UNSW

THE UNIVERSITY OF NEW SOUTH WALES

Exploring SIFT

Implementation, Application and GPU Acceleration of SIFT

COMP4121

Abstract

SIFT (scale-invariant feature transform) is a popular algorithm used for image feature extraction. The SIFT algorithm is widely applied to various computer vision algorithms such as object recognition, robotic mapping, video analysis and object tracking. We present a CPU implementation of the SIFT algorithm and demonstrate its application towards scale invariant feature matching between images. We also explore the opportunities and challenges of parallelising parts of the SIFT algorithm on a GPU using the CUDA framework. We present implementations of select components of SIFT on a GPU and describe methods and challenges when implementing for a highly parallel architecture. We compare the implementation against our reference CPU implementation to demonstrate the speedup benefits from running SIFT in a data parallel environment.

Contents

1 Introduction

With the introduction of new technologies such as self driving vehicles, robotics and even mobile phone cameras, computer vision has never been so ubiquitous. One commonly required capability in the computer vision field is that of image recognition. This involves, but is not limited to, image to image comparison, finding objects within images, scene comparison and other tasks.

To achieve this, we use a technique called Feature Extraction, which allows us to isolate and extract unique keypoints in the image. These keypoints would ideally also exist in other images of the same object. As an example, imagine a white cube object with 4 pink dots on one face. One could take two images of this cube with the pink dots facing the camera in some orientation, and then, in the second image, determine the presence of the cube by looking for the presence of the dots. If we consider these dots to be the features, then we would only need to look for these dots in future images to determine the presence of our cube.

Of course, you may like to ask, what if the object I am looking for doesn't have pink dots? This is a reasonable question. We need a generalised approach to finding keypoints such as these, with robustness to environmental and spatial differences, such as lighting or rotation. This is where SIFT comes in.

SIFT, or Scale Invariant Feature Transform, is an algorithm developed by [3], and is used for effectively and robustly extracting features from an image. It does so by considering specific properties of the feature in such a way as to make them scale and rotation invariant (the feature will be detected regardless of the size of the image, or the orientation of the image). It is also reasonably robust against lighting changes and also camera pose - the change in orientation of a camera between two captured images.

This part of the report will explain each part of this algorithm in detail and show, with our own implementation, some of the applications of such an algorithm.

2 SIFT Implementation

2.1 General Overview

The SIFT algorithm has several stages, each of which are reasonably computationally expensive. To reduce the computational cost of executing the algorithm, a cascade filtering approach is implemented, whereby the more complex algorithms are only applied to regions of interest in the original image that have passed earlier tests. We can break the algorithm up into several parts, which are executed sequentially in the following order:

a. Generate scale space and Difference of Gaussians"

b. Identify and localise keypoints

c. Orientation assignment

d. Generate keypoint descriptors

Firstly we generate a scale space of the image. We must then isolate the extreme values in the image - this is achieved through a subtractive operation applied to images in each octave. We then look for keypoints in the image. A keypoint is a location in an image that may have a strong visual feature, which means it is good for using as an identifier for the image.

As we identify these keypoints in our image, we will also filter out points that have the potential to be poor features. We then consider the keypoint and determine an orientation value for it - this orientation value is registered with that keypoint and provides the algorithm with its scale invariant quality. Next we generate our actual keypoint descriptors - these are special numerical representations of visual features that can be calculated and compared with other descriptors to provide a way of finding a particular feature in an image.

The following sections will explore these ideas in detail.

2.2 Scale Space Construction

The first step in the SIFT algorithm is the identification of locations of features in the image that can be reliably identified under differing conditions. This achieved through the creation of a scale space. A scale space refers to a sequence of images that have had increasing amounts of detail removed from them - we do this by applying a Gaussian blur to the image, making each image progressively blurrier. We will explain why the Gaussian blur is important in a moment. We introduce some new terminology to describe this: the "octave", and the "scale" - an octave describes a set of blurred images, which are all of the same size, and the scale represents the amount of blur applied to each image in that set. The image below shows a single octave of images, and the effect of increasing scale - the image on the left, an unmodified image of Lena, and the image on the far right of the octave, which has the most amount of blur applied to it.

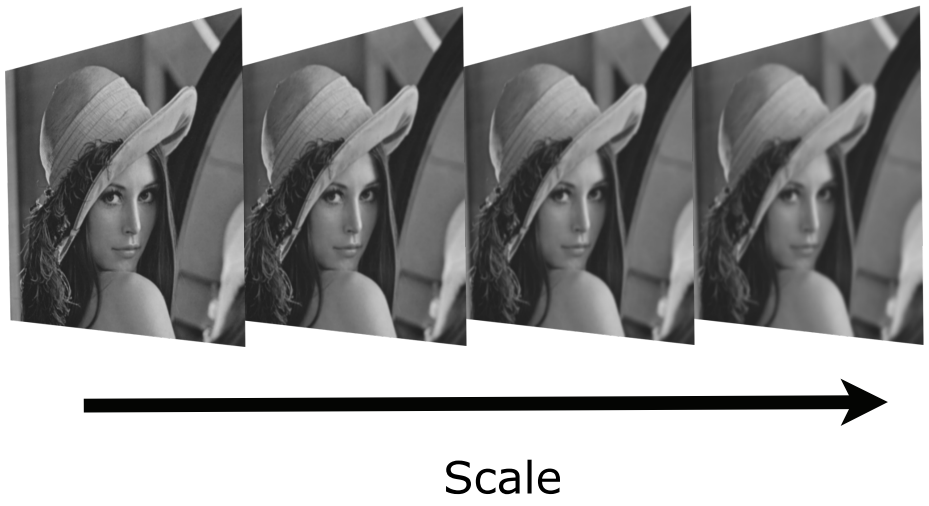

Figure 1: An image octave

The image above shows one octave, but the SIFT algorithm requires that this same operation be carried out on multiple octaves. Each increase in octave halves the image size, which can be seen in the image below.

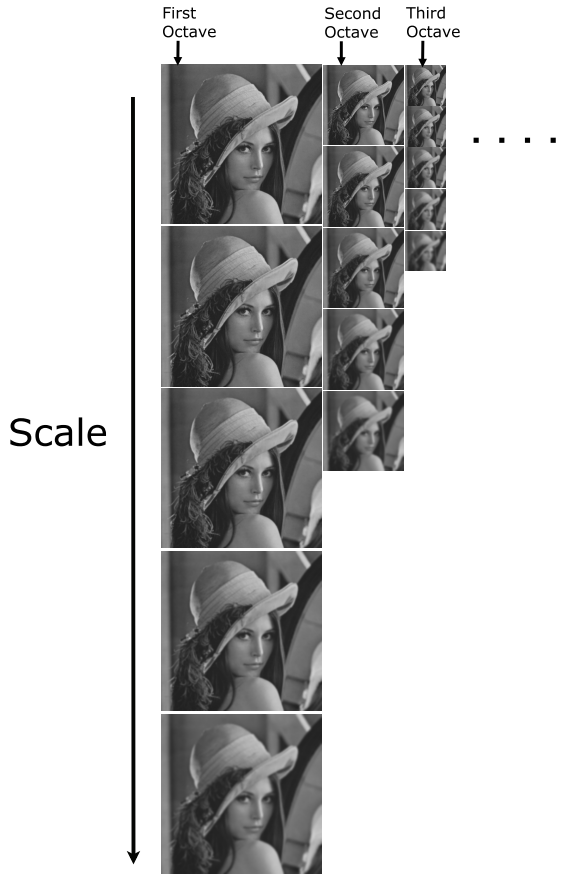

Figure 2: Octaves and scales

We use this change of scale to identify locations in the image that are reliably detected across all of these scales, which provides us with potential locations of keypoints that are invariant to scale. Previous work by Lindeberg [2] demonstrated that, under "reasonable assumptions" that the only possible scale-space kernel is the Gaussian function. We can define the scale space, L, of an image, I, as such:

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

Where I is our original image at point x, y, and L, our scale space, is a function of three variables: x and y are the coordinates in the image, for our purposes, these can be treated as the location of a pixel. The variable σ describes the value of the scale in an octave. We convolve the image, I with the Gaussian, defined as:

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

Now, to extract these stable keypoints from our scale space, we can use the previously generated images to produce a Difference of Gaussians. To explain why we do this, we must remember that we are trying to find extrema in the image. To do this properly, we would use a Laplacian of Gaussian based approach, as described by Lindeberg [2]. This involves finding the second order derivative, or the Laplacian as defined by:

$$\sigma^2 \nabla^2 G$$

We can use the Heat Diffusion equation to model the relationship between our difference of Gaussian function:

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$$

Which shows that we can compute $\nabla^2 G$ from the finite difference approximation to $\frac{\partial G}{\partial \sigma}$, using the difference of the nearby scales at $k\sigma$ and σ:

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

Where k is a real, constant multiplier.

We can now see that:

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

By using a difference of Gaussian function with constant factor, k, we are already incorporating the σ^2 scale normalisation required for the scale invariant Laplacian.

However, the Laplacian calculation is computationally expensive, but this result allows us to use a simplification:

We can approximate the Laplacian of Gaussians with the Difference of Gaussians (DoG), which we will call D. We can represent D like so:

$$\begin{aligned} D(x, y, \sigma) &= \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma) \end{aligned}$$

What this means practically, is that we can simply subtract our scale space images from one another to create the DoG, rather than computing the entire Laplacian. This is a significant optimisation. The generation of DoG images is visualised in Figure 3.

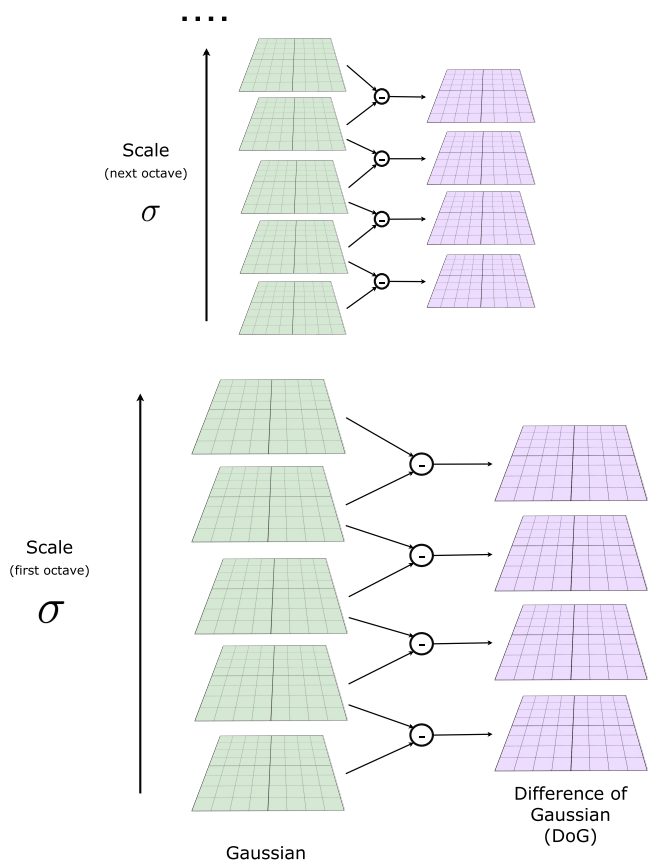

Figure 3: DoG Creation

Each image in an octave is subtracted from the previous and the DoG images are generated. This process is repeated for all octaves - we now have potential regions of interest that can be examined for maxima and minima. We have also filtered out large parts of the image from further consideration by the algorithm, thus greatly improving performance.

2.3 Keypoint Localisation

Now that we have generated our scale space, we can move through it to try and locate keypoints. We begin this process by iterating through each of the DoG images and look for maxima and minima. We do this by iterating through every pixel in each of the images and comparing that pixel to each of its neighbours, not only in the same image, but the image above and below it, referring to the DoG images that were calculated from the higher and lower scale images. This idea can be seen in Figure 4.

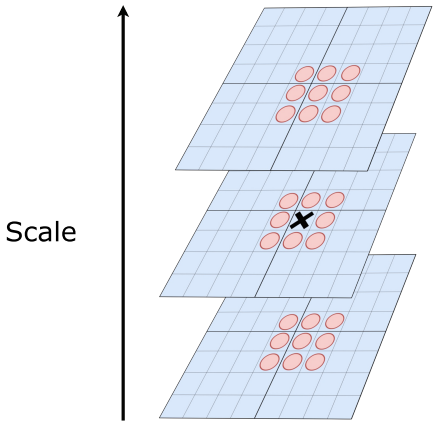

Figure 4: Extrema Comparison Neighbourhood

For each pixel checked, it will only be marked as an extrema if it has the lowest value or the highest value amongst its neighbours. Remember that the value of each pixel here is the result of the DoG subtraction we executed in the previous step of the algorithm. Because we check each pixel against the minima and maxima, only pixels with the extrema values will need to be checked against all of their neighbours - most will exit early, which reduces the amount of checks required. Likewise, the DoG at the top and bottom of the scale will not have upper or lower neighbours, so we simply skip them.

Once we have completed this comparison for the entire set of DoG images, we have a collection of keypoint candidates. We need to reject any potential keypoints if they are low contrast, or if they are located along an edge. The reasoning for removing low contrast values is that they will not make for good features, so we can pre-emptively remove them. We do this by simply comparing the absolute value of keypoint pixel intensity with a threshold value that we define, and simply remove the keypoint candidate if it falls below the threshold.

For features on an edge, we note that the Gaussian function will have a strong response on edges, even if the edge is an otherwise poor candidate for a keypoint. As such, we need to correct for this behaviour and remove candidate keypoints that lie on an edge. We do this by looking at the principal curvature of the keypoint - you can think of it as taking the gradient at a keypoint in two directions, one perpendicular to the other. We can calculate this principle curvature at a point using the 2x2 Hessian Matrix at the point of interest:

$$\boldsymbol{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

We take advantage of the fact that the eigenvalues of $\mathbf{H}$ are proportional to the principal curvatures of D. Since it is the ratio of these eigenvalues, that we are interested in, we can calculate the trace and determinant of $\mathbf{H}$ rather than their exact values, which is cheaper and easier to implement. We also introduce the two eigenvalues α and β. α is the larger of the two. Lowe gives the following equations:

$$Tr(\boldsymbol{H}) = D_{xx} + D_{yy} = \alpha + \beta$$
$$Det(\boldsymbol{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta$$

Discarding any points with a negative determinant, we add another variable, r representing $\frac{\alpha}{\beta}$. Now:

$$\frac{Tr(\boldsymbol{H})^2}{Det(\boldsymbol{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r}$$

A minimum can be found when the eigenvalues are equal, so, checking:

$$\frac{Tr(\boldsymbol{H})^2}{Det(\boldsymbol{H})} < \frac{(r+1)^2}{r}$$

Allows us to check to see if the principal curvatures are below the required threshold. This is the explanation given in the paper presented by Lowe, however, in our implementation, we simply determine the trace and determinant of **H** and check that:

$$\frac{Tr(\boldsymbol{H})^2}{Det(\boldsymbol{H})} < \text{CURVATURE\_THRESHOLD}$$

Where CURVATURE_THRESHOLD is a constant value. Lowe suggests a value of 10 for this, however, we have settled on a value of 5. The larger the value, the more tolerant of edges your algorithm will be, and obviously, the lower the value, the less tolerant.

This rejection of keypoints has provided two benefits: we have further reduced the number of locations where a keypoint may be found, and have also rejected locations that would have given a weaker keypoint, and therefore a less robust feature. At this stage, we consider any remaining keypoints to be reasonably valid descriptor locations.

2.4 Orientation Assignment

We will now calculate the orientation of the feature at each keypoint. We will look at the region surrounding each keypoint and look at the gradients associated with that area. We will then determine the aggregate magnitude of those gradients and consider this to be the main orientation of this feature. We will then describe the feature itself relative to that orientation, which gives rotation invariance. This means that the feature (and by extension, the larger image) can be rotated, and the keypoint will still be detected. There are two parts to this next procedure - we will calculate the orientation and magnitudes at each keypoint, and then generate a histogram. We create 36 bins in our histogram, representing the 360 degrees at each point. We then populate each of these bins with the magnitudes of the orientations in each of the neighbouring pixels. We use this histogram approach so that if there are keypoints that have multiple orientation magnitudes of a similar value, then a separate keypoint is created for each of the dominant orientations. For example, imagine an approximately single-axis symmetric feature in the image - we would create two keypoints for it, one facing each direction. This improves the robustness of rotational invariance.

To calculate the magnitude and orientation, we use the following formulas:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$
$$\theta(x,y) = tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

2.5 Key Point Descriptor

We will now generate unique feature descriptors for each of the keypoints. In the previous step of the process, we generated the gradient magnitudes and orientations of the regions surrounding the keypoints in our image. At each keypoint, we will take a 16x16 pixel window around each point. This 16x16 region is broken up into 4x4 blocks, and the magnitude and orientation of the gradients in each of those smaller regions is calculated. A simplified visualisation of this can be seen in Figure 5. This uses a 2x2 descriptor array computed from an 8x8 set of pixels. This image is taken from Lowe's paper, but perfectly visualises the process.

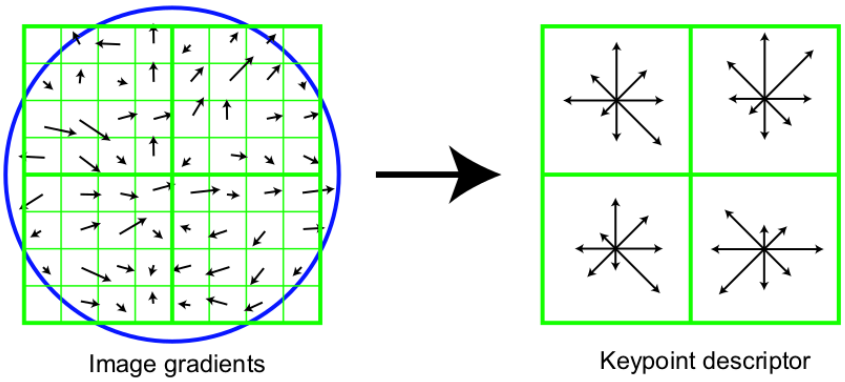

Figure 5: Key Point Descriptor gradient bins, taken from Lowe [3]

These gradient magnitudes are binned, like in the previous part of the algorithm, but this time into 8x8 bins. We also apply a Gaussian weighting to each of these gradient magnitudes and orientations, which decreases the weight of the orientation and magnitude as the point moves further away from the center of the keypoint. This is represented in the figure above by the blue circle. We do this for each of the 4x4 pixel blocks, and each of the gradients and magnitudes are binned into the 8 bins mentioned previously.

We now have 8 bins for each of the 4x4 blocks, giving us a 128 element feature vector that represents our keypoint. However, we are not yet finished. In order to deal with irregular lighting conditions, primarily non-linear illumination, we apply a threshold to the gradient values. Lowe suggests capping the values at 0.2, as this will effectively reduce the impact of large lighting changes on the gradient of the feature and preserving illumination invariance.

Now that we have completely extracted the features from our image, we can store them for later use. Figure 6 shows our Lena image with the best features identified, along with the orientation of these features. Note the pattern of orientations along edges, which we expect. (Image is on next page.)

Figure 6: Extracted Keypoint set on Lena Image

We have now generated descriptor for our keypoint, which should be able to be independently re-calculated with the same result from the same or another picture of the same subject (within a reasonable amount of change). These keypoint descriptors are stored and can be used in a variety of ways, which will be the subject of the next section of this report.

3 Applications of SIFT

As you may well imagine, being able to extract features from an image has many applications. Typically, these uses relate to the detection of objects in an image, or determining the orientation of an object in an image where its location is known. However, we will employ our implementation here to try and achieve something interesting.

Firstly, it should be noted that there is an OpenCV (a common and widely used open source computer vision library) implementation of SIFT already available - however, because we needed access to the inner workings of the algorithm for our CUDA implementation and we felt it would be improper to simply use an off the shelf implementation for our assignment, we implemented our own SIFT algorithm. We used c++ and basic OpenCV functionality to do this.

An interesting application of SIFT is as a feature extraction tool for image homography. Homography refers to the relationship between two image planes that we can infer from the extracted features of each image. This has valuable applications in image correction, object detection and even pose estimation for robotics. We will explain this further, but first, we need to start with our SIFT implementation and how we can use it for this purpose.

When we last looked at our SIFT algorithm, we had just described keypoint generation and showed the result of that on our demonstration image, Lena. In order to be able to achieve something as complicated as homography using our SIFT algorithm, we need to do a few things first.

a. Establish a way of reliably comparing keypoint descriptors and match pairs of points with similar keypoint descriptors

b. Select the Best matches

c. Determine, based on the matched pairs, a relationship between the sets of points

d. Use our homographic matrix

3.1 Comparing Keypoint Descriptors

Consider we have two images of the same object under reasonably similar lighting conditions and camera position. The object in question has a very prominent feature that is visible in both images. We would like to have a way of associating those two keypoints in both images so that we can infer some sort of relationship between them.

To do this, we will need to extend our SIFT implementation to allow us to compare our feature descriptors. To compare our keypoint descriptors from different images, we will use the Euclidean distance of the feature vectors of the keypoints like so:

$$Dist = \sqrt{\sum_{i=1}^{128}(fv1_i - fv2_i)^2}$$

Where $fv1$ is the 128 element feature vector of keypoint 1 in image 1, and $fv2$ is that of keypoint 2 in image 2.

For each point under examination in image 1, we will repeat this process, and select the keypoint in image 2 that has the lowest $Dist$ value, and consider these two points to be matched (for now). This is an expensive process, but for a reasonably small number of keypoints, is negligible in practice.

It is very possible that two keypoints are matched with this approach that are not actually related to the same feature - this is not a problem. In our implementation, we will correct this behaviour in the next section.

Note: this is not the only way to approach this problem. For larger numbers of points, it would be far faster to use a nearest neighbour search. We investigated implementing this approach using a K-D tree, but abandoned it due to time constraints.

3.2 Selecting the Best Matches

We can select the best matches in our set by sorting our matched pairs by how close they are in their respective images. For example, consider a point in image 1, at position (x_1, y_1), and a keypoint in image 2 at location (x_2, y_2). We now calculate the Euclidean distance between these two points and store that distance value along with the pair. What we are considering with this calculation is how far away the feature is in the second image compared to its location in the first image. The reasoning being that, the closer the keypoint in image 2 is to the location of keypoint 1 in the first image, the more likely they are to be the same feature.

We repeat this process for each matched pair that we have from the previous step. Once each matched pair has a distance associated with it, we sort them and choose the best n pairs, or, rather the pairs with the lowest distance. We now have fewer pairs to compare when we execute the next part of our algorithm.

3.3 Determining a relationship between matched pairs of points

Now that we have a set of matched pairs from the two images,we can try and determine if there is a relationship that exists between them. We will use a RANSAC based method to do this.

RANSAC, or **RAN**dom **SA**mple **C**onsensus is an algorithm for finding consensus in data sets. We will not delve too far into RANSAC itself, but it works in the following way: We randomly choose four pairs in the set of matched pairs and consider their gradients when compared to one another. Consider the image below:

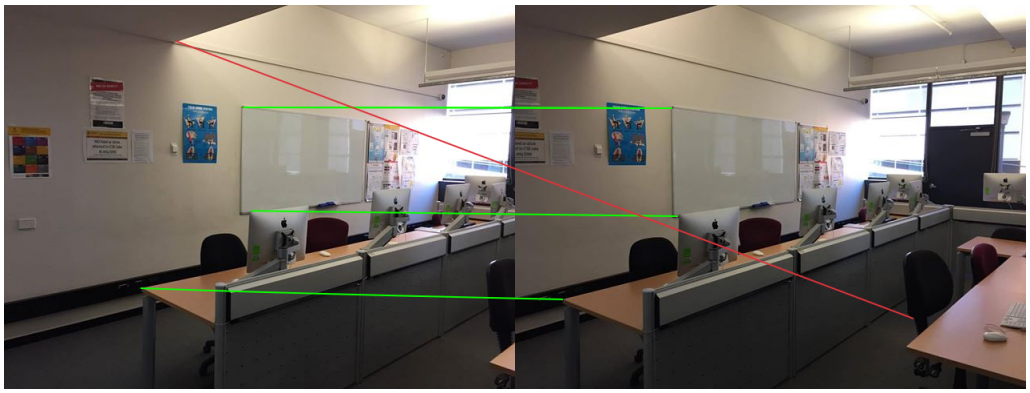

Figure 7: Simplified example of an image with matched pairs.

Notice that there are three more or less aligned pairs (highlighted in green) and one matched pair with an outlier gradient (highlighted in red). RANSAC will look at the pairs here and consider the general consensus: here we have three pairs of points who agree within a certain margin on the correct gradient, and one pair that does not agree. We take note of the agreeing pairs and the disagreeing pair and repeat the process until we can be happy enough that there is a general consensus and that the disagreeing pairs are outliers. The outlier pairs are removed from consideration. Any pairs remaining after this step should be reliable and can be used for our homography.

We can now calculate the homography of our two sets of matched pairs. Our implementation takes advantage of the OpenCV $findHomography$ function, using our points. It takes the planes associated with the points that we have, and finds the perspective transform matrix between them. We will hold onto this homographic matrix, **H** for later use. We give an example of its use below.

3.4 Using our homographic matrix

We can use our homographic matrix to de-skew images, create panoramas and for various other purposes. We will step through an example of using our SIFT implementation and the above homography calculation to de-skew an image.

We begin with two images of the ID card of one of the authors, which can be seen below. The one on the left is essentially straight on, the image on the right was taken by the same camera from a completely different angle.

Figure 8: An unskewed image of an ID card.

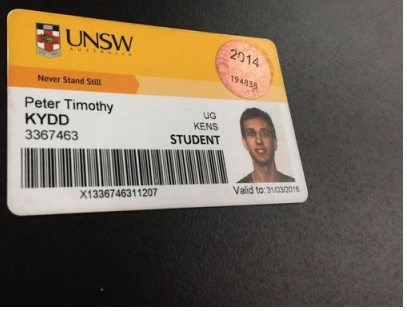

Figure 9: A skewed image of an ID card.

Note that these are two separate images, taken at different times with the same camera. No digital modifications have been made to the images.

We now run the images through our SIFT algorithm and calculate our keypoints and features. The result is shown below - you should be able to visually identify the corresponding keypoints, or at least some of them!

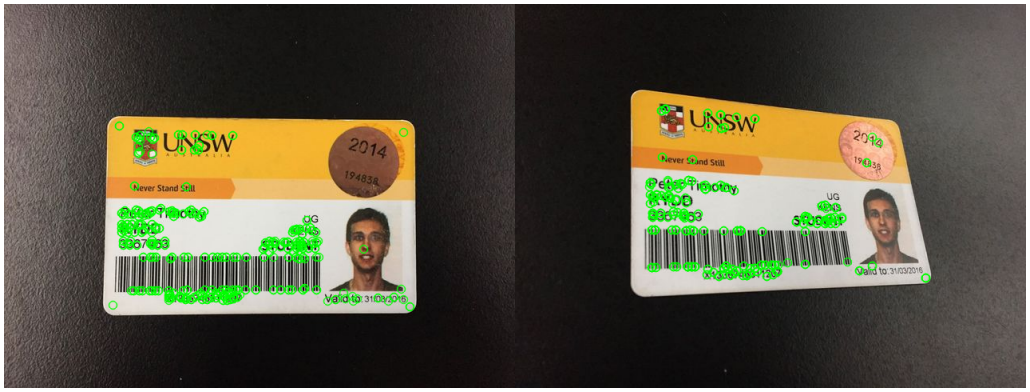

Figure 10: IDs with keypoints identified.

We will now algorithmically compute what your eyes can naturally do - identify pairs in the keypoints. We use the aforementioned method (Euclidean distance method) to determine the matches. Note that if you look closely, you can visually determine that there are many pairs that agree on an approximate gradient, and several that do not (outliers). The intention of RANSAC is to remove those outlier pairs.

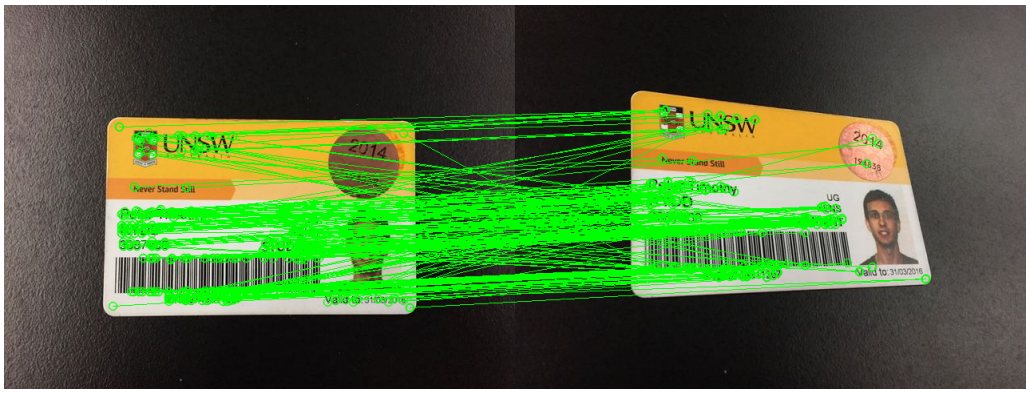

Figure 11: IDs with matched pairs.

And now, we can use our RANSAC method to compute a homographic transform and apply that to the original skewed image. The result is shown below.

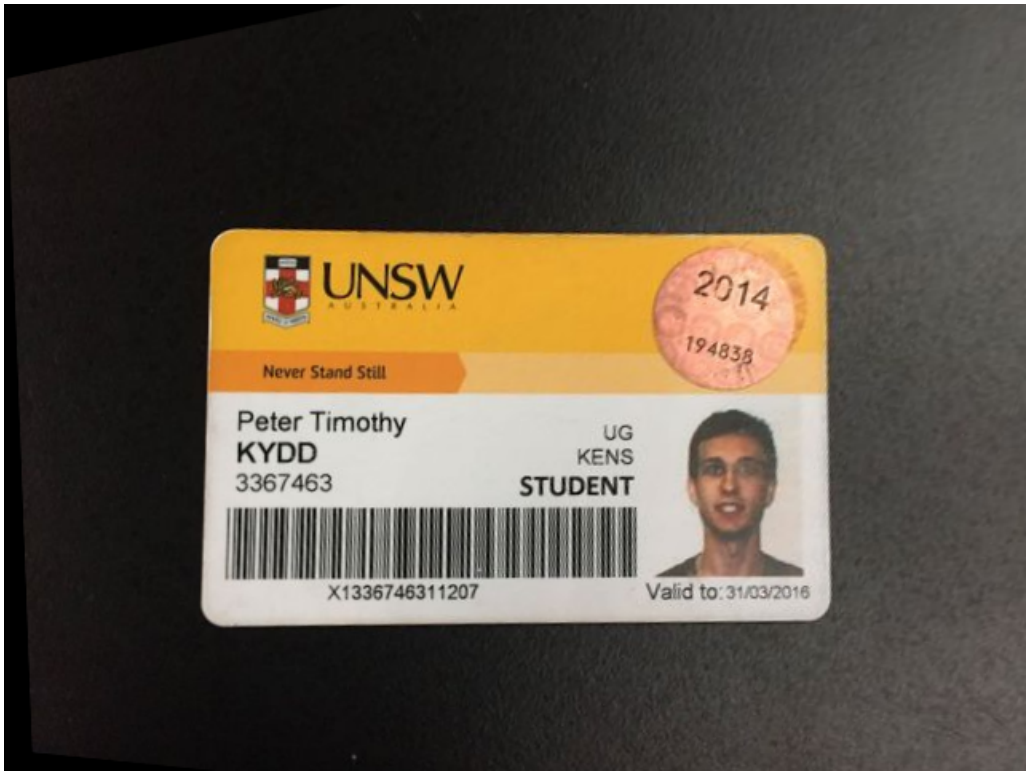

Figure 12: The formerly skewed ID with transform applied.

We should be able to see in Figure 12 that the skew present in the original image, Figure 9 has been undone. You can tell the two images apart by the diffuse illumination that is present on the right hand side of the skewed image, which is not present on the original unskewed image, Figure 8. You can also see the transformed image has a trapezoidal shape to it - the result of the deskewing.

4 Optimisations and GPU Acceleration

In this section we will explore optimisations that can be made to SIFT by implementing the key stages of the algorithm on a GPU using the NVIDIA CUDA framework. We will identify the architectural differences when programming for a GPU and an approach to modify the SIFT algorithm such that it suits the highly parallel nature of GPU based programming. SIFT is a computationally expensive algorithm due to the amount of data required to be processed. We will however identify that the underlying computations within each stage can be calculated in parallel allowing us to achieve a significant speed-up in execution time.

4.1 Background & Motivation

4.1.1 Parallel Computing

Traditional methods of improving performance in early years of computing looked towards increasing processor clock speeds and instruction throughput on single core chips. However it is no longer feasible to rely on this method as we rapidly approach the fundamental limitations of chip fabrication and transistor size. In recent years however a significant trend in computing has looked towards increasing the number of computing cores on chips as a mechanism for improving processor performance. From this trend we are introduced to parallel computing thereby allowing our programs to utilize multiple cores on a single chip as a means of improving performance [5].

4.1.2 Data parallelism

Efficient and correct computation within parallel environments remains a difficult task. In response various programming paradigms and models have been introduced that allow the programmer to effectively and safely use a multi-core environment. A programming model one can adopt to make use of the underlying multi-core hardware is data parallelism.

Data parallelism involves distributing data across various computing cores, where each processor will perform the same task of a different segment within the distributed data set [4]. Particularly advantageous to data parallelism is that by distributing our data into an appropriate number of segments, it can effectively scale to any number of processes/cores. Naturally lending well to such a model are GPUs in that its architecture can offer thousands of cores to be utilized.

4.1.3 GPGPU Programming & CUDA

General Purpose GPU (GPGPU) programming is the utilization of GPUs to execute programs that would typically be performed on a CPU. By performing computations on a GPU we introduce our computation to a highly parallelised architecture with thousands of cores available [4]. An architecture that can be used to achieve this is CUDA. The CUDA architecture is a parallel computing platform developed by NVIDIA found on their GPUs. Through CUDA, developers can access the massively threaded, multi-core architecture of the GPU for which they can develop general purpose computing applications.

To develop applications CUDA offers an extension of the C/C++ programming language where programmers can distinguish between writing code that runs on either a *host (CPU)* or *device (GPU)*. Developing device code is achieved by programming special functions called *kernels*, whereby the programmer specifies how many kernels are to be run in parallel on the GPU. The programmer specifies m parallel invocations of a kernel, also being known as *blocks*. We can further allow these blocks to be split into n *threads*, allowing parallel execution of threads within each block. To manage memory, we can transfer memory from the host to the GPU device memory. This is stored in a *global* memory region that all GPU threads can access. We can further manage memory between threads in each block, where threads can cooperatively share data through the use of *shared memory*. Shared memory can only be accessed between threads in any given block, with access times being faster than reading from the global memory region. An example illustrating the relationship between threads and blocks can be seen through a GPU vector addition

example in Figure 13 [5]. Here we present a CUDA kernel, *add*, invoked over a 4×4 block/thread configuration to add two 16 element vectors a and b. The execution of the *add* kernel is denoted by the special <<<>>> syntax used to specify the number of blocks and threads per block.

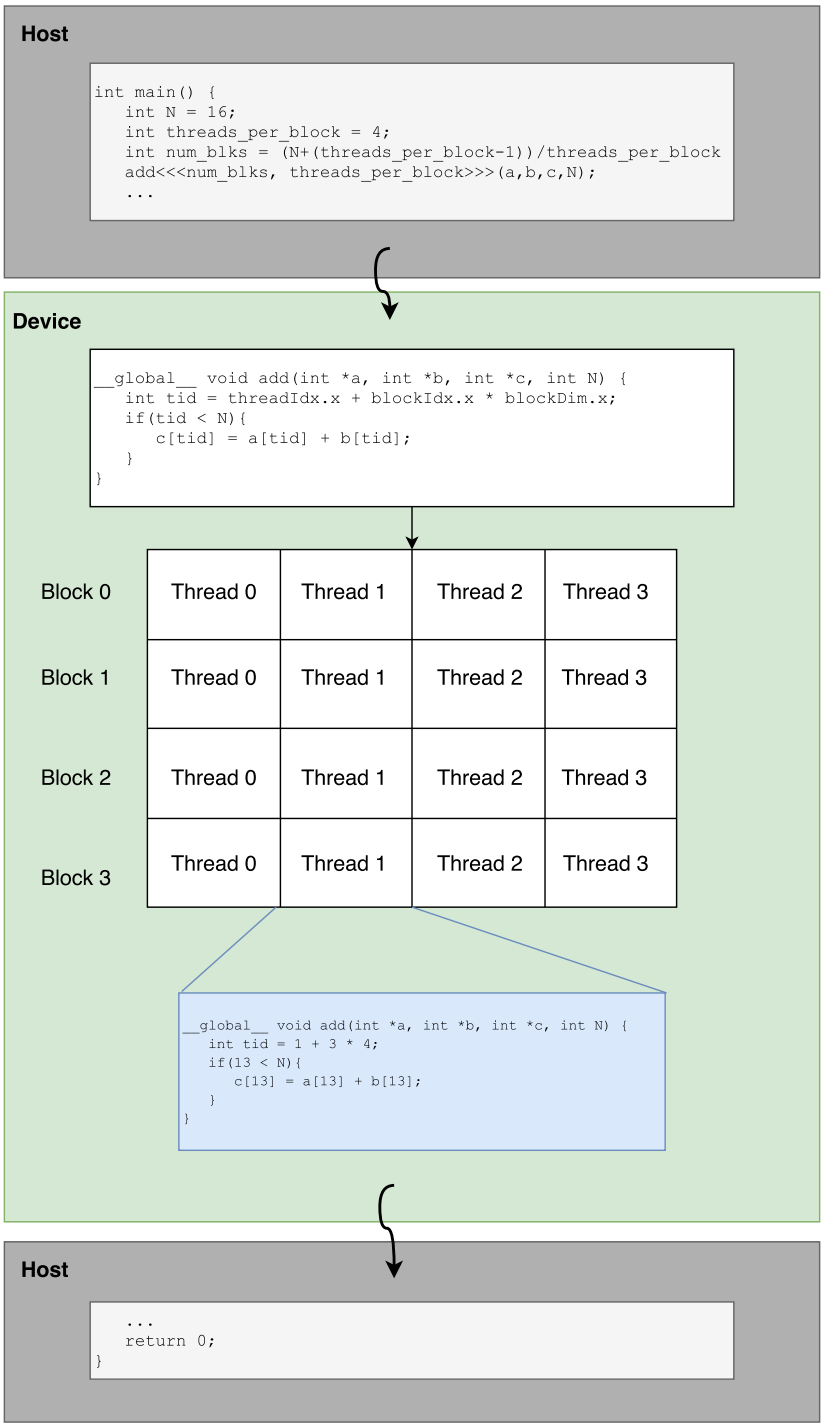

Figure 13: CUDA Vector Addition Example

Lastly, relevant to our application of SIFT on the GPU is that we can further represent a block as a 2D group of threads, illustrated in Figure 14. This allowing us to index threads in both the

x & y dimension.

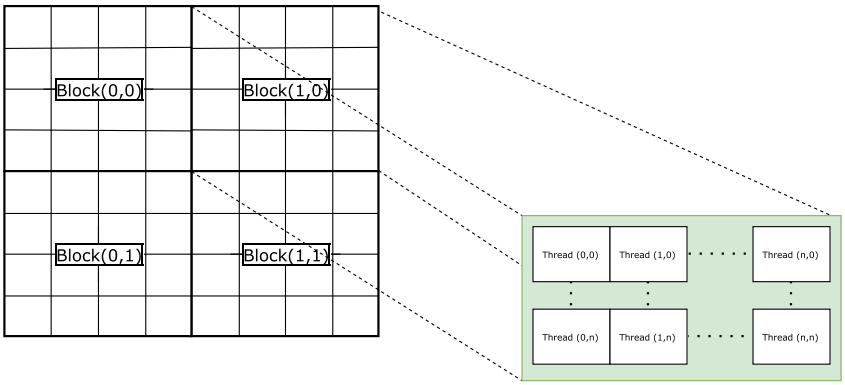

Figure 14: 2D hierarchy of blocks and threads

4.2 Scale Space Construction on the GPU

The first step of the SIFT algorithm requires us to construct a scale space for our given image. This process involves two parts, a repeated Gaussian convolution for n scales over m octaves followed by a difference calculation between our Gaussian images within each octave. The large number of arithmetic operations needed to be performed in conjunction with a significant amount of data to compute makes this problem suitable to adapt to a data parallel focused approach.

4.2.1 Gaussian Convolution

The first stage of the scale space construction step is the repeated application of a 2D Gaussian stencil over the image at various octaves (image sizes). A stencil computation involves repeatedly updating the values of a multi-dimensional grid of points using the values at a set of neighbouring points. These neighbouring points are defined by a stencil or a convolution kernel. An illustration of applying a 5×5 stencil can be seen in Figure 15. To perform a Gaussian blur we represent our Gaussian equation over a 5×5 point spread function where the values in the function correspond to the values in a Gaussian curve. We iterate the stencil over each pixel in the image, multiplying the values in the stencil with a neighbourhood of surrounding pixels, summing the result and storing it in the destination pixel. A complete application of the stencil across all pixels results in a Gaussian blurred image. An example of a Gaussian stencil initialised with $\sigma = 1.0$ is illustrated in Figure 16.

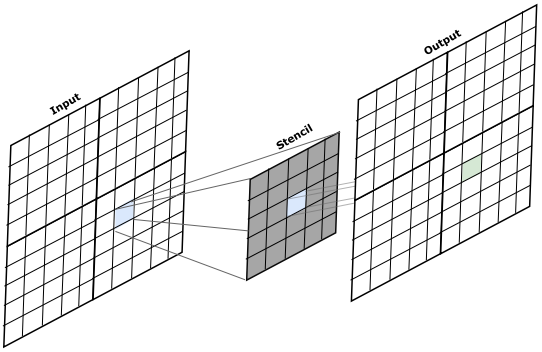

Figure 15: 5x5 Stencil Convolution Example

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 16: 5x5 Gaussian Stencil with $\sigma = 1.0$

Given the above example we can see that an application of a Gaussian stencil isn't a cheap operation where a single pixel calculation requires N^2 calculations, N being the square dimensions

of the stencil. In addition, the cost of producing a Gaussian blurred image scales with the size of the image, being the number of pixels we need to compute. An immediate optimisation we can make when computing the convolution is by recognising the Gaussian stencil is symmetrical and linearly separable [1]. Mathematically, a linearly separable $M \times N$ kernel can be decomposed in two matrices, being $M \times 1$ and $N \times 1$. Thus we can split our $N \times N$ Gaussian stencil into two separate $N \times 1$ stencils. We multiply a given pixel value and its neighbourhood with the values in each 1D stencil and take the sum. This reduces the number of computations for any given pixel down to $2N$ calculations. We can take the 5×5 Gaussian stencil illustrated in Figure 16 and present it in its decomposed form below:

$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} = \frac{1}{256} \cdot \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

4.2.2 Gaussian Convolution on CUDA

Taking the above principles we look to implementing a Gaussian convolution on the CUDA architecture. We can frame the stencil problem space through a data parallel based approach. We find that every pixel requires the same computation, each of which has the same stencil applied to its neighbourhood. Thus it is desirable to geometrically segment our image data-space into a into chunks that can be processed by different threads in parallel. To implement this we can naturally extend our 2D hierarchy of blocks and threads (illustrated in Figure 14) to match the image space such that each thread will perform a Gaussian stencil computation for a single pixel. A simplified example of this is shown below in Figure 17. We divide our image into blocks, each block containing a 2D set of threads. A thread computes the image pixel it is responsible for by calculating its position in the entire thread/block space. In the example below, the thread with (x, y) coordinates $(2, 2)$, relative to block $(1, 0)$, is responsible for computing image pixel with coordinates:

$$\begin{aligned} &(\text{blockId.x} \times \text{blockDim.x} + \text{threadIdx.x}, \text{blockId.y} \times \text{blockDim.y} + \text{threadIdx.y}) \\ &= (1 * 4 + 2, 0 * 4 + 2) \\ &= (6, 2) \end{aligned}$$

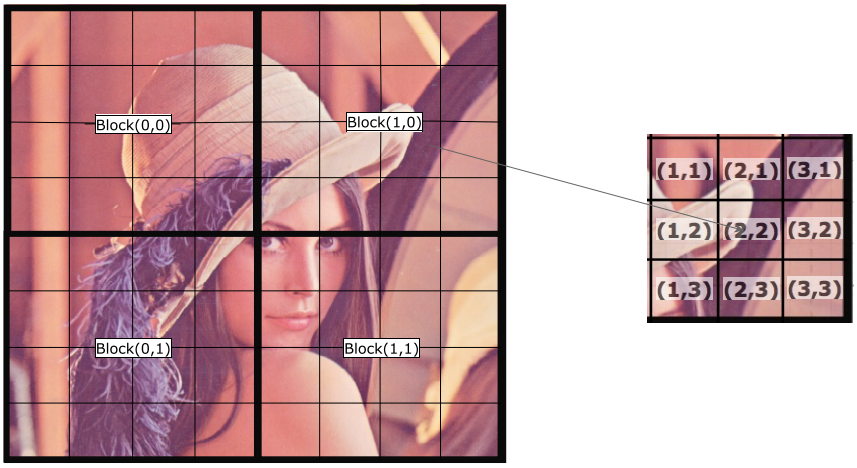

Figure 17: Blocks and Threads over image space

Taking this approach we implement two CUDA kernels, being a horizontal 1D convolution followed by a vertical 1D convolution to produce a Gaussian blurred image. To avoid the overhead of continually copying the result of the convolutions back and forth between host and device

memory, we take the result of the previous convolution (still in device memory) and pipeline it to another horizontal and vertical kernel execution to create the next scale within the octave. We repeat this process for all scales within an octave before copying the resulting blurred image data back to host memory. We again perform this process for all octaves.

4.2.3 DoG Construction

To create our DoG set we require to subtract pairs of Gaussian blurred images within our scale space. While the computation is relatively simple we must still iterate over all pixel data within each scale to take the difference. Taking a similar approach detailed above, we implement a CUDA kernel that performs a difference of an individual pixel between scales. We perform this computation directly after the convolution step by using the Gaussian scale space data resident in device memory, hence avoiding the cost of copying the data to and from the device and host. Since the entire Gaussian scale space for a given octave is resident in memory we implement our CUDA kernel to perform the difference calculation between all scales for an individual pixel. This avoids the cost of having to continually re-synchronise with the host after a single pairwise difference operation allowing a single kernel execution to perform the DoG calculation for a complete octave. Figure 18 illustrates the DoG construction process where a thread will calculate all scale differences for a single pixel, denoted by a blue dot.

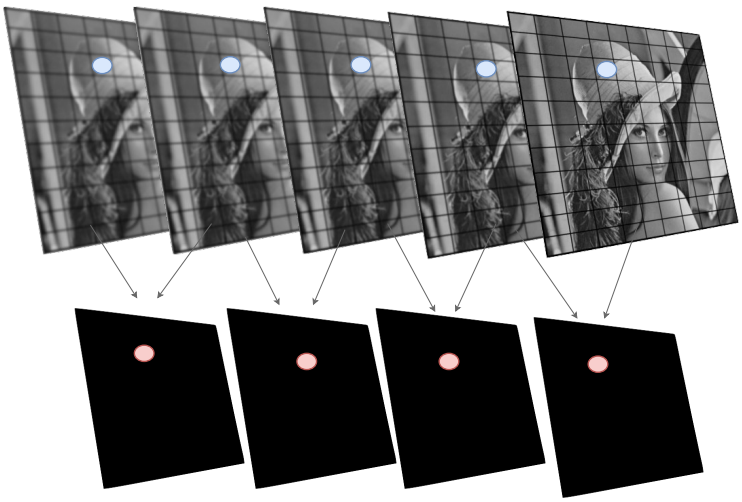

Figure 18: Constructing DoG space

4.3 Extrema Detection on the GPU

The next step of the SIFT algorithm is to detect the local extremas in the DoG scale space. This involves iterating through the scales of an octave, generating a candidate keypoint list if a given pixel within a scale is either the maxima/minima value among its 26 neighbourhood values (including the scale space above and below the given pixel). We could adopt a similar approach used in the DoG construction phase where a CUDA kernel can perform all the maxima/minima calculations for a pixel across all scales within an octave. Problematic with this approach however is that a given pixel within each scale we will be accessing 26 neighborhood values from global memory. Global memory accesses are relatively expensive, being orders of magnitude slower than shared memory accesses. Thus it is ideal for us to minimize the overall number of global memory accesses for our SIFT implementation. As mentioned earlier, shared memory is memory that is exclusively shared between threads within a given block. Shared memory can be thought of as a local cache for a block. We find when a thread calculates an extrema for a given pixel, it will use

a significant number of pixels that are accessed by neighbouring threads in the same block. By caching these accesses in shared memory we can optimize the extrema calculation on a GPU.

Due to shared memory size limitations, we are unable to cache all the scales of a given octave. Instead our CUDA kernel implementation caches three scales at a time within a given block, this being enough for a local extrema calculation. The threads cooperatively load the pixel data into shared memory and perform the necessary maxima/minima check. At the conclusion of the maxima/minima calculation threads within a block will proceed to replace the cached data of the oldest scale as it will no longer be used in subsequent calculations. The CUDA kernel repeats this step for all scales. The stages of this extrema calculation can be seen in Figure 19. To account for the case where a pixel requires data from outside the boundaries of a block, we execute our blocks with an additional set of redundant threads that act as a halo region. The application of a halo region is illustrated in Figure 20 such that an $N \times N$ block becomes an $(N+1) \times (N+1)$ block to account for the overlapping stencil region. These threads load the additional halo region data into shared memory but do not perform any extrema calculation. To avoid race conditions, the threads synchronize around loading new data into the shared memory region. Synchronisation only occurs with other threads in the same block, hence we pay careful attention to balancing the number of threads within a block and minimizing the branching behaviour of our kernel implementation to mitigate the cost of synchronisation.

Lastly, if an extrema is found we store keypoint candidate information in a global keypoint candidate list. A thread will atomically increment a global counter which in return will receive an index in the global candidate list to store the candidate information. It is important to note that this is a potential bottleneck for the algorithm such that all threads incrementing the global counter need to synchronize for the operation to be atomic. An alternative solution would be to maintain a local counter and candidate list for each block, storing the keypoints found within shared memory. After computing all extremas in a block a single thread can be responsible for updating the global candidate list, thus reducing the number of concurrent writers. We were unable to implement this solution however as atomic operations with respect to threads in a block was introduced with later versions of CUDA, being compute capability 6.x, whilst we only have 3.5 compute capable GPUs.

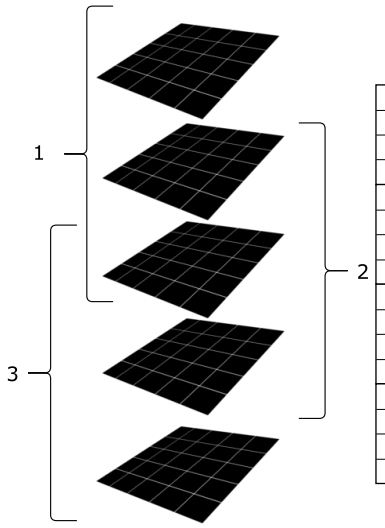

Figure 19: Extrema Comparisons

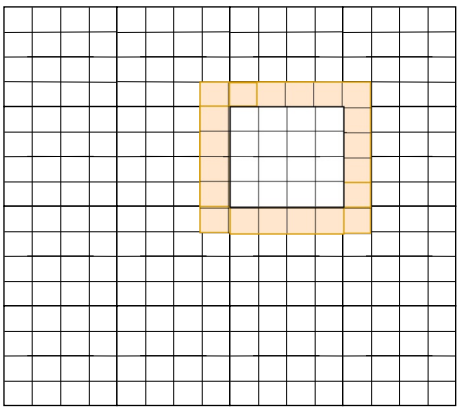

Figure 20: Halo Region around block

4.4 Keypoint Detection on the GPU

The final component we implement on the GPU is a stage of the keypoint detection process. We focus on the process of keypoint refinement, leaving the keypoint descriptor calculation to our CPU-based implementation. At the completion of the extrema calculation stage, we have a set of global keypoint candidates that we wish to refine. Refinement of the candidates is a two step process involving the interpolation of a candidate keypoint and its subsequent filtering. We implement both stages within a single CUDA kernel, performing the calculation of each keypoint in parallel. We formulate this by dedicating a single block for each keypoint (x, y), thus having N parallel blocks for N candidate keypoints. Each block has a set of threads representing the neighbourhood of pixel values around the keypoint. The interpolation and filtering steps require frequent access to neighbouring pixels thus we again exploit shared memory within each block. Each thread within a block, being relative to the keypoints neighbourhood, will load its corresponding pixel value into a shared memory cache.

The first step involves the interpolation of nearby pixels of a given keypoint candidate to calculate the subpixel maxima/minima. For a given keypoint candidate, $\mathbf{x} = (x, y, \sigma)^T$, we take the scale space function $D(x, y, \sigma)$ and construct a quadratic expression from its Taylor expansion. The quadratic expansion of this function, being shifted such that the origin is the key point candidate, is given by Lowe with the following expression:

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

By taking the derivative of $D(\mathbf{x})$ and setting it to zero we can find the maximum of the function. This yielding the equation:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}}$$

Taking the Hessian (introduced in Section 2.3) and our derivative, we sample the neighbourhood points around the keypoint to produce a 3×3 system of linear equations. By solving this we can solve to get the subpixel keypoint locations, $(\hat{x}, \hat{y}, \hat{\sigma})$. We use a specialised OpenCV CUDA implementation to efficiently solve the system of linear equations within our CUDA kernel. The last step of our GPU implementation is the filtering of unstable keypoints. Taking our calculated subpixel location, we filter out keypoints with low contrast and a low cornerness measure. These being the same curvature calculations introduced in Section 2.3.

5 Evaluation

The focus of evaluation is to implement a core subset of the algorithms within SIFT for a GPU. By doing so we aim to evaluate the possible performance benefits of bringing SIFT to a data parallel environment. When evaluating the performance of our GPU implementation it is important to note that the GPU implementation is not a complete version of SIFT. Thus we are unable compare a complete GPU implemented SIFT algorithm against our CPU-based version of SIFT. Secondly, though GPUs provide high data bandwidth, they also introduce high latency. Integrating our implemented GPU stages with our CPU implementation provides little performance gains due to the high latency cost of copying memory back and forth from host to device.

These points considered we focused on benchmarking a subset of the stages of SIFT individually. These stages being the scale space construction and extrema detection steps. We choose not to benchmark the keypoint filtering step as our GPU version is not a complete implementation of this stage. We also omit the cost of allocating and initialising memory within both the CPU and GPU versions.

5.1 Benchmarking Configuration

5.1.1 Harware Setup

We benchmark both implementations on the same system. The configuration of the system is described in Table 1.

Spec	**Value**
Arch	x86
ISA	x86_64
SoC	Sandy Bridge
CPU	i5-2500K
Cores	4
RAM	8GB
Max CLK	3.30Ghz
GPU	GTX 780Ti
GPU Memory	3072 MB
CUDA Cores	2880
CUDA Compute Version	3.5
Operating System	Ubuntu 14.04.5 LTS

Table 1: Testing Hardware

We measure the performance of our implementations in milliseconds (ms), taking the timestamps at the beginning and end of the algorithm. We use the 'clockgettime' library to retrieve monotonically increasing clock values, being independent of system time. Each benchmark calculates the average performance and standard error across 50 tests. We perform a series of warm-up runs to account for cold caches. Lastly all benchmarks use the same image data set.

5.2 Benchmarking Scale Space Construction

To evaluate the scale space construction stage of SIFT, we independently benchmark the performance of the convolution and image subtraction stage. We use these benchmarks to base our evaluation of the overall performance of scale space construction.

5.2.1 Gaussian Convolution

We firstly benchmark the cost of convolving a 5×5 Gaussian stencil across increasing image sizes. We evaluate the time it takes for a complete convolution iteration to complete on both our

GPU and CPU implementation. Our CPU implementation utilises the OpenCV API function, "GaussianBlur". The OpenCV implementation is considered optimised but only uses a single CPU core for the computation. We expect GPU performance benefits will come from running the algorithm in a parallelised environment. The results are show in Figure 21.

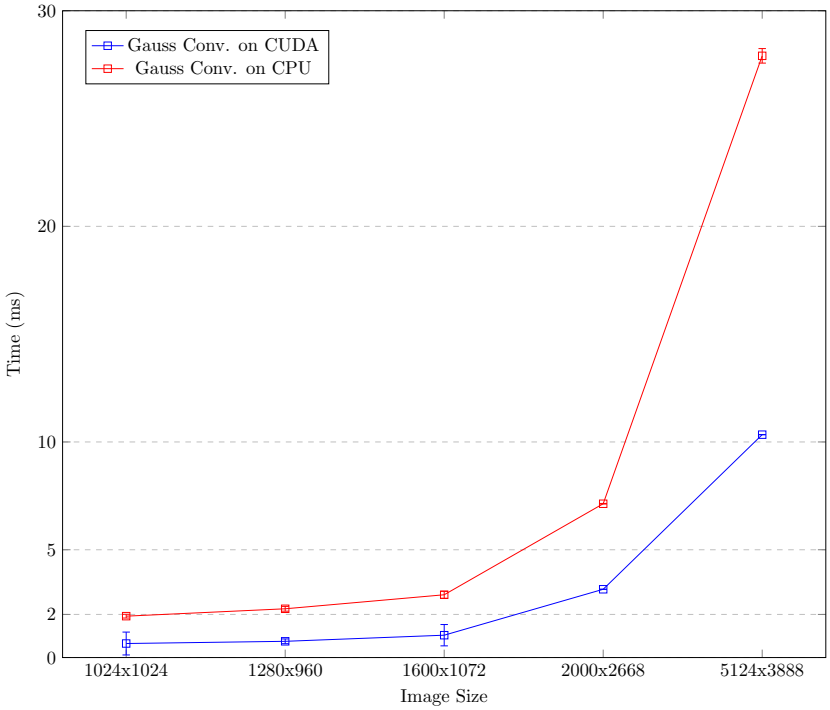

Figure 21: Gaussian Convolution Benchmark

As seen from the benchmarks above, implementing a Gaussian convolution on the GPU gives a 2-3 times speedup factor across various image sizes. We also see the cost of computation for the both the CPU and GPU implementation proportionally increase based on image size simply due to the extra data needed to be processed.

5.2.2 Difference Operation

We next benchmark the cost of performing image subtraction on both the GPU and CPU. This step being used to construct a DoG scale set within SIFT. Our GPU implementation involves subtracting all the scales for a given pixel within a single CUDA kernel execution. Thus we also measure the cost of $s-1$ consecutive DoG subtractions in our CPU implementation, s being the number of scales within a given octave. In our benchmarking tests, s is equal to 5. Our CPU implementation utilises the OpenCV API function, "subtract". We again expect GPU performance benefits will come from running the algorithm in a parallelised environment. The results are show in Figure 22.

From the benchmark results we receive significant speedup factors across all the images sizes. A benefit to implementing the difference step on the GPU is due to not having any race conditions or having to implement synchronisation between threads. Each thread in parallel is able to complete a subtraction operation for all scales within an octave irrespective of other threads. We again see the cost of computation for the both the CPU and GPU implementation proportionally increase based on image size due to the extra data needed to be processed.

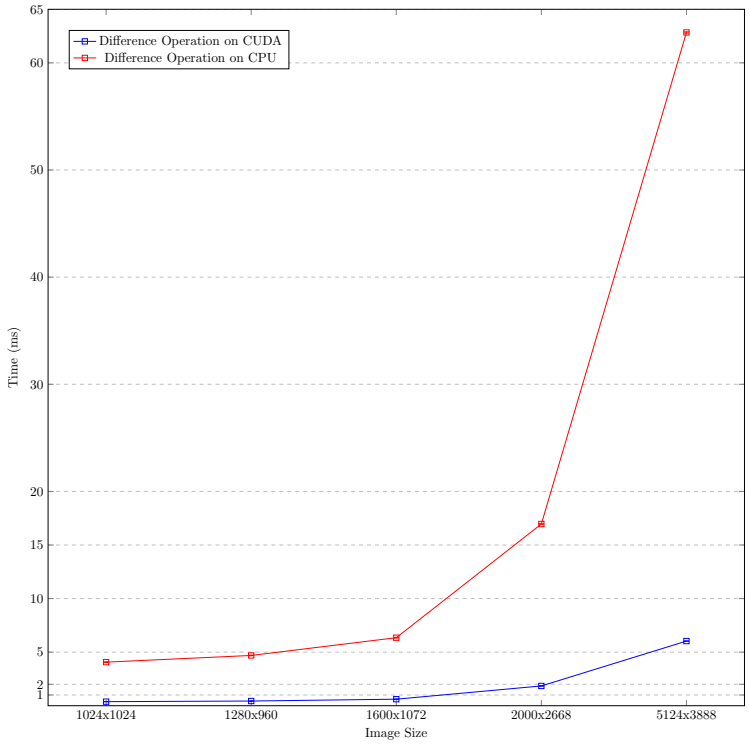

Figure 22: Image Difference Benchmark

5.2.3 Scale Space Construction

We lastly benchmark the performance of constructing a single octave. This is involves a series of 5 Gaussian convolutions and 4 image differences to create each scale. The benchmarks are shown in Figure 23. The results align with our previous benchmarks. We observe a speed up factor ranging from 2-3 with our GPU implementation. We find the DoG calculation is the least expensive step in the scale space construction step, consuming 30-40% of the overall calculation whilst the Gaussian convolution step is the most expensive operation in the scale space construction phase.

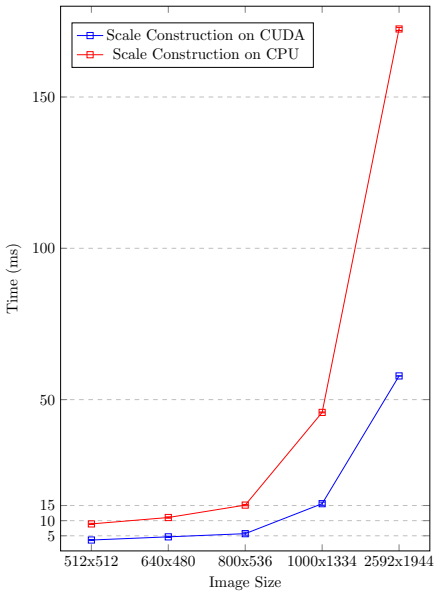

Figure 23: Octave Construction

5.3 Benchmarking Extrema Detection

We next focus on benchmarking the performance of extrema detection within SIFT. The step focuses on the generation of candidate keypoints from our DoG scale space. We benchmark the extrema detection within a single octave containing 5 scales (4 DoG datasets) and perform this operation on varying image sizes to evaluate the cost of calculation relative to the amount of data to process. The benchmarking results are illustrated below in Figure 24.

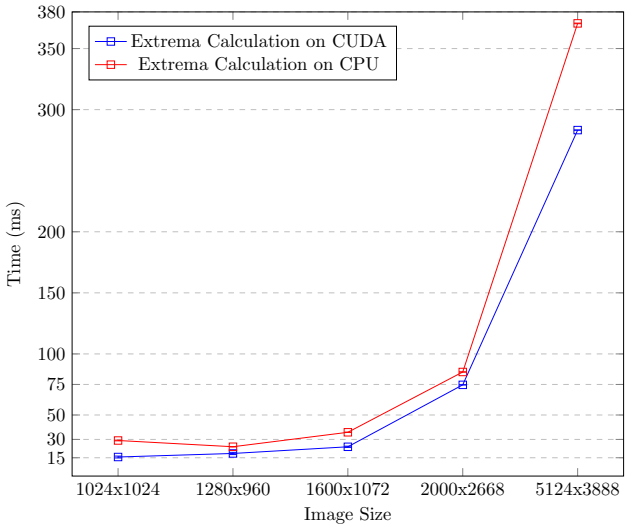

Figure 24: Extrema Detection

The GPU performance shows a speed up over the CPU implementation however is not as significant as the previous results. We believe this is due to the use of an atomic counter within our CUDA kernel implementation. The cost of synchronising over a counter offsets the performance benefits of parallelisation. As discussed, a solution would involve maintaining a local counter in shared memory. However atomic operations relative to threads in a block is not supported for our version of CUDA, being CUDA compute 3.5. To understand the impact of atomic operations on the performance of our extrema calculation we modify our implementation such that it has no atomic counters. We illustrate the performance of this modification in Figure 25. The modification produces an incorrect output however we see on average a 2x speedup when not using atomic variables. Ideally we would like to reach a balance the two results.

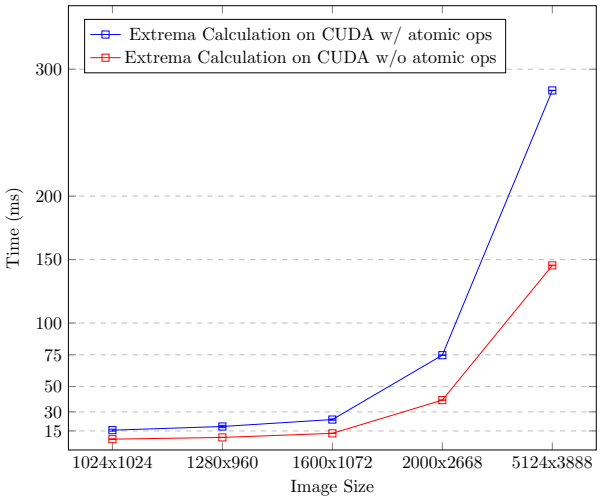

Figure 25: Extrema Detection

6 Conclusion

The goal of this project was to explore SIFT and investigate optimisations that could be made through accelerating stages of the algorithm on a GPU. To achieve this goal we presented a CPU implementation of the SIFT algorithm and an implementation of a subset of the SIFT algorithm stages for a GPU.

Through this project we outlined the various stages of the SIFT algorithm and how they work in conjunction to detect reliable scale invariant keypoints for a given image. We evidenced this through implementing SIFT and demonstrating its ability to extract these keypoints. Taking our implementation we demonstrated applications of the SIFT algorithm. We explored how SIFT can be used as a mechanism for feature tracking and matching between images.

With our exploration of SIFT we looked at potential opportunities to accelerate the algorithm on a GPU. We explored the programming approach of data parallelisation and how we can transform stages of the SIFT algorithm to suite this model. We discussed our approach to implementing Gaussian convolutions, DoG construction, scale space construction, extrema detection and keypoint filtering on NVIDIA's CUDA framework.

We lastly evaluated the performance of the various GPU implementations against our CPU based SIFT. We identified the speedup benefits from parallelising sub-stages of the SIFT algorithm whilst also finding potential bottlenecks and improvements that could be made to our GPU implementation.

References

[1] R. Fisher. Gaussian smoothing. `https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`. Accessed: 2017-11-17.

[2] Tony Lindeberg and Bart M. ter Haar Romeny. *Linear Scale-Space I: Basic Theory*, pages 1–38. Springer Netherlands, Dordrecht, 1994.

[3] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004.

[4] Trevor McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, School of Computer Science and Engineering, 2014.

[5] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.