Internship Report

Parallelizing Applications using OpenCL

Ananthatejas R, Madhuri Shanbhogue

SERC, IISc Bangalore

Abstract

The goal of this project was to explore the potential performance improvements that could be gained through the use GPU processing techniques within the OpenCL architecture for several different types of computer vision algorithms.

The choice of computer vision algorithms as the focus was based on examples of data level parallelism found within the algorithms and a desire to explore the performance variation of these between various GPU clusters.

This report includes an analysis of Gaussian Blur, Median Blur, JPEG Compression and Canny Edge Detection.

Gaussian Blur

Algorithm

The normalized image needs to be convolved with a Gaussian kernel generated using a suitable Gaussian function. Convolution involves the following steps:

- Neighbouring pixel values are taken on a weighted basis with weights as per the convolution kernel.
- The pixel under consideration is then replaced with the sum of the weighted pixel values of the neighbouring pixels and the weighted value of the pixel itself.

Parallelization

In the parallel implementation the process of convolution is done in a kernel i.e. for the convolution of each pixel a work item is deployed. Each work item computes the weighted sum of all the neighboring pixels as per the size of the convolution kernel and the pixel itself. The original pixel value is replaced with the sum.

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	90.000	147.658	0.61
1024 x 1024	395.000	182.120	2.169
2048 x 2048	2,010.000	355.298	5.657
4096 x 4096	6,445.000	1,007.891	6.395

Results on T10 Tesla GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	90.000	273.064	0.330
1024 x 1024	395.000	493.557	0.800
2048 x 2048	2,010.000	1,409.562	1.426

Results on 4870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	90.000	311.680	.289
1024 x 1024	395.000	358.498	1.102
2048 x 2048	2,010.000	553.208	3.633
4096 x 4096	6,445.000	2,595.523	2.483

Results on 5870 ATI GPU

Median Filter

Algorithm

Median filtering is nothing but replacing the pixel value with the median of the neighbouring pixels.

This implementation calculates the median of a 9x9 neighbouring area around each pixel

The sequential implementation involves the process of finding the median for each pixel is carried out using “for” loops

Parallelization

In the parallel implementation, pixel level parallelism is obtained. The process of finding the median of the neighbouring pixels is done by a single work item for the respective pixel. Parallelizing this process gives a good speed up over sequential as it involves sorting of the pixel values to find the median.

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	6345.000	805.309	7.878963
1024 x 1024	2.3E+04	2,857.275	8.049628
2048 x 2048	7.7E+04	9,418.611	8.175303

Results on T10 Tesla GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	6345.000	2,588.826	2.450918
1024 x 1024	2.3E+04	8,277.578	2.778591
2048 x 2048	7.7E+04	25,423.801	3.028658

Results on 4870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	6345.000	1,015.046	6.250948
1024 x 1024	2.3E+04	2,489.273	9.239645
2048 x 2048	7.7E+04	7,139.649	10.78484

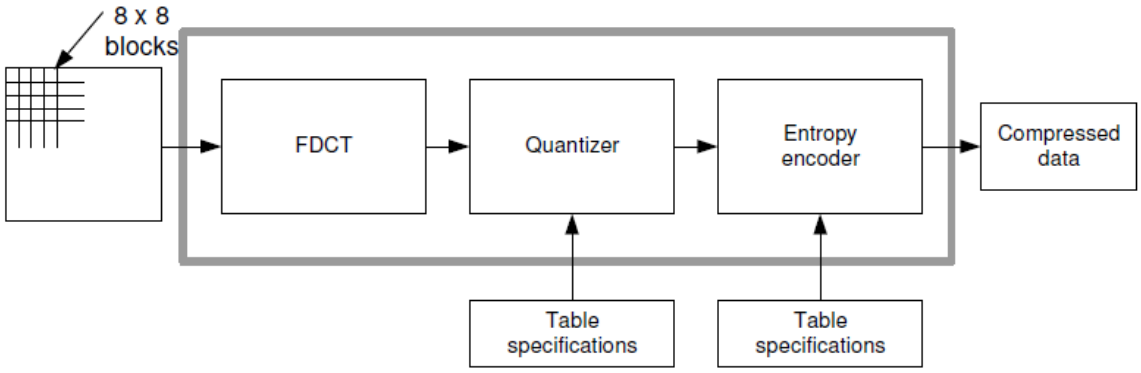
Results on 5870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	6345.000	313.076	20.2666
1024 x 1024	2.3E+04	1115.452	20.8660
2048 x 2048	7.7E+04	3909.243	19.6560

Results on NVIDIA C2070 GPU

JPEG Compression

Algorithm



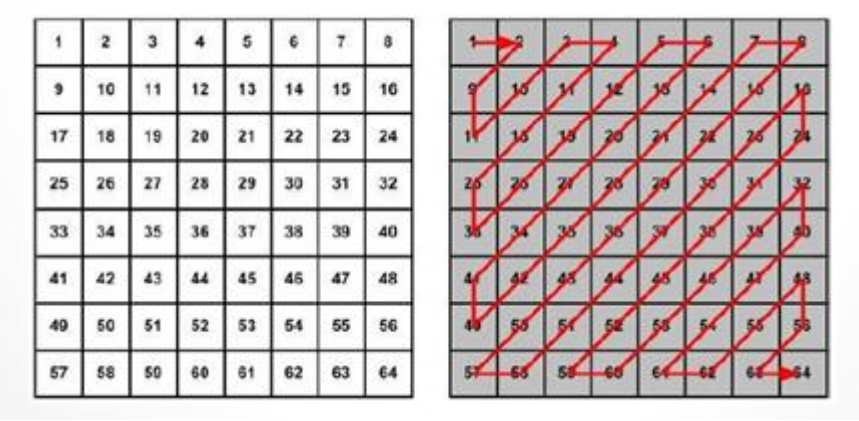
The algorithm involves the following steps:

- Conversion from (r,g,b) to (y,cb,cr)
- Splitting each channel into 8x8 blocks
- Applying Discrete Cosine Transform to each of those blocks
- Quantization of each block
- Zigzag scan
- Entropy coding of each block

The image is loaded into the main memory and the image is broken down into blocks of 8x8 pixels called Macro blocks. A 2D discrete cosine transform (DCT) is performed on each 8x8 macro block to separate it into its frequency components.
The DCT equation is as shown below

$$X_{k_1,k_2} = \sum_{n1=0}^{N_1-1} \sum_{n2=0}^{N_2-1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_1}\left(n_1+\frac{1}{2}\right)k_1\right] \cos\left[\frac{\pi}{N_2}\left(n_2+\frac{1}{2}\right)k_2\right]$$

The result at the top left corner (k1=0, k2=0) is the DC coefficient and the other 63 results of the macro blocks are the AC coefficients. As k1 and k2 increase (up to 8 in either direction), the result represents higher frequencies in that direction. The higher frequencies end up in the lower right corner (higher k1 and k2) and the coefficients there are usually low since most macro blocks contain less high frequency information. The purpose of the DCT is to remove the higher frequency information since the eye is less sensitive to it.
The quantization step divides the DCT transform by a quantization table so that the higher frequency coefficients become 0. The quantization table is defined by the compression percentage and has higher values at higher frequencies. The resulting matrix usually has a lot of zeros towards the right and bottom (higher k1 and k2).
At this point, all the lossy compression has occurred, meaning that high frequency components have been removed. The final step is to encode the data in a lossless fashion to conserve the most space. This involves two steps.
First, zigzag reordering reorders each macro blocks from the top left to the bottom right in a zigzag fashion so that the 0's end up at the end of the stream. This way, all the repeated zeros can be cut.



The final step is to use Huffman encoding to encode the whole picture by replacing the statistically higher occurring bits with the smallest symbols. This can be done with a standard Huffman table or can be generated based on the image statistics.

Parallelization

Parallelization involves 2 kernels each achieving pixel level parallelism for the functions DCT and quantization across barriers. One kernel is used to covert input data from the ppm file into (r,g,b) and then convert it into (y,cb,cr) format. Another kernel performs DCT on the pixel of (y,cb,cr) separately and then also quantizes it. Both the kernels achieve pixel level parallelism.

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
1024 x 1024	145.000	199.0194	0.728572
2048 x 2048	515.000	636.6885	0.808873
4096 x 4096	2005.000	2108.026	0.951127

Results on T10 Tesla GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	30.000	1017.796	0.014978
1024 x 1024	145.000	1382.594	0.104875
2048 x 2048	515.000	1665.709	0.309178

Results on 4870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
1024 x 1024	145.000	1382.594	0.142465
2048 x 2048	515.000	1665.709	0.369084
4096 x 4096	2005.000	8,066.383	0.468986

Results on 5870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	30.000	53.963	0.55724
1024 x 1024	145.000	136.593	1.02663
2048 x 2048	515.000	461.935	1.1257
4096 x 4096	2005.000	1925.305	1.03360

Results on NVIDIA C2070 GPU

Image Size	Sequential	ATI 5870	Tesla T10	ATI 5870 (Speedup)	Tesla T10 (Speedup)
512 x 512	190	39.98	19.099	5.235	9.948
1024 x 1024	780	103.98	74.255	5.656	10.504
2048 x 2048	3080	389.058	296.12	5.707	10.401
4096 x 4096	12290	1521.623	1186.652	5.679	10.356

Kernel Speed Up (Opteron)

Image Size	Sequential	ATI 5870	Tesla T10	ATI 5870 (Speedup)	Tesla T10 (Speedup)
512 x 512	100	39.98	19.099	2.501	5.235
1024 x 1024	420	103.98	74.255	4.039	5.656
2048 x 2048	1690	389.058	296.12	4.343	5.707
4096 x 4096	6740	1521.623	1186.652	4.429	5.679

Kernel Speed Up (Xeon)

Canny Edge Detection

Algorithm

The steps involved are:

- Noise Reduction
- Finding the intensity gradient of the image
- Non-maximum suppression
- Tracing edges through the image and hysteresis thresholding

- Noise reduction is carried out by applying a Gaussian blur on the image
- Then the intensity gradient magnitude (G) and the angle (Θ) is found out convolving each pixel with the matrix whose elements are derivatives of a Gaussian function in both x and y direction

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$
$$\mathbf{\Theta} = \arctan\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right).$$

Where G_x and G_y are the matrices after convolution with the derivative of Gaussian Function in x and y directions

- By finding the direction of the edge, a search is done to decide whether the gradient magnitude is a local maximum
- Also if the magnitude is high, then it is more likely to be an edge than if it is low
- Thresholding thus defines a lower and higher thresholds and then filters the image to give a better edge detection

In the sequential implementation, the process of convolution for Gaussian blur and to find the intensity gradient is performed using loops

Parallelization

In the parallel version the process of convolution with Gaussian kernel and finding the intensity gradient are parallelized by creating two kernels to carry out those processes. The first set of kernels launched convolve the image with the Gaussian kernel and the second set of kernels launched compute the intensity gradient and retain it as an edge if it is a local maximum. The intensity gradient is computed in the direction calculated using the above formula

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	70.000	204.592	0.342144
1024 x 1024	270.000	265.425	1.017237
2048 x 2048	1090.000	558.128	1.952957
4096 x 4096	4440.000	1594.599	2.784399

Results on T10 Tesla GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	70.000	7041.858	0.009941
1024 x 1024	270.000	7152.892	0.037747
2048 x 2048	1090.000	7732.059	0.140972

Results on 4870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	70.000	895.126	0.078201
1024 x 1024	270.000	1026.397	0.263056
2048 x 2048	1090.000	1423.070	0.76595
4096 x 4096	4440.000	3286.191	1.351108

Results on 5870 ATI GPU

Image Size	Sequential (Xeon)	Parallel	Speed Up (Xeon)
512 x 512	70.000	22.255	2.69602
1024 x 1024	270.000	69.770	4.01319
2048 x 2048	1090.000	250.655	4.22892
4096 x 4096	4440.000	943.676	4.78978

Results on NVIDIA C2070 GP

Conclusion

The Gaussian blur algorithm performs better for images of larger size on the GPU as for smaller images the overhead for kernel call and read and writes from the device is more. Also as the image size goes beyond 4096 the overhead of reading and writing from and to the image has to be taken into account. As the read write from device memory is substantial in the program Tesla T10, with a higher frequency memory clock performs better than the ATI 5870 though it has more number of cores

In case of Median Blur, due to the amount of computation involved in finding the median even for small images considerable speed up is observed. Also as the AMD 5870 has a better core clock speed the kernel performance is much better compared to Tesla T10 (higher computation). From the results obtained, it can be observed that the core clock speed is a bottle neck for the Tesla T10 in case of this algorithm. The C2070 gives the maximum speed up compared to the others because it has a large number of thread processors (448).

Jpeg Compression though from the timing values it can be seen that the kernel speed has increased, the bottleneck of the program being the read write into the files, overall speedup is not as high. Again because of the slower memory clock, the performance of the ATI 5870 is not as good as Tesla T10.

In Canny Edge Detection, due to considerable amount of read-write ATI 5870 slower than the Tesla T10. The ATI 4870 is very slow owing to a number of conditional statements for which it does not seem to be optimized. This causes considerable increase in the number of fetches compared to Tesla T10 and ATI 5870 and C2070.

Gaussian Blur

Median Blur

JPEG Compression

Canny Edge Detection

References

1. Wallace, G. The JPEG Still Picture Compression Standard. IEEE Transactions on Consumer , Electronics. December 1991.
2. M. Benes,. S. Nowick., A Wolfe,. “A Fast Asynchronous Huffman Decoder for CompressedCode Embedded Processors” , 1998
3. S.T. Klein,. Y. Wiseman, “Parallel Huffman Decoding with Applications to JPEG Files“, 2003
4. Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence
5. Image Processing on the GPU: Implementing the Canny Edge Detection Algorithm.