ECE1724 Project Report
Rev: 1.0
April 18, 2009

ECE1724 Project
Speeded-Up Speeded-Up Robust Features

Paul Furgale, Chi Hay Tong, and Gaetan Kenway
<paul.furgale@utoronto.ca>
<chihay.tong@utoronto.ca>
<kenway@utias.utoronto.ca>

1 Introduction

Feature detection and matching is one of the fundamental problems in low-level computer vision. Given a series of images of the same object, feature detection and matching algorithms try to (1) repeatably detect the same point of interest in every image, regardless of the scale and orientation of the object, (2) match each point of interest from one image with the same point in another image. Feature detection and matching form the basis for many higher-level algorithms such as object recognition [1], robot navigation [2], panorama stitching [3], and three-dimensional scene modeling [4].

In the last 10 years, the Scale Invariant Feature Transform (SIFT) [5] has become the standard algorithm for this task. However, while its performance is quite impressive on difficult matching tasks, it is too computationally expensive to use in online or real-time applications. In response to this, Bay et al. proposed the Speeded-Up Robust Feature (SURF) [6], which uses an integral image (essentially a 2D prefix sum) to approximate many of the calculations in the SIFT algorithm. The SURF algorithm is much faster than SIFT, but it is still not fast enough to use in real-time applications.

Our interest is the use of the SURF algorithm for real-time robot navigation. Traditionally, visual motion esti-

mation has used more efficient, less robust algorithms for this task (e.g. [7, 8]). However, with the introduction of efficient numerical methods that estimate the motion of a vehicle using features tracked over multiple frames [8], we believe that using a better feature detection and matching algorithm should (1) increase the accuracy of visual motion estimation, and (2) allow robots to determine their unique location within a map of visual features. SURF is much faster to compute than SIFT (interestingly, SURF is about as fast as GPU-accelerated SIFT), but it is still not fast enough to use in real-time. Therefore, it was our goal to use the GPU to accelerate the SURF algorithm to a point that it may be used in real-time robot navigation.

Rather than choosing a very simple algorithm and optimizing it as far as it could go, we have implemented what ended up being a very complicated algorithm. We were first and foremost concerned with verifying that the algorithm was correct, and second to that, optimization. To do this, we had to verify that the algorithm implemented on the CPU matched the original paper as closely as possible. Then we checked that each part of our GPU implementation matched the CPU implementation. Where it doesn't match, we have been careful to track down exactly why.

2 Related Work

2.1 GPU Feature Detector Implementations

We based our algorithm on the original SURF paper [6], with hints from previous GPU implementations of similar algorithms [9, 10]. We also investigated implementing improvements to the algorithm, as long as there was no degradation in performance. No one has previously implemented and released a CUDA version of SURF. Although the SIFT algorithm is currently the gold standard for scale/rotation invariant feature detection, it is too computationally expensive to use in real-time or large-scale projects. Several groups have put time into developing more efficient detectors and descriptors [11, 6] and others have ported the algorithms to the GPU. As it is most relevant to this project, we briefly outline work to date in porting these algorithms to the GPU.

Work on the GPU port of SIFT started as early as 2007 [12]. This open-source project is still in active development and the authors have gone through several iterations. Their work started as shader programs and now has been ported to CUDA. They report speeds of up to 30 Hz on 640×480 [1]. There is also a pure CUDA version of SIFT available but it comes with no evaluation and no documentation [13].

Two groups have produced closed-source GPU implementations of SURF. Terriberry et al. produced an implementation that runs at 30 Hz on 1280×960 images with 1000 features [10]. They have put a lot of thought into the acceleration of every part of the SURF algorithm and it is well detailed in their report. Although no part of their algorithm uses CUDA, most of their implementation tips would transfer in a straightforward way to the CUDA framework. The authors of the original SURF paper have also published a GPU implementation [9]. Their work deviates from the original SURF algorithm in the feature detection step, focusing on shader programs and GPU-specific memory usage. Their explanation of their CUDA implementation of orientation assignment and descriptor generation is low on detail, but it seems, on the surface, to be very similar to our own.

2.2 OpenSURF

With the idea of speeding development, we started with an open-source implementation of the SURF algorithm, OpenSURF [14]. The library is has a very clean interface and many comments. Our plan of action as we dug in to

[1] 4 to 6 Hz on my laptop.

the code was as follows:

1. Verify that the OpenSURF implementation matches the description in the original paper [6] and, where possible, matches the results from the original SURF binary.
2. Implement the GPU equivalent code.
3. Verify that the GPU equivalent code produces the same results.

Our implementation is a mix of OpenSURF, and our interpretation of the original SURF paper. In addition to changing parts of the algorithm, we modified it to match our GPU implementation so we could verify our calculations. Specifically, the GPU offers fast bilinear filtering of texture lookups which allow us to compute the orientation and descriptor at the exact scale they are found (as opposed to the nearest integer scale). We implemented this bilinear filtering on the CPU which increases the computational complexity by a factor of four. In this paper we will reference several implementations of the SURF algorithm. We will use the following tags when talking about the different implementations:

SURF	The closed-source implementation of SURF distributed by the original paper writers [6].
OpenSURF	The original version of OpenSURF [14].
CPU-SURF	Our modified version of OpenSURF.
GPU-SURF	Our implementation of SURF on the GPU.

3 Algorithm Description

This algorithm is split into a number of discrete steps. Each step of the algorithm and its GPU implementation is described below.

3.1 Compute Integral Image

Integral images allow for efficient and fast computation of box-type convolution filters, of which the SURF algorithm is largely based upon. The entry of an integral image $I_\Sigma(x, y)$ represents the sum of all pixel intensities in the input image I above and to the left of the location (x, y).

$$I_\Sigma(x, y) = \sum_{i=0}^{i \le x} \sum_{j=0}^{j \le y} I(i, j) \tag{1}$$

Once the integral image is computed, it takes only four lookups and three additions to calculate the sum of the pixel intensities over any upright, rectangular area (See Figure 1). These access requirements are independent of size, which is the key reason for the SURF algorithm's speed.

In considering the algorithm for computing the integral image on the GPU, we noticed that could be computed by conducting two inclusive parallel-prefix sums: one along the rows, and one along the columns. As a result, to generate the integral image on the GPU, four kernel calls are used.

1. Transpose and convert the image to normalized float representation.

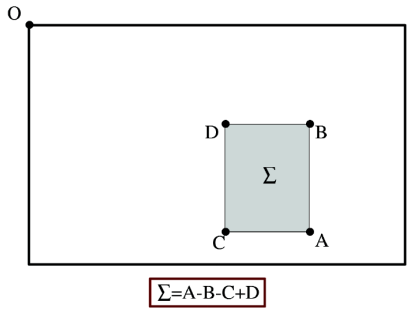


Figure 1: Illustration of an area lookup using an integral image. Image credit: Bay et al. [6]

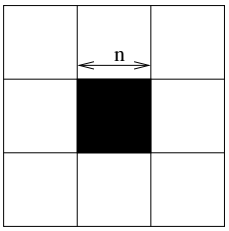


Figure 2: CenSurE kernel response, Image credit: Konolige et al. [8]

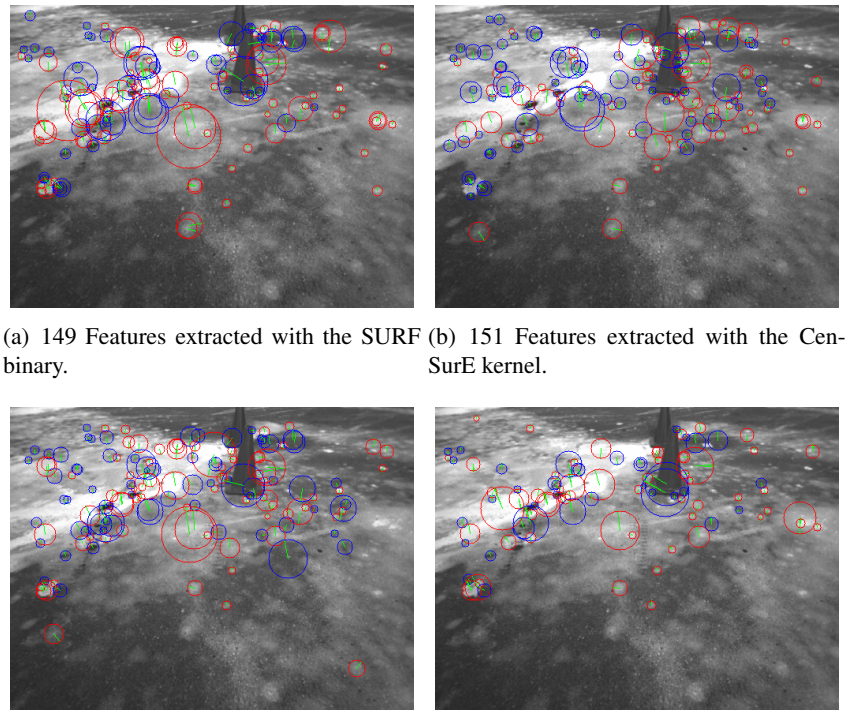
2. Compute the scans for all rows (columns) of the image.
3. Transpose the column-scanned image back to the original orientation.
4. Compute the scans for all rows of the column-scanned image.
5. Transfer integral image from linear memory to texture memory, and bind the texture.

The reason the computation was split into four separate kernels was so that the CUDA Parallel Primitives (CUDPP) library could be utilized for the efficient computation of multiple-row scans. Since the `cudppMultiScan()` function operates on rows of data, transpose operations are needed to efficiently compute the column scans as well. Some speedup could be gained by combining the kernels, saving global memory reads and writes. However, it was felt that the CUDPP library should be well-written, and optimized for performance.

Additionally, we wanted to use the sub-pixel indexing capabilities of textures. Since linear filtering is only supported for floats, we needed to convert the image pixel intensities from integers ($[0, 255]$) to normalized floats ($(0, 1)$). For efficiency, this conversion is conducted during the initial transpose step, while loading the data into shared memory.

3.2 Compute Interest Point Operator

To identify interesting points in an image, we evaluate a function at each pixel that describes how "interesting" that location is. In multi-scale algorithms such as SIFT and SURF, the interest operator is computed at different scales as well. The interest point operator used in the original SURF implementation [6] uses the determinant of the Hessian.



(a) 149 Features extracted with the SURF binary.

(b) 151 Features extracted with the CenSurE kernel.

(c) 151 Features extracted with the CPU-SURF kernel.

(d) 151 Features extracted with the original OpenSURF kernel.

Figure 3: A comparison of the different detection kernels.

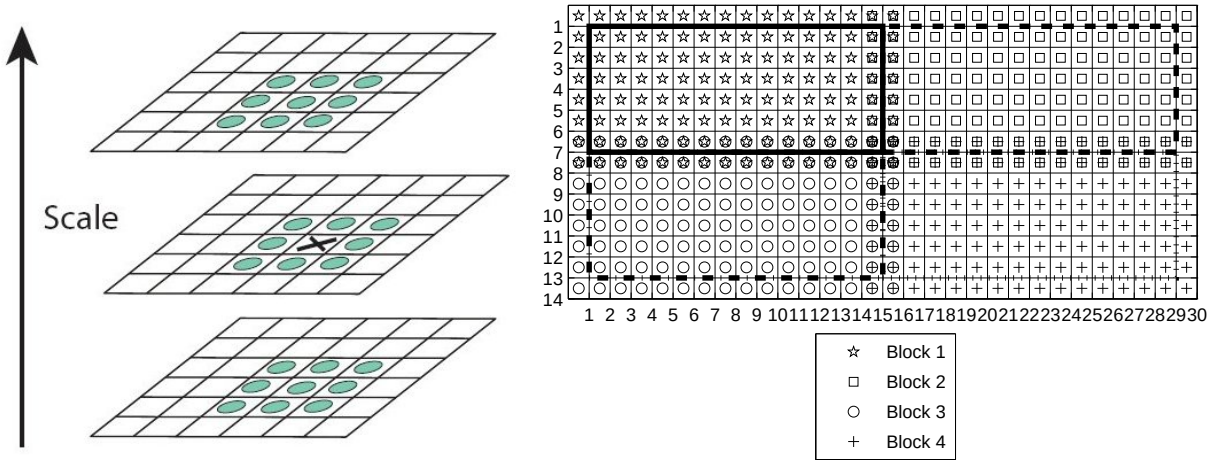
For a given smooth function f of two variables,(x, y) , the Hessian is the matrix of partial derivatives given by:

$$H(x, y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{2}$$

The partial derivatives of the image intensity at each point are approximated using a convolution of the image with a series of box filters. This convolution can be evaluated in constant time regardless of scale by using the integral image. Unfortunately, we were not able to match the results of the original SURF binary. For comparison, we also implemented a simpler interest point operator known as Center Surround Extremas (CenSurE). Simply, this filter identifies blobs in the image, by producing a maxima or minima when there is a light image patch on a dark background, or a dark image patch on a light background. The filter response for CenSurE is shown in Figure 2. A comparison of the different detectors is shown in Figure 3. We are currently investigating which operator exhibits the best performance.

3.3 Find Min/Max of the Interest Point Operator

After the interest point operator has been computed for each pixel, at each scale and for each octave, we wish to locate extrema in the responses in pixel directions, as well as in scale. This is accomplished by comparing a given



(a) The sample mask for finding maximum/minimum values Image credit: Lowe [5]

(b) A 2D slice of interest point data loaded onto shared memory

Figure 4

pixel value to its 8 neighbors in the given scale and the 9 neighbors in each of the scales above and below. This procedure amounts to finding values that are greater than/less than its 26 neighbors arranged in a $3 \times 3 \times 3$ cube as shown in Figure 4(a).

The basic flow of the algorithm on the GPU is as follows:

- Each block loads a $16 \times 8 \times 4$ (x-y-scale) section of interest point operator into shared memory.
- A border of 1 in each dimension is used.
- Extrema calculation is computed for a centered $14 \times 6 \times 2$ sub-block of loaded data.
- Data block loads must be overlapped due to border effects.
- If an extrema is found add to extrema array using an atomic increment to get a unique index.

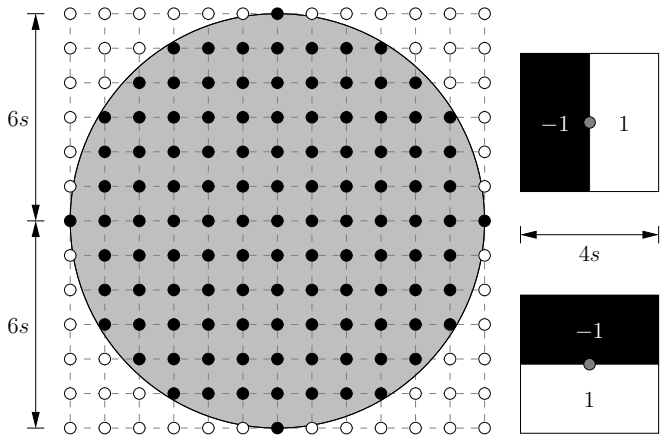
A sketch of the overlapping image blocks is shown in Figure 4(b). The bold outlines show which elements each block operates on.

3.4 Find Sub-Pixel/Sub-Scale Interest Point

After all the response extrema have been located, a sub-pixel (and scale) interpolation is performed to further improve the localization of the interest point. The step is called as a separate kernel with a linear set of blocks each corresponding to an extremum.

Equation 3 shows a second order Taylor series expansion of the interest point operator function denoted as L. This defines a parabola fixed by the value and the first and second derivatives of L.

$$L(\mathbf{x} + \Delta\mathbf{x}) = L(\mathbf{x}) + \left(\frac{\partial L}{\partial \mathbf{x}}\right)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \left(\frac{\partial^2 L}{\partial \mathbf{x}^2}\right) \Delta\mathbf{x} \tag{3}$$



(a) Lattice Points (b) Haar filters

Figure 5: The sampling layout and masks for the orientation assignment. Image credit: Terriberry et al. [10]

We are interested in the vertex of this parabola, so we take the derivative with respect to $\Delta\mathbf{x}$ and set it equal to 0. This results in the system of equations shown by Equation 4. All derivative terms are approximated using centered finite difference schemes. This results in a 3x3 system of equations.

$$\Delta\mathbf{x} = -\left(\frac{\partial^2 L}{\partial \mathbf{x}^2}\right)^{-1} \frac{\partial L}{\partial \mathbf{x}} \tag{4}$$

If the resulting $\Delta\mathbf{x}$ is greater than 0.5 in either x, y or scale, that interest point is not included in the list.

3.5 Compute Interest Point Orientation

To find a repeatable orientation for a feature point, the distribution of image gradients around it is examined. Haar wavelet responses corresponding to d_x and d_y are computed at 13×13 sample points distributed in a grid around the feature point. This layout is shown in Figure 5. The d_x and d_y responses are scaled by a Gaussian ($\sigma = 2s$) and the angle $\theta = \text{atan2}(d_y, d_x)$ is computed for each sample point. Next, the scaled values of d_x and d_y are summed over an angular sliding window of size $\frac{\pi}{3}$. The window with the largest vector magnitude $\sqrt{(\sum d_x)^2 + (\sum d_y)^2}$ is crowned the dominant orientation and the value $\phi = -\text{atan2}(\sum d_y, \sum d_x)$ is used as the interest point orientation. Orientation assignment is the slowest part of our GPU implementation. However, this is fits with reports from other papers that have implemented SURF. Our GPU implementation is described below.

First, it was noted that the integral image lookups needed to compute the axis-aligned grid of Haar wavelets overlap exactly. To save memory bandwidth, 17×17 threads are launched for each kernel and they make the 17×17 texture calls, storing the results in shared memory. Then the first 13×13 threads compute the scaled Haar wavelets d_x and d_y and use the results to produce θ. The result is stored in an array of `float3` data as d_x, d_y, θ.

The resulting list of 169 sample points is sorted using a parallel-bitonic sort. This was adapted from the NVIDIA sample code but changed in two ways. First, instead of sorting integers, the code here sorts an array of `float3` data, so the result is ordered by θ. Second, as the sort only works on arrays that are a power of 2, the input array is padded with dummy elements (θ = FLOATMAX) up to the array size of 256.

Finally, each of the 169 threads selects an angle in the sorted list and marches along the sorted array summing up all d_x and d_y values in the $\frac{\pi}{3}$ window. This is followed by a simple reduction to find the largest squared vector length $(\sum d_x)^2 + (\sum d_y)^2$. Again, the array is padded with dummy values so that the reduction can work on 256 elements.

As stated before, this ends up being the slowest part of our GPU algorithm. Several different optimization methods have been tried including:

1. Reducing the amount of shared memory used.
2. Replacing the sort with a linear search.
3. Calculating the final angle after the reduction to avoid 169 calls to atan2$(\cdot)$

For whatever reason, each of these optimizations makes the kernel slower.

3.6 Compute Interest Point Descriptor

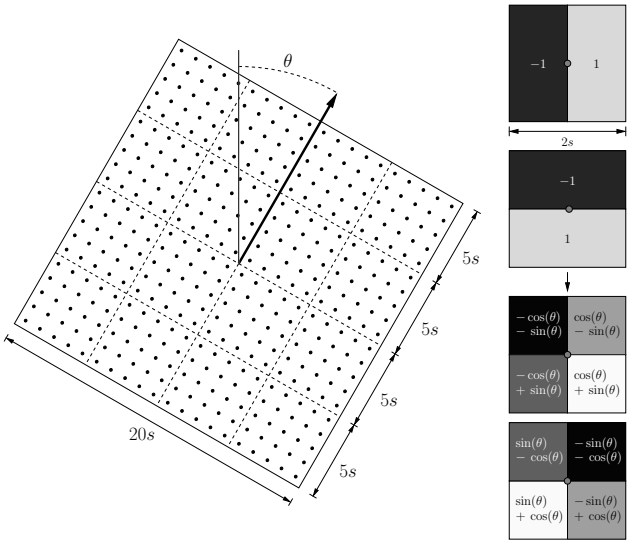
To construct the interest point descriptors, first, a square region of $20s \times 20s$ centered around the interest point and oriented along the orientation computed in the previous section is defined. The region is then split up equally into smaller 4×4 square subregions. For each sub-region, Haar wavelet responses are computed at 5×5 regularly spaced sample points. The lattice of sample points is illustrated in Figure 6. While the wavelet responses, d_x and d_y are defined to be aligned to the primary orientation, it is more efficient to compute the axis-aligned responses, and rotate them accordingly. Each of the responses are weighted by a Gaussian ($\sigma = 3.3s$), centered at the interest point.

The wavelet responses and their absolute values are then summed up over each sub-region, forming a four-entry descriptor vector for each sub-region ($\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$). Concatenating this for all 4×4 sub-regions results in a descriptor vector of length 64. Finally, the descriptor vector is normalized into a unit vector.

For the GPU implementation, we decided to use two kernels: one to compute the unnormalized descriptors, and a second to conduct the normalization step. This substantially simplifies the code, since thread indexing is clearer without the need for bit masking. Initially, this was implemented as a single kernel, but too much shared memory was used per block. Upon profiling, it was observed that `cubin` allocated local memory for this kernel when compiled for our laptop, which prompted the split. Using the desktop computers, this split made the kernel marginally slower (likely due to the extra global memory read and write).

The first kernel call allocates 16 blocks per interest point (reflecting the 4×4 subregions), and 25 threads per block (5×5 sample points per subregion). The algorithm then progresses as follows:

1. Load interest point parameters (x, y, s, ϕ) into shared memory.
2. Compute trigonometric rotations $(\sin(\phi), \cos(\phi))$.
3. Compute sample point locations.
4. Load integral image lookups (9 per sample point).
5. Compute axis-aligned Haar filter responses (d_x, d_y).
6. Rotate and store the Haar filter responses in shared memory.



(a) Lattice Points (b) Haar filters

Figure 6: The setup used to calculate a feature vector. (a) The lattice points where Haar responses are sampled. (b) The Haar filters used to compute the feature vector values. The responses from the two axis-aligned filters at the top are rotated to effectively achieve the pair of filters at the bottom. Image credit: Terriberry et al. [10]

7. Load and store absolute values $|d_x|$ into another shared memory block.
8. Sum the $d_x, |d_x|$ responses (reduction).
9. Write back the unnormalized responses to global memory.
10. Repeat for $d_y, |d_y|$.

For this algorithm, since the sample points are aligned to the rotated axis, the integral image texture lookups do not line up, like in the orientation computation. However, there are some savings in that lookups are reused between the d_x and d_y filters. Since memory read coalescing for textures is not well documented, spatial locality of memory accesses were attempted, but not confirmed. Additionally, in the reduction steps, 25 numbers needed to be summed. Though the standard approach is to extend the shared memory block to the next highest power of two (32), in the interest of shared memory requirements, a reduction step was first taken using 9 threads to reduce from 25 to 16 numbers. Then, conventional unrolled reduction was used. Some indexing tricks were used to overcome the restriction that grids can only be 2D, while this kernel configuration makes more sense as a 3D grid.

For the second kernel, 1 block was allocated per interest point, with 64 threads per block (corresponding to the dimensionality of the descriptor vectors). The algorithm is as follows:

1. Load the values of the descriptor into local registers.
2. Square the values and store them into shared memory.
3. Sum (reduce) the squared values.

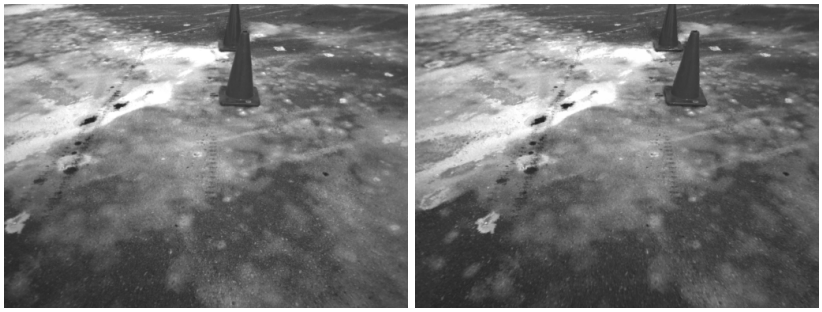


Figure 7: A sample stereo pair of images from our test set.

4. Compute the square root of the sum (length of the vector).
5. Divide each of the descriptor values by the length and write back to global memory.

This algorithm is fairly straightforward, and easily coalesces the memory reads and writes. Bank conflicts are avoided, since each thread only accesses one memory location each.

4 Methodology and Evaluation

As we mentioned in Section 2.2, our basic methodology was as follows:

1. Verify that the OpenSURF implementation matches the description in the original paper [6] and, where possible, matches the results from the original SURF binary.
2. Implement the GPU equivalent code.
3. Verify that the GPU equivalent code produces the same results.

For evaluation, we used a series of stereo test images of various sizes collected at the University of Toronto Institute for Aerospace Studies. A sample stereo pair is shown in Figure 7. We used a sequence of eight stereo pairs at resolutions of 512×384, 640×480, 1024×768, and 1280×960. This section will detail the method and metrics we used to evaluate each step of the algorithm and the associated results.

4.1 Timing Summary

Here we present the summary of the timing performance for each of the algorithm's subcomponents. Table 1 shows the percentage of computation time spent on the feature point independent tasks in the upper half and the feature point dependant tasks in the lower half. We see for the first three tasks, their computational cost is split relatively evenly, with slightly more time spent on the find min/max operations for the larger image sizes. For the feature point dependant operations, by far the most time consuming step is the determination of the orientation of the feature point.

Figure 8 graphically shows the time requires to process the 20 test images. The first thing we notice, it a very nearly linear speed up with image size. Secondly, there is not one particular kernel operation that consumes vastly

Image Size	512x384	640x480	1024x768	1280x960
Integral Image	35.1%	29.9%	27.3%	28.2%
Interest Point Operator	28.4%	31.0%	32.1%	31.9%
Find Min/Max	36.5%	39.1%	40.6%	39.9%
Sub pixel-scale Interpolation	11.4%	10.9%	10.2%	9.9%
Orientation	64.9%	64.7%	64.4%	65.0%
Descriptor	23.7%	24.4%	25.5%	25.1%

Table 1: Percentage of time for computations feature point independent calculations [top] and dependant [bottom]

more time than the others. This simply indicates a balanced approached must be taken when optimization speedups are considered.

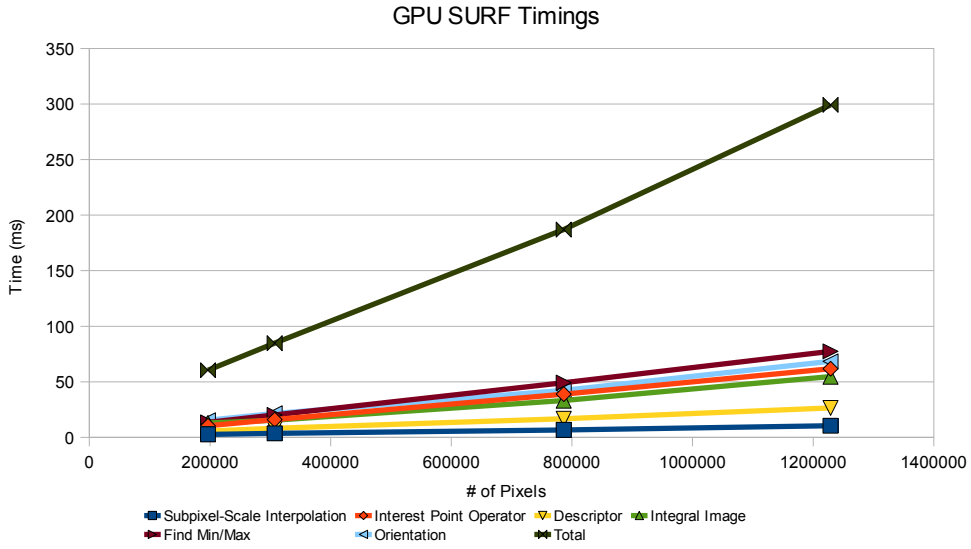


Figure 8: Computation time required to process all images of varying sizes

4.2 Overall Speedup

In addition to timing the individual parts of the algorithm, the processing time for complete runs were also profiled. Using a single test image over the four image sizes, the average processing time of 1000 trials for the GPU-SURF implementation was compared against average times over 10 trials for the CPU implementations (OpenSURF and CPU-SURF). The timings do not include memory initializations, but do include memory transfers to and from the GPU. SURF was not profiled for these trials since without source code, the memory initializations could not be isolated and excluded from the timing. Table 2 shows the results of the experiments on the SF2202 desktop computers.

As can be seen, CPU-SURF takes about 4x as long as the unmodified OpenSURF, which corresponds to the

Image Size	512x384	640x480	1024x768	1280x960
Feature Count	957	1509	3032	3218
CPU-SURF	961.86ms	1461.65ms	3409.69ms	4153.78ms
OpenSURF	251.27ms	384.95ms	998.29ms	1276.63ms
GPU-SURF	6.75ms	10.83ms	21.10ms	28.00ms
CPU-SURF Percent Speedup	14250%	13496%	16160%	14835%
OpenSURF Percent Speedup	3722%	3554%	4731%	4559%

Table 2: Overall speedup comparison between the CPU and GPU implementations.

4x increase in time complexity due to the implementation of bilinear filtering. All and all, these numbers show a substantial speedup vs. the CPU-based algorithms. These numbers can be improved upon with asynchronous memory transfers to and from the GPU.

4.3 Compute Integral Image

(a) Input test image. (b) CPU implementation. (c) GPU implementation.

Figure 9: Plots of rounding errors between float and integer (exact) representation of the integral image (lighter value indicates more error).

As mentioned in Section 3.1, we wished to utilize the sub-pixel texture indexing capabilities of the GPU. However, to use this feature, the texture must be composed of floats. As a result, the intensity values of the input image were converted from integers ($[0, 255]$) to normalized floats ($(0, 1)$). This results in a loss of precision, which is noticeable when compounded through addition, as in the case of the integral image. To gauge the error, we compared the integral images generated by the CPU and the GPU to an exact integral image computed from the integer values. Figure 9 shows a sample image, and plots of the errors for our CPU and GPU implementations. Computing the RMS errors, the CPU had a value of 0.0397, whereas the GPU had a value of 0.0066. While the GPU implementation has a much lower error, we have not been able to explain this. We suspect that a combination of low precision and rounding resulted in a series of fortunate errors.

To compare the following parts of the algorithm, we needed to ensure the inputs were the same. Since the GPU obtained more correct results, we decided to use the GPU's results, by copying the computed integral image back to host memory. However, we discovered that although the same integral image is used, sub-pixel lookups still remain

slightly different. Appendix D.2 of the CUDA Programming Guide details the algorithm for how bilinear filtering is done, but the actual hardware implementation details are not revealed. Therefore, we could not reverse-engineer and emulate the results produced by the GPU on the CPU, so the algorithms past this point will not produce exactly the same results. However, it is expected that the results are very similar in the majority of cases. For some series of computations, the values may be unstable, due to the lack of denormals on the GPU (which then order of operations matter). However, robustness is still present due to the large number of feature points identified by the algorithm, and these spurious features would simply not be used. One possible hint to the cause of the difference in texture lookup values is that the Appendix mentions that the fractional part is stored in 9-bit fixed point format, with 8 bits of fractional value.

4.4 Compute Interest Point Operator

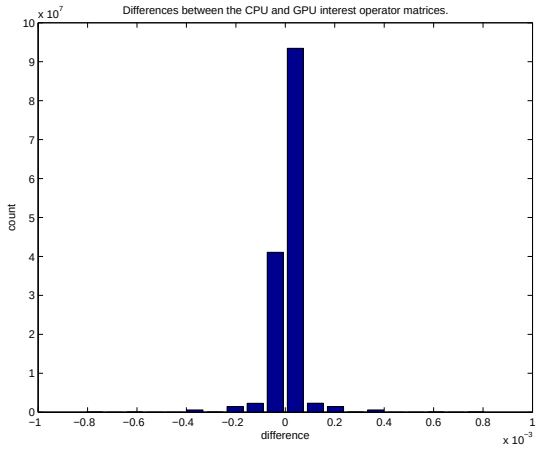


Figure 10: A histogram of difference between the CPU and GPU implementations of the interest point operator over all images.

Figure 10 shows the error difference between the interest operator computed on the CPU and GPU over all test images. Once again, we believe the errors that are observed are due to the inexact texture lookups on the GPU.

4.5 Find Min/Max of the Interest Point Operator

When the CPU and GPU operate on identical interest point operator data, the CPU and GPU implementation determined the same extrema points. For the complete GPU implementation, slight discrepancies resulting from the texture lookups in the interest point calculation resulted in a set of features which was very similar, but not identical.

4.6 Find Sub-Pixel/Sub-Scale Interest Point

Again, for verification purposes, we ensure the CPU and GPU versions are operating on the same interest point data to ensure we are only verifying the algorithm's accuracy. As we see from figure 11 the majority of the errors are less than 0.2×10^{-6} which is approaching on the machine precision accuracy of the float data type.

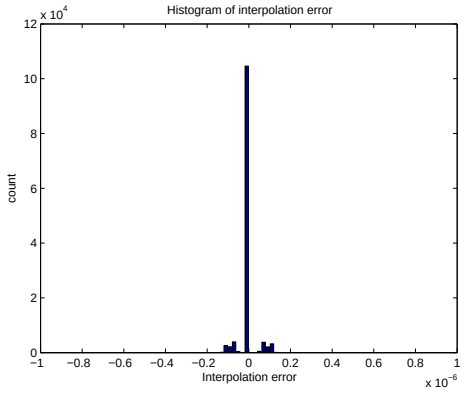


Figure 11: A histogram of difference between the CPU and GPU implementations of the subpixel interpolation over all images

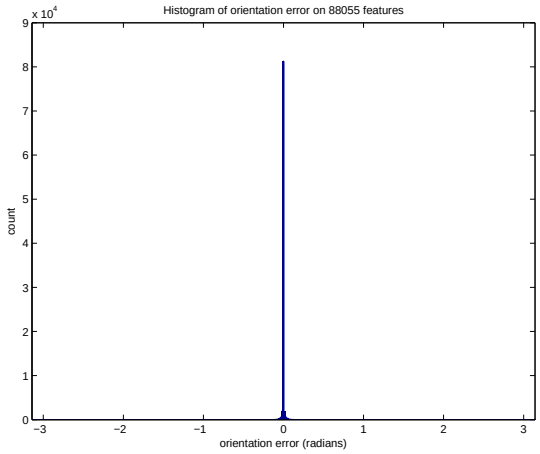


Figure 12: A histogram of orientation errors over all features in the image sequence.

4.7 Compute Interest Point Orientation

To evaluate the interest point orientations, we extracted feature points with our CPU implementation of the algorithm, and then computed the orientation with both the CPU and GPU. A histogram of orientation differences is shown in Figure 12. A small number of features had very different orientations but the vast majority had very small differences stemming from the texture lookups. The features with large differences in orientation are most likely from areas in the image with no clearly dominant gradient. In this case, the small differences in texture lookups cause a different result when a feature has two closely competing orientations.

4.8 Compute Interest Point Descriptor

To evaluate the interest point descriptors, the same interest point locations and orientations were provided to both the CPU and GPU implementations. The Euclidean distance between the resulting descriptor vectors were computed,

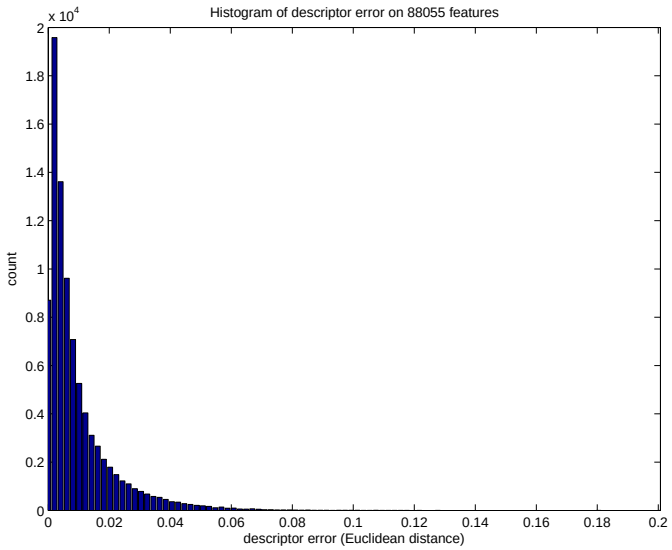


Figure 13: A histogram of descriptor error Euclidean distances over all features in the image sequence.

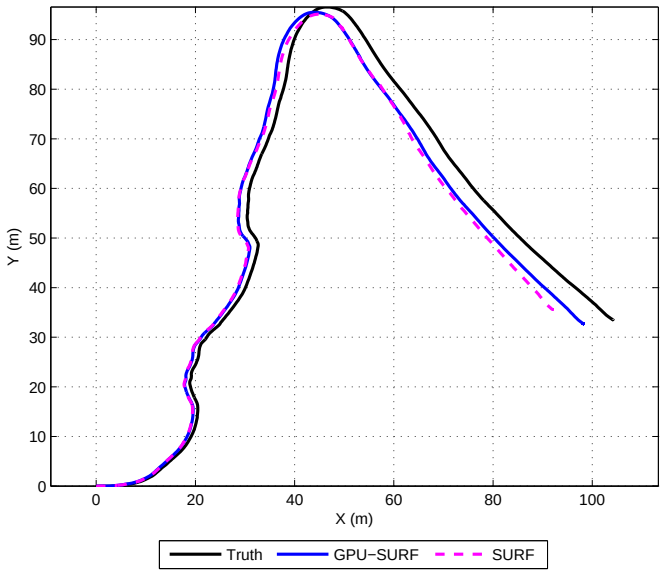
which can be seen as a histogram in Figure 13. Much like the previous sections, the algorithms do not match exactly, as expected. However, two points to note are the relatively small maximum error of 0.2, as well as that the largest peak is the one next to zero. This is again, as expected, as the integral image lookups between the CPU and the GPU are slightly different. These results match that of the orientation error histogram - the only reason it is more spread out is because the histogram bins in this case are smaller.

4.9 Visual Odometry

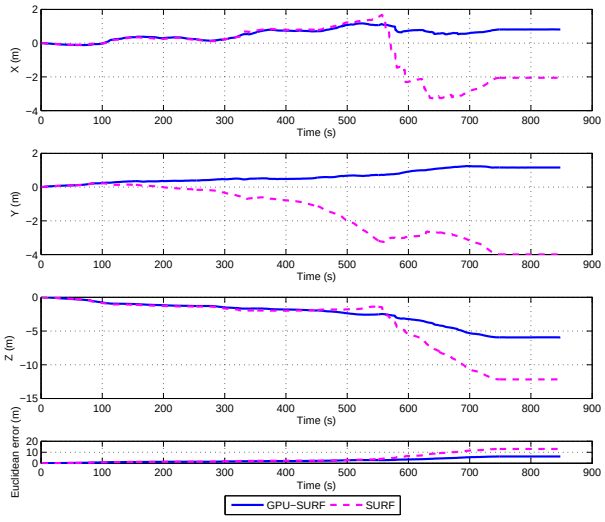
As we stated in the introduction, our interest in this algorithm is it's use in mobile robot navigation. To this end, the current code has been integrated into code that processes stereo images, tracks features through the images and produces an estimate of the camera's motion. We ran the algorithm using both SURF and GPU-SURF. The results are shown in Figure 14. All settings passed to the algorithm were the same, the only difference was the feature detection/description algorithm. This is not a rigorous evaluation of one algorithm against another, it is simply a demonstration that our GPU-SURF implementation can be used for tracking features through image sequences. The GPU code was called synchronously and GPU computations were not overlapped with CPU computations. Still, this resulted in a general speedup. The images used were 512×384 and the SURF algorithm ran at 3.7 Hz, while the GPU-SURF algorithm ran at 9.1 Hz.

5 Conclusions

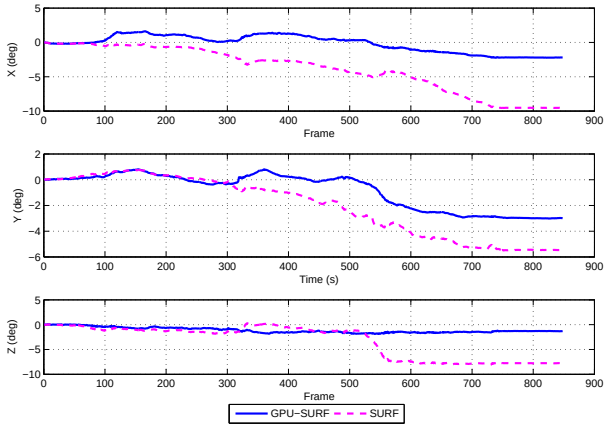
We have successfully implemented a version of the SURF feature detector on the GPU. Where possible, we have verified that our implementation matches the original paper and validated our results against a parallel CPU implementation. Over the next few weeks, we will be integrating the algorithm into a real-time robot navigation system. This will involve the following steps:



(a) Top-down view of the robot's path



(b) Translational error



(c) Rotational error

Figure 14: Comparison of SURF and GPU-SURF on a short visual odometry dataset.

1. Refactoring of the code to embed it in a C++ framework for autonomous robot navigation.
2. Implementation of feature matching to find matches between features in stereo images.
3. Use of CUDA streams to overlap image transfer and GPU computation.
4. Asynchronous execution of CPU and GPU code.

Once the code is integrated into this framework, we can perform the evaluation to determine the performance of our algorithm compared to other feature tracking schemes.

References

[1] David Nistér and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2161–2168, Washington, DC, USA, 2006. IEEE Computer Society.

[2] T. D. Barfoot. Online visual motion estimation using FastSLAM with SIFT features. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 579–585, 2005.

[3] M. Brown and D.G. Lowe. Recognising panoramas. *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1218–1225 vol.2, Oct. 2003.

[4] Stephen Se and Piotr Jasiobedzki. Stereo-vision based 3d modeling and localization for unmanned vehicles. *International Journal of Intelligent Control and Systems*, 13(1):47–58, March 2008. Special Issue on Field Robotics and Intelligent Systems.

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

[6] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110(3):346–359, 2008.

[7] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23(1):3, 2006.

[8] Kurt Konolige, Motilal Agrawal, and Joan Solà. Large scale visual odometry for rough terrain. In *Proceedings of the International Symposium on Research in Robotics (ISRR)*, November 2007.

[9] N. Cornelis and L. Van Gool. Fast scale invariant feature detection and matching on programmable graphics hardware. pages 1–8, June 2008.

[10] Timothy B. Terriberry, Lindley M. French, and John Helmsen. GPU accelerating speeded-up robust features. In *Proceedings of the 4th International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08)*, pages 355—362, Atlanta, Georgia, June 2008.

[11] Motilal Agrawal, Kurt Konolige, and Morten Blas. Censure: Center surround extremas for realtime feature detection and matching. In *Computer Vision ECCV 2008*, volume 5305/2008 of *Lecture Notes in Computer Science*, pages 102–115. Springer Berlin / Heidelberg, 2008.

[12] Changchang Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). http://cs.unc.edu/~ccwu/siftgpu, 2007.

[13] Marten Bj orkman. A CUDA implementation of SIFT. http://www.csc.kth.se/~celle/, 2009.

[14] Chris Evans. Open source SURF feature extraction library. http://code.google.com/p/opensurf1/, 2009.