On Fingerprint Processing Involving Modern Massively Parallel Architectures

Stanislav Sartasov, SPSU, St. Petersburg, Russia.

Scientific advisor: prof. A.N. Terekhov.

# Introduction

Fingerprint recognition is still the most widely accepted mean for a person identification and authentication. Two recent trends can be observed in automated fingerprint identification systems (AFIS): a shift towards mobile security systems and a querying the very large fingerprint databases (of order 106-107 and higher). Consequently two separate challenges arose in the field of fingerprinting: how to robustly identify a person with limited computational resources and how to speed up the database processing? The scalability problem is a long known issue in biometrics. With the number of records increasing so does the response time of the biometric system, and for a subset of AFIS, the real-time AFIS, the time is one of the quality characteristics.

In a generic AFIS there are following processing steps:

1. Image acquisition on a scanner
2. Image transmission from a scanner to the processing unit
3. Image enhancement (IE), can be omitted
4. Feature extraction and template building
5. Matching the template against database or part of it
6. Decision making whether to grant or deny the access given the matching results

In order to improve the overall system performance one should improve the performance of every step, which is usually obtained through the introduction of parallelism. A note should be taken that steps 1 and 2 usually don’t require speed up due to the hardware implementation of those steps, and their addition to a total processing time tends to be relatively small. As most systems implement step 6 in a form of a rule-based module with a single rule, the parallel implementation in this case might also be redundant.

With the hyperthreading and multi-core processors being available for some time now the simple way to improve the system performance was implemented in commercially available systems. The template database is chunked equally by the number of physical or logical cores and each chunk is processed independently, not unlike map-reduce approach or multi-agent systems. This approach is also scaled to the computing clusters, yet the price of such solutions is high. There exists a cheaper alternative, as General Purpose Graphical Processing Unit (GPGPU) computations is now a proven way to enhance a system performance in HPC. Among the many GPGPU technologies (e.g. OpenCL, DirectCompute) NVIDIA CUDA is the most advanced and readily available.

In this paper we demonstrate our results of implementing various steps of AFIS processing using NVIDIA CUDA technology. The paper is organized as follows. In section 2 an overview of parallel fingerprinting algorithms research to date is given. A small overview of CUDA technology is provided in section 3. The discussion of various aspects in parallel fingerprint processing is given in sections 4 and 5 and a conclusion is made in section 6.

# Related works

Prabhakar and Kameswara in [1] propose a scheme of a coarse minutia-based matcher for filtering out stored fingerprint templates that are unlikely to match the query. The proposed scheme is a de-facto continuous classifier [2] with a percentage of matched minutiae serving as a point in a 1D space. Each feature of a query vector is loaded to a separate processor element (PE). Every feature of a stored vector is matched independently with every query feature. If at least one of the PE produces a positive result, total matched count is increased. However the algorithm is made in assumption that two fingerprints are already aligned in orientation and the core point is detected in both of them. Authors notice a convenience of Single Instruction Multiple Data (SIMD) architecture for implementing the algorithm.

A similar approach is proposed by Ratha and Jain in [3] and [4] for FPGA which are well known for their parallel capabilities. Additional steps are done to prevent the same feature from being matched more than once. Matching speed of 1.1·105 matches per second is reported. This algorithm uses the Hough transform for fingerprints alignment, although it is only stated that parallel implementation on the chosen FPGA is available. SIMD processor efficiency for the task is once again noted. Authors also make a statement about two levels of parallelism in matching algorithm: macro-level for inter-template matching and micro-level for intra-template matching.

Gowrishankar in [5] describes a scheme of a feature extraction algorithm for a binary skeletonized image based on the neighbor colored pixels count and proposes the usage of massively parallel architecture of a computing device due to the information being extracted from a small neighborhood around each pixel.

Bai et al. in [6] investigate the capabilities of a NVIDIA graphic boards supporting CUDA for the parallel implementation of a well-known FingerCode algorithm [7]. Authors use a slightly modified version of a FingerCode due to technological limitations of a chosen matching scheme, but report a performance of 2.03 million of matches per second. Our previous studies also indicate a great potential in FingerCode’s parallel implementation, especially after spatial characteristics optimization (band number, band width, innermost radius, fnumber of filters, and number of sectors).

Cappelli, Ferrara and Maltoni report of a new local minutiae template based on a metric calculated for a minutiae around a sample one in a 3D lattice points inside a cylinder structure and its matching technique in [8] with a consequent consolidation step. The optimized version of such a template can be reduced to be a bit vector which can be efficiently matched with each other. Different approaches for consolidation step are then proposed. In a later press-release [9] authors state the increase of performance from 2.5 million matches per second on a CPU to 4 million on a GPU.

# NVIDIA CUDA overview

The NVIDIA CUDA technology is used in a GeForce family of graphic boards [10]. Its programming model is shown on fig.1.



Fig.1. CUDA programming model

According to this model all computations are divided to CPU-based and GPU-based with CPU aliased as host and GPU as device. The host sends a program kernel – a compiled GPU shader – to the device. Thread configuration occurs on a device prior to the kernel execution. It is assumed a large number of threads will be created during execution and those threads are united as blocks. All threads of a single block are executed on a single device multiprocessor, and the threads in the block are physically united in the warps where instructions are executed simultaneously. The set of blocks is organized as a grid. Currently the number of threads in a warp is 32, block and grid can’t index threads more than in 3 dimensions simultaneously with a limitation of a 1024 threads maximum per a single block. Up to 16 blocks with a total number of threads less than or equal to 2048 can be executed simultaneously on a single multiprocessor. There are some notable limitations:

1. The block size should be chosen with a register memory consumption per thread in mind as its amount is limited in a multiprocessor.
2. Global memory I/O is recommended to be optimized in such a way that, simply put, threads with a neighboring indexes are accessing neighboring 4-bytes words. This will allow to access the data in batches instead of single word access for each thread.
3. It’s forbidden to use recursive functions.
4. Branching to different paths is executed slower than branching to a single path by all threads.

Therefore implementing some algorithms in CUDA might be challenging.

# Image enhancement and minutiae extraction

Image enhancement is a crucial step for real-time AFIS, as the impression quality is generally lower during system usage than during enrollment. Different methods were proposed to improve the quality of the acquired fingerprint, a survey can be found in [11]. It was shown that contextual enhancement is better than context-less enhancement, and Fronthaler, Kollreider and Bigun in [12] demonstrate the superiority of the pixel-wise enhancement versus block enhancement, e.g. it doesn’t produce artifacts at blocks’ borders.

The enhancement method involving Laplacian-like pyramid image decomposition and Linear Symmetry (LS) estimation is proposed by Fronthaler, Kollreider and Bigun in [12]. The main steps of this algorithm include image scaling, matrix subtractions and additions, convolutions of the source image with 2 kernels based on a Gaussian partial derivative, representing the result as a field of complex number and squaring it, and a final scalar multiplication with another kernel. This approach is considered to be efficient if implemented on massively parallel architectures due to the following reasons:

* It may be viewed as a sequence of transformations where each transformation uses only the previous results to obtain the next one – every transformation is non-iterative.
* Every transformation operates on a 2D matrices in such a way that each element’s computation is independent from the computation of other elements other than by the source data, which is immutable.
* Furthermore, most of the transformations include 2D convolutions, either ordinary or complex, that can be efficiently parallelized [10], and most kernels are Gaussians or its derivatives, so their separability property may as well be used.

In our experiments we’ve implemented convolutions in a straightforward way without using separability due to the relatively small size of the kernels and the coalescing memory access pattern while a parallel processing was done on a matrix element level, so our results can further be improved. The size of Gaussian kernels was determined using a “3 sigma” rule by a formula . It could also be observed that amount of operations per pixel is constant, so the processing time is a linear function from the amount of pixels in the image or, in other words, image area. Parameters used a summarized in table I, and processing times are shown in table II.

TABLE I

Parameters for image enhancement routine

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| σ1 | 0.6 | f0 | 1.7 | f2,f3 | 1.3 | τ2 | 0.3 | σdirections | 2.0 |
| σ2 | 3.2 | f1 | 1.21 | τ1 | 0.1 | Annulus radii | Inner 4, outer 6 | Number of directions | 20 |

TABLE II

Image enhancement processing time on different graphic boards

|  |  |  |
| --- | --- | --- |
| Database (resolution) | NVIDIA GeForce 560, ms | NVIDIA GeForce 570 |
| FVC2000 DB2 (256 x 364) | 24.5 | 16.7 |
| FVC2004 DB1 (640 x 480) | 65.6 | 46.4 |

Minutiae extraction, also described in [12] is performed afterwards. LS and Parabolic Symmetry (PS) are estimated in order to produce the final measure . The area of each local maximum is checked for a consistent high , and the list of maximums is then sorted by a descending as a certainty measure resulting in a minutiae list. For the reasons described below we’re storing only top 32 minutiae as a template. The same considerations as for image enhancement are valid in this case as well. Convolutions and local maximum estimations are implemented in parallel while the minutiae list generation and sorting is done sequentially. Generally, those operations are lightweight compared to LS and PS estimation. Table III contains the parameters of the process being used. Those parameters should be chosen carefully in order to minimize the amount of spurious minutiae. However we’ve observed a consistency in their detection among different impressions of the same finger. The performance results are shown in Table IV.

TABLE III

Parameters for minutiae extraction routine

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| σ1 | 0.6 | ΤPS | 0.5 | Neighborhood radius for max | 9 |
| σ2 | 3.2 | ΤLS | 0.9 | Ring radii | Inner 4, outer 6 |

TABLE IV

Minutiae extraction time on different graphic boards

|  |  |  |
| --- | --- | --- |
| Database (resolution) | NVIDIA GeForce 560, ms | NVIDIA GeForce 570, ms |
| FVC2000 DB2 (256 x 364) | 24.7 | 17.8 |
| FVC2004 DB1 (640 x 480) | 62.8 | 48.4 |

# Minutia-based fingerprint matching

Minutiae-based fingerprint matching is a cornerstone of a modern AFIS. As noted in [3], database matching can be done in parallel on 2 levels: micro-level where different minutiae are matched in parallel with a given one and macro-level where different fingerprints are matched in parallel. However we need to point out that methods proposed in [1] and [3] are operating on an already aligned set of fingerprints, and Hough transform-based alignment is noted as a plausible for a selected FPGA without quality estimations. Fingerprint alignment is a challenging task that can be formulated in a following way: given two fingerprint templates and what are the best parameters for translation and rotation to maximize the matching score? A detailed survey of the developed methods can be found in [11].

In our approach the fingerprint database is preloaded to the GPU global RAM. This step adds up to the system security, as GPU RAM is separated from CPU RAM and is available only through driver operations. We’re assigning one CUDA block to each of the database fingerprints so it scales naturally on different and more advanced graphic cards. Each block consists of 32x32 threads, and currently it is the technological maximum for CUDA. Thus we’re limiting the template size to 32 minutiae. Both query template and database template are loaded to the cache memory of a multiprocessor. After that for every minutia and in these templates a template copy ( and accordingly) is made and translation is done to make the minutia and coordinates equal to (0; 0), so indices n and k can be used to indicate translation parameters. The thread then uses templates and to calculate the possible rotational alignment.

In order to do this we’re assuming that fingerprints are already coarsely rotationally aligned by a scanner, that is, its acquisition window or similar sensor is designed in a way that makes it difficult or inconvenient to acquire a rotated fingerprint, which is usually the case for modern scanners. For every minutiae pair in the difference between Euclidean distances and is computed. After that the slope angles of and and their difference are calculated. If distance difference and angle difference are below certain thresholds (in our experiments 3 and accordingly), is considered to be a plausible alignment parameters and is stored in a cache memory.

The second step is matching two fingerprints given a set of plausible alignment parameters. The set is distributed equally among 1024 working threads. Matching procedure is similar to the one described by Wegstein in [13], but it uses Euclidean distance differences instead of coordinate differences (6 in our experiments). Due to the constant amount of the minutiae in the templates the thread output is a number of the matched minutiae. Its maximum is searched among all thread outputs (and consequently among all the alignment parameters) and is considered to be the final similarity score.

FVC 2000 DB2 was used in conjunction with the image enhancement and minutiae extraction techniques discussed earlier to obtain the quality estimations for 3960 genuine and 386760 impostor matches. These results are displayed in Fig. 3. We report an average time of 0.41 ms for a single fingerprint match on GeForce 570 and 0.78 ms for GeForce 560. The performance difference between two boards is explained by the difference in the number of multiprocessors (15 and 7 accordingly).

Fig.3 ROC of the method discussed

# Discussion and conclusion

Our results demonstrate that convolution-based and/or scalar multiplication-based pixel-wise image enhancement and direct grayscale minutiae extraction can be implemented efficiently on a modern GPUs. Despite our focus on a direct grayscale minutiae extraction, GPGPU usage is also a perspective approach for a skeleton-based techniques, as there are GPU-accelerated ways for producing image skeletons [14], and minutia extraction from them (e.g. [5]) is highly parallelizable.

As for the minutiae matching, the matching speed is much lower than one could expect from a massively parallel architecture implementation. First of all, there’s definitely a place for a further optimizations in the implemented algorithm. Secondly, the alignment step of our approach is essentially a “brute force” search which can produce several thousand of possible alignments. Despite the fact this algorithm and matching procedure are highly parallelizable by data, internally they’re implemented in a sequential manner due to the lack of a single block resources. Thus with a finer alignment algorithm resulting in a lesser amount of possible alignment parameters or with a later version of CUDA technology a noticeable speedup is expected. With a pre-alignment step being done prior to the matching routine the implementation of a micro-level parallelism for minutiae matching on CUDA is a straightforward decision.

Another concern is the recognition quality. 32 minutiae is, on average, only 50 to 75% of its total amount in a given fingerprint. While some fingerprints in a database contain unrecoverable regions, we consider the spurious minutiae and operating on a subset of a template to be the main reasons of the high FRR.

The code for the algorithms discussed is available at <http://code.google.com/p/cuda-fingerprinting/> and requires CUDA 2.0 compatible graphic board to run.

# References

1. Prabhakar, R.V.S.N. and Rao, C.V.K. A parallel algorithm for fingerprint matching. in *Fourth IEEE Region 10 International Conference TENCON* (1989)*,* 373 – 376.
2. Lumini, A., Maio, D. and Maltoni, D. Continuous versus exclusive classification for fingerprint retrieval.  *Pattern Recognition Letters*, *18* (10),  1027 -1034.
3. Ratha N.K., Jain, A.K., Rover, D.T., Cantoni, V., Lombardi, L., Mosconi, M., Savini, M. and Setti, A. An FPGA-based point pattern matching processor with application to fingerprint matching. in Proceedings of the Computer Architectures for Machine Perception (CAMP '95), IEEE Computer Society, 394-.
4. Ratha, N.K., Karu, K., Chen, S. and Jain, A.K. A Real-Time Matching System for Large Fingerprint Databases. IEEE Transactions on Pattern Analysis and Machine Intelligence, *18* (8), 799-813.
5. Gowrishankar, T.R. Fingerprint Identification on a Massively Parallel Architecture. in *Proceedings of the 2nd Symposium on Frontiers of Massively Parallel Computation* (1989), 331–334.
6. Bai, S., Marques, J.P., McMahon, M.T., Barry, S.H. GPU-Accelerated Fingerprint Matching, 2011. Retrieved May 16, 2013, from GPU Technology Conference 2012: <http://on-demand.gputechconf.com/gtc/2012/posters/P0373_11-2610_GTC2011_POSTER-MITRE_GPU-Accelerated_Fingerprint_Matching_v1.pdf>
7. Jain, A.K., Prabhakar, S., Hong, L. and Pankanti, S. FingerCode: A Filterbank for Fingerprint Representation and Matching. in *Conference on Computer Vision and Pattern Recognition* (CVPR'99), IEEE Computer Society, vol. 2, 2187-.
8. Cappelli, R., Ferrara, M. and Maltoni D.Minutia cylinder-code: A new representation and matching technique for fingerprint recognition IEEE Transactions on Pattern Analysis and Machine Intelligence, 32 (12),  2128 -2141 .
9. Minutia Cylinder-Code, 2011. Retrieved May 16, 2013, from Biometric System Laboratory, University of Bologna: <http://biolab.csr.unibo.it/research.asp?organize=Activities&select=&selObj=81&pathSubj=111||8||81&Req=&>
10. Sanders, J., Kandrot, E. CUDA by Example: An Introduction to General-purpose GPU Programming, Addison Wesley Professional, 2011.
11. Maltoni, D., Maio, D., Jain, A.K. and Prabhakar, S. Handbook of Fingerprint Recognition, 2nd ed. Springer Professional Computing, 2009.
12. Fronthaler, H., Kollreider, K. and Bigun, J. Local features for enhancement and minutiae extraction in fingerprints. *IEEE Transactions on Image Processing, 17* (3), 354–363.
13. Wegstein, J.H. The M40 Fingerprint Matcher. U.S. Government Publication, National Bureau of Standards, Technical Note 878, U.S Government Printing Office, Washington, DC, 1972.
14. Real-time 2D skeletonization using CUDA, 2011. Retrieved May 16, 2013, from University of Groningen: <http://www.cs.rug.nl/svcg/Shapes/CUDASkel>