Parallel Canny Edge Detection

Hunter Jans, Andrew Danciak, Sam Younglove

1. Introduction

Edge detection is a critical building block in the field of computer vision that presents a complex computation that is a prime candidate for parallel-based speedup. Despite the overall complexity of computation, the canny edge detection algorithm was selected for parallelisation as it presents a multi-stage process that can be easily compartmentalized into parallel modules to improve speedup. As a fundamental component in the world of computer vision, significant performance improvements discovered to this implementation of the canny edge detection can also improve the performance of more complex algorithms and applications.

2. Design Overview

The canny edge detection algorithm can be broken down into five distinct steps, each of these having aspects exploitable by a parallel application.

1. Apply a Gaussian filter to the image:

Edge detection is typically sensitive to image noise, as a result this operation will blur the image to minimize the effect of said noise. Since the Gaussian filter is a weighted convolution operation, it is easily parallelizable via tiling the input data. Additionally, constant memory can be used for the kernel to reduce load times and speed up the operation.

2. Find per-pixel gradients and directions:

Looking at each pixel and its neighbors, this stage will calculate the edge gradient via the Sobel Filters for rudimentary edge detection. Once again this step can be parallelized via tiling and constant memory. A parallel max reduction and a strided linear operation are performed to compute normalized edge gradients using the gradient value found in the image. Resulting from this stage is an image with detected edges, but with undesired variability in the quality of said edges.

3. Non-maximum suppression:

Taking the edge gradient strength and directions calculated previously, the image pixels are conditionally suppressed to reduce weaker detections; if a neighboring pixel along the gradient direction has a stronger detection value than the center one, then the center one is disregarded. This process can be parallelized fairly easily via a strided operation.

4. Double thresholding:

A parallel strided operation will once more filter out weak edges. Given a high and low threshold, all pixels stronger than the threshold are painted white and all pixels weaker than the low threshold as black, any remaining pixels are painted grey and passed to the next step.

5. Edge tracking via hysteresis:
Another filtering operation, this step looks at each grey pixel and each of its neighbors. If any neighbor is white, then the grey pixel is painted white, otherwise it is painted black. This step is once again parallelized via a tiled implementation with shared memory.

3. Implementation

As stated above, there are five major stages of the canny edge detection algorithm, each of which was implemented as one or more kernel functions depending on resynchronization requirements. Note that all device memory allocated to a function is freed after the next function finishes its execution. This is done to save memory, especially when running on very large images. Additionally, timing and recording calls lie between each major step.

1. Apply a Gaussian filter to the image:
Prior to this stage, the input image from the host is copied to a device array. The kernel function is launched with 1:1 ratio block/tile sizes, and elements are loaded such that the block has a halo padding of two elements to accommodate the 5x5 gaussian filter. From here a normal weighted convolution is executed. Each thread within the bounds of the image will write their result to an output array, given the thread itself is not designated as a padding pixel. Shared memory is loaded at the start of the kernel call, with the convolution filter being loaded into constant memory.

2. Find per-pixel gradients and directions:
The previous steps output is fed into the first of three kernel functions used by this stage. Once again using a 1:1 ratio block/tile size, though this time with a halo padding of only one element. Two separate 3x3 convolutions kernels are computing for each element, utilizing a horizontal and vertical Sobel Filter. Each pixel's gradient strength and direction are computed using hypot() and arctan() operations on the result of the prior convolution. The gradient strength and direction are then output to separate arrays.

Because the results need to be normalized, a series of strided, max value reduction kernel calls, dependent on input size, are performed to find the maximum float value. The result is then fed to a normalization function to normalize the gradient strengths in parallel before outputting the result to the edge gradient array.

3. Non-maximum suppression:
This step again uses 1:1 ratio block/tile size with one pixel of halo padding. It receives the edge gradient strengths and selectively filters out pixels whose gradient directional neighbors have a higher strength, and outputs the results to a new array.

4. Double thresholding:
Running the thresholding operation on each pixel in parallel, setting pixels exceeding the high threshold to 255, eliminating pixels below the low threshold by setting them to 0, and all other pixels are set to 25 (an arbitrary middle value). Once again relying on the max value of the working image, the threshold is given as a ratio. Thus max reduction must be run once more before the double threshold kernel can run. The double thresholding function takes this as its input, and sends its results to the final step.

5. Edge tracking via hysteresis:
This step again runs with 1:1 ratio tile/block size and one pixel of halo padding. It receives the output of the double thresholding operation and for every pixel with a value of 25, it checks if any neighboring pixels have a value of 255. If so, the center pixel is set to 255, otherwise it is set to 0.

After each of these steps have been executed, the result of hysteresis is given as the input to another kernel function that casts each float to an unsigned char in parallel. This is done to ensure compatibility with OpenCV display utilities used to show the image. Finally, the result of the casting is copied to a host array and any remaining memory allocations are freed.

4. Verification

In order to verify the correctness and efficiency gains of a parallel implementation, a point of reference was needed. A serial implementation of the canny edge detection algorithm was created to report timing data for each of the five major steps. A float to unsigned char cast operation was also included with the timing reports along with the total time taken by the whole algorithm.

A special debug operation mode was implemented to aid in independent development and verification of the parallel stage implementations. Utilizing command line arguments, the user could select specific stages to be returned from the serial implementation that could then be fed as input to later parallel stages, effectively removing the dependency of prior stages.

After both serial and parallel calls have been run, the correctness of the parallel implementation is verified via a simple element-wise subtraction with zero-value assertions. Additionally, a few modes of operation were created to make the means of testing more diverse and robust. A mode was added to run the parallel implementation in real time on a 4k video stream in order to visually present the speed of the implementation. Another mode was added which runs both serial and parallel implementations on a series of increasingly larger resolution images, this mode was added to showcase the scalability of improvements with a parallel implementation.

5. Performance

Once every stage of the parallel implementation was confirmed to be working, significant performance improvements over a serial implementation were seen across all tiers of hardware and image resolutions:

Image	Impl	Step 1	Step 2	Step 3	Step 4	Step 5	Cast Time	Total	Pass/Speedup
256x256	Serial	0.701444ms	1.42259ms	0.993324ms	0.071863ms	0.205947ms	0.00565ms	3.40105ms	PASSED
256x256	Parallel	1.1735ms	0.355392ms	0.077824ms	0.17408ms	0.051456ms	0.0648ms	2.14659ms	1.584393X
256x256	Speedups	0.597735X	4.00287X	12.7637X	0.412816X	4.00239X	0.0871914X	1.58439X	
512x512	Serial	2.89174ms	5.32229ms	3.07565ms	0.276239ms	0.921612ms	0.034681ms	12.5224ms	PASSED
512x512	Parallel	0.942688ms	0.864256ms	0.28672ms	0.493824ms	0.06368ms	0.07088ms	3.24771ms	3.855764X
512x512	Speedups	3.06755X	6.15823X	10.727X	0.559388X	14.4726X	0.489292X	3.85576X	
1Kx1K	Serial	11.4899ms	21.8849ms	12.1898ms	1.05272ms	3.45011ms	0.084323ms	50.1523ms	PASSED
1Kx1K	Parallel	0.774144ms	1.17862ms	0.345376ms	0.759808ms	0.325632ms	0.263168ms	4.62378ms	10.846618X
1Kx1K	Speedups	14.842X	18.5682X	35.2944X	1.3855X	10.5951X	0.320415X	10.8466X	
2Kx2K	Serial	39.6394ms	73.7913ms	36.1017ms	3.92205ms	13.0263ms	0.57675ms	167.058ms	PASSED
2Kx2K	Parallel	1.37968ms	1.84627ms	0.796672ms	1.11709ms	0.73728ms	0.306176ms	7.22093ms	23.135317X
2Kx2K	Speedups	28.7309X	39.9677X	45.3156X	3.51096X	17.6681X	1.88372X	23.1353X	
4Kx4K	Serial	172.324ms	338.753ms	185.277ms	14.5016ms	57.3302ms	2.39142ms	770.579ms	PASSED
4Kx4K	Parallel	4.19299ms	4.98995ms	2.30912ms	2.69005ms	2.22106ms	0.750464ms	18.2768ms	42.161507X
4Kx4K	Speedups	41.0981X	67.8869X	80.2372X	5.39084X	25.8121X	3.18659X	42.1615X	
4Kx6K	Serial	256.894ms	415.301ms	183.981ms	21.942ms	65.6349ms	3.29781ms	947.051ms	PASSED
4Kx6K	Parallel	5.71011ms	7.5209ms	3.29421ms	3.59117ms	3.1488ms	1.05677ms	26.6833ms	35.492325X
4Kx6K	Speedups	44.9893X	55.2196X	55.8499X	6.11X	20.8444X	3.12066X	35.4923X	

(Results from running on an AMD Ryzen 9 7900 (3.7-5.4GHz) and an RTX 3080TI (1.37-1.67GHz) 80SMs.)

Already, significant performance improvements were being observed, however with several notable points of resynchronization being present in this first implementation, it was clear there was more that could be done to improve speedups.

The most obvious improvement could involve any attempt at removal of resynchronization points via clever kernel design at the cost of some unused calculations; one such potential removal is between the gaussian filter application and the gradient edge calculation. In theory, if the halo size of the gaussian filter tile were increased by one, then the Sobel Filter convolution could be applied in the same kernel call, removing the requirement of inter-block resynchronization between the operations.

Another improvement, which was developed and tested, involves the presence of very sparse data being fed to the hysteresis step. The double thresholding step already sets some points to the intermediate weak values, so an experimental implementation was created to simply mark these points when set. Hysteresis could then be performed in the same kernel call operating only on the marked points, removing the need to search through all of the non-marked pixels, thus saving computation time. Following that improvement, the type cast from float to unsigned char can be appended to the end of this kernel function as well. This approach effectively merges steps 4, 5, and the casting, and in execution produced notable improvements.

Image	Impl	Step 1	Step 2	Step 3	Step 4	Step 5	Cast Time	Total	Pass/Speedup
256x256	Serial	0.697794ms	1.40939ms	0.775396ms	0.077773ms	0.201887ms	0.00528ms	3.16789ms	PASSED
256x256	Parallel	1.89338ms	0.249952ms	0.064512ms	0ms	0ms	0.232448ms	2.58458ms	1.225690X
256x256	Speedups	0.368545X	5.63863X	12.0194X	infX	infX	0.0227148X	1.22569X	
512x512	Serial	2.78938ms	5.23501ms	2.99338ms	0.27847ms	0.89269ms	0.024481ms	12.2136ms	PASSED
512x512	Parallel	0.460992ms	0.589824ms	0.285696ms	0ms	0ms	0.225504ms	2.00954ms	6.077834X
512x512	Speedups	6.05083X	8.87554X	10.4775X	infX	infX	0.108561X	6.07783X	
1Kx1K	Serial	11.496ms	22.0923ms	12.9079ms	1.12564ms	3.29882ms	0.087193ms	51.0084ms	PASSED
1Kx1K	Parallel	0.77728ms	1.14381ms	0.35024ms	0ms	0ms	0.629984ms	3.61286ms	14.118541X
1Kx1K	Speedups	14.7901X	19.3147X	36.8545X	infX	infX	0.138405X	14.1185X	
2Kx2K	Serial	40.0069ms	74.3268ms	35.074ms	3.34219ms	12.7782ms	0.543129ms	166.072ms	PASSED
2Kx2K	Parallel	2.55562ms	3.98848ms	0.819456ms	0ms	0ms	1.02605ms	9.12186ms	18.205965X
2Kx2K	Speedups	15.6545X	18.6354X	42.8015X	infX	infX	0.529341X	18.206X	
4Kx4K	Serial	170.135ms	340.375ms	182.198ms	14.6657ms	58.0551ms	2.46843ms	767.899ms	PASSED
4Kx4K	Parallel	4.05725ms	4.97459ms	2.29926ms	0ms	0ms	1.44282ms	13.5706ms	56.585537X
4Kx4K	Speedups	41.9336X	68.4228X	79.242X	infX	infX	1.71084X	56.5855X	
4Kx6K	Serial	253.359ms	413.187ms	182.379ms	23.0236ms	65.2318ms	3.49044ms	940.673ms	PASSED
4Kx6K	Parallel	5.86061ms	6.94461ms	3.2736ms	0ms	0ms	1.9927ms	19.888ms	47.298439X
4Kx6K	Speedups	43.2308X	59.4976X	55.7122X	infX	infX	1.75161X	47.2984X	

(Results after merging steps 4, 5, and casting. The Cast Time column shows the speed of this single kernel call.)

These improvements, with little change to code design, made it clear that there are definitely further improvements that could have been made if time had permitted. Additionally, it was attempted to use the fast math compilation flag to improve performance further, however this resulted in no significant improvement at lower resolutions and actually caused the parallel implementation to be incorrect at higher (4k+) image resolutions. This is possibly due to fast math having a trade off of speed at the cost of precision, building up over the various steps and causing a pixel here or there to be incorrect. It is also unlikely that pinned memory or streams would significantly increase the performance of the implementation, as significant data is only ever passed between host and device at the beginning and end of the process. It may be possible to achieve more significant speed ups with more powerful hardware as well. Overall, it is well within reason to estimate that with further improvements and running on better hardware, performance improvements of 80x to 100x vs serial could be achieved.

6. Conclusions

Through application of the concepts and practices of parallelism, it was proven beyond a reasonable doubt that the canny edge detection algorithm is a valid candidate for significant performance improvements via parallelization. Through several benchmark series with a baseline working implementation, peak improvements of ~40x were achieved. These improvements were later boosted by 10-40% by a minor experimental improvement done simply to prove the implementation could be more efficient. While said improvements themselves are not necessarily enough to be deemed groundbreaking, it is clear that they are significant enough to prove the contrary given enough development time, optimization, and advances in implementation.