

# JPEG Compression Algorithm Using CUDA

Course Project for ECE 1724

Pranit Patel, Jeff Wong, Manisha Tatikonda, and Jarek Marczewski

Department of Computer Engineering

University of Toronto

## 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 CUDA architecture for JPEG compression algorithm. The choice of compression algorithms as the focus was based on examples of data level parallelism found within the algorithms and a desire to explore the effectiveness of cooperative algorithm management between the system CPU and an available GPU. In our project we ported JPEG Compression algorithm in CUDA and achieved upto 61% performance.*

## 1. Introduction

The advent of modern GPUs for general purpose computing has provided a platform to write applications that can run on hundreds of small cores. CUDA (Compute Unified Device Architecture) is NVIDIA's implementation of this parallel architecture on their GPUs and provides APIs for programmers to develop parallel applications using the C programming language. CUDA provides a software abstraction for the hardware called blocks which are a group of threads that can share memory. These blocks are then assigned to the many scalar processors that are available with the hardware. Eight scalar processors make up a multiprocessor and different models of GPUs contain different multiprocessor counts.

Video compression applications are essential to satisfy the growth of digital imaging applications. The JPEG standard provides great efficiency for compressing still images. The JPEG coding standard provides compression and coding method for a continuous-tone still images. In the JPEG standard, information that cannot be easily seen with the human eye is eliminated to obtain greater compression.

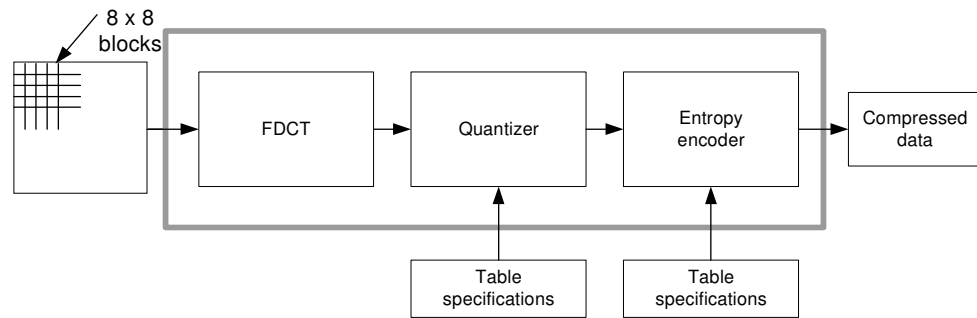
As explained, JPEG compression is an important and useful technique that can be slow on current CPUs. We wanted to investigate ways to parallelize compression specifically on GPUs where hundreds of cores are available for computation. By investigating JPEG compression, we can characterize which part of algorithm suited to parallelization. In addition, our goal was to not only parallelize applications for a new architecture but also to find the strengths and weaknesses of that architecture.

## 2. JPEG Compression Algorithm

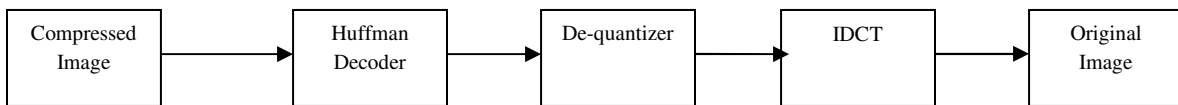
JPEG image compression is inherently different than sound or file compression in that image sizes are typically in the range of megabytes, meaning that the working set will usually fit in the device global memory of GPUs. Whereas larger files usually will require some form of streaming of data to and from the GPU, JPEG requires only a few memory copies to and from the device, reducing the required GPU memory bandwidth and overhead. In addition, the independent nature of the pixels lends itself to data level parallelism in CUDA. Because of these two factors, we predicted that JPEG compression can take advantage of CUDA's massive parallelism. To compare performance with our CUDA implementation, we implemented serial implementation of JPEG algorithm in CUDA which is optimized for processing the whole image rather than processing individual rows.

### 2.1 JPEG Algorithm

In order to find which part of JPEG compression could be parallelized, we investigated the algorithm which performed the compression itself [3] and [4]. Figure 1a and Figure 1b shows the general progression of the JPEG algorithm. The steps of the algorithm must be performed in a sequential manner because the operations in each block depend on the output from the previous block. Where the parallelism can be extracted is limited to the operations within each block.



**Figure 1a: DCT-based encoder simplified diagram**



**Figure 1b: DCT-based decoder simplified diagram**

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 equation for the DCT is shown in (1)

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

The result at the top left corner ( $k_1=0, k_2=0$ ) is the DC coefficient and the other 63 results of the macro blocks are the AC coefficients. As  $k_1$  and  $k_2$  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  $k_1$  and  $k_2$ ) 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  $k_1$  and  $k_2$ ).

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, zig-zag reordering reorders each macro blocks from the top left to the bottom right in a zig-zag 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.

Once the image is compressed we can get the decompressed image by reversing all encoder steps. First we apply Huffman decoding in-order to revert Huffman encoding. After that we de-quantized the Huffman decoding matrix by multiplying quantization table. And final step in Decoding is applying Inverse Discrete Cosine Transform (IDCT). Upon applying IDCT, we get the image which is as close to original image.

### 3. Related Work

#### 3.1 DCT/IDCT

Because DCT is the main component of MPEG and other video compression, there are many related researches in accelerating DCT with multi-processors or vector DSP to meet real-time performance requirement. Because DCT is a block-based computation, DCT is a good match to the multi-processor environment. In article [5], the author uses Stream SIMD (SSE) Extensions instructions in order to utilize the Single Instruction Multiple Data (SIMD) unit inside of the CPU to accelerate DCT computation. The SIMD unit allows single instructions to execute on multiple data. Although, the SIMD unit in CPU has much less number of parallel processing units comparing to GPU, but because of the high CPU clock frequency, it is still able to achieve reasonable performance.

In paper [6], the authors use DirectX 9.0 language to implement shader program to compute DCT on GPU. Before general purpose computation language such CUDA and OpenCL are available, shader program is the most common method to execute user defined computation on GPU. However, shader program is difficult to write because shader program is designed to work with graphics applications. To perform general purpose computing, input data are stored as texture in GPU memory, and the computation is applied as a transform to the texture. In this paper, the authors are able achieve 1.5x performance gain with shader program compares to CPU with SSE implementation.

### **3.2 Huffman Coding**

Encoding methods, including Huffman, were quite popular in research in mid to late 90s and it resurfaced in 2003 for parallel computing. Earlier papers were mainly concerned with ASIC or FPGA implementation of effective shift-register approach [7] and [8], or efficient comparators [8]. [8] suffers the problems on pre-defined Huffman Encoding, and can not be used in the generic case. It seems that decoding with Decoding Tree (one input bit at a time) is favourable to decoding with a table (one output char per cycle). Unfortunately, structures described in those early do not scale well and are not ideal for CUDA due to the same asynchronous input to output problem.

Klein and Wiseman[9] had a novel approach where they utilized self-correcting Huffman property. Variable length codes have a tendency to self correct when starting point is wrong. Decoding will always find a matching symbol, but it can wrong length, this leads to stumbling unto symbol boundary. This self correction is not very deterministic. Problem is in how to detect where the self correction did took place and then synchronize it. Self correcting property shows promise. If for example it could be mathematically proven that self correction is guaranteed over N input bits, it could mean that each thread would need to decode 2N with 1N overlap. Positions for output characters in the start, middle, and end of thread would need to be stored for comparison. This would reduce the wasted treads to 1 out of 2. Applied almost directly to CUDA that should result in at least 3x improvement in the straight forward implementation (IFF memory conflicts can be resolved).

[9] proposes methods isolating the starting point as insertion of synchronization codes, or padding at block boundaries. Padding in particular makes it trivial to divide the bit-stream into blocks and allows for computation without redundancies, all for cost of few bits at a block boundary. Unfortunately, such methods can not be applied to classic Huffman encoding, as it would not be backward compatible. However this becomes method is incorporated into the new JPEG standard, it will make it possible for optimize the algorithm for HardWare.

JPEG compression has been a new topic on CUDA forums. One of such forums states that NVIDIA graphics cards actually have hardware accelerators for JPEG with Huffman, but it is not within CUDA programming guide. General consensus is that Huffman encoding is either CPU or ASIC domain. General Purpose Processing accelerators might not be suitable.

## 4. Methodology and Implementation

Our approach in parallelizing the JPEG algorithm was to break the data level parallelism up in each block in Figure 1a and Figure 1b and write a kernel for each of the parallel tasks. Because of the hardware limitations for each CUDA block to access shared memory (512 threads, 16KB), it was important to keep shared data within a block for minimum latency. Fortunately for JPEG, the shared data within each macro block is small and does not approach the 16KB limit.

The load image I/O step cannot be parallelized in any fashion. In fact, CUDA introduces an overhead since the image has to be transferred between the device (GPU) memory and the host (CPU) memory. We further enhancement to minimize the CUDA overhead by loading whole image into global device memory once and perform encoding, then bring the result back to host memory.

### 4.1 DCT and IDCT

The DCT is a separable transform which means DCT is first applied to the column of the 8x8 matrix, and then DCT is applied to the rows of the results. To avoid calculating cosine values on the fly, the cosines of different phases are pre-calculated and stores into an 8x8 constant float matrix C. To perform DCT on the column of the macro block M, 8x8 constant matrix is cross multiply with the 8x8 macro block as shown in equation (2).

$$C \times M = \text{DCT\_R} \quad (2)$$

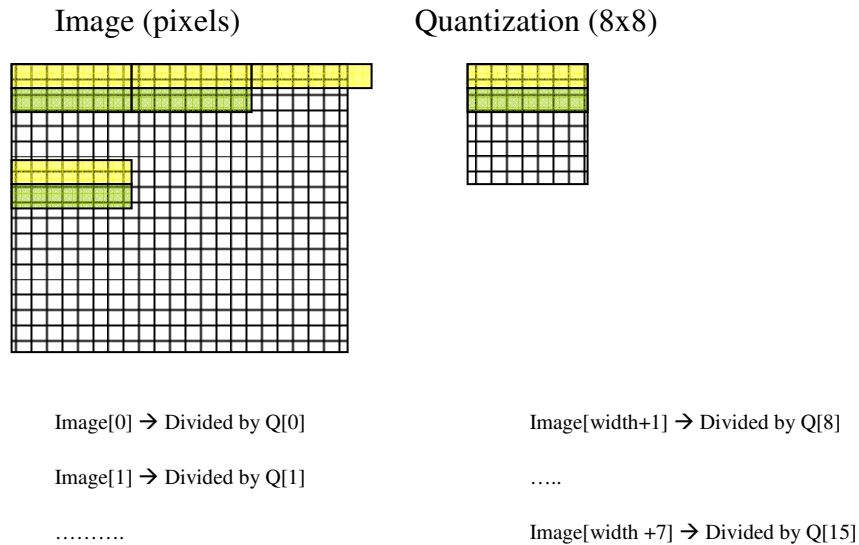
Then the second pass of DCT is performed to the row of column DCT result DCT\_R as shown in equations (2).

$$\text{DCT\_R} \times C^T = \text{DCT\_2D} \quad (3)$$

Two implementation approaches are used to implement DCT. The first implementation uses 64 threads for each thread blocks. In the kernel, each thread loads one pixel from raster. The read is assigned to each thread based on thread X and Y index. For each half warp, the memory reads consecutive locations in the global memory, which allows maximum global memory read efficiency. The read pixel data are then stored in shared memory. Each thread is responsible for calculating one element of the output matrix, which is multiple and accumulated one row of the macro block with one column of the constant matrix. Each thread will write back its result back to the global memory.

The second implementation approach utilizes all 512 threads for each thread block. The processing is nearly identical to the first implementation. The only difference is that each thread block is assigned to process 8 macro blocks. For IDCT, the calculation is identical to DCT with a different set of constant matrix.

## 4.2 Quantization



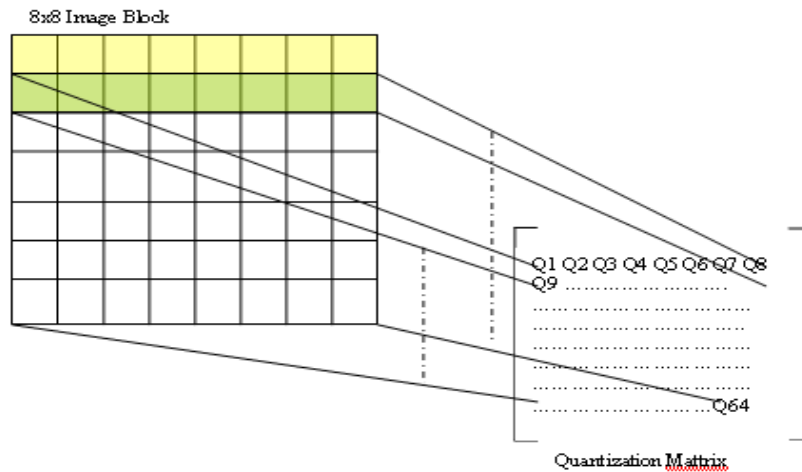
**Figure 4.2a.**

The Quantization algorithm consists of a very simple set of calculations. The image is divided into strides which equals to the nearest 16<sup>th</sup> divisor. Then the Quantization matrix is multiplied to the image width by width. Quantization matrix below is used as a multiplier/divisor.

```
Q[] = 32, 33, 51, 81, 66, 39, 34, 17,  
      33, 36, 48, 47, 28, 23, 12, 12,  
      51, 48, 47, 28, 23, 12, 12, 12,  
      81, 47, 28, 23, 12, 12, 12, 12,  
      66, 28, 23, 12, 12, 12, 12, 12,  
      39, 23, 12, 12, 12, 12, 12, 12,  
      34, 12, 12, 12, 12, 12, 12, 12,  
      17, 12, 12, 12, 12, 12, 12, 12
```

The first row from 0-7 is divided by the Quantized indices from 0-7 and then the same quantized indices are used for the rest of the row till the width of the image. The second row of the image is divided with the 2<sup>nd</sup> row of the quantized matrix 8-15. Again the same indices of the quantized matrix are used for the rest of this row. This process continues till the whole image is divided by the Quantized matrix as shown in the Figure 4.2a. The division is rounded and to the nearest integer and the same corresponding Quantized matrix is multiplied again to the same image using the same procedure as above. CPU implementation is straight forward and uses 2 for loops one to loop around the height and one to loop around the width.

Threads are set are to 8x8 and the grid is set up for width of the image divided by 8 and the height of the image divided by 8. Each 8x8 thread of the image is multiplied with the corresponding index in the quantization matrix. Figure 4.2b below shows how the CUDA threads are used.



**Figure 4.2b CUDA thread assignments for Quantization**

The first CUDA implementation used the exact same logic as the CPU. The threads are multiplied and divided to quantize the image but the image is still a part of global memory.

The second implementation used shared memory. The global image was divided into 8x8 chunks and this matrix grid was quantized with the quantized matrix and then copy back to global memory.

The third implementation used shared memory as well but with a small difference when compared with the second implementation. The shared memory was loaded with the division of the global image and the quantized values and then the shared copy was used to multiply back the quantized matrix.

### 4.3 Huffman Encode/Decode

Huffman encoding is a very effective lossless compression. It utilizes the frequency of occurrence of each symbol to generate a variable length encoding. In the encoding the frequent symbols are shorter and less frequent symbols are longer, with the overall effect that the entire encoded bitstream is shorter than the original. For images, this provides between 2x to 4x compression for typical images.

The algorithm for building the encoding follows this algorithm

```

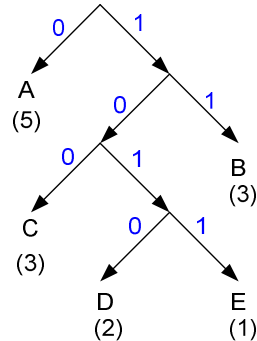
Each symbol is a leaf and a root.
While (forrest is not a single node) {
    Create new node
    new node is parent node of two least frequent nodes
    Remove two least frequent nodes from the forest
    Insert new node into the forest
}

```

Encoding is the last step in JPEG processing. Encoding can be broken down into few steps:

1. gathering frequency information
2. building encoding tree
3. building encoding table
4. actual encoding

A - freq = 7  
 B - freq = 3  
 C - freq = 3  
 D - freq = 2  
 E - freq = 1



## ENCODING

Symbol	Encoding	Length
A	0	1
B	11	2
C	100	3
D	1010	4
E	1011	4

## Encoding Table

### Step 1, gathering frequency information.

In this step, the goal is to count each symbol in the entire image. At this point the image is just a collection of bytes, and hence there are 256. Individual symbol counters need to be unsigned integers, which is large enough for images up to 4GB. There are various ways to utilize CUDA to count symbols. One way is to run 256 threads. Each thread would count one symbol. Second way is to divide the image into manageable blocks, let's say 1K each. Each thread would count symbols in each block, and in the end add them all up. The third way is to perform the counting at the same time that the image is quantities. This experiment was not performed in this project.

### Step 2, building encoding tree

In this step, we performed the classic Huffman algorithm on the CPU. At first glance, there is parallelism opportunity to perform sort operation in CUDA. When the encoding tree is built, the first step is to sort it by frequency. Then two lowest nodes are combined into an intermediate node, and that new node is inserted back to the forest. This cycle is repeated until there are no nodes left. Sorting is performed during each iteration.

What's not obvious is that during consecutive iterations, sorting operation is most efficient as a single iteration of bubble sort, while sorting in CUDA should be performed as RADIX sort.

There exists opportunity to optimization, however there are only 256 symbols. This part of encoding is too minor to consider for CUDA.

### Step 3. Building encoding table



Encoding table can be easily built by traversing up the tree for each symbol. Each symbol is addressed by the original value. It required two pieces of information, encoded symbol and encoded symbol length. This process is straight forward enough that it could be optimized for CUDA, but tree traversal would likely result in memory conflicts on every access. Same argument applies as in step 2, there are only 256 symbols.

## Parallel Encoding

In this experiment the encoding table is used. The goal is to produce write one encoded symbol with one memory access. Traversing through a tree structure does not seem effective for CUDA. The encoding table holds the encoded value in a short. The assumption is that 256 symbols would result in encoding were the least frequent symbol is at most 16 bits long. This is an invalid assumption, as in the worst case that encoding could be 257 bits long.

Example:

A - 0

B - 10

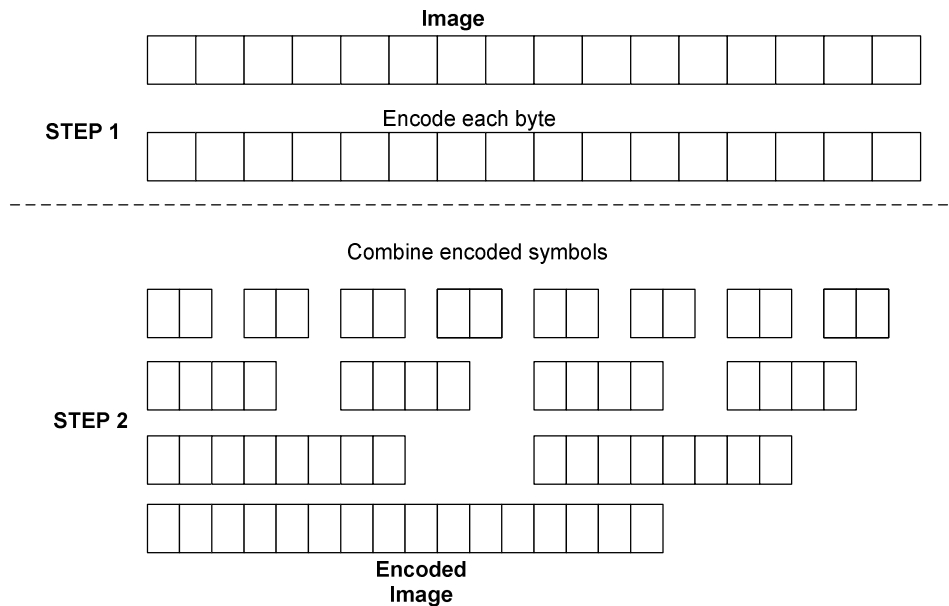
C - 110

....

Z - 1111...11110

This length assumption is necessary in order to use a table rather than encoding tree. Encoding tree is a table and it needs to have a fixed size of each element. By adopting Short as a data character, it allows us to fix the encodings in 4 data lines, and length information (chars) in 2 data lines. Each thread in a halfwarp can access the entire table in max of 4 accesses. To speed it up, the encoding table is moved to shared memory.

Encoding is divided up to into two phases. Phase 1 is encoding each byte individually. Phase 2 is combining the results.



Phase 1. Each byte is encoded individually by accessing the translation table. This step is extremely fast and it offers 100x speed up.

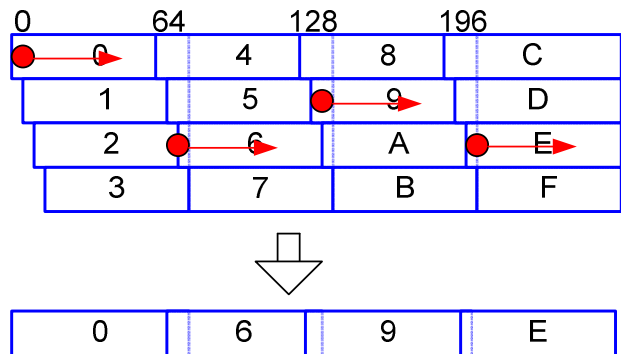
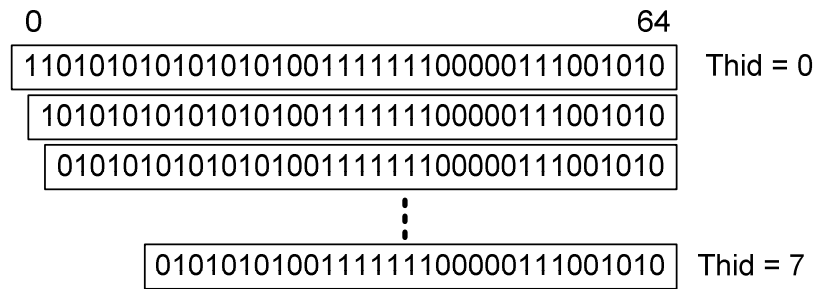
Phase 2. is an iterative phase. In the First iteration two symbols are combined, by concatenating the second symbol on top of the first one. In the second two sets of two symbols are combined into four, and so on. This seemed to be the simple algorithm and it showed 10x speed up for 1K image. What was not obvious is that the algorithm,  $O(n \log n)$  is invalid because it is of higher complexity then canonical CPU algorithm –  $O(n)$ . This algorithm is  $n \log n$  because in each iteration half of the image data is copied over to new location.

Other methods were not attempted.

## Parallel Decoding

Decoding is tricky because the only way to get some speed is to decode at several places at the same time. The problem is that in a variable-length encoding one does not know where to symbols start.

Obvious solution is to start multiple times at different bit offsets. One of threads must have started at a symbol boundary. One characteristic of variable length is that they are unable to detect that decoding is invalid due to wrong starting point. Decoding will just continue until the input bitstream is complete. Deciding which thread is the correct one will not be known until the previous section of the image was decoded. Combining resulting fragments is a serial process, but that can be divided into two parts: determining which segments are valid and coping data. Only the first part is serial in nature; copying data can be done in parallel.



In this project, we concentrated on parallel decoding streams. There were various challenges in developing the algorithm. Primary problem is how to avoid memory conflicts when the data stream is not regular in length. In the input each symbol has variable length, which can result in a very short or very long decoded stream. In any Huffman decode table it is a valid solution that the most common symbol has 1 bit encoding. This means that any 64B segment of input data can result up to 512B of output data. With 512B output per thread and multiple redundant execution threads, we quickly run out of memory.

This experiment was treated as proof of concept rather than complete solution. Following shortcuts were made. Each thread would be limited to 128B output. This can be guaranteed by using a encoding with minimal length of 4 bits per symbol. Another simplification made was to assume that maximum symbol length is 8 bits. The reason for that was to reduce the size of encoding table.

Our decode used following CUDA methods:

- encoding table into shared memory, symbols and lengths separately
- 64B input bitstream is moved into shared memory.
- each thread loops 128, once per output byte.
  - Thread sync on each iteration was attempted.
- Output bytes are written vertically to avoid collisions.
- Each 64B input stream is run 8 times
  - Each iteration has different bit offset on first byte.
- CUDA is programmed to 32 threads per block

- This means 16 threads use only 2 x shared 64B of input data without collisions. And the entire input bitstream for a half-warp fits on one address line.

Results of the experiment were not good. Code was tested on 1kB, 2kB, 4kB, and 16kB data streams, and results were consistent. CUDA code was 3 times slower than naïve CPU implementation. I expect that there were memory collisions, but I was unable to find the cause.

One curious observation is that, when decoding table was moved to read only memory, performance dropped to 46 times slower than CPU.

## **5. Evaluation Setup and Methods**

### **5.1 Image Format**

The images used by the jpeg algorithm are of bitmap format. The bitmap library consists of some basic operations format for the bmp images. These are some of the basic operations

- Calculation of image dimensions
- Division of image into planes and strides of 16 or 8
- Conversion of image into grayscale
- Dumping contents of an array into bmp image and vice versa

OpenGL library in CUDA seems to have a viewing plugin(cudaGLMapBufferObject) for pgm formatted images. There is no plugin in OpenGL for bmp, this imposed a challenge to convert the image processing library to be based on pgm and this enhancement has not been implemented. Current implementation of the JPEG executable takes in only bmp images.

## 5.2 Image types

The JPEG processing time largely depends on the sizes of the images used. The following images and sizes were used for the evaluation of the JPEG performance for the CPU run and the CUDA execution.



Image size 512MB



Image Size 1024x768

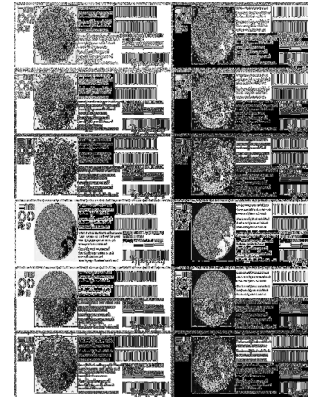


Image Size 2048x2048

**Figure 5. Images used for performance evaluation**

## 5.3 Metrics

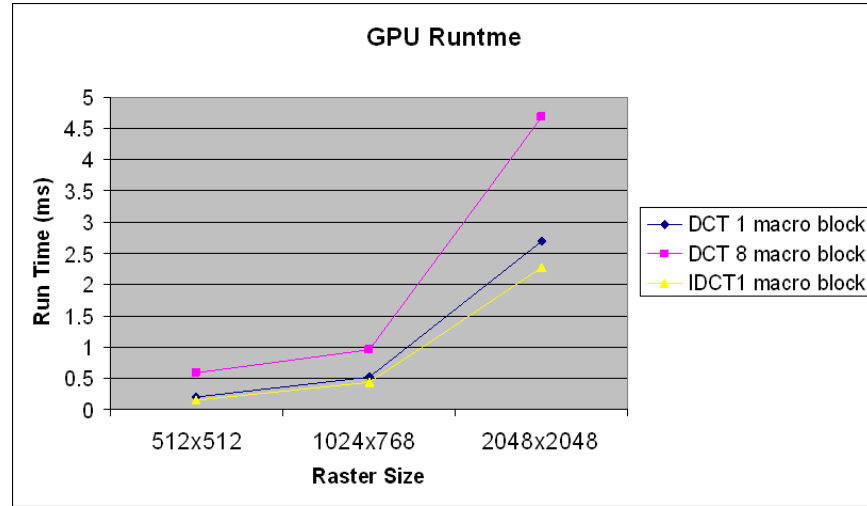
These are the measurements and results taken.

- Processing time for each stage of the jpeg pipeline is measured. Both time taken by the CPU and GPU were measured. These measurements do not include time to copy the image from CPU to device space. All times are measured in ms using the CUDA timer start/stop routines. DCT/IDCT and quantization also had couple of CUDA optimizations and each method was also timed.
- Processing time for the complete jpeg is measured as a sum of all the execution time of the individual stages of the pipeline.
- Speed is calculated a division of the time image processing takes in CPU verses in GPU.
- PSNR ratio – PSNR is measured between the CPU processed image and the GPU processed image. This is the peak signal to noise ratio and is calculated by comparing 2 images pixel by pixel and accumulating the difference. The exact formula used is  $10\log_{10}(255*255/(\text{Accumulated difference}/\text{Image dimensions}))$ . The larger the value the more close the images are.
- Image comparison – Manually check to make sure images are visibly same as the original image and the cpu processed image.

## 6. Results

### 6.1 DCT & IDCT

The runtime for two DCT implementation and IDCT are shown in Figure.

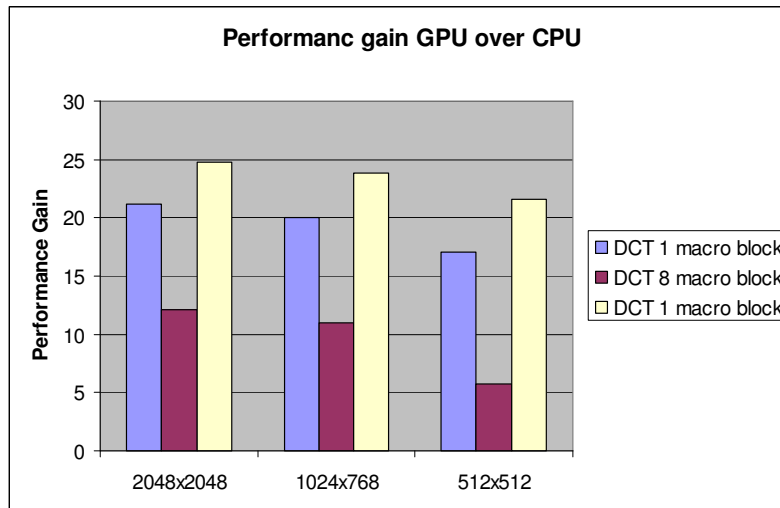


**Figure. Runtime for DCT and IDCT for different GPU implementation**

For all three results, the runtime does not increase linearly with image size. The overhead in large image could be caused by contention to the shared floating point unit, as the DCT computation is floating point intensive. It is unlikely that non-linearity is resulted from limitation in global memory bandwidth as the image size is relatively small compares available global memory bandwidth.

For DCT, the implementation with one macro block per thread block has much better performance compares with the eight macro block implementation. By utilizing more thread blocks, it allows more pipelining between thread blocks while thread blocks are waiting for pixels data read from global memory.

For IDCT, the kernel is nearly identical to the DCT 1 macro block implementation. IDCT has better performance because the pixel values are quantized. Because of the quantization, half of the operands are round to the nearest integer which speeds up the floating point multiply for the matrix multiplication.



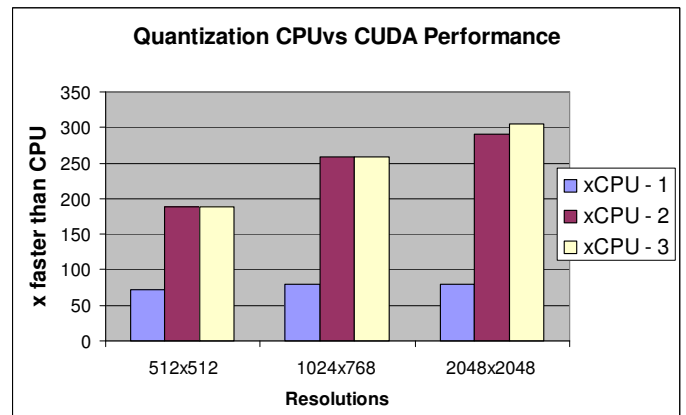
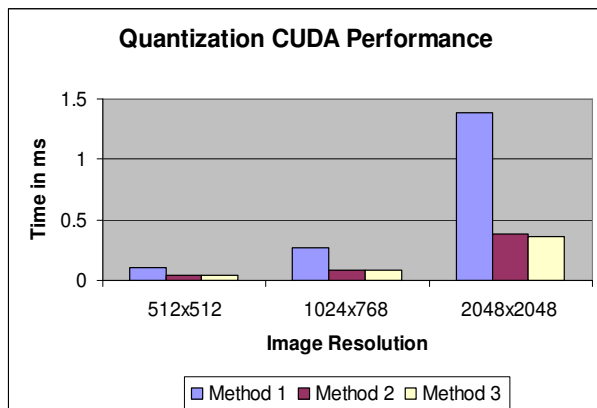
**Figure. Performance gain of GPU over CPU for different DCT and IDCT implementation**

In Figure, it shows the performance gain of GPU over CPU implementation. Using one macro block per thread block, DCT and IDCT both have over 20x performance gain over the CPU implementation. For eight macro blocks per thread block, DCT has only 10x performance gain because the lack of pipelining effect between thread blocks.

## 6.2 Quantization

### CUDA Programming

- Method 1 – Exact implementation as in CPU
- Method 2 – Shared memory to copy 8x8 image
- Method 3 – Load divided values into shared memory.



**Figure 6.2 :** a) the processing time for all 3 CUDA implementation  
b) comparison of each method against CPU vs. GPU

Figure 6.2a shows the processing time for all the 3 CUDA implementations. Figure 6.2b shows how each method performs compared with CPU. The processing time for CPU was divided by the time taken by the CUDA implementation for all the different resolution sizes.

- Method 2 and Method 3 have similar performance on small image sizes
- Method 3 might perform better on images bigger than 2048x2048
- Quantization is ~x70 faster (512MB) for the first method and much more as resolution increases.
- Quantization is ~ x180 faster (512MB) for method2 and 3 and much more as resolution increases.

	Method 1	Method 2	Method 3		xCPU - CPU1	xCPU - 2	xCPU - 3
<b>512x512</b>	0.102	0.039	0.039	7.37	72.2549	188.9744	188.9744
<b>1024x768</b>	0.274	0.085	0.085	22	80.29197	258.8235	258.8235
<b>2048x2048</b>	1.39	0.379	0.36	110	79.13669	290.2375	305.5556

**Table: Result of Quantization**

### 6.3 Overall Results

In table, it shows the combined runtime result of DCT, quantization and IDCT for 2K x 2K images. By combining 3 stages, CUDA implementation is able to achieve 13 times speed up compared to the CPU implementation. In terms of overhead, CUDA has 64 percent of overhead spent for copying data from host memory to GPU memory and back. Even with the significant amount of overhead, GPU is still an optimized platform for parallelizing DCT and Quantization.

	CPU (ms)	%	CUDA(ms)	%
<b>DCT</b>	63.17	31.82	2.7	17.85
<b>Quant</b>	79.01	39.80	0.379	2.50
<b>IDCT</b>	56.3	28.36	2.26	14.94
<b>Overhead</b>	0.02	0.01	9.791	64.71
<b>Total</b>	198.5		15.13	
<b>Speedup</b>	13.11963			

**Table: Combined Results of DCT, Quantization and IDCT for 2Kx2K images**



## 7. Conclusion

It has been shown that JPEG compression algorithms has the potential for significant speedup using GPU processing techniques within the CUDA architecture. This speedup can be achieved only when writing algorithms and function calls specifically with data level parallelism in mind and when the cost of moving data across a narrow memory channel can be amortized by many calculation based operations which can be execute in parallel across many vector processors.

It has also been demonstrated that cooperative algorithm management between the system CPU and an available GPU is an effective method for performance improvement when the respective strengths of the two chipsets are kept in mind. This was demonstrated in the rewriting of the JPEG compression algorithm to perform lossy encoding on the GPU and Huffman encoding and decoding on the CPU.

A further conclusion that can be drawn is that the increased memory bandwidth between the system CPU and GPU will lead to more effective utilization of GPUs for performance improvement. This can be further extended as future work in current implementation by benchmarking how data can be rearrange and calculated in JPEG. In the event that GPUs become integrated into shared memory processor cores or the memory bandwidth of today's systems increases significantly, the implementations developed will all achieve greater speedups.

## 8. References

- [1] NVIDIA CUDA SDK Brower, [http://www.nvidia.com/object/cuda\\_get.html](http://www.nvidia.com/object/cuda_get.html)
- [2] NVIDIA CUDA SDK Code Samples:  
<http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html>
- [3] Wallace, G. The JPEG Still Picture Compression Standard. IEEE Transactions on Consumer Electronics. December 1991.
- [4] International Telecommunication Union. Recommendation T.81 Information Technology – Digital Compression And Coding Of Continuous-Tone Still Images – Requirements And Guidelines. September 1992.
- [5] C.Y. Yam. Optimizing Video Compression for Intel Digital Security Surveillance applications with SIMD and Hyper-threading Technology. Intel Corporation. 2005.
- [6] B. Fang. G. Shen Techniques for Efficient DCT/IDCT Implementation on Generic CPU. May 2005
- [7] M. Benes,. S. Nowick., A Wolfe,. “A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors” , 1998

[8] M.K. Rudberg,. L. Wanhammer, “ New Approach to High Speed Huffman Decoding“, 1996

[9] S.T. Klein,. Y. Wiseman, “Parallel Huffman Decoding with Applications to JPEG Files“, 2003