Reconfigurable System Development and Evaluation using Standard JPEG Kernels

Timothy J. Davis
Muhammad Z. Hasan

Abstract- Applications have different computation intensive segments called kernels. Computation requirements vary from kernel to kernel in an application. If these kernels could be implemented on hardware, execution time of the application could be reduced as compared to its software implementation. However, due to transitory nature of these kernels and limitation of the hardware resources, it is preferred to implement the kernels on reconfigurable hardware. It ensures better utilization of the precious hardware in embedded systems.

I. INTRODUCTION
There are many embedded applications that must run in computationally constrained environments. These environments can best be taken advantage of by carrying out certain operations in real hardware rather than in software running on general purpose processors. As systems become more and more specialized, it is useful to do complex and repetitive tasks very quickly.

FPGAs (Field Programmable Gate Arrays) can be programmed with relative ease to accomplish difficult tasks in real hardware that is many times faster than accomplishing them in software. This makes FPGAs an ideal choice for something like compressing raw image data into a JPEG (Joint Photographic Experts Group) format. FPGAs have several advantages over both conventional microprocessors and ASICs (Application Specific Integrated Circuits). With an FPGA, changes can be made to the data path inside the hardware to change the flow of control, whereas a microprocessor has a fixed data path. When using custom hardware, like ASICs, if a change needs to be made to the circuit, a new chip (and probably a new device to use the chip) must be designed, manufactured, shipped and installed. If an FPGA were to be used instead, it could be reprogrammed in the field without the need for new or replacement hardware. The FPGA could also be programmed with different kernels at different times for increased flexibility. This ability is useful for JPEG compression because it allows for FPGAs to be programmed with a different step of the compression process, depending on the need of the user. If one step were to take a particularly long time, then multiple FPGAs could be programmed to use the same step. Also, if the JPEG standard was to be changed or a faster compression algorithm was to be developed, then the FPGAs could be programmed with the new algorithm or standard change without the need for replacement hardware.

II. STANDARD BEING USED
The JPEG standard was finalized in 1994 as ISO/IEC 10918-1, to allow raw image data, which can be very large, into much smaller file with a minimum amount of loss. It can be used to achieve up to 10:1 compression ratio without any visible loss [1] for color images. Compression ratios of up to 50:1 can be used with only a small amount of defects. It’s important to note that grayscale images do not compress so well as color images. Grayscale images can be compressed up to about 3:1 without too much visible defect [1]. The limits on the human vision system allow JPEG compression to be effective. The human eye cannot distinguish between very subtle differences in colors, especially when they are grouped close together in an image.

JPEG compression is comprised of three different stages:
1. DCT (Discrete Cosine Transform).
2. Quantization.
3. Encoding.

The compression process begins with the DCT phase [2]. The image is first broken up into eight-by-eight blocks of pixels. If the image is grayscale, each pixel is only represented by a single eight bit number. If the image is color, then each pixel is represented either by three eight bit numbers (representing R, G, and B color components) or by two three dimensional arrays (representing chrominance and luminance). After the image is broken up, each 8x8 matrix (which will be called a P matrix from now on) is processed according to the following equation:

\[
T(i,j) = 0.25 \times C_i \times C_j \times \sum_{x=0}^{7} \sum_{y=0}^{7} P(x,y) \times \cos \left( \frac{(2x+1)\pi i}{16} \right) \times \cos \left( \frac{(2y+1)\pi j}{16} \right),
\]

where \(C_i, C_j\) = \(\begin{cases} \frac{1}{\sqrt{2}} & \text{if } i = 0, j = 0 \\ 1 & \text{otherwise} \end{cases}\)

The output \(T\) matrix now contains spatial frequencies. A spatial frequency is a measure of how much the pixel value changes with respect to their position in the block [3]. The first value in the \(T\) matrix is called a DC coefficient, is closely related to the average of the values in the \(P\) array. The rest of the values in the matrix are called AC coefficients. Higher values of \(i\) and \(j\) are associated with higher cosine frequencies. This means that if the \(P\) matrix contains a single pixel value (meaning that all the pixels are the same), then the cosine functions will cancel each other out and all of the AC coefficients will be zero. It also means that images with lots of change in a small area (the given 8x8 matrix) will have non-zero AC coefficients that represent an image with a lot of fine detail.
The second phase of the compression process is the quantization. It is the process of transforming full, 24 bit color into fewer distinct colors. This makes the image smaller on disk and if the quantization is done correctly, human eye cannot detect any color loss. Since quantization is a lossy process, if done incorrectly or too many times, it can produce significant and visible distortion in the image. A simple way to quantize would be to divide every value in the \( T \) matrix by a set constant to produce a new matrix – the \( Q \) matrix. Because the divisor values are large, and the AC coefficients are generally fairly small, many zeroes will be produced in the \( Q \) matrix by this division, further reducing the amount of data that needs to be stored on disk.

Now that the image has been both transformed and quantized, it can be encoded. The incoming matrix of information will now, for most real images, contain a large number of zeroes and can be easily encoded. There are three different types of encoding that are used to compress the data and produce the final image: Differential Pulse Code Modulation (DCPM), Run Length Encoding and Huffman Coding. Because sometimes the DC coefficient of an image is very large, it isn’t beneficial to encode it with Huffman coding directly. However, the DC coefficients in successive blocks are very close in value. That makes it beneficial to encode the DC coefficient of a \( Q \) matrix as the difference from the DC coefficient of the previous \( Q \) matrix. Then, since many of the AC coefficients are zero, they can be compressed very well with Run Length Encoding (RLE). Consecutive runs of zeroes are looked for in a zig-zag pattern to maximize the length of the runs of zeroes. Figure 1 shows, graphically, how Run-Length Encoding works. The algorithm starts in the upper, left hand corner and works its way down to the bottom, right hand corner.

![Figure 1: Run-Length Encoding](image)

After both DPCM (over multiple blocks) and Run-Length Encoding have been applied on the block, the non-zero numbers in the block will generally be small – either the difference between the DC coefficients in two successive blocks for the DPCM or the AC values after the Run-Length encoding in a single block. Since all of these values are small, they will generally have many of the same bit patterns and will be easily compressible with Huffman Coding. This code replaces the values in the block with a variable length code that is chosen by the frequency of the value.

### III. IMPLEMENTATION

The JPEG compression algorithm was implemented by three separate VHDL modules representing three phases; all linked together in a top module that instantiated one of each kernel and passed data between them.

The first kernel is the quantization kernel, because it was the easiest to code. It involved a divisor matrix of predefined values as shown in Table 1.

<table>
<thead>
<tr>
<th>Divisor Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 5 7 9 11 13 15</td>
</tr>
<tr>
<td>3 5 7 9 11 13 15 17</td>
</tr>
<tr>
<td>5 7 9 11 13 15 17 19</td>
</tr>
<tr>
<td>7 9 11 13 15 17 19 21</td>
</tr>
<tr>
<td>9 11 13 15 17 19 21 23</td>
</tr>
<tr>
<td>11 13 15 17 19 21 23 25</td>
</tr>
<tr>
<td>13 15 17 19 21 23 25 27</td>
</tr>
<tr>
<td>15 17 19 21 23 25 27 29</td>
</tr>
</tbody>
</table>

This is used to produce the \( Q \) matrix as the output from the Quantization kernel. The biggest challenge in writing this phase was that the division of the \( Q \) matrix by the quantization matrix. Both the \( Q \) and quantization matrix contain integers, and when divided should produce another set of integers. A function to divide the numbers was implemented so that values could be rounded up appropriately. It simply made sure that any divisions that had remainders that were equal to or greater than half of the divisor were rounded up and any of the divisions with remainders less than half of the divisor were rounded down.

The second kernel that was created was the DCT kernel. It was the most challenging piece of this entire body of work. The DCT kernel was created first in C for a detailed analysis of the math involved in completing this kernel. It wasn’t a simple task and required much more time for analysis and debugging than was initially thought of. It turns out that the FPGA hardware has, at very best, limited tool support for floating point math, which is an essential component of the DCT kernel.

The final kernel is the Run-Length Encoding kernel. The code that was written follows a custom algorithm that scanned through the matrix in the zig-zag pattern of Figure 1 and found the runs of zeroes. It employed a series of loops that correctly implements the zig-zag pattern in the code. This final kernel is the last step in the JPEG compression process.

### IV. SIMULATION

One problem that was considered in the creation of the JPEG compressor in an FPGA was the use of FPGAs to run portions of the JPEG compression process, each in an individual FPGA. A special type of computer was used (a Starbridge HC-62) to measure the execution speed of each piece (or
kernel) of the JPEG compression algorithm that was used when the kernel was given a set of data.

A software simulator was written in C to emulate the behavior of this real, dynamically reconfigurable system. The simulator was written to gather data from several different methods of running five different kernels on three to five FPGAs with three possible sizes for data sets. These kernels and data sets were used to create a random ‘task graph’ that will be run on the FPGAs and the Host computer. This task graph simply contains an ordered list of the kernels to be executed, each with a specified size data set. Along with the original Break-Even policy (which tried to execute the kernels on the fastest platform – either the Host or an FPGA), there were four other schemes for running kernels on an FPGA:

1. Uniform Utilization I
2. Uniform Utilization II
3. Load Balancing with Uniform Utilization
4. Load Balancing with Look-Ahead

The first two policies, Uniform Utilization I and II, were to ensure, in slightly different ways, that the FPGAs had similar utilization times. It is important to keep each FPGA utilized similarly to minimize disparity of usage and local temperature build-up. Uniform Utilization I will always relocate a kernel if it will lower the disparity in the utilization of the FPGAs. This algorithm creates the most uniform utilization of all of the algorithms.

Uniform Utilization II ensures uniform utilization by moving the kernel to be executed to a different FPGA if the overhead for moving it will not increase run time beyond the host time. For example, if kernel A will take 10s to execute on the Host and 7s on an FPGA, then the algorithm will execute the kernel on an FPGA. After choosing an FPGA for execution, it will determine if kernel A is already on one FPGA or not. If it is already on FPGA 1, but FPGA 1’s disparity is higher than FPGA 2, then the kernel will be moved to FPGA 2 if the overhead required to reprogram FPGA 2 doesn’t make the execute time greater than the time required to execute on the Host computer (otherwise it would defeat the purpose of executing on an FPGA).

The third and fourth algorithms are Load Balancing algorithms. Load Balancing is the process of splitting the data set to be processed between the Host and an FPGA so that the run time is equal (or very close to equal) on both the Host and the FPGA. Thus, the overall runtime is minimized. Under this policy, there are two different ways of determining which FPGA to use for a particular kernel at execution time: one ensures uniform utilization at the expense of some speed and the other to guarantee speed at all costs. The method of load balancing with uniform utilization simply selects the FPGA that has the lowest utilization and executes the kernel on that FPGA. The Look Ahead method looks forward to the next two kernels to be executed and if either or both of those kernels are already in an FPGA, then the algorithm will not reprogram those FPGAs. This decreases the amount of overhead time spent reconfiguring FPGAs and maximizes the speed of the execution of the task graph.

V. RESULTS

The system used to obtain some FPGA implementation data was a Starbridge Systems HC-62 Hypercomputer. It is a high-performance, scalable, reconfigurable computer with 11 Virtex FPGAs, ten of which are user programmable. The peak disparity and the execution time of the Uniform Utilization I and II policies are shown in Figure 2 and Figure 3, respectively.

![Figure 2: Peak Disparity under Three Uniform Utilization Policies](image)

![Figure 3: Execution Time under Three Policies for Uniform Utilization](image)

The execution time of the Uniform Utilization policies were also measured with variable FPGA resources, as shown in Figure 4.

![Figure 4: Execution Time of a Task Graph with Variable FPGA Resources](image)

It should come as no surprise that Uniform Utilization code would reduce the peak disparity between the FPGAs as compared to the original, Break-Even code. It should also be
obvious that when more FPGAs are available for kernel execution, less time will be needed to execute the task graph since the FPGAs will need to be reconfigured fewer times, resulting in less overhead time.

Next, the results from the Load Balancing algorithms are shown in Figure 5 and Figure 6.

![Figure 5: Execution Time under Both Load Balancing Policies](image1)

![Figure 6: Peak Disparity for Load Balancing](image2)

Both the load balancing algorithms reduce execution time, but the load balancing with Look-Ahead can reduce the execution time by about 30% for larger task graphs, as compared to the original Break Even policy. Reducing the disparity between each FPGA can help to preserve the life of the FPGA and extend its usefulness. The load balancing with uniform utilization is the obvious choice if your objective is to keep the disparity between FPGAs low. The load balancing with look-ahead also produces lower peak disparity than the original code, but not nearly as good as the load balancing with uniform utilization.

VI. CONCLUSION

From the graphs, we can conclude, fairly decisively, that the different execution schemes can be useful for different applications. If the requirement is to reduce execution time, load balancing by Look-Ahead seems to be the best way. If a small peak disparity between FPGAs is needed to ensure the long life of a set of FPGAs, then Load Balancing by Uniform Utilization can almost completely reduce the peak disparity between FPGAs to zero.

The implementation of the JPEG compression standard on the Xilinx Spartan3E FPGAs turned out to be a very difficult problem to solve. The biggest hurdle to overcome was that we did not have the necessary and expensive software that would allow us to transfer data to and from the RAM on the FPGA development board. Without this ability, getting a raw image for processing onto the board and then taking a processed image out for inspection was impossible. Once we knew which software would allow us to do these things, we ordered the software and will begin using it to test our JPEG compression as soon as possible.

ACKNOWLEDGMENT

Timothy Davis would like to thank Dr. Muhammad Hasan for his support in the writing of this document and in performing the experiments contained herein.

REFERENCES