EPFL

Finger Vein Biometric Matching

Lara Sofie Lenz

Cyber Security, ETHZ / EPFL

Semester Project

February 19, 2026

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

LASEC

Abstract

Biometric identification through finger vein patterns is a secure alternative to conventional identification through fingerprints. To make it possible to use the method of finger vein identification in real life, we do not only have to ensure its accuracy in matching fingers but also have to consider the time it takes to extract veins from images and to compare them. In this work, we continue development of a vein extraction pipeline previously created in the LASEC group to improve its execution time. We recreate this pipeline in C++, optimize it algorithmically, and ensure its matching-accuracy by comparing its output to the previous implementation's output.

Contents

1 Introduction

Biometric authentication is a concept that has been around for centuries [19]. Derived from the ancient greek words *bios*, and *metron*, it describes the recognition and identification of humans based on their physical properties [14]. In recent years, biometric authentication has seen a rise in popularity. With the employment of face recognition or fingerprint scanners on most smartphones, biometric identification has become a widely acknowledged way of proving one's identity in everyday settings. Conventional methods for authentication require the possession and carrying along of an identifying document (e.g. a government-issued ID). This document can expire or get lost. The biometric properties of a person on the other hand are persistent and usually available at all times making it a convenient tool for authentication purposes.

Most biometric authentication systems, however, have a weakness. The identifying physical features of the person using them are not only visible to the system but also to everyone else. Faces can be captured on video, fingerprints can be taken from surfaces, and modern technologies enable us to imitate these properties in a variety of ways. It would therefore be of the essence to bind the identity of a human to a physical property that can not be easily observed by an outsider of the system.

One way to do this is by taking internal physical features into consideration. A popular choice for this are finger veins. Finger veins are usually not traceable during everyday activities. They are unique enough to be a differentiating factor between humans [20]. And most importantly, in contrast to other internal features, they are more easily obtainable by making use of special purpose devices. These devices usually function similarly. The finger is illuminated by a near-infrared light. Near-infrared light has the property that it passes through human tissue but is blocked by pigments like hemoglobin, or melanin. Since blood vessels are densely packed with hemoglobin, veins appear as dark shadows [7]. If we take an image of this illuminated finger, we can use it to extract the finger vein pattern.

For this project, we use a finger vein scanner that takes two images of the finger, each one from a different perspective. The finger needs to be horizontally inserted into the device. The two cameras are situated below the finger and the finger is illuminated by four near-infrared lights from above. So there are two lights per camera.

Inconsistencies in the near-infrared image can appear due to varying lighting conditions, the presence of other, more dense, matter in the finger (like bones), or changes in the volume of blood flow [4, 15, 16]. Much research has been conducted on how to extract finger vein features in the most accurate and robust way. It is of utmost importance to remove these inconsistencies in order to avoid any false matches between two sets of images that do not belong to the same person.

Depending on the quality of the image, we usually have to go through several steps of pre- and post-processing to be able to extract the vein features of a finger. We denote these necessary steps before, during, and after feature extraction as a feature extraction pipeline. This feature extraction pipeline is usually run at least twice, once to obtain a *model* image, and once to obtain a *probe* image. The *model* is used as a reference. Whenever a person wants to identify themselves, their finger vein image (the *probe*) will be matched with the *model* image of the person they want to authenticate as.

2 The Old Pipeline

This project builds directly upon the work of Simon Sommerhalder [18] who worked on finding a feature extraction pipeline which would yield the best matching results. The following chapter will give an insight into his work and findings.

Previous Work

One of the main contributions of the previous work was the extensive comparison of different feature extraction methods to obtain a finger vein image. The flow of the original pipeline can be seen in Fig. 1. In the rest of this chapter, we will describe what each pipeline stage was for, how the pipeline changed, followed by a detailed description of the algorithms used in this new pipeline.

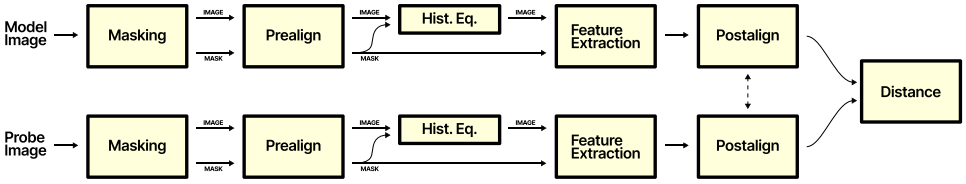

Figure 1: The flow of the original pipeline used in previous projects.

For every image, as a first step, a mask is generated that defines the region-of-interest (ROI). This region should contain the part of the image that actually holds the finger pixels and in that way separates the finger from the background. If we would skip this separation process, we could risk, due to noisy signals, the vein extraction algorithm finding finger vein patterns in the background. After this extraction step, both the unaltered finger image and the mask are passed to the next pipeline stage.

We will call the next stage *Prealignment*. This step uses the images obtained during the masking procedure and roughly aligns the finger before the finger veins are extracted.

The *Histogram Equalization* stage computes the histogram of the intensity values of the masked image region, spreads out the most frequent values, and afterwards rescales the entire image using this histogram. This was done to enhance the contrast of the input image.

The next stage is by far the most important in this pipeline, as it is responsible for actually extracting the finger vein features from the image, thus we call it *Feature Extraction*. This step produces a boolean image,

where each 1 denotes the presence of a vein at the respective pixel position. There exist different methods for extracting features from an image but we will primarily focus on the so called *Maximum Curvature* algorithm proposed by Miura et al. [16].

The next step, called *Postalignment*, aligns an extracted finger vein image. Depending on whether a *model* is available, the image may be post-aligned using information about the *model* to obtain a better matching for the distance measuring.

In the final stage, the distance between two finger vein images is computed (using a suitable distance metric) to obtain a score that indicates how well the images match with each other.

Previous Findings

After the previous work exhaustively ran through all combinations of algorithms for each pipeline step, they found the pipeline which yielded the best results. In Fig. 2, we can find the adapted pipeline with the respective algorithms used in each pipeline step. We can see that the *Histogram Equalization* stage was completely removed from the pipeline, and furthermore, that the *Postalignment* stage was no longer relevant for the *model*'s finger vein extraction.

The Python implementation of this pipeline is the starting point for this project.

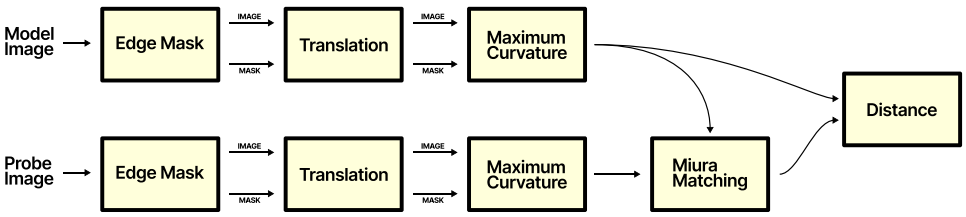

Figure 2: The flow of the pipeline determined to produce the best matching results. Instead of the pipeline steps, the corresponding methods used are written in each block.

2.1 Edge Mask Extraction

The finger vein scanner that we are working with takes two 240×376 (height $\times$ width) grayscale pixel images. These images are taken from two different angles, having a look at Figures 3 and 4, we can see what these images may look like.

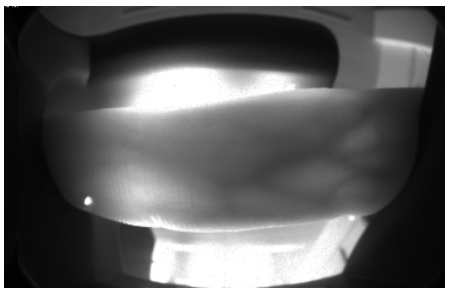

Figure 3: An image taken from the first camera of the scanner.

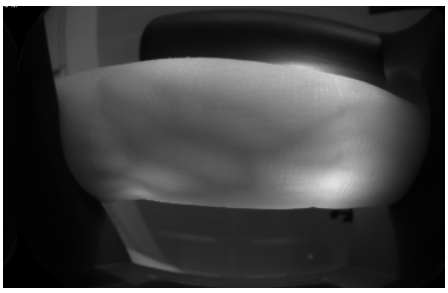

Figure 4: An image taken from the second camera of the scanner.

It is apparent that the camera does not only take a picture of the finger, but inevitably also captures parts of the mechanical assembly the finger is resting on (the foreground), and the space behind the finger (the background). If we would try to obtain finger vein patterns from this image directly, we might end up with a finger vein pattern ranging over the entire image instead of the region where the actual pattern should be found, namely the region of the finger (see Fig. 5). The reason is that due to structures and noise in the background we may find pixel positions out of the region-of-interest (ROI) where the curvature of a cross-sectional profile is a local maximum. Due to the finger vein extraction algorithm that we use [16] which makes use of this exact property, we would then falsely identify these pixels as containing finger veins. To mitigate this risk, it is of utmost importance to restrict the finger vein extraction algorithm to the region that actually contains the finger, i.e. we need to mask our image. A mask is usually a boolean image where each 1 indicates that the pixel of the original image is part of the ROI, and a 0 denotes that the pixel is not part of the ROI. To do the masking, we use a method called *Edge Mask*.

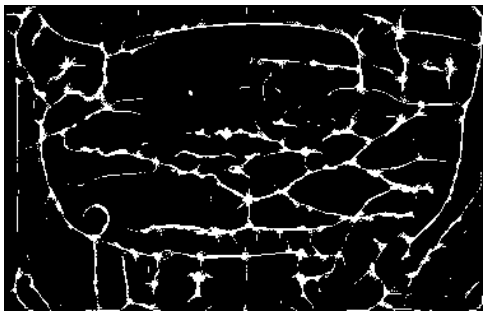

Figure 5: An image showing the extracted veins if no mask is used.

For each camera position, we know which pixels will show the foreground

instead of the actual finger image, thus, we can define a static ROI in the second dimension (over the width of the image, i.e. the columns). Note that we define the first dimension of the image to represent the rows, and the second dimension to represent the columns of the image. This interpretation corresponds to the interpretation of the dimensions of an image in software (more specifically memory), and we will use this interpretation for the rest of this report.

All the columns outside of this ROI will not be considered, i.e. the mask will be 0 for those regions. For the remaining mask computation we proceed as follows: we first compute the gradient of the entire image over both dimensions and then compute the hypotenuse over these gradient values. By doing this, we obtain the gradient magnitude for each pixel which represents by how much the image's intensity values are changing. This result, together with an index in the range of our predetermined, second-dimensional ROI, and a fixed center point is then given as a parameter to the `edge_points` function which returns the upper and lower edge point of each column. How exactly this function operates can be seen in Section 2.1.1.

After obtaining these edge points, we instantiate a mask image where all column values between the upper edge point (inclusive) and the lower edge point (exclusive) are set to 1. We have seen in Figures 3 and 4 that the very bright spots in the image usually lie outside of the expected ROI, thus, we set all the pixels in the mask to 0 for which the pixel value in the original image exceeds a specific threshold. Afterwards, we do morphological closing and opening operations to remove any artifacts that might be present around the border of the mask. Since we know that the edge around a finger is usually convex, we compute the convex hull over all white pixels (value 1). With that we obtain the mask that denotes the overall ROI for our image.

2.1.1 `edge_points`

The `edge_points` function takes as input the gradient magnitudes of an image, a column index i, and a center point for the column. It then extracts the gradient values from the i-th column, computes the average over all these values, and then divides the gradient values element-wise by their average. The resulting one-dimensional array is then given as a parameter to the `max_thresh` function. This function also takes as input the center point, a direction, and a threshold and returns, respectively, the upper or lower edge point of the given column depending on the direction. The `edge_points` function ultimately returns a tuple containing the index value of the column, and the respective row indices of the computed edge points.

2.1.2 `max_thresh`

As already stated, `max_thresh` takes as input a one-dimensional averaged gradient array, a center point (denoted as starting point), a direction, and a threshold value. It then traverses the array, starting from the center point in the given direction, and returns an index. This index either belongs to the maximum value found in the traversed range, where all the values were smaller or equal to the threshold; or the index equals the index of the first occurrence of a value greater than the threshold + 1 (i.e. the next value in traversing direction). The index found denotes an edge point of the column. Depending on the direction, the index either points to the upper edge point (if direction was *"up"*), or the lower edge point (if direction was *"down"*).

2.2 Translation Alignment

This function takes as input an image, and a mask. It then extracts the indices for each non-zero pixel of the mask and separates them by dimension (i.e., we have an array of indices of the first dimension, and a separate array of indices for the second dimension). After that, it fits a linear *model* through these two arrays to obtain a line that best fits all of the values. The estimated coefficient of this line is then used to compute an angle by which the image and mask will be rotated (using `rotateMat`), such that the finger will be horizontally aligned in the image and mask. The center of the rotation is the index (x, y) where x is the average over the indices of the first dimension, and y is the average over the indices of the second dimension. In the end, the rotated mask and input image are shifted (using `shiftMat`) to the absolute center of the image. The goal of this alignment method is to align the finger to the center of the image. This alignment method is an adapted version of the Huang normalization method proposed in [9]. For Huang normalization one needs to compute the center points for each column and fit the center line through these center points. However, since our mask is quite precise in the center but lacks this precision in the edges of the right- and leftmost columns, we would introduce a non-negligible margin of error when computing the center points to fit [18]. We therefore slightly adapt the Huang normalization method [9] by fitting the line through all points that are part of the mask instead of only the center points. In that way, we mitigate the errors introduced by the imprecision of the outermost edges and obtain a center line that more accurately represents the center of our finger.

rotateMat

The functionality of `rotateMat` is quite straight-forward, as it makes use of the **OpenCV** [3] `getRotationMatrix2D`, and `warpAffine` functions. These two functions, given the image to shift, an angle, and a center point, will

calculate a two-dimensional, affine rotation matrix, and apply this transformation to the image.

OpenCV is a great choice for image processing and computer vision tasks, as it has been used in a variety of research papers, and academic projects which demonstrates its suitability for this project. Moreover, **OpenCV** is cross-platform compatible, and its functions are highly performant which gives us more flexibility when designing our pipeline.

shiftMat

This function takes as input an image, and two shift values. Depending on the sign of the shift values, specific edges of the image will be cropped. Afterwards, the opposing edges will be padded with zeros. The padding is done in a way, that the image will remain the same size.

2.3 Maximum Curvature for Feature Extraction

The finger vein feature extraction is done making use of the *Maximum Curvature* implementation provided by the **Biometric Vein Recognition Library (Bob)** [10]. This implementation is based on the idea from Miura et al.'s paper [16] to use the curvature of a cross-sectional profile to find local maxima in the image indicating the presence of a vein in that position. This method is especially suitable for vein images with varying vein widths and brightnesses leading to improved matching accuracy when compared to prior methods [8, 15, 16].

Our `maximum_curvature` function makes heavy use of code taken from the **Bob** Python library, specifically the parts that relate to maximum curvature. The following sections (Sections 2.3.1-2.3.4) describe the inner workings of the functions taken from the **Bob** library.

To extract the vein pixels, we first have to obtain the curvatures of all four cross-sections (horizontal, vertical, 45°, -45° diagonal) that we are interested in. The curvature for each of these directions indicates where valleys in the image could be detected. These valleys indicate a region where we expect a vein to be. This computation is done, using the `detect_valleys` function which, given a masked image, returns a three-dimensional array filled with the curvatures for each cross-section, or in other words, a valley detector for each direction. With these valley detectors, we can compute the probability for each pixel to be the center of a vein. To do that, we call the function `eval_vein_probabilities` which returns a two-dimensional array where each value presents the vein center probability. We then only have to connect the center points of the veins (using `connect_centers`). We can do that by applying a filtering operation on the probability matrix. This filtering operation not only has the effect of connecting the pixels but also

eliminating noise. The filtered probabilities are again a three-dimensional array, since we have to filter for each direction. We cannot directly use this result, but rather have to find the maximum vein probability of all directions for each pixel. This leads us to a two-dimensional array which is of the same size as the input image. Using this array, we can call `binarise` to transform our probabilities into a boolean array where each 1 indicates the presence of a vein at that pixel position, and a 0 denotes its absence.

The details on how exactly the functions `detect_valleys`, `eval_vein_probabilities`, `connect_centers`, and `binarise` work can be found below.

2.3.1 `detect_valleys`

This function takes as input an image, a mask, and a parameter σ denoting the variance of the Gaussian filter that we want. The computation done in this function corresponds to **Step 1-1** described in the original paper by Miura et al. [16]. A copy of the pseudocode description can be found in Appendix A. The goal of this step is to obtain a valley detector κ for each direction of interest. The directions that we are most interested in are the horizontal and vertical directions, as well as the 45°, and -45° diagonals.

As a first step, we construct a two-dimensional Gaussian filter with the use of the input parameter σ, and calculate its first and second derivatives. Afterwards, we smooth the image given our filter derivatives, and calculate the first and second derivatives for all directions that we are interested in. By doing this, we obtain four valley detectors which will be given as an input to the next function, `eval_vein_probabilities`.

2.3.2 `eval_vein_probabilities`

This function takes as input a three-dimensional array of valley detectors and returns a two-dimensional array of vein center probabilities for each pixel. The computation follows **Steps 1-2** to **1-4** of the paper [16] (found in Appendix A) and is done as follows: Each direction (horizontal, vertical, 45°, -45°) is scanned for concavities. If such a concavity is found, its width is measured and the center of the concavity identified. This computation is done in function `_prob_1d`. We then assign a value to this center which depends on the width and depth of the concavity. The concavity center values are added up for all directions leaving us with a two-dimensional array of accumulated vein probabilities for each pixel of the original image.

2.3.2.1 `_prob_1d`

`_prob_1d` takes as input a one-dimensional array. This array is either a row of the horizontal valley detector, a column of the vertical valley detector, a diagonal of the 45° valley detector, or, respectively, an antidiagonal from the

-45° valley detector. The function first creates a thresholded signal of the input array where each 1 denotes that the input array's value was greater than 0. It then computes the difference between the thresholded signal, and a shifted version of this signal to obtain the start and end points of regions with positive values. These regions are then each searched for the maximum value (which coincides with the greatest depth of the cavity). We then generate an array of the same size as the input. At each position where a maximum was found, we assign a value that is equal to the maximum value times the width of the cavity.

2.3.3 `connect_centers`

Given a two-dimensional array of vein probabilities, this function connects vein centers by filtering the vein probabilities for each direction. This computation corresponds to **Step 2 / Equation 4** of [16] (found in Appendix A). For each direction, we either traverse through the rows, columns, diagonals, or antidiagonals of the two-dimensional vein probability array and provide them as input to `_connect_1d`. With that, we will then receive the filtered pixel values for each direction which we can then again store in a three-dimensional array.

2.3.3.1 `_connect_1d`

This function takes as input a one-dimensional array and outputs a one-dimensional array of size (size(input array) - 4). The filtering operation is done as follows:

$$\begin{aligned}\forall w \in \{2, \ldots,\ \text{length(input)} - 3\} :& \\ \text{output}[w] = \min(\ &\max(\text{input}[w+1], \text{input}[w+2]), \\ &\max(\text{input}[w-1], \text{input}[w-2]))\end{aligned}$$

2.3.4 `binarise`

Binarisation is the last step in our finger vein feature extraction algorithm. It takes as input a two-dimensional array containing the result of the maximum filtered vein probabilities. It then computes the median over all positive vein probabilities, and outputs a two-dimensional array of the same size as the input image where each 1 denotes that the vein probability was greater than the computed median. Thus, the resulting output is a two-dimensional boolean array containing the extracted finger veins. A 1 denotes the presence of a vein at that specific pixel, and a 0 denotes the absence of it.

2.4 Miura Matching

This method is used to align the finger vein image after the feature extraction has been performed. Since this function needs an image for comparison, we have to have a *model* and a *probe* available to perform this operation. This is also the main reason, why the *Postalignment* pipeline step is missing in our new pipeline scheme for the computation of the *model* (Fig. 2). To align the *model* and *probe* image, we first compute shift parameters for the *probe*, making use of the `miura_score` function. How exactly these shift parameters are computed can be seen below. After retrieving these shift parameters, we shift the *probe* according to the parameters and obtain a *probe* image that should be maximally aligned with the *model*. It should be noted here, that the *Postalignment* step only performs translation of the *probe* and no rotation of the image. If we would consider rotation for the *Postalignment* step, we would drastically increase the computation time as we would not only have to perform rotations and translations independently, but rather had to exhaustively find the best shift offsets for each possible rotation angle. Moreover, since we previously already rotated the finger image using the line center, we can expect the vein images of the *model* and *probe* to be fairly close in orientation.

2.4.1 `miura_score`

This function takes as input the two-dimensional boolean arrays of the *model* and *probe*, and computes two integer values denoting by how much the *probe* should be shifted in the respective dimensions. This function makes use of the method proposed in [15], where the shift parameters are computed as follows: first, we crop the *model* image by removing 30 pixels on the upper and lower edge, and 90 pixels on the left and right edge, respectively. The cropped image will then be rotated by 180° and used as a kernel for a convolution computation on the *probe*, i.e. we perform correlation on the image and a cropped version of the *model*. The output of this computation should be a matrix, where each element contains the number of pixels overlapping for this specific shift position. We then find the maximum value in this matrix denoting the position at which the *probe* image had the greatest overlap with the cropped *model* or, in other words, at what offset the images should best match each other, and extract the indices of this value. The indices themselves are already the shift parameters that we need.

We additionally compute a score using this shift position to find out how well the images could be aligned. We do this by taking the maximum value found and dividing it by the sum of the number of non-zero pixels of the cropped *model* and the best-aligned region of the *probe*. This means that we also only consider a sub-region of the *probe*. This sub-region is by definition the same size as the cropped *model*, so if the entire images are of

size 240×376 pixels, our regions considered will be of size 180×196 pixels. Using the entire *probe*, and *model* images, and defining the found shift offset to be (t, s), we can compute the score with the following equations:

$$
\begin{aligned}
&\text{\# Slice probe to obtain region with best alignment} \\
&P_{\text{slice}} := \mathit{probe}[t \dots t+180, s \dots s+196] \\
&\text{\# Crop model as defined before} \\
&M_{\text{crop}} := \mathit{model}[30 \dots 210, 90 \dots 286] \\
&\text{\# Compute score on the sliced and cropped regions of the images} \\
&\text{Miura\_Score}(\mathit{probe}, \mathit{model}) := \frac{\text{\# vein-pixels}(P_{\text{slice}} \cap M_{\text{crop}})}{(\text{\# vein-pixels}(P_{\text{slice}})) + (\text{\# vein-pixels}(M_{\text{crop}}))}
\end{aligned}
$$

We define vein-pixels(X) to be a function that returns the pixels in image X that were identified as being part of a vein. If the images could be perfectly matched, the score will equal 0.5. The worse the images match, the lower the score.

To obtain the best shift parameters, we need to perform a correlation on the *probe* and cropped *model*. This correlation is done using Fast-Fourier transforms (FFTs). FFTs are especially efficient for large matrix correlations, as in Fourier space this computation can be obtained through element-wise multiplication.

To make it more clear, how much more efficient it is to use FFT for correlation, we will look at an example: we know that our vein images have a size of $240 \times 376 = 90,240$ pixels. Using a cropped version of the *model*, we have a kernel size of $180 \times 196 = 35,280$ pixels. For simple convolution without FFT, we then would have to do $90,240 \times 35,280 = 3,183,667,200$ multiplications and $3,183,576,960$ additions. In the Fourier domain, we would only need to do $90,240$ complex multiplications using a padded kernel. Transforming to or from Fourier space takes $\frac{m \times n}{2} \times (\log_2(n) + \log_2(m))$ complex multiplications and $m \times n \times (\log_2(n) + \log_2(m))$ complex additions where m denotes the number of rows of the image, and n the number of columns. With a size of 240×376, we will need to do $\frac{240 \times 376}{2} \times (\log_2(240) + \log_2(376)) = 742,742$ complex multiplications and $240 \times 376 \times (\log_2(240) + \log_2(376)) = 1,485,484$ complex additions. Complex additions can be done by adding the real and imaginary part separately, thus, one complex addition needs 2 real-valued additions. For complex multiplications the computation looks as follow $(a + ib)(c + id) = (ac - bd) + i(ad + bc)$. This results in having to compute 4 real-valued multiplications and 2 real-valued additions for a complex multiplication. In total, we will therefore do $2,970,968$ multiplications and $4,456,452$ additions for a transform of a single image to or from Fourier space. For the entire convolution we, hence, will need $90,240 \times 4 + 2,970,968 \times 4 = 12,244,832$ multiplications and

$90,240 \times 2 + 4,456,452 \times 4 = 18,006,288$ additions. This will undeniably be much faster than doing the convolution directly in time space as we perform much fewer operations when using FFT.

2.5 Miura Distance Measuring

The distance computation itself is quite straight-forward, and is an adaptation of the score computation from the function above [15]. Instead of only considering the cropped regions of the *model* and *probe*, we now compare the entire vein images. The distance between two images (denoted here as A, and B) is computed as follows:

$$\text{Miura\_Distance}(A, B) := 1 - \frac{\#\text{ vein-pixels}(A \cap B)}{(\#\text{ vein-pixels}(A)) + (\#\text{ vein-pixels}(B))}$$

where, again, vein-pixels(X) returns the pixels in image X that were identified as being part of a vein. It still holds that if the images were equal, the computed distance will be 0.5. However, now the closer this distance value is to 1, the less vein pixels overlapped and, thus, the worse the images were matching.

3 The Optimized Pipeline

While the pipeline described in Section 2 computationally yielded the best results, the implementation done in Python turned out to be incredibly slow [13]. It was therefore a logical next step to implement this pipeline in a programming language that is considered to be more efficient. The programming language for which we decided was C++. An extraordinarily helpful library that we made use of for this project was the **NumCpp** library [5] which provides C++ functions that are comparable to functions from the **NumPy** library [6]. This **NumCpp** library made a lot of the implementation process easier as the original Python implementation relied heavily on **NumPy**.

While rewriting the code, it became quite obvious that due to the exploratory nature of the previous project, the implementation was not always maximally efficient in regards to how computations were performed. So while prior work focused on improving the extraction and matching results obtained through the pipeline, our focus was to make the computations for the given methods and algorithms as efficient, and in that sense, as fast as possible. To have a better overview of what functions in Python and C++ logically correspond, we tried to keep the function names as similar as possible. In the following sections there are detailed descriptions of the exact optimizations done in and for each pipeline step. However, we would also like to make some general remarks about the Python codebase and our optimizations.

The optimizations were applied in two phases. The first phase mainly consisted of rewriting the code in C++. During this rewriting phase, we sometimes encountered code that was unreachable or did not have any functionality, so instead of blindly following the implementation and applying optimizations afterwards, we directly removed this code from the implementation. We also found that some memory was stored and accessed in a way that was highly inefficient. While accessing and extracting memory in any order (e.g. column-wise) in Python is simple (although still inefficient and time-consuming), it is not possible to imitate the same accessing pattern in C++ without a considerable overhead in complexity. That is why we also opted for the first major optimization during this first optimization pass as we wanted to simplify the implementation and avoid any mistakes that could have happened in the course of this increase in complexity. A more precise description of this optimization can be found in Section 3.2.

During the second phase, we primarily optimized in a targeted way. We analysed the time spent in each stack frame for a call to the pipeline to figure out in what function we spent most of our computational time. After identifying the greatest bottlenecks, we optimized the functions that lead to those bottlenecks. We, additionally, optimized functions that were not

directly bottlenecks but for which we noticed inefficiencies.

We would also like to note that the original code was missing proper documentation for the functionality of each component. This made all the optimization steps (including the rewriting) much more difficult as instead of putting the main effort into optimizing, we had to spend a non-negligible amount of time investigating what the code was doing exactly before we could even think of improving it. So part of our optimizations was not only the improvement of the code, but also adding extensive documentation to the code such that for future developers and researchers it would be easier to understand the abstract concepts of functions as well as the specific inner workings of them.

3.1 Edge Mask Extraction

While the main bottlenecks of the program were inside the *Maximum Curvature* implementation, we quickly recognized that the implementation of the *Edge Mask Extraction* was not ideal either.

As we have already seen in Section 2.1, part of the extraction step was generating edge points for each column, and then using these edge points to create a first version of the mask. Now while there is nothing wrong with this approach, the actual implementation of it was rather inefficient. In the former implementation, the edge points were computed by looping over the ROI in the second dimension, and for each column index calling the `edge_points` function which averaged each element of the column and then used the `max_thresh` function to find the edge point for a specific direction. The result of this computation, originally a list of tuples of the form `[(column_index, upper_edge_point), (column_index, lower_edge_point)]`, was then split into four arrays, each containing one of the specified elements from the result (i.e. an array contained either the first element of the first tuple, the second element of the first tuple, the first element of the second tuple, or the second element of the second tuple). After the edge points were computed for each column and the four arrays respectively filled, we looped over the arrays' contents and set every pixel between edge points to 1. Only after that, we went over the mask again and removed the bright spots by setting every pixel that exceeded a threshold in the original image to 0. It is trivial to see that by doing this, we loop twice over the second-dimensional ROI, and additionally once over the entire mask.

For our rewritten and optimized implementation, we followed a different approach. Instead of calling `edge_points`, we directly compute the average over the entire gradient making use of the **NumCpp** library's [5] `average` function. This function is especially fast since it makes use of the **C++ standard library**'s [11] `accumulate` function which is highly optimized and efficient. Afterwards, we iterate over the second-dimensional ROI, and

average each column, directly call an optimized version of `max_thresh`, and instead of storing the result, we immediately set the values between the edges to 1, but only if the original pixel was not too bright. In that way, we reduced the number of loops from three to one, while also doing inline computations where appropriate.

As a side-note, the former implementation made inconsistent use of the edge points. When setting the values to 1 between edge points, the upper edge point was always included, while the lower edge point was always excluded. Since both edge points should be handled equally, we removed this ambiguity by including both edge points in the mask.

3.1.1 `max_thresh`

The first thing we noticed about this function was that it was always called twice. Once for the upwards direction, and once for the downwards direction. So our first optimization consisted of combining these two directions and traversing through both directions during one call. The respective output was then a tuple containing the upper and lower edge points found.

The second optimization was less concerned with efficiency but rather tried to eliminate inconsistencies during the computation of an edge point. As stated in Section 2.1.2, the function always returned an index. This index either belonged to the maximum value of all traversed values where all the values were below the threshold; or the index belonged to the value that followed the first value above the threshold. Now looking at what these values represent, we know that the gradient magnitude indicates by how much the pixel values are changing in that direction which also means that the higher the value the greater and faster the change. We can safely imply by this that a large gradient magnitude indicates the presence of an edge at that pixel. We know that in the region that we are traversing, there exists the edge point contouring the finger. Since the pixels within the finger are fairly similar, we expect the gradient magnitudes within the finger to be low. While traversing through the gradient magnitudes in one direction, we expect to find one that denotes the edge of the finger. If all gradient magnitudes are below the threshold that means that the edge was not as clear and we have to assume that the edge was at the pixel position with the highest magnitude encountered. This, we correctly compute by taking the maximum of all traversed values if all values were below the threshold. However, the first gradient magnitude above the threshold that we encounter indicates an unambiguous edge of the finger, so instead of taking the index of the value after that, we should be taking the index of this exact value. This inconsistency was most likely introduced due to a faulty loop design changing the index by one before exiting the loop.

3.2 Maximum Curvature for Feature Extraction

As expected, this pipeline step was the biggest bottleneck in our execution. Since most of the computational work is done in this step, it was also the one that benefited the most from optimizations. We can see in Fig. 6, that the `maximum_curvature` function takes up 95.59% of the entire execution time.

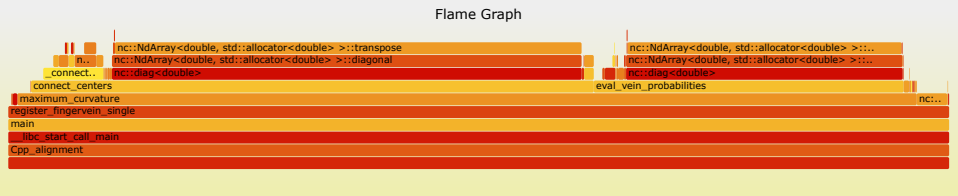

Figure 6: A flamegraph showing the percentage of time spent in each stack frame of a function, i.e. a representation of which functions take up most of the execution time.

One of the biggest optimizations for this particular pipeline step was the improvement of memory access patterns. As already broached in the beginning of the chapter, the original implementation provided by the **Bob** library [10] was written in Python. Python, and especially the **NumPy** library [6] make it very easy to access and extract elements from a n-dimensional matrix where $n > 1$. We have to bear in mind however that while the **NumPy** interface is easy to use, the memory access patterns, and thus the memory access times, for such matrices are the same for Python and C++. We therefore need to be careful when defining dimensions for a multi-dimensional matrix in order to avoid out-of-order retrieval of elements.

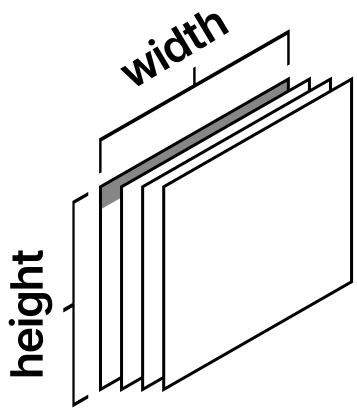

Figure 7: A visualization of how the four 240×376 matrices are aligned in the former implementation. The gray area denotes the values that we intend to access.

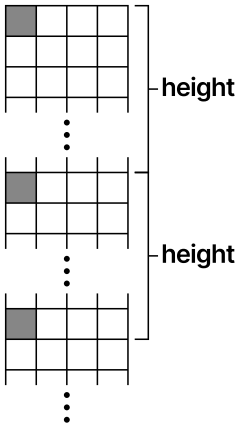

Figure 8: A visualization of the memory access pattern produced by Fig. 7. The gray boxes indicate the memory regions that need to be accessed when wanting to read the first row of the first matrix.

After the `detect_valleys` function executed, we obtain a three-dimensional array. The **Bob** library's implementation defines the dimensions of this array to be of size $240 \times 376 \times 4$. A visual representation of what this matrix looks like can be found in Fig. 7. Depending on how the memory is accessed, this matrix definition can be optimal, but we can already see in function `eval_vein_probabilities` that this is not the case in our implementation. We access this three-dimensional matrix, by always fixing the third dimension, depending on the direction fixing the first or second dimension, and then traversing through the remaining one-dimensional array. Explaining this in a more abstract way and making use of the explanations in Section 2.3.2, for each direction, we traverse through its derivative matrix via rows, columns, or diagonals. In Fig. 8, we can see how the memory would be accessed when the dimensions are defined as in [10]. As we can see, no element is ever accessed sequentially. We know that caches make use of the property that memory accesses are most often spatially and temporally related, meaning that if we access a specific element in memory we will most likely access it again soon, or will access an element that is close to this element in memory. The cache will therefore load entire lines of memory. With an access pattern as in Fig. 8, we only access memory that is spatially close after long periods of time, and the same memory after an even longer period. This dramatically increases the risk of the cache line already having been evicted the next time we access an element that is spatially close. This

will result in a high number of cache misses which are usually penalized as it takes time to search the cache, and reload the line back into it.

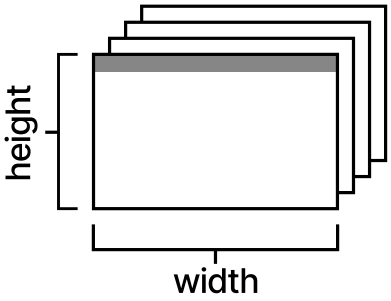

Figure 9: A visualization of how the four 240×376 matrices are aligned in the optimized implementation. The gray area denotes the values that we intend to access.

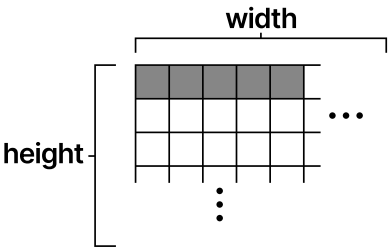

Figure 10: A visualization of the memory access pattern produced by Fig. 9. The gray boxes indicate the memory regions that need to be accessed when wanting to read the first row of the first matrix.

We therefore redefined the dimensions of this matrix. The resulting dimensions then had the size $4 \times 240 \times 376$, and a visual representation can be found in Fig. 9. As we can see, the derivative matrices will now be much more naturally aligned in memory. Having another look at the memory access patterns produced by this matrix definition, we can see (Fig. 10) that memory is accessed with more spatial locality now, resulting in more cache hits and, thus, a decreased number of cache misses which allows us to reduce the memory retrieval time.

While we did not change anything about the main structure of the pipeline, we did rewrite some helper functions to speed-up the computations. The helper functions that we had a look at were `_prob_1d`, `_connect_1d`, and **NumCpp**'s `diag`.

3.2.1 NumCpp's `diag`

While examining Fig. 6, we could see that an incredible amount of time was spent in this function. And more specifically, in transposing the matrix for this function. For our implementation, we only needed very simple functionality. We needed a function that would return an array of the n-th diagonal elements of a two-dimensional matrix. As the **NumCpp** implementation provided the same functionality as the **NumPy** implementation, namely that a diagonal could not only be extracted, but also a diagonal matrix constructed, this function introduced much more complexity than what we needed. We, thus, decided to implement our own function called `diag_elems` which given a matrix, its size, and the index of the diagonal in

question, would extract the diagonal. Using this new function instead of the one provided by **NumCpp** reduced the execution time of the entire pipeline dramatically.

3.2.2 `_prob_1d`

In the original implementation, the vein centers were found by using a 1-shift difference of a filtered input array. Each non-zero difference then denoted the beginning and end of a vein valley for which the maximum value (depth of concavity) was found and scaled (Section 2.3.2.1). While this approach was mathematically correct, it was a rather complicated way to get to what we wanted. We therefore opted for a more straight-forward way. Instead of doing all of these operations on the array, we traverse once over the array using a state machine. We know that positive values in the array indicate a valley, thus, whenever we encounter the first positive value, we start searching for a maximum. While we encounter more positive values, we keep track of the maximum value found and its index. As soon as we encounter a value that is ≤ 0, we know that the valley has ended. We can then proceed as before and scale the maximum value. With this, we reduce the time spent on searching and computing on the array.

3.2.3 `_connect_1d`

As described in Section 2.3.3.1, we corrected the vein probabilities by shifting the input matrix twice to the left and twice to the right, and applying a vector operation to compute the filter. In a similar fashion as in the optimized `_prob_1d` function, in our optimized version, we traverse through the input array once. Instead of applying vector operations, we directly filter the pixels element-wise, and return the resulting filtered output.

3.3 Miura Matching & Distance Measuring

The next optimization is more involved and presents an improvement regarding repetition of specific pipeline steps. As we know from Fig. 2, the *Postalignment* and Distance Measure pipeline steps are only relevant when we want to compare two vein images. It is expected that we will match finger vein images much more often than we will actually run the pipeline on them. Thus, it is important to also reduce the computation time for matching veins.

In the original pipeline, we correlated the *probe* and a cropped version of the *model* to find out at what position the *probe* and *model* best match. The correlation was done using Fast-Fourier transform (FFT) as especially for large matrices, computing the FFT and doing element-wise multiplication in Fourier space is much more efficient than computing the correlation directly.

We should not forget that in order to use this Fourier space property, we need our vein images to be in Fourier space. As we have seen in Section 2.4.1, computing the FFT adds great overhead to the correlation computation. In the original pipeline, for each *Postalignment* step, we had to transform the *probe* and cropped *model* to Fourier space, carry out the multiplications, and then apply an inverse Fourier transform to obtain the results that we needed for shifting. Only to then use this shifted *probe*, to compute a score for the correlation of the entire vein images.

As our optimization, we instead computed the Fourier transform for each vein image immediately after the extraction pipeline was completed. Whenever two vein images need to be matched, we can directly use their Fourier representation to compute the correlation between them. Then we only need to do the inverse Fourier transform to obtain a matrix in which the maximum value denotes how many pixels were maximally overlapping in the **entire** *probe* and *model* images, i.e. how well the **whole** images could be matched. We can directly plug this result into our *Miura Distance* computation, as it is equivalent to the # vein-pixels ($A \cap B$) term. In addition, computing the FFT representation of an image allows us to easily obtain the number of non-zero pixels by extracting the real part of the first element of the Fourier matrix. By doing that, we spare ourselves from having to count the number of non-zero pixels for each image.

Moreover, this optimization leads to more accurate results, as the entire *probe* and *model* are considered for the best match instead of only sub-regions of them. In the end, we save one Fast-Fourier transform computation for each additional matching operation.

Nonetheless, using FFT does not only entail advantages. The Fourier transform cannot be represented using binary values, thus, storing Fourier matrices will take up much more space than just storing the boolean arrays directly. In the end, it is a trade-off between the efficiency in time and the efficiency in space, and depending on the circumstances, one or the other might be preferred.

Applying all of the optimizations above left us with a pipeline where we could completely remove the *Postalignment* step, and in which we explicitly compute a Fast-Fourier transform (FFT) for the captured finger vein images. We use the word explicit here, as instead of before where the FFT was hidden in the convolution computation of the *Postalignment* step and repeated for each matching operation, we now store the Fourier representation of the images making it a separate computation step in the pipeline. A visualization of the new pipeline can be found in Fig. 11.

The reason why the performance of this particular pipeline is of such importance is that this pipeline is an essential part of VeinoCert [12, 13].

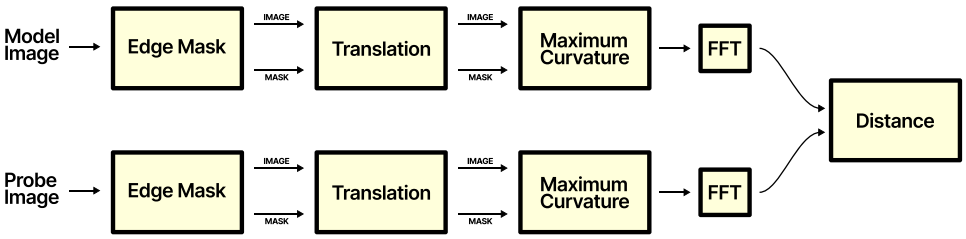

Figure 11: The flow of the fully optimized pipeline.

VeinoCert is a system and protocol that offers biometrical binding to physical documents with the help of Smartcards. Instead of fingerprints, it uses finger vein patterns to identify humans.

The VeinoCert system is implemented in Rust. To make our implementation callable, we needed to provide a C library of our C++ implementation. This library consists of 7 functions that can be called from the Rust interface. The first two functions are called `register_fingervein_single`, and `register_fingerveins`. Both of these functions run the pipeline on a grayscale input image to obtain a *model*. The only difference is, while the first function runs the pipeline for a single image, the second runs the pipeline for the images provided by both cameras, i.e. generates two *models*, one for each perspective. The functions `compare_model_with_input_single` and `compare_model_with_input` work in a similar fashion. Each function takes as input a finger image (the *probe*), and a vein image (the *model*). Again, the second function takes two finger and vein images, respectively. The pipeline is then run on the finger images, and the extracted *probe* veins are matched to the *model*'s veins. The remarkable thing about these two functions is that if the pipeline was already run for a *probe*, the pipeline result can be reused for the next matching operation and does not need to be recomputed.

This is possible due to a custom caching structure that we introduced. This caching structure is opaquely returned to Rust after every call to the matching function. If an initialized caching structure is given as input, the function will take the pipeline result from the structure and in that way avoid rerunning the pipeline which makes the matching between one *probe* and many *models* particularly efficient.

4 Pipeline Analysis

For our analysis we will consider three versions of the code. The first version we considered is the original Python implementation presented in [18]. We will use this as the baseline of our comparisons. The second version is the first implementation of the algorithm in C++. This version as described in Section 3, contains an almost faithful implementation of the Python code but transcribed in C++. As noted before, only a few optimizations were applied for this implementation. The last version considered is the current, optimized implementation of the pipeline.

We want to investigate how well these versions perform, and what speedups we might have obtained by rewriting and optimizing the code in C++. We compare the execution times for each version of the entire pipeline and each pipeline step. The execution times were measured on an **Intel Core i7 Ice Lake** processor under two different conditions (with and without Turbo Boost enabled). Turbo Boost is a technology that allows the CPU to opportunistically run at up to the Max Turbo Frequency while not transgressing the limits of safe temperatures and power consumption. During light workloads the CPU runs at base clock speed whereas during heavy workloads it can jump to higher clock speeds. This can lead to increased performance during processing [1].

However, due to the nature of Turbo Boost, these jumps in clock speed are not predictable (they are highly dependent on the current load of the entire system and the temperature of the processors). That is why execution times can vary substantially between runs. It is therefore recommended to disable Turbo Boost when wanting to measure the execution time of specific processes.

Since this application will be used in real-world settings, we did not want to exclusively report the results obtained without Turbo Boost. In reality, all of these applications will be run with Turbo Boost enabled, thus, we decided to perform a second benchmark. While the results for this second benchmark will have a higher variance, they will show more realistically how the pipeline actually performs.

We additionally carried out measurements of the performance of the API. Knowing the execution time of the pipeline is a great first step to evaluate the efficiency and performance of our implementation. But in the end, what matters is how much time of the entire protocol is spent in our function. Timing measurements were taken for each API function with the special amendment that the comparison functions were measured twice. Once without using the cached result (i.e., running the pipeline), and once with using the cached result. These experiments were exclusively measured on the C++ implementations as those were the only ones providing an API.

For all of these timing measurements, we ensured consistent and comparable results by starting the experiments on a cooled down processor and

with no active workload on the CPU. We additionally warmed up the caches and repeatedly measured the execution time to reduce timing variance between experiments and avoid outliers.

To analyse the pipeline we not only have to compare the execution times of the Python and C++ implementation but we also have to check that the accuracy of the matching results is similar. One of the problems that we have to consider is that while floating point numbers are standardized [2], and operations on them should yield the same results no matter which programming language is used, in practice there do exist differences. For a single computation these differences might not be as apparent but they can get propagated and amplified as the algorithm goes on. We therefore have to check whether our optimized implementation can keep up with the results obtained through the former implementation. The accuracy of the pipeline results will be measured computing the equal error rate (EER). The EER can be computed through the *Failed Acceptance Rate (FAR, false positive rate)*, and the *False Recognition Rate (FRR, false negative rate)* and is the point where *FAR=FRR*.

We will further provide a graph containing the receiver operating characteristics (ROC) curves for each version that we compare. ROC curves are a useful visualization tool to show and compare the performance of one or multiple binary classifiers for different thresholds, as they display the ratio between the *true positive rate (tpr)*, and the *false positive rate (fpr)*.

As an additional note, our experiments were run on more *model-probe* pairs than the experiments presented by [18]. We ran the experiments with 2390 unique *model-probe* pairs for the same finger, and 157,210 *model-probe* pairs for different fingers. Moreover, we optimized the experiments in a way that they would avoid unnecessary recomputations, and would instead reuse prior results if they were not an essential part of performance testing.

5 Results

The feature extraction pipelines yielded fairly similar results. In Figures 13 - 18, we can see the resulting output after each pipeline step for the Python and fully optimized versions. It is apparent that due to the optimizations the outputs do slightly differ, but that the results are generally quite close to each other. It is therefore to no surprise that the distance results between the different versions were also quite comparable. To accurately compute the EER and ROC curves, we needed to sample matching scores for as many unique *model-probe* pairs as possible. As already stated in Section 4, we computed the distances for 2390 same-finger pairs, and 157,210 different-finger pairs. The distance scores generally ranged from 0.5 to 0.94 for images of the same finger, and 0.83 to 0.96 for images of different fingers. We observed that independently of the implementation, it seemed to be impossible to obtain a score below 0.65 for images of the same finger if the images were not identical, i.e. if the images were taken during different interactions with the scanner. Besides the peak at 0.5, the distance scores for every version were relatively normally-distributed around 0.82. From the obtained scores, we could compute the EER for each implementation version and possible threshold. The Python implementation (*V1*) achieved an EER of 0.020887 for a threshold of 0.9, the first C++ implementation (*V2*) had its lowest EER at a threshold of 0.91 which was an EER of 0.020837, and the fully optimized implementation (*V3*) achieved an EER of 0.019534 for a threshold of 0.9. Due to the similarity of results, and the resulting correspondence in *false positive rate*s *(fpr)* and *true positive rate*s *(tpr)*, the ROC curves for each implementation are quite overlapping as can be seen in Fig. 12. Nonetheless, when having a close look at the curves, we can observe that the fully optimized version (*V3*) produced slightly better results than the Python (*V1*) or unoptimized C++ version (*V2*). Whereas the results produced by *V1* and *V2* were almost identically accurate.

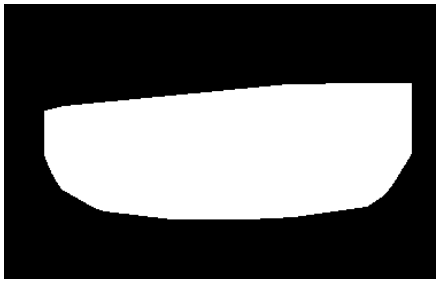

Figure 13: The edge mask computed by the Python implementation (*V1*).

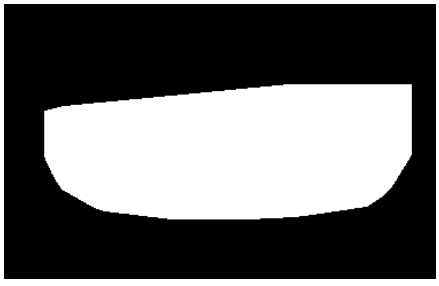

Figure 14: The edge mask computed by the fully optimized C++ implementation (*V3*).

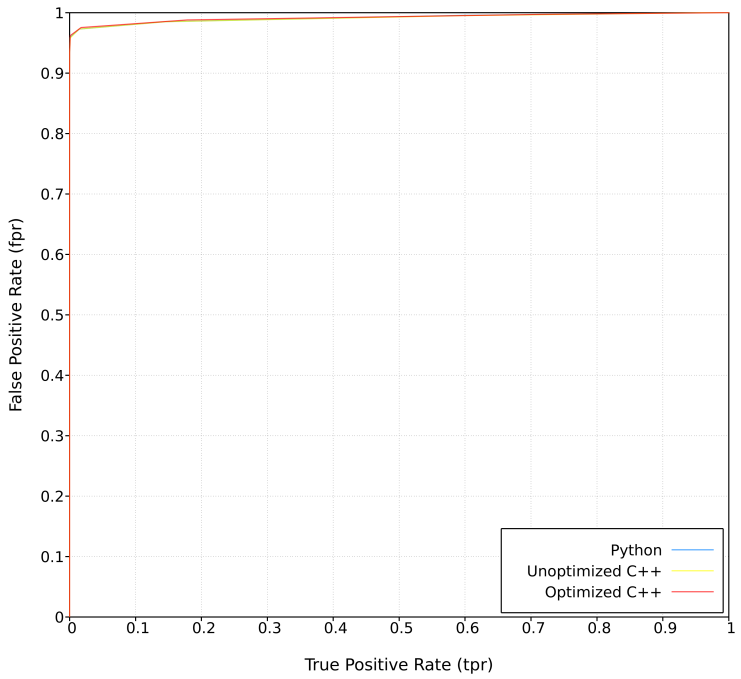

Figure 12: The ROC curve graph showing the matching accuracy of all considered implementation version.

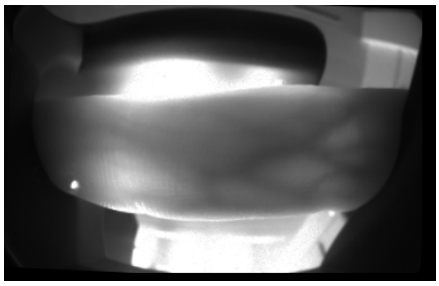

Figure 15: The prealigned image produced by the Python implementation (*V1*).

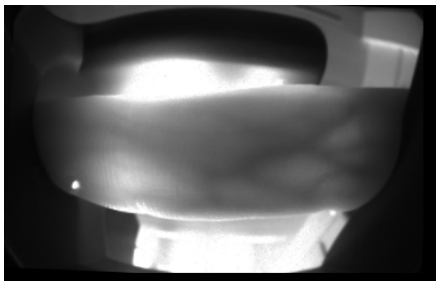

Figure 16: The prealigned image produced by the fully optimized C++ implementation (*V3*).

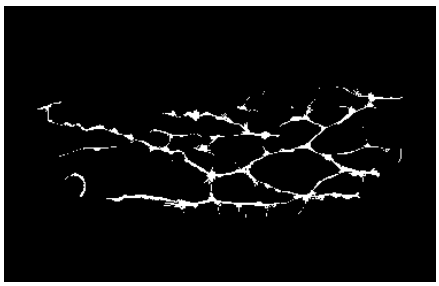

Figure 17: The extracted vein pattern produced by the Python implementation (*V1*).

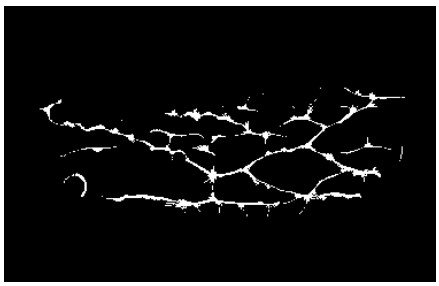

Figure 18: The extracted vein pattern produced by the fully optimized C++ implementation (*V3*).

Now that we have seen how accurate the different versions are, we want to find out how much time they take to execute. As a first benchmark, we timed the execution of the complete pipeline and each pipeline step while Turbo Boost was disabled. A visualization of the obtained results can be found in Figures 19 - 25.

Below, we provided a table (see Table 1) containing the mean and standard deviation values for each timing experiment. As we can see, the execution times drastically decrease for the C++ implementations, and we obtain the best results for the fully optimized version. Since our fully optimized version merged the functionality of the *Postalignment* and Distance computation step, we report the timing values for the *Postalignment* and Distance steps for *V1* and *V2* separately, and added an additional row where the Distance computation time of *V3* is compared to the combined *Postalignment* and Distance computation time of *V1* and *V2*.

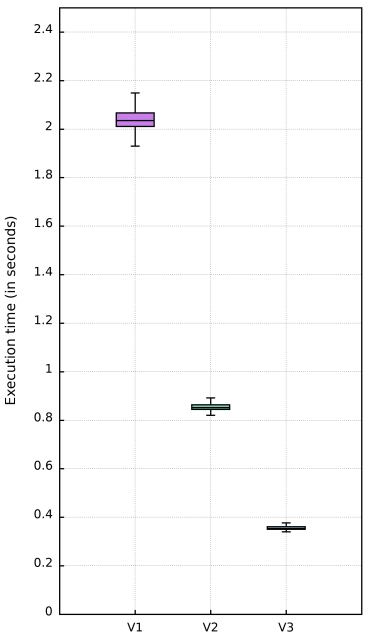

Figure 19: A graph showing the measured execution times with Turbo Boost disabled for the entire pipeline.

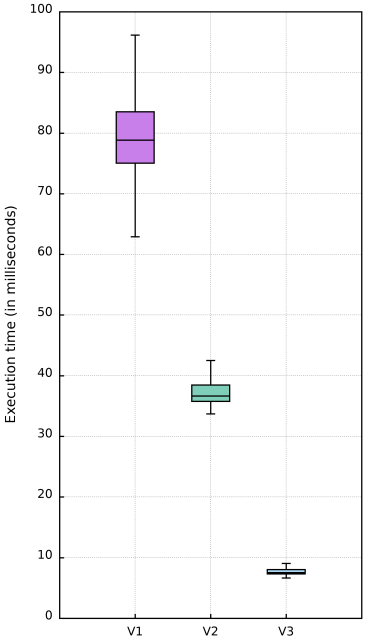

Figure 20: A graph showing the measured execution times with Turbo Boost disabled for the *Edge Mask* computation.

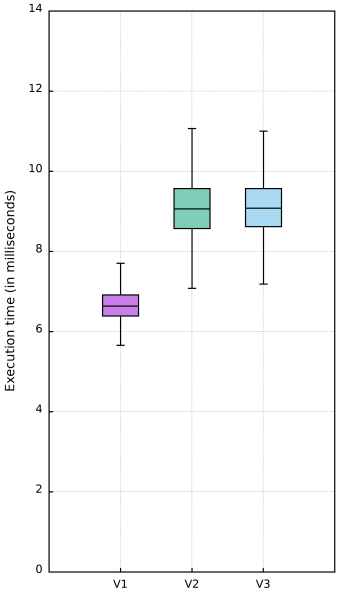

Figure 21: A graph showing the measured execution times with Turbo Boost disabled for the *Prealignment* step.

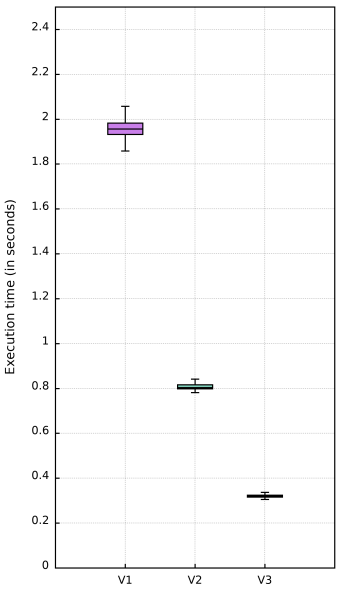

Figure 22: A graph showing the measured execution times with Turbo Boost disabled for the *Maximum Curvature* computation.

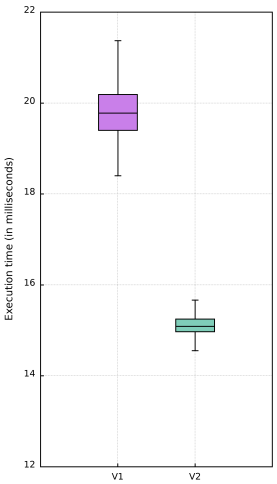

Figure 23: A graph showing the measured execution times with Turbo Boost disabled for the *Postalignment* step.

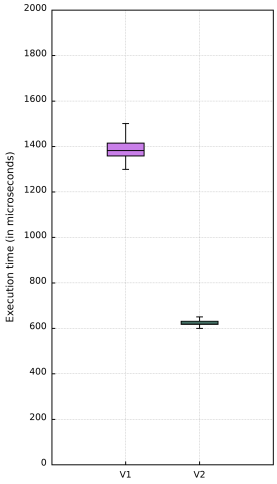

Figure 24: A graph showing the measured execution times with Turbo Boost disabled for the plain distance computation of *V1* and *V2*.

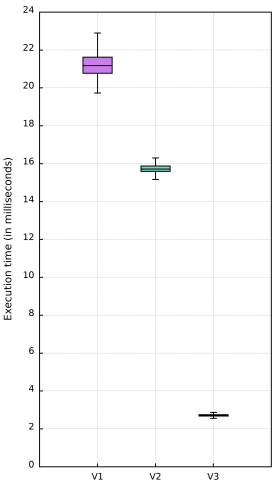

Figure 25: A graph showing the measured execution times with Turbo Boost disabled for the corrected distance computation.

		V1 (in s)	*V2* (in s)	*V3* (in s)
Complete Pipeline	μ	2.044722	0.859715	0.359969
	σ	0.057245	0.031744	0.018146
Edge Mask	μ	0.08056	0.03947	0.00828
	σ	0.009019	0.008892	0.002312
Prealignment	μ	0.006799	0.009221	0.009292
	σ	0.001112	0.00178	0.001871
Maximum Curvature	μ	1.960239	0.818776	0.323334
	σ	0.044181	0.049599	0.01474
Postalignment	μ	0.019849	0.015189	
	σ	0.000841	0.000504	
Plain Distance	μ	0.001407	0.00063	
	σ	0.00012	$6.3921 \cdot 10^{-5}$	
Corrected Distance	μ	0.021256	0.015818	0.002764
	σ	0.000888	0.000514	0.000276

Table 1: Time measurement results (measured in seconds) for the entire pipeline and each pipeline step while Turbo Boost was disabled.

Comparing the Python and fully optimized C++ implementation, we ob-

tain a speedup of **5.7x** for the complete pipeline execution, and a speedup of **9.7x** for the *Edge Mask* computation. There is no speedup for the *Prealignment* step, as we did not optimize it, but we do obtain a speedup of **6x** for the *Maximum Curvature* computation which makes a significant difference as it was the main bottleneck of the entire computation. As already described above, we cannot really compare the Distance functions directly but rather have to compare the speedup between the combination of the *Postalignment* and Distance computation, and the new Distance implementation. This speedup was **7.7x**.

Since these results were obtained disabling Turbo Boost, it is to be expected to see a drastic increase in speedup when Turbo Boost is enabled. As before a visualization of the results obtained with Turbo Boost can be found in Fig. 26 - 32, and the mean and standard deviation for each version can be found in Table 2. We can see that the variance of timing results drastically increased for the Python implementation and that the Python version did not really profit from Turbo Boost while the C++ versions experienced a general speedup. As expected we still achieve the best results with the fully optimized C++ implementation. The difference made by Turbo Boost can best be seen by comparing the speedups of the Python and optimized C++ implementations. The speedup for the entire pipeline execution was **11.5x**, and for the *Edge Mask* computation it was even **18.7x**. The *Prealignment* step, again, had not that much of a speedup as it was not optimized. The *Maximum Curvature* computation had a speedup of **11.2x**, whereas the new Distance function yielded a speedup of **19x**.

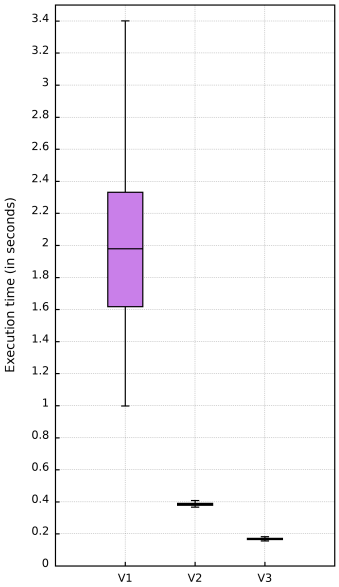

Figure 26: A graph showing the measured execution times with Turbo Boost enabled for the entire pipeline.

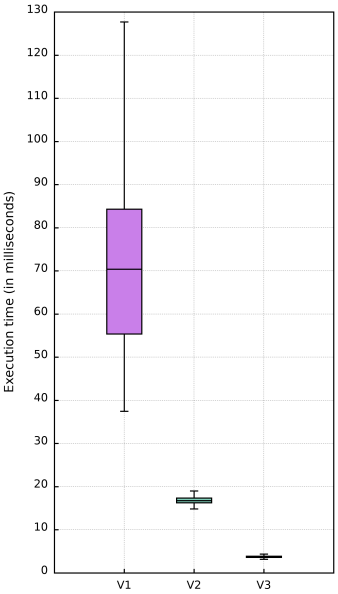

Figure 27: A graph showing the measured execution times with Turbo Boost enabled for the *Edge Mask* computation.

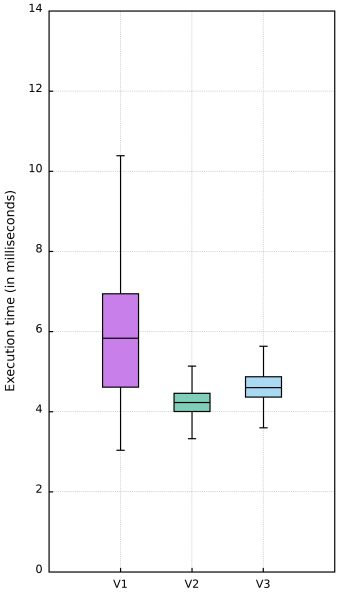

Figure 28: A graph showing the measured execution times with Turbo Boost enabled for the *Prealignment* step.

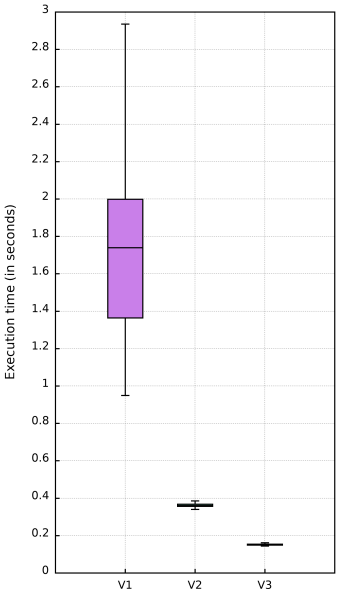

Figure 29: A graph showing the measured execution times with Turbo Boost enabled for the *Maximum Curvature* computation.

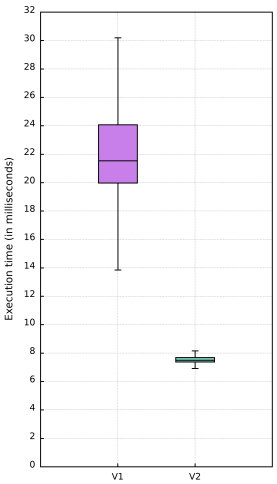

Figure 30: A graph showing the measured execution times with Turbo Boost enabled for the *Postalignment* step.

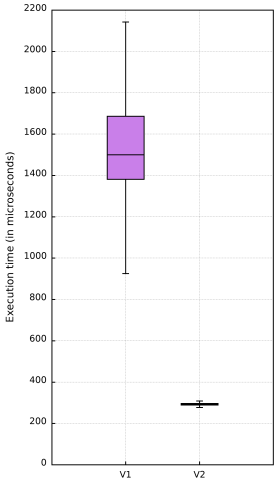

Figure 31: A graph showing the measured execution times with Turbo Boost enabled for the plain distance computation of *V1* and *V2*.

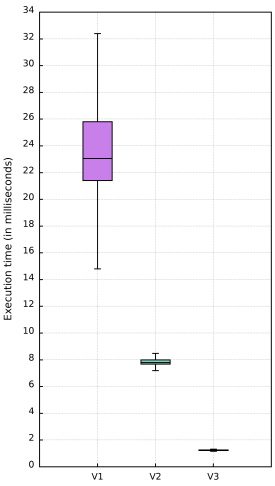

Figure 32: A graph showing the measured execution times with Turbo Boost enabled for the corrected distance computation.

		V1 (in s)	*V2* (in s)	*V3* (in s)
Complete Pipeline	μ	1.969198	0.387614	0.170519
	σ	0.424002	0.017806	0.009598
Edge Mask	μ	0.072014	0.017158	0.003849
	σ	0.019911	0.002081	0.000473
Prealignment	μ	0.006043	0.004284	0.004659
	σ	0.002194	0.001518	0.00058
Maximum Curvature	μ	1.715232	0.363249	0.153078
	σ	0.384366	0.015857	0.006914
Postalignment	μ	0.022223	0.007586	
	σ	0.003242	0.000427	
Plain Distance	μ	0.001574	0.000298	
	σ	0.000329	$6.0327 \cdot 10^{-5}$	
Corrected Distance	μ	0.023797	0.007884	0.001254
	σ	0.003467	0.000433	0.000103

Table 2: Time measurement results (measured in seconds) for the entire pipeline and each pipeline step while Turbo Boost was enabled.

At the end, we computed the execution times for each API function with and without Turbo Boost. We further differentiated the matching times for images of the same and different fingers, and even measured the time it takes to compare two images if the caching feature is used or not. The respective results can be found in Tables 3 and 4. The second table shows the API measurements for different finger comparisons, we also measured the timing for same finger comparisons but since the results are so similar, we opted for a more concise table here. The entire table can be found in Appendix B.

Function name	Turbo Boost	*V2* (in s)	*V3* (in s)
`register_fingervein_single`	no	$\mu = 0.82397$ $\sigma = 0.00388$	$\mu = 0.32702$ $\sigma = 0.00659$
	yes	$\mu = 0.29469$ $\sigma = 0.00362$	$\mu = 0.12566$ $\sigma = 0.00235$
`register_fingerveins`	no	$\mu = 1.64889$ $\sigma = 0.00827$	$\mu = 0.64586$ $\sigma = 0.00354$
	yes	$\mu = 0.58636$ $\sigma = 0.00589$	$\mu = 0.23997$ $\sigma = 0.00494$

Table 3: API timing measurements (measured in seconds) for functions `register_fingervein_single`, and `register_fingerveins`.

Function	Caching enabled	Turbo Boost enabled	*V2* (in s)	*V3* (in s)
single compare	no	no	$\mu = 0.83512$ $\sigma = 0.00342$	$\mu = 0.32303$ $\sigma = 0.00171$
		yes	$\mu = 0.29425$ $\sigma = 0.00311$	$\mu = 0.11794$ $\sigma = 0.00349$
	yes	no	$\mu = 0.0156$ $\sigma = 0.00197$	$\mu = 0.00166$ $\sigma = 5.26 \cdot 10^{-5}$
		yes	$\mu = 0.00537$ $\sigma = 0.00017$	$\mu = 0.00059$ $\sigma = 6.39 \cdot 10^{-5}$
two-image compare	no	no	$\mu = 1.67699$ $\sigma = 0.00405$	$\mu = 0.64298$ $\sigma = 0.00378$
		yes	$\mu = 0.57361$ $\sigma = 0.01017$	$\mu = 0.23496$ $\sigma = 0.00473$
	yes	no	$\mu = 0.03036$ $\sigma = 0.00039$	$\mu = 0.003266$ $\sigma = 5.74 \cdot 10^{-5}$
		yes	$\mu = 0.01055$ $\sigma = 0.00037$	$\mu = 0.00113$ $\sigma = 7.93 \cdot 10^{-5}$

Table 4: API timing measurements (measured in seconds) of the compare functions for different finger comparisons.

6 Discussion & Future Work

As we could see in Section 5, some implementations performed notably better with Turbo Boost while others did not achieve much of a speedup. This is most likely due to the fact that the longer a processor computes the higher its temperature rises which results in Turbo Boost reducing the clock speed in order not to damage the processor. Since for example, the Python execution time is much higher than the one of the C++ implementations, it is much more likely that the clock speed is reduced during the Python computation leading to a reduced speedup. This also explains why we obtain better results for the API measurements than for the pipeline step measurements that are called by the API function. Since the API measurements were done on a much smaller scale, namely not for every image of the dataset but a random, representative sample set, the processors did not overheat during the API function executions leading to faster computation times.

From the results in Section 5, we can, nonetheless, deduce that the pipeline execution greatly benefited from the port to C++ and the succeeding optimizations. Especially the improvement in execution time for the *Maximum Curvature* and Distance computation are important for our work. The *Maximum Curvature* step was the greatest bottleneck in our pipeline, thus, reducing its computation time greatly improved the execution time of the entire pipeline. As the pipeline is not only run for each registration of a finger but also for each comparison to extract the veins from the *probe* image, it was important to make the pipeline execution as fast as possible. Furthermore, as we will much more often compare veins than register them, it is a great success having reduced the Distance computation time to 1.254 milliseconds, and the API comparison for an image to 590.75 microseconds.

We can further report that the parts of the pipeline that were optimized are not likely to yield further performance gains as having another extensive look at the timings of each stack frame after optimization showed that for each function most time was spent in the **C++ standard library**. As this library is heavily optimized and performant, we can safely deduce that the time we currently need for these pipeline steps is the time we need to perform the actual computations. At this point, additional speedup could only be achieved by changing the methods used for the optimized pipeline steps. It is important to note though, that the methods currently used were chosen due to their performance in accuracy and not their performance in speed. Switching to different methods might cause the matching results to perform worse. We should undeniably put a greater emphasis on the accuracy of our results than on their timing, given that an implementation that is fast, but falsely matches veins is unusable for our purposes. Moreover, as our current pipeline implementation executes well under one second, it is reasonable to say that the pipeline is sufficiently fast for our real-world applications.

Future projects should therefore focus on further improving the accuracy of the matching results instead of decreasing the execution time. The next finding is a good example for why we still need to improve accuracy.

A rather surprising aspect of the results was the fact that images of the same finger that are not identical yield matching scores that are fairly high (starting from 0.66). This implies that the extracted vein images, although they are from the same finger, still differ in a lot of places. Our goal in the future should therefore be to improve the pipeline in a way that images of the same finger can be matched with a higher probability while not increasing or even better decreasing the probability for different finger matches. As a first step, it needs to be investigated which part of the pipeline causes these drastic differences. It is a possibility that either the feature extraction step identifies veins at different positions for different images indicating that the feature extraction method might not be as stable as we would have hoped for, or it could be that the veins are correctly extracted and would perfectly match if the images were transformed in a different way. We might even think about considering small rotations during the matching phase. In the end it is subject to future work to find a way to improve the matching results.

Due to the fact that the VeinoCert protocol uses Smartcards, part of this project was to consider whether the matching (compare) operation could be done on a Javacard itself. This would have allowed us to reduce the image information send over the network and in that way decrease the risk of information leakage.

Unfortunately after our investigations, we consider it highly unlikely to perform the matching step on a Javacard. To perform the comparison, we would need to permanently store the *model* on the card, and we would additionally need space for the *probe* image during a comparison. Since both 240×376 images are in Fourier space, we would need at least $2 \times 737,284 = 1,474,568$ bytes (≈ 1.5 MB) of storage. Most Javacards have an EEPROM of size $32 - 64$ KB, and a RAM size of 4 KB. Not even the *model* image alone would fit into the storage of the card, let alone two images and the program for the comparison computation.

But even if we disregard the memory constraint, we would still be bound by the Javacard's CPU. Most of these CPUs have a clock frequency of $1-10$ MHz. We usually cannot compute during every cycle but for the sake of the following example, we will assume exactly that and that the CPU has the highest possible clock frequency (10 MHz). On a Javacard, a single multiplication takes around 112 cycles, and an addition around 89 cycles [17]. With our best case CPU we would therefore need $11.2\mu s$ for a multiplication and $8.9\mu s$ for an addition. We know from our analysis above that a convolution using FFT takes $12,244,832$ multiplications and $18,006,288$ additions. Since our images are already in Fourier space, we can deduct

some of these operations and end up with a total of $6,302,896$ multiplications, and $9,093,384$ additions. The time it would take to compute all of the multiplications is $70.6s$, and accordingly $80.9s$ for all additions. We would therefore need at least $151.5s$ to complete the correlation part of the matching function. It is obvious that the comparison computation on the Javacard would take an unacceptably long time.

This brings us to the conclusion that the usage of Javacards for the comparison of vein images is unsuitable. Following projects should find an alternative storage space where the vein information can be securely stored and computed on without revealing any data.

7 Conclusion

In this work, we have given an extensive description of the pipeline architecture used in previous projects. We have rewritten the former Python implementation in C++ and additionally optimized it. Part of our contribution was adding detailed documentation to the rewritten code to make it easier to work with our implementation in the future.

After optimization, we compared the different versions of the pipeline making sure that our new pipeline implementation produced comparable results to the old one while greatly improving upon the execution time. We showed that our new fully optimized C++ implementation achieved a lower EER than the Python version and that it yielded a speedup of up to **11.5x** for the entire pipeline computation.

We put the reported results into perspective and explained what consequences these findings have and how they will shape future work.

References

[1] What is Intel Turbo Boost Technology? https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html. Accessed: 2024-05-29.

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] Joon Hwan Choi, Wonseok Song, Taejeong Kim, Seung-Rae Lee, and Hee Chan Kim. Finger vein extraction using gradient normalization and principal curvature. In Kurt S. Niel and David Fofi, editors, *Image Processing: Machine Vision Applications II*, volume 7251, page 725111. International Society for Optics and Photonics, SPIE, 2009. doi:10.1117/12.810458.

[5] David Pilger. Numcpp: A templatized header only c++ implementation of the python numpy library, 2023. URL: https://dpilger26.github.io/NumCpp/doxygen/html/index.html.

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[7] J. Hashimoto. Finger vein authentication technology and its future. In *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers.*, pages 5–8, 2006. doi:10.1109/VLSIC.2006.1705285.

[8] A.D. Hoover, V. Kouznetsova, and M. Goldbaum. Locating blood vessels in retinal images by piecewise threshold probing of a matched filter response. *IEEE Transactions on Medical Imaging*, 19(3):203–210, 2000. doi:10.1109/42.845178.

[9] Beining Huang, Yanggang Dai, Rongfeng Li, Darun Tang, and Wenxin Li. Finger-vein authentication based on wide line detector and pattern normalization. In *2010 20th International Conference on Pattern Recognition*, pages 1269–1272, 2010. doi:10.1109/ICPR.2010.316.

[10] Idiap Research Institute. Biometric vein recognition library, 2022. URL: https://www.idiap.ch/software/bob/docs/bob/bob.bio.vein/stable/index.html.

[11] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fifth edition, December 2017. URL: https://www.iso.org/standard/68564.html.

[12] Leonhard Koblitz. Veinocert. Semester project thesis, EPFL, June 2024.

[13] Piotr Kulpinski. Implementation of veinocert protocol. Semester project thesis, EPFL, January 2024.

[14] Mark Maguire. The birth of biometric security. *Anthropology Today*, 25:9–14, 04 2009. doi:10.1111/j.1467-8322.2009.00654.x.

[15] Naoto Miura, Akio Nagasaka, and Takafumi Miyatake. Feature extraction of finger vein patterns based on iterative line tracking and its application to personal identification. *Syst. Comput. Japan*, 35(7):61–71, jun 2004.

[16] Naoto Miura, Akio Nagasaka, and Takafumi Miyatake. Extraction of finger-vein patterns using maximum curvature points in image profiles. *IEICE - Trans. Inf. Syst.*, E90-D(8):1185–1194, aug 2007.

[17] Harald Schlatte-Schatte. Performance estimation for smartcard applications ported onto java cards. Master thesis, TU Graz, March 2012.

[18] Simon Sommerhalder. Fuzzy extraction for finger veins. Semester project thesis, EPFL, June 2022.

[19] Houbing Song, Glenn A. Fink, and Sabina Jeschke. *Front Matter*, pages i–xxx. Wiley-IEEE Press, 2017. doi:10.1002/9781119226079.fmatter.

[20] Yapeng Ye, He Zheng, Liao Ni, Shilei Liu, and Wenxin Li. A study on the individuality of finger vein based on statistical analysis. In *2016 International Conference on Biometrics (ICB)*, pages 1–5, 2016. doi:10.1109/ICB.2016.7550089.

A The Maximum Curvature Algorithm

We provide here a transcript of the pseudocode referenced in Section 2.3. The pseudocode is taken from the description of Miura et al.'s *Maximum Curvature* algorithm in [16].

Step 1: Extraction of the center positions of veins:

Due to the various widths and brightnesses of finger veins, they suggest a method checking cross-sectional profiles of a finger-vein image. The cross-sectional profile of a vein looks like a dent since the vein is darker than the surrounding area. These concave curves have large curvatures independently of the width and brightness. The center positions of veins can be obtained by calculating local maximum curvatures in these cross-sectional profiles.

To make the feature extraction more robust, only the positions of the center lines are emphasized. A score is assigned to each position, and it is larger when its dent is deeper or wider.

Step 1-1: Calculation of the curvatures:

F is a finger image, and $F(x, y)$ is the intensity of pixel (x, y). We define $P_f(z)$ as a cross-sectional profile acquired from $F(x, y)$ at any direction and position, where z is a position in a profile. To relate a position of $P_f(z)$ to that of $F(x, y)$, the mapping function T_{rs} is defined as $F(x, y) = T_{rs}(P_f(z))$.

The curvature, $\kappa(z)$, can be represented as

$$\kappa(z) = \frac{d^2 P_f(z)/dz^2}{(1 + (dP_f(z)/dz)^2)^{\frac{3}{2}}}. \tag{1}$$

Step 1-2: Detection if the centers of veins:

A profile is classified as concave or convex depending on whether $\kappa(z)$ is positive or negative. If $\kappa(z)$ is positive, the profile $P_f(z)$ is a dent (concave).

In this step, the local maximums of $\kappa(z)$ in each concave area are calculated. These points indicate the center positions of the veins.

The positions of these points are defines as z'_i, where $i = 0, 1, \ldots, N-1$, and N is the number of local maximum points in the profile.

Step 1-3: Assignment of scores to the center positions:

Scores indicating the provability that the center positions are on veins are assigned to each center positions.

A score, $S_{cr}(z)$, is defined as follows:

$$S_{cr}(z'_i) = \kappa(z'_i) \times W_r(i), \tag{2}$$

where $W_r(i)$ is the width of the region where the curvature is positive and one of the z'_i is located.

If $W_r(i)$, which represents the width of a vein, is large, the probability that it is a vein is also large. Moreover, the curvature at the center of a vein is large when it appears clearly. Therefore, the width and the curvatures of regions are considered in the scores.

Scores are assigned to a plane, V, which is a result of the emphasis of the veins. That is,

$$V(x'_i, y'_i) = V(x'_i, y'_i) + S_{cr}(z'_i), \tag{3}$$

where (x'_i, y'_i) represents the points defined by $F(x'_i, y'_i) = T_{rs}(P_f(z'_i))$.

Step 1-4: Calculation of all the profiles:

To obtain the vein pattern spreading in an entire image, all the profiles in a direction are analyzed. To obtain the vein pattern spreading in all directions, all the profiles in four directions are also analyzed. The directions used are horizontal, vertical, and the two oblique directions intersecting the horizontal and vertical at 45°. Thus, all the center positions of the veins are detected by calculating the local maximum curvatures.

Step 2: Connections if vein centers:

To connect the centers of veins and eliminate noise, the following filtering operation is conducted. First, two neighboring pixels on the right side and two neighboring pixels on the left side of pixel (x, y) are checked.

If (x, y) and the pixels on both sides have large values, a line is drawn horizontally. When (x, y) has a small value and the pixels on both sides have large values, a line is drawn with a gap at (x, y). Therefore, the value of (x, y) should be increased to connect the line. When (x, y) has a large value and the pixels on both sides of (x, y) have small values, a dot of noise is at (x, y). Therefore, the value of (x, y) should be reduced to eliminate the noise.

This operation can be represented as follows.

$$\begin{aligned} C_{d1}(x, y) = \quad & \min\{\max(V(x+1, y), V(x+2, y)) \\ & + \max(V(x-1, y), V(x-2, y))\}. \end{aligned} \tag{4}$$

The operation is applied to all pixels.

Second, this calculation is made for each of the four directions in the same way, and C_{d2}, C_{d3}, C_{d4} are obtained.

Finally, a final image, $G(x, y)$, is obtained by selecting the maximum of C_{d1}, C_{d2}, C_{d3}, and C_{d4} for each pixel. That is, $G = \max(C_{d1}, C_{d2}, C_{d3}C_{d4})$.

Step 3: Labeling the images:

The vein pattern, $G(x, y)$, is binarized by using a threshold. Pixels with values smaller than the threshold are labeled as parts of the background, and those with values greater than or equal to the threshold are labeled as parts of the vein region. We determined the threshold such that the dispersion between the groups of values in $G(x, y)$ was maximized, assuming that the histogram of values in $G(x, y)$ was diphasic in form.

B API measurements for compare functions for both same and different fingers

Function	Compare between same or different finger	Caching enabled	Turbo Boost enabled	*V2* (in s)	*V3* (in s)
single compare	same	no	no	$\mu = 0.83512$ $\sigma = 0.00271$	$\mu = 0.32303$ $\sigma = 0.00264$
			yes	$\mu = 0.2921$ $\sigma = 0.00443$	$\mu = 0.11864$ $\sigma = 0.00221$
		yes	no	$\mu = 0.01502$ $\sigma = 0.00034$	$\mu = 0.0017$ $\sigma = 5.73 \cdot 10^{-5}$
			yes	$\mu = 0.00548$ $\sigma = 0.00034$	$\mu = 0.0006$ $\sigma = 5.14 \cdot 10^{-5}$
	diff	no	no	$\mu = 0.8389$ $\sigma = 0.00342$	$\mu = 0.32264$ $\sigma = 0.00171$
			yes	$\mu = 0.29425$ $\sigma = 0.00311$	$\mu = 0.11794$ $\sigma = 0.00349$
		yes	no	$\mu = 0.0156$ $\sigma = 0.00197$	$\mu = 0.00166$ $\sigma = 5.26 \cdot 10^{-5}$
			yes	$\mu = 0.00537$ $\sigma = 0.00017$	$\mu = 0.00059$ $\sigma = 6.39 \cdot 10^{-5}$
two-image compare	same	no	no	$\mu = 1.67699$ $\sigma = 0.00494$	$\mu = 0.64298$ $\sigma = 0.00625$
			yes	$\mu = 0.59007$ $\sigma = 0.00583$	$\mu = 0.23536$ $\sigma = 0.00401$
		yes	no	$\mu = 0.02954$ $\sigma = 0.00158$	$\mu = 0.0033$ $\sigma = 9.59 \cdot 10^{-5}$
			yes	$\mu = 0.01065$ $\sigma = 0.00062$	$\mu = 0.00113$ $\sigma = 9.81 \cdot 10^{-5}$
	diff	no	no	$\mu = 1.67385$ $\sigma = 0.00405$	$\mu = 0.6414$ $\sigma = 0.003781$
			yes	$\mu = 0.57361$ $\sigma = 0.01017$	$\mu = 0.23496$ $\sigma = 0.00473$
		yes	no	$\mu = 0.03036$ $\sigma = 0.00039$	$\mu = 0.00327$ $\sigma = 5.74 \cdot 10^{-5}$
			yes	$\mu = 0.01055$ $\sigma = 0.00037$	$\mu = 0.00113$ $\sigma = 7.93 \cdot 10^{-5}$

Table 5: API timing measurements (measured in seconds) of the compare functions for both same and different finger comparisons.