15-418/618
Final Project Report: Parallel Huffman Decoding

Joelle Lim (joellexl), Srinidhi Kaushik (srudrapa)
Team Parallex

Apr-May 2022

1 Summary

For our Final Project, we parallelize Huffman decoding process using CUDA and test its performance on different data-sets using the GHC machines. We compare the computation speed-ups obtained with different settings of the parameters of the decoding process. We attempt to analyze several possible ways how the segment size and number of threads used to run the code affects the speed-up. Given the speed of our implementation, it seems like parallel Huffman Decoding (which has not been attempted much in literature due to the complications of the decoding process) is effective and viable.

2 Background

Data compression is a basic component of a wide range of applications, and Huffman coding is one of such lossless compression techniques. Although the main intended purpose of compression is thought of as to save valuable storage, it plays another important role in increasing effective bandwidth to the said storage when incorporated into file systems. With the shift to data-intensive workloads and applications, the importance of being able to conduct these compression and decompression operations on-the-fly fast and efficiently is becoming more apparent. Hence, for this final project, we want to look into parallelizing the decoding part of this compression scheme.

2.1 Introduction to Huffman Encoding

The basic idea of Huffman Encoding is to assign variable length code-words to each of the input characters, and these lengths of the assigned codes are based on the frequencies of occurrence of the corresponding character in the input text. Note that none of the prefix can be repeated amongst the code-words to avoid ambiguity in decoding. To encode an input text stream, we first build a frequency dictionary, and repeatedly select 2 minimum frequency symbols and merge them using a Min Heap Tree. Finally, the code-word can be

assigned to characters by recursively traversing the tree and recording the path taken by the traversal.

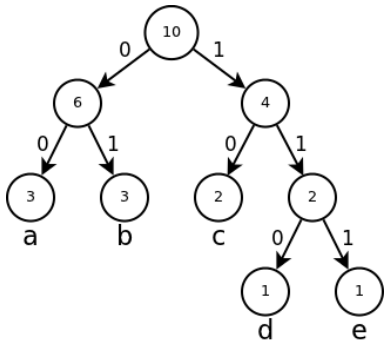

Fig: A Huffman tree.

An example of this Huffman Encoding Tree is shown above. The value of a leaf node represents the frequency of occurrence of the particular character in the input text; the value of an internal tree is the sum of all the values its sub-tree. Each branch of the tree takes on a bit-code of 0 or 1, and the code assigned to each character is the concatenation of all these bit-codes along the traversal path to the character node. For instance, the character a appears 3 times in the input text and is assigned a code-word 00, which has length 2; on the other hand, the character d appears only 1 time in the input text and is hence assigned a longer code-word of 110, which has length 3.

2.2 Why Parallelism?

This problem might be apt for parallelism because we are usually working with huge sized inputs for compression/ decompression. The decoding process for the compressed text seems to be doing very repeated work for each character where we lookup the Huffman encoding tress and translate it from code-words back to plain-text. Both these factors seem to suggest that this problem might lend very well to **data-parallelism**.

This process is however fraught with synchronization challenges, which has been a slight deterrent in attempts to parallelize this decoding. However, a recent paper published in 2018 showed that this process can actually achieve massive parallelism, and this solution is what we intend to replicate in our project.

2.3 Decoding as a Huge Challenge

Huffman decoding has always been a challenge, and there were not many efficient parallel solutions until several experimental algorithms were proposed in the last 5 years. The reason for its difficulty lies in the fact that there is **no way to tell when the encoded code-word starts and end**. Huffman encoding is simple because we can break up the input into chunks and encode each chunk separately, but in the case for Huffman decoding, there is no clear separation to identify each code-word, keeping in mind that each code-word can be of different lengths.

3 Our Approach

In the following section, we first explain in high level concept of our approach, the rationale behind it, and how it works; this is then followed by, a more detailed explanation of how we implemented it in terms of the code design, data structures and how CUDA constructs are used.

3.1 High-Level Concept Idea

Several state of art parallel solutions proposed to solve this issue exploit the **self-synchronizing property** of Huffman codes, and this is going to be the approach taken in our implementation. This property is explained using the diagram below.

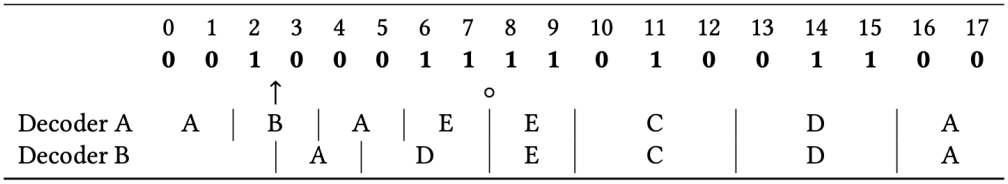

Fig: Illustration of the *self-synchronizing* property.

Suppose that the two decoders, A and B, are processing the message shown in the diagram below starting from different offsets: A starts at index 0 and B starts at index 3. The first row in the diagram represents the indices of the bits in the encoded input, the second row contains the message itself and the last two rows contain the output from the decoders A and B. Notice that if B starts decoding at index 3, the initial decoded message is erroneous, however starting from bit 7, decoder B starts decoding the rest of the message correctly and gives the same output as A. Bit 7 is referred to as the **synchronization point** because both A and B decoders have become synchronous now.

Hence the idea to parallelize this problem starts from breaking the encoded input into chunks, but also making sure to incorporate the above property to synchronize the outputs from each decoder to ensure correctness.

3.2 Algorithm Overview

We wrote and implemented both a Huffman encoder and decoder that runs sequentially first to test for correctness. The encoder takes in an input text file and writes out both the compressed text together with meta information about the compressed text and the frequency table of the characters which we can read in to construct the Huffman encoding tree. The following code snippet shows the meta data structures and the map we will be output after encoding, and this will be used for decoding:

```
/* Stores the metadata for the encoded file. */
struct meta {
    uint8_t tree_lb_sh_pos; /* Bit shift position of the last tree byte. */
    uint16_t nr_tree_bytes; /* Number of bytes in the encoded tree buffer. */
    uint64_t nr_src_bytes;  /* Number of bytes in the original file. */
    uint64_t nr_enc_bytes;  /* Number of encoded bytes (excluding headers). */
};

/* Maps a character to its frequency in the data. */
struct map {
    uint8_t ch;
    uint32_t freq;
};
struct map *frequency_map;
```

The above is the starting point of our project. We read in the frequency table and use a min heap to construct the Huffman encoding tree. In the sequential decoder, all we have to do is read the encoded text from most signficant bit to the least significant bit and walk down the tree to decode each codeword. Once we reach the leaf node, we successfully decode a character, and we restart at the root node again.

3.3 Parallelizing the Decoder

First, in our algorithm we divide the input data into N equally sized chunks (padded if this cannot be equally divided), which we refer to as **sub-sequences**. B adjacent sub-sequences are grouped into a **sequence** (if N is not a multiple of B, the last sequence will contain less than B sub-sequences) and there should hence be $\lceil N/B \rceil$ number of these sub-sequences. The setup is described as follows:

- We prepare for the decoding by reading in a `char *` array of encoded bits from the encoded file, and copying this array to the CUDA device. This will be the **bit string** we will be decoding.
- We construct the Huffman decoding tree and covert this to a Huffman decoding table. For each character, we pad all possible combinations of bits to the end of the codeword up to `MAX_CODEWORD_LENGTH`, which is the maximum length of a codeword. We add all these combinations of entries to the decoding table. This is so that later on during the decoding process we just have to lookup the decoding table with a fixed `MAX_CODEWORD_LENGTH`) input, and we can translate that into a plaintext character.
- We set up the array of `sync_info` structures, where the i-th `sync_info` structure corresponds to the synchronization information for the i-th sub-sequence. The use of this structure will be more clear in the following section when the algorithm is explained

```
struct sync_info {
    /*
     * Index of the last bit that was decoded in the
     * sub-sequence.
     */
    uint64_t last_word_bit;

    /* Number of bytes  decoded in this sub-sequence. */
    uint64_t num_symbols;

    /*
     * Index to write the first decoded byte of this
     * sub-sequence in the output array.
     */
    uint64_t out_pos;

    /* Synchronization status of this sub-sequence. */
    bool sync;
};
```

3.3.1 Algorithm Outline

1. Phase 1: Intra-sequence synchronization.
2. Phase 2: Inter-sequence synchronization.
3. Phase 3: Exclusive prefix sum over the `num_symbols` field in the `sync_info` array to get the `out_pos` (position to write the decoded characters into the output array for each sub-sequence.
4. Phase 4: Write the output in parallel using the `out_pos` in the `sync_info` array.

3.4 Phase 1

In the following, we show a pseudo code snippet describing the main idea of what we're doing in Phase 1.

Algorithm 2: Phase 1, from a thread's perspective

input : An array of subsequences SQ of size N
output: An array of triples sync_info of size N

```
1  current_subseq ← global thread id;
2  current_subseq_in_block ← local thread id;
3  last_codeword ← decode_subseq(SQ, sync_info,
     current_subseq, false);
4  ++current_subseq;
5  ++current_subseq_in_block;
6  __syncthreads();
7  sync_achieved ← false;
8  for i ← 1 to B do
9      if not sync_achieved and current_subseq_in_block < B
         then
10         last_codeword ← sync_info[current_subseq];
11         current_codeword ← decode_subseq(SQ, sync_info,
             current_subseq, true);
12         if last_codeword equals current_codeword then
13             sync_achieved ← true;
14         end
15         else
16             sync_info[current_subseq] ← current_codeword;
17         end
18     end
19     ++current_subseq;
20     ++current_subseq_in_block;
21     __syncthreads();
22 end
```

Fig: Pseudo-code for "Phase 1" of the decoding process.

The main idea of this phase is to "locate" the correct code-word boundaries inside each sub-sequence. In this phase, we launch N threads, each of which will be decoding a different sub-sequence. Each thread will decode its own sub-sequence and once it is done decoding the last complete code-word, it will write the index of that last decoded bit, to its corresponding `sync_info` entry. We insert a **thread barrier** here to make sure all of the threads in the same block are at the same iteration, and have also finished decoding their assigned sub-sequence.

When we decode sub-sequence, we start from the start of the sub-sequence at the first iteration, or the `last_word_bit` from the left adjacent sub-sequence for all proceeding iterations. After decoding the sub-sequence, we check if the `last_word_bit` detected by the previous thread is the same as the one detected by the current thread. If so, then we have reached a **synchronization point** and we can disable this current thread. This procedure is repeated B times so that the first thread in the sequence (in the block) will be able to synchronize all the sub-sequences in the sequence.

The following diagram illustrates Phase 1 in practice. In this case, we have threads 0-4 working on adjacent sub-sequences in parallel. The bold black bars represent where the

`last_word_bit` ends at for each iteration. In the next iteration, each thread starts decoding from the bold black bars again until all of the bars are synchronized.

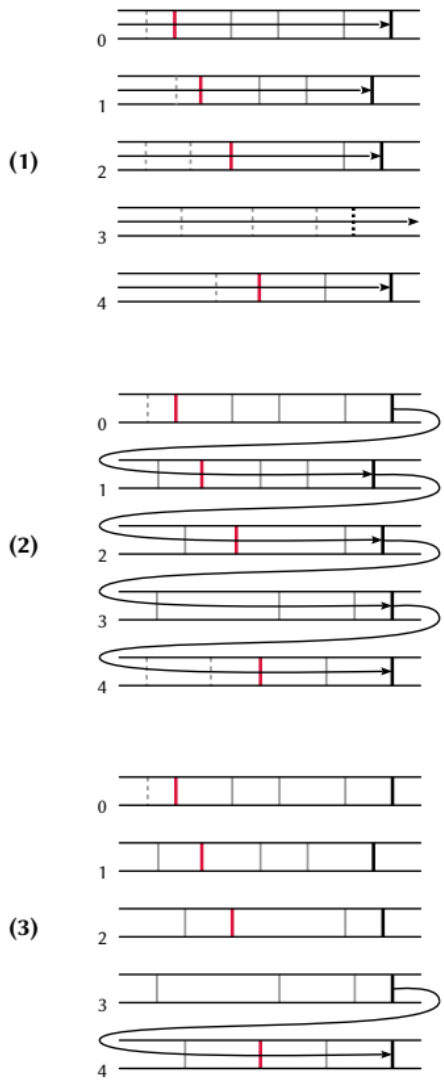

Fig: Illustration of intra-sequence synchronization ("Phase 1").

3.5 Phase 2

Phase 2 is similar to Phase 1 but instead of doing inter sub-sequence (intra-sequence) synchronization, we are doing inter sequence synchronization. In the following, we show a pseudo code snippet describing the main idea of what we're doing in Phase 2.

Algorithm 4: Phase 2, from a thread's perspective

input : An array of subsequences SQ of size N
An array of triples sync_info of size N
An array of flags sequence_synced of size N_b
output: The modified sync_info array
The modified sequence_synced array

```
1  current_subseq ← (global thread id) ·B + B;
2  current_block ← thread block id +1;
3  sync_achieved ← false;
4  for i ← 0 to B do
5      if not sync_achieved then
6          last_codeword ← sync_info[current_subseq];
7          current_codeword ← decode_subseq(SQ, sync_info,
             current_subseq, true);
8          if last_codeword equals current_codeword then
9              sync_achieved ← true;
10             sequence_synced[current_block] ← true;
11         end
12         else
13             sync_info[current_subseq] ← current_codeword;
14         end
15     end
16     ++current_subseq;
17     __syncthreads();
18 end
```

Fig: Pseudo-code for "Phase 2" of the decoding process.

Each thread is now assigned to a different sequence. We start to decode the first sub-sequence, by continuing from the last sub-sequence of the previous sequence. We overflow the synchronization information to the following sub-sequences until we reach a synchronization point is reached. When this sync point is reached, we set the `sync` flag in the corresponding `sync_info` structure.

We insert a **thread barrier** again to make sure that all the threads are at the same iteration, and that the `sync_info structs` are updated before they move on to the next iteration. Again, this procedure is repeated B times until we reach the last sub-sequence of the sequence it was assigned to.

3.6 Phases 3 & 4

These 2 phases should be very similar to the outputting process of assignment 2, as we simply have to get the exclusive prefix sum of the `num_symbols` field. This will give the index at which we should write the decoded characters to the output. The writing of the output can be done in parallel too,

We simply copied our implementation of exclusive scan from assignment 2 to conduct the parallel exclusive scan.

To write out the output, we simply have to allocate an appropriately sized array in the device, run one kernel such that all threads write to the array at the appropriate locations in parallel, and then copy that output array from the CUDA device to host. Lastly, we can `fwrite` the data from this host array to a file to conclude the whole operation.

3.7 Workload Breakdown

Notice that the main issue with parallelizing Huffman decoding was originally because there were **too many "synchronization" and "dependencies"** in the code. For example, to know where the next code-word starts, we needed to know where the current code-word ends. This issue **cannot be avoided** fully, but with parallelization we can make this process **iterative**. Each iteration we are essentially making a guess at where the code-word starts, and on a large scale, the synchronization points will line up eventually. In the research paper, it was shown that it takes 73 bits on average for synchronization to be achieved.

This seems to lend itself to the idea of **thread barriers** that we learnt in class. With threads and thread blocks (sub-sequences and sequences), we can use the thread block synchronization primitives like **barriers** to make sure that all the threads are at the same iteration. Besides this is **very favorable to cache locality**, since all threads in a thread block should work in adjacent bits in the encoded bit string, and should also be writing to adjacent `sync_point` structures.

Furthermore, since we are usually working with large inputs and are doing roughly the same operations on different inputs, this seems like an apt-case of data-parallelism, as mentioned before. The computationally expensive part of the operation is decoding the code-words itself and making sure that they are all synchronized.

4 Discussion: Challenges and Attempted Optimizations

In the following, we also describe several of our initial attempts to implement the algorithm and optimizations that did not work out well. The algorithm itself is very complicated to implement and getting all the parts working together took a lot of effort. It took us quite some time to even get the sequential part of the code working and output the decoding tree for us to even get started on parallelizing the decoder.

4.1 Decoding Tree vs Decoding Array vs Decoding Table Lookup

Although this was skimmed over in the above, the actual decoding of each sub-sequence plays a huge part in the implementation of the algorithm. In the initial sequential version, we simply had a Huffman decoding tree, and at each bit, we simply move to the left or right child node to decode each character.

However, we quickly realize that this implementation cannot be easily adapted to the parallel version since we cannot "copy" the tree directly over to the CUDA device – each node in the tree has many indirect pointers to other nodes and we will have to recursively copy and create each of these nodes.

Our first attempt was to directly **convert this decoding tree to a decoding tree array**, where we simply convert the tree to an array representation: the left child a node currently at index i will be at index $2 * i$ and the right child will be at index $2 * i + 1$. This however, was not very efficient since we would take up to O(`MAX_CODEWORD_LENGTH`) time to traverse the tree to decode one character. This made our implementation very very slow at first, and even perform worse than the sequential version.

We then attempted the **decoding table** approach (final solution uses this) where we will pad each code-word with all possible combinations of bits up to `MAX_CODEWORD_LENGTH` to make up entries of this table. Now, we simply have to grab a `MAX_CODEWORD_LENGTH` window and look it up the decoding table to get the translated plain-text. We take O(1) time to lookup and decode one character now.

The downsides of the both attempts we tried is the cost of space – both implementations will require around $O(2^{\texttt{MAX\_CODEWORD\_LENGTH}})$ space, which can be huge if the tree is very imbalanced. However, we think that this is an appropriate trade-off if this `MAX_CODEWORD_LENGTH` is not ridiculous. In our solution test case, we have this set to around 23-24.

4.2 Skipping Phase 1

One fun experiment we decided to try was varying the algorithm and skipping Phase 1. Essentially, we would **ignore dividing the sequence into sub-sequence,** such that we simply work with sequences and synchronization between sequences. This implementation was indeed much slower than the implementation that we have now since each iteration would be **much slower and there are no checkpoints** from the `sync_info`s that we can use to skip redundant and extra work.

4.3 Bit String Representation

We also had multiple iterations of our code when we tried to figure out the best way to represent the bit string. The easiest way we tried was to use `bool *`, where 1 and 0 is `true` and `false` respectively; however this was very **space inefficient**. Each `bool` term would take up a byte of space (8 bits) even though it was representing 1 bit of information.

We then decided to change to use `char *`, and conduct bit manipulations to extract the exact window of bits we are interested in. This initially resulted in a lot of complications just to get the code to work correctly, especially when working with boundaries of sub-sequences/ sequences. However it proved to be much **efficient and faster** than the previous solution. Perhaps memory thrashes less because of the less space required. Also the lookup in the decoding table using `char *` was easier too rather than having to form the word by concatenating in `MAX_CODEWORD_LENGTH` `bool`s.

4.4 Sync Array Movement from Host and Device

An array `bool`s is used to keep track of the synchronized sequences at every iteration of "Phase 2" to terminate the convergence loop. In our initial implementation, the entire array was copied back-and-forth from the device and the host to check if all the elements in the array was `true`, essentially invoking `cudaMemcpy` of the entire array every time. We

optimized this by initializing the array on device only and running the check inside another kernel and only copying back result (a single `bool`) from device to host at every iteration, and thus reducing the running time.

5 Results

Finally, we discuss and show some results of our project. We conduct all our tests on the GHC machines since we needed to use CUDA libraries, and on the data-sets listed below:

- enwik9: Excerpt of the English Wikipedia, consisting of UTF-8 encoded XML data (raw: 1 GB, compressed: 0.42 GB, compression-ratio: 2.38).
- dickens: Collected works of Charles Dickens (text-file; raw: 9.8 MB, compressed: 5.6 MB, compression-ratio: 1.74).
- mozilla: Tarred executables of Mozilla 1.0 (raw: 51.2 MB, compressed: 40 MB, compression-ratio: 1.29).
- webster: 1913 Webster Unabridged Dictionary in HTML format (raw: 40 MB, compressed: 25 MB, compression-ratio: 1.6).
- xml: Archive of collected XML files (raw: 5.2 MB, compressed: 3.6 MB, compression-ratio: 1.43).
- shakespeare: Complete Works of William Shakespeare (text-file; raw: 5.3 MB, compressed: 3.1 MB, compression-ratio: 1.72).

5.1 Compression Ratio and Correctness

To conduct a baseline double check that the Huffman encoding and decoding is correct, we make sure that the compression ratio looks appropriate. These can be seen in the following graphs where we first show the input size of each data-set, and the resultant compression ratio we achieved using Huffman encoding:

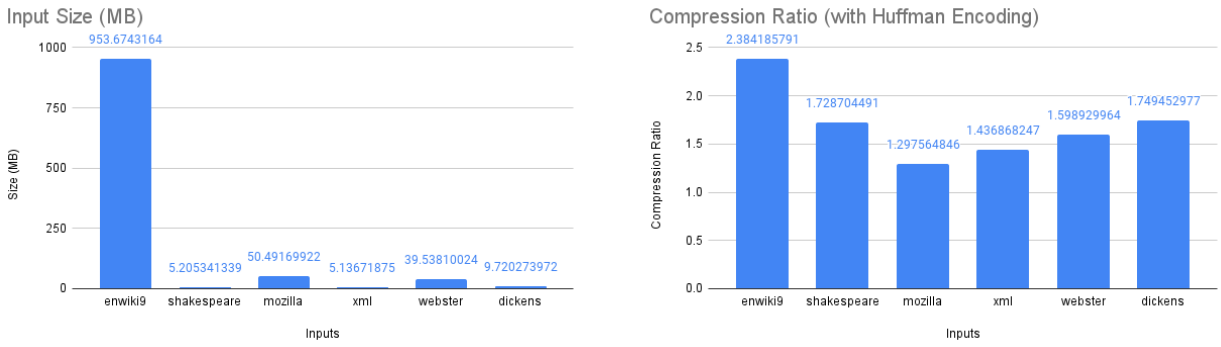

Fig: Input sizes and compression ratios of encoded inputs used for running experiments.

5.2 Speed-up Graphs

We conducted some experiments varying the **number of threads per block** (`THREADS_PER_BLOCK`) and the **sub-sequence size** (`SUBSEQ_SIZE`), and we compare the speed-ups obtained in the following. Recall that both parameters above are parameters that can be fixed by the user. We divide the input into sub-sequences of the `SUBSEQ_SIZE`), and then group `THREADS_PER_BLOCK` number of adjacent sub-sequences together to form a sequence.

This is an overview of the speed-ups we obtain running all the experiments on different data-sets.

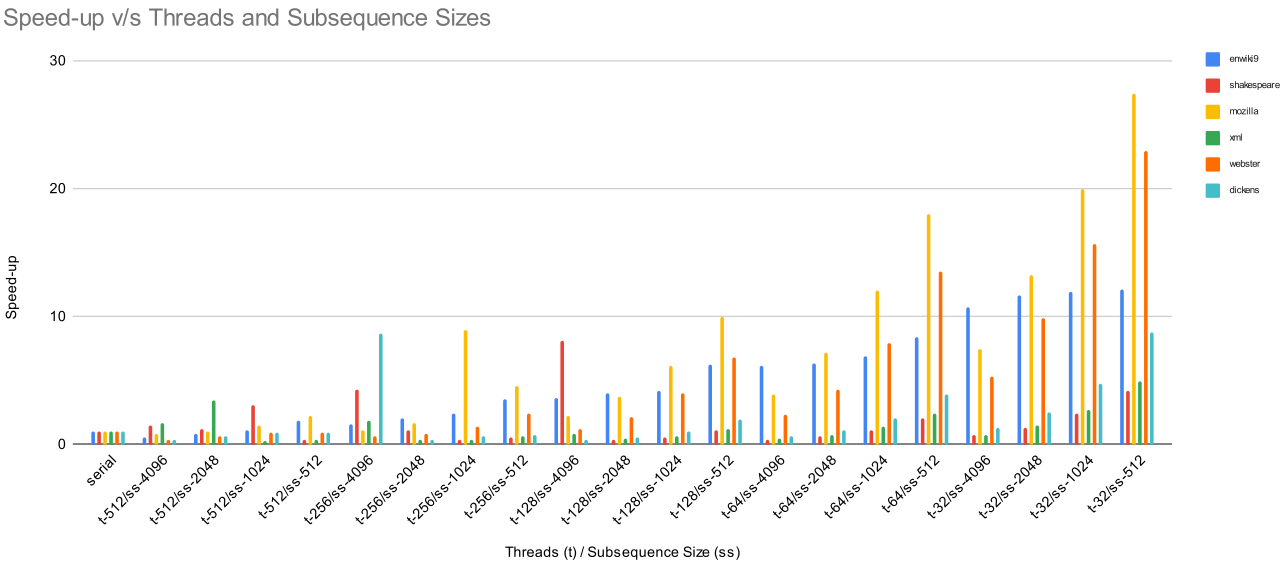

Fig: Overview of the speed-ups obtained for different inputs by varying the number of threads-per-block and the sub-sequence size.

5.2.1 Varying Sub-sequence Sizes

Now, we take a closer look at how **varying the sub-sequence size** affects performance. When we fix the `THREADS_PER_BLOCK` to 32, we get the following speed-up graphs:

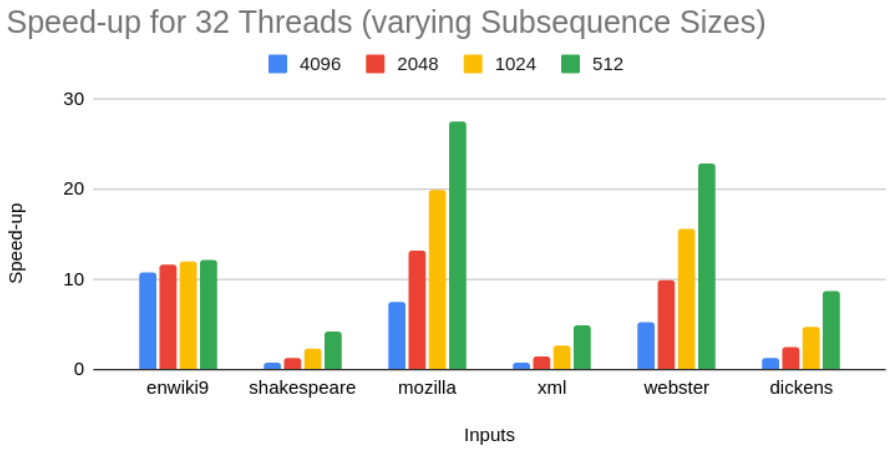

Fig: Speed-ups obtained when `THREADS_PER_BLOCK` = 32 for different sub-sequence sizes.

In the above graph, it seems that the apparent trend is that **speed-up is higher when the `SUBSEQ_SIZE` is smaller**. This makes sense because the `SUBSEQ_SIZE` controls **how big a unit of parallelization is**: the smaller it is, the more parallelism we can have. However, there is a trade-off: if it is too small, we will have too many sub-sequences, and have **too much synchronization**. It seems like in this case a `SUBSEQ_SIZE` of 512 seems to work well.

In contrast, when we fix the `THREADS_PER_BLOCK` to 256, we get the following speed-up graphs:

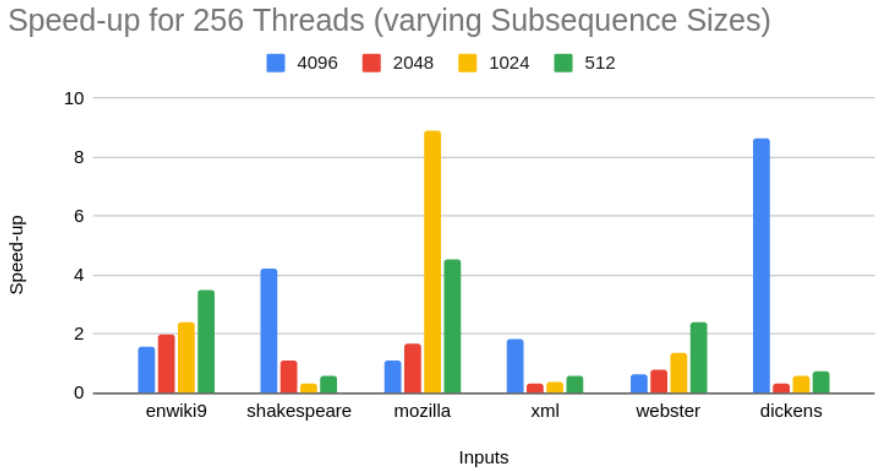

Fig: Speed-ups obtained when `THREADS_PER_BLOCK` = 256 for different sub-sequence sizes.

The general trend is still that we have an increasing trend of speed-up when we decrease the sub-sequence size, similar to how it was before. However, there are many outliers: for example, the *dickens* data-set has the best speed-up when the sub-sequence size was 4096, and has increasing but low speed-ups for any other sub-sequence size. The outliers are

probably just due to certain specific qualities of the data-set (frequency of characters, for instance) that created some noise and local optimums.

Note that there is an interesting relationship between the `THREADS_PER_BLOCK` and `SUBSEQ_SIZE` since they both affect the size of sub-sequence together, and they might have different and possibly effects on Phase 1 and Phase 2. This might have caused the more uncertain trends when we have the number of threads set to an extremity of 256 which is a pretty large number. These effects are further explored in the section XXXX.

5.2.2 Varying number of Threads-Per-Block

Now, we take a closer look at how **varying the threads per block** affects performance. When we fix the `SUBSEQ_SIZE` to 512, we get the following speed-up graphs:

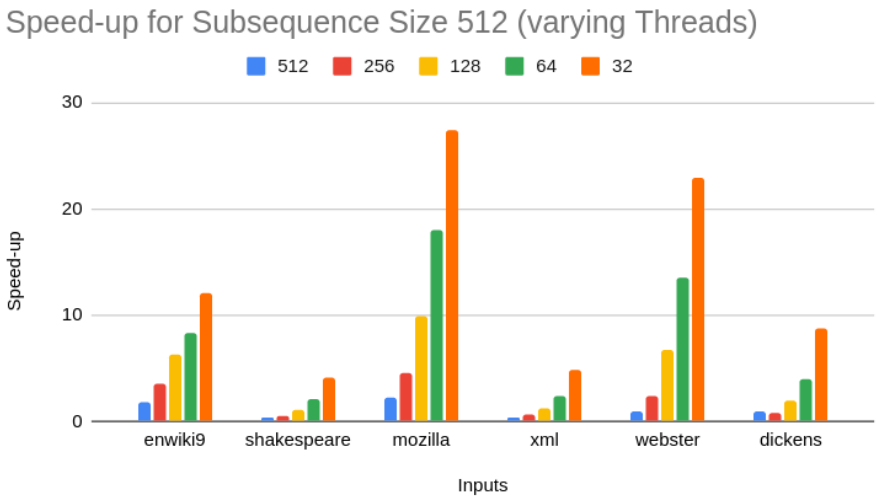

Fig: Speed-ups obtained when `SUBSEQ_SIZE` = 256 for varying number of threads-per-block.

In the above graph, it seems that the apparent trend is that **speed-up is higher when the `THREADS_PER_BLOCK` is smaller**. This makes sense because the `THREADS_PER_BLOCK` also controls both how many sub-sequences make up a sequence and hence controls how big a sequence is. This affects **how big a unit of parallelization is** for phase 2: the smaller it is, the more parallelism we can have. However, there is a trade-off: if it is too small, we will have too many sequences, and have **too much synchronization** for phase 2. Also if it is too small, we won't have much parallelization in phase 1 too: imagine the extreme case of having `THREADS_PER_BLOCK` = 1, that would mean that we're essentially skipping phase 1. It seems like in this case a `THREADS_PER_BLOCK` of 32 seems to work well for most data-sets.

In contrast, when we fix the `SUBSEQ_SIZE` to 4096, we get the following speed-up graphs:

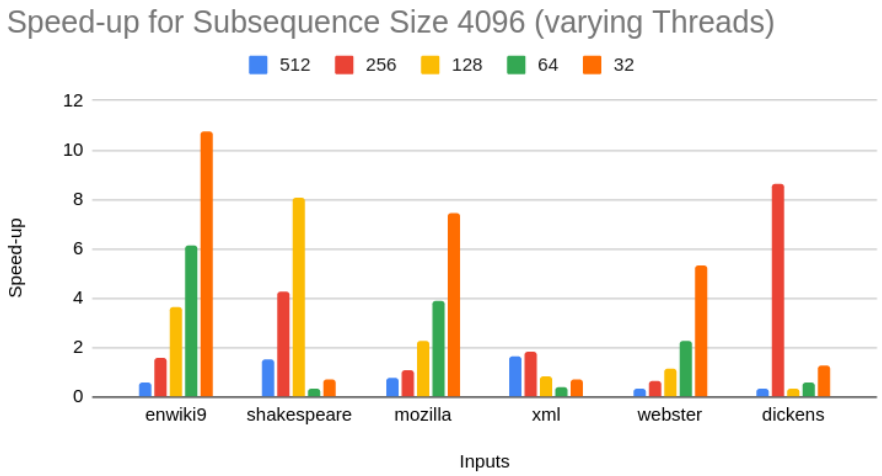

Fig: Speed-ups obtained when `SUBSEQ_SIZE` = 4096 for varying number of threads-per-block.

Although the trend is not as clear as before, the general trend is still that we have an increasing trend of speed-up when we decrease the sub-sequence size, similar to how it was before. However, this time, we may reach a peak of speed up at a larger `THREADS_PER_BLOCK` size. For instance, in the *dickens* data-set, decreasing `THREADS_PER_BLOCK` smaller than 256 causes smaller speed-ups instead of increasing speed-ups. The outliers are again probably just due to certain specific qualities of the data-set that created some noise and local optimum.

5.3 Breakdown of Time Spent in Phases

Next, to further investigate into the trends encountered in the previous section, we now also try to investigate the breakdown of the time spent in the different phases and how this might relate to previous observations in speed-up. The following graphs show the breakdown of time spent in Phase 1, Phase 2 and Phase 3, on the *enwik9* data-set for when `THREADS_PER_BLOCK` and `SUBSEQ_SIZE` are set to (32, 512) and (32, 4096).

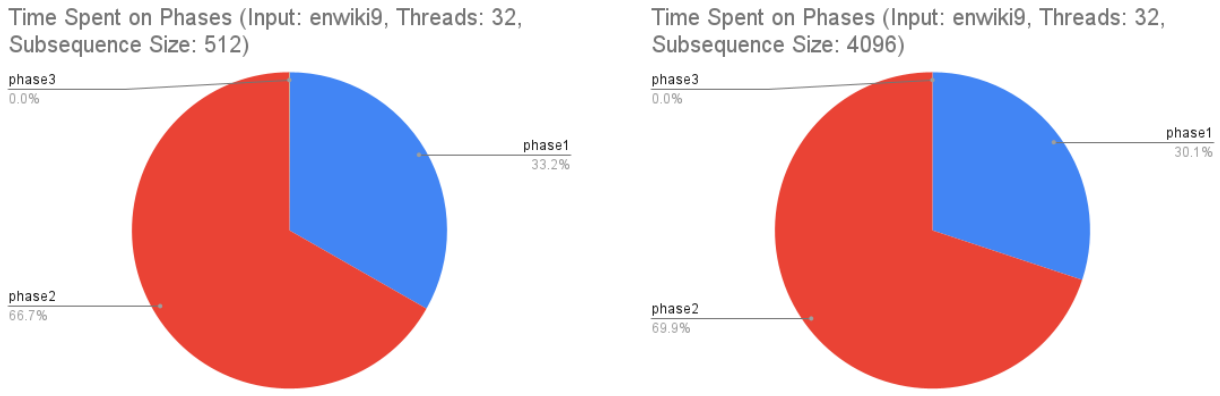

Fig: *enwik9*: Breakdown of time spent across the three phases. We see that most of the time here is spent in "Phase 2".

Notice that Phase 3 is almost negligible in this chart and only Phase 1 and Phase 2 composes the graph. **Phase 2 takes a longer time than Phase 1** in this setting, but the ratio of time spent in both phases for the 2 settings are around the same.

Now we set the `THREADS_PER_BLOCK` and `SUBSEQ_SIZE` to (256, 512) and (256, 4096).

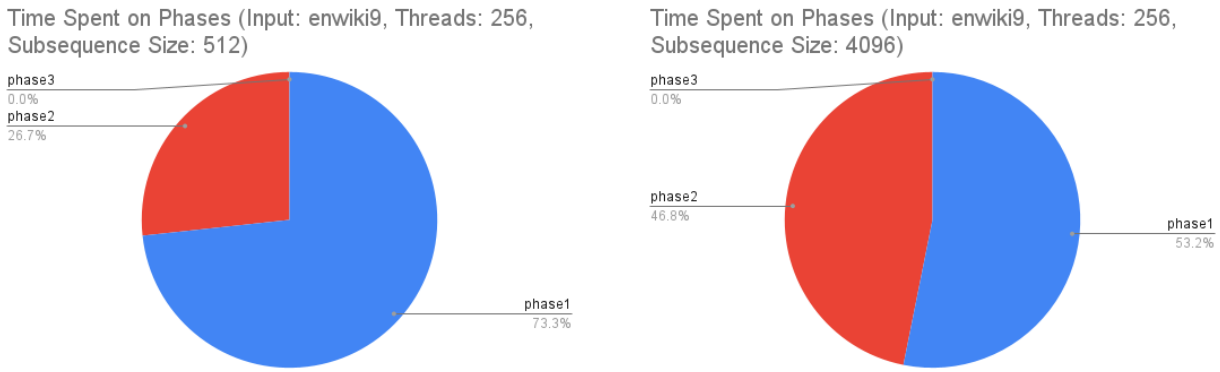

Fig: *enwik9*: Breakdown of time spent across the three phases. We see that most of the time here is spent in "Phase 1".

Again Phase 3 is almost negligible in this chart and only Phase 1 and Phase 2 composes the graph. However, now **Phase 1 takes a longer time than Phase 2** in this setting, but the ratio of time spent in both phases for the 2 settings are around the same.

From both sets of experiments, we notice that **increasing the `SUBSEQ_SIZE` increases the proportion of time spent in Phase 2**. As mentioned before, the `SUBSEQ_SIZE` controls how the size of each unit of work, and the smaller the sequence this size the smaller the granularity of work done by thread for both Phases. However, there is a trade-off because if this granularity is too small/fine, there will be too much overhead – this might be why Phase 1 takes longer when the sub-sequence size is smaller at 512, when it might be too small.

Also, increasing the `SUBSEQ_SIZE` increases the corresponding **size of the sequence** in Phase 2, which fewer intervals at which we have the `sync_infos` for checkpoints and hence less skipping of redundant work. This makes Phase 2 less efficient and hence the increase in time spent in Phase 2.

Increasing the `THREADS_PER_BLOCK` however, decreases the proportion of time spent in Phase 2 (red portions of the second set of graphs is smaller). In Phase 1, we need to iterate `THREADS_PER_BLOCK` number of times to sync all the threads in the block together. The higher the number of `THREADS_PER_BLOCK`, **more thread barriers** we will have to use, and the longer it might take for Phase 1. However, having more `THREADS_PER_BLOCK` may help with Phase 2, since there are many more opportunities where we can potentially skip redundant work in. This is probably why the proportion of time in Phase 2 decreases when we increase the `THREADS_PER_BLOCK`.

Lastly, in the below we show the breakdown of time spent for the `dickens` data-set when we set the `THREADS_PER_BLOCK` and `SUBSEQ_SIZE` to (256, 512).

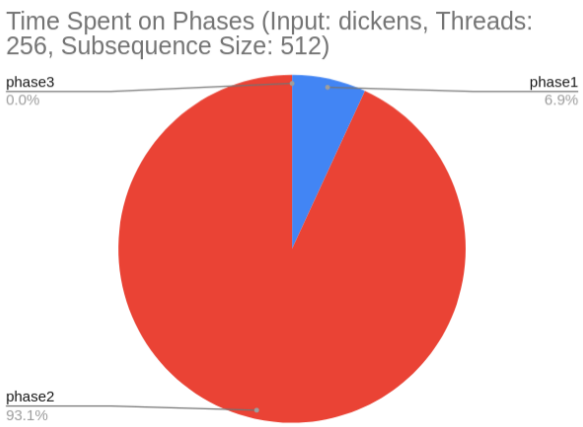

Fig: *dickens*: Breakdown of time spent across the three phases. We see that most of the time here is spent in "Phase 2", here as well.

Notice that compared to the breakdown for the *enwik9* data-set the pie-chart is more unbalanced and there is a greater disparity of time spent in phase 1 and phase 2. Theoretically speaking the best setting of `THREADS_PER_BLOCK` and `SUBSEQ_SIZE` should balance the time spent in both phases equally, to achieve the best possible speed-up. Because this piechart is so unbalanced, this probably explains why the speed-ups for the dickens data-set was less desirable than for the *enwik9* data-set.

5.4 Bottlenecks

In our project, it is very apparent that the main bottleneck is the **dependencies** between different parts of the code and the **synchronization**.

This is evident from the fact that changing the `SUBSEQ_SIZE` and `THREADS_PER_BLOCK` which directly affects these dependencies has a huge effect on the speed-ups observed.

Cache usage should not be an issue since most memory accesses are "sequential access" (e.g., reading of the bit string, reading and writing of the `sync_info`). Communication required is mainly done by reading and writing into `sync_info`s. This communication cost is managed by the usage of **thread barriers** to synchronize across multiple sub-sequences every iteration. We could also say that communication is one of the bottlenecks of the program, but we believe this is more accurately described as a "dependency" limitation that is inherent to the problem.

Workload distributed across the different threads should also be roughly equal and not very unbalanced, since the sub-sequence and sequences they have to decode are of similar length. Hence the bottleneck should not be in workload distribution too.

As analyzed above, Phase 3 and Phase 4, which is concerned with the computing the exclusive prefix sum and writing of the output is also negligible work compared to the rest of the decoding. Hence I/O is not a bottleneck.

6 Work Done

The work was equally split between the two of us for parallelization, integration, testing and experimentation.

7 References

André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018). Association for Computing Machinery, New York, NY, USA, Article 27, 1–10. https://doi.org/10.1145/3225058.3225076