Hardware Acceleration of the JPEG2000 Kakadu Library

Michael Dyer, Amit Gupta, Natalia Galin and Saeid Nooshabadi
School of Electrical Engineering and Telecommunications
The University of New South Wales, Sydney, Australia

***Abstract*— The NIOS II soft core processor from Altera allows for the easy interfacing of new peripheral blocks to existing software. Using the Kakadu software implementation of JPEG2000, we have added dedicated hardware block encoders to produce an accelerated implementation. Simulation demonstrates that our simple architecture can provide a speedup of 2.55 times, compared to a pure software implementation. The use of both hardware and software allows for an implementation of the entire JPEG2000 standard. We also identify future options for further hardware acceleration of the Kakadu library.**

I. Introduction

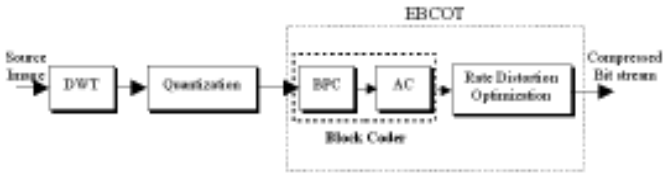


Fig. 1: Block Diagram of JPEG2000 Compression Scheme

JPEG2000 [1] is a relatively recent image compression scheme. The heart of the compression scheme is the algorithm known as Embedded Block Coding with Optimal Truncation [2]. EBCOT is responsible for modelling and entropy coding independent blocks of quantized Discrete Wavelet Transform (DWT) coefficients. This algorithm is made up of two fundamental blocks, namely the block coder (termed Tier-I) and the rate distortion optimisation system (Tier-II). The block coder in turn is built from the bit-plane coder and the MQ variant of the arithmetic coder. The bit-plane coder scans through a block of coefficients in a bit-plane oriented fashion, producing a context for each bit. This context is used to drive the probability model of the MQ coder. A block diagram of the scheme can be found in Figure 1.

Due to its bit-based operation, software implementations of the block coding algorithm are less than ideal. Previous work by our group [3], [4] has developed efficient hardware implementations of both the bit-plane coder and the MQ coder. These blocks have been combined to produce a hardware block coder.

The majority of research into JPEG2000 hardware has concentrated on the efficient implementation of individual processing units shown in Figure 1. Little has been done on combining these improved systems to produce an efficient system capable of implementing the entire JPEG2000 standard. Existing hardware implementations of the standard such as [5], [6] use limited tile sizes and choice of transform in order to reduce implementation complexity. Our approach to this is to use existing software libraries and interface them to efficient implementations of the processing units. This codesign approach allows for the partitioning of the compression scheme into operations that are more suited to either hardware or software. In particular, operations such as the Discrete Wavelet Transform and EBCOT, which are difficult to implement in software can be moved into dedicated hardware. This also allows for the parallel implementation of these tasks, providing even greater speed. Things more suited to software, such as packet formation and file formatting, can be left in software and run on a simple processor. This approach has the major advantage of being able to implement the entire JPEG2000 standard, without limitations. Of course, this does require more resources in terms of memory.

This work uses a port of the Kakadu software implementation of JPEG2000 (provided with [7]) on the Altera NIOS II processor. This implementation is profiled to determine which of the compression operations consumes the most processor time. The major operation, namely block coding is implemented in hardware using our previous work and interfaced to Kakadu. This is achieved by integrating the hardware block coder into the NIOS II processor as a custom peripheral. We are able to demonstrate the tremendous improvement that a dedicated hardware block coder can make to the implementation of JPEG2000. This then allows us to identify future candidates for hardware optimization.

II. Software Profiling

The Kakadu library was ported to the NIOS II processor and used to compile a simple compression application, using full optimization. The 8 bit gray scale ISO test image 'Cafe', with an image size of 2560×2048 pixels was used to test the software. The truncated results of the GPROF profiling tool are shown in Figure 2.

```
Flat profile:
 Each sample counts as 0.001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 28.80     48.13    48.13     1286     0.04     0.08  encode()
 17.23     76.92    28.79     9420     0.00     0.00  encode_cleanup_pass()
 15.62    103.02    26.10     8134     0.00     0.00  encode_mag_ref_pass()
  3.89    109.52     6.51     4960     0.00     0.00  perform_vertical_lifting_step()
  3.54    115.44     5.92                             __floatsidf()
  3.48    121.25     5.82     2560     0.00     0.00  transfer_bytes()
  3.12    126.47     5.22                             __pack_d()
  2.90    131.32     4.85     4960     0.00     0.02  horizontal_analysis()
  2.89    136.16     4.83      122     0.04     0.92  encode_row_of_blocks()
  2.69    140.64     4.49                             __unpack_d()
  2.47    144.77     4.13     4960     0.00     0.00  push(kdu_line_buf&)
   ...
  0.00    167.08     0.00        1     0.00   133.29  main()
```

Fig. 2: GPROF profile of a pure software compressor

According to the profile, compression takes 167.08 sec, of which block coding accounts for 103.02 sec. This accounts for 61.65% of the total processing time. The function "encode()" is called for each block, thus there are 1286 blocks coded by the compressor for this image. The functions "encode_cleanup_pass()" and "encode_mag_ref_pass()" are called in turn by the "encode()" function. Readers familiar with Kakadu may notice the absence of "encode_sig_prop_pass()", as this was inlined by the compiler into the "encode()" function. It is obvious from this profile that our optimizations effort is well placed.

The functions "perform_vertical_lifting_step()" and "horizontal_analysis()" are also possible candidates for optimization. These functions are used to perform the DWT on the image. These account for 11.36 sec of processor time.

III. Existing Hardware

A. Bit Plane Coder

The bit plane coder developed by our group [4] is based on the 2 state memory system developed in [8]. This block coder, unlike its predecessors, processes an entire stripe column per clock cycle. This means that up to 10 context data pairs may be produced in a single cycle. This coder is also generic - it can process a block in either normal or causal mode [1]. Additionally, due to its concurrent column processing, it can produce on average 1.1 context data pairs per cycle, compared with 0.7 context data pairs of the best alternative architecture.

In order to couple this system to the arithmetic coder, a special context buffer was developed [9]. This buffer is used to prevent excessive stalling of the bit plane coder while the arithmetic coder processes bursts of context data pairs.

B. Arithmetic Coder

In order to enable the use of a bit plane coder that produces more than one context data pair per clock cycle, new arithmetic coder architectures were required that absorb this high symbol rate. Because of the sequential nature of the MQ coder algorithm, producing hardware MQ coders that can consume one or more context data pairs per clock cycle presents a significant challenge.

Our group has produced multiple architectures each capable of consuming more than one context data pair per clock cycle [10]. These coder architectures present a variety of design choices. In particular, the use of a brute force coding technique with simplified byte emission, allows for the coding of 2 symbols per clock cycle, whilst maintaining a comparable critical path to that of the block coder.

C. DWT

Two possible optimization methods have been identified. Firstly, the lifting step itself (Figure 3) may be implemented as a custom ALU instruction in the DWT. The step must be performed four times for each pair of input samples. The lifting step itself consumes approximately 5 assembly instructions, a combination of multiply, shift and accumulate. A custom instruction should be able to decrease this to one or two instructions. Alternatively, the entire DWT process could be moved into a custom peripheral. Our group has implemented a custom ALU "step" instruction for the NIOS II processor, and is currently completing a complete DWT engine.

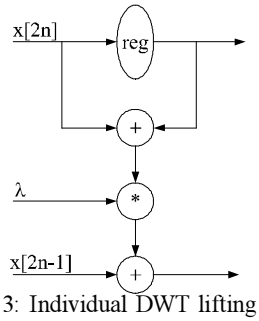


Fig. 3: Individual DWT lifting step

IV. Proposed Architecture

The Altera NIOS II soft core processor [11] is an ideal platform for interfacing new hardware to existing software. The processor is synthesizable onto the Stratix range of FPGAs, allowing for 'system-on-programmable-chip' (SOPC) designs. The NIOS processor has two separate bus masters, one for instructions and one for data. The processor uses Avalon Bus Specification [11] for implementing the system bus. This bus architecture uses a switching fabric, allowing multiple masters to use the same bus simultaneously, as long as they are not accessing the same physical device. It also has the added benefit of allowing peripherals to operate at a different clock frequency to the bus master.

Figure 5 shows the proposed NIOS II SOPC configuration. The development kit used to test this architecture was fitted with 16MB of SDRAM, and 2MB of SRAM. Individual hardware block coders (BCs) are connected to the bus directly and via DMA controllers. The direct connection is used to transfer control and status information, as well as compressed data. The DMA access is used to transfer code block coefficients into the local code block RAM of each BC (see Figure 6).

In our proposed architecture, the SDRAM is used to hold code and data (the 'cafe' image used to test the system was embedded as a C array and compiled into the program). The 2MB SRAM was used as special storage for codeblock objects. Kakadu is a single threaded library, and only requests block coding when a row of blocks is ready for processing. Thus we can be assured that while Kakadu is waiting for the block coders to complete processing, the NIOS processor will only be accessing the SDRAM. Thus, the DMA controllers will have free access to the SRAM, allowing for uninterrupted burst transfers of block coefficients into the block coder peripheral.

The block coder developed in [4] has certain data requirements the demand special consideration. As mentioned, block coding is a bit-based operation. The block coder processes a code block in a bit plane fashion. First, the MSBs of each coefficient are coded, followed by the next most significant bit

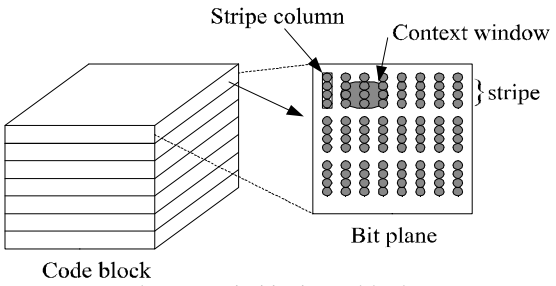


Fig. 4: Code block partitioning

and so on. Within each bit plane, the codeblock is divided into stripes, and these stripes further divided into stripe columns of 4 bits (see Figure 4). Each stripe is processed in a raster scan fashion. The bit plane is also divided in to three passes, namely the Significance Propagation, Magnitude Refinement and Clean Up passes. The bit plane coder produces a context for each bit, based on its own state and the state of its 8 immediate neighbors. This context, along with the bit value is passed to the MQ Coder for entropy coding. Each bit may belong to only one pass. For efficient operation, our bit plane coder requires that a stripe column be provided during each clock cycle. The code block coefficients must then be reorganized into stripe columns in order to be processed by the bit plane coder.

Each code block coefficient is 16 bits wide, however to better utilize the bus, the code block ram uses a 32 bit data bus. The "reorg" unit of the block coder performs the required reorganization of code block coefficients into bit plane columns. The coefficients are stored in row-major fashion in memory. Thus a single 32 bit read will yield two coefficients from neighboring stripe columns. From these two coefficients two bits are produced, corresponding to the current bit plane. These two bits are written into two stripe column registers, in the correct position. When the column registers are full, they are written into the FIFO, ready for processing by the bit plane coder. In this way, we can produce two stripe columns every four clock cycles. However, as the bit plane coder requires a column every cycle, the "reorg" unit must be run at least twice the clock rate of the bit plane coder.

Figure **??** shows the custom instruction logic for implementing the DWT step. The instruction format provides for two 32 bit operands (A and B) and one 32 bit result (R). This is not the complete implementation as shown in Figure 3. We have developed this simpler step based on the original instructions actually performed by Kakadu. The custom instruction performs the first add and multiply of Figure 3 and then truncates fixed point result.

V. Results

The proposed system was synthesized utilizing a single dedicated block coder. Tables I and II show the synthesis results for the block coder and NIOS processor respectively. As the reorganization system must run at least 2 times faster than the bit plane coder, for the following analysis we assume that the reorganization unit is clocked at over 90 MHz, and that the bit plane coder is clocked at 45 MHz.

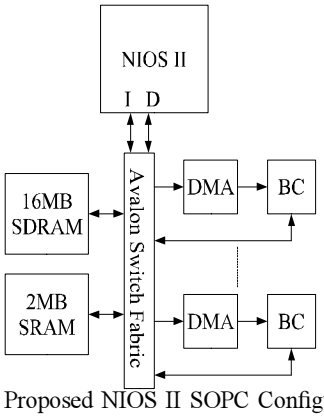


Fig. 5: Proposed NIOS II SOPC Configuration

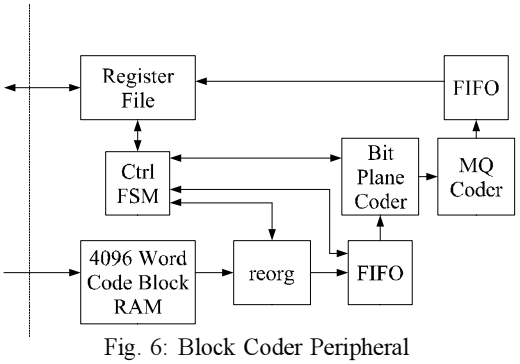
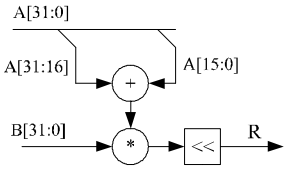


Fig. 6: Block Coder Peripheral

The bit plane coder will take, on average, $CYC_{BC} = 46080$ cycles to process a code block. In addition to this, each code block contains $64 \times 64 = 4096$ 16 bit samples. These samples must be transferred across the avalon bus. As stated in Section IV, we can safely assume that the bus will be free for this transfer. This means the number of cycles taken to transfer the samples, using a 32 bit bus is $CYC_{DMA} = \frac{4096}{2} = 2048$ cycles. If F_{BC} and F_{DMA} are the block coder and the avalon bus clock frequencies, the time taken to process a code block is then

$$\begin{aligned} T_{BC} &= \frac{CYC_{BC}}{F_{BC}} + \frac{CYC_{DMA}}{F_{DMA}} \\ &= \frac{46.08 \times 10^3}{45 \times 10^6} + \frac{2048}{50 \times 10^6} \\ &= 1.06 \times 10^{-3}\,\text{sec} \end{aligned}$$

We can apply these results to the software profile in Section II, where we saw that the Cafe test image requires 1286 code blocks to be processed. The total time required to code the code blocks from Cafe is then $T_{BC,Cafe} = 1286 \times 1.06 \times 10^{-3}\,\text{sec} = 1.37\,\text{sec}$. The total time now taken to process the image is $T_{Cafe} = 167.08 - 103.02 + 1.37 = 65.43\,\text{sec}$. The estimated speed up of the compression scheme for this image



is then $S = \frac{167.08}{68.22} = 2.55$ times.

Comparing the assembly language generated by the compiler for the DWT analysis stage of Kakadu, we note that the number of instructions inside the lifting loop has decreased from 5 to 2. This corresponds to a 40% reduction in the number of instructions inside the lifting analysis loop.

TABLE I: Block Coder Synthesis Results

logic elements	3998
reorg clk	99.91MHz
bit plane clk	51.22MHz
memory bits	90528

TABLE II: NIOS Processor Synthesis Results

logic elements	4919
clk	50MHz
memory bits	71808

VI. Further Work

This section discusses some of the speed up techniques that can improve the systems performance. The techniques include multithreading of Kakadu, implementation of the DWT engine as a hardware unit, and multiple block coders.

A. Multithreading Kakadu

Work on multithreading the Kakadu JPEG2000 library is progressing. With the addition of threads, the Kakadu will be able to support multiple block coders and DWT engines operating concurrently. The single threaded Kakadu library must wait for a partial DWT to complete before beginning block coding, and then must wait for block coding to complete before resuming the DWT. With multiple threads and dedicated hardware, these two tasks can be performed simultaneously. In this way, the only tasks the NIOS processor need perform would be sequencing and control and file creation.

B. Parallel Block Coders

This work has depended on average values for the determination of the possible speed up time. Further work is investigating the true statistics of block coding time, such that a more accurate model of system performance may be obtained. As more block coders are added, bus throughput becomes more and more significant, and will eventual prove to bottleneck the block coding speed and hence limit the number of block coders that can be supported by a single processor. The bottleneck occurs when the time taken to load all code block RAM in each of the available block coders exceeds the block coding time of the first loaded block coder. For N block coders, this occurs when $CYC_{BC} < NCYC_{DMA}$. Using the results of our proposed architecture, we predict that the maximum number of block coders is $N < \left\lfloor \frac{46080}{2048} \right\rfloor = 22$. Using this number of block coders, block coding time would be reduced to $T_{BC,Cafe} = \frac{1}{22} \times 1286 \times 1.06 \times 10^{-3}\,\text{sec} = 0.06\,\text{sec}$. This is an extremely simple estimation of the number of block coders that can be supported and takes no account of the statistical variation of block coding time. The investigation of the true statistics of block coding time will provide a more accurate analysis. Additionally, the amount of speed up obtained by parallel block coders will be relatively insignificant to that achieved from moving from a pure software implementation to an implementation using a single hardware block coder. The improvement however, will prove beneficial when other parts of the compression scheme are moved into hardware, and when Kakadu supports multithreading.

C. DWT

It may be possible to more tightly couple the DWT to the block coder peripherals, removing some of the NIOS II processor overhead. Tighter coupling may make better use of parallel block coders, as it will remove much of the redundant memory overhead used in copying DWT coefficients. It will also allow for the parallel operation of these two units.

VII. Conclusion

We have performed a profile of the Kakadu library when run on the soft core embedded processor NIOS II. From this profile we have identified those operations requiring hardware acceleration. We have demonstrated that the use of dedicated hardware block coders can be used to substantially increase the speed of the Kakadu JPEG2000 software library, by a factor of 2.55 times. We have outlined future work that will further improve the speed of the Kakadu library running on an embedded processor.

References

[1] "ISO/IEC international standard 15444-1 JPEG2000 image coding system."

[2] D. Taubman, E. Ordentlich, M. Weinberger, G. Seroussi, I. Ueno, and F. Ono, "Embedded block coding in JPEG2000," in *Proc. International Conference on Image Processing (ICIP 02)*, vol. 2, September 2000, pp. 33–36.

[3] M. Dyer, D. Taubman, and S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000," in *Proc. Internation Conference on Image Processing (ICIP '04)*, 2004.

[4] A. K. Gupta, S. Nooshabadi, and D. Taubman, "Concurrent symbol processing capable VLSI architecture for bit plane coder of JPEG2000," *IEICE Transactions on Information and Systems, Special Section on Recent Advances in Circuits and Systems*, vol. E88-D, pp. 1878–1884, 2005.

[5] K. Andra, C. Chakrabarti, and T. Acharya, "A high-performance JPEG2000 architecture," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, pp. 209 – 218, 2003.

[6] H.-C. Fang, C.-T. Huang, Y.-W. Chang, T.-C. Wang, P.-C. Tseng, C.-J. Lian, and L.-G. Chen, "81MS/s JPEG2000 single-chip encoder with rate-distortion optimization," in *Proc. IEEE International Solid-State Circuits Conference*, 2004.

[7] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[8] Y.-T. Hsiao, H.-D. Lin, K.-B. Lee, and C.-W. Jen, "High-speed memory-saving architecture for the embedded block coding in JPEG2000," in *IEEE International Symposium on Circuits and Systems*, vol. 5, May 2002, pp. V–133 – V–136.

[9] A. K. Gupta, S. Nooshabadi, and D. Taubman, "Efficient VLSI architecture for buffer used in EBCOT of JPEG2000 encoder," in *Proc. IEEE International Conference of Circuits and Systems (ISCASŠ05)*, 2005.

[10] M. Dyer, D. Taubman, S. Nooshabadi, and A. Gupta, "Concurrency techniques for arithmetic coding in JPEG2000," 2005, unpublished, submitted to IEEE Transactions on Circuits and Systems I.

[11] Altera Corporation, "Nios II processor," *http://www.altera.com/products/ip/processors/nios2/ni2-index.html*.