Scalability of a Parallel JPEG Encoder on Shared Memory Architectures

David Castells-Rufas[1], Jaume Joven[2], Jordi Carrabina[3]

CEPHIS-Universitat Autònoma de Barcelona
Edifici Enginyeria, Campus UAB, Bellaterra, Spain

[1]david.castells@uab.es
[2]jaume.joven@uab.es
[3]jordi.carrabina@uab.es

***Abstract*— Embedded multimedia systems are expected to fully embrace the future many-core wave. As a consequence parallel programming is being revamped as the only way to exploit the power of coming chips. While waiting for them we try to extrapolate some lessons learned from current multi-cores to influence future architectures and programming methods. In this paper we investigate the parallelism and scalability of a JPEG image encoder, which is a typical embedded application, on several shared memory machines using the OpenMP programming framework. We identify the Huffman coding as the bottleneck that blocks the application from scaling above a 7x factor. We propose a strategy to parallelize the Huffman coding, which introduces a small degradation in some parts of the image, allowing to reach higher speedup factors. A factor of 18.8x has been reached in SGI Altix 4700 using 22 threads. Contrasting these results with some previous works using message passing architectures we consider that the use of OpenMP on top of shared memory architectures should be reconsidered for future chips in favor of message passing architectures and programming models.**

***Keywords*— JPEG, encoder, OpenMP, performance, parallelization**

I. Introduction

JPEG encoding is present as an important function of many embedded devices such as smart phones and still cameras. In such embedded environments it is cost effective to have dedicated hardware to assist the JPEG encoding, which is a computational demanding task. However, with a new wave of many-core processors on the horizon, which are expected to land in the embedded arena, maybe it will not be cost effective to add a specific core to assist the task if the already present many-core processor can do the job.

There is no consensus about what paradigm will be used in future many-core processors. Many of the current proposals are based on the shared memory paradigm. Shared memory is perceived as a beneficial paradigm because it offers an easy programming environment that relies upon having good compilers and runtimes. However we would like to show how shared memory paradigm limits the scalability of applications, like JPEG encoding, which could be further accelerated using other more scalable paradigms like message passing or stream processing.

II. Previous Work

The high compression ratios achieved by the JPEG standard are based on the fact that only few values can be used to describe each block of an image after DCT transform and quantization processes has been applied. These few values are then Huffman coded to reduce the final number of bits that are finally stored (see Figure 1).

JPEG encoding has received much attention because of its commercial interest and has been implemented in many different platforms. There is a vast amount of literature about the different parts of the encoder and also many implementations in several different platforms. Since it is difficult to cover the entire work in the literature, we only provide some pointers to some representative contributions.

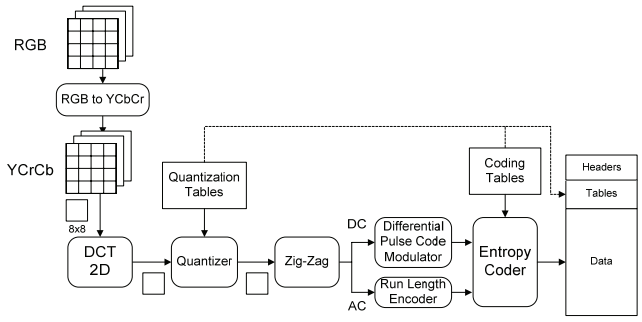

Fig. 1 Block diagram of the JPEG encoding process

To the best of our knowledge, the maximum JPEG encoding performance is obtained when custom hardware is used, usually as part of an ASIC. Nethra claims in [1] that his commercially available NI2065/66 chip offers a performance of 75MPixels/s. Kovac and Ranganathan claim in [2] that, if implemented, their Jaguar chip design would give 100MPixels/s working at 100Mhz clock frequency. ASIC designs offer a very good performance, but this comes with a sacrifice to flexibility since the chips can not be reused for other tasks.

Some more flexibility is offered by DSPs, ASIPs and FPGAs. For instance, Cast, Inc. claims in [3] that their encoder IP can encode more than 30 frames per second at a 4/3 HDTV resolution in an Altera EP3SE50 device running at 250Mhz. This is equivalent to more than 50MPixels/s. The

0190-3918/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPP.2010.58

same performance is claimed by Texas Intruments in [4] using the TMS320DM355 chip running at 216Mhz.

Hardware implementations often require using the minimal amount of memory, so they are commonly based on a pipelined design (see Figure 2). The way to achieve the maximum performance is to split the process in the greater possible number of stages and reduce the pipelining period, which is determined by the time needed by the slowest stages.

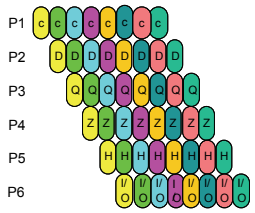

Fig. 2 Hardware pipeline

This parallelization strategy gives impressive performance and can also be implemented in software quite effectively as reported by Shee in [5] and Osorio in [6]. These works use Xtensa SL, and Ambric many-core processors respectively, which are message passing architectures in which pipeline can be easily implemented. The maximum reported performance is 31MPixels/s.

On the other hand, Shared Memory architectures are not specially addressed to pipelined processes and their natural programming frameworks, like OpenMP, encourage exploiting the parallelism at the data level. However it is shocking to notice the few number of works that have addressed the parallelization of embedded applications in this kind of architectures, and the few number of processors used in most scalability studies. For instance, Oh ([7]), Kodaka ([8]), and more recently Tumeo ([11]) have addressed the parallelization of JPEG encoding on shared memory architectures and its scalability up to just four processors.

III. Identification of the Algorithmic Bottlenecks

The sequential JPEG encoding process could be described by the following pseudo-code.

```
write headers
for each block of the image
{
 r0 = rgb2yuv(block)
 r1 = dct(r0)
 r2 = q(r1)
 r3 = zigzag(r2)
 eDC = dcEncode(r3[0], lastDC)
 write(eDC)
 lastDC = r3[0];
 rles = RLE (r3)
 eAC = Huffman(rles)
 write(eACs)
}
```

With a simple profiling of the above sequential encoder implementation we can prove, as reported by innumerable works before, that the most time-consuming functions of the encoding processes are 2D DCT transform, color conversion, quantization, and Huffman coding, respectively.

All the functions work with a small amount of memory (a block of 64 values at most) and each block is almost processed totally independently from the rest. A first problem that prevents us from treating each block independently is the data dependency between a block and the previous one, because the previous DC value is needed to compute the differential encoding of the DC coefficients.

If we split the loop in two, we can have a fist loop with the DCT and Quantization and Zig-Zag and a second loop with the remainder. But in order to do that, we need a large intermediate memory to store the results of the Zig-Zag function for each block of the image. So the modified pseudo-code would be as follows.

```
write headers
for each block of the image
{
 r0 = rgb2yuv(block)
 r1 = dct(r0)
 r2 = q(r1)
 r3[block] = zigzag(r2)
}

for each block of the image
{
 eDC = dcEncode(r3[block][0],
                r3[prev_block][0])
 write(eDC)
 rles = RLE (r3)
 eAC = Huffman(rles)
 write(eACs)
}
```

Now the first loop contains the most computing demanding functions and can be totally parallelized. But as we do that we observe the Amdahl's Law effects as the second sequential loop becomes dominant. Figure 3 depicts this effect in a hypothetical 4 processor system encoding a 8 block image. Four processors are simultaneously computing Color, DCT, Q, ZZ functions of different blocks until all blocks are computed. Afterwards Huffman codes must be serially processed. In the image different colors represent the data dependencies among functions. Each box represents a function that is executed in a processor.

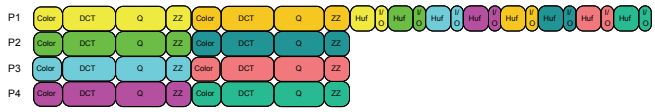

Fig. 3 First loop parallelization

Most works in the literature pay much attention to the parallelization of this first loop, resulting in designs that show poor scalability. We have implemented a first parallel version of the encoder that only parallelizes the first loop.

The scalability results are shown in Figure 4. In this simple test we have encoded the *bruno* image on several Shared Memory platforms. In section 5 more details will be given about test images and execution platforms.

As previously discussed, the scalability of the encoder is poor due to the sequential Huffman loop. The maximum achieved speedup is 7x in Altix machine which is the slowest system of the set. In faster Xeon machines the maximum speedup is below 4x.

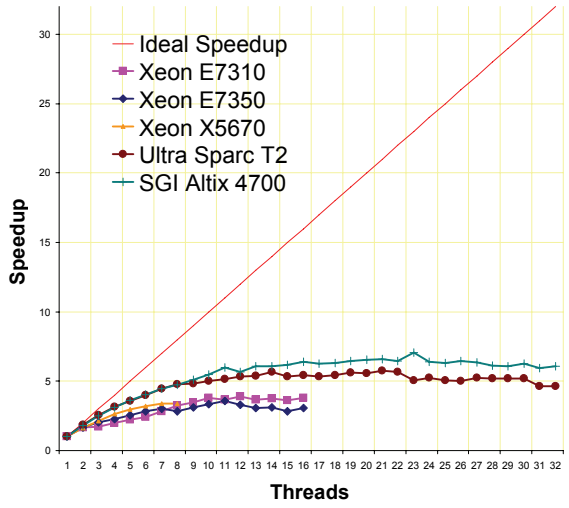

Fig. 4 Scaling of trivial parallelization of the JPEG encoding in various Shared Memory Platforms.

The parallelization of the second loop is harder because it writes the Huffman codes to the output bitstream. This is a problem because writing should be done in order and the written codes have a variable length, so it is not possible to calculate the offset for each result into the output bitstream. But even more important is the fact that resulting codes are bit aligned. So, in case we could compute the codes in parallel we would need to join them properly by a costly process of bit alignment, which is much more complex than a desirable simple memory copy.

Cook presented in [9] and [10] various strategies to perform the bit alignment and overcome this bottleneck, showing how JPEG could be effectively parallelized. Later on some extensions were added to JPEG coding ([12]) to allow the tiled coding of images, providing better error correction features and enabling the parallel encoding and decoding of the images. In order to do that, some new restart markers where introduced to identify the boundaries of the tiles and ensure that the critical information was byte aligned. These features have received the attention of the industry as shown in [13] but their use is far from universal.

IV. Proposed Parallelization

Having a variable length output from the Huffman coding is very beneficial to provide a high compression ratio but complicates the concatenation of several parallely computed bitstreams. Recall that, if the results from the Huffman coding were byte aligned, we could produce them in parallel and join them easily in a final step. Could we force a byte alignment resynchronization in certain parts of the bistream ?

We propose a way to parallelize the JPEG encoder without using JPEG extensions or the realignment techniques used by Cook. Instead, we propose to use the cause of the problem as the solution to the problem. If the variable Huffman codes create a byte misalignment, we could use an arbitrary number of "convenient" Huffman codes to realign the bitstream to byte boundaries.

Figure 5 describes this method. The top bitstream corresponds to a hypothetic JPEG image. In the figure we can see a complete block and the beginning of the following block. A block starts with the encoded value of the DC coefficient, and is followed by an undetermined number of encoded run-lengths of AC coefficients. Notice that encoded values have a variable length and are not byte aligned. The bottom bitstream is the result of applying the proposed method. When the first shown block is encoded, it is specified that realignment must be applied. So, after the DC a number of Huffman codes with odd length are written to the bitstream until byte boundaries are reached.

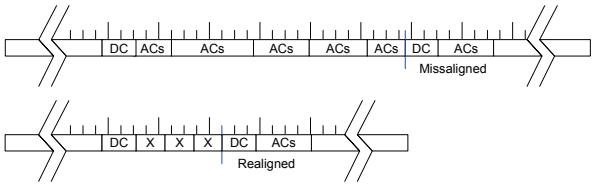

Fig. 5 Realignment to byte boundaries by replacing some Huffman codes by convenient ones

Since we parallelize the outer loop of the block iteration (the row loop) we apply the substitution at the last block of the row. This process eventually degrades the quality of the image since important AC component are removed. But, since the selected blocks contain the Cr color component information, and their associated Y and Cb components are preserved, the quality loss will not be noticeable for most large photographic images.

Fig. 6 Detail of large original images (left) and their parallely encoded results (right) for photographic images

The effect of the parallelization strategy on the image quality is shown in Figures 6 and 7. As we eliminate the high frequency coefficients from the last block of each row, a blurred band is created at the right edge of the image. Synthetic images are more affected than photographic images because they usually contain more high frequency information. Photographic images usually have less variability in the

chrominance fields so the degradation is less evident to the human eye.

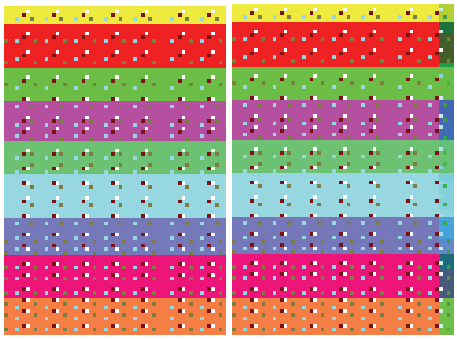

Fig. 7 Detail of large original images (left) and their parallely encoded results (right) for synthetic images

We have to modify the encoding algorithm in order to incorporate the byte realignment feature we are proposing. The second loop has to be spitted again in two loops and partial bitstreams must be created for each row. So, now the second loop writes the encoded values in the bitstream associated to each row, and when the end of the row is detected the last block is forced to realign to byte boundaries. Finally a third loop concatenates all row bitstreams to the final bitstream by simple memory copy operations.

This algorithm can be effectively parallelized because the first loop (DCT, Q, Zig-Zag) and second loop (Byte-aligned Huffman by row) have no data dependencies between iterations.

```
write headers
for each block of the image
{
 r0 = rgb2yuv(block)
 r1 = dct(r0)
 r2 = q(r1)
 r3[block] = zigzag(r2)
}

for each row of the image
 for each block of the row
 {
  eDC = dcEncode(r3[block][0],
                 r3[prev_block][0])
  write(eDC, encRow[row])

  if (block is last)
  {
    realign(encRow)
  }
  else
  {
   rles = RLE(r3)
   eAC = Huffman(rles)
   write(eACs, encRow[row])
  }
 }

for each row of the image
{
  write(encRow[row])
}
```

The second loop no longer represents a bottleneck to prevent the application scaling, and greater speedups can be achieved. The parallelization strategy is graphically depicted in Figure 8.

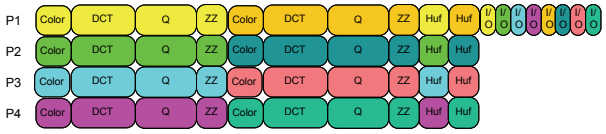

Fig. 8 Proposed parallelization strategy

V. Implementation & Results

We test the encoder performance with two big resolution images *bruno* and *women*. The *bruno* image is a 3MPixels (2000x1502) color image, the *women* image is a 16MPixels (4992x3328) color image.

The test platforms are Shared Memory architectures. Intel Xeon 7310, Intel Xeon 7350, Intel Xeon X5560, Ultra Sparc T2 (see Table 1). Although not addressed to the embedded market, these CPUs can give us an idea of the features that would offer future embedded Shared-Memory many-core processors.

TABLE I
Test Platforms

	Xeon 7310	**Xeon 7350**	**Xeon X5570**	**Ultra Sparc T2**	**SGI Altix 4700**
Processors	16	16	8	4	256
Threads x processor	1	1	1	8	1
Threads	16	16	8	32	256
Clock Freq.	1.6GHz	2.93GHz	2.93GHz	1.2GHz	1.6GHz
Cache	2x2MB	2x4MB	8MB	4MB	8MB

Figure 9 shows the time taken by different processors to compress the *women* image. The Xeon X5570 (Nehalem architecture) offers the best Single Thread performance, and when using multiple threads gets the lowest encoding times: 51ms and 275ms to encode the *bruno* and *women* images respectively. In terms of pixel performance this is almost 60 MPixels/s, which is higher than all FPGA and DSP reported implementations and very close to pure ASIC ones. But this is not a very fair comparison because hardware assisted encoders run at lower clock frequencies offering a much better power efficiency ratio.

If we just focus on performance, although ASIC and DSP implementations could try to increase their clock frequency, this is not a feasible task for current FPGA devices. On the other hand, message passing many-core architectures, like Ambric, have shown less performance but running at a significantly lower clock frequency.

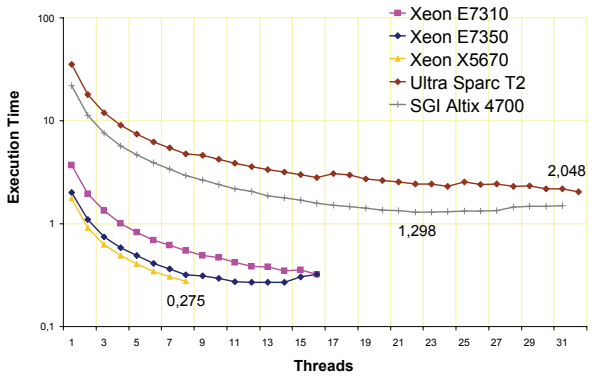

Fig. 9 Time to compress women image on different platforms for different threads (Y axis is in logarithmic scale)

We have scaled up to 32 threads to analyze the scalability of the application on the different platforms (see Figure 11). We get a maximum speedup factor of 18x in Altix machine using 22 threads. After 22 threads the Altix machine has an erratic performance and the T2 offers little speedup after adding threads.

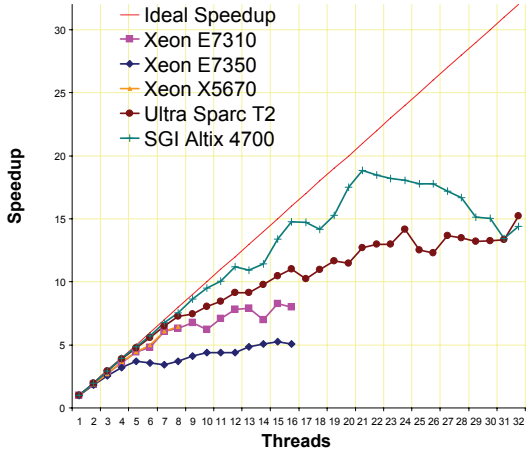

Fig. 10 Application scalability on different plaforms for the bruno image

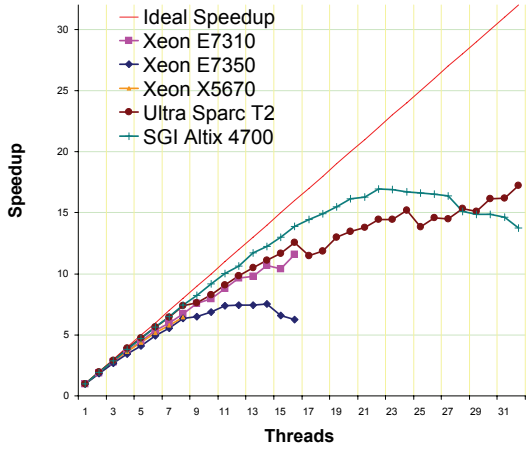

Fig. 11 Application scalability on different platforms for the women image

We cannot measure the power consumption of the tested machines to observe the energy efficiency of the different computing platforms. But what we can do is use clock cycles instead of time to normalize the disparity of frequencies of operation. Since clock frequency is one of the main drivers of dynamic power consumption this can be a feasible way to have an idea of the efficiency of the different platforms. Of course, this analysis should be taken with skepticism because frequency is not the only driver to energy consumption. In Figure 12 we can see that, although shared memory approach is feasible and gives acceptable performance, it has a much lower efficiency of all the other alternatives. Looking at results published by Osorio (in [6]) it is interesting to see their message passing implementation has a much better ratio than our developed shared memory ones.

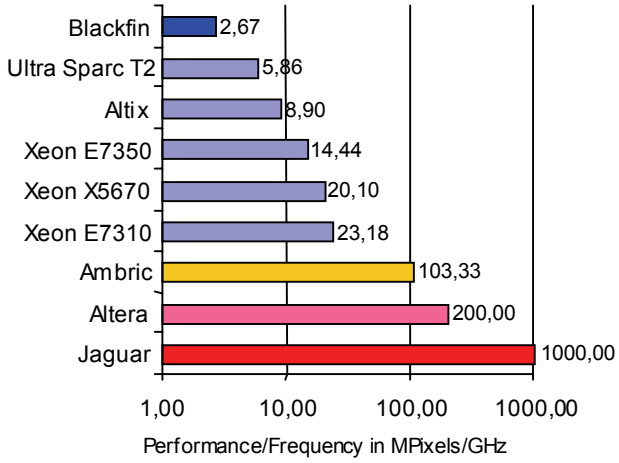

Fig. 12 Performace/Frequency ratio for different platforms in MPixels/GHz.

VI. Conclusions

We have proved that, introducing an small reduction of quality in the image, JPEG encoding can be speeded up to 20x by performing the Huffman coding on independent byte-aligned bitstreams that can be finally joined using a fast memory copy operation. This allows surpassing the performance offered by previous referenced shared memory implementations by more two fold.

However, the scalability depends on the image size and the platform used. It is important to notice that in most platforms the encoder shows a linear speedup for less than 16 threads, but after that the application scales poorly. Extrapolating the results one could see no significant benefits going above 32 processors.

Data locality is the key to good scalability, but shared memory architectures must provide a logical common global memory view. These contradictory requirements are projected to the programming frameworks. Shared Memory frameworks, like OpenMP, are embraced for the sake of programmer's ease of use, expecting to have a compilation tool-chain or runtime environment that will do a reasonable good job. But to efficiently place data into the global space some complex *pragmas* or access patterns must be performed. In fact, this is

introducing some explicit communication details and, indeed, deriving into a more complex code.

On the other hand, being free from a global memory view, message passing architectures can exploit data locality at its best at the cost of more programming effort. In addition, we should argue that this claim is sometimes gratuitous. If we take, for instance, the common pipelined design of a JPEG encoder it turns out to be more complex to implement in OpenMP than in MPI.

From the obtained results we observe that a typical embedded application like JPEG encoder do not scale above 32 threads using the shared memory paradigm, and programming frameworks like OpenMP cannot easily express convenient pipeline designs or control data locality. We advocate for adopting message passing architectures and programming methods for future embedded many-cores that allow closing the gap between them and ASICs in terms of energy efficiency.

ACKNOWLEDGMENTS

The authors want to thank Alejandro Duran, Eduard Ayguade and Barcelona Supercomputing Center for granting access to their Altix machine and their valuable support.

This work was partly supported by the European ITEA2 ParMA (Parallel programming for Multicore Architectures) Project, the Spanish Ministerio de Industria, Turismo y Comercio project TSI-020400-2009-26 and Ministerio de Ciencia y Innovacion project TEC2008-03835/TEC, the Catalan Government Grant Agency Ref. 2009SGR700.

REFERENCES

[1] Nethra Imaging, NI-2065/66. 3.2 Megapixel Smart Camera Module Image Processor with JPEG Encoder (available at http://www.nethra.us.com/pdf_files/ProdBrief_NI2065-66.pdf)

[2] Kovac, M. & Ranganathan, P. JAGUAR: a high speed VLSI chip for JPEG image compression standard VLSI Design, International Conference on, IEEE Computer Society, 1995, 0, 220

[3] http://www.altera.com/products/ip/dsp/image_video_processing/m-cas-jpeg-e.html

[4] http://focus.ti.com/docs/prod/folders/print/tms320dm355.html

[5] Shee, S.; Erdos, A. & Parameswaran, S. Heterogeneous multiprocessor implementations for JPEG:: a case study Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, 2006, 222

[6] Osorio, R. R.; Díaz-Resco, C. & Bruguera, J. D. Highly Parallel Image Processing on a Massively Parallel Processor Array XX Jornadas de Paralelismo, A Coruña, 2009

[7] Oh, J.; Kim, S. & Kim, C. OpenMP and Compilation Issues in Embedded Applications Lecture notes in computer science, Springer, 2003, 109-121

[8] Kodaka, T.; Kimura, K. & Kasahara, H. Multigrain parallel processing for jpeg encoding on a single chip multiprocessor Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02), 2002, 57

[9] Cook, G. W. & Delp, E. J. The Use of High Performance Computing in JPEG Image Compression the Twenty-Seventh Asilomar Conference on Signals, Systems, and Computers, 1993, 846-851.

[10] Cook, G. & Delp, E. An investigation of JPEG image and video compression using parallel processing IEEE International Conference on Acoustics, Speech and Signal Processing, 1994, 5

[11] Tumeo, A.; Monchiero, M.; Palermo, G.; Ferrandi, F. & Sciuto, D. A design kit for a fully working shared memory multiprocessor on FPGA Proceedings of the 17th ACM Great Lakes symposium on VLSI, 2007, 222

[12] ITU-T Recommendation T.84 | ISO/IEC 10918-3:1996, Information Technology Digital Compression and Coding of Continuous-Tone Still Images: Extensions

[13] Moussavi, F.; Lin, S.; Kopet, T. & Jabbi, A. Method and apparatus for parallelization of image compression encoders, US Patent App. 11/730,718, 2007.