A First Step to Accelerating Fingerprint Matching based on Deformable Minutiae Clustering [⋆]

A.J. Sanchez[1][0000−0001−6743−3570], L.F. Romero[1][0000−0003−2959−2030], S. Tabik[2][0000−0003−4093−5356], M.A. Medina-Pérez[3][0000−0003−4511−2252], and F. Herrera[2][0000−0002−7283−312X]

[1] Dpt. of Computer Architecture, University of Malaga, Spain
ajsanchez@ac.uma.es felipe@uma.es

[2] Dpt. Computer Science and Artificial Intelligence, University of Granada, Spain
siham@ugr.es herrera@decsai.ugr.es

[3] Tecnologico de Monterrey, Campus Estado de México, Carretera al Lago de Guadalupe Km. 3.5, Atizapán de Zaragoza, Estado de México, 52926, México
migue@itesm.mx

Abstract. Fingerprint recognition is one of the most used biometric methods for authentication. The identification of a query fingerprint requires matching its minutiae against every minutiae of all the fingerprints of the database. The state-of-the-art matching algorithms are costly, from a computational point of view, and inefficient on large datasets. In this work, we include faster methods to accelerating DMC (the most accurate fingerprint matching algorithm based only on minutiae). In particular, we translate into C++ the functions of the algorithm which represent the most costly tasks of the code; we create a library with the new code and we link the library to the original C# code using a CLR Class Library project by means of a C++/CLI Wrapper. Our solution re-implements critical functions, e.g., the bit population count including a fast C++ PopCount library and the use of the squared Euclidean distance for calculating the minutiae neighborhood. The experimental results show a significant reduction of the execution time in the optimized functions of the matching algorithm. Finally, a novel approach to improve the matching algorithm, considering cache memory blocking and parallel data processing, is presented as future work.

Keywords: Fingerprint Recognition · Cache Optimization · Language Interoperability.

[⋆] This work was partly supported by the National Council of Science and Technology of Mexico (CONACYT) under the grant PN-720 and by the TIN2016-80920R project (Spanish Ministry of Science and Technology) and the University of Malaga (Campus de Excelencia Internacional Andalucia Tech).

1 Introduction

The task of fingerprint recognition has been widely studied in the recent years. The reason why this human characteristic is very popular to identify people lies in the fact that fingerprints are unique and do not change over time. That is, there are not two people with the same fingerprint, making them a perfect recognition feature. The major characteristics of fingerprints, based on ridge features, are known as minutiae and represent ridge endings and bifurcations as standard. Thereby, in almost all most accurate matching algorithms, the comparison method relies on the minutiae details computation carried out in the CPU [1], while other approaches perform the calculations by using GPUs [2, 3].

Fingerprint matching algorithms are usually implemented using high-level programming languages such as C#, Java or Python [4]. This choice is made due to the large amount of programming tools that are available to make the programming work easier for the researcher. Nevertheless, the mayor drawback from this languages and similar ones consists in the lower execution speed of the code, in contrast with lower-level programming languages. For example, in C# programming language, when the file is compiled, the code is firstly translated into an intermediate managed language named *Intermediate Language* (IL) and then into machine code, during runtime, by the *Common Language Runtime* (CLR) tool from the .NET framework. This is why this type of code is also known as managed language, causing execution overhead.

This paper focuses on currently one of the most accurate fingerprint matching algorithms, based on minutiae processing, patented by Medina-Pérez et al. [5, 6] and will be referred as "DMC" in the rest of the paper. This matching method was initially developed using C# for its implementation and uses the following three algorithms:

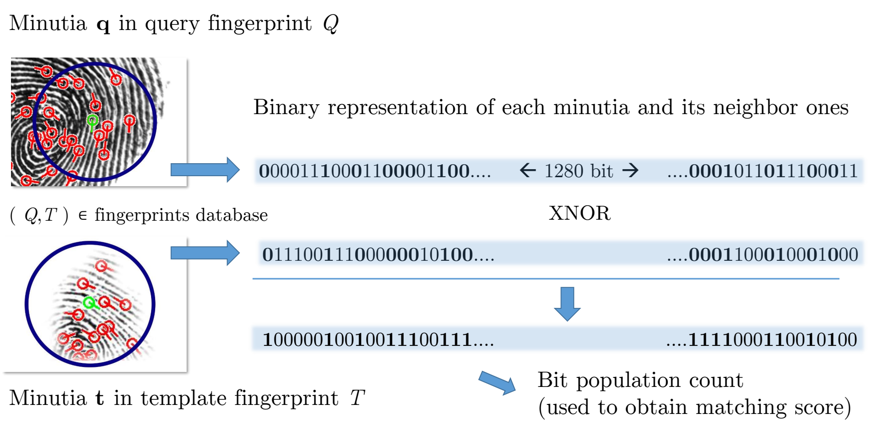
1. The *Minutia Cylinder-Code* method [7] which is based on 3D data structures, called Binary Cylinder Codes, built from minutiae positions and angles from merging local structures [8]. DMC uses this minutia representation to perform the matching processing.
2. The *Minutiae Discrimination* method [9] that calculates a quality value for each minutia based on the minutiae direction consistency inside its minutiae neighborhood.
3. The *Deformable Minutiae Clustering* method [5, 10] which is used to avoid data loss due to fingerprint deformation. Similarity results are obtained by adding the weights of each matched minutiae pair from merging the clusters that are similar. In addition, the weight of each minutiae pair (q, p) is calculated by evaluating the minutiae from the matched cluster in which (q, p) is the centre. Statistical outcomes are obtained based on the performance evaluation proposed by Cappelli et al. [11].

We propose two techniques to accelerate the DMC matching algorithm in CPU. First, the profile of the matching algorithm is obtained to determine the highest computational cost functions. Afterwards, we apply novel optimizations by including a CLR Class Library project to work as a linker between the C# code, and a C++ Library where the DMC matching algorithm is actually executed. Focusing on the improvements, we optimize the bit population count operation performed within *Minutia Cylinder-Code* algorithm and the minutiae direction consistency algorithm from the *Minutiae Discrimination* method. Moreover, a future parallel approach to optimize the DMC algorithm is also proposed taking into account matching decomposition [12].

This paper is organized as follows. Section 2 introduces the structure of the DMC matching algorithm along with its profiling outcomes. Section 3 describes the optimization techniques performed on the original DMC matching algorithm. Section 4 presents the experimental results. Section 5 proposes a future approach to enhance matching algorithm execution. Finally, Section 6 discusses the result of this paper and future work.

2 DMC structure and profiling

The DMC fingerprint matching algorithm was developed from merging three independent matching methods whose procedures can be summarized as follows:



Minutia **q** in query fingerprint Q
Binary representation of each minutia and its neighbor ones
0000111000110000110 0.... ← 1280 bit → 0001011011100011
$(Q,T) \in$ fingerprints database
XNOR
01110011100000010100.... 0001100010001000
10000010010011100111.... 1111000110010100
Minutia **t** in template fingerprint T
Bit population count
(used to obtain matching score)

Fig. 1. Example of the bit population count operation used in the first step of the matching algorithm. The set bits, resulted from the XNOR operation performed on two minutiae descriptors, are counted and used to calculate the minutiae matching score.

1. Let Q and T be the minutiae sets of query and template fingerprints from a particular database. Each minutia $\mathbf{q} \in Q$ is compared to all minutiae $\mathbf{t} \in T$ based on their minutia descriptor. In our case study, the minutia descriptor is an array of 1280 bits which stores the positions and angles of all minutiae

inside a circumference of radius r, centered on the selected minutia. Thus, two similar minutiae should have a large amount of bits with same position and value inside both minutia descriptors, so that if the XNOR operation is done between them as it is shown in Figure 1, it will result in a new vector with one bits in the concurring positions. Therefore, a function cited as *PopCount* is necessary to calculate the number of set bits, which is then used to obtain a similarity value between minutiae. Each matching minutiae pairs $(\mathbf{q}, \mathbf{t})$ is then included inside a set of local matching minutiae pairs (A) and sorted by its similarity value.

2. For each minutia $\mathbf{q} \in Q$, a quality value is computed relying on the minutiae direction consistency of all minutiae inside its neighborhood, and prior to achieve this goal, it is necessary to compute the three minutiae closest to the selected one by means of a function named as *UpdateNearest*. Moreover, the same procedure is performed on each minutia $\mathbf{t} \in T$. Finally, two sets containing all minutiae from each fingerprint and their particular quality value are obtained.
3. From the sorted A list, clusters of matching minutiae pairs are determined for each pair $(\mathbf{q}, \mathbf{t})$, along with a weight value which is obtained taking into account the matched minutiae pairs inside the matched cluster in which $(\mathbf{q}, \mathbf{t})$ is located in the centre. Then, all clusters inside both fingerprints are merged to find global matching minutiae pairs. Afterwards, a new search is performed, focusing on each minutia $\mathbf{q} \in Q$ and $\mathbf{t} \in T$, using the Thin Plate Spline method to find new matching minutiae pairs [13], which could have been discarded in the previous algorithms due to deformations.

Table 1. Call tree performance profiling of the original DMC where the functions are listed by the collected CPU exclusives samples in descending order. BBC denotes Binary Cylinder-Code and DMC-BCC.Match the main function in which the DMC matching algorithm is executed. The functions in bold are located within DMC whereas the other functions are native tools from .NET Framework.

Function Name	Inclusive Samples	Exclusive Samples
clr.dll	2094	2094
mscorlib.ni.dll	1685	1187
Map	1242	358
UpdateNearest	315	231
DMC-BCC.Match	6193	209
MtiaPairComparer.Compare	381	205
BinaryCylinderCode.Match	309	189
DMC-BCC.MatchMinutiae	281	107
BCC.PopCount	95	95

Once the matching algorithm process has been studied, the next target consists in determining which functions are the most computationally expensive inside the four steps already mentioned. Thus, a thorough analysis of the original DMC implementation needs to be made, so the Performance Profiler Tools from Visual

Studio 2017 is used to obtain two types of outcomes: inclusive and exclusive samples from the CPU. In our case, the exclusive samples are used to identify the functions with the highest computing lines of code because it only takes account of the samples obtained inside the function and not the ones that are called within it, representing the bottlenecks of the matching algorithm. Table 1 shows the functions with the highest computing load based on the CPU exclusives samples. Highlighted function names are functions which are situated inside the matching algorithm, while the rest of functions are tools used by the .NET Framework. From this list, our research will be focused on the *UpdateNearest* and *PopCount* functions due to the high number of exclusive samples that they have obtained from the profiling in contrast with the few amount of lines of code that they contains. In the next section, novel optimizations performed over this two functions will be presented.

3 Software optimizations

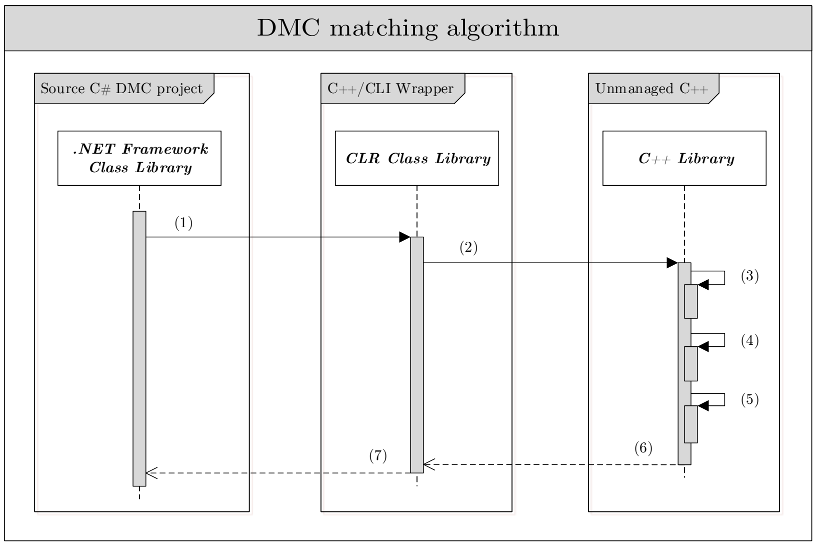


Fig. 2. Schematic structure of the modified DMC matching algorithm which includes a CLR Class Library and an unmanaged C++ Library where the fingerprint matching algorithm is performed.

Regarding the DMC algorithm, it was implemented using C# along .NET Framework 4 and hence, the first step constitutes the translation of the matching algorithm, which includes the functions previously determined from the profiling, to an unmanaged and lower-level programming language preserving the structure of the original solution. Therefore, C++ was chosen to replace C# as the programming language owing to its better performance and the fact that it is also an Object-Oriented Programming (OOP) language which results in less compatibility issues. Then, a connection between the original C# project and the new C++ library needed to be established in order to share the necessary data between them and hence, two .NET framework tools of interoperability were tested: Platform Invoke using extern "C" methods [14] and a C++/CLI Wrapper [15]. Both strategies speed up the part of the code which is translated into C++, however, the second method enables objects creation, name-spaces usage, and passing values, and it also enables the assembly along with managed codes.

Figure 2 shows the modified DMC algorithm using the C++/CLI Wrapper, where a CLR Class Library project and a C++ Library are included. The first one is written using C++/CLI code which provides several .NET Framework tools for working with managed and unmanaged memory. Thus, all the fingerprints data loaded inside the C# managed code can be passed as arguments and stored within the CLR Class Library with few transformations (1). Then, this data can be easily exported to the unmanaged C++ Library (2) simply by calling the functions defined inside the last one. Afterwards, the C++ library executes the optimized matching algorithm, which includes the three main processes: *Minutia Cylinder-Code Creation* (3), *Minutia Quality Computation* (4) and *Clustering Computation and Merging* (5). Finally, all the results achieved are returned to the CLR Class Library project first (6) and then, to the C# project (7).

Regarding the first step of the matching algorithm, the bit population count procedure is needed to compute the similarity value between minutiae. This bit operation is also known as pop count and define the action of calculating the number of set bits from a certain word. This technique is implemented in the *PopCount* function by using a SWAR (SIMD Within A Register) method. This technique deals with counting bits of duos to afterwards, add the duo-counts to a four-bit aggregation and then bytes inside a 64-bit register, to finally sum all bytes together. Although this algorithm is known for a good performance, we wanted to implement the best approach for the bit population count based on the computer characteristics where the DMC fingerprint matching algorithm was running. Therefore, the fast C++ bit population count library developed by Wojciech Mula et al. [16] was implemented, which chooses the best algorithm depending on the computer features and the size of the bit array.

On the other hand, focusing on the second step of the matching algorithm, the minutiae neighborhood of each minutia is calculated within all minutiae that set the fingerprint, based on the Euclidean distance. The square root results from this operation are obtained by using two methods: accessing to a pre-established table with the 1024 former square root results or, if the value is not contained in this table, performing the square root operation using the C# Math Class. To avoid the overhead of this two methods in the translated matching algorithm, the square Euclidean distance is used instead.

4 Experimental results

The experiments were executed using an Intel i5-8600K processor running at 3.60 GHz. The machine has six L1 and L2 cache memory of 32 and 256 kBytes, respectively, and a single L3 cache of 9 MBytes. Also, it has a 8 GB of RAM (DDR4 with dual-channel). Time results were obtained by using the Stopwatch Class from .NET Framework in C# and the C++11 Chrono Library in the C++ library. The FVC2004_DB1A database, which contains 800 hundred latent fingerprints with perturbations deliberately introduced, has been used to perform the experiments. Regarding the optimization of the bit population count, a fast C++ library (*libpopcnt*) was used inside the *PopCount* function in spite of the original SWAR algorithm from C#. The results achieved from the execution of both methods are shown in Table 2. The C# method runtime was reduced from 90.176 to 0.440 microseconds in C++ per iteration of counting the set bits from an array which takes about 160 bytes of memory (20 values of 8 bytes). Thus, since the array size is less than 512 bytes, an unrolled pop count algorithm is selected by the library to perform the operation. On the other hand, an AVX2 algorithm would be executed if the array size were over 512 bytes, however, this situation does not take place in the original DMC algorithm but will be addressed in a future work to improve the accuracy without increasing the execution time. In speed-up terms, the function acceleration achieved in C++ is up to 204.8 times faster than the original C# one.

Table 2. Time, in microseconds, and speed-up results per bit population count iteration inside the *PopCount* function, where the performances of the SWAR algorithm from C# and the fast C++ *libpopcnt* library are compared.

	C# SWAR algorithm	Fast C++ library
Time per iteration	90.176 μs	0.440 μs
Speed-up	1	204.8

With respect to the *UpdateNearest* function, the use of the Euclidean distance was replaced by the squared Euclidean distance (SED), eliminating the square root operation and improving time results per iteration as shown in the Table 3. This function was accelerated up to 2.83 times using the squared Euclidean distance, in contrast with the 2.33 achieved with the standard Euclidean distance.

Table 3. Time, in microseconds, and speed-up results per iteration of the *UpdateNearest* function using three approaches: original C# method, this method translated into C++ and the final method implementing the squared Euclidean distance (SED) in C++.

	C#	C++	C++ (SED)
Time per iteration	25.47 μs	10.93 μs	9.00 μs
Speed-up	1	2.33	2.83

5 Proposed improvements

Fingerprint matching algorithms are useful methods which commonly perform the comparison between one latent fingerprint and all the fingerprints stored in a particular database. This technique reduces the number of fingerprints to analyze to a reduced set of similar ones, however, the time necessary to perform this operation depends heavily on the size of the database. In addition, every fingerprint, either rolled, latent or plain, do not necessarily contain a fixed number of minutiae, owing to the high complexity of the fingerprint sample taking, which involves several aspect such as pressure variations, finger area (partial or total sample), large nonlinear distortion, among others [17]. Thus, if the query fingerprint and the selected template one contain a large amount of minutiae each, the total data used to perform the comparison will take more memory space than the available inside the L1 cache memory, increasing the cache misses and the execution time thereby [18]. This situation is shown in Figure 3 where the average size of a fingerprints pair, obtained from executing the DMC matching algorithm on the FVC2004_DB1A fingerprints database, is close to the maximum L1 memory cache capacity.

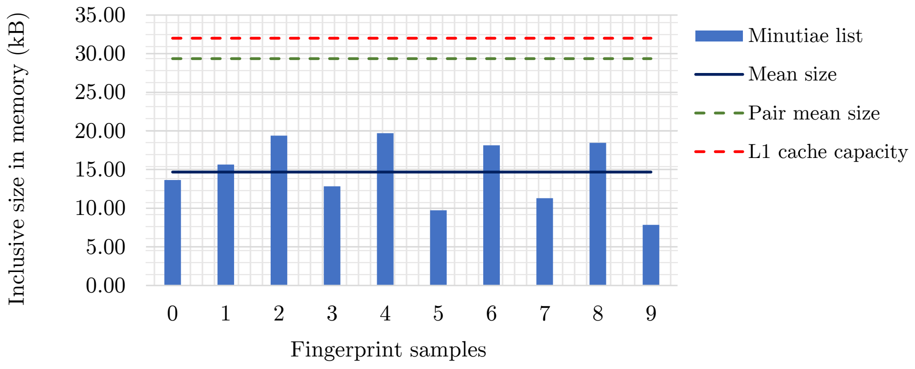


Fig. 3. Minutiae lists sizes results, in managed memory, from a sample of 10 fingerprints from the FVC2004_DB1A database. The average size and the mean pair size of the minutiae lists sample are presented along the L1 cache memory capacity.

In order to address this issue, the implementation of a parallel program structure for the DMC matching algorithm is proposed. A diagram of this structure is presented in Figure 4, where each core from the CPU will have a copy of the query fingerprint and will perform the matching algorithm between this one and a block of fingerprints from the database. This block will contain entire fingerprints or sections, depending on its size in memory, e.g., F5-S1 and F5-S2 from the previous figure represent a division of the fifth fingerprint from the database in two sets of minutiae to be processed in two different cores. This processing block size will be calculated so that all minutiae data would fit properly in the L1 memory cache when processing the first two steps of the matching algorithm. If a match occurs between minutiae, the similarity value will be stored and then computed to obtain a global similarity outcome. Hence, this proposal tries to solve two common problems of fingerprint matching algorithms: poor scalability for large fingerprints databases and low efficiency of cache memory usage.

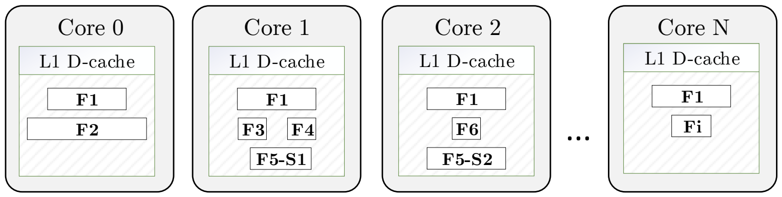


Fig. 4. Structure of the proposed parallel DMC matching algorithm where each thread has a copy of the query fingerprint ($F1$) to be compared with a block of selected fingerprints from the database whose data fit properly inside the L1 D-cache memory.

6 Conclusions

Two approaches to improve the speed of the fingerprint matching algorithm DMC proposed by Medina-Pérez et al. [5] has been presented in this work. The use of C++ as the programming language for executing the DMC matching algorithm has proved to be the critical step to reduce runtime, owing to the computational overhead that C# causes. To achieve this goal, a CLR Class Library was implemented inside the original DMC code to work as a linker between the original C# project and the unmanaged C++ file, enabling the pass of the fingerprints data in both directions. The unmanaged C++ was deployed into a static C++ Library (.lib) which contains the optimized matching algorithm written in this low-level programming language. Regarding more specific optimizations, the implementation of a fast C++ library for the *PopCount* function within the matching algorithm improves the acceleration of the set bits counting operation up to 204.8 times, in contrast to the SWAR C# method. Concerning the *UpdateNearest* function, the same algorithm as C# was implemented in the C++ library, obtaining an enhancement up to 2.33 times faster by translating this method into C++. Then, the use of the squared Euclidean distance inside the *UpdateNearest* function is addressed for calculating the minutiae neighborhood, proving to be up to 2.83 times faster than the no-squared approach in C#. To conclude, a new approach to enhancement the DMC matching algorithm performance has been proposed using parallel programming techniques where each thread compares the copied query fingerprint with blocks of template fingerprints which would fit properly in the L1 D-cache memory for the two first steps of the matching algorithm. It has been shown that, focusing on the DMC matching algorithm, the total data size in managed memory from two fingerprints is, in some cases, bigger than the L1 cache memory capacity. This situation could slow down the calculation performance owing to cache memory misses and the poor scalability for large databases, issues that could be addressed by using our novel suggestion.

References

1. Peralta, D., et al.: A survey on fingerprint minutiae-based local matching for verification and identification: Taxonomy and experimental evaluation. Information Sciences **315**, 67–87 (2015)
2. Lastra, M., Carabao, J., Gutierrez, P.D., Benitez, J.M., and Herrera, F.: Fast fingerprint identification using GPUs. Information Sciences **301**, 195–214 (2015)
3. Gutierrez, P.D., Lastra, M., Herrera, F., and Benitez, J.M.: A high performance fingerprint matching system for large databases based on GPU. IEEE Transactions on Information Forensics and Security **9**(1), 62–71 (2014)
4. Medina-Perez, M.A., et al.: Introducing an experimental framework in C# for fingerprint recognition. In: MCPR 2014, pp. 132–141. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07491-7_14
5. Medina-Perez, M. A., et al.: Latent fingerprint identification using deformable minutiae clustering. Neurocomputing **175**, 851–865 (2016)
6. Medina-Perez, M.A. et al.: "Sistema y método para la comparación de huellas dactilares y palmares basada en múltiples clústeres deformables de minucias coincidentes", No. ES2581593, Spain, 2018. (In Spanish)
7. 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 (2010)
8. Jiang, X., and Wei-Yun Y.: Fingerprint minutiae matching based on the local and global structures. Pattern recognition **2000**(2), 1038–1041 (2000)
9. Cao, K., Liu, E., Pang, L., Liang, J., and Tian, J.: Fingerprint matching by incorporating minutiae discriminability. In: International Joint Conference on Biometrics (IJCB) 2011, pp. 1–6. IEEE (2011) https://doi.org/10.1109/IJCB.2011.6117537
10. Medina-Perez, M.A., et al.: Improving fingerprint verification using minutiae triplets. Sensors **12**(3), 3418–3437 (2012)
11. Cappelli, R., et al.: Performance evaluation of fingerprint verification systems. IEEE transactions on pattern analysis and machine intelligence **28**(1), 3–18 (2006)
12. Peralta, D., Garcia, S., Benitez, J.M., and Herrera, F.: Minutiae-based fingerprint matching decomposition: methodology for big data frameworks. Information Sciences **408**, 198–212 (2017)
13. Bazen, A.M., and Sabih H.G.: Fingerprint matching by thin-plate spline modelling of elastic deformations. Pattern Recognition **36**(8), 1859-1867 (2003)
14. Using C++ Interop (Implicit PInvoke), https://msdn.microsoft.com/en-gb/library/2x8kf7zx.aspx. Last accessed 16 Feb 2018
15. .NET Programming with C++/CLI (Visual C++), https://msdn.microsoft.com/en-gb/library/68td296t. Last accessed 22 May 2018
16. Mua, W., Kurz, N., and Lemire, D.: Faster population counts using AVX2 instructions. The Computer Journal **61**(1), 111–120 (2017)
17. Jain, A. K., and Feng, J.: Latent fingerprint matching. IEEE Transactions on pattern analysis and machine intelligence **33**(1), 88–100 (2011)
18. Kowarschik, M., Weiss, C.: An overview of cache optimization techniques and cache-aware numerical algorithms. In: Algorithms for Memory Hierarchies, pp. 213–232. Springer, Berlin, Heidelberg (2003) https://doi.org/10.1007/3-540-36574-5_10