Modern Architecture for Image Processing

Isaac B. Martin[*]

Abstract

This paper is a review of an ongoing research project aimed at solving problems present by large scale image processing computing. Specifically, it seeks to develop a general hardware architecture specifically designed for image processing algorithms. This architecture must improve upon execution time and energy efficiency of existing off-the-shelf processing hardware (CPU and GPU), while avoiding the workload required for FPGAs and ASICS. As a first step in this process, we seek to compare the performance of several common image processing (IP) algorithms on the two most common processing hardwares, CPU and GPU. Using C++, OpenCV and CUDA, we have implemented versions of several IP algorithms (ranging from simple RGB2Gray, to complex K-means and SIFT) for CPU and GPU based execution.

We find that in general, our GPU-oriented implementations perform measurably better than CPU-oriented in terms of execution time. This result is not surprising, as IP algorithms typically lend well to parallelization due to the need for computation on every pixel of an output image, however some algorithms appear to benefit a great deal more than others. Continuing forward, we seek to gather more IP algorithms that will better represent the most frequently used strategies, and learn more about how the balance of parallelization and processing power will impact the effectiveness of our theoretical architecture as a general purpose IP processor.

Keywords: GPGPU, image processing, architecture

1 Introduction

Modern technological advances in devices such as smartphones and networking have created a social environment built upon the sharing of data, and among the most prevalent and is images. Through social networks like Facebook and Flickr, people across the planet are cataloging and sharing their experiences globally, using images to describe what cannot be explained in words. Amidst such a wealth of data, many of these networking and tech giants are looking to use image processing to gather useful information from this vast ocean of pixels. Therein lies the motivation for our work on this project.

As of now, there does not exist a commercially available general processing architecture designed explicitly for image processing applications. For anyone wishing to engage in large scale image processing operations, this may be a serious problem; running a costly algorithm on an architecture to which it is poorly suited results in a great deal of lost energy or time, both of which can cost significant amounts of money. Hence, settling for using a processing cluster made up of high-end CPUs or GPUs for image processing may not be the best course of action. More specialized solutions do exist, such as FPGAs and ASICs, but architectures such as these must be configured manually, or through newly developed tools such as Rigel [Hegarty et al. 2016]. However, almost every solution in this category is meant to be tailored to a small group of algorithms, not a broad area of computing such as Image Processing. A general purpose IP architecture would provide a middle ground between specialization and adaptability, allowing for high performance and energy efficiency for a broad range of algorithms without needing to be reconfigured for every new application. This is our long term goal, to develop such an architecture that is both efficient and adaptable to a broad range of image processing algorithms. Before we get there, however, we must start by understanding what common IP algorithms need in an architecture to execute efficiently.

Over the course of the last six months, we have worked to implement a number of common and important IP algorithms on existing commercial processing architectures, namely CPU and GPU, in order to better understand how these systems benefit or restrict image processing applications. In section 2, we will describe these algorithms, how they work at a high level, and why comparing the different implementations is useful for this discussion. In section 3, we will discuss the details of our experimental software, hardware, and choices for test data. Lastly, in section 4, we will discuss the results of our tests, speculate on the underlying causes and implications, and detail some of our plans for the continuation of our work.

2 Algorithm Implementation

While there are many readily available implementations of common IP algorithms, many of them exist within libraries or projects that have been highly optimized, and had most of their internal workings either obscured from end users or integrated into larger systems. Additionally, hardware specific implementations will have been adapted to take advantage of the underlying architecture. Such issues limit how well publicly available implementations can work for comparison. In order to get a fair measurement, it is necessary to implement these algorithms ourselves. This allows us to guarantee that both versions of each algorithm follow comparable logical structure, avoid using datastructures that require more computation and memory than we need, and strip back our algorithms to only include as much logic as is necessary for a tightly controlled input dataset.

The algorithms for this project were chosen based on the following criteria: 1) How frequently they are used in real-world applications, 2) Whether or not our algorithms can reasonably be adapted to a C++/CUDA implementation, and 3) Difficulty of implementation. As this project is still in the early stages, most of our choices are simple, easy-to-implement but important algorithms. We hope to add more interesting and complex algorithms in the future. Unless stated otherwise, assume that the data for both input and output is of type unsigned char. We will now describe the features of each implemented algorithm.

2.1 RGB Color to Grayscale *(RGB2Gray)*

Converting an RGB color image to grayscale is one of the simplest examples of an IP algorithm, but it is nonetheless common and useful. Each pixel in the output image is calculated by a weighted summation of the color channels in the corresponding input pixel:

$$p_o = 0.299p_{iR} + 0.587p_{iG} + 0.114p_{iB}$$

Where p_o is the output pixel color (one channel), p_{iR} is the input pixel's red value. The coefficients are based on the responses of the rod cells in the human eye to each wavelength of color. This algorithm is interesting because it requires very little memory or computation per output pixel. Our hope is that by using it for comparison,

[*]e-mail: ibmartin@iastate.edu

it may expose features about the two architectures that would be obscured in more rigorous algorithms.

2.2 Color Inversion *(Reverse)*

Color inversion is another simple algorithm similar to RGB2Gray. Each output pixel color channel is calculated by $p_{oX} = 255 - p_{iX}$, which applied to all pixels produces the color negative of the image (p_{iX} is converted to a float). While not complex, calculating the output image requires 0.5 times more memory than for RGB2Gray.

2.3 Gamma Correction *(Gamma)*

Gamma Correction is commonly used to brighten or darken images without distorting color balance. The intensity value of each color channel (3 per pixel) in the output image is calculated by: $p_{oX} = 255(p_{iX}/255)^{\gamma}$, where γ is given as input. Values of $\gamma > 1$ will darken the image, and $\gamma < 1$ will brighten it. This algorithm is interesting because of the potential for non-integral exponential operations, which are more computationally intensive per pixel than either RGB2Gray or Reverse.

2.4 Direct Resize *(dResize)*

Direct Resize is an image scaling algorithm that does not use any interpolation to determine pixel values, instead choosing a single pixel from the source image to use for a given output pixel. This implementation takes advantage of integer truncation, and uses the ratio of the original image to the output image to get an index for the desired input pixel. This strategy is cheap and simple, but results in severe aliasing for significant scale changes.

2.5 Linear Resize *(lResize)*

Linear Resize is another image scaling algorithm, but does make use of linear interpolation to mitigate aliasing in the output image. For each output pixel, lResize uses the same strategy as dResize to select a matching location in the input image, but does not truncate the index. If this float index is significantly far enough in space from an integral index in the input image ($> 10^{-10}$), the algorithm retrieves the four pixels closest to the calculated point. Their values are then weighted by their distance from the calculated point, and summed together.

2.6 Gaussian Blur *(gaussFilter)*

Gaussian blur is the process of blurring an image by a Gaussian equation, and it is extremely important in many image processing applications. Unlike other kinds of blurs, Gaussian blur does not add any artifacts or distortions to the image. It is also vitally important as the Gaussian equation is capable of creating scale invariant smoothed images, which are needed for creating scale-space representations. Scale-space is the notion that complex real world objects exhibit structural details and many different scales, such as how a tree has both fine details in its leaves and an overall shape defined by all its leaves and branches. An accurate scale-space representation allows IP applications to observe these attributes without needing to specify the scales at which they occur. This is an important consideration when we discuss *gaussPyramid* and *SIFT*. The Gaussian equation for image processing is given as

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

where x and y are the distance from the origin on the horizontal and vertical axes, and σ is the standard deviation, or the size of the bell curve created by the function. *gaussFilter* first uses this equation to create a kernel matrix centered at the origin of the Gaussian equation. The size of this kernel can be arbitrary, but for our measurments we use a 7x7 kernel to limit computational complexity. To generate the output image, this kernel is then convolved with the input image at every pixel location.

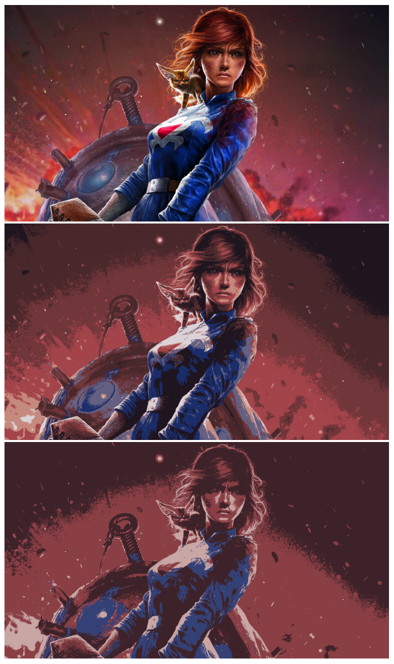
Figure 1: *Examples of the kMeans algorithm run on the test image* ***nausicaa.jpg****. (top): Original image (center): 8-means (bottom): 4-means.*

2.7 Sobel Edge Detection *(sobelFilter)*

Sobel Edge Detection is another convolution-based algorithm that is used to find visible edges in an image by locating areas where neighboring pixels have very different intensities. Two separate convolutions are involved, using one kernel to quantify horizontal pixel gradients, and the other for vertical:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

These two kernels are convolved with the input image at each pixel to get a 2D vector describing the strength of the edge at this pixel, and the length of that vector is supplied to the corresponding output pixel as its intensity. In this implementation however, the intensity of these gradients can have a much wider range than standard pixel values (0 - 255 for a unsigned char based image), so an extra step is necessary to fit that data into range properly. The actual range is data-dependent, and differs from image to image, so this step is necessary to prevent overflow or underflow.

2.8 K-Means *(kMeans)*

K-means is a segmentation algorithm that is used to determine a combination of K colors that best represent every color in the image. The output of this algorithm is the most accurate approximation of the input image using only K distinct colors. EXAMPLE. The general premise behind this algorithm is as follows:

1. Randomly choose K pixels from the input image. Let the colors of these pixels v_k be the initial color value of each "color group" k.
2. For each pixel in the input image p_i, assign it to the color group k with the shortest distance between its color value v_k and the color of p_i.
3. For each color group k, set the color value v_k to the mean-average color of all p_i assigned to group k.
4. If any pixels changed group assignments from previous iterations in step 2, repeat steps 2 and 3. Else, the algorithm has reached convergence. Proceed to step 5.
5. For every input pixel p_i, if p_i is assigned to group k, set the color of the output pixel at the same index p_o to v_k.

The random nature of this algorithm would normally mean that a fair time-based comparison between implementations would require thousands of runs. To account for that, we have adjusted both implementations to use an identical random seed, and execute the loop defined by steps 2, 3 and 4 to run exactly 200 times, regardless of convergence of the algorithm. We hope that this will be sufficient to provide a reasonable comparison, as K-means takes a great deal of time to run on large images. For testing purposes, we provide a K value of 8. An example of *kMeans* is given in Figure 1.

2.9 Gaussian Pyramid *(gaussPyramid)*

Gaussian Pyramid is a preliminary step in creating a scale space representation. There are two primary operations within a loop in *gaussPyramid*: apply Gaussian blur, then subsample. The blur obscures the finer details that are more apparent at lower scales, reducing their impact on subsequent images, and subsampling reduces the size of the image such that shapes that appear at larger scales occupy few pixels, and therefore require traversal of fewer pixels to observe. The bulk of our implementation involves managing the memory for each level of the Gaussian pyramid; the blur and scaling is handled by *gaussFilter* and *lResize* as described previously. Our implementation builds this pyramid, then returns a single image containing every image in the pyramid. The degree of blur and subsampling can be arbitrary, but for our implementation we fix the call to double σ and 0.5X subsampling as these are the most commonly used scaling factors, and it gives us a deterministic limit on the amount of memory our output image requires.

2.10 Scale Invariant Feature Transform *(SIFT)*

Scale Invariant Feature Transform is a complex and powerful algorithm designed to use information from scale-space to learn the defining features of an object such that it can be identified accurately in other images. Proposed by David Lowe [Lowe 1999], the driving factor behind this algorithm is the *keypoint descriptor*, a measurement of the relative pixel gradients centered at local extrema in the scale-space representation. These descriptors are highly robust, completely invariant to scale and orientation changes, and resistant to changes in local image distortions and changes in illumination, factors that few previous approaches could compensate for.

In order to calculate keypoint descriptors for a given image, *SIFT* executes four major tasks on the input [Lowe 2004]:

1. Scale-space extrema detection: This stage builds and searches through the entire scale-space of the input image. Using a difference-of-Gaussian *(doG)* approach (an efficient approximation of Laplacian of Gaussian), the algorithm identifies local extrema in the *doG* as potential keypoints that will be invariant to scale and orientation.
2. Keypoint localization: Each candidate keypoint is measured to determine its actual location and scale, and any keypoint candidates determined to be too unstable eliminated.
3. Orientation assignment: Each keypoint is assigned an orientation based on the pixel gradients around it. For all future operations, the orientation, scale, and location assigned to a keypoint will serve as the "origin" to which all other data is aligned. This makes the keypoints invariant to transformations in these attributes.
4. Keypoint descriptor: The gradients local to the keypoint (in both scale and location) are used to create a group of histograms describing the area around the keypoint. This representation is robust, and can withstand significant shape distortion or illumination changes without losing much accuracy.

This algorithm is rigorous and extensive, requiring a great deal of computation and memory to gather the data it needs. As of now we do not have a working GPU-based implementation, but our CPU-based implementation is functional, including our search algorithm (a variation on k-d tree search called best-bin-first) which will be used to identify previously observed objects in a new image.

3 Details and Methodology

As part of creating a testing environment with we could make reasonable comparisons, we decided to use C++ as our programming language, with CUDA as our GPU programming library. We also chosen to use OpenCV to help handle managing image data, such as loading and writing. However, we have taken steps to limit usage of OpenCV functionality in our algorithms and operate only on the underlying memory of OpenCV data structures manually. By doing this, we ensure OpenCV does not add to the execution time of our algorithms without our knowledge, and it enables us to use the same basic logic to execute on both our CPU and GPU based implementations (CUDA kernels cannot use OpenCV functions or data structures). In the rest of this section, we will describe the details of the computer used to gather our data, and the test images we use.

3.1 Hardware and Software Details

Our execution platform is a PC running Windows 10 OS (64-bit). The major hardware details are as follows:

- CPU: Intel Core i7-6700K CPU @ 4.00GHz
- RAM: 16.0 GB
- GPU: NVIDIA GeForce GTX Titan X (Maxwell), 12GB memory, 3072 CUDA cores, 1000 MHz base clock

It is worth mentioning that while this CPU is new and very powerful, the GPU is more than a year old and not quite state of the art anymore. However, the Titan X line is designed for GPU programming, and we still expect it to be competitive for our purposes.

The details of our software versions are as follows:

- Programming Language: C++
- OpenCV version: 3.0.0
- CUDA version: 7.5

The details of all test images used for this project are given in Table 1. All of the material used for this project, including the test images

Image	Dimensions	Uncompressed Filesize
einstein.png	292 x 285	250 KB
castle.png	512 x 512	786 KB
lena.png	512 x 512	786 KB
boat.png	640 x 480	922 KB
bike.jpg	640 x 480	922 KB
valve.png	640 x 480	922 KB
koala.png	1024 x 768	2.36 MB
nausicaa.jpg	1920 x 1080	6.22 MB
oranges.jpg	4672 x 3104	43.5 MB
mountains.jpg	5215 x 3468	54.2 MB
tiger.jpg	7680 x 4320	100 MB

Table 1: *Dimensions and approximate uncompressed filesize of each test image used in data collection for this project. Filesize is calculated assuming each image is read in as a 3-color channel RGB image using unsigned char as its data type.*

and full data results, are available at: https://github.com/ibmartin/research_project_2016.git

3.2 Runtime Execution Details

The program file Source1.cu contains all of the logic related to gather data about the execution times of our algorithms, excluding *SIFT* which has no CUDA version with which to compare. At the beginning of the main loop, a test image is loaded using OpenCV's imread function, which loads it as a row-major 3-channel unsigned char Mat data structure. The order in which the images are processed is the same as it appears in Table 1. For each image, we run each pair of implementations sequentially, first the CPU-based version followed by the GPU-based version. Timing is independent of any reads or writes to data files; the timer we use to track execution time is started immediately before the call to the algorithm we want to test, and the timer is stopped immediately after the algorithm returns. The time we are using is limited to a resolution of approximately 0.5ms, which may cause somewhat erratic behavior on the simple algorithms running on the smallest images, but we do not perceive this to be of great concern. For each image, this pattern of CPU and GPU versions is looped 10 times for *RGB2Gray, Reverse, Gamma, dResize*, and *lResize*. *gaussFilter* is only run 5 times, and the rest are only run once. We also run two versions of both *dResize* and *lResize*. The first call scales the image by 2.0 in both dimensions, and the second scales the image by 0.5 in both dimensions. This is to determine whether the performance of these implementations relative to each other show a noticeable difference. All of the data in Table 2 is from a single execution. What is not included with this paper are the raw data on execution times, averages, and throughput of each algorithm for each test image. Those will be available at https://github.com/ibmartin/research_project_2016.git.

4 Results and Discussion

Table 2 contains the gist of all our collected data, represented as the value calculated by dividing the mean-average CPU-based implementation time by the mean-average GPU-based implementation time. Values < 1 indicate that the CPU version completed in less time than the GPU version, and vice-versa. In the rest of this section, we will use Table 2 and any other collected data we see fit to mention to assess the relative performance of each pair of algorithms and discuss what we believe the underlying implications may be.

4.1 *RGB2Gray, Reverse, Gamma*

The behavior of *RGB2Gray* on einstein is extremely odd, implying a $100\times$ speedup for CPU over GPU, but we are disinclined to consider this accurate. The timer functions we use for this project have a resolution of $0.5ms$, which is a reasonable execution time for a simple algorithm like *RGB2Gray* running on a small image like einstein. Occasionally, the execution time will be short enough that the timers will measure no time difference, and a zero is entered as the execution time, as we have observed in the past. The rest of the data values appear to be reasonable however.

RGB2Gray: For the CUDA version of this algorithm, one thread is executed per pixel in the output image. Both versions of this algorithm would enjoy high locality in both source data and output data, and there is no discernible advantage for one implementation over the other, except for the parallelization enjoyed by the GPU version. As we'd expect, as the input image grows larger, parallelization will become more valuable, which is supported by the stead growth we see in the larger images.

Reverse: The results of this test are not what would be expected for an algorithm as simple as *Reverse*. This is because of a simple but important design oversight. It could be fixed easily, but for the sake of discussion we will use these results. The GPU version of *Reverse* creates one thread for each color channel in each pixel in the output image. As such, there will be $3\times$ as many threads as pixels. The CPU version iterates over every pixel, operating on three color channels at a time before moving on. Attempting to use so many threads for data with such high locality creates unnecessary overhead costs for the GPU version, while the CPU version only needs to iterate and execute subtractions. This gives the CPU version an advantage for small images. Even so, the parallelization benefit for larger images still bears out.

Gamma: It is not entirely clear why the CUDA version performs so much better here. Both versions are executed per color channel (CPU iterates over them and GPU creates one thread each), and both call the pow function with a non-integral exponent. It may be the way in which the CPU version accesses memory, or the pointer arithmetic instructions, or the ability of the CUDA threads to execute the pow function competitively. This will require more testing to understand, which is part of the reason for this project.

4.2 *dResize, lResize*

dResize: The results for this algorithm are very interesting. For the $\times 2$ calls, both versions appear competitive. Both versions enjoy high memory locality, and there appears to be no inherent advantage for either implementation. The $\times 0.5$ calls show a serious disadvantage for the GPU version, which almost certainly stems from poor source memory locality for CUDA threads. One thread is created per output color channel, but the data needed for each of those threads is spread out over a wider area, resulting in more cache misses and potentially thrashing, especially at smaller scaling factors.

lResize: This algorithm behaves similarly to *dResize*, with the addition that the GPU version enjoys better memory cohesion for accessing neighboring pixels. While this causes it to improve significantly over CPU for the $\times 2$ calls, it still suffers greatly in the $\times 0.5$ calls for the same reasons. This issue is definitely worth keeping in mind in the future. Also, we expect the result for einstein to be another anomaly caused by timer resolution.

Image	*RGB2Gray*	*Reverse*	*Gamma*	*dResize(x2)*	*dResize(x0.5)*	*lResize(x2)*	*lResize(x0.5)*	*gaussFilter*	*sobelFilter*	*kMeans*	*gaussPyramid*
einstein	0.011792	0.500000	10.123154	0.857166	0.285633	4.697648	1.555149	29.931319	20.563271	14.878110	12.555556
castle	1.317527	0.713929	14.318726	1.007629	0.409054	5.350820	0.481232	36.312825	24.642857	9.168963	20.784340
lena	1.221680	0.599808	15.095208	1.007556	0.347736	5.320891	0.608590	36.321993	24.690492	18.689231	21.220578
boat	1.624855	0.817737	14.035977	1.000000	0.260802	5.325793	0.636190	35.658933	24.905961	7.893338	21.199705
bike	1.349596	0.768877	15.389279	1.013024	0.434669	5.404260	0.666500	36.826568	25.270861	17.959786	22.214034
valve	1.944198	0.851611	15.112856	1.033052	0.434669	5.330773	0.707980	36.786297	25.291681	18.195436	22.285450
koala	1.914164	0.979600	17.279905	1.044180	0.634130	5.463870	1.000000	39.198833	26.373058	8.202881	27.448619
nausicaa	2.386807	1.074383	18.166981	1.085451	0.683650	5.582563	1.183663	39.890668	26.885400	25.899375	30.353902
oranges	2.609510	1.168948	17.728765	1.121448	0.812378	5.931122	1.432885	42.911041	26.759713	26.988703	34.626200
mountains	2.612948	1.173004	18.355146	1.140798	0.821972	6.103778	1.439372	43.906666	27.039346	28.789218	35.220700
tiger	2.639279	1.180924	19.097727	1.181693	0.863564	6.282781	1.449501	45.430880	25.926641	26.998289	36.172493

Table 2: *CPU/GPU ratio of the mean-average execution time of each algorithm on test images. RGB2Gray through sobelFilter each run 10 times, kMeans and gaussPyramid run once due to long execution time on large images.*

4.3 *gaussFilter*

The clear advantage appears to be to the GPU version of the *gaussFilter* algorithm. The inherent computational complexity of a convolution over an entire image is H^2K^2, where W is the image size along one dimension and K is the convolution kernel. Properly implemented parallelization can ameliorate the impact of the H^2 term, resulting in effectively lower complexity. That seems to be the case for this algorithm. Additionally, a great deal of overlapping memory accesses for nearby CUDA threads would result in high memory locality, which again benefits the parallelized execution more than the sequential execution.

4.4 *sobelFilter*

Edge detection behaves generally as we'd expect, showing a distinct advantage for the GPU implementation, though not as much as for *gaussFilter*. The smaller kernel size of *sobelFilter* (3×3 vs 7×7) would result in much less memory overlap and fewer hits, as well as less computational impact on the CPU version.

4.5 *kMeans*

The speedups for *kMeans* do suggest the expected improvement of GPU over CPU for larger images, but with a very high degree of variance. As we stated previously, the random nature of the *kMeans* algorithm may require a very large sample size in order to get an accurate approximation of the average execution time, but that is simply not feasible (and perhaps not worthwhile) for our project. Running the CPU version of this 8-means on our largest image, *tiger*, took more than 38 minutes to complete.

4.6 *gaussPyramid*

gaussPyramid works as a combination of both *gaussFilter* and *lResize*, but as *lResize* is used to subsample each step in the pyramid, we expect to encounter the same weaknesses experienced in *lResize(*$\times 0.5$*)*. The data appears to bear this out; while the advantage is significantly in favor of the GPU implementation, it is markedly less so than it as for *gaussFilter*. However, the rate of improvement for GPU / CPU appears to increase faster for this algorithm than for *gaussFilter*. It is not clear why this is; further investigation may reveal something useful about this phenomenon.

4.7 Future Work

While the cadre of algorithms we've deliberated on so far have given us some useful insight, they do not adequately represent the breadth of image processing. We hope to continue adding more complex and useful algorithms to this list, as well as expanding the number of metrics we can measure in their execution. In order to understand the kind of architecture most IP algorithms need, we must take into consideration deeper issues, such as caching, energy consumption, qualities and challenges of parallelism, and optimizing our implementations for their intended hardware. For the near future, we plan on continuing development of our GPU-based *SIFT* implentation, improving our understanding of caching on GPUs, and studying the characteristics of other common IP algorithms.

References

HEGARTY, J., DALY, R., DEVITO, Z., RAGAN-KELLEY, J., HOROWITZ, M., AND HANRAHAN, P. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG) 35*, 4, 85.

LOWE, D. G. 1999. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, Ieee, 1150–1157.

LOWE, D. G. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision 60*, 2, 91–110.