Low-Complexity Hyperspectral Image Compression on a Multi-tiled Architecture

Karel H.G. Walters, André B.J. Kokkeler, Gerard J.M. Smit
University of Twente
Department of Electrical Engineering, Mathematics and Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
k.h.g.walters@utwente.nl

***Abstract*—The increasing amount of data produced in satellites poses a downlink communication problem due to the limited data rate of the downlink. This bottleneck is solved by introducing more and more processing power on-board to compress data to a satisfiable rate. Currently, this processing power is often provided by custom off the shelf hardware which is needed to run the complex image compression standards. The increase in required processing power often increases the energy required to power the hardware. This in turn pushes algorithm developers to develop lower complexity algorithms which are able to compress the data for the least amount of processing per data element. On the other hand hardware developers are pushed to develop flexible hardware which can be used on multiple missions to cut development cost and can be re-used for different missions.**

This paper introduces an algorithm which has been developed to compress hyperspectral images at low complexity and describes its mapping to a new hardware platform which has been developed to offer flexibility as well as high performance processing power called the Xentium tile processor.

I. Introduction

The improvements in hyperspectral sensors make it more difficult to achieve the compression needed to download all the data from the orbiting platform. Although the processing power of the hardware on-board has increased, so has its power consumption. The constraints on processing power consumption have in turn pushed scientists to develop less complex algorithms to process this increasing amount of data.

The algorithm described in this paper has been developed by Abrardo et. al. [1] to have a lossless algorithm which combines a good compression ratio with low complexity. The algorithm has been proposed to the CCSDS hyperspectral data compression group for standardization. Abrardo et. al proposed the algorithm but without any implementation other than a C program run on an standard Linux desktop PC. This leads to the question whether this algorithm is indeed such a good choice to be run on any DSP architecture or alike.

Not only are the scientists pushed to develop better algorithms, the hardware engineers are pushed to develop more efficient and flexible processing platforms. The hardware platform needs to be able to adapt to changes in the algorithm during development. Moreover, if a different algorithm would improve the overall platform performance it would be a welcome feature when this could be implemented even after launch.

A new tile processor called the Xentium is under development at Recore Systems [2] which will become available as an IP in the future. This highly parallel processing platform is designed to be part of a system-on-chip in which several of these tiles will be connected with a network-on-chip. This allows to distribute programs over different tiles and employ more tiles when needed. Tiles which are not used can be turned off to save power. Another benefit of this tiled approach is that if a single tile fails the tasks can easily be mapped on a different still available tile.

This paper investigates the possibilities to map the hyperspectral image compression algorithm to the Xentium platform. Although the platform's main development drive is not related to space affiliated applications it is meant to be an energy efficient highly parallel platform. Section II explains the algorithm after which section III will provide a more detailed description of the Xentium platform. Section IV will show how different parts of the algorithm can be mapped to the Xentium and section V will show the obtained results. This paper concludes in section VI after which the future work is discussed in section VII.

II. Compression algorithm

The following compression algorithm has been developed by the universities of Siena and Torino in Italy together with Carlo Gavazzi Space and the On-Board Payload Data Processing section of ESA's ESTEC. This section will give a short overview of the encoding part of the algorithm. The decoding is forseen to be done as part of the ground segment and as such not on the platform proposed here. The algorithm has been described in detail in a paper by Abrardo et. al. [1].

The main purpose of the algorithm is to compress hyperspectral image data lossless at a competitive rate and with low complexity. A hyperspectral image is an image in which the same scene is observed in different frequency bands. Because it is the same scene, the frequency bands show a high correlation between each of them. The algorithm is based upon the ideas of distributed source coding (DSC) [3] and has been proposed to the CCSDS for standardization. Distributed source coding considers a situation in which two or more statistically dependent data sources must be encoded by separate encoders that are not allowed to communicate with each other. Performing separate lossless compression may

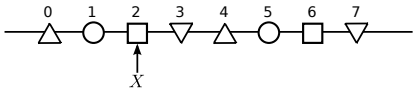

Fig. 1. DSC example

seem less efficient than joint encoding. However, distributed source coding theory proves that, separate encoding is optimal, provided that the sources are decoded jointly and are correlated [4, 5].

The principle of DSC can be explained by the following example: Figure 1 depicts 8 different pixel values of different frequency bands of a hyperspectral image. We choose to encode the value of the frequency band indicated by X. In the case there would not be any side information available, the value of X would cost 3 bits to encode. Now we introduce the different cosets, bins, in which the value can reside. In figure 1 they are indicated by triangle point upwards, circle, square and triangle point downwards. Encoding only this coset requires 2 bits and by using the previous decoded band as side information the actual value can be retrieved. The value of X without side information can be decoded as either 2 (010) or 6 (110) as the encoded value of the coset in which X resides is 10 in binary. When the value in the previous band, Y, is 3, X will be decoded as 2 (010) because the requirement is that the closest coset is chosen in the decoding stage. The probability of error, i.e that Y is closer to another element of the coset other than the true value of X, is really small. The number of cosets depends on the amount of correlation between the different bands, more correlation results in less cosets and vice versa [1].

The encoding part of the algorithm is as follows. Let $x_{m,n,i}$ denote the pixel of an hyperspectral image X in the $m-th$ line, $n-th$ column and $i-th$ band. Each band is partitioned into blocks of size 16×16 and each pixel $x_{m,n,i}$ can be recovered from the pixel of the previous band $x_{m,n,i-1}$ with a linear model

$$x_{m,n,i} = f(x_{m,n,i-1}) + v_{m,n} \tag{1}$$

where $v_{m,n}$ is a noise process determining the correlation. So $f(x)$ is a linear function from which the current band can be determined with the previous band as input. The value of pixels in X is coded by retaining only the least significant bits (LSBs) of $x_{m,n,i}$, where the selected number of LSBs depends on the amount of correlation between the current and the previous band. The number of LSBs is selected in such a way as to guarantee that the decoder will be able to exactly reconstruct X.

To compress $x_{m,n,i}$ the following steps are applied.

1) In order to make the current 16×16 block, $x_{m,n,i}$, as similar as possible to $x_{m,n,i-1}$ in a Minimum Mean-Squared Error sense, a Least-Squared estimator is computed as follows

$$\alpha_{[-1]} = \frac{\sum_{m,n}(x_{m,n,i-1} - \mu_{i-1}) \cdot (x_{m,n,i} - \mu_i)}{\sum_{m,n}(x_{m,n,i-1} - \mu_{i-1})^2} \tag{2}$$

where μ_i and μ_{i-1} are the average value of the corresponding blocks of bands i and $i-1$.

2) A quantized version of $\alpha_{[-1]}$ is generated with a uniform scalar quantizer with 256 levels starting from 0 to 1.99. The predicted values within the block will be

$$\tilde{x}_{m,n,i} = \mu_i + \hat{\alpha}_{[-1]}[x_{m,n,i-1} - \mu_{i-1}] \tag{3}$$

3) The error vector is calculated in each block as

$$e^{[-1]}_{m,n} = x_{m,n,i} - \tilde{x}_{m,n,i} \tag{4}$$

and the maximum error is used to set the number of LSBs to be retained as

$$k_{[-1]} = \left\lfloor \log_2(\|e^{[-1]}_{m,n}\|_\infty) \right\rfloor + 2 \tag{5}$$

Where number of bits to encode the absolute maximum of $e^{[-1]}_{m,n}$ is determined.

4) The $k_{[-1]}-1$ LSBs are transmitted and for the $k_{[-1]}$ least significant bit-plane the following map is computed

$$M_{[-1]} = \begin{cases} 0 & |e^{[-1]}_{m,n}| < 2^{k_{[-1]}-2} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

where the maximum absolute error in the block is $2^{k_{[-1]}-1}$. This results in a 16×16 matrix of 0s and 1s. The map $M_{[-1]}$, as well as the $k_{[-1]}$-th LSB of those pixels for which $M_{[-1]} = 1$, need to be written in the compressed file. Most samples of $M_{[-1]}$ are zero. This means that $M_{[-1]}$ is a highly compressible signal. Only the positions of the bits where $M_{[-1]}$ equals to one are coded using a differential Huffman encoder. The Huffman code table is pre-defined and built based on simulation data.

5) For each block the compressed bitstream also contains the $k_{[-1]}$ value as well as the value of $\hat{\alpha}_{[-1]}$ and μ_i.

The algorithm also has some provisions for error resilience. The error resilience in the algorithm is achieved by using not only $x_{m,n,i-1}$ as side information to create the map but also $x_{m,n,i-2}$ hereby if one band is garbled or lost during transmission there is still a high probability that the information can be recovered. Since this only affects the number of operations and not so much the mapping possibility, it has been discarded in our implementation.

III. Xentium tile processor

The Xentium tile processor is currently under development by Recore Systems. The Xentium is a programmable digital signal processing tile that is being designed for high performance computing. The Xentium datapath has by default a width of 32-bits, but is customizable at design time. In this paper we assume a default un-customized Xentium. A default implementation of the Xentium design is depicted in figure 2.

DSP operations are performed on different processing units. One operation can be issued on each unit in each clock cycle. All operations require a single clock cycle (with the exception for load operations which require two cycles). Multiple units can be used in parallel to perform operations used in DSP

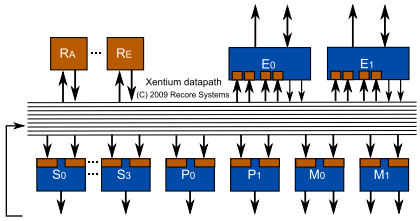

Fig. 2. Xentium core

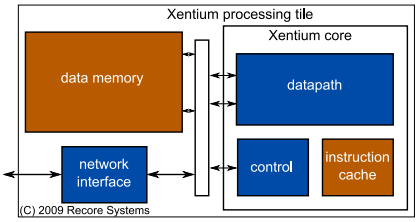

Fig. 3. Xentium together with memory and NoC interface

kernels such as one complex multiplication per clock cycle, four multiply accumulates (MACs) per clock cycle or a radix-2 FFT butterfly per clock cycle.

The processing units in the Xentium datapath can operate in two different modes; 32-bit word or vector mode. In vector mode, the unit operands are interpreted as 2-element vectors. The elements of these vectors are the low and high half-word parts of a word. Vector operations perform the same operation on the low and high parts of the vectors. Moreover, the datapath is equipped with dedicated 40-bit accumulators for improved accuracy.

The default Xentium design has two M units, four S units, two P units and two E units. The M units can perform multiply operations. The S units can perform ALU (addition, subtraction) and shift operations. The P units can perform ALU, compare and pack operations. The E units can perform load and store operations. Each operation can be executed conditionally and a 16-bit immediate value can be selected as a second operand on the M, S and P units.

A preliminary synthesis of the design has been performed in 90nm CMOS technology. The estimated area of the design (excl. SRAM cells) is about 0.3mm^2 (100k equivalent gates, excl. embedded SRAM). The clock frequency of the processing tile is estimated over 200Mhz. Those preliminary results are before place and route. Hence, area and size should be taken as lowerbound.

The Xentium is developed to be part of a SoC like depicted as in figure 4 in which TP denotes the different tile processors. Each of the tile processors is connected to a router. The routers in their turn are connected to a network-on-chip. The network-on-chip can be connected to a more conventional bus architecture like AMBA to communicate with other peripherals.

Currently, program development for the Xentium is done in C with intrinsics or directly in assembly. As a development environment, the Eclipse IDE is used together with a Xentium

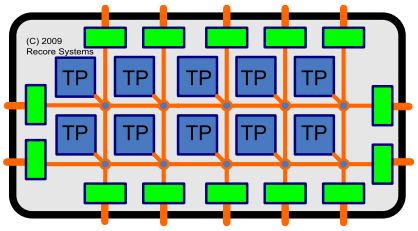

Fig. 4. Xentium tiles connected via a network-on-chip

C library with which a functionally correct implementation can be made. A cycle accurate DSP kernel implementation for execution on the Xentium processing tile currently requires Xentium assembly coding effort.

IV. Mapping

The algorithm lends itself well for data parallelism since all the bands can be divided into blocks of 16x16 pixels. Therefore a mapping has been made in which the whole process of a single block has been implemented on a single processing tile. This way more throughput is easily obtained by employing more processing tiles. In order to verify the functional correctness of the algorithm, a reference implementation was made in Matlab. As input for the algorithm a 50 band, 16 bits per pixel image from the AVIRIS instrument was used.

Different parts of the algorithm will be discussed now and how these are mapped to the Xentium platform

A. Average

The average value of each block, μ, is one of least complex operations.

$$\mu_i = \frac{\sum_{m,n}(x_{m,n,i})}{16 \times 16} \tag{7}$$

The values are loaded into the Xentium and added after which the result is shifted to the right by 8 positions. Before shifting 128 is added to the result to make sure that the result is properly rounded.

B. Least-squared estimator

The least-squared estimator, α, in a straight forward manner consists of $5 * 256$ additions and subtractions, and 512 multiplications followed by a single division. An easy optimization can be made by using the difference between the mean and the pixel value in the previous band,$(x_{m,n,i-1} - \mu_{i-1})$, twice for both the denominator as well as the numerator. This will save 256 subtractions. Calculating the numerator and denominator separately does not cause many problems on the Xentium other than the possibility of a saturation. This risk is minimal since the input values are 16-bit, the Xentium has 32 bit adders, and there are only 256 additions performed.

The division is a bit more challenging. The Xentium, like many DSPs, does not offer any hardware division possibilities. It is possible to do multiplication on the Xentium, therefore we choose to do an approximation. The basic idea is to multiply with the inverse and calculating the inverse with the help of the Newton-Rhapson approach [6], which is depicted in equation

8. In equation 8 x_{k+1} relates to the increasing accuracy of $\frac{1}{\sum_{m,n}(x_{m,n,i-1}-\mu_{i-1})^2}$ of equation 2, while a in equation 8 is the value of $\sum_{m,n}(x_{m,n,i-1}-\mu_{i-1})^2$.

The main benefit is now that the division has been transformed in multiplications and subtractions only.

$$\begin{aligned} x_{k+1} &:= x_k - \frac{f(x_k)}{f'(x_k)} \\ &:= x_k - \frac{(1/x_k - a)}{(-1/x_k^2)} = x_k + (x_k - ax_k^2) \\ &:= x_k(2 - ax_k) \end{aligned} \quad (8)$$

The default Xentium design offers a 16×16 multiplier with which the approximation can be done. It takes 4 iterations to calculate a good enough approximation. Since it is known in what range α should be, the starting value, x_k, can be determined at design time. The input however does need to be scaled down since the numerator and denominator are larger than 16 bits. This is not so much of a problem for this specific implementation since the result, α, will be quantized. The 256 different values α relate to the 8 most significant bits of α this means that the effect of less accurate approximation is discarded since the 8 least significant bits of α have little effect on the final result.

C. LSBs

The error vector and following calculation to determine the maximum number of LSBs is pretty straight forward. The Xentium has a `max` instruction with which it is easy to determine the maximum error by means of a simple `for` loop. The log_2 of this value is determined with the use of the `exp` instruction which returns the number of sign bits. For example the binary representation of the decimal value 3 on an 8 bit machine would be 00000011. The `exp` function will return 6 in this case. This together with a subtraction determines the number of LSBs, $k-1$, that need to be retained for each pixel value in the current block.

D. Bit packing

One of the more difficult operations is to pack the $k-1$ LSBs in a 32-bit word. In hardware this is rather simple since a simple shift register would suffice and the same holds for a higher level language of which an implementation is shown in listing 1. The bits currently available in the word are represented by `bits_a` and `k` represents the number of bits that need to be retained for each value. The reason this proved to be a difficult operation on the Xentium is the conditional store on line 10 and because of the relation between the output value within the conditional part and the value outside the conditional part, lines 12 and 14. An implementation has been made but it took quite some time to get the schedule right and make optimal use of the hardware available.

```
1  int bits_a = 32;
2
3  for(int i=0; i<256; i++)
4  {
5          bits_a = k;
6          value = test_data[i] & ((1<<k)-1);
7
8          if(bits_a<0)
9          {
10                 output |= (value >>abs(bits_a));
11                 output = 0;
12                 bits_a = 32-abs(bits_a);
13         }
14         output |= (value << bits_a);
15 }
```

Listing 1. Shift c++ pseudocode

E. Map

To create the encoded map, several operations are combined. The indexes of the values which are larger than the predefined maximum error are determined first. From those specific values the k-th bit is also stored. The stored bits are packed together in pretty much the same way as described in section IV-D. The indexes gathered from the previous operation are subtracted from eachother so their respective difference remains. This is to get the values needed for the differential Huffman encoder. A small example of this is illustrated in figure 5, in which part of an image block of 16×16 is shown. All of the values in the block are used as input for equation 6. In this example the values on indexes 1 and 18 cause a 1 in the map afterwhich their respective difference is used as input for the Huffman encoder.

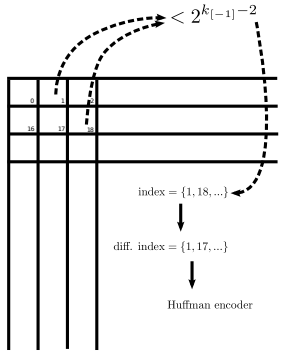

Fig. 5. Map encoding example

The Huffman encoder is very sequential since each differential index value needs to be matched with its encoded value according to the code table. The code table is predefined and build at design time based on simulation data. The specific codeword is loaded from memory and packed. The packing is done in the same way as in IV-D with the exception that the length of the code word is used instead of k. The length of the code word is determined by a subtraction with the word length, 32, with the result of the `exp` function on that codeword.

V. Results

The results presented here indicate the performance of the Xentium processing tile that is currently still under develop-

ment; prototype software development tools have been applied for the given implementation and some of the features offered by the Xentium hardware have not been fully exploited.

The results have been verified with a reference Matlab implementation. The main difference between the Matlab result and the Xentium implementation is the accuracy of the division before quantization. This does not affect the final result because after quantization the results are equal again.

Table I shows the results obtained for the different parts of the algorithm. The cycle count is split up in the number of cycles required before entering the main loop of the kernel, the main loop and the number of cycles after the loop. In the case of the leased-squared-error the cycles after the loop are for the division operation. The total throughput for a single tile at 200Mhz would be around , $\frac{200*10^6}{(4149/256)} = 12.3 * 10^6$, 12.3 MSamples per second. A single block can be processed in, $\frac{4149}{200*10^6} \approx 20$us. Do note that this is for a single tile and by employing more tiles an almost linear speedup is expected.

Kernel	cycle count
Average	2+ 2*254 + 2
LSE	4 + 2*256 + 28
LSBs	2+256+2
Bit packing	5+5*255+5
Map	6+6*256+6
Total	4149

TABLE I
CYCLE COUNT FOR DIFFERENT KERNELS OF A 16×16 BLOCK

From the results it can be seen that the architecture does perform well in cases where a lot of operations can be done on the same data. In the case of the least squared estimator which roughly requires 1790 operations can be done in 544 clock cycles. Looking at the map and the bit packing kernel which both do not require that many operations on the same data can not benefit much from the high degree of parallelism available on the Xentium.

VI. CONCLUSIONS

Based on the results obtained the platform looks very promising. It can also be said that the algorithm is indeed of low complexity for embedded DSP like systems. Two things that are lacking on the current platform are hardware division and an efficient solution to the bit packing problem. If an higher accuracy solution is needed for the division a look-up-table solution might be employed. The result is quantized over 256 levels which means that a look-up-table holding the 256 results of the division can suffice.

An efficient solution to the bit packing problem does not exist for the current platform. When the current implementation is not sufficient, as a last resort a special unit could be designed and this problem can then be offloaded into hardware. This means that the amount of clock cycles for bit packing goes down from $5 + 5 * 255 + 5$ to about 256 clock cycles, depending on the hardware implementation. This saves about 1030 clock cycles.

The algorithm works on blocks of 16×16 samples. Because the blocks can be processed independently the algorithm lends itself very well to parallel processing. The algorithm only needs the samples of one block, the corresponding samples of the previous band and a few other constants. The amount of data fits easily in the local memory of the Xentium. Because most memory accesses are local this means that the algorithm runs efficiently in the Xentium tile.

For an image of 1024×1024 pixels with 50 bands this mend that the whole image of 64×4 blocks can be encoded in $64 \times 64 \times 50 \times 20$us ≈ 4s. Using a chip with 64 Xentium tiles, this would mean an encoding rate of $\frac{64}{4} \approx 16$ images with 50 bands per second.

Overall it can be said that this multi-tiled architecture does have potential when it comes to high throughput demanding algorithms like hyperspectral image compression. A better conclusion can be made when an actual instrument is chosen and accurate specifications are available.

VII. FUTURE WORK

As the Xentium architecture is still under development it will be interesting to see how the platform performs when it reaches a more mature stage. Currently the tooling does not make it trivial to implement the algorithm as a whole and the developer is encouraged to split up the problem into several smaller kernels. When the tooling is better equipped to deal with the algorithm as a single unit it will be of interrest to see what the overall performance gain is. This way a better assessment can be made whether or not this platform could be used on-board.

One of the other things that is of great interrest is the power consumption of the whole platform when this algorithm is run. Another interesting question is how the data should be routed to the tiles of the SoC. In the EU FP7 CRISP project [7] we are working on a prototype implementation of a SoC with 8 Xentium cores interconnected by a NoC.

Finally we will investigate how the implementation results (speed as well as energy consumption) compares with other implementations like standard DSP processors and FPGAs.

ACKNOWLEDGMENTS

We would like to thank Recore Systems for the support with the implementation. We would like to thank ESA's ESTEC TEC-EDP section for providing support with the algorithm.

REFERENCES

[1] A. Abrardo, M. Barni, A. Bertoli, A. Garzelli, E. Magli, F. Nencini, B. Penna, and R. Vitulli, "Low-complexity, secure and error-resilient hyper-spectral image compression," *On-board payload data compression workshop 2008 ESA*, 2008.

[2] Recore Systems, "www.recoresystems.com."

[3] Z. Xiong, A. Liveris, and S. Cheng, "Distributed source coding for sensor networks," *IEEE Signal Processing Magazine*, vol. 21, no. 5, pp. 80–94, September 2004.
[4] D. Slepian and J. Wolf, "Noiseless coding of correlated information sources," *IEEE Transactions on Information Theory*, vol. 19, no. 4, pp. 471–480, July 1973.
[5] A. Wyner and J. Ziv, "The rate-distortion function for source coding with side information at the decoder," *IEEE Transactions on Information Theory*, vol. 22, no. 1, pp. 1–10, January 1976.
[6] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," *SIGPLAN Not.*, vol. 29, no. 6, pp. 61–72, 1994.
[7] Cutting edge Reconfigurable Ics for Stream Processing, "www.crisp-project.eu."