database. We observe that in our approach the speedup grows with the number of matrices. Moreover, we can easily distribute demanding calculations of signatures on many GPU units and obtain very good scalability of the system. Although GPU-based approach of four routines of generalized eigenvalues problem have been already proposed, they are optimized for solving only one large matrix. In our approach we use many very small matrices, which can be efficiently stored in a shared memory onto many GPUs. This requires different memory alignments, memory access optimization and simplification of some sub-procedures, but as tested experimentally could yield much better performance.

References

1. S. Bak, E. Corvee, F. Bremond, and M. Thonnat. Person re-identification using spatial covariance regions of human body parts. In *AVSS*, 2010.
2. J. Demmel and M. Heath. Applied numerical linear algebra. In *Society for Industrial and Applied Mathematics*. SIAM, 1997.
3. W. Förstner and B. Moonen. A metric for covariance matrices. In *Quo vadis geodesia ...?, Festschrift for Erik W. Grafarend on the occasion of his 60th birthday, TR Dept. of Geodesy and Geoinformatics, Stuttgart University*, 1999.

4. A. S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM 5*, 1958.
5. C. Lessig. Eigenvalue computation with cuda. In *NVIDIA techreport*, 2007.
6. G. Poli, J. H. Saito, J. a. F. Mari, and M. R. Zorzan. Processing neocognitron of face recognition on high performance environment based on gpu with cuda architecture. In *SBAC-PAD*, pages 81–88. IEEE Computer Society, 2008.
7. V. Sriram. Design-space exploration of biologically-inspired visual object recognition algorithms using cpus, gpus, and fpgas. In *MRSC*, 2010.
8. D. Tosato, M. Farenzena, M. Spera, V. Murino, and M. Cristani. Multi-class classification on riemannian manifolds for video surveillance. In *ECCV*, 2010.
9. O. Tuzel, F. Porikli, and P. Meer. Region covariance: A fast descriptor for detection and classification. In *ECCV '06*, pages 589–600, May 2006.
10. P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR '01*, pages 511–518, 2001.