Institut für Parallele und
Verteilte Systeme
Abteilung Parallele Systeme
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit Nr. 3049

Parallelization of JPEG-LS Encoder for Lossless Compression of Images in CUDA

Haitham Assem Tantawy

Course of Study: INFOTECH

Examiner: Prof. Dr. Sven Simon

Supervisor: Dipl. Inf. Simeon Wahl

Commenced: June 01, 2010

Completed: November 30, 2010

CR-Classification: E.4, F.1.2, I.3.1, I.4.2

Acknowlegement

I would like to thank each of Simeon Wahl and Tjark Bringewat for their supervision and continuous guidance throughout my work on this thesis. I would also like to thank professor Sven Simon for giving me the opportunity to contribute to such a research field, marking one of my most important achievements.
This work wouldn't have been completed without the encouragement I have always received from my parents Ola and Assem, as well as my little sister Doha. I would also like to dedicate special thanks to my best friend Melanie, who constantly motivated me to proceed further.

Haitham Assem Tantawy

Abstract

This thesis investigates the tendency of encoding parallelism of the lossless JPEG-LS encoder. We analyze the existing data dependencies arising from the context modeling process used in the standard, and which dictates a sequential encoding of pixels that are classified to the same context. Previous work, found in the literature, has approached parallelism in JPEG-LS either by pipelined hardware implementations or software implementations, which modify the context update procedures in order to break the data dependency loop, and thus become no more compliant to the established standard. In our work, we present an encoder design, referred to as *round encoder*, which achieves parallelism in JPEG-LS by encoding the image in terms of context rounds. Each round contains pixels of different contexts which can be encoded in parallel. The proposed round encoder design accomplishes the same number and order of the performed context updates like the standard, and thus is fully compatible. The degree of maximum achievable parallelism however depends on the context distribution of pixels in the image, and thus parallelism is not guaranteed.
The thesis also presents a parallel software implementation of the JPEG-LS encoder, which realizes the round encoding concept being fully compatible with the standard. A GPU is used as a target hardware architecture for our implementation in conjunction with a CPU for the sequential working parts. A speedup is achieved for some of the parallelized algorithmic blocks. However, no speedup is obtained for the whole parallel JPEG-LS encoder with respect to the original sequential implementation. Possible reasons behind the low performance are discussed with further suggestions for future work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Image compression is considered to be one of the applied concepts used in the last decades to reduce the storage size occupied by images and save bandwidth upon their transfer. The fast development in the Internet, multimedia technologies and exchange of information has always demanded a rising and accelerating progress in that field of study, allowing a number of image compression algorithms, methods and standards to evolve and serve various applications.
Focusing on the lossless type of image compression, the image is compressed in such a way that it can be reconstructed without any loss in the image quality. This has been an important issue in a variety of applications where images are subject to further processing, intensive editing or also when they are obtained at high cost. Such applications include image archiving, remote sensing, medical imaging and certain military applications that work with target recognition. These applications strictly demand that compression of the image does not lead to any change in the raw image data when performing decompression.
In lossless compression schemes, two consecutive working phases are to be recognized: modeling and coding. The modeling phase is aimed at gathering information about the image data in the form of a probabilistic model, which is then to be utilized by the coding phase in generating the output bit sequence. The modeling can be done either statically, where a single constructed model is available for the data, or it can be done adaptively, where the model gets dynamically updated during the compression process. Most of the present lossless image coders such as CALIC coder, Sunset algorithm and JPEG-LS realize an adaptive modeling paradigm.
Although the modeling/coding architecture has achieved high coding competence in terms of compression rates, however it often resulted in high computational costs because of the statistical analysis in the modeling phase and the subsequent computations required in the coding phase. Accordingly, speed performance and computation time have recently become one of the crucial factors judging the efficiency of an image compression algorithm.
On the other side, there has been a massive revolution in the last decades in parallel hardware architectures and subsequent parallel computing, pushing the limits of processing speeds to wider boundaries, which were harder to achieve before by single processing hardware units. The fact that increasing performance through parallel processing is far more energy-efficient than increasing microprocessor clock frequencies, resulted in having parallel computing becoming a mainstream based on multi-core processors and Graphic Processing Cards (GPUs). Most desktop and laptop systems nowadays ship with built in Graphic cards making them readily available

everywhere. This consequently opened new horizons in software development and made software parallelism an indispensible task.
Despite the sequential nature of lossless image compression algorithms, which can be dictated by either the performed adaptive modeling process or the variable length encoding scheme, there has been previous research done to exploit parallelism in these algorithms, trying to modify them into a multi process manner in order to make advantage of the existing parallel hardware architectures.
This thesis is intended to investigate how far a sequential image compression algorithm is to be parallelized, and how much increase in performance is to be gained, while taking into account the data and instruction dependencies set by the standard for the encoding process. JPEG-LS encoder is chosen for investigation as a lossless image compression algorithm because of its simplified coding scheme, low complexity and high attained compression ratios. The parallel implementation is to be exploited on a GPU because of its flexible architecture allowing the most efficient software hardware mapping, as will be demonstrated in the rest of this work.

1.2 Thesis Structure

This thesis is organized as follows. Chapter 2 introduces the necessary background of image encoding using the JPEG-LS standard, highlighting the main concepts needed for the understanding of the presented work. Chapter 3 demonstrates the general theory of parallelism by presenting types of parallelism and performance limitations. Also the hardware architecture of the GPU is introduced, discussing about its parallel execution model and the available memory hierarchy. In Chapter 4, the tendency of encoding parallelism in the JPEG-LS standard is exploited, pointing out the main data and instruction dependencies found in the standard, and how previous work has approached corresponding solutions. Chapter 4 introduces a new encoding concept, referred to as *round encoding*, which aims at achieving encoding parallelism in JPEG-LS. The proposed design methodology and implementation strategy of parallel JPEG-LS are discussed in Chapter 5, where a detailed description is provided of how the round encoding concept is realized to achieve parallelism. Chapter 5 also presents the possible bottlenecks and limitations of the suggested implementation. The Performance of parallel JPEG-LS is analyzed in Chapter 6 where a comparison to the sequential version is carried out and the achieved speedup results are presented. The thesis finally concludes in Chapter 7.

Chapter 2

JPEG-LS for Lossless Compression of Images

The fact that the lossless coding mode of JPEG has never been very much successful, producing compression ratios inferior to other lossless compression algorithms [1], the ISO in cooperation with IEC have decided to develop a new standard for lossless image compression for continuous tone images known as JPEG-LS. The standard is based on the ideas of LOCO-I (Low COmplexity LOssless COmpression for Images) algorithm [2], which realizes a low complexity implementation of the universal context modeling paradigm. The overall reduction in complexity is mainly achieved by a simplified implementation of the modeling unit realizing a simple context model. The model is capable of capturing high order dependencies, and is further refined for efficient performance alongside with an extended family of adaptively chosen Golomb type codes [2]. In this manner LOCO-I/JPEG-LS becomes capable of attaining higher compression ratios compared to other algorithms, which are only based on arithmetic/Huffman coding, and to run at a much lower computational complexity. JPEG-LS has been also found to be superior to the lossless mode of the succeeding JPEG 2000 [3].

2.1 Basic encoding process of JPEG-LS

In the JPEG-LS standard, the image is sampled at the input of the encoder in a raster scan order. The encoding of each pixel is based on the so called context modeling, which provides an classification of each pixel into a specific context according to its neighborhood pixels. All subsequent encoding operations are dependent on the pixel's assigned context. Two types of encoding modes are present in the JPEG-LS standard as follows.

Regular mode

Pixels encoded in this mode of operation are called regular-mode pixels. A regular mode pixel is characterized by having a value, which is different from its previously scanned neighborhood pixel of the same image line. Encoding of regular mode pixels is achieved by making a prediction about their values based on their assigned contexts. The error in the made prediction with respect to the pixel's actual value is to be encoded by a Golomb coder.

Run mode

Pixels encoded in this mode of operation are called run-mode pixels. A run mode pixel is characterized by having the same value as its previously scanned neighborhood pixel of the same image line. Encoding of run-mode pixels does not include prediction. Only the count of the identified consecutive run-mode pixels, constituting a run-segment, needs to be encoded. In addition, encoding of how the run-segment has ended is also performed. More details will follow in the coming sections.

2.2 Detailed Description of Lossless JPEG-LS Encoder

Figure 2.1 shows the encoder block diagram of JPEG-LS comprising both the modeling and coding parts. A detailed description of the individual functioning blocks, obtained from the standard documentation in [4], is to be demonstrated in the next sections. It should be however noted that all the code belonging to the near lossless mode of JPEG-LS has been omitted, as this mode of operation is not handled in this work.

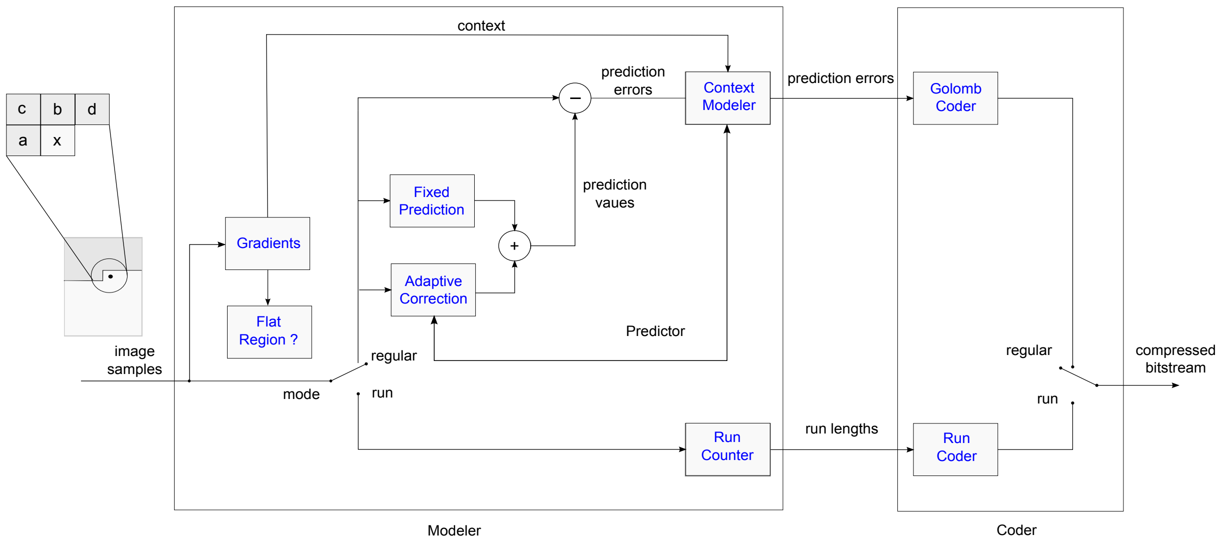


Figure 2.1: JPEG-LS Blockdiagram

2.2.1 Context Classification

When coding the current pixel, whose position x is shown in Figure 2.2, JPEG-LS examines four surrounding, previously scanned pixels of the current pixel x, referred to as the causal template. These are the pixels at positions a ,b ,c and d as shown in Figure 2.1. The values of the four surrounding pixels are used to classify pixel x to a certain context. For pixels on the boundaries of the image, certain initializations for the causal template are to be performed as described in [4].

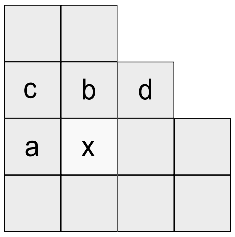


Figure 2.2: Context for pixel x

Local gradient estimation

The context classificaton for pixel *x* starts by estimating the differences *D1*, *D2* and *D3* of the causal template as indicated in Listing 2.1, with *Ra*, *Rb* and *Rc* being the reconstructed values[1] of pixels at positions *a*, *b* and *c* respectively. These differences are referred to as the local gradients of pixel *x* ,and they capture the pixel's surrounding level of activity (smoothness, edginess). They also determine the statistical behavior of the prediction errors which are to be encoded later in the process.
Based on the values of the three local gradients, a decision is made on whether to encode pixel *x* in a regular or a run mode as shown in Listing 2.1. If the run mode is selected, the encoder proceeds with the protocol specified in Section 2.2.5.

Listing 2.1: Local gradient estimation and mode selection

```
D1 = Rd - Rb
D2 = Rb - Rc
D3 = Rc - Ra

if((D1==0) && (D2==0) && (D3==0))
        RunModeProcessing;
else
        RegularModeProcessing;
```

Local gradient quantization and merging

In case of a regular mode encoding, quantization of the local gradients follows as specified in Listing 2.2. This step in the context classification process is performed in order to reduce the total number of contexts that can be generated. This results in lowering the amount of memory required for encoding and allows for a larger number of samples to be coded in the same context, and thus achieving better statistics.
In the procedure specified in Listing 2.2, the three local gradients *D1*, *D2* and *D3* are compared to three non-negative threshold parameters *T1*, *T2* and *T3*[2]. According to the examined relation, a vector (*Q1*,*Q2*,*Q3*) is obtained with each of *Qi* taking a value between -4 and 4. To further

[1] In the lossless mode of JPEG-LS, the reconstructed values of all pixels are exactly the same as the original ones. They only differ in the case of near-lossless encoding which is not discussed in this work

[2] The values of the threshod parameters are to be chosen freely. For an 8-bit precision image, suggested values could be T1 = 3; T2 = 7 and T3 = 21. More details about choosing the values of these parameters are provided in [4]

reduce the number of generated contexts, quantized context triplets of opposite signs are merged together into one context triplet (for instance (*Q1*,*Q2*,*Q3*) and (-*Q1*,-*Q2*,-*Q3*). This merging is then compensated by changing the sign of the prediction error in the coming step.
After the merging process has been completed, the vector (*Q1*,*Q2*,*Q3*) is then used to calculate the integer *q* which represents the context of pixel *x* having a range of [0..364]. This integer is to be required later to index arrays *A*,*B*,*C* and *N*, as will be shown shortly.

Listing 2.2: Local gradient quantization

```
for(i=0; i<3; i++){
  if(D[i] <= -T3)       Q[i] = -4;
  else if(D[i] <= -T2)  Q[i] = -3;
  else if(D[i] <= -T1)  Q[i] = -2;
  else if(D[i] < 0 )    Q[i] = -1;
  else if(D[i] <= 0)    Q[i] = 0;
  else if(D[i] < T1)    Q[i] = 1;
  else if(D[i] < T2)    Q[i] = 2;
  else if(D[i] < T3)    Q[i] = 3;
  else                  Q[i] = 4;
}
q = 81*Q[0] + 9*Q[1] + Q[2];
```

2.2.2 Prediction

After the determination of the context in the regular mode, the encoder proceeds with the next step predicting the value *Px* of the current pixel *x*. The prediction process includes two steps as shown in Listing 2.3. These are fixed prediction, based on edge rules, and adaptive correction of the predicted value. The error resulting from the difference between the pixel's predicted and actual value is then computed and encoded.

Edge-detection and predection correction

The fixed predictor in JPEG-LS performs a primitive test to detect the horizontal and vertical edges found above and to the left of the encoded pixel *x* respectively . For this test, the three samples of the causal template in positions *a*, *b* and *c* are required for comparison as depicited in Listing 2.3. The predictor selects the value of pixel *b* when a vertical edge is detected. If a horizontal edge is found, the value of pixel *a* is chosen. Otherwise, when no edges are found, a combined value of the three pixels is predicted.
The prediction procedure is then followed by the adaptive correction mechanism shown in Listing 2.4, which aims at compensation for systematic biases in the prediction. The corrected value of *Px* depends on both the *SIGN* variable, determined from the merging process, and the correction values stored in *C[q]*. The bias computation of these correction values is discussed in Section 2.2.4.

Listing 2.3: Edge prediction

```
if(Rc >= MAX(Ra,Rb))
```

```
  Px = MIN(Ra,Rb);
else {
  if(Rc <= MIN(Ra,Rb))
    Px = MAX(Ra,Rb);
  else
    Px = Ra + Rb - Rc;
}
```

Listing 2.4: Prediction correction

```
if(SIGN == 1)
  Px = Px + C[q];
else
  Px = Px - C[q];

if(Px > MAXVAL)
  Px = MAXVAL;
else if(Px < 0)
  Px = 0;
```

Computation of prediction error

Using the corrected value *Px*, the prediction error Errval is then computed as shown in Listing 2.5, where the *SIGN* variable is again used to determine if the sign of *Errval* should be reversed. A modulo reduction is then applied to the prediction error to reduce it to a range relevant for coding.

Listing 2.5: Computation of prediction error

```
Errval = Ix - Px;
if (SIGN == -1)
   Errval = -Errval;
Errval = ModRange(Errval, RANGE);
```

2.2.3 Golomb coding of the prediction error

The encoding step in the regular mode is performed by a scheme derived from Golomb coder. The error value is first mapped to a non-negative integer *MErrval*, and is then encoded using the code function LG(*k*,*LIMIT*). The variable *k*, for the limited length code function, is computed according to the procedure shown in Listing 2.6. This parameter depends on both variables *A[q]* and *N[q]*, and thus is considered to be context dependent. The value of *k* for a given context is updated each time a pixel with the same context is found.

Listing 2.6: Computation of Golomb variable

```
for(k=0; (N[q]<<k) < A[q]; k++);
```

2.2.4 Context variables update

The encoding of the sample *x* in the regular mode is ended by updating the context variables *A[q]*, *B[q]*, *C[q]* and *N[q]*. This update process has to be performed after both the values of *k* and *MErrval* have been computed.

Update

The update of variables *A[q]*, *B[q]* and *N[q]* follows according to the procedure depicted in Listing 2.7. Further details on the update procedure are found in [4].

Listing 2.7: Context variables update

```
B[q] = B[q] + Errval;
A[q] = A[q] + ABS(Errval);
if (N[q] == RESET) {
  A[q] = A[q]>>1;
  B[q] = B[q]>>1;
  N[q] = N[q]>>1;
}
N[q] = N[q]+1;
```

Bias estimation

As has been discussed in Section 2.2.2, the prediction correction values, which are needed for the bias cancellation, are stored in *C[q]*. At the end of the encoding process of pixel *x*, this variable needs to be updated to account for bias changes within the same context. The variables *N[q]* and *B[q]* are required for the update procedure as shown in Listing 2.8. Further details on the bias computation procedure are provided in [4].

Listing 2.8: Bias estimation

```
if(B[q] <= -N[q]) {
  B[q] = B[q] + N[q];
  if(C[q] > MIN_C)
    C[q] = C[q] - 1;
  if(B[q] <= -N[q])
    B[q] = -N[q] +1;
}
else if(B[q] > 0) {
  B[q] = B[q] - N[q];
  if(C[q] < MAX_C)
    C[q] = C[q] + 1;
  if(B[q] > 0)
    B[q] = 0;
}
```

2.2.5 Run mode

Encoding in the run mode is done in a different manner. The encoder selects this mode for pixel x when the values of the surrounding context pixels *a*, *b*, *c* and d are identical. In this mode of operation, neither prediction nor error encoding is carried out. The encoder however starts at pixel *x* and looks for the longest sequence of run pixels whose values are identical to the reconstructed value *Ra* of context pixel *a*. This is called a run segment. The run mode terminates by either encountering a sample with different value (interruption sample), or by reaching the end of the image line. The two main tasks in this mode are (1) run scanning and run-length coding, and (2) run interruption coding.

Run scanning and run-length coding

Listing 2.9 shows the procedure according to which the run-length, referred to as *RUNcnt*, is computed. After the procedure finishes, the variable *EOLine* indicates whether the run segment was interrupted or not. The encoder then continues by encoding the value of *RUNcnt*, which provides information about the length of the identified run-segment. More details on the encoding of run-lengths are found in [4]

Listing 2.9: Run scanning

```
RUNval = Ra;
RUNcnt = 0;
while(Ix == RUNval){
  RUNcnt += 1;
  Rx = RUNval;
  if(EOLine == 1)
     break;
  else
     GetNextSample();
}
```

Run interruption coding

If the-run segment was terminated before reaching the end of the current image line, the new sample that casued the interuption shall be encoded in a quite similar manner to the regular encoding mode. The encoding steps are as follows.

- Computation of the index *RItype*, which defines the context of the interruption sample as shown in Listing 2.10. This parameter obtains a value of either 0 or 1 depending on the relation between *Ra* and *Rb*.
- According to the estimated *RItype*, the value of the interruption sample is predicted and the prediction error *Errval* is computed as shown in Listing 2.10.
- The sign of *Errval* is then corrected, if necessary, and a modulo reduction is further applied as shown in Listing 2.11.

- The context integer *q* is computed as (*q* = *RItype*+365), and the Golomb variable *k* is determined in a similar manner as in the regular mode encoding case.
- The prediction error is mapped to a non negative integer *EMErrval*, and is then encoded by the limited length Golomb code function LG(*k*,*glimit*).
- The Context variables *A[q]*, *N[q]* and *Nn[q]* are updated according to the procedure in Listing 2.12.

Listing 2.10: Computation of prediction error of interruption sample

```
if(Ra == Rb)
  RItype = 1;
else
  RItype = 0;

if(RItype == 1)
  Errval = Ix - Ra;
else
  Errval = Ix - Rb;
```

Listing 2.11: Sign correction and mapping of the prediction error

```
if((RItype == 0) && (Ra>Rb)){
  Errval = -Errval;
Errval = ModRange(Errval, RANGE);
```

Listing 2.12: Context variables update for the interruption sample

```
 if(Errval < 0)
  Nn[q] = Nn[q] + 1;
A[q] = A[q] + ((EMErrval+1-RItype)>>1);
  if(N[q] == RESET){
     A[q] = A[q]>>1;
     N[q] = N[q]>>1;
     Nn[q] = Nn[q]>>1;
}
N[q] = N[q] + 1;
```

Chapter 3

Theory of Parallelism

3.1 Types of Parallelism

The concept of parallel computing relies on breaking large problems, being serially executed, into a set of smaller problems that can be solved concurrently, and thus gaining an enhanced speed performance. Parallelism can be exploited in many forms, which differ in their abstraction levels. Each of these forms realize parallel computing in a different manner suiting the specific sequential nature of various problems. Below is a detailed description of two types of parallelism, data and task parallelism.

3.1.1 Data Parallelism

Many problems can be best understood as a sequence of operations being executed on a core data structure, where the elements of the structure are updated or used for computation. Data Parallelism implies the geometrical decomposition of the structure into chunks of data, on which the same computation task is performed concurrently. This kind of parallelism is usually exploited in serial programs, where a set of computationally intensive data loops dominate the execution time. In such a case, parallelism can be approached by unfolding the loops and distributing the individual iterations among the execution units, keeping the existing loop dependencies into account. The following key points should be considered when applying data parallelism.

- **Data decomposition**
  Partitioning the global data structure into chunks requires finding the optimum decomposition size which ensures optimum synchronization, communication and load balancing among the execution units. Types of decomposition include : 1) Coarse-grained decomposition, with smaller number of large data chunks, 2) Fine-grained decomposition, with larger number of small data chunks. Coarse-grained decomposition offers a lower amount of data sharing between the execution units, however a worse load balancing in comparison to fine-grained decomposition.

- **Exchange Operation**
  During the computation process, data from neighboring chunks might be required for access. This essentially demands an efficient data exchange between nearby chunks. The exchange process is usually achieved by either taking advantage of a shared memory or by copying the extra required data to local structures before starting the computation. In both cases coordination between the execution units is required to ensure data availability.

- **Update/Computation operation**
  parallelization is considered to be easier and probably much more efficient, when the non-local data, copied from other chunks before the computation, are not altered in the computation phase. Otherwise, extra code must be added to ensure that the correct data is found and to ensure a proper access protocol.

- **Data distribution and scheduling among execution units**
  Determining which execution units will work on which data chunks strongly affects the gain in performance, especially when the amount of computation per chunk is not uniform. Distributions can be either done statically or dynamically, depending on the degree of computation uniformity among the chunks, and the desired level of load balancing.

3.1.2 Task Parallelism

Task parallelism, on the other hand, is the characteristic of a program to be decomposed into a set of different tasks, involving different computations, that are applied on the same or different sets of data. In contrast to data parallelism, task parallelism usually does not scale with problem size, however it is recommended in programs where the computation is naturally divided among a large set of slightly dependent working blocks. Three key elements are to be noted when applying task parallelism as follows.

- **Task definition**
  Defining the size and the total number of tasks available strongly affects both scheduling flexibility and load balance. In general the amount of tasks available should exceed the number of parallel execution units.

- **Task dependencies**
  Most of the existing dependencies, found between the tasks, arise from data dependencies. Possible strategies to handle those dependencies can be applied by either replicating data or by grouping single updates and executing them at a delayed stage.

- **Task scheduling**
  Although static task scheduling is simple to implement and requires the least overhead, however it often becomes less suitable in cases where the task length is unpredictable. Dynamic task scheduling on the other hand achieves an almost equal distributed load among the execution units. However, It results in more overhead and requires additional scheduling structures to implement.

3.2 Amdahl's Law

Amdahl's law defines the maximum theoretical amount of speed improvement that can be gained for a certain program, when a portion of the computations is parallelized by a number of parallel working processors. Equation 3.1 shows the calculation of the speedup, with P representing the proportion of the program that is parallelized, and N being the number of parallel processing units involved in the computation of P.

$$Speedup = \frac{1}{(1-P)+P/N} \tag{3.1}$$

Basically, the law states that the maximum achievable speedup of a program is always limited by its sequential portion (1-*P*), regardless of how many processors are devoted for the parallelization of portion *P*. The maximum speedup will always remain 1/(1-*P*).
Both careful analysis of program dependencies (finding out the sequential limit) , and inspection of the minimum overhead that is created with parallelization, are necessary procedures that need to be considered in the design phase of parallel algorithms.

3.3 Parallel Hardware Architecture

3.3.1 Flynn's taxonomy

Flynn's taxonomy is a specific classification of parallel computer architectures that is based on the number of concurrent instructions (single or multiple) and data streams (single or multiple) available in the architecture. The four categories in Flynn's taxonomy are the following:

Single Instruction, Single Data stream (SISD)

This describes a sequential architecture which exploits no parallelism in either the instructions or data streams. Examples of SISD architecture are the traditional uniprocessor machines.

Single Instruction, Multiple Data streams (SIMD)

This describes an architecture which exploits parallelism by performing a single operational task (instruction) on multiple data streams. Examples of SIMD architecture are array processors and GPU .

Multiple Instruction, Single Data stream (MISD)

This type of architectures exploits parallelism by performing multiple operations (instructions) on a single data stream. MISD architectures are used for fault tolerance.

Multiple Instruction, Multiple Data streams (MIMD)

This describes an architecture, where multiple processors are simultaneously executing different instructions on different data streams. Examples of MIMD architecture are distributed systems .

3.4 General-Purpose Graphic Processing Unit GPGPU

A GPGPU is a parallel hardware SIMD architecture, with a very high computational power in floating point operations [5]. The reason behind the discrepancy in floating-point capability is that the GPU is designed such that more transistors are devoted to data processing, rather than data caching and flow control. This makes the GPU specialized for compute-intensive problems, which can be expressed as data parallel computations. Because the computations are usually of a high arithmetic intensity and they are executed on many data elements, the memory access latency becomes hidden within the massive amount of performed calculations.

3.4.1 GPU hardware architecture

As shown in Figure 3.1, the GPU consists of a set of multiprocessors, with each multiprocessors having a set of 32-bit processors. The processors share a single instruction unit, and thus implementing a single instruction multiple data architecture, SIMD, as was previously mentioned. At each clock cycle, the multiprocessors execute the same instructions on a group of threads called a warp, with the warp size being 32.
The threads have access to different memory spaces, as shown in Figure 3.1, constituting a memory hierarchy as follows.

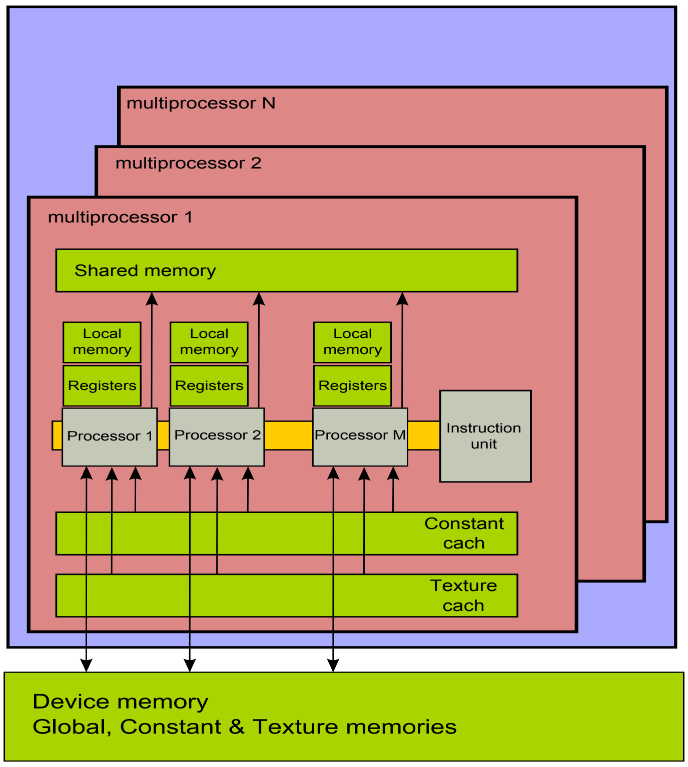


Figure 3.1: GPU architecture

Global memory

A non cached read/write memory space accessed by all threads within all multiprocessors. Accesses to the global memory (device memory) are considered to be costly and therefore the right access pattern has to be applied by taking the following into account.

- **First**, The device is capable of reading 4-byte, 8-byte or 16-byte words from global memory into registers in a single instruction.
- **Second**, global memory bandwidth is maximized when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 32, 64 or 128 bytes. Further details on coalescence rules are provided in [5].

Constant memory

A cached read-only memory space accessed by all threads within all multiprocessors. Reading from the constant memory cache is as fast as reading from a register as long as all threads of a half-warp read the same address. On a cash miss, a read from the constant memory costs one read from device memory (global memory).

Texture memory

A cached read-only memory space accessed by all threads within all multiprocessors. The texture cache is optimized for 2D spatial locality and is designed for streaming fetches with a constant latency.

Local memory

A non cached read/write memory space, private for each thread. Accesses to the local memory are as expensive as accesses to the global memory, however they are always coalesced since they are per-thread by definition.

Shared memory

An on-chip read/write memory accessed by all threads within one multiprocessor. The shared memory is divided into equally-sized memory banks, which can be accessed simultaneously. Any memory request made for n addresses that fall in n distinct memory banks can be served simultaneously, resulting in an n times higher bandwidth than that of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access is serialized.
The banks of a shared memory are organized in such a way that successive 32-bit words are assigned to successive banks. For a warp size of 32 threads and a number of shared memory banks equal to 16[1], a memory request for a warp is split into two separate requests. One request for the first half of the warp and a second request for the second half of the warp. This means that there can be no bank conflict between threads belonging to different half warps. Within one request for the 16 threads of a half warp, the memory access is as fast as accessing the registers as long as there are no bank conflicts, otherwise the request is serialized and a lower bandwidth is obtained. Further details on avoiding bank conflicts are provided in [5].

Registers

A set of 32-bit registers is available per processor, with each thread accessing its own registers independently.

3.4.2 CUDA programming model

CUDA (Compute Unified Device Architecture) is NVIDIA's programming model for parallel processing on the GPU. The model assumes the device (GPU) to be a separately working co-processor to the host (CPU) running the program. Executions on the device are done through

[1]The number of available shared memory banks differs from one device to another. Devices with compute capability of 1.x have 16 shared memory banks

the initiation and the launching of a kernel, which is considered to be a single unit of arithmetic operation. A kernel is launched by the host and is executed as a grid of parallel running thread blocks on the device as shown in Figure 3.2. Each thread block is executed by one multiprocessor. The number of blocks per grid as well as the number of threads per block are to be specified upon kernel launch time.
CUDA's programming model also assumes that the communication between the host and the device is to be done through the device memory, which is visible to all kernels as shown in Figure 3.3. CUDA runtime calls are used for device memory allocation and de-allocation as well as data transfer between host and device memories. Since each block of threads is executed by one multiprocessor, threads of different blocks can not communicate via shared memory, as shown in Figure 3.3.

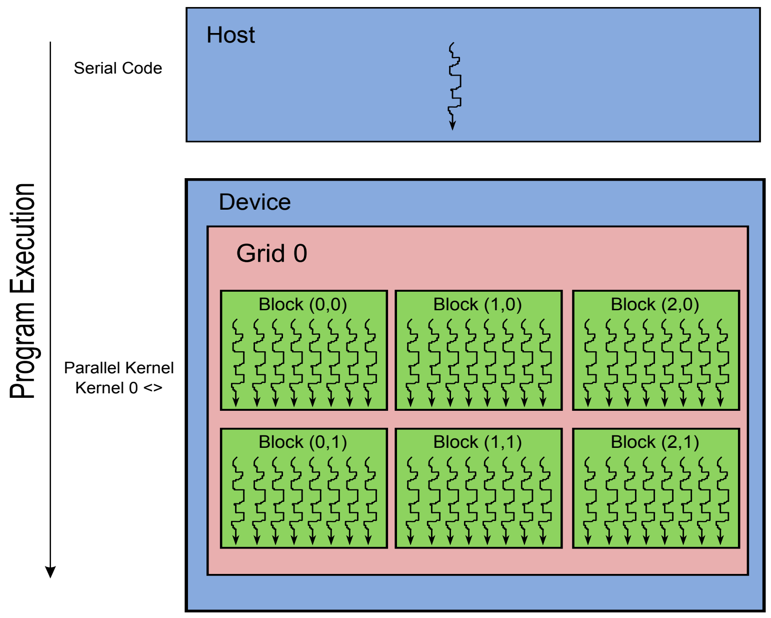


Figure 3.2: Program execution on host and device

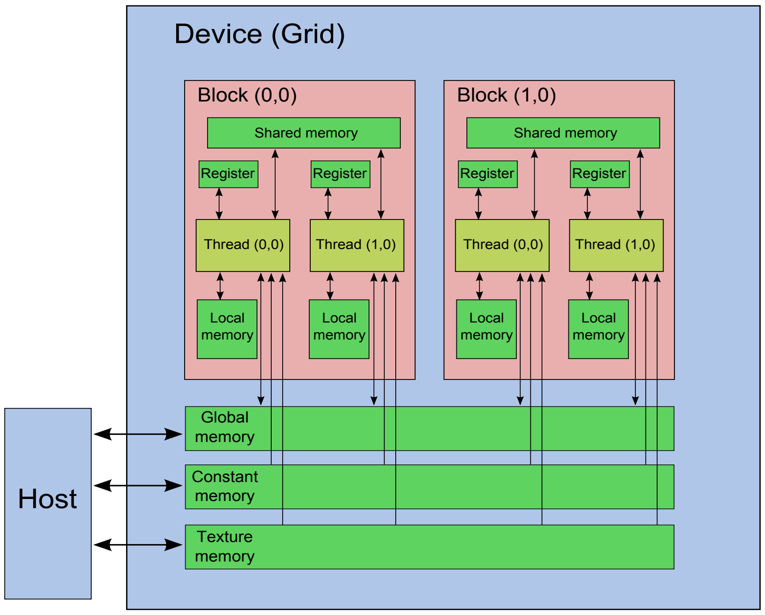


Figure 3.3: CUDA programming model

Chapter 4

Parallelism in JPEG-LS

4.1 Data Dependencies in JPEG-LS

The nature of the applied context encoding in the JPEG-LS algorithm suggests a tendency of parallelization as a result of the independent encoding routine exhibited by each context. However, a careful inspection of the encoder data-flow diagram in Figure 4.1 shows that a data dependency loop is encountered each time the modeler determines that two pixels, scanned in a raster order, belong to the same quantized context Q. In such a case, and as was pointed out in Chapter 2, the parameter *C[Q]*, being previously updated by the first pixel, is used to cancel the bias of the prediction error of the second pixel. This means that both the calculation of the prediction error and the update of the context's variables are data dependent, and thus a sequential processing is forced for pixels having the same context.
Encoding in the run mode is also considered to be a second factor which weakens the possibilities of having a fully parallel JPEG-LS. This mode of encoding is purely sequential by definition since the run lengths are only determined by serially counting the scanned pixels till the end of the current image line is reached or the interruption pixel is encountered.
For the mentioned reasons, a fully parallelized software implementation of JPEG-LS encoder, which is compliant to the established standard, has never been reported. However, other different implementations achieving a speedup in the algorithm execution time have been proposed in the literature as will be discussed in the next section.

4.2 Previous Approaches of Parallel LOCO-I/JPEG-LS

An early hardware implementation based on FPGA was found in [6]. This modified implementation of the LOCO-I algorithm, referred to as FPGA-LOCO, was designed to lower the complexity of the encoder by ignoring the run mode. Also a different context window was used to model the pixels, which resulted in having a total of 767 contexts, against the 365 contexts of the standard. Although this implementation reported 15% better compression than the Rice algorithm, it has been considered to be incompatible with the LOCO-I standard.
A more recent implementation using VHDL and which is capable of standard-compliant encoding and decoding was proposed in [7]. A limited parallelism was obtained by pipelining the computations that do not depend on the previous ones. For example, updating the statistical context parameters of the current pixel and retrieving the pixel values from the memory for the context of the next pixel. By this improvement, software simulations showed an increase

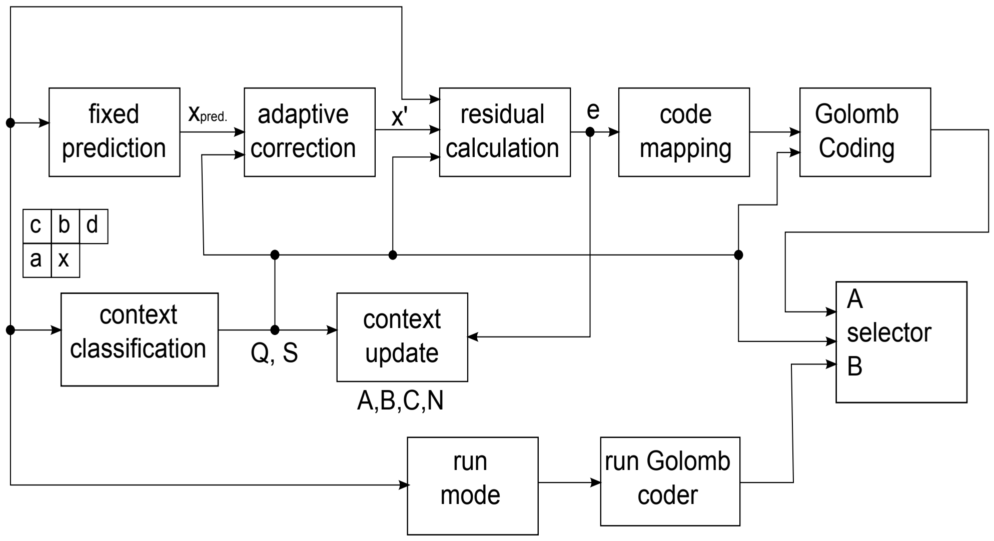


Figure 4.1: Data flow of JPEG-LS encoder

in performance of only 17% in average compared to the HP LOCO-I software implementation. The amount of on-chip memory used for the context table and for two image rows was found to consume 86% of the total area and was considered to be the slowest part of the design.

Another parallel pipelined version of a modified LOCO-I implementation was reported in [8]. The implementation, referred to as N2C-EX, consists of 3 pipelines: one handles the run mode, while the other two are concerned with processing two neighboring regular-mode pixels independently. In order to achieve parallelism, the context statistics for the two encoded neighboring pixels are replicated and stored in two separate memories. This doubling of the context set suggests a proportional increase of the required memory with increasing the number of parallel encoded pixels, and thus leading to scaling problems. The design has been reported to achieve an overall speedup of almost 2 (but not in all cases), and with a minimal decrease in compression ratios. The design however did not provide any improvements in run mode encoding which remained sequential.

Two fully pipelined hardware implementations which are compatible with the standared were proposed in [9] and [10]. The implementation in [9] used a double-frequency architecture to overcome the problem of context dependencies and thus avoid pipeline stalls. The implementation in [10], on the other hand, used a look-ahead technique for the computation of prediction residuals to solve the same problem. Both designs attained a higher throughput. However, still an output of one coded pixel is available every clock cycle.

The most recent publishe work is found in [11]. The proposed design achieves parallelism by delaying the context update procedure by n pixels, and thus being capable of encoding n pixels in parallel. The design however deviates from the standard by the fact that only one context update, based on an average error value, is performed for all same-context pixels involved in one parallel encoding operation. This means that the total number of updates made per context over the whole image is less than standard for $n>1$. Another deviation results from the order in which the pixels are processed. Instead of a raster scan order, n lines are processed in parallel and pixels within each line are processed sequentially. This causes a different context update order than the standard and also demands large bandwidth buffers for creating interleaved bitstreams. Encoding in run-mode remained unaffected because of the sequential line processing.

The loss in compression ratios, as a result of the relaxed context update, has been reported to be less than 0.1% (but not in all cases).

4.3 Context Round Encoding Method

As it can be concluded from the previous discussions, parallelism in JPEG-LS is better exploited as a case of data parallelism rather than task parallelism. This is due to the fact that the number of encoding tasks performed on each pixel is not large enough to produce a significant speedup upon parallelization when compared to the number of pixels that need to be processed. Moreover, the encoding tasks in JPEG-LS are greatly dependent on each other and eliminating such dependencies is almost impossible. For example, The Golomb coding stage needs the Rice mapped error, which in turn requires the calculation of the prediction residual, based on the output of the fixed predictor.

Discussions in the last section revealed that the work found in the literature has mainly approached parallelism in JPEG-LS in two different ways. Firstly, by creating an overlap in the processing of different pixels, as in [7], which did not provide a significant gain in the speedup because of the narrow overlap window size. Secondly, by breaking the feedback loop created by the context update procedure, as in [11], and consequently providing implementations which are not compatible with the standard.

In this work, parallelism is approached in a slightly different manner. We make use of the fact that pixels having different contexts undergo a totally independent coding scheme. We suggest that parallel encoding can be then achieved in terms of what is called context rounds. Each round contains a group of pixels which have different contexts, and thus ready to be processed in parallel. Rounds are encoded sequentially, however the pixels within each round are encoded in parallel. After processing each round, the context update procedure is performed for all pixels in that round. Pixels of the same contexts, following in the next round, can then use the updated context variables from previous rounds. In order to ensure that the context update process is performed following the same order specified by the standard, pixels of the same context should be assigned to the encoding rounds in such a manner that preserves the raster scan order. This means that moving in a raster scan fashion and assuming that rounds are encoded in an ascending order, a pixel with context Qi can be only assigned to a round n if all Qi pixels, occurring first in the image, have been assigned among the previous n-1 rounds. Figure 4.2 illustrates this idea. Although applying such a round assignment routine ensures compatibility with the standard in terms of the number and the order of updates performed per context, the encoding of pixels with different contexts is however done in an out of order fashion. This consequently demands a reordering of the output bit stream after the processing has completed, so that a full compliance to the JPEG-LS standard is achieved.

The concept of round encoding can be best understood from the context distribution of pixels within the image. Figures 4.3 and 4.4 show the distributions of pixels among the 364 contexts used in the regular mode encoding of the JPEG-LS standard. Six high-resolution gray 8-bit images [12] are used for investigation. As it can be observed from the figures, some images exhibit a relatively uniform context distribution as in Figure 4.3, while others have clusters of pixels concentrated among a few contexts, Figure 4.4. The horizontal lines shown in the figures represent different rounds. All contexts intersecting with the same horizontal line can have their corresponding pixels encoded in the same round and thus in parallel. The encoding process can be visualized as a line upwardly scanning the context distribution graph. As we proceed with the

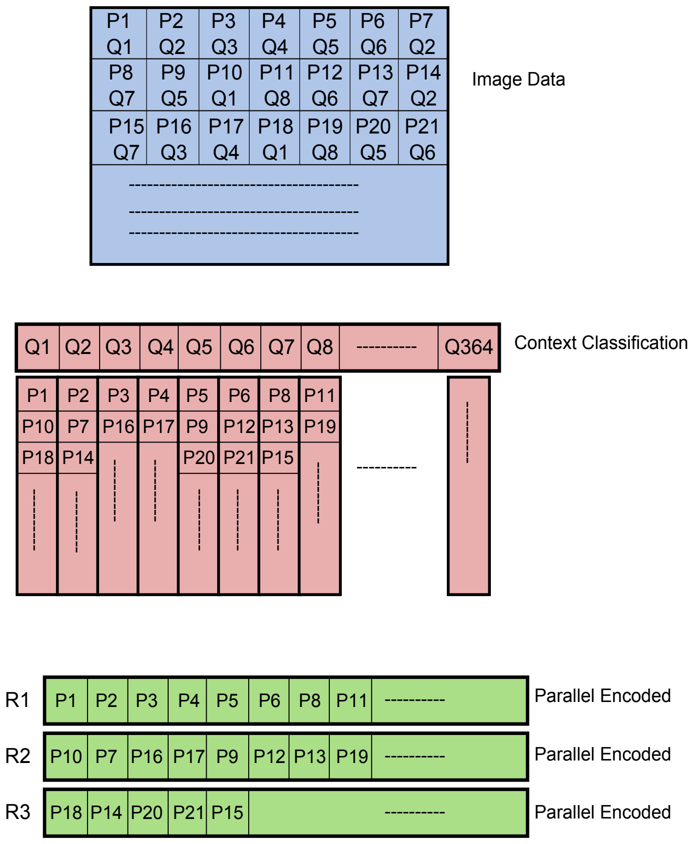


Image Data
P1 Q1, P2 Q2, P3 Q3, P4 Q4, P5 Q5, P6 Q6, P7 Q2
P8 Q7, P9 Q5, P10 Q1, P11 Q8, P12 Q6, P13 Q7, P14 Q2
P15 Q7, P16 Q3, P17 Q4, P18 Q1, P19 Q8, P20 Q5, P21 Q6
Context Classification
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 ---------- Q364
P1 P2 P3 P4 P5 P6 P8 P11
P10 P7 P16 P17 P9 P12 P13 P19
P18 P14 P20 P21 P15
R1 P1 P2 P3 P4 P5 P6 P8 P11 ---------- Parallel Encoded
R2 P10 P7 P16 P17 P9 P12 P13 P19 ---------- Parallel Encoded
R3 P18 P14 P20 P21 P15 ---------- Parallel Encoded

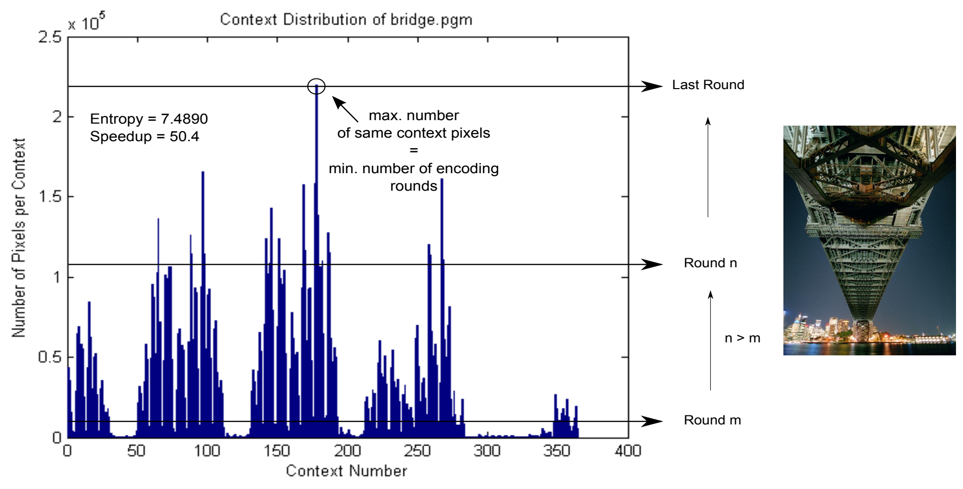
Figure 4.2: Round Encoding Method ('P' stands for pixels, 'Q' for Contexts)

encoding rounds, the number of pixels that can be processed in parallel decreases until the last round is reached having in most cases one pixel available for encoding. Assuming a maximum number of 364 pixels being encoded in each round, the minimum number of rounds required to encode the whole image is equal to the highest number of same-context pixels within the image. This is shown in the figures by the highest peak in the distribution. Alternative designs for assigning less number of pixels per round are discussed Section 4.4.

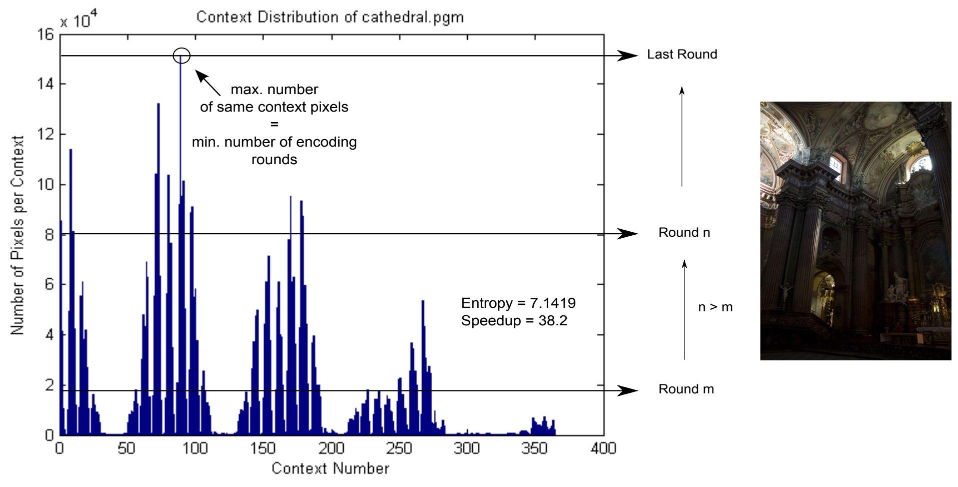
The degree of parallelism attained by the round encoding method is greatly dependent on the context distribution of the image, and thus is not guaranteed. Images with a more uniform context distribution are expected to achieve a higher parallelism than their counterparts with less uniform distributions. A maximum theoretical increase in the performance of regular mode encoding can be calculated by dividing the minimum number of required encoding rounds by the total number of regular mode pixels as described by Equation 4.1. This maximum theoretical speedup is computed for each image in Figures 4.3 and 4.4.

$$\text{Max. speedup in regular mode} = \frac{\text{Total number of regular mode pixels}}{\text{Min. number of rounds}} \tag{4.1}$$

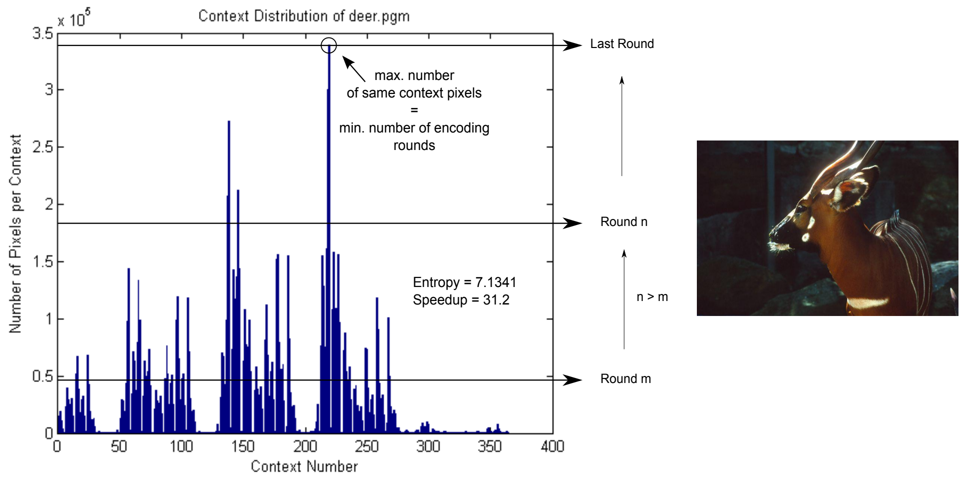
In order to explore the relation between the context distribution and the expected maximum speedup attained by the round encoding method, we regard the context distribution as a histogram and calculate the corresponding entropy according to Equation 4.2, where P represents the probability of occurrence of context Q within the image. The entropy shall give an indica-



(a) bridge.pgm-2749x4049

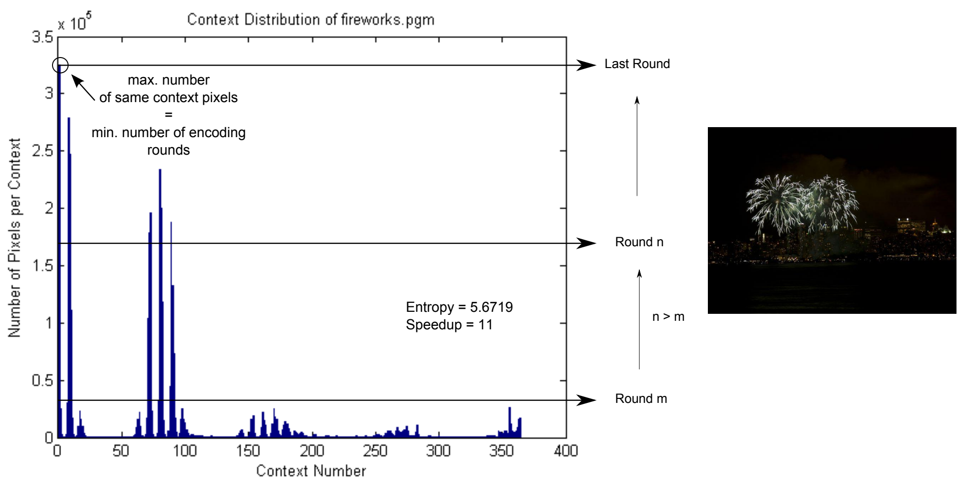


(b) cathedral.pgm-2000x3008

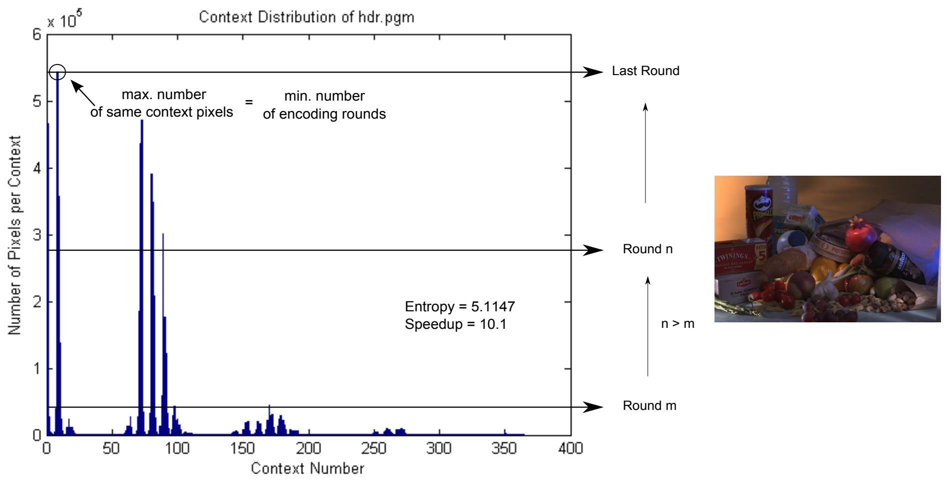


(c) deer.pgm-4043x2641

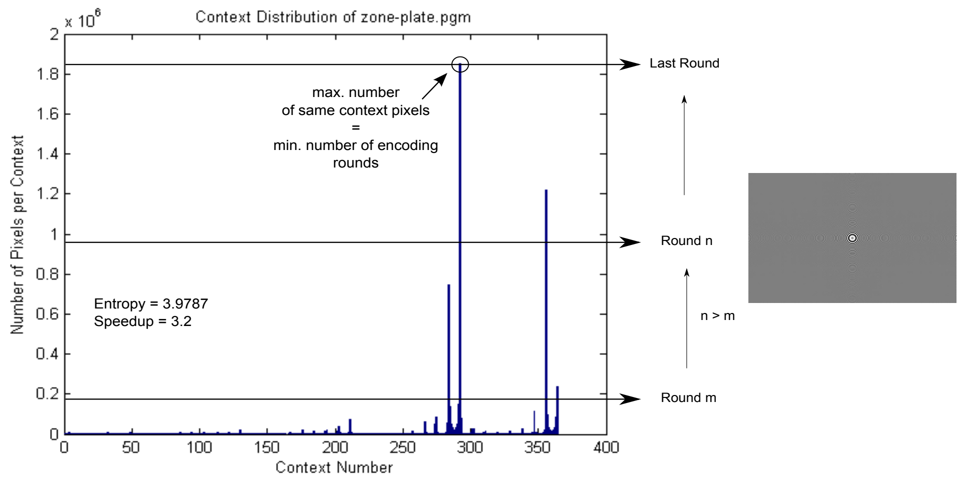
Figure 4.3: Context distribution of images with high distribution entropy



(a) fireworks.pgm-3136x2352



(b) hdr.pgm-3072x2048



(c) zone-plate.pgm-3000x2000

Figure 4.4: Context distribution of images with low distribution entropy

tion about the uniformity of pixels distribution among the contexts, and thus an estimation of the expected parallelism. Figure 4.5 plots the computed entropies of the investigated images versus the corresponding maximum theoretical speedup. As it can be observed, a higher entropy indicates a higher speedup, since the maximum entropy is when all the contexts have the same cardinality. This relation can be seen in Figure 4.5.

$$Entropy = -\sum_{Q=1}^{364} P * Log_2(P) \tag{4.2}$$

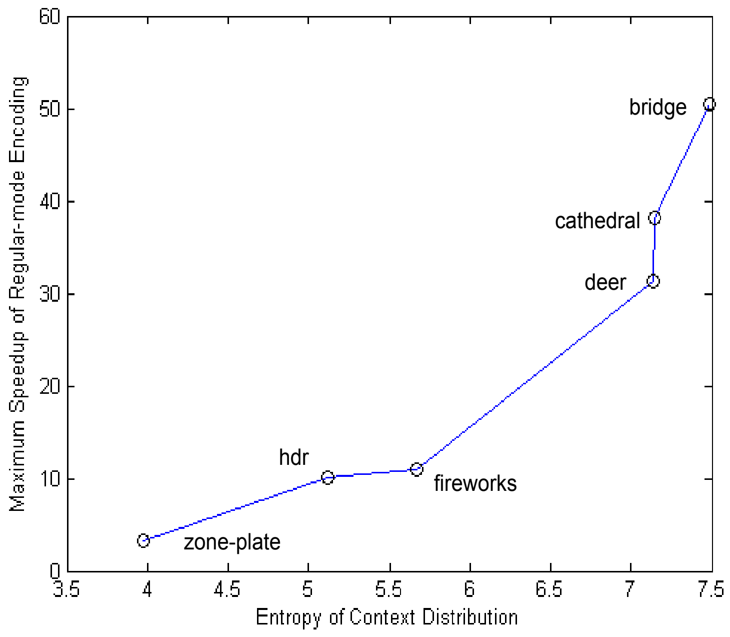


Figure 4.5: Relation between the context distribution entropy of an image and the maximum expected speedup in regular mode encoding

4.4 Context Round Assignements

4.4.1 Maximum Load Assignment

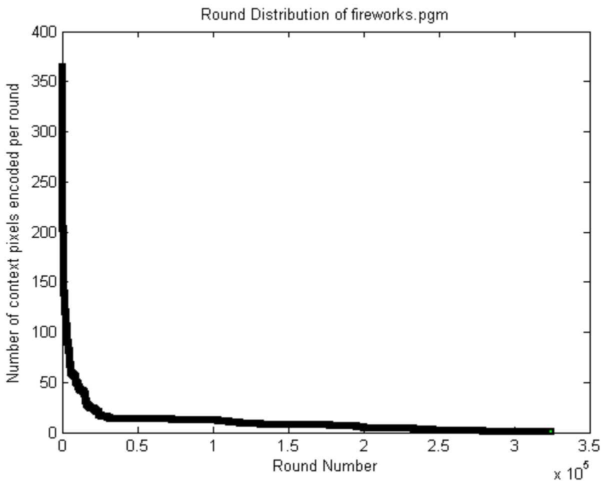
The first proposed round assignment scheme suggests maximizing the number of parallel encoded context pixels per round so as to achieve the least total amount of encoding rounds, and thus the lowest possible processing time. Figures 4.6 and 4.7 plot the results of software simulations performed on the same set of previously investigated images to examine the number of processed context pixels assigned to each round. In this scheme, rounds were allowed to have a maximum number of 364 context pixels encoded in parallel. As it can observed from the graphs, the distribution of pixels among the rounds suffers from a large load imbalancing. some images like *zone-plate*, *fireworks* and *hdr*, basically the ones with lower context distribution entropy, have their first few rounds fully loaded with pixels, while the majority of the remaining rounds have less than 20 context pixels assigned to them. The high entropy images on the other hand, Figure 4.7, show a continuously changing pixels load among the intermediate rounds. This is because maximizing the round encoding capacity for these images results in having the contexts running out of pixels at different round numbers, and thus a rather smooth change of workload is experienced along the whole coarse of encoding.

4.4.2 Round Robin Assignment

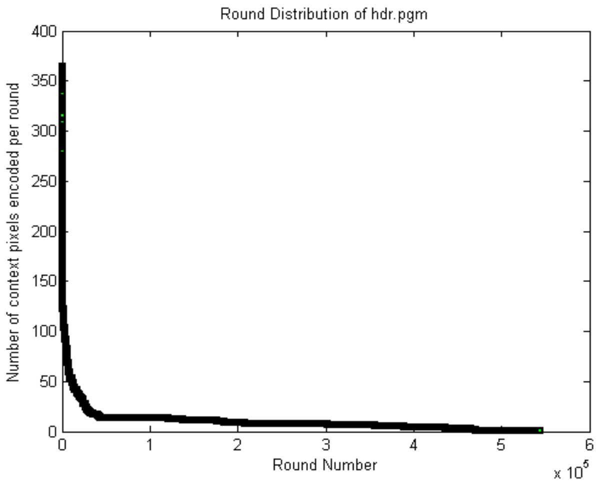
In order to achieve a rather constant workload distribution among the largest amount of encoding rounds, a second assignement scheme is suggested where the maximum number of parallel encoded pixels is limited to a value below 364 pixels per round. This means that in each round, an *n* number of contexts are chosen for encoding out of the 364 available contexts. The fact that *n* is less than 364 gives a higher degree of freedome in choosing the encoded contexts in each round. In this proposed assignment scheme, the contexts are assigned to the rounds in a round robin fashion. Figure 4.8 illustrates the assignment procedure. The first round is assigned pixels from the first *n* contexts, the second round follows with pixels from the subsequent *n* contexts and so on till context number 364 is reached, then the assignment proceeds again starting from context number 1.
Figure 4.9 shows the simulation results only for the images with high context distribution entropy, as these are the ones which show a high tendency for parallelization and thus worth consideration. In our simulations, the maximum number of parallel encoded pixels per round was experimentally determined for each image individually. Section 4.4.4 provides more details on estimating this parameter, referred to as the round capacity parameter. As it can be observed from the figure, a more balanced round distribution is obtained in which an average of 56% of the encoding rounds are equally loaded with pixels for the three investigated images. The last remaining 44% of rounds still exhibit a changing workload, which appears in the form of a short tail in the distribution graph. This proposed round assignment scheme also suffers from an increase in the total amount of rounds required for encoding the whole image, which is a consequence of limiting the round encoding capacity. An increase of 51%, 55% and 59% in the total number of rounds has been measured for *bridge*, *cathedral* and *deer* respectively.

4.4.3 Context Priority Assignment

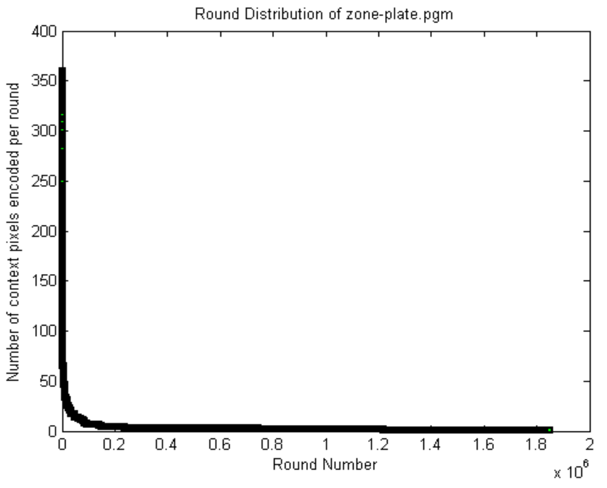
It should be noted that the used round robin context assignment, discussed in the last section, gives no encoding priority for contexts with large pixel count. Therefore, eventhough pixels in such contexts can undergo consecutive round assignments, the round robin scheduler might delay their encoding to later rounds and replace them with contexts with lower number of pixels, and thus of lower priority. This leads to first, an increase in the total number of rounds required for the whole encoding process and second, having the last encoding rounds only occupied by those few contexts with the large pixel count. This consequently leads to the observed tail problem.
A last proposed context assignment scheme, referred to as context priority assignment, is discussed in this section. In this scheme, contexts are prioritized based on their corresponding pixel count and are then encoded in order of their assigned priorities. This means that in each round a pixel of context *Qi* is allowed to be encoded only when all the pixels of the *n* higher priority contexts have been already encoded in previous rounds (*n* denotes the allowed number of parallel encoded pixels per round). This assignment scheme shall then minimize the added delay, in



(a) fireworks.pgm-3136x2352

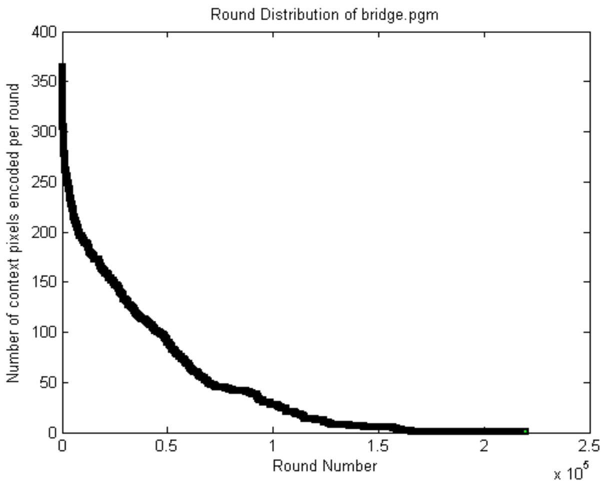


(b) hdr.pgm-3072x2048

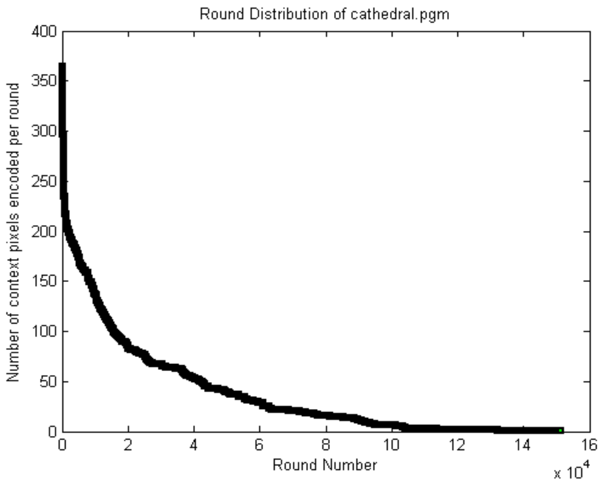


(c) zone-plate.pgm-3000x2000

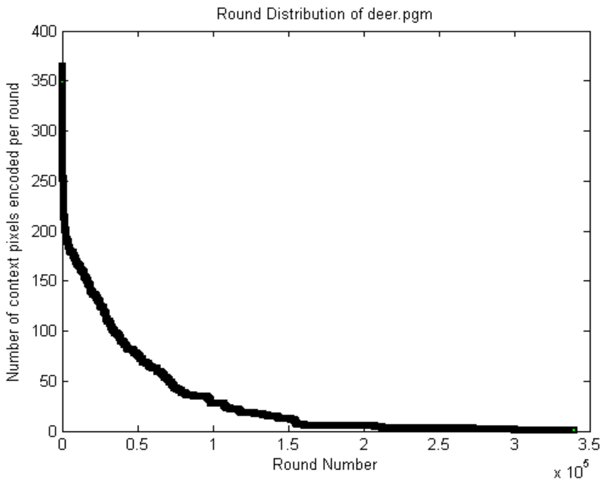
Figure 4.6: Round distribution of low-entropy images with maximum load assignment scheme



(a) bridge.pgm-2749x4049



(b) cathedral.pgm-2000x3008



(c) deer.pgm-4043x2641

Figure 4.7: Round distribution of high-entropy images with maximum load assignment scheme

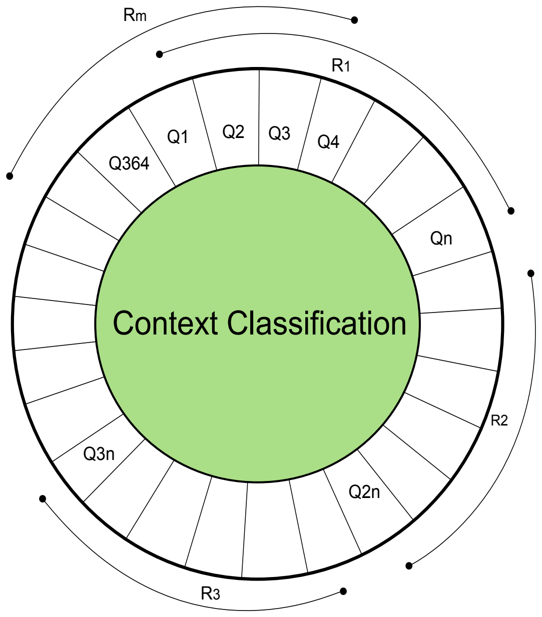
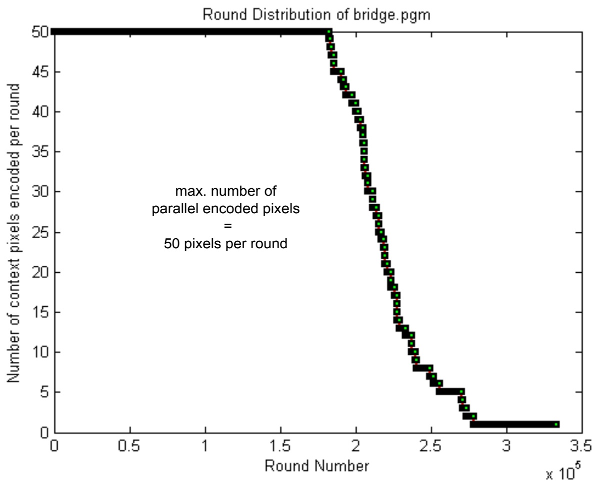


Figure 4.8: Round robin assignment scheme

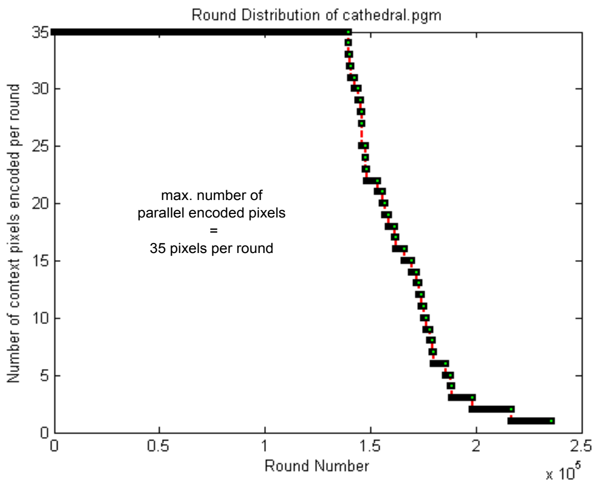
terms of number of rounds, that results from limiting the round encoding capacity to a value less than 364 context pixels per round.
Figure 4.10 illustrates how the context priority assignment scheme is implemented. First, pixels are classified according to their contexts. A sorting is then applied to arrange the 364 contexts in an increasing order of their pixel count. Rounds are then only assigned pixels of the first n contexts. When one of these contexts has all of its pixels encoded, a swapping is performed, where the next high priority context replaces the empty one in position and therefore pixels of that context are allowed to be encoded. This process continues on until the end of the context list is reached and all pixels have been encoded.
Figure 4.11 shows the results of the context priority assignment scheme. As it can be seen from the figure, almost 99.5% of the encoding rounds have a constant number of pixels to encode. Moreover, the tail problem, experienced by the round robin assignment, has now disappeared and a rather abrupt change of workload takes place at the very end of the encoding process. The increase in the total number of encoding rounds did not exceed 9% with respect to the one obtained by the maximum load scheme. Because of the superior obtained results, context priority round assignment scheme will be chosen for the implementation of parallel JPEG-LS, which will be discussed in the next chapter.

4.4.4 Estimating the Round Encoding Capacity

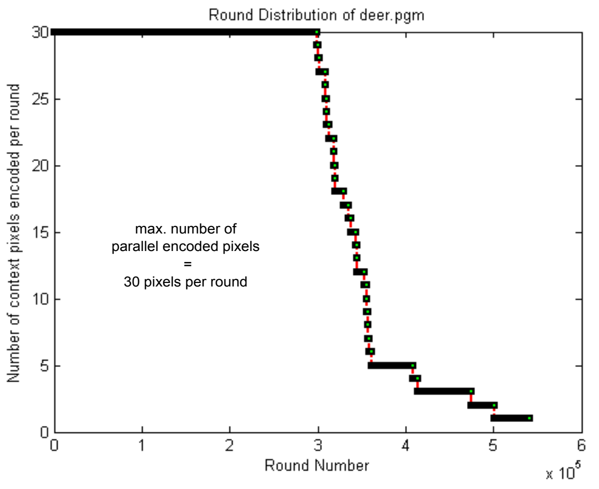
In estimating the number of parallel encoded pixels per round, two factors are to be considered. First, the total number of encoding rounds required to encode the whole image. Second, the desired percentage of rounds having the same pixels load. Increasing the round encoding capacity (number of parallel encoded pixels) results in lowering the total amount of encoding rounds. However, it also decreases the percentage of equally loaded rounds. The first boundary case is defined by setting the round capacity to 364 context pixels, which results in the minimum amount of encoding rounds and the worst load distribution. The second boundary case, on the



(a) bridge.pgm-2749x4049



(b) cathedral.pgm-2000x3008



(c) deer.pgm-4043x2641

Figure 4.9: Round distribution of images with Round robin assignment scheme

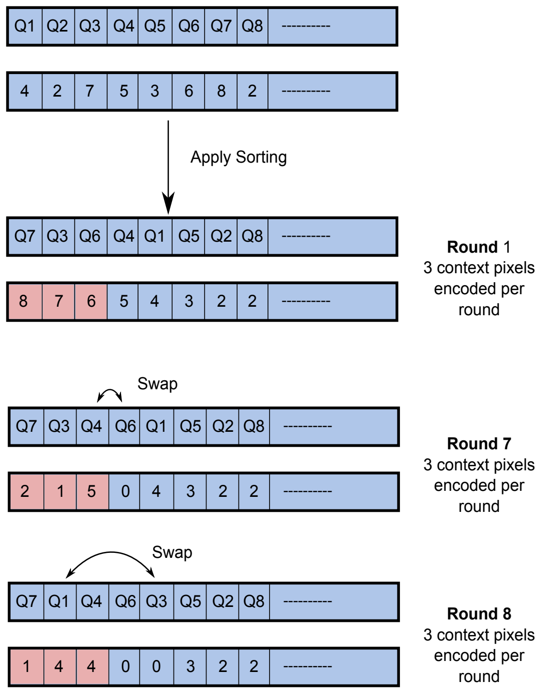
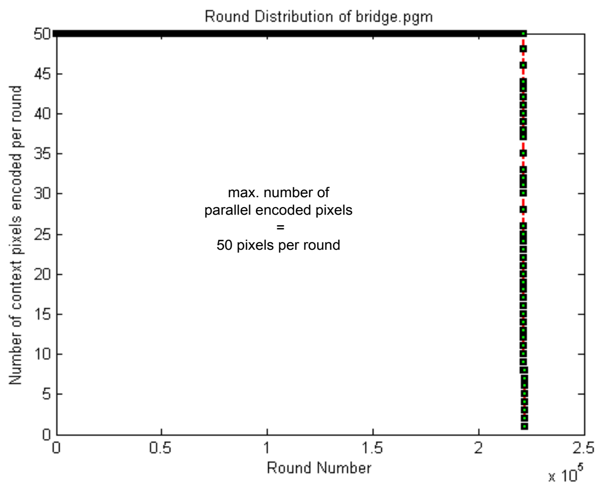

Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 ---------
4 2 7 5 3 6 8 2 ---------
Apply Sorting
Q7 Q3 Q6 Q4 Q1 Q5 Q2 Q8 ---------
8 7 6 5 4 3 2 2 ---------
Round 1
3 context pixels encoded per round
Swap
Q7 Q3 Q4 Q6 Q1 Q5 Q2 Q8 ---------
2 1 5 0 4 3 2 2 ---------
Round 7
3 context pixels encoded per round
Swap
Q7 Q1 Q4 Q6 Q3 Q5 Q2 Q8 ---------
1 4 4 0 0 3 2 2 ---------
Round 8
3 context pixels encoded per round

Figure 4.10: Context priority assignment scheme

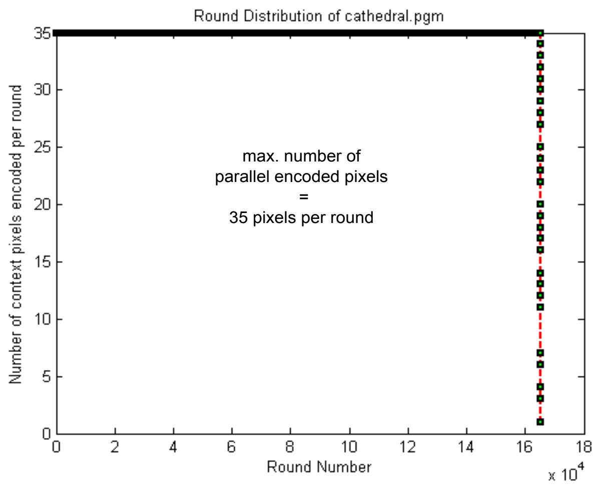
other hand, is defined by setting the round capacity to only 1 context pixel, and this results in having the largest amount of encoding rounds and the best load distribution (this is a case of no parallelism, with one encoded pixel per round). In between these two boundary cases, an optimization can be applied to find the optimum value of the round capacity parameter, which constitutes a trade-off between the two mentioned factors. In our simulations, This parameter was experimentally determined for each individual image to achieve an increase of no more than 10% in the total number of encoding rounds, and a 99% of equally loaded rounds (this applies to the context priority assignment scheme).

4.5 Adjusting Run Mode Encoding

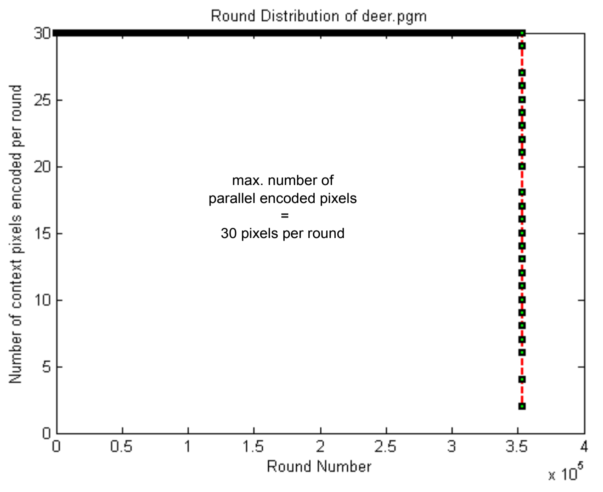
As far as the run mode has been concerned, the encoding of the run interrupt sample has always directly followed the run-length coding, and thus in a sequential manner. As it has been discussed in Chapter 2, encoding run interrupt samples resembles that of the regular mode ones, with the contexts 365 and 366 allocated for run interrupt encoding. We have examined the occurrences of run interrupt samples by running software simulations on the same panel of images used previously. It has been verified that almost 99% of the identified run segments within the image are terminated by the interruption sample. Moreover, it has been also found that the number of interrupt pixels within the image (contexts 365 and 366) is quite comparable to that of pixels in other regular mode contexts. Therefore, based on the previously stated facts, we suggest that a limited increase in parallelism can be achieved by treating the interrupt samples as regular mode samples, and by encoding them with the proposed context round encoding method. This means that the interrupt samples shall be assigned to rounds in the same manner as with the regular mode samples. Therefore, the set of contexts that can be processed in parallel becomes



(a) bridge.pgm-2749x4049



(b) cathedral.pgm-2000x3008



(c) deer.pgm-4043x2641

Figure 4.11: Round distribution of images with context priority assignment scheme

366 contexts instead of 364. The applied modification to the run mode shall also reduce the size of the sequential part in the JPEG-LS algorithm.

Chapter 5

Design and Implementation of Parallel JPEG-LS

In the last chapter, the concept of round encoding has been introduced, where the potential of parallelism in the encoding using JPEG-LS has been analyzed for a collection of images. Three round assignment schemes were also proposed, with each scheme presenting a unique trade-off between the total number of required encoding rounds and the achieved level of workload balance among the rounds. This chapter presents both the design methodology and the implementation strategy of parallel JPEG-LS encoder, making use of the round encoding method as an approach to achieve parallelism. Further applied techniques for achieving parallelism will be also shortly discussed.
Referring to Chapter 3, a GPGPU is used as a target hardware architecture for the parallel implementation of JPEG-LS encoder, along with the CPU for the sequential part. A mapping of the different algorithmic blocks to the hardware will be investigated, and possible overhead issues, resulting from the proposed design, will be presented.

5.1 Design Methodology

Our implementation is divided into three distinct phases. Each phase is responsible for the execution of a group of encoding tasks, which include both the tasks defined by the standard and additional tasks required for the adaption to the round encoding scheme. Although the order of execution of some tasks or their application to certain pixels might be different from that in the reference implementation, the overall encoder design shall provide an output bit stream that is identical to that of the established standard, and therefore achieving the same compression ratios. Figure 5.1 shows the encoder data flow diagram of the proposed parallel JPEG-LS. The following sections discuss each of the encoding phases in detail.

5.1.1 Phase One : Pre-computation of Pixels

The first phase in the encoding process, as shown in Figure 5.1, is concerned with the context classification and the fixed prediction of pixels. The fact that any of these two encoding tasks exhibit data dependencies shall suggest parallelizing their execution so that they are applied on all the pixels of the image at the same time. This accordingly requires allocating this phase on the GPU, where each working thread becomes responsible for the execution of these two tasks

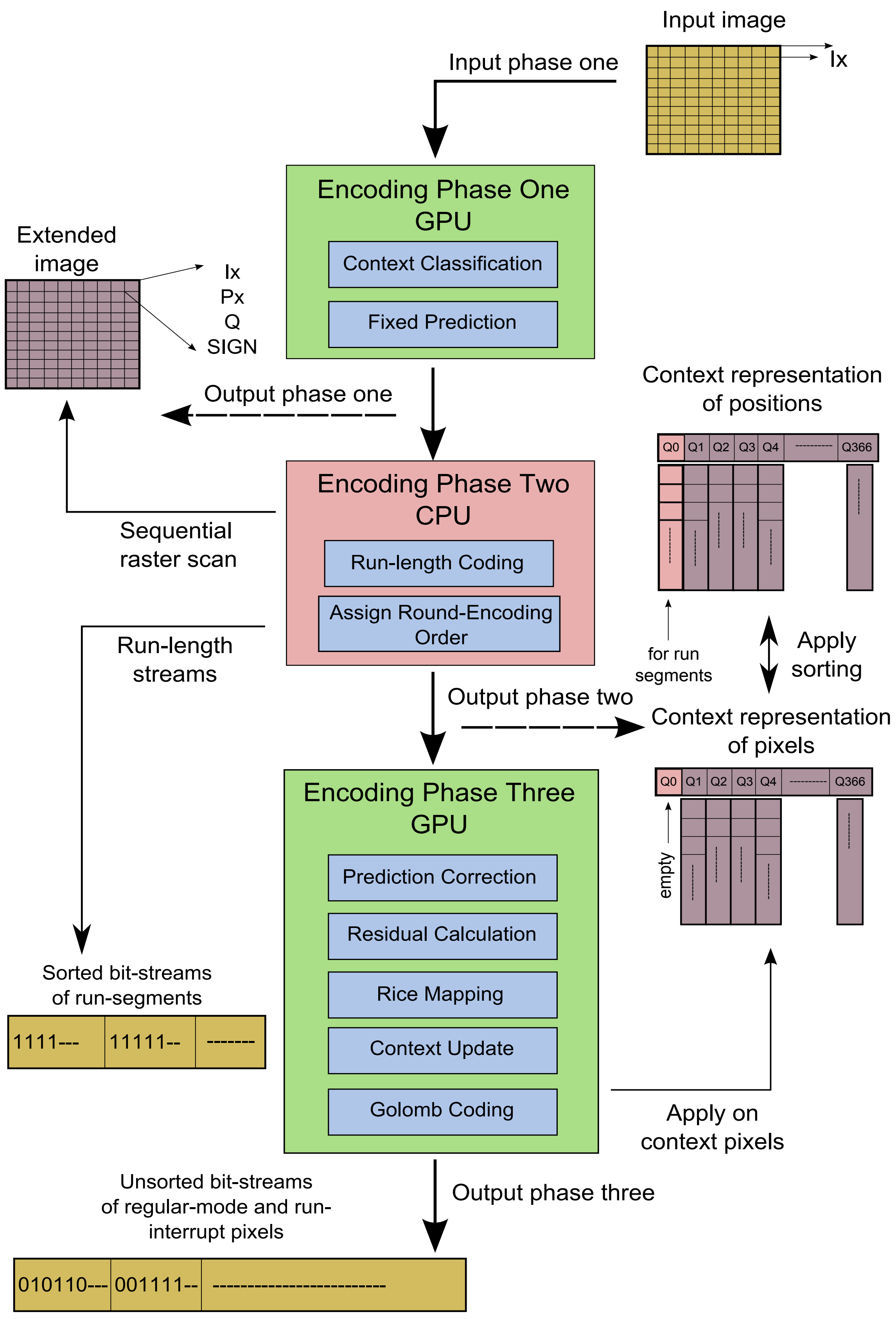


Figure 5.1: Blockdiagram of parallel JPEG-LS encoder

on one corresponding pixel. Although further parallelization can be still achieved by executing the two mentioned tasks concurrently within the encoding of one pixel, as they do not depend on each other. However, the fact that the architecture of the GPU is more suited to SIMD execution model, as was discussed in Section 3.4.1, suggests a sequential execution of the two tasks. This achieves a uniform task distribution among the working threads of the GPU and thus meeting the hardware execution model.
The main goal of this encoding phase is to compute those parameters, which are needed for the pixel's further processing, and which are also dependent on the values of the surrounding pixels. Such parameters include the context value Q, the prediction Px and the sign $SIGN$ of each pixel. After the completion of this phase, each pixel (except for the run-mode pixels) can be then regarded as an independent structure, on which the rest of the encoding tasks can be applied without the need to access its neighborhood pixels [1]. This shall serve as a first preparation step for the round encoding scheme, which will alter the pixel's local properties. Figure 5.1 shows the output of phase one. Basically the same image is available, but with each pixel locally equipped with the computed values needed in further stages.
Two main drawbacks exist in this encoding phase:
First, although context classification and fixed prediction are considered to be regular-mode encoding tasks, they will be still applied on all the pixels of the image including the run-mode pixels as well. This adds an extra amount of computation, which is not needed in the rest of the encoding process. The added amount of work shall however be non critical, since the execution is done in parallel, and there are plenty of resources available on the GPU in terms of parallel working threads[2]. Also trying to solve this problem by adding an extra task for extracting the run-mode pixels is considered to be of a high computational cost, and will violate with the GPU SIMD execution model. Therefore, no distinction of run-mode pixels is made at this stage of encoding, and they are treated as regular-mode ones. Only the start of run-segments is identified by those pixels having a context number $Q = 0$.
The second drawback of this encoding phase is represented by the added data overhead. As it was previously mentioned, each pixel becomes a structure which holds not only the value of the individual pixel, but also other values needed for further processing (Q, Px and $SIGN$). This may subsequently result in a higher data transfer overhead between the GPU and the CPU and will demand higher memory bandwidth. Again limitations depend on the size of the encoded image.

5.1.2 Phase Two : The Sequential Raster Scan

After receiving the extended image (the original image with the added values to each pixel) from the first encoding phase, the second encoding phase is ready to execute its tasks. At this stage of encoding, as shown in Figure 5.1, two main tasks take place. First, encoding run-lengths of run-mode pixels. Second, assigning the round encoding order to regular-mode and run-interrupt pixels according to the context priority assignment scheme discussed in Section 4.4. This involves arranging same-context pixels according to their raster scan order and sorting contexts based on their corresponding pixel count.
Because of the serial nature of the performed tasks, this phase of encoding is mapped to the CPU, where a sequential raster scan of the image (output of phase one) is performed. During the

[1]It should be noted that the encoding dependency of same context pixels still exists, and is to be solved by the round encoding concept

[2]Problems may only appear with very high-resolution images with sizes exceeding the available number of parallel working threads on a GPU

scanning process, the start of run-segments is identified by those pixels which have their *Q* value equal to zero. The run segment-length is then obtained by sequentially counting the following run piexls until the end of the image line is reached, or the interruption sample is encountered. The run-length is encoded according to the standard, as was described in Section 2.2.5, and the produced bit stream is then stored as shown in Figure 5.1. In case of an interrupted run-segment, the values of *Px* and *Q* of the run-interrupt pixel are corrected and the *RunIndex* parameter, needed for run interruption encoding[3], is stored.

During the sequential scanning, both regular-mode and run-interrupt pixels are stored according to their context numbers and raster scan order. After the scanning process has finished, a sorting of the 366 contexts is then applied, so that contexts are arranged in an increasing order of their corresponding pixel count. This phase of encoding is thus considered to be a transformation of the image representation from the original raster scanned image into a context based representation, on which the round encoding scheme can be applied in the following phase as shown in Figure 5.1. Although none of the pixels is literally assigned a round number, however their arrangement and the classification they are adjusted to in this phase implicitly dictate their order of encoding in the following phase, taking into account that the number of encoded pixels per round is known.

It should be also noted that since at this phase of encoding, the original image is disturbed in terms of the local properties of each pixel, it is quite essential to store the pixels' original positions while raster scanning the image, as this will be needed afterwards to arrange the output bit-stream in the same raster scan order. For regular-mode and run-interrupt pixels, the exact position in the image should be stored. For run-segment pixels on the other hand, only the position of the starting pixel in the segment is stored. This is basically because for run segments, an output of one codeword is produced for the whole segment, and thus we treat the it as one entity in our design. The positions can be either stored within the created pixel structure (as the case with the values *Q*, *Px* and *SIGN*), or they can be stored separately. The first case will result in an increased data overhead, since the size of the pixel structure will get bigger and this will lead to a slower data transfer rate, when the pixels are transferred to the GPU for encoding in the following phase. Also for run segments, no pixel structures are available, since these pixels are not assigned to contexts, and therefore there is no possibility of storing their positions according to the first method. The second method represents then a better choice, where the positions are separately stored in the same arrangement as their corresponding pixels, indicated in Figure 5.1, and they do not get transferred to the GPU in the following phase. The sorting should be however applied to the positions as well, so that a one-to-one mapping between the pixel and its position is available[4].

The main drawback of this sequential encoding phase is that its size and computational complexity define the lowest possible encoding time of the whole implementation, as described by Amdahl's law in Section 3.2. The fact that a sequential raster scan of the image is performed, makes the computational time of this phase a function of the image size. An added computational overhead is also present due to the applied context sorting scheme, which is not part of the standard, however it is needed for the application of the round encoding method in the fol-

[3]RunIndex is used to indicate the run-mode order required for run-mode encoding. more details are to be found in [4]

[4]It should be noted that for the context representation of positions, the context cell *Q*=0 holds the position of the starting pixel for each run segment. The corresponding cell in the context representation of pixels is however empty, since no pixel structures are created for run-segment pixels. The sorting should be therefore applied on the cells starting from *Q*=1 till *Q*=366. See Figure 5.1

lowing phase. This sorting scheme shall nevertheless be a non critical factor, since it is applied only once with an average complexity of O (366), and only one swap of each context is needed afterwards to keep the contexts sorted during encoding, as was described in Section 4.4.

5.1.3 Phase Three : Round Encoding of Pixels

After all the context pixels have been arranged, and have been extended by the values needed for the rest of the processing, the round encoding scheme is then applied, where five sequential tasks are performed on each pixel in each round. These are the tasks specified by the standard for regular-mode and run-interrupt encoding as indicated in Figure 5.1. Since this is a parallel working phase, it is mapped to the GPU, where a number *Rcap* of parallel working threads are launched for the encoding of *Rcap* context pixels per round , with *Rcap* being the round capacity parameter. In this phase, we make use of the context representation, obtained from the previous phase, so that the encoding threads are directly applied on the first *Rcap* contexts in the context list (as these are the contexts with higher pixel count and thus higher priority). Once one of the *Rcap* contexts has all of its pixels encoded, it gets replaced by the awaiting context with the next highest pixel count. Pixels of the replacing context are then allowed encoding. This implements the context priority assignment scheme discussed in Section 4.4
Beside the fact that the encoding of this phase is an out-of-order encoding, and that an extra re-ordering step is required afterwards to arrange the output bit-stream in the original raster scan order. Another drawback of this design block can be observed as follows. The output bit-stream generated from the Golomb coder for each context pixel is of a different length that depends on pixel-specific encoding parameters. The sequential nature of encoding in the original JPEG-LS standard allows for the bit-streams to be appended behind each other. However, this can not be achieved in the case of parallel encoding, since the encoder will not know where to place the output stream of pixel P_i, when previous pixel P_{i-1} is being encoded at the same time. This means that for each context pixel, a maximum size buffer shall be allocated for its output bit-stream, so that parallel encoding becomes feasible. This in turn results in a waste of memory space that might not be fully used.
Data transfer overhead between the CPU and the GPU might again constitute a bottleneck, where pixel structures are transferred to the GPU for encoding, and output bit-streams are transferred back to the CPU for final re-ordering. The size of the transfer overhead is expected to scale with the image size.

5.1.4 Re-ordering of Bit-streams

The final step in the proposed encoder design takes place on the CPU, in which the compressed bit-streams of run-segments, regular-mode pixels and run-interrupt pixels are arranged according to their original raster scan order. This is done by the help of the stored pixels' position information available from the second encoding phase. Since this re-ordering scheme is done sequentially, it also affects the minimum overall encoding time. After the re-ordering has completed, the bit-streams are written to the output file, which is then considered to be fully compatible with the output of the standard.

5.2 Implementation in C++/CUDA

A lossless mode codec implementation of the JPEG-LS standard in C language has been used as a reference implementation [13]. The code has been adapted to C++ language for the sequential parts running on the CPU, and the CUDA programming model was used for the parallel implementation on the GPU. The following sections discuss the software implementation of the proposed parallel JPEG-LS encoder design presented in Section 5.1

5.2.1 Data Structures

The following represents the used data structures in our implementation. Alternative choices for some structures are also discussed, with reasoning about the selected choices.

Context_Pixel Structure

This is the core structure representing regular-mode and run-interrupt pixels. It carries all the information these pixels need for their processing as shown in Listing 5.1.

Listing 5.1: Context_Pixel structure

```
struct Context_Pixel{
 int Ix;        \\pixel value
 int Q;         \\pixel context number
 int Px;        \\pixel prediction value
 int SIGN;      \\pixel sign value
 int xPos;      \\pixel x position in the image
 int yPos;      \\pixel y position in the image
 int RunIndex;  \\only used by run-interrupt pixels
}
```

Due to the large size of the *Context_Pixel* structure, occupying 28 bytes of memory, optimization has been made to reduce the size of this structure, while at the same time maintaining the required information. First, we make use of the fact that the positions of the pixels will not be needed in the third encoding phase, and thus we eliminate the position parameters from the *Context_Pixel* structure. We store them separately, as was discussed in Section 5.1.2. Second, since our implementation is concerned with grey images with pixels having a maximum value of 255, we can use a **short** representation instead of **int** for the *Ix* and *Px* values. The same applies to the context number *Q*, which has a maximum value of 366, and *RunIndex* which has a maximum value of 30. A final reduction in the size can be achieved by eliminating the *SIGN* parameter as well, and allowing the context number *Q* to have either positive or negative values indicating the corresponding sign of the pixel. Listing 5.2 shows the modified *Context_Pixel* structure with a reduced size of only 8 bytes. This shall lower the data transfer overhead between the CPU and the GPU, and shall help in achieving coalescence in device memory access on the GPU as was discussed in Section 3.4.1.

Listing 5.2: Modified Context_Pixel structure

```
struct Context_Pixel{
 short Ix;       \\pixel value
 short Q;        \\pixel context number (+/-)
 short Px;       \\pixel prediction value
 short RIndex;   \\only used by run-interrupt pixels
}
```

ContextPixelImageArray

This represents the two-dimensional array structure of the original image, but with all the pixels adjusted to the Context_Pixel format. This shall be the output of the first encoding phase.

ContextPixelArray

This is the context based representation of the image as an array of pointers, with each cell pointing to another array containing the *Context_Pixel* structures of the corresponding context. This shall be the output of the second encoding phase.
An alternative option for the context representation of the image was to use a linked-list data structure instead of an array. This offers the advantage of dynamically expanding the size of the context lists as needed, which in fact meets our regarded case since the number of pixels per context is only known at run time. The drawbacks of using linked-lists are however the large space they occupy, and the time they need for the allocation and de-allocation of memory segments resulting in an increase of the overall encoding time. For this reason, an array structure has been used instead, and we overcame the problem of the unknown context sizes by creating another array structure, *ContextCountArray*, which holds the corresponding pixel count per context. The added structure shall be created in the second encoding phase by performing an additional raster scan[5] prior to the originally performed raster scan discussed in Section 5.1.2. During this additional raster scan, the size of each context is calculated by adding the pixels with the same context number *Q* and storing their corresponding count in *ContextCountArray[Q]*. After finishing this step, the *ContextPixelArray* can be then initialized with the context sizes found in the *ContextCountArray*. It should be however noted that the calculated sizes of some contexts will be of an upper-limit value since the run-mode pixels are being accounted for in the context classification process performed in phase one. This shall be however corrected during the second raster scan of encoding phase two, as will be shown in Section 5.2.3.

ContextCountArray

The *ContextCountArray* holds the upper-limited sizes of each of the 366 contexts. It gets created in the initial raster scan of the second encoding phase, and its values are corrected during the second raster scan, when the number of run-mode pixels are known.

Pixel_Position Structure

This structure represents the *x* and *y* coordinates of pixels in the original image. As it was previously mentioned, for regular-mode and run-interrupt pixels, we store the pixel's exact position in

[5]Although an added raster scan worsen the performance of the second encoding phase, it is still considered to be much more efficient than working with linked lists. Also, in the added raster scan, only a counting operation is performed, which shall not require a lot of time

the image, while for run-segment pixels, only the position of the starting pixel in the segment is stored. Another variable is also added to this structure, which indicates the number of pixels the position variables are pointing to. In case of regular-mode and run-interrupt pixels, this variable will have a value of 1 pixel, while in case of run-segments it will have a value equivalent to the run-segment length. The *Pixel_Position* structure is indicated in Listing 5.3.

Listing 5.3: Pixel_Position structure

```
struct Pixel_Position{
 int xPos;      \\x-coordinate of the pixel
 int yPos;      \\y-coordinate of the pixel
 int pixelCount \\number of pixels this position indicates
}
```

PixelPositionArray

The *PixelPositionArray* has the exact same structure as the *ContextPixelArray*, holding the position properties of pixels in terms of the introduced *Pixel_Position* structure . The elements of this array follow the same arrangement as their corresponding pixels in the *ContextPixelArray*, so that each output bit-stream can be mapped to its equivalent position in the original image at the end of the encoding process.

ContextBitStreamArray

The *BitStreamArray* holds the compressed bit-streams of regular-mode and run-interrupt pixels. Therefore, it has a size equal to the total number of context pixels found in the image (the image without the run-segment pixels). The elements of this array are pointers, pointing to character arrays which act as buffers holding the bit-stream of each pixel. As was discussed in section 5.1.3, the size of the bit-stream buffer is set to a maximum size ensuring that the longest output bit-stream will fit inside. In our implementation, the size is set to 50 characters.

RunBitStreamArray

The *RunBitStreamArray* has the same structure as the *ContextBitStreamArray*, except that it has a size equal to the total number of identified run segments in the image. It holds the compressed bit-streams of run segments.

5.2.2 Implementation of Encoding Phase One

The encoding of phase one starts after the whole image has been read and stored in memory, and after the initialization of the non defined samples of the causal template has been made [4]. The image is then transferred to the global memory of the GPU in preparation for its parallel processing. We use a two-dimensional array representation of the input image on the global memory, in order to be able to easily locate the context samples of each pixel. Blocks of 16x16 pixels are processed in parallel by GPU thread blocks of size 16x16 threads. This is shown in Figure 5.2. Since each pixel of the image is accessed more than once as a part of the context of different pixels, we make use of the fast access shared memory on the GPU. A two dimensional array of size 18x17 pixels is allocated on the shared memory for each block of threads in order to hold the pixels encoded by that block. Each of the 256 threads inside a block copy the corresponding 256 pixels, which are to be encoded, from the global memory to the shared memory. For the pixels residing on the encoded block boundary, the complete set of context samples is not available within the copied 256 pixels. Therefore, the corresponding threads, responsible for the encoding of these boundary pixels, perform extra copying of the missing context samples to the shared memory as well (Figure 5.2). For this reason, the size of the allocated array on the shared memory is set to 18x17 pixels, so that the context samples of the block boundary pixels can be stored. Listing 5.4 shows a part of the code implementation of encoding phase one. A pixel structure is created for each pixel in the image, in which the pixel's value *Ix*, context number with the sign *+/- Q*, and prediction *Px* are stored. The pixel structures are also written to the shared memory (16x16 pixels in size), and they get transferred to the global memory at the end of the processing, Again a two-dimensional array is used for the representation of the output image in the global memory, in which the pixel structures are arranged in the same raster scan order as the original image.

Listing 5.4: Implementation of encoding phase one

```
EncodingPhaseOne{

 struct Context_Pixel pix;
 \\get context samples Ra,Rb,Rc,Rd;
 pix.Ix = SMem_input[tx+1][ty+1];
 pix.Q = context_classification();
 pix.Px = edge_prediction();
 if(SIGN == 1)
   pix.Q = - pix.Q;
 SMem_output[tx][ty] = pix;
 __syncthreads();
}
```

5.2.3 Implementation of Encoding Phase Two

Listings 5.5-5.9 show parts of the code implementation of the second encoding phase. As it has been discussed earlier, this phase implements a serial scanning of the output image of phase one, where the *ContextPixelArray* and *PixelPositionArray* are filled with context pixels and their positions respectively. The contexts' sizes in both arrays are determined by the upper-limited values found in *ContextCountArray*, and which are obtained from the initial raster scan discussed in Section 5.2.1. For both contexts 365 and 366, their sizes are set to the number of identified run-segments (number of pixels with context value $Q = 0$ found in *ContextCountArray*[0]). This is also an upper-limited value, assuming that all the run-segments, which were identified in phase one, are terminated by the interrupt sample.
All the identified run-segments are encoded sequentially and their compressed bit-streams are stored in *RunBitStreamArray*. For interrupted run-segments, the values of the run-interrupt pixels are corrected according to Listing 5.6, where they get assigned their correct context numbers and prediction values, and also have their *RunIndex* parameter set. A correction of the values in *ContextCountArray* is also performed by subtracting the count of run-mode pixels (run-segment

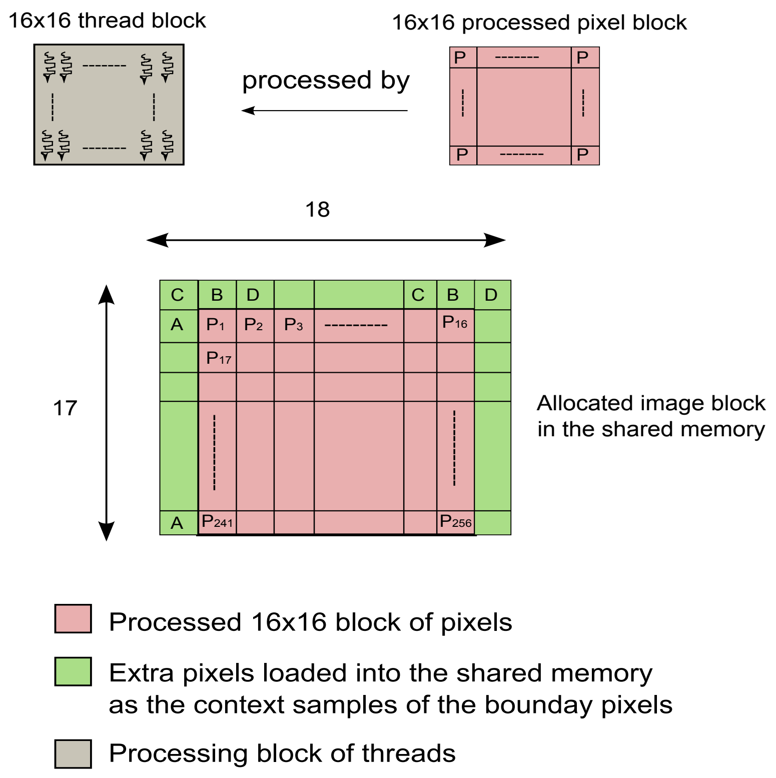

16x16 thread block
16x16 processed pixel block
processed by
P
18
17
C
B
D
A
P1
P2
P3
P16
P17
P241
P256
Allocated image block in the shared memory
Processed 16x16 block of pixels
Extra pixels loaded into the shared memory as the context samples of the bounday pixels
Processing block of threads

Figure 5.2: Block processing of pixels

and run-interrupts) as show in Listings 5.6 and 5.7. A bubble sort[6] is then applied on the *ContextPixelArray* and *PixelPositionArray* to arrange the contexts according to the corrected values of *ContextCountArray*. The three sorted arrays represent the output of this phase and the input of encoding phase three.

[6]Another efficient sorting algorithm, like quick sort, can be used instead of bubble sort. However the effect of the applied sorting algorithm on the overall encoder performacne is not expected to be much, as will be shown in Chapter 6

Listing 5.5: Encoding phase two

```
EncodingPhaseTwo(){

perform initial raster scan;
initialize ContextPixelArray;
initialize PixelPositionArray;
for (;;){
 if(counter == dim.x*dim.y )
   break;
     GetNextSample();
 if (ABS(pix.q) == 0)
   RunMode_Processing();
 else
   Update_ContextPixelArray();
   Update_PixelPositionArray();
 }
 sort(ContextCountArray,
      ContextPixelArray,
      ContextPositionArray);

}
```

Listing 5.6: Run-interrupt handling

```
RunInterrupt_Handling(){

 ContextCountArray[ABS(pix.Q)]
 -=1;
 pix.Q = 365 || 366; \\correct
 if (SIGN == 1)
   pix.Q = -pix.Q
 pix.Px = Ra || Rb;  \\correct
 set pix.RunIndex;
 Update_ContextPixelArray();
 Update_PixelPositionArray();

}
```

Listing 5.7: Run mode processing

```
RunMode_Processing(){

 while (run-segment == true){
   count +=1;
   ContextCountArray[ABS
   (pix.Q)] -=1;
 }
 RunSegment_Coding();
 Update_PixelPositionArray();
 if (interruption == true)
   RunInterrupt_Handling();

}
```

Listing 5.8: Update of context array

```
Update_ContextPixelArray(){

 ContextPixelArray[ABS(pix.Q)]
 [index] = pix;

}
```

Listing 5.9: Update of position array

```
Update_PixelPositionArray(){

 struct pos;
 pos.x = xPos;
 pos.y = yPos;
 pos.pixelCount = count;
 ContextPositionArray[ABS
 (pix.Q)][index] = pos;

}
```

5.2.4 Implementation of Encoding Phase Three

Both the *ContextPixelArray* and the *ContextCountArray* are copied to the global memory of the GPU for processing in phase three. Listing 5.10 shows a part of the implementation code. A one dimensional thread block of size *Rcap*, representing the round encoding capacity, is launched for computation. No shared memory is used for storing the context pixels in this phase, since each pixel is accessed only once, where it gets stored in the corresponding thread register and all the encoding is then applied on the register stored copy. Also no enough shared memory space is available for storing the whole pixals of the image. A shared memory however is allocated to store the context variables *A*, *B*, *C*, *N* and *Nn*, since a regular update to these variables is performed each time a context pixel is encoded. Two additional variables are used : *ToalContextPixels*, which indicates the sum of all pixels in the 366 contexts, and *marker*, which indicates the position of the context cell that is awaiting to be swapped once one of the currently encoded *Rcap* contexts has finished with the encoding of its pixels. In such a case, the corresponding encoding thread performs a swap operation with the context cell pointed to by marker. The thread then increments the *marker* variable by one. An *atomicAdd* operation is also used here in the incrementation process to serialize possible concurrent access of threads to the *marker* variable. Finally, in The *Prediction_Error_Coding()* function, the output bit-stream of the pixel is stored in the *ContextBitSreamArray*, in a position corresponding to the pixel's encoding order. Since the size of the bit-stream buffer of each pixel is set to a maximum size, as was discussed earlier, a special character shall be appended to the pixel's output stream to mark its end.

Listing 5.10: Encoding phase three

```
for(i=threadIdx.x ; i<numberOfRounds*Rcap ; i+=Rcap)
{
  if((i<ToalContextPixels) &&
  (ContextCountArray[threadIdx.x+1]!=0))
  {
     pix = ContextPixelArray[threadIdx.x+1][counter];
     counter +=1;
     if (ContextCountArray[threadIdx.x+1] == counter)
     {
        threadmarker = atomicAdd(marker,1);
        ContextCountArray[threadIdx.x+1] = 0;
        swap = true;
     }
  }
  syncthreads()
  if((swap == true) &&(threadMarker < 367) &&
  (ContextCountArray[threadMarker]!=0) )
  {
     ContextPixelArray[threadIdx.x+1] =
     ContextPixelsArray[threadMarker];

     ContextCountArray[threadIdx.x+1] =
     ContextCountArray[threadMarker];

     ContextCountArray[threadMarker] = 0;
     counter = 0;
     threadMarker = 367;
     swapp = false;
  }
  if(i<ToalContextPixels)
  {
     if ((ABS(pix.Q) != 365) && (ABS(pix.Q) != 366))
     {
       Prediction_Correction();
     }
     Coumpute_Prediction_Error();
     Prediction_Error_Mapping();
     Prediction_Error_Coding();
     Context_Variables_Update();
  }
  __syncthreads();
}
  __syncthreads();
```

Chapter 6

Implementation Results

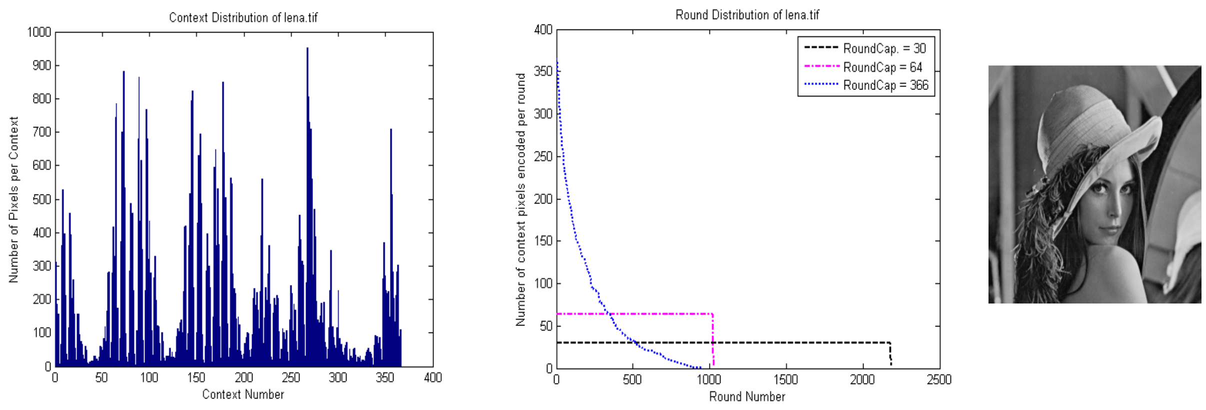
The compatibility of our proposed design to the standard has been proved by comparing the resulting output bit streams of both the original source code and our parallel implementation. The streams have been found to be identical after performing the re-ordering step discussed in Section 5.1.4. In our performance analysis, six grey-scale 8 bit images with different sizes have been used for time measurements. The images were allowed to be encoded by both the sequential JPEG-LS encoder and our parallel presented round encoder. For both implementations, the encoding time was measured from the instant after the image has been read and all the initializations have been made till the output bit-streams have been generated. No storage of the bit-streams is performed for either the sequential or the parallel versions[1]. Also the time consumed by the stream re-ordering step, performed in the round encoder implementation, is not accounted for in our measurements. In order to avoid the time overhead resulting from the first initializations made by CUDA on the device, the image was allowed to be encoded for several runs where the measurements were taken on one of the middle runs. This consequently ensures that this time overhead needed for the first run on the device is no longer included in our measurements. A GeForce GTX 280 graphics card with compute capability of 1.3 having 30 multiprocessors has been used for testing our parallel JPEG-LS implementation.
Figures 6.1 and 6.2 show the context distributions of the images. The distributions have the run-interrupt contexts 365 and 366 included. The figures also include the round distributions of the images performed for three different round capacity values. These are 30, 64 and 366 context pixels per round[2].

6.1 Performance Analysis

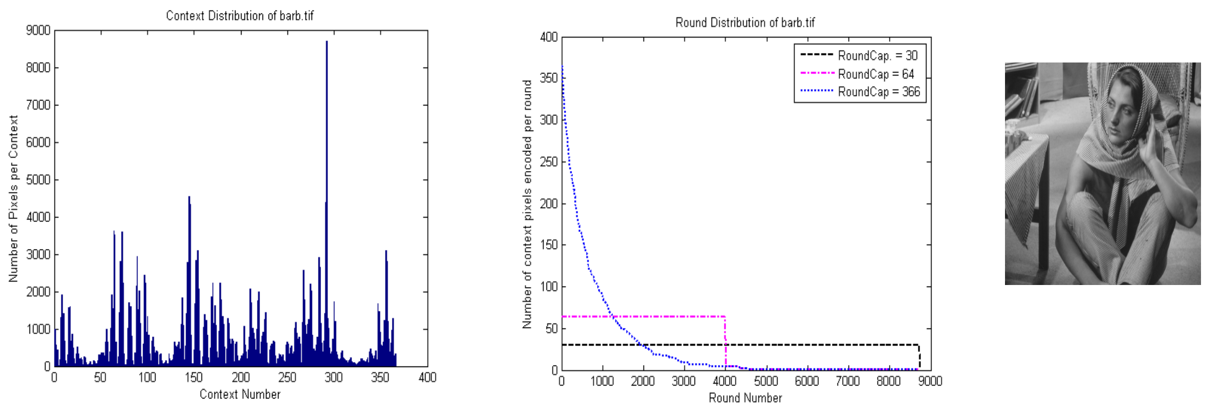
Figure 6.3 shows the amount of time spent in each of the three encoding phases for three of the investigated images. As it can be observed from the figure, the third encoding phase consumes the majority of the encoding time especially for small size images. The amount of time spent in this phase of encoding decreases with increasing the round capacity parameter, as shown in the figure. For middle size images, like *mountain*, around 74%,68% and 58% of the total

[1]This is because for the parallel version, a sequential writing of the output bit stream of each pixel to the GPU memory consumes a lot of time, which results in slowing down the parallel implementation with respect to the sequential one. Therefore our measurements do not account for this operation in both implementations. More discussion is to follow in the coming sections

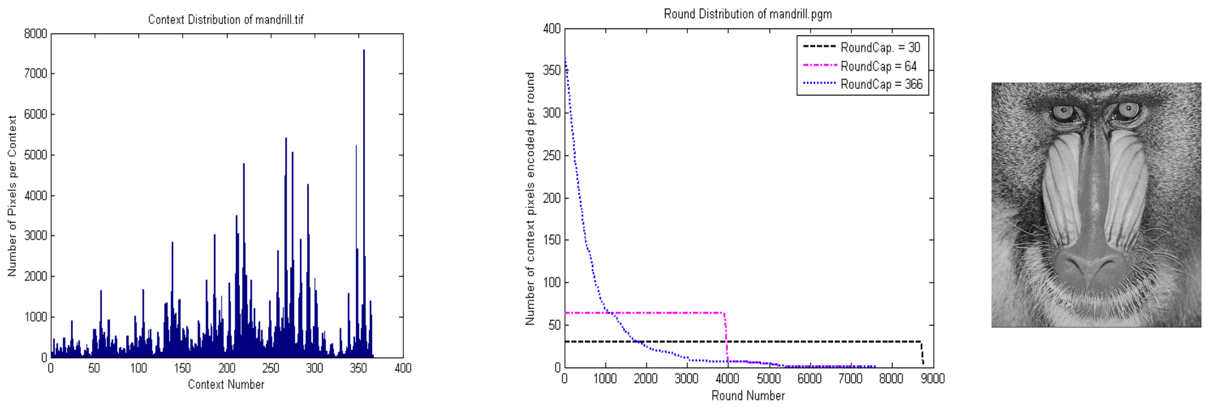
[2]It should be noticed that although some of the round distributions in Figures 6.1 and 6.2 might seem to have a box-shaped distribution, they do actually have a tail extention defined by the context with the highest pixel count



(a) lena.tif-256x256

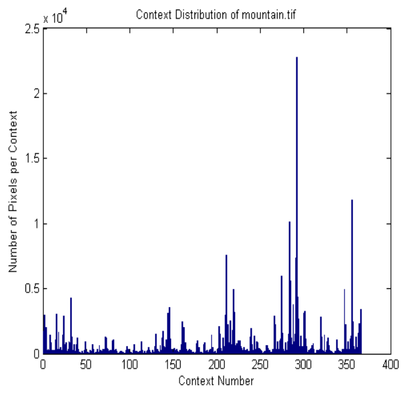


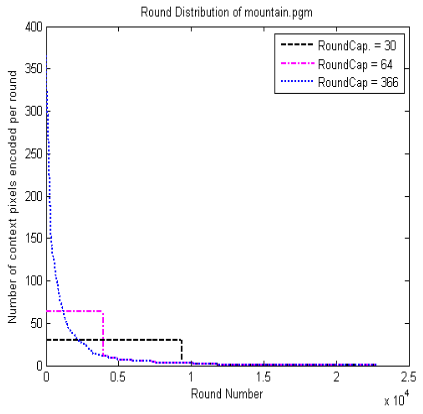
(b) barb.tif-512x512

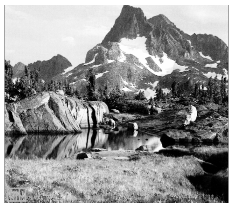


(c) mandrill.tif-512x512

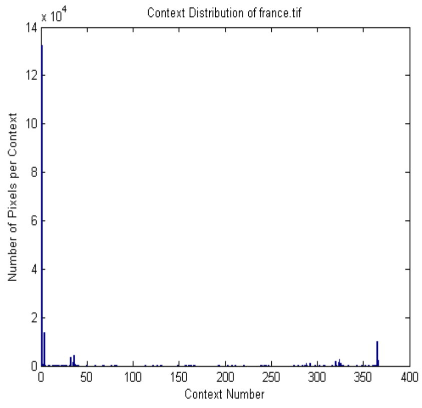
Figure 6.1: Context distribution of images

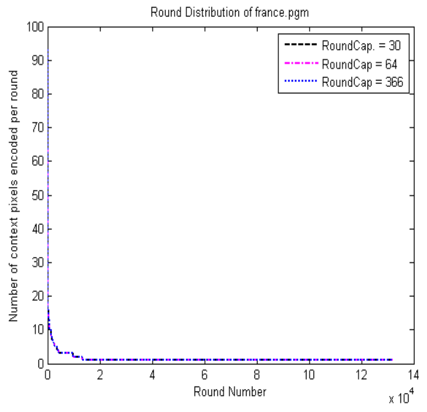


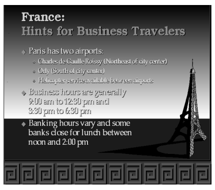




(a) mountain.tif-640x480



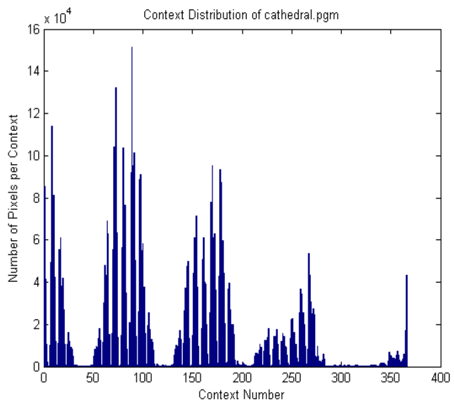


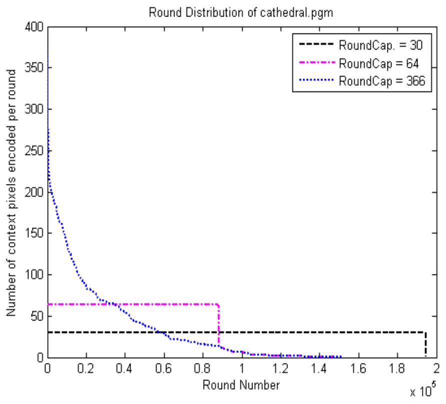


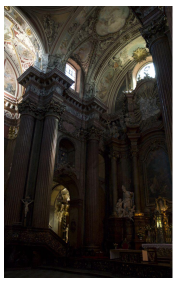
France:
Hints for Business Travelers

- Paris has two airports:
  - Charles de Gaulle Roissy (Northeast of city center)
  - Orly (South of city center)
  - Helicopter service available between airports
- Business hours are generally 9:00 am to 12:30 pm and 3:30 pm to 6:30 pm
- Banking hours vary and some banks close for lunch between noon and 2:00 pm

(b) france.tif-672x496

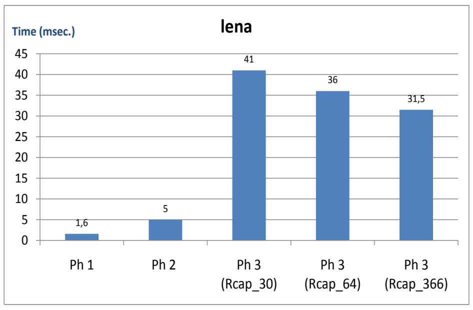




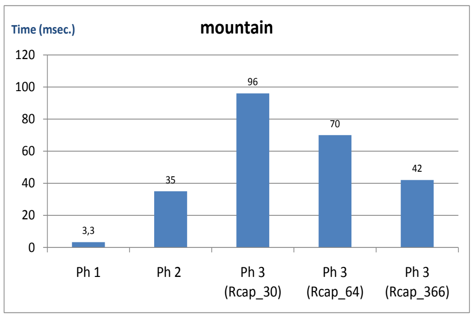


(c) cathedral.tif-2000x3008

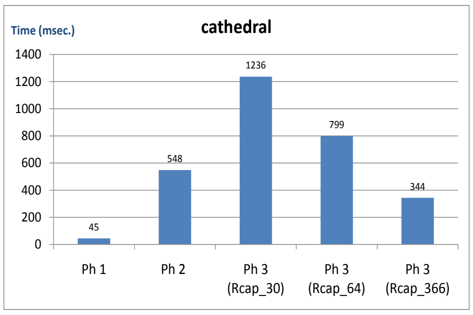
Figure 6.2: Context distribution of images



(a) lena.tif-256x256



(b) mountain.tif-640x480



(c) cathedral.pgm-2000x3008

Figure 6.3: Time analysis of the three encoding phases

encoding time is spent in the third encoding phase for the round capacities 30, 64 and 366 respectively. The contribution of the second encoding phase to the total encoding time becomes more dominant as the size of the image is made larger. For *cathedral*, for instance, around 58% of the total encoding time is consumed by the second phase, when encoding with the maximum round capacity of 366 pixels. The first encoding phase, on the other hand , shows the lowest contribution to the total encoding time constituting around 3% for the case with a round capacity of 64 pixels. The following sections provide more detailed analysis on the performance of each of the three encoding phases.

6.1.1 Analysis of Encoding Phase One

Performing timing analysis on the first encoding phase has revealed a dependency of its encoding time on the size of the image as shown in Figure 6.4. Increasing the image size leads to an increase in both the time needed by the data transfer between the CPU and the GPU, and also the computation time on the GPU. Figure 6.4 shows these results. The increase of the GPU computation time can be explained by the amount of global memory access done while transferring the encoded blocks of the image from the global memory to the shared memory in preparation

for their encoding as was discussed in Section 5.2.2. Increasing the image size results in a corresponding increase in the number of processed image blocks, leading to a higher amount of global memory access, and thus a higher overall encoding time.
Comparing the performance of the first encoding phase to its sequential counterpart in the original JPEG-LS implementation has shown an average speedup of 4.3, with a variance of 2.7 for the six investigated images, as indicated in Table 6.1. This has been obtained by taking both the data transfer and computation times on the GPU into account. By disregarding the data transfer time and evaluating only the computation performance on the GPU with respect to the sequential version, an average speedup of 28.6 with a variance of 85.2 is obtained as shown in Table 6.1. This indicates the existence of a large data transfer overhead that slows down the overall processing process on the GPU. From Figure 6.4, it can be seen that the GPU computation constitutes only around 17% of the total encoding time of phase one, while the rest is spent in transferring the data between the CPU and the GPU.

Image	**Speedup with data transfer**	**speedup without data transfer**
lena-256x256	2.3	18.5
barb-512x512	4.5	29
mandrill-512x512	4.5	36
mountain-640x480	5.2	34
france-672x496	2.5	16.4
cathedral-2000x3008	6.6	37.5
Average	4.3	28.6
Avg. variance	2.7	85.2

Table 6.1: Speedup of phase one

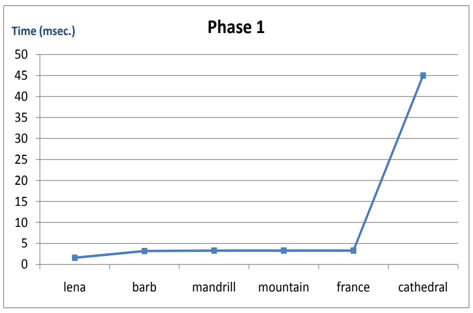
6.1.2 Analysis of Encoding Phase Two

For the second encoding phase, a similar dependency of the encoding time on the image size has been also observed as shown in Figure 6.5. The performance of this phase however is mainly affected by the sequential raster scan executed on the image. The time needed for the raster scan scales linearly with the image size. The sorting, on the other hand, did not show any dependency on the image size, but a rather constant execution time, which did not exceed $0.3 msec$.

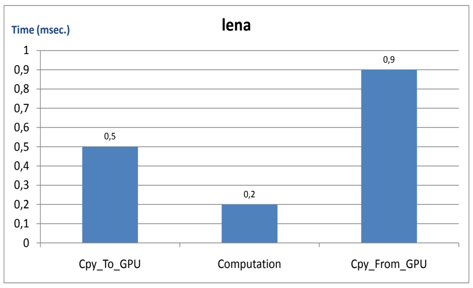
6.1.3 Analysis of Encoding Phase Three

The third encoding phase shows also an increase in its encoding time with increasing the image size, as indicated in Figure 6.7 for the three investigated round capacities. Similar to the first encoding phase, the data transfer constitutes a large overhead especially for small size images encoded with a high round capacity parameter [3]. From Figure 6.7, it can be seen that encoding of *lena* with a round capacity of 366 pixels requires 95% of the total encoding time of phase three only for data transfer.
Based on the obtained timing results, we compare the performance of the parallel phase three

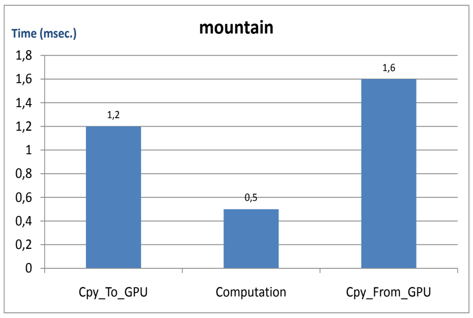
[3]Although no storage of the bit-streams was performed, and thus no output is available for phase three to be transferred to the CPU, however we separately measured the time needed for the bit-streams to be transferred and we added it to our results



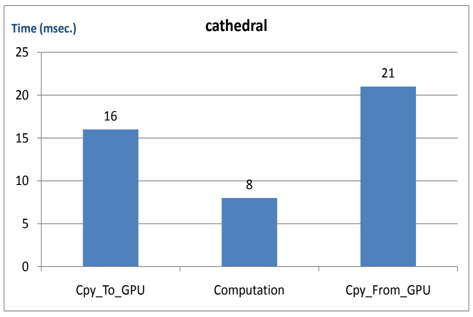
(a) phase 1



(b) lena.tif-256x256



(c) mountain.tif-640x480



(d) cathedral.pgm-2000x3008

Figure 6.4: Timing analysis of encoding phase One

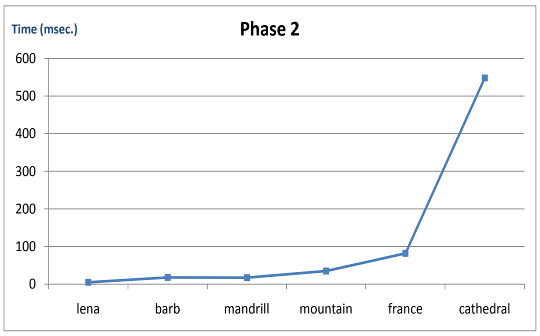


Figure 6.5: Timing analysis of encoding phase two

with its sequential counterpart of the original JPEG-LS implementation. Tables 6.2 and 6.3 indicate the speedup results with and without the data transfer time respectively. Table 6.3 also shows the theoretical speedup of phase three obtained by Equation 6.1.

$$Theoretical\,Speedup = \frac{Total\ number\ of\ context\ pixels}{Total\ number\ of\ rounds} \tag{6.1}$$

As it can be observed, a low performance is obtained for encoding phase three on the GPU, when compared to the CPU sequential performance. No speedup is achieved when taking the data transfer time into account. An average speedup of 2 is obtained for the maximum round capacity case, when comparing only the computation part on the GPU (without the data transfer time) with the sequential version. Such a low performance is thought to be caused by the following.

First, We have previously assumed that the total number of encoding rounds is the main factor affecting the computation speed of our round encoder in phase three. This however has proven not to be the case. By observing the round distributions of Figures 6.1 and 6.2 and the corresponding round numbers in Table 6.3, we find that for *mountain*, the same total number of encoding rounds, 22769, is obtained for the three different round capacities. Yet, we still observe a quite noticeable decrease in the GPU computation time depicted in Figure 6.7 when varying the encoder round capacity. This consequently suggests that the amount of context swapping, made on the GPU to achieve a sorted context encoding, has a large effect on the encoding time of this phase. The lower the round capacity value, the more is the performed swapping, and thus the slower is our round encoder. Since the swapping operation is performed on global memory data, it becomes affected by the slow global memory access routine. Storing the pixel structures in the shared memory is also not possible because of the limited size of the shared memory. The same results have been obtained for *cathedral*, *mandrill* and *barb*, which have the same number of encoding rounds for the round capacities 64 and 366, and still show a different GPU computation time in the two cases.

Second, We have used a shared memory to store the context variables A,B,C,D, N and Nn, so that they can be frequently accessed during the performed context update procedures defined in Listings 2.7, 2.8 and 2.12 of Chapter 2. According to the discussion of Section 3.4.1, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between threads. Figure 6.6 shows an example of how one of the context arrays, for instance context array A, is stored in the shared memory with the 366 context elements being successively distributed among the 16 memory banks. As it has been previously mentioned, a shared memory request for a warp of size 32 is split into one request for the first half of the warp and one request for the second half of the warp. This means that no bank conflicts will occur only if the 16 threads of a half warp access elements in different memory banks. This is however not always the case in our implementation. The decision of which bank each thread has to access is fully determined by the context number Q of the encoded pixel. Since this information is only available at run-time, when the pixel is read by the corresponding encoding thread, such bank conflicts become difficult to avoid. A best case scenario is when the 16 context elements lie in different banks resulting in a one memory request for the whole half warp. A worst case scenario, on the other hand, is when the 16 context elements lie in the same bank resulting in serialized 16 memory requests. Our implementation is thought to be running in between the two cases. The same situation is encountered for the rest of the context variable arrays, which increases the amount of possible bank conflicts. Therefore, a considerable slow down of the GPU computation in this phase of encoding is observed as a result of the performed context update procedures.

Third, another drawback of performing the update of the context variables in parallel, is that the time needed for this update procedure to be finished is context dependent. In terms of parallel computation, this means that all the threads of a parallel working warp will not be able to start encoding their next pixels (in the next round), until the thread with the longest execution time has finished its computation. This depends on the evaluated conditions of the update procedures of Listings 2.7, 2.8 and 2.12. If one thread in the warp requires an x amount of time to update its context variables, the rest of the threads will remain idle for a period of time defined from the instant they have finished computation until an x amount of time has passed. In a sequential processing, on the other hand, the encoding of a pixel starts right after the encoding of the previous pixel has finished, and therefore no idle time is added.
Fourth, because of the structure of the *ContextPixelArray* as an array of pointers to another *Context_Pixel* arrays, it is not possible to transfer this structure from the CPU to the GPU by a single copy operation. Instead, an initial copying of each of the 366 context arrays is first separately performed on the device, followed by a later copying of the pointers pointing to these context arrays. This results in an overall of 367 copy operations on the device, which consume a lot of time for their execution, as indicated in the timing analysis of Figure 6.7.

Image	**Speedup(Rcap-30)**	**Speedup(Rcap-64)**	**Speedup(Rcap-366)**
lena-256x256	0.08	0.09	0.1
barb-512x512	0.14	0.17	0.2
mandrill-512x512	0.14	0.18	0.25
mountain-640x480	0.15	0.19	0.3
france-672x496	0.17	0.24	0.4
cathedral-2000x3008	0.25	0.39	0.89
Average	0.15	0.21	0.38
Avg. variance	0.03	0.009	0.07

Table 6.2: Speedup of phase three with the data transfer time included

6.1.4 Parallel versus Sequential JPEG-LS

Figure 6.8 compares the performance of our parallel implemented JPEG-LS versus the original sequential version. As it can be observed from the figure, the parallel implementation, consisting of the three encoding phases, runs at a lower speed than the sequential version. An average speedup of 0.52 is achieved in the case of maximum round capacity, encoding 366 pixels per round. The computation slowdown is not only caused by the third encoding phase, with the reasons pointed out to in the last section, but also the second encoding phase. By observing the graphs of Figure 6.8, we notice that with larger images, the second encoding phase starts to have an encoding time which is quite comparable to the total encoding time of the whole original sequential implementation. Since the second encoding phase is a sequential working phase, this means that its computation time defines the minimum achievable computation time of the whole implementation (see Amdahl's law Section 3.2). Assuming an imaginary computation time of zero for both encoding phases one and three, a medium size image, like *mountain*, can achieve a maximum speedup of 1.3 based on the sequential working phase two.

Image	lena	barb	mandrill	mountain	france	cathedral
Size	256x256	512x512	512x512	640x480	672x496	2000x3008
Rounds(Rcap-30	2186	8749	8749	22769	132230	194822
Rounds(Rcap-64)	1028	8691	7598	22769	132230	151242
Rounds(Rcap-366)	953	8691	7598	22769	132230	151242
Speedup(Rcap-30) Theoretical	29.9	29.9	29.9	13.09	1.36	29.9
Speedup(Rcap-30) GPU-computation	0.32	0.29	0.28	0.26	0.19	0.29
Speedup(Rcap-64) Theoretical	63	30	34	13.09	1.36	38
Speedup(Rcap-64) GPU-computation	0.58	0.5	0.5	0.4	0.28	0.5
Speedup(Rcap-366) Theoretical	68	30	34	13.09	1.36	38
Speedup(Rcap-366) GPU-computation	2.3	2	2	1.84	0.5	2.13

Table 6.3: Speedup of phase three without including the data transfer time

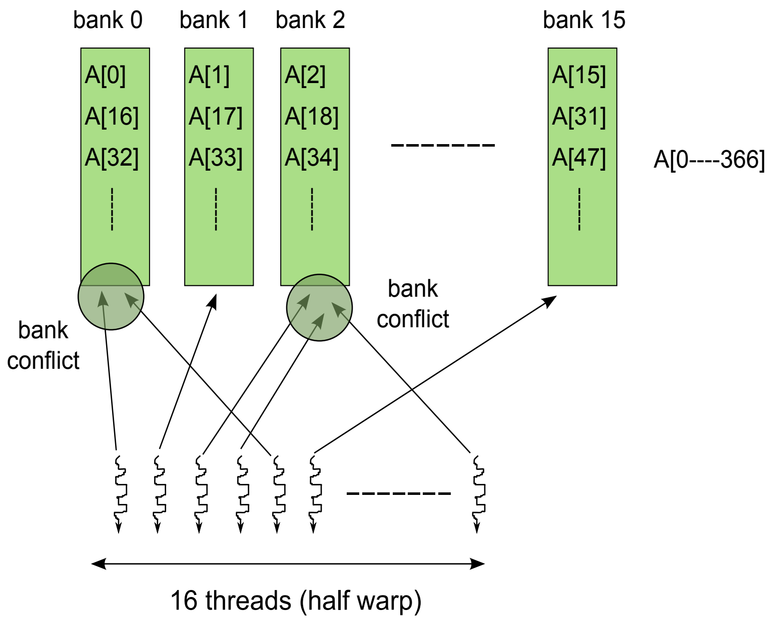
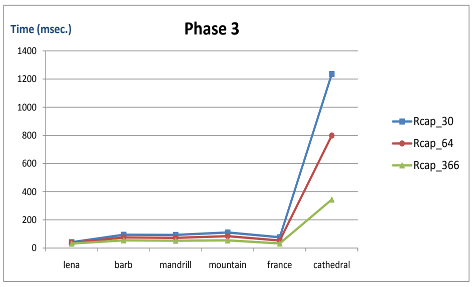


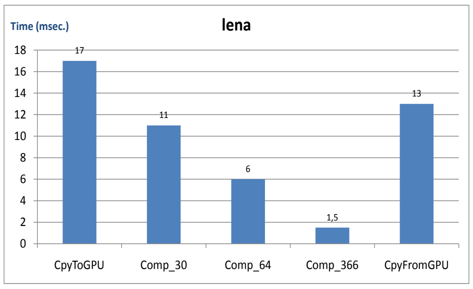
Figure 6.6: Bank conflicts in shared memory

6.2 Future Work

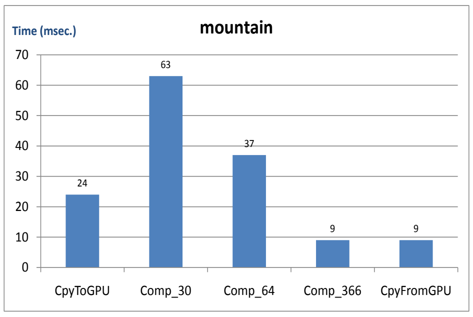
From the presented results, it can be seen that there are two main factors limiting the maximum achievable speedup of our paralell JPEG-LS design. These are the large data transfer overhead, resulting from copying the data between the CPU and the GPU in both directions, and the sequential computation of the second encoding phase. The following are suggestions to overcome



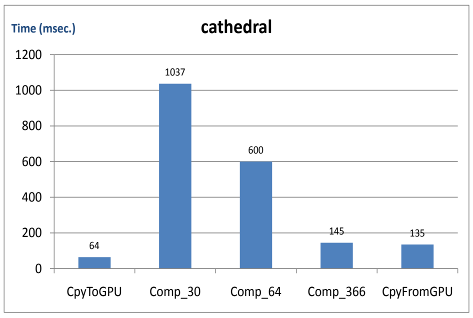
(a) phase 3



(b) lena.tif-256x256



(c) mountain.tif-640x480



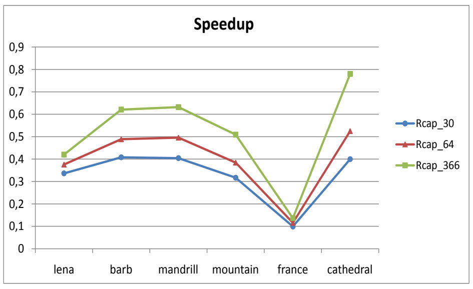
(d) cathedral.pgm-2000x3008

Figure 6.7: Time analysis of encoding phase three

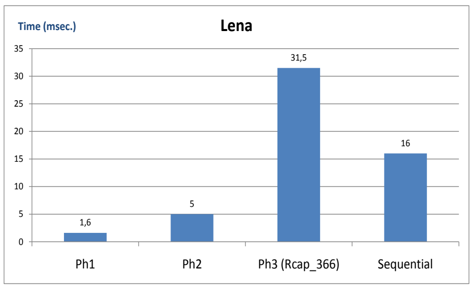
Image	**Speedup(Rcap-30)**	**Speedup(Rcap-64)**	**Speedup(Rcap-366)**
lena-256x256	0.34	0.37	0.4
barb-512x512	0.4	0.49	0.6
mandrill-512x512	0.4	0.49	0.6
mountain-640x480	0.34	0.43	0.58
france-672x496	0.1	0.11	0.13
cathedral-2000x3008	0.4	0.5	0.78
Average	0.33	0.4	0.52
Avg. variance	0.014	0.02	0.05

Table 6.4: Speedup of prallel JPEG-LS

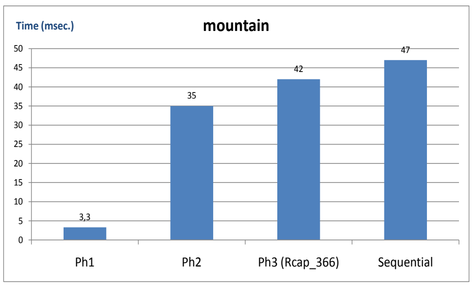
these problems and to increase the speed of our parallel presented JPEG-LS implementation in general.
First, the data transfer overhead can be minimized by overlapping both the copying and the computation on the GPU. CUDA provides ways for data streaming aiming at overlapping both



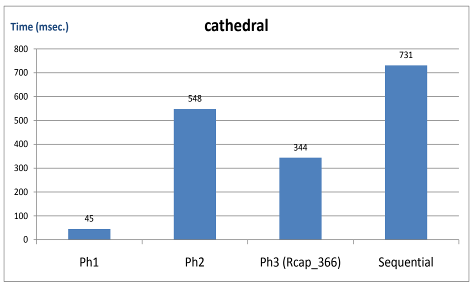
(a) Speedup of parallel JPEG-LS



(b) lena.tif-256x256



(c) mountain.tif-640x480



(d) cathedral.pgm-2000x3008

Figure 6.8: Parallel versus sequential JPEG-LS

processes in order to hide the overhead of transmission, and thus to reduce the overall time needed by the GPU [5]. Application of such methods can result in an improved performance of parallel JPEG-LS.

Second, another way of overcoming the large data transfer overhead, is to encode phase two on the GPU, with one working thread doing the sequential work. This will eliminate the need to transfer both the output of phase one from the GPU to the CPU, and the input of phase three from the CPU to the GPU.

Third, For the third encoding phase, eliminating the need to perform context swapping can help accelerate the round encoding of this phase, when a round capacity of less than 366 pixels is used. An attempt has been made to try to change the pixels' representation from context to round representation on the CPU (see Figure 4.2), so that when the data is transferred to the GPU, no swapping operation is required. This however has resulted in both increasing the amount of encoding time of phase two on the CPU, and also increasing the data transfer overhead.

Chapter 7

Conclusion

In the presented work, the compression of images using the lossless mode of JPEG-LS standard has been investigated for tendency of encoding parallelism. The nature of the applied context encoding performed by the JPEG-LS standard has revealed a data dependency loop, which is encountered when two pixels, scanned in a raster order, are found to belong to the same quantized context Q. This leads to the later occurring pixel requiring the context variables, updated by the previous same-context pixel. Previous approaches to parallelize JPEG-LS have mostly modified the standard, so as to break the feedback loop caused by the context update procedure, and thus provide implementations that are not compatible with the standard. Our proposed design however presents a parallel software implementation, which is fully compliant to the JPEG-LS standard. In our implementation, parallelism is approached through the concept of round encoding, which makes use of the fact that regular mode pixels with different contexts can undergo a totally independent encoding scheme. An image is therefore encoded in terms of context rounds which contain pixels of different contexts, and thus can be processed in parallel. At the end of encoding of each round, the context variables of the encoded contexts are updated, so that processing of the next pixeles in the next round can follow. The presented round encoder adheres to the standard in terms of the number and order of the performed context updates. Three round assignment schemes were presented, which differ in the way they schedule the pixels among the encoding rounds. The first round assignment scheme, called maximum load assignment, maximizes the round encoding capacity by processing the maximum number of available contexts per round. This achieves the minimum number of encoding rounds, however the worst workload distribution. The second round assignement scheme, called round robin assignment, improves the workload load distribution by assigning less number of contexts per round. This however leads to a higher number of total encoding rounds. The third round assignment scheme, called context priority assignment, combines the advantages of the first two schemes by assigning less number of pixels per round, like the round robin scheme, however with prioritizing the contexts according to the number of pixels they have. This consequently achieves a total number of encoding rounds which is quite comparable to the maximum load scheme, and with an almost equally distributed workload. The only disadvantage of the context priority assignement scheme is the complexity of its implementation, which involves pre-sorting of the contexts, and a number of context swapping operations done during the encoding to keep the priority condition valid. A final design proposal suggests the encoding of the run interrupt pixels as regular mode ones to achieve an increase in the amount of parallelism.

A parallel JPEG-LS implementation, based on the round encoding concept, has been introduced for running on the GPU, with the sequential parts running on the CPU. Three encoding phases

constitute our implementation. The first encoding phase is a parallel working phase, which is responsible for making the context classification and edge prediction of all pixels, and thus computing those parameters needed for the further processing of regular-mode pixels. A large data transfer overhead has been obtained for the first encoding phase. The second encoding phase is a sequential working phase, which is concerned with the coding of run-segment lengths and also arranging the pixels in a context representation form, on which the round encoding process can be applied. The final encoding phase is again a parallel working phase, which implements the presented round encoding scheme in performing the rest of the encoding tasks of regular-mode and run-interrupt pixels. These tasks include prediction correction, residual calculation, rice mapping, context update and Golomb coding, as defined by the standard. The encoding of the context rounds has been made according to the context priority assignment scheme which is implemented by both encoding phase two, providing an initial sort of the contexts, and encoding phase three, making the necessary context swaps during the encoding process.
The timing analysis results of our implementation have shown that most of the encoding time is consumed by the second and third encoding phases. An average speedup of 4.3 has been obtained for the first encoding phase with respect to its sequential counterpart of the original JPEG-LS implementation. The speedup increase to an average of 28.6, when the data transfer time is not taken into account. The encoding time of phase two has been found to become more dominant as the size of the image is made larger reaching to 58% of the total encoding time for the largest image used in our investigation. No speedup has been achieved for the third encoding phase, which has shown both a large data transfer overhead, similar to phase one, and a low computation performance of the round encoder, when compared to the theoretical predicted speedup values. Possible reasons are first, the swapping operation performed on the context pixels residing in the device global memory, and which suffers from a low access speed. Second, the existence of shared memory bank conflicts which occur when the threads perform the context update procedures, accessing different context elements in the same memory bank. Third, the fact that some threads have to remain idle, waiting for other threads to finish their context update process, which vary in its execution time depending on the encoded context. This consequently leads to the encoding of some pixels lasting for a longer time period than it actually requires.
For the above mentioned reasons, along with the large data transfer overhead and the relatively long execution of the sequential second encoding phase, no speedup was achieved for our parallel implemented JPEG-LS with respect to the original sequential implementation.
Future work can proceed with the optimization of the GPU encoding phases by applying data streaming methods which aim at hiding the data transfer time within the computation time. Also, re-organizing the used data structures and modifying the representation of context pixels can lead to achieving a higher computation speed on the GPU for the third encoding phase. A further reduction of the sequential part of our implementation can result in a considerable improvement, since its execution time defines the maximum achievable performance according to Amdahl's law.
Finally, as far as parallelization of a sequential compression algorithm is concerned, an initial design, which takes into account the minimization of data and task dependencies from the very beginning, can lead to a significant increase in speed performance upon parallelization. Therefore, it is now becoming a necessity for evolving compression standards to include the concept of parallel encoding into their design criteria, so that less efforts with better results can follow afterwards to increase the encoding speed performance of these algorithms.

Bibliography

[1] D. Salomon, *Data Compression, The Complete Reference*. Springer, second ed., 2000.

[2] M. Weinberger, G. Seroussi, and G. Spario, "The LOCO-I lossless image compression. algorithm: principles and standarization into JPEG-LS," *IEEE transactions on image processing*, vol. 9, no. 8, pp. 1309–1324, 2000.

[3] D. Santa-Cruz, T. Ebrahimi, J. Askelof, M. Larsson, C. Christopoulos, and S. Simon, "JPEG 2000 still image coding versus other standards," *Proceedings of SPIE*, vol. 4115, pp. 446–454, 2000.

[4] *FCD 14495 - Lossless and near-lossless coding of continous tone still images (JPEG-LS)*, 1997. ISO/IEC JTC1/SC29 WG1 (JPEG-JPIG).

[5] *NVIDIA CUDA programming Guide*. V.2.3.1.

[6] M. Klimesh, V. Stanton, and D. Watola, "Hardware implementation of a lossless image compression algorithm using a field programmable gate array," *TMO Progress Report*, pp. 42–144, February 2001.

[7] A. Savakis and M. Piorun, "Benchmarking and hardware implementation of JPEG-LS," *Proceedings of the International Conference on Image Processing*, vol. 2, pp. II–949–II–952, 2002.

[8] M. Ferretti and M. Boffadossi, "A parallel pipelined implementation of LOCO-I for JPEG-LS," *Proceedings of the 17th International Conference on Pattern Recognition (ICPR'04)*, vol. 1, pp. 769–772, 2004.

[9] P. Merlino and A. Abramo, "A fully pipelined architecture for the LOCO-I compression algorithm," *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 17, pp. 967–971, April 2009.

[10] M. E.Papadonikolakis, A. P.Kakarountas, and C. E.Goutis, "Efficient high-performance implementation of JPEG-LS encoder," *Journal of Real-Time Image Proc*, vol. 3, pp. 303–310, 2008.

[11] S. Wahl, Z. Wang, Q. Chensheng, M. Wroblewski, L. Rockstroh, and S. Simon, "Memory-efficient parallelization of JPEG-LS with relaxed context update," pp. 142–145, 2010.

[12] *Image Compression Benchmark*. http://www.imagecompression.info/test-images/.

[13] *http://www.stat.columbia.edu/ jakulin/jpeg-ls/mirror.htm*, 2009.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged, was
solely the effort of the author. At no
stage was any collaboration entered into
with any other party.

(Haitham Assem Tantawy)