EFFICIENT PARALLEL TEXT COMPRESSION ON GPUS

A Thesis

by

XIAOXI ZHANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Science

EFFICIENT PARALLEL TEXT COMPRESSION ON GPUS

A Thesis

by

XIAOXI ZHANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Dmitri Loguinov
Committee Members,	Donald Friesen
	A. L. Narasimha Reddy
Head of Department,	Duncan M. Walker

December 2011

Major Subject: Computer Science

ABSTRACT

Efficient Parallel Text Compression on GPUs. (December 2011)

Xiaoxi Zhang, B.E., National University of Defense Technology

Chair of Advisory Committee: Dr. Dmitri Loguinov

This paper demonstrates an efficient text compressor with parallel Lempel-Ziv-Markov chain algorithm (LZMA) on graphics processing units (GPUs). We divide LZMA into two parts, match finder and range encoder. We parallel both parts and achieve competitive performance with freeArc on AMD 6-core 2.81 GHz CPU. We measure match finder time, range encoder compression time and demonstrate real-time performance on a large dataset: 10 GB web pages crawled by IRLbot. Our parallel range encoder is 15 times faster than sequential algorithm (FastAC) with static model.

To my parents and sister

ACKNOWLEDGMENTS

I would like to sincerely thank Dr. Loguinov for giving me the opportunity to work with him. I believe that working with him has prepared me to deal with any situation both professional and personal that I may encounter in the future. I would also like to thank Dr. Reddy and Dr. Friesen for being on my committee.

I would also like to thank my colleagues at Internet Research lab, especially Sid and Sadhan. Finally, all my achievements have been a direct result of the support and sacrifices of my parents and my sister.

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
	A. Our Contribution	2
II	RELATED WORK	4
III	BACKGROUND AND OUR APPROACH	6
	A. Basics	6
	B. Design Overview	8
IV	PARALLEL FINDER AND MERGER	10
	A. Parallel Match Finder	10
	1. Find Longer Match	11
	B. Merge Contiguous Codewords	12
	1. Merge Case 1	13
	2. Merge Case 2	16
V	PARALLEL RANGE ENCODING ON GPUS	18
	A. Sequential Arithmetic Coding	18
	B. Character Frequency Statistics	23
	C. Parallel Big Number Multiplication	24
	D. Parallel Big Number Addition	27
VI	RESULTS AND ANALYSIS	33
VII	CONCLUSION	36
REFERENCES		37
VITA		39

LIST OF TABLES

TABLE Page

I Arithmetic encoding example 19

II Arithmetic decoding example 20

LIST OF FIGURES

FIGURE Page

1 Parallel text compressor design on GPUs. 9

2 Find longer match. 11

3 Merge case 1. 13

4 Our merge process (compute element counters with prefix reduction and output corresponding positions with prefix scan). 14

5 Merge case 2. 16

6 Big number multiplication. 25

7 Example of big number multiplication. 26

8 Big number parallel addition output phase. 27

9 Big number parallel addition merging phase. 27

10 Compression rate and ratio for parallel LZMA (pLZMA). 34

11 Compression rate and ratio for parallel arithmetic coding. 34

12 Compression rate and ratio comparison with FreeArc (0.66) on 6x2.8 GHz core, FreeArc (0.66) on 1x2.8GHz core, winrar (4.01), winzip (15.5), gzip (1.3.12) and 7zip (9.20). 35

CHAPTER I

INTRODUCTION

The textual content of the Web is growing at a such stunning rate that compressing it has become mandatory. In fact, although modern technology provides ever increasing storage capacities, the reduction of storage usage can still bring rich dividends because of its impact on the number of machines/disks required for a given computation.

Parallel processing is a widely used technique for speeding up many algorithms. Recent advances in Graphics Processing Units (GPUs) open a new era of parallel computing. Commercial GPUs like NVIDIA GTX 480 has 480 processing cores and can achieve more than a teraflop of peak processing power. Traditionally, GPUs are mainly used for graphical applications. The release of the NVIDIA CUDA programming model makes it easier to develop non-graphical applications on GPUs. CUDA treats the GPU as a dedicated coprocessor of the host CPU, and allows the same code to be simultaneously running on different GPU cores as threads.

As we know, compression speed and ratio is a trade-off. We can improve compression ratio by searching more repeated substrings at a larger distance, which depends on faster compression speed. Therefore, we propose parallel compression algorithm on GPUs to speedup compression speed and then compression ratio.

However, three problems make the development of efficient parallel compression implementations on GPUs nontrivial. The first problem is that parallel compression algorithm is hard to achieve same compression ratio with corresponding sequential algorithm. The first reason is that typical parallel compression algorithm is to split data to blocks and assign a thread to each block which definitely sacrifices compres-

---

The journal model is *IEEE Transactions on Automatic Control.*

sion ratio because it cannot find the repeated substrings between different blocks. For example, Intel IPP compression library [1] implements its parallel algorithm in this way, which leads to IPP gzip compression ratio decrement from 5.7 to 4.6 (uncompressed size/compressed size). The second reason is that it is difficult to merge parallel matching results effectively and efficiently, which depends on merge strategy and parallel scan algorithm. The second problem is that efficient compression algorithm like LZMA is not inherently parallel. Two reasons cause this problem. One reason is that the data dependencies in LZMA algorithm require the result of step i before step $i+1$ can start. The other reason is that data matching algorithm based on hash table search cannot be translated to the highly parallel environment of the GPUs. The third problem is that we need new design on GPUs to resolve non-natural parallel compression algorithm, memory conflict and barrier synchronization since GPU architecture is different with CPU.

A. Our Contribution

In this paper, we propose novel design and algorithm to resolve the above problems. To improve compression ratio, we search redundant data in larger hash table for better compression ratio. Also, we design parallel match finder to match longer substring and merger to solve shorter match problem of parallel match finder.

To achieve fast compression speed, we split LZMA to two phases and parallel them separately: one is parallel matching and merging, the other is parallel range coding. In phase one, we find duplicate substrings by parallel building and searching hash table [2], and then merge matching results to keep same compression ratio with sequential algorithm. In phase two, we encode unmatched substrings, matched offset and length with parallel range encoding.

We implement our parallel algorithm and achieve high performance on GPUs. To achieve high performance on GPUs, our parallel algorithm design is based on classical parallel algorithms such as prefix sum, parallel reduction and compaction which are optimized and perform with high performance on GPUs. Also we involve other optimization techniques in our algorithm, e.g., avoiding memory conflict by padding data, minimizing the need for barrier synchronization by using warp-wise and block-wise execution.

CHAPTER II

RELATED WORK

Numerous sequential algorithms have been invented to improve compression ratio and compression speed. For text compression, basically there are two kinds of algorithm family. One is dictionary methods (e.g., LZ77, LZ78 and LZP) and the other is statistical methods (e.g., Huffman coding, arithmetic coding [3] and PPM). Modern compressors usually combine them together, e.g., Gzip combines LZ77 and Huffman coding, LZMA combines LZ77 and arithmetic coding. Other algorithms include various compression techniques. For example, Burrows-Wheeler Transformation [4] (BWT, also called block-sorting) is based on a permutation of the input sequence, and Bzip2 is an implementation of BWT. The performance of these algorithms on textual web are tested with various switches on state-of-the-art compressors and compared in paper [5], where LZMA is proved to be the best one.

Preprocessors/filters are involved to eliminate large distance duplicate data using hash table search. BMI [6] works well with gzip since BMI can find long distance redundant substrings which cannot be found by gzip since it only searches repeated substrings in 32KB blocks. However, BMI is pretty slow becuase it uses a naive hash function.

Novel techniques keep coming up. FreeArc [7] is the best contemporary compressor, which involves more than 11 algorithms and filters, and LZMA is one of them. Srep is used as the preprocessor in FreeArc to match large chunk (default 512 bytes), which is like BMI but uses strong hash function: SHA-1 and MD5. And grzip is used as text compressor which integrates 4 algorithms: LZP, BWT, WFC and EC.

All algorithms and techniques mentioned above are essentially sequential. FreeArc can parallel run on multi-core CPU while their approach is splitting a big file to blocks

and then assigning them to different cores, therefore the algorithm is in fact sequential. PBZIP2 [8] is a parallel Bzip implementation which is inherently feasible since bzip is based on block split and sorting. However, the performance is not good enough because of the natural limitation of bzip. Bzip compression speed is much slower compared with optimized LZMA. Parallel arithmetic encoding algorithm was proposed in paper [9]. But no experiment result was showed in this paper. The method they presented is not feasible in practice because of limited machine precision, and the most important issue is that their mathematical derivation actually has a fatal defect.

Variety of optimization techniques of fundamental parallel algorithm on GPUs are proposed. Parallel compaction, prefix sum, sorting and parallel reduction algorithm on GPUs are proposed in paper [10, 11, 12, 13], and most of their implementations are provided in related library. These algorithms show very impressive performance on GTX 480, e.g., parallel reduction can finish adding 16 millions elements in 0.77 ms. Two efficient histogram algorithms designed for CUDA have been presented in paper [14]. The first algorithm is based on simulating a mutex by tagging the memory location and continuing to update the memory until the data is successfully written and the tag is preserved. It is designed for NVIDIA GPUs of 'compute capability' 1.0 and atomic memory updates has been provided for GPUs of 'compute capability' 2.0. The second method maintains a histogram matrix of B $\times$ N size, where B is the number of bins and N is the number of threads. This provides a collision free structure for memory updates by each thread.

Real-time parallel hashing on GPUs [2] is implemented with hybrid approach combining classical perfect hashing and cuckoo hashing. This efficient data-parallel algorithm combines the advantages of fast on-chip memory and large global memory, it takes 107 ms building and 59.1 ms retrieving time for large hash table of 32 millions elements.

CHAPTER III

BACKGROUND AND OUR APPROACH

The modern GPUs' massive parallelism architecture offers very high throughput on certain problems, and General-purpose computing on graphics processing units (GPGPU) gives its near universal use, which means that GPU is a cheap and ubiquitous source of processing power. Therefore we leverage GPU powerful computing capability and choose GPU as our parallel architecture. To achieve faster duplicate data matching speed, we introduce parallel match finder to search redundant data and implement the parallel match finder based on parallel hash table on GPUs. After finding out the duplicate substrings, we parallel merge the result to minimize the merging time, and design optimal merging method to keep compression ratio same with corresponding sequential algorithm. In last phase, we use our parallel range encoder to speedup the encoding of unmatched substrings, match offset and match length.

A. Basics

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the input (uncompressed) data stream. It searches repeated substrings in a sliding window, and then a match is encoded by a pair of numbers called a length-distance pair.

Arithmetic coding stores frequently used characters with fewer bits and not-so-frequently occurring characters with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding such as Huffman coding in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number.

Arithmetic coding has better compression ratio than Huffman coding since it is able to compress data at rates much better than 1 bit per byte when the symbol probability are right.

Range coding [15] is a variation of arithmetic coding, it performs renormalization in bytes instead of bits thus running twice faster, and with 0.01% worse compression than a standard implementation of arithmetic coding.

LZMA is the combination of LZ77 and arithmetic coding. The dictionary compressor produces a stream of literal symbols and phrase references, which encodes one symbol at a time by the range encoder, using a model to make a probability prediction of each bit.

The GPU has a multi-core processor containing an array of Streaming Multiprocessors (SMs). A SM is an array of SPs, which consists of 8 Streaming Processors (SPs), along with two more processors called Special Function Units (SFUs). CUDA, Compute Unified Device Architecture, is a general-purpose hardware interface designed to let programmers use NVIDIA graphics hardware for purposes other than graphics in a more familiar way. At the hardware level, the GTX 480 processor is a collection of 15 multiprocessors, with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 32 processors inside it. At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each a SIMD processor. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors. Access to global memory has a high latency (in the order of 400-600 clock cycles), which makes reading from and writing to the global memory particularly expensive. The performance of global memory accesses can be severely reduced unless access to adjacent memory locations is coalesced. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for

a specific GPU. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared.

CUDA can be used to offload data-parallel and compute intensive tasks to the GPU. The computation is distributed in a grid of thread blocks. All blocks contain the same number of threads that execute a program on the device 2, known as the kernel. Each block is identified by a two-dimensional block ID and each thread within a block can be identified by an up to three-dimensional ID for easy indexing of the data being processed. The block and grid dimensions, which are collectively known as the execution configuration, can be set at run-time and are typically based on the size and dimensions of the data to be processed.

Each GPU thread only reads 4 bytes data in our implementation to achieve coalesced global memory access. Let tid denote thread id, bid denote the block id, N denote the maximum thread number of a block (i.e., block size), p denote data address, the mapping between thread id and data position is $p = s + N \times bid + tid$, where tid is incremental number from 0 to $N-1$ by 1, s is data start address. Assume $N = 128$, when thread 1 finished read data from the first slot, it will move to position $s+128$, and then $s+256$, and so on. Based on this GPU thread access model, we can find the match result is only 4 bytes, so we need to merge them with parallel merger.

B. Design Overview

The main stages of our parallel text compressor is described in Fig. 1. Firstly CPU loads file data to host memory, and then we copy the data from host (CPU) to

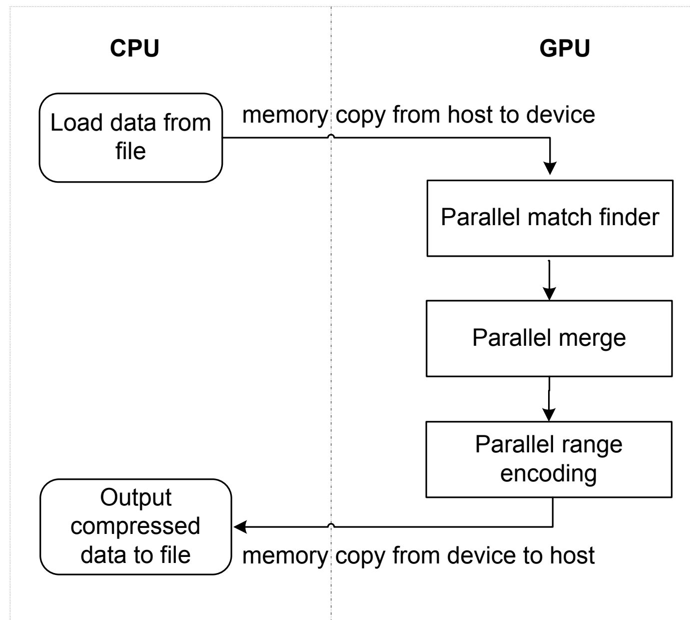

Fig. 1. Parallel text compressor design on GPUs.

device (GPU). Secondly we parallel find repeated data with our match finder and encode the match with LZ77 method. We parallel match finder by paralleling hash table building and search on GPUs. Thirdly we merge the parallel matching result of previous phase to achieve same compression ratio with sequential algorithm. The reason is parallel match finder can only find out multiple short match simultaneously, in fact lots of matches are contiguous and we can expand the match length by merging them. Fourthly, after merging, we have unmatched literal, matched distance and match length, and then we encode them with range coding. To speedup, we design and implement parallel range encoding on GPUs. Finally we copy the compressed data from device to host and output them to the compressed file.

CHAPTER IV

PARALLEL FINDER AND MERGER

A. Parallel Match Finder

We design parallel match finder based on parallel hash table [2] and we use their source code. The parallel hash table input are integer keys and integer values. Basically they use cuckoo hashing, the hash function is ((constants.x XOR key) + constants.y) mod kPrimeDivisor, where constants.x and constants.y respectively represent two different constants which are generated by random number function. They have 3 hash functions with 6 constant, parallel build 3 hash tables in shared memory and them write to global memory. And retrieval need to search the 3 tables. The parallel multi-value construction produces a hash table in which a key k is associated with a count c_k of the number of values with key k, and an index i_k into a data table in which the values are stored in locations $i_k ... i_k + c_k - 1$. The multiple hash table building process has three phases. Firstly they sort keys and values. Secondly they find first key-value pair for each key and assign a unique index for each of the keys, and then do compaction to find out i_k and c_k. Thirdly they find out all unique keys and their associated values to build unique key-value hash table, and then we search from this hash table, unique keys location i_k and the number c_k can be retrieved.

For our match finder, firstly we construct keys and values for the hash table. We convert every 4 chars from input stream to an integer key, and put start address of the 4 chars to a value. Using LZ77, we need to output match length and the offset of key address and closest match address of the key. Since we hash every 4 chars to a key, so our match length is 4. The problem is we do not have the offset since the result of multi-value hash table search is the key first occurrence position L_x in sorted

Initial string					
abcd	1234	abcd	efgh	abcd	1234

Default match					
abcd	1234	8	efgh	8	16

Optimal match					
abcd	1234	8	efgh	16	16

Fig. 2. Find longer match.

value array and the number of values with this key L_y. The solution is we combine sorted value array with L_x and L_y. After hash table construction, we have sorted key array and sorted value array. In searching hash table phase, we have L_x and L_y, so we can fetch the value from sorted value array (i.e., first occurrence address of the key) based on L_x, the index of all the addresses of key k in sorted value array is from L_x to $L_x + L_y$. We use a flag to indicate if the key can be matched or not. For our match finder, if the key only occurs once, we directly write the key to output array, and set the flag as 0; if the key occurs multiple times, we do binary search from the sorted value array and get the closest match key, computer the offset of current key and the closest match key and write the offset to output array, and set the flag value as 1.

1. Find Longer Match

An example is depicted in Fig. 2. In this case, the third "abcd" has two matched elements. We default choose the closest one, as the second line. However, it causes shorter match substring. If we choose the first "abcd", we can merge last two elements and the match length can be expanded from 4 to 8, which is the optimal case.

To match longer string, our approach is that we iterate more previous matched

positions for contiguous substrings and try to find longer sequence with same match position if their current match substring are not consecutive. Firstly we compute 32 closest matching offset and load to shared memory. After building hash table, we have a sorted hash value array, and another location array which holds start address of all unique elements and numbers of their duplicate elements. We calculate the distances of current element with previous 32 duplicate substrings and save to shared memory for comparing. Secondly, we iterate twice to find longer contiguous elements with common matching offset. In first iteration, we assign thread i to access $data[2i+1]$, thread i compare data[i] with two adjacent elements $data[2i-1]$ and $data[2i+1]$, we compare previous 32 match position for the three elements in order since the 32 matching offset is ascending ordered, if all three match offset is same, we update this position with new position, otherwise choose position matching 2 elements and update current position. In the second iteration, we assign threads i to $data[2i]$ to compare with two adjacent elements, repeat the same process with first iteration.

The number of previous matching elements we can compare is limited by shared memory size. Assume we have 256 threads to parallel run, each thread loads 16 previous match positions for three elements, each match positions is 4 bytes, so shared memory usage is $256 \times 32 \times 4 = 32K$ bytes. GTX 480 has 48 KB shared memory, and shared memory is also used for registers, so 32 is close to the maximum value.

B. Merge Contiguous Codewords

After parallel hash construction and retrieval, we get lots of matched and unmatched substrings. Since each thread only matches one substring, so compared with sequential algorithm, our match length is shorter. Therefore, we need to merge contiguous matched substrings to achieve better compression ratio. Normally matched substring

Initial string

abcd	defg	ghij	1234	abcd	defg	ghij	1234	...	...

Sequential match

abcd	defg	ghij	1234	(12, 16)				...	...

Parallel match

abcd	defg	ghij	1234	(12, 4)	(12, 4)	(12, 4)	(12, 4)	...	...

Fig. 3. Merge case 1.

length is 4 bytes since it is easy to convert 4 chars to an integer to search duplicate keys in hash table. Based on the hash table design, we have two cases need to merge.

1. Merge Case 1

We can merge two consecutive match substrings when their match offsets are equal. The first case example is described in Fig. 3. The first line in the figure is the original text need to compress. The second line is sequential compressed result, we call (12, 12) a codeword. The first number of codeword is backwards relative position, the second one is the matched length. The third line is our parallel intermediate compressed result. We need to convert parallel intermediate result to sequential match result to keep compression ratio same.

The basic idea is we can parallel merge contiguous codewords if the two relative positions/offsets are equal. We assign each thread to compare two codewords and perform it recursively, if contiguous matching occurs in the two substrings, i.e., two consecutive offsets is equal, that means we can expand the first matching data length by adding the second match length. However, the shortage of this algorithm is obvious: It can only merge even numbers codewords, and we need to scan twice to catch

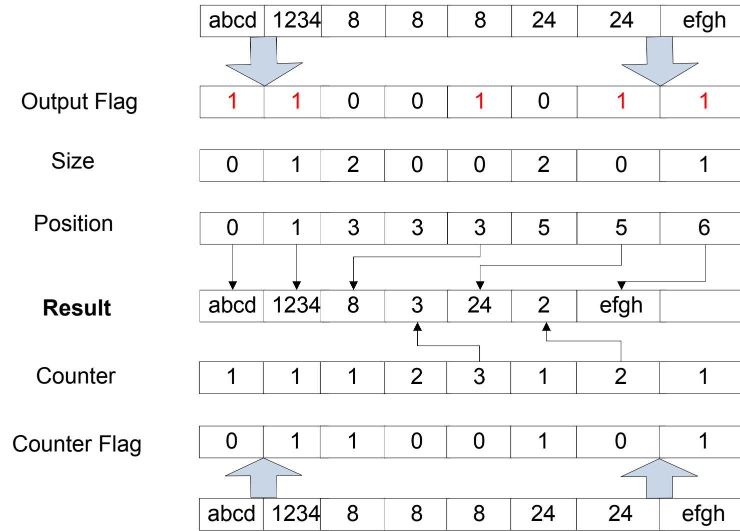

Fig. 4. Our merge process (compute element counters with prefix reduction and output corresponding positions with prefix scan).

the merge when the first codewords index is odd; Another problem is that we need to compact the sequence. Thus we propose another merging algorithm described in Fig. 4 which can keep result same with sequential. The main steps are shown below:

- Find out valid elements which should be output.
- Compute duplicate number and offsets for valid elements.
- Move to proper position.

We denote common notations here for all the followings algorithms. k is the GPU thread id, n is thread number in a GPU block, and N is the number of all input elements. B is GPU block number, and $B = N/n$.

In algorithms 1, *pos* is to indicate positions for elements in compacted sequence. *counter* is to save duplicated codewords number. $flag$ is for calculating *counter*

in segment scan process. $flag$ values are changed after segment scan, thus we use $outputFlag$ to indicate which elements should be output. As depicted in the algorithm, we have 4 steps and we omit the details of step 2 and 3. After this process, duplicated codewords are compacted by extending match length.

In practice, we generate a one-byte flag for each element to indicate current value is unmatched literal or matched offset in parallel match finder. After merging, we set 0 as flag value to indicate an element is unmatched literal, 1 as flag value to indicate it is a 4 byte matched length, 2 as flag value to indicate it is matched length, and 3 as flag value to indicate it is a match offset. We need to read next element as match length when flag value is 3. When performing arithmetic encoding, we need to compute the frequency of each symbol including match distance and length.

Multi-block segment scan is also involved in parallel merging. Algorithm 1 is performed in GPU thread block level, which is called intra-block merge. After this, we also need to merge inter-block duplicate matching. Firstly we need another array on global memory to save the compacted length of each block after intra-block merging, and then we assign one thread for each block to read the compacted length to locate the last element. Next step is to check the flag. If the flag is one or two, then we check if it is equal to next one and merge; otherwise we move back one step to compare or just skip when the flag value is three. There is another option is that we do not perform compaction for the matched sequence which could be faster. The problem is we can not merge inter-block redundancy, and it causes higher cost to know the exact valid elements number and calculate frequency for these symbols to do arithmetic encoding with static model.

Initial string

abcd	1234	abcd	12gf

Sequential match

abcd	1234	(8,8)	12gf

Parallel match

abcd	1234	(8,4)	12gf

Shift every 2 bytes

cd12	34ab	(8,4)	gf..

Fig. 5. Merge case 2.

2. Merge Case 2

The second case example is described in Fig. 5. For converting the parallel match result to sequential result, we generate a hash value when shifting every 2 bytes. The merging pattern is basically same with case 1, we compare the offset and update the length with number of matching elements multiply 4. In this case, the difference is that we update the match length to a multiple of 2 and add 2. Let N denote the number of matching substrings, and the expression of merged match length is $2N+2$.

This case can be extended to hash interval 1 to exactly match every repeated substring and make sure the result is same with sequential match. The potential issue is that twice increment of the hash table size would cause slower hash table building and searching. Moreover, we can change the hash size to 3 and generate small chunk matching, which is another way to increase matching substrings. The implementation of this case is a little different with first case. Since GPUs can only access every 4 bytes or a multiple of 4 bytes, we can not directly move 2 bytes to read them. We need to combine low 16 bits of previous hash key with high 16 bits of next hash key to produce a hash value whose offset is a multiple of 2.

Algorithm 1 Parallel merging algorithm.

```
1: /* 1. Initial pos, flag, outputFlag */
2: for k = 1 to n − 1 in parallel do
3:     if input[k] = input[k − 1] then
4:         pos[k] = 0
5:     else
6:         if input[k] = input[k + 1] then
7:             pos[k] = 2
8:         else
9:             pos[k] = 1
10:        end if
11:    end if
12: end for
13: for k = 0 to n − 1 in parallel do
14:    if input[k] ≠ input[k + 1] or k = n − 1 then
15:        outputFlag[k] = 1
16:    else
17:        outputFlag[k] = 0
18:    end if
19: end for
20: /* 2. Compute pos[k] with prefix sum */
21: /* 3. Compute counter[k] with segmented scan */
22: /* 4. Output elements and match length to proper position */
```

CHAPTER V

PARALLEL RANGE ENCODING ON GPUS

A. Sequential Arithmetic Coding

Fundamentally, the arithmetic encoding process consists of creating a sequence of nested intervals, for a simpler way to describe we represent intervals in the form $[b, l)$, where b is called base or starting point of the interval, and l the length of the interval [16].

Let Ω be a data source that puts out symbols s_k coded as integer numbers in the set $0, 1, ..., M-1$, and let $S = s_1, s_2, ..., s_N$ be a sequence of N random symbols. For now, we assume that the source symbols are independent and identically distributed, with probability

$$p(m) = Prob\left\{s_k = m\right\}, m = 0, 1, 2, ..., M-1, k = 1, 2, ..., N. \tag{5.1}$$

We also assume that for all symbols we have $p(m) \neq 0$, define $c(m)$ to be the cumulative distribution,

$$c(m) = \sum_{s=0}^{m-1} p(s), m = 0, 1, ..., M. \tag{5.2}$$

Note that $c(0) \equiv 0, c(M) \equiv 1$, and

$$p(m) = c(m+1) - c(m). \tag{5.3}$$

Basic arithmetic encoding algorithm can be described with the following two equations,

$$b_k = b_{k-1} + l_{k-1} c(s_k), \tag{5.4}$$

Table I. Arithmetic encoding example

Iteration	Input Symbol	Interval base	Interval length
0	—	1	—
1	2	0.7	0.2
2	1	0.74	0.1
3	0	0.74	0.02
4	0	0.74	0.004
5	1	0.7408	0.002
6	3	0.7426	0.0002

$$l_k = l_{k-1} p(s_k), k = 1, 2, ..., N. \tag{5.5}$$

Let us give an example from paper [16] to demonstrate the iterative process. Assume that source Ω has four symbols (M = 4), 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}, the whole input sequence is {2, 1, 0, 0, 1, 3, 1, 1, 1, 2}. The encoding example is demonstrated in Table I.

$b_0 = 0, l_0 = 1,$

$\Phi_0(S) = [0, 1),$

$b_1 = b_0 + c(s_1)l_0 = 0 + 1 \times 0.7 = 0.7,$

$l_1 = p(s_1)l_0 = 1 \times 0.2 = 0.2,$

$\Phi_1(S) = [0.7, 0.9),$

$b_2 = b_1 + c(s_2)l_1 = 0.7 + 0.2 \times 0.2 = 0.74,$

$l_2 = p(s_2)l_1 = 0.5 \times 0.2 = 0.1,$

Table II. Arithmetic decoding example

Iteration	Decoder updated value	Output symbol
0	0.74267578125	2
1	0.21337890625	1
2	0.0267578125	0
3	0.1337890625	0
4	0.6689453125	1
5	0.937890625	3

$\Phi_2(S) = [0.74, 0.84),$

...

$b_6 = b_5 + c(s_6)l_4 = 0.7426,$

$l_6 = p(s_6)l_5 = 0.0002$

$\Phi_6(S) = [0.7426, 0.7428),$

The final task in arithmetic encoding is to define a code value v(S) that will represent data sequence S. We can choose any value in the final interval.

The decoding process start from $v(S)$, the recursion formulas are

$$v'_1 = v(S), \tag{5.6}$$

$$s'_k = \{s : c(s) \leq v'_k < c(s+1)\}, k = 1, 2, ..., N, \tag{5.7}$$

$$v'_{k+1} = \frac{v'_k - c(s'_k)}{p(s'_k)}, k = 1, 2, ..., N-1. \tag{5.8}$$

In equation (5.7), the colon means "s that satisfies the inequalities". The decod-

ing example is demonstrated in Table II.

This process can make sure different sequence produce different code value. We can compare with the idea that we represent ASCII symbol sequence 'abc' $= 97 \times 256^2 + 98 \times 256 + 99$, we obviously know this value is unique.

To implement arithmetic coding with fixed-precision, we need to solve two problems. One is multiplication precision issue: the number of digits required to represent the interval length exactly grows when a symbol is coded. We solve this problem using the fact we do not need exact multiplications by the interval length. Practical implementations use P-bit registers to store approximations of the mantissa of the interval length and the results of the multiplications. All bits with significance smaller than those in the register are assumed to be zero. We do not have to worry about the exact distribution values as long as the decoder is synchronized with the encoder, i.e., if the decoder is making exactly the same approximations as the encoder, then the encoder and decoder distributions must be identical. The price to pay for inexact arithmetic is degraded compression performance. Arithmetic coding is optimal only as long as the source model probabilities are equal to the true data symbol probabilities; any difference reduces the compression ratios. In fact, if we can make multiplication accurately to 4 digits, the loss in compression performance can be reasonably small.

The other problem is addition precision problem when there is a large difference between the magnitudes of the interval base and interval length. This problem can be solved by interval rescaling. One important property of arithmetic coding is that the actual intervals used during coding depend on the initial interval and the previously coded data, but the proportions within subdivided intervals do not. For example, if we change the initial interval to 2 in arithmetic example, not 1, the coding process remains the same, expect all intervals are scaled by a factor of two, and shifted by one.

We also can apply rescaling in the middle of the coding process. Suppose that at a certain stage m we change the interval according to

$$b'_m = \gamma(b_m - \delta), l'_m = \gamma l_m, \tag{5.9}$$

We can use the following equations to recover the interval and code value that we would have obtained without rescaling:

$$b_N = \frac{b'_N}{\gamma} + \delta, l_N = \frac{l'_N}{\gamma}. \tag{5.10}$$

Sequential arithmetic encoding algorithm is depicted in algorithm 2 [16], which is the implementation of FastAC, one of the fast arithmetic encoder. In this algorithm, let L_{max} denote the initial value is l, L_{min} denote minimum value of l. For 4 bytes unsigned integer arithmetic coding implementation, $L_{max} = 2^{32} - 1$, $L_{min} = 2^{24}$, $D = 65536$. D is 65536 since FastAC split large file to 64 KB blocks, thus all denominators of $p(s_k)$ is 65536, i.e., right shift 16 bits. We can change D value depending on the block size. If we set block size 32768, then $D = 32768$. We also can change L_{max} and L_{min}. If we implement with 8 bytes unsigned integer, then $L_{max} = 2^{64} - 1$, $L_{min} = 2^{56}$, and this implementation would improve the compression ratio a little bit [17]. We calculate each input char based on the two equations (4) and (5) and produce new base and new length, finally the base is a big number. Since our computer has limited precision, we need to perform interval rescale and output the highest byte to output buffer to keep the precision of new base and length. We call the interval rescale and highest byte output process renormalization. For simple explanation, we use 256 symbols to explain our algorithm in following section.

Let us discuss two extreme cases here to have an intuitive impression how algorithm 2 works. Assume we compress a data source with 256 symbols, and we take 64 KB as block size. One case is all symbols have same frequency in the block, that

means all 256 symbols have same probability $p(s_i) = 1/256$. After code line 5, the length is changed to less than L_{min}, so interval rescale once for each input char, and each interval rescale output one byte, thus we can simply say the output byte is same with input, and no compression is produced here. The other case is the input data source only has two symbols. The probability of symbol '0' is 65535/65536, and the probability of symbol EOF is 1/65536. Since the probability of symbol '0' is close to 1, we can simply say that the length is always greater than L_{min}, and no interval rescale causes no output till the last symbol is encoded. We can estimate the whole block is compressed to couple of bytes. These two examples simply reflect how compression occurs and the relation between symbol frequency and compression ratio. In normal case, we output 2 bytes for 3 symbols.

In sequential arithmetic encoding, for N bytes, we need to calculate N steps, and in each step we do at least two multiplications and one addition, hence the time complexity is $O(N)$.

We propose novel parallel algorithm which time complexity can be $O(logN)$. The general idea is to separate the arithmetic encoding to big number multiplication and big number addition. Both can be parallel using variant algorithm of prefix sum.

Based on equation (4) and (5), we can derive the following equation without iterative process,

$$b_N = b_0 + l_0 c(1), N = 1 \tag{5.11}$$

$$b_N = b_1 + \sum_{i=2}^{N} \left(l_0 \prod_{j=1}^{i-1} p(s_j) c(s_i) \right), N > 1. \tag{5.12}$$

B. Character Frequency Statistics

Firstly we need to calculate the frequency of each symbol to figure out $p(s_k)$ and $c(s_k)$. Our parallel character frequency statistic algorithm employs source code of CUDA

histogram algorithm of Ramtin Shams and R. A. Kennedy [14]. We use the second collision free method of this paper. They maintain a histogram matrix of $B \times N$ size, where B is the number of bins and N is the number of threads. This method provides a collision free structure for memory update by each thread. A parallel reduction is ultimately performed on the matrix to combine data counters along the rows and produce the final histogram. Two problems need to be addressed for this method. One is slow zero initialization on global memory. They implemented a method for initializing floating point arrays in the kernel with a throughput of around 35 Gb/s on GTX 8800 and solved this problem. The other is non-coalesced read/writes per input data on global memory is inefficient. They pack multiple bins in a double work in the shared memory and only update the corresponding bin in the global memory when the packed bin overflows. This method greatly reduced the global memory update and our test result showed 6.6 GB/s high performance on GTX 480.

C. Parallel Big Number Multiplication

Our parallel arithmetic coding is based on this 4 bytes integer implementation. Firstly we can parallel calculate $\prod_{j=1}^{N-1} p(s_j)$ in equation (12) based on prefix sum algorithm, described in Figure 6.

The key problem is to resolve limited precision problem. Our solution is to represent $\prod_{j=1}^{N-1} p(s_j)$ with two parts, one is a float number, the other is right shift number based on the denominator of $p(s_j)$. We represent numerator of $p(s_j)$ with 4 bytes float, and the denominator depends on block size. In FastAC, they choose 65536 as block size to perform arithmetic coding, thus the denominator is 65536. In algorithm 3, the input are numerators of $p(s_j)$, k is thread id, n is thread number in a block.

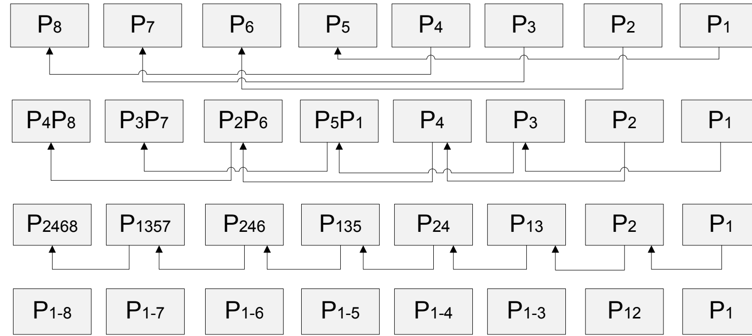

Fig. 6. Big number multiplication.

In algorithm 3, there is no precision loss introduced since in fact we are repeating the same process with sequential algorithm 2. The only difference is we firstly multiply numerator till it is larger than $2^{15}-1$, then divide the denominator. We use float type so actually no precision loss is introduced in this process. The purpose of dividing denominator is to prevent multiplication overflow for 32 bits float type, and $2^{15}-1$ is set as overflow threshold here since it is half of maximum value of 32 bits float type. In phase 1, we assign an initial right offset for each symbol, where base number is 256. Phase 2 basically is a prefix sum process. We also need to change offset in this stage when the product is larger than the overflow threshold. At the end of algorithm 3, we multiply $l_0, c(s_i)$, and amend the product by multiplying 256 and changing right offset value if it is less than 2^{24} to keep the value same with sequential.

In Fig. 7, we give an example with base number 16. When multiplying 7 with 4, we can not directly multiply them since the result 28 would cause an overflow. We firstly check if each multiplicator is large than $\log_2 16$. Since $7 > 4$, 7 is divided by 16, and the result is 0.4375. After this, we update its exponent by adding 1, then multiply by 4, and the result is 1.75. We apply same operation on 5 and 11. Finally the result is 6.015625, the exponent is 2. We can see $6.015625 \times 16^2 = 1540 = 5 \times 11 \times 4 \times 7$.

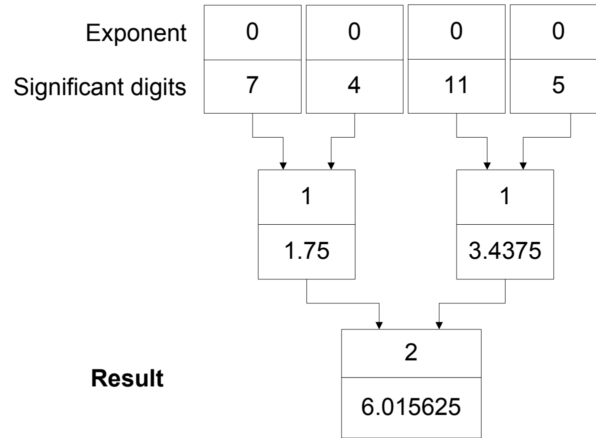

Fig. 7. Example of big number multiplication.

Large data is split to multiple blocks, thus we need to perform intra-block scan in shared memory and inter-block scan in global memory. In our implementation, algorithm 3 actually performs in GPU thread block which is intra-block operation. After this, we start inter-block scan. Let b denote GPU block id, and d denote iteration times, $0 \leq d < \log_2 B$, where B is the number of blocks, $B = N/n$. From second block, we load elements of each block and multiply them with last element of block $(b - 2^d)$, and then iterate this operation $\log_2 B$ times for all blocks, finally we output the final result to global memory. For example, assume we have 3 blocks, and each block has 8 elements. After GPU intra-block multiplication, the first block contains p_1, p_{1-2}, ..., p_{1-8}, the second block contains p_9, p_{9-10}, ..., p_{9-16}, and the third block contains p_{17}, p_{17-18}, ..., p_{17-24}. In first iteration, we multiply each element in second block with last element p_{1-8} in block 0 $(1 - 2^0 = 0)$; also we multiply each element in third block with p_{9-16}. After first iteration, elements in second block are p_{1-9}, p_{1-10}, ..., p_{1-16}, elements in third block are p_{9-17}, p_{9-18}, ..., p_{9-24}. In second iteration, we respectively multiply elements in third block with last element of block

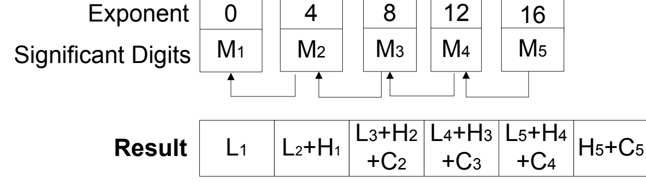

Fig. 8. Big number parallel addition output phase.

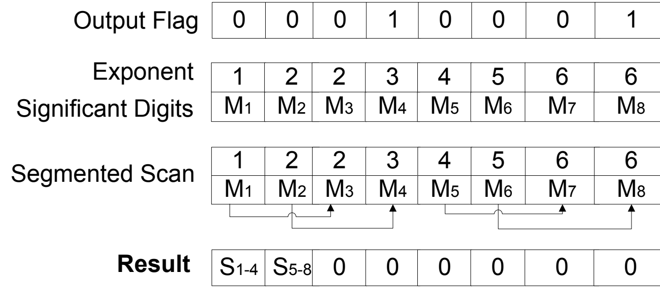

Fig. 9. Big number parallel addition merging phase.

$2 - 2^1$, i.e., the first block. After second iteration, the third block contains p_{1-17}, p_{1-18}, ..., p_{1-24} and the process is terminated. Let N represent element number we need to multiply, B and T represent the block and thread number, $N = B \times T$, the step complexity is $O(\log_2 T + \log_2 B) = O(\log_2 N)$.

D. Parallel Big Number Addition

Secondly we can parallel compute the summation $\sum_{i=1}^{N} I_i$ in equation (12), where I_i denotes $l_0 \prod_{j=1}^{i-1} p(s_j)c(s_i)$. Based on the first step, let S denote left shift offset, we obviously know $S[i]$ is a sorted array and $S[i+1] - S[i] \leq 1$ (i.e, 1 or 0) since the minimum multiplicand in the algorithm is 1/256. Let M denote the 4 byte float number, we have $I_i = M_i/(256^{S[i]})$.

In sequential arithmetic coding algorithm, when $range < L_{min}$, we rescale the range by left shift 8 bits and output highest byte of base to keep the precision of addition, otherwise small range would cause precision loss problem. After rescaling the length and left shift operation, we can represent $I_i = M_i(256^{S[N]-S[i]})$, where S[i] is left shift offset.

The key problem is how to hold the result in memory and how to parallel output it to compressed file. We can not maintain the precision of the final result since it could be a huge number which even can not fit in shared memory. Therefore our approach is to output the huge number by bytes to shared memory and then to global memory. We know the maximum left shift offset, thus we know the output memory size. Considering the carry of highest position, we increase 4 bytes for the total memory size. The process of parallel output results is depicted in Fig. 8. In this figure, the top line is the left shift offset of all input number, the second line is the 8 bytes number representing significant digits of I_i. Let L_i denote the low 32 bits, H_i denote the high 32 bits of the 8 bytes number and C_i denote the carry of sum of $L_i + H_{i-1}$. Firstly we shift 32 bits to get high and low 32 bits for each number. Secondly we add low 4 bytes value of current offset with previous high 4 bytes value and carry from previous offset, and output the sum which is the final result on this offset to global memory.

Another problem is our shift offsets are not a multiple of 4. They are monotonically increasing, can be repeated and the biggest gap is one. There are two extreme cases: one is all shift numbers are same, the other is all is different and monotonically increase one. In real situation, after match finding phase, arithmetic coding produce around 2 bytes for every 3 symbols. We know a new base is generated for each symbol, therefore two base numbers could have same offsets.

Our solution is firstly computing summation of all I_i whose shift offset from $4i$ to

$4(i+1)-1$, $0 \leq i \leq (S[N]/4)$, and then save the sum to 8 bytes long variable. Fig. 9 demonstrates the process. Output flag is to indicate which number should be output, the first line of input data is the left shift number, and the second line is 4 byte integer numbers. Furthermore, we generate a flag which is described in algorithm 4 for segmented scan. After segmented scan, all left shift number is a multiple of 4, so we can parallel arrange 4 bytes output for each thread and output them to global memory.

The whole parallel big number addition process is described in algorithm 4, it is performed in GPU thread blocks. After intra-block operation, multi-block merge process starts. We use another array to save the shift offset of last valid byte in each block. If the shift offset of last valid byte in block k is s, and the shift offset of start valid byte in block $k+1$ is $s+1$, then we do not merge them; if they are same, then we need to merge by adding the two; if a carry is propagated, then previous element is updated by adding 1, also we update the start element in block $k+1$ to zero. We apply this merging manner to all adjacent blocks and compact them to remove zero. The compacted result is the final arithmetic coding result, which is exactly same with sequential algorithm.

In our implementation, we also do optimization on memory coalescing, divergent branching, bank conflicts and latency hiding. In practice, we avoid shared memory bank conflicts by replacing interleave addressing with sequential addressing. We unroll loops to remove instruction overhead since multiply operation has low arithmetic intensity. We can unroll last 9 iterations of the inner loop using templates since we know the block size on GTX480 is limited to 768 threads. Moreover, we can remove CUDA function _syncthreads which introduces 4 clock cycles because instructions are SIMD synchronous within a warp.

Algorithm 2 Sequential arithmetic coding: FastAC

1: **Input:** $p(s_k)$ and $c(s_k)$ respectively contain numerators of all symbols' probabilities and cumulative distribution in the input sequence, D is the denominator of the probability and cumulative distribution, N is the number of characters in the input sequence.
2: **Output:** *output* contains compressed data, a big number.
3: **for** $k = 1$ to N **do**
4:     /* Compute new base and length according to equation (4) and (5) */
5:     $l = l/D$
6:     $b{+}= l * c[s_k]$
7:     $l{*}= p[s_k]$
8:     **if** propagate carry **then**
9:         $p = idx - 1$
10:         **while** $output[p] = 255$ **do**
11:             $output[p--] = 0$
12:         **end while**
13:         $output[p]++$
14:     **end if**
15:     **while** $l < 2^{24} - 1$ **do**
16:         $l <<= 8$
17:         $output[idx++] = b >> 24$
18:         $b <<= 8$
19:     **end while**
20: **end for**

Algorithm 3 Parallel big number multiplication.

1: **Input:** $data$ = numerators of $p(s_j)$, D = denominator of $p(s_j)$.
2: **Output:** $power$ contains right shift offset of each symbol, $data$ = respective significant digits of $\prod_{i=1}^{1} p(s_i), \prod_{i=1}^{2} p(s_i), ..., \prod_{i=1}^{N} p(s_i)$.
3: /* Initialize offset, k is thread id, n is total thread number in a GPU thread block. */
4: **for** all $k = 0$ to $n - 1$ in parallel **do**
5: $power[k] = \log_{256} D$
6: **end for**
7: /* 2. Compute $\prod_{i=1}^{N} p(s_i)$. */
8: **for** $d = \log_2 n - 1$ down to 0 **do**
9: **for** all $k = 0$ to $n - 2^d - 1$ in parallel **do**
10: $data[k] *= data[k + 2^d]$
11: $power[k] += power[k + 2^d]$
12: **while** $data[k] > 2^{15} - 1$ **do**
13: $data[k] = data[k]/D$
14: $power[k] -= \log_{256} D$
15: **end while**
16: **end for**
17: **end for**

Algorithm 4 Parallel big number addition.

1: **Input:** $input = (I_1, I_2, ..., I_N)$, $power$ contains
all left shift offsets of I_N.
2: **Output:** $output = (\sum_{i=1}^{N} I_i)$.
3: /* 1. Load input to 8 bytes data[k], initialize flag */
4: **for** all $k = 1$ to $n - 1$ in parallel **do**
5: $data[i] = data[i] << ((power[i] \bmod 4) * 8)$
6: **if** $power[k] \bmod 4 = 0$ AND
$power[k]! = power[k-1]$ **then**
7: $flag[k] = 1$
8: $outFlag[k-1] = 1$
9: **end if**
10: **end for**
11: /* 2. Compute partial sum with segmented scan */
12: /* 3. Compact the data array */
13: /* 4. Each thread output 4 bytes to global memory */
14: **for** all $k = 0$ to $n - 1$ in parallel **do**
15: $outValue+ = data[k]$ AND $2^{32} - 1$
16: $outValue+ = data[k-1] >> 32$
17: **if** propagate carry **then**
18: $carry[k]+ = 1$
19: **end if**
20: $outValue+ = carry[k-1]$
21: $output[k] = outValue$
22: **end for**

CHAPTER VI

RESULTS AND ANALYSIS

We test with 10 GB file which is crawled by IRLbot [18] crawler. We randomly truncate 32 MB, 128 MB, 256 MB, 512 MB from the 10 GB file and then test the compression ratio and time. We test the data with Hash 4 bytes, hash 2 bytes and longer match three methods.

As for testing hardware, we use an AMD Phenom(tm) II 2.8 GHz six-core desktop machine with 3MB L2 Cache for all our experiments. The machine runs Windows Server 2008 R2 with 16 GB of RAM support and 5 TB of disk space available. The GPU is GeForce GTX 480, CUDA driver version is 3.20 and CUDA capability major/minor version number is 2.0. GeForce GTX 480 has 1576599552 bytes total amount of global memory, 15 Multiprocessors, and each MP has 32 cores.

In Fig. 10(a), we can see compression time is close to linear increase. Our implementation is splitting a big file to small blocks (128 MB, 192 MB), thus it is linear increase. We can observe MatchLonger method is increase more sharply, the reason is we use 192 MB block size for 512 MB files, and the larger hash table is slower.

In Fig. 10(b), firstly we can find when file size is less than 32 MB, the compression ratio is lower than file larger than 100 MB. The reason is fewer match substrings can be found in files less than 128 MB. When file size is larger than 128 MB, the large file is split to 128 MB, so compression ratio is similar. We also can observe compression ratio of Hash4 method improve faster than Hash2, and Hash2 is faster than MatchLonger method, which proves more hash values can produce more matching and really work efficiently for both small and large file. In small file, method Hash4 is limited by hash table size, so Hash2 and MatchLonger work better for small file.

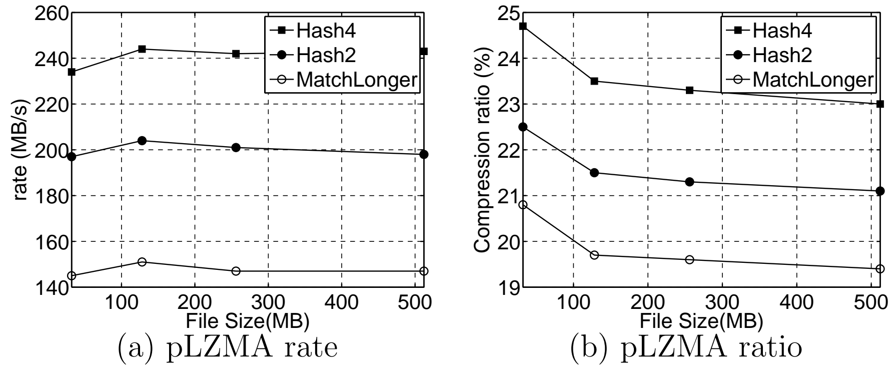

Fig. 10. Compression rate and ratio for parallel LZMA (pLZMA).

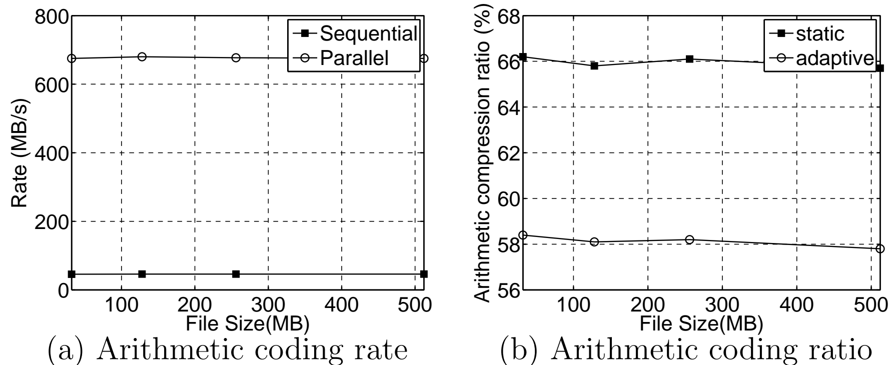

Fig. 11. Compression rate and ratio for parallel arithmetic coding.

FastAc is one of the fastest arithmetic encoder [19], we take it as sequential arithmetic encoder example and compare with our parallel arithmetic compressor on GTX 480. Fig. 11(a) is our compression time test result, which shows that the compression speed of FastAC on CPU is around 46 MB/s, while our parallel algorithm speed is around 680 MB/s. However, we can only parallel static model. The difference between compression ratio of static model and adaptive model on our dataset is around 8% as showed in Fig. 11(b). We minimize the difference with match finder pre-processing and produce good final compression results compared with popular compressors.

We test FreeArc which is the fastest compressor in the environment. The com-

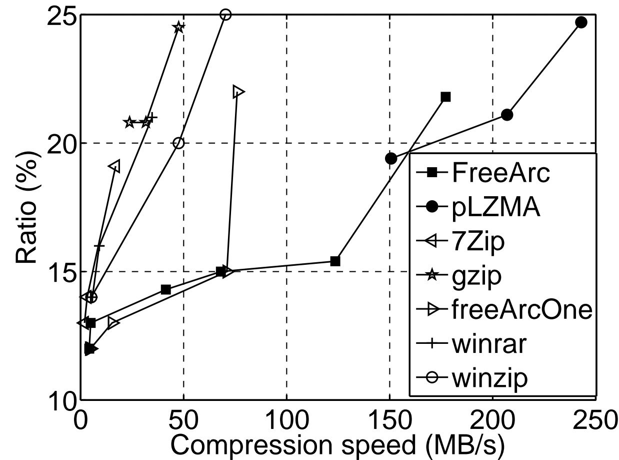

Fig. 12. Compression rate and ratio comparison with FreeArc (0.66) on 6x2.8 GHz core, FreeArc (0.66) on 1x2.8GHz core, winrar (4.01), winzip (15.5), gzip (1.3.12) and 7zip (9.20).

pression speed is 177 MB/s with compression ratio 21.8%. In Fig. 12, we can observe our algorithm (pLZMA) on GPUs can achieve competitive compression ratio with FreeArc on 6-core CPU when compression ratio is around 20%. And our result is 3.5-10 times faster than popular compression software such as gzip, 7zip and winrar, whose compression speed is around 17-70 MB/s on the 6x2.8 GHz CPU with compression ratio around 20%. Our compression ratio is around 4% larger than normal LZMA algorithm since we parallel static arithmetic encoding, while sequential arithmetic encoding can use adaptive model and achieve better compression ratio.

CHAPTER VII

CONCLUSION

In this paper, we proposed a novel parallel text compression design on GPUs, a novel parallel matching and merging algorithm to keep the compression ratio approximate with the sequential, improved parallel range coding design and implementation on GPUs. We showed with our experiment that our compressor is 3.5-10 x faster than sequential approaches on modern CPUs with around 20% ratio. Future work involves exploring methods to design more efficient algorithms for building large hash table and parallel adaptive arithmetic coding.

REFERENCES

[1] Intel Performance Primitives, "IPP," http://software.intel.com/en-us/intel-ipp, Mar. 2010.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," *ACM Trans. on Graphics*, vol. 28, no. 5, pp. 154:1–154:9, Dec. 2009.

[3] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[4] P.M. Fenwick, "The burrows-wheeler transform for block sorting text compression - principles and improvements," *The Comp. Journal*, vol. 39, no. 9, pp. 731–740, Jun. 1996.

[5] P. Ferragina and G. Manzini, "On compressing the textual web," in *Web Search and Data Mining*, Feb. 2010, pp. 391–400.

[6] J. Bentley and M. McIlroy, "Data compression with long repeated strings," *Inform. Sci.*, vol. 135, no. 1-2, pp. 1–11, Jun. 2001.

[7] B. Ziganshin, "Freearc," http://www.freearc.org, Dec. 2009.

[8] J. Gilchrist, "Parallel bzip2," http://www.compression.ca/pbzip2, Apr. 2010.

[9] J. Jiang and S. Jones, "Parallel design of arithmetic coding," *Comp. and Digital Techniques*, vol. 141, no. 6, pp. 327–333, Nov. 1994.

[10] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *High Performance Graphics*, Aug. 2009, pp. 159–166.

[11] D. G. Merril and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Parallel Architecture and Compilation Techniques*, Sep. 2010, pp. 545–546.

[12] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," Tech. Rep. NVR-2008-003, NVIDIA, Dec. 2008.

[13] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *International Parallel and Distributed Processing Symposium*, May. 2009, pp. 1–10.

[14] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *International Conference on Signal Processing Systems*, Dec. 2007, pp. 418–422.

[15] G. Nigel and N. Martin, "Range encoding: an algorithm for removing redundancy from a digitized message," in *Video Data Recording Conference*, Southampton, UK, Mar. 1979.

[16] A. Said, "Introduction to arithmetic coding theory and practice," Tech. Rep. HPL-2004-76, Hewlett-Packard Laboratories, Apr. 2004.

[17] S. Garg, "64-bit range coding and arithmetic coding," http://www.sachingarg.com/compression/entropy_coding/64bit, Apr. 2005.

[18] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "Irlbot: Scaling to 6 billion pages and beyond," Tech. Rep. 2008-2-2, Texas A&M University, Feb. 2008.

[19] A. Said, "Comparative analysis of arithmetic coding computational complexity," Tech. Rep. HPL-2004-75, Hewlett-Packard Laboratories, Apr. 2004.

VITA

Xiaoxi Zhang received his B.E. degree in computer science from National University of Defense Technology, China. He received his Masters of Science degree in computer science in December 2011 at Texas A&M University, College Station.

His research interest include GPUs parallel computing and compression. He can be reached at:

c/o Department of Computer Science and Engineering

Texas A&M University

College Station

Texas - 77843

The typist for this thesis was Xiaoxi Zhang.