1. ABSTRACT:

The aim of this project is to explore and test the new potential performance improvement that can be achieved by the use of GPU processing architecture for several different types of compression algorithms. The compression algorithms are the choice of focus on the data parallelism on the GPU device. The specific algorithms ported to the CUDA architecture included lossless data compression techniques, Vorbis Audio Encoding and JPEG. The lossless data compression has been chosen as being distinctly unique from the lossy compression techniques like Vorbis and libjpeg. JPEG was considered to be different from audio in that the encoding concepts and significantly enough to contain divergent opportunities for exploitation of vector processing and data level parallelism.

2. INTRODUCTION:

The GPU processing with the data parallelism is making a revolutionary breakthrough in computation. By sharing the work of CPU, the GPU computation is making the computers to be faster and less power consuming. To make this process even better, the compression techniques are being used by the NVIDIA Corporation. CUDA (Compute Unified Device Architecture) is the implementation on NVIDIA Corporation. They implemented this parallel architecture in their GPUs and provide APIs for the programmers to develop parallel applications using the C programming. CUDA provides software abstractions for the hardware architecture called blocks which are group of threads (512 threads per block) those can share memory and synchronize the processing. Eight scalar processors make up a multiprocessor and different models of GPUs contain different multiprocessor counts.

Compression is vital and useful technique that can be slow on CPUs. I wanted to investigate the ways to parallelize the compression specifically on GPUs where hundreds of cores are available for computation. By investigating four forms of compression (DXT, ogg, JPEG, zip), we can characterize what components and forms of compression are suited to parallelization. In addition, my goal was not only parallelize application for a new architecture but also to find strengths and weakness of the architecture.


3. PREVIOUS WORK:

High Quality compression algorithms implemented are highly expensive when the other compression methods are available in [1] and [2], the resulting quality of those simplified
methods is not very high. Cheap compression algorithms have been implemented using the traditional GPGPU approaches [3]. But [7] addresses few improvements over these algorithms. This is a new field which has to be explored more. The developers from NVIDIA is keep on trying to optimize the compression techniques in GPU. [8] is one of the good example of it, where they implement DXT algorithm.

4. OGG (VORBIS) AUDIO COMPRESSION:

4.1. Algorithm:

The Ogg audio format uses the Vorbis encoder to transform time domain sampled audio data into a compressed format using frequency transformation techniques and Huffman coding. The Modified Discrete Cosine Transform (MDCT) is used which blends one frame into the next. Frames are passed from the Vorbis tools application into the Vorbis encoder library. Once the library is called, the transform’s first step is to multiply the time domain sampled data by a Pulse Coded Modulation (PCM) windowing function which emphasizes the middle portions of an individual frame over the boundaries. Subsequently, it passes the data to the transform section wherein butterfly patterns of the variety typically used in Fourier transforms are applied to it. Noise and tone masks are also applied as well as manipulations using a spectral floor which eliminate unimportant information from the audio data such as inaudible frequencies.

The time taken to encode a wav file to this format varies significantly depending on the input data set size, but can take up to five minutes for a single song of about ten minutes. The vast majority of the time spent in encoding a song is in the vorbis_analysis function call which maps to the mapping0_forward call in the Vorbis library. This function itself calls many distinct functions within the library, many of which employ pointer arithmetic.

4.2. Optimizations:

The first CUDA enabled optimization implemented on the library was in the application of the PCM window function. This operation involved multiple floating point multiply operations on two one dimensional data arrays, one being the sampled PCM data and the other being a predefined windowing array. Since the window arrays were predefined based on length, they were passed to the global memory of the GPU processor on initialization of the encode engine and only freed when subsequent encoding was complete. Since the Vorbis tools application that initiated encode called vorbis_analysis for every frame of data, each function call involved a memory copy of the data to the GPU, the subsequent floating point calculations, and a copy operation to move the data back to the CPU memory for continued processing. In order to avoid the performance penalty of divergent control paths, the CUDA kernel that parallelized this operation was only invoked in the common case where the window sizes used were of equal length. In cases where the window sizes were different, the CPU only function was called. In practice, using the tested input data, the uncommon case of unequal windows occurred less than one percent of the time. Several benchmarks were run with this CUDA kernel as the only CUDA enabled function.
The next operation that lent itself well to vector processing according to program analysis was the function which removed the spectral floor from the nearly transformed data. This operation stored the results of the multiplication of two floating point arrays to a third value. Unlike the PCM windowing function, wherein only the frame data needed to be transferred with each iteration, all three arrays needed to be transferred to the GPU in this case and the result copied back, yielding a total of four memory operations. The third optimization implemented was in the mdct_forward function which performed the beginning of the MDCT. This involved three loops of different execution patterns and the previously mentioned complex pointer arithmetic operations. An example of a loop and its transformed vectorizable counterpart is given below.

```c
int n=init->n;
int n2=n>>1;
int n4=n>>2;
int n8=n>>3;
DATA_TYPE *w=alloca(n*sizeof(*w));  //Allocate memory
DATA_TYPE *w2=w+n2;
REG_TYPE r0;
REG_TYPE r1;
DATA_TYPE *x0=init+n2+n4;
DATA_TYPE *x1=x0+1;
DATA_TYPE *T=init->trig+n2;
for(int i=0;i<n8;i+=2)
{
    x0 -=4;
    T-=2;
    r0= x0[2] + x1[0];
    r1= x0[0] + x1[2];
    w2[i]= MULT_NORM(r1*T[1] + r0*T[0]);
    w2[i+1]= MULT_NORM(r1*T[0] - r0*T[1]);
    x1 +=4;
}
```

**Code: 1 Sequential MDCT for Loop**

```c
__global__ void
mdct_for_loop1(float* dev_w2, float* device_d, float* dev_trig,
int n2,
int n4)  //This function will run in GPU
{
    int i = ((threadIdx.x * WINDOW_GRID_SIZE) + blockIdx.x) * 2;
    dev_w2[i]= ((device_d[n2+n4 - 4*(i/2)+1]) + device_d[n2+n4 +
        4*(i/2)+3])*dev_trig[n2 - (2*(i/2)+1)] + (device_d[n2+n4 -
        4*(i/2+1)
        + 2] + device_d[n2+n4 + 4*(i/2) +1])*dev_trig[n2 -
        (2*(i/2+1))];
    dev_w2[i+1]= ((device_d[n2+n4 - 4*(i/2+1]) + device_d[n2+n4 +
        4*(i/2))
```
\begin{align*}
+3])^* & \text{dev} \_\text{trig}[n2 - (2*(i/2+1))] - (\text{device} \_d[n2+n4 - 4*(i/2+1) + 2] + \\
\text{device} \_d[n2+n4 + 4*(i/2) +1])^* & \text{dev} \_\text{trig}[n2 - (2*(i/2+1))+1])
\end{align*}

\textbf{Code:2 Kernel Implementation of CUDA Program}

\section*{4.3. Performance Benchmarks:}

The performance benchmarks listed in Table 4 were taken on a Red Hat Linux based system using an NVidia 8800 GTS 512 which had a memory bandwidth between CPU and GPU space of 1.5 GB/s.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU only sequential version</td>
<td>1.20</td>
</tr>
<tr>
<td>GPU Spectral Floor Calc &amp; PCM Window</td>
<td>0.32</td>
</tr>
<tr>
<td>GPU PCM Window only</td>
<td>0.62</td>
</tr>
<tr>
<td>GPU PCM Window only, memcpy removed</td>
<td>1.25</td>
</tr>
<tr>
<td>GPU Spectral Floor Calc &amp; PCM Windows &amp; MDCT for loops</td>
<td>0.84</td>
</tr>
<tr>
<td>GPU PCM Windows &amp; MDCT for loops</td>
<td>0.98</td>
</tr>
<tr>
<td>GPU PCM Windows &amp; MDCT for loops, memcpy removed</td>
<td>1.37</td>
</tr>
</tbody>
</table>

\textbf{Table: 1 Benchmarks}

From the data gathered, several conclusions can be drawn. The first and most obvious conclusion is that the spectral floor calculation, which requires the overhead of copying three floating point arrays to the GPU and only involves one floating point calculation, never results in a performance gain. In fact, performance suffers significantly overall. Removing the spectral floor calculation the PCM windowing function by itself can be seen to negatively affect performance. However, if the memory copy operations are removed, the speedup can be seen to be 1.25. Assuming, for the purposes of our benchmarks, that the GPU is otherwise idle when not processing the CUDA application, any speedup can be considered an improvement (without taking into account any efficiency metrics).

After replacing the pointer arithmetic of the original beginning for loops of the MDCT with complex index calculations, thereby amortizing the cost of the memory copy operations, the speedup can be seen to be 0.98. It is clear from this benchmark, which brings the performance of the CUDA enabled system to near parity, that amortizing the cost of transfers between CPU and
GPU across many calculate intensive benchmarks is the only possible method to achieve performance gains. Removing the memory copy operations, the speedup achieved is 1.37.

4.4. Future work:

Given the architecture of today’s systems, it is possible to continue to re-architect the Vorbis library in order to take further advantage of an available GPU. However, the porting exercise involved in transforming all of the portions of this algorithm into a format that is more conducive to vector processing is time consuming, tedious, difficult to debug and involves the removal of many intuitive elements achievable by pointer arithmetic. A specific example of an area which could be further improved is in the substitution of the standard qsort with a GPU floating point quicksort. An implementation of this type of quicksort is planned for release by the Chalmers University of Technology [4]. Other examples include more instances of pointer unrolling and transformation of additional functions as well as the removal of control statements to optimize with CUDA for the common cases.

5. LOSSLESS COMPRESSION:

5.1. Algorithm:

Data compression, unlike JPEG and Vorbis, must be a lossless compression, so that data can be exactly recovered from the compressed data. The Lempel-Ziv 77 (LZ77) compression technique was used to represent lossless data compression. The LZ77 algorithm (unlike the LZ78) is not patent protected, and is used in many open-source compression tools, such as gzip. Rather than using an existing library implementation, the LZ77 algorithm was coded from scratch. Most compression implementations also use Huffman coding after the LZ77 pass to further compress the data, but in this implementation only LZ77 is used.

The LZ77 algorithm works by using two buffers: a backwards search buffer, and a lookahead buffer. The input is scanned from beginning to end. Given a certain position in the input file, the search buffer will contain the $S$ previously scanned bytes, and the lookahead buffer will contain the next $L$ bytes. At each new input, the longest substring in the search buffer that is a prefix of the lookahead buffer is determined. The output at that location is then coded as $(B, C, N)$, where $C$ is the number of bytes to copy from the search buffer to the output, starting at the current location minus $C$ bytes, and $N$ is the new byte (not contained in search buffer) to append at the end. In other words, the algorithm works by finding data that has already been seen, and encoding it in a shorter format.

The algorithm doesn’t use very much complex arithmetic. Rather, most of the operations involve reading memory locations and comparing their contents (the inner loop is the substring matching).
5.2. Parallelism:

The parallelism in compression is data level parallelism. The input file can be broken into chunks of contiguous data, and each chunk can be compressed independently. Because LZ77 doesn’t use any explicit dictionary, the data can be split at arbitrary positions and the results will still be correct. However, the compression ratio may suffer, since the search buffer is effectively empty at each chunk boundary, resulting in the first $S$ bytes of each chunk potentially being sub-optimally compressed. In this implementation, the search and lookahead buffers were small (Less than 127 elements), so this had minimal impact on the compression ratio. The algorithm was first parallelized using pthreads. The input data was broken up into chunks of a defined size, and each pthread was responsible for compressing its designated portion of the input. Locking was avoided by giving each pthread its own output buffer (big enough to hold output of the worst case compression ratio). Then, after all pthreads had finished, the output was written by reading the correct number of bytes from each output buffer. While this approach may not be very efficient in terms of memory usage, it allowed for a more direct comparison with the CUDA implementation, since synchronization and locking would have been very expensive in CUDA. The implementation in CUDA was similar to pthreads. First, the input data was transferred from CPU (host) memory to GPU (device) memory. The input file was kept relatively small so that it could all fit in memory at once.

However, processing a larger file would simply be a matter of streaming the input file into memory in some defined memory block size. After being transferred to device memory, the CUDA kernel operated on the data and wrote the results to an output buffer in device memory. Each thread would access its respective portion of the input buffer, and write results to the output buffer. The size of each thread’s input and output was dependent on the number of threads and the total input size. Various combinations of number of blocks and threads per block were evaluated.

5.3. Benchmarks and Results:

The compression methods were tested using a 50 MB input text file. The results using pthreads are shown below in Figure 5. Note that the output file size was about 26 MB, and did not vary significantly when increasing the number of threads. These results were obtained from running on an 8 core Xeon configuration (Cyclades). The results of the CUDA implementation, using various numbers and sizes of blocks, are shown in Figure 6. The speedups are relative to a single threaded CPU implementation. The CUDA results were obtained using a GeForce 9600M GT graphics card. The compression results using CUDA varied by a few hundred kilobytes for all test cases except the 400 x 400 case, in which the output was about 7 MB larger.

In analyzing the CUDA results, it can be seen that, firstly, the arrangement of the data into blocks and the size of the blocks is not as important as the total number of threads (block size x number of blocks). The best results were achieved with the highest total thread count, of 400 x 400, which resulted in a 1.36x speedup over the single threaded version. Interestingly, using only a single block with a single thread results in only a minor slowdown compared with a single threaded CPU version. However, the CUDA results in general pale in comparison to the pthreads results.
6. JPEG:

The first compression algorithm we attempted to port to CUDA was JPEG image compression. 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 used Jpeglib 6b provided by the Independent JPEG group. This is the standard serial JPEG implementation used in most JPEG compression and decompression code. At first, I attempted to port the code from the library and convert many of the serial functions to CUDA functions. After reading through the library, I found that the pointer intensive nature of the code and the row streaming algorithm did not lend itself to CUDA code. As a result, I decided to write our own CUDA implementation of JPEG compression following the same algorithm and flow of Jpeglib but optimized for processing the whole image rather than processing individual rows.

6.1. Algorithm:

![Figure 2: Process of JPEG Compression](image)

In order to find which part of JPEG compression could be parallelized, I investigated the algorithm which performed the compression itself [1] and [2]. Figure 1 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.

Compression begins by loading an uncompressed image in the form of 24-bit bitmap image with 8-bit red, green and blue values for each pixel. The image is loaded into the main memory and the Jpeglib performs operations on blocks of 8x8 pixels called MCUs (minimum coded units). These blocks facilitate frequency operations such as the discrete cosine transform (DCT) later in the algorithm. The serial Jpeglib works on MCUs row by row. The first step after loading the image is to transform the RGB values to YCC (luma, chroma blue, chroma red). This is simply a set of linear operations on the three RGB values for each pixel in the image. The purpose of the conversion is to separate the brightness (luma) of the image from the individual colors. Since the eye is more sensitive to brightness, we can save space by downsampling the chroma component and using a different quantization table. Downsampling can be performed at ratios of 4:4:4 (no downsampling), 4:2:2 (reduce by factor of 2 in horizontal direction), or 4:2:0 (reduce by factor of 2 in horizontal and vertical directions). All operations now operate on the 3 channels separately and in the same manner. The results are shifted to center around 0 (subtract by 128) to prepare for the next step.

\[
G_{u,v} = \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} g_{x,y} \cos \left( \frac{\pi}{8} \left( x + \frac{1}{2} \right) u \right) \cos \left( \frac{\pi}{8} \left( y + \frac{1}{2} \right) v \right)
\]
After the optional downsampling, a 2D discrete cosine transform (DCT) is performed on each 8x8 pixel MCU to separate it into its frequency components. The equation for the DCT is shown above. The result at the top left corner (u=0, v=0) is the DC coefficient and the other 63 results of the MCU are the AC coefficients. As u and v 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 u and v) and the coefficients there are usually low since most MCUs 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 u and v). At this point, all the lossy compression has occurred, meaning that high frequency and some chroma 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 MCU 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. The result is that Huffman encoding works on the entire bit stream and the MCU boundaries are lost in the final JPEG encoding.

6.2. Approach:

Our approach in parallelizing the JPEG algorithm was to break the data level parallelism up in each block in Figure 1 and write a kernel for each of the parallel tasks. The green blocks in the figure were parallelized in our implementation. 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 MCUs 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 to the device (GPU) memory in addition to the host (CPU) memory. After the kernels execute, the image data then has to be loaded from the device back to the host and written to disk. Fortunately for JPEG, there are a couple of reasons why this introduces very little overhead compared to ogg and zip compression. First, the size of images is small enough to fit in the entire device memory. Second, we only need two calls to cudaMemcpy since there is no need to stream data in or out, just one call to bring the whole image into global device memory and one call to bring the results back to host main memory.

The color conversion step is very conducive to the CUDA parallel architecture since there are absolutely no dependencies between any of the pixels. Once the 3 components of the image were brought in to the device global memory, we created three threads for each pixel to do the
conversion. There is no need to bring the data into shared memory, which can be thought of as a cache, since we do not reuse the data during processing. For each block in the CUDA abstraction, we assigned a thread dimension of 8x8x3 so that three threads operate on the 8x8 pixels in an MCU per block. This left the block dimension to be width/8 by height/8 per grid, and resulted in 8x8x3=192 threads per block and width*height/64 blocks for a total of width*height*3 threads for the whole image.

In our implementation of JPEG compressor, we did not downsample the chroma components since it is optional. The downsampling can be easily added to the above kernel by not calculating and writing back all of the chroma components after the calculation to the device global memory. Parallelizing the discrete cosine transform (DCT) was more complex because for every element in an 8x8 matrix, we needed to add all the elements in the same column and row and use additional multiplication operations. To parallel this 2D DCT, we performed 2 1D DCT operations and added the results to each other to get the final 2D DCT coefficient matrix. We broke each 1D DCT into 8 threads, one thread for each row/column. In the parallel DCT, shared memory becomes important since we need to re-access many of the elements in the matrix. In CUDA’s block abstraction, we assign 4x4x24 threads to each block, resulting in width/32 by height/32 blocks. This configuration resulted in 16 MCUs (4x4) per block and 24 threads (8 threads per component * 3 components) for each MCU for the 1D DCT. Each block, therefore, is sharing about 3KB of data for the 16 MCUs. The total number of threads for the whole image, then, is width/8*width/8*24, or 24 threads for each MCU.

Parallelizing the quantization after the DCT step was much like the conversion step since it is a per pixel operation independent of other pixels. We simply took the DCT coefficient calculated above at that pixel and divided it by the quantization coefficient at the MCU location specified by the quantization table (generated from the image quality per JPEG specification). In terms of CUDA’s abstraction, we have 8x8x3 threads per block and width/8 by height/8 blocks. Each block operates on one MCU with 3 threads per pixel (one for each component of YCC) for a total of width*height*3 threads. No shared memory is necessary since there is no need to reuse data. Zig-zag reordering is another per pixel operation. Each element in the quantized DCT matrix is placed in a new location to gather all the high frequency coefficients at the end of the stream. We used the same CUDA block construct as above (8x8x3 threads per block and width/8 by height/8 blocks) and a position lookup table to determine where to place the coefficient.

The final step of entropy coding using Huffman encoding is not parallelized for CUDA for several reasons. First, generating the Huffman table is a serial process that has a lot of dependencies on previous bits of the stream. Also, CUDA is not optimized for control structures so that the decision making steps during Huffman encoding will be slow when executed on the GPU. Huffman is not a math intensive algorithm, but rather control and dependency intensive. Finally, Huffman encoding gets rid of the MCU boundaries and looks at the bit stream as a whole, so the previous approaches at CUDA parallelization is not effective.

My approach in parallelizing JPEG compression for CUDA was to find the math intensive functions and develop kernels that can operate on a per-pixel or per-MCU fashion. I attempted to limit the decision and control structures in CUDA by using lookup tables and more calculation based approach. I found that the lossy portion of the encoder, which includes removing high frequency data and downsampling could be parallelized efficiently with CUDA because they are
more mathematically and calculation based. The lossless compression is much harder because of
the dependencies on the entire bitstream.

6.3. Results:

I compare the performance of our CUDA JPEG implementation with Jpeglib’s standard
compression algorithm. Because we have full control of our implementation, we are able to
breakdown the times into finer granularity than the Jpeglib’s implementation. Jpeglib
encapsulates many of its operations into a JPEG class and operates on data in a per-row manner
so it is difficult to breakdown the operations into its components. The two algorithms are run on
a set of randomly generated images of various sizes (256x256, 512x512, etc) with no
downsampling and equal image quality and quantization tables. Jpeglib is run on an Intel Core 2
Duo E6750 while the CUDA JPEG is run on the same machine with an NVIDIA 8800 GTS 512
(16 mp x 8 sp/mp = 128 scalar processors). The results are summarized below.

<table>
<thead>
<tr>
<th>Image Dim</th>
<th>256px</th>
<th>512px</th>
<th>1024px</th>
<th>2048px</th>
<th>4096px</th>
<th>8192px</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host IO</td>
<td>0.078</td>
<td>0.078</td>
<td>0.28</td>
<td>1.154</td>
<td>4.57</td>
<td>20.55</td>
</tr>
<tr>
<td>Computation</td>
<td>0.045</td>
<td>0.046</td>
<td>0.172</td>
<td>0.686</td>
<td>2.68</td>
<td>10.78</td>
</tr>
<tr>
<td>Total</td>
<td><strong>0.123</strong></td>
<td><strong>0.124</strong></td>
<td><strong>0.452</strong></td>
<td><strong>1.84</strong></td>
<td><strong>7.25</strong></td>
<td><strong>31.33</strong></td>
</tr>
</tbody>
</table>

Table 3: Jpeglib Execution Times (in seconds)

<table>
<thead>
<tr>
<th>Image Dim</th>
<th>256px</th>
<th>512px</th>
<th>1024px</th>
<th>2048px</th>
<th>4096px</th>
<th>8192px</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host IO</td>
<td>0.092</td>
<td>0.094</td>
<td>0.374</td>
<td>1.24</td>
<td>4.82</td>
<td>19.6</td>
</tr>
<tr>
<td>Device IO</td>
<td>0.035</td>
<td>0.031</td>
<td>0.037</td>
<td>0.064</td>
<td>0.156</td>
<td>0.467</td>
</tr>
<tr>
<td>Kernel</td>
<td>0.001</td>
<td>0.001</td>
<td>0.003</td>
<td>0.013</td>
<td>0.042</td>
<td>0.155</td>
</tr>
<tr>
<td>Encoding</td>
<td>0.018</td>
<td>0.015</td>
<td>0.055</td>
<td>0.228</td>
<td>0.92</td>
<td>3.725</td>
</tr>
<tr>
<td>Total</td>
<td><strong>0.146</strong></td>
<td><strong>0.141</strong></td>
<td><strong>0.469</strong></td>
<td><strong>1.545</strong></td>
<td><strong>5.938</strong></td>
<td><strong>23.947</strong></td>
</tr>
</tbody>
</table>

Table 4: CUDA JPEG Execution Times (in seconds)
In both the Jpeglib and CUDA JPEG tables, the host IO refers to the transfer of data from the disk to host (CPU) main memory. Since almost 2/3 of the time is spent simply reading and writing the large files, the overall speedup is reduced. Looking at the Jpeglib’s computation time, this includes everything else from color conversion to encoding. The Jpeglib code does not provide a clear way to breakdown these times.

In CUDA JPEG, we see that the host IO time is very similar to the Jpeglib’s case, since we still need to bring in the data and write out the result. The rest of the time can be broken down as we please in our implementation. We see that the actual time spent running the kernel is very small. Some time is spent on device IO, which is the time to transfer the data between the host (CPU) and device (GPU). This time is small, consistent with expectations for image files in the range of megabytes (for size of 8192x8192, 2*192MB/0.467s = 822 MB/s, a little lower than the measured transfer bandwidth). This can be attributed to the fact that the device IO time also includes the time to allocate and set up the device memory. We see from this data that device IO time is not a significant factor in JPEG compression. The bottleneck rests in the host IO (disk transfer) and the serial encoding process.

Table 5 summarizes the kernel and overall speedup of CUDA JPEG. For small image sizes, there is no benefit from using CUDA, but also no downside as the CPU is freed to perform other operations. For larger images, there is a noticeable speedup (~1.3x) but the disk IO lessens the computation speedup. We also look at the estimated kernel speedup by dividing the computation time of Jpeglib (subtracting out the encoding time) by the kernel time of CUDA JPEG. The kernel speedup reveals a lot more about the improvements of CUDA for computationally intensive parallel applications (~40x). CUDA provided a decent improvement in JPEG compression mainly due to the mathematically intensive and pixel independent nature of image
compression. Specifically, the lossy compression component which includes image conversion, downsampling, DCT, and quantization saw considerable gains. The bottle neck remains the host I/O (disk to memory) and the lossless (Huffman) encoding. But even with the bottlenecks, the parallel computational speedups due to CUDA make up for the serial portion and provide an overall speedup for larger image sizes.

7. CONCLUSION:

In this project, I have compared different compression algorithms and showed several benchmarks and speedups. Implementing new algorithms for compressions through GPU can speed up the process. Future games require more compressing techniques on GPU to make them faster and affordable.

CUDA’s main strength lies in its ability to perform computationally intensive parallel math operations on large datasets residing in its device/shared memory. But its weakness is that it is slow in performing control logic and data dependent operations. CUDA’s need to fit the hardware model of the GPU and the limitations of language also add to its weakness. Finally, for large data sizes, memory bandwidth contributes a large part to the bottleneck.

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 was shown by benchmarking data across all algorithms explored. 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.
REFERENCES:


