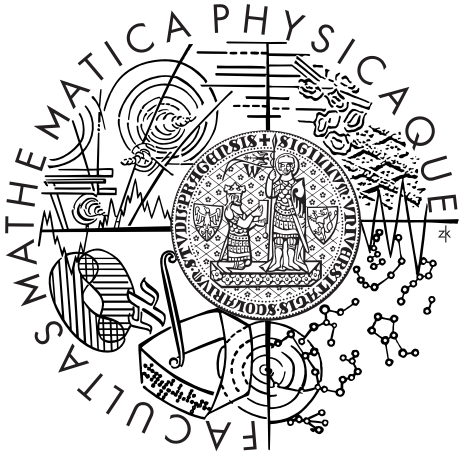
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Leonid Buneev

Arithmetic coding on GPU

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. Jan Horacek
Study programme: Computer Science
Specialization: Programming and software systems

Prague 2013

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Aritmeticke kodovani pomoci GPU

Autor: Leonid Buneev

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Jan Horáček, Kabinet software a výuky informatiky

Abstrakt: Cílem této práce je prozkoumat možnosti vytváření implementace paralelního aritmetického kódování a změřit míru zlepšení výkonu.

V první části, krátký přehled aritmetického kódování s jeho seriovou implementací (Amir Said, FastAC) je popsaná. Práce dále popisuje zásady práce s GPU a identifikuje možnosti zlepšení algoritmu a jeho paralelizace. Několik implementaci jsou uvedeny, s měnícími se mírami zlepšení výkonu a nedostatky.

V závěru práce poskytuje výsledky různých testů naší implementace, stejně jako diskuse o proveditelnosti uplatnění GPU-orientované verze algoritmu místo sériové v reálném světě.

Klíčová slova: aritmetické kódování, komprese, gpu, cuda, kódování entropie

Title: Arithmetic coding on GPU

Author: Leonid Buneev

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Jan Horáček, Department of Software and Computer Science Education

Abstract: The aim of this thesis is to investigate possibilities for creating parallel arithmetic coding implementation and measure performance improvements.

In the first part, short overview of Arithmetic coding with its serial implementation (FastAC by Amir Said) is presented. The thesis then describes principles of work with GPUs and identifies possibilities of algorithm improvement and parallelization. Several parallel implementations are given, with varying performance improvements and occasional drawbacks.

In conclusion, thesis provides results of performance tests of our implementation, as well as discussion about feasibility of applying GPU-oriented version of algorithm instead of serial one in real-world applications.

Keywords: arithmetic coding, compression, gpu, cuda, entropy encoding

Contents

1. Introduction

1.1 Why to speed up AC

In the modern world data compression is a very important area of research, because the volume of data that need to be transported and stored is growing very fast - faster, than technologies of data storage and transfer. However, quite often, especially in the current world of home computers, computing powers are not wholly used. That is why compression is useful nowadays - while increasing computational costs, it reduces usage of other resources, such as data storage space or transmission capacity.

Arithmetic coding is one of the most popular lossless compressing algorithms out there. It is universal algorithm - it works just with data stream, regardless of their type, and therefore can be applied to any data. It is very effective for entropy-based compression algorithm - in fact, it achieves almost the best theoretically possible compression rates for given statistical models. But encoding and decoding processes are quite computationally expensive in comparison with, say, Huffman coding (less effective but faster entropy-based compression algorithm). Sometimes computational price may be too expensive, especially with large data streams. For example, with compression of high definition video usually less efficient and less complex algorithms are used, such as Huffman coding, because AC is too slow for real-time encoding even with the most modern processors.

1.2 Goal of this paper

The popularity of Graphic Processing Units (GPUs) opens new possibilities for general-purpose computation including the acceleration of algorithms. Massively parallel computations using GPUs have been applied in various fields by researchers. This paper tries to implement data-parallel arithmetic coding on GPUs using the NVIDIA GPU and the Computer Unified Device Architecture (CUDA) programming model.

1.3 Paper structure

The second chapter will give general overview of arithmetic encoding and decoding principles in theory. In the third chapter theory behind arithmetic coding is explained. Fourth chapter introduces sequential implementation of AC, called FastAC, written in C++ by Amir Said. The fifth chapter fill focus on general-purpose computations of GPU, and after it our parallel AC implementation using CUDA is described. In Chapter 7 we will overview Class library for AC compression, that use our parallel implementation and FastAC as sequential implementation, and provide developer instructions for using it. The last chapter will provide test results, overview them and provide some use cases for our algorithm.

2. Prior work

2.1 Arithmetic coding development history

The first step toward arithmetic coding was taken by Shannon[1], who observed in a 1948 paper that messages N symbols long could be encoded by first sorting the messages in order of their probabilities and then sending the cumulative probability of the preceding messages in the ordering. The code string was a binary fraction and was decoded by magnitude comparison. The next step was taken by Peter Elias in an unpublished result; Abramson [2] described Elias' improvement in 1963 in a note at the end of a chapter. Elias' code was studied by Jelinek[3]. The codes of Shannon and Elias suffered from a serious problem: As the message increased in length the arithmetic involved required increasing precision. By using fixed-width arithmetic units for these codes, the time to encode each symbol is increased linearly with the length of the code string.

Meanwhile, another approach to coding was having a similar problem with precision. In 1972, Schalkwijk [4] studied coding from the standpoint of providing an index to the encoded string within a set of possible strings. This is a last-in-jirst-out (LIFO) code, because the last symbol encoded was the first symbol decoded. Cover [5] made improvements to this scheme, which is now called enumerative coding. These codes suffered from the same precision problem.

Both Shannon's code and the Schalkwijk-Cover code can be viewed as a mapping of strings to a number, forming two branches of pre-arithmetic codes, called FIFO (first-in-first-out) and LIFO. Both branches use a double recursion, and both have a precision problem. Rissanen [6] alleviated the precision problem by suitable approximations in designing a LIFO arithmetic code. Code strings of any length could be generated with a fixed calculation time per data symbol using fixed-precision arithmetic.

Pasco [7] discovered a FIFO arithmetic code, discussed earlier, which controlled the precision problem by essentially the same idea proposed by Rissanen. In Pasco's work, the code string was kept in computer memory until the last symbol was encoded. This strategy allowed a carry-over to be propagated over a long carry chain. Pasco also conjectured on the family of arithmetic codes based on their mechanization.

In Rissanen [6] and Pasco [7], the original (given, or presumed) symbol probabilities were used. In [8], Rissanen and Langdon introduced the notion of coding parameters "based" on the symbol probabilities. The uncoupling of the coding parameters from the symbol probabilities simplifies the implementation of the code at very little compression loss, and gives the code designer some tradeoff possibilities. In [7] it was stated that there were ways to block the carry-over.

Rissanen and Langdon [8] successfully generalized and characterized the family of arithmetic codes through the notion of the decodability criterion which applies to all such codes, be they LIFO or FIFO, L-based or P-based. The arithmetic coding family is seen to be a practical generalization of many pre-arithmetic coding algorithms, including Elias' code, Schalkwijk [4], and Cover [5].

In 1979 Nigel and Martin [9] rediscovered FIFO Arithmetic code while using slightly different approach to representing coder internal values and probablities

and defining Range code method based on it. While Range code is essentially Arithmetic coding, it provides more practical view on coding process by using more integer and fixed-precision arithmetic.

At this point arithmetic coding was well-defined, and later research was focused on increasing encoding speed without sacrificing compression ratio, either by using approximate multiplication, or approximate division.

2.2 Arithmetic coding parallelization

In 1992, Howard and Vitter [10] made first attempt at designing parallel arithmetic encoder for image compression, and defined "quasi-arithmetic coding". The main idea behind it was by limiting amount of possible arithmetic coder "states" and represent coder in lookup tables. While achieving good results performance wise, quasi-arithmetic coding still have huge memory requirements for building lookup tables and, as a result of limiting coder states, worse compression ratio. In 1999 Mahapatra et al. [11] introduced multialphabet arithmetic coding algorithm and its parallel pipelined implementation, where different coders work on different levels of a binary tree representing the symbol.

With introducing of massively-parallel general purpose GPUs the idea of arithmetic coding parallelization gained new breath. In 2008, Balevic et al. [12] have shown that usage of block-parallel arithmetic coding directly on GPU may significantly reduce resulting simulation data size, which will to smaller and faster data flow between host and device memory. In 2010, Rusňák [13] focused on implementing parallel EBCOT (JPEG2000 entropy encoding algorithm) - and gained significant overall speed increase, but the arithmetic coding step of EBCOT have shown slowdown that prevented gaining optimal speed for realtime HD-video compression.

In 2011, Xiao et al [14] have studied negative effects of block-parallel arithmetic encoding on compression ratio (as opposed to serial version) and concluded, that there is "the loss by breaking the probability prediction process and the loss by breaking the information for context decision at the starting of each slice <...>the loss by breaking probability adaptation process takes the most of losses". Meanwhile, Chen[15] tried to implement arithmetic coding on GPU and gained "speedup values ranging from 26x to 42x" as opposed to CPU version.

This paper is aimed on implementing particular type of arithmetic coder - bitwise arithmetic coder - on GPU. Using bit model leads to slightly better compression ratios, have much less requirements on memory (which is very limited while working on GPU), but is way more computationally-expensive. As a reference CPU implementation, Amir Saids FastAC is used, which is currently state of the art (in terms of performance) serial bitwise arithmetic coder.

3. Arithmetic coding in general

3.1 Lossy and lossless compression

There are different approaches to data compression problem. Compression can be either lossless or lossy.

Lossy data compression algorithms allow discarding (using) some of it. Lossy compression algorithms are mainly used for multimedia data - images, audio and video. They exploit knowledge about the nature of data being compressed, and try to discard less valuable data parts (for example, discard less recognised by human ear frequencies from audio file). Compression ratio (that is, the size of the compressed file compared to that of the uncompressed file) , that can be achieved by lossy algorithms without easily visible differences from source, is sufficient - up to 1:10 for audio and still images, and up to 1:100 for video.

Usually lossy-compressed file is quite different from original at the bit-level. Lossless algorithms, on the other hand, allow the exact original data to be reconstructed from the compressed data, but they cant offer as efficient compression rates as lossy algorithms do. It is common practice to apply lossy and lossless compression technique as different steps in the same compression program - for example, H.264 video codec, apart from lossy video encoding techniques, can use Huffman or arithmetic coding for slightly improving compression rates. Main idea behind lossless compression is mapping input data to a bit sequences in such way that "probable" (e.g. frequently encountered) data will produce shorter output than "improbable" data. This process is known as entropy coding.

3.2 Entropy

In information theory, entropy is a measure of the uncertainty in a random variable. The term usually refers to the Shannon entropy H - measurement of average unpredictability in a random variable. Shannon entropy measures actual data "value", and provides an absolute limit on the best possible lossless compression ratio.

Shannon denoted the entropy H of a discrete random variable X with possible values $x_1, ..., x_n$ and probability mass function $P(X)$ as

$$H(X) = E[I(X)] = E[-\ln(P(X))] \tag{3.1}$$

Here E is the expected value operator, and I is the information content of X. $I(X)$ is itself a random variable.

When taken from the finite sample, the entropy can be explicitly written as

$$\begin{aligned} H(X) &= \textstyle\sum_i P(x_i)\, I(x_i) \\ &= -\textstyle\sum_i P(x_i) \log_2 P(x_i) \\ &= -\textstyle\sum_i \frac{n_i}{N} \log_2 \frac{n_i}{N} \\ &= \log_2 N - \frac{1}{N} \textstyle\sum_i n_i \log_2 n_i, \end{aligned} \tag{3.2}$$

Data with high entropy value is unpredictable and cannot be efficiently compressed with lossless algorithm. But data with low entropy value contain rela-

tively predictable values, and can be efficiently encoded to high-entropy message with lower size. One of the most basic entropy encoding algorithms is Huffman Coding.

3.3 Huffman coding

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol), that expresses the most common source symbols using shorter strings of bits than are used for less common source symbols. Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code. The running time of Huffman's method is fairly efficient, it takes $O(n \log n)$ operations to construct it. A method was later found to design a Huffman code in linear time if input probabilities (also known as weights) are sorted.

3.4 Arithmetic coding

3.4.1 Encoding process

Huffman coding maps each symbol to its most efficient bit representation, and this approach works perfectly, when symbol probabilities are powers of 0.5. But with distributions, where symbols have frequencies far from a power of 0.5, such as 0.75 or 0.375, Huffman code cannot achieve best possible compression rates.

Arithmetic coding - one more entropy encoding algorithm - uses different approach. It doesn't map symbols - instead, it maps whole message to a single number, a fraction n where $(0.0 \leq n < 1.0)$. Fundamentally, the arithmetic encoding process consists of creating a sequence of nested intervals in the form $\Phi_k(S) = [\alpha_k, \beta_k), k = 0, 1, ..., N$, where S is the source data sequence, α_k and β_k are real numbers such that $0 \leq \alpha_k \leq \alpha_{k+1}$ and $0 \leq \beta_k \leq \beta_{k+1}$. For better understanding of arithmetic coding principles, we will represent intervals in the form $|b, l\rangle$, where b is the base (starting point) of the interval, and l - the length of the interval. The relationship between the traditional and the new interval notation is

$$|b, l\rangle = [\alpha, \beta) \text{ if } b = \alpha \text{ and } l = \beta - \alpha \tag{3.3}$$

The intervals used during the arithmetic coding process are, in this new notation, defined by set of recursive equations

$$\begin{aligned} &\Phi_0(S) = |b_0, l_0\rangle = |0, 1\rangle, \\ &\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c(s_k) l_{k-1}, p(s_k) l_{k-1}\rangle, k = 1, 2, ..., N, \end{aligned} \tag{3.4}$$

where S is data sequence, s_k is symbol at position k, and N is total symbols count in S. The properties of the intervals guarantee that $0 \leq b_k \leq b_{k+1} < 1$, and $0 < l_{k+1} < l_k \leq 1$.

The final task is to define a code value $C(S)$, that will represent data sequence S. Later we will show that decoding process works properly for any code value $C \in \Phi(S)$. But we cannot provide the code value to the decoder as pure real number - it must be stored, using a conventional number representation. Since we have the freedom to choose any value in the final interval, we will choose the values with the shortest representation.

Example 3.1. Let us assume that source alphabet A has four ($M = 4$) symbols, the probabilities and distribution of the symbols are $p = [0.2, 0.5, 0.2, 0.1]$ and $c = [0, 0.2, 0.7, 0.9, 1]$, and the sequence of ($N = 6$) symbols to be encoded is $S = \{2, 1, 0, 0, 1, 3\}$

Figure 1.1 illustrates, how the encoding process corresponds to the selection of intervals in the line of real numbers. We begin at the top of the figure, with the interval $\Phi_0 = [0, 1)$, which is divided into four subintervals, each with length proportional to symbol probabilities, and one of this interval will be chosen depending on first symbol in sequence S. Specifically, interval $[0, 0.2)$ corresponds to $s_1 = 0$, interval $[0.2, 0.7)$ corresponds to $s_1 = 1$, interval $[0.7, 0.9)$ corresponds to $s_1 = 2$, and $[0.9, 1)$ corresponds to $s_1 = 3$. First symbol of data sequence is $s_1 = 2$, and interval $\Phi_1 = [0.7, 0.9)$ is chosen. Interval Φ_1 is again divided into four subintervals, with lengths, proportional to symbol probabilities (note that their lengths are proportional to the length of interval they belong to as well). One from these intervals is chosen based on second data sequence symbol s_2, and so on.

The interval lengths are reduced by factors equal to symbol probabilities in order to obtain code values that are uniformly distributed in the interval $[0, 1)$. For example, if 20% of all sequences start with symbol $s_1 = 0$, then 20% of the code values must be in the interval assigned to those sequences, which can only be achieved if the first symbol is assigned with 0 an interval with length equal to its probability, 0.2. Same reasoning applies to the assignment of the subinterval lengths: every occurrence of symbol 0 will reduce the interval length to 20% of its current length. This way, after encoding all symbols the distribution of code values should be close approximation of a uniform distribution, which means that the code will have nearly highest possible entropy.

The process of finding the shortest binary representation is quite simple. We will show it by induction. The main idea is that for large intervals we can find the optimal value by testing a few binary sequences, and as the interval lengths are halved, the number of sequences to be tested has to double, increasing the number of bits by one. Thus for interval length l_N use following rules:

- If $l_N \in [0.5, 1)$, then choose $\hat{v}(S) \in \{0, 0.5\} = \{0.0_2, 0.1_2\}$ for 1-bit representation.
- If $l_N \in [0.25, 0.5)$, then choose $\hat{v}(S) \in \{0, 0.25, 0.5, 0.75\} = \{0.00_2, 0.01_2, 0.10_2, 0.11_2\}$ for 2-bit representation.
- If $l_N \in [0.125, 0.25)$, then choose $\hat{v}(S) \in \{0, 0.125, ..., 0.75, 0.875\} = \{0.000_2, 0.001_2, ..., 0.110_2, 0.111_2\}$ for 3-bit representation.

By looking at this pattern conclusion can be made, that the minimum number of bits required for representing $\hat{v} \in \Phi_N(S)$ is

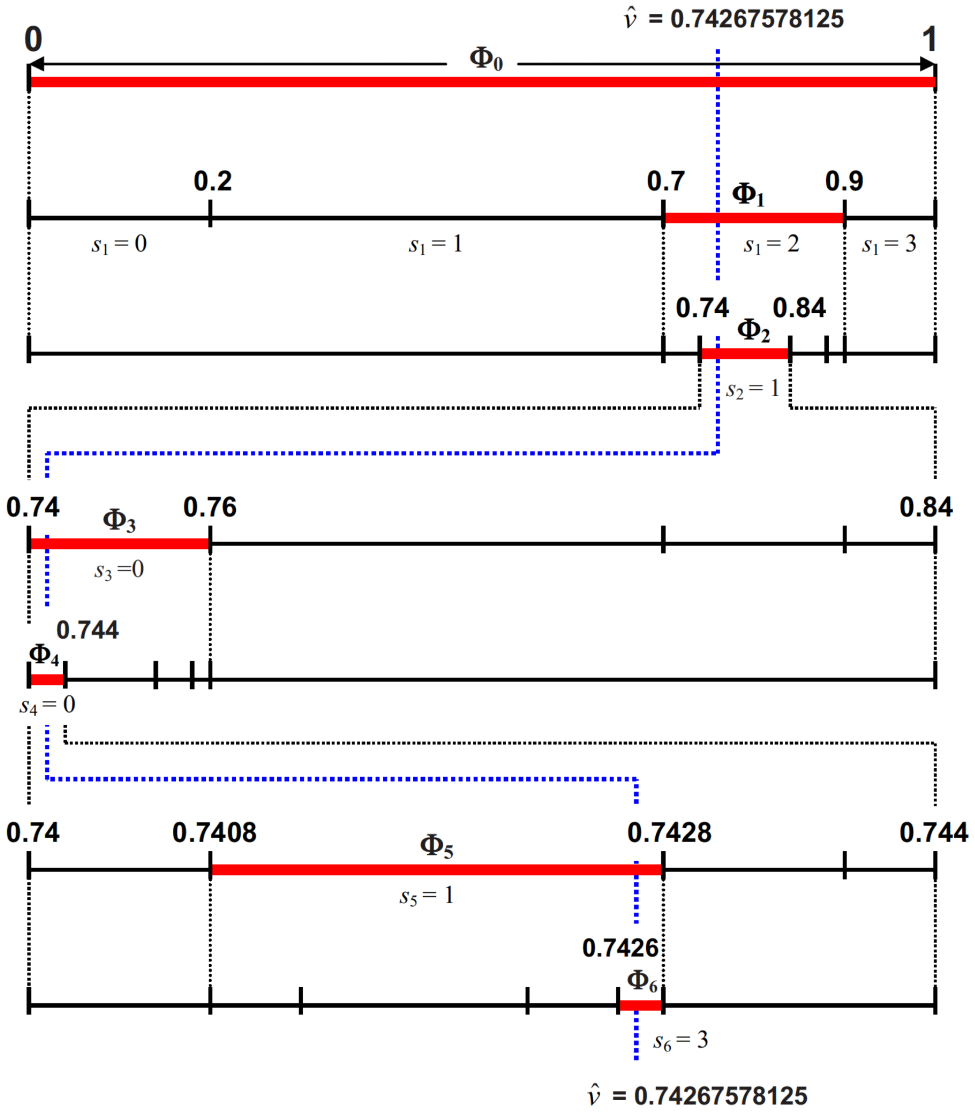


Figure 3.1: Graphical representation of the arithmetic coding process of Example 1: the interval $\Phi_0 = [0, 1)$ is divided in nested intervals according to the probability of the data symbols. The selected intervals, corresponding to data sequence $S = 2, 1, 0, 0, 1, 3$
are indicated by thicker lines.

$$B_min = \lceil -log_2(l_n) \rceil \text{bits} \tag{3.5}$$

Actually we may use even less bits, because trailing zeros can be safely removed. But with optimal coding average number of saved bits is one, and for that reason it is not useful in practice.

3.4.2 Decoding process

In arithmetic coding the decoded sequence is determined by the code value $\hat{v}$. We will represent decoded sequence as

$$S(\hat{v}) = \{s_1(\hat{v}), s_2(\hat{v}), ..., s_N(\hat{v})\}. \tag{3.6}$$

In decoding process any code $C \in \Phi_N(S)$ can be used for decoding the correct sequence. The decoding process recovers the data symbols in the same order that they were encoded. Fundamentally, decoding process recovers the sequence of intervals created by the encoder, and searches for the correct value in each of these intervals. It is defined as

$$\begin{array}{lcl} \Phi_0(S) & = & |b_0, l_0\rangle = |0, 1\rangle, \\ s_k(\hat{v}) & = & \{s : c(s) \leq (C - b_{k-1})/l_{k-1} < c(s+1)\}, k = 1, 2, ..., N \\ \Phi_k(\hat{v}) & = & |b_k, l_k\rangle = |b_{k-1} + c(s_k)l_{k-1}, p(s_k)l_{k-1}\rangle, k = 1, 2, ..., N. \end{array} \tag{3.7}$$

Example 3.2. Lets decode the data obtained in Example 1.1. In Figure 1.1 we can see graphical meaning of code value $\hat{v}$ - it is a value, that belongs to all nested intervals, created during coding. The dotted line shows that, while moving during magnifying, its value remains the same. Therefore, we can start we can start decoding from the first interval $\Phi_0(S) = [0, 1)$ - we will compare $\hat{v}$ with the cumulative distribution c to find the only possible value of s_1

$$\hat{s_1}(\hat{v}) = \{s : c(s) \leq \hat{v} = 0.74267578125 < c(s+1)\} = 2 \tag{3.8}$$

The value of $\hat{s_1}$ can be used for determining $\hat{s_2}$. In fact, we can "remove" effect of s_1 in $\hat{v}$ by defining the normalized code value

$$\hat{v_2} = \frac{\hat{v} - c(s_1)}{p(s_1)} = 0.21337890625 \tag{3.9}$$

$\hat{v_2} \in [0, 1)$, i. e. its value is normalized to its initial interval. In this interval we can use the same process to find

$$\hat{s_2}(\hat{v}) = \{s : c(s) \leq \hat{v_2} = 0.21337890625 < c(s+1)\} = 1 \tag{3.10}$$

Then process continues, and the updated values computed while decoding.

But the decoder has only the initial code value $\hat{v}$, and cannot know right time to stop decoding. This happens because intervals are mapped to sets of sequences, and each real number actually corresponds to one infinite sequence. For example, the sequences corresponding to $\Phi_6(S) = [0.7426, 0.7428)$ are all those that start as $\{2, 1, 0, 0, 1, 3, ...\}$. Our code value $\hat{v} = 0.74267578125$ corresponds to one such infinite sequence, and the decoding process can go on forever.

In practice there are two solutions of this problem:

1. provide the number of data symbols (N) at the beginning of compressed file

2. Use a special symbol as "end-of-file", and assign the smallest allowed by coder probability value to it.

The fact that $S \neq S' \iff \Phi_N(S) \cap \Phi_N(S') = \emptyset$ guarantees, that decoded sequence is correct.

4. FastAC

4.1 Overview

There is a lot of different serial AC implementations out there. As a reference serial implementation we will use Fast Arithmetic Coding (FastAC) by Amir Said. The source code is free and transparent, and at the same time it contains several tweaks that help "achieve optimal compression and higher throughput by better exploiting the great numerical capabilities of the current processors". It also contains several advanced tricks for bitwise AC in particular, and those will help us create parallel version as well.

FastAC contains its own set of classes to create an interface for advanced using of this implementation in real-world projects. But this interface contain some excess functionality for us, like advanced data-source model classes, and is not suitable to embed GPU implementation without some code refactoring. We have created our own interface instead, and took just core encode and decode functionality from FastAC. Our interface allows using of both FastAC as serial and our parallel implementations. It will be described in detail in Chapter 6.

FastAC actually consists of four slightly different versions of AC implementation - we will use one with 32-bit integer variables and 32-bit products, which "is the most portable, reliable and usually the fastest, integrating all the main acceleration techniques". It is simpler as well, because it uses 32-bit arithmetic all the time. This restriction have a potential drawback - the "total number of bits of precision assigned to interval length and probability must not exceed 32" (the reason will be explained later in this chapter). However, as long as we focus on binary model, 32 bit for storing length and probability is more than enough.

4.2 Encoding process

We need to take a look into the source code to better understand arithmetic coding principles and tweaks used in FastAC. First of all, there are some variables that contain current state of encoder and constants specific for binary data model. You can see them in listing 3.1.

Listing 4.1: FastAC state variables

```
1  // I/O
2  unsigned char * code_buffer, * new_buffer, * ac_pointer;
3  // Arithmetic coder state
4  unsigned base, value, length;
5  // Binary model settings - zero bit probability
6  unsigned bit_0_prob;
7
8  // threshold for renormalization
9  const unsigned AC__MinLength = 0x01000000U;
10 // maximum AC interval length
11 const unsigned AC__MaxLength = 0xFFFFFFFFU;
12 // length bits discarded before multiplication
13 const unsigned BM__LengthShift = 13;
```

The meaning of variables is straightforward, and constants will be explained later in this chapter.

The encoding process is focused in function encode(), that encodes one bit of information and changes coder state accordingly. You can see it in listing 3.2.

Listing 4.2: encode()

```
1  void encode(unsigned bit)
2  {
3    unsigned x = bit_0_prob * (length >> BM__LengthShift);
4    if (bit == 0)
5      length  = x;
6    else {
7      unsigned init_base = base;
8      base   += x;
9      length -= x;
10     if (init_base > base) propagate_carry();
11   }
12   if (length < AC__MinLength) renorm_enc_interval();
13 }
```

Here some explanation is needed. We calculate `x` - new interval length, in case that new symbol is zero. For that we just need to multiply current length by probability of zero symbol (between 0.0 and 1.0). But we don't want to use floating-point arithmetic, and that is why the type of `bit_0_prob` is `unsigned int`. To be clear, that is how `bit_0_prob` could be calculated:

Listing 4.3: set_zero_probability()

```
1  void set_zero_probability(double p0)
2  {
3    bit_0_prob = unsigned(p0 * (1 << BM__LengthShift));
4  }
```

`BM__LengthShift` is the trick here. Normally, we would map probability onto whole range of unsigned int values, and then make something like `x = length*((float)bit_0_prob/max_unsigned_value)`, but here division and float conversion operations are involved, and those are expensive. Instead, we decide to use just 12 lower bits of `unsigned int` for storing probability, and other 20 bits for length calculation. In other words, `length >> BM__LengthShift` substitutes division and doesn't involve `float`, but loses some precision by discarding lower 12 bits of length. Amount of bits for storing probability can be changed - more bits for probability increases its precision but decreases precision of calculation.

Then we update our interval. If our bit is actually zero, we just set interval length to `x`, base remains the same. Otherwise, base is increased and length is decreased by `x`.

Note, that in theory we operate with absolute values of interval base and length, but in practice it will lead to extremely large numbers and expensive calculations. To address this issue, when length become small enough we can discard some top bits of base, send them directly to output, update length accordingly (multiply by $2^{discarded\_bits\_count}$), continue encoding. This procedure is called renormalisation. For this reason at line 12 of listing 3.2 we check if the length is small enough and call `renorm_enc_interval()` if needed. Listing 3.3 shows this procedure.

Listing 4.4: renorm_enc_interval()

```
1 void renorm_enc_interval()
2 {
3   do {
4     // output and discard top byte
5     *ac_pointer++ = (unsigned char)(base >> 24);
6     base <<= 8;
7   }
8   while ((length <<= 8) < AC__MinLength); // length*256
9 }
```

For performance reasons FastAC sends to output whole bytes only when they are ready. We need to code some number between *base* and $base + length$, and that is why `AC__MinLength` is set to `0x01000000` - when length will be less than this value, first 8 bytes in binary representation of every number in this interval would equal first 8 bytes of base.

Line 8 of listing 3.2 contains hidden trap. The sum of `base` and `x` can exceed the maximum possible value of `unsigned int`. Line 10 takes care about such situation - it compares new base with old base, base can only be increased or remain the same, therefore if value of new base is lesser overflow has happened. The `propagate\_carry()` function is called then, it is straightforward as well and you can find it at listing 3.5.

Listing 4.5: propagate_carry()

```
1 void propagate_carry()
2 {
3   unsigned char * p;            // carry propagation on
        compressed data buffer
4   for (p = ac_pointer - 1; *p == 0xFFU; p--) *p = 0;
5   ++*p;
6 }
```

And this is everything that needs to be noted about encoding process. The `encode()` function is just called for every bit from input, then some simple clean up is made to finalize output and compressed stream is ready. The next section will explain how FastAC can decompress it back.

4.3 Decoding process

The same set of state variables and constants from listing 3.1 is used during decoding. Listing 3.6 provides `decode()` function.

Listing 4.6: propagate_carry()

```
1 unsigned decode()
2 {
3   unsigned x = bit_0_prob * (length >> BM__LengthShift);
4   unsigned bit = (value >= x);
5   if (bit == 0)
6     length  = x;
7   else {
8     value  -= x;
9     length -= x;
```

```
10    }
11    if (length < AC__MinLength) renorm_dec_interval();
12    return bit;
13  }
```

Decode process is just inverted encode. The `value` variable is set to first 32 bits of input (compressed stream) at initialization step, and then additional data is read from input during renormalisation when needed.

Firstly we calculate the length of interval for symbol 0 same way as we did during encoding. Then we decide what bit will be sent to output (uncompressed stream) - zero if value falls into `x` interval, and 1 otherwise. Then we need to update coder state accordingly and make renormalization if needed. Renormalisation function for decoding is shown at listing 3.7.

Listing 4.7: renorm_dec_interval()

```
1 void renorm_dec_interval()
2 {
3   do {
4     // read least-significant byte
5     value = (value << 8) | unsigned(*++ac_pointer);
6   } while ((length <<= 8) < AC__MinLength);
7 }
```

There are several interesting things to note about decoding and encoding differences.

1. The `decode()` doesn't need to propagate carry to output, because `value` is never increased like `base` in `encode`.
2. In spite of input for `decode()` (compressed stream) being lesser than for `encode()` (original stream), `decode()` is potentially slower, because both functions are called once for every bit of their output stream. In other words, `decode()` is $O(uncompressed\_stream\_size)$, while `encode` will be $O(compressed\_stream\_size)$.
3. Renormalisation in both functions work with compressed stream - it reads from it during `decode()` and writes during `encode()`.

4.4 Data models

Everything said earlier was about static binary data model. It means, that there are only two symbols in the alphabet - '0' and '1', and zero probability is given to the algorithm and is fixed. This is the simplest algorithm variant and it needs to be understood first, but there are many better data models.

One enhancement is extending alphabet - storing probabilities of 2-bit, 3-bit, or 8-bit symbols. It needs more memory but leads to way better compression ratios. Extended alphabet variant is not targeted by this paper, but there is a CPU implementation of 8-bit adaptive data model in the program, just to compare binary coding with non-binary.

Another enhancement is adaptivity - both encoder and decoder start with zero symbol probability = 50%, and according to certain rules modify this value on the

fly based on symbols already encoded / decoded. There may be many different rules how to react on already encoded. Both CPU and GPU implementations of Adaptive binary data model are available in our implementation.

Third variant is context - there are several binary models (either static or adaptive), and for each symbol program decides which model to use according to previous symbols. Needs more memory but leads to way better compression ratios, especially when model-choosing rules are carefully crafted according to data stream type. Both CPU and GPU implementations of context-adaptive binary model are available in our implementation (although in our implementation model choosing rules are very primitive - just choose model, indexed by last 8 bits).

5. NVIDIA CUDA Basics

The fact that the performance of graphic processing units (GPUs) is much bigger than the central processing units (CPUs) of nowadays1 is hardly surprising. GPUs were formerly focused on such limited field of computing graphic scenes. Within the course of time, GPUs became very powerful and the area of use dramatically grew. So, we can come together on the term General Purpose GPU (GPGPU) denoting modern graphic accelerators. The driving force of rapid raising of the performance are computer games and the entertainment industry that evolves economic pressure on the developers of GPUs to perform a vast number of floating-point calculations within the unit of time. The research in the field of GPGPU started in late 70's2. In the last few years, we can observe the renaissance in this area caused by rapid development of graphic accelerators. Two main players in the field of GPGPUs are AMD with their ATI Stream Technology and NVIDIA which introduced Compute Unified Device Architecture (CUDA), the parallel computing engine accessing GPGPUs resources to software developers. Through the frameworks extending commonly used programming and scripting languages such as C, Java, Python or MATLAB, CUDA enables easy way to make applications using NVIDIA GPUs.

This chapter introduces the architecture of CUDA-capable graphic accelerators in comparison with the present multi-core CPUs. The overview of memory hierarchy and programming model as well as advantages and limitations of CUDA are discussed.

5.1 Hardware Architecture

Present multi-core CPUs usually have 2 - 8 cores. These cores usually work asynchronously and independently. Thus, each core can execute different instructions over different data at the same time. According to the Flynn's taxonomy, we are talking about Multiple Instruction stream, Multiple Data stream (MIMD) class of computer architectures.

On the other hand, GPUs are designed for parallel computing with an emphasis on arithmetic operations, which originate from their main purpose - to compute graphic scene which is finally displayed. Current graphic accelerators consist of several multiprocessors (up to 30). Each multiprocessor contains several (e.g., 8, 12 or 16) Arithmetic Logic Units (ALUs). Up to 480 processors is in total on the current high-end GPUs. Figure 4.1 shows the general overview of the CPU and GPU.

Also the memory hierarchy is specific in the case of graphic accelerators. Each multiprocessor has registers that are used by ALUs. Processors within a multiprocessor can access shared memory of typical size 16KB, or a main memory of the accelerator, which is not cached on the majority of present accelerators4. Global memory in terminology of CUDA, is accessible from all the processors on the accelerator. In addition, there are two separate memory areas (constant memory and texture memory) also shared across the multiprocessor and both cached and read-only. When accessing some element from the texture memory, a couple of surrounding elements are also loaded. This feature is called spatial

Figure 5.1: Comparison CPU and GPU architectures. Source: http://ixbtlabs.com/articles3/video/cuda-1-p1.html

locality. One of the most limiting factors is a very small capacity of shared memory and registers. If application uses more variables per thread than available registers, they are stored in a local memory which is, in fact, the dedicated part of global memory. Accessing these variables is as time-consuming as accessing any other variable stored in the global memory. For better understanding, we can see the memory hierarchy in Fig. 4.2

5.2 Programming model

A CUDA-capable GPU is referred to as a device and the CPU as a host. Thread is the finest grain unit of parallelism in CUDA. Thousands of threads are able to run concurrently on the device. Threads are grouped into the warps. Size of a warp is usually 32 threads. Warp is the smallest unit that can be processed by multiprocessors. Warps scheduled across processors of one multiprocessor are coupled into a thread blocks. Block is a unit of the resource assignment. Typical size of a thread block is 64-512 threads and depends on the particular application what is the optimal size of a thread block to ensure the best utilization of the device. Thread blocks form a grid. Grid can be viewed as a 1-dimensional, 2-dimensional or 3-dimensional array. Figure 4.3 is depicting the described hierarchy.

5.3 Features and Limitations of CUDA

It is easy to learn the CUDA API, but hard to program efficient applications which utilize the GPU's performance. CUDA API is a set of extensions based on the standard C language. Counterweight to many features of this massively parallel architecture is that there are limitations mostly caused by HW architecture. CUDA belongs to the class of Single Instruction, Multiple Thread (SIMT) according the Flynn's taxonomy. SIMT originates in Single Instruction Stream, Multiple Data Stream (SIMD) class known for example from the supercomputers based on vector processors (e.g., Cray 1). SIMT also implies the divergence in the program that usually leads to the serialization of the run. Recursive functions are not supported either. As introduced before, graphic accelerators were

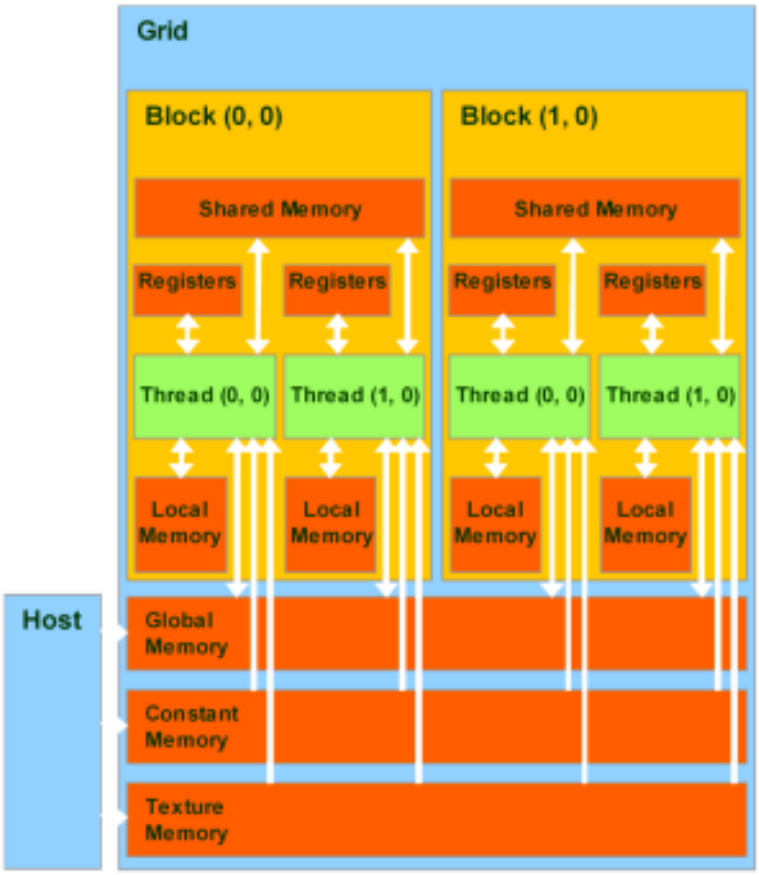


Figure 5.2: CUDA memory model. Source: http://ixbtlabs.com/articles3/video/cuda-1-p5.html

developed with the focus on computing vast amounts of arithmetic operations. Many of them are implemented directly in the hardware with a cost of units of warp-cycles. Besides arithmetic functions there is a set of bitwise operations also implemented "in hardware".

Of course, a set of constructs used in parallel programming is present in CUDA. For example several methods of barrier synchronisation primitives, native broadcast of a single variable, scatter and gather functions or atomic operations which prevents from race conditions.

The use of shared memory has also significant impact on the overall performance but the limiting factor is its size (8-48 KB). Talking about memory, CUDA brought more efficient data transfer operation between the host and the device. Unlike OpenCL6, CUDA is closed source belonging to NVIDIA corp. which can be considered as a limitation as well.

There is a lot of information about CUDA on the web. For more detailed information see http://www.nvidia.com.

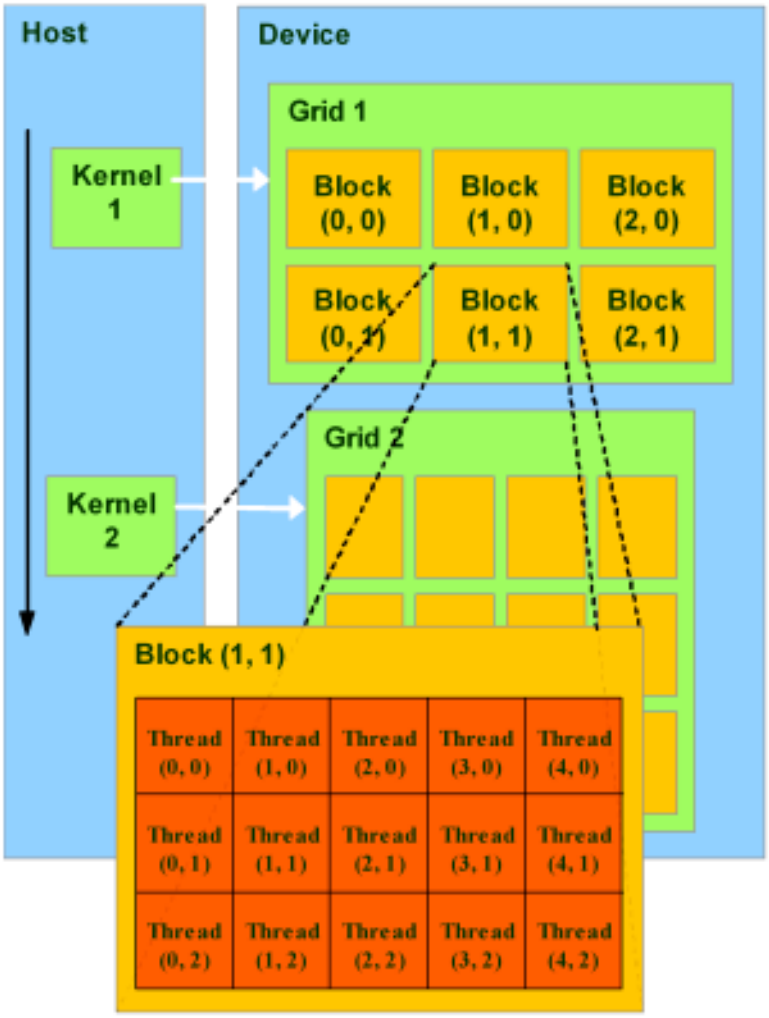


Figure 5.3: NVIDIA CUDA programming model. Source: http://ixbtlabs.com/articles3/video/cuda-1-p5.html

6. Parallel AC implementation

6.1 The main idea

There are two main forms of parallelization algorithms - task parallelism and data parallelism.

Task parallelism analyses individual algorithm steps, search independent ones and calculates them at the same time. Often this is a hard task and involves some algorithm modifications to make some steps independent from others. There are no major independent steps in arithmetic encoding or decoding, so we cannot parallelize the algorithm itself. Moreover, this class of parallel algorithms can utilise multi-core nature of modern processors, but it is very tough to make effective use of GPU resources because GPUs have single-instruction (SIMT) architecture.

Another approach, data parallelism, is used more often and generally is easier to implement. It analyses algorithm to find, which parts of data stream can be processed independently, and then processes many data chunks at the same time. This class of parallel algorithms, in contrast to task parallelism, can effectively utilise GPU resources, because the same procedure is used to process every data chunk - perfect hit for SIMD architectures.

Unfortunately, it is impossible to process different small parts of input stream (bytes, for example) for arithmetic coder at the same time without serious drawbacks, because each new calculation requires result of previous calculation to be done. In other words, every call of function `encode()` from FastAC relies on data calculated by previous calls. Instead, input stream is divided to rather large parts (chunks), and all chunks are processed independently but simultaneously. It allowed us to write a kernel that effectively utilises GPU resources without making major modifications to serial algorithm - which is very important, because serial decoder on CPU should be able to process data encoded in parallel by GPU.

The fact that each chunk is compressed individually means that overall compression ratio of data stream should theoretically be somewhat worse than with processing whole stream sequentially. That turned out to be true only for static models - adaptive, context-adaptive and byte-adaptive models all provided better compression when applied to chunks individually.

6.2 Encode kernel

First of all, data must be split into chunks. Chunk size and amount of chunks should be chosen very carefully - it directly impacts effectiveness of GPU usage. Then, we process each chunk with its own thread, which will create compressed version of every chunk. But this is not everything - compressed chunks must be united in continuous stream. This operation is special case of reduce algorithm, which is commonly used in data processing - to form a resulting output stream. Whole process is illustrated by figure 4.1.

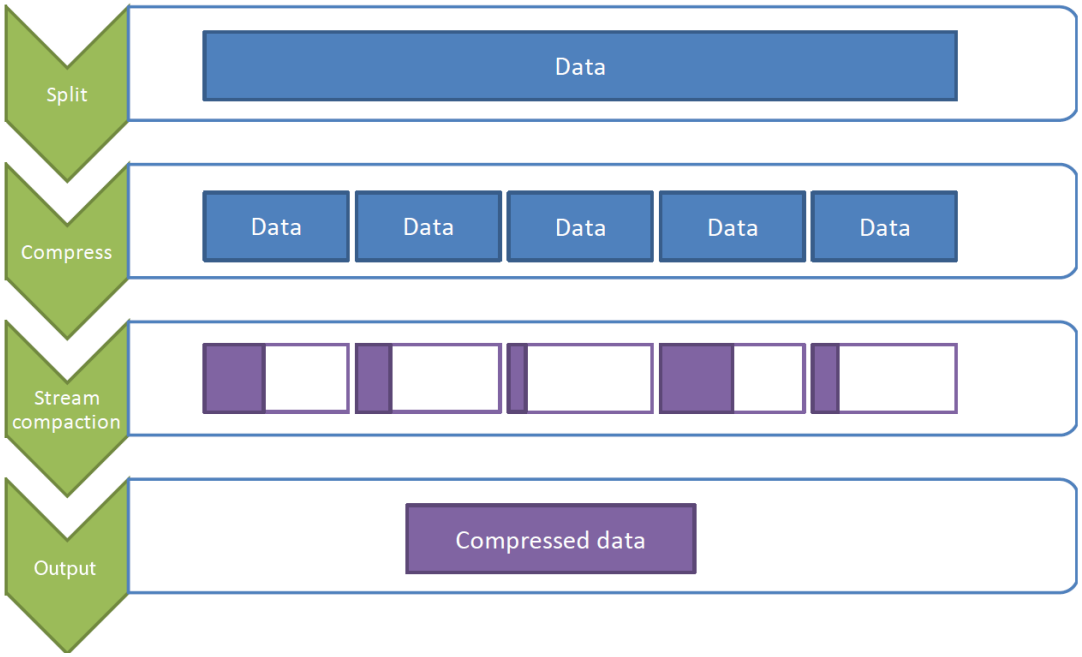


Figure 6.1: Parallel arithmetic encoding steps

6.2.1 Split

The most important part here is choosing chunk size. Too small chunks will lead to significant loss in compression ratio, and too big chunks will mean that we cannot place enough of them into device memory at the same time (usually about 1-2 Gb) to fully utilize GPU parallel processing potential. The implementation allows setting chunk size explicitly, because concrete usage scenarios and target hardware may vary. Default chunk size is 16 kilobytes, which means that theoretically up to 65536 chunks can be stored simultaneously on device with 1Gb memory.

After compressing one chunk of data information about compressed chunk size must be stored. In our implementation by default we store it in first two bytes of every chunk, which means that algorithm can handle chunks up to 64 Kb in size.

6.2.2 Compress

Algorithm for compressing of one separate chunk is fairly straightforward modification of serial version. Firstly, thread must determine correct address of input chunk start from blockId and threadId, and after that just apply serial algorithm. In its easiest version (with given static probability of zero symbol) algorithm uses very little additional memory - which usually is a bottleneck for GPU computations. This operation produces compressed data chunks, but we still need to provide compaction - e. g. move every chunk right to the end of previous chunk. This operation is called ”Stream compaction”

6.2.3 Stream compaction

In usual circumstances, stream compaction should be done on CPU. Our tests have shown, that on usual media files stream compaction takes about 2% of whole encoding time, which is neglicible. Moreover, in some situations (such as streaming) stream compaction on CPU and compressing on GPU can be done at the same time.

However, there is another approach - stream compaction on GPU itself. It can be very useful in situations, where data is already on gpu before compressing (for example, as a result of some scientific computation), is very redundant (which will lead to high compression ratios), and needs to be transformed to CPU. In such systems memory transfer itself is a bottleneck, and compressing data on GPU with decompressing on CPU afterwards can lead to some performance improvements.

Stream compaction on GPU. Straightforward GPU implementation is time-consuming and has $O(nlog_2n)$ addition operations, while serial version performs only $O(n)$ operations. Harris et al [16] have shown in 2007 more effective approach for stream compaction, which use tree-like sum ordering to effectively limit sum operations count by $O(n)$, and his algorithm can be used as separate kernel right after compression step. The tests have shown that stream compaction is significantly faster on GPU compared to CPU. Unfortunately, our implementation of GPU-based stream compaction in some cases introduced instability. We didn't investigate it much further - it didn't promise any significant performance improvements in usual use cases. Theoretically it must work, though, and we will be happy if someone will enhance our parallel compression implementation with flawless parallel prefix sum.

6.2.4 Limitations

There are several limitations that need to be considered while using our implementation. First of all, it works better for large datasets and large chunkSizes, and developer should choose chunk size to split whole dataset in about 4096 chunks to fully utilize GPU powers - so, for significant performance improvement the size of uncompressed data must be more than two megabytes. Usually this is the case - serial implementation is fast enough for almost-instant compression of small files, but with larger file sizes (streaming HD video) you may want to use the power of GPU. Although our GPU implementation can work with input of any size, it works more effectively when dealing with amount of chunks, that is a power of two. Thus, for further acceleration it makes sense to delegate correct amount of work to GPU (chunk size * some power of two) and process everything that was left on CPU at the same time.

Also, there are some memory considerations. For more complex data models, you will need more memory. If algorithm needs isolated copy of those data models (which is usually the case when using adaptive data models), this may become a problem - if all additional data wont fit in shared memory, global memory access overhead will most likely heavily slow down the algorithm.

6.3 Decode kernel

Decode kernel is not part of this work, although the parallelization process will be similar and easy to implement. But our compressed data stream model (every chunk has 2 bytes header with compressed size) doesn't fit parallel decompression - to correctly split work between threads, you need to know all chunks sizes before decoding. Simple fix will be storing all chunks sizes right at the beginning of the file - this approach is a little worse for some use cases (video streaming), but other than that there is no drawbacks to it. After you extract chunk sizes (on cpu), you can calculate starting position in compressed data for every thread, and after that just slightly adapt serial algorithm for parallelization and use it in CUDA kernel. Decoding process needs access to the same data models as encoding process, and perform almost the same operations, so the same memory considerations as for encode kernel apply here.

7. Software architecture, documentation

7.1 Codecs

Part of this work is quite simple class library, which contains implementation of both GPU and CPU arithmetic coders. They are compatible with each other - CPU and GPU encoder will produce absolutely identical outputs for given input.

First of all, there is an `ArithmeticCodec` abstract class, which defines abstract methods for encoding and decoding data. Important definitions are shown in listing 6.1.

Listing 7.1: ArithmeticCodec

```
1  class ArithmeticCodec
2  {
3    uint chunkSize;
4    uint chunkSizeBytes;
5
6    uint64 EncodeData(
7      byte* iData,
8      uint64 iDataSize,
9      byte* oData,
10     DataModel& model);
11
12   void DecodeData(
13     byte* iData,
14     byte* oData,
15     uint64 oDataSize,
16     DataModel& model);
17 }
```

Several things are worth noting here. There are no streams, as one might expect from program that supposed to work which large datasets, because for effective parallel processing as much data as possible must be loaded in memory at the same time. There are `chunkSize` and `chunkSizeBytes`, that are needed for implementing splitting logic (it will have different implementations on GPU and CPU, but will lead to same results). Function `DecodeData` needs `oDataSize`, that is needed for determining correct ending of decoded data stream. Finally, Datamodel is reference on instance that describes data model that should be used. There are several simple data models predefined:

- `StaticBinaryModel` - probability of zero bit is given and fixed.
- `AdaptiveBinaryModel` - probability of zero bit is modified during execution according to already-encoded symbols.
- `AdaptiveByteModel` - table of probabilities for each byte symbol. Is adapting during execution.
- `ContextAdaptiveBinaryModel` - several `AdaptiveBinaryModel`s, which one to choose is determined from previous symbols.

All of them are defined in the file 'DataModels.cu'. They are compiled for both CPU and GPU - it is very useful to define some simple helper functions inside data model class to use them in both CPU and GPU coders.

There are two key classes, that implement `ArithmeticCodec` - `CPUCodec` and `GPUCodec`. Not all of them implement every method from `ArithmeticCodec` - for example, `AdaptiveByteModel` is supported only by CPU codec. `CPUCodec` uses "only" CPU for encoding, and `GPUCodec` - "only" GPU. If you want to mix both approaches, it is advised to write some wrapper, for example `MixedCodec`, that will contain `CPUCodec` and `GPUCodec` instances and delegate work between them - should be fairly simple to do.

Both codecs implement encoding logic for static, adaptive and context-adaptive binary models. `CPUCodec` also supports adaptive byte model and contains decoding logic for all of them as well. All variants support situation, where potential output is actually bigger than input - in this case, "00 00" is written instead of output chunk size, and a copy of corresponding input data chunk after that.

7.2 Developer manual

There is wide variety of scenarios, where our algorithm may be useful. You will need to modify the code itself to achieve optimal performance on your system. The most important file for customization is "_Constants.h" - it contains several `#define` directives, which define codec behaviour. The single most important constant is `THREADS_PER_BLOCK` - it is amount of threads that will be spawned by CUDA per single block. Feel free to experiment with it - higher values will lead to better performance, but larger memory requirements. On our system, 32 and 64 were the most effective values, although on more modern devices this value will be higher.

If you want to compress some large data stream and send it to another PC, or pack in a file, you will need to perform some extra work - namely, you probably will need to manually split data into large, "level 0" chunks, and call `EncodeData` for each one. Moreover, you will need to add some logic to store uncompressed size of whole data stream somehow (for example, in compressed file header). Nice example of it is given in file "helpers.cpp". The method "EncodeFile" divides files in large "Level one chunks" - 32 megabytes each - and then encodes them with selected encoding method. After that it stores safety header and compressed data stream in the resulting file itself. The method "DecodeFile" does corresponding inverse operation as well.

If you want to write your own data model, you need to:

1. Write `MyDataModel` class
2. Implement corresponding overloads of `EncodeData` and `DecodeData` functions in `CPUCodec` and/or `GPUCodec`.
3. For `GPUCodec` you will possibly need to implement kernel itself in separate file with extension ".cu".

If you have followed the rules, you may even use your new data model with our sample program and test your results.

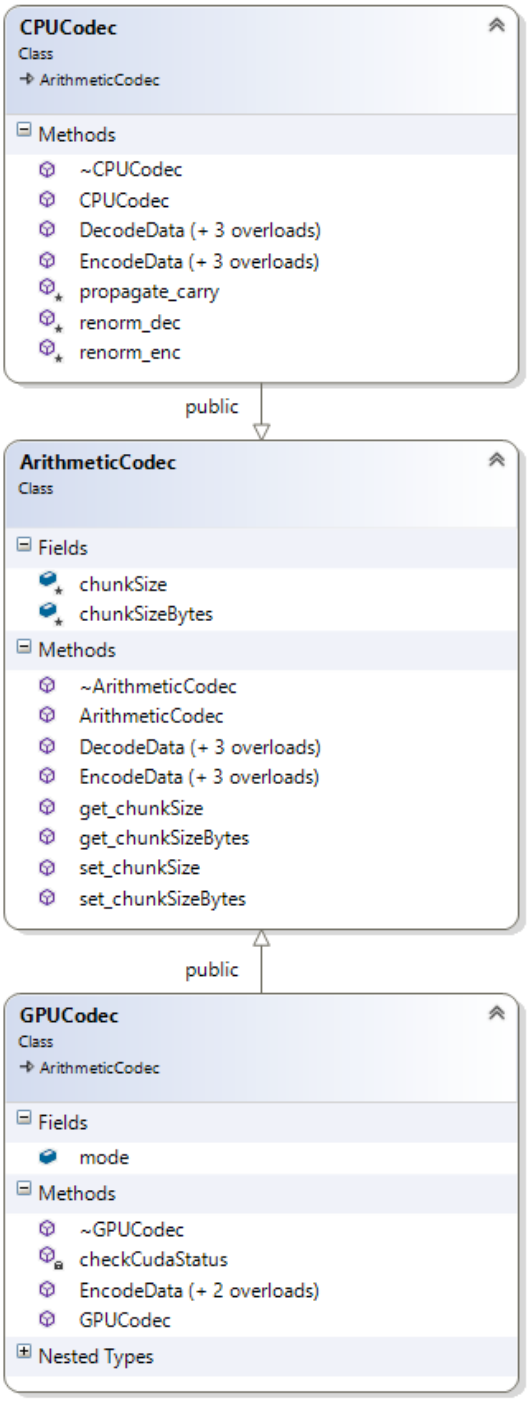


Figure 7.1: GARCoder static library core classes diagram

7.2.1 Compiling

Program was written with using as few advanced c++ constructs as possible. The only modern class in entire program (from C++11) `std::chrono::high_resolution_clock` - without it, program should be compilable by older compilers. Only STL and Cuda libraries are used. You need to have CUDA Toolkit (at least version 4.0) installed and correctly configured on your system (https://developer.nvidia.com/cuda-downloads) to compile kernels.

Folders description:

- "Bin" contains compiled Windows executable
- "Tests" contains several files for compression testing
- "Code - MSVS Solution" archive contains Microsoft Visual Studio 2013 solution
- "Code - clean" archive contains source files only

Note: "main.cpp" and "helpers.cpp" are parts of the sample program. Every other code file - part of the library.

7.3 User manual

There is a sample command line program provided, which is able to encode file, decode it, compare two files to find out if they are identical. It's interface is straightforward, you just need to follow command line instructions (not all combinations of parameters work, though. For example, decoding is implemented only on CPU). It also measures execution time using `std::chrono::high_resolution_clock` from C++11 standard. This program was used in "Testing" chapter of this paper. Its source code will provide you an example of correct using GARCoder library.

To run the program you will need to have CUDA-compatible NVIDIA GPU in your system with Compute Capability at least 2.1. You can find list of all CUDA-enabled GPUS here https://developer.nvidia.com/cuda-gpus. It is advised to have the latest Display Driver installed for optimal performance.

Known bugs: after many iterations under some circumstances, GPU may refuse to allocate huge chunks of memory due to memory fragmentation. In this case command line will show "Bad Allocation" message. Usually restarting program is enough to resolve the issue.

Program was tested with NVIDIA GeForce GTX 460M.

8. Tests

8.1 Environment

Our AC implementations were tested in terms of performance, correctness and compression efficiency on the following data sets:

- Generated files - random bit sequencies with different zero probabilities $p(0) = 0.001, 0.01, 0.05, 0.25, 0.5$. Size of each file is 128 MB = 134 217 728 bytes
- Generated files - random bit sequencies with zero probability $p(0) = 0.25$ and sizes from 1 KB to 128 MB.
- 1995 CIA World Fact Book - english text, txt, 2.8 MB = 2 988 578 bytes
- Audio sample - wav (Waveform Audio File), 100 MB = 104 768 684 bytes
- Image - high-definition png, 3 MB = 3 094 761 bytes
- Video sample - high-definition yuv (uncompressed video), 748 MB = 783 820 800 bytes

During every test every file was compressed and decompressed. Then every decompressed file was compared with source file to be exactly same. Every file was loaded to RAM before coding so hard disk read/write operations duration does not affect results. Tests were executed 10 times, the best and the worst results were discarded and the mean of remaining 8 measurements was used as the result.

8.2 Compression ratio

First of all, to give context about different compression methods, let's compare different data models - binary static, binary adaptive, simple binary context-adaptive (just use 8 last bits as context), and adaptive with byte symbols.

Figure 8.1 shows compression ratios of different data models with different data types. Results are unsurprising - algorithms with "memory" (context-adaptive ant byte table) perform significantly better, than plain algorithms. In real applications, context-adaptive algorithm should perform way better - our implementation is very simple, while real implementations have specific context selection rules for getting best results out of their data. Also note, that plain adaptive algorithm always performs slightly better than static. Static data model knows exact zero probability right at the beginning, while adaptive model starts with uniform probability - but adaptive model also reflects changes in data stream with time, while static only looks at data stream as a whole. So, adaptivity is more important than knowing exact probabilities.

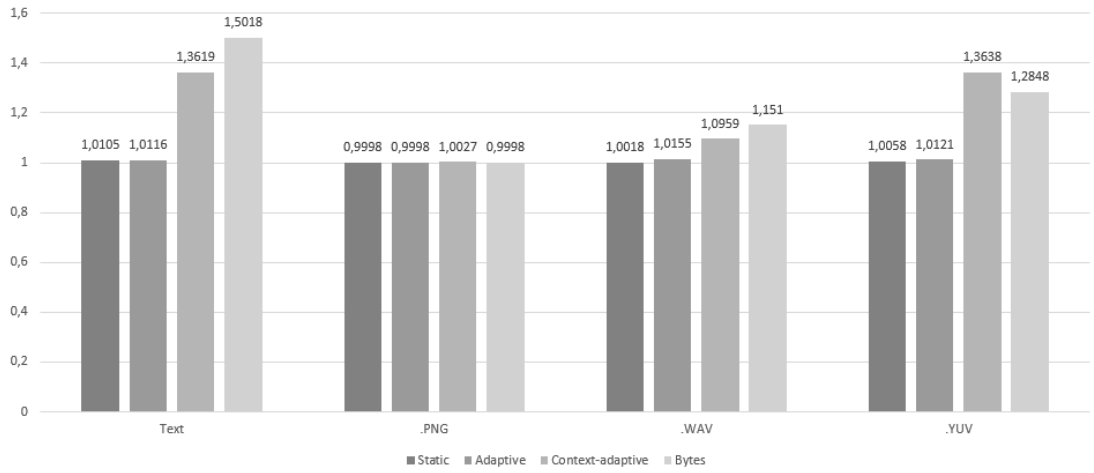


Figure 8.1: Compression ratios for different data models. Higher is better.

8.3 Performance

Finally, lets look at our results.

Figure 8.2 shows performance of different approaches while compressing large text file (The 1995 CIA World Factbook). Even at 3MB data size there is solid lead (70%) of GPU version for simple algorithms. Surprisingly, more complex context-adaptive version doesn't perform as well.

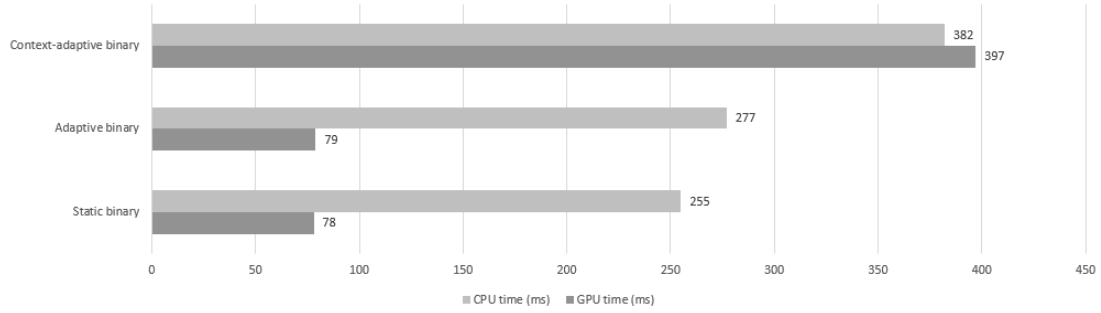


Figure 8.2: Time needed to compress 3MB .TXT file. Shorter is better.

Results for compressing .PNG file you can see at figure 8.3. It shows almost exactly the same picture as previous chart. This is important, because, while chunks of .TXT file almost always were successfully compressed, chunks of .PNG file usually weren't compressed at all. This means, that result of compression doesn't affect compression performance at all.

After that much larger .WAV file (about 100 MB) was compressed. Results are displayed in chart 8.4. GPU still performs about 70% better than CPU on simple algorithms, and, suddenly, context-adaptive GPU version is almost 2 times faster than CPU one as well.

Finally, figure 8.5 shows results for the largest file from our dataset, .YUV file (about 800 MB). GPU performs better with all three algorithms used - static binary coding is 81% faster, adaptive binary coding is 86% faster, and context-adaptive binary coding with 8 bits context is 46% faster. Those are results using GPU exclusively - so, using GPU and CPU at the same time, context-adaptive algorithm will perform about 65% faster than using only CPU.

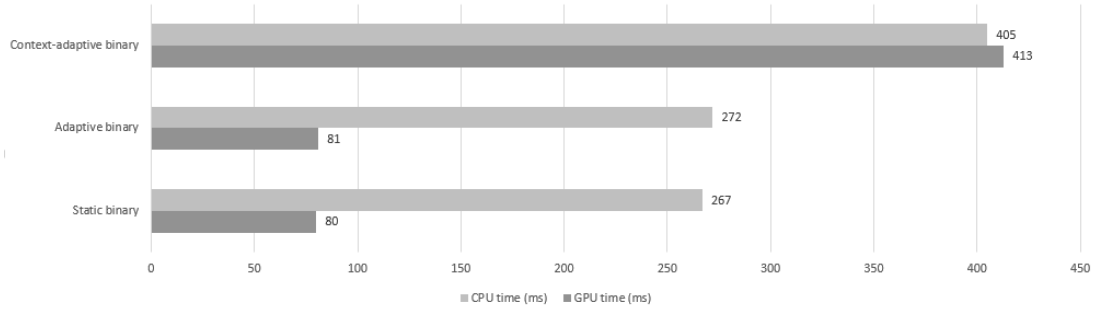


Figure 8.3: Time needed to compress 3MB .PNG file. Shorter is better.

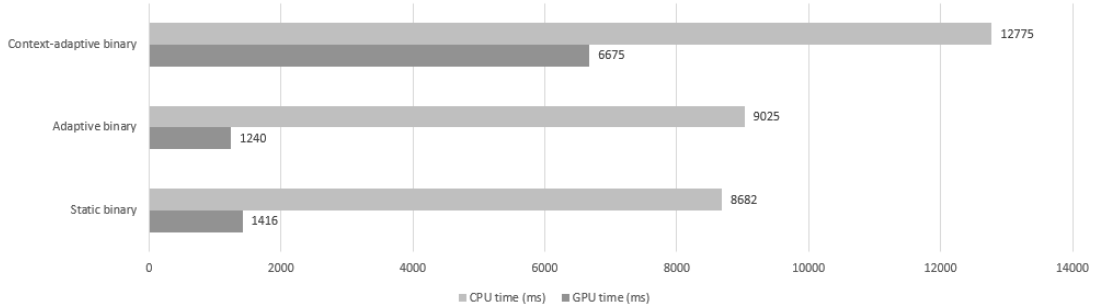


Figure 8.4: Time needed to compress 101MB .WAV file. Shorter is better.

What is wrong with context-adaptive algorithm, though? The problem is in memory usage. Thread needs to remember probabilities for every possible context - 256 variants for 8 bit context. Actually, not only probabilities, but 4 more internal variables that adaptive model is used. It is already 5120 bytes. And every thread on GPU needs his own copy of probabilities table to update it. 5120 bytes is too much to store in registers (the fastest memory), so CUDA tries to store it into shared memory. But with 64 threads per warp, 327680 bytes need to be stored in memory - it is too much, so CUDA stores entire probabilities table in global memory - which is not ideal, as its values are always accessed and changed. So, to achieve optimal performance, there is requirement to fit all DataModel variables into shared memory somehow. When context was cut from 8 bits to 4, probability table became 16 times smaller, so kernel needs 16 times less shared memory - and indeed, when tests show that with 4bit context GPU performance gain was the same as with simpler algorithms (70-80%), although compression rate has slightly fallen.

Bear in mind, though, that everything becomes much simpler if we will use non-adaptive version. Threads don't need to update context-probability table, so all threads may read from the same table - that means that all shared memory (48KB for modern devices) can be used to store probability tables. Sometimes it is preferable to use non-adaptive version - in such cases, GPU will provide acceleration even for large or complex probability models.

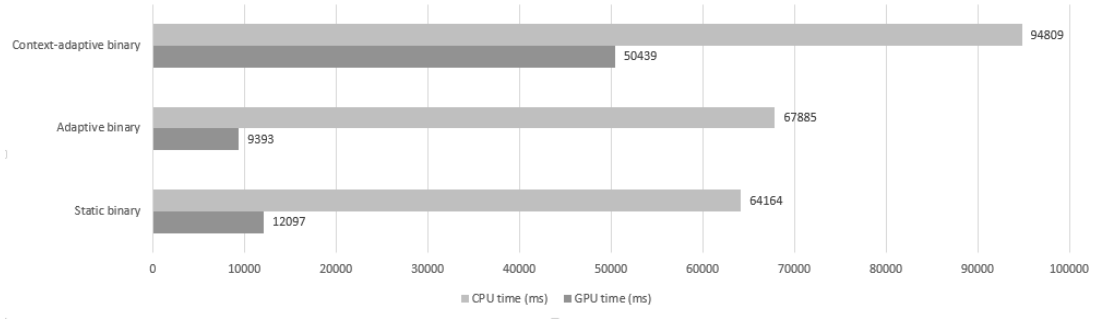


Figure 8.5: Time needed to compress 765MB .YUV file. Shorter is better.

Conclusion

We have successfully developed static, adaptive and context-adaptive binary arithmetic encoder, adapted for use on GPU. It has shown significant acceleration as opposed to serial arithmetic coding (FastAC), mainly on large data streams - up to 13x in ideal circumstances - while negative effect on compression efficiency is almost negligible on large data chunks (less than 0.2% on 16Kb chunks, adaptive versions performed even better). As opposed to previous research, we have used binary data model, which allowed our basic algorithm to be much more memory-effective - it doesn't use shared memory, which means that shared memory is available for some complex data models.

Although results are good, our implementation is more about proof-of-concept, than real-world ready algorithm. It still needs to be adapted for particular data types to be useful, and some more sophisticated adaptive data models may be problematic to parallelize due to memory restrictions, as it was shown with 8-bit-context-adaptive data model in our work. At the same time, it shows that static context data models are ideal for parallelization, as they offer good compression efficiency and dont need as much memory as adaptive variants.

CUDA has proven to be mature, stable, powerful and fairly easy-to-use technology. We have chosen very parallelization-unfriendly serial algorithm (linear, very fast even in serial version, data depend on each other), and still succeeded - so there must be many more algorithms that can be accelerated by unmatched performance power of GPU.

In future we will try to develop parallel AC with some more practical data models, as well as explore possibility of unification GPU and CPU code - the first step to it was shown in our DataModels, which use same functions on both devices. Probably, it is possible to unify entire codebase and just switch between cpu and gpu modes as needed.

Bibliography

[1] Shannon, C. *A Mathematical Theory of Communication.* The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, July, October, 1948

[2] Abramson, N. *Information Theory and Coding.* McGraw-Hill Book Co., Inc., New York, 1963.

[3] Jelinek, F. *Probabilistic Information Theory.* McGraw-Hill Book Co., Inc., New York, 1968.

[4] Schalkwijk, J. *An Algorithm for Source Coding.* IEEE Trans. Info. Theory IT-18, 395 (1972).

[5] Cover, T. M. *Enumerative Source Coding.* IEEE Trans. Info. Theory IT-19, 73 (1973).

[6] Rissanen, J. J. *Generalized Kraft Inequality and Arithmetic Coding.* IBMJ. Res. Develop. 20, 198-203 (1976).

[7] Pasco, R. *Source Coding Algorithms for Fast Data Compression.* Ph.D. Thesis, Department of Electrical Engineering, Stanford University, CA, 1976.

[8] Rissanen, J. J. and Langdon, G. G., Jr. *Arithmetic Coding.* IBM J. Res. Develop. 23, 149-162 (1979).

[9] Nigel, G. and Martin, N. *Range encoding - An algorithm for removing redundancy from a digitized message.* Video and Data Recording Conference, Southampton, UK, July 24–27, 1979.

[10] Howard, P. G. and Vitter, J. S. *Parallel Lossless Image Compression Using Huffman and Arithmetic Coding.* Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 23-26, 1992, 299-308.

[11] Mahapatra, S. *Parallel implementation of multialphabet arithmetic coding algorithm.*

[12] Balevic, A.; Rockstroh, L.; Wroblewski, M.; Simon, S. *Using Arithmetic Coding for Reduction of Resulting Simulation Data Size on Massively Parallel GPGPUs.* Recent Advances in Parallel Virtual Machine and Message Passing Interface 2008, pp 295-302, ISBN 978-3-540-87474-4

[13] Rusňák, V. *Design and Implementation of Arithmetic Coder for CUDA Platform.* Ph.D. Thesis, Faculty of informatics, Masaryk University, Brno, 2010.

[14] Xiao, W.; Zhou, Y.; Jizheng, X.; Guangming, S. *A scheme of parallel arithmetic coding* . Circuits and Systems (ISCAS), 2011 IEEE International Symposium, 15-18 May 2011, ISBN 978-1-4244-9473-6

[15] Chen, L.; Fang, Y.; Huang, B. *Accelerating arithmetic coding on a graphic processing unit.* Proc. SPIE 8183, High-Performance Computing in Remote Sensing, 81830B (November 02, 2011); doi:10.1117/12.897112

[16] Harris, M.; Sengupta , S.; Owens, J. D. *Parallel Prefix Sum (Scan) with CUDA*. GPU Gems 3, Chapter 39, pp. 145-158, ISBN 978-0321515261, 2007

Attachments

Attached CD-ROM Contents

- **thesis.pdf** - copy of thesis
- **Bin** - compiled windows executable
- **Code - clean** - source files only
- **Code - MSVS Solution** - Microsoft Visual Studio 2013 Solution
- **Tests** - Several files for testing
  - **small.txt** - small file (16KB) where every byte is $0x67$
  - **test.txt** - The 1995 CIA World Factbook text file, 2935KB
  - **test.png** - PNG picture (3023KB). Rarely compresses (probably is already compressed by .png format itself), but useful for testing worst-case scenario.
  - **test.wav** - WAV audio file (102314KB)

Reference GPU specification

Listing 8.1: GPU specs

```
Device 0: "GeForce GTX 460M"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1536 MBytes
     (1610612736 bytes)
  ( 4) Multiprocessors x ( 48) CUDA Cores/MP:    192 CUDA Cores
  GPU Clock rate:                                1417 MHz (1.42
     GHz)
  Memory Clock rate:                             1250 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 393216 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D
     =(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x
     2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x
     65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy
     engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows
     Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode: < Default (multiple host threads can use ::
     cudaSetDevice() with device simultaneously) >

  CUDA Driver = CUDART, CUDA Driver Version = 5.5, CUDA Runtime
     Version = 5.5, NumDevs = 1, Device0 = GeForce GTX 460M
```

Basic encode kernel

Listing 8.2: encodeKernel.cu

```
__global__ void encodeKernel
(unsigned const bit_0_prob,
unsigned char* iData,
unsigned const iSize,
unsigned char* oData,
unsigned* oChunksSizes,
unsigned const chunkSize)
{
  int chunkId = blockIdx.x*blockDim.x + threadIdx.x;
  iData += chunkId*chunkSize;
  oData += chunkId*(chunkSize + 2);

  unsigned char* ac_pointer = oData + 2;
  unsigned length = AC_MaxLength;
  unsigned base = 0;
  if (chunkId*chunkSize < iSize) {
    unsigned sizeToProcess = min(chunkSize, iSize - chunkId*
       chunkSize);
    unsigned char* end = ac_pointer + sizeToProcess;

    for (unsigned i = 0; i < sizeToProcess; ++i)
    {
      unsigned char byte = iData[i];
      for (unsigned j = 0; j < 8; ++j)
      {
        unsigned x = bit_0_prob * (length >> BM_LengthShift);
        if ((byte & (1 << j)) == 0)
          length = x;
        else
        {
          unsigned init_base = base;
          base += x;
          length -= x;
          if (init_base > base) //Overflow? Carry.
          {
            unsigned char * p;
            for (p = ac_pointer - 1; *p == 0xFFU; p--)
              *p = 0;
            ++*p;
          }
        }
        if (length < AC_MinLength) //Renormalization
        {
          do
          {
            if (ac_pointer >= end)
              break;
            *ac_pointer++ = (unsigned char)(base >> 24);
            base <<= 8;
          } while ((length <<= 8) < AC_MinLength);
        }
      }
    }
    if (ac_pointer < end) {
```

```
54        unsigned init_base = base;
55        if (length > 2 * AC_MinLength) {
56          base += AC_MinLength;
57          length = AC_MinLength >> 1;
58        }
59        else {
60          base += AC_MinLength >> 1;
61          length = AC_MinLength >> 9;
62        }
63
64        if (init_base > base)
65        {
66          unsigned char * p;
67          for (p = ac_pointer - 1; *p == 0xFFU; p--)
68            *p = 0;
69          ++*p;
70        }
71        do
72        {
73          if (ac_pointer >= end)
74            break;
75          *ac_pointer++ = (unsigned char)(base >> 24);
76          base <<= 8;
77
78        } while ((length <<= 8) < AC_MinLength);
79      }
80
81      unsigned codeBytes = 0;
82      if (ac_pointer < end) {
83        codeBytes = unsigned(ac_pointer - oData);
84      }
85
86      oData[0] = codeBytes >> 8;
87      oData[1] = codeBytes;
88      oChunksSizes[chunkId] = codeBytes;
89    }
90  }
```