# Parallel Corner Detection

Kunal Jobanputra (kjobanpu), Ryan Stentz (rstentz)

***Abstract*—For the computer vision community, quick feature detection in a given image is an important task. The Harris Corner Detection algorithm is popular for being able to be find corners in images due to being translation, rotation, and illumination independent. This paper will discuss different techniques to compute the algorithm in parallel, as computing this quickly is important to feed into a pipeline of tasks in modern computer vision, such as homography estimation. We targeted embedded devices, since it is becoming more popular to run computer vision tasks on these devices. We implemented serial, shared memory, and GPU versions of the Harris Corner Detection algorithm on the popular NVIDIA Jetson Nano as well as the 2019 MacBook Pro with 2.6GHz 6-Core Hyperthreaded Intel Core i7 (can run 12 threads at once). Using the GPU, we were able to achieve more than a 3.2x speedup over the serial version. With the MacBook, we were able to achieve more than an 8x speedup. With these speedups, it is possible to process 720p, 1080p, and 4k images on embedded devices in real time, which has the potential to increase the accuracy of many computer vision tasks.**

## I. Background

### A. Main Algorithm:

Below is a general overview of the Harris Corner Detection Algorithm. Each block is a distinct phase. Blocks with the same colors can occur simultaneously.

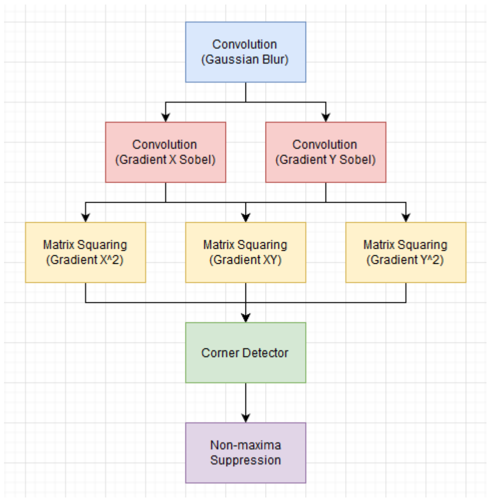

Convolution (Gaussian Blur)
Convolution (Gradient X Sobel)
Convolution (Gradient Y Sobel)
Matrix Squaring (Gradient X^2)
Matrix Squaring (Gradient XY)
Matrix Squaring (Gradient Y^2)
Corner Detector
Non-maxima Suppression

Figure 1: A high-level overview of the Harris Corner Detection Algorithm

Now, we'll walk through the algorithm on an example image as shown in Figure 2:

Figure 2: Original Image

Let I(x,y) denote the intensity of the pixel at the coordinate (x,y) in a given image we are running corner detection on. Then, we proceed as follows:

1) **Apply Gaussian Blur:** In this step, we apply a Gaussian blur to make some features less defined so that the algorithm can filter out *fake* edges.

An example would be tracing the clouds in the sky. Use $w = \frac{1}{16}\begin{bmatrix}1&2&1\\1&2&1\\2&4&2\end{bmatrix}$ as the Gaussian filter to convolve about the entire image. The resulting image after this step is shown in Figure 3.

Figure 3: Blurred Image

2) **Compute partials:** For each pixel at coordinate (x,y), compute the partials $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ using the Sobel Operator. Essentially, for $G_x$ = $\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}$ and $G_y$ = $\begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix}$, apply $G_x$ and $G_y$ centered at coordinate (x,y) to compute $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ respectively. Then, compute G = $\begin{bmatrix}g_{xx} = (\frac{\partial I}{\partial x})^2 & g_{xy} = (\frac{\partial I}{\partial x})(\frac{\partial I}{\partial y})\\ g_{xy} = (\frac{\partial I}{\partial x})(\frac{\partial I}{\partial y}) & g_{yy} = (\frac{\partial I}{\partial y})^2\end{bmatrix}$.
3) **Compute c(x,y):** For a constant $k$ determined from trial and error, compute c(x,y) = det(G) - $k$(trace(G))$^2$ for all pixels at a given (x,y) coordinate.
4) **Threshold:** Choose a threshold $t$ from trial and error, and highlight (x, y) coordinates whose c(x,y) $> t$. This produces a binary mask, where a 1 indicates a corner. When overlaying on the original image, Figure 4 is the output.
5) **(Optional) Perform Non-maxima Suppression**: Non-maxima suppression (NMS) can help refine the output. NMS is a non-linear filter which removes pixels that are not maximums in their local neighborhood. It does this by iterating over every pixel. If a pixel is not the maximum in its neighborhood of a selected size, then it is removed from the output. Otherwise, it is left unchanged.

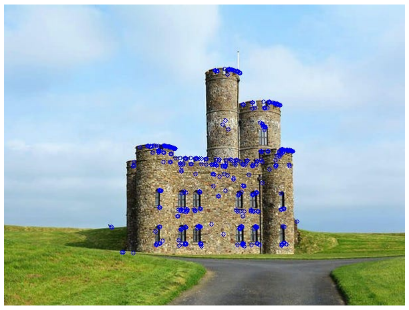


Figure 4: Output Image

## B. *Parallelization:*

We see that there are 5 distinct phases in the algorithm. Each pixel calculation is independent in a given phase, allowing for great amount of parallelism, while each phase must be completed before the next can proceed. If we store intermediate results between each phase, we don't have to worry about any contention after having a barrier after each phase, as we are writing to a separate buffer to a distinct (x,y) coordinate each time. This allows for a high degree of parallelism. There is high locality, as each phase computes pixels in a convolution type operation, scanning along the image. The two approaches we thought would be interesting to explore would be on a GPU and a shared address space on a CPU via

OpenMP. Specifics of the approaches are discussed in the next section.

## II. Approach

### A. Technologies Used

For some tasks, we saw that it was easier to just use some OpenCV functions. Particularly, we used OpenCV to convert the RGB image to a grayscale image (allows us to just use one intensity value instead of calculating over the different color channels), storing the intermediate results, highlighting the corners, and writing out the final image.

We used C++ for the serial and shared memory implementation (for which we also employed OpenMP) and used CUDA and Thrust for the GPU implementation. We used C++, as it allowed us to use OpenCV, which made the tasks discussed above much easier. Other than the libraries we mentioned here, all the code is written by us in the project.

We targeted the NVIDIA Jetson Nano for the GPU approach. This embedded device contains an ARM Cortex-A57 quad-core CPU operating at 1.43 GHz. For its GPU, the NVIDIA Jetson Nano 128-cuda core Maxwell architecture runs at 640 MHz. The device contains 4 GB of LPDDR4 shared memory between the GPU and CPU.

For the shared memory approach, we used a 2019 Macbook Pro with a 2.6GHz 6-Core Intel Core i7 Hyperthreaded architecture. We had problems building using the OpenMP library, which is the reason why we didn't test directly on the NVIDIA Jetson Nano. We talk towards the end of the paper about how fair of a comparison this is, as the Nano only had 4 lower powered cores available and costs less.

### B. GPU

This section will discuss the various CUDA implementations we developed for each of the five steps of the Harris Corner Detector shown in the previous section. As a reminder, Figure 1 shows a high-level overview of each stage in the Harris Corner Detector. Stages that appear in the same color occur at the same time.

*Convolutions:* As shown in Figure 1, there are three separate convolutions steps in the Harris Corner Detector. This section will discuss the various approaches to implementing the 3x3 Sobel convolution in CUDA. For the first approaches, each thread was mapped to a single pixel, and each thread block was mapped to a 2D image patch. We chose a thread block size of 32x32.

1) Naive Approach: The naive approach to implementing the Sobel convolution onto the GPU consists of loading the image into global memory and mapping a thread to every pixel in the image. Each thread is responsible for computing the value of the convolution for its corresponding pixel. This naive implementation utilizes the slowest of the memory types available to the GPU (global memory). In

our case, the convolution kernel is of size 3x3, so each thread performs 9 global memory reads and 18 arithmetic instructions (9 additions and 9 multiplications). This implementation has an arithmetic intensity of 2, which is less than desirable.

2) Unrolled Loop: Since the Sobel kernel is small and fixed in size, one easy way to improve performance is to unroll the typical kernel loop into a single expression for calculating the convolution for a single pixel. Since the Sobel kernel itself contains a row or column of zero weights we do not have to consider these rows or columns in the final calculation. This approach can safely ignore these values without creating any divergent execution. This saves 3 global memory reads, and 6 arithmetic operations.

3) Utilizing Shared Memory - first approach: The naive approach used global memory, which is the slowest memory available on the GPU. As such, switching to shared memory has potential performance gains due to its higher bandwidth and lower access time. An easy first implementation would have each thread first load its associated pixel from global memory into shared memory, synchronize with other threads, and then carry out the convolution utilizing shared memory. However, threads at the edge of the thread block require data beyond the image patch to complete their calculation. Since this data is not in shared memory, these threads cannot carry out their calculation and remain idle after loading their pixel value into shared memory. These idle threads were assigned pixels in the convolution's apron, which is shown below:

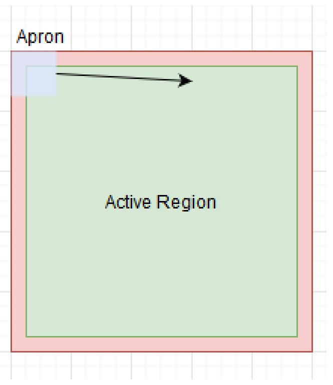


Figure 5

With this first approach of utilizing shared memory, the threads at the border of the thread block are mostly idle during the computation. In our case with a 3x3 Sobel filter and a 32x32 thread block, we have an apron size of $33 * 4 = 132$. This means that about 13% of our threads are idle during the computation. This leaves room for improvement.

4) Utilizing Shared Memory - second approach: In the previous approach, we saw that around 13% of the threads in a thread block remain idle after loading their pixel value into shared memory. We can fix this by loading a bit more into shared memory before performing the convolution. In the

3x3 Sobel filter case with a 32x32 thread block, we need to load a 34x34 image patch into shared memory to carry out the convolution with no idle threads. As before, each thread will load its pixel into shared memory. Additionally, some threads will be responsible for loading in the 1-pixel thick border surrounding the thread block. The mapping used to determine which thread loads which extra pixel is rather tricky to compute on the fly and would create much divergent execution. Thus, we used a predefined mapping stored as a 132x2 image in the GPU's constant memory. This array maps a thread's linear index in its thread block to a pixel location in the thread block's apron. The mapping is visualized in Figure 6:

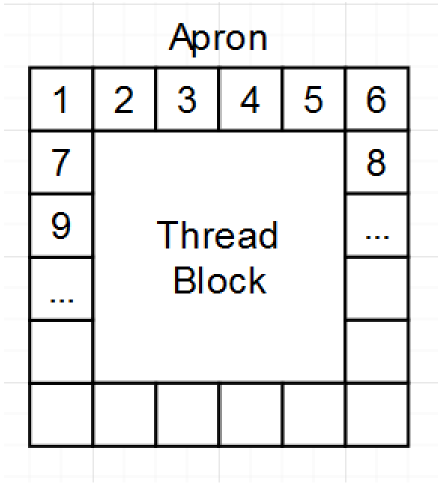


Figure 6

With this approach, we utilize shared memory and eliminate any idle threads.

5) Reducing Memory Overhead - streaming approach: The typical workflow of a GPU computation involves copying the input to GPU memory, running the computation, and then copying the output to CPU memory. Depending on the input and output size, the two memory transfers take a lot of time. We can help hide this memory overhead by switching to a streaming approach. The typical serialization of the memory transfer and convolution as pictured in Figure 7:

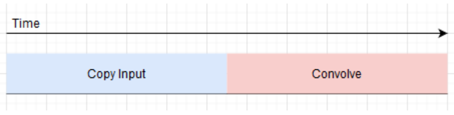


Figure 7

Instead of serializing, we can stream the first row of the input and perform a 1D convolution on it while we stream the second row. This pattern of memory transfer overlapped with computation is shown in Figure 8:

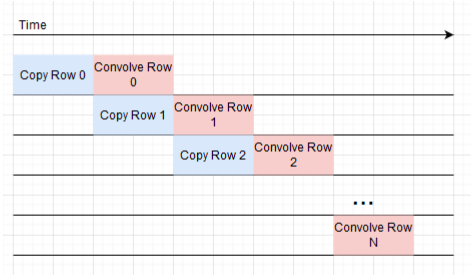


Figure 8

Under ideal conditions, this approach has the potential to nearly hide all memory overhead in the transferring of the input to the GPU. However, this is a potential downside. This approach requires the convolution operation to be performed in 2 steps:

a horizontal convolution followed by a vertical convolution. As such, one potential implementation of this approach must store the intermediate results from the horizontal convolution and then the final results. If both are stored in global memory, this approach could double the amount of required memory bandwidth.

*Computing the Cornerness Map:* The cornerness map step is the most computationally intense steps in the Harris Corner Detector. It takes the image gradients as inputs, multiplies them, and computes a value for each pixel that correlates with the pixel's probability of being a corner. This section will review the different approaches for parallelizing this step. Since this step can be parallelized over all pixels, we reused the same 32x32 thread block layout from the convolution step.

1) Naive Approach: Each thread in the naive approach computes the cornerness map by scanning over the neighborhood of its corresponding pixel. For each pixel in its neighborhood, it loads the gradients $I_x, I_y$ from global memory, computes $I_x^2, I_y^2$, and $I_{xy}$, and adds these values to $G_{xx}, G_{yy}, G_{xy}$ respectively. It then computes the trace and determinant of the $G$ matrix, and writes the final cornerness value to global memory. Each thread in this naive approach issues $3N^2$ global memory accesses, where $N$ is the length of the neighborhood, and slightly more than $6N^2$ arithmetic instructions. This implies that the operation has an arithmetic intensity of a little over 2, which shows room for improvement.
2) Shared Memory Approach: Since the cornerness calculation is, by nature, very similar to a convolution, it also can benefit from the utilization of shared memory. Instead of computing $I_x^2, I_y^2, I_{xy}$ for all neighbors, the shared memory approach has each thread compute $I_x^2, I_y^2, I_{xy}$ for their pixel and store the results in shared memory. Shared memory is then used when each thread scans over their neighborhoods. This approach reduces the total number of computations and global memory accesses.
3) Integral Images: The number of memory accesses for the cornerness computation can be drastically improved via the use of integral images. In an integral image, the point at $(x, y)$ is the sum of the pixels in the rectangle formed by $(0, 0), (x, y)$. An integral image can be used to compute a sum of pixels for any sized rectangle using just 3 additions and 4 memory accesses. The process for computing the sum of the purple rectangle in image 1 via the integral image 2 is shown in Figure 9:

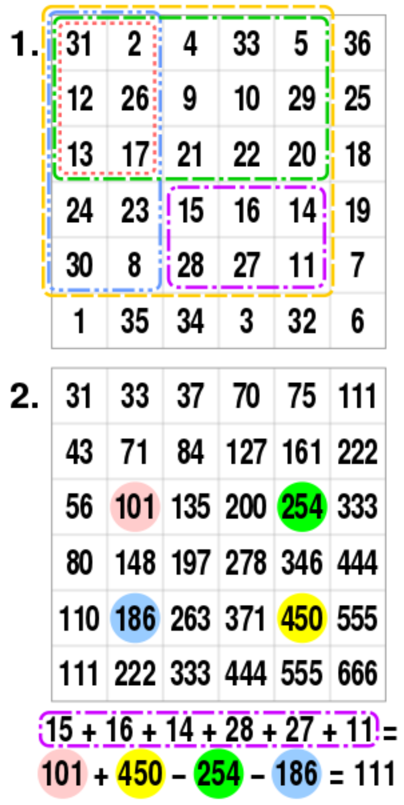


Figure 9

Integral images can be computed by performing a prefix sum across the rows and then performing another prefix sum of this result across the columns. In our implementation, we used the inclusive scan by key operation provided by the Thrust library to perform a scan over the rows. Then, we transposed the image and scanned over the rows again. In the cornerness calculation, we take the integral images of $I_x^2, I_y^2$, and $I_{xy}$. Afterwards, we use these images to compute each $G_{xx}, Gyy$ and $G_{xy}$ via 3 additions and 4 memory accesses. The rest of the calculation proceeds as described in the naive approach.

Using integral images has the benefit of a constant 3 additions and 4 memory accesses, regardless of the neighborhood window size used in the cornerness calculation. However, building these images takes time and involves a lot of memory accesses. Thus, this approach reaches its maximum effectiveness when the neighborhood window size is large.

*Non-maxima Suppression:* Non-maxima suppression (NMS) is another non-linear filter which removes any local non-maximum values of the cornerness map. It does this by sweeping a window over each pixel and determining if the pixel is a local maximum. Since each pixel is processed independently, this operation can be parallelized on a GPU. The naive approach assigns a thread to each pixel and a thread block to a 2D image patch. Each thread then looks at its pixel's neighbors to determine if it is the local maximum. If a thread finds a neighbor with a pixel value that's larger, it removes its pixel from the output. Otherwise, it keeps the pixel.

There are many potential optimizations to speedup NMS, such as spiral access patterns and corner segmentation. However, we found that our naive implementation NMS performs was not a major bottleneck in our pipeline, so optimization efforts were spent elsewhere.

It is important to note that the algorithm for NMS presented here is actually a popular approximation of NMS. For a true NMS algorithm suited for both a GPU and CPU, please refer to [1].

*Output Compression:* As mentioned earlier, GPU computations typically consist of 3 major operations: copy input, perform computation, and copy output. To gain the most precision, our inputs were grayscale images represented by an array of floating point values (4 bytes). Naturally, for our first implementation, we output a binary mask of the same size and data type as the input. We then compressed this output to an array of characters (1 byte) as a slight compression effort.

The output of Harris Corner Detection is particularly sparse, especially after performing NMS, so more aggressive compression techniques can be used. Specifically, one can compress the output from a binary image to an array of pixel locations for each corner. This representation, while requiring more data per corner, has the same length as the number of detected corners, which is typically significantly smaller than the number of pixels in the image. To create this compressed output, we took the binary mask as input and performed an inclusive scan via Thrust. We then ran our compression kernel, which used the scanned output to transform the 1s in the Harris Corner binary output to a vector of pixel locations.

### C. Shared Address Space

When implementing the GPU based solution, we noticed that copying to and from the GPU was expensive, we thought that implementing a shared memory version may be useful for evaluation. Even though there are fewer cores, we wanted to see if the expense of copying data to and from the GPU was worth the extra parallelism we got.

For this implementation, we went with OpenMP, as it was a nice way to translate our serial implementation to an efficient shared memory implementation. The idea was that each pixel in each phase is a given task that should be executed by a thread. Since the tasks have a pretty uniform workload across the board, we can simply statically assign these tasks across the available threads we have at the time. Since we are not dynamically assigning tasks, there is no overhead of contention of grabbing the next task: Each thread knows exactly which tasks to execute. Additionally, there is high locality, as blocks of contiguous memory would be scheduled to each thread due to how iterations are split up amongst tasks as well as the nature of how convolution operations exploit temporal and spatial locality. Lastly, we know that there is a low amount of false sharing between processors, as processors are starting at far and separate offsets into the image.

Since we were iterating through pixels, we have two dimensions that we are to consider. Specifying to OpenMP that each (i,j) pair was a distinct task would give us the static assignment that we described above. Initially, we used statically assigned both the inner and outer loop. The problem that this causes that each task $i$ now has $j$ distinct tasks, not using the threads available to their fullest potential since task $i$ is in charge of one entire inner loop. Instead, we want $i*j$ separate tasks to be evenly split amongst the threads. To do this, we condensed the loop to one for loop in which we got our row and column index respectively by dividing the value of our iterator by our width and modding the value of our iterator by our width. Although this only really helped our execution times a bit, it was an interesting optimization that allowed us to gain even better speedups.

## III. Results

To do an apples to apple comparison across different architectures and different methods, we used the same images to compare results. The following table summarizes the images that we used for testing:

| Name of Image | Dimensions | Shorthand |
|---|---|---|
| callibration.jpg | 256x256 | cal |
| checker.jpg | 300x300 | ch |
| castle.jpg | 600x450 | cas |
| 1080p.png | 1920x1080 | 1080 |
| 4k.jpg | 3840x2160 | 4k |

We tested using high-precision timers on various problem sizes.

### A. *Results - GPU*

We tested our GPU implementation of the Harris Corner Detector on the NVIDIA Jetson Nano. We compare its performance against an optimized serial version of the Harris Corner Detector running on the same system. Below is are the results for our best implementation of the GPU Harris Corner Detector:

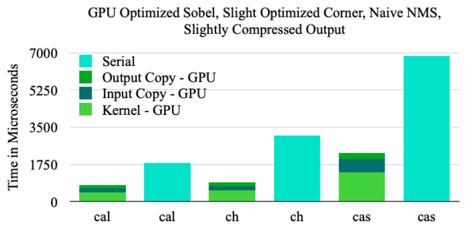


Figure 11

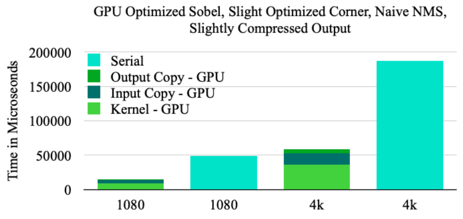


Figure 12

Now, we'll dive into each component of the Harris Corner Detector and evaluate the performance of the optimizations discussed in the previous section. *Sobel Optimizations Results:* Below is a comparison of the different Sobel convolution implementations discussed above, minus the streamed approach.

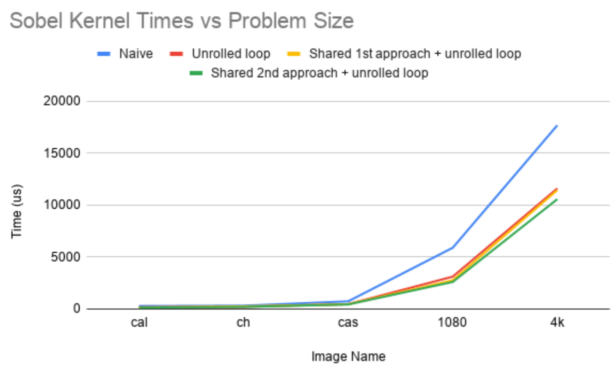


Figure 10

Below is a variety of metrics for each of these kernels running on our 1080p image:

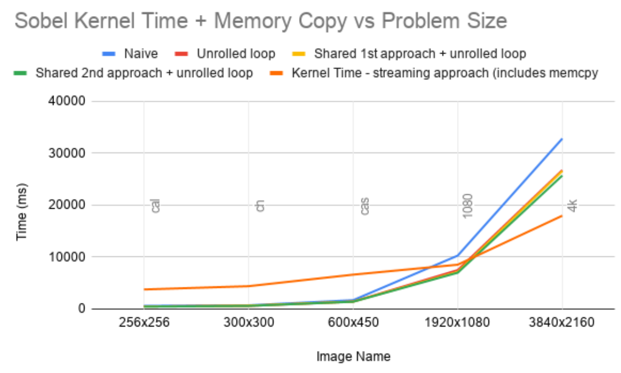
| Metric | Naive | Unrolled Loop | SM - 1st | SM - 2nd |
|---|---|---|---|---|
| GM Load | 46.75 GB/s | 23.15 GB/s | 4.01 GB/s | 3.5272GB/s |
| GM Store | 7.925 GB/s | 5.7899 GB/s | 7.13 GB/s | 6.20 GB/s |
| SM Load | 0 GB/s | 0 GB/s | 25.37 GB/s | 24.81 GB/s |
| SM Store | 0 GB/s | 0 GB/s | 3.38 GB/s | 3.11 GB/s |
| Stall-Data | 41.34% | 38.07% | 38.45% | 38.51% |
| Stall-Sync | 0.00% | 0.00% | 14.70% | 15.67% |

As we can see, the various optimizations made to the Sobel kernel caused incremental improvement. The first optimization, unrolling the loop, caused a significant reduction in kernel runtime. Unrolling the loop had the advantage of reducing memory accesses from 9 to 6, and, indeed, the metrics reveal a reduction in global memory loads. Switching to shared memory via our first approach also lowered kernel runtimes from 3120 us to 2775 us on a 1080p image. The metrics show that our first implementation of shared memory drastically lowered our global memory throughput while increasing our shared memory throughput. Shared memory is significantly faster than global memory, so this is likely the reason for the performance improvement. The metrics also reveal that our first shared memory implementation has more stalls due to synchronization. In our implementation, synchronization occurs after loading from global to shared memory, so this increase makes sense. It could also explain why the speedup was not as much as expected.

Finally, moving to our last optimization, our second shared memory approach reduced our kernel times slightly from 2775 $\mu$s to 2611 $\mu$s. Looking to the metrics, we see that our second implementation has lower memory shared memory utilization. This is because in the process of eliminating idle threads, we actually reduce the number of total thread blocks needed to cover an image. This stems from the fact that with our second implementation, thread blocks do not overlap spatially. This results in fewer redundant memory transactions (both global and shared) and give a slight performance improvement. One other important takeaway is that synchronization stalls increased. This is because our second implementation includes more memory transactions for some threads before the synchronization barrier, which results in longer waiting times for other threads.

In the above discussion, we did not discuss our final optimization approach: streams. This is because the streaming approach includes transferring the input.

Below is a comparison of all the Sobel kernel results, including the streaming approach.



As one can see, the streaming approach is worse for smaller images but shows promise for larger images. Unfortunately, due to asynchronous the nature of the streaming approach, we could not generate metrics and can only speculate on this approach's poor performance. One major difference between this approach and others is the number of kernel invocations. The streaming approach invokes a memory copy and kernel at each row of the image. Although there is very slight overhead for a single kernel invocation, the overhead from thousands invocations starts to add up. In addition to kernel invocations, the streamed approach relies on 1D convolutions instead of 2D. Our mapping of threads to pixels may have not been optimized for 1D, resulting in reduced parallelism.

*Computing the Cornerness Map:* We considered three approaches for computing the cornerness map: naive, naive with shared memory, and integral images. Comparing the naive and shared naive approach, we get the following table of the speedup over the naive approach:

| cal | ch | cas | 1080 | 4k |
|---|---|---|---|---|
| 3 $\mu$s | 5 $\mu$s | 10 $\mu$s | 418 $\mu$s | 642 $\mu$s |

From above, we can see that using shared memory during the cornerness calculation improves speeds of the corner detector considerably at larger images resolutions and less so at smaller resolutions. In addition to implementing a shared memory version of the cornerness kernel, we also implemented one that utilized shared kernels. This approach was abandoned rather quickly, however, since the time it took to compute the integral images overpowered the time saved while using them. For example, on a 1080p image, computing the three integral images takes 63496 $\mu$s. For perspective, our best implementation of the GPU Harris Corner takes 4350 $\mu$s to complete. When analyzing our computation of integral images, we noticed that the majority of the time (53%) was spent transporting the various results. Perhaps with a less naive implementation of matrix transpose would make this approach more feasible.

*Output Compression:* Our final set of optimizations focused on reducing the size of the algorithm's output, as this is copied to the CPU for the last step. Our first approach at this, switching from an array of float to an array of characters, had significant performance improvements. Below is a chart comparing the time to copy the output to the host for both the uncompressed and slightly compressed output:

| | cal | ch | cas | 1080 | 4k |
|---|---|---|---|---|---|
| uncompressed | 203 $\mu$s | 204 $\mu$s | 655 $\mu$s | 4768 $\mu$s | 15651 $\mu$s |
| compressed | 148 $\mu$s | 171 $\mu$s | 272 $\mu$s | 1686 $\mu$s | 6777 $\mu$s |

As one can see, this simple compression scheme nets about 3x speedup in copying the output to the CPU. With the compression route promising, we implemented a stronger compression algorithm described in the previous section. While this algorithm drastically lowered the time to copy the output to the CPU, it also significantly increased the kernel time. Further analysis revealed that compression increased the kernel runtime by about 5000 $\mu$s on a 1080p image while reducing the output copy time by 1000 $\mu$s.

## *B. Results - GPU Deeper Analysis*

Below is a ranking of the percent time spent doing each step of our GPU Harris Corner Detection algorithm:

1) 32%: copying the input image to the GPU
2) 31%: computing cornerness values
3) 18%: performing the Sobel convolution
4) 16%: non-maximum suppression

To further improve the algorithm, one should strive to optimize copying the input to the GPU and computing the cornerness values. For copying the input, the streaming approach has great potential. Although we were not able to see any performance benefits on reasonably sized images for embedded devices, it has the potential to half the time spent copying the input. For computing the cornerness values, optimized integral images could provide significant speedup with a larger window size.

## *C. Results - Shared Memory*

We tested our shared memory implementation on the 2019 Macbook Pro and compared the serial version the same machine so that we are comparing performances on a similar strength core. Similarly to the GPU implementation measuring, we used high-precision timers on a variety of images of differing sizes. In this section, we will primarily discuss the speedups relative to the amount of threads specified to OpenMP at runtime. The speedup graph across different images and different thread counts is given in Figure 17.

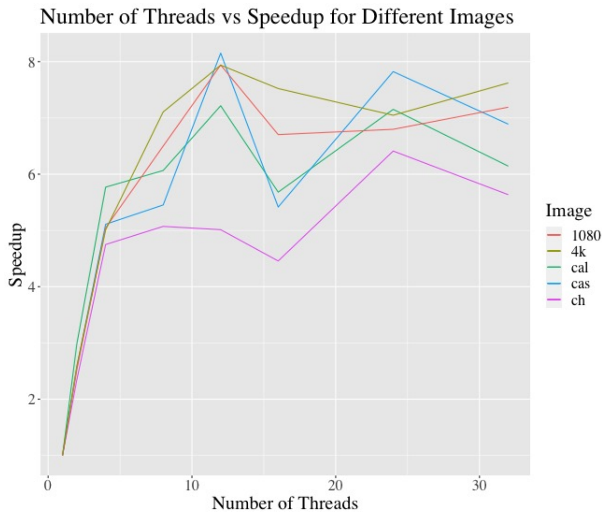


Figure 17

We tested the implementation with thread counts 1, 2, 4, 8, 12, 16, 24, and 32. There is a general trend that is happens across images of different sizes. Speedup shoots up to a superscalar level for thread counts 2 and 4. Then, speedup slows down but still increases until a thread count of 12, at which most images get their highest speedup (with exception of checker.jpg). After that, there is a slowdown in the speedup until a thread count of 16. There is again a pickup in the amount of speedup at a thread count of 24 threads for all images except for the largest two images, which slowly increase the speed up up to 32 threads. All of the other images decrease again at a thread count of 32 threads.

A likely reason as to why speedup increases at a superscalar level for low thread counts is due to the fact that data is staying in different processor caches and is being used for different convolution operations. There is faster access to more data at once. This slows down overtime, as there is less work for each processor, resulting in less data being stored in higher level caches.

As described above, there were two real anomalies from the general trend: checker.jpg not reaching it's maximum speedup at 12 threads and the largest two images not slowing down at a thread count of 32. In the usual case, it makes sense why 12 threads results in the most amount of speed up; for the Macbook Pro, 12 threads can run in parallel. There is no context switching overhead, allowing the CPU to just execute the statically assigned work. There are a couple reasons that could explain the odd behavior in checker.jpg. First of all, it is the second smallest image, so there may not be that much work that each thread really needs to do. However, we still see speedup in the smallest image, callibration.jpg, so another reason may be about the intensity values on the pixels. It is a boxy image of white and black squares, potentially making calculation very quick for the high density of white pixels (as these pixels would have an intensity value of 0, zeroing out some calculations).

We see a dropoff in speedup across the board after increasing the thread count past 12 threads, as the overhead of context switching between threads that can't run in parallel outweighs the benefits of having more threads. However, we see that for smaller images, we increase at a thread count of 24. The reason behind this may be because data stall latency can be hidden by running the next task on another thread. There is a decrease after increasing the thread count for a similar reason to above; smaller images don't have enough work for these threads so there is overhead in just context switching to other threads. Finally, we can address the anomaly of the larger images increasing speedup gradually after a thread count of 16. Since there is more data, it gives way to increasing data stalls from cold misses; it implies that the computation hasn't gotten to the point where context switching is wasted work.

## IV. Ending Remarks

Our optimized GPU implementation of the Harris Corner Detector yielded more than a 3x speedup over the serial version. This speedup over the serial version enabled our embedded device to process both 1080p and 4k images in real time (62 fps and 17 fps respectively).

However, we saw that the speedups were greater on the shared memory implementation; it may not be a fair apples to apples comparison, as we are using two very different architectures. It's possible that our GPU was not up to par with the caliber CPU we were employing with the shared memory implementation. With a better GPU with increased bandwidth and increased core count, the results may be drastically different.

One way to make a fair comparison would be how much it costs an individual. The Nvidia Jetson Nano costs around $100, while a chip with similar configurations as to one that exist in the Macbook Pro costs around $300. If we had either tested on a more expensive GPU or got a CPU that costs as much as the Nano did, the comparison could have been more stable. However, even considering the fact that the speedup on the shared memory approach was higher, the ability to process both 1080p and 4k images in real time on the GPU at a cheaper cost would suit most for their purposes of computer vision tasks, including object recognition, optical flow, feature matching, and more.

## V. References

[1] Teixeira, L., Celes, W., Gattass, M.: Accelerated Corner-Detector Algorithms (2008) 10.5244/C.22.62.

[2] Hosseini, F., Fijany, A., Fontaine, J.G.: Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture (2010)