Astronomical Image Warping and Convolution

Brad Huang, Jonathan Kuck

***Abstract*—We prototype astronomical image processing language primitives in this paper. We implemented primitives for warping and convolving astronomical images. In the context of a fully developed astronomical image processing language our primitives will facilitate easy creation and adjustment of high performance image processing pipelines. We measured a performance improvement of 10x using our warp and 5x using our convolution over a current astronomical image processing system.**

I. Introduction

The Large Synoptic Survey Telescope (LSST) is currently under construction in Chile and scheduled to begin operating by the end of 2022. The LSST will survey 20,000 square degrees of sky (nearly half the entire sky) for 10 years, imaging this entire area roughly every 3-4 nights. Over the course of the survey every sky location will be imaged 1,000 times with the 3.2 gigapixel camera. Every night the LSST will capture 1,000 exposures, each composed of two separate images, and produce 15 terabytes of data.

The LSST has two main goals, building a high resolution representation of the sky and detecting changes in real time. At every location high signal to noise ratio images will be produced by stacking many exposures and best seeing images will be created by selectively stacking exposures with clean point spread functions. To detect changes in real time every exposure will be differenced against a stacked image of the same location. These transient differences may include moving objects or fixed objects whose brightness has changed (supernovae). Some of these transients may disappear quickly so a 60 second window has been allocated for detection. Within this time bound other telescopes will be alerted for follow up study.

Given the quantity of image data that will be processed every night image processing efficiency is critical, but differencing quality cannot be sacrificed. It is expected that 10 million alerts will be sent out every night. False positives must be kept to a minimum so that truly interesting discoveries can be made. There is an inherent trade-off between efficiency and quality, the algorithmic complexity of the differencing pipeline can be reduced to improve efficiency but this may compromise quality. Improving efficiency thus has the potential to reduce compute time, required resource, or improve the differencing quality.

When computing the difference between the just captured science exposure and template image (stacked image representing the previous state of the sky) both images are fed into a pipeline. First the template image is warped to the science exposure so that both images are in the same coordinate system. Next the difference between the two images' point spread functions must be accounted for. This is done by computing a psf matching kernel that varies spatially throughout the image. The psf matching kernel is then convolved with the template image. The warped and psf matched template image is then subtracted from the science exposure. This difference image is then cross-correlated with the psf of the science exposure to find a maximum likelihood estimate that a difference exists at every pixel. Finally, differences are detected as groups of connected pixels whose maximum likelihood difference estimate is above a threshold.

We have prototyped a language that implements the computationally expensive steps involved in astronomical image differencing. These steps are image warping and convolution. After benchmarking an example LSST differencing pipeline we found that 50% of computation time is spent on the psf matching convolution while 25% is spent warping the template image to the science exposure. Our image warping and convolution language primitives improve differencing performance. Eventually, as part of a complete astronomical imaging language, they may facilitating exploration of the trade-off between end to end pipeline performance and differencing quality.

Our warper implementation for the image warping problem relied on OpenGL and GLSL at its core. The implementation will be detailed in the next secion. A Warper instance can be used to repeatedly warp an image given at its initialization time. Kernel sizes and vertex fetching methods can be tuned, making it easy to experiment with tradeoffs between quality and performance of the warp.

Our convolution language primitive performs convolution with a spatially varying kernel. The kernel is the linear combination of fixed basis kernels weighted by polynomials of output image coordinates. The kernel size, degree of polynomial weighting functions, number of basis kernels, machine's vector width, and floating point precision of the basis kernels are specified at compile time. We generate code that takes specific basis kernels and polynomial coefficients at runtime and performs the convolution.

II. Image Warping Implementation

A. Image warping using texture mapping

Figure 1 shows the high level picture of how the original pipeline performs image warping. Before photometric alignment and image differencing the pipeline first needs to align the positions of the stars in the two images. Grid points are evenly sampled across the science image (bottom right), and the recorded difference in telescope position and angle between the times that the two pictures were taken is used to compute the corresponding positions of the grid points on

the template image (left). The transformation is computed by function calls to the LSST software stack and we do not focus on the optimization of this process.

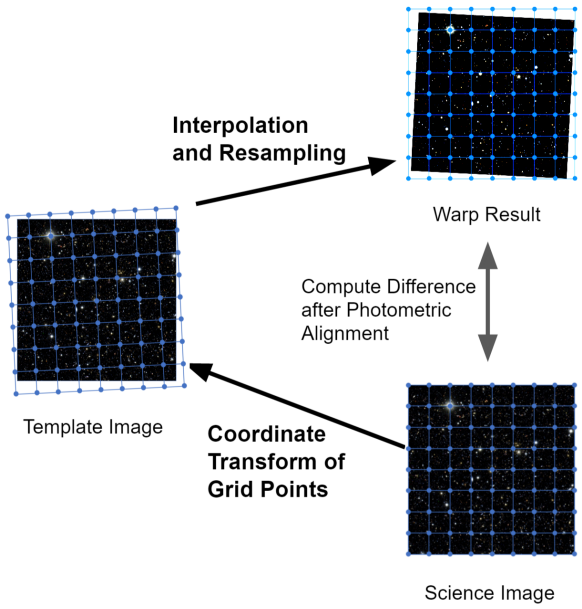

Fig. 1: Image warping as performed in the astrometric alignment part of the current pipeline. The goal of the image differencing pipeline is to compute the difference between the science image, the picture taken for the current sky, and the template image, the picture aggregated from previous records of roughly the same part of sky. The pipeline performs image warping to align the positions of objects in the template image to those in the science image.

The pipeline then computes the values of each plane for each pixel in the warp result. The FITS image format that LSST is using includes three planes, the image plane representing received intensity as a single precision floating point value, the variance plane representing variance of the intensity in the given pixel as a single precision floating point value, and the 16 bit mask plane containing flags such as the pixel should be discarded because it lies on an image edge or was hit with a cosmic ray. For each pixel in the warp result, its corresponding position in the template image is computed from linearly interpolating the transformed grid point coordinates. The pipeline then filters the surrounding area with a 2D 8x8 Lanczos kernel and normalizes the area occupied by the given pixel in the template image to obtain the final values of each plane.

Our implementation framed the image warping process as a texture mapping problem in OpenGL and GLSL. The linear interpolation of coordinates and kernel filtering steps in the original pipeline resembled a standard texture mapping process. If we view the template image as a texture to filter on and resample from, the coordinates of grid points in the science image would be analogous to final pixel coordinates to be shaded, and the coordinates that these correspond to in the template image would be analogous to texture coordinates. We can then let OpenGL handle the linear interpolation of template coordinates for each output pixel, and focus on developing different resampling strategies in the GLSL shader. Figure 2 shows the resampling process in the original pipeline modeled as texture mapping.

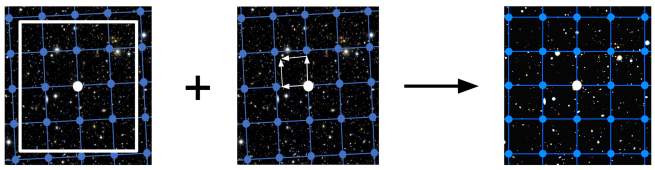

Fig. 2: Resampling broken down into two steps, intensity calculation and area normalization. The white dot in the center of the rightmost image represents the fragment to be shaded, and those in the other two images represent its corresponding location on the texture. Left to right: filtering with a square kenel on the texture, the corresponding area of one pixel, and an example output of the shader.

B. Intensity Calculation

The original pipeline calculates the average intensity of a pixel in the warp result using 2D Lanczos kernel of size 8x8. We assumed the Lanczos kernel values in the X and Y direction to be independent to compute the 2D kernel. For every pixel in the warp result, the image plane values came directly from interpolating the 8x8 texels surrounding the position that the pixel corresponds to in the texture. We treated the 64 pixels as independent variables, so the variance plane value can be computed as the interpolation of the 8x8 variance texels with the squared kernel values as coefficients. Finally, the mask plane was calculated as the union (OR operation) of mask plane bits from all the neighboring texels with positive kernel values.

We implemented a shader program following the exact same method in the original pipeline. Since this method fetches each of the 64 texels in the texture using GL_NEAREST and interpolates them individually with kernel values, we view this as the direct texture fetching method. However, we can reduce the number of texture fetching operations in the process by taking advantage of the bilinear interpolation of texels as performed by OpenGL using GL_LINEAR. If we decompose the kernel into the combination of 16 two-by-two pixel grids, for every grid of four pixels, we can use the relative weights of each pixel to determine a texture coordinate that we should fetch to best approximates the pixels surrounding it. We can then weight the fetched result by the sum of the original kernel values in the four pixels.

In the bilinear texture fetching method, we would need to do more computations on our own, but can reduce the number of texture fetching by a factor of four. However, determining the mask plane bits using the bilinear method would be difficult as OpenGL does not perform the desired OR operation when bilinearly interpolating to fetch the texels.

C. Area Normalization

Additionally, the resampling included an area normalization step for the image and variance planes. We calculated the parallelogram area on the texture corresponding to each pixel at a given location in the fragment shader using derivatives of texture coordinates, and then used the area to scale the image and variance values as filtered from the texture.

III. Image Warping Evaluation

A. Quality Evaluation

Evaluating the quality of image warping output is difficult. The best warp results are ones that produce the most informative analysis results with the entire pipeline. However, we did not integrate our warping into a complete differencing pipeline and cannot measure its impact on differencing analysis. Instead, we used Peak Signal-to-Noise Ratio (PSNR) to evaluate how closely our results matched reference output from the LSST pipeline.

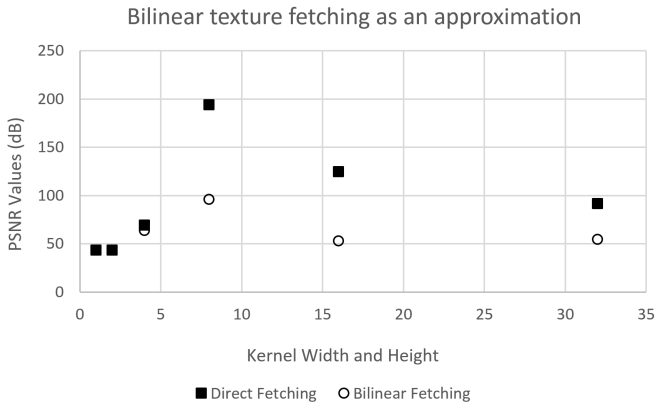

Fig. 3: Pixel Signal-to-Noise Ratio (PSNR) as Evaluation Metric. We normalized the range of values on the image plane to be between zero and one, and then calculated the PSNR on the image plane. The kernel sizes sampled are 1x1, 2x2, 4x4, 8x8, 16x16 and 32x32 with the x-axis representing width. Solid rectangles represents PSNR values of the results from the direct fetching method, while the empty circle represents those from the bilinear fetching method.

Observe from Figure 1 that the outputs with 8x8 kernel size have the highest PSNR among different kernel sizes with the same fetching method. This is an expected result since the original pipeline used an 8x8 Lanczos kernel. The reduced PSNR values with bilinear texture fetching method show that the method is only an approximation to the original pipeline, yielding results quite different from the reference output except in the case of an 8x8 kernel.

B. Efficiency Measure

Running the warping part of the original pipeline on an Intel 4th Gen Core took around 5 seconds for image size of 2046x4094. In comparison, we tested our implementation on a machine with Intel Core i7-4510U CPU with an Nvidia GeForce 840M GPU. Moving most of the calculations in the resampling process from the CPU to the GPU reduced the runtime by 10x and 20x, respectively for direct and bilinear texture fetching methods. In addition, from Figure 2 we can see that, although results from bilinear texture fetching were only an approximation, the runtimes were further reduced by a factor of 2 to 3 compared to the direct fetching method.

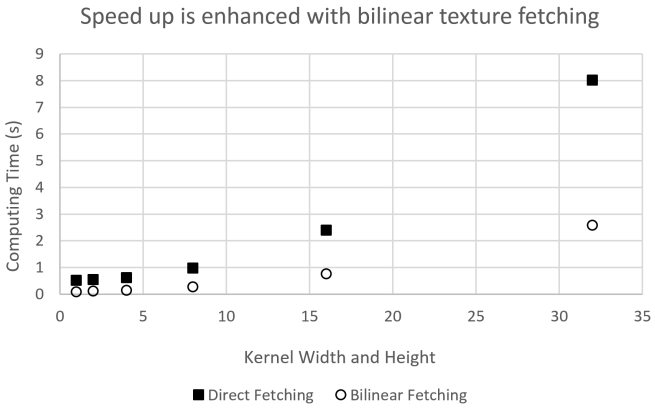

Fig. 4: Runtime reduced with our implementation, especially using the bilinear texture fetching method. The tests were run on kernel sizes of 1x1, 2x2, 4x4, 8x8, 16x16 and 32x32 with the x-axis representing width, and showed a quadratic relationship between the computing time and the kernel size.

However, resampling in the GPU required us to send the texture data to the GPU and read the final results back from the GPU, and each involved transferring a total 80MB of data for the three planes. Table 1 shows that this is an expensive process that took more time (around one second) than resampling for kernels of sizes up to 8x8, even though it is constant time for different sizes.

IV. Convolution

A. Overview

The convolution used for psf matching in the LSST image differencing pipeline uses a kernel that is a spatially varying linear combination of fixed basis kernels. Half the time in our reference LSST differencing pipeline is spent on a single convolution of this type. This specific convolution is also used elsewhere in astronomical image processing. We implemented this convolution as a language primitive in our prototype astronomy image processing language because it is the largest bottleneck in the differencing pipeline and also used

Kernel Size	Resampling Time Only (s)	Include Data Transfer Time (s)
0	0.0431 ± 0.0062	1.0807 ± 0.1579
1x1	0.0900 ± 0.0119	1.1016 ± 0.1559
2x2	0.1235 ± 0.0100	1.1131 ± 0.1325
4x4	0.2089 ± 0.0098	1.1747 ± 0.1384
8x8	0.5744 ± 0.0105	1.527 ± 0.1452
16x16	1.9914 ± 0.0115	2.9456 ± 0.1580
32x32	7.597 ± 0.0296	8.6285 ± 0.2110
64x64	30.4196 ± 0.150	31.5397 ± 0.2559

TABLE I: We timed 30 runs of the warping with the direct fetching method, comparing the time the shaders took to resample, and the time that includes transferring data between CPU and GPU. A kernel size of 0 indicates a direct texture fetch of the nearest texel without any resampling

elsewhere. The kernel is the linear combination of a set of basis kernels, all of the same size, weighted by 2 dimensional polynomials with the image coordinates for variables.

LSST images record three separate planes, image, mask, and variance. Convolution of the image plane is standard. Convolution of the variance plane is treated as finding the variance of the weighted sum of independent random variables, which is equivalent to convolving the variance plane with the square of the kernel values used on the image plane. The mask output is the bitwise or each input mask that overlaps a non-zero kernel value. Each output mask pixel has every flag set that was set in any mask pixel that contributed to the corresponding output image value.

The parameters required to specify this convolution are the basis kernels and the weighting polynomials. The number of basis kernels, size of the basis kernels, and number of coefficients in the weighting polynomials are implicit parameters. We implemented our convolution using Terra to meta-program with parameters specified at compile time. The following convolution parameters must be set at compile time:

- Floating point precision of the spatially varying kernel calculation
- Treatment of denormal floating point kernel values
- SIMD vector width
- Whether to use bilinear interpolation of the spatially varying kernel between points on a sparse grid where it is precisely calculated
- Whether to refactor the basis kernels into a smaller set of basis kernels when there are more original basis kernels than polynomial terms

These parameters may be specified at either compile time or runtime:

- Number of basis kernels
- Kernel size
- Kernel values
- Degree of polynomial weighting functions
- Polynomial coefficients

In this section we present the performance impact of parameters that must be set at compile time and explore the tradeoff in performance and functionality when the other parameters are moved between compile time and runtime.

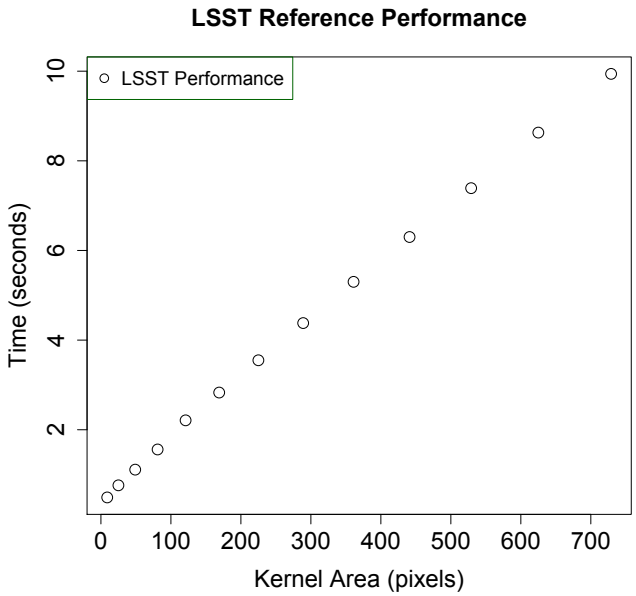

Fig. 5: This graph shows current LSST convolution performance using a kernel that is the spatially varying linear combination of 5 basis kernels. The spatial weighting functions are 3rd degree polynomials. The kernel is explicitly evaluated at every pixel location in the image without interpolation.

All Terra code was compiled using Terra release-2015-08-03. We ran all convolution performance tests on a single Intel core i7-4790 CPU @ 3.60GHz. The LSST camera sensor is composed of 189 chips and the image differencing pipeline is planned to run on 378 cores, each processing half the image from a single sensor chip. In this paper we focus on single core performance because of the inherent parallelism in this setup. However, we have separately explored multi-core performance of linear combination kernels implemented in Halide and found that performance scales well to multiple cores.

As a reference, figure 5 shows the performance of the current LSST spatially varying linear combination convolution implementation. Five basis kernels of size 23x23 with 3rd degree weighting polynomials were used in this example.

B. Compile Time Optimizations

In this section we look at the performance impact of optimization that must be set at compile time. All examples in this section use the spatially varying linear combination of 5 basis kernels specified as arrays of values at runtime. Third degree polynomials are used as weighting functions.

First we consider the effect denormal kernel values can have on performance. In this example our basis kernels are

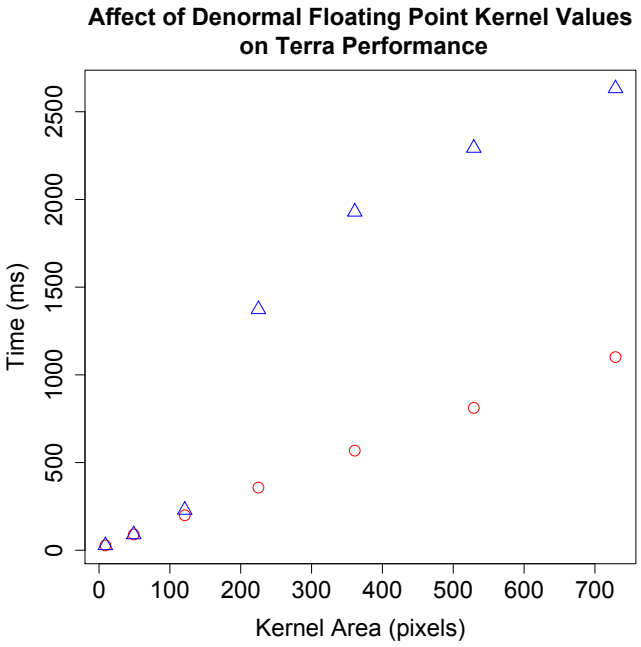

Fig. 6: This graph shows Terra convolution performance when using basis kernels that contain denormal single precision floating point values and performance of the same convolution with denormal basis kernel values set to zero. We used a spatially varying linear combination kernel composed of five basis kernels. The spatial weighting functions were 3rd degree polynomials.

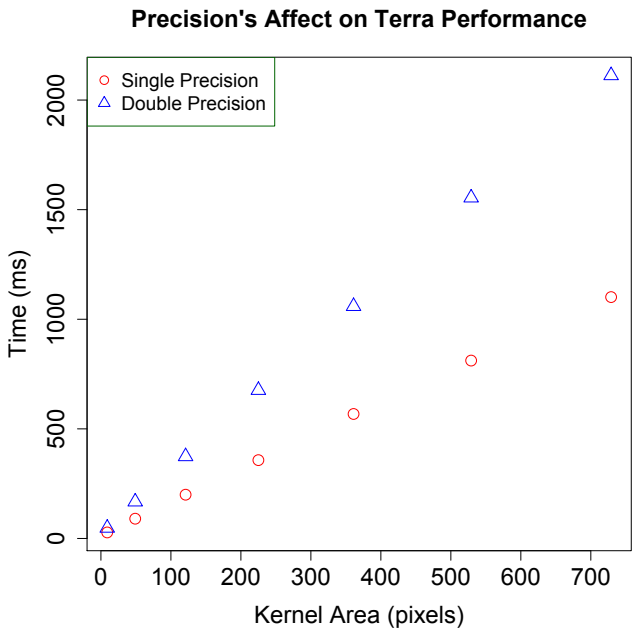

Fig. 7: This graph shows Terra convolution performance when calculating kernel values using single precision and double precision floating point numbers. We used a spatially varying linear combination kernel composed of five basis kernels. The spatial weighting functions were 3rd degree polynomials.

2 dimensional Gaussians with standard deviations in the x dimension of 2, .5, .5, .5, and 4 pixels, standard deviations in the y dimension of 2, 4, 4, 4, and 4 pixels, and rotations of 0, 0, $\pi/4$, $\pi/2$, and 0 radians. These basis kernels were used because they were chosen in an LSST program that demonstrates timing the LSST spatially varying linear combination convolution. The basis kernels are stored as single precision floating point numbers and the linear combination of basis kernels is computed using single precision arithmetic. We found that these kernels contain denormalized numbers with absolute values less than 10^{-38}. Figure 6 shows performance with denormal numbers present and with all kernel values less than 10^{-30} set to zero. The three smallest kernels have nearly identical performance in both cases. For these three kernels 0, 0, and .3 percent of all basis kernel values are denormal numbers. The percent of basis kernel values that are denormal numbers increases with kernel size in this example. There is a significant performance difference for the next four kernel sizes. Among these kernels 8, 18, 24, and 29 percent of all basis kernel values are denormal numbers. These percentages do not correspond with the speedup acheived by zeroing denormal numbers. Speedups, in the same order, are 3.8, 3.4, 2.8, and 2.4x. We are not sure how to account for this, but if denormal numbers may be present in basis kernel values it is clear they should be set to zero.

Next we consider the affect of floating point precision on convolution performance. In this example we use the same Gaussian basis kernels as in the last but set all single precision kernel values less than 10^{-30} to zero. There are no denormal double precision kernel values present in any size kernel so we leave them unadjusted. Figure 7 compares performance when double vs. single precision values are used to represent the basis kernels and perform the linear combination. Execution time using double precision kernels is nearly double that of single precision. For our test image and example kernel of size 19x19 the maximum percent error among image pixels, compared to the reference LSST output, was $1.13\text{x}10^{-4}$ using double precision and $1.63\text{x}10^{-4}$ using single precision.

The performance benefit of vectorization is shown in Figure 9. In this example denormal kernel values are set to zero, kernel values are calculated with single precision, and a kernel size of 23x23 is used.

When the number of basis kernels is larger than the number polynomial terms, convolution can be performed with a set of refactored kernels. The number of refactored kernels will be equal to the number of polynomial terms. Take for example the set of b basis kernels $\{K_1, K_2, ..., K_b\}$. Each basis kernel has a corresponding n^{th} degree weighting polynomial in the set $\{P_1, P_2, ..., P_b\}$. Each polynomial has $\frac{(n+1)(n+2)}{2}$ terms. We refer to the j^{th} coefficient of the i^{th} polynomial as $P_i{}^j$ and

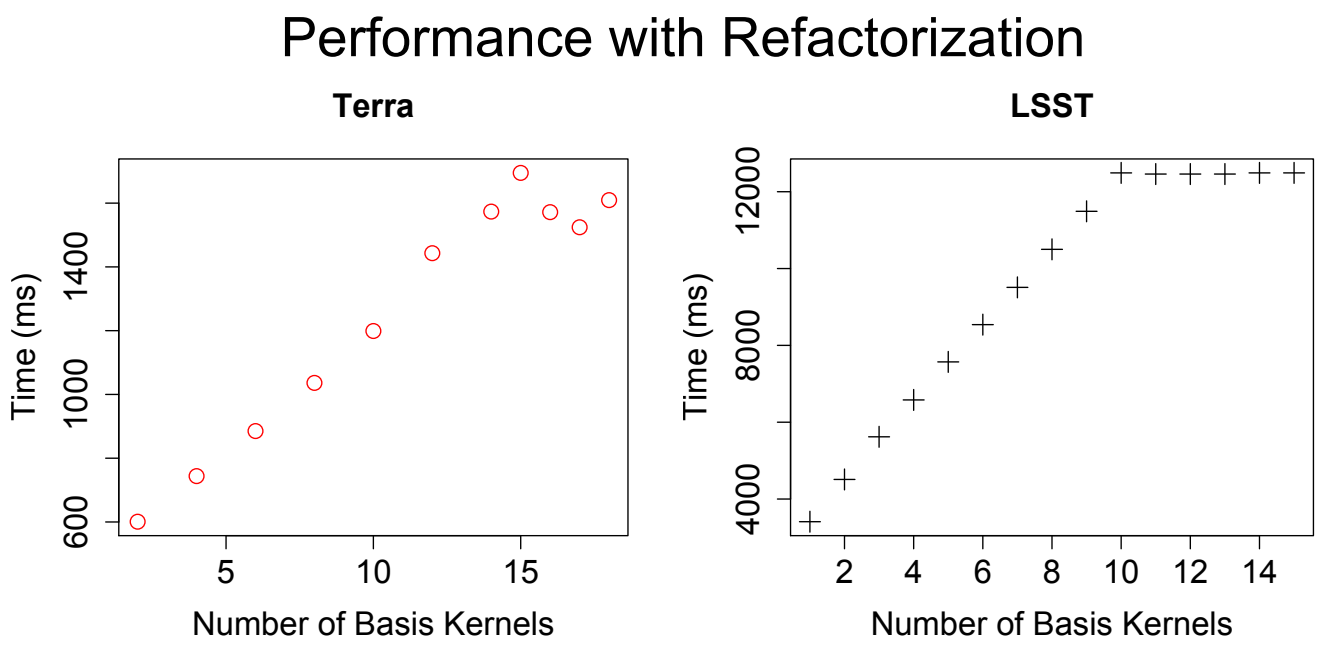

Fig. 8: The right graphs shows LSST convolution performance using kernel refactorization for kernels with size greater than 10. The right graph shows Terra performance, hand chosen to use refactorization when the kernel size reaches 15 and it improves performance. The spatial weighting polynomials each had 10 terms. We used a 23x23 spatially varying linear combination kernel, composed of five basis kernels.

the remaining component of the j^{th} term as ${P_{xy}}^j$ (which will be some combination of x and y, $x^c y^d$ where $c + d = n$ and the same for all weighting polynomials). The spatially varying linear combination of the basis kernels at location (x, y) in the image is then given by:

$$K_{total} = \sum_{i=1}^{b} K_i \sum_{j=1}^{(n+1)(n+2)/2} P_i^j {P_{xy}}^j \quad (1)$$

$$K_{total} = \sum_{j=1}^{(n+1)(n+2)/2} {P_{xy}}^j \sum_{i=1}^{b} K_i P_i^j \quad (2)$$

When the number of basis kernels b is larger than the number of terms in the weighting polynomials $\frac{(n+1)(n+2)}{2}$, refactoring the kernels reduces the number of basis kernels. Equation 2 shows the refactorization we employ. After refactoring there will be $\frac{(n+1)(n+2)}{2}$ new basis kernels that are linear combinations of the original basis kernels. The l^{th} refactored kernel R_l is given by:

$$R_l = \sum_{i=1}^{b} K_i P_i^l \quad (3)$$

The weighting polynomials are also changed, but the new l^{th} weighting polynomial is simply ${P_{xy}}^l$. Figure 8 shows Terra and LSST performance using refactorization. LSST performance benefits from refactorization immediately when the number of basis kernels is larger than the number of polynomial terms. Our Terra implementation does not benefit from refactorization until the kernel size reaches 15. We are not sure why our refactorization performance is suboptimal, and leave improvement as an area for future work.

C. Convolution Parameters that may be Set at Compile or Run Time

In this section we consider the performance impact of parameters that may be set at compile time or runtime:

- Number of basis kernels
- Kernel size
- Kernel values
- Degree of polynomial weighting functions
- Polynomial coefficients

When the number of basis kernels is known at compile time we can unroll the innermost convolution loop that calculates the linear combination of basis kernel values at every kernel location. This loop cannot be unrolled when the number of basis kernels is specified at runtime, which results in a significant performance penalty. In the case of convolution with a 23x23 kernel that is the linear of combination of 5 basis kernels each weighted by a third degree polynomial, the execution time with unrolling when the number of basis kernels is known at compile time is 810ms while the execution time without unrolling is 13s, 16 times slower. This performance penalty is large because we have introduced an additional loop that

Terra Vector Performance

Vector Width	1	2	4	8
Execution Time (ms)	4700	3000	1300	800
Speedup Over No Vectorization	1x	1.6x	3.6x	5.9x

Fig. 9: This table shows the effect of vectorization on Terra convolution performance. We used a 23x23 spatially varying linear combination kernel, composed of five basis kernels. The spatial weighting functions were 3rd degree polynomials. Denormal kernel values were set to zero and the kernel was calculated using single precision.

will be executed (image area)x(kernel area) times. A better alternative when the number of basis kernels is unkown at compile time is to compute the convolution with each fixed basis kernel for a specific output point separately before adding these results. This strategy moves the additional loop outside the loops over the kernel dimensions. This method reduces the execution time without unrolling in our example to 2.5s, now three times slower than with the loop unrolled. This is still a significant performance penalty when the number of basis kernels is unknown at compile time.

The kernel size is used to determine loop bounds for the convolution at a single point. We found that specifying these bounds at compile time results in a 20% overall performance improvement. In the case of a 23x23 kernel with 5 basis kernels and 3rd degree polynomials for weighting functions, execution time is reduced from 1 second to .8 seconds when the kernel size is known at compile time. We find this result surprising, but believe the compiler can optimize bounds checking on these loops which are executed at every point in the image.

We found that performance is equivalent when the basis kernel values are specified at compile time or runtime for randomly generated basis kernels. When basis kernels have symmetry, causing repeated values, performance can be improved when the kernels are known at compile time. Preliminary results indicate that performance may be improved by nearly a factor of two when kernels have two axes of symmetry. For the particular case of using convolution for psf matching, this optimization is not useful. Our reference LSST psf matching convolution contains 27 basis kernels each weighted by functions with three terms. By refactoring the 27 original basis kernels to form 3 new basis kernels, each a linear combination of the originals, convolution performance is significantly improved. This refactorization must be performed at runtime after the function coefficients are computed, so the basis kernels are unknown at compile time.

Our convolution takes the weighting polynomial coefficients as runtime parameters, which is necessary for our differencing application. We have not found a performance improvement from specifying the coefficients at compile time, given that all coefficients are non zero. Our implementation sets the number of coefficients as a compile time constant. We are not aware of a use case where this number is unknown at compile time and have not explored the performance impact of setting it at runtime.

D. Convolution Summary

The kernel used for psf matching in the LSST image differencing pipeline contains 27 basis kernels of size 23x23 with weighting functions that contain 3 terms. We have not integrated our convolution with the differencing pipeline and made a direct performance comparison. We have run the LSST differencing pipeline on our test machine and timed the convolution step at 9.5 seconds on an 8.4 megapixel image. We also timed a separate LSST convolution composed of 27 basis kernels of size 23x23 with weighting functions that are 1st degree polynomials with 3 terms. We have confirmed our output matches this convolution given different parameters. We find this separate convolution takes 5.4 seconds without interpolation and 3.3 seconds with interpolation over a 10x10 grid on a 3 megapixel image. This performance with interpolation is comparable to the performance we measured in the differencing pipeline. We timed our convolution without interpolation on the 3 megapixel image at .68 seconds, 4.8x faster than LSST with interpolation and 7.9 times faster than LSST without interpolation. This comparison has a layer indirection, but demonstrates that our current convolution performance is roughly 5x faster than LSST performance for the critical psf matching use case. We believe we are missing another factor of two or more due to poor refactorization and lack of interpolation. Figure 10 shows a comparison of LSST and Terra single core convolution performance without interpolation given 5 basis kernels and weighting polynomials with 10 terms for a range of kernel sizes.

V. Future Work

A. Image Warping Limitations

One limitation of the image warping implementation we had is the difficulty in adding different kernels. Since we manually code up the kernel computation in the shaders, every time we want to add a new type of kernel, we would have to create new shaders for the specific kernel, limiting the flexibility of sampling methods.

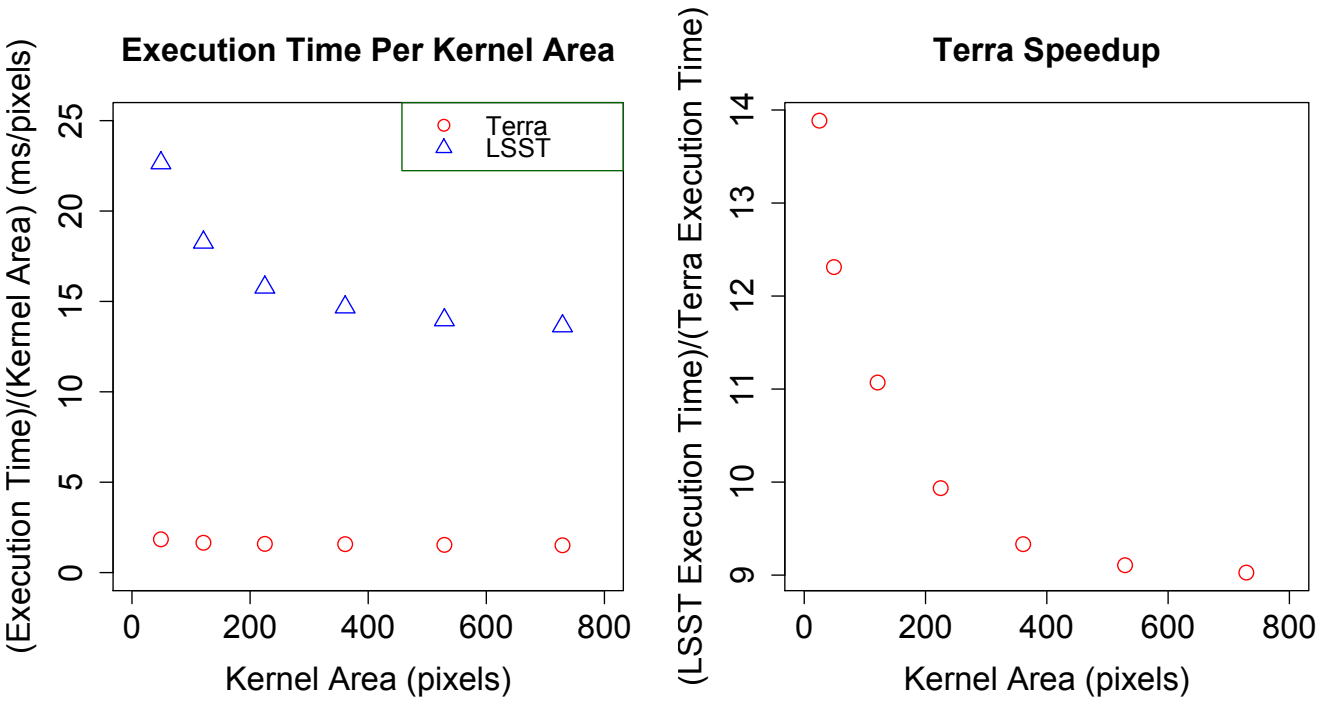

Fig. 10: The left graph shows LSST and Terra convolution execution time relative to the area of the convolving kernel. The right graph shows the Terra convolution speedup for a range of kernel sizes. We used a spatially varying linear combination kernel, composed of five basis kernels. The spatial weighting functions were 3rd degree polynomials. In Terra, the kernel was calculated using single precision and basis kernels were assigned random values, avoiding zero and denormal numbers. Both LSST and Terra convolutions were run on a single core without kernel interpolation.

B. Convolution Limitations

Improving the performance of our convolution refactorization is an area for future work. Another area is implementing interpolation of the spatially varying kernel. LSST convolution performance is increased by 2-3x by computing the exact spatially varying kernel values on a sparse grid and then using bilinear interpolation to compute kernels between grid points. Our initial attempts to implement interpolation have failed to improve performance.

C. Language Completion and LSST Integration

We would like to expand our language to be able to implement an end to end image differencing pipeline. This may enable additional optimizations between steps. With a complete pipeline we could measure and explore the trade off between differencing quality and speed. Integrating our implementations into the existing LSST pipeline is also an area for future work.

VI. Conclusion

We improved the efficiency and versatility of the image warping process through building a simple library using OpenGL and GLSL. Moving computation from the CPU to the GPU reduced warping runtime significantly with both the direct and bilinear texture fetching methods. We had to account for the time transferring data between the processors, but since computation of other parts of the pipeline can also be transferred to the GPU in the future, we expect the data transfer time to be amortized over a larger piece of the pipeline.

We explored the tradeoff between flexibility and performance in astronomical convolution. We found that performance is optimized without restricting functionality in a reference astronomical image difference pipeline by specifying all parameters except kernel values and weighting polynomial coefficients at compile time. We measured a 5x performance improvement using parameters for a convolution that currently occupies half the execution time of a time sensitive image differencing pipeline that will be used in the LSST.