FAST FINGERPRINT RECOGNITION USING SPIRAL

Woon Ho Jung

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Identification based on fingerprints is an active area of research in biometrics. In this work we present a fast implementation of a recently developed fingerprint identification algorithm based on wavelet packets. We applied various general and domain-specific code optimization techniques to efficiently implement the registration phase, which takes as input a set of fingerprint images and produces an adapted packet tree and an associated wavelet domain template. The code for the actual identification is then generated automatically from a mathematical description of this packet tree using SPIRAL. We discuss the optimization techniques used and present various benchmarks that demonstrate the efficiency of our implementation.

1. INTRODUCTION

The fingerprint identification algorithm that we are going to discuss in this paper requires several mathematical computations. The algorithm makes heavy use of matrix operations such as multiplications and inversions, DFTs of different sizes. Throughout the paper we will discuss not only the hot-spots of the implementations but also the techniques and tools used to improve the hot-spots. We used gprof in order to detect hot-spots of the implementation.

1.1. Motivation

As stated above the fingerprint identification algorithm requires heavy mathematical computations, thus a single algorithm could have different runtimes depending on the implementation of the algorithm. The algorithm consists of 2 stages, the training stage and the verification stage. The training stage is performed off-line thus it is not very sensitive in terms of runtime because the end users will not suffer from it. However it is important to have an efficient implementation of the training stage in order to allow the algorithm developers to quickly run and test different test cases. The verification stage is performed on-line therefore it needs to be as fast as possible. The quality of a fingerprint identification system not only depends on the accuracy of the system but also in the time that it takes to compute the answer. It is clear that an end user will not be satisfied with the system if he or she needs to wait 30 minutes to obtain access to a building.

1.2. Previous Work

We start out with a Matlab implementation of the algorithm [1]. This implementation was developed in order to verify the correctness and the accuracy of the algorithm. However it has sub-optimal runtime performance.

1.3. Our Contribution

In this paper we will introduce a C-version of the algorithm that was developed with the efficiency problem in mind. In addition we explore the possibilities of using SPIRAL to generate code for mathematical expressions slightly different from the ordinary DSP transforms such as the DFT or the DCT.

1.4. Organization of the Paper

In section 2 we are going to describe the architecture of the system and the fingerprint detection algorithm in question. This description is rather simplistic however it is sufficient to understand the algorithm and understand the tuning strategies used. In section 3 we discuss about the theoretical arithmetic cost. In section 4 we introduce different hot-spots of the algorithm and the techniques applied to tune the hot-spots. In section 5 we present SPIRAL, which is a tool used to implement the verification stage of the system. In section 6 we provide benchmarks of the system and the impact of each of the tuning techniques presented in section 4.

2. DESCRIPTION OF THE SYSTEM AND THE ALGORITHM

The system is divided into 2 stages, verification and training. First we will describe the algorithm and describe the system architecture that implements the described algorithm.

2.1. Fingerprint identification algorithm

2.1.1. Training Stage

- Full decomposition of an image: Given $Img = [img_1, img_2, ..., img_n]$, where $img_n$is a training image. The full decomposition routine will construct a full tree for each image in the Img list. Given an image img_1 we apply wavelet packet decompositions in order to obtain 4 subspaces of the original image. This will give us 4 direct child nodes of the initial image img_1. Recursively apply the same routine to the child nodes until we construct a full tree with height equal to 4 (Figure 1.a). We apply the full decomposition to all the images in the Img list and obtain a list of trees.

- Score Tree: Given the list of trees generated in the full decomposition step we combine the data (subspaces generated by the wavelet packet decomposition) stored in the nodes of the trees to generate a single score tree. For instance to generate the root of the score tree we will need the root of all the trees in the tree list generated in the full decomposition step. The score tree contains the Fitness score of each subspace in the nodes of the tree. The fitness score is used to determine whether a node or a subspace contains a enough information or not (Figure 1.b). The fitness score can be computed by the following formula:

  Fitness Score =$1/(U^T(X^+D^{-1}X)^{-1}U)$.

  $X = [DFT(img_1); DFT(img_2); ...DFT(img_n)]$

  each column of matrix X holds the spectrum of one of the training images after projecting it onto the wavelet subspace. D is a diagonal matrix with the average power spectrum of the training images along its diagonal, for simplicity we consider U to be a column matrix with all ones.

- Pruning: Given a score tree generated in the previous step the pruning routine will prune the score tree based on the fitness score stored in each node of the score tree. After the score tree pruning we will be left with only the nodes or subspaces that have better score than the parent node (Figure 1.c).

- Correlation Filter: Given a pruned score tree this step will compute the correlation filter for the nodes or subspaces that are located at the leaves of the score tree. The correlation filter can be computed by the following formula: $H = \overline{D}^{-1}X(X^+\overline{D}^{-1}X)^{-1}U$.

  $\overline{D} = \alpha D + \beta I$. Where α and β are constants parameters between 1 and 0. Finally we save both the correlation filter and the score tree in each node of the pruned score tree.

2.1.2. Verification Stage

- Decompose Input Image: The decomposition in the verification is similar to the one in the training stage. However since we know the structure of the pruned tree we decompose the image into subspaces that match the pruned tree (Figure 2.c).

- Apply DFT: Apply Discrete Fourier Transform to each of the subspaces obtained in the previous step. (Figure 2.d)

- Pointwise multiplication: Pointwise multiply the correlation filter obtained in the training stage with the matrix produced in the previous step (Figure 2.d).

- Apply IFFT: Apply the inverse Discrete Fourier Transform to each of the subspaces obtained to the new matrix produced in the previous step (Figure 2.e).

- Verification: Search for a peak in the generated matrix. If the image is authentic a sharp peak should be present in the matrix (Figure 2.f).

2.2. System Architecture

2.2.1. Training Stage

The training stage takes a set of authentic images and a set of impostor images for each user. The system generates a correlation filter together with a set of functions to be applied to the filter and the input image to determine whether an image is authentic. The correlation filter is generated following the steps explained in the previous section. The functions are generated by SPIRAL. In the training stage once that we have the pruned tree it is possible to parse the tree and generate a formula that SPIRAL can understand. With this formula we can make SPIRAL to generate a fast code for each user that can be compiled into dynamic libraries (Figure 3).

2.2.2. Verification Stage

The verification stage reads the correlation filter generated in the training stage and the image to test. It loads the corresponding dynamic library and it applies the function to the correlation filter and the test image. Finally search for a peak in the matrix generated by applying the functions to the test image (Figure 3).

3. ARITHMETIC COST

In this paper we are not going to analyze the theoretical arithmetic cost of the system.

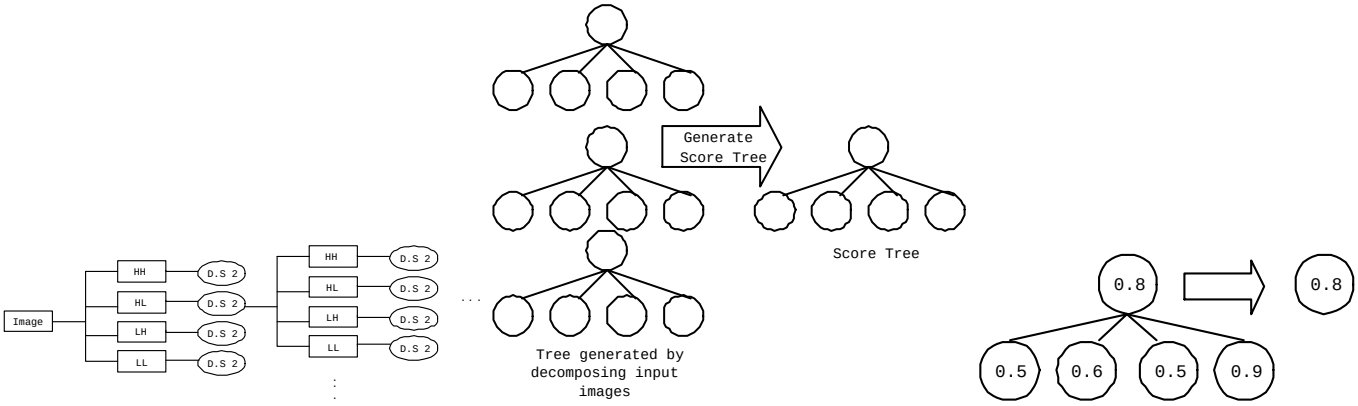

Fig. 1. (a) Recursively decompose the input image to form a tree of subspaces. (b) Take all the trees that has the subspaces of the input images and compute a single score tree. (c) Prune the full score tree.

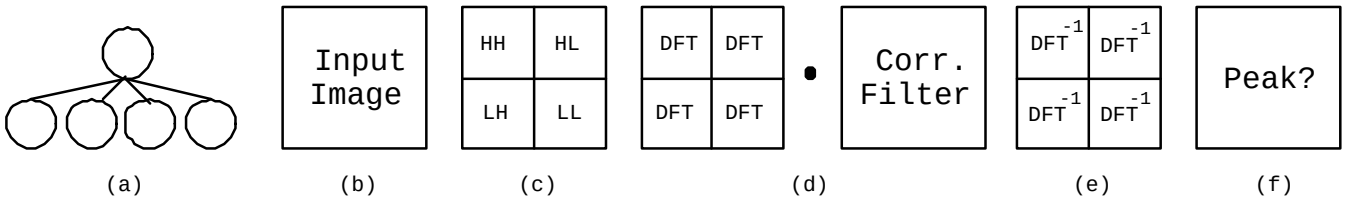

Fig. 2. (a) Pruned score tree, generated in the training stage. (b) Input image, image to identify. (c) Given the pruned score tree divide the input image into subspaces. (d) Apply DFT to each of the subspace and then pointwise multiply with the correlation filter generated in the training stage. (e) After the pointwise multiplication apply the inverse DFT to each subspace. (f) Search for peak, if the input image(b) belongs to the set the final matrix will have a sharp peak.

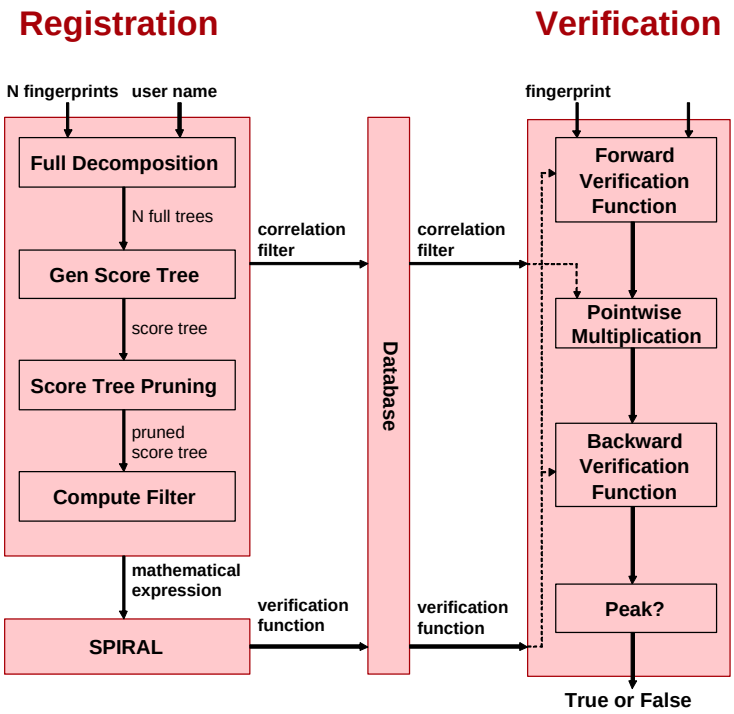

Fig. 3. The training stage generates the correlation filter (user.filt) and the verification functions (user.so). The verification routine receives a fingerprint and applies the function in the dynamic library (user.so) together with the correlation filter (user.filt) to verify the input fingerprint

- The training stage performs a large number of floating operations however the bottleneck of the routine is the call to SPIRAL. In addition all users will have the same arithmetic cost in computing the score tree however they will differ in the total arithmetic cost because we only compute the correlation filters at the leaves of the pruned score tree. Therefore we cannot compute a single generic formula that gives the exact number of floating point operations.

- It is important to compute the arithmetic cost of the verification stage however SPIRAL generates different identification routine for different users. Therefore it is not possible to compute a single generic formula that gives the number of floating point operations.

4. HOT-SPOT TUNING

The bottleneck of the training stage is the call to SPIRAL to generate the code for the verification stage. With this approach we made a tradeoff between the runtime in the training stage and the runtime in the verification stage. Although the training stage becomes substantially slow, SPIRAL guarantees a fast implementation for the verification stage. SPIRAL will automatically optimize the verification stage code for each user. Having a generic verification routine will prevent us from calling SPIRAL every time in order to generate the verification stage code however this generic routine will be much slower than the user specific code generated by SPIRAL. Therefore this can be thought as a trade-off between the training stage runtime and the verification stage runtime. Although the runtime performance of the training stage is not very relevant, we put some effort in optimizing it. The runtime of the training stage is not going to affect the end users because the training is performed "off-line" however it is important for the developers to be able to run experiments in a reasonable time length. Next we will describe some of the hot-spots of the system and techniques used to fix the bottlenecks of the training stage (besides the call to SPIRAL).

- Avoid Multiple Copies of Matrix X

  In computing the fitness score and the correlation filter we need to perform several matrix-matrix multiplications. However we should pay close attention to the term $(X^+D^{-1}X)$. The matrix X is being multiplied by X^+. The math kernel libraries only provide generic routines that perform generic matrix-matrix multiplications. Therfore if we decide to use a math

library we need to have both the matrix X and the matrix X^+ in memory and perform an explicit matrix conjugation. Having two copies of the matrix X in memory is not desirable because it causes a large memory footprint and degrades performance. We decided to implement our own matrix-matrix-matrix multiplication that computes the $(X^+D^{-1}X)$ term in a single routine having a single copy of the matrix X.

- Compute and Store X transposed instead of X

  Our FFT routines (generated by SPIRAL) compute the DFT of the input matrix and it stores the answer in a stride-1. However in order to construct the matrix X we need to have the DFT of the subspaces in the columns of the matrix X. This forces us to use a temporary storage to compute the DFT's and then move the data to the corresponding column in a column major order. This method is not desirable because it requires to have a temporary storage, to copy redundant data and to access the X matrix in a column major order. We decided to compute and store the matrix X^T instead of the matrix X, with this approach we can pass the pointers of the matrix X directly to the FFT routines.

- Trade Divisions for Multiplications

  The matrix D is a diagonal matrix therefore a multiplication by D^{-1} implies division by the elements in the diagonal. Floating point divisions are very expensive compared to floating point multiplications. We decided to pre-compute and to store the matrix D^{-1} in order to replace all the divisions by multiplications.

- Opportunistic computation

  We can observe that the computation involved in computing the fitness score and the correlation filter is somewhat similar, $(X^+D^{-1}X) \simeq (X^+\overline{D}^{-1}X)$. Thus with a very small overhead in the fitness score computation we can pre-compute the $(X^+\overline{D}^{-1}X)$ term of the correlation filter. Therefore our implementation pre-computes part of the correlation filter in the fitness score computation in an optimistic matter.

5. USING SPIRAL TO GENERATE VERIFICATION CODE

As stated before the runtime of the verification stage is very important. Therefore we decided to let SPIRAL automatically generate and optimize code for each distinct users. SPIRAL can take an mathematical expression and automatically generate code that implements the given expression. For each user the training routine will generate an mathematical expression that describes the users score tree and call SPIRAL with the derived expression in order generate the code for the verification stage. It is hard to implement a single generic routine that could satisfy all the users and system platforms. However using SPIRAL we are able to automatically generate a verification routine that is specifically implemented and optimized for each user and the platform that the system is running on.

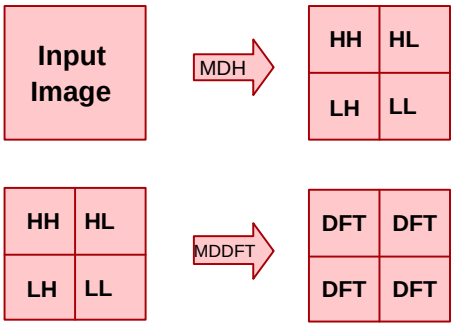

Fig. 4. MDH(n) Splits an image of size nxn into 4 subspaces of size n/2xn/2. MDDFT(n) Applies multi-dimensional DFT of size n/2 to each subspace.

5.1. Generate expressions for SPIRAL

The verification algorithm consists of 2 building blocks : MDH(n) and MDDFT(n). Given a score tree all the leaves represents a MDDFT(n) and all other intermediate nodes represent a MDH(n). We can recursively apply this simple rule to generate the full expression (Figure 4).

$MDH(n) = 1/4L^n_2(I_{n/2} \otimes DFT_2)$

$MDFFT(n) = (I_2 \otimes L^n_{n/2} \otimes I_{n/2})(DFT_{n/2} \otimes DFT_{n/2})(I_2 \otimes L^n_2 \otimes I_{n/2})$

Our mathematical description of the algorithm has the following syntax:

Exp(n) ::= MDFFT(n) |
DirectSum(Exp(n/2),Exp(n/2),Exp(n/2),Exp(n/2))*MDH(n)

6. BENCHMARK

In this section, We present experimental results that compare the performance of the different optimizations presented in section 4. Furthermore, we present comparative performance results for our implementation and the Matlab implementation. The experiments were performed with a Pentium 4 3.0 GHz system with 1024KB of cache and 1GB of RAM. Compilation : gcc -O3 -I. -march=pentium3 -mtune= pentium3 -funroll-loops -msse3.

6.1. Training Routine

We trained different users using 3, 5, 10, 15, 20 authentic training images and measured the runtime of the training

stage. In these measurements we removed the call to SPIRAL. The call to SPIRAL is definitely the bottleneck of the training routine however we decide to remove it because we are trying to show the performance improvements of the training routine due to the changes presented in section 4. The effect on the runtime for each optimization is shown in Figure 5 and discussed next:

- Base: Is the base code for the training routine. It implements the training routine using our own matrix-matrix-matrix multiplication described in section 4. However it does not have any other optimization mentioned in section 4.
- Optimization 1: This is the code fixes the problem discussed in "Compute X Transposed instead of X". This optimization made our system up to 3 times faster than the base-line implementation. The training algorithm constructs several X matrices. Therefore an improvement in the routine that constructs the matrix X gave us a huge improvement in the runtime.
- Optimization 2: This code applies the technique discussed in "Trade Divisions for Multiplications". This optimization made our system up to 2 times faster than the optimization 1. This is because in a Pentium 4 a double precision floating point division requires at least 38 cycles however a floating point multiplication only needs 6 cycles.
- Best: This code is the best code that we currently have. It applies all the techniques mentioned in section 4.

6.2. Verification Routine

We measured 3 different users verification stage runtime. We used 3 different filters from different users. We used fingerprints that are 128x128 in size. Our experiment shows that the code generated by SPIRAL performs between 2X slower and 5X faster than the Matlab implementation. SPIRAL is a code generator/optimizer for DSP transforms such as DFT or DCT and was not specifically designed to generate some arbitrary mathematical expression like in our case. We believe that future versions of SPIRAL will generate even better code for the mathematical descriptions that we are using in this project.

7. CONCLUSIONS

We spent a considerable amount of work in optimizing our system and in some cases we achieved performance improvements up to 8 times faster than the baseline. This research shows that for a numerical computation software not only the algorithm is important but also the efficiency of

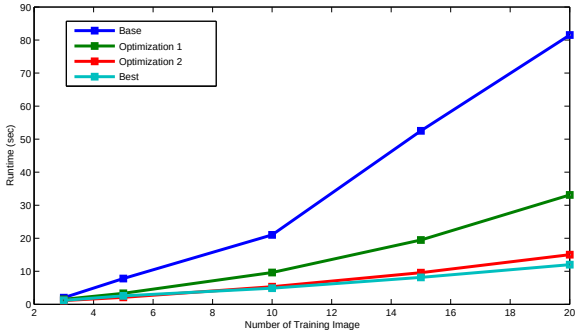

Fig. 5. Runtime plot for different implementations of the training routine.

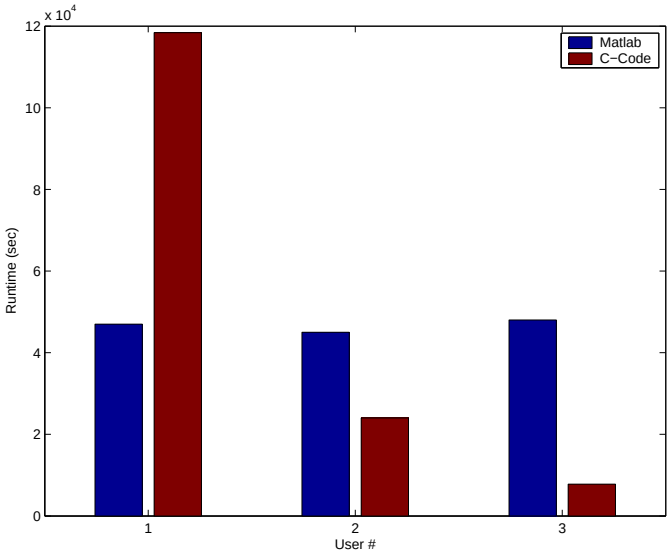

Fig. 6. Verification stage benchmark (3 different users with 3 different score tree).

the actual implementation. Our research also addresses the importance of using an automatic code generators such as SPIRAL. By letting SPIRAL generate the verification code we saved a huge amount time that could be spent in optimizing the training stage, which is not possible to generate using code generators.

8. REFERENCE

[1] Wavelet Packet Correlation Methods in Biometrics (Jason Thornton, Pablo Hennings, Jelena Kovacevic, B.V.K. Vijaya Kumar)