

# Parallel Lossless Compression

(Namita Pradhan, Arjav Naik, Anuja Patil)

## Overview:

We are implementing the DEFLATE algorithm which is used for G-Zip compression. Compression saves bandwidth and speeds up the task. DEFLATE constitutes LZ77 algorithm and Huffman coding. We used two kernels - one for LZ77 and the other for Huffman coding. Currently we have implemented both the algorithms in parallel and compressed the data separately.

**Libraries used:** bitset, pqueue

## Data Pre-processing:

We read the files using C++ standard IO libraries, into character array to pass to the device kernel.

## Lempel-Ziv 77 (LZ77):

This algorithm encodes data as a string and points to a dictionary where the previously saved characters are stored. So, if the same character occurs again then, it will be used from the dictionary.

## Parallel implementation of LZ77:

The following flowchart explains the parallel implementation of LZ77 algorithm. Let us go step-by-step through the flowchart and understand the methodologies behind each step.

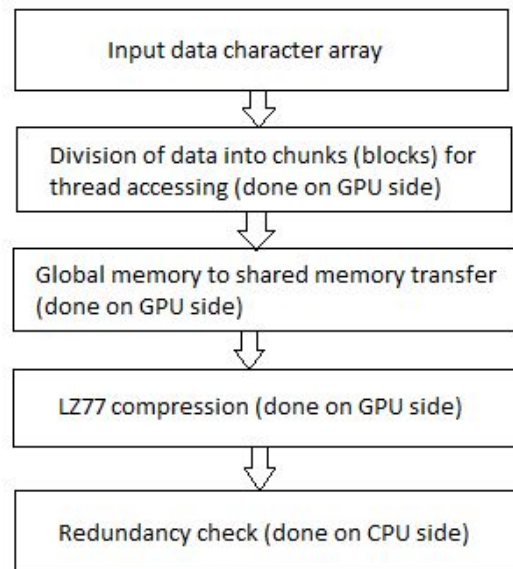


Fig.1: LZ77 flowchart

First the input data character array is divided into chunks of data, represented by a thread block. We have considered the block size to be 512. That means, the input data is divided into 512 sized parts which are given to thread blocks; each thread works with one character. This data is transferred to shared memory from global memory. All this is done on the GPU side of the processor.

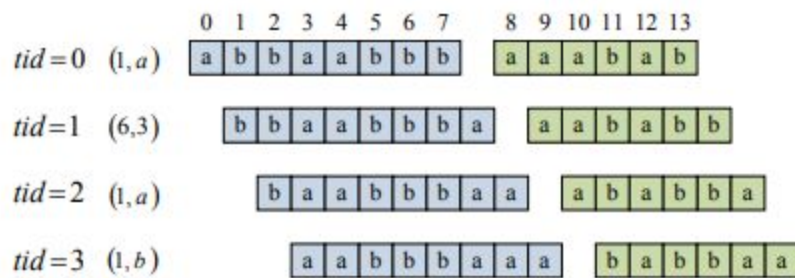


Fig. 2: LZ77 sliding window

For the LZ77 compression, we used the sliding window technique. As shown in the Fig. 2, there are two parts in the 14-bit window, i.e., search buffer and uncoded buffer. The search buffer is 8-bit long and the uncoded buffer is 6-bit long. The characters in the uncoded buffer are searched in the search buffer. If they do not match then they are stored in the form, (length, substring). But if they match then the code is stored in the form, (offset, substring length). Once this is done, the entire sliding window is slid to the right by the number of bits that are checked. This process is repeated until the entire input data shifts out of the uncoded buffer. The offset of each thread's starting point is determined by its thread ID.

But, sliding of the window by the number of bits checked is only theoretical. In practice the shifts happen one bit-by-bit. This gives redundancy, for example, if 3-bit substring is checked at a time, then the next two 1-bit shifts will be redundant as they have been already accommodated in the previous substring. We have checked for this redundancy and counted the number of redundant elements in the output. Thus, if we remove these redundant elements from the output, then we will have a much more compressed output!

### Huffman encoding:

In Huffman coding, each character in the input is converted to sequences of codewords. In this algorithm, we generate Huffman trees based on the frequency of each character and sums calculations. Then we traverse the tree to determine codewords for each character.

### Parallel implementation of Huffman encoding:

The following flowchart explains the parallel implementation of Huffman coding. Let us go step-by-step through the flowchart and understand the methodologies behind each step.

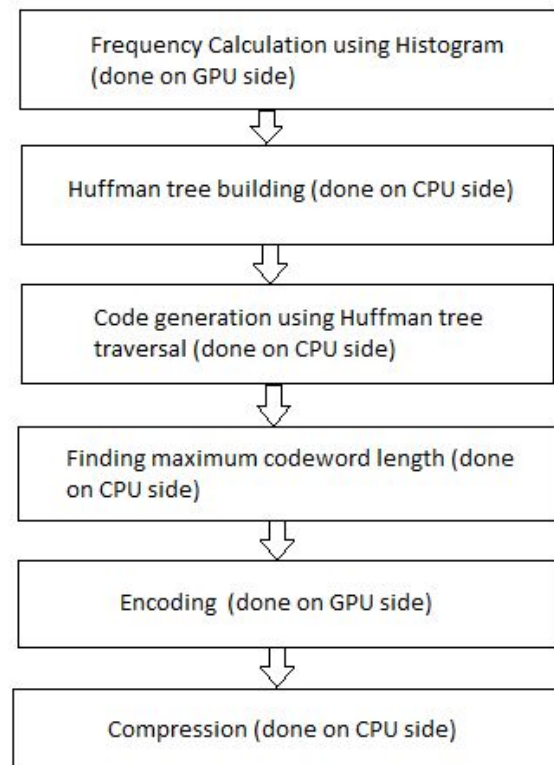


Fig. 3: Huffman encoding flowchart

We read a text file in the format .txt which consists of random alphanumeric characters, into an input array. We then use this input array to calculate the frequency of each different character. The function `calculateFrequency` in the file `kernel.cu`, calculates the frequency of these different characters in the input array. For this purpose we used a private histogram which is then merged with the global histogram. The histogram technique uses `atomicAdds` for maximizing the parallelism. The number of bins of our array is 40, which includes the 26 alphabets from a-z, 10 numbers from 0-9, space, period, comma and new line. We have not used capital alphabets for simplicity. The Histogram is performed on the GPU as we can exploit the parallelism to make it work faster.

Then we built a Huffman tree using a class `HuffmanNode`, with variables; `character`, `count`, `left (0)` and `right (1)`, which are the two children of the node. 0 and 1 are set to the two children because we want to generate binary codes for the characters. Using such nodes, the Huffman tree is created with the frequencies calculated from the histogram. All the frequencies are set as priorities of a priority queue, the lowest frequency being at the head of the queue. The Huffman tree creation begins with dequeuing two elements, adding them to form a huffman node and then enqueueing it into the queue. The next node will be added to the left or right subtree depending on whether it's value is larger or smaller than its parent node; if it is smaller, then it is

inserted in the right subtree and if greater, then in the right subtree. This procedure is repeated until there is only one element in the queue.

Then we move on to code generation using this created Huffman tree by traversing it. We started with the root node and kept traversing till we reached a leaf node. If we go to the right of a node, then the code is 0 and to the left it is 1. When the leaf node is reached then the entire codeword that is traversed is stored and assigned to the character held with the leaf node. The characters are identified by using their ASCII representation.

All the codeword lengths are stored in an array which is used to find the maximum length. This is done because, the threads in the GPU, which will access each codeword, do not know where to start the next codeword from. Thus, each thread accesses number of bits equal to this maximum length. We use this maximum length to calculate the offset of each thread. The codeword is written from this offset onwards and the other short codewords are padded with '2'. This writing is done in such a way that each bit is written into a byte. For example, if the code for the character 'a' is 101 then, 1 is written in one byte, 0 in the next and 1 in the last byte. Therefore, 3 bytes are required to write a 3-bit long codeword. This step is performed on the GPU side.

Then comes the final stage which is the compression. Here, we read 8 bytes together which do not consist of the padded '2'. Then the bytes are compressed using the library bitset. This is then written into a binary file, which is our compressed output!

#### **Limitations of the project:**

1. We are not implementing the Huffman coding for capital alphabets.
2. The redundancy check is done in LZ77, but we do not remove the redundancies.
3. Both our algorithms are not integrated to improve efficiency, they compress data individually.
4. LZ77 gives segmentation fault for bigger input.

#### **Evaluation/Results:**

#### **LZ77 Compression:**

```

bender /home/cegrad/anaik/GPUProject $ ./a.out example11.txt
Allocating device variables...
Copying host variables to device...
Launching kernel...0.000054 s
Copying device variables to host...
1b1a1a635554536374731a1b1a1b33531b8366656463835656555474731b1a8483464646468584831
b5554531a858483767574731b2354531a1b131b86868685848373531a65646374731b56566356555
453431b1a53868584831354531a1b1a1a43631a1b33531b1a1524331a1a1b14231b1b656473531b1a
131a83656463531a1b2626464666668584831b1a1626364554631a1a1b262544431a1b1b1a431b13
64631a76767583731b5334331a532476767574731b1a14231a1a1b14231b1b6666748366666665758
58483531b1b1a1a1a836463531a1b767574731a131a7574731b444354531b1a1a1a1b1a757473858
483533534868584831a1b1b832354531b1a648686868686858483331a1423631a1b1b331b46467344
831b1a1a1a86858483464646466868584831b1a53834431a731a142364631b1a1b131b1b83656475
83731b65647355545384831a1b1a656463331b16253443831b1a1a1b23531a631a1b5343868584834
67344831a1b1b1b868685848375747325243731b831b1a733534331b5584831a1b1b74732366656
463464544831a632583431b1a1a1a1b1524331b1b767686868584831b1a162534431a1a1b1b1b1a1b
86858483162534431b1b83331b1a14231a731b858483666564631b1a252443837473331b431b1a1a
63531b331b14236666656473531b1a65ABABABABABABABABABABABABABABABABABABABABABABABABABAB
count = 372
Freeing memory...

```

Fig. 4: Output for input file size 512

Here, we have given an input file which has 512 characters consisting of only a and b, to show the correct parallel LZ77 implementation. Before checking the redundancy, the output of the LZ77 compression will be of size 1024. Therefore, the output has to be checked for redundancy. After checking redundancy on the CPU side, the output will be of size 372. The reduction in size shows correctness of the LZ77 implementation. The time required to launch the kernel is 0.000052 s.

```

bender /home/cegrad/anaik/GPUProject $ ./a.out example3.txt
Allocating device variables...
Copying host variables to device...
Launching kernel...0.000053 s
Copying device variables to host...
666564634335767574731b1a8356565656565554531a631a43331a831b64631a7343631a1b34331b1
35684835663565655546573631b1a1a331a44431b6463531a1b162534431b1b86858483631b1a1a8
483631b1a848314231a1a1b734544431b7574731a84831b6666656463531b1a23565554531b1a8333
1a16253443831a1b131b1b631a34331a8685848353431b84831423656463363534631b1a131a6665
646354531b631b4646454483731b1a1a732363554531a1b656463331a142366666667483631a1
b1b83531a1b736666666665646335384836463431a565655545364631a1b2375748483331a731b1
a1b1b8375747325244383747334331a1b1b1b7675747363431a53831b236463738334431a1a1b1b1b
1a1b8314231a1a1b131b1b1a23757484833544531b1b1a1b1a33565554531a1b131b66656463531b
438336736564631b13831b1a131a1a1b23735453331a431a1b8584831b646333538483631b7356565
6565554531a8483331b54831a631a1b14231b1b1a2384836665648584834564631b1a14231a83633
5343383731a1b848331b13631b53331b1a1a531b137473631a1b848374731a131a747384831b5335
3485848323631b1a747356565554531b1a731a8685848364637383331a1a1b1a1b1b1b76767676
7675747353486858483457343565554ABABABABABABABABABABABABABABABABABABABABABABABABABAB
43431a1b23531a66666668363767574731a1b33545376757473331b15243374731a1b1a1a83767
675747334331a431a1b8354531a4383331b1a1b1b731a14231a8686868685848334431a1a1b231b53
1a1b1b7473236564631b1a1a635545363767574731b34331b74731b15243384831a1a1a83656463
73858483731b1a262646466564831a1b7334331b1a14231a731b84835453431b1a646336353463868
584831a1b633534331a1b1a14231a63831a1b1b436463858483731b1a131a631b14231b1b1a1b1a1
a4443331a1b131b656463431b338483431a1b1a747384836665646356555453431b1a6564631b7353
1a1423858483131b1b666564748483431b1b8483431b1a131a1a53331a1b1b632484831a1a668483
6463767574731a1b1b464544431b1a5554531b464544431a1b1a34331a44731b65646375747353358
58483731a1b1b747363354537473431a1b1b1a14231a1a1b238483631a1b747324231b858483444
731b631b74731a131a767675747365646374731b53363683666564631b1a1524331a831b1b1b1a23
635453331b1423631b1a1a331a44431b6463531a1b63331b16263544531b1b1a1b1a131a54533585
8483731a1b1b34338483131a1a832626464666668584831b1a14231a1a1b1b83531a1b75747343656
463531b1aABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
count = 764
Freeing memory...

```

Fig. 5: Output for input file size 1024

Here, we have given an input file which has 1024 characters consisting of only a and b, to show the correct parallel LZ77 implementation. Before checking the redundancy, the output of the LZ77 compression will be of size 2048. Therefore, the output has to be checked for redundancy. After checking redundancy on the CPU side, the output will be of size 764. The reduction in size shows correctness of the LZ77 implementation. The time required to launch the kernel is 0.000053 s.

## Huffman coding:



The following screenshot is the output of our first kernel which calculates frequency of characters of the input file. This is done on GPU side.

```
bender /home/cegrad/apatil/GPUProject $ ./compressFile example3.txt
Allocating device variables...
Copying host variables to device...
Launching kernel...
(Copying device variables to host...)
a frequency is 4
b frequency is 4
c frequency is 0
d frequency is 17
e frequency is 9
f frequency is 7
g frequency is 4
h frequency is 2
i frequency is 2
j frequency is 4
k frequency is 11
l frequency is 11
m frequency is 0
n frequency is 8
o frequency is 0
p frequency is 0
q frequency is 0
r frequency is 3
s frequency is 11
t frequency is 4
u frequency is 0
v frequency is 1
w frequency is 6
x frequency is 0
y frequency is 0
z frequency is 2
0 frequency is 0
1 frequency is 2
2 frequency is 2
3 frequency is 2
4 frequency is 2
5 frequency is 1
6 frequency is 0
7 frequency is 0
8 frequency is 0
```

Fig. 6: Frequency calculation using Histogram

## Compilation and running:

### LZ77:

For LZ77 implementation we have included two files mainlz.cu and lzkernel.cu. To compile these two files use command `nvcc mainlz.cu`.

We have also included two files for testing the code example11.txt (512 characters) and example3.txt (1024 characters).

So to test use the command : `./a.out example3.txt` and `./a.out example11.txt`

The output for example3.txt shows a segmentation fault after displaying the correct result.

The two screenshots attached are the expected result.

For `./a.out example11.txt` the expected output is Fig. 4.

For ./a.out example3.txt the expected output is Fig. 5.

### Huffman encoding:

We have following files for Huffman encoding:

fileCompressor.cu (which contains the main() function), kerne.cu (which has 2 kernels), huffmanNode.h, huffmanNode.cu, huffmanUtil.h, huffmanUtil.cu, huffmanNodeComparator.cu and Makefile.

To compile the code, we run: make

This should generate compressFile executable file.

Then run this command to run:

./compressFile example2.txt

This should create an output file "outfile.bin" which should contain the binary encoded output of the file example2.txt.

However, we are getting following issues (discussed on Piazza):

1) We have 2 kernels - one for calculating frequency which works fine. The second kernel is to encode the input file. This kernel launches after its corresponding copying of data from device to host, which is out of the desired order of execution. This results in fatal error of not being able to copy device variables to host variables. Subsequently after this kernel launch, the device to host copy resumes again. This circular execution pattern wasn't resolved.

### Miscellaneous:

We also wrote serial implementations for both LZ77 and Huffman encoding. We have included them for your reference.

The Huffman code was not working due to Library errors. We were not able to figure them out. The serial code is in Serial folder of zip file submitted.

The LZ77 serial code did not give correct results when checked manually with the parallel implementation. The serial code of LZ77 is in lz77.cpp file.

### Task distribution:

Task	Breakdown
Implementation of Huffman encoding	Anuja Patil - 65%, Arjav Naik - 10%, Namita Pradhan - 25%
Implementation of LZ77	Anuja Patil - 10%, Arjav Naik - 65%, Namita Pradhan - 25%
Project Report	Anuja Patil - 25%, Arjav Naik - 25%, Namita Pradhan - 50%

## References:

- [1] A Parallel Huffman Coder on the CUDA Architecture, Habibelahe Rahmani, Cihan Topal, Cuneyt Akinlar
- [2] GLZSS: LZSS Lossless Data Compression Can Be Faster, Yuan Zu and Bei Hua
- [3] CULZSS: LZSS Lossless Data Compression on CUDA , Adnan Ozsoy and Martin Swamy.