Hardware Architecture for Lossless Image Compression Based on Context-based Modeling and Arithmetic Coding

Xiaolin Chen, Nishan Canagarajah, Jose L. Nunez-Yanez, Raffaele Vitulli
Department of Electrical and Electronic Engineering, University of Bristol, UK
Email:{eezxxc, eecnc, eejlny}@bristol.ac.uk
On-Board Payload Data Processing Section European Space Agency (ESA), Netherlands
Email: raffaele.vitulli@esa.int

ABSTRACT
In this paper we present a novel hardware architecture for context-based statistical lossless image compression, as part of a dynamically reconfigurable architecture for universal lossless compression. A gradient-adjusted prediction and context modeling algorithm is adapted to a pipelined scheme for low complexity and high throughput. Our proposed system improves image compression ratio while keeping low hardware complexity. This system is designed for a Xilinx Virtex4 FPGA core and optimized to achieve a 123 MHz clock frequency for real-time processing.

I. INTRODUCTION
Lossless compression has been successfully used in reducing the bandwidth of communication networks and the storage requirement of digital data. Moreover, lossless image compression is increasingly significant since it is required by many upcoming applications, such as Tele-medicine, space related research, professional visual data studios [1], and next-generation lithography systems [2]. A lot of effort has been made to improve the compression methods both in software [3], [4], and hardware [5], [6]. While software implementation can often yield reasonably good compression ratios at the price of low speed, hardware implementation often suffers from unsatisfactory compression ratios. Furthermore, hardware compressions of general and visual data are usually handled separately. This is not a very efficient approach due to the current trend of network convergence where visual and general data are transmitted along the same physical channel. This fact suggests a technology capable of fast adaptation to the nature of the data and delivering optimal compression ratios.

Given the above requirements, we propose an original reconfigurable FPGA architecture, shown in Fig. 1, to handle general and visual data using context-based modeling and arithmetic coding. We aim at producing an efficient combination of compression schemes for different data types with the final objective of achieving high compression ratio, high throughput and low complexity.

As part of this project, this paper focuses on a novel architecture for lossless gray-scale image compression, which uses context-based modeling, probability estimation and arithmetic coding.

II. LOSSLESS IMAGE MODELING ALGORITHM
The characteristics of images: large alphabet size, two dimensions and big size potentially imply high statistical model complexity and large memory usage. State-of-the-art compression schemes, e.g. CALIC [3], use complex edge detection technique and arithmetic coder to obtain optimal compression ratios. From the view of hardware implementation, however, a higher priority should be given to...
controlling model complexity, which means that memory usage should be minimized and complex operations such as multiplication and division should be avoided if possible. Following these guidelines, we use a simple edge-detecting predictor and a binary arithmetic coder to obtain good compression ratio and hardware amenability.

Image modeling is divided into two phases: prediction and context modeling. In this work, the predictor is inspired by the GAP (Gradient-Adjusted Prediction) from CALIC, a software-based algorithm. However, our method is simplified as fewer contexts (512 vs. 576 in CALIC), and hence less memory, are used. It combines with the probability estimator and binary arithmetic coder to contribute in a new low-complexity scheme.

Fig. 2 shows the denotation of 7 neighboring symbols of the current symbol \( X \), according to their geographical locations. They constitute the context of the current symbol \( X \). In the prediction phase, we estimate the local edge feature by calculating the vertical and horizontal gradients intensity \( dv \), \( dh \), using the differences between context symbols vertically and horizontally. The predicted symbol value \( \hat{X} \) is a linear combination of its contexts, according to the gradient direction and magnitude. The predictor is designed to be suitable for hardware, involving only addition/subtraction and shifting. Thus we obtain the prediction error \( e \), which is the difference between the original and the predicted symbol value. The prediction error \( e \) is also remapped from the range \(-2^{e-1}\) to \(2^{e-1}\), to the range 0 to \(2^{e-1}\) to reduce the alphabet size.

In the context modeling phase, context selection is essential to reduce memory usage. We use 6 context symbols to compare with the predicted value \( \hat{X} \) to obtain a texture pattern \( t \), representing the local texture feature. Also, to indicate the activity of errors in a context, a coding context is generated with the local gradients \( dv \), \( dh \) and an previous prediction error \( e \) of \( W \). The coding context is quantized into 8 levels to form a coding context index \( QE \). Combining the texture pattern and coding context, a set of 512 compound contexts are formed by 6 bits texture pattern \( t \) and 3 bits coding context index \( QE \). These contexts are used to generate an error feedback to adjust the bias of prediction, which will be discussed in the next section. The 8 coding contexts are also used to encode symbols in the probability estimator presented in Section IV.

III. LOSSLESS IMAGE MODELING
ARCHITECTURE

We present our new hardware architecture of the image modeling module in Fig. 3, where the prediction part is illustrated in the solid-line box and the context modeling part is the rest.

Implementation of the modeling can be done in two pipelines running in parallel.

Line 1: a) Calculate prediction error \( e = X - \hat{X} \), where \( \hat{X} \) is the adjusted predicted value;
b) Update sum and number of prediction errors in each compound context;
c) Map prediction error \( e \) to \( \tilde{e} \);
d) Update coding context index \( QE \);

Line 2: a) Update context with new symbol;
b) Calculate gradients \( dv \), \( dh \);
c) Calculate primary prediction value \( \hat{X} \) and quantized coding context \( QE \);
d) Calculate texture pattern and update context index;
e) Calculate the mean of errors \( \bar{e} \) in context for error feedback, and obtain an adjusted predicted value \( \tilde{X} = \hat{X} + \bar{e} \).

Figure 2: Neighbouring Pixels in Prediction and Modeling

Figure 3: Architecture of Image Modeling Module
for next symbol. So the Line 1 operation for the
next symbol can utilize the result of Line 2 from
current symbol when the two lines work in parallel.
Consequently, pipelining can be achieved by
breaking the operation into small pieces and
executing them simultaneously, resulting in a
reduction of clock period. Managing the context
memory and the error feedback technique, which
calculates the mean of errors in a context, are the
most difficult parts. The details are given below.

In the compression process, we need to store
3 lines of image pixel values in memory as context
and use 3 pointers to indicate symbol locations in
each line. At the end of processing each line, the 3
pointers to each line have to be rotated in a certain
order so that the oldest line will be discarded and
the newly formed line will be saved.

An error feedback technique is used in the
prediction schemes to adjust the prediction in a
certain context, because the mean of errors is
the most probable prediction error in each context.
This error feedback can help with reducing the bias
of the primary prediction. The mean of errors in a
context is simply given by
\[
\overline{e} = \frac{\text{sum}}{\text{count}}
\]  
(1)

where \(\text{sum}\) and \(\text{count}\) are the sum and occurrence
of errors in the context, respectively. In order to
calculate the mean, \(\text{sum}\) and \(\text{count}\) in each context
are stored in memory, as shown in Fig. 3. For
memory efficiency, we only use 5 bits to store the
error occurrence. When the \(\text{count}\) reaches its
maximal value 31, it is halved by right-shifting one
bit; meanwhile \(\text{sum}\) is halved so as to maintain the
mean \(\overline{e}\). Thus we only need 13 bits \((2^5 \times 2^8 = 2^{13})\)
plus one sign bit to store the sum of errors safely.
This rescaling function is completed by the block of
Overflow Guard. Experimental results prove that
this rescaling technique slightly improves the
compression ratio by “aging” the observed data.
However, division is always a difficult problem in
hardware, especially when the dividend can be as
large as 13 bits. To make this division practical, we
bound the dividend \(\text{sum}\) by 10 bits for two reasons:
firstly experiments on our image test set show that
the chance of the sum being larger than 1023 is
less than 0.001%; secondly, extraordinary large
ersors tend not to reflect the true behavior of the
context because prediction errors should be
moderately small given an adequate predictor. We
use the most significant bits of the divisor \(\text{count}\) in
the division, with the dividend being rescaled
accordingly to maintain the same result.
Consequently, we only need a lookup table of
1KByte \((2 \times 512 = 1024)\) to complete fast division.
Although the result of division is only an
approximation, it does not affect the compression
performance in our experiments.

The outputs of the modeling module are the
mapped error \(\overline{e}\) and the coding context index \(\text{QE}\)
which are sent to the probability estimator.

IV. PROBABILITY ESTIMATOR AND BINARY
ARITHMETIC CODER

The probability estimator adaptively calculates
the probability of symbol occurrence in each
code in a SRAM. It enables the application of a
simple and fast binary arithmetic coder and the
decomposition of the coding procedure into bit
level operations, and hence full pipelining and high
throughput.

Each context is represented by a balanced
binary tree with \(2^n\) (\(n\) is the bits per pixel) nodes
associated with each symbol in the alphabet. A
number of bits are used to store the symbol
frequency count in each node. Initially, all the
symbols in the alphabet are assigned an equal
probability, e.g. 1/256 in an alphabet of 256
symbols. When one symbol is received, the value
of the corresponding tree node increases to reflect
the probability distribution of symbol occurrence.
There are 8 coding contexts for image
compression, corresponding to 8 “dynamic” trees
and one “static” tree for coding the escape symbols.
Escape happens when a valid probability of a
symbol cannot be found, e.g. when its probability is
0, in which case the symbol is “escaped” to the
“static” tree and is sent as it is. Escape is
undesirable as it does not achieve any
compression. It takes place when some symbol
counts reach the maximal frequency count, e.g. 14
bits for \((2^{14} - 1)\), in which case all the symbol
counts in the tree will be halved. Consequently, the
counts of symbols that have not been seen before
will be rescaled from 1 to 0, resulting in escape
when those symbols occur later for the first time.
Therefore, the frequency count bits have to be
carefully chosen. Experimental results of average
compression bit rates under different frequency
count bits are shown in Fig. 4. Note that when too
Figure 4: Average Bit Rates under Different Probability Precision
few bits are used, more escapes happen; when too many bits are used, fewer escapes happen but the probability distribution is so skewed that more bits are needed to encode the symbols with small probability. Therefore we choose 14 bits for the symbol counts.

As tree node value increment works from the root down to the leaves through the tree, a symbol can be fully encoded by the binary decision (left or right) taking place in each level of the tree.

The binary arithmetic coder is driven by the binary decision bits and the probability data from probability estimator. More details regarding the arithmetic coder can be found in [7].

V. IMPLEMENTATION AND PERFORMANCE COMPARISON
In this section we present the experiment results of the proposed lossless image compressor. We compared the performance of different image compressors in Table. 1, using a set of grey-scale test images of size 512*512 pixels. JPEG-LS (LOCO-I) [4] and SLP (Switched Linear Prediction) are low complexity compression schemes using Golomb-Rice coder. Clearly, the proposed scheme outperforms JPEG-LS and SLP in terms of compression ratio with comparable low complexity, though yields slightly bigger compression ratio compared to software CALIC since fewer contexts and simpler hardware-amenable modeling techniques are used in our system.

Table 2 is the device utilization summary of the hardware implementation on a Xilinx Virtex4 FPGA chip. Memory usage for modeling is 3.7KBytes, for probability estimator is 4KBytes. The design was synthesized and optimized using Xilinx ISE 8.1 and achieved a clock frequency of 123 MHz, and a throughput of 123Mbits/sec. The low complexity means that a multi-core solution could be used to scale up the performance.

VI. CONCLUSIONS
A novel hardware architecture for context-based lossless image compression is proposed in this paper. As a result, lossless compression is achieved efficiently with low complexity hardware design. Our experiments show improvement in terms of compression ratio when comparing to other low complexity schemes. The combination of different modeling modules optimized for various data types, has the potential of achieving a high efficient universal lossless compressor.

REFERENCES

Table 1. Bit Rates Comparison of a few Selected Schemes

<table>
<thead>
<tr>
<th>Image</th>
<th>JPEG-LS</th>
<th>SLP(M0)</th>
<th>CALIC</th>
<th>proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>barb</td>
<td>4.86</td>
<td>4.79</td>
<td>4.59</td>
<td>4.68</td>
</tr>
<tr>
<td>boat</td>
<td>4.25</td>
<td>4.28</td>
<td>4.12</td>
<td>4.18</td>
</tr>
<tr>
<td>goldhill</td>
<td>4.71</td>
<td>4.74</td>
<td>4.61</td>
<td>4.65</td>
</tr>
<tr>
<td>lena</td>
<td>4.24</td>
<td>4.17</td>
<td>4.09</td>
<td>4.14</td>
</tr>
<tr>
<td>mandrill</td>
<td>6.04</td>
<td>5.99</td>
<td>5.9</td>
<td>5.93</td>
</tr>
<tr>
<td>peppers</td>
<td>4.49</td>
<td>4.49</td>
<td>4.35</td>
<td>4.39</td>
</tr>
<tr>
<td>zelda</td>
<td>4.01</td>
<td>3.97</td>
<td>3.84</td>
<td>3.90</td>
</tr>
<tr>
<td>average</td>
<td>4.66</td>
<td>4.63</td>
<td>4.50</td>
<td>4.55</td>
</tr>
</tbody>
</table>

Table 2: Device Utilization Summary

<table>
<thead>
<tr>
<th></th>
<th>Modelling</th>
<th>Probability</th>
<th>Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Slices</td>
<td>508</td>
<td>297</td>
<td>1123</td>
</tr>
<tr>
<td>No. of Slice Flip-flop</td>
<td>224</td>
<td>124</td>
<td>283</td>
</tr>
<tr>
<td>No. of 4 input LUT</td>
<td>912</td>
<td>561</td>
<td>2131</td>
</tr>
<tr>
<td>No. of bonded IOBs</td>
<td>31</td>
<td>60</td>
<td>53</td>
</tr>
<tr>
<td>No. of GCLK</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>