SKYE: Motion Detection & Tracking with SEJITS

Mihir Patil
Armando Fox, Ed.
Charles Markley, Ed.

SKYE: Motion Detection & Tracking with SEJITS

Mihir Patil[1] mihir.patil@berkeley.edu
Armando Fox[1] fox@cs.berkeley.edu
Chick Markley[1] chick@berkeley.edu

[1]UC Berkeley, Department of Electrical Engineering & Computer Science

ABSTRACT

Optimized libraries with callables in Python are very popular for scientific development. They provide an accessible interface, complemented with strong performance. However, there are often operations that these specialized libraries do not support, but are nevertheless necessary in custom applications. This presents a problem, since these unsupported operations are often implemented in Python by the application designer, and therefore are usually unoptimized. Applications that mandate significant performance cannot be effective when they also require these unsupported operations. This performance gap can be greatly reduced with selective embedded just-in-time specialization, or SE**JIT**S. Instead of writing simple Python code for the custom operations, SEJITS allows users to write an optimized version of the routine that can be called in Python. The JIT-specialized code drastically improves performance of the custom routine, allowing the overall computing pipeline for the application to witness a significant speedup. This report explores SKYE, an application that uses OpenCV and a custom SEJITS-optimized method to perform motion detection and tracking.

INTRODUCTION

Scientists often use optimized libraries in Python, as they provide a familiar interface and high performance. The performance benefit of these libraries is significantly increased when they are written with parallelism in mind, using frameworks such as OpenCL or OpenMP.

However, one problem that arises from the use of these libraries is when the application requires a **custom primitive**, a subroutine that is implemented by the application designer exclusively, and not by the optimized library. When working in Python or another scripting language of choice, such custom primitives can be written to produce the correct output, but almost never match the performance of the optimized libraries. Therefore, if one is using optimized libraries and needs to implement one's own custom primitive, the performance of the overall pipeline (the optimized library functions plus the custom primitive) is significantly reduced due to the slowness of the custom primitive.

With SEJITS, this performance gap can be greatly reduced. SKYE is an application of SEJITS specific to motion tracking. In the creation of the SKYE pipeline, the optimized OpenCV library [4] was used extensively. However, there is one routine in SKYE, which is termed the "screener" method that is a custom primitive and thus does not have an OpenCV implementation. Using

SEJITS to optimize the screener method improves the performance of the SKYE pipeline by 10.4x.

While other real-time motion detection algorithms with similar processing speeds and accuracy exist, SKYE is unique in that it incorporates an optimized Python library and a JIT framework to implement custom methods for the application. In other words, it is an example of how an application designer can augment optimized library functionality, while maintaining high performance.

THE ORIGINAL SKYE PIPELINE

The Purpose of SKYE

The SKYE system's function is to track motion from objects, mostly using the OpenCV framework. The original SKYE pipeline achieves this goal, but is inferior in accuracy to subsequent versions. Additionally, it is important to note that this original pipeline involves *only* a series of OpenCV calls, with no custom primitives, and therefore serves as a "control" for accuracy and performance testing of other iterations of the pipeline that use custom primitives and SEJITS specialization.

Computing the Difference of Consecutive Frames

This pipeline starts by taking as input consecutive frames of video footage. The frames are then are made black-and-white for simplicity and a the difference is computed between the two images. This difference image is black where the two images are the same, and white where the two images are different. Figure 1 shows an example of a difference image.

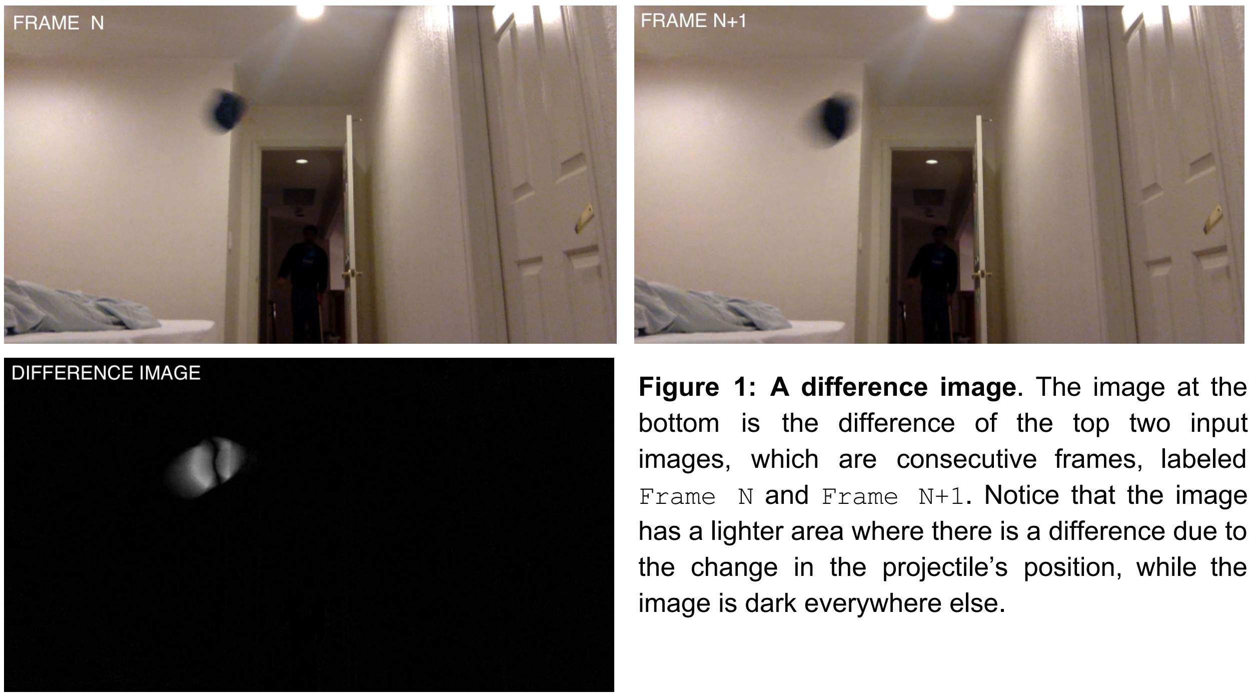


Figure 1: A difference image. The image at the bottom is the difference of the top two input images, which are consecutive frames, labeled `Frame N` and `Frame N+1`. Notice that the image has a lighter area where there is a difference due to the change in the projectile's position, while the image is dark everywhere else.

Blurring and Thresholding the Difference Image

After computing the difference of the two images, the pipeline blurs the difference image to smooth out any anomalous differences due to lighting and other factors, while still keeping the major differences intact. This blurred difference image is then subjected to an OpenCV thresholding operator [4], which makes any pixel with lightness greater than or equal to a particular value (`20` for this implementation) completely white, and makes all pixels less than that value completely black. Figure 2 shows the difference image after the blurring and thresholding steps.

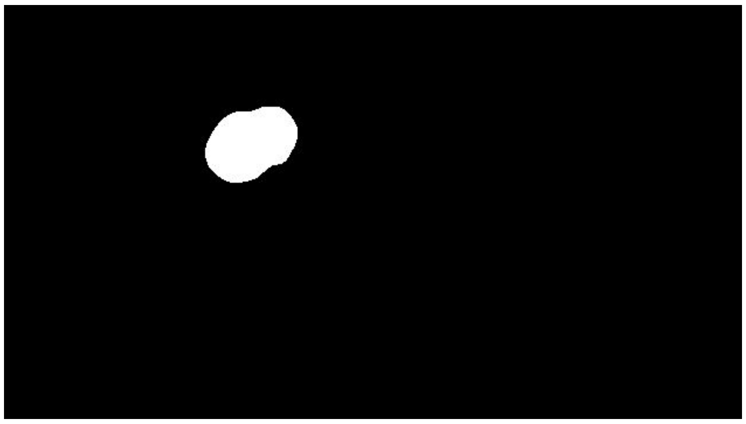


Figure 2: The difference image after blurring and thresholding. The SKYE pipeline takes the difference image, as shown in Figure 1 and blurs it to smooth out any anomalous differences. Then, all the surviving differences are amplified by a thresholding function, such that any pixel with greater than or equal to a specified amount of lightness becomes completely white, while all others are made black.

Finding Contours

After thresholding the difference image, an OpenCV contour operator [4] is applied to the image. As the name of the operator suggests, it helps find contours in the image. Because the pipeline uses a thresholding operator to amplify differences, the contours are quite manifest. Ultimately, this contour operator produces the final bounding boxes around the projectile. Figure 3 shows the final bounding box in blue for a projectile.

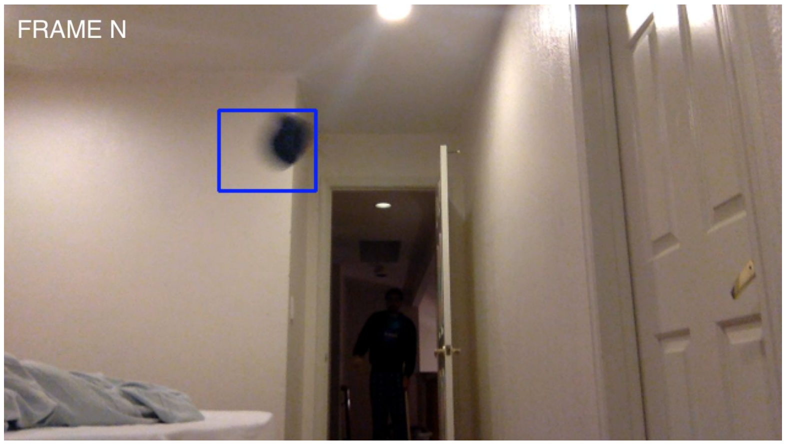


Figure 3: Output of the original SKYE pipeline. Using a series of OpenCV operations, the original SKYE pipeline is able to produce a bounding box, such as the one shown in blue in this image, to describe the location of the projectile. Notice that the blue bounding box is rather large compared to the projectile itself, a problem that is eased by the use of the "screener" method in subsequent SKYE pipelines.

Additionally, Figure 4 shows all the steps for the original pipeline.

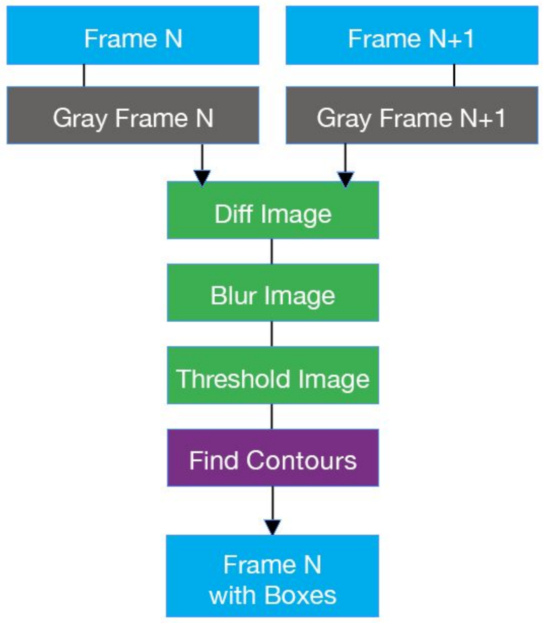


Figure 4: The original pipeline. This is the version of the SKYE motion-detection pipeline does not have any custom primitives and relies solely on OpenCV built-in functions.

Original SKYE Pipeline Performance

All the operations of the pipeline were done using out-of-the-box OpenCV primitives, and therefore SKYE was able to reach a fast processing speed. On a 3 GHz Intel Core i7 processor (Haswell), on Mac OSX, this original pipeline achieved performance of 35.14 frames-per-second.

SLOW SCREENER SKYE PIPELINE

OpenCV with Unoptimized Custom Primitive

Problems with the Original SKYE Pipeline

While the original version of the SKYE pipeline successfully tracks motion, the tracking accuracy of the original version is limited. This can be seen in Figure 3, where the blue bounding box created by the SKYE algorithm is much larger than the projectile itself.

The reason the bounding box is this large is that it is created based on the contours of an image that is a processed version of the difference image; specifically, this is because the difference image has *two representations* of the projectile (one from `Frame N` and one from `Frame N+1`). This means that the bounding box that encompasses both those representations is much larger than the shape of the projectile itself.

The "Screener" Method

This next version of the algorithm aims to solve this issue by using a custom primitive (not included in OpenCV or another optimized library). This custom primitive is known as the **screener method** in this report.

The screener performs a very simple task — whenever the difference at a particular pixel between the Sobel images of the two frames (`SOBEL N` and `SOBEL N+1` in Figure 7) is less than the constant `MIN_PIX_DIFF`, that pixel is thrown out in the resulting image. In other words, this operation eliminates insignificant differences between the Sobel versions of the two images. Note that in the existing implementation, `MIN_PIX_DIFF` was set to `60`, but is a tunable constant depending on the application.

Should the difference at a particular pixel be some value `X` greater than `MIN_PIX_DIFF`, then the screener method checks to see if the pixel is inside one of the contour boxes that were found in the output of the original SKYE pipeline. If the pixel is inside one of those boxes, the pixel is allowed to retain its value `X`. Otherwise; the pixel is also thrown out and assigned the value `0`.

Overall, because the screener method takes Sobel versions of the two images as inputs and only keeps pixels that are shared between the inputs and are in a bounding box that was found in the original SKYE pipeline, it achieves a much more accurate version of the bounding box. Figure 5 shows the output of the screener for the same frames that were used as input to create the thresholded image in Figure 2.

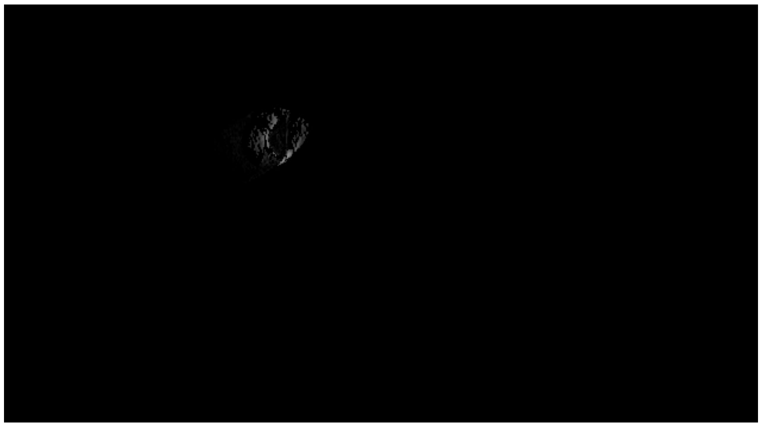


Figure 5: The screener output. This image is the output of the screener method. Note that it has a box-area that is much smaller than that of the image in Figure 2.

Bilateral Filter and Thresholding

The light portion of Figure 5 is very faint, and for the OpenCV contour finding algorithm to properly find contours, the light parts must be prominent.

To remedy this, the SKYE pipeline runs an OpenCV bilateral filter [4] on the image. The bilateral filter is a blurring operator, but also has the property of edge preservation. This operator eliminates most non-edge-defining light pixels, and thus reduces anomalous contouring.

Additionally, to increase the contrast of the image, another OpenCV thresholding operator is applied to the output. The result of these two operators on the output of the screener is shown in Figure 6.

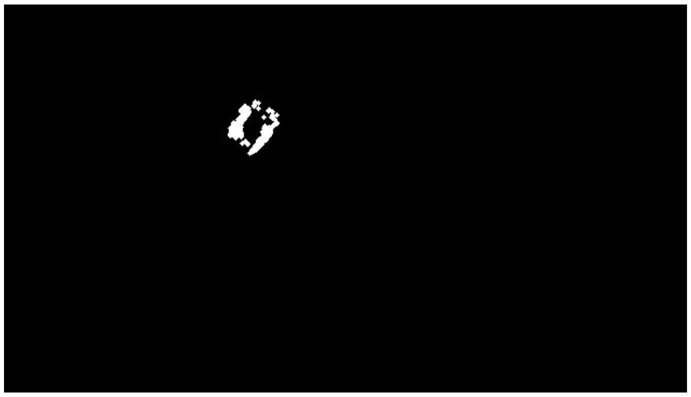


Figure 6: Screener image after bilateral filtering and thresholding. After applying an OpenCV bilateral filter and the OpenCV thresholding function to the output of the screener method (shown in Figure 5), the relevant parts of the image are much more clearly delineated for the contouring algorithm at the end of the pipeline.

Additionally, Figure 7 shows a diagram of this pipeline.

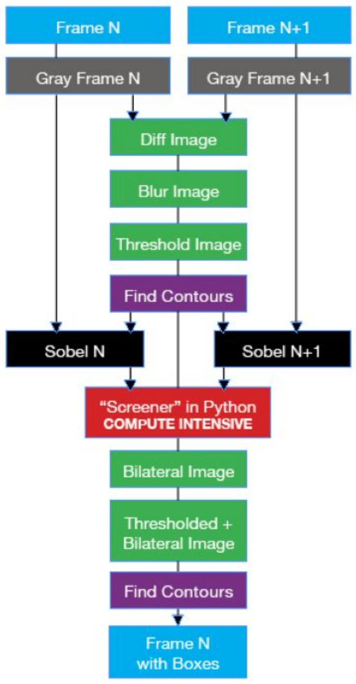


Figure 7: The slow screener. This is the version of the SKYE pipeline that uses the slow Python screener. In the faster version of this pipeline, the screener method is SEJITS optimized.

Final Output with the Screener

After applying the bilateral filter and thresholding operator to the output of the screener method, the pipeline uses the same OpenCV contour operator as before to get bounding boxes. Figure 8 shows the difference between the original SKYE pipeline's output bounding box and the screener SKYE pipeline's bounding box.

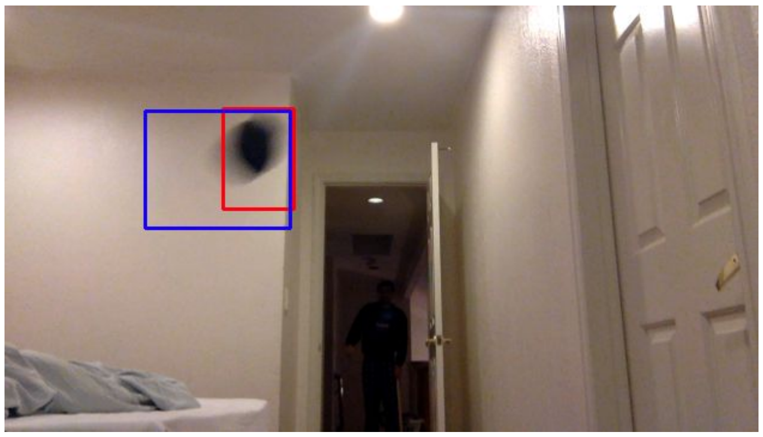


Figure 8: Comparing the outputs of the original SKYE pipeline and the screener SKYE pipeline. In this image, the blue (larger) bounding box is the output of the original SKYE pipeline, and the red (smaller) bounding box is the output of the screener SKYE pipeline. It is clear from the sizes of the boxes that the use of the screener method increases accuracy in motion tracking.

Performance Limitations of the Slow Screener

While this version of the SKYE pipeline has greater accuracy than the original version, the extra computation due to the screener method results in a significant slowdown. This is because the screener method is implemented in Python, and simply cannot match the speed of the optimized routines that are implemented in the OpenCV library.

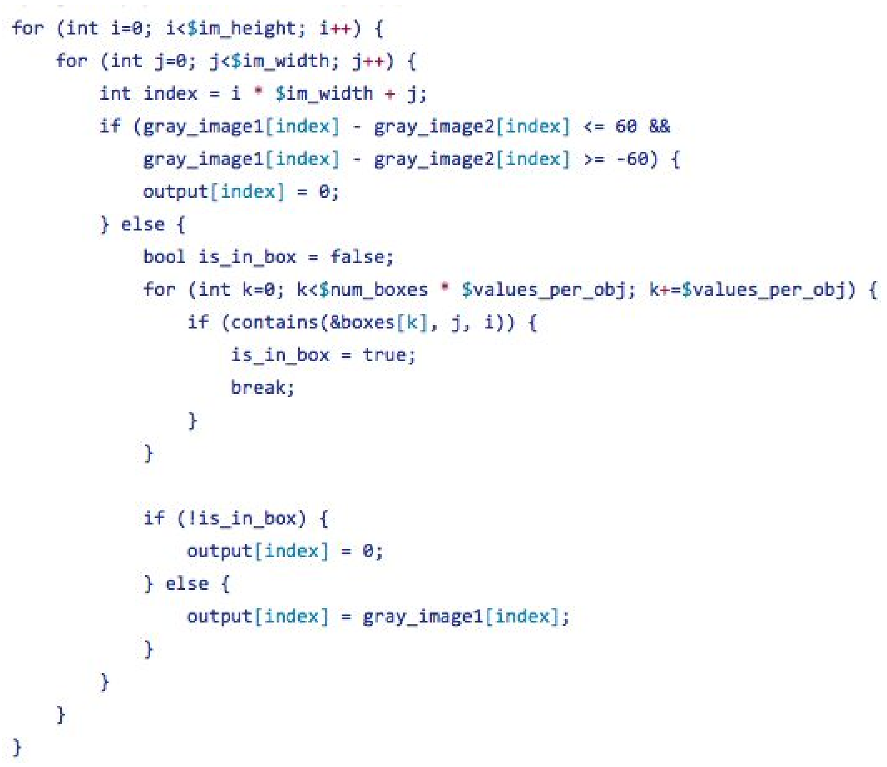
Overall, this pipeline is able to process 2.69 frames-per-second, an order of magnitude worse performance compared to the original pipeline.

FAST SCREENER SKYE PIPELINE

OpenCV with a SEJITS-optimized Custom Primitive

With the screener we get increased tracking accuracy, at the cost of speed. The reason for the loss of speed is the custom primitive screener method, which is implemented in Python, while the rest of the pipeline is comprised of optimized OpenCV functions.

To remedy this, a faster, SEJITS-optimized screener method was used for this pipeline, instead of the unoptimized Python version. This version involved the same screener, with the exact same logic, except it was written through a SEJITS specializer, with a C backend. Figure 9 shows the template code for this optimized screener.



```
for (int i=0; i<$im_height; i++) {
    for (int j=0; j<$im_width; j++) {
        int index = i * $im_width + j;
        if (gray_image1[index] - gray_image2[index] <= 60 &&
            gray_image1[index] - gray_image2[index] >= -60) {
            output[index] = 0;
        } else {
            bool is_in_box = false;
            for (int k=0; k<$num_boxes * $values_per_obj; k+=$values_per_obj) {
                if (contains(&boxes[k], j, i)) {
                    is_in_box = true;
                    break;
                }
            }

            if (!is_in_box) {
                output[index] = 0;
            } else {
                output[index] = gray_image1[index];
            }
        }
    }
}
```

Figure 9: SEJITS Screener Code. The SEJITS template of the screener method maintains the increased tracking accuracy of the Python version, but makes the pipeline 10.4x faster than the second iteration of the pipeline. The `contains` method (not shown) is also written in a C string template in the SEJITS specializer so it can be called in the screener.

The SEJITS specializer is written as a Python string template in C. The template values such as `$num_boxes`, as well as `$im_height` and `$im_width` are code-generated as integer literals just-in-time, based on the number of boxes found as output from the original SKYE pipeline and the dimensions of the image. The output of this generates a C file named `generated.c`, which contains this template with the just-in-time variables filled in. Additionally, the `contains` method, which is not shown in Figure 9, is also added as C code so it can be used in the screener method.

The SEJITS output file `generated.c`, which has the hardcoded values specific to the video stream, is cached using the SEJITS caching mechanism [2], and therefore does not have to be repeatedly code-generated for every pair of input frames. Instead, after the initial code generation of the file, SEJITS runs the screener method directly from the cache.

With the use of SEJITS, the performance of the pipeline improves significantly. After the initial code-generation step is completed for the first pair of input frames, the performance jumps to 28.08 frames-per-second.

PIPELINE PERFORMANCE COMPARISON

The three different pipelines have disparate performances. Because the original pipeline does not have any custom parts, it is faster than the others. However, the latter two pipelines both use the screener and therefore have the same tracking accuracy, but the SEJITS version performs 10.4x better than the Python version in terms of speed. Table 1 shows the processing frames-per-second for each of the pipelines.

SKYE Pipeline Version	**Frames-per-second**
Original (OpenCV only)	35.14
Slow Screener (Python)	2.69
Fast Screener (SEJITS)	28.08

Table 1: Performance comparison of the 3 SKYE pipeline versions. The SEJITS “screener” version performs 10.4x better than the Python “screener” version.

SKYE USAGE

While SKYE was originally meant to be for motion detection and tracking, today it is used as a part of an endeavor known as Project Hermione, which is a system that tracks lecturers as they pace the classroom, allowing a camera to follow them without a camera-person guiding the camera. Hermione takes footage at a wide angle, and during post-processing SKYE is used to identify the movements of the lecturer and thus filter out the unnecessary parts of the footage. The Hermione system is currently in use at UC Berkeley, for EECS 160 (UI Design) and EECS 16A/B, the electrical engineering introductory courses.

While the current implementation of Hermione is suitable for offline processing, planned implementations call for near-real-time streaming, making the performance improvements to SKYE very useful for future development.

POTENTIAL FUTURE WORK

While this 10.4x speedup is significant on its own, further improvements can be made as well. When processing video on a more powerful computer, with many cores, parallelizing the code has the potential to increase performance.

Additionally, future work could try to harness SEJITS's ability to support an OpenCL backend for GPU development. In essence the screener method can potentially be run on GPUs for much faster performance. While GPUs require time to copy to and from, OpenCV methods can also be run on GPUs [4], therefore allowing the entire pipeline to run with just one set of copies to and from the GPU. Additionally, because the same frame is used in two subsequent iterations of the pipeline (i.e. `Frame N` is used when comparing `Frame N-1` to `Frame N`, as well as when comparing `Frame N` to `Frame N+1`), only one image has to be copied to the GPU every iteration, and only four values (the four corners of the bounding box for the image) need to copied back.

RELATED WORK

The idea of writing code in a productivity language, such as Python, while retaining high-level performance is not exclusive to SEJITS. Two notable frameworks that try to achieve this are Weave and Cython.

Weave, a part of the SciPy package, allows users to write C code inline within Python functions [3]. However, while this approach is similar to the way SEJITS is used for SKYE, Weave cannot take advantage of JIT specialization to improve performance, and has the additional effect of cluttering the code with the optimizations it does make. In contrast, SEJITS hides the optimized code away in a specializer, allowing the main application logic to be clean and easy to read [2].

Cython on the other hand allows the user to annotate Python functions to provide information like static types to a C compiler [5]. Like Weave, Cython's performance optimizations can clutter the main logic of the application, and has the additional downside of not being executable through a standard Python compiler [2].

Change detection itself is a common problem in various fields, ranging from traffic safety to medical imaging. Many algorithms exist for change detection offline, which doesn't require fast performance [1,6]. For real-time applications, such as collision detection on roads, algorithms such as the one proposed by Yu & Chen [7] are comparable in speed and performance to SKYE. However, SKYE is unique due to how the application is implemented: the JIT specialization engine takes a optimized Python-only algorithm and improves accuracy without significant drops in performance. Other motion-detection implementations should be able to benefit from this same JIT technique.

CONCLUSION

The majority of the SKYE implementation is completed with a series of calls to optimized methods in the OpenCV library. But the screener method is the part of the implementation that does not have an optimized implementation, and thus significantly slows down the entire pipeline. However, after optimizing the screener method with SEJITS, the SKYE pipeline saw a 10.4x increase in performance overall. In general, with SEJITS, application designers can create optimized versions of custom primitives, which can significantly improve the performance of those applications.

REFERENCES

[1] Ardekani, B.A., Bachman, A.H. et al. (2001). A quantitative comparison of motion detection algorithm of FMRI. Magn Reson. Imaging 19, 959–963.

[2] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: Getting productivity and performance with Selective Embedded JIT Specialization."

[3] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001-2017.

[4] G. Bradski *Dr. Dobb's Journal of Software Tools*

[5] R. Bradshaw, S. Behnel, D. S. Seljebotn, G. Ewing, et al., The Cython compiler, http://cython.org.

[6] R. Radke, S. Andra, O. Al-Kofahi, B. Roysam, "Image change detection algorithms: A systematic survey", *IEEE Trans. Image Process.*, vol. 14, pp. 294-307, Mar. 2005.

[7] Z. Yu, Y. Chen, "A real-time motion detection algorithm for traffic monitoring systems based on consecutive temporal difference", *Proc. 7th Asian Control Conf.*, pp. 1594-1599, 2009-Aug.-2729.