USC Viterbi
School of Engineering

Parallel and Distributed Computing Final Project Report
EE 451/CSCI 452

CUDA Parallelization of the SIFT Image Classification Algorithm

University of Southern California

Charles Bennett, Alexander Valente, Justin Kleist
Team 16

Date of Report: 04/28/2019

3710 McClintock Ave, RTH Suite 403 · Los Angeles, CA 90089-1456
www.cs.usc.edu · Email: kleist@usc.edu · valentea@usc.edu ·

Abstract

This paper describes our methods and thought process behind using the USC HPC to implement a Block Distributed Sift Algorithm. By using CUDA we paralyzed individual steps in the algorithm, primarily applying the Gaussian Filter on each octave of the given input image, using cyclic distribution to find the Taylor approximation of each octave, and take the Laplacian of this to find the local extrema of the picture. After rotating the orientation of the images and filtering the local extrema to only include the points of interest. Then use a paralyzed process to generate a unique fingerprint for the photo to use for comparison for image searching.

https://minghsiehee.usc.edu · Email: · valentea@usc.edu · kleist@usc.edu · cfbennet@usc.edu

Table of Contents

Introduction

Our team members, Charles Bennett, Alexander Valente, and Justin Kleist used the knowledge gained over the semester of taking EE 451 to create a paralyzed version of the Scale-Invariant Feature Transformation Image (SIFT) Comparison Algorithm for our final project. The SIFT We implemented this on the USC's High-Performance Computing Center using C++ using CUDA for the paralyzation on the steps that required graphic manipulation.

We originally were going to use PThread to paralyze this, but upon further review of the process we decided that using Nvidia's CUDA paralyzation would be better. Since our project is heavily based around image comparison and graphical manipulation, we thought it would be better to use CUDA because it runs the parallel segments of code on GPUs which are better built for image and graphic processing.

To run the program, the user would input an image and then would be returned images that are within a specified similarity to the original image. First, we hope to have it function with only images but after that we want to be able use object detection to recognize objects in an image and then search a database of images and return images that contain the same object. Unfortunately with the given time constraints we did not have time to implement object detection or being able to search/compare with a remote pre-existing database.

SIFT Algorithm Steps

The SIFT Algorithm can be easily be broken down to six steps. First we construct a Scale Space of the input image. To do this we must implement the Gaussian Filter using what fundamentally a parallelized matrix multiplication. After we have created our scaled octaves on the images that have been blurred by the Gaussian filter, we must subtract the successive gaussian filtered images of different octaves and then get the Taylor approximation of this series. Using this we find the Laplacian of Gaussian filters to get the extrema in the input image. We then must filter those extrema to only the points of interest. With those points of interest we must assign the orientation to make the feature orientation invariant. Lastly, we take the remaining points of interest that have been modified to be orientation invariant, and use them to create a histogram with a unique dense fingerprint that is used for comparison. Then return all images that are with in a specified similarity of the input image.

1- Scale Space Construction

In this step, we take the given input image and apply a Gaussian Filter to it multiple times (we chose to apply it 5 times sequentially) to create a single octave (Figure 1). Applying the Gaussian Filter creates a slight blur on the image, to increase the intensity of the blur (or convolution) on the image we must change the standard deviations of the filter. In order to increase the

standard deviation of the kernel we must also increase the size of the kernel. In order to capture an adequate amount of data from the gaussian curve, the kernel must have a size of at least (3*standard deviation)^2. Additionally, by successively increasing the size of our kernel with each octave, we increase the likelihood that an edge point will be included a given index of a given octave that is not present in another.

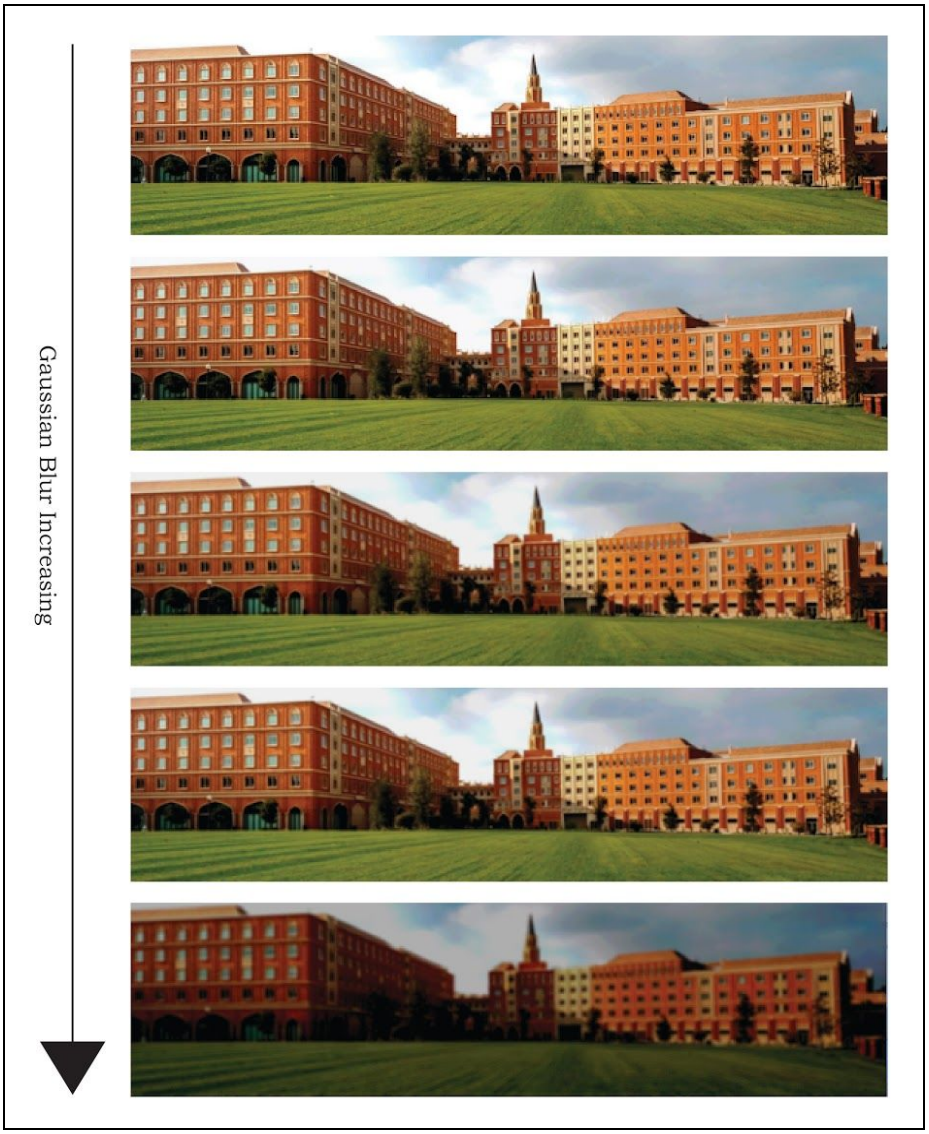


Figure 1

We do this with CUDA parallelization using two main equations (Figures 2 and 3) This creates multiple octaves of different size, and with-in each octave are the images with the different blur intensities.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}.$$

Figure 2

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y),$$

Figure 3

Each block has side length ((Image Size/Number of Blocks) + Kernel Size). This is because in order to do the convolution operation there needs to be data redundancies between blocks of size ((2*kernel size * (Image Size/Number of Blocks)) - kernel size^2).

K	Global Cuda Mem (us)	Sequential (us)
3	212	71514
5	214	200742
7	214	398913
9	210	660559
11	210	993265

2- Laplacian of Gaussian Approximation

When running the SIFT algorithm, one of the most intensive steps is taking the Laplacian of the Gaussian (LoG). Since calculating all those second

order derivatives is extremely computationally intensive. To save runtime we take a bit of a difference approach that gives a very close approximation of the LoG (Figure 4).

$$D(x,y,\sigma) = (G(x,y,k\sigma) - G(x,y,\sigma)) * I(x,y)$$
$$= L(x,y,k\sigma) - L(x,y,\sigma).$$

$$\sigma\nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma}$$

$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2\nabla^2 G.$$

Figure 4

To do this we must find the Difference of Gaussian images. Once we have our multiple octaves we take all the images in a single octave and compare each image to one another. We subtract the differences between two images that are one standard deviation apart (Figure 5).

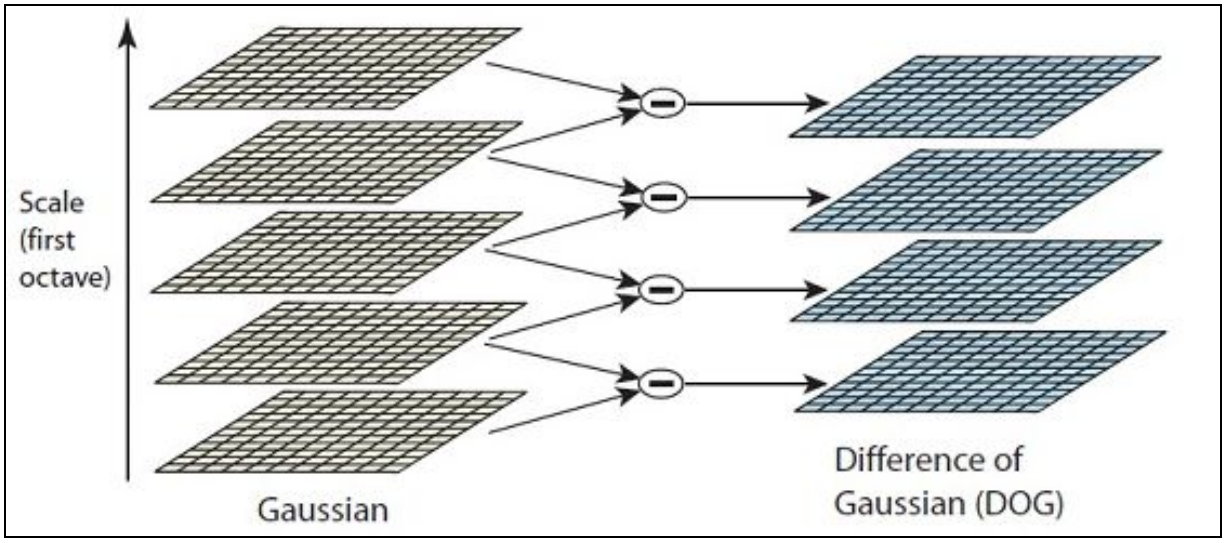


Figure 5

This creates output images that only show the differences between the image (Figure 6). Then take the Taylor Approximation of this series of output images. Then, take the Laplacian of the series to find all local extrema.

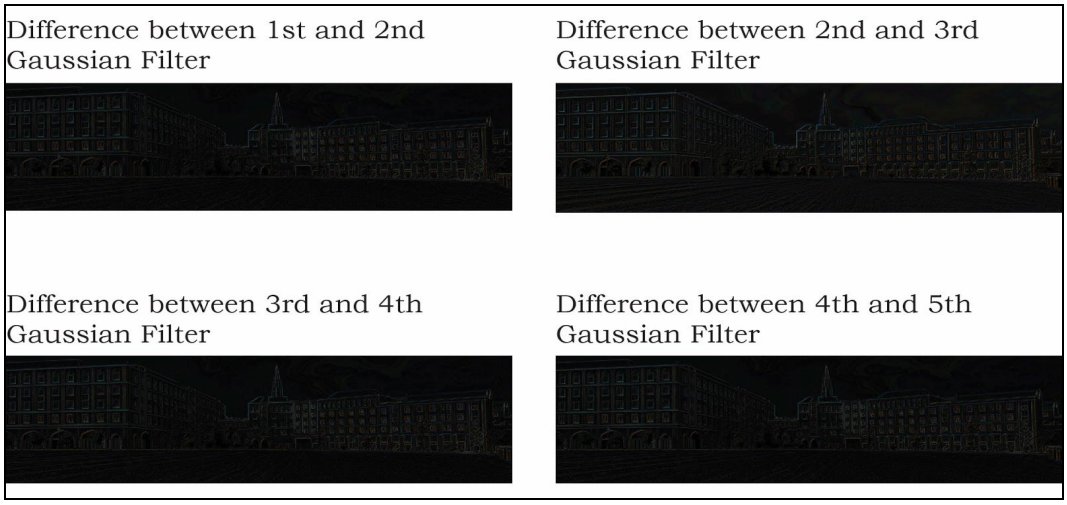
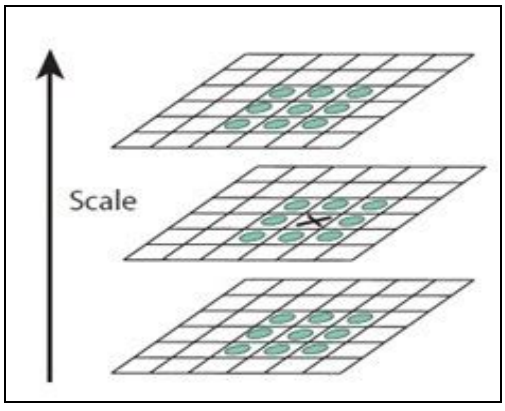


Figure 6

3- Identifying Keypoints

After applying the Gaussian Filter and taking the differences between the octaves we want to look at each pixel in the image and check if it is a local extrema. For each pixel, calculate the difference in value to each of its neighbors. We must not only check the neighboring pixels



touching the pixel we are looking at, but also the neighboring pixels of the neighboring images in the octave (Figure 7). As such we must perform a total of 26 comparisons for each image. If a pixel has a value greater than all its surrounding 26 neighbors, we mark it as a point of interest that we will look at and filter in the next step. Once we go through all the pixels in an image, we move one deeper into the octave list and repeat the process. Since we must compare pixels with the images “above and below” the pixel we are working on, we can not perform this operation on the top and bottom layer of the octave. Therefor, we just skip the first and last image in the octave. For example, we took all the pixels that we found as extrema, changed them to bright blue, and lay them over the orignal image we ran this program on (Figure 8). This returns much too many points of interest, so in the next step we must filter them and only leave the points that are most valuable to create the image fingerprint.

Figure 7

Figure 8

4- Filtering Keypoints

With the generated keypoints we must first find the Sub-Pixel Extrema. To do this run a simple taylor expansion on the points we found (Figure 9). This will find the key points locations and greatly increase our chances of finding an image that is within the predefined similarity.

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

Figure 9

After this, we have two simple steps to remove all unwanted points found in the last step and leaves us with only the Keypoints that we use for the fingerprint generation.

First, we must remove all low and high contrast points. For each of the extrema found, simply compare its value with a specified value. This is one way to change the specified similarity that images are compared to. If the pixel value is less than the value we compare it to, then it is rejected and we must continue to the next point. The higher that comparison value is the more points will be accepted. The more points that are accepted give a much looser depiction of the image, and the inverse with a lower value. With a loose fingerprint to compare with, there is more room for error and therefor, a higher variance in the images that may be returned.

Next, it is important to filter edges and corners in the picture. For many images, edges, such as along walls, are initially identified as keypoints. But many images have edges, and for that reason it creates noise that must be

filtered out before generating the image fingerprint. But when implementing this step we must make sure to include corners because they are known to be valuable for image comparison. To do this we implement a variation of the Harris Corner Detector. For each point of interest we must create two perpendicular gradients (Figure 10).

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

Figure 10

Then, based on the surrounding image there are only 3 possabilities. First, if both gradients are small the pixel is most likely on a surface with little to no variance and as such is excluded from being a final keypoint. Second, if only one gradient is large (one big gradient [perpendicular to the edge] and the other small [along the edge]), then the point most likely is on an edge and must be filtered out for the reasons above. The last possibility is that both gradients are large, because they both run parallel to edges along a corner, and as such this is a valuable point and we want to keep it.

5- Keypoint orientation

Now that we have truly filtered out all the extrema points, we are only left with points of interest that we will directly use to generate the fingerprint. But since we only found the Laplacian of Gaussian approximation, by finding the Difference in Gaussian, the points we are left with are purely orientation dependent. That is to that if we have two identical images, one rotated 90 degrees, this algorithm would not see them as equal because “UP” on one image would be “LEFT/RIGHT” on the other.

To do this, we must calculate the magnitude and direction of the gradients around each keypoint at all 360 degrees (Figure 11 bellow).

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

After finding all 360 gradients, we construct a simple histogram with 36 bins each with a size of 10 degrees (Figure 12). This tells us how often and prominent a gradient is at any point, and with the bin that has the highest value is the orientation that the pixel is generated. Not only this, but if there are multiple bins that have at least 80% of the size that the largest bin has we add another keypoint. This new key point has both the exact same location and scale that the original does, but the new keypoint will be assigned the orientation of the corresponding bin.

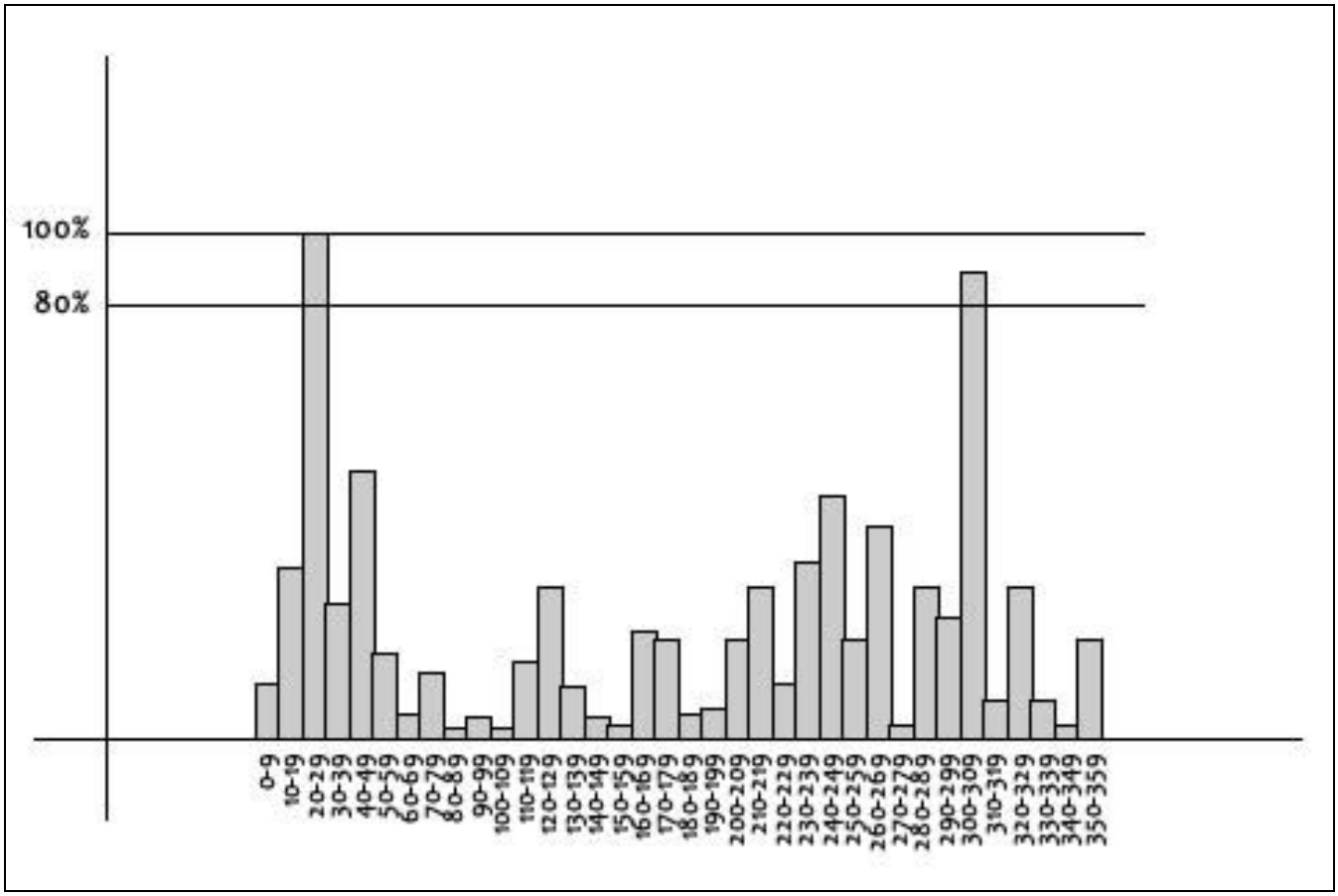


Figure 12

As you can see in our example above, in Figure 12, the bin that is most prominent is Bin 3. Therefore the pixel that corresponds to this gets assigned Orientation Three (20-29 degrees). But you can also see that Bin 31 has over 80% of the size that Bin 3 does. As such we must add a new keypoint. The new keypoint we make has all the same values as the original keypoint that we were working with, only that its orientation will be assigned differently. So after looking at the orientation of the point in question, we exit this section with not one, but two keypoints that are identical, bar the fact that one has on Orientation 3 and the other on Orientation of 31.

6- Generate SIFT Image Features

Now that we have all the keypoint, and they have all been assigned an orientation, we take these values and generate a 128 number identifying vector for each keypoint. The vector generated for each keypoint is called a feature. We store all the features that correspond with the current image, and this becomes our unique image fingerprint.

To do this, we take a single key point and view a 16x16 grid around it centered around the key point, and from there it must be broken up again into a total of sixteen 4x4 segmented areas (Figure 13).

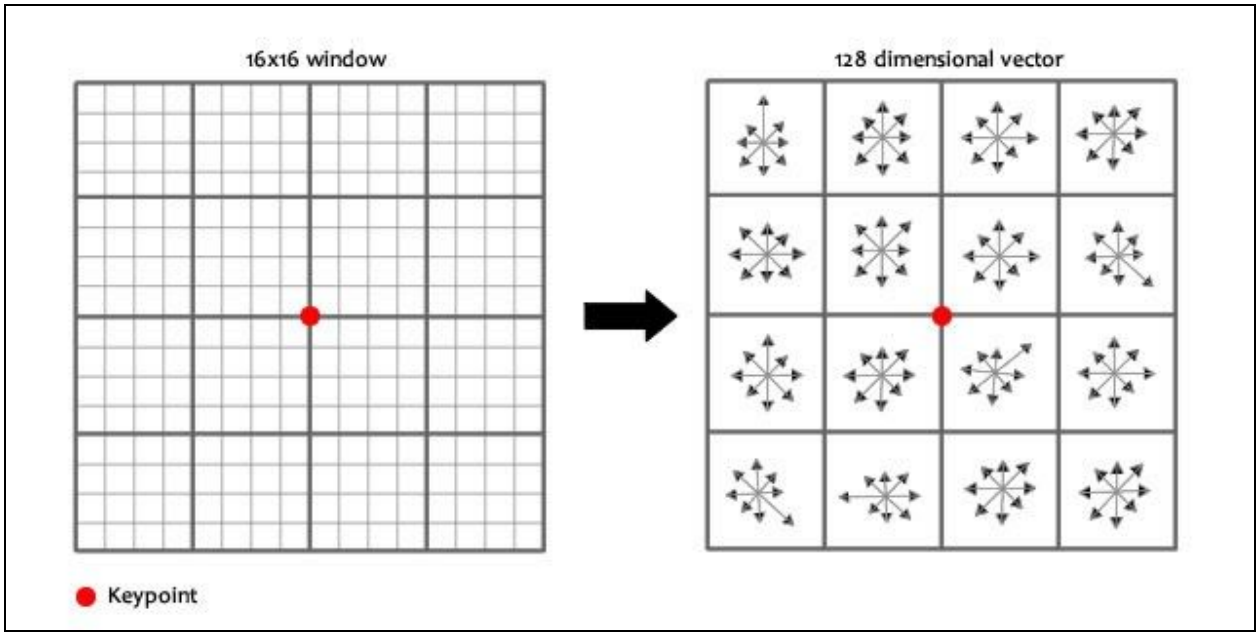


Figure 13

Then for each of these areas, we must again calculate the gradients (both the magnitude and direction) for every pixel. Same as we did when finding the

orientation of the keypoints, we take the 16 gradients (with in each 4x4 sub-segments) and put them into a histogram. But this time, we only have 8 bins each with a size of 45. Our Figure 14 is a visual representation of how we divide the histogram bins in to directional orientations. Take the bin (direction/orientation) with the largest size and that is the orientation assigned to the sub-segment (Figure 15).

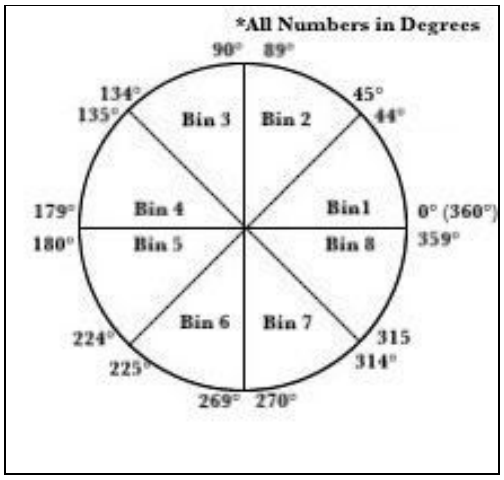


Figure 14

Doing this for all 16 pixels, you would've "compiled" 16 totally random orientations into 8 predetermined bins. You do this for all sixteen 4x4 regions. So you end up with 4x4x8 = 128 numbers. Once you have all 128 numbers, you normalize them. These 128 numbers form the feature vector that key point. And with the feature vectors you have now generated a unique fingerprint.

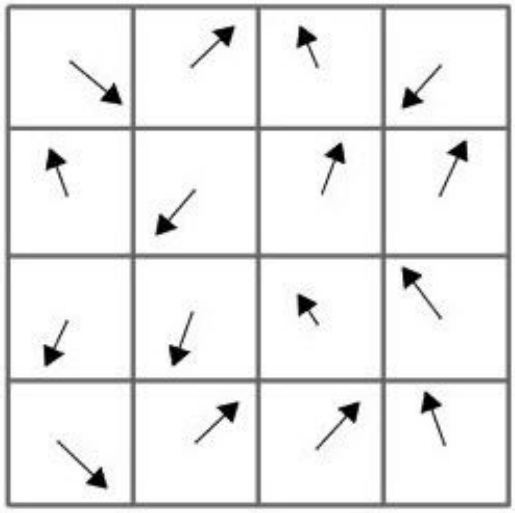


Figure 15

Code

1. ssh into USC HPC (make sure X11 forwarding is enabled)
2. setup CUDA toolchain "source /usr/usc/cuda/default/setup.sh"
3. setup OpenCV toolchain "source /usr/usc/opencv/default/setup.sh"
4. compilation "nvcc -O3 -std=c++11 [sourceFile] -I/usr/usc/opencv/default/include -lopencv_core -lopencv_highgui -lopencv_imgcodecs -lopencv_imgproc -L/usr/usc/opencv/default/lib -o [outputFileName]"
5. run "srun -n[number of nodes] --x11 --gres=gpu:1 ./[outputFileName] [imageFile]"

Conclusion and Future Work

This project demonstrates the advantages of utilizing GPU technology to process images with the SIFT algorithm. The most computationally expensive and conceptually difficult portion of this algorithm is generating the Gaussian blurred images. As such, most of the effort in this project was creating an efficient SIMT version of this computation. Although the given CUDA algorithm showed strong improvement over the serial algorithm, more work can be done to improve this. Performing a convolution on a GPU poses many challenges. The efficiency of the GPU means that the programmer does not have the same tools as with a CPU. Dynamic memory allocation in the kernel, for example, comes at a huge cost to the performance of the GPU. This means that, for constructing 5 octaves, there must have been 5 calls to the kernel from the host and thus 5 transfers of memory to the host. The only alternative would be to transfer massive amounts of memory to the thread local memory at the

offset of the program. Both of these cause massive bottlenecks. Furthermore, since images vary in size and, since the block size must be decided at run time, it is not possible to optimize thread and block size to accommodate the maximum number of warps. While GPU's offer large amounts of concurrency, they also provide less flexibility to the programmer. In future work, it would be advantageous to explore using a parallelized shift algorithm on large amounts of images. Since the images would be of varying sizes, images of similar sizes could be processed together by similar programs to achieve maximum efficiency with respect to block size. Furthermore, the bottleneck of returning to host memory could be lessened by computing multiple images at a time since memory would be transferred to the host less often. GPU's could also be pipelined so that memory transfer could overlap with computation.