Can you do it without looking? : Fully Homomorphic Image Processing

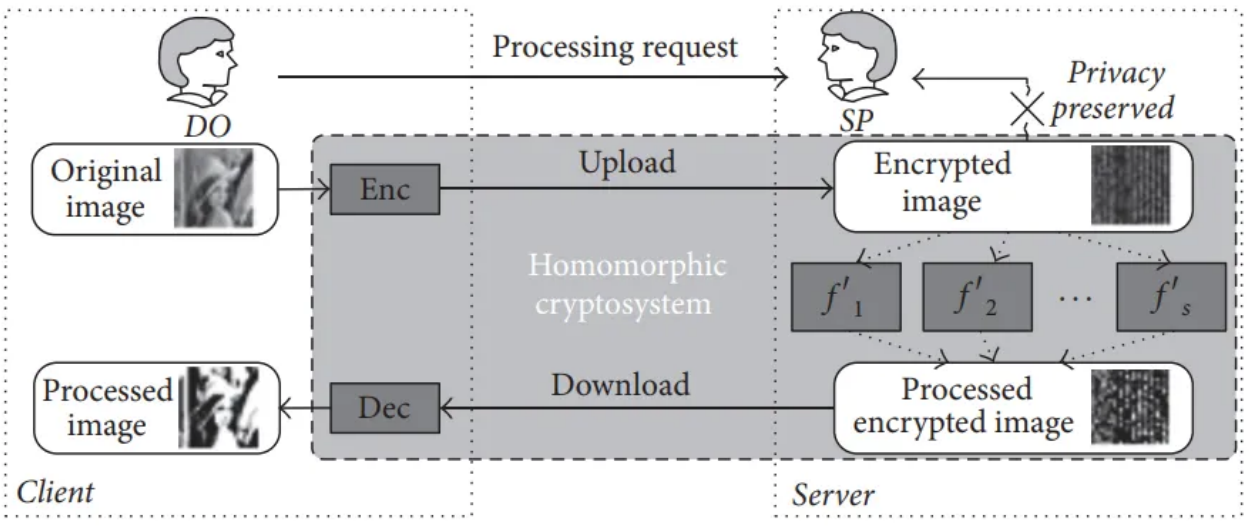
Team - Compare and Contrast; Ishwar B Balappanawar, 2021101023; Bhargav Srinivas, 2021101065

Problem definition

The problem is to perform image stitching on two encrypted images and produce a stitched image which on decryption will give the stitched version of original images.

You want to do expensive/heavy image processing on your images, but your device is not powerful enough. You can send it to a cloud service to do the processing, but you don't want to send personal information. What do you do? We propose using homomorphic encryption, which lets us perform computation on data that is encrypted.

We picked SIFT as a way to demonstrate the feasibility of modern fully homomorphic encryption (FHE) for image processing. This is because the various concepts involved in the image stitching pose interesting and sometimes new unseen challenges of working with FHE.



(b) Encrypted image processing service model

Methodology

We will be using one of CKKS or BFV encryption schemes for FHE. These schemes give us the capability of performing *addition, multiplication and subtraction* on encrypted data. Our plan is to encrypt each pixel individual and do computation on them.

Relevant Papers

The paper that got us motivated in this direction Fully Homomorphic Image Processing - though which we can do things like resizing, scaling, compression etc.

We found a few papers which extract **SIFT features in homomorphic encryption setting -** Image Feature Extraction in Encrypted Domain With Privacy-Preserving SIFT , Towards Efficient Privacy-preserving Image Feature Extraction in Cloud Computing

However these papers reveal some amount of information about the SIFT points to the server which we want to eliminate. The primary reason for revealing the information is because the **comparison operator is extremely expensive to implement**, however there have been recent works on **implementing efficient comparison** operators.

Like this latest 2020 AsiaCrypt paper : Efficient Homomorphic Comparison Methods with Optimal Complexity

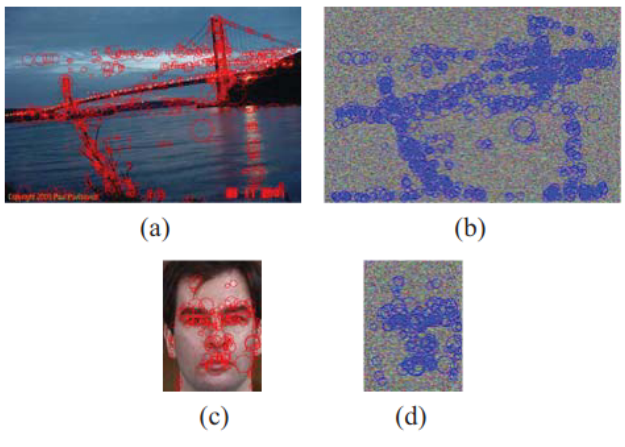


Fig. 5. (a) and (c) Detection of SIFT features in the plaintext domain. (b) and (d) Ciphertext domain. (Best viewed on a color display).

Secure SIFT (Homomorphic)

Deliverables

Given the "***novel/exploratory***" nature of the project especially towards the later half. We had promised the following -

- Implementations of SIFT from scratch
- Perform Image Stitching using SIFT implemented Above
- Implement basic Image Processing Manipulations in Homomorphic setting - Brightness/Contrast Manipulation, Convolutions, Image Resizing etc.
- Implement SIFT in Homomorphic setting

Methedology

Breaking Down SIFT

The sift algorithm:

The SIFT algorithm involves the following steps and computational tasks:

1. **Scale-space Extrema Detection**

- **Goal**: Identify points that are scale-invariant by searching for local extrema in the scale space.
- **Steps**:
  - Create a *scale-space* by convolving the image with Gaussian filters at multiple scales.
  - Subtract adjacent Gaussian-blurred images to produce *Difference of Gaussians (DoG)*.
  - Find local extrema in the DoG images by comparing each pixel with its neighbors in both the spatial (3×3 region) and scale dimensions.
- **Computational Tasks**:
  - Convolve the image with Gaussian kernels (computationally expensive).

    - Subtract Gaussian images to create the DoG images.
    - Iterate through the image and scales to find local extrema.

2. Keypoint Localization

- **Goal**: Refine the location of detected keypoints and remove unstable ones.
- **Steps**:
    - Use interpolation (e.g., Taylor expansion) to refine keypoint locations.
    - Discard keypoints with low contrast or those lying on edges (unstable keypoints).
        - Low contrast: Magnitude of DoG value is below a threshold.
        - Edge response: Use the Hessian matrix's eigenvalues to determine edge-like regions and discard them.
- **Computational Tasks**:
    - Interpolate extrema locations to sub-pixel accuracy.
    - Compute Hessian matrices and eigenvalues for edge detection.

3. Orientation Assignment

- **Goal**: Assign an orientation to each keypoint for rotation invariance.
- **Steps**:
    - Compute gradient magnitudes and orientations for pixels around the keypoint.
    - Create an orientation histogram of these gradients (e.g., 36 bins covering 360°).
    - Assign the dominant orientation (peak of the histogram) to the keypoint. Additional keypoints can be created for significant secondary peaks.
- **Computational Tasks**:
    - Compute gradient magnitude and direction for surrounding pixels.
    - Construct and analyze the orientation histogram.

4. Keypoint Descriptor Generation

- **Goal**: Generate a descriptor vector that describes the local image region around the keypoint in a robust and invariant way.
- **Steps**:
    - Take a 16×16 neighborhood around the keypoint.
    - Divide the neighborhood into a 4×4 grid of 4×4 subregions.
    - Compute gradient magnitudes and orientations for each subregion.
    - Create an 8-bin orientation histogram for each subregion.
    - Concatenate these histograms to form a 128-dimensional feature vector.
- **Computational Tasks**:
    - Compute gradients and orientations for the 16×16 neighborhood.
    - Build orientation histograms for each subregion.
    - Normalize the descriptor vector for illumination invariance.

Our goal was to implement all these computational tasks using the primitives provided to us for an FHE scheme. Some of the tasks involve computing gradients and hessians which warrant the use of floating point numbers. For this reason we chose to use the CKKS encoding scheme. We used the tenseal python library, as it is a well supported wrapper around the popular Microsoft SEAL library. The primitives that we utilize are addition and multiplication. So we implemented all the necessary computational tasks using addition and multiplication. There is the further constraint of multiplicative depth, which prevents us from performing a long chain of multiplications on an encrypted number. This is one of the most significant hurdles with adapting any algorithm to the FHE setting.

Some of the tasks, such as convolution and matrix multiplication, were trivial to implement using addition and multiplication. However there were multiple tasks that were far from trivial, we will discuss these in the next section.

The Easy Parts

- Addition, Blurring, Resizing
- Convolution
- Difference of Gaussian

The Hard Parts

Comparison

The computation of scale-space extrema in a Fully Homomorphic Encryption (FHE) setting requires comparing encrypted values, a challenging task due to the complexity of implementing comparison
functions. Integer domain comparisons can leverage Fermat's Little Theorem for exact results, while approximate methods exist for floating-point numbers. However, both approaches are resource intensive, requiring significant multiplication depth, which is costly in FHE.

A practical solution involves an interactive approach where the server sends the two numbers to the client for comparison, and the client returns an encrypted boolean result with minimal noise [?].

While this method introduces interactivity and potential algorithm exposure (as the client sees the values being compared), the risk can be mitigated by sending spurious comparison requests alongside the real ones. This approach balances practicality with security, leveraging modern communication speeds to maintain efficiency

Division

Integer division is surprisingly hard to perform using FHE primitives, it often requires the use comparison function which itself is expensive. Floating point division is a bit simpler; approximations
such as Taylor or Chebyshev series are commonly used in FHE computations . To get a good approximation we still need to use excessive multiplicative depth.

To avoid this, we store the numerator and denominator of the division separately and modify the subsequent uses of the algorithm. To use the numerator and denominator separately. For example, if $c = a < b$ and later we use it for, say, comparison, $c < d$, we modify this comparison to $a < d \cdot b$. This allows us to avoid calculating the inverse of b, which would have been expensive.

Other Functions Requiring Approximation

There are steps in SIFT which require the computation of the magnitude of vectors. This involves computing the square root of a number. Again there are polynomial approximations of this using
Chebyshev polynomials (used by openFHE). Any sort of polynomial approximation would require a lot of multiplicative depth to get a good approximate value, so we do the same that we did in comparison and delegate this calculation to the client.

Conditional Blocks

The result of a comparison is often used to decide which code block to execute. In algorithms implemented with FHE primitives, we do not have a luxury to choose. If the boolean value of an if else condition is encrypted then we need to

execute both the if-block and the else-block and mask the effects of the block by multiplying with the boolean value.

This is branchless coding and there has already been a lot of research done in converting regular algorithms to their branchless version [14, 15]. A big reason for this is that to take advantage of
the min-maxed parallelism of GPUs, one needs to reduce the number of branches. We believe this field is worth consulting before one starts implementing their algorithms.

Some optimizations that help in regular algorithms can become unnecessary when using FHE, because we execute both if and else block anyway. So early stopping strategies like breaking in loops is pointless. This allowed us to skip a very large chunk of code in SIFT. One should greedily look for such blocks to eliminate while adapting their algorithms.

Histograms and Binning

In one of the steps we need to calculate the magnitude and angle of gradients in a neighborhood. A histogram of these angles need to be calculated weighted by their magnitudes. An efficient representation of the histogram bins is critical. This kind of optimization has been explored in privacy-preserving image processing [16]. Typically, we would implement this by creating an array with $\frac{360}{numbins}$ entries. For a given angle, one would calculate the index of the array that needs to be incremented. In the FHE setting we cannot choose which element of the array to increment, we need to update every element but multiplied with a mask so that only the correct element
is incremented. The mask is generated by comparing the candidate element with the limits of each of the bins.
$a_i < \theta < a_{i+1}$. for all i. This means that indices in FHE space are one-hot vectors.

Before we start calculating the histogram, we need to calculate the angles of the gradient according to the following equation.

$\theta = arctan(\frac{dy}{dx})$

We have the values for dx and dy but using approximations to calculate inverse of dx and arctan is just too expensive, so we modify the condition we use to calculate the mask.
tan(ai) · dx < dy < tan(bi) · dx
Now we have the bins in terms of multiplications and additions. There is still a tan to calculate but that is over an unencrypted value, so can do it regularly. Would a simple compiler pass of the

Finding max in an array:

In the step of orientation calculation, one needs to find the maximum element in the histogram. The usual method of doing this would be maintaining a running maximum of the elements and comparing that running maximum with subsequent elements. Update the running maximum according to the boolean result of the comparison.

```
max = 0
for element in array:
    b = element > max
    max = b * element + (1 - b) * max
```

If we analyze the multiplicative depth of this algorithm, it would be O(N), where N is size of array, ignoring the requirement for comparison.

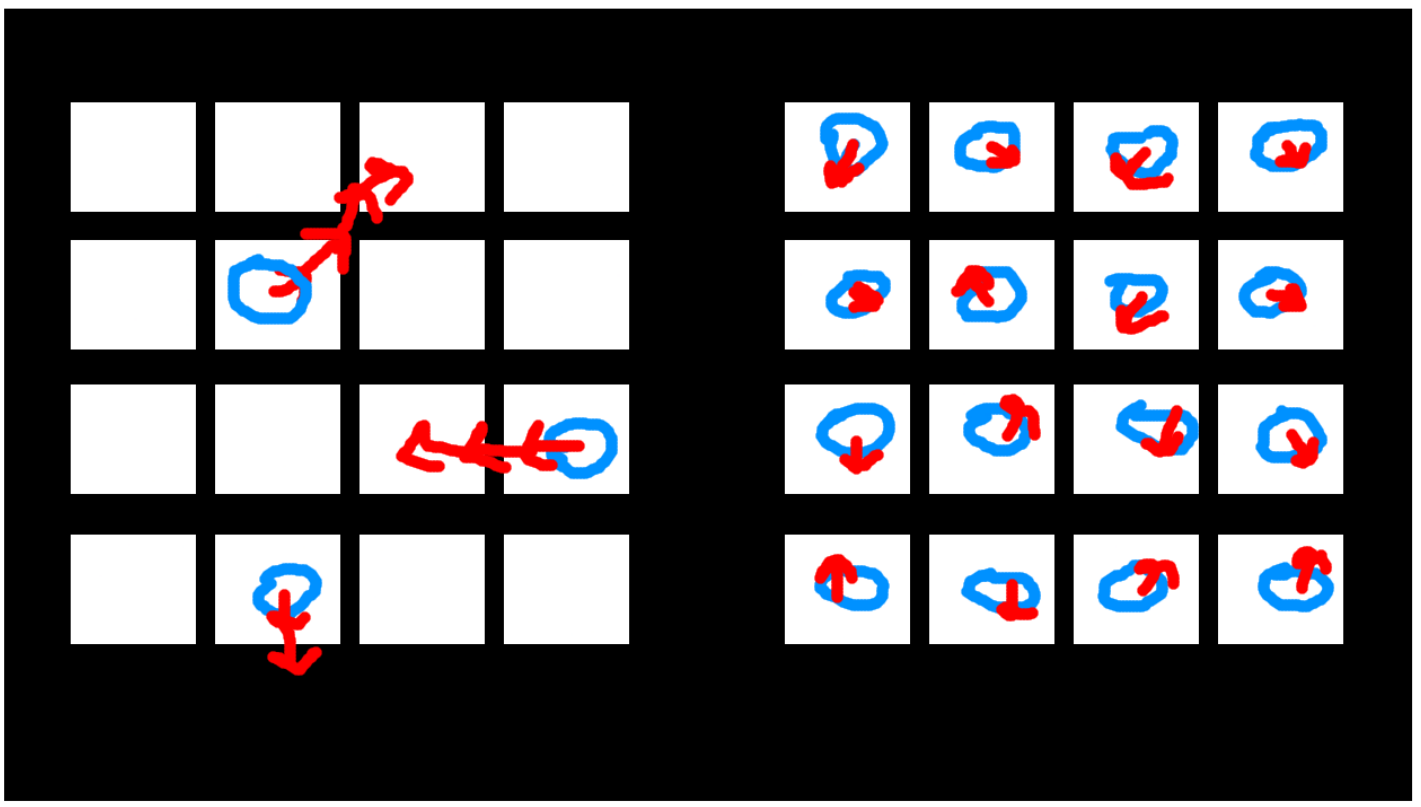
Since length of arrays can be large, we instead use a different way of calculating max, where we compare adjacent elements of the array and remove the smaller values. This would cut down the array size in half. We perform this step till only one element is left. The multiplicative depth of this algorithm ignoring comparison is O(logN).

```
def max(a, b):
    cond = a > b
    return cond * a + (1-cond) * b

def vecmax(ls):
    l = len(ls)
    if l == 1:
            return ls[0]
    elif l == 2:
            return max(ls[0], ls[1])
    else:
            return max(vecmax(ls[:l//2]), vecmax(ls[l//2:]))
```

Calculating coordinates of subpixel extrema

In SIFT we need to calculate the subpixel extrema of a neighbour. A function is fit on the values of pixels in that neighbourhood and hessian and a method similar to gradient ascent is used to calculate the maxima point. Doing it for every pixel neighbourhood is however expensive. In standard implementation of SIFT and optimization is done by checking if the current pixel value is the maximum among it's 27 neighbours. If it is then it becomes a candidate key point pixel and subpixel maximum is calculated starting from this pixel. This allows us to skip the expensive gradient ascent computation in the neighbouring pixels.



In FHE, we don't have the luxury of skipping computation based on a simpler condition, therefore we need to perform the subpixel maximum calculation anyway. However, this actually opens up a possibility to optimize. The gradient ascent now doesn't need to go beyond the current pixel's influence to determine whether to remove or keep the current pixel. We can get away with using only one gradient ascent step. This fact is illustrated in the figure above.

Deferred Computation

In order to delegate expensive operations, such as comparison and square root, to the client in a non-interactive manner, we propose a novel method of deferring computation. In this method, the server sends a single response to the client. The client can then perform the requisite computation to extract the result from the response.

How It Works

The server assumes it does not know the boolean value and treats it as a variable. As operations are performed on this boolean variable, a function is constructed with the variable as a parameter. The server sends the following to the client:

- The comparison required to compute the boolean value.

- A function that uses this boolean value as a parameter.

The client performs the necessary comparison and substitutes the resulting values into the function to compute the final result.

Construction of the Function

To construct this function, a dynamic computation graph is created, which tracks how the comparison results are manipulated. This graph contains comparison nodes and represents the structure of the computation. Post-processing may be applied to simplify the graph and produce a more compact function

Example

Consider the following expression:

$$a = (x > y) \cdot c + (z > w) \cdot d + (y \geq x) \cdot e$$

The server will return to the client:

$$f(c1, c2) = c1 \cdot (c - e) + c2 \cdot d + e$$

where:

$$c1 = (x > y),\ c2 = (z > w)$$

Downside of the Approach

One potential downside of this approach is that, in the worst case, the client could extract the exact algorithm used by the server. For example, if the entire algorithm depends on a single initial
comparison, the computation graph will store all the subsequent computations. Thus, the complexity of the function depends heavily on the algorithm itself. However, in most cases, the resulting function is likely to be simpler.

Thoughts on the FHE scene

Through this exercise we see that there is an absence of a framework that can be used to implement general algorithms. While execution time might be a problem, applications such as CNNs and Neural Networks are trivial to implement given the primitives. The vast majority of the libraries that we observed either are too low level only providing few primitives or jump the gun and provide bad abstractions aiming to turn any arbitrary code into the FHE setting.

The former is not friendly to beginners who are trying to use the technology but don’t want to know the details of the field while the latter is inconsistent in how it works.

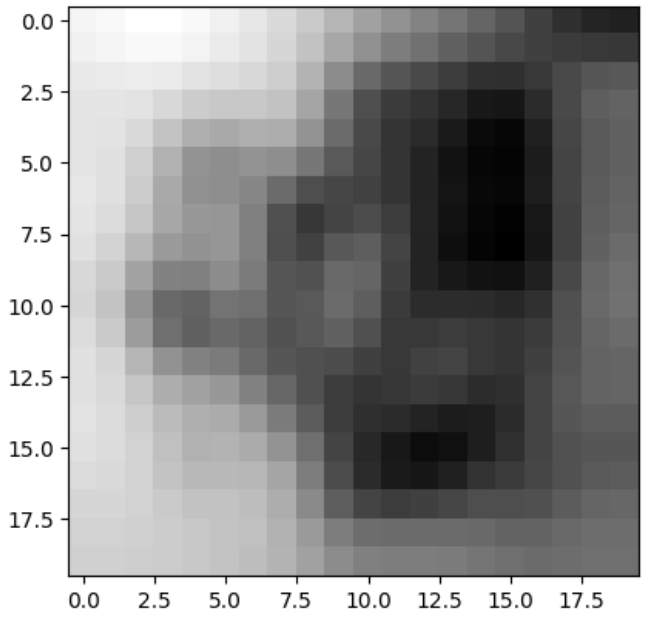
There needs to be an intermediate framework which abstracts what can trivially be abstracted away but still make it explicit where the user needs to be careful, so that the user knows the tradeoffs of using a particular feature. If we want practicality, then we need to embrace
the limitations of the current FHE scene.

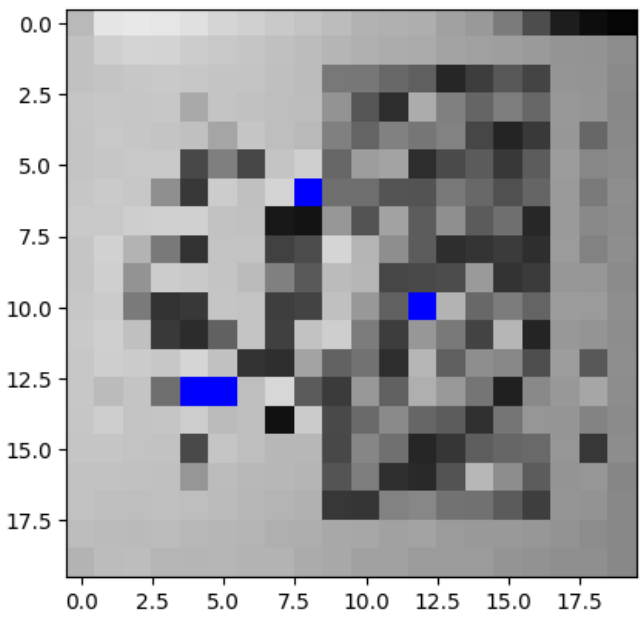
Results

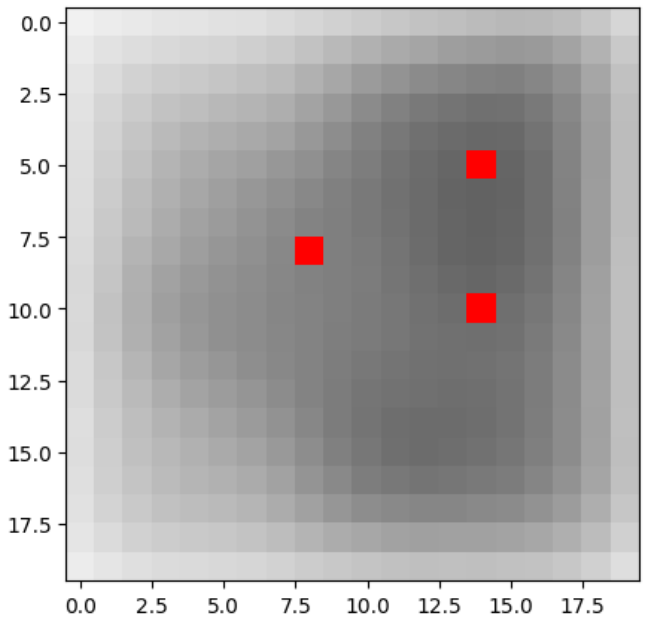
Time Analysis for a 20 × 20 grey scale image.

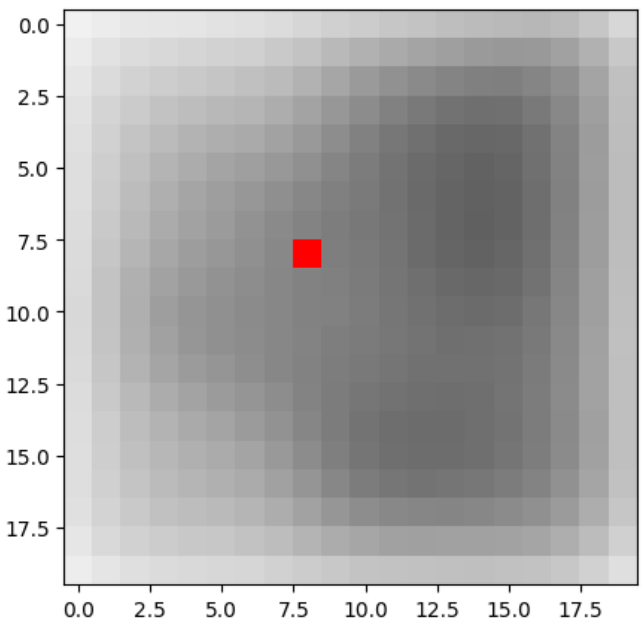
- Generating Base Image - 19.5 seconds
- Num Octaves - 0.0 seconds
- gaussian_kernels - 0.0 seconds
- Gaussian Images - 6 minute 56.7 seconds
- dog_images - 2.5 seconds
- keypoints -
- findScaleSpaceExtrema
    - Orientation - 1 keypoint takes 2 minutes

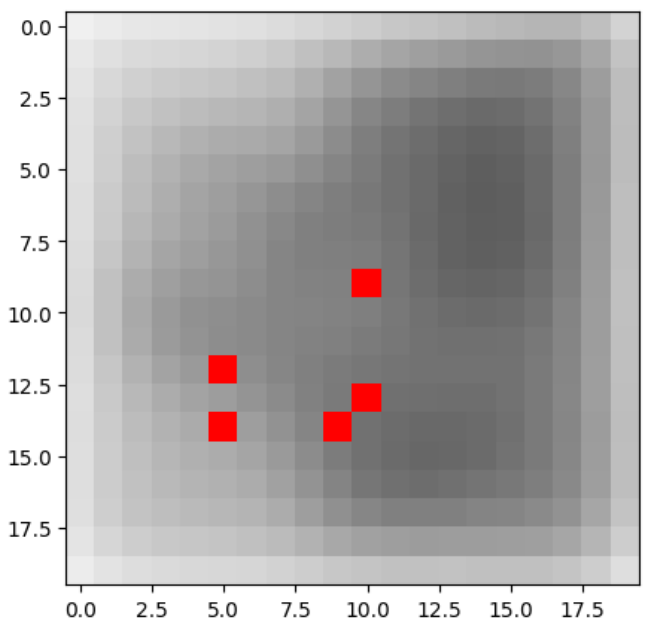
- Descriptors - 1 keypoint takes 5 minutes











Future Work

Parallelization

- Scope of optimizing convolution operations.

- Computation for each pixel is independent at the key point finding stage
- Same is the case for every keypoint while finding orientation and descriptors
- Parallelization here will significantly decrease

More efficient deferred computation

- Can delegate more items and not just comparisons for deferred computations.
- Parser to compress the computation graph generated as a result of minimize algorithm leak and computation.
- Checkout `src/secsift/defer.py` for the initial prototype of the computation graph.

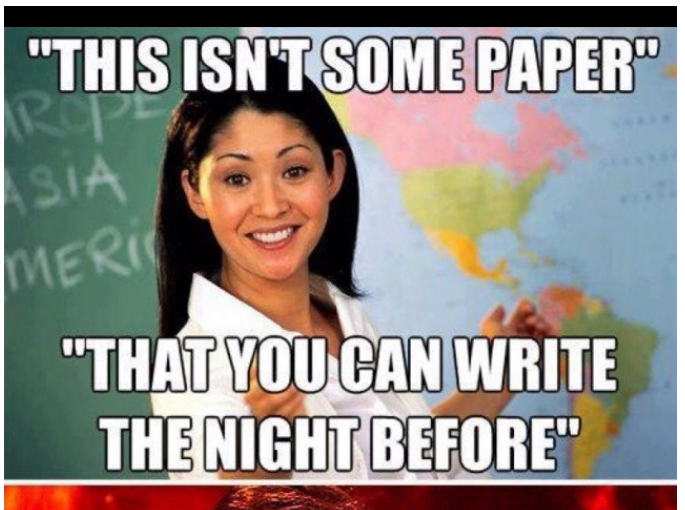
Image stitching and more image stitching algorithms

- Descriptors of different images can be compared to check/compared in the encrypted domain.
- Rotation is trivial once they are matched (only 4 closest keypoint descriptors are needed to do so)

General Image Processing in FHE

- This opens a pandora's box, any image processing algorithm can be moved to the encryption domain.
- Solves the problem of remote computation mentioned in the problem statement.

Us realising the true complexity of SIFT
(We just wanted an easy project)



References

[1] T. Lindeberg, Scale Invariant Feature Transform, 05 2012, vol. 7.
[2] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" Proceedings of the 3rd ACM Workshop on Cloud Computing Security, pp. 113–124, 2010.
[3] S. Chhabra and et al., "A survey on fully homomorphic encryption: An engineering perspective," IEEE Access, vol. 8, pp. 217 050–217 073, 2020.
[4] Microsoft, "Seal (simple encrypted arithmetic library)," 2017. [Online]. Available: https://github.com/microsoft/SEAL
[5] S. Halevi and V. Shoup, "Helib," 2014. [Online]. Available: https://github.com/homenc/HElib
[6] T. Contributors, "Tenseal: A library for encrypted tensor computation," 2020. [Online]. Available:

https://github.com/OpenMined/TenSEAL
[7] C.-Y. Hsu, C.-S. Lu, and S.-C. Pei, “Image feature extraction in encrypted domain with privacy-preserving sift,” IEEE transactions on image processing : a publication of the IEEE Signal
Processing Society, vol. 21, pp. 4593–607, 06 2012.
[8] X. Liu, X. Zhao, Z. Xia, Q. Feng, P. Yu, and J. Weng, “Secure outsourced sift: Accurate and efficient privacy-preserving image sift feature extraction,” IEEE Transactions on Image Processing,
vol. 32, pp. 4635–4648, 2023.
[9] I. Rey Otero and M. Delbracio, “Anatomy of the SIFT Method,” Image Processing On Line, vol. 4, pp. 370–396, 2014, https://doi.org/10.5201/ipol.2014.82.
[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in Advances in Cryptology – ASIACRYPT 2017, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.
[11] W. Fu, R. Lin, and D. Inge, “Fully homomorphic image processing,” 10 2018.
[12] J. H. Cheon, A. Kim et al., “Comparison algorithms for homomorphic encryption,” Cryptology ePrint Archive, vol. 2017, p. 1164, 2017. [Online]. Available:
https://eprint.iacr.org/2017/1164
[13] D. Bootland, I. Iliashenko, and C. Martindale, “Optimal polynomial approximations for modular exponentiation,” Cryptology ePrint Archive, vol. 2020, p. 1213, 2020.
[14] G. S. Brodal, “Cache-oblivious algorithms and data structures,” Lecture Notes in Computer Science, vol. 2497, pp. 268–286, 2001.
[15] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, A. Ravi, and J. McMahan, “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy,” International
Conference on Machine Learning, pp. 201–210, 2016.
[16] H. Qin and et al., “Privacy-preserving sift: A secure framework for image feature extraction,” IEEE Transactions on Information Forensics and Security, vol. 13, no. 8, pp. 2068–2084, 2018.
[17] M. Abadi and et al., “Tensorflow: A system for large-scale machine learning,” Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp.
265–283, 2016.
[18] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. Fitzek, and N. Aaraj, “Survey on fully homomorphic encryption, theory, and applications,” Cryptology ePrint Archive, Paper 2022/1602,
2022. [Online]. Available:
https://eprint.iacr.org/2022/1602
[19] O. D. Team, “Openfhe library,” 2021. [Online]. Available:

https://github.com/openfheorg/openfhe-developmen